

Lecture notes on Algorithms

Neal E. Young

draft of September 29, 2026

© 2018–2026 Neal E. Young. Text: CC BY-NC-SA 4.0; code: MIT; except where noted. See <https://github.com/nealeyoung/algs101/blob/main/LICENSE>.

Contents

1	Preliminaries	1
1.1	External sources to complement these notes	2
1.2	The 4-step recipe for presenting an algorithm	3
1.3	Meta-cognition to the rescue	6
1.4	Background topics to review	10
1.5	Exercises on background topics	12
2	Long-Form Proofs	14
2.1	Introduction	15
2.2	The long-form proof format	17
2.3	Proofs about algorithms (a few tips, including use of invariants)	24
2.4	How to write long-form proofs in LaTeX	25
2.5	External sources on proofs for algorithms	27
2.6	Exercises	27
3	Stable Matching	37
3.1	Problem definition	38
3.2	Algorithm definition (Gale-Shapley algorithm)	39
3.3	Bounding the worst-case run-time	40
3.4	Proving that the algorithm is correct	41
3.5	Advanced topic: hospital-optimality	43
3.6	External sources on Stable Matching	45
3.7	Exercises	45
4	Divide and Conquer	49
4.1	Bounding sums	50
4.2	Recursion trees for recursive algorithms	52
4.3	An algorithm for the Closest Pair problem	56
4.4	An algorithm for Large-Integer Multiplication	58
4.5	An algorithm for Matrix Multiplication	60
4.6	An algorithm for selecting the k th smallest	62
4.7	An incorrect algorithm for finding a 2D Local Maximum	65
4.8	Additional exercises on Divide and Conquer	68
5	Greedy Algorithms	69

5.1	Introduction	70
5.2	Making Change	70
5.3	Activity Selection	73
5.4	Covering by Intervals	78
5.5	Huffman Coding	81
5.6	Interval Partitioning	85
5.7	Minimizing Total Wait Time	87
5.8	Dijkstra's algorithm for Single-Source Shortest Paths	90
5.9	Minimum Spanning Trees	95
5.10	Additional Exercises	103
6	Graph Traversal	104
6.1	Graph terminology and data structures	105
6.2	Breadth-first Search	105
6.3	Depth-first Search	106
7	Dynamic Programming	114
7.1	Pascal's rule for computing n choose k (with long-form proof)	115
7.2	Paths in DAGs	117
7.3	Making Change	122
7.4	Four steps for presenting a dynamic-programming algorithm	124
7.5	Subset Sum	125
7.6	Knapsack (with long-form proof)	126
7.7	Weighted Activity Selection	128
7.8	Longest Common Subsequence (with long-form proof)	131
7.9	Edit Distance and Sequence Alignment	133
7.10	Longest Palindromic Subsequence (with long-form proof)	135
7.11	Paragraph Layout	137
7.12	Minimum-Weight Polygon Triangulation	138
7.13	Optimal Binary Search Tree	139
7.14	Single-Source Shortest Paths with Negative Weights	140
7.15	All Pairs Shortest Paths / Floyd-Warshall Alg. (w/ l. f. proof)	142
7.16	External sources on dynamic programming	144
8	Max Flow, Reductions, and Linear Programming	145
8.1	Maximum Flow	146
8.2	Polynomial-time reductions	152
8.3	Linear Programming and Integer Linear Programming	159
9	The complexity classes P and NP	164
9.1	Intractable computational problems	165
9.2	Reductions revisited	165
9.3	Reductions between some problems that are new to us	166
9.4	Review of key concepts	168
9.5	The class NP—problems that have poly-time verifiers	169

9.6	External sources on P, NP, and NP-completeness	175
A	Python code	176
A.1	Lecture Note 3: Stable Matching	178
A.2	Lecture Note 4: Divide and Conquer	181
A.3	Lecture Note 5: Greedy Algorithms	186
A.4	Lecture Note 6: Graph Traversal	195
A.5	Lecture Note 7: Dynamic Programming	201
A.6	Lecture Note 8: Max Flow, Reductions, and Linear Programming	221
B	Homework assignments	226
B.1	Preliminaries, Long-Form Proofs, Stable Matching	226
B.2	Preliminaries, Stable Matching, Divide and Conquer	226
B.3	Divide and Conquer	226
B.4	Greedy Algorithms	227
B.5	Greedy Algorithms, Graph Traversal	227
B.6	Greedy Algorithms, Graph Traversal	227
B.7	Dynamic Programming	227
B.8	Dynamic Programming, Greedy Algorithms	227
B.9	Dynamic Programming, Flow	228
B.10	Flow, Reductions, LP	228
B.11	Flow, Reductions, P and NP	228
B.12	Using LaTeX for homeworks	229

Lecture Note 1

Preliminaries

[↑ CONTENTS](#)

Contents

1.1	External sources to complement these notes	2
1.2	The 4-step recipe for presenting an algorithm	3
1.2.1	How to define an algorithm	4
1.2.2	How to write pseudo-code	4
1.2.3	External sources on how to describe algorithms	6
1.3	Meta-cognition to the rescue	6
1.3.1	Learning algorithms $\equiv$ learning a new language and culture	6
1.3.2	Advice from previous students	9
1.4	Background topics to review	10
1.4.1	Big-O notation definitions and useful rules of thumb	10
1.4.2	Properties of logarithms that are good to know	11
1.4.3	External sources on background material	11
1.5	Exercises on background topics	12
1.5.1	Exercises with solutions	12
1.5.2	Exercises without solutions	12

- (**DPV**) *Algorithms*, by Das Gupta, Papadimitriou, and Vazirani.
pros: Concise, accessible, good for building intuition. Drafts available online.
cons: So high level and informal that it hides many underlying details.
- (**JE**) *Algorithms*, by Jeff Erickson. <https://jeffe.cs.illinois.edu/teaching/algorithms/>
pros: Individual chapters online. Articulates the thought processes behind the results.
cons: Relatively new. Not yet widely used.
- (**KT**) *Algorithm Design*, by Kleinberg and Tardos.
pros: Provides motivation with modern examples. Solutions to exercises can be found online.
cons: Wordy and expensive. Solutions to all exercises can be found online.
pro: lecture slides by K. Wayne (KW): <https://www.cs.princeton.edu/~wayne/kleinberg-tardos/>
- (**CLRS**) *Introduction to Algorithms*, by Cormen et al.
pros: Classic reference text. Very thorough. Widely used. Soln's to some exercises are online.
cons: A bit too thorough. Solutions to some exercises are online.
- Those four are widely available. We also refer to the following secondary sources:
- (**LLM**) *Mathematics for Computer Science*, by Lehman, Leighton, and Meyer. Has a few good sections on proofs. <https://courses.csail.mit.edu/6.042/spring18/mcs.pdf>
- (**MF**) *Building Blocks for Computer Science*, by Fleck. Covers background material.
<https://mfleck.cs.illinois.edu/building-blocks/>
- MIT lecture videos.** Available for some topics, mainly from these two courses:
<https://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-006-introduction-to-algorithms-fall-2011/>
<https://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-046j-design-and-analysis-of-algorithms-spring-2015/>

Figure 1.1: External sources to complement the lecture notes. The four main sources are listed first. We recommend that you find one or two that suit you, and learn your way around them.

1.1 External sources to complement these notes

These lecture notes are meant as the primary reference material. They are concise, and they include the low-level details that lectures and textbooks often omit in order to focus on developing intuition.

On the other hand, the lecture notes are terse, and provide neither motivation of the topic nor many examples. These are important, so we suggest that you complement your reading of the lecture notes by using one or more sources in Figure 1.1. Please find copies online or at the library. The various lecture notes will give more specific pointers into these resources.

Prerequisites. See Section 1.4 (and exercises in Section 1.5) for background topics you should know.

1.2 The 4-step recipe for presenting an algorithm

Throughout these notes, we use the following 4-step recipe for presenting each algorithm:

1. Define the problem that the algorithm solves. Specify the possible inputs, and what the acceptable outputs are for each input. Mathematically speaking, the problem definition defines a function mapping each input to a set of outputs. For precision, the definition is typically given in terms of mathematically well-defined objects such as numbers, strings, trees, graphs, sequences, sets, etc. The definition may introduce some problem-specific terminology for convenience.

2. Define the algorithm. Clearly specify the steps that the algorithm takes on any input. Describe the algorithm precisely in English, or pseudo-code if necessary for precision. Avoid giving code—algorithms are not programs. They are read by people, not computers. A competent programmer should be able to easily translate your description into code. Learning to describe algorithms well is one of the main goals of these notes. Section 1.2.1 of this lecture note has more details.

3. Bound the worst-case run-time. We model the worst-case run time as a function T such that, for any positive integer N , $T(N)$ is the maximum, over all inputs of size N , of the number of steps the algorithm takes on that input. Then we give big-O bounds on $T(N)$ in terms of N .

The *actual* number of steps in practice will depend on implementation details (for example: the programming language used, the compiler, and even the particular computer that the implementation runs on). We want our analysis to be independent of these implementation details. Generally speaking, these factors will affect the number of steps by at most some *constant factor*. Also, we are mainly concerned about the run-time for *large* inputs. For these reasons, we give big-O bounds—asymptotic bounds, neglecting constant factors and lower-order terms—on the run time.

4. Prove that the algorithm is correct. (That is, for every input, it returns a correct output.)

In these notes, we model algorithms and the objects they work with mathematically. That is, we define algorithms as well-defined processes that operate in a precise way on mathematical objects such as numbers, strings, trees, graphs, sequences, sets, and so on. The advantage of this approach is *precision*—when we think about or discuss an algorithm, we can understand, and agree on, precisely what we mean. For example, we know precisely what we mean by an assertion such as “Algorithm A is correct” (assuming Algorithm A is itself well-defined).

A “proof” of such a statement is just a careful explanation of a precise line of reasoning by which it is clear that the statement is true (e.g., why, on *every* possible input, the algorithm produces a correct output). The guiding standard for written proofs is that, just by reading the proof, the intended reader should be able to *easily* understand and verify that reasoning, step by step. (As an aside: this means that the level of detail required in a proof depends on the intended reader.)

Learning to read and verify proofs about algorithms is another main goal. As discussed later, learning to do this is similar to learning a new language and culture—proofs are expressed using various standard conventions, and to understand or present a proof you must know these conventions. You’ll need to immerse yourself in this language and culture. At first, you may feel that you have been thrown into the deep end of the pool, but *don’t be intimidated*. Many algorithms work in intuitively accessible ways, with intuitively accessible and familiar structures. With exposure, as you learn the standard conventions and terminology, you will find algorithms, and reasoning about them, to be natural.

We'll do proofs in what we call *long-form* format, which is described in [Lecture Note 2](#), on proofs. Read that lecture note after this one.

1.2.1 How to define an algorithm

This section is about how to do the second step—defining the algorithm. This step must present the algorithm in a precise and unambiguous way, so that from reading what is written the reader can know exactly what the algorithm does. The definition should be (i) *precise* and (ii) *accessible* (easy to read). For any competent programmer, translating the description into code in their favorite programming language should be a straightforward task. Describing algorithms well takes a lot of practice. Here is some guidance.

To achieve both precision and accessibility, you typically begin with an accessible, high-level description of the algorithm, to develop the reader's intuition for the main ideas. This is written in English. Depending on the algorithm, this could be a single sentence, a paragraph, or (for algorithms that are more complicated than those in these notes) a page or more. This description should assume that the reader is a competent programmer (more on this below) and is familiar with common algorithm-design techniques (divide-and-conquer, dynamic programming, greedy algorithms, etc.). Also, it should assume that the reader understands the problem definition.

But it is not always easy to give a description in English that specifies the details in a completely precise way. If necessary, to make sure the details are clear, it often helps to follow the English description by *pseudo-code*—for the algorithm or parts of it. But what is pseudo-code?

1.2.2 How to write pseudo-code

Think of pseudo-code as a high-level programming language, one designed to be read not by a computer, but by a *competent programmer*. Given the pseudo-code, the competent programmer should be able to “compile” the given algorithm into actual code in any programming language (an *implementation* of the algorithm). But what knowledge can you assume that this programmer (the reader) has? Don't assume that, a-priori, the reader knows anything about the algorithm you have in mind. But you can assume the reader knows the following basics:

- The definition of the problem that the algorithm solves, including terminology defined there.
- Programming constructs that are common to most programming languages: **if-then-else** statements, **for** and **while** loops, function calls (including recursion), variables, assignment statements, and nested / indented code blocks. Also, standard data types, including numbers, boolean values, strings, arrays, lists, dictionaries (e.g. hash tables), and so on.
- Standard mathematical objects (e.g., sets, sequences, trees, graphs) and standard mathematical notation (e.g. logical and arithmetical operators, set notation, summations, quantifiers).
- How to implement common programming sub-tasks, such as iterating over a collection of items, computing a sum, computing the maximum or minimum element in a collection, and so on. Also, the worst-case asymptotic time that such a sub-task will take (e.g., $\Theta(n)$ time to iterate over a collection of n items).

For the pseudo-code to be precise, it should have correct overall structure, and each line should have a clear and unambiguous meaning when read by any reader who knows the basics above. To keep it accessible (easy to read and understand at a high level), it should judiciously *omit low-level implementation details*. But what details are okay to omit? Any details that any reader (who knows the basics above) will be able to easily figure out.

```

Sort-and-Count(L)
  If the list has one element then
    there are no inversions
  Else
    Divide the list into two halves:
      A contains the first  $\lfloor n/2 \rfloor$  elements
      B contains the remaining  $\lfloor n/2 \rfloor$  elements
     $(r_A, A) = \text{Sort-and-Count}(A)$ 
     $(r_B, B) = \text{Sort-and-Count}(B)$ 
     $(r, L) = \text{Merge-and-Count}(A, B)$ 
  Endif
  Return  $r = r_A + r_B + r$ , and the sorted list  $L$ 

```

Figure 1.2: Pseudo-code from [KT](#). (It also gives pseudo-code for **Merge-and-Count**, not shown.) This figure is © the authors of [KT](#), reproduced here for commentary; it is not covered by these notes' license.

<pre> closest-pair($P = (p_1, p_2, \dots, p_n)$) 1. if $n \leq 1$: return ∞ 2. if $n = 2$: return $d(p_1, p_2)$ 3. let $m = \lfloor n/2 \rfloor$ 4. let $L = (p_1, p_2, \dots, p_m)$; let $R = (p_{m+1}, \dots, p_n)$ 5. let $\delta_L = \text{closest-pair}(L)$; let $\delta_R = \text{closest-pair}(R)$ 6. let $\delta = \min(\delta_L, \delta_R)$ 7. let $M = \{p_i : x_i - x_m \leq \delta\}$ 8. let $\delta_M = \min\{d(p_i, p_j) : p_i, p_j \in M, i \neq j, y_i \leq y_j \leq y_i + \delta\}$, computed as described in the text 9. return $\min(\delta, \delta_M)$ </pre>	$(p_i = (x_i, y_i) \in \mathbb{R}^2, \text{ with } x_1 < x_2 < \dots < x_n)$
--	--

Figure 1.3: Pseudo-code for finding the closest pair. [Appendix B of the book](#) shows its LaTeX source.

For example, see Figure 1.2, which shows [KT](#)'s pseudo-code for an algorithm to count inversions. In that example, consider the statement starting with “**Divide the list into two halves:...**”. This description precisely defines the task to be solved. From reading just the description, any competent programmer should know how to write code that implements it. So, it is okay that the low-level implementation details of this step are omitted. Indeed, omitting them makes the description more accessible, because the low-level details would be distracting if present! So, *learn to use an appropriately high level of detail!*

Mathematical language is, by design, very precise. For this reason, mathematical language is very useful to precisely define relevant quantities. Consider the example in Figure 1.3. In Step 7, mathematical notation is used to define M as a set of points. In Step 8, it is used to define δ_M .¹ Assuming the reader is comfortable with mathematical notation, defining these quantities mathematically is much clearer than, say, giving a for loop that computes them. So, *learn to use mathematical notation and terminology correctly!*

We recommend that you use LaTeX to write up your solutions. [Appendix B of the book](#) explains how to use LaTeX to prepare the homeworks, including how to write pseudo-code with the macros

¹However, in that step, because the actual method used to compute δ_M is complicated and requires further explanation, its implementation is described in the subsequent text, and Step 8 refers the reader to that description, which we don't repeat here.

that come with the homework templates.

1.2.3 External sources on how to describe algorithms

- [JE](#) Page 10, [Introduction](#) 0.5–0.7
- [CLRS](#) Page 16, Chapter 2.1 (but we prefer that you follow the style in the lecture notes here)

1.3 Meta-cognition to the rescue

Some things to keep in mind, as we proceed, about learning algorithms

1.3.1 Learning algorithms $\equiv$ learning a new language and culture

When you are conversing or writing in a language that you are fluent in, you are generally able to focus on the high-level meaning that you want to convey, without being distracted by having to think about the lower-level mechanics such as grammar and vocabulary. You internalized those lower-level mechanics through previous practice, and now they are automatic for you.

Learning to design and analyze algorithms is like learning a new language. You start at the bottom of a learning curve, where even basic terms and concepts may be unfamiliar, and many aspects seem opaque. You start by learning basic notation and terminology, and practice those until they become natural. Then you learn the basic concepts, and work with them until they become natural. As you practice, these building blocks become automatic for you. You are no longer distracted by the lower-level details, and can focus on understanding algorithms at a higher level. A similar process happens as you practice learning to communicate your ideas clearly.

You are also learning a new culture — the culture of a large “algorithms” community with a rich history, a rich literature, and established (and useful) norms and language.

Immersion, feedback, and interaction. The best way forward is to immerse yourself in it in some substantial way, to engage in regular, very focused practice, with lots of feedback and discussion as you go. Your active engagement will come mostly from working on problems — spending hours thinking about them and struggling with them, and practicing communicating your ideas.

Make a special effort to get (and pay careful attention to) detailed feedback on your attempts. Seek regular feedback from others: fellow students and, if you are taking a course, the course staff. Feedback is crucial for discovering gaps in your understanding and in your explanations. Feedback can also help you stay on track, and not waste as much time when you do have a gap or a roadblock. Look particularly carefully at any feedback on work that you hand in.

In this spirit, we encourage you to discuss the material, including your ideas for problems, with others who are learning it, both to get help understanding the material, and to help others learn it. Teaching others will increase your own understanding. But if you are taking a course, follow its academic-integrity policy: typically you must write up your own solutions on your own, and you must not share your written solutions with other students!

But do your own pull-ups! Watching someone else do pull-ups won’t make you stronger, and reading or hearing solutions won’t make you good at solving problems. The way to learn the material is to work through problems yourself, so engage fully with the exercises.

Meta-strategies for intuition in the face of abstraction. You will encounter many precise but abstract definitions of problems and algorithms. Likewise, the proofs you'll see are given in full generality, and so are also abstract. These definitions and proofs will condense a *lot* of information and insight into a few terse lines, and their abstract nature may be a barrier to understanding. Just reading them once is often not enough to develop a solid intuition for their meaning.

But maintaining an intuitive understanding is crucial! To do that it helps to actively employ some meta-strategies:

1. Start by being mindful of when something (e.g., a definition) is not intuitive or is not clear for you. When that happens, the first thing to do is slow down. Review what you just read, or ask questions about what you just heard. Sit for a minute and consider it. Consider a few specific examples. Lectures and textbooks do this, and it's a good practice to do it on your own. Often, drawing out an example, or a picture that represents the concept, helps.
2. A second meta-strategy: Realize it takes your brain *time to digest* new problems, algorithms, and lines of reasoning. After you first read, say, a proof, or an algorithm, you may have a vague intuition for it, but you might not have digested it enough so that you understand all the pieces, or how they work together. Put it aside for a while, think about something else, then come back to it. You may find that the material has become easier to work with. It takes time for your brain to learn what's relevant. Then opaque concepts can become clear.
3. A third meta-strategy: Talk it out with someone else. Interacting (explaining and listening) engages your brain in ways that thinking on your own may not.

Start problems early. For the more challenging problems, it may take a day or two just to digest the problem, and then another day before you have some good ideas. Your first tentative solutions are likely to have gaps or other errors that you can't see right away. So, you may need to revisit your first drafts after a day or so to find and fix gaps or errors. That is, there will be several phases: understanding the problem, coming up with ideas, developing them, and checking and correcting them. Each phase will take time. So, you should read the problems and start thinking about them early—if they are assigned in a course, at least a week before they are due for the harder problems—and then work on them again several times, a day or two apart. If you can get help (for example, office hours or a discussion forum), consider using it in each phase—early on to get clarification on your understanding of the problem, midway to get feedback on the direction you are pursuing, and towards the end to get feedback on your full attempts.

Read lecture notes and do exercises before lecture. If you are attending lectures on this material, note that each lecture will cover lots of ground. We will introduce a new problem, discuss one or more algorithms for it, then consider how to analyze those algorithms. In short, we'll introduce several layers of concepts, each building on the previous. This will be difficult to follow in real time if it is all new to you. For this reason, we encourage you to read the lecture notes and do some of the easier exercises before the lecture, so you come to lecture already having some idea of the problem definitions and the algorithms we will do that day. Then you will follow more easily when we get into the analysis.

Take-away: work hard to stay ahead of the curve! In short, it takes time to digest each new topic before it clicks. As previous students have noted (see Section 1.3.2), there is a risk of not having a topic click until after the problems on it are due, or after an exam on it. To avoid this, we strongly encourage you to stay ahead of the curve as much as possible. Read lecture notes and do sample exercises at least a few days before the lecture on the topic. Start each set of problems more

than a week before it is due, and continue to work on it in stages. This way, by exam time, you will have had as much time as possible to digest the topics. Also, when you have a time-consuming project in another course, you will not be forced to do this work at the last minute.

1.3.2 Advice from previous students

At the end of a semester-long course based on these notes, students graciously left the following advice for future students:

1. Be honest with yourself and the professor when you don't understand something. At times I felt like because there were no questions from other people and I didn't understand a topic, that maybe everyone already understood it and I was simply behind. This might have been true sometimes, but the stress of snowballing into a topic that you don't understand is far greater than the stress of drawing a bit of attention for a reasonable question. In reference to the analogy of watching someone do pull-ups, just because a professor says something makes sense (especially with proofs), doesn't mean it actually makes sense to you. Proofs are complicated, so give yourself a break. Intuition takes time to build, not always just a lecture. The worst thing that can happen is that the professor will ask you to hold your question if there isn't enough time.

2. Procrastinating will not work for this class. I wish I didn't fight so hard against that fact, but it's simply impossible to determine how long an assignment will take you. There are too many variables that will affect your understanding of the homework for a course like this. It really just depends on what kinds of intuition you have coming into Algorithms. Sometimes it doesn't feel possible (as some people mentioned, we have various other difficult classes to deal with), but it does get easier with practice, and with support from your professors (most of them actually want you to succeed).

3. Review your data structures and practice with them. It didn't take me long to realize that just because I understand an algorithm, doesn't mean I can implement it efficiently. I do wish our semesters were long enough to cover implementations in more detail, but in the meantime, try to understand the things that make the algorithms click. Efficient programming will become important at one point if not now, so be sure that you're at least getting an idea of how things are implemented in actual code and why they work in practice vs theory. Again I'll refer to the pull-up analogy. This is something that I wish I had done more during the semester.

Form study group to revise by teaching each other, but complete the assignments independently. The former helped me to catch gaps in understanding and recall better in exams, the latter prepared me for exam condition. Go for office hour after coming up with a hand-written draft for the assignments, then format them in LaTeX afterward. Also, writing down a detailed cheat sheet helps.

1. Review the lecture notes for specific details on problems. More often than not, I would only get a high level understanding of the algorithms that were being discussed in class, and only understand them intuitively. The lecture notes have a lot of the specific details which actually put the higher level intuition of the algorithms into a concrete, logical, and provable concept.

2. Get feedback on the homeworks after they are graded. Sometimes when I got homework back, I sort of realized what I did wrong but didn't take the time to fully understand why I got points taken off. Usually it was for small details that were crucial to the proofs or correctness of an algorithm, which can make or break a problem. This also helped when preparing for the midterms or the final exam.

1.4 Background topics to review

We assume that from the prerequisite courses you already know various background topics such as how to program, recursion, sorting algorithms (e.g. mergesort), basic data structures (e.g. hash tables, heaps, binary search trees, graphs), and basic discrete math: proof, induction, asymptotic (big-O) notation, set notation, summation notation, quantifiers, basic properties of logarithms, polynomials, and exponential functions, doing algebra with inequalities, and so on.

Below, for reference, are definitions and properties of the big-O notations, and some properties of logarithms that you may want to know. Sources for reviewing other background material are listed at the end of this section.

1.4.1 Big-O notation definitions and useful rules of thumb

Here are formal definitions of the big-O, big-Ω, and big-Θ notations. You'll need them for the exercises.

Definition 1.1. Fix any two functions $f, g : \mathbb{R} \rightarrow \mathbb{R}$.

- $f(n) = O(g(n))$ if there are positive constants $c, n_0 > 0$ such that, for all $n \geq n_0$,

$$0 \leq f(n) \leq c g(n).$$

- $f(n) = \Omega(g(n))$ if $g(n)$ is $O(f(n))$. Equivalently, there are $c, n_0 > 0$ such that, for all $n \geq n_0$,

$$f(n) \geq c g(n) \geq 0.$$

- $f(n) = \Theta(g(n))$ if $f(n)$ is $O(g(n))$ and $f(n)$ is $\Omega(g(n))$. Equivalently, there are $c_1, c_2, n_0 > 0$ such that, for all $n \geq n_0$,

$$0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n).$$

Two rules of thumb for easy cases:

You can use the limit of the ratio f/g to determine the big-O relationship, when the limit exists:

Lemma 1.1. Suppose $f(n)$ and $g(n)$ are non-negative for all sufficiently large n , and the limit $\ell = \lim_{n \rightarrow \infty} f(n)/g(n)$ exists. Then

- $f(n) = O(g(n))$ if and only if $\ell < \infty$
- $f(n) = \Omega(g(n))$ if and only if $\ell > 0$
- $f(n) = \Theta(g(n))$ if and only if $0 < \ell < \infty$

(continued)

Most functions we deal with will have just logarithmic, polynomial, and exponential terms. The next rule of thumb makes it easy to determine the big-O relationships between such functions.

Lemma 1.2. *Suppose, for some constants $a, b, c, a', b', c' \in \mathbb{R}$ (with $a, a' > 0$), that*

$$\begin{aligned} f(n) &= a^n \times n^b \times \log^c n \quad \text{and} \\ g(n) &= (a')^n \times n^{b'} \times \log^{c'} n. \end{aligned}$$

Then $f(n) = O(g(n))$ if and only if (a, b, c) is lexicographically at most (a', b', c') . That is, iff

- $a < a'$, or
- $a = a'$ and $b < b'$, or
- $a = a'$, $b = b'$, and $c \leq c'$.

For example, $n^3 \log^2 n = O(n^{3.1})$, but $n^3 \log^2 n \neq O(n^3)$.

And $2^n = O(3^n/n^3)$, but $2^n \neq O(2^n/n^3)$.

We omit the proofs of these lemmas, but here is the idea for Lemma 1.2. Following Lemma 1.1 consider the limit of the ratio:

$$\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = \lim_{n \rightarrow \infty} \log_2^{c-c'}(n) n^{b-b'} \left(\frac{a}{a'}\right)^n.$$

This limit exists and (by calculation) is finite if and only if the given condition holds.

1.4.2 Properties of logarithms that are good to know

For all $a, b, c > 0$ (with $b, c \neq 1$ wherever they appear as bases):

1. $\log_b(ac) = \log_b a + \log_b c$
2. $\log_b(a^c) = c \log_b a$
3. $\log_b a = \log_c a / \log_c b$... change of base
4. $b^{\log_c a} = a^{\log_c b}$... often useful for recurrence relations
5. $\log(a!) = \Theta(a \log a)$ for positive integers a ... because $(a/2)^{a/2} \leq a! \leq a^a$

1.4.3 External sources on background material

- [DPV](#) Chapter 0 (algorithms, big-O notation)
- [JE Preface](#) (has a detailed list of background topics and further reading), and [Introduction](#)
- [KT](#) Chapters 1 (algorithms) and 2 (big-O notation). (See also the [KW slides](#) for those chapters.)
- [CLRS](#) Chapters 1 & 2 (algorithms), 3 (big-O notation), Appendices A and B (basic math)
- [MF](#), especially

[Chapter 1: Math Review](#)

[Chapter 3: Proofs](#)

[Chapter 11: Induction](#)

[Chapter 14: Big-O](#)

[Chapter 15: Algorithms](#)

1.5 Exercises on background topics

1.5.1 Exercises with solutions

External problems with solutions

- [KT](#) Page 65, Solved Exercises 1 and 2 (big-O notation)

Exercise 1.1. (*Big-O notation, proof*) Suppose $f(n)$, $g(n)$, and $h(n)$ are non-negative functions with $f(n) = O(h(n))$ and $g(n) = O(h(n))$. Use the definition of big-O in [Lecture Note 1](#) to prove that $f(n) + g(n) = O(h(n))$. Give this proof in long form (as described in [Lecture Note 2](#) on proofs). Be careful to get your reasoning about the quantifiers (in the definitions of big-O) right.

Solution.

Proof (long form).

1. Consider any functions f , g , and h as described in the theorem.
2. Let constants $c_1, n_1 > 0$ be such that $(\forall n \geq n_1) f(n) \leq c_1 h(n)$. (These exist by def'n of big-O.)
3. Likewise, let constants $c_2, n_2 > 0$ be such that $(\forall n \geq n_2) g(n) \leq c_2 h(n)$.
4. Fix $n_3 = \max\{n_1, n_2\}$ and $c_3 = c_1 + c_2$.
5. By algebra and the previous steps, then, $(\forall n \geq n_3) f(n) + g(n) \leq c_3 h(n)$.
6. Therefore, by the definition of big-O, $f(n) + g(n) = O(h(n))$. □

Alternative proof with explicit “for all” block (a little more detail than necessary):

Proof (long form).

1. Consider any functions f , g , and h as described in the theorem.
2. Let constants $c_1, n_1 > 0$ be such that $(\forall n \geq n_1) f(n) \leq c_1 h(n)$. (These exist by def'n of big-O.)
3. Likewise, let constants $c_2, n_2 > 0$ be such that $(\forall n \geq n_2) g(n) \leq c_2 h(n)$.
4. Fix $n_3 = \max\{n_1, n_2\}$ and $c_3 = c_1 + c_2$.
- 5.1. Consider any $n \geq n_3$.
- 5.2. Since $n_3 \geq n_1$, $f(n) \leq c_1 h(n)$.
- 5.3. Since $n_3 \geq n_2$, $g(n) \leq c_2 h(n)$.
- 5.4. By algebra, $f(n) + g(n) \leq (c_1 + c_2)h(n) = c_3 h(n)$.
6. By [Block 5](#), for all $n \geq n_3$, we have $f(n) + g(n) \leq c_3 h(n)$.
7. Therefore, by the definition of big-O, $f(n) + g(n) = O(h(n))$. □

1.5.2 Exercises without solutions

Exercise 1.2. (*Big-O notation*) Give the asymptotic values of the following functions, using the Θ -notation. Give the simplest possible answer.

- (a) $\frac{1}{2}n^5 + 7n^4 - n^3 + 13n$
- (b) $3 + 2n^{-2} + 1/(\log^2 n)$
- (c) $n(7n^3 \log n + 9n \log^5 n) + 15n^3$
- (d) $n^2 2^n + 13n^4 + n^3 \log n$
- (e) $n^5 3^n + n4^n$

Exercise 1.3. (*Big-O notation. Similar to Solved Exercise 1, [KT Page 65](#).*) Rank the following functions in increasing order of asymptotic growth rate. That is, reorder the functions so that each function is big-O of the following function. No justification is required.

$$n^3 \quad 7^{\log_2 n} \quad n! \quad 12^n \quad \log_2(n!) \quad 2^{4n} \quad 100n^{3/2} \quad 10n \quad 2^{\log_3 n} \quad \log_2^3 n$$

Exercise 1.4. (*Induction and recursion*) You encounter the following mysterious piece of code.

```
def C(a, n):
1. if n = 0: return (1, a)
2. let (u, w) = C(a, ⌊n/2⌋)
3. if n is even: return (u · u, u · w)
4. if n is odd: return (u · w, w · w)
```

- (a) What are the outputs of $C(a, 1)$, $C(a, 2)$, $C(a, 3)$, and $C(a, 4)$ (in terms of a)? You do not need to justify your answers.
- (b) Give a closed-form expression (in terms of a and n) for the value output by $C(a, n)$, assuming n and a are non-negative integers.
- (c) Prove by induction on $n \geq 0$ that $C(a, n)$ indeed outputs the value you gave.
- (d) Prove that for each positive integer n , during the execution of $C(a, n)$, it does at most $2 + 2 \cdot \log_2 n$ multiplications.

External exercises without solutions²

- [KT Page 67](#), Exercises 1 through 8
- [CLRS Page 52](#), Exercises 3.1–1 through 3.1–8
- [DPV Page 18](#), Exercises 0.1 through 0.3

²As mentioned in Section [1.1](#), solutions for exercises in [KT](#), and some exercises in [CLRS](#), can be found online.

Lecture Note 2

Long-Form Proofs

[↑ CONTENTS](#)

Contents

2.1	Introduction	15
2.2	The long-form proof format	17
2.2.1	There are four kinds of proof steps	17
2.2.2	There are four kinds of blocks	18
2.2.3	Checklist and cheat sheet	21
2.3	Proofs about algorithms (a few tips, including use of invariants)	24
2.4	How to write long-form proofs in LaTeX	25
2.5	External sources on proofs for algorithms	27
2.6	Exercises	27
2.6.1	Exercises with solutions	27
2.6.2	Exercises without solutions	32

2.1 Introduction

Why algorithms? Information surrounds us, but much of it is out of our reach. Here’s a simple example. Suppose you want to know the prime factors of the following 186-digit number: 11679847981112819759721399310592745791658017001955007325132913837831330495881519756453703742878526148\8414688067442512219413748768010657572575384986457405973985247465176041951676954461208131403777. Like every positive integer, this one has a unique set of prime factors. In principle, you could find the factors by exhaustive search—after all, there are only finitely many possibilities. But there are about 10^{93} possible factors to check, so that would take too long, even for a computer. In fact, no algorithm is known that can factor large integers in any reasonable amount of time (e.g., the lifetime of the universe).¹ The information is (in some sense) “there”, but it is out of reach, because we don’t know an efficient algorithm to compute it.

This kind of barrier to knowledge is common. For example, Google has a database with details of all the roads in the United States. From that data, Google Maps can quickly compute, say, a shortest possible route from New York City to San Francisco. But what about a shortest possible route that *visits all 50 U.S. capitols in any order*? Nobody knows how to find such a route with a reasonable amount of computation time.

Whether or not we can learn a desired piece of information often depends on whether there is an *efficient algorithm to solve the underlying computational problem*. Further, for many important and easily stated problems, whether the problem has an efficient algorithm *is not currently known*. New algorithms can lead to advances in science, medicine, biology, business, and so on.

Why proofs? Algorithms are important, but can be difficult to reason about. For example, consider the following simple algorithm:

```

input: integer  $N \geq 1$ 
1. while  $N > 1$ :
2.1. if  $N$  is even:  $N \leftarrow N/2$ 
2.2. else:  $N \leftarrow 3 * N + 1$ 
3. print “done”

```

In 1937, L. Collatz conjectured that, for every input, this algorithm eventually halts. But even after years of study, nobody knows for sure!² Many simple algorithms are hard to reason about. So, when searching for an algorithm to solve a particular problem, we limit our search to algorithms that we *do* know some way to reason about—that is, algorithms that we can prove do work. For this reason, *proofs drive algorithm design*. (More on this later.)

Do your own pull-ups! Textbooks and lectures generally present algorithms that are already very well studied. The author and reader (or instructor and student) generally *already know* that the algorithms are correct. So textbooks and lectures focus on conveying the main underlying ideas. Listening to such a lecture or reading such a text, it is not hard to get an intuitive sense of the algorithm and why it works. And in many algorithms courses, this is all you would do.

But this experience does not teach you how to design your own algorithms. To find an algorithm that works, you typically must search through a variety of ideas. Ideas that seem promising often

¹No such algorithm is known—indeed, standard encryption schemes rely on this fact. But nobody knows for sure that there isn’t one. (We do know that, if a so-called quantum computer can be built, it would be able to factor efficiently.)

²See https://en.wikipedia.org/wiki/Collatz_conjecture.

fail, sometimes for deviously subtle reasons. To know that an idea works, you must also find an airtight line of reasoning as to why. That is, the search for an algorithm proceeds hand-in-hand along with the search for a proof that it works. Doing this is much harder than following a proof in a lecture or textbook. The difference is like the difference between watching someone else do pull-ups, versus doing them yourself. If you want to build the strength to do pull-ups, it doesn't help to just watch someone else do them. You have to do them yourself! So, what does this entail?

Mind the gap! (Check the details.) When you are developing a new proof, you are usually not quite sure it is right. With experience, you will learn that your intuition has blind spots. Because of this, you have to check the details carefully by writing out your reasoning. Any gap in your *explanation* can hide a gap or error in your *reasoning*. So, when developing a new proof, it is important to explain (and check!) your reasoning step by step, in more detail than you see in a typical textbook or lecture. Also, the more clearly you communicate your reasoning, the more a reader can understand about the line of reasoning you have in mind, so the more useful their feedback can be.

Exploring the logical consequences of an assumption. Suppose we don't know that a given statement is true. We can still assume for the moment that the statement is true, and explore the logical consequences if it is. This kind of imaginative thinking can help figure out what is actually true. It is so common that keeping track of assumptions is one of the main organizational challenges in presenting a proof. Conventionally, this is done via four standard proof patterns, *proof by contradiction*, reasoning by *cases*, and when proving *if then* and *for all* statements. When creating complex proofs, it is useful to know these forms like the back of your hand, so you can keep track of your assumptions and organize your reasoning clearly.

These notes: *long-form proofs*. To learn to develop proofs with enough detail, and to learn the four standard forms for managing assumptions, we'll use *long-form proofs*. A long-form proof is given as a sequence of numbered steps, where each step is small, has a clear meaning, and follows easily from previous steps. The numbering allows each step to be identified unambiguously. (This is useful during the proof to make it easy to tell the reader which previous steps a particular step follows from. It also makes communicating about the proof, including giving feedback on it, easier.)

In long-form proofs, assumptions are explicitly managed by using what we call *blocks*. Each block introduces an assumption, which then holds for the scope of the block. Blocks nest like code blocks in computer programs. There are four block types—one for each of the four standard proof forms mentioned above. Section 2.2 describes the long-form proof format in detail.

Basic terms: *problem, algorithm, proof* To design algorithms, and to reason and communicate clearly about them, it is crucial that the language we use is clear, with a precise and unambiguous meaning. To this end, we use the language of mathematics.

A computational *problem* is defined by specifying what the possible inputs are, and what the output should be for each input. (This means that defining a problem is just defining a function.) We cast our problems in terms of standard mathematical objects such as numbers, strings, trees, graphs, sequence, sets and so on.

As described previously in [Lecture Note 1](#), to define an *algorithm*, one specifies exactly how, given any input, the algorithm processes the input to produce the output. The definition should be mathematically precise, and written to be clear and accessible to any competent reader. For this reason, use words, not code. Any competent programmer should be able to translate the definition

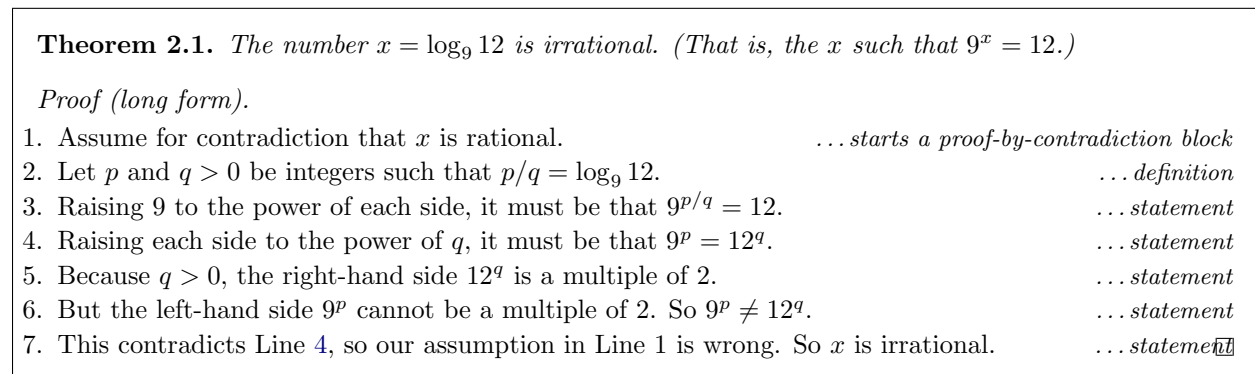


Figure 2.1: A long-form proof with one *proof by contradiction* block.

into code. An algorithm is **correct** (for a given problem) if, for every input, the algorithm produces a correct output.

A **proof** of a fact is a clear explanation, typically in writing, of a complete and correct line of reasoning that establishes that the fact must be true. Typically, the proof is preceded by a clear statement of the fact being proven, e.g. stated as a theorem or lemma. Before you try to write a proof for a fact, you should already have in mind a line of reasoning as to why the fact must be true.³ The reasoning must be complete (covering all cases) and without logical gaps. The goal of the written proof is to communicate your line of reasoning to the reader. The fundamental guiding principle is this: *from what you write, the reader should be able to easily reconstruct your reasoning in detail, and to easily verify that it is correct.*

2.2 The long-form proof format

The long-form proof format uses a particular syntactical form that helps make the details and logical structure of the proof particularly clear. This section defines the format and gives examples. Section 2.6.1 has more examples.

2.2.1 There are four kinds of proof steps

Each long-form proof is given as a sequence of numbered steps. Each step takes up about one line on the page. Figure 2.1 shows an example. There are just four kinds of steps:

1. A *definition* of a term or variable (for example, Step 2 in the figure defines p and q),
2. A *statement of fact*, with a justification (Steps 3–7),
3. A *comment* to guide the reader (the proof above has none),
4. The first line of a *block* (such as in Step 1); per Section 2.2.2, blocks are used to manage assumptions. Each block takes one of four forms:
 - (i) a proof by contradiction,
 - (ii) a proof of an “if then” statement,
 - (iii) a proof by cases,
 - (iv) a proof of a “for all” statement.
5. Some combination of the above (e.g. a definition and a statement; do this rarely, though).

³But of course it is also useful to write partial proofs to guide your thinking.

A **definition** gives a name to some quantity (beware: it must be clear that the quantity does exist). The definition should be mathematically precise, with no ambiguity. The definition holds throughout the remainder of the block in which it occurs. (Just as declaring a variable in a computer program makes the variable available for the rest of the code block in which the declaration occurs.)

Each **statement of fact** states some fact, along with a short justification. *The given fact must follow directly from the facts established in previous steps*, and your anticipated reader should be able to easily see why this is the case. The level of detail you give thus depends on your anticipated reader. For example, the proof in Figure 2.1 assumes the reader is familiar with the definitions of “rational” and “irrational”, and with the basic properties of integers, and it uses such properties without explaining why they hold. When writing up exercises for these notes, you should assume the reader is another student who knows the material in the notes, but who doesn’t yet know anything about the line of reasoning you have in mind.

When you are starting out, if you’re not sure how much detail to give, err on the side of giving more. For the reader, it’s easier to skip unnecessary detail than to figure out what you might have had in mind when details are missing. And gaps (missing details) often hide errors.

As necessary, the statement of fact is accompanied by a justification to guide the reader, along with explicit references to any previous steps that it depends on. (By default, each step depends on the previous step, so if that’s the only dependence you don’t have to mention it.)

2.2.2 There are four kinds of blocks

Long-form proofs can have multiple, possibly nested **blocks**. (The proofs in Figures 2.1 and 2.2 each have just one, though, the main outer block.) Blocks are used to keep track of assumptions made during the proof. *The first line of each block introduces an assumption* (in some cases only implicitly). The assumption is then in effect until the end of the block. That is, within the block we use the assumption as if it were a true fact. We don’t need to know whether the assumption is actually true. Rather, within the block we are imagining that it is true, and exploring what the logical consequences would be if it were. This kind of exploration lets us figure out facts that hold even without any assumptions.

Blocks can be used in long-form proofs in (only) the following four specific ways:

1. Blocks for proof by contradiction. The proof in Figure 2.1 has just one block—its main block, consisting of Steps 1–7. The assumption that starts the block is that the given quantity $\log_9 12$ is rational. The block explores the logical consequences of that assumption, showing that, if the assumption is true, then $9^p = 12^q$ and $9^p \neq 12^q$ (for some p and q). That is, a statement and its negation both hold. This is called a *contradiction*. Since it cannot be true, we deduce that our original assumption (which implies it) must be false. That is, $\log_9 12$ cannot be rational. It must be irrational.

This is called *proof by contradiction*. To do proof by contradiction, use a block whose first line explicitly assumes the opposite of what you want to prove. To signal the reader, in this first line, explicitly use a phrase including the word “contradiction” such as “*Assume for contradiction that...*” Then, in the body of the block, somehow deduce a contradiction—a statement and its negation. If you can do that, you can conclude that the assumption that started the block must be false, and the negation of that assumption must be true. (By the way, proof by contradiction is usually just a convenient way to organize a proof. That is, it is usually possible to give an equivalent “direct” proof without using proof by contradiction.)

2. Blocks to prove “if then” statements. The logical meaning of a statement of the form “if A then B” is that, if A is true, then B must also be true. You can prove such a statement without

Theorem 2.2. *Let $n = 31415923576480135764801$. If $\log_7 n$ is rational, then it is an integer.*

Proof (long form).

1. Assume that $\log_7 n$ is rational. ... starts an “if then” block
2. Let p and q be integers with $q > 0$ such that $p/q = \log_7 n$ definition
3. Raising 7 to the power of each side gives $7^{p/q} = n$ statement
4. Raising each side to the power of q gives $7^p = n^q$.
5. As each integer has a unique prime factorization, this implies that n is a power of 7. ⋮
6. That is, $n = 7^i$ for some integer $i > 0$.
7. Substituting that into the equation in Step 4, by algebra we have

$$\begin{array}{ll}
 7^p = n^q = (7^i)^q = 7^{iq} & \\
 p = iq & \text{(taking the } \log_7 \text{ of each side)} \\
 p/q = i & \text{(dividing each side by } q > 0\text{)} \\
 \log_7 n = i & \text{(using } \log_7 n = p/q\text{).}
 \end{array}$$

8. It follows that $\log_7 n$ is an integer (if it is rational). □

Figure 2.2: A long-form proof with one *if-then* block.

knowing whether or not A is true. Start a block by assuming A, then, under that assumption, show that B must also hold. If you can do that, then B is a logical consequence of A, so, if A is true, then B must also be true. That is, “if A then B”.

Figure 2.2 shows an example. For some particular n , it shows that if $\log_7 n$ is rational, then $\log_7 n$ is an integer. It doesn’t prove whether $\log_7 n$ is rational or not. Rather, it shows that, if we assume that $\log_7 n$ is rational, it follows by logic and algebra that $\log_7 n$ must in fact be an integer.

3. Blocks to prove “for all” statements. A “for all” statement is a statement of the following form, given some set N and predicate Q :

“for all n in N , the predicate $Q(n)$ holds”.

To prove such a statement, use a “for all” block. Start the block with a line such as ⁴

“... 1. Consider an arbitrary n in N .”

This asks the reader to consider a generic element of the set N , referred to by the name n . To let the reader know that you are starting a “for all” block, make sure you include the qualifier “arbitrary” in the first step. Within the block, assuming only that n is in N (and knowing nothing else about n) prove that $Q(n)$ must hold. Suppose you can indeed prove that. Because your reasoning could be applied generically to any particular element of N , it must be that $Q(n)$ holds for every particular n . That is, you can deduce the following statement (outside the block!): *for all n in N , the predicate $Q(n)$ holds*.

For example, note that the proof in Figure 2.2 doesn’t use the specific properties of the particular integer n defined there. So, we can apply the reasoning in that proof to *any* particular positive integer n , to show that if $\log_7 n$ is rational, then it is an integer. In other words, *for all positive integers n , if $\log_7 n$ is rational, then it is an integer*. Figure 2.3 shows the corresponding long-form proof. It uses a “for all” block as described above. The logical structure of the proof mirrors the logical structure of the statement: the main, outer block (Steps 1 through 3) is a “for all” block; the inner block (Block 2) is an “if then” block.

⁴For “for all” blocks, the underlying assumption (not explicitly stated) is that N is non-empty, so n is well-defined.

Theorem 2.3. *For all positive integers n , if $\log_7 n$ is rational, then it is an integer.*

Proof (long form).

1. Consider an arbitrary positive integer n starts a “for all” block
- 2.1. Assume that $\log_7 n$ is rational. ... starts an “if then” block
- 2.2. Let p and q be integers with $q > 0$ such that $p/q = \log_7 n$ definition
- 2.3. Raising 7 to the power of each side gives $7^{p/q} = n$ statement
- 2.4. Raising each side to the power of q gives $7^p = n^q$.
- 2.5. As each integer has a unique prime factorization, this implies that n is a power of 7. ⋮
- 2.6. That is, $n = 7^i$ for some integer $i > 0$.
- 2.7. Substituting that into the equation in Step 2.4, by algebra we have

$$\begin{aligned}
 7^p &= n^q = (7^i)^q = 7^{iq} \\
 p &= iq && \text{(taking the } \log_7 \text{ of each side)} \\
 p/q &= i && \text{(dividing each side by } q > 0) \\
 \log_7 n &= i && \text{(using } \log_7 n = p/q).
 \end{aligned}$$

- 2.8. It follows that $\log_7 n$ is an integer.
3. By Block 2, if $\log_7 n$ is rational, then $\log_7 n$ is an integer. □

Figure 2.3: This proof has a *for-all* block containing an *if-then* block.

4. Blocks for proof by cases. The fourth and final type of block is used for case analysis. Suppose you know that “ C_1 or C_2 ” holds, and you want to show that some proposition X holds. (You may not know which of C_1 or C_2 holds, but you know at least one has to.)

First, consider the case that C_1 holds. Start a block with the assumption that C_1 holds. Under that assumption (within that block) show somehow that X holds. Then you know that, if C_1 holds, then X holds.

Then consider the other case, that C_2 holds. Start a block with the assumption that C_2 holds. Under that assumption (within that block), show somehow that X holds. Then you know that, if C_2 holds, then X holds.

Finally, since you know that C_1 or C_2 holds, and you know that in either case X must hold, you can conclude (without any assumptions) that X holds.

This is the principle of proof by cases. It extends naturally to situations with more than two cases, as long as you know that at least one of case’s assumption must hold. When you start a block for a proof by cases, signal it by starting the statement with a label, such as “Case 1. ...”. You can also use a phrase such as “*Consider the case that ...*” to signal the reader. If necessary, use a comment to the reader before the cases to help the reader understand their structure.

Figure 2.4 shows an example. Here’s the context for the theorem shown there. Assume that, for every pair of people in the world, either the two have met, or they have not. For any given group G of people, if every pair in G has met, call G a *club*. If every pair in G has not met, call G a group of *strangers*. The theorem says that in every group of six, there must be either a group of three who form a club, or a group of three strangers. The proof uses two cases, each with two subcases.

It is interesting to note that in that proof, the part that proves that G is good is constructive. That is, it yields an efficient algorithm, which, given any group G of six people, finds three that are a club or are strangers. (Choose any person x . If x has met at least three of the others, check any pair in those three have met, and so on...)

Theorem 2.4. *Within every group of six people, there exists a club of three or a group of three strangers.*

Proof (long form).

- | | |
|---|------------------------------|
| 1. Let G be an arbitrary group of six people. | ... starts a “for all” block |
| 2. Say that G is <i>good</i> if F includes a club of 3 or a group of 3 strangers. | ... definition |
| 3. Our goal is to prove G is good. | ... comment |
| 4. Fix x to be any one of the six people in G . | ... definition |
| 5. There two cases, each with two subcases: | ... comment |
| 6.1. <u>Case 1:</u> Suppose that, among the five other than x , at least three have met x . | ... start case block |
| 6.2.1. <u>Subcase 1.1:</u> Suppose that no two among those three have met. | ... start case block |
| 6.2.2. Then those three people are a group of strangers, so G is good. | ... statement |
| 6.3.1. <u>Subcase 1.2:</u> Otherwise some pair among those three have met. | ... start case block |
| 6.3.2. That pair, and x , have all met each other, so form a club of three, so G is good. | ... statement |
| 6.4. One of the Subcases 1.1 or 1.2 must hold, and G is good in each, so G is good (in Case 1). | ... stmt |
| 7.1. <u>Case 2:</u> Among the five other than x , at least three have <i>not</i> met x . | ... start case block |
| 7.2.1. <u>Subcase 2.1:</u> Suppose that every pair among those three have met. | ... start case block |
| 7.2.2. Those three are a club, so G is good. | ... statement |
| 7.3.1. <u>Subcase 2.2:</u> Otherwise some pair among those three have not met. | ... start case block |
| 7.3.2. That pair, and x , have all not met, so form a group of three strangers, so G is good. | ... statement |
| 7.4. One of the Subcases 2.1 or 2.2 must hold, and G is good in each, so G is good (in Case 2). | ... stmt |
| 8. One of the Cases 1 or 2 must hold, and G is good in each, so G is good. | ... statement |

Figure 2.4: An example of using blocks for cases. (See the definitions in the text for context.)

Remark. In doing proof by cases, if the reasoning for a case is the same as for a previously proved case, you can just point out that the reasoning is the same, instead of repeating it.

Follow each inner block by its conclusion. For any inner block, the deduction that follows from the reasoning in the block should be stated in the first line following the end of the block (or sequence of blocks for proof by cases). This is true for all of the four block types. For example, in Figure 2.3, just after Block 2 ends, Step 3 states the conclusion implied by the reasoning in the block. In Figure 2.4, just after Blocks 6 and 7 end, Step 8 states the conclusion implied by the reasoning in those blocks.

The main (outer) block should prove the desired fact. The conclusion that follows from the reasoning in the main (outer) block should be the fact that the proof claims to prove.

Lemmas. A lemma is like a “small” theorem. In these notes, we use lemmas within the context of the main proof, to make it modular and therefore easier to understand (just as subroutines are used to make programs more modular). See Problem 2.5 in Section 2.6 for an example.

Section 2.6 has many other examples of long-form proofs for study.

2.2.3 Checklist and cheat sheet

After you’ve written a proof, put it aside for a while, then come back with fresh eyes to check it. Put yourself in the place of the reader, who does not yet know anything about the line of reasoning you have in mind. The high-level criteria is that *from reading just what you wrote, the anticipated reader will be able to easily reconstruct and verify your entire line of reasoning.*

1. The proof is preceded by the statement to be proved (typically as a theorem or lemma).
2. Each step in the proof is either a statement, definition, comment, or the start of a block.
3. Each definition and statement is mathematically precise and well-defined,
4. Each assumption is introduced only in the first line of some block.
5. Each block is of one of the four block types: contradiction, if then, for all, or cases. (A definition does not need its own block!)
6. The line following each inner block (or a series of case blocks) states the fact that follows from the reasoning in the block, according to the block's type.
7. The statement you are trying to prove follows from the main (outer) block.
8. Each name (variable) is defined *before* it is used.
9. Each variable is referred to only within the block in which it is defined.
10. Each statement follows directly by the justification given in that step from the previous steps, either by domain-specific reasoning, basic logic, or via one of the four block structures. The reader should be able to see why easily, without having to think very much, and without having to guess at what you had in mind.
11. The proof doesn't assume that the reader already knows anything specific about the line of reasoning.
12. You *can* assume the reader knows basic facts about the basic structures you are reasoning with (e.g., numbers, trees, graphs, sets, algorithms). You *should* use standard mathematical structures: numbers, tuples, sets, sequences, permutations, graphs, trees, paths, matrices, etc.
13. Resort to formal notation (rather than precise English) only when you can't say what you need to say precisely in words alone. You can define a new term or notation for a concept that you introduce, but avoid this if possible as it increases the cognitive load on the reader.
14. Most importantly, from just reading what you write, the reader should be able to easily reconstruct, and hopefully verify, your entire line of reasoning.

Figure 2.5: A checklist that will help you avoid common errors in proofs.

The main things to check are (i) that each statement and definition is mathematically precise, (ii) that the overall logical structure is correct, (iii) the reasoning is complete (covers all cases), and (iv) that there are no large gaps. Consider each step, and make sure that if asked you could easily explain in more detail exactly what that step says and why it holds. Checking a proof carefully and skeptically is a learned skill.

Figure 2.5 gives checklist that will help you avoid common errors. Figure 2.6 gives a cheat sheet summarizing how to use the four block types.

<u>Proof by cases, given that “A or B” holds</u> ... <u>Case 1.</u> Assume A. ... <i>within this scope, A holds</i> <i>somehow prove C</i> ... - C <u>Case 2.</u> Assume B. ... <i>within this scope, B holds</i> <i>somehow prove C</i> ... - C - By Cases 1 and 2, C holds.	<u>Proof by contradiction</u> ... - Assume for contradiction that B holds. ... <i>within this scope, B holds</i> <i>prove any contradiction</i> ... - contradiction - By Block 2, B is false.
<u>“If then” blocks</u> ... - Assume A. ... <i>within this scope, A holds</i> <i>somehow prove B</i> ... - B - By Block 2, if A, then B. <i>Alternatively, show the contrapositive:</i> “if not B, then not A”.	<u>“For all” blocks</u> ... - Let x be an arbitrary element of S.^a ... <i>within this scope, x is defined</i> <i>somehow prove $P(x)$</i> ... $P(x)$ - By Block 2, every x in S has property $P(x)$. ^aThe definition in Step 2.1 assumes S is non-empty. This assumption is implicitly introduced by the block. For other types of definitions, blocks are not needed.
	<u>Basic logic:</u> (“$\equiv$” means “is equivalent to”) “if A then B” $\equiv$ “if not B then not A” “if A then B” $\equiv$ “B or not A” “not (A and B)” $\equiv$ “(not A) or (not B)” “not (A or B)” $\equiv$ “(not A) and (not B)” “not (not A)” $\equiv$ “A” “not $\forall x \in S. P(x)$” $\equiv$ “$\exists x \in S. \text{not } P(x)$” “not $\exists x \in S. P(x)$” $\equiv$ “$\forall x \in S. \text{not } P(x)$”

Each block introduces an assumption that lasts for the scope of the block. The general principle is that *whenever we want to explore the consequences of some assumption, we begin a new block, with a step that makes that assumption.* The block delimits the scope of that assumption. There are four block types: “proof by contradiction”, “if then”, “for all”, and “cases”. (For the “for all” block, what is the underlying assumption that necessitates a block? That S is not empty!)

Figure 2.6: A cheat sheet summarizing the four block types and how to use them.

2.3 Proofs about algorithms (a few tips, including use of invariants)

Proofs about *algorithms* can be particularly challenging. One reason is that during the execution of an algorithm its state changes. Try to follow these tips when doing proofs about algorithms:

1. When you refer to the state of the algorithm (such as the value of some variable) make sure that it makes clear *exactly what point in time* you have in mind.
2. When you define an object in your proof (i.e., introduce a name for some quantity), by convention, the named object never changes after that—it is static. Be aware of this convention and try to follow it.

It is okay to violate this convention when you are referring in the proof to the name of variable in the algorithm. But (i) if you do this, make sure your wording signals it clearly. For example, say something like this:

1. Consider any execution of the algorithm.
2. Consider the value of the algorithm's variable x just before the first iteration.
- ⋮

And (ii) whenever you refer to the value of x , make sure you make clear exactly *when* during the execution you mean.

3. Learn to use *invariants*. An invariant is a property that is maintained throughout the execution of the algorithm. Invariants are a basic proof technique for reasoning about iterative algorithms. Exercise 2.4 in Section 2.6 gives an example. We'll see more throughout these notes.
4. As a rule, reasoning about processes is generally harder than reasoning about static objects. So, when you can present your proof as reasoning only about static objects, do that. For example, in reasoning about a static object such as a set or a spanning tree, don't present the proof in a way that implicitly or explicitly introduces some new process (e.g. one that builds the object step by step).

Exercise 2.4 in Section 2.6 illustrates the first three points above.

```

\begin{theorem}\label{thm: bad cases}
  For every real number, its square is positive.
  That is,  $\forall x \in \mathbb{R}, x^2 > 0$ .
\end{theorem}

\begin{longFormProof}
  \step Let  $x \in \mathbb{R}$  be an arbitrary real number.

  \begin{block}
    \step \underline{Case 1.} Suppose  $x$  is positive ( $x > 0$ ).
    \step Then  $x^2$  is a product of two positive numbers, so  $x^2$  is positive.
  \end{block}

  \begin{block}
    \step \underline{Case 2.} Suppose  $x$  is negative ( $x < 0$ ).
    \step Then  $x^2$  is a product of two negative numbers, so  $x^2$  is positive.
  \end{block}

  \step In either Case 1 or 2,  $x^2$  is positive, so  $x^2$  is positive.
\end{longFormProof}

```

Figure 2.7: How to use the `theorem`, `longFormProof`, and `block` environments, and the `\step` command. See Exercise 2.7 for the result.

2.4 How to write long-form proofs in LaTeX

Instructions for getting started with LaTeX are in [Appendix B of the book](#). This section assumes you've read them. This one is about how to use LaTeX (and the macros in the homework templates) to write long-form proofs.

Within the body of the document, you can state a theorem or lemma using the `theorem` or `lemma` environments. LaTeX will automatically number the theorem or lemma for you. Follow your theorem or lemma statement by a long-form proof, using the `longFormProof` environment. With that, start each step with a `\step` command. To introduce an inner block, use the `block` environment. Figure 2.7 illustrates these environments and the `\step` command.

Optionally, you can use a `\label` command to record the theorem or lemma number, using a name of your choice. Then you can recall the number by using the `\ref` command. Specifically, `\ref{thm: bad cases}` will reproduce the number (assuming you use the same name as you gave to `\label`). You can use `\label` and `\ref` to label and refer to any type of numbered object in your document, such as a proof step or block. When the numbers change, LaTeX automatically updates the references. (It may take two iterations of running LaTeX for the references to fully update.)

To introduce a lemma inside a long form proof, use the `lemma` environment to state the lemma, then use the `longFormSubProof` environment for the proof of the lemma. An example is shown in Figure 2.8.

```

\begin{theorem}
  Any finite, connected graph  $G$  that has Property  $A$  also has Property  $B$ .
\end{theorem}

\begin{longFormProof}[First long-form proof.]
  \step Let  $G$  be an arbitrary finite, connected graph having Property  $A$ .

  \step Before we give the proof, we prove the following utility lemma:

  \begin{lemma}\label{lemma: high low}
    Let  $v$  be any vertex in  $G$  that has a neighbor  $x$  with larger weight
    ( $w(x) > w(v)$ ). Then  $v$  has a neighbor  $y$  with a smaller weight ( $w(y) < w(v)$ ).
  \end{lemma}

  \begin{longFormSubProof}[Proof of lemma.]

    \step ...

  \end{longFormSubProof}

  \begin{block} \label{A}
    \step Now assume for contradiction that  $G$  does not have Property  $B$ .

    \step ...

    \step So, by Lemma~\ref{lemma: high low} ...

    \step This is a contradiction (as  $G$  is finite).
  \end{block}

  \step By Block~\ref{A},  $G$  must have Property  $B$ .

\end{longFormProof}

```

Figure 2.8: Making a lemma with the `lemma` and `longFormSubProof` environments. See the solution to Problem 2.5 in Section 2.6 for the output.

2.5 External sources on proofs for algorithms

- [LLM](#) Chapter 1 (proofs by contradiction, cases, proofs of “if then” statements, but nothing about “for all” statements), Chapter 6 (invariants), Chapter 5 (induction).

Those topics are also covered in the videos of the first three lectures here:

<https://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-042j-mathematics-for-computer-science-fall-2010/video-lectures/>

- [MF](#) Chapters 2 (Logic), 3 (Proofs), and 11 (Algorithms)
- More practice proof problems (marked “P”) are at <https://www.cs.ucr.edu/~neal/puzzles>.
- *Book of Proof*, by Hammack, Parts II and III (about 70 pages with examples from discrete math). <http://www.people.vcu.edu/~rhammack/BookOfProof/>
- Long-form proofs are similar in spirit to Lamport’s “structured proofs,” which you can read about here: <https://www.microsoft.com/en-us/research/publication/write-21st-century-proof/>.

2.6 Exercises

2.6.1 Exercises with solutions

Exercise 2.1. (*Long-form proof*) Convert the following short-form proof into long form.

Theorem 2.5. *There are infinitely many primes.*

The proof will use that, if a is a common factor of integers b and c , then a is a factor of $c - b$.

Proof (short form). Suppose for contradiction that there are only finitely many primes: $p_1, p_2, \dots, p_n$. Consider the product P of all the primes. Let $p_i > 1$ be some particular prime in the prime factorization of $P + 1$. Then p_i is a factor of both P and $P + 1$, so it is a factor of $(P + 1) - P = 1$, which is impossible. $\square$

Solution. Here’s the proof in long form.

Proof (long form).

1. Suppose for contradiction that there are only finitely many primes.
2. Let $p_1, p_2, \dots, p_n$ be a list of all the primes (larger than 1).
3. Let $P = \prod_{i=1}^n p_i$ be their product. (Note $P > 1$ so $n \geq 1$.)
4. Fix p_i to be any particular prime in the prime factorization of $P + 1$.
5. Then p_i evenly divides P , and also evenly divides $P + 1$.
6. So p_i evenly divides their difference, $(P + 1) - P$, which equals 1.
7. But this is a contradiction, as 1 has no factor other than 1. $\square$

Step 4 defines p_i to be a specific number (one that we know exists at that point) for use later in the proof. Step 4 does not open a new “for all” block—we would only use a “for all” block there to prove something about *all* primes in the prime factorization of $P + 1$.

Exercise 2.2. (*Long-form proof*) In the kingdom of Platonica:

- (P1) Every native is either a knight or a knave (and nobody is both).
- (P2) Every statement a knight makes is true.
- (P3) Every statement a knave makes is false.

Convert the short-form proof of the following theorem into long form:

Theorem 2.6. *Given P1–P3, any person who states “I am a knave” is not a native of Platonica.*

Proof (short form). Assume for contradiction that X is a native of Platonica. If X is a knight, then (by P2) X ’s statement “I am a knave” must be true, so X is a knight and a knave, contradicting P1. Otherwise (by P2) X is a knave, so (by P3) X ’s statement is false, so X is not a knave (contradicting that X is a knave). $\square$

Solution. Here is the long-form proof.

Proof (long form).

1. Let X be an arbitrary person who states “I am a knave”.
- 2.1. Suppose for contradiction that X is a native of Platonica.
- 2.2. By P1, X is either a knight or a knave. We consider each case separately.
- 2.3.1. Case 1. First consider the case that X is a knight.
- 2.3.2. By P2, X ’s statement “I am a knave” is true, so X is a knave.
- 2.3.3. By this and P1, X is not a knight.
- 2.3.4. This is a contradiction (as X is a knight).
- 2.4.1. Case 2. Otherwise, by P1, X is a knave.
- 2.4.2. By P3, X ’s statement “I am a knave” is false.
- 2.4.3. That is, X is not a knave.
- 2.4.4. This is a contradiction (as X is a knave).
- 2.5. By P1, Case 1 or 2 must hold, and each case leads to a contradiction, so we have a contradiction.
3. By Block 2, X is not a native of Platonica. $\square$

Bonus problem: Could we replace (P1) by “(P1) Every native is a knight, a knave, or both”?

Exercise 2.3. (*Big-O definition*) Give a long-form proof of the following theorem:

Theorem 2.7 (KT Chapter 2 Fact 2.4, Page 39). *Suppose $f(n)$, $g(n)$, and $h(n)$ are non-decreasing, non-negative functions and that $f(n) = O(h(n))$ and $g(n) = O(h(n))$. Then $f(n) + g(n) = O(h(n))$.*

Solution.

Proof (long form).

1. Consider any such functions f , g , and h .
2. Let constants $c_1, n_1 > 0$ be such that $(\forall n \geq n_1) f(n) \leq c_1 h(n)$. (These exist by def’n of big-O.)
3. Likewise, let constants $c_2, n_2 > 0$ be such that $(\forall n \geq n_2) g(n) \leq c_2 h(n)$.
4. Fix $n_3 = \max\{n_1, n_2\}$ and $c_3 = c_1 + c_2$.
5. By algebra and the previous steps, then, $(\forall n \geq n_3) f(n) + g(n) \leq c_3 h(n)$.
6. Therefore, by the definition of big-O, $f(n) + g(n) = O(h(n))$. $\square$

Exercise 2.4. (*Invariants*) Here's a puzzle to illustrate the first three points from Section 2.3, including the use of an invariant.

Elizabeth has a secret hiding place where she presently has 35 coins, 38 baseball cards, and 39 pieces of candy. Her younger brother John has his own secret hiding place, but that does not prevent him from taking things from Elizabeth's (her place isn't as secret as she thinks). John always takes two items at a time, each of a different kind so that the number won't decrease too quickly, and he always adds one item of the third kind. On day, some time later, Elizabeth is shocked to find that all the items in her hiding place were of the same kind. Is it possible that it is all candy?

Solution.

Theorem 2.8. *It is not possible that it is all candy.*

Proof (long form).

1. Consider an arbitrary finite sequence, X , of such actions by John.
2. We will prove that, that throughout the sequence, the following invariant holds:
invariant: *the number of coins plus the number of cards is odd.*
3. At the start, the invariant is true because the number of coins plus the number of cards is $35+38$.
4. We will show that each action that John takes preserves the invariant:
 - 5.1. Consider any action in the sequence, where the invariant holds just before the action.
 - 5.2. That is, just before the action the number of coins plus the number of cards is odd.
 - 5.3.1. Case 1. Suppose John takes a coin and a card, and puts back a candy.
 - 5.3.2. Then the number of coins plus cards decreases by two, so remains odd.
 - 5.4.1. Case 2. Suppose John takes a coin and a candy, and puts back a card.
 - 5.4.2. Then the number of coins plus cards doesn't change, so remains odd.
 - 5.5.1. Case 3. Otherwise John takes a card and a candy, and puts back a coin.
 - 5.5.2. Then the number of coins plus cards doesn't change, so remains odd.
 - 5.6. One of the cases holds, and in each case the number of coins plus cards remains odd.
 - 5.7. So the invariant holds just after the action.
6. By Block 5, for each action in X , if the invariant holds just before it, then it holds just after it.
7. As observed in Step 3, the invariant holds initially.
8. So by induction the invariant holds throughout the sequence.
9. In particular, it holds at the end, so at the end the number of coins plus cards is odd.
10. So at the end, the number of coins plus cards is not zero. □

Exercise 2.5. (*Complex long-form proof*) Let $G = (V, E)$ be any graph, where each vertex $v \in V$ has a numeric weight $w(v) \in \mathbb{R}$. Consider the following two properties:

Property A: For every vertex v in the graph, if v has neighbors, then the weight $w(v)$ of v equals the average of v 's neighbor's weights.

Property B: All vertices in G have the same weight.

Give a long-form proof of the following theorem:

Theorem 2.9. *Any finite, connected graph G that has Property A also has Property B.*

Solution. We give two correct long-form proofs. Exercise 2.6 gives an incorrect proof by induction.

First long-form proof.

1. Let G be an arbitrary finite, connected graph having Property A .
2. Before we give the proof, we prove the following utility lemma:

Lemma 2.1. *Let v be any vertex in G that has a neighbor x with larger weight ($w(x) > w(v)$). Then v has a neighbor y with a smaller weight ($w(y) < w(v)$).*

Proof of lemma.

- 3.1. Let v be an arbitrary vertex as in the lemma (so G has a neighbor x with larger weight).
- 3.2.1. Assume for contradiction that v 's neighbors all have weight at least $w(v)$.
- 3.2.2. Then, since v also has at least one neighbor whose weight is strictly larger, the average of v 's neighbors' weights is strictly larger than $w(v)$.
- 3.2.3. This is a contradiction (as G has Property A .)
- 3.3. By Block 3.2, vertex v has some neighbor with weight less than $w(v)$, proving the lemma. $\square$
- 4.1. Now assume for contradiction that G does not have Property B .
- 4.2. Let u and v be two vertices of different weight (they exist because G doesn't have Property B).
- 4.3. Consider any path from u to v in G (it exists as G is connected).
- 4.4. The path must have two adjacent vertices with different weights. (Otherwise, by transitivity, all vertices along the path would have the same weight, but u and v have different weights.)
- 4.5. Of these two adjacent vertices with different weights, let v_1 be the one with smaller weight.
- 4.6. Then v_1 has a neighbor with larger weight.
- 4.7. So, by Lemma 2.1 applied to v_1 , v_1 has some neighbor v_2 with smaller weight ($w(v_1) > w(v_2)$).
- 4.8. So, by Lemma 2.1 applied to v_2 , v_2 has some neighbor v_3 with smaller weight ($w(v_2) > w(v_3)$).
- 4.9. Inductively, there is an infinite path $v_1, v_2, v_3, \dots$ with strictly decreasing vertex weights.
- 4.10. Since the weights strictly decrease along the path, no vertex in G occurs twice on the path.
- 4.11. So G has infinitely many vertices.
- 4.12. This is a contradiction (as G is finite).
5. By Block 4, G must have Property B . $\square$

Here's the same proof in short form.

Same proof, short form. Let G be any finite connected graph with Property A . We use the following observation: *If a vertex v in G has a neighbor x with larger weight ($w(x) > w(v)$) then v also has a neighbor y with smaller weight ($w(y) < w(v)$).* (Otherwise the weight of v would not equal the average of its neighbors' weights.) Suppose for contradiction that G doesn't have Property B . So it has two vertices with different weights. There is a path between these two vertices, along which there must be two neighbors whose weights differ. The smaller-weight neighbor, say v_1 (having a larger-weight neighbor) must also (by the observation) have a smaller-weight neighbor, say v_2 . In turn, v_2 (with larger-weight neighbor v_1) has a smaller-weight neighbor v_3 . Inductively, G contains an infinite path along which the vertex-weights strictly decrease, contradicting that G is finite. $\square$

Here's the second long-form proof.

Second long-form proof.

1. Let G be an arbitrary finite, connected graph having Property A .

2. Let $W = \min_{v \in V} w(v)$ be the minimum vertex weight. (W is well-defined as G is finite.)

We will use the following utility lemma:

Lemma 2.2. *If v is any vertex of weight W , then each of v 's neighbors has weight W .*

Proof of lemma.

- 3.1. Let v be an arbitrary vertex with weight W .
- 3.2. Each of v 's neighbors has weight *at least* W (by definition of W).
- 3.3.1. Assume for contradiction that v has a neighbor of weight strictly more than W .
- 3.3.2. Then the average of the neighbors' weights is strictly more than W .
- 3.3.3. This is a contradiction (as v has weight W and G has Property A).
- 3.4. By Block 3.3, each of v 's neighbors has weight at most W .
- 3.5. By this and Line 3.2, each of v 's neighbors has weight W . □
4. Let v_1 be any vertex of weight W . (Vertex v_1 exists by definition of W .)
5. By the lemma, all neighbors of v_1 have weight W .
6. Applying the lemma to each neighbor of v_1 , all of the neighbors' neighbors also have weight W .
7. Continuing inductively, all vertices reachable from v_1 have weight W .
8. Since G is connected, all vertices are reachable from v_1 , so all vertices have weight W .
9. So G has Property B. □

Exercise 2.6. (*Incorrect induction*) Here's a third proof for the theorem in Problem 2.5. This one has an error. Find the error. Specifically, identify the first step of the proof that asserts a statement that does not follow from the previous steps and the definitions. Explain why the statement does not follow.

Theorem 2.9. *Any finite, connected graph G that has Property A also has Property B.*

Third long-form proof (INCORRECT! FIND THE ERROR!).

1. The proof is by induction on the size of G .
2. Base case: G has two vertices. Then it is easy to verify that Property A implies Property B.
- 3.1. For the induction step, for any $n \geq 2$, assume the theorem holds for all graphs with n vertices.
- 3.2.1. Let G be an arbitrary connected $(n + 1)$ -vertex graph having property A.
- 3.2.2. Let v_1 be any vertex in G , and let G' be obtained from G by deleting v_1 and its edges.
- 3.2.3. G' has Property A, because G does.
- 3.2.4. So, by the inductive assumption, G' has Property B.
- 3.2.5. That is, all vertices in G' have equal weight, say W .
- 3.2.6. Now consider v_1 in G . Each neighbor of v_1 is in G' , so has weight W .
- 3.2.7. Since G has Property A, then, v_1 also has weight W .
- 3.2.8. So all vertices in G have weight W . That is, G has Property B.
- 3.3. By Block 3.2, the theorem holds for all graphs with $n + 1$ vertices.
4. By Line 2, the theorem holds for any graph with two vertices. By Block 3, for every $n \geq 2$, it holds for any graph with $n + 1$ vertices, as long as it holds for every graph with n vertices. Inductively, it holds for all graphs with $n \geq 2$ vertices. □

Solution. The first error is in Step 3.2.3. Nothing established so far implies that G' has Property A: deleting v_1 changes the neighborhoods (and so the neighbor averages) of v_1 's neighbors, so Property A for G does not, by itself, give Property A for G' . (For finite graphs the statement

happens to be true, but only as a consequence of the theorem we are trying to prove, so using it here is circular.) Another error is in the following step: G' does not have to be connected, so the theorem does not apply to G' (however this one can be fixed).

Bonus question. Suppose $G = (V, E)$ is an infinite connected graph with vertex weights and Property A . If all the vertex weights are non-negative, must G also have Property B ? (The answer is no. Can you find an example?)

2.6.2 Exercises without solutions

Exercise 2.7. (*Proof by cases*) Consider the following long-form proof. Note that “positive” means “strictly greater than zero”, while “negative” means “strictly less than zero.”

Theorem 2.10 (False!). *For every real number, its square is positive. That is, $\forall x \in \mathbb{R}, x^2 > 0$.*

Proof (long form). Find the error!

1. Let $x \in \mathbb{R}$ be an arbitrary real number.
- 2.1. Case 1. Suppose x is positive ($x > 0$).
- 2.2. Then x^2 is a product of two positive numbers, so x^2 is positive.
- 3.1. Case 2. Suppose x is negative ($x < 0$).
- 3.2. Then x^2 is a product of two negative numbers, so x^2 is positive.
4. In either Case 1 or 2, x^2 is positive, so x^2 is positive. □

Here is the same proof in short form.

Proof (short form). The proof is by cases. If x is positive, then x^2 is a product of two positive numbers, so is positive. If x is negative, then x^2 is a product of two negative numbers, so is positive. In either case, x^2 is positive. So x^2 is positive. □

Find the error. What is the first step of the long-form proof that makes an assertion that is not necessarily true? In what situation does that assertion fail?

Exercise 2.8. (*Proof practice—red/blue/round/square*)

Complete the proof of the following theorem.

Theorem 2.11. *In a set S of objects, each is either red or blue, and each is either round or square. S has at least one red object, at least one blue object, at least one round object, and at least one square object. Then S has two objects that differ both in color and in shape.*

Proof (long form).

1. Let S be the set of objects in question.
- 2.1. Case 1. Suppose all red objects in S are the same shape.
- 2.2. Assume that the red objects are all square (the case when they are all round is symmetric).
- 2.3. ...
- 2.4. So, in Case 1, there exist two objects that differ both in color and in shape.
- 3.1. Case 2. Otherwise S has a red-round object and a red-square object.
- 3.2. ...
- 3.3. So, in Case 2, there exist two objects that differ both in color and in shape.
4. Case 1 or Case 2 must hold, and in either case the theorem holds, so the theorem holds. □

Exercise 2.9. (*Understanding induction*) Find the error in the following proof by induction.

Theorem 2.12. *In any finite set of marbles, all the marbles are the same color.*

Proof (long form). Where's the error?

1. Define $P(n) \equiv$ “in any set S of n marbles, the marbles are the same color.”
2. We show by induction on n that $P(n)$ holds for all integers $n \geq 0$.
3. First we consider the base cases, $n \in \{0, 1\}$.
4. In any set of zero or one marbles, all marbles are trivially the same color. I.e., $P(0)$ and $P(1)$ hold.
- 5.1. Next we prove the inductive step. Fix an arbitrary integer $n \geq 2$.
- 5.2.1. Assume that $P(n - 1)$ holds. (We'll show that $P(n)$ holds.)
- 5.2.2.1. Let $S = \{m_1, m_2, \dots, m_n\}$ be an arbitrary set of n marbles.
- 5.2.2.2. Let set $S_1 = \{m_1, m_2, \dots, m_{n-1}\}$ contain the first $n - 1$ marbles in S .
- 5.2.2.3. Let set $S_2 = \{m_2, m_3, \dots, m_n\}$ contain the last $n - 1$ marbles in S .
- 5.2.2.4. S_1 has size $n - 1$, and $P(n - 1)$ holds (Step 5.2.1), so all marbles in S_1 are the same color.
- 5.2.2.5. Likewise all marbles in S_2 are the same color.
- 5.2.2.6. Marble m_2 is in S_1 , so by Step 5.2.2.4, each marble in S_1 is the same color as m_2 .
- 5.2.2.7. Marble m_2 is also in S_2 , so by Step 5.2.2.5, each marble in S_2 is the same color as m_2 .
- 5.2.2.8. Each marble in S is in S_1 or S_2 , so (by the previous two steps) is the same color as m_2 .
- 5.2.2.9. That is, all marbles in S are all the same color.
- 5.2.3. By Block 5.2.2, $P(n)$ holds.
- 5.3. By Block 5.2, if $P(n - 1)$ holds, then $P(n)$ holds.
6. By Block 5, for any $n \geq 2$, if $P(n - 1)$ holds, then $P(n)$ holds.
7. By Step 4, $P(0)$ and $P(1)$ hold.
8. By the preceding two steps, and induction, $P(n)$ holds for all integers $n \geq 0$.
9. That is, within any finite set of marbles, all the marbles are the same color. □

For comparison, here's the proof in short form.

Proof (short form). We show by induction on $n \geq 0$ that, within any set S of n marbles, all the marbles are the same color. For the base cases $n \in \{0, 1\}$ this is true because S contains at most one marble. Consider any set S of $n \geq 2$ marbles. Assume the inductive hypothesis holds for any set of $n - 1$ marbles. Order the n marbles in S arbitrarily. Let S_1 contain the first $n - 1$ marbles in S . Let S_2 contain the last $n - 1$ marbles in S . By the inductive assumption, within S_1 all marbles are the same color. Likewise, within S_2 all marbles are the same color. Since S_1 and S_2 both contain marble m_2 , all marbles in $S = S_1 \cup S_2$ are the same color. □

Exercise 2.10. (*Proof practice—rationality (Problem 1.13 of LLM (2018))*)

Complete the proof of the following theorem.

Theorem 2.13. *Let*

$$p(x) = a_0 + a_1x + a_2x^2 + \dots + a_{m-1}x^{m-1} + x^m$$

be an arbitrary polynomial with integer coefficients, whose highest-degree term has coefficient 1. Then any rational root of $p(x)$ is an integer.

Proof (long form).

1. Let p be an arbitrary polynomial as in the theorem, and let x be any rational root of p .
2. Fix integers i and j with $j > 0$ such that $x = i/j$.
3. ...
4. So x is an integer. □

Exercise 2.11. (*Proof practice, greedy algorithms—partition the integers*)

The positive integers are partitioned into several subsets $A_1, A_2, \dots, A_n$ such that, for each $i \in \{1, 2, \dots, n\}$ and x in A_i , $2x$ is not in A_i . What is the minimum possible value of n ?

Complete the proof of the following theorem.

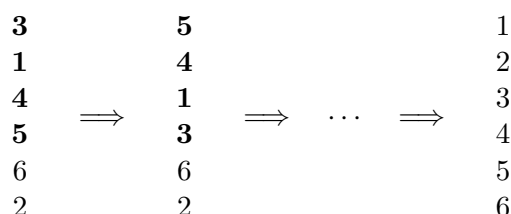
Theorem 2.14. *The minimum possible value of n is 2.*

Proof (long form).

1. For any such partition n is at least 2. (Otherwise 1 and 2 would be in A_1 .)
2. Define $A_1 = \dots$ and $A_2 = \dots$
3. To finish we show that A_1, A_2 is a partition with the claimed property... □

Bonus question: what if you want neither $2x$ nor $3x$ to be in A_i with x ?

Exercise 2.12. (*Reasoning about processes—flipping disks upper bound*) You have n discs labeled $1, 2, 3, \dots, n$ (not necessarily in that order), stacked one on top of each other (sort of like Towers of Hanoi). The basic operation you can do on the stack is the following: take some disc, and all the discs above it, and flip over that part of the stack. For example, you can take the stack on the left, and by applying the basic operation starting at disc 5, you can get the stack in the middle:



The goal is to get to the stack on the right, in which the discs are ordered. Give a long-form proof of the following theorem.

Theorem 2.15. *For any initial stack of n discs, there is a sequence of at most $2n$ basic operations that ends with the discs stacked in order (1 on top, 2 next, ..., n on the bottom).*

Exercise 2.13. (*Reasoning about processes—flipping disks lower bound*) Give a long-form proof of the following theorem, for the problem defined in Exercise 2.12.

Theorem 2.16. *For any $n \geq 1$, there is some initial way to stack the n discs such that any sequence of basic operations that ends with the discs stacked in order has at least $n - 1$ operations.*

(For every n , define a particular initial stacking of the n discs, then prove that there is no sequence of less than $n - 1$ operations that orders them.)

Exercise 2.14. (*Proof practice—boys and girls around a table*) Complete the proof of the following theorem. (*Hint:* make sure your reasoning is complete and covers all possible arrangements. It's not enough to observe that one particular kind of arrangement fails.)

Theorem 2.17. *Fifty chairs are arranged evenly spaced around a round table. For any seating arrangement of 25 boys and 25 girls (placing one boy or girl in each chair), at least one person has a girl on each side.*

Proof (long form).

1. Suppose for contradiction that there is an arrangement where nobody sits between two girls.
2. ...
3. This is a contradiction, so some person sits between two girls. □

Exercise 2.15. (*Proof practice—inclusion/exclusion for reals*) ★ Prove the following theorem.

Theorem 2.18. *Define $f(x, y, z) = |x| + |y| + |z| - |x + y| - |x + z| - |y + z| + |x + y + z|$. Every triple x, y, z of real numbers satisfies the inequality $f(x, y, z) \geq 0$.*

Exercise 2.16. (*Proof practice—handshakes after the party*) ★

Setup: $N + 1$ couples (where $N \geq 0$) attend a dinner. At the end of the dinner, one person (Alice) asks each other person to write down the number of distinct people with whom he or she shook hands at the dinner. Surprisingly, all numbers $0, 1, 2, \dots, 2N$ are written down. Assuming that no person shook hands with their partner, how many people did Alice shake hands with at the dinner? (*Hint:* one person shook hands with $2N$ people. What about that person's partner?)

Complete the proof of the following theorem.

Theorem 2.19. *Alice shook hands with N people.*

Proof (long form).

1. We prove by induction on N that, in any such scenario, Alice shakes hands with N people.
2. In the base case, $N = 0$, ...
- 3.1. For the induction step, consider any $n \geq 1$, and assume that the statement holds for any such scenario with $N = n - 1$ couples. We show that the statement holds for any such party scenario with $N = n$ couples.
- 3.2. ...
4. ... □

Exercise 2.17. (*Induction—Hanoi lower bound*) ★ Check out the towers of hanoi puzzle (https://en.wikipedia.org/wiki/Tower_of_Hanoi), then complete the proof of the following theorem.

Theorem 2.20. *Any legal sequence of moves (which never puts any disc on top of a smaller disk, and moves N discs from one peg to another) takes at least $2^N - 1$ moves.*

Proof (long form).

1. The proof is by induction on $N \in \{1, 2, 3, \dots\}$.
2. The base case $N = 1$ holds because moving one disc requires at least $2^1 - 1 = 1$ move.
- 3.1. For the induction step, consider any $n \geq 1$, and assume that the theorem holds for $N = n$.
- 3.2. We show that the theorem holds for $N = n + 1$...
4. ... □

Exercise 2.18. (*Proof practice—all pairs with different sums*) ★★

Complete the proof of the following theorem.

Theorem 2.21. *A subset S of $\{1, 2, \dots, 100\}$ has the following property: for every quadruple u, v, w, x of distinct numbers in S , the sum of u and v differs from the sum of w and x . Then the size of S is at most fifteen.*

Proof (long form).

1. Consider any such subset S .
2. ...
3. Therefore the size of S is at most fifteen.

□

For more problems, see <https://www.cs.ucr.edu/~neal/puzzles> (those marked “P”).

Lecture Note 3

Stable Matching

[↑ CONTENTS](#)

Contents

3.1	Problem definition	38
3.2	Algorithm definition (Gale-Shapley algorithm)	39
3.3	Bounding the worst-case run-time	40
3.4	Proving that the algorithm is correct	41
3.5	Advanced topic: hospital-optimality	43
3.6	External sources on Stable Matching	45
3.7	Exercises	45
3.7.1	Exercises with solutions	45
3.7.2	Exercises without solutions	47

We use the 4-step recipe for presenting an algorithm, as described in [Lecture Note 1](#). Below we mostly follow the definitions and analysis from [KT Section 1.1](#).

3.1 Problem definition

The *Stable Matching* (historically known as *Stable Marriage*) problem is defined as follows.

input:

- a positive integer n , implicitly defining n hospitals $(h_1, \dots, h_n)$ and n doctors $(d_1, \dots, d_n)$
- $2n$ *preference lists*: for each hospital, a permutation of the doctors, and for each doctor, a permutation of the hospitals.

output: A *stable matching*, defined as follows.

A *matching* M is a collection of hospital/doctor pairs such that each hospital or doctor is in at most one pair in M . For each pair (h, d) in M , we say M *matches* hospital h and doctor d , and that h and d are *matched*. The matching M is *perfect* if M matches each hospital and doctor (i.e., each is in one pair in M , so $|M| = n$). In a perfect matching, a pair (h, d) is an *unstable pair* (also known as an *instability*) if h and d prefer each other to their matches in M . A *stable matching* is a perfect matching with no instabilities.

[To build intuition, consider examples. See [Section 3.6](#) for pointers to textbooks that have motivation and examples.]

There can be more than one stable matching. For a given instance (input), there can be more than one stable matching. (For example, suppose each hospital has a different first choice. Then matching each hospital to his first choice is stable. Likewise, if each doctor has a different first choice, matching each doctor to her first choice is stable. Use these observations to build an example with $n = 2$ that has more than one stable matching. *Challenge:* show that, for every $n \geq 2$, there is an instance with at least $2^{n/2}$ distinct stable matchings.)

Problem variants. There are many possible variants of this problem. For example: Allow having more hospitals than doctors. Or allow incomplete preference lists (where a doctor prefers to be unmatched than matched to someone not on their list). What we discuss below extends cleanly to these variants. One can consider a variant without two different types (the “roommates problem”: given n people (where n is even), each with a preference list ordering all $n - 1$ other people, find a stable matching. ([KW slides](#) have an example for this variant that has no stable matching.)

Exercise: Find an algorithm that, given an instance and a perfect matching, outputs all the unstable pairs (if any). It should run in $O(n^2)$ (linear) time. (It helps to precompute arrays A and B such that $A[h][d]$ gives the rank of doctor d in h ’s preference list, while $B[d][h]$ gives the rank of h in d ’s preference list.)

1. Maintain a matching, initially empty.
2. While there is an unmatched hospital who has not made an offer to every doctor:
 - 3.1. Choose any such hospital h .
 - 3.2. Among doctors that hospital h has not previously made an offer to, let d be the one h prefers most.
 - 3.3. Hospital h makes an offer to doctor d , with the following outcome:
 - 3.4. If doctor d is unmatched, d tentatively accepts the offer, and h and d become matched.
 - 3.5. Else, if d prefers his or her current match, say h' , to h , then d rejects h , and stays matched to h' .
 - 3.6. Else, d tentatively accepts the offer: d rejects h' , and becomes matched to h instead of h' .
4. Finally, return the (possibly incomplete) matching formed by the currently matched pairs.

Figure 3.1: The Gale-Shapley (hospitals-propose) algorithm for Stable Matching.

3.2 Algorithm definition (Gale-Shapley algorithm)

First consider the brute-force algorithm: *Enumerate all $n!$ possible perfect matchings. For each, check if it is stable, and if so halt and return that matching.* This algorithm takes at least $\Omega(n!n^2)$ time, which is super-exponential, way too slow to be practical for large n . Considering that modern CPUs have GHz clocks—they do billions (10^9) of operations per second—this algorithm would currently require more than on the order of $n!n^2/10^9$ seconds, over a year already for $n \approx 17$.

Instead, Gale and Shapley proposed the following algorithm. It is conventionally described as an iterative process that maintains a matching. The matching is initially empty. In each iteration, the algorithm chooses any unmatched hospital. That hospital makes an offer to some doctor—the doctor that it prefers the most, among doctors that it has not previously made offers to. The doctor (tentatively) accepts the offer and becomes matched to that hospital if that new match is better for the doctor than the doctor's current match (or if the doctor is not yet matched). The process halts when every hospital is either matched or has made offers to all doctors. Pseudo-code is shown in Figure 3.1.

Next we consider the following questions:

1. Does this algorithm always terminate?
2. Does it always return a perfect matching?
3. Does it always return a stable matching? (That is, is it correct?)
4. How can we implement this algorithm efficiently? What run time can we achieve?

For intuition, trace the execution of the algorithm on this input:¹

hospitals	1st	2nd	3rd	4th	5th	doctors	1st	2nd	3rd	4th	5th
A	V	W	X	Y	Z	V	B	C	D	E	A
B	W	X	Y	V	Z	W	C	D	E	A	B
C	X	Y	V	W	Z	X	D	E	A	B	C
D	Y	V	X	W	Z	Y	E	A	B	C	D
E	V	X	W	Y	Z	Z	A	B	C	D	E

(continued)

¹source: KW slides

3.3 Bounding the worst-case run-time

Does the algorithm have to terminate? Why? We start by bounding the number of iterations.

Lemma 3.1 (e.g. [KT](#) p.7 (1.3)). *In any execution of the algorithm, there are at most n^2 iterations.*

Proof (long form).

1. Consider an arbitrary execution of the algorithm.
2. In each iteration, one hospital makes an offer to one doctor.
3. By Lines [3.2](#) and [3.3](#) of the algorithm, no hospital makes an offer to the same doctor twice.
4. By two steps above, there are at most n^2 iterations (one for each hospital/doctor pair). □

(Question about long-form proofs: which of the four types of block does the above proof use?)

The algorithm can be implemented so that each iteration takes constant time. (How? Of course, we have to keep track of the current matching, and, for each hospital h , which doctor h should next make an offer to. This is easy to do. The main challenge is to somehow preprocess the input so that, given any doctor and two hospitals, one can check in $O(1)$ time which hospital the doctor prefers. To do this, precompute an array A such that $A[d][h]$ is the rank of h in d 's preference list. See, e.g., [KT](#) Section 2.3.) This and the lemma imply that the algorithm can be implemented to run in $O(n^2)$ time. Note that the input has size $\Theta(n^2)$. We conclude that the algorithm can be implemented to run in time *linear* in the input size:

Lemma 3.2. *The Gale-Shapley algorithm can be implemented to run in time linear in the input size.*

Python code: [stable_matching.py](#) (Section [A.1](#)). It implements the Gale-Shapley algorithm and, for comparison, exhaustive search. Running it shows that Gale-Shapley is much faster than exhaustive search.

(continued)

3.4 Proving that the algorithm is correct

Recall that an algorithm is *correct* if, for every possible input, the algorithm returns a correct output (in this case, a stable matching). We prove this next. We start with some useful observations and a utility lemma. Consider any execution of the algorithm.

Observation 3.1 (KT p.7 (1.1)). *Consider any doctor d . Doctor d is not matched until the first iteration in which d receives an offer. After that d always remains matched, and in each iteration, d either keeps their current match, or rejects that match and becomes matched to a hospital that d prefers more.*

Observation 3.1 follows by inspection of Lines 3.4–3.6 of the algorithm.

Observation 3.2 (KT p.7 (1.2)). *Fix any hospital h . Let $d'_1, d'_2, \dots, d'_k$ be the doctors h makes offers to over the course of the execution, in the order that h makes offers to them. Then $d'_1, d'_2, \dots, d'_k$ is a prefix of h 's preference list. In other words, h prefers d'_1 to d'_2 , and d'_2 to d'_3 , and so on, and h prefers d'_k to each doctor that h does not make an offer to.*

Observation 3.2 follows by inspection of Line 3.2 of the algorithm.

We need one more utility lemma before we prove correctness.

Lemma 3.3 (KT p.8 (1.5)). *Any execution of the algorithm returns a perfect matching.*

Proof (long form).

1. Consider an arbitrary execution of the algorithm.
2. By Lemma 3.1, it terminates, so (by inspection of the algorithm) returns some matching M .
- 3.1. Suppose for contradiction that at least one hospital or doctor is unmatched in M .
- 3.2. M matches each hospital to at most one doctor, and vice versa.
- 3.3. Also, there are n hospitals and n doctors.
- 3.4. So, there must be both an unmatched hospital, say, h , and an unmatched doctors, say, d .
- 3.5. By Line 2 of the algorithm (the termination condition), h made an offer to d at some point.
- 3.6. So, by Observation 3.1, d remains matched at the end (so is matched in M).
- 3.7. This is a contradiction, since d is not matched in M (per Step 3.4 above).
4. By Block 3, each person is matched in M . That is, M is a perfect matching. □

(continued)

To finish we show that the perfect matching is stable.

Lemma 3.4 (correctness; [KT](#) p. 8 (1.6)). *Any execution of the algorithm returns a stable matching.*

Proof (long form).

1. Consider an arbitrary execution of the algorithm.
2. Let M be the perfect matching it returns (by Lemma 3.3).
 - 3.1. Consider any hospital/doctor pair (h, d) . We will show that (h, d) is not an unstable pair.
 - 3.2. Let doctor d' and hospital h' be their matches in M . (They exist, since M is perfect.)
 - 3.3. Note that $d' \neq d$ and $h' \neq h$ (otherwise h and d are a matched pair, so are not an unstable pair).
 - 3.4.1. Case 1. First consider the case that h never made an offer to d during the execution.
 - 3.4.2. Hospital h did make an offer to doctor d' , so, by Observation 3.2, h prefers d' over d .
 - 3.4.3. So (h, d) is not an unstable pair.
 - 3.5.1. Case 2. Otherwise h did make an offer to d during the execution.
 - 3.5.2. By alg. Lines 3.3–3.6, just after that, doctor d was matched to h or a hospital d prefers more.
 - 3.5.3. By Obs. 3.1, d prefers his/her final match h' at least as much, so (h, d) is not an unstable pair.
 - 3.6. Case 1 or Case 2 holds, so (h, d) is not an unstable pair.
4. By Block 3, there are no unstable pairs, so M is stable. □

We've proven that the algorithm runs in linear time and is correct.

(continued)

3.5 Advanced topic: hospital-optimality

We consider another interesting aspect. (See also [KT](#) p.9, “Extensions”.)

First we define the following terms. Fix any Stable Matching instance, say, I .

- Doctor d and hospital h are *feasible* for each other if there exists *any* stable matching (for the instance I) that matches the pair (h, d) . Otherwise we say d is infeasible for h (and vice versa).
- The *best* feasible doctor for a hospital h is the doctor, among the feasible doctors for h , whom h prefers most. Let $\text{best}(h)$ denote this doctor.
- A stable matching is *hospital-optimal* if it matches each hospital h to $\text{best}(h)$.

The only possible hospital-optimal matching is $M = \{(h, \text{best}(h)) : h \in \{1, \dots, n\}\}$. Note that a-priori, this M might not be a matching. (For example, it could be that two different hospitals have the same best feasible doctor d . No matching can match them both to d , so if this happens, there is no hospital-optimal matching.) Also, a-priori, the hospital-optimal matching might not be stable!

Next we show that, not only does the hospital-optimal matching exist for every instance, but the Gale-Shapley algorithm (in the hospitals-propose form) always returns it! The proof requires a careful use of induction. It takes some work to get the underlying intuition!

Consider any execution of the algorithm on any input. We first prove two utility lemmas.

Lemma 3.5. *Suppose a doctor d rejects some feasible hospital h in favor of another hospital h' (Line 3.5 or 3.6 of the algorithm). Then h' has a feasible doctor d' that h' prefers to d .*

Proof (long form).

1. Let d , h , and h' be as in the statement of the lemma.
2. Let M be a stable matching that matches h to d . (M exists by def'n of “feasible”.)
3. M is stable, so (h', d) is not an unstable pair for M .
4. Doctor d rejects h in favor of h' , so d prefers h' to h (algorithm Line 3.5 or 3.6).
5. But (h', d) is not an unstable pair for M , so hospital h' must prefer its match in M , say d' , to d .
6. So d' is a feasible doctor for h' that h' prefers to d . □

Lemma 3.6. *During the execution of the algorithm, each hospital is rejected only by its infeasible doctors.*

Proof (long form).

1. Suppose for contradiction that some hospital is rejected by one of its feasible doctors.
2. Let t be the *first* time that this happens. Let h and d be the hospital and doctor in question.
3. Let h' be the hospital in favor of which d rejects h , i.e., the hospital d is matched to just after time t . Note that $h' \neq h$.
4. By Lemma 3.5, h' has a feasible doctor, say d' , that h' prefers to d .
5. Hospital h' made an offer to d (at or before time t), but prefers d' more.
6. So (by Obs. 3.2), h' previously made an offer to d' .
7. So hospital h' must have been rejected by d' before making an offer to d .
8. This contradicts that t is the first time that a hospital was rejected by one of its feasible doctors. □

From Lemma 3.6, hospital-optimality of the final matching M follows easily.

Lemma 3.7 (hospital optimality; KT p. 10 (1.7)). *The matching output by the algorithm is hospital-optimal.*

Proof (long form).

1. Let M be the matching output by the algorithm.
- 2.1. Consider any hospital h . Let d be its matched doctor in M .
- 2.2. M is stable, so, by definition of feasible, d is feasible for h .
- 2.3. By Observation 3.2, h was rejected by every doctor that h prefers to d .
- 2.4. By this and Lemma 3.6, every doctor that h prefers to d is infeasible for h .
- 2.5. So, d , who is h 's doctor in M , is h 's most-preferred feasible doctor. That is, $d = \text{best}(h)$.
3. By Block 2, M matches each hospital h to $\text{best}(h)$. That is, M is hospital-optimal. □

We've proved that there is always a hospital-optimal matching, and the algorithm always outputs it. One consequence is that, for any instance, every possible execution of the algorithm yields the same matching. This was not obvious a-priori, as different executions can choose different hospitals in each iteration.

Exercise 3.5 is to show that the algorithm's matching is *doctor-pessimal*.

(continued)

3.6 External sources on Stable Matching

- [KT](#) Sections 1.1 and 2.3 (on implementation) and [KW slides](#) (for those chapters)
- [JE Chapter 4](#) Section 4.5; [LLM](#) (MIT Math for CS text) Section 6.4
- Lecture video (U. Vazirani)²
- Can the participants gain an advantage by misrepresenting their preferences?³

3.7 Exercises

3.7.1 Exercises with solutions

Exercise 3.1. (*Unstable pairs*) Consider this instance of Stable Matching with $n = 3$ (with hospitals X, Y, Z and doctors A, B, C).

$X:$	A	B	C
$Y:$	B	C	A
$Z:$	C	A	B

$A:$	Z	Y	X
$B:$	X	Y	Z
$C:$	X	Z	Y

List all pairs that are unstable for this input and matching $\{(X, B), (Y, C), (Z, A)\}$.

Solution. (Z, C) is the only unstable pair.

(Each of the hospitals get its second choice, so a pair (h, d) can be unstable only if doctor d is hospital h 's first choice. So the only potentially unstable pairs are (X, A) , (Y, B) , and (Z, C) . These are the pairs (h, d) where hospital h prefers doctor d to its matched doctor. Among these, pairs (X, A) and (Y, B) are not unstable, because A and B are matched to their first choices. But pair (Z, C) is unstable, because C is matched to his or her last choice.)

Exercise 3.2. (*Gale-Shapley practice*) Consider executing the Gale-Shapley algorithm (hospitals-propose, as described in the lecture notes) on the input from Exercise 3.1. Can there be more than three iterations before the algorithm halts?

Solution. No. Each hospital has a different first choice, so after the first three iterations, all hospitals are matched.

Exercise 3.3. (*Gale-Shapley practice. Similar to the example on Page 173 of JE Chapter 4.*) Show an execution of the Gale-Shapley (hospitals-propose) algorithm on the input from Exercise 3.1 modified so that the hospitals are A, B, C and the doctors are X, Y, Z .

Solution.

1. Hospital A makes an offer to doctor Z . Z accepts. Current matching is $\{(A, Z)\}$.
2. Hospital B makes an offer to doctor X . X accepts. Current matching is $\{(A, Z), (B, X)\}$.
3. Hospital C makes an offer to doctor X . X rejects C . Matching is unchanged.
4. Hospital C makes an offer to doctor Z . Z accepts C and rejects A . Current matching is $\{(B, X), (C, Z)\}$.
5. Hospital A makes an offer to doctor Y . Y accepts. Final matching is $\{(A, Y), (B, X), (C, Z)\}$.

Exercise 3.4. (*Practice with another algorithm*) In 1951, before Gale and Shapley proposed their algorithm, a different algorithm was in use. For instances with $n = 3$, that algorithm is shown in Figure 3.2. What is the output of that algorithm on the input from Exercise 3.1?

²https://archive.org/details/ucberkeley_webcast_8qhxIAWp-zk

³<https://math.stackexchange.com/questions/2775243>

1. For each pair (i, j) in $\langle (1, 1), (1, 2), (2, 1), (1, 3), (3, 1), (2, 2), (2, 3), (3, 2), (3, 3) \rangle$, do:
 - 1.1. Consider those hospital/doctor pairs (h, d) such that d is h 's i th choice, h is d 's j th choice, and neither h nor d are yet matched. For each such pair (h, d) , match h to d .
2. Return the matched pairs.

Figure 3.2: In 1951, before Gale and Shapley proposed their algorithm, a different algorithm was in use. For instances with $n = 3$, the algorithm is shown above.

Solution. First, here is the input in the form of a table of the ranks (i, j) for each hospital/doctor pair:

	A	B	C
X	(1, 3)	(2, 1)	(3, 1)
Y	(3, 2)	(1, 2)	(2, 3)
Z	(2, 1)	(3, 3)	(1, 2)

(For example, X ranks A first, while A ranks X third, so the entry for hospital X and doctor A is $(1, 3)$.)

In the first iteration, when $(i, j) = (1, 1)$, the algorithm doesn't match any pairs (because there are none with those ranks). In the second iteration, when $(i, j) = (1, 2)$, the algorithm matches Y to B and Z to C . In the third iteration, when $(i, j) = (2, 1)$, the algorithm doesn't match any pairs (because B and Z are already matched). In the fourth iteration, when $(i, j) = (1, 3)$, the algorithm matches X to A . After that, no more matches are made (because everybody is already matched).

The algorithm returns matching $\{(X, A), (Y, B), (Z, C)\}$. In this case, each hospital is matched to his first choice, so cannot be in any unstable pair. So there are no unstable pairs, and the matching is stable.

Exercise 3.5. (*prove Doctor-pessimality*) Section 3.5 shows that the Gale-Shapley algorithm (with hospitals proposing) outputs a hospital-optimal matching. In this exercise you will show that the matching is *doctor-pessimal*, defined as follows (in the terminology from Section 3.5). Fix any Stable Matching instance I . For any doctor d , let $\text{worst}(d)$ denote d 's least-preferred feasible hospital. A stable matching is *doctor-pessimal* if it matches each doctor d to $\text{worst}(d)$. (In this case the matching must be $\{(\text{worst}(d), d) : d \in \{1, 2, \dots, n\}\}$, but note that a-priori this set of n pairs might not be a matching, or might not be stable.)

Prove the following lemma. Hint: use hospital-optimality (Lemma 3.7).

Lemma 3.8 (doctor pessimality; KT p.11 (1.8)). *The matching output by the Gale-Shapley algorithm (hospitals-propose) is doctor-pessimal.*

Solution.

Proof (long form).

1. Assume for contradiction that some execution outputs a matching M that isn't doctor pessimal.
2. Let (h, d) be a matched pair in M such that $h \neq \text{worst}(d)$. Let $h' = \text{worst}(d)$.
3. Let H' be a stable matching that matches h' and d . (H' exists by definition of $h' = \text{worst}(d)$.)
4. By Lemma 3.7, M is hospital-optimal, so $d = \text{best}(h)$, and h prefers d to its matched doctor in H' .
5. Since H' is stable, d must prefer his or her match in H' (that is h') to h .
6. Both h and h' are feasible for d , so this contradicts that $h' = \text{worst}(d)$. □

Exercise 3.6. (*Implement Gale-Shapley*) Do the following Hacker-Rank challenge (implement the algorithm): <https://www.hackerrank.com/neaileyong-algs101-stable-matching>.

Solution. Python code: `stable_matching.py` (Section A.1).

External exercises with solutions

- **KT:** Page 19, Solved Exercises 1 (reducing another problem to Stable Matching) and 2 (a variant of Stable Matching).

3.7.2 Exercises without solutions

Exercise 3.7. (*Stable matching definition*) Describe a specific instance I of Stable Matching with $n = 3$ that has only one stable matching. Describe the stable matching. Explain clearly why it is stable. Explain clearly why no other matching is.

Exercise 3.8. (*Another algorithm, proof problem 1*) Does the algorithm in Exercise 3.4 return a *perfect* matching for every input with $n = 3$? If so, give a long-form proof that it does. If not, give a counter-example: describe a particular Stable Matching instance with $n = 3$ and show that the algorithm does not give a perfect matching for that instance.

Exercise 3.9. (*Another algorithm, proof problem 2*) Consider the following lemma and proof about the algorithm in Exercise 3.4:

Lemma 3.9. *For any execution of this algorithm, if it returns a perfect matching M , then M is stable.*

Proof (long form).

1. Consider any execution of the algorithm that returns a perfect matching M .
 - 2.1. Consider any pair (h, d) .
 - 2.2.1. Suppose for contradiction that (h, d) is unstable for M .
 - 2.2.2. Let d' and h' be the respective matches of h and d in M (they exist, as M is perfect).
 - 2.2.3. Because (h, d) is an unstable pair, h ranks d above d' .
 - 2.2.4. So, by inspection of the alg. (Line 1), it considered the pair (h, d) before the pair (h, d') .
 - 2.2.5. So, when the algorithm considered the pair (h, d) , hospital h was not yet matched.
 - 2.2.6. Likewise, since (h, d) is unstable for M , doctor d ranks h above h' .
 - 2.2.7. So, by inspection of the alg. (Line 1), it considered the pair (h, d) before the pair (h', d) .
 - 2.2.8. So, when the algorithm considered the pair (h, d) , doctor d was not yet matched.
 - 2.2.9. By Steps 2.2.5 and 2.2.8, when the algorithm considered (h, d) , neither h nor d were matched.
 - 2.2.10. So, by inspection of the algorithm, it must have matched h to d in that iteration.
 - 2.2.11. But once a pair is matched, it stays matched, so (h, d) is in M .
 - 2.2.12. This is a contradiction, as no matched pair is unstable.
 - 2.3. By Block 2.2, (h, d) is not unstable for M .
 3. By Block 2, no pair is unstable for M . That is, M is stable. □

What is the first line in this proof that makes an assertion that is not necessarily true? Describe an example (an input and an execution of the algorithm on that input) for which the assertion fails.

Exercise 3.10. (*Another algorithm, proof problem 3*) Does the algorithm in Exercise 3.4 return a *stable* matching for every input with $n = 3$? If so, prove it. If not, give a counter-example: describe a particular Stable Matching instance with $n = 3$ and show that the algorithm does not give a stable matching for that instance.

Exercise 3.11. (*Gale-Shapley proof problem*) Prove or disprove the following assertion:

In any execution of the hospitals-propose Stable Matching algorithm, there is some doctor that does not receive an offer before the last iteration. (Note: we mean the algorithm described in the lecture notes—in each iteration, one hospital makes an offer to one doctor.)

If the assertion is false, describe a counter-example: a particular instance, and an execution of the algorithm on that instance, in which every doctor receives an offer before the start of the last iteration. If the assertion

is true, give a long-form proof. You may reuse without proof any observations or lemmas from the lecture notes (but cite the observation or lemma by number when you do, to let the reader know).

Exercise 3.12. (*Gale-Shapley run-time*) Let I be any instance of the Stable Matching problem in which all n hospitals have the exact same preferences. As a function of n , how many iterations will any execution of the Gale-Shapley algorithm (hospitals-propose, as described in the lecture notes) do on input I ? Explain your answer clearly.

Exercise 3.13. (*Gale-Shapley proof problem*) Prove or disprove the following assertion:

In every execution of the hospitals-propose Stable Matching algorithm, there is at most one hospital that makes offers to every doctor. (Note: we mean the algorithm described in the lecture notes—in each iteration, one hospital makes an offer to one doctor.)

If the assertion is false, describe a counter-example: a particular instance, and an execution of the algorithm on that instance, in which at least two hospitals make offers to every doctor. If the assertion is true, give a long-form proof. You may reuse without proof any observations or lemmas from the lecture notes (but cite the observation or lemma by number when you do, to let the reader know).

External exercises without solutions

- [JE](#) Chapter 4 (Page 179), Exercises 9–13
- [KT](#) (Page 22), Exercises 1–8
- [LLM](#) (2018) Page 207, Problems 6.16–6.32

Lecture Note 4

Divide and Conquer

[↑ CONTENTS](#)

Contents

4.1	Bounding sums	50
4.1.1	Naive upper and lower bounds	50
4.1.2	Geometric sums	50
4.1.3	Exercises	51
4.1.4	External sources on bounding sums	51
4.2	Recursion trees for recursive algorithms	52
4.2.1	Exercises	53
4.2.2	External sources on recursion trees and recursive algorithms	55
4.3	An algorithm for the Closest Pair problem	56
4.3.1	Exercises	57
4.3.2	External sources on Closest Pair	57
4.4	An algorithm for Large-Integer Multiplication	58
4.4.1	Exercise	59
4.4.2	External sources on Large-Integer Multiplication	59
4.5	An algorithm for Matrix Multiplication	60
4.5.1	Exercise	61
4.5.2	External sources on fast matrix multiplication	61
4.6	An algorithm for selecting the k th smallest	62
4.6.1	Exercise	64
4.6.2	External sources on deterministic selection	64
4.7	An incorrect algorithm for finding a 2D Local Maximum	65
4.7.1	1D Local Maximum	65
4.7.2	2D Local Maximum	65
4.7.3	Exercise	67
4.7.4	External sources on finding a local maximum	67
4.8	Additional exercises on Divide and Conquer	68

4.1 Bounding sums

Recall that, for example, $\sum_{i=1}^n i^2$ denotes the sum $1^2 + 2^2 + \dots + n^2$. You may have previously learned *closed forms* for some sums, such as $\sum_{i=1}^n i = n(n+1)/2$. Unfortunately, for most sums of interest, closed forms are difficult to find or simply don't exist. Fortunately, when we are analyzing asymptotic running times, we don't care about lower-order terms or constant factors, so we don't usually need exact closed forms for the sums that arise. Instead, for example, we only need to know that $\sum_{i=1}^n i = \Theta(n^2)$. Here are three rules for quickly estimating the big- Θ values of sums.

4.1.1 Naive upper and lower bounds

Here are two rules that work well for sums whose terms grow polynomially (or less) with the index.

Observation 4.1 (Naive upper bound). *Any sum is at most the number of terms times the max. term.*

Three examples:

$$\begin{aligned}\sum_{i=1}^n i &\leq n \cdot \max_{i=1}^n i = n \cdot n = O(n^2) \\ \sum_{i=1}^n i^2 &\leq n \cdot \max_{i=1}^n i^2 = n \cdot n^2 = O(n^3) \\ \sum_{i=1}^n \log i &\leq n \cdot \max_{i=1}^n \log i = n \log n = O(n \log n)\end{aligned}$$

Observation 4.2 (Naive lower bound). *Any sum with non-negative terms is at least half the number of terms times the median term.*

Three examples:

$$\begin{aligned}\sum_{i=1}^n i &\geq (n/2) \cdot (n/2) = \Omega(n^2) \\ \sum_{i=1}^n i^2 &\geq (n/2) \cdot (n/2)^2 = \Omega(n^3) \\ \sum_{i=1}^n \log i &\geq (n/2) \cdot \log(n/2) = \Omega(n \log n)\end{aligned}$$

For the three examples above, the naive upper and lower bounds match (up to constant factors). So together they are enough to determine the value of the sum, up to constant factors (the big- Θ value). For example, $\sum_{i=1}^n \log i = \Theta(n \log n)$.

The bounds will match for any sum whose terms don't increase (or decrease) too fast—specifically, for any sum where the median term is within a constant factor of the maximum term. For example, it will work for sums whose terms grow only polynomially with the index.

4.1.2 Geometric sums

However, it won't work for all sums. For example, try $\sum_{i=1}^n 2^i$. The naive upper bound gives $\sum_{i=1}^n 2^i = O(n2^n)$. The naive lower bound gives $\sum_{i=1}^n 2^i = \Omega(n2^{n/2})$. But these upper and lower bounds are far apart! Indeed their ratio $2^n/2^{n/2} = 2^{n/2}$ is quite large. The answer is actually $\sum_{i=1}^n 2^i = \Theta(2^n)$. Neither the naive upper or lower bound is tight in this case.

We have another rule for such sums. First we formally define the type of sums it applies to:

Definition 4.1. A sum $\sum_{i=1}^n \alpha_i$ is geometric if, for some constants $c > 1$ and i_0 (indep. of n) either
(i) for each $i \geq i_0$ the i th term is at least c times the previous term (that is, $\alpha_i \geq c \cdot \alpha_{i-1}$), or
(ii) for each $i \geq i_0$ the i th term is at most the previous term divided by c (that is, $\alpha_i \leq \alpha_{i-1}/c$).

Rule of thumb: if the sum has an exponential term in the summand, it is probably geometric.

Three examples

- $\sum_{i=1}^n 2^i$ is geometric: the ratio of successive terms $2^i/2^{i-1}$ is at least two.
- $\sum_{i=1}^n 2^i/i$ is also geometric: for $i \geq 3$ the ratio of successive terms is $(2^i/i)/(2^{i-1}/(i-1)) = 2(i-1)/i \geq 4/3$.
- $\sum_{i=1}^n i^2 + 1$ is not geometric: the ratio of successive terms $((i+1)^2 + 1)/(i^2 + 1)$ gets arbitrarily close to 1 for large i .

Here's our third rule. It handles any geometric sum:

Observation 4.3 (geometric sums). *Any geometric sum is proportional to its largest term.*

Three examples

$$\sum_{i=0}^n 2^i = \Theta(\max_{i=0}^n 2^i) = \Theta(2^n).$$

$$\sum_{i=1}^n 2^i/i = \Theta(\max_{i=1}^n 2^i/i) = \Theta(2^n/n).$$

$$\sum_{i=0}^{\infty} i^2/2^i = \Theta(\max_{i=0}^{\infty} i^2/2^i) = \Theta(1).$$

4.1.3 Exercises

Exercise 4.1. (*Bounding sums*) Use the so-called naive upper and lower bounds, and the rule for geometric sums, to determine the big- Θ value of each of the following sums.

(a) $\sum_{i=1}^n 1.1^i/i^2$

(b) $\sum_{i=1}^{\log_2 n} i^2 \log_3 i$

(c) $\sum_{i=1}^{2^n} (\log_3 i)^2$

(d) $\sum_{i=0}^n i^3/2^i$

4.1.4 External sources on bounding sums

[CLRS](#) Appendices A-1 and A-2

mergesort ($A[1..n]$)	
1. if $n = 1$, return	
2. set $m = \lfloor n/2 \rfloor$	
3. mergesort ($A[1..m]$)	(sort the first $n/2$ elements)
4. mergesort ($A[m+1..n]$)	(sort the last $n/2$ elements)
5. merge ($A[1..n], m$)	(merge the two sorted halves, in time $\Theta(n)$)

Figure 4.1: Mergesort. The **merge** procedure runs in linear time. We don't give its code.

	level	# of subprobs	subprob. size	work per subprob	work in level
	0	1	16	16	16
	1	2	8	8	16
	2	4	4	4	16
	3	8	2	2	16
	4	16	1	1	16
total:					5×16

Figure 4.2: On the left, the recursion tree for **mergesort** on an array of size 16. On the right, a table of the work done in each level of the recursion tree.

4.2 Recursion trees for recursive algorithms

Next we discuss how to analyze the run times of recursive algorithms, using **mergesort** (shown in Figure 4.1) as an example. Let $T(n)$ denote the worst-case running time of **mergesort** on any array of size n . Then $T(1) = \Theta(1)$. For $n \geq 2$, by inspection of **mergesort** and using that **merge** takes linear time, we have the recurrence relation

$$T(n) = 2T(n/2) + n.$$

(For now, we assume n is a power of 2, so $n/2$ is an integer.)

Using a recursion tree to count the work. A *recursion tree* represents the recursive structure of an execution of a recursive algorithm. The root node represents the top-level call. Each non-root node represents one recursive call. Each node is labeled by the size of the subarray that the corresponding call sorts. The children of each node for a given call represent the two recursive calls that that call makes. Figure 4.2 shows the recursion tree for **mergesort** on an array of size $n = 16$.

As illustrated in the table to the right in the figure, if the input array size n is a power of 2, there are $1 + \log_2 n$ levels. For each $i \in \{0, \dots, \log_2 n\}$, level i has 2^i subproblems, each of size $n/2^i$. The work associated with a subproblem of size $n/2^i$ is $\Theta(n/2^i)$, so the total work at level i is $\Theta(2^i \times n/2^i) = \Theta(n)$. Summing over the $\log n$ levels, the total work is $\Theta(n \log n)$. Hence, $T(n) = \Theta(n \log n)$. Note that the work is balanced evenly across the levels.

If n is not a power of 2, then there are $1 + \lceil \log_2 n \rceil$ levels. In each level i except the last, there are 2^i subproblems, each of size at most $\lceil n/2^i \rceil \leq 2n/2^i$. So, by a similar calculation, the total work $T(n)$ is also $\Theta(n \log n)$ in this case.

Theorem 4.1. *Mergesort runs in $O(n \log n)$ time.*

Variation with heavy leaves. Consider the recurrence relation $T(n) = 3 \cdot T(n/2) + n$. The same approach applies, but now each non-leaf node has **three** children. As before we have

$$T(n) = \sum_{i=0}^{\max \text{ level}} (\# \text{ subprobs on level } i) \times (\text{work per subprob. on level } i)$$

but now the number of subproblems on level i is 3^i , so the calculation proceeds as follows:

$$\begin{aligned} T(n) &= \sum_{i=0}^{\log_2 n} 3^i (n/2^i) = n \sum_{i=0}^{\log_2 n} (3/2)^i = \Theta(n (3/2)^{\log_2 n}) \quad (\text{because the sum is geometric}) \\ &= \Theta(n \cdot n^{\log_2 3/2}) = \Theta(n^{\log_2 3}) = \Theta(n^{1.58\dots}) \quad (\text{using } b^{\log n} = n^{\log b} \text{ and } \log_2 3/2 = \log_2 3 - 1). \end{aligned}$$

For mergesort, the work was balanced across the levels. But here we get a qualitatively different behavior: *the total work is dominated by the work done at the leaves.*

Variation with heavy root. Consider the recurrence relation $T(n) = 3T(n/2) + n^2$. Then

$$T(n) = \sum_{i=0}^{\log_2 n} 3^i (n/2^i)^2 = n^2 \sum_{i=0}^{\log_2 n} (3/4)^i = \Theta(n^2).$$

In this case *the total work is dominated by the work at the root.*

These are the three common cases: either (a) the work is balanced across the levels, or the sum is geometric, so either (b) the work at the leaves dominates, or (c) the work at the root dominates. A particularly important special case of the latter case (when the work at the root dominates) is when *the algorithm does linear work, then recurses on subproblems whose combined size is a constant factor less than the size of the given problem.*

For example, consider the recurrence relation $T(n) = aT(n/b) + n$, for some $a < b$. That is, the algorithm does linear work, then recurses on a problems, each of size n/b , so having total size $an/b = (a/b)n$ where $a/b < 1$. Then we get

$$T(n) = \sum_{i=0}^{\log_b n} a^i (n/b^i) = n \sum_{i=0}^{\log_b n} (a/b)^i = \Theta(n).$$

The last step follows because $a < b$, so the sum is geometric and dominated by the first term. See also the Selection algorithm in Section 4.6.

Python code: [divide_and_conquer/mergesort.py](#) (Section A.2).

4.2.1 Exercises

With solution

Exercise 4.2. (*Recursion-tree method*) For the algorithm below, use the recursion-tree method to find the asymptotic value (using the Θ -notation) for the number of letters printed by the algorithm as a function of n .

def PrintYs(n):	<i>Assume n is a power of 3.</i>
1. for $i = 1$ to $n \log_2 n$: print(“Y”)	<i>... print $n \log_2 n$ Y’s</i>
2. if $n > 1$:	
3. for $j = 1$ to 3: PrintYs($\lfloor n/3 \rfloor$)	<i>... make 3 recursive calls, each with input $\lfloor n/3 \rfloor$</i>

The solution should give

- (i) an appropriate recurrence relation,
- (ii) as a function of the level i in the recursion tree: the number of nodes (that is, calls) in level i , the problem size (the argument to each call) at each node in level i , and the number of letters printed by each call in level i ,
- (iii) the number of levels in the recursion tree,
- (iv) the sum, over all levels, of the number of letters printed by calls in that level, and finally,
- (v) a big- Θ estimate of the value of the sum (as a function of n), obtained by using either the rule for geometric sums or the naive upper and lower bounds (show your work for (v) if you want feedback).

Solution.

- (i) Define $Y(n)$ to be the number of letters printed by PRINTYS given input $n \geq 1$.
Then $Y(1) = 0$ and, for $n > 1$, $Y(n) = n \log_2 n + 3Y(n/3)$.
- (ii) In level i , there are 3^i nodes. Each call has input $n/3^i$ and prints $(n/3^i) \log_2(n/3^i)$ letters.
- (iii) There are $1 + \log_3 n$ levels.
- (iv) The total number of letters printed is $Y(n) = \sum_{i=0}^{\log_3 n} 3^i (n/3^i) \log_2(n/3^i) = \sum_{i=0}^{\log_3 n} n \log_2(n/3^i)$.
- (v) By the naive upper bound, the sum is at most $(1 + \log_3 n)n \log_2 n = O(n \log^2 n)$.
By the naive lower bound, the sum is at least $\frac{1}{2}(1 + \log_3 n)$ times the median term, which is $n \log_2(n/3^i)$ where $i = (1 + \log_3 n)/2$. For this i , 3^i simplifies to $\Theta(\sqrt{n})$, so the median term simplifies to $n \log_2(n/\Theta(\sqrt{n})) = n \log_2 \Theta(\sqrt{n}) = \Theta(n \log n)$. So the sum is $\Omega(n \log^2 n)$.
Thus, the sum, which equals $Y(n)$, is $\Theta(n \log^2 n)$.

Exercise without solutions

Exercise 4.3. (*Mergesort variation*) Consider a variant of Mergesort that, given an array of size N , recurses as usual just until the subproblems are of size $\sqrt{N}$, then solves each subproblem of size $\sqrt{N}$ directly using insertion sort (a quadratic-time algorithm). Use a recursion-tree analysis to determine the asymptotic worst-case running time of this algorithm on arrays of size n . Show your work as described for Exercise 4.2.

Exercise 4.4. (*Recursion-tree method*) Repeat Exercise 4.2 for each of the algorithms below. Give your answers in the same format as the solution given for that exercise. To make the calculations easier, ignore floors in your calculations. (That is, for (a), (c), and (e) assume n is a power of 3. For (b) assume n is a power of 4. For (d) assume n is a power of 2.)

- (a) `def PrintAs( $n$ ):` *Assume n is a power of 3.*
 - 1. `for  $i = 1$  to  $3n$ : print("A")` *... print $3n$ A's*
 - 2. `if  $n > 1$ :`
 - 3. `for  $j = 1$  to 4: PrintAs( $\lfloor n/3 \rfloor$ )` *... make 4 recursive calls, each with input $\lfloor n/3 \rfloor$*
- (b) `def PrintBs( $n$ ):` *Assume n is a power of 4.*
 - 1. `for  $i = 1$  to  $n^2$ : print("B")` *... print n^2 B's*
 - 2. `if  $n > 1$ :`
 - 3. `for  $j = 1$  to 16: PrintBs( $\lfloor n/4 \rfloor$ )` *... make 16 recursive calls, each with input $\lfloor n/4 \rfloor$*
- (c) `def PrintCs( $n$ ):` *Assume n is a power of 3.*
 - 1. `for  $i = 1$  to  $5n$ : print("C")` *... print $5n$ C's*
 - 2. `if  $n > 1$ :`
 - 3. `for  $j = 1$  to 2: PrintCs( $\lfloor n/3 \rfloor$ )` *... make 2 recursive calls, each with input $\lfloor n/3 \rfloor$*

- (d) `def PrintDs( $n$ ):` *Assume n is a power of 2.*
 1. `for  $i = 1$  to 11: print("D")` *... print 11 D's*
 2. `if  $n > 1$ :`
 3. `for  $j = 1$  to 2: PrintDs( $\lfloor n/2 \rfloor$ )` *... make 2 recursive calls, each with input $\lfloor n/2 \rfloor$*
- (e) `def PrintEs( $n$ ):` *Assume n is a power of 3.*
 1. `for  $i = 1$  to  $2n^3$ : print("E")` *... print $2n^3$ E's*
 2. `if  $n > 1$ :`
 3. `for  $j = 1$  to 9: PrintEs( $\lfloor n/3 \rfloor$ )` *... make 9 recursive calls, each with input $\lfloor n/3 \rfloor$*

4.2.2 External sources on recursion trees and recursive algorithms

- [DPV](#) Chapter 2.2; [KT](#) Chapter 5; [CLRS](#) Chapter 4 (especially 4.4); [LLM](#) Chapter 22;
- [JE](#) Chapter 1, and Appendix II Section 3 (& exercises starting on Page 20), at <https://jeffe.cs.illinois.edu/teaching/algorithms/notes/99-recurrences.pdf>
- An MIT lecture video
 - <https://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-006-introduction-to-algorithms-fall-2011/lecture-videos/lecture-3-insertion-sort-merge-sort/>

4.3 An algorithm for the Closest Pair problem

Problem definition. The Closest Pair problem is defined as follows:

input: 2D points $P = (p_1, p_2, \dots, p_n)$ where each $p_i = (x_i, y_i) \in \mathbb{R}^2$

output: the minimum Euclidean distance $\delta = \min\{d(p_i, p_j) : 1 \leq i < j \leq n\}$ between any two of the points. Here $d((x_i, y_i), (x_j, y_j)) = \sqrt{(x_i - x_j)^2 + (y_i - y_j)^2}$. For $n \leq 1$, the answer is ∞ .

Algorithm. The obvious algorithm (compare all points) takes time $O(n^2)$. We want $O(n \log n)$.

We'll assume the points are sorted by x-coordinate. If not, we can sort them in $O(n \log n)$ time in a pre-processing step. To keep things simple, for now, we also assume that the x-coordinates are distinct. Our goal is to use a divide-and-conquer approach. What about this algorithm?

closest-pair-1 ($P = (p_1, p_2, \dots, p_n)$)	$(p_i = (x_i, y_i) \in \mathbb{R}^2, \text{ with } x_1 < x_2 < \dots < x_n)$
1. if $n \leq 1$: return ∞ 2. if $n = 2$: return $d(p_1, p_2)$ 3. let $m = \lfloor n/2 \rfloor$ 4. let $L = (p_1, p_2, \dots, p_m)$; let $R = (p_{m+1}, \dots, p_n)$ (<i>partition P into left and right halves by x-coordinate</i>) 5. let $\delta_L = \text{closest-pair-1}(L)$; let $\delta_R = \text{closest-pair-1}(R)$ 6. let $\delta = \min(\delta_L, \delta_R)$ 7. return δ	
(WRONG! See below)	

The running time of this algorithm satisfies $T(n) = 2T(n/2) + O(n)$, so is $O(n \log n)$, as desired. But the algorithm is wrong, because it doesn't consider the pairs (p_i, p_j) such that $i \leq m < j$. That is, $(p_i, p_j) \in L \times R$. To fix it, we need to consider those pairs.

Among those pairs, it is enough to consider only those with $d(p_i, p_j) < \delta$ (for δ from Step 6). (Why?)

Let $X = x_m$ be the median x-coordinate (separating L from R) and let $M = \{p_i : |x_i - X| \leq \delta\}$ contain the points lying in a vertical strip of width 2δ centered at X . Any point $p_i \in L \setminus M$ (that is, in L but not M) has distance at least δ from any point in R . Likewise any $p_i \in R \setminus M$ has distance at least δ from any point in L . So we only need to consider pairs $(p_i, p_j) \in M \times M$.

[Draw a picture to help intuition.]

Further, within M , it is enough to compare each point p_i in M only to those points p_j above p_i in M such that $y_i \leq y_j < y_i + \delta$. (Any other point above p_i in M has distance at least δ to p_i .) And there are at most eight such points:

Lemma 4.1. Assume δ_L and δ_R are calculated correctly. For any point $p_i \in M$, there are at most 8 points p_j above p_i in M such that $y_i \leq y_j < y_i + \delta$.

Proof (short form). We use that any $\delta \times \delta$ square contains at most 4 points in L and at most 4 points in R . To prove this observation, imagine partitioning the square into its four $(\delta/2) \times (\delta/2)$ quadrants. Within each quadrant, every pair of points has distance at most $\sqrt{(\delta/2)^2 + (\delta/2)^2} = \delta/\sqrt{2} < \delta$. So R has at most one point in each quadrant (otherwise R would have a pair with distance less than $\delta \leq \delta_R$, contradicting the correctness of δ_R). Likewise for L .

Now, consider any $p_i \in M$. Consider the two $\delta \times \delta$ squares $S_L = [X - \delta, X] \times [y_i, y_i + \delta]$ and $S_R = [X, X + \delta] \times [y_i, y_i + \delta]$. Any point p_j in M with $y_i \leq y_j < y_i + \delta$ must lie in either S_L or S_R . S_L contains no points from R , and (by the observation above) contains at most four points from L . So S_L contains at most four points from P . Likewise S_R contains at most four points from P . The lemma follows. $\square$

Summarizing, it is enough to compare each point p_i in M with the points p_j in M such that $y_i \leq y_j < y_i + \delta$, and there are at most eight such points (per p_i). To fix the algorithm, we have it sort the points in M by y-coordinate, then have it compare each point p_i in M to the points p_j in M such that $y_i \leq y_j < y_i + \delta$:

<code>closest-pair($P = (p_1, p_2, \dots, p_n)$)</code>	$(p_i = (x_i, y_i) \in \mathbb{R}^2, \text{ with } x_1 < x_2 < \dots < x_n)$
1. if $n \leq 1$: return ∞	
2. if $n = 2$: return $d(p_1, p_2)$	
3. let $m = \lfloor n/2 \rfloor$	
4. let $L = (p_1, p_2, \dots, p_m)$; let $R = (p_{m+1}, \dots, p_n)$	(partition P into left and right halves by x -coordinate)
5. let $\delta_L = \text{closest-pair}(L)$; let $\delta_R = \text{closest-pair}(R)$	
6. let $\delta = \min(\delta_L, \delta_R)$	
7. let $M = \{p_i : x_i - x_m \leq \delta\}$	(compute this in $O(n)$ time)
8. let $\delta_M = \min\{d(p_i, p_j) : p_i, p_j \in M, i \neq j, y_i \leq y_j \leq y_i + \delta\}$.	(compute this as described in the text below)
9. return $\min(\delta, \delta_M)$	

In Step 8, Compute δ_M efficiently as follows. Sort the points in M by y -coordinate, then enumerate the points p_i in M in order of increasing y -coordinate. For each such p_i , consider the points p_j in the list after p_i , such that $y_j \leq y_i + \delta$. There will be at most eight of these for each p_i , so the total time (after sorting) is $O(n)$.

As described, this would give time $T(n) = 2T(n/2) + \Theta(n \log n)$, which gives $T(n) = \Theta(n \log^2 n)$.

To save a $\log n$ factor, instead of sorting the points in M by y -coordinate within each recursive call, do the following. Along with P , pass in a second list P' of the points in P , this one sorted by y -coordinate. From P' , the sorted list of points in M can be computed in $O(n)$ time. For the recursive calls on L and R , let L' and R' denote the lists L and R sorted by y -coordinate. Compute each of these (from P' again) in time $O(n)$, then pass L' with L , and pass R' with R , to each recursive call.

The time to preprocess all the points to sort them by x and y coordinate (before any recursion) is $O(n \log n)$. The remaining time satisfies $T(n) = 2T(n/2) + O(n)$, which gives $T(n) = O(n \log n)$.

Theorem 4.2. *There is an $O(n \log n)$ -time algorithm for the Closest Pair problem.*

Above we assumed that all points have distinct x -coordinates. Exercise 4.6 asks you to consider what happens if that's not the case.

Python code: [divide_and_conquer/closest_pair.py](#) (Section A.2).

4.3.1 Exercises

Exercise 4.5. (*implementation*) Implement the algorithm and test your implementation on Hacker-rank.com, at <https://www.hackerrank.com/neaileyong-algs101-closest-pair>.

Solution. Python code: [divide_and_conquer/closest_pair.py](#) (Section A.2).

Exercise 4.6. (*Closest Pair*) In analyzing the Closest Pair algorithm in Section 4.3, we assumed that each point had a unique x -coordinate. On inputs where that's not the case, can the algorithm give an incorrect output, or fail to run in time $O(n \log n)$? If so, how would you fix the algorithm to deal with such inputs?

4.3.2 External sources on Closest Pair

- [DPV](#) Problem 2.32; [KT](#) Chapter 5.4; [CLRS](#) Chapter 9.2

4.4 An algorithm for Large-Integer Multiplication

Problem definition

input: n -bit integers A and B , with binary representations $A = a_1a_2 \dots a_n$ and $B = b_1b_2 \dots b_n$.
output: the product $C = AB$.

This problem is about multiplying integers that are too large to fit in a machine word. For example, RSA encryption currently requires arithmetic on 2048-bit integers, or larger, while current machine words are typically 64 bits.

Algorithm—first attempt. The standard grade-school algorithm takes time $\Theta(n^2)$. We want something faster. We start by considering the following recursive algorithm:

int-product-1($n, A = a_1a_2 \dots a_n, B = b_1b_2 \dots b_n$) (A and B are n -bit integers; n is a power of two)

1. if $n = 1$: return a_1b_1 (base case)
2. split A into $A_1 = a_1 \dots a_{n/2}$ and $A_2 = a_{n/2+1} \dots a_n$ so that $A = 2^{n/2}A_1 + A_2$.
3. split B into $B_1 = b_1 \dots b_{n/2}$ and $B_2 = b_{n/2+1} \dots b_n$ so that $B = 2^{n/2}B_1 + B_2$.
4. return $2^n(A_1 \times B_1) + 2^{n/2}(A_1 \times B_2 + A_2 \times B_1) + (A_2 \times B_2)$

The last step requires four multiplications of $n/2$ -bit integers. *These are done recursively!*

The two multiplications by powers of 2 are done directly just by shifting the bits.

Correctness. Correctness of the algorithm follows by induction from the equation

$$\begin{aligned} A \times B &= (2^{n/2}A_1 + A_2) \times (2^{n/2}B_1 + B_2) \\ &= 2^n(A_1 \times B_1) + 2^{n/2}(A_1 \times B_2 + A_2 \times B_1) + (A_2 \times B_2). \end{aligned} \quad (4.1)$$

Because of this equation, as long as the four sub-products are calculated correctly (and the additions are done correctly) the result returned will be correct.

Run time. Each addition can be done using the standard algorithm in $O(n)$ time. And each multiplication by a power of two just shifts the bits, so can be done in $O(n)$ time. So the running time satisfies $T(n) = 4T(n/2) + n$. The recursion-tree approach gives

$$\begin{aligned} T(n) &= \sum_{i=0}^{\# \text{levels}} (\# \text{ subproblems in level } i) \times (\text{work per subproblem in level } i) \\ &= \sum_{i=0}^{\log_2 n} (4^i)(n/2^i) = n \sum_{i=0}^{\log_2 n} 2^i = n \Theta(2^{\log_2 n}) = \Theta(n^2). \end{aligned}$$

This isn't any faster than the standard algorithm! Can we do better?

Karatsuba's algorithm. The middle term in Step 4, namely $A_1 \times B_2 + A_2 \times B_1$, satisfies

$$A_1 \times B_2 + A_2 \times B_1 = (A_1 + A_2) \times (B_1 + B_2) - A_1 \times B_1 - A_2 \times B_2. \quad (4.2)$$

The terms $A_1 \times B_1$ and $A_2 \times B_2$ are computed anyway for Step 4, so by computing the right-hand side with this equation, we replace two recursive calls by *one*. This gives us Karatsuba's algorithm:

karatsuba($n, A = a_1a_2 \dots a_n, B = b_1b_2 \dots b_n$) (A and B are n -bit integers; n is a power of two)

1. if $n = 1$: return a_1b_1 (base case)
2. split A into $A_1 = a_1 \dots a_{n/2}$ and $A_2 = a_{n/2+1} \dots a_n$ so that $A = 2^{n/2}A_1 + A_2$.
3. split B into $B_1 = b_1 \dots b_{n/2}$ and $B_2 = b_{n/2+1} \dots b_n$ so that $B = 2^{n/2}B_1 + B_2$.
4. let $M_1 = \text{karatsuba}(n/2, A_1, B_1)$ ($M_1 = A_1 \times B_1$)
5. let $M_2 = \text{karatsuba}(n/2, A_2, B_2)$ ($M_2 = A_2 \times B_2$)
6. let $M_3 = \text{karatsuba}(n/2, A_1 + A_2, B_1 + B_2)$ ($M_3 = (A_1 + A_2) \times (B_1 + B_2)$)
7. return $2^n M_1 + 2^{n/2}(M_3 - M_1 - M_2) + M_2$

Correctness. Correctness follows by induction from Equations (4.1) and (4.2).

Run time. Recalling that each addition or multiplication by a power of two can be done in $O(n)$ time, the running time now satisfies $T(n) = 3T(n/2) + n$, which implies

$$T(n) = \sum_{i=0}^{\log_2 n} (3^i)(n/2^i) = n \sum_{i=0}^{\log_2 n} (3/2)^i = n \Theta((3/2)^{\log_2 n}) = \Theta(n^{\log_2 3}) = \Theta(n^{1.58\dots}).$$

(Above we use $x^{\log y} = y^{\log x}$ and $\log_2(3/2) = \log_2(3) - 1$.) We have the following result:

Theorem 4.3 (Karatsuba). *There is an $O(n^{\log_2 3})$ -time algorithm for multiplying n -bit integers.*

Note: A recent (2019) algorithm achieves run time $O(n \log n)$.

4.4.1 Exercise

Exercise 4.7. (*Karatsuba's variation*) The sums $A_1 + A_2$ and $B_1 + B_2$ may have $n/2 + 1$ bits due to carry. This complicates the algorithm and the recurrence relation. Professor Bo Zo proposes the following variant to fix this. Does it work?

int-product-2($n, A = a_1a_2 \dots a_n, B = b_1b_2 \dots b_n$)	(A and B are n -bit integers; n is a power of two)
1. if $n = 1$: return a_1b_1	(base case)
2. split A into $A_1 = a_1 \dots a_{n/2}$ and $A_2 = a_{n/2+1} \dots a_n$ so that $A = 2^{n/2}A_1 + A_2$.	
3. split B into $B_1 = b_1 \dots b_{n/2}$ and $B_2 = b_{n/2+1} \dots b_n$ so that $B = 2^{n/2}B_1 + B_2$.	
4. let $M_1 = \text{int-product-2}(n/2, A_1, B_1)$	($M_1 = A_1 \times B_1$)
5. let $M_2 = \text{int-product-2}(n/2, A_2, B_2)$	($M_2 = A_2 \times B_2$)
6. let $M_3 = \text{int-product-2}(n/2, A_2 - A_1 , B_2 - B_1)$	← CHANGED THIS LINE AND BELOW!
7. let $s_A = \text{sign}(A_2 - A_1)$ and $s_B = \text{sign}(B_2 - B_1)$	
8. return $2^n M_1 + 2^{n/2}(M_1 + M_2 - s_A s_B M_3) + M_2$	

4.4.2 External sources on Large-Integer Multiplication

- DPV Chapter 2.1, KT Chapter 5.5
- JE Chapter 1, Section 1.9 (has information about the 2019 $O(n \log n)$ -time algorithm)
- An MIT lecture video
- <https://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-006-introduction-to-algorithms-fall-2011/lecture-videos/lecture-11-integer-arithmetic-karatsuba-multiplication/>

4.5 An algorithm for Matrix Multiplication

Next we describe a similar algorithm for Matrix Multiplication.

Problem definition

input: $n \times n$ matrices A and B .

output: the matrix product $C = A \times B$. (So $c_{ij} = \sum_{k=1}^n a_{ik}b_{kj}$.)

We assume throughout that we are working with 64-bit integers, so multiplication and addition of two numbers can be done in constant time. We also assume n is a power of 2 (we can reduce the general case to this one).

Algorithm—first attempt. The obvious algorithm takes time $O(n^3)$. We want time $o(n^3)$.

To warm up, consider the following recursive algorithm. Let $A_{11}, A_{12}, A_{21}, A_{22}, B_{11}, B_{12}, B_{21}, B_{22}, C_{11}, C_{12}, C_{21}$ and C_{22} be $\frac{n}{2} \times \frac{n}{2}$ submatrices such that

$$A = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix}, \quad B = \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix}, \quad \text{and } C = \begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix}.$$

We are given A and B , and want to compute C . The following matrix equation holds for each of the four pairs $i, j \in \{1, 2\}$:

$$C_{ij} = A_{i1} \times B_{1j} + A_{i2} \times B_{2j}. \quad (4.3)$$

This gives the following recursive algorithm:

matrix-product-1 (n, A, B)	(A and B are $n \times n$ matrices; n is a power of two)
1. if $n = 1$: return $[a_{11}b_{11}]$	(base case)
2. for each of the four pairs $i, j \in \{1, 2\}$, compute A_{ij} and B_{ij} as defined above	(time $O(n^2)$)
3. for each of the four pairs $i, j \in \{1, 2\}$:	
4.1. let $C_{ij} = \text{matrix-product-1}(n/2, A_{i1}, B_{1j}) + \text{matrix-product-1}(n/2, A_{i2}, B_{2j})$	
5. return $C = \begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix}$	

Correctness. Correctness of the algorithm follows from Equation (4.3), by induction.

Running time. To compute the product of two $n \times n$ matrices, the algorithm makes eight recursive calls on $(n/2) \times (n/2)$ matrices. The time spent outside of the recursive calls is $O(n^2)$. So the running time $T(n)$ satisfies $T(n) = 8T(n/2) + O(n^2)$. Using the recursion-tree method, the total time is proportional to

$$T(n) = \sum_{i=0}^{\log_2 n} (8^i) \times (n/2^i)^2 = n^2 \sum_{i=0}^{\log_2 n} 2^i = n^2 \Theta(2^{\log_2 n}) = \Theta(n^3).$$

No better!

Strassen's Algorithm. To improve the running time, we'll reduce the number of recursive calls from eight to seven. To do this, replace Steps 3 and 4.1 with the following steps. First, compute the following seven $\frac{n}{2} \times \frac{n}{2}$ matrices, using seven recursive matrix multiplications:

$$\begin{aligned} M_1 &= (A_{11} + A_{22}) \times (B_{11} + B_{22}) & M_2 &= (A_{21} + A_{22}) \times B_{11} \\ M_3 &= A_{11} \times (B_{12} - B_{22}) & M_4 &= A_{22} \times (B_{21} - B_{11}) \\ M_5 &= (A_{11} + A_{12}) \times B_{22} & M_6 &= (A_{21} - A_{11}) \times (B_{11} + B_{12}) \\ M_7 &= (A_{12} - A_{22}) \times (B_{21} + B_{22}) \end{aligned}$$

Strassen figured out that those seven matrices satisfy the following equations:

Lemma 4.2 (Strassen).

$$C_{11} = M_1 + M_4 - M_5 + M_7$$

$$C_{12} = M_3 + M_5$$

$$C_{21} = M_2 + M_4$$

$$C_{22} = M_1 - M_2 + M_3 + M_6$$

We don't give the proof, but you can check each equation if you like.

Now we modify the Algorithm in Step 4.1 so it computes each C_{ij} using the four equations above in time $O(n^2)$. This gives Strassen's algorithm.

Correctness. Correctness of the algorithm follows by the lemma (and induction on n).

Run time. The time now $T(n)$ now satisfies $T(n) = 7T(n/2) + O(n^2)$, so the total time is proportional to

$$\sum_{i=0}^{\log_2 n} (7^i) \times (n/2^i)^2 = n^2 \sum_{i=0}^{\log_2 n} (7/4)^i = n^2 \Theta((7/4)^{\log_2 n}) = \Theta(n^{\log_2 7}) = \Theta(n^{2.80\dots}).$$

(Above we use $x^{\log y} = y^{\log x}$ and $\log_2(7/4) = \log_2(7) - 2$.) We have the following result:

Theorem 4.4 (Strassen). *There is an $O(n^{\log_2 7})$ -time algorithm for multiplying $n \times n$ matrices.*

4.5.1 Exercise

Exercise 4.8. (*Strassen variation*) The standard way of multiply two 3×3 matrices uses 27 multiplications. Professor Bo Zo found a way to do it using 24 multiplications and a constant number of additions (without assuming that multiplication is commutative). He uses this observation as the basis for a variant of Strassen's algorithm that makes 24 recursive calls, each multiplying two matrices of size $n/3 \times n/3$. Does he get a faster algorithm? Show your work!

4.5.2 External sources on fast matrix multiplication

- CLRS Chapter 4.2 (includes exercises), and an MIT lecture video:
- <https://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-046j-introduction-to-algorithms-sma-5503-fall-2005/video-lectures/lecture-3-divide-and-conquer-strassen-fibonacci-polynomial-multiplication/>

4.6 An algorithm for selecting the k th smallest

Problem definition. The Selection problem is defined as follows:

input: array $A[1..n]$ of numbers, integer k in $\{1, \dots, n\}$.

output: the k th smallest number in the array

When $k = 1$, the output is $\min_i A[i]$. When $k = n$, the output is $\max_i A[i]$. When $k = n/2$, the output is the median.

Algorithm. One simple way to solve the problem is to sort A , then return $A[k]$. This takes $\Theta(n \log n)$ time. Next we present a faster, linear-time algorithm. We assume for simplicity that all elements in A are distinct.

```

select( $A[1..n], k$ )
1. if  $n \leq 15$ , sort  $A$  in place and return  $A[k]$                                 (since  $n \leq 15$ , this takes constant time)
2. choose pivot  $p \in A[1..n]$  as described in the text below
3. partition  $A$  around  $p$  into  $A[1..i-1]$ ,  $A[i] = p$ , and  $A[i+1..n]$ 
   (so that elements smaller than  $p$  are in  $A[1..i-1]$  and elements larger than  $p$  are in  $A[i+1..n]$ )
4. if  $k = i$ : return  $A[i]$ 
5. else if  $k < i$ : return select( $A[1..i-1], k$ )                                (discard  $A[i..n]$ )
6. else ( $k > i$ ): return select( $A[i+1..n], k-i$ )                            (discard  $A[1..i]$ )

```

Correctness. We omit the proof. The idea is that if $k < i$, then the k th smallest in $A[1..n]$ (after Step 3) is the k th smallest in $A[1..i-1]$, and if $k > i$ it is the $k-i$ th smallest in $A[i+1..n]$. Correctness follows from this and induction.

Run time. Step 3 (the partition step) takes linear time. After that, **select** discards some elements of A and calls itself recursively on the rest. If the pivot is chosen badly, this could lead to quadratic running time. For example, if $k = 1$ and the pivot is chosen as the maximum element each time, then each recursive call would discard just one element, so there would be $n-1$ recursive calls, with the i th call taking time proportional to $n-i$, yielding total time $\Theta(\sum_{i=0}^n n-i) = \Theta(n^2)$.

To avoid this, the algorithm uses a complicated scheme to choose the pivot in Step 2. It first partitions the array A into $n/5$ columns, each with 5 elements, as shown below (the diagram assumes n is a multiple of 2 and 5, that is, a multiple of 10):

$A[1]$	$A[6]$		$A[n/2-4]$		$A[n-4]$
$A[2]$	$A[7]$		$A[n/2-3]$		$A[n-3]$
$A[3]$	$A[8]$	$\dots$	$A[n/2-2]$	$\dots$	$A[n-2]$
$A[4]$	$A[9]$		$A[n/2-1]$		$A[n-1]$
$A[5]$	$A[10]$		$A[n/2]$		$A[n]$
$\underbrace{\hspace{15em}}_{n/5}$					

It then finds the median within each individual column of 5. This takes constant time for each column, so the total time for this step is linear. There are $n/5$ of these medians. It then *recursively computes the median of these $n/5$ medians, and chooses the pivot p to be the result*. This completes the description of the algorithm. To analyze the run time, we use the following lemma:

Lemma 4.3. *The rank i of the pivot satisfies $3n/10 \leq i \leq 7n/10$, so the recursive call in Line 5 or 6 has at most $7n/10$ elements.*¹

Proof (short form). The pivot p , being the median of the $n/5$ column medians, is larger than $n/10$ of those medians. And each such median is in turn larger than 2 elements in its column. So p is larger than at least

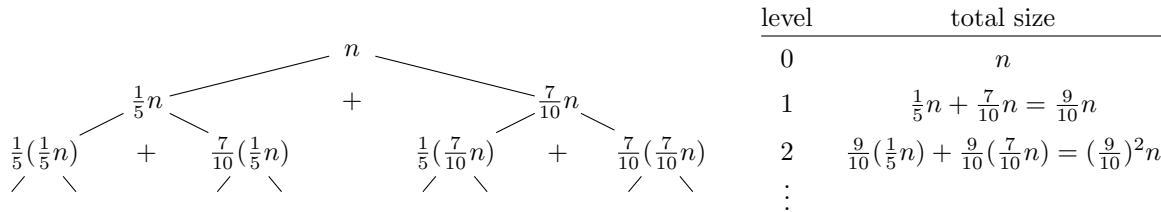
¹This observation and some of the following discussion is not necessarily true when n is not a multiple of 10, because we are ignoring rounding issues. See the discussion at the end of the note.

$3n/10$ other elements. By similar reasoning, p is smaller than at least $3n/10$ other elements. So at least $3n/10$ elements are discarded in the recursive call. $\square$

Define $T(n)$ to be the worst-case running time of **select** on n elements. The above reasoning gives (for some constant $c > 0$) the inequality

$$T(n) \leq c \cdot n + T(n/5) + T(7n/10).$$

The next crucial observation is that sum of the subproblem sizes is $n/5 + 7n/10 = 9n/10$, so the total problem size drops by a factor of $9/10$:



This happens at each level of the recursion tree. For each subproblem of any size n' , its two subproblems on the level below have combined size at most $\frac{9}{10}n'$, so the total size at each level is $\frac{9}{10}$ th of the total size at the level above. So, by induction, the sum of the problem sizes in level i is at most $(9/10)^i n$. Since the *work* for each subproblem is linear in the subproblem size, the total work in level i is also $O((9/10)^i n)$. So the total work is $O(\sum_{i=0}^{O(\log n)} (9/10)^i n) = O(n)$ (using here that the sum is geometric).

Correcting for rounding issues. The previous discussion is wrong when n is not a multiple of 10, because it ignores rounding issues. For completeness, we give a more careful, correct analysis, starting with the choice of the pivot. When n is not a multiple of 10, to choose the pivot p , the algorithm partitions into columns of 5 something like this:

1	2	...	$m = \lfloor n/10 \rfloor$	...	$\lceil n/5 \rceil$
$A[1]$	$A[6]$	...	$A[5m-4]$	...	empty
$A[2]$	$A[7]$		$A[5m-3]$	...	empty
$A[3]$	$A[8]$	...	$A[5m-2]$	...	$A[n-1]$
$A[4]$	$A[9]$	...	$A[5m-1]$	...	$A[n]$
$A[5]$	$A[10]$	...	$A[5m]$	...	empty

There are $\lceil n/5 \rceil$ columns. The last column has between 1 and 5 elements. After finding the $\lceil n/5 \rceil$ medians of those columns in $\Theta(n)$ time the algorithm recursively finds the median of those medians—specifically, the m th smallest, where $m = \lfloor n/10 \rfloor$, and takes that to be the pivot p .

In the case $k > i$ in Step 6 of the algorithm, it discards p and all elements smaller than p . This will be at least $3m - 2$ elements (not necessarily $3m$, because the last column may have fewer than three elements). Recalling that $m = \lfloor n/10 \rfloor$, in this case at least $3\lfloor n/10 \rfloor - 2$ elements are discarded. In the other case ($k < i$ in Step 5), the algorithm discards p and all elements larger than p . The number of such elements is at least

$$\begin{aligned}
 3(\lceil n/5 \rceil - m + 1) - 2 &= 1 + 3(\lceil n/5 \rceil - m) \\
 &= 1 + 3(\lceil n/5 \rceil - \lfloor n/10 \rfloor) \\
 &\geq 1 + 3(n/5 - n/10) \\
 &= 1 + 3n/10 \geq 3\lfloor n/10 \rfloor.
 \end{aligned}$$

Hence the number of remaining elements in the recursive call is at most $n - 3\lfloor n/10 \rfloor + 2$, and

$$T(n) \leq n + T(\lceil n/5 \rceil) + T(n - 3\lfloor n/10 \rfloor + 2).$$

Lemma 4.4. $T(n) = O(n)$.

Proof. We give a proof by induction that $T(n) \leq cn$ for some (large) constant c .

The base cases $n \leq 80$ will hold as long as $c \geq \max\{T(n)/n : n \leq 80\}$.

For $n > 80$, we have

$$\begin{aligned}
 T(n) &\leq n + T(\lceil n/5 \rceil) + T(n - 3\lfloor n/10 \rfloor + 2) && \text{(see above)} \\
 &\leq n + c \lceil n/5 \rceil + c(n - 3\lfloor n/10 \rfloor + 2) && \text{(by induction, using that } n > 80\text{)} \\
 &< n + c(n/5 + 1 + n - 3(n/10 - 1) + 2) && \text{(using } \lceil x \rceil < x + 1 \text{ and } \lfloor x \rfloor > x - 1\text{)} \\
 &= cn(1/5 + 6/n + 9/10) && \text{(algebra)} \\
 &\leq cn(1/5 + 3/40 + 9/10) && \text{(using } n \geq 80\text{)} \\
 &\leq cn && \text{(as long as } c \geq 40\text{).}
 \end{aligned}$$

It follows that $T(n) \leq cn$, where $c = \max(40, \max\{T(n)/n : n \leq 80\})$. □

We have proved the following theorem:

Theorem 4.5. *There is a (deterministic) linear-time algorithm for Selection (and Median Finding).*

Above we assumed that all numbers in the array are distinct. Exercise 4.9 asks you to consider what happens if that's not the case.

Python code: [divide_and_conquer/selection.py](#) (Section A.2).

4.6.1 Exercise

Exercise 4.9. (*Selection algorithm*) In analyzing the Selection algorithm in Section 4.6, we assumed that each number was distinct. On inputs where that's not the case, can the algorithm give an incorrect output, or fail to run in time $O(n)$? If so, how would you fix the algorithm to deal with such inputs?

4.6.2 External sources on deterministic selection

- [JE](#) Chapter 1, Section 1.8 (& exercises); [CLRS](#) Chapter 9.3 (& exercises); MIT lecture video: <https://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-046j-introduction-to-algorithms-sma-5503-fall-2005/video-lectures/lecture-6-order-statistics-median/>
- [DPV](#) Ch. 2.4 and [KT](#) Ch. 1.5 discuss a simpler randomized algorithm (random pivot)

4.7 An incorrect algorithm for finding a 2D Local Maximum

4.7.1 1D Local Maximum

For warm-up, we first consider the 1-dimensional problem.

Problem definition. The 1D Local Maximum problem is defined as follows:

input: An array $A[1..n]$ of numbers.

output: A *local maximum*, which is a cell $A[i]$ such that every neighboring cell in the array has a value that is at most $A[i]$. (The neighboring cells are those among $A[i-1]$ and $A[i+1]$ that are actually present in A .)

Algorithm. There is a linear-time (that is, $O(n)$ -time) algorithm: just check each cell.

Here is an $O(\log n)$ -time algorithm using divide and conquer:

```

local-max-1D( $A[1..n]$ )
1. if  $n = 1$ , return  $A[1]$ 
2. let  $m = \lceil n/2 \rceil$ 
3. if  $A[m]$  is a local maximum of  $A[1..n]$ , return  $A[m]$ 
4. if  $A[m+1] > A[m]$ , return local-max-1D( $A[m+1..n]$ )
5. else ( $A[m-1] > A[m]$ ) return local-max-1D( $A[1..m-1]$ )
    
```

Running time. The running time $T(n)$ satisfies $T(n) = O(1) + T(n/2)$. So $T(n) = O(\log n)$.

Correctness.

Lemma 4.5. *The 1D algorithm is correct.*

Proof (long form).

1. The proof is by induction on n .
2. Let $P(n)$ be the predicate “the algorithm is correct on all inputs of size n ”.
3. By inspection of algorithm Line 1 and the definition of local maximum, $P(1)$ holds.
 - 4.1. Consider any execution of the algorithm on any input A with $n \geq 2$. Assume $P(i)$ for $1 \leq i < n$.
 - 4.2.1. Case 1. Suppose $A[m]$ is a local maximum of $A[1..n]$.
 - 4.2.2. By inspection of algorithm Line 3, the algorithm is correct on this input.
 - 4.3.1. Case 2. Otherwise, suppose $A[m+1] > A[m]$.
 - 4.3.2. Then any local maximum of $A[m+1..n]$ is also a local maximum of $A[1..n]$.
(Note: in the case that the local max is $A[m+1]$, this follows from $A[m+1] > A[m]$.)
 - 4.3.3. So (by inspection of Line 4), the algorithm is correct on this input.
 - 4.4.1. Case 3. In the remaining case, $A[m-1] > A[m]$.
 - 4.4.2. Then any local maximum of $A[1..m-1]$ is also a local maximum of $A[1..n]$.
 - 4.4.3. So (by inspection of Line 5), the algorithm is correct on this input.
 - 4.5. One of the three cases holds, so the algorithm is correct on this input.
5. By Block 4, $P(n)$ holds as long as $P(i)$ holds for $1 \leq i < n$.
6. Since $P(1)$ holds, by induction $P(n)$ holds for all $n \geq 1$. □

4.7.2 2D Local Maximum

Next we consider two dimensions.

Problem definition. The 2D Local Maximum problem is defined as follows:

input: An $n \times n$ array $A[1..n, 1..n]$ of numbers.

output: A *local maximum*, which is a cell $A[i, j]$ such that every neighboring cell in the array has a value that is at most $A[i, j]$. (The neighboring cells are those among $A[i, j+1]$, $A[i, j-1]$, $A[i-1, j]$, $A[i+1, j]$ that are actually present in A .)

Note that the neighbors of a local max $A[i, j]$ can have the *same* value as $A[i, j]$. For example, in the array to the right, the local maxima are in bold.

0	0	0	0	0	0	0
0	1	1	0	3	4	0
3	2	0	1	3	1	0
2	5	0	3	0	1	0
4	1	0	0	1	3	0
3	3	0	1	2	1	0
0	0	0	0	0	0	0

Algorithm. There is a linear-time (that is, $O(n^2)$ -time) algorithm: just check each cell. We want a faster algorithm. We propose the following algorithm. Assume for simplicity that n is of the form $2^i - 1$ for some integer i . (The “cross” X in Step 2 below is shown in the array above.)

- We assume $n \geq 1$.*

local-max($A[1..n, 1..n]$)

 1. Let X be the set consisting of the $2n - 1$ cells lying in column $(n + 1)/2$ or row $(n + 1)/2$ of A .
 2. Check each cell in X to see if it is a local maximum in A . If so, return such a cell. Otherwise:
 3. Let $A[i, j]$ be any cell in X with maximum value (among cells in X).
 4. Let $A[i', j']$ be a neighbor of $A[i, j]$ in A with $A[i', j'] > A[i, j]$. (It exists, as $A[i, j]$ is not a local max.)
 5. Partition $A \setminus X$ (the array A , minus the cells in X) around X into its four $(n - 1)/2 \times (n - 1)/2$ quadrants—the upper left, upper right, lower left, and lower right—in the natural way.
 6. Among those, let A' be the quadrant containing the cell $A[i', j']$.
 7. Recursively compute a local maximum of the subarray A' , and return that cell.

Run time. Recall that the input has size $n \times n = n^2$.

Lemma 4.6. *The algorithm above can be implemented to run in $O(n)$ time.*

Proof sketch. Let $T(n)$ denote the worst-case run time on an $n \times n$ input array. Then $T(1) = O(1)$. Consider running the algorithm on any input with $n \geq 2$. The cross X has $O(n)$ cells, so Lines 2–7 can be implemented to take $O(n)$ time. So $T(n) = T((n - 1)/2) + O(n)$. So $T(n) = O(n)$. □

Correctness. Before we prove correctness, we prove the following utility lemma:

Lemma 4.7. *During any execution of the algorithm that reaches Line 6, the quadrant A' defined there contains a local maximum of A .*

Proof (long form).

1. Consider the following process.

- (i) Put a token on the cell $A[i', j']$.
 - (ii) While the token's cell is not a local maximum in A do:
 - (iii) Move the token to a neighboring cell with larger value.

2. Each cell in X is no larger than $A[i, j]$ (by Line 3 of the algorithm), and $A[i, j] < A[i', j']$.
3. So each cell in X is smaller than $A[i', j']$.
4. So by inspection of the process it never moves the token into X . The token stays in A' .
5. The process must stop, because each move increases the value of the cell with the token.
6. So the token ends on a cell in A' that is also a local maximum of A . □

Now we prove correctness...

Lemma 4.8 (Wrong!). *The algorithm is correct.*

Proof (long form).

1. The proof is by induction on n .
2. Let $P(n)$ be the predicate “the algorithm is correct on all $n \times n$ inputs”.
3. For any input with $n = 1$, the cross X contains the entire array, so Line 2 returns a local max.
4. So $P(1)$ holds.
- 5.1. Consider any execution of the algorithm on any input A with $n \geq 2$. Assume $P(i)$ for $1 \leq i < n$.
- 5.2.1. Case 1. Suppose the algorithm returns in Line 2.
- 5.2.2. By inspection of Line 2, it returns a local max of A .
- 5.3.1. Case 2. Otherwise the algorithm returns in Line 7.
- 5.3.2. The quadrant A' contains a local max of A (by Lemma 4.7).
- 5.3.3. By Step 5.1, $P((n-1)/2)$ holds, so the recursive call (on A') returns a local max of A' .
- 5.3.4. By the previous two steps, the algorithm (Line 7) returns a local max of A .
- 5.4. So, in either case, the algorithm returns a local max of A . That is, it is correct on input A .
6. By Block 5, for any $n \geq 2$, if $P(i)$ holds for $1 \leq i < n$, then $P(n)$ holds.
7. By Step 4, $P(1)$ holds.
8. By the previous two steps and induction, $P(n)$ holds for all $n \geq 1$.
9. That is, the algorithm is correct. □

Python code: [divide_and_conquer/local_maximum.py](#) (Section A.2).

4.7.3 Exercise

Exercise 4.10. (*Local Max proof*) The proof of Lemma 4.8 is wrong.

- (a) In the proof of correctness (Lemma 4.8), where precisely is the error? More specifically, what is the *first* step in the long-form proof that asserts a statement that is not necessarily true? What is the mistake in the reasoning in that line?
- (b) Give an input (with $n \leq 7$) on which the 2D Local Max algorithm in Section 4.7 fails. Describe the execution of the algorithm on that input. We encourage you to verify your answer using the Python implementation in [divide_and_conquer/local_maximum.py](#).

4.7.4 External sources on finding a local maximum

- MIT (“Peak Finding”):
slides: <https://courses.csail.mit.edu/6.006/spring11/lectures/lec02.pdf>
video and lecture notes:
- <https://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-006-introduction-to-algorithms-fall-2011/lecture-videos/lecture-1-algorithmic-thinking-peak-finding/>

4.8 Additional exercises on Divide and Conquer

Each previous section of this note also contains at least one exercise.

External exercises with solutions

- [KT](#) Page 242, Solved Exercises 1 and 2 (divide-and-conquer algorithms)

Exercise without solution

Exercise 4.11. (*Recursive Bus Routes*) You have been commissioned to design a new bus system that will run along Huntington Avenue. The bus system must provide service to n stops (numbered $1, 2, \dots, n$) on the eastbound route. Commuters may begin their trip at any stop i and end at any step j where $1 \leq i < j \leq n$. Here are some naive ways to design the system:

- You could have a bus run from stop 1 to stop n , stopping at every stop in between. The system would be cheap because it would only require $n - 1$ route segments, $\{1 \rightarrow 2, 2 \rightarrow 3, \dots, (n - 1) \rightarrow n\}$, for the entire system. However, a person traveling from stop 1 to stop n would have to wait while the bus makes $n - 1$ stops.
- You could have, for every pair of stops i, j with $i < j$, a special express bus from i to j . No commuter would ever have to make more than one stop. However, this system would be expensive as it requires $\binom{n}{2} = \Theta(n^2)$ route segments, $\{i \rightarrow j : 1 \leq i < j \leq n\}$.

Using divide and conquer, you will find a compromise—a set $S(n)$ of $\Theta(n \log n)$ route segments, with the property that a user can get from any stop i to any stop $j > i$ using at most two segments in $S(n)$. That is, *for every i and j with $1 \leq i < j \leq n$, either the (direct) route segment $i \rightarrow j$ is in $S(n)$, or, for some m with $i < m < j$, the set $S(n)$ contains the two segments $i \rightarrow m$ and $m \rightarrow j$.*

- For each base case $n \in \{1, 2\}$, describe a set $S(n)$ with these properties, and having size $n - 1$.
- For each $n > 2$, give a definition for a set $T(n)$ of at most n route segments such that, for every pair of stops (i, j) where i is in the first half ($1 \leq i \leq \lceil n/2 \rceil$) and j is in the second half ($\lceil n/2 \rceil < j \leq n$), a commuter can get from i to j using one or two segments in $T(n)$.
- For each $n > 2$, give a definition for a full set $S(n)$ as in the problem statement. (Extend the partial solution $T(n)$ by recursing twice, once on the first half and again on the second half.)
- Let $N(n) = |S(n)|$ (the number of segments in $S(n)$). Give a recurrence relation for $N(n)$. Use the the recursion-tree method to solve it. Show your work as described for Exercise 4.2.

External exercises without solutions²

- [KT](#) Exercises 1–7 (Page 246). [DPV](#) 2.1–2.32 (Page 83). [LLM](#) 22.1–22.4 (Page 1017).
- [CLRS](#) Exercises in Section 4.1 (Page 74), and Problems 4–1 through 4–6 (Page 107).
- [JE](#) Chapter 1 Exercises 1–41 (Page 44)

External sources on Divide and Conquer: See the previous sections, especially Section 4.2.2.

²As mentioned in [Lecture Note 1](#), solutions for exercises in [KT](#) and some in [CLRS](#) can be found online.

Lecture Note 5

Greedy Algorithms

[↑ CONTENTS](#)

Contents

5.1	Introduction	70
5.2	Making Change	70
5.2.1	Exercises	72
5.3	Activity Selection	73
5.3.1	Exercises	76
5.3.2	External sources on Activity Selection	77
5.4	Covering by Intervals	78
5.4.1	Exercises	80
5.4.2	External sources on Covering by Intervals	80
5.5	Huffman Coding	81
5.5.1	Exercises	84
5.5.2	External sources on Huffman Coding	84
5.6	Interval Partitioning	85
5.6.1	Exercises	86
5.6.2	External sources on Interval Partitioning	86
5.7	Minimizing Total Wait Time	87
5.7.1	Exercises	88
5.7.2	External sources on Minimizing Total Wait Time	89
5.8	Dijkstra's algorithm for Single-Source Shortest Paths	90
5.8.1	Exercises	94
5.8.2	External sources on Dijkstra's algorithm	94
5.9	Minimum Spanning Trees	95
5.9.1	Iterative approach (Prim's algorithm)	95
5.9.2	A recursive view	99
5.9.3	Exercises	100
5.9.4	External sources on Minimum Spanning Tree	102
5.10	Additional Exercises	103
5.10.1	External exercises with solutions	103
5.10.2	Exercises without solutions	103

5.1 Introduction

An algorithm is *greedy* if it builds a solution piece by piece. Once all the pieces have been added, and the solution is complete, the algorithm returns the solution. A guiding principle is that the algorithm must maintain the following “greedy” invariant at all times:

greedy invariant: *The current (partial) solution can be extended to a correct (full) solution.*

To design the algorithm, a good strategy is to start by finding a rule for the *first* greedy step, and a proof that the first piece chosen by the rule is part of some correct solution. Doing this helps build intuition. Furthermore, in many cases, after the first step, the problem that remains to be solved is a smaller instance of the same problem, so the algorithm can simply recurse to solve that problem. Or, if that doesn’t work, we can take a different approach: try to generalize the rule for the first step, to find a rule for extending the partial solution by one piece at every step, and a proof that the generalized rule maintains the greedy invariant.

We will also see an example that doesn’t fit either of these patterns.

5.2 Making Change

Here’s an example where, after the first greedy choice, the remaining problem is just a smaller instance of the original problem. So, the algorithm can simply recurse to solve that problem. The example is simple, but nicely illustrates this particular schema for greedy algorithms.

Problem Definition. Making Change with U.S. coins.

input: an integer $N \geq 0$ (the desired total)

output: A minimum-length multiset $\{c_1, c_2, \dots, c_\ell\}$ such that each item c_d is in $\{1, 5, 10, 25\}$ and the multiset sums to N , that is, $\sum_{i=1}^{\ell} c_i = N$.

(A *multiset* is an unordered collection of items in which—in contrast to a set—any given item can occur more than once.) Intuitively, the problem is to make change for a given value N , using as few (U.S.) coins as possible. If item c_i is, respectively, 1, 5, 10, or 25, we think of it as being a penny, nickel, dime, or quarter.

Algorithm. There is a natural greedy algorithm—use the largest possible coin, then recurse:

rchangeUS(N):	
1. if $N = 0$: return $\emptyset$.	...base case — return the empty multiset
2. let $c' = \max \{c \in \{1, 5, 10, 25\} : c \leq N\}$	
3. let $\{c_1, c_2, \dots, c_{\ell'}\} = \text{rchangeUS}(N - c')$	...recursively make change for $N - c'$
4. return multiset $\{c', c_1, c_2, \dots, c_{\ell'}\}$	...return $\ell' + 1$ coins of total value N

Unwinding the tail recursion gives the following equivalent iterative implementation:

ichangeUS(N):	
1. while $N > 0$:	
2. let $c' = \max \{c \in \{1, 5, 10, 25\} : c \leq N\}$	
3. output c'	
4. set $N = N - c'$	

The run time is proportional to the number of coins output, which is $O(N)$.

Correctness. To prove correctness, the main challenge is to show that the first greedy choice is correct—that is, there is always some correct solution that uses c' , the largest coin of value at most N . Once we know that, we can commit to choosing coin c' first, and then (clearly?) the problem that remains is to make change optimally for the remaining total, $N - c'$.

We start with a useful observation:

Observation 5.1. *For any N , every optimal way of making change has at most four pennies, at most one nickel, and at most two coins that are nickels or dimes.*

Each claim in the observation follows by an exchange argument. (If there were more than four pennies, the way of making change would not be optimal because five pennies could be exchanged for one nickel. If there more than one nickel, two nickels could be exchanged for one dime. And then, if there were more than two coins that are nickels or dimes, there would have to be either three dimes (replaceable by a quarter and a nickel) or two dimes and a nickel (replaceable by a quarter).)

Using the observation, we show that the first choice of the algorithm is correct:

Lemma 5.1. *Fix any $N \geq 1$ and let $c' = \max \{c \in \{1, 5, 10, 25\} : c \leq N\}$. Let X^* be any optimal way of making change totaling N . Then X^* contains a coin of value c' .*

Proof (long form).

1. Consider any such N , c' , and X^* .
2. All available values that are larger than c' exceed N , so no coin of value greater than c' is in X^* .
3. To finish, we show that there is a coin of value at least c' in X^* .
 - 4.1. Case 1. Suppose $c' = 1$. That is, $N \leq 4$.
 - 4.2. X^* has total value $N \geq 1$, so X^* must have a coin of value at least 1.
 - 5.1. Case 2. Suppose $c' = 5$. That is, $5 \leq N \leq 9$.
 - 5.2. By Observation 5.1, the total value of the pennies in X^* is at most 4.
 - 5.3. But $N \geq 5$, so there is at least one other kind of coin in X^* , necessarily of value at least 5.
 - 6.1. Case 3. Suppose $c' = 10$. That is, $10 \leq N \leq 24$.
 - 6.2. By Observation 5.1, the total value of the pennies and nickels in X^* is at most $4 + 5 = 9$.
 - 6.3. But $N \geq 10$, so there is at least one other kind of coin in X^* , necessarily of value at least 10.
 - 7.1. Case 4. Otherwise $c' = 25$. That is, $N \geq 25$.
 - 7.2. By Observation 5.1 the total value of the pennies, nickels, and dimes in X^* is at most $4 + 20 = 24$.
 - 7.3. But $N \geq 25$, so there is at least one other kind of coin in X^* , necessarily of value 25.
8. By the case analysis, there must be a coin of value at least c' in X^* .
9. By this and Step 2, there must be a coin of value c' in X^* . □

By the lemma, we know that the first step made by the greedy algorithm is correct for any N . That is, the value c' that it chooses in $\{1, 5, 10, 25\}$ is guaranteed to be in any optimal solution with total N , so it can commit to adding c' to its solution. Next we prove carefully that to make change of value N optimally it suffices to add c' to any optimal way of making change of total value $N - c'$:

Lemma 5.2. *Fix any $N \geq 1$ and let $c' = \max \{c \in \{1, 5, 10, 25\} : c \leq N\}$. Let $\{c_1, c_2, \dots, c_{\ell'}\}$ be any optimal way to make change of value $N - c'$. Then $\{c', c_1, c_2, \dots, c_{\ell'}\}$ is an optimal way to make change of value N .*

Proof (long form).

1. Consider any such N , c' , and $c_1, c_2, \dots, c_{\ell'}$.
2. The $\ell' + 1$ coins in $c', c_1, c_2, \dots, c_{\ell'}$ do total N , because the coins in $c_1, c_2, \dots, c_{\ell'}$ total $N - c'$.
3. To finish, we show that to make change for N , at least $\ell' + 1$ coins are required.
4. Let $X^* = \{c_1^*, c_2^*, \dots, c_{\ell}^*\}$ be an optimal way of making change for N .
5. By Lemma 5.1, c' is one of those coins.
6. So X^* contains $\ell - 1$ coins that make change for $N - c'$.
7. Since ℓ' is the minimum number of coins to make change for $N - c'$, it must be that $\ell - 1 \geq \ell'$.
8. Equivalently, $\ell \geq \ell' + 1$ —i.e., making change of total value N requires at least $\ell' + 1$ coins. □

Finally, we show correctness.

Lemma 5.3. *For all $N \geq 0$, $\text{rchangeUS}(N)$ is correct—that is, it makes change totaling N using the fewest possible coins.*

Proof (long form).

1. The proof is by induction on N .
2. For base case $N = 0$, `rchangeUS(0)` uses no coins, which is clearly the minimum possible.
- 3.1. Consider any $N \geq 1$, and assume that, for all $N' < N$, `rchangeUS(N')` is correct.
- 3.2. Consider the execution of `rchangeUS(N)`. As in Lines 2 and 3 of the algorithm,
- 3.3. define $c' = \max\{c \in \{1, 5, 10, 25\} : c \leq N\}$ and $\{c_1, c_2, \dots, c_{\ell'}\} = \text{rchangeUS}(N - c')$.
- 3.4. By the inductive assumption, $\{c_1, c_2, \dots, c_{\ell'}\}$ is an optimal way to make change totaling $N - c'$.
- 3.5. So, by Lemma 5.2, $\{c', c_1, \dots, c_{\ell'}\}$ is an optimal way to make change totaling N .
- 3.6. But $\{c', c_1, \dots, c_{\ell'}\}$ is also the output of `rchangeUS(N)`.
- 3.7. So `rchangeUS(N)` outputs an optimal way to make change totaling N .
4. By Block 3, for all $N \geq 1$, `rchangeUS(N)` is correct if `rchangeUS(N')` is correct for all $N' < N$.
5. By Step 2, `rchangeUS(0)` is correct.
6. By the preceding two steps and induction, for all $N \geq 0$, `rchangeUS(N)` is correct. □

The theorem below summarizes what we've figured out.

Theorem 5.1. *There is an $O(N)$ -time algorithm for Making Change with U.S. Coins.*

5.2.1 Exercises

Exercise 5.1. (*non-greedy denominations*) The proof above relies heavily on the particular denominations $\{1, 5, 10, 25\}$. For each $d \in \{6, 7, 8, 9, 11\}$, do the following. Consider using instead the set of denominations $\{1, 5, d\}$. Figure out whether, for all integers N , the greedy algorithm (with available denominations $\{1, 5, d\}$) always makes change for N using a minimum possible number of coins. For each value of d , explain clearly why your answer is correct.

External exercise

- [JE](#) Chapter 4, Exercise 14

5.3 Activity Selection

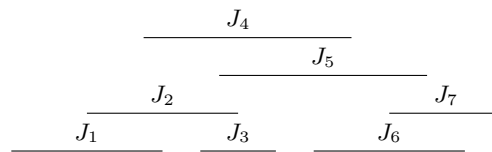
Here is another example where the algorithm makes one greedy choice, then recurses.

Problem definition. The (unweighted) Activity Selection problem is defined as follows:

input: A collection of real intervals $I = ([s_1, e_1], [s_2, e_2], \dots, [s_n, e_n])$, with $s_j \leq e_j$ for all j . We use J_j to denote $[s_j, e_j]$, and call J_j a *job*. We think of $[s_j, e_j]$ as the time interval the job requires.

output: A maximum-size pairwise-disjoint subset S of I .

Pairwise disjoint means that every two intervals in S are disjoint (that is, for every two intervals, their intersection is empty). Intuitively, we want to do as many jobs as we can, but for any two jobs whose intervals overlap, we can't do both. To build intuition, consider an example:



One correct solution for this input is $S^* = \{J_1, J_3, J_6\}$.

The goal is to maximize the size $|S^*|$ of S^* . Note that the size is the *number of intervals* in S , not the total area covered by intervals in S .

Coming up with an algorithm. A greedy algorithm for this problem should maintain a subset $X \subseteq I$ of the jobs. It should start with X empty, then add jobs one by one, until no more jobs can be added to X . It should then return the final set X . That is, the algorithm should have the following form:

input: Activity Selection instance $I = \{J_j = [s_j, e_j] : j \in \{1, 2, \dots, n\}\}$

output: solution X

1. $X \leftarrow \emptyset$ (X is the partial solution so far.)
2. while some job in I can be added to X :
- 3.1. *somehow* choose some job J_j in I (*But how?*)
- 3.2. $X \leftarrow X \cup \{J_j\}$. (*Add the next piece to the partial solution.*)
4. return X

Our goal is to guarantee that the final set X returned by the algorithm will be a correct solution, that is, a maximum-size pairwise-disjoint subset of I .

To get started and build intuition, consider just the *first step* of the algorithm. Let's look for a rule that, given I , identifies a job that *has to* be in a correct solution for I .¹ Here are some possible candidates for the rule:

1. Choose a job with earliest start time (minimizing s_j).
2. Choose a smallest job (minimizing $e_j - s_j$).
3. Choose one with the fewest *conflicts* (i.e., other jobs that overlap).
4. Choose one with earliest end time (minimizing e_j).

Exercise 5.2 asks you to find a counter-example for each of the first three rules (that is, an input for which the job chosen by the rule is not in any optimal solution). To build intuition, review that exercise and its solution. But what about the fourth rule—choose the *earliest-ending* job, J_1 ? Is it the case that, for every instance I , this rule always gives a good first choice? That is, for every instance I , is the earliest-ending job guaranteed to be in at least one correct solution for I ? Let's make this a formal conjecture:

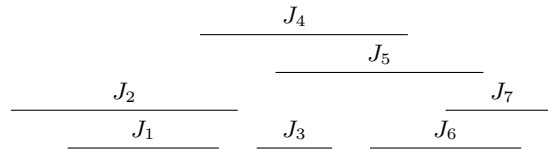
Conjecture 5.1. *For every Activity Selection instance $I = \{J_1, \dots, J_n\}$ with $n \geq 1$, the earliest-ending job J_1 is in some correct solution.*

¹Note that it would be too much to ask for a job that has to be in *every* correct solution! There are instances where no one job is in every correct solution, so that's too much to hope for. We'll see that it's enough to choose a job that's in *at least one* correct solution.

Intuitively, this conjecture is plausible. The reason seems to be something like this. The correct solution has to have *some* first job, and job J_1 is *as good a choice as any*, in the following sense: *compared to any other choice for the first job, choosing J_1 leaves open the most options for the remaining jobs.*² Can we develop this into a solid proof of the conjecture?

Consider any instance $I = \{J_1, J_2, \dots, J_n\}$, and a hypothetical solution S^* for I .

Can we say that the first job in S^* has to be the earliest-ending job in I (that is, J_1)? No! Consider this example:



In this example, $S^* = \{J_2, J_3, J_6\}$ is one correct solution, and it does not contain J_1 . So there can be a correct solution that does not contain J_1 .

But does there have to be *some* correct solution that contains J_1 ? Yes! If we take S^* and replace J_2 by J_1 , the resulting set will still be pairwise-disjoint (why?) and will have the same size as S^* , so must also be a correct solution. This kind of reasoning is called an *exchange argument*. Let's try to extend it to prove the general case.

Lemma 5.4. *Let I be any non-empty problem instance. Let J_1 be an earliest-ending job in I . Then I has a correct solution that contains J_1 .*

Proof (long form).

1. Consider any I and J_1 as described in the lemma, and let S^* be any correct solution for I .
 - 2.1. Case 1. Suppose that J_1 is in S^* .
 - 2.2. Then I has a correct solution that contains J_1 (namely, S^*).
 - 3.1. Case 2. In the remaining case, J_1 is not in S^* .
 - 3.2. Let J_1^* be the first (earliest ending) job in S^* . Note $J_1^* \neq J_1$.
 - 3.3. We'll show that exchanging J_1 for J_1^* in S^* gives a correct solution (containing J_1).
 - 3.4. Let $S' = \{J_1\} \cup S^* \setminus \{J_1^*\}$ be obtained from S^* by replacing J_1^* by J_1 .
 - 3.5. Job J_1 ends no later than J_1^* ends, and J_1^* ends before any other job in S^* starts.
 - 3.6. So J_1 ends before any job (other than J_1^*) in S^* starts.
 - 3.7. So S' is also pairwise disjoint. And S' has the same size as S^* .
 - 3.8. So S' is also a correct solution. So I has a correct solution (namely S') that contains J_1 .
4. So I has a correct solution that contains J_1 . □

How to compute the rest of the correct solution? Recurse! The preceding discussion establishes that the earliest-ending job J_1 is in some correct solution. So we can commit to putting J_1 in our solution. How do we find the remaining jobs?

Given that J_1 is in, no job that overlaps with J_1 (that is, that starts before J_1 ends) can be. So we can remove all those jobs from consideration. From among the remaining jobs, we can choose any pairwise-disjoint subset to complete the solution. And of course we should choose as many of those jobs as possible. That is, the problem that remains is the Activity Selection instance defined by the remaining jobs. So we should just *recurse on those remaining jobs*.

Algorithm. This gives us an algorithm: choose the earliest-ending job J_1 , then recurse on the set containing those jobs that are disjoint from J_1 . Here is pseudo-code:

²KT call this “greedy stays ahead”.

rselect($I = \{J_1, J_2, \dots, J_n\}$):
 1. if $n = 0$: return $\emptyset$.
 2. Recall that J_1 is the earliest-ending job in I .
 3. Let D contain the jobs in I that are disjoint from J_1 .
 4. Recursively compute $R = \text{rselect}(D)$, then return $\{J_1\} \cup R$.

— jobs are ordered by end time
 — if I is empty, return the empty set

Correctness. Let's try to turn our intuition into a full proof of correctness. First we give a utility lemma that formally captures the intuition. It shows formally that, after we choose the earliest-ending job J_1 , it suffices add to J_1 any maximum-size pairwise-disjoint subset of the jobs that are disjoint from J_1 :

Lemma 5.5. *Let I be any non-empty Activity Selection instance. Let J_1 be an earliest-ending job in I . Let D contain the jobs in I that are disjoint from J_1 . Let R be any correct solution for D (as an Activity Selection instance). Then $\{J_1\} \cup R$ must be a correct solution for I .*

Proof (long form).

1. Consider any I , J_1 , D , and R as described in the lemma.
2. Let S^* be a correct solution for I that contains J_1 (it exists by Lemma 5.4).
3. Let $R' = S^* \setminus \{J_1\}$ be obtained by removing J_1 from S^* .
4. Then R' is a pairwise-disjoint subset of D (using here that S^* is pairwise disjoint).
5. Since R is a maximum-size pairwise-disjoint subset of D , we have $|R| \geq |R'|$.
6. So $|S^*| = 1 + |R'| \leq 1 + |R| = |\{J_1\} \cup R|$.
7. That is, $\{J_1\} \cup R$ is as large as S^* .
8. And $\{J_1\} \cup R$ is pairwise disjoint (because R is, and $R \subseteq D$ so all jobs in R are disjoint from J_1).
9. By the previous two steps, $\{J_1\} \cup R$ must also be a correct solution for I . □

Given Lemma 5.5, the reasoning for correctness is as usual for recursive algorithms—by induction. Here is a (short-form) proof.

Theorem 5.2. *The recursive algorithm is correct.*

Proof (short form). The proof is by induction on $|I|$. The base case is when $|I| = 0$. Then I is the empty set, and the pairwise-disjoint subset is the empty set, so **rselect**(I) is correct in this case.

For the induction step, consider the execution of **rselect**(I) on any I of positive size. Let J_1 , D , and $R = \text{rselect}(D)$ be as computed by Lines 2–4 of the algorithm. Note that $|D| < |I|$. Assume inductively that **rselect**(D) is correct. So R is a correct solution for D . By Lemma 5.5, the solution $\{J_1\} \cup R$ returned by **rselect**(I) is correct for I . □

Run time. With care we can obtain a linear-time implementation. First, unwinding the tail recursion in the recursive algorithm (and deleting jobs “lazily”) yields the following equivalent implementation. It fits the iterative greedy template we originally had in mind:

iselect($I = \{J_1, J_2, \dots, J_n\}$):
 1. initialize X to be $\emptyset$ (the empty set of jobs)
 2. for $t = 1$ to n :
 3. if J_t is disjoint from each job already in X :
 4. add J_t to X
 5. return X

— assumes jobs are ordered by end time

Because jobs are ordered by end time, in iteration t , the job J_t will be disjoint from X if and only if it starts after the last job in X (so far) ends. This allows an easy linear-time implementation:

```

iselect( $I = \{J_1, J_2, \dots, J_n\}$ ):
1. initialize  $X = \emptyset$  and  $\ell = -\infty$ 
2. for  $t = 1$  to  $n$ :
3.   if  $s_t > \ell$ :
4.     add  $J_t$  to  $X$ 
5.     set  $\ell = e_t$ 
6. return  $X$ 

```

— assumes jobs are ordered by end time
— maintain ℓ as max end time of any job in X

Here's what we've figured out:

Theorem 5.3. *There is a linear-time algorithm for (unweighted) Activity Selection (or $O(n \log n)$ time if jobs are not sorted by end time).*

Python code: [greedy_algorithms/activity_selection.py](#) (Section A.3).

5.3.1 Exercises

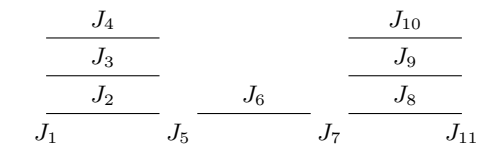
Exercise with solution

Exercise 5.2. (*Activity Selection counter-examples*) Find a counter-example for each of the first three rules proposed in Section 5.3 for choosing a first job for Activity Selection. (Earliest start time, shortest job, job with fewest conflicts.)

Solution. The first two rules are easy to break. Here's a counter-example for the third:

$$I = \{[0, 2], [1, 4], [1, 4], [1, 4], [3, 6], [5, 8], [7, 10], [9, 12], [9, 12], [9, 12], [11, 13]\}.$$

Graphically:



The third rule chooses $J_6 = [5, 8]$ first, but the only correct solution is

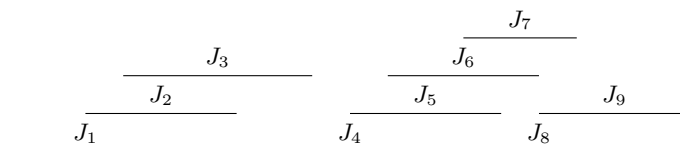
$$S = \{J_1, J_5, J_7, J_{11}\} = \{[0, 2], [3, 6], [7, 10], [11, 13]\},$$

which does not contain J_6 .

Exercises without solutions

Exercise 5.3. What is a correct output for the Activity Selection instance below?

$$I = \{[0, 4], [2, 6], [3, 8], [7, 11], [9, 13], [10, 14], [12, 15], [12, 16], [14, 18]\}.$$



External exercises without solutions

- [KT](#) Chapter 4, Exercise 17. [CLRS](#) Chapter 16, Exercises in Section 16.1.
- [JE](#) Chapter 4, Exercises 1, 2, 3 (Covering by Intervals), 4 (Puncturing)

5.3.2 External sources on Activity Selection

- [KT](#) Section 4.1 and [KW slides](#) (“Interval Scheduling”)
- [CLRS](#) Chapter 16, Section 16.1. [JE](#) Chapter 4, Section 4.2.

5.4 Covering by Intervals

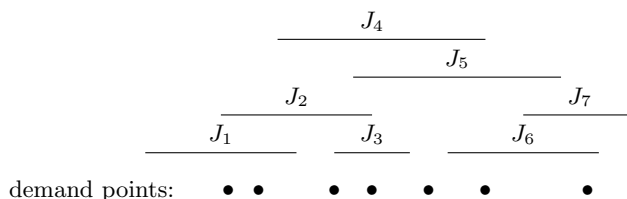
Here is another example where the algorithm makes one greedy choice, then recurses.

Problem definition. The (unweighted) Covering by Intervals problem is defined as follows:

input: A collection of real intervals (called “jobs”) $J = ([s_1, e_1], [s_2, e_2], \dots, [s_m, e_m])$, with $s_j \leq e_j$ for all j , and a set of “demand” points $X = \{x_1, x_2, \dots, x_n\}$. We use J_j to denote the job $[s_j, e_j]$.

output: A minimum-size subset S of J such that every demand point is in at least one interval in S .

We want to use as few intervals as possible, while “covering” the demands. To build intuition, consider an example:



One correct solution for this input is $S^* = \{J_2, J_5, J_7\}$. The goal is to minimize the size $|S^*|$ of S^* . Note that the size is the *number of intervals* in S , not the total area covered by intervals in S .

Coming up with an algorithm. A greedy algorithm for this problem should maintain a subset X of the jobs. It should start with X empty, then add jobs one by one, until all demand points are covered by intervals in X . It should then return the final set X . Our goal is to guarantee that the final set X returned by the algorithm will be a correct solution, that is, a minimum-size set of intervals covering the demands.

To start we consider just the first step of the algorithm. Let’s look for a rule that, given I , identifies a job that *has to* be in some correct solution for I . Here are some possible candidates for the rule:

1. Choose a job with earliest start time (minimizing s_j).
2. Choose a largest job (maximizing $e_j - s_j$).
3. Choose one that covers the most demand points.
4. Among jobs covering the first demand point, choose one with latest end time (maximizing e_j).

To build intuition, find a counter-example for each of the first three rules (that is, an input for which the job chosen by the rule is not in any optimal solution). But what about the fourth rule? Is it the case that, for every instance I , this rule always gives a good first choice? That is, for every instance I , is the job selected by that rule guaranteed to be in at least one correct solution for I ?

The intuition is that any correct solution has to have some job covering the first demand point, and, among jobs that do, the latest-ending job is as good a choice as any: *compared to any other choice among jobs covering the first demand point, choosing the latest-ending job covers all the demand points that any such job can.* Here’s a careful proof:

Lemma 5.6. *Let (J, X) be any problem instance with at least one demand point. Among jobs covering the first demand point, let J_i be one that ends latest. Then (J, X) has a correct solution that contains J_i .*

Proof (long form).

1. Consider any such (J, X) and J_i . Let S^* be any correct solution for (J, X) .
 - 2.1. Case 1. Suppose that J_i is in S^* .
 - 2.2. Then (J, X) has a correct solution that contains J_i (namely, S^*).
 - 3.1. Case 2. In the remaining case, J_i is not in S^* .
 - 3.2. Let J_j^* be any job in S^* that contains the first demand point. (There must be one.)
 - 3.3. We’ll show that exchanging J_i for J_j^* in S^* gives a correct solution (containing J_i).
 - 3.4. Let $S' = \{J_i\} \cup S^* \setminus \{J_j^*\}$ be obtained from S^* by replacing J_j^* by J_i .
 - 3.5. Job J_i ends no earlier than J_j^* ends, so J_i contains all the demand points that J_j^* does.
 - 3.6. So S' covers all the demand points. And S' has the same size as S^* .

- 3.7. So S' is also a correct solution. So (J, X) has a correct solution (namely S') that contains J_i .
 4. So (J, X) has a correct solution that contains J_i . □

Computing the rest of the solution. We know now that the latest-ending job J_i that contains the first demand point is in some correct solution. So we can commit to putting that job J_i in our solution. Given that J_i is in, the demand points that J_i covers are taken care of—we can remove all those from consideration. The problem that remains is to find a minimum-size set of jobs to cover the remaining points. That is, to solve the Covering by Intervals instance defined by the given jobs and the remaining demand points. We solve that recursively.

Algorithm. This gives us an algorithm: choose the latest-ending job J_i among jobs that contain the first demand point x_1 , then recurse on the instance formed by all the jobs and the remaining demand points. Here is pseudo-code:

$\text{rcover}(J, X)$ where $J = (J_1, J_2, \dots, J_m)$ and $X = (x_1, x_2, \dots, x_n)$: — assuming $x_1 \leq x_2 \leq \dots \leq x_n$
 1. If $n = 0$ then return $\emptyset$. — no demand points, return the empty set
 2. Let J_i be the latest-ending job that contains x_1 .
 3. Let X' contain the demand points that are not in J_i .
 4. Recursively compute $R = \text{rcover}(J, X')$, then return $\{J_i\} \cup R$.

Correctness. Next we give a utility lemma that formally proves that, after the first choice of J_i , the remaining problem is equivalent to the instance (J, X') as defined in the algorithm.

Lemma 5.7. *Let (J, X) be any Covering by Intervals instance with at least one demand point. Let J_i be a latest-ending job among jobs that contain the first demand point x_1 . Let X' contain the demand points that are not in J_i . Let R be any correct solution for (J, X') (as a Covering by Intervals instance). Then $\{J_i\} \cup R$ must be a correct solution for (J, X) .*

Proof (long form).

1. Consider any such (J, X) , J_i , X' , and R .
2. All demands in X' are covered by R , so $\{J_i\} \cup R$ covers all points in X , as it should.
3. To finish we show that $\{J_i\} \cup R$ has minimum possible size.
4. Let S^* be a correct solution for (J, X) that contains J_i (it exists by Lemma 5.6).
5. Let $R' = S^* \setminus \{J_i\}$ be obtained by removing J_i from S^* .
6. The jobs in R' must cover all demand points in X' , so R' is a possible solution for (J, X') .
7. Since R is a minimum-size solution for (J, X') , we have $|R| \leq |R'|$.
8. So $|\{J_i\} \cup R| = 1 + |R| \leq 1 + |R'| = |S^*|$. So $\{J_i\} \cup R$ also has minimum possible size. □

Given Lemma 5.7, the reasoning for correctness is as usual for recursive algorithms—by induction. Here is a (short-form) proof.

Theorem 5.4. *The recursive algorithm is correct. That is, for every instance (J, X) , it gives a correct output.*

Proof (short form). The proof is by induction on the number of demand points in X . The base case is when there are no demand points ($X = \emptyset$). Then the empty set is clearly the optimal solution, so $\text{rcover}(J, X)$ is correct in this case.

For the induction step, consider the execution of $\text{rcover}(J, X)$ on any instance with $|X| \geq 1$. Let J_i , X' , and $R = \text{rcover}(J, X')$ be as computed by Lines 2–4 of the algorithm. Assume inductively that $\text{rcover}(J, X')$ is correct. So R is a correct solution for (J, X') . By Lemma 5.7, the solution $\{J_i\} \cup R$ returned by $\text{rcover}(J, X)$ is correct for (J, X) . □

Run time. Assume the demand points are given in non-decreasing order ($x_1 \leq x_2 \leq \dots \leq x_n$) and the jobs are given in order of non-decreasing end time ($e_1 \leq e_2 \leq \dots \leq e_m$). If not, preprocess the input to make it so. Unwinding the tail recursion in the recursive algorithm yields the following equivalent iterative implementation.

`icover`($J = (J_1, J_2, \dots, J_m), X = (x_1, x_2, \dots, x_n)$): — assumes $e_1 \leq \dots \leq e_m$ and $x_1 \leq \dots \leq x_n$

1. initialize A to be $\emptyset$ (the empty set of jobs)
2. for $t = 1$ to n :
3. if x_t is not covered by any job in A :
4. let J_i be the latest-ending job that covers x_t
5. add J_i to A
6. return A

If J_i is the latest-ending job that covers x_t , then the latest-ending job that covers x_{t+1} will be some job $J_{i'}$ where $i' \geq i$. Using this fact, and carefully keeping track of which jobs we've considered, we can obtain a linear-time implementation. We omit the details here.³

Here's what we've figured out:

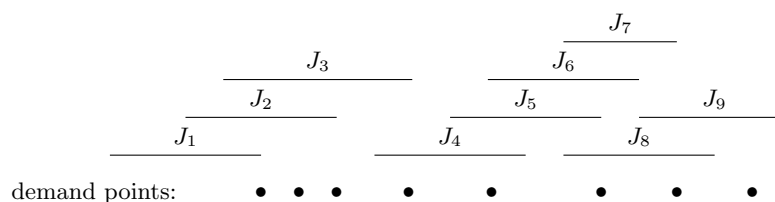
Theorem 5.5. *There is a linear-time algorithm for (unweighted) Covering by Intervals (or $O((n+m) \log(n+m))$ time if jobs and demand points are not already sorted).*

5.4.1 Exercises

Exercise 5.4. (*Covering by Intervals counter-examples*) Find a counter-example for each of the first three rules proposed in Section 5.4 for choosing a first job for Covering by Intervals. (Earliest start time, largest job, job that covers the most demand points.)

Exercise 5.5. What is a correct output for the Covering by Intervals instance below?

$$J = \{[0, 4], [2, 6], [3, 8], [7, 11], [9, 13], [10, 14], [12, 15], [12, 16], [14, 18]\}; X = \{4, 5, 6, 8, 10, 13, 15, 17\}.$$



5.4.2 External sources on Covering by Intervals

- [JE](#) Chapter 4 (and Exercises 1, 2, 3)

³Assume that each point x_t is in at least one interval. (Otherwise there is no solution.) For each point x_t , define L_t to be the index of the latest-ending job that contains x_t . Note that $L_t \leq L_{t+1}$ for all t , because the points are in increasing order and the jobs are ordered by increasing end-time. First, in a linear-time preprocessing step, compute L_t for all t . To do this, enumerate the points in reverse (from latest to earliest). For each point x_t , find L_t by starting at L_{t+1} , and scanning backwards. Here's a linear-time implementation of this part:

1. let $j = m$
2. for $t = n, n-1, \dots, 1$:
3. while $j > 1$ and interval j does not contain x_t : decrement j
4. $L_t = j$

To see that this takes linear time, note that in Line 3, throughout the entire execution, the number of times the while loop executes is at most m (once for each interval). Now, in `icover`, the latest-ending job that contains a particular point x_t can be found in constant time—it's the job with index L_t .

5.5 Huffman Coding

Here is another example where the algorithm makes one greedy choice, then recurses.

Problem definition. The Huffman Coding problem is defined as follows:

input: An increasing sequence $p = (p_1, \dots, p_n)$ of *frequencies* ($0 \leq p_1 \leq p_2 \leq \dots \leq p_n$).

output: A rooted binary tree T whose n leaves consist of the n frequencies. The tree should have minimum cost, defined as $\text{cost}(T) = \sum_{i=1}^n p_i \text{depth}_T(p_i)$, where $\text{depth}_T(p_i)$ is the depth of the leaf for p_i in T . The cost is the (weighted) average leaf depth.

The problem models compressing a given file in a certain way. (See e.g. [here](#).)

It's not obvious a-priori what a greedy algorithm for this problem should look like, because a solution T is not just a subset of objects that can be chosen one by one. But we apply the same basic idea: to start, we look for some piece that must be “part” of an optimal tree. Let's consider some examples. Our goal is to discover properties that an optimal solution should have.

Consider an example where $p = (0.5, 0.5)$. The optimal tree T has two leaves, each of depth 1, and has cost 1. In fact, this tree is optimal for any instance with $n = 2$ frequencies. Next consider an example where $p = (0.25, 0.25, 0.5)$. We should use the tree T with $n = 3$ leaves, whose two leftmost leaves are at depth 2 (holding p_1 and p_2 , respectively), and with p_3 as the right child of the root. (Why is that?) This way the tree has cost $.5 \cdot 1 + 0.25 \cdot 2 + 0.25 \cdot 2 = 1.5$

Fix an arbitrary instance p . Define $\mathcal{S}$ to be the set of trees T for p in which the two leaves p_1 and p_2 (with lowest frequencies) are siblings. We argue next that there must be an optimal tree for p that is in $\mathcal{S}$.

First, does an optimal tree T always have to have *some* pair of leaves that are siblings? Yes—consider any deepest leaf ℓ in T ; if ℓ doesn't have a sibling, then its parent only has one child. So we could replace the parent by ℓ and get a better tree.

Observation 5.2. *There is an optimal tree T for p in which two of the deepest leaves are siblings.*

Next we show by an exchange argument that these siblings may as well be p_1 and p_2 .

Lemma 5.8. *There is an optimal tree T for p that is in $\mathcal{S}$ (i.e., p_1 and p_2 are siblings in T).*

Proof (short form). The proof is an exchange argument. Let T be an optimal tree in which two deepest leaves are siblings (one exists by Observation 5.2). Let p_i and p_j (with $i < j$) be two deepest leaves that are siblings. Let tree T' be obtained from T by exchanging p_1 and p_i . By the following calculation, $\text{cost}(T') \leq \text{cost}(T)$:

$$\begin{aligned} \text{cost}(T') - \text{cost}(T) &= p_1[\text{depth}_{T'}(p_1) - \text{depth}_T(p_1)] + p_i[\text{depth}_{T'}(p_i) - \text{depth}_T(p_i)] \\ &= p_1[\text{depth}_T(p_i) - \text{depth}_T(p_1)] + p_i[\text{depth}_T(p_1) - \text{depth}_T(p_i)] \\ &= (p_1 - p_i)[\text{depth}_T(p_i) - \text{depth}_T(p_1)] \\ &\leq 0. \end{aligned}$$

The last line follows from $p_1 \leq p_i$ and $\text{depth}_T(p_i) \geq \text{depth}_T(p_1)$.

Next, let T'' be the tree obtained from T' by exchanging p_2 and p_j . Note that $p_2 \leq p_j$ (as $j > i \geq 1$), and $\text{depth}_T(p_j) \geq \text{depth}_T(p_2)$, so the same calculation as above shows $\text{cost}(T'') \leq \text{cost}(T')$. From this it follows that T'' is also optimal. And since p_1 and p_2 in T'' replace p_i and p_j in T , leaves p_1 and p_2 are siblings in T'' . So $T'' \in \mathcal{S}$. $\square$

By Lemma 5.8, p has an optimal tree in $\mathcal{S}$. So it's enough to find a tree that is optimal among trees in $\mathcal{S}$? How do we do that? The next key insight is that in any tree $T \in \mathcal{S}$, we can think of the 3-node subtree consisting of p_1 , p_2 , and their parent as a single leaf having frequency $p_1 + p_2$.

Next we make this intuition precise.

Definition 5.1. *Define p' to be the instance obtained from instance p by removing the two smallest frequencies p_1 and p_2 , replacing them by a single frequency $p_1 + p_2$, and reordering the frequencies as necessary. So p' has $n - 1$ frequencies.*

Define $\mathcal{T}'$ to be the set of possible trees for p' .

Given any $T \in \mathcal{S}$, let R be the tree obtained from T by replacing the leaves p_1 , p_2 , and their parent (a 3-node subtree) by a single new leaf with frequency $p'_i = p_1 + p_2$. Call the process of obtaining R from T in this way merging p_1 and p_2 in T .

Let $\mu : \mathcal{S} \rightarrow \mathcal{T}'$ be the function where $\mu(T) = R$, where R is obtained by merging p_1 and p_2 in T .

The merging process is invertible as follows. Given any tree $R \in \mathcal{T}'$, obtain tree T from R by finding the leaf with the frequency $p_1 + p_2$ in R , and replacing it by a new node having two new children: leaves p_1 and p_2 . Call the process of obtaining T this way *splitting* p'_i in R . If we then merge p_1 and p_2 in T , we get R back. That is, $\mu(T) = R$ and $T = \mu^{-1}(R)$.

In short, μ is a bijection between the set $\mathcal{S}$ of trees for p (having p_1 and p_2 as siblings) and the set $\mathcal{T}'$ of trees for p' . So solutions for p (of the kind that we are after) correspond one-to-one with solutions for p' . But does the bijection preserve the *cost*? For each $T \in \mathcal{S}$, if $R = \mu(T)$, is it the case that $\text{cost}(T) = \text{cost}(R)$? By considering an example, we can see that it's not the quite the case. However, the function just *shifts* the cost in a predictable way: it decreases it by $p_1 + p_2$:

Observation 5.3. For each tree $T \in \mathcal{S}$ and tree $R = \mu(T)$, $\text{cost}(T) = \text{cost}(R) + p_1 + p_2$.

To see why the observation holds, let d be the depth of p_1 and p_2 in T . The leaves p_1 and p_2 in T contribute $d \cdot (p_1 + p_2)$ to $\text{cost}(T)$, whereas the leaf p'_i in R contributes $(d - 1) \cdot (p_1 + p_2)$ to $\text{cost}(R)$. (And otherwise T and R are identical.)

In short, the function μ (merging p_1 and p_2) is a bijection (a one-to-one mapping, a.k.a. correspondence) between $\mathcal{S}$ (the trees for p in which p_1 and p_2 are siblings) and $\mathcal{T}'$ (the trees for p'). Furthermore, the bijection preserves relative cost: given two trees T_1 and T_2 for p , it is the case that $\text{cost}(T_1) \leq \text{cost}(T_2)$ if and only if $\text{cost}(R_1) \leq \text{cost}(R_2)$, where $R_1 = \mu(T_1)$ and $R_2 = \mu(T_2)$.

It follows that a tree $T \in \mathcal{S}$ is optimal for p if and only if $\mu(T)$ is optimal for p' . So, to compute an optimal tree T , it is enough to compute an optimal tree R for p' , and then take $T = \mu^{-1}(R)$. (That is, obtain T by splitting its leaf of frequency $p_1 + p_2$.) This is the gist of the next lemma.

Recall that p is an arbitrary instance with $n \geq 2$, and p' is the instance obtained from p by replacing p_1 and p_2 with $p_1 + p_2$. The lemma says that, to compute an optimal tree for p , it suffices to compute an optimal tree for p' , then split $p_1 + p_2$:

Lemma 5.9. Let R be any optimal tree for p' . Let $T = \mu^{-1}(R)$ be obtained by splitting the leaf of frequency $p_1 + p_2$ in R . Then T is optimal for p .

Proof (long form).

1. Consider any p , p' , R , and $T = \mu^{-1}(R)$ as defined above. We need to show T is optimal for p .
2. Let T^* be an optimal tree for p in $\mathcal{S}$. (T^* exists by Lemma 5.8.)
3. Let $R^* = \mu(T^*)$ be obtained by contracting p_1 and p_2 in T^* .
4. By Observation 5.3, $\text{cost}(T^*) = \text{cost}(R^*) + p_1 + p_2$, and $\text{cost}(T) = \text{cost}(R) + p_1 + p_2$.
5. Tree R is optimal for p' , so $\text{cost}(R) \leq \text{cost}(R^*)$.
6. By Steps 4 and 5, $\text{cost}(T) = \text{cost}(R) + p_1 + p_2 \leq \text{cost}(R^*) + p_1 + p_2 = \text{cost}(T^*)$.
7. That is, $\text{cost}(T) \leq \text{cost}(T^*)$.
8. Since T^* has minimum cost for p , it follows that T does also. So T is optimal for p . □

This gives us the following recursive algorithm:

Huffman($p = (p_1, p_2, \dots, p_n)$): — recursive algorithm for Huffman Coding —

1. If $n = 1$: return the tree consisting of a single leaf p_1 .
2. Let p' be the instance obtained from p by replacing p_1 and p_2 by $p_1 + p_2$ and reordering.
3. Let $R = \text{Huffman}(p')$. (Recursively compute a tree for p')
4. Obtain new tree $T = \mu^{-1}(R)$ for p by splitting the leaf of frequency $p_1 + p_2$ in R (replacing leaf $p_1 + p_2$ by a new node with two new children, leaves p_1 and p_2).
5. Return T .

The correctness of the recursive algorithm follows from Lemma 5.9 by a simple induction.

Lemma 5.10. *The recursive algorithm is correct.*

Proof (long form).

1. We show by induction on $n = |p|$ that **Huffman** is correct for all instances p .
2. For any instance with $n = 1$, the one-node tree is optimal, so the algorithm is correct.
- 3.1. Consider any p with $n \geq 2$.
- 3.2.1. Assume that the algorithm is correct for all smaller instances.
- 3.2.2. Consider the execution of the algorithm on p .
- 3.2.3. Let p' , $R = \text{Huffman}(p')$, and $T = \mu^{-1}(R)$ be as computed in the algorithm.
- 3.2.4. By the inductive assumption, R is optimal for the smaller instance p' .
- 3.2.5. So, by Lemma 5.9, the solution $T = \mu^{-1}(R)$ returned by **Huffman**(p) is optimal for p .
- 3.3. By Block 3.2, if **Huffman** is correct for all smaller instances, then it is correct for p .
4. By Block 3, for any p with $n \geq 2$, if **Huffman** is correct for smaller instances, it is correct for p .
5. By Step 2, **Huffman** is correct for all instances of size 1.
6. So, inductively, **Huffman** is correct for all instances. □

We proved correctness of the recursive algorithm using the optimal-substructure property for the greedy choice. We showed how to *reduce* the problem of computing an optimal tree for p to the problem of computing an optimal tree for the smaller instance p' .

The algorithm is tail recursive except for the step of splitting the $p_1 + p_2$. Try executing it on few examples to get intuition. We can unwind the tail recursion to get the following iterative algorithm:

iHuffman($p = (p_1, p_2, \dots, p_n)$): — iterative algorithm for Huffman Coding —

1. Initialize $C \leftarrow (t_1, t_2, \dots, t_n)$, where t_i is a subtree consisting of a single leaf p_i .
2. For $i \leftarrow 1, 2, \dots, n - 1$:
 - 3.1. Remove the two minimum-frequency subtrees t_1 and t_2 from C .
 - 3.2. Construct a new subtree t' , whose root is a new node with left and right subtrees t_1 and t_2 .
The frequency of t' is the sum of the frequencies of t_1 and t_2 .
 - 3.3. Insert t' into C , and reorder the trees in C by increasing frequency.
4. Return the single tree t_1 in C .

The iterative algorithm maintains a collection $C = (t_1, t_2, \dots, t_k)$ of subtrees. Initially, there are n subtrees: subtree t_i consists of a single leaf with frequency p_i . It maintains the *frequency* of each subtree, which is defined to be the sum of the frequencies of its leaves. It keeps the collection C ordered by increasing frequency. The algorithm repeats the following step: it removes the two minimum-frequency subtrees t_1 and t_2 from C , then replaces them by a single new subtree T , formed by a new root, having left subtree t_i and right subtree t_j . It stops after $n - 1$ iterations, when the collection has just one tree, and returns that tree.

What is the greedy invariant? If we tried to prove directly that the iterative algorithm is correct, we would do so by showing that it maintains the greedy invariant. In this case, that would be:

greedy invariant: *There is an optimal tree T that includes each subtree t_i in C .*

One can prove that (i) this invariant is maintained in each iteration, and (ii) after the final iteration (when there is just one tree t_1 in C) the invariant implies that t_1 is an optimal tree for p . The proof of (i) generalizes the proof of Lemma 5.8. We leave the proofs of (i) and (ii) as an exercise.

Run time. By storing the subtrees in C in a heap, it is easy to obtain run time $O(n \log n)$. It is possible to reduce this to linear, $O(n)$ by the following clever observation. The initial frequencies are given in increasing order. In the sequence of subtrees that are produced and inserted into C , the frequencies increase with time. That is, each subtree's frequency is at least as large as the frequency of the previously produced subtree. So, instead of holding C in a heap, we can store C as two ordered lists: the first containing the original singleton subtrees $t_i, t_{i+1}, \dots, t_n$ in C and the second containing the other subtrees (each with more than one node) in C . When we create a new subtree, we add it to the end of the second list (in constant time). Then we

can find the minimum-frequency subtree in C in constant time—it will be the first subtree on one of these two lists. Here is what we have figured out:

Theorem 5.6 (Huffman). *There is an $O(n)$ -time algorithm for Huffman Coding (or $O(n \log n)$ time if the given frequencies are not sorted).*

Python code: [greedy_algorithms/Huffman.py](#) (Section A.3).

5.5.1 Exercises

- [JE](#) Chapter 4, Exercises 6–8.
- [DPV](#) Chapter 5, Exercises 5.11–5.19, 5.28–5.30.
- [CLRS](#): Exercises in Section 16.3 (Page 436).

5.5.2 External sources on Huffman Coding

- [JE](#) Chapter 4. [KT](#) Chapter 4, Section 4.8. [CLRS](#) Section 16.3

5.6 Interval Partitioning

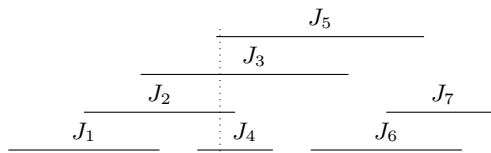
This example does *not* follow the usual “make one choice then recurse” pattern, nor is the design motivated by maintaining the greedy invariant. The algorithm does build the solution iteratively, but we prove optimality of the solution by a combinatorial argument (a matching lower bound).

Problem definition. The Interval Partitioning problem is defined as follows:

input: A collection of real intervals $I = ([s_1, e_1], [s_2, e_2], \dots, [s_n, e_n])$, with $s_i \leq e_i$ for all i . We use J_j to denote $[s_j, e_j]$, and call J_j a *job*. We think of $[s_j, e_j]$ as the time interval the job requires.

output: A minimum-size partition of I into pairwise-disjoint subsets.

Pairwise disjoint means that every two intervals in S are disjoint (that is, for every two intervals, their intersection is empty). Intuitively, we want to partition the jobs into a small number of groups, so that in each group no two jobs overlap. We can think of this as assigning each job to a server, so that each server can do all the jobs assigned to it while doing at most one job at a time. We want to minimize the number of servers. To build intuition, consider an example (ignore the dotted vertical line for now):



One correct solution is to partition the jobs by level (as shown above). That is, partition I into the following four groups:

$$\{J_1, J_4, J_6\}, \{J_2, J_7\}, \{J_5\}, \{J_3\}.$$

Recall that the goal is to minimize the number of groups. Does this partition, with four groups, do that? Note that the four jobs J_2, J_3, J_4 , and J_5 all share a common point (the dotted vertical line). That is, the intersection of their four intervals is non-empty. So no two of those jobs can be in the same group. So at least four groups are necessary. So the above partition is optimal.

Algorithm. We assume the jobs given in order of increasing *start* time, so $s_1 \leq s_2 \leq \dots \leq s_n$. (If not, sort the jobs first in $O(n \log n)$ time.) The algorithm considers the jobs in the given order, maintaining as it goes a partition of the jobs seen so far. The partition is initially empty. For each job J_j , if there is already a group that J_j can be added to (where J_j is disjoint from all jobs already in that group), it adds J_j to that group. Otherwise, it starts a new group containing just J_j :

partition($I = \{J_1, J_2, \dots, J_n\}$):	<i>Assume jobs are ordered by start time</i>
1. initialize $P = \emptyset$.	<i>... initialize p to the empty partition</i>
2. for $j = 1$ to n do:	
3. if there is a group $g \in P$ such that J_j is disjoint from every job J_i in g :	
4. set $g = \{J_j\} \cup g$	<i>... add job J_j to group g</i>
5. else:	
6. set $P = \{\{J_j\}\} \cup P$	<i>... add a new group $\{J_j\}$ to P</i>
7. return P	

Correctness

Lemma 5.11. *The algorithm is correct.*

Proof (long form).

1. Consider any execution of the algorithm on a Interval Partitioning instance I .
2. Let j' be the last iteration in which a new group was added to P in Line 6 of the algorithm.
3. Consider the time just before Line 6 was executed in iteration j' .

4. Let P' be the partition P at that time. (So P' doesn't include the new group $\{J_{j'}\}$.)
5. Each group $g \in P'$ contains some job, say $J_{c(g)}$, that overlaps $J_{j'}$ because otherwise the algorithm (Lines 3 and 4) would have added $J_{j'}$ to group g .
6. So, for each $g \in P'$, the interval $[s_{c(g)}, e_{c(g)}]$ of job $J_{c(g)}$ overlaps the interval $[s_{j'}, e_{j'}]$ of $J_{j'}$.
7. The jobs are sorted by start time, so each start time $s_{c(g)}$ is at most the start time $s_{j'}$ of $J_{j'}$.
8. Hence each interval $[s_{c(g)}, e_{c(g)}]$ contains that start time $s_{j'}$.
9. Hence, there are $|P'| + 1$ jobs (the jobs $\{J_{c(g)} : g \in P'\}$ and $J_{j'}$) that all contain $s_{j'}$.
10. In any valid partition, no pair of those $|P'| + 1$ jobs can be in the same group (each pair overlaps).
11. Hence, in *any* valid partition, the number of groups is at least $|P'| + 1$.
12. After iteration j' , the algorithm doesn't add groups to P , so its final partition has size $|P'| + 1$.
13. So any valid partition has as many groups as the partition returned by the algorithm. □

Exercise 5.6 asks you to consider whether the algorithm would work if it were to consider the jobs by increasing *end time* (instead of start time).

5.6.1 Exercises

Exercise 5.6. Modify the Interval Partitioning algorithm in Section 5.6 so that it considers the jobs in order of increasing *end time* (instead of start time). Consider the following invariant:

invariant: For any two groups g and g' in the current partition P , the last job added to g and the last job added to g' overlap.

- (a) Prove that, in any execution of the algorithm, if the invariant holds throughout, then the output is an optimal solution.
- (b) Is the algorithm correct? Prove that it is, or describe a counter-example (an input for which the algorithm fails).

External exercises without solutions

- [LLM](#) Exercise 10.21 (Page 459). [JE](#) Chapter 4, Exercise 5

5.6.2 External sources on Interval Partitioning

- [KT](#) Section 4.2 (“Scheduling All Intervals”) and [KW slides](#) (“Interval Partitioning”)

5.7 Minimizing Total Wait Time

Problem Definition

input: a sequence $T = (t_1, t_2, \dots, t_n)$ of n tasks, each with an associated length $\ell(t_i)$.

output: a permutation $T' = (t'_1, t'_2, \dots, t'_n)$ of T minimizing the *total wait time*, denoted $W(T')$ and defined as $W(T') = \sum_{i=1}^n \sum_{j=1}^i \ell(t'_j)$.

Intuitively, we must choose an order in which to do the tasks so as to minimize the sum of their respective times until completion, assuming the j th task we do (task t'_j) starts when the $(j-1)$ st task finishes, and then takes time $\ell(t'_j)$ equal to the task's length.

We start with some observations that will be useful for developing an algorithm.

Observation 5.4. *The wait time $W(T')$ can also be expressed as*

$$W(T') = \sum_{j=1}^n (n-j+1) \ell(t'_j).$$

The observation holds by considering the sum that defines $W(T')$: the j th task t'_j contributes its length $\ell(t'_j)$ to its own completion time, and to the completion times of the $n-j$ following jobs.

Next we consider how swapping two adjacent tasks changes the wait time.

Observation 5.5. *For any permutation $T' = (t'_1, \dots, t'_n)$, and any j with $1 \leq j \leq n-1$, let $T'' = (t''_1, \dots, t''_n)$ be the permutation obtained from T' by swapping the j th job t'_j with the following job t'_{j+1} . That is, $t''_j = t'_{j+1}$, $t''_{j+1} = t'_j$, and $t''_i = t'_i$ for all $i \notin \{j, j+1\}$.*

Then $W(T'') = W(T') + \ell(t'_{j+1}) - \ell(t'_j)$.

The observation follows from the previous one. (Swapping increases the coefficient of $\ell(t'_{j+1})$ by 1, and decreases the coefficient of $\ell(t'_j)$ by 1.)

We'll use the observation to show that t'_1 should be the shortest task. The next observation will then allow us to decouple the problem of choosing the first task t'_1 from the problem of ordering the remaining $n-1$ tasks, $(t'_2, \dots, t'_n)$.

Observation 5.6. *For any permutation $T' = (t'_1, \dots, t'_n)$, let $R' = (r'_1, r'_2, \dots, r'_{n-1}) = (t'_2, t'_3, \dots, t'_n)$ be the permutation that T' induces on the $n-1$ tasks other than t'_1 . Then*

$$W(T') = n \ell(t'_1) + W(R').$$

Note that $W(R')$ is the total wait time of the permutation R' of the remaining $n-1$ tasks, considered as an independent instance with just those $n-1$ tasks. (Observation 5.6 can be verified by using Observation 5.4 to expand $W(T')$ and $W(R')$, then using $r'_j = t'_{j+1}$.)

Algorithm. Now we consider the first greedy choice. We use Observation 5.5 to show that we can put any shortest task first:

Lemma 5.12. *Consider any instance $T = (t_1, \dots, t_n)$ with $n \geq 1$. Let t_m be any task of minimum length, so $\ell(t_m) = \min_i \ell(t_i)$. Then there is an optimal solution T' that starts with job t_m .*

Proof of Lemma 5.12.

1. Consider any instance T with $n \geq 1$ and let t_m be any task of minimum length.
2. Let T' be an optimal permutation (there is one, as the number of permutations is finite).
- 3.1. Case 1. Suppose T' starts with task t_m (that is, $t'_1 = t_m$).
- 3.2. Then there is an optimal solution (namely T') that starts with task t_m .
- 4.1. Case 2. Otherwise task t_m is not first in T' .
- 4.2. Consider bringing task t_m to the front by repeatedly swapping it with the task before it.
- 4.3. By Observation 5.5 (and $\ell(t_m) \leq \ell(t_j)$ for all j) each swap does not increase the total wait time.

4.4. So the resulting permutation (which starts with task t_m) is also optimal. $\square$

By Lemma 5.12, it is safe to choose any minimum-length task t_m to go first. Next we use Observation 5.6, to show that, once the first task is chosen as t_m , we can simply recurse on the remaining $n - 1$ tasks. That is, any optimal permutation of those tasks (considered as an independent instance with $n - 1$ tasks) will work.

Lemma 5.13. *Consider any instance $T = (t_1, t_2, \dots, t_n)$ with $n \geq 1$. Let t_m be any task of minimum length, so $\ell(t_m) = \min_i \ell(t_i)$. Let $R = (r_1, r_2, \dots, r_{n-1})$ be the $n - 1$ remaining tasks (other than t_m). Let $R' = (r'_1, r'_2, \dots, r'_{n-1})$ be any optimal permutation of those $n - 1$ tasks.*

Then the permutation $T' = (t_m, r'_1, r'_2, \dots, r'_{n-1})$ is optimal for T .

Proof (long form).

1. Consider any such T , m , R , R' , and T' .
2. Let $T^* = (t_1^*, t_2^*, \dots, t_n^*)$ be an optimal permutation of T with $t_1^* = t_m$ (T^* exists by Lemma 5.12).
3. Let $R^* = (t_2^*, \dots, t_n^*)$ be the permutation that T^* induces on the tasks other than t_m .
4. Since R' is an optimal permutation of those $n - 1$ tasks, $W(R') \leq W(R^*)$.
5. Using Observation 5.6, then $W(R') \leq W(R^*)$, then Observation 5.6 again,

$$W(T') = n\ell(t_m) + W(R') \leq n\ell(t_m) + W(R^*) = W(T^*).$$

$\square$

This gives us the following algorithm:

```
rMinWaitTime( $t_1, t_2, \dots, t_n$ ):
1. if  $n = 0$ : return the empty sequence
2. let  $t_m$  be a job of minimum length
3. let  $(r_1, r_2, \dots, r_{n-1})$  be the  $n - 1$  jobs other than  $t_m$ , in any order
4. let  $(r'_1, r'_2, \dots, r'_{n-1}) = \text{rMinWaitTime}(r_1, r_2, \dots, r_{n-1})$ 
5. return  $T' = (t_m, r'_1, \dots, r'_{n-1})$ 
```

Correctness

Lemma 5.14. *The algorithm is correct.*

Proof (short form). The proof is by induction on n . For the base case $n = 0$, the algorithm is clearly correct. Consider any $n \geq 1$. Assume inductively that the algorithm is correct on all inputs of length $n - 1$. Consider executing the algorithm on any input $T = (t_1, t_2, \dots, t_n)$ of length n . Let t_m , $(r_1, r_2, \dots, r_{n-1})$, $(r'_1, r'_2, \dots, r'_{n-1})$, and T' be as computed in the algorithm. (See the algorithm for their definitions.) By the inductive assumption $(r'_1, r'_2, \dots, r'_{n-1})$ is optimal for $(r_1, r_2, \dots, r_{n-1})$. Hence, by Lemma 5.13, T' is optimal for T . $\square$

Run time. Clearly the algorithm can be implemented to run in $O(n^2)$ time. But we can do better. Observe that the algorithm simply sorts the jobs by increasing length. This can be done via Mergesort in $O(n \log n)$ time. So the algorithm can be implemented to run in $O(n \log n)$ time.

The next theorem summarizes what we've figured out:

Theorem 5.7. *There is an $O(n \log n)$ -time algorithm for Minimizing Total Wait Time.*

Exercise 5.7 asks you to extend this result to a more general problem where each task also has a positive *weight*, and the goal is to minimize the *weighted* sum of completion times.

5.7.1 Exercises

Exercise without solution

Exercise 5.7. Consider the following generalization of the problem in Section 5.7.

input: an integer $n \geq 0$, and for each $i \in \{1, \dots, n\}$, a *job length* ℓ_i and positive *weight* w_i .

output: a permutation π of the n jobs minimizing the *weighted sum of completion times*, defined as $\sum_{i=1}^n w_{\pi(i)} \sum_{j=1}^i \ell_{\pi(j)}$.

Prove carefully that ordering the jobs by increasing ℓ_i/w_i works (generalizing the algorithm in Section 5.7). Make sure your proof correctly handles instances with ties, where $\ell_i/w_i = \ell_j/w_j$ for some jobs i and j . This can happen even when $(\ell_i, w_i) \neq (\ell_j, w_j)$!

5.7.2 External sources on Minimizing Total Wait Time

- [JE Chapter 4](#) Section 4.1 (As of August 2019, Erickson’s presentation has a subtle error. In our terminology, his Lemma 4.1 states that *any permutation that orders the tasks by increasing length is optimal*. But what his proof actually shows is the converse: *any permutation that doesn’t order the tasks by increasing length isn’t optimal*. These are not the same! We’ve filed a bug report.)

5.8 Dijkstra's algorithm for Single-Source Shortest Paths

Distances in edge-weighted graphs. Given an edge-weighted graph $G = (V, E)$ (directed or undirected) the (*shortest-path*) distance from vertex v to vertex w is the minimum weight of any path from v to w :

$$\text{dist}(v, w) = \min\{\text{wt}(P) : P \text{ is a path from } v \text{ to } w \text{ in } G\}.$$

We think of a path P as a set of edges, and use $\text{wt}(P)$ to denote the sum of the edges' weights, so $\text{wt}(P) = \sum_{e \in P} \text{wt}(e)$. If there is no path from v to w , then $\text{dist}(v, w) = \infty$.

[Consider examples to get intuition.]

Observation 5.7. Let $G = (V, E)$ be any edge-weighted graph (directed or undirected). Suppose G has no negative-weight cycles (the weight of each cycle C in G is non-negative, so $\sum_{e \in C} \text{wt}(e) \geq 0$). Then:

- (a) For every two vertices $v, w \in V$, the shortest-path distance from v to w is well-defined.
- (b) If there is a path from v to w , then there is a shortest path that is simple (has no cycles).

The rest of the note focuses on finding shortest paths from a given source vertex s , assuming that the graph has no negative-weight edges.

Problem Definition. The Single-Source Shortest Paths problem is defined as follows:

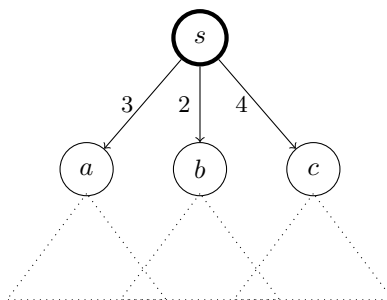
input: Directed graph $G = (V, E)$ with *non-negative* edge weights and source vertex $s \in V$.

output: For each vertex $v \in V$, the shortest-path distance from s to v .

(For computing shortest paths, any undirected graph G can be considered as a directed graph—just replace each undirected edge (u, v) by the two directed edges (u, v) and (v, u) .)

Algorithm. Since we are looking for a greedy algorithm, let's start by finding a rule for determining some *first part* of a correct solution. In this case, we note that by Observation 5.7(b), $\text{dist}(s, s) = 0$. So the algorithm can go ahead and set $d[s] = 0$. That step is easy, but what's next? Let's consider the out-neighbors of s . Is there always an out-neighbor w of s whose shortest-path distance $\text{dist}(s, w)$ we can easily determine?

Suppose you know that there are three edges out of s , weighted as follows:



You don't know anything else about the rest of the graph. Is there an edge among these three that you know *must* be a shortest path? It seems plausible that the cheapest edge (s, b) should be. Why? Can there be a shorter path from s to b ? No, because any path from s to b has to leave s along one of the three edges. That edge has weight at least 2, and after that (because edge weights are non-negative) the path can only get longer.

This reasoning works in general, giving us a rule for determining the first greedy step:

Lemma 5.15. Among edges leaving s , let (s, w') be one of minimum weight. Then the edge (s, w') is a shortest path from s to w' . That is, $\text{dist}(s, w') = \text{wt}(s, w')$.

Proof (short form). We first show $\text{dist}(s, w') \leq \text{wt}(s, w')$, then show $\text{dist}(s, w') \geq \text{wt}(s, w')$.

Part (i). The edge (s, w') is one possible path from s to w' . So by definition, $\text{dist}(s, w') \leq \text{wt}(s, w')$.

Part (ii). To show $\text{dist}(s, w') \geq \text{wt}(s, w')$, we show that every path from s to w' must have weight at least $\text{wt}(s, w')$. The idea is that any path from s to w' must leave s , and leaving s (on any edge) costs at least $\text{wt}(s, w')$. And all remaining edges on the path have non-negative weight.

Consider an arbitrary path $P_{w'}$ from s to w' . Let (s, x) be the first edge on $P_{w'}$. Let $P_x = P_{w'} \setminus \{(s, x)\}$ be the remaining path from x to w' . Then

$$\begin{aligned} \text{wt}(P_{w'}) &= \text{wt}(s, x) + \text{wt}(P_x) && (\text{since } P_{w'} = \{(s, x)\} \cup P_x) \\ &\geq \text{wt}(s, x) + 0 && (\text{since } \text{wt}(P_x) \geq 0) \\ &\geq \text{wt}(s, w') && (\text{by the choice of } w'). \end{aligned}$$

So $\text{wt}(P_{w'}) \geq \text{wt}(s, w')$. This holds for an arbitrary $s \rightsquigarrow w'$ path $P_{w'}$, so every $s \rightsquigarrow w'$ path has weight at least $\text{wt}(s, w')$. That is, $\text{dist}(s, w') \geq \text{wt}(s, w')$.

By Parts (i) and (ii) above, $\text{dist}(s, w') = \text{wt}(s, w')$. □

Extending the first step. The next step in the greedy-algorithm design pattern is to generalize the first step. We want a rule that lets us determine the distance $\text{dist}(s, w)$ from s to some vertex w that is not yet in K . If we had such a rule, the algorithm could apply it repeatedly, building up the set K of known vertices one at a time. It would stop when the distance to each vertex is known. Such an algorithm would have the following general form:

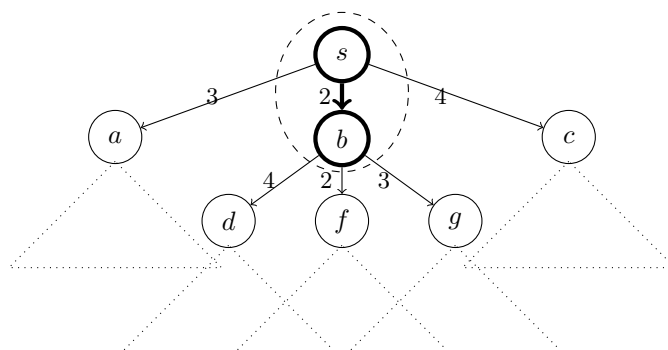
```
greedyShortestPaths( $G = (V, E), s$ ):
1. initialize  $K \leftarrow \{s\}$  and  $d[s] = 0$            ...  $K$  will be the set of vertices whose distances are known
2. until the distance from  $s$  to every vertex is known:
3.1. somehow select a vertex  $w' \in V \setminus K$          note:  $V \setminus K$  is the set of vertices in  $V$  but not  $K$ 
3.2. somehow compute  $d[w'] = \text{dist}(s, w')$ 
3.3. add  $w'$  to  $K$ 
4. return the array  $d$ 
```

As long as it can somehow correctly set $d[w'] = \text{dist}(s, w')$ in Step 3.2, it will maintain the greedy invariant—that the partial solution so far is correct:

greedy invariant: Each vertex $v \in K$ satisfies $d[v] = \text{dist}(s, v)$.

So, in each iteration, how can we find a suitable vertex $w' \in V \setminus K$, one for which the algorithm can easily compute $\text{dist}(s, w')$?

Returning to the previous example, after determining that the edge (s, b) is a shortest path, suppose you find that there are three edges out of b , as follows:



You know that $\text{dist}(s, s) = 0$ and $\text{dist}(s, b) = 2$. Can you determine the distance from s to a third vertex?

What about vertex f ? Do you know that the distance from s to f is 4? No, because there could (for example) be a path from a to f of weight 0.

What about a ? Does the edge (s, a) have to be a shortest path? Can there be a path from s to a of weight less than 3? No, because any other path to a would have to leave the set $\{s, b\}$ via one of the five edges that leaves the set. By checking each edge, we can see that any path that leaves through that edge would cost

3 or more. For example, a path that leaves through the edge (b, f) incurs weight at least 2 to get to b , and then another 2 for (b, f) .

Let's try to generalize this reasoning. Returning to the general case, say that a *path out of K* is a path P from s to any vertex w not in K . Among all paths out of K , let P^* be one of minimum total weight. Let w' be the destination vertex of P^* . Then P^* must be a shortest path from s to w' . Every path from s to w' must have weight at least $w(P^*)$, because it must start with some path out of K . So, if we could find such a path P^* , we would know that $\text{dist}(s, w') = \text{wt}(P^*)$.

The algorithm will find such a path P^* as follows. Consider just the edges leaving K , that is, $(u, w) \in E$ such that $u \in K$ and $w \notin K$. Among these edges, choose an edge (u', w') that minimizes

$$d[u'] + \text{wt}(u', w') = \text{dist}(s, u') + \text{wt}(u', w').$$

Take P^* to be a path consisting of a shortest path from s to u' , followed by the edge (u', w') . We'll show later that this path is indeed a minimum-weight path out of K . Here is the algorithm:

Dijkstra($G = (V, E), s$): — *Dijkstra's algorithm for Single-Source Shortest Paths* —

1. initialize $K \leftarrow \{s\}$ and $d[s] = 0$
2. while some edge in E leaves K (that is, $\exists(u, w) \in E : u \in K, w \notin K$):
 - 3.1. among edges leaving K , choose edge (u', w') that minimizes $d[u'] + \text{wt}(u', w')$
 - 3.2. set $d[w'] = d[u'] + \text{wt}(u', w')$
 - 3.3. add w' to K
4. set $d[w] = \infty$ for all $w \in V \setminus K$
5. return d

We explain Steps 2 and Step 4, later. First we show that the loop maintains the invariant:

Lemma 5.16. *The loop maintains the invariant $d[v] = \text{dist}(s, v) < \infty$ for all $v \in K$.*

Proof (long form).

1. Consider any execution of the algorithm on some input $(G = (V, E), s)$.
2. The invariant holds at the start of the first iteration, when $K = \{s\}$ and $d[s] = 0 = \text{dist}(s, s)$. (Because G has no negative-weight cycles.)
 - 3.1. Consider any iteration of the loop such that the invariant holds at the start of the iteration.
 - 3.2. Let K and d denote the set K and the array d at the *start* of the iteration.
 - 3.3. Let (u', w') be the edge chosen in the iteration.
 - 3.4. To show that the iteration maintains the invariant, we show $\text{dist}(s, w') = d[u'] + \text{wt}(u', w')$.
 - 3.5. First we show $\text{dist}(s, w') \leq d[u'] + \text{wt}(u', w')$.
 - 3.6. Since $u' \in K$, by the invariant, $\text{dist}(s, u') = d[u'] < \infty$.
 - 3.7. So there exists a path from s to u' of weight $d[u']$.
 - 3.8. This path plus the edge (u', w') form a path from s to w' of weight $d[u'] + \text{wt}(u', w')$.
 - 3.9. So there exists a path from s to w' of weight $d[u'] + \text{wt}(u', w')$.
 - 3.10. So $\text{dist}(s, w') \leq d[u'] + \text{wt}(u', w')$, as desired.
 - 3.11. To finish we'll show $\text{dist}(s, w') \geq d[u'] + \text{wt}(u', w')$.
 - 3.12.1. Let P be any path from s to w' . We'll show $\text{wt}(P) \geq d[u'] + \text{wt}(u', w')$.
 - 3.12.2. P starts in K , but ends outside of K , so some edge on P leaves K .
 - 3.12.3. Let (x, y) be an edge on P that leaves K (so $x \in K, y \notin K$).
 - 3.12.4. Separate the path P around the edge (x, y) into the part before, P_x , and the part after, P_y .
 - 3.12.5. So $P = P_x \cup \{(x, y)\} \cup P_y$ and P_x is a path from s to x . Then

$$\begin{aligned}
 \text{wt}(P) &= \text{wt}(P_x) + \text{wt}(x, y) + \text{wt}(P_y) && (\text{Since } P = P_x \cup \{(x, y)\} \cup P_y.) \\
 &\geq \text{wt}(P_x) + \text{wt}(x, y) && (\text{As all edge weights are non-negative, so } \text{wt}(P_y) \geq 0.) \\
 &\geq \text{dist}(s, x) + \text{wt}(x, y) && (\text{Since } P_x \text{ is a path from } s \text{ to } x.) \\
 &= d[x] + \text{wt}(x, y) && (\text{Since } x \in K \text{ so } d[x] = \text{dist}(s, x) \text{ by the invariant.}) \\
 &\geq d[u'] + \text{wt}(u', w') && (\text{By the alg's choice of } (u', w'), \text{ and } x \in K, y \notin K)
 \end{aligned}$$

- 3.13. By Block 3.12, every path from s to w' has weight at least $d[u'] + \text{wt}(u', w')$.
- 3.14. That is, $\text{dist}(s, w') \geq d[u'] + \text{wt}(u', w')$.
- 3.15. By this and Step 3.10, $\text{dist}(s, w') = d[u'] + \text{wt}(u', w')$.
- 4. The invariant holds initially. By Block 3, each iteration maintains it, so it holds throughout. □

Given that the invariant holds, correctness is easy to verify:

Theorem 5.8. *Dijkstra is correct.*

Proof (long form).

- 1. Consider the execution of the algorithm on an arbitrary input (G, s) .
- 2. Consider the time just before Line 4 executes.
- 3. By Lemma 5.16, at that time, $d[v] = \text{dist}(s, v)$ for each $v \in K$.
- 4. By the loop condition, no edges leave K , so vertices in $V \setminus K$ are not reachable from s .
- 5. So each remaining vertex $v \in V \setminus K$ has $\text{dist}(s, v) = \infty$.
- 6. So, after Line 4 executes, each vertex $v \in V \setminus K$ also has $d[v] = \text{dist}(s, v)$.
- 7. So the distances returned by the algorithm are correct. □

Shortest-path trees. Given an instance (G, s) , a *shortest-path tree* T is a (directed) tree, rooted at s , whose vertices are the vertices in G that are reachable from s , such that, for each reachable vertex v , the path from s to v in T is a shortest path from s to v .

For any execution of the algorithm, each iteration selects some edge (u', w') . The collection of these edges (over all iterations) form such a tree T (where the parent of w' in T is u'). So the algorithm can easily be augmented to compute a shortest-path tree T , if desired.

Efficient implementation. Let $n = |V|$ and $m = |E|$ denote the number of edges of a given instance. The algorithm has at most n iterations, as each iteration adds a vertex to K . A naive implementation will take linear time per iteration, giving overall run time of $O(n(m + n))$. But it is easy to improve this.

Lemma 5.17. *Dijkstra's algorithm can be implemented to run in time $O((m + n) \log n)$.*

Proof sketch. Maintain the edges leaving K (that is, $(u, w) \in E \cap (K \times (V \setminus K))$) in a heap⁴ keyed by $d[u] + \text{wt}(u, w)$. Then implement each iteration as follows: to find the edge (u', w') , use a **delete-min** operation on the heap (if the edge returned no longer leaves K , discard it and repeat). Then, when adding w' to K , maintain the heap by adding all edges from w' into $V \setminus K$ to the heap. (Edges that lead into w' are left in the heap; they are discarded when **delete-min** later returns them.)

Each heap operation takes time $O(\log n)$. Each edge is added to the heap at most once and removed from it at most once, so the total time is $O((m + n) \log n)$. □

Be aware that most texts describe the algorithm and its implementation differently. Instead of considering $d[u] + \text{wt}(u, w)$ for each edge (u, w) leaving K , they present the algorithm as considering, for each vertex $w \notin K$, the quantity

$$\delta(w) = \min\{d[u] + \text{wt}(u, w) : u \in K, (u, w) \in E\}.$$

That is, for each $w \notin K$, the algorithm groups all edges into w into a single group, and considers only the best edge (u, w) into w . Then, the implementation maintains a heap of the *vertices* in $V \setminus K$, where each vertex $w \in V \setminus K$ is keyed by $\delta(w)$. In each iteration, it chooses vertex $w \notin K$ minimizing $\delta(w)$, and sets $d[w] = \delta(w)$.

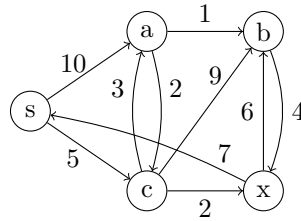
This algorithm can be viewed as a particular way of implementing the algorithm we presented earlier, and in that sense the proof of correctness applies to this implementation too.

Python code: [greedy_algorithms/Dijkstra.py](#) (Section A.3).

⁴a.k.a. a priority queue, see KT 2.5

5.8.1 Exercises

Exercise 5.8. (*Simulate Dijkstra*) Manually simulate Dijkstra's algorithm (from the lecture notes) on the directed graph G below with source s .



- For each of the first four iterations, what is the state of the algorithm at the start of iteration? (The state consists of the value of the variable K , and the values of $d[v]$ for $v \in V$.) What edge (u', w') does the algorithm choose in the iteration?
- What shortest-path tree (from s) is computed by the algorithm on this input? (This is the tree formed by edges chosen by the algorithm in each iteration.)
- Let G' be the graph obtained from G (shown above) by reducing the weight of edge (a, c) to -10 . What is the array d of distances returned by Dijkstra's algorithm given input G' with source s ? Show $d[s]$, $d[a]$, $d[b]$, $d[c]$, and $d[x]$.

Exercise 5.9. (*Dijkstra counter-example*)

- Describe an edge-weighted digraph G and source s on which Dijkstra's algorithm *fails*. That is, it returns an array d such that $\exists v \in V : d[v] \neq \text{dist}(s, v)$. Your instance must have three vertices, no cycles, and three (directed) edges. *All but one* of the edge-weights must be non-negative. (The third must be negative.)
- Explain clearly what the two iterations of the algorithm do when it executes on the instance you give in Part (a).

Exercise 5.10. (*Dijkstra proof*) Is the following conjecture true or false? Either give a long-form proof of the conjecture (as a theorem), or prove that it is false by giving a counter-example, and explaining clearly how the algorithm fails on your counter-example.

Conjecture 5.2. Let $G = (V, E)$ be any edge-weighted digraph, and $s \in V$ any vertex. Suppose that every edge that doesn't touch s has non-negative weight. (That is, for every edge $(u, w) \in E$, if $s \notin \{u, w\}$, then $\text{wt}(u, w) \geq 0$. Edges entering or leaving s can have negative weight.) Suppose also that G has no negative-weight cycles. Then Dijkstra's algorithm, if run on input (G, s) , correctly computes the shortest-path distance from s to each vertex $v \in V$.

External exercises without solutions

- DPV 4.1, 4.6–4.9, 4.14, 4.15. JE: some of the exercises in Chapter 8.
- CLRS: ten exercises in Chapter 24 Section 24.3.

5.8.2 External sources on Dijkstra's algorithm

- KT Chapter 4, Section 4.4. DPV Chapter 4, Section 4.4.
- CLRS Chapter 24, Section 24.3. JE Chapter 8, Section 8.6.
- MIT lecture (video):
<https://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-006-introduction-to-algorithms-fall-2011/lecture-videos/lecture-15-single-source-shortest-paths-problem/>

5.9 Minimum Spanning Trees

In this section, we first consider an iterative algorithm that is designed to maintain the greedy invariant:

greedy invariant: the current (partial) solution can be extended to some correct (full) solution.

This invariant implies correctness. We then consider how the algorithm can instead be viewed as making one greedy choice, then recursing. This view also yields other algorithms.

Problem Definition. The Minimum Spanning Tree (MST) problem is defined as follows:

input: A connected, undirected (!), edge-weighted graph $G = (V, E)$.

output: A minimum-weight *spanning tree* of G . (A spanning tree is a tree T in G whose vertex set includes all the vertices. The weight of T is the sum of its edges' weights, so $\text{wt}(T) = \sum_{e \in T} \text{wt}(e)$.) We represent T as the set of its edges.

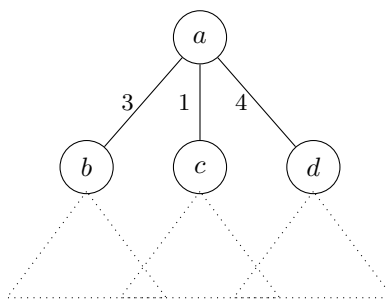
Intuitively, the problem is to connect all vertices in the graph at minimum cost.

[Consider some examples to get intuition.]

5.9.1 Iterative approach (Prim's algorithm)

We have in mind that the greedy algorithm will build a tree edge by edge. To build intuition, let's start by considering candidate rules for the first step.

Suppose you know that, for some vertex v , there are three edges out of v , weighted as follows:



You don't know anything else about the rest of the graph. Is there an edge among these three that you know *must* in at least one MST?

It seems plausible that the cheapest edge (a, c) should be. Let's conjecture as follows:

Conjecture 5.3. Let $G = (V, E)$ be any connected, edge-weighted graph. Let u' be any vertex in V . Among edges leaving u' , let (u', w') be one of minimum weight. Then there exists an MST of G that contains (u', w') .

If the conjecture is true, then, in the example, (a, c) must be in some MST.

Can we prove the conjecture in general? Consider any graph G and edge (u', w') , as in the conjecture. Consider any minimum-spanning tree T^* for G . What can we say about T^* ? In the case that (u', w') happens to be in T^* , then the conjecture holds for G . But what about the case where (u', w') is not in T^* ? Can we use an exchange argument, somehow modifying T^* to add (u', w') so as to obtain an MST that does contain (u', w') ?

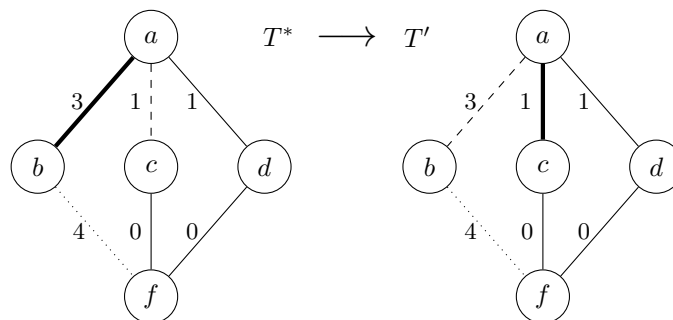
Because T^* is connected and every vertex is in T^* , there has to be at least one edge in T^* into u' . Let (u', w^*) be an edge in T^* incident to u' . What if we construct T' from T^* by removing (u', w^*) and replacing it by (u', w') ? That is, let $T' = \{(u', w')\} \cup T^* \setminus \{(u', w^*)\}$.

Then the weight of T' is no larger than the weight of T^* , because

$$\text{wt}(T') = \text{wt}(T^*) + \text{wt}(u', w') - \text{wt}(u', w^*) \leq \text{wt}(T^*).$$

(Here we are using that (u', w') has minimum weight among edges leaving u' , so $\text{wt}(u', w') \leq \text{wt}(u', w^*)$.) And T' does contain (u', w') . Can we conclude that T' is a minimum-spanning tree that contains (u', w') , as desired?

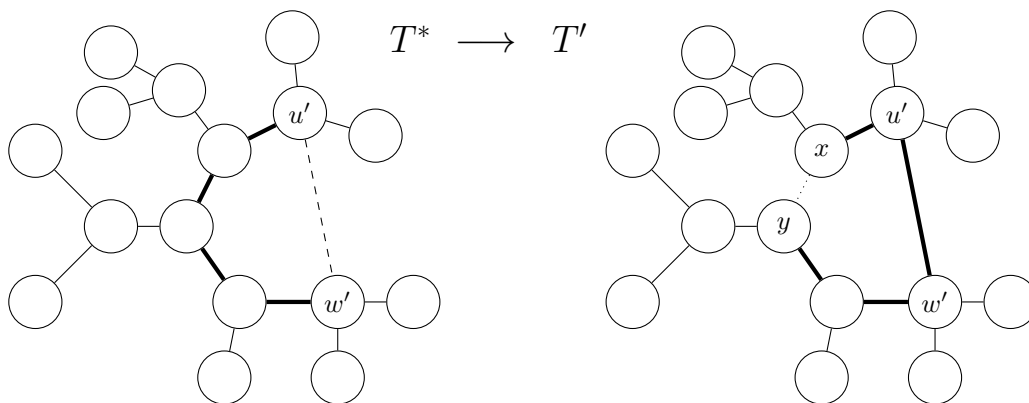
No! Our reasoning has a gap that hides an error. As we've defined T' , it might not be a spanning tree! Consider the 5-vertex graph G shown on the left below. Consider the minimum spanning tree T^* formed by the four solid edges in the graph on the left.



The edge (a, c) is a minimum-weight edge among edges out of a , but (a, c) is not in T^* . The proposed reasoning suggests that we obtain T' from T^* by adding (a, c) and removing any other edge out of a . So we could, say, remove edge (a, b) . (In the notation above, a is u' , c is w' , and b is w^* .) So $T' = \{(a, c)\} \cup T^* \setminus \{(a, b)\}$. This is the set of solid edges shown to the right above. But this set T' is not a spanning tree, so our proposed reasoning is flawed!

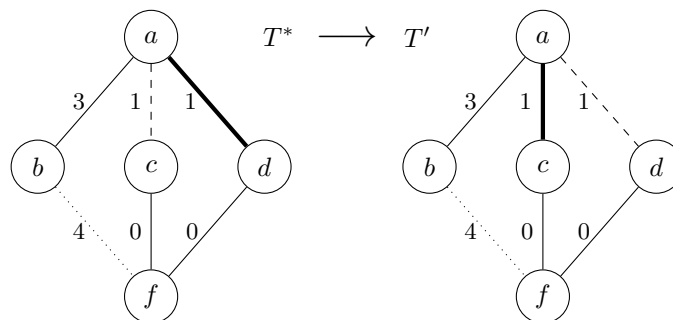
We need to study more carefully what kinds of edge exchanges we can do with spanning trees. The next observation describes the main kind of edge exchange that we'll use.

Observation 5.8. Let T^* be any spanning tree of G . Let (u', w') be any edge not in T^* , and let (x, y) be any edge that lies on the path in T^* between u' and w' . Then adding (u', w') to T^* and removing (x, y) gives a spanning tree $T' = \{(u', w')\} \cup T^* \setminus \{(x, y)\}$.



As illustrated above, the observation holds because adding (u', w') to T^* creates a single cycle (with the path from u' to w' in T^*). Edge (x, y) lies on that cycle, so after removing (x, y) the resulting set of edges T' is acyclic, and therefore a spanning tree. (Alternatively, removing (x, y) from T^* disconnects T^* into two disconnected subtrees. One of these contains u' , the other contains w' . So adding (u', w') joins the two subtrees into a single spanning tree T' .)

Returning to the example, with $(u', w') = (a, c)$, we can take (x, y) to be any edge on the path from a to c in T^* . That is, one of the edges (a, d) , (d, f) , or (f, c) . Let's choose (x, y) to be (a, d) , the other edge incident to a . Using this edge exchange gives the tree T' shown on the right below:



Now let's try again to prove the conjecture. Consider any graph G and edge (u', w') , as in the conjecture. (So (u', w') has minimum weight among edges leaving u' .) Consider any minimum-spanning tree T^* for G . If (u', w') happens to be in T^* , then the conjecture holds for (u', w') . But what about the case where (u', w') is not in T^* ? Let's try an exchange argument, with a more careful choice of edge to replace.

Let P be the path from u' to w' in T^* . Let (u', w^*) be the edge incident to u' on this path P . (In the example, (u', w^*) is (a, d) .) Construct T' from T^* by removing (u', w^*) and replacing it by (u', w') , so $T' = \{(u', w')\} \cup T^* \setminus \{(u', w^*)\}$. Now, by Observation 5.8, because (u', w^*) is on the path from u' to w' in T^* , the resulting set T' of edges must be a spanning tree.

As before, the weight of (u', w') is at most the weight of (u', w^*) , because, among edges leaving u' , edge (u', w') (by definition) has minimum weight, so

$$\text{wt}(T') = \text{wt}(T^*) + \text{wt}(u', w') - \text{wt}(u', w^*) \leq \text{wt}(T^*).$$

Because T^* is an MST, this implies that T' is also an MST.⁵ And T' does contain (u', w') . Hence, there is an MST that contains (u', w') .

So we think we have a line of reasoning by which we know that the conjecture is true. Here is the conjecture, stated as a lemma, with a proof that records the details of our reasoning.

Lemma 5.18. *Let $G = (V, E)$ be any connected, edge-weighted graph. Let u' be any vertex in V . Among edges leaving u' , let (u', w') be one of minimum weight. Then there exists an MST of G that contains (u', w') .*

Proof (short form). Let T^* be any minimum spanning tree in G . If edge (u', w') is in T^* , we are done. Otherwise, obtain a spanning tree T' from T^* using Observation 5.8 to add (u', w') , selecting the edge (u', w^*) to remove carefully, as follows. By definition, the edge (u', w') has minimum weight among edges that are incident to u' . Let P be the path in T^* connecting u' to w' . Let (u', w^*) be the edge incident to u' on that path P . Then take $T' = \{(u', w')\} \cup T^* \setminus \{(u', w^*)\}$.

By Observation 5.8, T' is a spanning tree. The weight of T' is at most the weight of T^* , because $\text{wt}(T') = \text{wt}(T^*) + \text{wt}(u', w') - \text{wt}(u', w^*)$, and (u', w') has minimum weight among all edges leaving u' , so $\text{wt}(u', w') \leq \text{wt}(u', w^*)$. So T' is also a minimum-weight spanning tree, and by construction it contains (u', w') . $\square$

Extending beyond the first edge. Fix some input $G = (V, E)$. The algorithm will start by choosing some arbitrary vertex v_0 in V , and, among edges leaving v_0 , choosing some edge (v_0, v_1) of minimum weight. It will choose (v_0, v_1) as its first edge. By the lemma, this edge is guaranteed to be in some MST.

After committing to connecting v_0 and v_1 via the edge (v_0, v_1) , one natural way to continue is to choose the cheapest edge leaving the set $\{v_0, v_1\}$. Then we will have three connected vertices, say $\{v_0, v_1, v_2\}$. We could then take the cheapest edge leaving those three vertices, and so on. This idea gives us the following algorithm:

⁵And so, in fact, it must be that $\text{wt}(T') = \text{wt}(T^*)$, and also $\text{wt}(u', w') = \text{wt}(u', w^*)$.

$\text{Prim}(G = (V, E))$: 1. choose any vertex v_0 in V 2. initialize $T = \emptyset$ and $V_T = \{v_0\}$ 3. for $t = 1, 2, \dots, n - 1$, where $n = V$: 4.1. among edges leaving V_T (that is, among $\{(u, w) \in E : u \in V_T, w \notin V_T\}$) let (u', w') be one of minimum weight 4.2. add (u', w') to T and add w' to V_T 5. return T	— G is a connected, edge-weighted graph
--	---

This is commonly called Prim's algorithm, although Prim was not the first to discover it. (See https://en.wikipedia.org/wiki/Prim's_algorithm.)

Correctness. As promised at the start, we'll show that the algorithm maintains the greedy invariant. For this algorithm, the invariant can be stated specifically as follows:

greedy invariant: *the tree T is a subtree of some MST of G*

Lemma 5.18 shows that the invariant holds after the first iteration. By using more or less the same reasoning, we show that every iteration maintains the invariant.

Lemma 5.19. *Throughout any execution of the algorithm, the invariant holds.*

Proof (long form).

1. Consider any execution of the algorithm on some connected graph $G = (V, E)$.
2. At the start of the first iteration, T is a singleton vertex.
3. Since G is connected, it has some MST, so the invariant holds initially.
4. Next we show that each iteration maintains the invariant.
 - 5.1. Consider any iteration such that the invariant holds at the start of the iteration.
 - 5.2. Let T be the subtree just before the iteration.
 - 5.3. Let (u', w') be the edge chosen in the iteration.
 - 5.4. So (u', w') leaves V_T , and has minimum weight among edges leaving V_T .
 - 5.5. The invariant holds at the start of the iteration, so T is a subtree of some MST, say T^* .
 - 5.6.1. Case 1. Suppose T^* contains (u', w') as well.
 - 5.6.2. Then $T \cup \{(u', w')\}$ is also a subtree of T^* , so the invariant holds at the end of the iteration.
 - 5.7.1. Case 2. Otherwise (u', w') is not in T^* .
 - 5.7.2. The edge (u', w') leaves V_T , so at least one edge on the path in T^* from u' to w' leaves V_T .
 - 5.7.3. Let (x, y) be such an edge.
 - 5.7.4. By Observation 5.8, adding (u', w') to T^* and removing (x, y) gives the spanning tree T' where

$$T' = \{(u', w')\} \cup T^* \setminus \{(x, y)\}.$$
 - 5.7.5. Edge (x, y) leaves V_T , while edge (u', w') has minimum weight among edges leaving V_T .
 - 5.7.6. So $\text{wt}(u', w') \leq \text{wt}(x, y)$. This implies that $\text{wt}(T') = \text{wt}(T^*) + \text{wt}(u', w') - \text{wt}(x, y) \leq \text{wt}(T^*)$.
 - 5.7.7. So T' is also a minimum-weight spanning tree of G .
 - 5.7.8. Edge (x, y) leaves V_T , so (x, y) is not in T .
 - 5.7.9. So, at the end of the iteration, the new subtree $T \cup \{(u', w')\}$ is a subtree of MST T' .
 - 5.7.10. So the invariant holds at the end of the iteration.
- 5.8. By Cases 1 and 2 above, the invariant holds at the end of the iteration.
6. As observed previously, the invariant holds at the start of the first iteration.
7. By Block 5, each iteration maintains the invariant. So it holds throughout. □

Correctness of the algorithm follows easily from the invariant:

Theorem 5.9. *Prim's algorithm is correct. Given any connected edge-weighted graph G , it returns a minimum spanning tree for G .*

Proof (short form). Consider any execution of the algorithm on any input G . By Lemma 5.19, the invariant holds after the last iteration of the algorithm, so the final subtree T is a subtree of some minimum spanning tree T^* of G . Since each of the $n - 1$ iterations adds a new vertex to T , at termination T contains all n vertices, so T is itself a spanning tree of G . So $T = T^*$. That is, T is a minimum spanning tree. $\square$

Run time. For efficiency, the algorithm can maintain the edges (u, w) leaving V_T in a heap, keyed by $\text{wt}(u, w)$. Then choosing the best edge (u', w') in each iteration takes $O(\log n)$ time. The algorithm maintains the heap when w' is added to V_T as follows. For each edge (w', x) incident to w' , if x is in V_T , then it removes (w', x) from the heap, and otherwise (x is not in V_T) it adds (w', x) to the heap. The time for the iteration is then $O((1 + \text{degree}(w')) \log n)$, so the total time is $O(\log n \sum_{w' \in V} 1 + \text{degree}(w')) = O((n + m) \log n)$:

We have the following result:

Theorem 5.10. *There is an $O((m + n) \log n)$ -time algorithm for the Minimum Spanning Tree problem.*

This algorithm is similar to Dijkstra's shortest-path algorithm.⁶

Python code: [greedy_algorithms/min_spanning_tree.py](#) (Section A.3).

5.9.2 A recursive view

Many greedy algorithms make one greedy choice, then recurse. Unwinding the tail recursion gives an iterative algorithm. Next we describe how Prim's algorithm (and others) can be developed via this recursive approach, instead of by the iterative view in the previous section.

Recall the first greedy step: choose any vertex u' and an edge (u', w') of minimum weight among edges leaving u' , then commit to taking edge (u', w') . By Lemma 5.18, this edge is in some MST.

To continue, don't proceed iteratively as in the previous section. Instead, imagine *contracting* the edge (u', w') into a single vertex. Contraction replaces u' and w' in the graph by a single new vertex, say v' , which inherits all of the edges incident to u' or w' (except for the edge (u', w') , which is deleted). Let G' be the graph obtained from G by contracting (u', w') .

With some thought, one can show, for any set R of edges (except (u', w')) that $\{(u', w')\} \cup R$ is a spanning tree of G if and only if R is a spanning tree of G' . Using this and an easy edge-weight calculation, one can prove this observation:

Observation 5.9. *Let R be any MST of G' . Then $\{(u', w')\} \cup R$ is an MST of G .*

Hence, to find a set R of edges such that $\{(u', w')\} \cup R$ is an MST, it suffices to compute an MST in G' , and take R to be that one. This gives the following recursive algorithm:

rGreedyMST($G = (V, E)$): 1. if $V = 1$: return $\emptyset$ 2. choose any vertex u' in V 3. among edges leaving u', let (u', w') be one of minimum weight 4. let G' be the graph obtained from G by contracting (u', w') 5. $R \leftarrow \text{rGreedyMST}(G')$ 6. return $\{(u', w')\} \cup R$	— G is a connected, edge-weighted graph
---	---

This algorithm is correct by induction, using Observation 5.9 to show the inductive step. (By induction, R is an MST of G' , so by the observation $\{(u', w')\} \cup R$ is an MST of G .)

Prim's algorithm can be viewed as the above algorithm, where in each recursive call the vertex u' is taken to be the new vertex that was just formed by the edge contraction in the parent call. This way, as the algorithm proceeds, the edges contracted so far form just one connected subtree in the original graph. This is the subtree T in the iterative version of Prim's algorithm.

⁶In each iteration, the algorithm chooses, among edges (u, w) leaving V_T , one that minimizes $\text{wt}(u, w)$, whereas Dijkstra's algorithm chooses, among edges (u, w) leaving K , one that minimizes $d[u] + \text{wt}(u, w)$.

Kruskal's algorithm. We can obtain other greedy MST algorithms by choosing the vertex u' in other particular ways in each call. Consider the following rule: *let (u', w') be any edge of minimum weight in the entire graph.* (We can have the algorithm choose this u' in Step 2, and, in Step 3, have it take edge (u', w') out of u' . This edge is guaranteed to be a minimum-weight edge out of u' , so it is a correct choice for the algorithm.) So the the following recursive algorithm is also correct:

rGreedy2($G = (V, E)$): — G is a connected, edge-weighted graph

1. if $|V| = 1$: return $\emptyset$
2. let (u', w') be a minimum-weight edge in G
3. let G' be the graph obtained from G by contracting (u', w')
4. $R \leftarrow \text{rGreedy2}(G')$
5. return $\{(u', w')\} \cup R$

Next we unwind the tail recursion in this algorithm, and, instead of contracting each chosen edge, we just maintain the set F of edges that have been chosen so far. (As the algorithm proceeds, F will form a forest—a collection of disjoint subtrees. If we were to contract the edges in F , each subtree would contract to a vertex giving us back the contracted graph in the algorithm above.) This yields the following iterative algorithm:

iGreedy2($G = (V, E)$):

1. initialize F to be the forest with vertex set V and no edges. let $n = |V|$.
2. for $t = 1, 2, \dots, n - 1$:
- 3.1. among edges connecting any two subtrees in F , let (u', w') be one of minimum weight
- 3.2. add (u', w') to F (joining the two trees into one)
4. return F

Each iteration adds a minimum-weight edge among edges connecting different subtrees, so we can also implement this algorithm as follows:

Kruskal($G = (V, E)$): — *Kruskal's algorithm* —

1. initialize F to be the forest with vertex set V and no edges. let $n = |V|$.
2. let $e_1, e_2, \dots, e_m$ be the edges in E , ordered by increasing weight
3. for $t = 1$ to m :
4. if e_t connects different subtrees in F : add e_t to F *but how do we check this?*
5. return F

This is known as Kruskal's algorithm. We haven't specified how to determine whether e_t connects different subtrees in F . One naive way is to use depth-first search in F , but that takes linear time, leading to a quadratic-time algorithm. A better way uses a *disjoint-set* data structure. We haven't studied it, but, using that data structure, the total time will be dominated by the time to sort the edges. So the total time will be $O((m + n) \log n)$, where $m = |E|$ and $n = |V|$.

Python code: [greedy_algorithms/min_spanning_tree.py](#) (Section A.3).

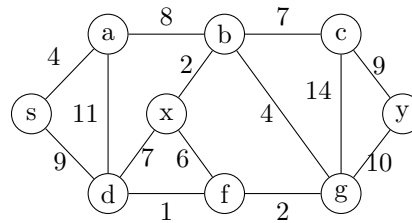
5.9.3 Exercises

External exercise with solutions

- [KT Solved Exercises 3 \(MSTs\)](#)

Exercises without solutions

Exercise 5.11. (*Simulate Prim*) Manually simulate Prim's algorithm (from the lecture notes) on the graph G below, starting from source s . List the edges that the algorithm chooses, in the order that it chooses them.



Exercise 5.12. (*MST proof check*)

Conjecture 5.4. *Let G be any connected edge-weighted graph G with at least two vertices. Let v be any vertex, and, among edges leaving v , let (v, w^*) be one of minimum weight. Suppose every other edge incident to v has larger weight. Then (v, w^*) is in every minimum spanning tree of G .*

Proof (long form).

1. Consider any such graph G , vertex v , and edge (v, w^*) ,
- 2.1. Consider any spanning tree T of G that does not contain (v, w^*) .
- 2.2. Since T is connected and contains all vertices, T contains some edge incident to v .
- 2.3. Let (v, w') be such an edge. Note that $w' \neq w^*$ as (v, w^*) is not in T .
- 2.4. Let $T' = \{(v, w^*)\} \cup T \setminus \{(v, w')\}$ be obtained from T by removing (v, w') and adding (v, w^*) .
- 2.5. Then $\text{wt}(v, w^*)$ is strictly less than $\text{wt}(v, w')$. That is, $\text{wt}(v, w^*) < \text{wt}(v, w')$.
- 2.6. So $\text{wt}(T') = \text{wt}(T) + \text{wt}(v, w^*) - \text{wt}(v, w') < \text{wt}(T)$.
- 2.7. So T' is a spanning tree whose total weight is strictly less than the total weight of T .
- 2.8. So T is not a minimum spanning tree.
3. By Block 2, every spanning tree of G that doesn't contain (v, w^*) isn't a minimum spanning tree.
4. So every minimum spanning tree of G contains (v, w^*) . □

Is the reasoning in the proof correct? If not, what is the first step that makes an assertion that is not necessarily true, given the assumptions? Why exactly is that assertion not necessarily true?

Exercise 5.13. (*Reverse greedy MST*) Here's another kind of greedy algorithm to consider. Start with all edges in the graph, and *remove* edges one by one, until the remaining edges form a spanning tree. Can you come up with an algorithm of this form?

- (a) Define an iterative MST algorithm based on the following rule:
Let C be any cycle in the remaining graph, and remove any maximum-weight edge on C .
- (b) Prove that the rule maintains this invariant:
Some minimum spanning tree of the original graph is a subset of the remaining edges.
- (c) Prove that the algorithm is correct (use that it maintains the invariant).

As much as possible, follow the structure of the proofs in Section 5.9.1.

External exercises without solutions

- [JE](#) Chapter 7, Exercises 1–10
- [DPV](#) Exercises 5.1–5.10, 5.21–5.23
- [KT](#) Chapter 4, Exercises 1–3, 8–11, 19–23, 27, 28, 31
- [CLRS](#) Exercises in Chapter 23

5.9.4 External sources on Minimum Spanning Tree

- [DPV](#) Section 5.1; [KT](#) Section 4.5 and [KW slides](#); [CLRS](#) Chapter 23; [JE](#) Chapter 7
- MIT lecture video:
<https://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-046j-introduction-to-algorithms-sma-5503-fall-2005/video-lectures/lecture-16-greedy-algorithms-minimum-spanning-trees/>

5.10 Additional Exercises

Each previous section also contains its own exercises.

5.10.1 External exercises with solutions

- [KT](#) Solved Exercises 1–3

5.10.2 Exercises without solutions

Exercise 5.14. (*Algorithm for Puncturing Intervals*)

The Puncturing Intervals problem is defined as follows:

input: Set $I = \{B_1, B_2, \dots, B_n\}$ of real intervals, where $B_i = [s_i, e_i]$.

output: A minimum-size finite set $P \subset \mathbb{R}$ of real values, such that $B_i \cap P \neq \emptyset$ for each $B_i \in I$.

For example, for the set I of intervals in Exercise 5.3 in Section 5.3 the set $P = \{1, 5, 9, 14\}$ of points has the property that it contains at least one point in each interval in I . However, this set is not a correct solution, because it doesn't have minimum possible size.

- Give a correct solution P for the instance defined by that set I of intervals.
- Define a correct greedy algorithm for this problem. Your algorithm should make one greedy choice for the first point, then recurse. It should be clear that the run time is polynomial.
- Prove that your algorithm is correct. The structure of your proof should be as similar as possible to the proof of correctness for the Activity Selection algorithm in Section 5.3.

External exercises without solutions ⁷

- [JE](#) Chapter 4, Exercises 15–25 (Page 176)
- [DPV](#) 5.1–5.33 (Page 161)
- [KT](#) Chapter 4 Exercises 1–33 (Page 188)
- [CLRS](#): Exercises in Section 16.2 (Page 427) and Problems 16–1 through 16–5 (Page 446)

⁷As mentioned in [Lecture Note 1](#), solutions for exercises in [KT](#) and some in [CLRS](#) can be found online.

Lecture Note 6

Graph Traversal

[↑ CONTENTS](#)

Contents

6.1	Graph terminology and data structures	105
6.2	Breadth-first Search	105
6.3	Depth-first Search	106
6.3.1	The DFS forest: forward, back, cross, and tree edges	107
6.3.2	Finding connected components in undirected graphs in linear time	107
6.3.3	Finding a cycle in an undirected graph in linear time	107
6.3.4	Discovery times and finish times	108
6.3.5	Topologically sorting a directed graph in linear time	108
6.3.6	Strongly connected components in directed graphs	109
6.3.7	Exercises	109
6.3.8	External sources on graphs and linear-time graph traversal	111
6.3.9	Appendix: Proof of correctness for Breadth-First Search	112

6.1 Graph terminology and data structures

Definitions and terminology related to graphs. We assume familiarity with graphs and the following graph-related terms: *edge*, *vertex*, *node*, *endpoint* of an edge, *directed graph*, *path*, *simple path*, *cycle*, *simple cycle*, *connected graph*, *strongly connected graph*, *distance* between two nodes. We also assume familiarity with trees and the following tree-related terms: *rooted tree*, *parent*, *child*, *descendant*, *ancestor*. Given a graph $G = (V, E)$, vertices $u, w \in V$ are *neighbors* if they are connected by an edge (that is, if $(u, w) \in E$). Given a digraph $G = (V, E)$, if the directed edge (u, w) is in E , then w is an *out-neighbor* of u and u is an *in-neighbor* of w .

Data structures for graphs. We generally assume that graphs are stored in *adjacency-list* representation. For given graph $G = (V, E)$, then, V is represented as a collection of vertices (for example, a linked list or array), and associated with each vertex $v \in V$ is a collection (again, a linked list or array) containing the edges leaving v . The space (memory) required for representing G this way is proportional to the *size* of G , that is, the number of edges and vertices, $|V| + |E|$. The representation allows us to enumerate V in time $O(|V|)$, and to enumerate the edges leaving a given vertex v in time proportional to the out-degree of v .

In some cases (for a directed graph), we may also preprocess the graph (in linear time) to compute, for every vertex v , a collection containing the edges *entering* v . Then we can enumerate the vertices into a given v in time proportional to the in-degree of v .

We don't use the *adjacency matrix* representation (which stores the edge set as a 2-dimensional array A such that $A[i, j]$ is 1 if there is an edge from the i th vertex to the j th vertex, and 0 otherwise). It can save some space (a constant factor) for dense graphs (where most pairs of vertices have an edge). But it is much less efficient for sparse graphs.

It can be convenient to store the collection of vertices and the collections of edges in a *hash table* (dictionary). This allows checking in constant time whether a given edge is present. Generally, we store an *undirected* graph $G = (V, E)$ as a directed graph that has both directed edges (u, w) and (w, u) for each (undirected) edge $\{u, w\} \in E$.

For detailed background information on graphs and graph representations, see [JE](#) Chapter 5, [KT](#) Chapter 3, [DPV](#) Chapters 3 and 4, and/or [CLRS](#) Chapter 22.

When designing algorithms that traverse a graph, keep in mind that the algorithm does not have a “global” view of the graph. Given a vertex v , all it has access to is the immediate neighbors of v .

6.2 Breadth-first Search

Given an graph $G = (V, E)$ and a source vertex s , how can we efficiently find all the vertices reachable from s ? *Breadth-first search* starts at s . It then discovers all out-neighbors of s (at distance 1 from s). From those it finds all out-neighbors of those out-neighbors (at distance 2 from s), and so on. The algorithm uses two data structures: an array d and a FIFO queue Q . The queue holds the vertices that have been discovered, but whose out-edges have not yet been explored. As the algorithm proceeds, for each vertex v that has been discovered so far, the algorithm records the distance from s to v (that is, $\text{dist}(s, v)$) in the array entry $d[v]$.

$\text{BFS}(G = (V, E), s)$ G is a directed or undirected graph, in adjacency-list representation

1. initialize FIFO queue $Q = (s)$; initialize distance array d with $d[s] = 0$; initialize array π
2. while Q is not empty:
 - 3.1. pop the next vertex, say u , from the head of the queue Q
 - 3.2. for each edge (u, w) out of u :
 - 3.3. if $d[w]$ is not set, then set $d[w] \leftarrow d[u] + 1$, set $\pi[w] \leftarrow u$, and add w to the tail of Q
4. for each vertex v such that $d[v]$ is not set, set $d[v] = \infty$
5. return d and π

Lemma 6.1 (run time). *Given any directed or undirected graph G and source s , $\text{BFS}(G = (V, E), s)$ runs in time linear in the size of G (that is, in time $O(|V| + |E|)$).*

Proof. Each vertex w is added to the queue at most once, because (by Line 3.3) $d[w]$ is set the first time w

is added, so won't be added again. So each vertex u is popped from the queue in line 3.1 at most once, and the total time for the algorithm is $O(\sum_{u \in V} 1 + \text{out-degree}(u)) = O(|V| + |E|)$. $\square$

Before we sketch the correctness proof, we formally prove some intuitive properties of distances. Fix an input $G = (V, E)$ with source s . For each integer $\ell \geq 0$, define *level* ℓ to be the set of vertices v with $\text{dist}(s, v) = \ell$.

Lemma 6.2 (levels). *Level 0 contains just the source s . Each level $\ell > 0$ consists of the vertices w such that (a) w is not in any level less than ℓ , and (b) there is an edge $(u, w) \in E$ from some vertex u in level $\ell - 1$. That is, each level $\ell > 0$ is $\{w \in V : \text{dist}(s, w) \geq \ell, (\exists(u, w) \in E) \text{dist}(s, u) = \ell - 1\}$.*

A detailed long-form proof (which just requires reasoning carefully about the definitions) is in an appendix to this note. Using the lemma, we can prove that BFS computes distances correctly:

Lemma 6.3 (correctness). *Given any directed or undirected graph G and source s , the array d returned by $\text{BFS}(G = (V, E), s)$ satisfies $d[v] = \text{dist}(s, v)$ for all $v \in V$.*

Roughly, the proof shows by induction that $\text{BFS}(G, s)$ discovers the vertices level by level. It starts with the queue containing just $\{s\}$ (level 0), with $d[s]$ set to 0. When it pops s off the queue, it puts the vertices in level 1 in the queue, and sets $d[w]$ for each level-1 vertex to 1. Then, as it pops the level-1 vertices off the queue, it appends the level-2 vertices to the queue (setting $d[w]$ for each to 2) and so on. In general, as it pops the vertices in any given level $\ell - 1$ off the queue, it appends the vertices in level ℓ to the queue (and correctly sets $d[w]$ for each such vertex to ℓ). A detailed long-form proof is in an appendix to this note.

Note that the edges $\{(\pi[v], v) : 0 < d[v] < \infty\}$ form the so-called *shortest-path tree* T —rooted at s , in which neighbors of s in G are children of s , neighbors of those neighbors are grandchildren, etc. so that, for each $\ell \geq 0$, the nodes at depth ℓ in T are the vertices in level ℓ of G , and the path from s to any node v in T is a shortest path from s to v .

Python code: [graph_traversal/breadth_first_search.py](#) (Section A.4).

External sources on Breadth-First Search

- [JE](#) Sections 5.5–5.6; [KT](#) Sections 3.2–3.3; [DPV](#) Section 4.2; [CLRS](#) Section 22.2.

6.3 Depth-first Search

Breadth-first search expands the search along an even frontier, postponing the exploration of a newly discovered vertex until all vertices closer to s have been explored. In contrast, depth-first-search explores each vertex immediately, as soon as it is discovered.

Depth-first-search does not solve a specific computational problem. Rather, it is used as the basis for designing algorithms for many problems. Here is pseudo-code. No start vertex is given.

DFS($G = (V, E)$)	<i>G is a directed or undirected graph</i>
1. initialize visited $\leftarrow \emptyset$	<i>(vertices visited so far, initially the empty set)</i>
2. define DFS1(u):	<i>(definition of internal subroutine DFS1)</i>
3.1. add u to visited	
3.2. for each edge $(u, w) \in E$ out of u :	
3.3. if $w \notin$ visited: DFS1(w)	
4. for each vertex $u \in V$:	<i>(outer loop that actually calls DFS1)</i>
5. if $u \notin$ visited: DFS1(u)	

Consider executing DFS on any graph $G = (V, E)$. During the execution, when a call to DFS1(u) is made, call this *visiting* vertex u . Each vertex $u \in V$ is visited exactly once, and when u is visited, the inner loop iterates over the edges out of u , so the total time for the execution is $O(\sum_{u \in V} 1 + \text{out-degree}(u)) = O(|V| + |E|)$. That is, DFS runs in linear time.

Python code: [graph_traversal/depth_first_search.py](#) (Section A.4).

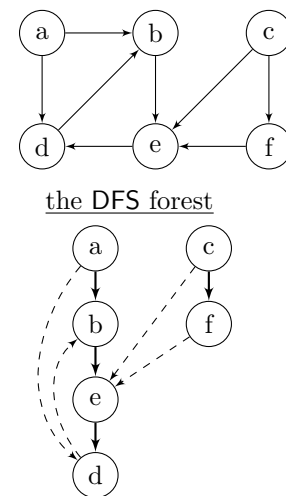
6.3.1 The DFS forest: forward, back, cross, and tree edges

Executing DFS on a graph implicitly defines a *DFS forest*. To explain, consider executing DFS on the graph G shown to the right with $V = \{a, b, c, d, e, f\}$. Assume that in Lines 3.2 and 4, vertices are chosen in alphabetical order. The recursive calls to DFS1 are nested as shown below.

```

DFS1(a)    --- called in outer loop, Line 5
  DFS1(b)  --- called from DFS1(a), Line 3.3
    DFS1(e) --- called from DFS1(b), Line 3.3
      DFS1(d) --- called from DFS1(e), Line 3.3
    DFS1(c) --- called in outer loop, Line 5
      DFS1(f) --- called from DFS1(c), Line 3.3

```



The resulting DFS forest is shown to the right. It consists of two *DFS trees*. The vertices of the DFS forest are the vertices u of G , each representing the single call to DFS1(u) made during the execution. If the call to DFS1(u) recursively calls DFS1(w) (following the edge (u, w)), then w is the child of u in the DFS forest. Then edge (u, w) is called a *tree edge*. In the diagram, the tree edges are the edges drawn with solid lines. The remaining (non-tree) edges in E are not part of the forest. They are nonetheless shown in the diagram, but with dashed lines. Each non-tree edge $(u, w) \in E$ is classified as one of three types, depending on how it interacts with the DFS forest:

If w is a descendant of u in the forest (in the example, (a, d)), then (u, w) is a *forward edge*.

If u is a descendant of w (in the example, (d, b)), then (u, w) is a *back edge*.

The remaining non-tree edges (in the example, (c, e) and (f, e)), are *cross edges*.

Cross edges can only go “right to left”—from a node u to a node w that was visited before u —because otherwise the edge (u, w) would have been traversed when u was visited.

What about *undirected* graphs? For undirected graphs there cannot be any cross edges! Also, each forward edge is also a back edge.

6.3.2 Finding connected components in undirected graphs in linear time

In an undirected graph $G = (V, E)$, the *connected components* are defined as follows. Consider two vertices u and w to be *equivalent* if there is a path between u and w . This relation is reflexive, transitive and symmetric, and partitions the vertex set into equivalence classes $C_1, C_2, \dots, C_k$, each called a *connected component*. For every two vertices u and w , there is a path between u and w if and only if u and w are in the same component C_i . There are no edges *between* different connected components in G .

Given an undirected graph $G = (V, E)$, the connected components of G can be found in linear time by doing a single DFS on G . Then (since there are no cross edges) each tree in the DFS forest is a single connected component.

Python code: [graph_traversal/depth_first_search.py](#) (Section A.4).

6.3.3 Finding a cycle in an undirected graph in linear time

Given an undirected graph $G = (V, E)$, DFS can be used to detect in linear time whether G has a cycle, as follows. After running DFS on G , if there is a back edge (u, w) other than the reverse of a tree edge (that is, with w not the parent of u), then that back edge forms a cycle with the path of tree edges between u and w . Conversely, suppose the DFS produces no such back edge. Since G is undirected, any forward edge would also be such a back edge, so there are no forward edges either. And DFS of an undirected graph cannot produce any cross edges. So, if the DFS produces no such back edge, all of the edges in G are tree edges (or reverses of tree edges). Since the trees in the DFS forest don’t have cycles, G has no cycle. It follows that an undirected graph G has a cycle if and only if DFS on G produces such a back edge.

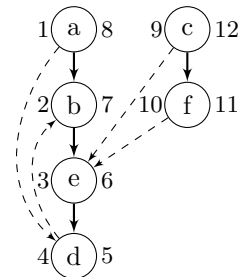
6.3.4 Discovery times and finish times

Executing DFS on a graph implicitly defines a *discovery time* $d[u]$ and a *finish time* $f[u]$ for each vertex $u \in V$. The discovery time is the time when u is first visited, that is, when $\text{DFS1}(u)$ is called. The finish time is the time when $\text{DFS1}(u)$ completes. To explicitly compute these times, we could augment DFS as follows:

```

DFS( $G = (V, E)$ )
1. initialize visited  $\leftarrow \emptyset$ ; initialize  $t \leftarrow 1$ 
2. define  $\text{DFS1}(u)$ :
3.1. add  $u$  to visited; set  $d[u] = t$ ; increment  $t$            (set discovery time for  $u$ )
3.2. for each edge  $(u, w) \in E$  out of  $u$ :
3.3.   if  $w \notin \text{visited}$ :  $\text{DFS1}(w)$ 
3.4. set  $f[u] = t$ ; increment  $t$            (set finish time for  $u$ )
4. for each vertex  $u \in V$ :
5.   if  $u \notin \text{visited}$ :  $\text{DFS1}(u)$ 

```



The discovery and finish times for the example are shown above to the right. The discovery time for each vertex v is shown just to the left of v ; the finish time is shown just to the right. Discovery and finish times of the endpoints of an edge are related, depending on the type of the edge:

Observation 6.1. *In any execution of DFS on a graph $G = (V, E)$, for any edge $(u, w) \in E$,*

If (u, w) is a tree or forward edge, then $d[u] < d[w] < f[w] < f[u]$.

If (u, w) is a back edge, then $d[w] < d[u] < f[u] < f[w]$.

If (u, w) is a cross edge, then $d[w] < f[w] < d[u] < f[u]$. (Happens only if G is a directed graph.)

6.3.5 Topologically sorting a directed graph in linear time

A directed graph $G = (V, E)$ is *topologically ordered* if the vertex set V is given as a sequence $V = (v_1, v_2, \dots, v_n)$ such that each edge $(v_i, v_j) \in E$ has $i < j$. Topological Ordering is the following problem: *Given a directed graph G , reorder G 's vertices so that G is topologically ordered.*

Lemma 6.4. *If digraph G has a cycle, then it cannot be topologically ordered.*

Proof. Suppose for contradiction that G has been topologically ordered, and has a cycle C . Let v_i be the vertex in C with largest index i , so that the edge out of (v_i, v_j) in C must have $i > j$, contradicting that G is topologically ordered. $\square$

So let's amend our problem to the following: *Given a digraph $G = (V, E)$, determine whether G has a cycle, and, if not, topologically order the vertices of G .* We will use DFS to solve it in linear time. Consider any execution of DFS on G . We start with the following observation:

Observation 6.2. (i) *If digraph G has a back edge (u, w) , then G has a cycle.*

(ii) *If G has no back edge, then $f[u] > f[w]$ for each edge $(u, w) \in E$.*

For Part (i), note that the back edge (u, w) forms a cycle with the path from u to its descendant w in the DFS forest. For Part (ii), if there are no back edges, each edge (u, w) is either a tree, forward, or cross edge, so, by inspection of Observation 6.1, $f[u] > f[w]$.

This gives us the following linear-time algorithm. Run DFS on G . If there is a back edge, then G has a cycle. Otherwise topologically sort G by reordering the vertices by reverse finishing times. (E.g. push each u on the front of a globally maintained list when $f[u]$ is set in Line 3.4 of $\text{DFS1}(u)$.) After reordering, we have $f[v_1] > f[v_2] > \dots > f[v_n]$. By Observation 6.2(ii) each edge (v_i, v_j) has $f[v_i] > f[v_j]$, so has $i < j$. That is, the graph is now topologically ordered.

Python code: [graph_traversal/depth_first_search.py](#) (Section A.4).

6.3.6 Strongly connected components in directed graphs

What is the meaning of “connected component” in a *directed* graph? The situation is more complicated than in undirected graphs, because it can be that there is a path from u to w , but no path from w to u . Two vertices u and w are said to be *strongly connected* if there is both a path from u to w and a path from w to u . This relation is reflexive, transitive and symmetric, so it partitions the vertex set into equivalence classes $C_1, C_2, \dots, C_k$, each called a *strongly connected component*. For each pair of vertices $u, w \in V$, the pair is strongly connected if and only if they are in the same component C_i . There can be edges from one strongly connected component to another, but any cycle in G must be wholly contained within a single component.

Given a source vertex s , an easy way to compute the strongly connected component containing s is as follows. Run DFS (or BFS) to find all vertices that s has a path to. Then run DFS (or BFS) on the graph *with all edges reversed* to find all vertices that have a path to s . (These will be the vertices that s has a path to in the reversed graph.) Then the strongly connected component containing s contains those vertices v in both sets—such that s has a path to v , and v has a path to s . Thus, one can identify the strongly connected component containing s in linear time.

The following DFS-based algorithm finds *all* the strongly connected components in linear time:

$\text{SCC}(G = (V, E))$ G is any directed graph

1. Run DFS on G . Record the finish time $f[u]$ of each vertex u .
2. Obtain digraph $G' = (V, E')$ from G by reversing each edge ($E' = \{(w, u) : (u, w) \in E\}$).
3. Run DFS on G' , ordering V in the outer loop (Line 4) by decreasing $f[u]$ (from the first DFS).
4. Let $T_1, T_2, \dots, T_k$ be the DFS trees in the resulting DFS forest.
5. Return $(C_1, C_2, \dots, C_k)$, where C_i contains the vertices in tree T_i .

Lemma 6.5. *The above algorithm returns the strongly connected components of G .*

For a proof, see [CLRS Chapter 22.5](#).

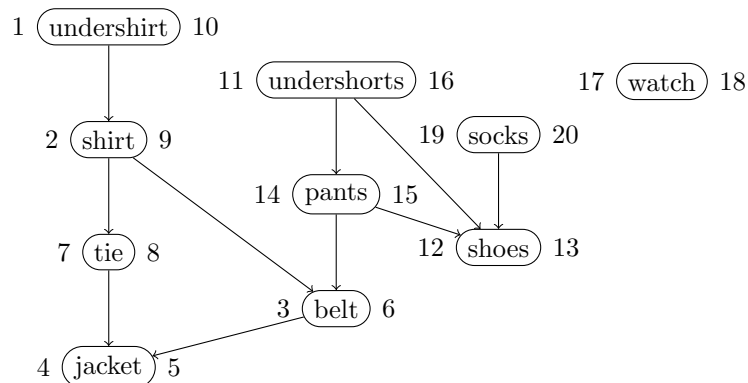
6.3.7 Exercises

External exercises with solutions

- [KT Chapter 3](#): Solved Exercises 1 (topological ordering) and 2

Exercises without solutions

Exercise 6.1. (*DFS*)



The drawing above shows a directed acyclic graph (DAG) $G = (V, E)$, on which a depth-first search (DFS) has been run. Each node is labeled with its discovery time and its finish time during the DFS. Show the DFS forest for this execution of DFS. Draw all the nodes. Draw only the tree edges, not the non-tree edges. How many forward edges are there? How many cross edges are there? How many back edges are there?

Exercise 6.2. (*Cut vertices in undirected graphs.*) Fix any connected, undirected graph $G = (V, E)$. A vertex $v \in V$ is a *cut vertex* if there are vertices $u, w \in V \setminus \{v\}$ such that every u -to- w path in G goes through v . In other words, removing v from G would increase the number of connected components.

Consider any execution of DFS on G . As observed in Section 6.3, because G is a connected, undirected graph, the DFS forest will consist of just one rooted tree, say T , and each non-tree edge is both a back edge and a forward edge. There are no cross edges. In T , say that any tree edge (v, v') is *bypassed* by a back edge (u, w) if u is a descendant of both v and v' , and w is a proper ancestor of both v and v' . (In T , by definition, each vertex is its own descendant and ancestor. A *proper* ancestor of v is an ancestor other than v .)

Theorem 6.1. *Let v be a vertex in V other than the root of the DFS tree T . Then v is a cut vertex if and only if, for some child v' of v in T , no back edge bypasses (v, v') .*

Theorem 6.2. *The root of the DFS tree T is a cut vertex if and only if the root has at least two children in T .*

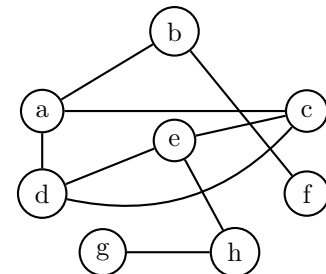
Give a long-form proof of Theorem 6.1.

(For context, these theorems are the basis of the following linear-time algorithm for finding the cut vertices in G . Run DFS on G . For each $v \in V$, define $\ell[v]$ (the *low number* of v) to be the minimum, over all back edges (u, w) out of any descendant u of v , of $d[w]$. Define $\ell[v] = \infty$ if there are no such back edges. When finishing the visit to vertex v , the low number of v can be computed via the recurrence

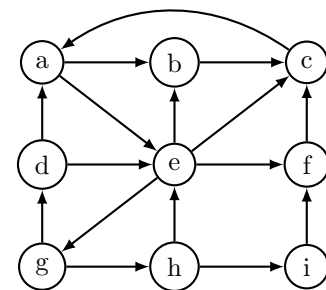
$$\ell[v] = \min \left(\min\{d[w] : (v, w) \text{ is a back edge}\}, \min\{\ell[v'] : v' \text{ is a child of } v\} \right).$$

By Theorem 6.1, (unless v is the root) v is a cut vertex iff it has a child v' such that (v, v') is not bypassed by some back edge (u, w) . This happens iff v has a child v' with $\ell[v'] \geq d[v]$. So, by computing and checking the low numbers as described above, and checking the number of children of the root, one can find all cut vertices in G in linear time.)

Exercise 6.3. (*BFS*) Simulate BFS on the graph to the right, starting from the node a . In your answer, just draw the resulting BFS tree and label each node with its distance from a .



Exercise 6.4. (*DFS*) Simulate DFS (as described in this note) on the graph to the right. When there are multiple choices for the next node to visit, break ties alphabetically (choose a before b , choose b before c , etc.).



- Draw the DFS forest that results.
- Label every edge as either tree, forward, backward, or cross.
- Label each vertex with its post-order number.
- Is this graph a directed acyclic graph (DAG)?
- If so, give the topological ordering that the DFS-based algorithm (from class and this note) would output after doing the DFS that you simulated above.
If not, how many strongly connected components does the graph have? List them all.

Exercise 6.5. (*Street Sweeper*) You have been hired to clean up the streets of Boston using your street sweeper. In order to do the job for the lowest cost, you want to devise a way to sweep each street once in each direction. (That is, for each street, you go up the street once and down the street once to sweep each side of the road.) All roads are two way. To find a good route, you formally define the following problem

STREET SWEEPER

input: a connected, undirected graph $G = (V, E)$

output: a (not necessarily simple) cycle C that traverses each edge (u, w) once in each direction¹

- Give a correct output for the input which is the graph from Exercise 6.3.
- Give a definition for a correct algorithm for STREET SWEEPER. For this part, explain clearly in a few English sentences exactly what your algorithm does.
- Define your algorithm in pseudocode.
- Prove your algorithm correct (i.e., gives the correct output for every input). You may cite (without reproving them) facts about DFS that we’ve observed in these notes. We recommend that you use the long-form proof format, as it will help you catch gaps and errors, and make your proof easier to check.
- Give the best bound you can on the worst-case running time of your algorithm in terms of $n = |V|$ and $m = |E|$. Explain carefully what your reasoning is as to why the bound you give holds for all inputs. You may use (without reproving them) any facts about DFS that we’ve observed in these notes. Your algorithm should run in linear time.
- Implement your algorithm and test your implementation on Hackerrank.com, at <https://www.hackerrank.com/neaileyoung-algs101-street-sweeper>.

Exercise 6.6. (*DFS Edge Count*) Fix an undirected graph $G = (V, E)$, run DFS on G , and let F be the resulting DFS forest. For each vertex $u \in V$, let E_u be the set of edges in E that have both endpoints among the descendants of u in F (including u itself). Give a linear-time algorithm that computes $|E_u|$ for every vertex $u \in V$. Implement your algorithm and test your implementation on Hackerrank.com, at <https://www.hackerrank.com/neaileyoung-algs101-dfs-edges-and-street-sweeper> (the “DFS Edge Count” challenge, which specifies how the DFS should break ties).

Exercises with solutions

Exercise 6.7. (*implementation*) Implement the algorithm for topological sorting in Section 6.3.5 and test your implementation on Hackerrank.com, at <https://www.hackerrank.com/neaileyoung-algs101-dag-paths-and-topological-sort> (the “Topologically sort or find a cycle” challenge).

Solution. Python code: [graph_traversal/depth_first_search.py](#) (Section A.4).

External exercises without solutions

- [JE](#) Chapter 5: Exercises 1–28
- [CLRS](#) Chapter 22: exercises in Sections 22.1 (background), 22.2 (BFS), 22.3 (DFS), 22.4 (topological sort), 22.5, (strongly connected components), and Problems 22–1 through 22–4 (Page 621)
- [DPV](#): exercises in Chapter 3 (DFS) and Chapter 4 Exercise 4.5
- [KT](#) Chapter 3: Exercises 1–12

6.3.8 External sources on graphs and linear-time graph traversal

- [JE](#) Chapter 5
- [CLRS](#) Chapter 22: Sections 22.1 (background), 22.2 (BFS), 22.3 (DFS), 22.4 (topological sort), 22.5, (strongly connected components)
- [DPV](#) Chapters 3 (DFS) and Sections 4.1–4.2 (BFS)

¹Formally, the cycle C can be specified as a sequence of *directed* edges $(v_1, v_2), (v_2, v_3), \dots, (v_{\ell-1}, v_\ell), (v_\ell, v_1)$. In any cycle, each edge must start with the vertex that the edge before ends with (except the *first* edge must start with the vertex that the the last edge ends with). Here, C should contain each edge $(u, w) \in E$ exactly twice: once as (u, w) and once as (w, u) .

- KT Chapter 3
- MIT lecture video
 - <https://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-006-introduction-to-algorithms-fall-2011/lecture-videos/lecture-14-depth-first-search-dfs-topological-sort/>

6.3.9 Appendix: Proof of correctness for Breadth-First Search

Here is the detailed proof of correctness for BFS. Fix any directed or undirected graph $G = (V, E)$ and source s . Consider executing $\text{BFS}(G = (V, E), s)$. Recall that *level* ℓ is $\{v \in V : \text{dist}(s, v) = \ell\}$, the set of vertices that are at distance ℓ from s . To start we prove some properties of distances.

Lemma 6.2 (levels). *Level 0 contains just the source s . Each level $\ell > 0$ consists of the vertices w such that (a) w is not in any level less than ℓ , and (b) there is an edge $(u, w) \in E$ from some vertex u in level $\ell - 1$. That is, each level $\ell > 0$ is $\{w \in V : \text{dist}(s, w) \geq \ell, (\exists(u, w) \in E) \text{dist}(s, u) = \ell - 1\}$.*

Proof (long form).

1. Level 0 contains just s because s is the only vertex reachable from s by a path of length 0.
2. Consider any level $\ell > 0$ and any vertex $w \in V$.
 - 3.1. Assume that Properties (a) and (b) hold for w .
 - 3.2. By Property (b), there is an edge (u, w) where $\text{dist}(s, u) = \ell - 1$.
 - 3.3. So there is a length- $(\ell - 1)$ path p from s to some vertex u with $(u, w) \in E$.
 - 3.4. Hence, there is a path from s to w of length ℓ . That is, $\text{dist}(s, w) \leq \ell$.
 - 3.5. This and (a) ($\text{dist}(s, w) \geq \ell$) imply $\text{dist}(s, w) = \ell$. That is, w is in level ℓ .
4. By the preceding block, if Properties (a) and (b) hold, then w is in level ℓ .
5. To finish we show the converse.
 - 6.1. Assume that w is in level ℓ , that is, that $\text{dist}(s, w) = \ell$. Property (a) clearly holds ($\text{dist}(s, w) \geq \ell$).
 - 6.2. Let p be a shortest path from s to w with length ℓ . (Such a path exists because $\text{dist}(s, w) = \ell$.)
 - 6.3. Let $(u, w) \in E$ be the last edge on path p . Let p' be the prefix of p from s to u .
 - 6.4. So p' has length $\ell - 1$, and no path from s to u has length less than $\ell - 1$ (because otherwise that path and (u, w) would form a path from s to w of length less than ℓ).
 - 6.5. So $\text{dist}(s, u) = \ell - 1$. That is, Property (b) also holds.
7. By the preceding block, if w is in level ℓ , then Properties (a) and (b) hold.
8. By this and Step 4, w is in level ℓ if and only if Properties (a) and (b) hold. □

Lemma 6.3 (correctness). *$\text{BFS}(G = (V, E), s)$ returns an array d with $d[v] = \text{dist}(s, v)$ for all $v \in V$.*

Proof (long form).

1. Consider any execution of BFS on some input $(G = (V, E), s)$.
2. We will show, for each level $\ell \geq 0$, by induction on ℓ , that the following three properties hold at some time t_ℓ during the execution of the while loop:
 - A_ℓ — The queue Q contains exactly those vertices in level ℓ .
 - B_ℓ — The set of vertices v for which $d[v]$ has been set is exactly the union of levels $0, 1, \dots, \ell$.
 - C_ℓ — Each array entry $d[v]$ that has been set has been set correctly, to $\text{dist}(s, v)$.
3. For the base case, level $\ell = 0$ is $\{s\}$, so A_0 , B_0 , and C_0 hold just before the first iteration.
 - 4.1. Assume inductively that, for some level $\ell \geq 1$, $A_{\ell-1}$, $B_{\ell-1}$, and $C_{\ell-1}$ hold at some time $t_{\ell-1}$.
 - 4.2. At time $t_{\ell-1}$, by property $A_{\ell-1}$, the queue contains exactly the vertices in level $\ell - 1$.
 - 4.3. And (by $B_{\ell-1}$ and $C_{\ell-1}$) each vertex u in level $\ell - 1$ has $d[u]$ set correctly to $\ell - 1$.
 - 4.4. Let t_ℓ be the end-time of the last iteration in which any level- $(\ell - 1)$ vertex is popped from Q , so that, between time $t_{\ell-1}$ and time t_ℓ , the alg. pops every vertex u in level $\ell - 1$ from Q (and, because Q is FIFO, pops no other vertex before time t_ℓ , so that at time t_ℓ the queue contains exactly the vertices added between times $t_{\ell-1}$ and t_ℓ).

- 4.5. For each such vertex u that it pops from Q , it adds each vertex w with an edge (u, w) such that $d[w]$ is not already set, and it sets $d[w]$ to $d[u] + 1 = (\ell - 1) + 1 = \ell$.
- 4.6. By $B_{\ell-1}$, $d[w]$ was set at time $t_{\ell-1}$ if and only if w is in level $\ell - 1$ or less.
- 4.7. So, between time $t_{\ell-1}$ and t_ℓ , the algorithm adds to Q those vertices w with an edge (u, w) such that u is a vertex in level $\ell - 1$ and w is not in a level less than ℓ .
- 4.8. By Lemma 6.2 these added vertices are just those in level ℓ .
- 4.9. So at time t_ℓ Property A_ℓ and B_ℓ hold. And $d[w] = \ell$ for each added w , so Property C_ℓ holds.
5. Inductively, for every level $\ell \geq 0$, Properties A_ℓ , B_ℓ , and C_ℓ hold at some time t_ℓ .
6. B_ℓ and C_ℓ hold for every level ℓ at some time during the while loop, and each $d[v]$ is only set once, for every vertex v with finite $\text{dist}(s, v)$, the while loop sets $d[v] = \text{dist}(s, v)$.
7. Line 4 of the algorithm sets $d[v] = \text{dist}(s, v) = \infty$ for each vertex v that is not in any level.
8. So, at termination, $d[v] = \text{dist}(s, v)$ for every $v \in V$. □

Lecture Note 7

Dynamic Programming

[↑ CONTENTS](#)

Contents

7.1	Pascal's rule for computing n choose k (with long-form proof)	115
7.2	Paths in DAGs	117
7.2.1	Counting Paths in a Grid	117
7.2.1.1	Memoizing	118
7.2.1.2	Iterating over the subproblems	118
7.2.2	Shortest Path in a Grid	119
7.2.3	Shortest Path in a DAG	120
7.2.4	Longest Path in a DAG	121
7.3	Making Change	122
7.3.1	Reducing to Shortest Path in a DAG	122
7.4	Four steps for presenting a dynamic-programming algorithm	124
7.5	Subset Sum	125
7.6	Knapsack (with long-form proof)	126
7.7	Weighted Activity Selection	128
7.8	Longest Common Subsequence (with long-form proof)	131
7.9	Edit Distance and Sequence Alignment	133
7.10	Longest Palindromic Subsequence (with long-form proof)	135
7.11	Paragraph Layout	137
7.12	Minimum-Weight Polygon Triangulation	138
7.13	Optimal Binary Search Tree	139
7.14	Single-Source Shortest Paths with Negative Weights	140
7.15	All Pairs Shortest Paths / Floyd-Warshall Alg. (w/ l. f. proof)	142
7.16	External sources on dynamic programming	144

7.1 Pascal's rule for computing n choose k (with long-form proof)

Given non-negative integers n and k , the number of size- k subsets of $\{1, 2, \dots, n\}$ is called n choose k , and denoted $\binom{n}{k}$. Consider the problem of computing it:

Computing n choose k

input: non-negative integers n and k

output: $\binom{n}{k}$.

To design an algorithm, first we'll prove Pascal's rule:

Lemma 7.1 (Pascal's rule).
$$\binom{n}{k} = \begin{cases} 1 & \text{if } k = 0, \\ 0 & \text{if } k > 0 \text{ and } n = 0, \\ \binom{n-1}{k} + \binom{n-1}{k-1} & \text{otherwise.} \end{cases}$$

Proof (long form).

1. For $m, \ell \geq 0$, let $\mathcal{C}(m, \ell)$ be the collection of all size- ℓ subsets of $\{1, 2, \dots, m\}$, so $\binom{m}{\ell} = |\mathcal{C}(m, \ell)|$.
2. If $k = 0$, the empty set is the only size- k subset of $\{1, 2, \dots, n\}$, so the rule holds in this case.
3. If $k > 0$ and $n = 0$, there is no size- k subset of $\{1, 2, \dots, n\}$, so the rule holds in this case.
4. So assume $n, k > 0$ and consider $\mathcal{C}(n, k)$.
5. Each set in $\mathcal{C}(n, k)$ either contains n or it doesn't.
6. The sets in $\mathcal{C}(n, k)$ that *don't* contain n are just the sets in $\mathcal{C}(n-1, k)$.
7. By definition of $\mathcal{C}$, there are $\binom{n-1}{k}$ of these.
8. So, the number of sets in $\mathcal{C}(n, k)$ that don't contain n is $\binom{n-1}{k}$.
9. To finish, we show that the number of sets in $\mathcal{C}(n, k)$ that *do* contain n is $\binom{n-1}{k-1}$.
10. For each set S in $\mathcal{C}(n, k)$ that does contain n , let $\phi(S) = S \setminus \{n\}$ be the set with n removed.
11. Then ϕ is a bijection (a one-to-one mapping) between the sets in $\mathcal{C}(n, k)$ that contain n and the sets in $\mathcal{C}(n-1, k-1)$. (That is, $\phi(S)$ is in $\mathcal{C}(n-1, k-1)$, and, for each $S' \in \mathcal{C}(n-1, k-1)$, there is exactly one set $S \in \mathcal{C}(n, k)$ that contains n such that $\phi(S) = S'$ —namely $S = S' \cup \{n\}$.)
12. Because there is such a bijection, the number of sets in $\mathcal{C}(n, k)$ that contain n must equal the number of sets in $\mathcal{C}(n-1, k-1)$. By definition this number is $\binom{n-1}{k-1}$.
13. By this and Step 8, the size of $\mathcal{C}(n, k)$ is $\binom{n-1}{k} + \binom{n-1}{k-1}$. □

The proof of Pascal's rule illustrates the first main principle of dynamic programming. Namely, to reason about a collection of objects (in this case the size- k subsets of $\{1, \dots, n\}$) the proof partitions the collection into a few types—in this case two:

- (A) Those subsets not containing the element n .
- (B) Those subsets containing the element n .

Then it handles each type by noting that, restricted to objects of that particular type, the problem at hand is somehow equivalent to a smaller instance of the original problem.

In the current example, the proof observes that the sets of type (A) are just the size- k subsets of $\{1, \dots, n-1\}$, so counting the type-(A) sets is equivalent to the problem of counting the size- k subsets of $\{1, \dots, n-1\}$. This is a smaller instance of the original problem.

Similarly, the sets of type (B) correspond bijectively (via the deletion of element n) to the size- $(k-1)$ subsets of $\{1, \dots, n-1\}$. So counting the type-(B) sets is equivalent to counting the size- $(k-1)$ subsets of $\{1, \dots, n-1\}$. Again, this is a smaller instance of the original problem.

This understanding of the recursive structure of the objects in question (subsets) yields a recurrence relation (in this case, Pascal's rule). This in turn yields a recursive, divide-and-conquer algorithm for the problem. In this case, for computing $\binom{n}{k}$:

countSubsets(n, k): — $n, k \geq 0$

1. return $\begin{cases} 1 & \text{if } k = 0, \\ 0 & \text{if } k > 0 \text{ and } n = 0, \\ \text{countSubsets}(n-1, k) + \text{countSubsets}(n-1, k-1) & \text{otherwise.} \end{cases}$

Correctness of the algorithm follows by induction and Lemma 7.1.

This seems like a classic *divide-and-conquer* approach. But beware! This implementation is not efficient—indeed, its run time is exponential! Consider its recursion tree, which has branching factor two, and depth between k and n , so somewhere between 2^k and 2^n nodes!

There are two standard work-arounds for this obstacle. We'll discuss them in the next example.

Exercises

Exercise 7.1. (*example with solution*) The proof of Lemma 7.1 uses a bijection ϕ between those sets in $\mathcal{C}(n, k)$ that contain n and the sets in $\mathcal{C}(n-1, k-1)$. Show this bijection for $n = 5$ and $k = 3$. For reference, here is all of $\mathcal{C}(5, 3)$, and its subset $\mathcal{C}(4, 3)$:

$$\mathcal{C}(5, 3) = \left\{ \begin{array}{l} \{1, 2, 3\} \\ \{1, 2, 4\} \\ \{1, 3, 4\} \\ \{2, 3, 4\} \\ \{1, 2, 5\} \\ \{1, 3, 5\} \\ \{1, 4, 5\} \\ \{2, 3, 5\} \\ \{2, 4, 5\} \\ \{3, 4, 5\} \end{array} \right\}$$

Solution.

sets in $\mathcal{C}(5, 3)$ that contain 5	ϕ $\longleftrightarrow$	sets in $\mathcal{C}(4, 2)$
$\{1, 2, 5\}$		$\{1, 2\}$
$\{1, 3, 5\}$		$\{1, 3\}$
$\{1, 4, 5\}$		$\{1, 4\}$
$\{2, 3, 5\}$		$\{2, 3\}$
$\{2, 4, 5\}$		$\{2, 4\}$
$\{3, 4, 5\}$		$\{3, 4\}$

External sources

- https://en.wikipedia.org/wiki/Pascal%27s_rule

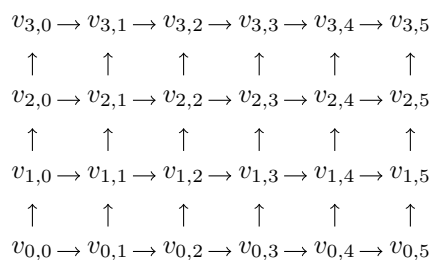
7.2 Paths in DAGs

7.2.1 Counting Paths in a Grid

Here's a toy problem to consider: Counting Paths in a Grid.

input: Non-negative integers i and j .

output: The number of paths from the origin $v_{0,0}$ to the node $v_{i,j}$ at location (i, j) in the 2D grid, as illustrated to the right, assuming each step in the path moves one unit up or one unit to the right.



For example, how many paths are there from $v_{0,0}$ to $v_{3,5}$ in the graph to the right? Let $P(i, j)$ denote the set of paths from $v_{0,0}$ to $v_{i,j}$. We want to count the size of $P(3, 5)$ (denoted $|P(3, 5)|$). The next lemma gives a divide-and-conquer approach:

Lemma 7.2. For $i, j \geq 0$, $|P(i, j)| = \begin{cases} 1 & \text{if } i = 0 \text{ or } j = 0 \\ |P(i, j - 1)| + |P(i - 1, j)| & \text{otherwise} \end{cases}$.

Proof. When $i = 0$ or $j = 0$, the lemma holds because (by inspection of the grid) there is only one path. So assume $i > 0$ and $j > 0$. Partition the set $P(i, j)$ of paths from $v_{0,0}$ to $v_{i,j}$ into two subsets:

- (A) The paths that enter $v_{i,j}$ through $v_{i,j-1}$.
- (B) The paths that enter $v_{i,j}$ through $v_{i-1,j}$.

By inspection of the grid, every path into $v_{i,j}$ enters either through $v_{i,j-1}$ or through $v_{i-1,j}$, so the two subsets (A) and (B) partition $P(i, j)$.

The number of paths entering $v_{i,j}$ through $v_{i-1,j}$ equals the number of paths to $v_{i-1,j}$, because each path to $v_{i-1,j}$, followed by the edge from there to $v_{i,j}$, yields a unique path into $v_{i,j}$. Similarly, the number of paths entering $v_{i,j}$ through $v_{i,j-1}$ equals the number of paths to $v_{i,j-1}$. $\square$

The lemma gives this recursive algorithm for Counting Paths in a grid:

countPaths(i, j) — return the number of paths from $v_{0,0}$ to $v_{i,j}$

1. return $\begin{cases} 1 & \text{if } i = 0 \text{ or } j = 0 \\ \text{countPaths}(i, j - 1) + \text{countPaths}(i - 1, j) & \text{otherwise} \end{cases}$

There is one problem with this algorithm: it takes time exponential in i and j . To see why, consider the recursion tree for **countPaths**(n, n). In the tree, each leaf has depth at least n , and each non-leaf node has two children, so the number of nodes in each level $i \leq n$ is 2^i . So executing **countPaths**(n, n) results in more than 2^n recursive calls altogether!

Two ways to avoid solving subproblems multiple times. In the execution of **countPaths**(i, j), exponentially many recursive calls can arise. But each recursive call that arises is of the form **countPaths**(a, b) where $0 \leq a \leq i$ and $0 \leq b \leq j$. And there are only $(i + 1) \cdot (j + 1)$ such pairs (a, b) (one for each point lying to the left of and below point (i, j)). So, the number of *different* subproblems solved during the execution is only $\Theta(ij)$.

The reason that the recursive implementation above takes exponential time is that *it solves each distinct subproblem many times*. You can see this by trying it out on the example, or diagramming the recursion tree. This redundancy is unnecessary. *It's enough to solve each distinct subproblem just once*. There are two standard ways to do this.

7.2.1.1 Memoizing

Memoizing caches the value returned by each call, to avoid computing it again:

<code>countPaths(i, j):</code>	— return the number of paths from $(0, 0)$ to (i, j) in the 2d grid
1. maintain global “cache” array $C[]$, initially empty	
2. if $C[i, j]$ has not previously been set:	(store the value in the cache)
3. $\text{set } C[i, j] = \begin{cases} 1 & \text{if } i = 0 \text{ or } j = 0 \\ \text{countPaths}(i, j - 1) + \text{countPaths}(i - 1, j) & \text{otherwise} \end{cases}$	
4. return $C[i, j]$	

Consider executing `countPaths( $i, j$ )` with this modification. Now, in the recursion tree, there will be just one non-leaf node for each point (a, b) with $0 < a \leq i$ and $0 < b \leq j$, representing the first time `countPaths( $a, b$ )` is called. Subsequent calls with identical parameter values (a, b) will return the cached value, without recursing. So the total number of non-leaf nodes is $\Theta(ij)$. And the number of leaf nodes is at most twice the number of non-leaf nodes (since each non-leaf node has two children). So the total number of nodes is also $\Theta(ij)$. The work associated with each node is constant, so the total time is $\Theta(ij)$, instead of exponential in i and j .

7.2.1.2 Iterating over the subproblems

The second standard fix is to avoid recursion altogether, by simply iterating over the distinct subproblems:

<code>countPaths(i, j):</code>	— return the number of paths from $(0, 0)$ to (i, j) in the 2d grid
1. for $a = 0, 1, 2, \dots, i$:	
2.1. for $b = 0, 1, 2, \dots, j$:	
2.2. $N[a, b] = \begin{cases} 1 & \text{if } a = 0 \text{ or } b = 0 \\ N[a, b - 1] + N[a - 1, b] & \text{otherwise} \end{cases}$	
3. return $N[i, j]$	

With this approach, *the iteration order matters!!!* Each $N[a, b]$ must be computed *after* $N[a, b - 1]$ and $N[a - 1, b]$. This requirement is easy to satisfy for this problem, but is more complicated to satisfy for some others.

With either approach, correctness of the resulting algorithm follows by induction on the problem size and Lemma 7.2. This implementation requires (by inspection) $\Theta(ij)$ additions, but the values involved can be roughly as large as 2^{i+j} , so large-integer arithmetic on $(i + j)$ -bit integers is required. Then each addition takes time $O(i + j)$, yielding the following theorem:

Theorem 7.1. *Given integers $i, j \geq 0$, we can compute, in $O((i + j)ij)$ time, the number of paths in the 2d grid graph (defined earlier) from the origin to $v_{i,j}$.*

Python code: [dynamic_programming/dynamic_programming_paths.py](#) (Section A.5).

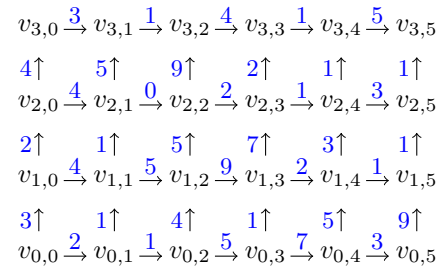
7.2.2 Shortest Path in a Grid

The same reasoning easily extends to solve other problems on paths in a grid, such as finding a shortest path.

input: Non-negative integers i and j , and a weight for each edge in the 2d grid graph.

output: The minimum, over all paths from $v_{0,0}$ to $v_{i,j}$, of the total weight of edges on the path.

To the right is the graph with some edge weights added (in blue).



Recall that $P(i, j)$ denotes the set of paths from $v_{0,0}$ to $v_{i,j}$. Our goal now is to compute the minimum, over all paths in this set, of the total weight of edges in the path.

Define $M(i, j)$ to be this minimum weight. Let $w(v_{a,b}, v_{c,d})$ denote the weight of the edge from $v_{a,b}$ to $v_{c,d}$ (if the edge is present, that is, if $(c, d) = (a + 1, b)$ or $(c, d) = (a, b + 1)$).

$$\textbf{Lemma 7.3.} \text{ For } i, j \geq 0, M(i, j) = \begin{cases} 0 & \text{if } i = 0 \text{ and } j = 0, \\ M(i - 1, j) + w(v_{i-1,j}, v_{i,j}) & \text{if } i > 0 \text{ and } j = 0, \\ M(i, j - 1) + w(v_{i,j-1}, v_{i,j}) & \text{if } i = 0 \text{ and } j > 0, \\ \min \left(\begin{array}{l} M(i - 1, j) + w(v_{i-1,j}, v_{i,j}), \\ M(i, j - 1) + w(v_{i,j-1}, v_{i,j}) \end{array} \right) & \text{if } i > 0 \text{ and } j > 0. \end{cases}$$

Proof sketch. When $i = 0$ and $j = 0$, the lemma holds because the only path is the empty path.

Next we give the proof just for the case $i > 0$ and $j > 0$. (The remaining cases are easier.) As before, partition the set $P(i, j)$ of paths from $v_{0,0}$ to $v_{i,j}$ into two types:

- (A) The paths that enter $v_{i,j}$ from $v_{i-1,j}$.
- (B) The paths that enter $v_{i,j}$ from $v_{i,j-1}$.

By inspection of the grid, every path into $v_{i,j}$ enters either through $v_{i,j-1}$ or through $v_{i-1,j}$, so the two subsets (A) and (B) partition $P(i, j)$. It follows that the minimum weight of any path in $P(i, j)$ is the minimum weight of any path of type (A), or the minimum weight of any path of type (B), whichever is smaller.

The paths of type (A) can be obtained by taking any path to $v_{i-1,j}$ and adding the edge from $v_{i-1,j}$ to $v_{i,j}$. So the minimum weight of any path of type (A) is the minimum weight of any path into $v_{i-1,j}$, plus the weight of that edge. That is, the minimum weight of any type-(A) path is $M(i - 1, j) + w(v_{i-1,j}, v_{i,j})$.

By similar reasoning, the minimum weight of any type-(B) path is $M(i, j - 1) + w(v_{i,j-1}, v_{i,j})$.

It follows (when $i > 0$ and $j > 0$), that $M(i, j) = \min \left(\begin{array}{l} M(i - 1, j) + w(v_{i-1,j}, v_{i,j}), \\ M(i, j - 1) + w(v_{i,j-1}, v_{i,j}) \end{array} \right)$. □

Here is the resulting algorithm (the iterative implementation):

shortestPath (i, j, w):		— return the minimum weight of any path from $v_{0,0}$ to $v_{i,j}$
1. for $a = 0, 1, 2, \dots, i$:		
2.1. for $b = 0, 1, 2, \dots, j$:		
2.2. $M[a, b] = \begin{cases} 0 & \text{if } a = 0 \text{ and } b = 0, \\ M[a - 1, b] + w(v_{a-1,b}, v_{a,b}) & \text{if } a > 0 \text{ and } b = 0, \\ M[a, b - 1] + w(v_{a,b-1}, v_{a,b}) & \text{if } a = 0 \text{ and } b > 0, \\ \min \left\{ \begin{array}{l} M[a - 1, b] + w(v_{a-1,b}, v_{a,b}) \\ M[a, b - 1] + w(v_{a,b-1}, v_{a,b}) \end{array} \right\} & \text{if } a > 0 \text{ and } b > 0. \end{cases}$		
3. return $M[i, j]$		

Correctness of the algorithm follows from the lemma and induction on the problem size.

By inspection the algorithm runs in time $\Theta(ij)$. This proves the following theorem:

Theorem 7.2. *Given integers $i, j \geq 0$ and edge weights for the 2d grid graph (defined earlier), we can compute, in $O(ij)$ time, the minimum weight of any path from the origin to $v_{i,j}$.*

7.2.3 Shortest Path in a DAG

The previous reasoning for grid graphs extends easily to any directed acyclic graph (DAG). To illustrate this, consider finding shortest paths in an edge-weighted DAG:

input: DAG $G = (V, E)$ with edge weights; source vertex s , destination vertex w

output: The shortest-path distance from s to w

Fix any such input. We start with the following recurrence relation:

Lemma 7.4. *For any vertex $w \in V$:*

$$\text{dist}(s, w) = \begin{cases} 0 & \text{if } w = s \\ \min \{ \text{dist}(s, u) + \text{wt}(u, w) : (u, w) \in E \} & \text{otherwise.} \end{cases}$$

Proof sketch. In the case that $w = s$, the lemma holds because the only path is the empty path.

Consider the case $w \neq s$. The shortest path from s to w is the shortest, over all possible edges (u, w) into w , of the shortest among the paths ending in that edge (u, w) . Among the paths from s to w ending in a given edge (u, w) , a shortest one can be obtained by taking any shortest path from s to u , and appending edge (u, w) . Hence the shortest path from s to w ending in (u, w) has weight $\text{dist}(s, u) + \text{wt}(u, w)$. $\square$

This gives the following recursive algorithm:

$\text{shortestPath}(G = (V, E), s, w):$	<i>return the shortest-path distance from s to w in DAG G</i>
1. return $\begin{cases} 0 & \text{if } w = s \\ \min \{ \text{shortestPath}(G, s, u) + \text{wt}(u, w) : (u, w) \in E \} & \text{otherwise.} \end{cases}$	

If w has no in-neighbors, so the set is empty, the minimum has value ∞ .

Correctness follows from Lemma 7.4 and induction. (Show by induction on the recursion depth that $\text{shortestPath}(G, s, w) = \text{dist}(s, w)$ for all $w \in V$.) We memoize the algorithm. Then the recursion tree will have at most one non-leaf node for each vertex $v \in V$, and the work associated with that node will be proportional to the in-degree of v (the number of edges into v), so the total time will be $\Theta(n + m)$.

If we want instead an iterative algorithm, we have to explicitly compute an order in which to evaluate the subproblems. We can only evaluate a given subproblem $\text{shortestPath}(G, s, w)$ *after* the subproblem for each in-neighbor u of w is done. This is exactly what topological ordering is for! The algorithm first computes a topological ordering of the vertices (in linear time using DFS, as described in [Lecture Note 6](#)), then considers the vertices in that order.

$\text{shortestPath}(G = (V, E), s, w):$	<i>return the shortest-path distance from s to w in DAG G</i>
1. let $v_1, v_2, \dots, v_n$ be a topological ordering of the vertices (computed using DFS; see Lecture Note 6)	
2. for $j = 1, 2, \dots, n$:	
3. set $d[v_j] = \begin{cases} 0 & \text{if } v_j = s \\ \min \{ d[v_i] + \text{wt}(v_i, v_j) : (v_i, v_j) \in E \} & \text{otherwise.} \end{cases}$	
4. return $d[w]$	

In Step 3 if v_j has no in-neighbors, so the set is empty, the minimum has value ∞ .

Correctness follows from Lemma 7.4 and induction. (Show by induction on j that that $d[v_j] = \text{dist}(s, v_j)$ for all $j \in \{1, 2, \dots, n\}$.) The time to evaluate the right-hand side in Step 3 is $\Theta(1 + \text{in-degree}(v_j))$. Summing over all the vertices, the total time is $\Theta(n + m)$. We conclude with a theorem that summarizes what we've figured out:

Theorem 7.3. *There is a linear-time algorithm for Shortest Path in a DAG.*

Python code: [dynamic_programming/dynamic_programming_paths.py](#) (Section A.5).

Exercise 7.2. (*implementation*) Implement the algorithm and test your implementation on Hacker-rank.com, at <https://www.hackerrank.com/neaileyong-algs101-dag-paths-and-topological-sort> (the “Shortest-path distances in a DAG” challenge; there, edge weights can be negative).

Solution. Python code: [dynamic_programming/dynamic_programming_paths.py](#) (Section A.5).

7.2.4 Longest Path in a DAG

The same reasoning yields a linear-time algorithm for Longest Path in a DAG (computing the *maximum* weight of any path from s to w). The algorithm and its derivation are exactly the same as for Shortest Path in a DAG, but each “min” is replaced by a “max”. We leave the details as an exercise.

Python code: [dynamic_programming/dynamic_programming_paths.py](#) (Section A.5).

7.3 Making Change

input: A pair $I = (D, T)$ where $D = \{d_1, d_2, \dots, d_k\}$ is a set of *denominations*, and T is a target (all non-negative integers, with each $d_i \geq 1$ and $1 \in D$).

output: The minimum size of any sequence $C = (c_1, \dots, c_\ell)$ s.t. $c_i \in D$ for all i and $\sum_i c_i = T$.

Lecture Note 5 described a greedy algorithm, and proved that it worked for denominations $D = \{1, 5, 10, 25\}$. However, that algorithm doesn't work for all denominations. For example, it fails for $D = \{1, 3, 4\}$ and target $T = 6$. Next we give a dynamic-programming algorithm that works in all cases. Fix any input instance $(D = \{d_1, d_2, \dots, d_k\}, T)$.

Our first goal is to come up with some divide-and-conquer (recursive or inductive) understanding of the possible ways of making change for T .

To this end, partition the ways of making change for T into up to k types, one for each denomination d_i that could be used first, which can be any denomination d_i that does not exceed T . Among ways to make change for T by first using a coin of denomination d_i , the best uses that coin d_i , and is followed by the best possible way of making change for $T - d_i$. The best way of making change overall is the best among these (up to) k types. So, we could find the best way overall if we knew the best way to make change for $T - d_i$, for each $d_i \leq T$.

Applying this reasoning recursively, for $t \in \{0, 1, \dots, T\}$, define $M(t)$ to be the minimum number of coins needed to make change for t . Then $1 + M(t - d_i)$ is the minimum number of coins needed to make change for t if the first coin is d_i . This reasoning gives the following recurrence relation:

Lemma 7.5. $M(0) = 0$. For $t \geq 1$, $M(t) = \min\{1 + M(t - d_i) : d_i \in D, d_i \leq t\}$.

This recurrence relation naturally gives a recursive algorithm, which we would memoize for efficiency. Alternatively it gives the following iterative algorithm:

```
makeChange( $D = \{d_1, d_2, \dots, d_k\}, T$ ):
1. For  $t = 0, 1, \dots, T$ :
2.    $m[t] = \begin{cases} 0 & \text{if } t = 0 \\ \min\{1 + m[t - d_i] : d_i \in D, d_i \leq t\} & \text{otherwise.} \end{cases}$ 
3. return  $m[T]$ 
```

Correctness follows by the lemma and induction on t , showing that $m[t]$ as computed by the algorithm equals $M(t)$ as defined earlier. Each of the $T + 1$ subproblems ($m[t]$ for each t) takes time $O(k)$ to evaluate, so the total time is $O(kT)$.

Let's summarize what we've figured out in a theorem:

Theorem 7.4. *There is an $O(kT)$ -time algorithm for Making Change.*

Python code: [dynamic_programming/making_change.py](#) (Section A.5).

7.3.1 Reducing to Shortest Path in a DAG

Many dynamic-programming problems reduce to finding an optimal path in some implicit DAG. For example, we can obtain the above algorithm for Making Change this way. Given an instance (D, T) of Making Change, consider the following instance $(G = (V, E), s, t)$ of Shortest Path in a DAG.

Take $V = \{0, 1, \dots, T\}$. That is, there is a vertex for each subproblem $M(t)$. For each t , add an edge from $t - d$ to t for each $d \in D$ with $d \leq t$. So $E = \{(t - d, t) : t \in V; d \in D; d \leq t\}$. (There is an edge $(t - d, t)$ if subproblem $t - d$ occurs in the right-hand side of the recurrence for subproblem t .) Give each edge weight 1.

Then each path p in G from 0 to T corresponds to a way of making change totaling T , and vice versa. Specifically, if a path p from 0 to T goes through a sequence of vertices such as $0, c_1, c_1 + c_2, \dots, c_1 + c_2 + \dots + c_n = T$, then it corresponds to making change totaling T using coins in $c = (c_1, c_2, \dots, c_n)$. The length of the path p is the number of coins in c . So, the minimum number of coins needed to make change totaling T equals the distance $\text{dist}(0, T)$ from 0 to T in G , and another algorithm for Making Change would work as follows:

`makeChange2( $D, T$ )` (return the min. num. of coins needed to make change for T with denominations in D)

1. construct the weighted DAG G described above for (D, T)
2. return `shortestPath( $G, 0, T$ )`

This algorithm is essentially equivalent to the previous dynamic-programming algorithm for Making Change.

Python code: [dynamic_programming/dynamic_programming_paths.py](#) (Section A.5).

How could you use this idea to solve the following problem? *Given (D, T) , compute the number of ways to make change totaling T using coins with denominations in D .* (There is an ambiguity here: do we consider two ways of making change different if they differ only in the order of their coins? Is $c = (1, 1, 3)$ different than $c' = (3, 1, 1)$? First answer the question assuming that order matters (so these c and c' are different). Then solve it assuming order doesn't matter. Hint: restrict to sequences c such that, say $c_i \geq c_{i+1}$ for each i .)

7.4 Four steps for presenting a dynamic-programming algorithm

In the rest of these notes, we use the following four steps to present each dynamic-programming algorithm:

- 1. Define the subproblems.** Define the subproblems to be solved for a given instance, and state how the final answer is computed from these subproblems. For example, to present the dynamic-programming algorithm for Making Change, the first step would be as follows:

1. Given the instance (D, T) of Making Change, define the following subproblems. For each $t \in \{0, 1, \dots, T\}$, define $M(t)$ to be the minimum number of coins needed to make change (using denominations in D) of total value t . The solution to the problem is $M(T)$.

This tells the reader exactly what subproblems the algorithm will solve, and also how (once they are solved) the algorithm will compute the output for the given instance.

- 2. State the recurrence relation.** The recurrence relation is a divide-and-conquer equation that can be used to compute the value for each subproblem from the values of some smaller subproblems. The left-hand side of the equation is the value of the subproblem. The right-hand side is an expression built from the values of smaller subproblems. For Making Change, the recurrence relation is in Lemma 7.5:

$$2. \ M(t) = \begin{cases} 0 & \text{if } t = 0 \\ \min\{1 + M(t - d_i) : d_i \in D, d_i \leq t\} & \text{otherwise.} \end{cases}$$

The recurrence relation does not *define* $M(t)$! Rather, it is an identity that holds for $M(t)$ as M is defined in the first step. (To prove correctness of the algorithm, we'll have to prove that the recurrence relation holds.)

The first and second step together determine the algorithm: the algorithm iterate (or recurses) over the set of subproblems to be solved. It solves each subproblem using the recurrence relation, evaluating the right-hand side to determine the value of the left-hand side.

For readers who are familiar with dynamic programming, the first two steps are enough to implicitly define the algorithm, so it is usually unnecessary to give any further definition.

- 3. Prove correctness.** The third step is to prove that (or explain why) the recurrence relation holds. As long as it holds, correctness of the algorithm (iterative, or recursive and memoized) will follow by induction.
- 4. Bound the run time.** Analyze the runtime of the algorithm. This is the sum, over all of the subproblems, of the time to evaluate the right-hand side of the recurrence relation (assuming the smaller subproblems are already solved). For Making Change, this would be enough:

4. There are $T + 1$ subproblems (one for each $t \in \{0, 1, \dots, T\}$; subproblem t is to compute $M(t)$). For each, the time to evaluate the right-hand side of the recurrence for $M(t)$ is $O(k)$ (the number of denominations). So the total time for the algorithm is $O(Tk)$.

7.5 Subset Sum

input: A sequence $x = (x_1, x_2, \dots, x_n)$ of non-negative integers, and an integer target T .

output: True if some subsequence of x sums to T , otherwise False.

(That is, True if there is a subset S of $\{1, 2, \dots, n\}$ such that $\sum_{i \in S} x_i = T$.)

The problem differs from Making Change in that each integer can be used at most once. In what follows, for any subset $S \subseteq \{1, 2, \dots, n\}$, we say the subset *yields* t if the corresponding subsequence of x sums to t , that is, if $\sum_{i \in S} x_i = t$. The underlying collection of objects we are working with is the collection of subsets of $\{1, 2, \dots, n\}$. For our divide-and-conquer approach, we partition this collection into two types of subsets:

- (A) Subsets that don't contain n . That is, subsets of $\{1, 2, \dots, n-1\}$.
- (B) Subsets that do contain n . That is, subsets of the form $S = S' \cup \{n\}$ where $S' \subseteq \{1, 2, \dots, n-1\}$.

Using this, we reduce our problem to two smaller problems:

- (i) There is a type-A subset yielding T if and only if some subset of $\{1, \dots, n-1\}$ yields T .
- (ii) There is a type-B subset yielding T if and only if some subset of $\{1, \dots, n-1\}$ yields $T - x_n$.

Applying this reasoning recursively gives us the algorithm. Here it is, using the four steps:

subproblems and recurrence relation. Fix any input $x = (x_1, \dots, x_n)$ with target T . For each $i \in \{0, 1, \dots, n\}$ and each target $t \in \{0, 1, \dots, T\}$, define

$$S(i, t) = \begin{cases} \text{True} & \text{if there is a subset of } \{1, 2, \dots, i\} \text{ that yields } t, \\ \text{False} & \text{otherwise.} \end{cases}$$

The final answer is given by $S(n, T)$. The recurrence relation is

$$S(i, t) = \begin{cases} \text{True} & \text{if } t = 0 \text{ and } i = 0, \\ \text{False} & \text{if } t > 0 \text{ and } i = 0, \\ S(i-1, t) \text{ or } S(i-1, t-x_i) & \text{if } i > 0 \text{ and } x_i \leq t, \\ S(i-1, t) & \text{if } i > 0 \text{ and } x_i > t. \end{cases}$$

correctness. The base cases of the recurrence relation are true by inspection. The rest holds by the reasoning above. That is, by definition $S(i, t)$ holds if and only if there is a subset of $\{1, \dots, i\}$ yielding t . This occurs if and only if (A) there is such a subset that *doesn't* contain i , or (B) there is one that *does* contain i . (A) occurs if and only if there is a subset of $\{1, \dots, i-1\}$ yielding t , that is, if and only if $S(i-1, t)$ holds. (B) occurs if and only if there is a subset of $\{1, \dots, i-1\}$ yielding $t - x_i$, that is, if and only if $x_i \leq t$ and $S(i-1, t - x_i)$ holds.

run time. There are $O(nT)$ subproblems, and for each the right-hand-side of the recurrence can be evaluated in constant time, so the total time for the algorithm (iterative, or recursive and memoized) is $O(nT)$.

Theorem 7.5. *There is an $O(nT)$ -time algorithm for Subset Sum.*

reduction to finding a path in a graph. Consider the graph with a vertex $v(i, t)$ for each subproblem $S(i, t)$, with edges representing the recurrence-relation dependencies as usual. The problem is then equivalent to asking whether this graph has a path from $v(0, 0)$ to $v(n, T)$.

Python code: [dynamic_programming/subset_sum.py](#) (Section A.5).

7.6 Knapsack (with long-form proof)

input: A triple $I = (v, w, L)$ where $v = (v_1, v_2, \dots, v_n)$ is a sequence of non-negative *values*, $w = (w_1, w_2, \dots, w_n)$ is a sequence of positive integer *weights*, and L is a non-negative integer *weight limit*. We call each integer $i \in \{1, 2, \dots, n\}$ an *item* of value v_i and weight w_i .

output: The maximum value of any subset $S \subseteq \{1, 2, \dots, n\}$ having weight at most L . (The value of a subset S is $\sum_{i \in S} v_i$. The weight of S is $\sum_{i \in S} w_i$.) In other words, output

$$\max \left\{ \sum_{i \in S} v_i \mid S \subseteq \{1, 2, \dots, n\}, \sum_{i \in S} w_i \leq L \right\}.$$

Here is the algorithm. Fix any input instance $I = (v, w, L)$.

subproblems and recurrence. For each $i \in \{0, 1, 2, \dots, n\}$ and $\ell \in \{0, 1, 2, \dots, L\}$, define $V(i, \ell)$ to be the maximum value achievable using only items $\{1, 2, \dots, i\}$ and total weight at most ℓ . The given instance I has solution $V(n, L)$. The recurrence relation is

$$V(i, \ell) = \begin{cases} 0 & \text{if } i = 0, \\ \max(V(i-1, \ell), V(i-1, \ell - w_i) + v_i) & \text{if } i > 0 \text{ and } w_i \leq \ell, \\ V(i-1, \ell) & \text{if } i > 0 \text{ and } w_i > \ell. \end{cases}$$

correctness. The intuition is as follows. If $i = 0$, then the only subset is the empty set, of value zero. For $i > 0$, consider the subsets for $V(i, \ell)$ —that is, subsets of $\{1, \dots, i\}$ having total weight at most ℓ . Partition these into two types: those that don't include i , and those that do. Those that don't include i are just the subsets for $V(i-1, \ell)$, so the maximum value of any subset of this type is $V(i-1, \ell)$. Those that do include i (which exist only if $w_i \leq \ell$) are of the form $S \cup \{i\}$ where S is any subset for $V(i-1, \ell - w_i)$ (having total weight at most $\ell - w_i$). The maximum value of any such subset $S \cup \{i\}$ is $V(i-1, \ell - w_i) + v_i$. A long-form proof of correctness is on the next page.

time. There are $O(nL)$ subproblems. For each, the right-hand side of the recurrence can be evaluated in constant time. So the total time to compute $V(n, L)$ is $O(nL)$.

Theorem 7.6. *There is an $O(nL)$ -time algorithm for Knapsack*

reduction to Longest Path in a DAG. Alternatively, we can reduce the problem to an instance of Longest Path in a DAG. The graph has a vertex $v(i, \ell)$ for each subproblem. If $i > 0$ then $v(i, \ell)$ has an incoming edge from $v(i-1, \ell)$ of weight zero, and (if $\ell \geq w_i$) an incoming edge from $v(i-1, \ell - w_i)$ of weight v_i . If $i = 0$ and $\ell > 0$, then give $v(i, \ell)$ an incoming edge of weight zero from $v(0, 0)$. Then the longest path from $v(0, 0)$ to $v(i, \ell)$ corresponds to a maximum-value subset of $\{1, 2, \dots, i\}$ of total weight at most ℓ .

(continued)

Long-form proof of correctness for Knapsack

Here is a long-form proof of correctness for the recurrence relation. For brevity in the proof, for any $S \subseteq \{1, \dots, n\}$, define $v(S) = \sum_{j \in S} v_j$ to be the total value of items in S . Likewise, define $w(S) = \sum_{j \in S} w_j$ to be the total weight of items in S . So, using this notation, $V(i, \ell) = \max \{v(S) : S \subseteq \{1, 2, \dots, i\}, w(S) \leq \ell\}$.

Proof (long form).

1. Consider any $i \in \{0, \dots, n\}$ and $\ell \in \{0, 1, \dots, L\}$.
2. Recall that $V(i, \ell) = \max \{v(S) : S \subseteq \{1, 2, \dots, i\}, w(S) \leq \ell\}$.
 - 3.1. Case 1. In the case $i = 0$, the set S must be empty, so $V(i, \ell) = 0$, so the recurrence holds.
 - 4.1. Case 2. Next consider the case when $i > 0$ and $w_i \leq \ell$.
 - 4.2. Consider the subsets $S \subseteq \{1, 2, \dots, i\}$ such that $w(S) \leq \ell$.
 - 4.3. Partition these into two types: (A) those that don't include i , and (B) those that do.
 - 4.4. $V(i, \ell)$ is the maximum value of any subset of either type.
 - 4.5. The type-A subsets are just the subsets of $\{1, \dots, i-1\}$ with $w(S) \leq \ell$.
 - 4.6. So (by definition of $V(i-1, \ell)$) the maximum value of any type-A subset is $V(i-1, \ell)$.
 - 4.7. The type-B subsets are those of the form $S' \cup \{i\}$ where $S' \subseteq \{1, \dots, i-1\}$ has $w(S') \leq \ell - w_i$.
 - 4.8. The maximum value of such a set $S' \cup \{i\}$ is achieved when S' itself has maximum total value.
 - 4.9. So, the maximum value of any type-B subset $S' \cup \{i\}$ is $V(i-1, \ell - w_i) + v_i$.
(Here we implicitly use that $w_i \leq \ell$, so that $V(i-1, \ell - w_i)$ is well-defined.)
 - 4.10. By Steps 4.4, 4.6 and 4.9, $V(i, \ell) = \max (V(i-1, \ell), V(i-1, \ell - w_i) + v_i)$.
 - 4.11. So the recurrence holds in this case.
 - 5.1. Case 3. Finally consider the remaining case, when $i > 0$ and $w_i > \ell$.
 - 5.2. Each w_j is non-negative, so no subset of $\{1, \dots, i\}$ that contains i has total weight at most ℓ .
 - 5.3. So the subsets $S \subseteq \{1, \dots, i\}$ with $w(S) \leq \ell$ are just the $S \subseteq \{1, \dots, i-1\}$ with $w(S) \leq \ell$.
 - 5.4. So $V(i, \ell) = V(i-1, \ell)$, and the recurrence holds in this case.
6. By the case analysis (Cases 1–3) above, the recurrence holds. □

Python code: [dynamic_programming/knapsack.py](#) (Section A.5).

7.7 Weighted Activity Selection

Warm-up—counting pairwise-disjoint subsets

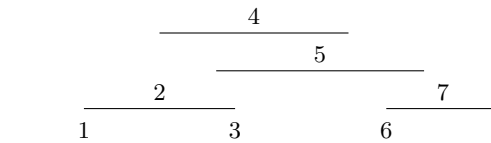
input: A collection $C = \{[s_1, e_1], [s_2, e_2], \dots, [s_n, e_n]\}$ of intervals.

For convenience we assume $e_1 \leq e_2 \leq \dots \leq e_n$. (If not, sort them.)

output: The number of subsets $C' \subseteq C$ of C that are pairwise disjoint.

(These include the subsets of size one or zero.)

Here is an example input, C , consisting of seven intervals. (The interval heights are irrelevant.)



Following Activity Selection, call each interval in C a *job*, and interpret each job $[s_i, e_i]$ as an activity that lasts from start time s_i to end time e_i . A pairwise-disjoint subset of C is then just a collection of jobs can be done without doing more than one job at any time.

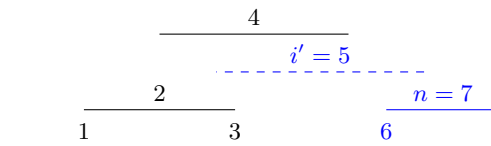
Our first goal is to come up with some recursive or inductive understanding of the collection of pairwise-disjoint subsets. To start, partition the pairwise-disjoint subsets into two types:

- (A) Pairwise-disjoint subsets of C that don't contain the last job $[s_n, e_n]$.
- (B) Pairwise-disjoint subsets of C that do contain the last job $[s_n, e_n]$.

The subsets of type (A) are just the pairwise-disjoint subsets of the first $n-1$ jobs, that is, of the collection $C_A = \{[s_1, e_1], [s_2, e_2], \dots, [s_{n-1}, e_{n-1}]\}$.

What about the subsets of type (B)? Any subset of type (B) contains the job $[s_n, e_n]$, so can't contain any job that overlaps with job $[s_n, e_n]$, that is, can't contain any job $[s_i, e_i]$ with $i \geq i'$, where $i' = \min\{i : e_i \geq s_n\}$ is the index of the first job that overlaps job $[s_n, e_n]$.

In the example, $n = 7$ and $i' = 5$. The jobs overlapping the last job $[s_n, e_n]$ are shown in blue:



So, the subsets of type (B) consist of $[s_n, e_n]$, together with any pairwise-disjoint subset of the first $i' - 1$ jobs, that is, of the collection $C_B = \{[s_1, e_1], [s_2, e_2], \dots, [s_{i'-1}, e_{i'-1}]\}$. The collection C_B can be empty. (In the example, C_B contains the first four jobs.)

This approach gives the following dynamic-programming algorithm.

subproblems and recurrence. For $j \in \{0, 1, 2, \dots, n\}$, define $N(j)$ to be the number of pairwise-disjoint subsets of $\{[s_1, e_1], [s_2, e_2], \dots, [s_j, e_j]\}$. The output will be $N(n)$. The recurrence relation is

$$N(j) = \begin{cases} 1 & \text{if } j = 0 \\ N(j-1) + N(i'-1), \text{ where } i' = \min\{i : e_i \geq s_j\} & \text{if } j > 0. \end{cases}$$

correctness. The recurrence relation follows from the preceding discussion. We don't give a careful proof here.

run time. There are $n + 1$ subproblems. The time to solve each depends on how we compute i' . Here is the iterative version of the algorithm:

```

countSchedules( $C = \{[s_1, e_1], [s_2, e_2], \dots, [s_n, e_n]\}$ ):
1. for  $j = 0, 1, \dots, n$ :
2.   set  $M[j] = \begin{cases} 1 & \text{if } j = 0 \\ M[j - 1] + M[i' - 1], \text{ where } i' = \min\{i : e_i \geq s_j\} & \text{if } j > 0 \end{cases}$ 
3. return  $M[n]$ 

```

— *iterative version*

If we assume constant-time arithmetic (which is not realistic here!) the algorithm can be implemented to run in $\Theta(n \log n)$ time. The main obstacle is computing i' in each iteration. Since the jobs are sorted by end times, i' can be computed in each iteration in $O(\log n)$ time using binary search. Then the n iterations take total time $\Theta(n \log n)$, as does the sort.

But the value of $M[i]$ can be as large as 2^i , so in the worst case requires n bits to store. Then each addition takes $\Theta(n)$ time, so each of the n iterations takes $\Theta(n)$ time, for a total time of $\Theta(n^2)$. Here is what we have figured out:

Theorem 7.7. *Given a collection C of n intervals, the number of pairwise-disjoint subsets of C can be computed in $O(n^2)$ time.*

Weighted Activity Selection

input: A collection $C = \{[s_1, e_1], [s_2, e_2], \dots, [s_n, e_n]\}$ of intervals (called *jobs*), and, for each job $[s_i, e_i]$, an associated value $v_i \geq 0$. Again assume $e_1 \leq e_2 \leq \dots \leq e_n$.

output: The maximum, over all pairwise-disjoint subsets C' of C , of the total value of intervals in C' , that is $\sum_{[s_i, e_i] \in C'} v_i$.

subproblems and recurrence relation. Fix an instance. For $j \in \{0, 1, \dots, n\}$, define $V(j)$ to be the maximum value of any pairwise-disjoint subset from the first j intervals $\{[s_1, e_1], [s_2, e_2], \dots, [s_j, e_j]\}$. The final answer is $V(n)$. The recurrence relation is

$$V(j) = \begin{cases} 0 & \text{if } j = 0 \\ \max(V(j-1), V(i'-1) + v_j), \text{ where } i' = \min\{i : e_i \geq s_j\} & \text{if } j > 0. \end{cases}$$

correctness. As before, we divide the pairwise-disjoint subsets of $\{[s_1, e_1], [s_2, e_2], \dots, [s_j, e_j]\}$ into two types:

- (A) The pairwise-disjoint subsets that don't include the last job $[s_j, e_j]$.
- (B) The pairwise-disjoint subsets that do include the last job $[s_j, e_j]$.

Each pairwise-disjoint subset is of type (A) or type (B), so $V(j)$ is the maximum value of any subset of type (A), or the maximum value of any subset of type (B), whichever is larger.

As before, the subsets of type (A) are just the pairwise-disjoint subsets from the first $j - 1$ jobs, so the maximum value of any subset of type (A) is $V(j - 1)$.

And, also as before, each subset of type (B) consists of $[s_j, e_j]$, together with any pairwise-disjoint subset from the first $i' - 1$ jobs, where i' is the index of the first job that overlaps job j . That is, where $i' = \min\{i : e_i \geq s_j\}$. It follows that the maximum value of any subset of type (B) is $V(i' - 1) + v_j$. This gives the recurrence relation.

Correctness of the algorithm (iterative, or recursive and memoized) follows from the recurrence relation by induction on j (that the algorithm does in fact set $W[j]$ to $V(j)$ as previously defined).

run time. Here is the iterative algorithm.

$\text{maxSchedule}(C = \{[s_1, e_1], [s_2, e_2], \dots, [s_n, e_n]\}, v):$ — *iterative version*

1. for $j = 0, 1, 2, \dots, n$:
2. set $W[j] = \begin{cases} 0 & \text{if } j = 0 \\ \max(W[j-1], W[i'-1] + v_j), \text{ where } i' = \min\{i : e_i \geq s_j\} & \text{if } j > 0. \end{cases}$
3. return $W[n]$

In each iteration, the index i' can be found using binary search on the jobs (which are ordered by end times) in $O(\log n)$ time. Then the n iterations take total time $\Theta(n \log n)$. This proves the following theorem:

Theorem 7.8. *There is an $O(n \log n)$ -time algorithm for Weighted Activity Selection.*

reduction to Longest Path. Consider the DAG that has a vertex for each subproblem $V(j)$, with edges for each dependency (of weight zero or v_j) in the recurrence relation. The problem is equivalent to finding a max-weight path in this graph from the vertex for $V(0)$ to the vertex $V(n)$.

7.8 Longest Common Subsequence (with long-form proof)

input: A pair $I = (A[1..m], B[1..n])$ of sequences.

output: The length of the longest common subsequence of A and B . (That is, the longest sequence that is a subsequence of both A and B .)

Here is the algorithm. Fix any input instance $(A[1..m], B[1..n])$.

subproblems and recurrence. For each pair (i, j) with $i \in \{0, 1, \dots, m\}$ and $j \in \{0, 1, \dots, n\}$, define $L[i, j]$ to be the length of the longest common subsequence between $A[1..i]$ and $B[1..j]$. (The first i elements in A , and the first j elements in B .) The final answer is $L(m, n)$. Here is the recurrence relation:

$$L(i, j) = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0, \\ 1 + L(i - 1, j - 1) & \text{if } i, j > 0 \text{ and } A[i] = B[j], \\ \max(L(i - 1, j), L(i, j - 1)) & \text{if } i, j > 0 \text{ and } A[i] \neq B[j]. \end{cases}$$

correctness. The intuition is as follows. In the case that $A[i] = B[j]$, there is a longest common subsequence of $A[1..i]$ and $B[1..j]$ that ends in the common final element $A[i]$, and, before that final element, consists of a longest common subsequence of $A[1..i - 1]$ and $B[1..j - 1]$ (of length $L(i - 1, j - 1)$). So in that case, $L(i, j) = 1 + L(i - 1, j - 1)$.

In the case that $A[i] \neq B[j]$, every common subsequence y of $A[1..i]$ and $B[1..j]$ either (i) doesn't end with $A[i]$, or (ii) doesn't end with $B[j]$ (or both). In Case (i), y is a common subsequence of $A[1..i - 1]$ and $B[1..j]$. In Case (ii), y is a common subsequence of $A[1..i]$ and $B[1..j - 1]$. Thus, the set of common subsequences of $A[1..i]$ and $B[1..j]$ is the union of two (not necessarily disjoint) subsets:

- (i) the set of common subsequences of $A[1..i - 1]$ and $B[1..j]$, and
- (ii) the set of common subsequences of $A[1..i]$ and $B[1..j - 1]$.

So the length of the longest common subsequence of $A[1..i]$ and $B[1..j]$ is the length of the longest sequence in these two subsets. That is, $L(i, j) = \max(L(i - 1, j), L(i, j - 1))$. A long-form proof of correctness is on the next page.

time. There are $(n + 1) \cdot (m + 1) = O(nm)$ subproblems, and for each, the right-hand side of the recurrence takes constant time to evaluate (assuming the smaller subproblems are already solved). So algorithm (iterative, or recursive and memoized) takes time $\Theta(nm)$.

Theorem 7.9. *There is an $O(nm)$ -time algorithm for Longest Common Subsequence.*

reduction to Longest Path. The problem is equivalent an instance of Longest Path on a graph with a vertex v_{ij} for each subproblem $L(i, j)$, and an edge for each dependency in the recurrence relation (with appropriate weight, 1 or 0).

(continued)

Long-form proof of correctness for Longest Common Subsequence

Here's a long-form proof that the recurrence relation is correct.

Proof (long form).

1. Consider any pair (i, j) with $i \in \{0, \dots, m\}$ and $j \in \{0, \dots, n\}$.
 - 2.1. Case 1. Consider the case that $0 \in \{i, j\}$. ($A[1..i]$ or $B[1..j]$ is the empty sequence.)
 - 2.2. The only common subsequence is the empty one, so $L(i, j) = 0$, and the recurrence is correct.
 - 3.1. Case 2. Otherwise $i \geq 1$ and $j \geq 1$.
 - 3.2.1. Case 2.1. First consider the sub-case when $A[i] = B[j]$.
 - 3.2.2. We first show that $L(i, j) \geq L(i-1, j-1) + 1$.
 - 3.2.3. Let $x = (x_1, x_2, \dots, x_\ell)$ be a common subsequence of $A[1..i-1]$ and $B[1..j-1]$, of length $L(i-1, j-1)$ (it exists by definition of L).
 - 3.2.4. Let $x' = (x_1, x_2, \dots, x_\ell, A[i])$ be obtained by appending $A[i]$ to x .
 - 3.2.5. Since $A[i] = B[j]$, x' is a subsequence of both $A[1..i]$ and $B[1..j]$.
 - 3.2.6. So $L(i, j) \geq |x'| = L(i-1, j-1) + 1$.
 - 3.2.7. Next we show $L(i, j) \leq L(i-1, j-1) + 1$.
 - 3.2.8. Let $y = (y_1, y_2, \dots, y_k)$ be a common subsequence of $A[1..i]$ and $B[1..j]$, of length $L(i, j)$.
 - 3.2.9. Let $y' = (y_1, y_2, \dots, y_{k-1})$ be obtained from y by removing the last element.
 - 3.2.10. Sequence y is a subsequence of $A[1..i]$, so y' is a subsequence of $A[1..i-1]$. (Verify!)
 - 3.2.11. Similarly, y' is a subsequence of $B[1..j-1]$.
 - 3.2.12. So y' is a common subsequence of $A[1..i-1]$ and $B[1..j-1]$.
 - 3.2.13. So $L(i-1, j-1)$ is at least $|y'| = L(i, j) - 1$. Rearranging gives $L(i, j) \leq L(i-1, j-1) + 1$.
 - 3.2.14. By this and Step 3.2.6, $L(i, j) = L(i-1, j-1) + 1$.
 - 3.2.15. So $L(i, j)$ equals the right-hand side of the recurrence in Case 2.1 (when $A[i] = B[j]$).
 - 3.3.1. Case 2.2. In the remaining sub-case $A[i] \neq B[j]$.
 - 3.3.2. First we show $L(i, j) \geq \max(L(i-1, j), L(i, j-1))$.
 - 3.3.3. Let z be a common subsequence of $A[1..i-1]$ and $B[1..j]$ of length $L(i-1, j)$.
 - 3.3.4. Then z is also a common subsequence of $A[1..i]$ and $B[1..j]$.
 - 3.3.5. So $L(i, j) \geq |z| = L(i-1, j)$.
 - 3.3.6. By a symmetric argument, $L(i, j) \geq L(i, j-1)$. So $L(i, j) \geq \max(L(i-1, j), L(i, j-1))$.
 - 3.3.7. Next we show that $L(i, j) \leq \max(L(i-1, j), L(i, j-1))$.
 - 3.3.8. Let $y = (y_1, y_2, \dots, y_\ell)$ be a common subsequence of $A[1..i]$ and $B[1..j]$ of length $L(i, j)$.
 - 3.3.9.1. Case 2.2.1 First consider the case that $y_\ell \neq A[i]$.
 - 3.3.9.2. Then y must be a subsequence of $A[1..i-1]$ (as well as $B[1..j]$).
 - 3.3.9.3. So $L(i-1, j) \geq |y| = L(i, j)$. That is $L(i, j) \leq L(i-1, j)$.
 - 3.3.9.4. So $L(i, j) \leq \max(L(i-1, j), L(i, j-1))$.
 - 3.3.10.1. Case 2.2.2 Otherwise (since $A[i] \neq B[j]$) it must be that $y_\ell \neq B[j]$.
 - 3.3.10.2. By an argument symmetric to the one in Case 2.2.1, $L(i, j) \leq L(i, j-1)$.
 - 3.3.10.3. So $L(i, j) \leq \max(L(i-1, j), L(i, j-1))$.
 - 3.3.11. By Blocks 3.3.9 and 3.3.10, $L(i, j) \leq \max(L(i-1, j), L(i, j-1))$.
 - 3.3.12. By this and Step 3.3.6, $L(i, j) = \max(L(i-1, j), L(i, j-1))$.
 - 3.3.13. So $L(i, j)$ equals the right-hand side of the recurrence in Case 2.2 ($A[i] \neq B[j]$).
 - 3.4. By Blocks 3.2 and 3.3, $L(i, j)$ equals the right-hand side of the recurrence in Case 2 ($0 \notin \{i, j\}$).
 4. By Blocks 2 and 3, $L(i, j)$ equals the right-hand side of the recurrence (for any i, j). □

Python code: [dynamic_programming/longest_common_subsequence.py](#) (Section A.5).

7.9 Edit Distance and Sequence Alignment

input: A pair $I = (A[1..m], B[1..n])$ of strings.

output: The *edit distance* between A and B . This is the minimum cost of any *alignment* between A and B . An alignment is a matching between characters of A and characters of B such that the matching doesn't *cross*: that is, there are no two matched pairs $(A[i], B[j])$ and $(A[i'], B[j'])$ where $i < i'$ but $j > j'$. The cost of an alignment is the total number of unmatched characters (in both sides) plus the number of matched pairs $(A[i], B[j])$ where $A[i] \neq B[j]$.

Equivalently, the edit distance is the minimum number of edit operations that transforms A into B , where each edit operation either deletes a character, inserts a character, or replaces any one character by some other character.

intuition. Fix any input instance $(A[1..m], B[1..n])$. The collection of objects we are working with is the set of alignments between A and B . To derive the algorithm, we classify these alignments into three types:

- (A) Those that don't match $A[m]$ at all.
- (B) Those that don't match $B[n]$ at all.
- (C) Those that match $A[m]$ to $B[n]$.

Every alignment is of one of these types, because an alignment cannot both match $A[m]$ to some character in $B[1..n-1]$ and match $B[n]$ to some character in $A[1..m-1]$. (This is because in such a matching those matched pairs would cross.) It is possible for an alignment to be both type (A) and type (B), but this is not a problem for the algorithm. Next we consider the three types of alignments, and for each we observe that the problem of finding a minimum-cost alignment of that type reduces to a smaller subproblem:

- (A) An alignment of type (A) is just an alignment of $A[1..m-1]$ and $B[1..n]$, so the minimum cost of any alignment of type (A) is just the minimum cost of any alignment between $A[1..m-1]$ and $B[1..n]$.
- (B) Likewise, the minimum cost of any alignment of type (B) is just the minimum cost of any alignment between $A[1..m]$ and $B[1..n-1]$.
- (C) Any alignment of type (C) matches $A[m]$ to $B[n]$, and then the rest of the alignment is just an alignment between $A[1..m-1]$ and $B[1..n-1]$. So, the minimum cost of any alignment of type (C) is just the minimum cost of any alignment between $A[1..m-1]$ and $B[1..n-1]$, plus the cost of matching $A[m]$ to $B[n]$ (1 if $A[m] \neq B[n]$, otherwise 0).

When we apply this reasoning recursively, each subproblem that arises is to compute the minimum cost of any alignment between some prefix of A and some prefix of B . That is, between $A[1..i]$ and $B[1..j]$ for some i and j . Here is the algorithm:

subproblems and recurrence relation. For each pair (i, j) with $i \in \{0, 1, \dots, m\}$ and $j \in \{0, 1, \dots, n\}$, define $M(i, j)$ to be the minimum cost of any alignment between $A[1..i]$ and $B[1..j]$. The final answer is $M(m, n)$. The recurrence relation is

$$M(i, j) = \begin{cases} \max(i, j) & \text{if } i = 0 \text{ or } j = 0, \\ \min(M(i-1, j) + 1, M(i, j-1) + 1, M(i-1, j-1)) & \text{if } i, j > 0 \text{ and } A[i] = B[j], \\ \min(M(i-1, j) + 1, M(i, j-1) + 1, M(i-1, j-1) + 1) & \text{if } i, j > 0 \text{ and } A[i] \neq B[j]. \end{cases}$$

correctness. Correctness follows from the ideas sketched above under “intuition.”

time. There are $(n+1) \cdot (m+1) = O(nm)$ subproblems, and for each, the right-hand side of the recurrence takes constant time to evaluate (assuming the smaller subproblems are already solved). So the algorithm (iterative, or recursive and memoized) take $O(nm)$.

Theorem 7.10. *There is an $O(nm)$ -time algorithm for Edit Distance.*

Python code: [dynamic_programming/edit_distance.py](#) (Section A.5).

Extending the algorithm to Sequence Alignment

KT Section 6.6 considers a generalization of Edit Distance called Sequence Alignment:

input: A pair $I = (A[1..m], B[1..n])$ of strings, *gap penalty* $\delta > 0$, and *mismatch cost function* α defining, for each character a in A and b in B , a *mismatch cost* $\alpha(a, b) \geq 0$.

output: The minimum cost of any alignment of A and B , where an alignment is a non-crossing matching (as defined for Edit Distance), but the cost of the alignment is δ times the total number of unmatched characters (in both sides), plus the sum over the matched pairs $(A[i], B[j])$ of $\alpha(A[i], B[j])$.

Edit Distance is the special case when $\delta = 1$ and $\alpha(a, b) = 1$ for $a \neq b$, and $\alpha(a, b) = 0$ for $a = b$. Following the same reasoning as for the algorithm for Edit Distance gives the following algorithm. Fix any input instance $(A[1..m], B[1..n])$.

subproblems and recurrence relation. For each pair (i, j) with $i \in \{0, 1, \dots, m\}$ and $j \in \{0, 1, \dots, n\}$, define $M(i, j)$ to be the minimum cost of any alignment between $A[1..i]$ and $B[1..j]$. The final answer is $M(m, n)$. The recurrence relation is

$$M(i, j) = \begin{cases} \delta \max(i, j) & \text{if } i = 0 \text{ or } j = 0 \\ \min(M(i-1, j) + \delta, M(i, j-1) + \delta, M(i-1, j-1) + \alpha(A[i], B[j])) & \text{if } i, j > 0. \end{cases}$$

correctness. Correctness follows by reasoning similar to the reasoning for the correctness of the recurrence relation for Edit Distance.

time. There are $(n+1) \cdot (m+1) = O(nm)$ subproblems, and for each, the right-hand side of the recurrence takes constant time to evaluate (assuming the smaller subproblems are already solved). So the algorithm (iterative, or recursive and memoized) takes time $\Theta(nm)$.

Theorem 7.11. *There is an $O(nm)$ -time algorithm for Sequence Alignment.*

7.10 Longest Palindromic Subsequence (with long-form proof)

input: A string $X[1..n]$ (a sequence of characters).

output: The maximum length of any subsequence of X that is a palindrome. (A palindrome is a sequence that is the same as its reverse. The subsequence need not be contiguous. For example, “yobananaboy” is a palindromic subsequence of “your urban cabana bat toy”.)

Here is the algorithm. Fix any input instance $X[1..n]$.

subproblems and recurrence. For each i and j with $1 \leq i \leq n+1$ and $i-1 \leq j \leq n$, define $M(i, j)$ to be the maximum length of any palindromic subsequence of $X[i..j]$. (When $j = i-1$, $X[i..j]$ is the empty string, so $M(i, i-1) = 0$.) The given instance X has solution $M(1, n)$. The recurrence relation is

$$M(i, j) = \begin{cases} 0 & \text{if } j < i, \\ 1 & \text{if } j = i, \\ \max(M(i, j-1), M(i+1, j), 2 + M(i+1, j-1)) & \text{if } j > i \text{ and } X[i] = X[j], \\ \max(M(i, j-1), M(i+1, j)) & \text{if } j > i \text{ and } X[i] \neq X[j]. \end{cases}$$

(We can simplify this somewhat per the comment at the end below.)

correctness. The base cases hold by inspection. Here’s the intuition for the remaining cases. Consider the palindromic subsequences of $X[i..j]$. Categorize these into three types: (A) those that include $X[i]$ and $X[j]$ (necessarily starting with $X[i]$ and ending with $X[j]$), (B) those that don’t include $X[i]$, (C) those that don’t include $X[j]$. Every subsequence is of one (or more) of these types. So the maximum length of any such subsequence is the maximum length of any type-A, type-B, or type-C subsequence.

First consider the case that $X[i] = X[j]$. Then those of type A are of the form $X[i], S, X[j]$ where S is any palindromic subsequence of $X[i+1..j-1]$; so the longest of type A has length $2 + M(i+1, j-1)$. Those of type B are just palindromic subsequences of $X[i+1..j]$; the longest of these has length $M(i+1, j)$. Those of type C are just palindromic subsequences of $X[i..j-1]$; the longest of these has length $M(i, j-1)$. So the recurrence holds in this case.

In the remaining case ($X[i] \neq X[j]$), there are none of type A, because any palindromic subsequence has to start and end with the same character. The analysis of the other two types is the same as in the case $X[i] = X[j]$. So the recurrence holds in this case. A long-form proof of correctness is on the next page.

time. There are $O(n^2)$ subproblems. For each, the right-hand side of the recurrence can be evaluated in constant time. So the algorithm (iterative, or memoized recursive) takes time $\Theta(n^2)$.

Theorem 7.12. *There is an $O(n^2)$ -time algorithm for Longest Palindromic Subsequence.*

reduction to Longest Path. Consider the DAG with a node for each subproblem and a directed edge for each dependency (with appropriate weight 0 or 2), and additional directed edges from the node for $M(0, 0)$ to the nodes for each $M(i, i)$ with $1 \leq i \leq n$ (of weight 1) and to the nodes for each $M(i+1, i)$ with $0 \leq i \leq n$ (the empty substrings, of weight 0). The problem is equivalent to finding a longest path in this DAG from the node for $M(0, 0)$ to the node for $M(1, n)$.

simplifying the recurrence. In the case that $X[i] = X[j]$, for any type-B or type-C subsequence, there is always a type-A subsequence that is at least as long. (We can use a “greedy” exchange argument to prove this; we omit the details.) So the maximum in the recurrence for this case is always achieved by the third term, $2 + M(i+1, j-1)$, and the recurrence simplifies to

$$M(i, j) = \begin{cases} 0 & \text{if } j < i, \\ 1 & \text{if } j = i, \\ 2 + M(i+1, j-1) & \text{if } j > i \text{ and } X[i] = X[j], \\ \max(M(i, j-1), M(i+1, j)) & \text{if } j > i \text{ and } X[i] \neq X[j]. \end{cases}$$

Long-form proof of correctness

Here is a long-form proof that the first recurrence is correct.

Proof (long form).

1. Consider any i and j with $1 \leq i \leq j \leq n$. Note that $j - i + 1$ is the length of the string $X[i..j]$.
 - 2.1. Case 1. In the case that $j - i + 1 \leq 1$, the string $X[i..j]$ has length one or zero.
 - 2.2. So it is its own longest palindromic subsequence, $M(i, j) = j - i + 1$, and the recurrence holds.
 - 3.1. Case 2. Next consider the case that $X[i] = X[j]$ (and $j - i + 1 \geq 2$).
 - 3.2. Consider all possible palindromic subsequences of $X[i..j]$.
 - 3.3. Categorize these into three types:
 - (A) those that include $X[i]$ and $X[j]$,
 - (B) those that don't include $X[i]$, and
 - (C) those that don't include $X[j]$.
 - 3.4. Each palindromic subsequence of $X[i..j]$ is of one (or more!) of these types.
 - 3.5. So $M(i, j)$ is the maximum length of any type-A, type-B, or type-C subsequence.
 - 3.6. The type-A's are of the form $X[i], s, X[j]$, where s is any palindromic subseq. of $X[i + 1..j - 1]$.
 - 3.7. The length of such a subsequence $X[i], s, X[j]$ is maximized when the length of s is maximized.
 - 3.8. By definition of $M(i + 1, j - 1)$, the maximum length of any such s is $M(i + 1, j - 1)$.
 - 3.9. So the maximum length of any type-A subsequence is $2 + M(i + 1, j - 1)$.
 - 3.10. The type-B subsequences are just palindromic subsequences of $X[i + 1..j]$.
 - 3.11. So (by definition of $M(i + 1, j)$) the maximum length of any type-B subsequence is $M(i + 1, j)$.
 - 3.12. Similarly, the maximum length of any type-C subsequence is $M(i, j - 1)$.
 - 3.13. By Steps 3.5, 3.9, 3.11, and 3.12, $M(i, j) = \max(2 + M(i + 1, j - 1), M(i + 1, j), M(i, j - 1))$.
 - 3.14. So the recurrence holds in this case.
 - 4.1. Case 3. Finally consider the remaining case, when $X[i] \neq X[j]$ and $j - i + 1 \geq 2$.
 - 4.2. Since $X[i] \neq X[j]$, no palindromic subsequence of $X[i..j]$ starts with $X[i]$ and ends with $X[j]$.
 - 4.3. So, the reasoning in Case 2 applies, except that there are no type-A subsequences.
 - 4.4. And by that reasoning, $M(i, j) = \max(M(i + 1, j), M(i, j - 1))$.
 - 4.5. So the recurrence holds in this case.
5. By the case analysis (Cases 1–3) above, the recurrence holds. □

Python code: [dynamic_programming/longest_palindrome.py](#) (Section A.5).

7.11 Paragraph Layout

input: A pair $I = (N, w[1..n])$ where N is positive integer, and $w[1..n]$ is an array of words (strings), each of length at most N .

output: A *minimum-cost layout* of w , of width at most N .

A layout breaks the sequence $w[1], w[2], \dots, w[n]$ of words into *lines*, by inserting line breaks after some of the words. For example, suppose $N = 6$ and the given sequence of words is $W[1] = \text{aaa}$, $W[2] = \text{bb}$, $W[3] = \text{cc}$, $W[4] = \text{dddd}$.

Two possible layouts (each with three lines) are shown to the right. The *width* of each line is the number of characters in the line, including the spaces that separate the words. The three lines in Layout 1 have widths 6, 2, and 4. The three lines in Layout 2 have widths 3, 5, and 4.

layout 1	layout 2
aaa.bb	aaa...
cc....	bb.cc.
dddd..	dddd..

Each line in the layout must have width at most N . We define the *cost* of a layout as follows. For each line, define the *gap* of the line to be N minus the width. The three lines in Layout 1 have gaps 0, 4, and 2. The three lines in Layout 2 have gaps 3, 1, and 2. The cost of a layout is the sum, over its lines, of the gap of the line *squared*. Layout 1 has cost $0^2 + 4^2 + 2^2 = 20$. Layout 2 has cost $3^2 + 1^2 + 2^2 = 14$. The goal is to compute a layout of minimum cost.

Minimizing the cost tends to reduce the variation in the gaps, giving a better-looking layout.

Here is the algorithm. Fix any instance (N, w) .

subproblems. For each $i \in \{0, 1, \dots, n\}$, let $M(i)$ be the optimal cost for the subproblem $(N, w[1..i])$ formed by the first i words. The solution to the given problem $(N, w[1..n])$ is $M(n)$.

recurrence. First, define $\text{width}(w[i..j])$ (for $i \leq j$) to be the width of a line formed by words $w[i], w[i+1], \dots, w[j]$. That is, $\text{width}(w[i..j]) = \sum_{h=i}^j (|w[h]| + 1) - 1$. Then the gap of such a line is $N - \text{width}(w[i..j])$. Here is the recurrence for M :

$$M(j) = \begin{cases} 0 & \text{if } j = 0 \\ \min \left\{ M(i-1) + (N - \text{width}(w[i..j]))^2 \mid i \in \{1, \dots, j\}, \text{width}(w[i..j]) \leq N \right\} & \text{if } j > 0. \end{cases}$$

correctness. The intuition is as follows. Each possible layout for words $w[1..j]$ consists (for some i) of a layout for the words in $w[1..i-1]$, followed by one last line with the words in $w[i..j]$. Any $i \in \{1, 2, \dots, j\}$ with $\text{width}(w[i..j]) \leq N$ gives such a layout.

time. There are $n+1 = \Theta(n)$ subproblems. For each, evaluating the right-hand side of the recurrence takes time proportional to the maximum number of words that fit in the line. This is at worst $O(\min(n, N))$. So the algorithm (iterative, or recursive and memoized) takes time $O(n \min(n, N))$.

Theorem 7.13. *There is an $O(n \min(n, N))$ -time algorithm for Paragraph Layout.*

The problem is equivalent to finding a shortest path, in the DAG with vertices for each subproblem and appropriate edge costs, from the node for $M(0)$ to the node for $M(n)$.

Python code: [dynamic_programming/paragraph_layout.py](#) (Section A.5).

7.12 Minimum-Weight Polygon Triangulation

input: A sequence $p[1..n]$ of points in the plane, forming (in clockwise order) the vertices $p[1], p[2], \dots, p[n]$ of a convex polygon.

output: The minimum weight of any *triangulation* of p . A triangulation of p is a set of *chords* (diagonals—straight-line segments connecting pairs of vertices) that subdivides the convex polygon into triangular faces. The weight of the triangulation is the sum of the lengths of the chords. We include the boundary segments (with endpoints p_i, p_{i+1}) as chords.

subproblems and recurrence relation. Fix any input instance $p[1..n]$. For each pair (i, j) with $1 \leq i < j \leq n$, and $j - i \geq 1$, define $T(i, j)$ to be the optimal cost for the subproblem $p[i..j]$ formed by the $j - i + 1 \geq 2$ points $p[i], p[i + 1], \dots, p[j]$. The final answer is $T(1, n)$. The recurrence relation is

$$T(i, j) = \begin{cases} d(p[i], p[j]) & \text{if } j = i + 1, \\ \min \left\{ T(i, k) + T(k, j) + d(p[i], p[j]) \mid k \in \{i + 1, i + 2, \dots, j - 1\} \right\} & \text{if } j > i + 1. \end{cases}$$

Above, $d(p[i], p[j])$ is the Euclidean distance between points $p[i] = (x_i, y_i)$ and $p[j] = (x_j, y_j)$, that is, $d(p[i], p[j]) = \sqrt{(x_i - x_j)^2 + (y_i - y_j)^2}$.

correctness. The intuition is as follows. Each triangulation of $p[i..j]$ can be obtained as follows for some $k \in \{i + 1, \dots, j - 1\}$: take any triangulation of $p[i..k]$, and any triangulation of $p[k..j]$, and add the chord with endpoints $p[i]$ and $p[j]$. (The latter chord must be in the triangulation, and point $p[k]$ is the third vertex on the triangle that has that chord as one side. Draw a picture and consider some examples for more intuition. Make sure to consider the boundary cases when $k \in \{i + 1, j - 1\}$.)

time. There are $\Theta(n^2)$ subproblems. For each, the right-hand side of the recurrence can be evaluated in time $O(n)$ (assuming the smaller subproblems are already solved). So the algorithm (iterative, or recursive and memoized) takes time $\Theta(n^3)$.

Theorem 7.14. *There is an $O(n^3)$ -time algorithm for Min-Weight Polygon Triangulation.*

does not reduce to Shortest Paths! The standard trick of creating a graph with one vertex per subproblem $T(i, j)$ doesn't work, because, in the right-hand side of the recurrence for $T(i, j)$, for each k , the term includes *two* subproblems $T(i, k)$ and $T(k, j)$.

Python code: [dynamic_programming/min_wt_triangulation.py](#) (Section A.5).

7.13 Optimal Binary Search Tree

input: A sequence $p[1..n]$ of *frequencies*, where $p[j]$ is the frequency of *key* j .

output: The minimum cost of any *binary search tree* T for the n keys. A binary search tree is a rooted binary tree with the n keys $\{1, 2, \dots, n\}$ as nodes, *in search-tree order*. (So, at each node, the key is larger than the keys in the node's left subtree, and smaller than the keys in the node's right subtree.) The cost of T is the weighted average depth of the keys (where the root has depth 1, so $\text{depth}_T(j)$ is the number of nodes visited when searching for key j):

$$\text{cost}(T) = \sum_{j=1}^n p[j] \text{depth}_T(j).$$

For any such tree T and key j , define T_j to be the subtree whose root node has key j , containing j and all its descendants.

Observation 7.1. *For any tree T and key j , the set of keys in the subtree T_j is some contiguous sequence $\{i, i+1, \dots, k\}$, where $i \leq j \leq k$. So any such subtree T_j is a binary search tree for the Optimal Binary Search Tree instance $p[i..k]$.*

The cost of T_j for its instance $p[i..k]$ is the weighted average depth of keys in T_j , that is, $\text{cost}(T_j) = \sum_{h=i}^k p[h] \text{depth}_{T_j}(h)$. For any such subproblem $p[i..k]$, define $\text{wt}(p[i..k]) = \sum_{h=i}^k p[h]$ to be the sum of the frequencies in the instance $p[i..k]$.

Observation 7.2. *Consider any subtree T_j . Let $p[i..k]$ be the subproblem it solves. Let T_ℓ and T_r be its left and right subtrees. Then $\text{cost}(T_j) = \text{wt}(p[i..k]) + \text{cost}(T_\ell) + \text{cost}(T_r)$.*

Also, the subproblems solved by T_ℓ and T_r are $p[i..j-1]$ and $p[j+1..k]$.

Here is the algorithm. Fix any input instance $p[1..n]$.

subproblems and recurrence relation. For each $i \in \{1, 2, \dots, n+1\}$ and $k \in \{0, 1, \dots, n\}$ with $i \leq k+1$, define $M(i, k)$ to be the minimum cost of any binary search tree for the subproblem $p[i..k]$ (which is empty if $i = k+1$). The given instance $p[1..n]$ has solution $M(1, n)$. Here's the recurrence relation:

$$M(i, k) = \begin{cases} 0 & \text{if } i > k \\ \min \left\{ \text{wt}(p[i..k]) + M(i, j-1) + M(j+1, k) \mid j \in \{i, i+1, \dots, k\} \right\} & \text{if } i \leq k. \end{cases}$$

correctness. The intuition is that an optimal tree for $p[i..k]$ can be obtained by choosing some key $j \in \{i, i+1, \dots, k\}$ for the root, then choosing any optimal tree T_ℓ for subproblem $p[i..j-1]$ as the left subtree, and any optimal tree T_r for subproblem $p[j+1..k]$ as the right subtree. The subtree for $p[i..k]$ is then a subtree T_j , and, by Observation 7.2, its cost is $\text{cost}(T_j) = \text{wt}(p[i..k]) + \text{cost}(T_\ell) + \text{cost}(T_r) = \text{wt}(p[i..k]) + M(i, j-1) + M(j+1, k)$.

time. There are $O(n^2)$ subproblems. For each, the right-hand side of the recurrence can be evaluated in time $O(n)$. So the algorithm (iterative, or recursive and memoized) takes time $\Theta(n^3)$.

Theorem 7.15. *There is an $O(n^3)$ -time algorithm for Optimal Binary Search Tree.*

does not reduce to Shortest Paths! The standard trick of creating a graph with one vertex per subproblem $M(i, k)$ doesn't work, because, in the right-hand side of the recurrence for $M(i, k)$, for each j , the term includes *two* subproblems $M(i, j-1)$ and $M(j+1, k)$.

Python code: [dynamic_programming/optimal_binary_search_tree.py](#) (Section A.5).

7.14 Single-Source Shortest Paths with Negative Weights

input: An edge-weighted digraph $G = (V, E)$, and source vertex s . (The weights can be negative.)

output: For each vertex $v \in V$, the shortest-path distance from s to v .

subproblems and recurrence relation. Fix an input instance $G = (V, E)$. For each $w \in V$ and $i \in \{0, 1, 2, \dots, n-1\}$ define $M(w, i)$ to be the minimum weight of any s -to- w path in G having exactly i edges. (Notes: (i) If there is no such path, then $M(w, i) = \infty$. (ii) The path doesn't have to be simple.) The final answer is an array d where $d[w] = \min \{M(w, i) : i \in \{0, 1, \dots, n-1\}\}$ for all $w \in V$. (If no negative-weight cycle is reachable from s , then by [Observation 5.7](#) of [Lecture Note 5](#) (applied to the subgraph of G reachable from s) there is a shortest path from s to w that is simple, so has at most $n-1$ edges, so this will be the true shortest-path distance.) Here is the recurrence relation:

$$M(w, i) = \begin{cases} 0 & \text{if } i = 0 \text{ and } w = s, \\ \infty & \text{if } i = 0 \text{ and } w \neq s, \\ \min \{M(u, i-1) + \text{wt}(u, w) \mid (u, w) \in E\} & \text{if } i > 0. \end{cases}$$

As usual we define the minimum of an empty set to be ∞ .

correctness. For the first two (base) cases, the recurrence holds by inspection. For the main case of the recurrence ($i > 0$), the intuition is as follows. $M(w, i)$ is the minimum weight of any i -edge path from s to w . As $i > 0$, each such path consists of an $(i-1)$ -edge path from s to u , followed by the edge (u, w) , for some in-neighbor u of w . For a given final edge (u, w) , among the i -edge paths ending in (u, w) , the shortest consists of a shortest $(i-1)$ -edge path to u , followed by the edge (u, w) , so has weight $M(u, i-1) + \text{wt}(u, w)$.

time. For a given $M(w, i)$, the time to evaluate the right-hand side of the recurrence is proportional to the in-degree of w , plus 1. Summing over the subproblems, the total time is proportional to $\sum_{i=0}^{n-1} \sum_{w \in V} 1 + \text{in-degree}(w) = \sum_{i=0}^{n-1} n + m = n(n+m)$. So the algorithm (iterative, or recursive and memoized) takes time $\Theta(n(n+m))$, where $n = |V|$ and $m = |E|$. We can assume without loss of generality that $m = \Omega(n)$ —if not we can preprocess the graph to remove vertices that have no incident edges, after which $m \geq n/2$. Then this is $\Theta(nm)$.

Theorem 7.16. *There is an $O(nm)$ -time algorithm for Single-Source Shortest Paths with Neg. Wts.*

Detecting negative cycles. The algorithm can be modified to detect negative-weight cycles reachable from s , if there are any, as follows. Recall that $d[v] = \min_{i \leq n-1} M(v, i)$ is the s -to- v distance as computed by the algorithm. The graph has a negative-weight cycle reachable from s if and only if there is an edge (u, w) such that $d[w] > d[u] + \text{wt}(u, w)$.¹

Bellman-Ford algorithm. The well-known and closely related Bellman-Ford algorithm achieves the same time bound. One variant of Bellman-Ford is as follows:

Bellman-Ford($G = (V, E), s$)

1. initialize $d[s] = 0$ and $d[v] = \infty$ for $v \neq s$
2. for $j = 1, 2, \dots, n-1$:
3. for $w \in V$:
4. $d[w] = \min(d[w], \min\{d[u] + \text{wt}(u, w) : (u, w) \in E\})$
5. return d

¹By the reasoning above, if there is no such cycle, the algorithm computes distances correctly, that is, $d[v] = \text{dist}(s, v)$ for all v . This implies $d[w] \leq d[u] + \text{wt}(u, w)$, because $\text{dist}(s, w) \leq \text{dist}(s, u) + \text{wt}(u, w)$ for all (u, w) .

Conversely, suppose $d[w] \leq d[u] + \text{wt}(u, w)$ for all edges (u, w) . Then $\text{wt}(u, w) \geq d[w] - d[u]$. For any cycle C reachable from s , summing this inequality over the edges in C gives

$$\text{wt}(C) = \sum_{(u,w) \in C} \text{wt}(u, w) \geq \sum_{(u,w) \in C} d[w] - d[u] = 0.$$

To show correctness, we can use a similar argument to show by induction on j that, after each iteration j , for all vertices $v \in V$, (i) $d[v]$ is *at most* $\min_{i \leq j} M(v, i)$ and (ii) $d[v]$ is the weight of some s -to- v path. It follows that, if the graph has no negative cycles reachable from s , then, after iteration $n - 1$, $d[v] = \text{dist}(s, v)$ for all $v \in V$.

Python code: [dynamic_programming/Bellman_Ford.py](#) (Section A.5).

7.15 All Pairs Shortest Paths / Floyd-Warshall Alg. (w/ l. f. proof)

input: An edge-weighted digraph $G = (V, E)$ with no negative-weight cycles. To ease notation assume $V = \{1, 2, \dots, n\}$.

output: An array of shortest-path distances d such that for all $u, w \in V$, $d[u, w] = \text{dist}(u, w)$ is the minimum shortest-path distance in G from u to w .

subproblems and recurrence relation. Fix an input instance $G = (V, E)$. For each $u, w \in V$ and $i \in \{0, 1, 2, \dots, n\}$ define $M(u, w, i)$ to be the minimum weight of any u -to- w path in G whose interior vertices are all in $\{1, 2, \dots, i\}$. (If there is no such path, then $M(u, w, i) = \infty$.) The final answer is an array d where $d[u, w] = M(u, w, n)$ for all $u, w \in V$. Here is the recurrence relation:

$$M(u, w, i) = \begin{cases} 0 & \text{if } u = w \\ \text{wt}(u, w) & \text{if } u \neq w, i = 0, \text{ and } (u, w) \in E \\ \infty & \text{if } u \neq w, i = 0, \text{ and } (u, w) \notin E \\ \min(M(u, w, i-1), M(u, i, i-1) + M(i, w, i-1)) & \text{if } u \neq w \text{ and } i \neq 0 \end{cases}$$

correctness. The intuition for the main case of the recurrence (when $i \neq 0$ and $u \neq w$) is as follows. Say a path p from u to w in G is *allowed* for $M(u, w, i)$ if the interior vertices of p (the vertices in p other than the endpoints u and w) are all in $\{1, 2, \dots, i\}$. Then $M(u, w, i)$ is the minimum weight of any allowed path for $M(u, w, i)$. Partition the allowed paths for $M(u, w, i)$ into three types: (A) those that don't have i as an interior vertex, (B) those on which i occurs once as an interior vertex, and (C) those on which i occurs more than once as an interior vertex.

The type-A paths are just those that are allowed for $M(u, w, i-1)$. So the minimum weight of any type-A path is $M(u, w, i-1)$.

The type-B paths are those paths p that can be obtained as follows. Let p_1 be any allowed path for $M(u, i, i-1)$. Let p_2 be any allowed path for $M(i, w, i-1)$. Obtain path $p = p_1 \cup p_2$ by appending p_2 to p_1 , so $\text{wt}(p) = \text{wt}(p_1) + \text{wt}(p_2)$. A minimum weight type-B path p can be obtained by taking p_1 to be a minimum-weight allowed path for $M(u, i, i-1)$, and taking p_2 to be a minimum-weight allowed path for $M(i, w, i-1)$, so p has weight $M(u, i, i-1) + M(i, w, i-1)$.

We can ignore any type-C path p , because by removing cycles (from i to i) from p , we can find a type-B path with $\text{wt}(p') \leq \text{wt}(p)$. A long-form proof of correctness is below.

time. There are $\Theta(n^3)$ subproblems. For each, the right-hand side of the recurrence can be evaluated in $\Theta(1)$ time. So the algorithm (iterative, or recursive and memoized) takes time $\Theta(n^3)$.

Theorem 7.17. *There is an $O(n^3)$ -time algorithm for All-Pairs Shortest Paths.*

Long-form proof of correctness for the Floyd-Warshall recurrence relation

Here's a long-form proof that the recurrence is correct.

Proof (long form).

1. Following the explanation, for any $u, w \in V$ and $i \in \{0, \dots, n\}$, say a path p from u to w is *allowed* for $M(u, w, i)$ if all interior vertices of p are in $\{1, 2, \dots, i\}$. Then $M(u, w, i)$ is the minimum weight of any allowed path for $M(u, w, i)$.
 - 2.1. Consider any $u, w \in V$ and $i \in \{0, 1, \dots, n\}$.
 - 2.2. First we consider the boundary cases of the recurrence.
 - 2.3.1. Case 1. Suppose $u = w$.
 - 2.3.2. The empty path (of weight zero) is a u -to- w path.
 - 2.3.3. There is no path of less weight (as the graph has no negative cycles).
 - 2.3.4. So $M(u, w, i) = 0$, and the recurrence holds in Case 1.
 - 2.4.1. Case 2. Suppose $u \neq w$, $i = 0$, and $(u, w) \in E$.

- 2.4.2. As $i = 0$, the allowed paths for $M(u, w, i)$ are the u -to- w paths with no interior vertices.
- 2.4.3. The only such path consists of just the edge (u, w) , with weight $\text{wt}(u, w)$.
- 2.4.4. So $M(u, w, i) = \text{wt}(u, w)$, and the recurrence holds in Case 2.
- 2.5.1. Case 3. Suppose $u \neq w$, $i = 0$, and $(u, w) \notin E$.
- 2.5.2. As $i = 0$, the allowed paths for $M(u, w, i)$ are the u -to- w paths with no interior vertices.
- 2.5.3. Since (u, w) is not an edge, there is no such allowed path.
- 2.5.4. So $M(u, w, i) = \infty$, and the recurrence holds in Case 3.
- 2.6. Next we consider the main case of the recurrence.
- 2.7.1. Case 4. In the remaining case, $u \neq w$, and $i \neq 0$.
- 2.7.2. We show that $M(u, w, i) = \min(M(u, w, i-1), M(u, i, i-1) + M(i, w, i-1))$.
- 2.7.3. Every allowed path for $M(u, w, i-1)$ is also allowed for $M(u, w, i)$.
- 2.7.4. So $M(u, w, i) \leq M(u, w, i-1)$.
- 2.7.5. Next we show $M(u, w, i) \leq M(u, i, i-1) + M(i, w, i-1)$.
- 2.7.6. If $M(u, i, i-1) + M(i, w, i-1) = \infty$, then the above inequality holds, so assume otherwise.
- 2.7.7. Let p_1 be an allowed path for $M(u, i, i-1)$ with $\text{wt}(p_1) = M(u, i, i-1)$.
(It exists, as $M(u, i, i-1)$ is finite.)
- 2.7.8. Let p_2 be an allowed path for $M(i, w, i-1)$ with $\text{wt}(p_2) = M(i, w, i-1)$.
(It exists, as $M(i, w, i-1)$ is finite.)
- 2.7.9. Then $p = p_1 \cup p_2$ is a path from u to w whose interior vertices are all in $\{1, \dots, i\}$.
- 2.7.10. That is, p is an allowed path for $M(u, w, i)$.
- 2.7.11. So $M(u, w, i) \leq \text{wt}(p) = \text{wt}(p_1) + \text{wt}(p_2) = M(u, i, i-1) + M(i, w, i-1)$.
- 2.7.12. By this and Step 2.7.4, $M(u, w, i) \leq \min(M(u, w, i-1), M(u, i, i-1) + M(i, w, i-1))$.
- 2.7.13. Next we show $M(u, w, i) \geq \min(M(u, w, i-1), M(u, i, i-1) + M(i, w, i-1))$.
- 2.7.14. If $M(u, w, i)$ is infinite, then the inequality above holds, so assume otherwise.
- 2.7.15. Let p be an *acyclic* allowed path for $M(u, w, i)$ with $\text{wt}(p) = M(u, w, i)$.
(It exists, because $M(u, w, i)$ is finite, and the graph has no negative-weight cycles.)
- 2.7.16.1. Case 4.1 First consider the case that i is not an interior vertex of p .
- 2.7.16.2. Then p is allowed for $M(u, w, i-1)$, so $M(u, w, i-1) \leq \text{wt}(p) = M(u, w, i)$.
- 2.7.16.3. So $M(u, w, i) \geq \min(M(u, w, i-1), M(u, i, i-1) + M(i, w, i-1))$.
- 2.7.17.1. Case 4.2 Otherwise i occurs as an interior vertex of p , just once since p is acyclic.
- 2.7.17.2. Let p_1 be the prefix of p ending at i . Let p_2 be the suffix of p starting at i .
- 2.7.17.3. Since i occurs just once as an interior vertex of p , i is not an interior vertex of p_1 or p_2 .
- 2.7.17.4. So p_1 is an allowed path for $M(u, i, i-1)$, and p_2 is an allowed path for $M(i, w, i-1)$.
- 2.7.17.5. So $\text{wt}(p_1) \geq M(u, i, i-1)$, and $\text{wt}(p_2) \geq M(i, w, i-1)$.
- 2.7.17.6. So $M(u, w, i) = \text{wt}(p) = \text{wt}(p_1) + \text{wt}(p_2) \geq M(u, i, i-1) + M(i, w, i-1)$.
- 2.7.17.7. So $M(u, w, i) \geq \min(M(u, w, i-1), M(u, i, i-1) + M(i, w, i-1))$.
- 2.7.18. By Blocks 2.7.16 and 2.7.17, $M(u, w, i) \geq \min(M(u, w, i-1), M(u, i, i-1) + M(i, w, i-1))$.
- 2.7.19. By this and Step 2.7.12, $M(u, w, i) = \min(M(u, w, i-1), M(u, i, i-1) + M(i, w, i-1))$.
- 2.7.20. So the recurrence holds in Case 4.
- 2.8. By Blocks 2.3-2.7, the recurrence holds for $M(u, w, i)$. □

Python code: [dynamic_programming/Floyd_Warshall.py](#) (Section A.5).

7.16 External sources on dynamic programming

	JE	DPV	KT	KT slides	CLRS	MIT video	Python code
intro	3	6.1	6.1	-	15	19 (2011)	-
paths in DAGS	8.5	4.7	-	-	24.2	-	py
Making Change	Ex. 1; Ch. 3	Ex. 6.17–6.19	-	-	-	-	py
Subset Sum and Knapsack	3.8	6.4	6.4	6.4	-	21 (2011)	py, py
Weighted Activity Selection	-	-	6.1	6.1	-	-	-
Longest Common Subsequence	-	-	-	-	15.4	15 (2005)	py
Edit Distance	3.7	6.3	6.6	6.6	-	21 (2011)	py
Longest Palindromic Subseq.	-	-	-	-	Prob. 15-2	10 (2015)	py
Paragraph Layout	-	-	-	-	Prob. 15-4	20 (2011)	py
Min-Wt Polygon Triangulation	-	Prob. 6.12	-	-	-	-	py
Optimal Binary Search Tree	3.9	-	-	-	15.5	-	py
Bellman-Ford	8.7	4.6	6.8	6.8	24.1	18 (2005)	py
Floyd-Warshall	9.8	6.6	-	-	25.2	11 (2015)	py
Segmented Least Squares	-	-	6.3	6.3	-	-	-

The table entries in blue above (for the slides, videos, and code) are hyperlinked to their sources. In case you have a printed copy of these notes, you can find them via the following URLs:

KT slides: <https://www.cs.princeton.edu/~wayne/kleinberg-tardos/>

2005 videos: <https://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-046j-introduction-to-algorithms-sma-5503-fall-2005/>

2011 videos: <https://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-006-introduction-to-algorithms-fall-2011/>

2015 videos: <https://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-046j-design-and-analysis-of-algorithms-spring-2015/>

Python code: <https://github.com/nealeyoung/algs101/tree/main/src/code>

Lecture Note 8

Max Flow, Reductions, and Linear Programming

[↑ CONTENTS](#)

Contents

8.1	Maximum Flow	146
8.1.1	The Minimum s - t Cut problem	147
8.1.2	A greedy Max-Flow algorithm that doesn't work	147
8.1.3	The Ford-Fulkerson algorithm	148
8.1.4	Consequences: The Max-Flow/Min-Cut Theorem; Integer Flows	150
8.1.5	Proof of Lemma 8.1	151
8.1.6	External sources on Max Flow	151
8.2	Polynomial-time reductions	152
8.2.1	Example 1: Bipartite Matching $\leq_P$ Max Flow	153
8.2.2	Example 2: Disjoint Paths $\leq_P$ Max Flow	155
8.2.3	Exercise: Bipartite Matching $\leq_P$ Disjoint Paths	156
8.2.4	The two key properties of polynomial-time reductions	156
8.2.5	External sources on polynomial-time reductions	157
8.3	Linear Programming and Integer Linear Programming	159
8.3.1	Max Flow as a linear program	160
8.3.2	Min Cut as a linear program	160
8.3.3	Shortest s - t path as a linear program	161
8.3.4	Subset Sum as an integer linear program	162
8.3.5	External sources on linear programming	163

8.1 Maximum Flow

The problem is defined as follows.

input: a flow network $(G = (V, E), \text{cap}, s, t)$, consisting of

digraph $G = (V, E)$; edge capacities $\text{cap} : E \rightarrow \mathbb{R}_+$; source $s \in V$ and sink $t \in V$

output: a maximum s - t flow f in G (defined below)

Fix any input $(G = (V, E), \text{cap}, s, t)$. What is an s - t flow? Imagine that there is a water source at s . Water must be sent from s , through the edges and vertices of the graph, to sink t . Each edge $(u, w) \in E$ can carry at most $\text{cap}(u, w)$ units of water per second from u to w . For each vertex v other than s or t , all water entering v must leave v . The goal is to maximize the rate at which water is sent.

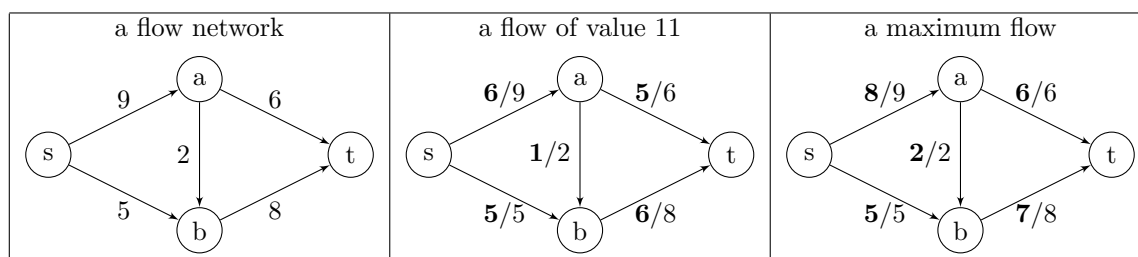
Formally, an s - t flow f in G is a function $f : E \rightarrow \mathbb{R}_+$, assigning a flow value $f(u, w) \geq 0$ to each edge $(u, w) \in E$, subject to the following constraints. For each edge $(u, w) \in E$, the flow $f(u, w)$ on the edge must be at most the capacity $\text{cap}(u, w)$. (These are called the *capacity* constraints.) For each vertex $v \in V \setminus \{s, t\}$ other than the source or sink, the flow on the edges into v must be equal to the flow on the edges leaving v . (These are called the *conservation* constraints.)

Any function $f : E \rightarrow \mathbb{R}_+$ meeting these constraints is called a *flow* (for the given flow network). The *value* (or *size*) of the flow, denoted $|f|$, is the net flow out of the source. (As we shall see, this necessarily equals the net flow into the sink.) The goal is to find a flow f of maximum value.

Here is a restatement of the problem. Let $\text{in}(v) = \{u \in V : (u, v) \in E\}$ denote the in-neighbors of v . Let $\text{out}(v) = \{w \in V : (v, w) \in E\}$ denote the out-neighbors. Using this notation, the problem is as follows:

$$\begin{aligned} \text{maximize } |f| &= \sum_{w \in \text{out}(s)} f(s, w) - \sum_{u \in \text{in}(s)} f(u, s) \quad \text{subject to:} \\ (\forall (u, w) \in E) \quad &0 \leq f(u, w) \leq \text{cap}(u, w) \quad \text{--- capacity constraints} \\ (\forall v \in V \setminus \{s, t\}) \quad &\sum_{w \in \text{out}(v)} f(v, w) = \sum_{u \in \text{in}(v)} f(u, v) \quad \text{--- conservation constraints} \end{aligned}$$

Here is one example of a flow network, a flow in it, and a maximum flow in it:



8.1.1 The Minimum s - t Cut problem

input: a flow network $(G = (V, E), \text{cap}, s, t)$

output: an s - t cut of minimum *capacity* (terms defined below)

An *cut* is a partition of the vertex set V into two parts¹ (S, T) . Given any such cut, define $E(S, T)$ to be the set of edges crossing the cut from S to T . That is, $E(S, T) = \{(u, w) \in E : u \in S, w \in T\} = E \cap (S \times T)$. Define the *capacity* of the cut, $\text{cap}(S, T)$, to be the total capacity of those edges. For any flow f , define the *flow from S to T* , $f(S, T)$, to be the total flow on those edges. That is

$$\text{cap}(S, T) = \sum_{(u, w) \in E(S, T)} \text{cap}(u, w), \quad \text{and} \quad f(S, T) = \sum_{(u, w) \in E(S, T)} f(u, w).$$

In the previous example,

$$\text{cap}(\{s\}, \{a, b, t\}) = 9 + 5 = 14,$$

$$\text{cap}(\{s, b\}, \{a, t\}) = 17, \text{ and}$$

$$\text{cap}(\{s, a, b\}, \{t\}) = 6 + 8 = 14.$$

An s - t cut is a cut (S, T) with the additional property that $s \in S$ and $t \in T$. Next we'll show that the capacity of *any* s - t cut is an upper bound on the value of *any* s - t flow. To prove it we use the following useful property of flows:

Lemma 8.1. *Let f and (S, T) be any flow and s - t cut. Then $|f| = f(S, T) - f(T, S)$.*

That is, the value of the flow equals the flow from S to T minus the flow from T to S .

In the previous example, for the cut $S = \{s, b\}$, the flow f in the third panel has $f(\{s, b\}, \{a, t\}) - f(\{a, t\}, \{s, b\}) = (8 + 7) - 2 = 13 = |f|$.

Summing the conservation constraints over all vertices $v \in S \setminus \{s\}$ gives the equation in the lemma. You can find the detailed proof at the end of this note. The bound we mentioned follows:

Lemma 8.2. *Let f and (S, T) be any flow and s - t cut. Then $|f| \leq \text{cap}(S, T)$.*

That is, the value of any s - t flow is at most the capacity of any s - t cut.

Proof. Lemma 8.1 implies $|f| = f(S, T) - f(T, S) \leq f(S, T) \leq \text{cap}(S, T)$.

(The two inequalities hold because $f(T, S) \geq 0$, and f satisfies the capacity constraints.) □

In the previous example, the s - t cut $S = \{s, a\}$ has capacity 13. So, by the lemma, every s - t flow has value at most 13. The flow in the third panel achieves value 13, so must be a maximum flow.

8.1.2 A greedy Max-Flow algorithm that doesn't work

Consider the following greedy algorithm for Maximum Flow:

```

greedy-flow( $G = (V, E), \text{cap}, s, t$ )
1. initialize  $f[u, w] = 0$  for all  $(u, w) \in E$ 
2. for any path  $p$  in  $G$ , define  $\delta(p) = \min\{\text{cap}(u, w) - f[u, w] : (u, w) \in p\}$            ( $p$ 's residual capacity)
3. while there exists an  $s$ - $t$  path  $p$  with positive residual capacity  $\delta(p) > 0$ :
4.1. let  $p$  be any such path
4.2. for each edge  $(u, w) \in p$ : increase  $f[u, w]$  by  $\delta(p)$            ( $\text{augment the flow along path } p$ )
5. return  $f$ 
    
```

In Step 3, the path p with positive residual capacity is called an *augmenting* path. Computing one is easy. First delete the *saturated* edges—those with $f[u, w] = \text{cap}(u, w)$. Then the possible augmenting paths (the s - t paths with positive residual capacity) are the s - t paths in the resulting graph. Such a path can be found (if it exists) in linear time using depth-first or breadth-first search.

¹So $S \subseteq V$ and $T = V \setminus S$.

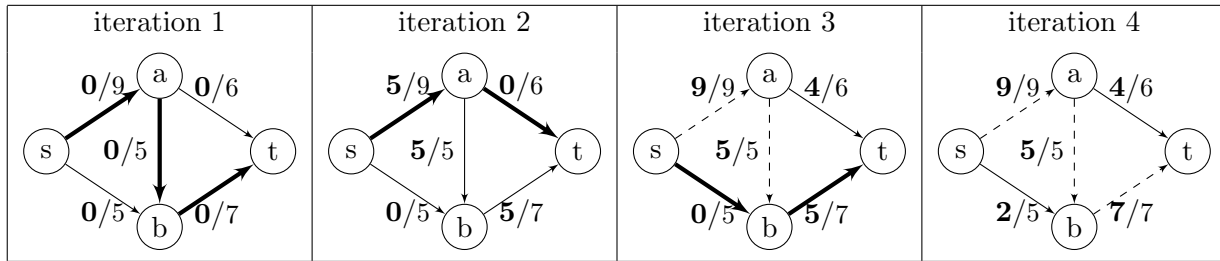


Figure 8.1: An example in which the greedy algorithm gets stuck.

The greedy algorithm maintains the invariant that f is a flow (that is, f satisfies the capacity and conservation constraints). The invariant holds initially because each edge has flow zero. In each iteration, in Step 4.2, increasing the flow by $\delta(p)$ along the s - t path p maintains the invariant, because it maintains the capacity constraints and the conservation constraints. (Verify!) So, if the algorithm terminates, it returns a flow. Unfortunately, the algorithm doesn't work—it can get stuck before it finds a maximum flow, as shown in Figure 8.1.

8.1.3 The Ford-Fulkerson algorithm

In the remainder of this note, we assume that in G no pair $u, w \in V$ of vertices has edges in both directions ($|\{(u, w), (w, u)\} \cap E| \leq 1$). This assumption is just to simplify the notation. It is easy to reduce the general case to this one by “splitting” edges as necessary.

Intuitively, in the example, the greedy algorithm failed because it pushed too much flow along edge (a, b) . To fix it, we introduce a mechanism for allowing the algorithm to “cancel” previously sent flow. Namely, for every edge $(u, w) \in E$, if the current flow f sends positive flow $f(u, w) > 0$ on the edge, we add an artificial *back edge* (w, u) , giving it capacity $f(u, w)$. We also remove all saturated edges. The resulting graph is the *residual graph* $G_f = (V, E_f)$ for f , where

$$E_f = \{(u, w) \in E : f(u, w) < \text{cap}(u, w)\} \cup \{(u, w) : (w, u) \in E, f(w, u) > 0\}.$$

The edge capacities in G_f , called *residual capacities* are

$$\text{cap}_f(u, w) = \begin{cases} \text{cap}(u, w) - f(u, w) & \text{if } (u, w) \in E \text{ and } f(u, w) < \text{cap}(u, w) \\ f(w, u) & \text{if } (w, u) \in E \text{ and } f(w, u) > 0. \end{cases}$$

Ford-Fulkerson ($G = (V, E), \text{cap}, s, t$)	
1. initialize $f[u, w] = 0$ for all $(u, w) \in E$	
2. for any s - t path p in G_f , define $\delta(p) = \min\{\text{cap}_f(u, w) : (u, w) \in p\}$	(p 's residual capacity)
3. while there is an s - t path in G_f :	
4.1. let p be any such path; let $\Delta = \delta(p)$	(note $\delta(p) > 0$ by def'n of G_f)
4.2. for each edge $(u, w) \in p$:	(augment flow along augmenting path p by Δ units)
4.3. if $(u, w) \in E$: increase $f[u, w]$ by Δ	
4.4. else: decrease $f[w, u]$ by Δ	(cancel Δ units of flow on the reverse edge (w, u))
5. return f	

Figure 8.2: The Ford-Fulkerson augmenting-paths algorithm for Maximum Flow.

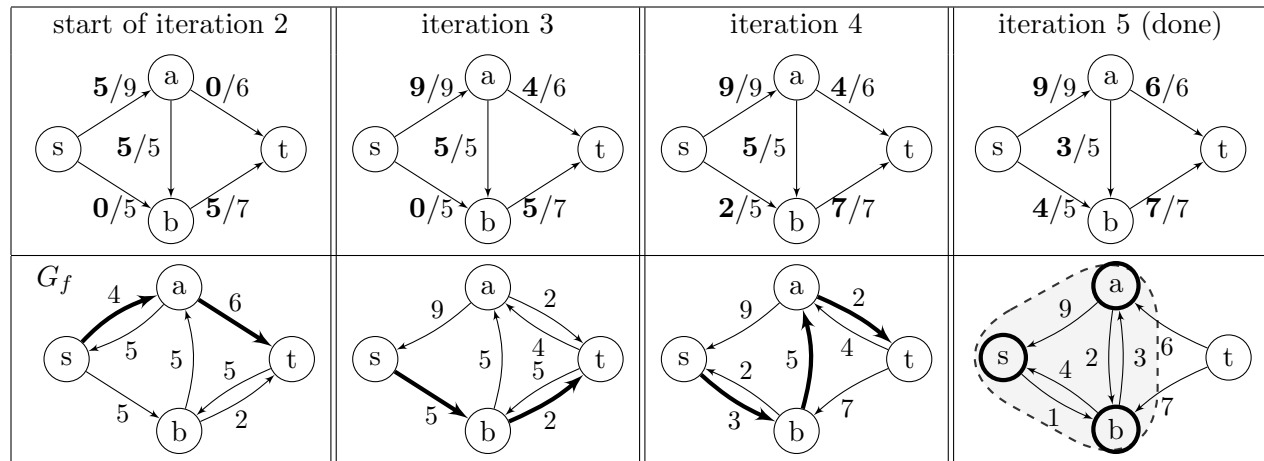


Figure 8.3: Example execution of the Ford-Fulkerson algorithm. The first row shows the flow f in the graph G at the start of each iteration (2–5). The second row shows the residual graph G_f and the augmenting path p in that iteration. After Iteration 5 there is no s - t path in the residual graph G_f (as there are no edges leaving the s - t cut with $S = \{s, a, b\}$), so the algorithm terminates.

We modify the greedy algorithm to maintain the residual graph G_f , and, in each iteration, to use any s - t path in G_f as the augmenting path. (Note that all residual capacities are positive.) This gives the Ford-Fulkerson algorithm, shown in Figure 8.2.

Observation 8.1. *The Ford-Fulkerson algorithm maintains the invariant that f is a flow in G .*

The algorithm maintains the invariant because, in each iteration, augmenting flow along the path p preserves the conservation and capacity constraints. (To see that this still holds when flow is “canceled” on an edge, try some examples.) Figure 8.3 shows the execution of the modified algorithm on the example from Figure 8.1.

Theorem 8.1 (Ford & Fulkerson). *If the algorithm terminates, it returns a maximum s - t flow f .*

Proof. Assume the algorithm terminates. Let f be the flow it returns. Let S contain the set of vertices reachable from the source s in the final residual graph G_f . Let $T = V \setminus S$. The sink t is not in S , because at termination there is no s - t path in the residual graph. So (S, T) is an s - t cut. (In Figure 8.4, $S = \{s, a, b\}$.)

We use the following observation, which follows from the definition of S : *in G_f there is no edge (u, w) from S to T (that is, with $u \in S$ and $w \in T$).* (Otherwise, since u is reachable from s in G_f , w would also be reachable from s , so would be in S .)

Consider any edge $(u, w) \in E$ from S to T in the original graph G . By the observation above, this edge is not present in the residual graph G_f . This implies that edge (u, w) must be saturated: $f[u, w] = \text{cap}(u, w)$.

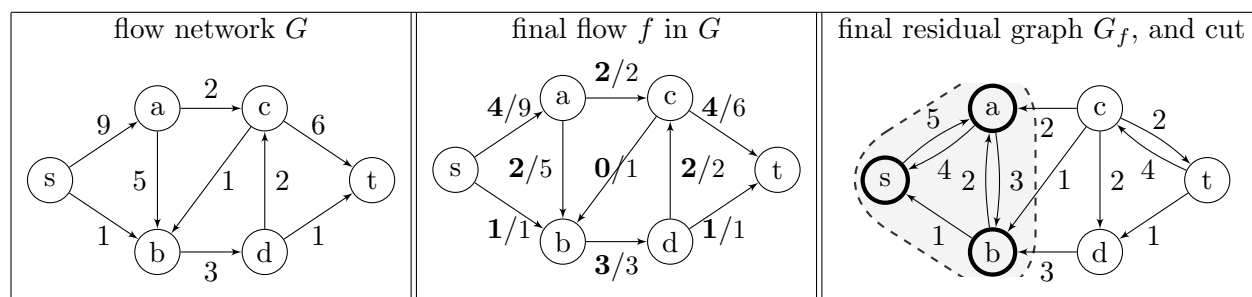


Figure 8.4: An example to illustrate the proof of Theorem 8.1.

Summing over all edges from S to T in E , we get $f(S, T) = \text{cap}(S, T)$. (In Figure 8.4, these edges are (a, c) and (b, d) .)

Consider any edge $(u, w) \in E$ from T to S in the original graph G . The reverse edge (w, u) (which goes from S to T) is not present in the residual graph (by the same observation). By definition of G_f , this implies that edge (u, w) has no flow: $f(u, w) = 0$. Summing over all edges from T to S in E , we get $f(T, S) = 0$. (In Figure 8.4, there is one such edge, (c, b) .)

By Lemma 8.1 and the previous two paragraphs, $|f| = f(S, T) - f(T, S) = f(S, T) = \text{cap}(S, T)$. That is, the value of f equals the capacity of the cut. By Lemma 8.2, no flow has value more than $\text{cap}(S, T)$. So f is a maximum flow. $\square$

Python code: `max_flow.py` (Section A.6).

8.1.4 Consequences: The Max-Flow/Min-Cut Theorem; Integer Flows

We conclude with two important consequences.

Theorem 8.2 (max-flow/min-cut). *In any flow network, the maximum s - t flow value equals the minimum capacity of any s - t cut.*

Proof. Let v^* denote the maximum value of any s - t flow. (It is well defined, as the set of flows is closed under limit.) Let f be a flow of value v^* . Let S contain the vertices reachable from s in G_f . Then $t \notin S$: otherwise an s - t path in G_f would be an augmenting path, and augmenting f along it would give a flow of value more than v^* . Following the reasoning in the proof of Theorem 8.1, (S, T) is an s - t cut with capacity equal to $|f|$. By Lemma 8.2, every cut has capacity at least $|f| = v^*$. So $\text{cap}(S, T) = v^*$ is the minimum possible s - t cut capacity. $\square$

For many applications, the edge capacities are integers, and we want an *integer-valued* flow, where the flow $f(u, w)$ on each edge (u, w) is an integer.

Theorem 8.3. *If the edge capacities are integers, the algorithm returns an integer-valued flow.*

Proof. Assume the edge capacities are integers. The algorithm maintains the invariant that f is integer-valued, because if f is integer valued and each capacity is an integer, then each capacity in the residual graph is a positive integer, so $\delta(p)$ is a positive integer in each iteration. And the algorithm terminates, because in each iteration the value of the flow increases by at least 1, but the value never exceeds the capacity of any s - t cut. $\square$

Note that this algorithm may take exponentially long. Indeed, there are flow networks (with irrational edge capacities) in which the algorithm might not even terminate (depending on how augmenting paths are chosen)! But there are variants of the algorithm that are guaranteed to run in polynomial time. For example, the Edmonds-Karp algorithm (discovered independently by Dinitz) specializes the Ford-Fulkerson algorithm by choosing the augmenting path in each iteration to have a minimum number of edges. (Such a path can be found using breadth-first search.) With this choice, one can prove there are at most $O(|V| \cdot |E|)$ iterations, so the running time is $O(|V| \cdot |E|^2)$.

Python code: `max_flow.py` (Section A.6).

8.1.5 Proof of Lemma 8.1

Fix any flow network $(G = (V, E), \text{cap}, s, t)$.

Lemma 8.1. *Let f and (S, T) be any flow and s - t cut. Then $|f| = f(S, T) - f(T, S)$.*

That is, the value of the flow equals the flow from S to T minus the flow from T to S .

Proof. Fix any such flow f and cost (S, T) . For ease of notation, if, for a pair of vertices $u, w \in V$ there is no edge $(u, w) \in E$, define $f(u, w) = 0$. Following this convention, for every vertex v other than s or t the conservation constraint for v is $\sum_{w \in V} f(v, w) = \sum_{u \in V} f(u, v)$. Then

$$\begin{aligned}
 |f| &= \sum_{w \in V} f(s, w) - \sum_{u \in V} f(u, s) && \text{(by definition of } |f| \text{)} \\
 &= \sum_{v \in S} \sum_{w \in V} f(v, w) - \sum_{v \in S} \sum_{u \in V} f(u, v) && \text{(adding all conservation constraints for } v \in S \setminus \{s\} \text{)} \\
 &= \sum_{v \in S} \sum_{w \in V \setminus S} f(v, w) - \sum_{v \in S} \sum_{u \in V \setminus S} f(u, v) && \text{(canceling the common terms for edges from } S \text{ to } S \text{)} \\
 &= \sum_{v \in S} \sum_{w \in T} f(v, w) - \sum_{v \in S} \sum_{u \in T} f(u, v) && \text{(because } T = V \setminus S \text{)} \\
 &= f(S, T) - f(T, S) && \text{(by definition of } f(S, T) \text{ and } f(T, S) \text{).}
 \end{aligned}$$

□

8.1.6 External sources on Max Flow

- DPV Chapter 7.2. CLRS Chapter 26. KT Chapter 7. JE Chapter 10. MIT lecture video:
 – <https://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-046j-design-and-analysis-of-algorithms-spring-2015/lecture-videos/lecture-13-incremental-improvement-max-flow-min-cut/>

8.2 Polynomial-time reductions

So far our primary strategy for solving a given computational problem has been to design an algorithm from scratch. To this end, we’ve studied basic algorithm-design techniques such as divide-and-conquer, dynamic programming, and greedy algorithms. In the remainder of these notes, we consider a different approach: solving the problem that you want to solve by *reducing* it to a different problem, one that already has a known algorithm. More broadly, our interest is in how different computational problems relate to each other. We start with some necessary formal definitions, then illustrate the concept by describing how to reduce some new problems to Max Flow.

Decision problems (definition). When studying reductions, for simplicity and for historical reasons, we usually cast the problems of interest as *decision problems*—problems with yes/no answers. That is, for any input, the output is either ‘yes’ or ‘no’. Most computational problems can easily be recast in this way. For example, here is the decision version of Max Flow:

Max Flow (decision version)
input: a flow network (digraph $G = (V, E)$ with edge capacities, source s and sink t), and a $k \geq 0$
output: ‘yes’ if G has an s - t flow with value at least k , and otherwise ‘no’

Polynomial-time (definition). The first question we consider when approaching a new computational problem is whether the problem has an algorithm that runs in *polynomial time*—that is, for every possible input, the algorithm computes a correct output in time bounded by some polynomial in the size of the input. The degree of the polynomial can be large, as long as it’s constant. (For example, $O(n^{1000})$ is polynomial time. $O(n \log n)$ is also polynomial time, as $O(n \log n) = O(n^2)$.)

Polynomial-time reductions (definition). Now we can formally define a *polynomial-time reduction*. Let A and B be any two decision problems. A *reduction* from A to B is a function r that maps each input x for A to a corresponding input, $r(x)$, for B . The reduction is *correct* if it has the following property:

For every possible input x to A ,
the answer to x is **yes** if and only if the answer to $r(x)$ (as an instance of B) is **yes**.

That is, if we knew the answer to $r(x)$, it would tell us the answer to x .

And the reduction r is a *polynomial time* reduction if there is an algorithm that, given any input x outputs $r(x)$, with worst-case run time bounded by a polynomial in the length of x . The notation

$$A \leq_P B$$

means “ A reduces in polynomial time to B ”. It means that (up to polynomial factors) A is “no harder” than B : if $A \leq_P B$, and B has a polynomial-time algorithm, then so does A . This is because composing the reduction with any polynomial-time algorithm for B gives a polynomial-time algorithm for A .

8.2.1 Example 1: Bipartite Matching $\leq_P$ Max Flow

To illustrate the concept, consider the following decision problem:

Bipartite Matching (decision version)

input: a bipartite graph $G = (U, W, E)$, and a value k

output: ‘yes’ if G has matching of size at least k , and otherwise ‘no’

Note that a *bipartite* graph $G = (U, W, E)$ is an undirected graph in which the vertices are divided into two parts (sets) U and W , and each edge (u, w) in E is in $U \times W$ (that is, u is in U and w is in W , so all edges go between the parts—no edge has both endpoints in one part).

[Consider an example for intuition.]

Next we define a polynomial-time reduction r from this problem to Max Flow. To define the reduction, one specifies, for every possible input I (for Bipartite Matching) what the output $r(I)$ is. The output must be some input for Max Flow. We have to explicitly define this input in terms of I (that is, from the bipartite graph G and k). Here is the definition of our reduction r :

Step 1. Define the reduction. Do this by specifying its output for every possible input.

Our reduction r from Bipartite Matching to Max Flow:

input: any Bipartite Matching instance $I = (G = (U, W, E), k)$

output: the output $r(I)$ is the Max Flow instance $(G' = (V', E'), s, t, k')$ where

$$k' = k$$

$$V' = U \cup W \cup \{s, t\}$$

$$E' = E \cup \{(s, u) : u \in U\} \cup \{(w, t) : w \in W\}$$

and all edges have capacity 1.

In words: The reduction takes $k' = k$, then obtains G' from G as follows. The vertices of G' are the vertices of G and two new vertices s and t . For each edge (u, w) from U to W in G , there is a directed edge (u, w) in G' with capacity 1. For each vertex $u \in U$, there is a directed edge (s, u) in G' with capacity 1. For each vertex $w \in W$, there is a directed edge (w, t) in G' with capacity 1.

[Consider an example for intuition. See also [KT Section 7.5](#), which presents the same reduction.]

Step 2. Explain how the reduction can be computed in polynomial time. To know that the reduction r is a *polynomial-time* reduction, we have to be sure that, given any input I , the output $r(I)$ can be computed in time polynomial in the size of I .

By inspection of the reduction r defined above, given G and k , G' (and k') can be computed in time linear in the size of G , by simply enumerating the vertices and edges of G appropriately. So the reduction can be computed in polynomial (even linear) time.

Step 3. Prove that the reduction is correct. To know that the reduction r is *correct*, we have to be sure that, for every possible input I , the answer to the output $r(I)$ is **yes** if and only if the answer to I is **yes**. (In other words, to solve I , it suffices to solve $r(I)$.)

Next we'll give a detailed proof. Study the form of this proof carefully. The reasoning for the correctness of any reduction always has the same overall structure. Only the details vary, depending on the specific reduction.

Note: Don't be intimidated by the length of the proof. Most of the steps in the proof are straightforward. Make sure you understand the high-level structure of the reasoning. A short-form proof follows the long-form proof. You may wish to review it first.

Lemma 8.3. *The reduction r is correct. That is, for every instance I of Bipartite Matching, the answer to I is **yes** if and only if the answer to $r(I)$ (as an instance of Max Flow) is **yes**.*

Proof (long form).

1. Consider any possible input $I = (G = (U, W, E), k)$. *block: $\forall$*
2. Let $G' = (V', E')$ and k' be the output $r(I)$ of the reduction given I .
3. First, we'll prove that, if the answer to I is **yes**, then the answer to $r(I)$ is **yes**.
Then, we'll prove the converse: if the answer to $r(I)$ is **yes**, then the answer to I is **yes**.
- 4.1. Suppose that the answer to I is **yes**. *block: if ... then ...*
- 4.2. That is, G has some matching, say M , of size at least k .
- 4.3. We'll show that G' has some s - t flow of size at least $k' = k$.
- 4.4. That is, we'll show that the answer to $r(I)$ is **yes**.
- 4.5. Define flow f as follows:
for every edge (u, w) in M , let $f(s, u) = 1$, let $f(u, w) = 1$, and let $f(w, t) = 1$.
(That is, send 1 unit of flow along the path (s, u, w, t) .)
For every other edge (u, w) in G' , let $f(u, w) = 0$.
- 4.6. By inspection of f , the value of f is the size of M , which is at least $k' = k$.
- 4.7. And f satisfies the conservation and capacity constraints, so f is a valid flow.
- 4.8. So G' has an s - t flow of size at least k' .
- 4.9. So, the answer to $r(I)$ is **yes**.
5. By Block 4, if the answer to I is **yes**, then the answer to $r(I)$ is **yes**.
6. Next we show the converse.
- 7.1. Suppose that the answer to $r(I)$ is **yes**. *block: if ... then ...*
- 7.2. That is, G' has some s - t flow, of value at least $k' = k$.
- 7.3. Since G' has integer capacities, G' has such a flow for which each edge flow is an integer.
- 7.4. Let f be such a flow. Since each edge capacity in G' is 1, each edge flow $f(u, w)$ is either 0 or 1.
- 7.5. We'll show that G has some matching of size at least k .
- 7.6. Define $M = \{(u, w) \in E : f(u, w) = 1\}$ to contain each edge in G that has flow in G' .
- 7.7. For each vertex $u \in U$, there is only one edge into u in G' .
- 7.8. So, because of the conservation constraints, each vertex $u \in U$ is in at most one edge in M .
- 7.9. Likewise, each vertex $w \in W$ is in at most one edge in M .
- 7.10. So M is a matching in G .
- 7.11. Also by the conservation constraints, for each edge (s, u) in G' with $f(s, u) = 1$ there is some edge (u, w) in M , so the size of M is at least the value of f , that is, at least k .
- 7.12. So M is a matching of size at least k in G .
- 7.13. So the answer to I is **yes**.
8. By Block 7, if the answer to $r(I)$ is **yes**, then the answer to I is **yes**.
9. By this and Step 5, the answer to I is **yes** if and only if the answer to $r(I)$ is **yes**. □

As mentioned above, the high-level structure of the proof of correctness is the same for all reductions. Only the details (within the two blocks) depend on the specific reduction. For readers who understand this, it is enough to give just those details. We could do that as follows:

Proof of correctness, short form. Consider any possible input $I = (G = (U, W, E), k)$ to the reduction. Let $G' = (V', E')$ and $k' = k$ be the output of the reduction for that input.

Suppose that G has some matching M of size at least k . Define a flow f in G' by taking $f(s, u) = f(u, w) = f(w, t) = 1$ for every matched edge (u, w) , and giving zero flow to the remaining edges. Then f is an s - t flow whose value is equal to the size of M , which is at least k . So, if G has a matching of size at least k , then G' has a flow of value at least k' .

Conversely, suppose that G' has a flow of value at least $k' = k$. Let f be such a flow that is also integer-valued (there must be one, since the capacities in G' are integers). So, the flow on each edge is either 1 or 0. Let M contain the edges in E whose corresponding edges in G' have flow 1. For each vertex $u \in U$, there is only one edge into u in G' , so (by the conservation constraints), each vertex $u \in U$ is in at most one edge in M . Likewise, each vertex $w \in W$ is in at most one edge in M . So M is a matching in G . Since the value of f is at least k , there are at least k edges out of s that have flow 1. So, by inspection of G' (and the conservation constraints), there are at least k edges in M . So M is a matching of size at least k in G .

By the above reasoning, G has a matching of size at least k if and only if G' has an s - t flow of value at least k' . That is, the answer to I is **yes** if and only if the answer to $r(I)$ is **yes**. So the reduction is correct. $\square$

8.2.2 Example 2: Disjoint Paths $\leq_P$ Max Flow

We give one more reduction, from the following problem to Max Flow.

Disjoint Paths (decision version)
input: a digraph $G = (V, E)$, source s , sink t , and a value k
output: ‘yes’ if G there are k s - t paths in G such that no two share an edge; otherwise ‘no’.

[Consider an example for intuition.]

Next we define a polynomial-time reduction r from this problem to Max Flow. To define the reduction, we specify, for every possible input I (for Disjoint Paths) what the output $r(I)$ is. The output must be some input for Max Flow, that is, a flow network and a value k . We have to say specifically how to obtain this flow network from I (that is, from G , s , t , and k). Here is the definition of our reduction r :

Step 1. Define the reduction. (Specify the output for every possible input.)

Here is our reduction r from Disjoint Paths to Max Flow:

input: any Disjoint Paths instance $I = (G = (V, E), s, t, k)$
output: the output $r(I)$ is the Max Flow instance (G', s, t, k) where G' is obtained from G by assigning capacity 1 to each edge.

[Consider an example for intuition. See also [KT Section 7.6](#), which presents the same reduction.]

Step 2. Explain how the reduction can be computed in polynomial time. By inspection of the reduction r (as defined above), given $I = (G, s, t, k)$, the flow network G' can be computed in time linear in the size of G , by simply adding a capacity to each edge. So the reduction can be computed in polynomial (even linear) time.

Step 3. Prove that the reduction is correct. Following the previous reduction from Bipartite Matching, we’ll assume the reader understands the general high-level structure common to all proofs of correctness for reductions. We’ll give a short-form proof. Before that, we need a utility lemma.

Lemma 8.4. *Given any s - t flow f with positive value, t is reachable from s using only edges that have positive flow.*

Proof (short form). Let G'' be the flow network obtained by reducing the capacity of each edge to equal the flow on the edge, then deleting all edges that have flow (and modified capacity) zero. The original flow remains valid in G'' .

Suppose for contradiction that there is no s - t path in G'' . Let S denote the set of vertices reachable from s in G'' , so that $(S, T = V \setminus S)$ is an s - t cut. Note that no edge (u, w) in G'' leaves S (otherwise u would be reachable from s , but w would not be, even though there is an edge from u to w). So, (S, T) is an s - t cut of capacity zero in G'' . So, every s - t flow in G'' has value at most zero. This is a contradiction, as the original flow is valid in G'' and has positive value. $\square$

Here is the proof that the reduction is correct.

Lemma 8.5. *The reduction r is correct. That is, for every input I , the answer to I is yes if and only if the answer to $r(I)$ is yes.*

Proof (short form). Consider any possible input $I = (G = (V, E), s, t, k)$ to the reduction. Let G' (and s, t, k) be the output of the reduction for that input.

First suppose that G does have k edge-disjoint s - t paths, say, $P_1, P_2, \dots, P_k$. Assume without loss of generality that each path is simple. (If not, just remove all cycles from each path to get k simple edge-disjoint paths.) Define a flow f in G' by setting $f(u, w)$ to be the number of times (u, w) occurs on these paths. Since the paths are edge-disjoint and simple, each flow value $f(u, w)$ will be 0 or 1. Since each edge capacity in G' is 1, f obeys the capacity constraints. Since each of the paths leaves s , the value of f is k . For each vertex v (other than s or t), for each edge (u, v) on some path P_i entering v , the next edge (v, w) on P_i leaves v , so the total flow into v equals the total flow leaving v . So f obeys the conservation constraints. So f is a valid flow in G' , of value k .

Conversely, suppose that G' does have a flow of value at least k . All edge capacities are integers, so there is also an *integer* flow of value at least k . Let f be such a flow. Since each edge capacity is 1, each flow value $f(u, w)$ is 0 or 1. Convert f into a sequence of paths $P_1, P_2, \dots, P_\ell$ as follows:

1. for $i = 1, 2, \dots$, while the value of flow f is positive:
 - 2.1. let P_i be an s - t path that uses only edges (u, w) such that $f(u, w) = 1$
 - 2.2. for each edge $(u, w) \in P_i$, reduce the flow $f(u, w)$ on (u, w) by 1 (to 0)

In each iteration, the s - t path P_i must exist by Lemma 8.4. The iteration reduces the flow on each edge in P_i by 1 (to 0). This preserves the conservation constraints, the capacity constraints, and the property that the flow on each edge is either 0 or 1. But it reduces the value of the flow by 1. It follows that the number of iterations will equal the original value of f . Since this is at least k , the process produces at least k paths. And no edge occurs twice in these paths, because an edge (u, w) is used in a path P_i only when $f(u, w)$ is 1, and after that $f(u, w)$ is reduced to 0. So, if G' has a flow of value at least k , then G has k edge-disjoint s - t paths. (If the process produces more than k paths, just ignore some.) $\square$

8.2.3 Exercise: Bipartite Matching $\leq_P$ Disjoint Paths

As an exercise, find a poly-time reduction from Bipartite Matching to Disjoint Paths.

8.2.4 The two key properties of polynomial-time reductions

Recall that $A \leq_P B$ denotes that there is a polynomial-time reduction from decision problem A to decision problem B . Intuitively, this notation suggests that (up to polynomial time), A is “no harder” than B . Formally, the following holds:

Lemma 8.6. *Let A and B be decision problems such that A reduces in polynomial time to B . If B has a polynomial-time algorithm, then A has a polynomial-time algorithm. That is, if $A \leq_P B$ and $B \in P$, then $A \in P$.*

Proof. Assume r is a poly-time reduction from A to B , and that B has a poly-time algorithm. Fix c and c' such that r is computable in time $O(n^c)$ and B has an algorithm running in time $O(n^{c'})$. Consider the following algorithm for A :

algorithm for A (on input x):

1. compute $y = r(x)$, in time $O(|x|^c)$
2. run the algorithm for B on input y , taking time $O(|y|^{c'})$
3. if the algorithm for B outputs yes, then output yes, otherwise no

This algorithm for A is correct by the following reasoning:

The algorithm for A outputs yes on input x

$\iff$ the algorithm for B outputs yes on input y *(by inspection of the algorithm for A)*

$\iff$ the answer for $r(x)$ is yes *(as the algorithm for B is correct)*

$\iff$ the answer for x is yes *(because r is a correct reduction from A to B)*

So the algorithm is correct.

The algorithm takes polynomial time, $O(|x|^{cc'})$, by the following reasoning. Step 1, computing $y = r(x)$, takes time $O(|x|^c)$ because $r(x)$ is computable in time $O(|x|^c)$. Also, this means that y has size $O(|x|^c)$. So Step 2, running the algorithm for B on y , takes time $O(|y|^{c'}) = O((|x|^c)^{c'}) = O(|x|^{cc'})$. $\square$

One more formal observation: poly-time reductions compose, so the relation $\leq_P$ is transitive:

Lemma 8.7. *Let A , B , and C be decision problems. If A reduces in polynomial time to B , and B reduces in polynomial time to C , then A reduces in polynomial time to C .*

That is, if $A \leq_P B$ and $B \leq_P C$, then $A \leq_P C$.

Proof. Let r_1 and r_2 be the poly-time reductions from A to B , and from B to C . Consider the function r_3 defined by composing r_1 and r_2 , that is, $r_3(x) = r_2(r_1(x))$. The function r_3 is a correct reduction from A to C by the following reasoning:

The answer for x is yes

$\iff$ the answer for $r_1(x)$ is yes *(because r_1 reduces A to B)*

$\iff$ the answer for $r_2(r_1(x))$ is yes *(because r_2 reduces B to C)*

$\iff$ the answer for $r_3(x)$ is yes *(because $r_3(x) = r_2(r_1(x))$)*

So the reduction r_3 is correct.

For the running time, fix c_1 and c_2 such that r_1 is computable in time $O(n^{c_1})$ and r_2 is computable in time $O(n^{c_2})$. The following algorithm computes r_3 in polynomial time, $O(n^{c_1 c_2})$:

algorithm to compute $r_3(x)$:

1. compute $y = r_1(x)$, in time $O(|x|^{c_1})$
2. compute and return $r_2(y)$, taking time $O(|y|^{c_2})$

Clearly the algorithm computes $r_3(x)$. To bound the run time, reason as in the previous proof: Step 1 takes time $O(|x|^{c_1})$, so y has size $|y| = O(|x|^{c_1})$, so Step 2 takes time $O(|y|^{c_2}) = O(|x|^{c_1 c_2})$. $\square$

8.2.5 External sources on polynomial-time reductions

- Reducing matching to flow: [KT Section 7.5](#); [DPV Section 7.3](#); [CLRS Section 26.3](#)
- Reducing disjoint paths to flow: [KT Section 7.6](#)
- For Seam Carving (which reduces to shortest s - t path in a DAG), see https://en.wikipedia.org/wiki/Seam_carving
<https://www.youtube.com/watch?v=6NcIJXTlugc>
- For Image Segmentation via Min Cut, see [KT Section 7.10](#). See also: <https://discovery.ucl.ac.uk/13383/1/13383.pdf>, Figures 1, 2, and 11
- Other reductions to Max Flow or Min Cut:
[KT Sections 7.5–7.12](#) (and [KW slides](#) for Ch. 7); see also [JE Chapter 11](#)

- Reductions in general:
KT Section 8.1 (and [KW slides](#) for Ch. 8); CLRS Section 34.3; and [JE](#) Chapter 12, and
<https://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-006-introduction-to-algorithms-fall-2011/lecture-videos/lecture-23-computational-complexity/>

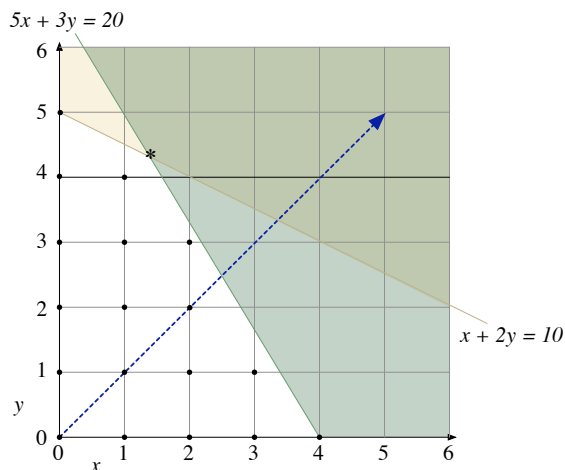


Figure 8.5: The two-dimensional linear program $\max\{x + y : x, y \geq 0; x + 2y \leq 10; 5x + 3y \leq 20\}$. The feasible region is the white portion of the positive quadrant. The optimal point is marked with an asterisk.

8.3 Linear Programming and Integer Linear Programming

Recall that linear algebra is the study of systems of linear equations, which are useful for modeling a wide variety of problems. *Linear programming* can be understood as a generalization of linear algebra, in that it is about systems of linear *inequalities* (including equations). Linear *inequalities* allow us to model an even wider variety of problems.

As a computational problem, an instance of Linear Programming requires finding a vector that optimizes a given linear objective function subject to given linear constraints. For example, given a matrix $A \in \mathbb{R}^{n \times m}$ and vectors $b \in \mathbb{R}^n$, $c \in \mathbb{R}^m$, find

$$\min\{c \cdot x \mid x \in \mathbb{R}^m; x \geq 0; Ax \geq b\}.$$

A vector inequality such as $x \geq 0$ is interpreted component-wise, i.e., as $x_i \geq 0$ for all $i \in \{1, 2, \dots, m\}$. Similarly, $Ax \geq b$ is interpreted component-wise.

A *feasible solution* is any vector x satisfying the constraints $x \in \mathbb{R}^m$, $x \geq 0$, and $Ax \geq b$. The objective function (or *cost*) is $c \cdot x = \sum_j c_j x_j$. Linear programs may take other forms, for example $\max\{c \cdot x \mid x \in \mathbb{R}^m; x \geq 0; Ax \leq b\}$.

Integer Linear Programming. *Integer Linear Programming* extends Linear Programming by allowing an additional constraint on the solution, namely, that each variable should be assigned an *integer* value. For example, given (A, b, c) as above, find $\min\{c \cdot x \mid x \in \mathbb{Z}^m; x \geq 0; Ax \geq b\}$. (Note that $\mathbb{Z}^m$ is the set of m -dimensional integer vectors.) Integer Linear Programming is a so-called *NP-hard* problem—it is powerful enough to model many computationally *intractable* combinatorial problems. So Integer Linear Programming is unlikely to have a polynomial-time algorithm. Nonetheless, many moderately sized Integer Linear Programs can be solved in practice, so formulating a problem as an integer linear program can be an effective way to solve it.

A small linear program. Consider the problem maximize $x + y$ subject to $x + 2y \leq 10$, $5x + 3y \leq 20$, and $x, y \geq 0$. The problem is illustrated in Figure 8.5. The feasible region is the white area in the positive quadrant. The dashed vector points in the direction of increasing cost ($x + y$). The optimal feasible solution is $(1\frac{3}{7}, 4\frac{2}{7})$ (marked by “*”), of cost $5\frac{5}{7}$. The feasible integer solutions (feasible solutions of the corresponding integer linear program) are marked with black dots. Three of these have maximum cost 5. The solution $(2, 2)$ of cost 4 is a $1\frac{1}{4}$ -approximate solution to the integer linear program and a $1\frac{3}{7}$ -approximate solution to the linear program.

Algorithms for Linear Programming. Linear Programming is efficiently solvable. That is, there are polynomial-time algorithms that, given any linear program, find an optimal solution. Historically the so-called *Simplex algorithm* is the most common. Its worst-case run time is not polynomial, but it usually works quickly in practice. In recent decades *Interior-point* algorithms have superseded the Simplex algorithm. These are polynomial-time algorithms. The *Ellipsoid* algorithm was the first polynomial-time algorithm to be discovered, but it is slow in practice, so is of mostly theoretical interest.

Algorithms via Linear Programming. Here's one technique for solving a given computational problem:

- (i) Model the problem as a linear program, that is, reduce the problem to Linear Programming.
- (ii) Use an existing LP solver to solve the linear program, and, therefore, your problem.

This approach can be easier than designing an algorithm from scratch. In the rest of this section we describe how various problems reduce to Linear Programming.

We note in passing that there is another, more sophisticated, approach:

- (i) Model the problem as a linear program.
- (ii) Design an efficient algorithm that mimics what the Simplex algorithm does for that linear program.

For example, the Ford-Fulkerson Max-Flow algorithm and Dijkstra's algorithm for Single-Source Shortest Paths can be viewed as special cases of the Simplex algorithm, obtained by applying the Simplex algorithm (with suitable choices of pivots) to the linear programs for those problems. We won't discuss this further here, as we don't discuss how the Simplex algorithm works. Chapter 7 of [DPV](#) discusses this approach specifically for Max Flow.

8.3.1 Max Flow as a linear program

Linear programs are often given by explicitly writing the constraints, rather than using a matrix. For example, a given Max Flow instance $(G = (V, E), s, t)$ translates directly to the following linear program:

$$\begin{aligned} & \text{maximize} && \sum_{w \in \text{out}(s)} f_{sw} - \sum_{u \in \text{in}(s)} f_{us} && \text{subject to:} \\ & (\forall (u, w) \in E) && 0 \leq f_{uw} \leq \text{cap}(u, w) && \text{--- capacity constraints} \\ & (\forall v \in V \setminus \{s, t\}) && \sum_{w \in \text{out}(v)} f_{vw} = \sum_{u \in \text{in}(v)} f_{uv} && \text{--- conservation constraints} \end{aligned}$$

This is a (very easy!) polynomial-time reduction from Max Flow to Linear Programming.

Recall that, if the edge capacities are integers, then there is always an optimal integer-valued flow. So, in this case, this linear program always has an optimal integer solution, and finding one is not hard.

Recall that Bipartite Matching and Disjoint Paths reduce in polynomial time to Max Flow. Since Max Flow reduces to Linear Programming, it follows that these two problems also reduce to Linear Programming. As an exercise, you could try formulating each of these problems as a linear program directly.

8.3.2 Min Cut as a linear program

A given Min Cut instance $(G = (V, E), s, t)$ is equivalent to the following linear program:

$$\begin{aligned} & \text{minimize} && \sum_{(u,w) \in E} \text{cap}(u, w) x_{uw} && \text{subject to:} \\ & && y_s = 0; y_t = 1 \\ & (\forall v \in V) && 0 \leq y_v \leq 1 \\ & (\forall (u, w) \in E) && 0 \leq x_{uw} \\ & (\forall (u, w) \in E) && y_w - y_u \leq x_{uw} \end{aligned}$$

The next lemma summarizes the relation to Min Cut:

Lemma 8.8. *For any Min Cut instance, every s - t cut (S, T) yields an integer solution to this linear program whose cost $\sum_{(u,w) \in E} \text{cap}(u,w)x_{uw}$ equals $\text{cap}(S, T)$, and every integer solution yields an s - t cut whose capacity is at most the cost of the solution. In particular, the minimum cost of any integer solution is the minimum capacity of any s - t cut.*

Proof sketch. Given any s - t cut (S, T) , the corresponding (integer) solution (x, y) to the linear program is obtained by taking $y_v = 0$ for $v \in S$ and $y_v = 1$ for $v \in T$, and then $x_{uw} = 1$ for each edge (u, w) that crosses the cut, and $x_{uw} = 0$ for each edge that doesn't cross the cut. One can verify that this must be a feasible solution with the correct cost. Conversely, given any integer solution (x, y) to the linear program, the corresponding s - t cut (S, T) is obtained by taking $S = \{v \in V : y_v = 0\}$ and $T = \{v \in V : y_v = 1\} = V \setminus S$. This is an s - t cut, and its capacity is at most the cost of (x, y) : each edge (u, w) that crosses the cut has $y_w - y_u = 1$, so $x_{uw} \geq 1$. $\square$

Also, there is always an optimal integer solution to this linear program:

Lemma 8.9. *For any Min Cut instance, there is an optimal integer solution to this linear program.*

Proof. Here's a clever proof using [the probabilistic method](#).² Fix an instance and any optimal LP solution (x, y) . Choose a number $r \in [0, 1]$ uniformly at random, then let

$$S = \{v \in V : y_v \leq r\} \text{ and } T = \{v \in V : y_v > r\}.$$

This gives a random s - t cut (S, T) . By [linearity of expectation](#), the expected capacity of this cut is

$$\begin{aligned} \sum_{(u,w) \in E} \text{cap}(u,w) \Pr[(u,w) \text{ crosses the cut}] &= \sum_{(u,w) \in E} \text{cap}(u,w) \Pr[y_u \leq r < y_w] \quad (\text{by definition of } (S, T)) \\ &= \sum_{(u,w) \in E} \text{cap}(u,w) \max(0, y_w - y_u) \quad (\text{by definition of } r) \\ &\leq \sum_{(u,w) \in E} \text{cap}(u,w) x_{uw} \quad (\text{by inspection of the LP}) \end{aligned}$$

That is, the expected capacity of the random cut (S, T) is at most the cost of (x, y) . Thus, there must exist a cut (S^*, T^*) whose capacity is at most the cost of (x, y) . By Lemma 8.8, this cut (S^*, T^*) has a corresponding integer solution (x^*, y^*) whose cost equals the capacity of (S^*, T^*) , which is in turn at most the cost of (x, y) . So (x^*, y^*) is also optimal. $\square$

8.3.3 Shortest s - t path as a linear program

Given a directed graph $G = (V, E)$ with non-negative edge weights, and two vertices s and t , consider problem of finding a shortest path from s to t in G . Here are two linear programs that model this problem. We call one the *primal* and the other the *dual*:³

<i>primal</i> minimize $\sum_{(u,w) \in E} \text{wt}(u,w) f_{uw}$ subject to $(\forall v \in V) \quad \sum_{w \in \text{out}(v)} f_{vw} - \sum_{w \in \text{in}(v)} f_{wv} = \begin{cases} 1 & \text{if } v = s \\ -1 & \text{if } v = t \\ 0 & \text{otherwise} \end{cases}$ $(\forall (u,w) \in E) \quad f_{uw} \geq 0$	<i>dual</i> maximize $\pi_t - \pi_s$ subject to $(\forall (u,w) \in E) \quad \pi_w - \pi_u \leq \text{wt}(u,w)$
---	--

For intuition, note that, in the primal linear program, f is a flow of value 1 from s to t . Integer flows of this kind correspond to s - t paths:

²If you're not familiar with probability, this proof won't make much sense.

³The existence of two such linear programs for a problem is not a coincidence. It happens for every linear-programming problem! The Max-Flow/Min-Cut relationship is another example.

Lemma 8.10. *Every s - t path P in G yields an integer solution f to the primal whose cost is $\text{wt}(P)$, and every integer solution f yields an s - t path whose weight is at most the cost of f . In particular, the minimum cost of any integer solution is $\text{dist}(s, t)$.*

Proof sketch. For any s - t path P , the corresponding solution f is defined by giving f_{uw} a value equal to the number of times edge (u, w) occurs in P . Because the path starts at s , ends at t , and enters each other vertex v the same number of times it leaves v , this assignment to f meets the constraints of the primal LP. By inspection the cost of f in the LP equals $\text{wt}(P)$.

Conversely, given any integer solution f to the LP, one can construct a corresponding path P by starting at s , then walking greedily, taking each step along any edge (u, w) where f_{uw} is positive. When traversing an edge (u, w) , append the edge to P and decrement f_{uw} . Because f satisfies the LP constraints, a greedy step is always possible until the walk reaches t . At that point P is an s - t walk of weight at most the cost of f (the flow left over on edges of f costs at least 0), and, because the edge weights are non-negative, P contains an s - t path of weight at most $\text{wt}(P)$. $\square$

Lemma 8.11. *The primal LP always has an optimal integer solution.*

Proof sketch. By an argument similar to the greedy construction in the previous proof, one can show that any solution f can be expressed as a weighted average of some integer solutions $f_1, f_2, \dots, f_\ell$. The cost of f equals the average cost of these solutions, so the cheapest of these integer solutions has cost at most the cost of f . $\square$

It follows that the value of the linear program (that is, the optimal objective value achieved by any feasible solution) is $\text{dist}(s, t)$. Next we consider the dual LP, and show the same for it:

Lemma 8.12. *The value of the dual LP is $\text{dist}(s, t)$.*

Proof. First we show that the value of the dual LP is at most $\text{dist}(s, t)$. Let π^* be an optimal solution to the dual LP, so $\pi_t^* - \pi_s^*$ is the value of the dual LP. Let P^* be a shortest s - t path, so $\text{wt}(P^*) = \text{dist}(s, t)$. Then, summing the dual LP constraints over edges in P^* gives

$$\text{dist}(s, t) = \text{wt}(P^*) = \sum_{(u,w) \in P^*} \text{wt}(u, w) \geq \sum_{(u,w) \in P^*} \pi_w^* - \pi_u^* = \pi_t^* - \pi_s^*.$$

The last step follows because the sum “telescopes”—for each vertex v on the interior of the path P^* , the term π_v^* occurs once with positive sign and once with negative sign, and these terms cancel each other.

To finish we show that the value of the dual LP is at least $\text{dist}(s, t)$. Define a dual solution π by taking $\pi_v = \text{dist}(s, v)$ for every v (assuming every vertex is reachable from s ; deleting the unreachable vertices changes neither $\text{dist}(s, t)$ nor the LPs’ values). This solution π is feasible, because by the triangle inequality,⁴ for any edge (u, w) ,

$$\pi_w - \pi_u = \text{dist}(s, w) - \text{dist}(s, u) \leq \text{wt}(u, w).$$

And the cost of π is $\pi_t - \pi_s = \text{dist}(s, t) - \text{dist}(s, s) = \text{dist}(s, t)$. So, the dual LP has a feasible solution of cost $\text{dist}(s, t)$, and is a maximization problem, so its value must be at least $\text{dist}(s, t)$. $\square$

8.3.4 Subset Sum as an integer linear program

For variety, consider the Smallest Subset Sum problem. Fix an instance $(v = (v_1, v_2, \dots, v_n), T)$. The problem is to determine the smallest subsequence of v that sums to T . Here is an equivalent instance of Integer Linear Programming:

$$\begin{aligned} & \text{minimize} && \sum_{i=1}^n x_i && \text{subject to:} \\ & (\forall i \in \{1, \dots, n\}) && 0 \leq x_i \leq 1, \\ & && \sum_{i=1}^n v_i x_i = T, \\ & && x \in \mathbb{Z}^n. \end{aligned}$$

⁴The triangle inequality is that $\text{dist}(s, w) \leq \text{dist}(s, u) + \text{wt}(u, w)$. It holds because one way to get from s to w is to take a shortest path from s to u , followed by the edge (u, w) .

Setting $x_i = 1$ corresponds to including v_i in the subsequence. (And setting $x_i = 0$ corresponds to omitting it.) In this way, the solutions to this integer linear program correspond bijectively to the subsequences of v that sum to T . The size of each subsequence equals the cost, $\sum_i x_i$, of its corresponding solution x .

What would happen if we were to drop the explicit integrality constraint, “ $x \in \mathbb{Z}^n$ ”? This would give a linear program instead of an integer linear program. But that linear program wouldn’t necessarily have optimal solutions that are integer valued. Consider for example the Smallest Subset Sum instance ($v = (1, 3), T = 2$). It has no solution—no subsequence of v sums to 2. But the corresponding linear program has many solutions, the optimum being $x = (0, 2/3)$.

8.3.5 External sources on linear programming

- DPV Chapter 7. KT Section 11.6 and KW slides. CLRS Chapter 29. MIT lecture video: <https://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-046j-design-and-analysis-of-algorithms-spring-2015/lecture-videos/lecture-15-linear-programming-lp-reductions-simplex/>
- JE notes: <https://jeffe.cs.illinois.edu/teaching/algorithms/notes/H-lp.pdf>
- Microsoft Excel has a free add-on for solving LPs in a spreadsheet. See e.g. <https://www.fm-magazine.com/issues/2019/feb/linear-programming-microsoft-excel.html>
- For more serious applications, Gurobi has a commercial LP solver, free for academic use, that you can install on your own computer. It can also solve Integer Linear Programs of moderate size. <https://gurobi.com>
- Many other free LP solvers are available.

Lecture Note 9

The complexity classes P and NP

[↑ CONTENTS](#)

Contents

9.1	Intractable computational problems	165
9.2	Reductions revisited	165
9.3	Reductions between some problems that are new to us	166
9.3.1	Three similar problems: Clique, Independent Set, and Vertex Cover	166
9.3.2	A very different (?) problem: 3-CNF-SAT	166
9.4	Review of key concepts	168
9.5	The class NP—problems that have poly-time verifiers	169
9.5.1	NP-hard problems; NP-complete problems	169
9.5.2	3-CNF-SAT reduces to Subset Sum	170
9.5.3	3-CNF-SAT reduces to Hamiltonian Path	172
9.6	External sources on P, NP, and NP-completeness	175

9.1 Intractable computational problems

Many important computational problems are *intractable*—not solvable by any reasonably fast algorithm. Since 1950 or so, computer scientists have developed a fairly deep understanding of intractable problems.

In designing algorithms, it is useful to be able tell when a problem is intractable, or likely to be intractable. If your problem is intractable, there is no algorithm that is guaranteed to compute a correct solution to every input in reasonable time. In this case, you need some other approach. For example, perhaps you only care about solving a few particular instances of the problem. Then you might focus on solving just those instances. Or, you might look for fast algorithms that are guaranteed to return provably good *approximate* solutions. Or, you might revisit your application and try to find a different way to formally model what you need to do.

We focus here on a particular kind of intractability: whether or not the problem has any algorithm whose worst-case running time is bounded by *any polynomial*—that is, for some constant $c > 0$, on any input x , the algorithm takes time $O(|x|^c)$. The class of problems that have such algorithms is called *Polynomial Time*, and is often denoted P (or $\mathbb{P}$). Note that this is a class of *problems*, not algorithms. P is one of many *complexity classes* studied by computer scientists.

We also focus on another particular class of algorithmically important problems, so-called “NP-complete” problems. We have *strong evidence* that these problems are not in P (have no polynomial-time algorithms) so, are very likely to be intractable in that sense.

9.2 Reductions revisited

[Lecture Note 8](#) formally defined polynomial-time reductions, and gave reductions from Bipartite Matching to Max Flow, from Disjoint Paths to Max Flow, from Max Flow to Linear Programming, and from Shortest s - t Path to Linear Programming. These enable us to, for example, use any algorithm for Linear Programming to solve Shortest s - t Path, Max Flow, and Bipartite Matching.

Recall that when considering reductions we focus on *decision problems*—computational problems with ‘yes’/‘no’ answers. For any decision problem A , we define the *language*, $L(A)$ associated with A to be the set of inputs for which the answer is “yes”. For example, the language for the decision problem for Shortest s - t Path is the set

$$L = \{(G, s, t, W) \mid \text{there is an } s\text{-}t \text{ path in } G \text{ of weight at most } W\}.$$

To be formally precise, we have in mind here that each input is encoded in a standard way as a binary string. So, each language is, technically, a set of binary strings. But we can safely forget about this most of the time. It is important mainly in that it gives us a robust and general notion of input size—the size of an input is the number of bits in the encoding of the input.

If an algorithm for a decision problem A says ‘yes’ for a given input x , we say the algorithm *accepts* x . So, if the algorithm is correct, then, for all inputs x the algorithm accepts x if and only if $x \in L(A)$.

Recall from [Lecture Note 8](#) that a reduction from a decision problem A (such as Bipartite Matching) to another decision problem B (such as Max Flow) is, formally, a function r such that, given any input x for problem A , the function produces an equivalent instance $r(x)$ of problem B . *Equivalent* means that x is in $L(A)$ if and only if $r(x)$ is in $L(B)$. The reduction is a *polynomial-time* reduction if the function r can be computed in polynomial time. That is, there is an algorithm that, given input x , outputs $r(x)$ in time polynomial in $|x|$. Recall that the notation

$$A \leq_P B$$

means that there is a polynomial-time reduction from A to B . Informally, it means that A is “no harder” than B . (To be more precise, A is at most “polynomially harder” than B .) Recall the following lemmas from [Lecture Note 8](#):

Lemma 9.1. *Let A and B be decision problems such that A reduces in polynomial time to B . If B has a polynomial-time algorithm, then A has a polynomial-time algorithm. That is, if $A \leq_P B$ and $B \in P$, then $A \in P$.*

Lemma 9.2. *Let A , B , and C be decision problems. If A reduces in polynomial time to B , and B reduces in polynomial time to C , then A reduces in polynomial time to C .*

That is, if $A \leq_P B$ and $B \leq_P C$, then $A \leq_P C$.

9.3 Reductions between some problems that are new to us

9.3.1 Three similar problems: Clique, Independent Set, and Vertex Cover

Given an undirected graph $G = (V, E)$, a *clique* is a subset $K \subseteq V$ of the vertices such that every pair of vertices in K has an edge in E . An *independent set* is a subset $I \subseteq V$ of the vertices such that no pair of vertices in I has an edge in E . A *vertex cover* is a subset $C \subseteq V$ of the vertices such that every edge in E has an endpoint in C . Here are the three corresponding decision problems.

For each, the input is $G = (V, E)$ and an integer k .

Clique— does G have a clique of size k or more?

Independent Set— does G have an independent set of size k or more?

Vertex Cover— does G have a vertex cover of size k or less?

Each of these three problems reduces in polynomial time to each of the others.

Reducing Clique to Independent Set. Here is a reduction f_1 of Clique to Independent Set. Given an instance $(G = (V, E), k)$ of Clique, define $f_1(G, k)$ to be the instance $(\bar{G}, k)$ of INDEPENDENT SET, where $\bar{G} = (V, \bar{E})$ has the same vertex set of G , and has exactly those edges that G does not have.

To verify correctness, observe that f_1 can be computed in time polynomial in the size of its input, and that, for all instances (G, k) of Clique, the answer to (G, k) is “yes” (G has a clique of size at least k) if and only if the answer to the instance $f_1(G, k)$ of Independent Set is “yes” ($\bar{G}$ has an independent set of size at least k). This is because a set S of vertices is a clique in G iff S is an independent set in $\bar{G}$.

Reducing Independent Set to Clique. The function f_1 also happens to be a poly-time reduction in the other direction: from INDEPENDENT SET to Clique.

Reducing Independent Set to Vertex Cover. Here is a reduction f_2 of Independent Set to Vertex Cover. Given an instance $(G = (V, E), k)$ of Independent Set, define $f_2(G, k)$ to be the instance (G, k') of Vertex Cover, where $k' = |V| - k$. To verify correctness, observe that f_2 can be computed in time polynomial in the size of its input, and that, for all instances (G, k) of Independent Set, the answer to (G, k) is “yes” (G has an independent set of size k) if and only if the answer to the instance $f_2(G, k)$ of Vertex Cover is “yes” (G has a vertex cover of size at most $k' = |V| - k$). This is because a set S of vertices is an independent set in G iff the complement $V \setminus S$ of S is a vertex cover in G .

Reducing Vertex Cover to Independent Set. The function f_2 also happens to be a poly-time reduction in the other direction: from VERTEX COVER to Independent Set.

Reducing Clique to Vertex Cover. And, by Lemma 9.2 (reductions compose), Clique reduces to Vertex Cover via the reduction $(G, k) \mapsto f_2(f_1(G, k))$.

Thus, we see that all three of these problems are equivalent, in the sense that if any one of them has a poly-time algorithm, then (via the reductions described above) each of them will have a poly-time algorithm.

It would be reasonable to conclude at this point that Clique, Independent Set, and Vertex Cover are, in some sense, just different names for the same problem, and that is why we can reduce each of these problem to the other. But the reason is deeper than that. Next we'll see reductions between problems that, on the surface at least, are not at all similar.

9.3.2 A very different (?) problem: 3-CNF-SAT

Here are two problems that seem very different from those three:

SAT —Given a Boolean formula ϕ , is there an assignment of values (true or false) to the variables of ϕ that *satisfies* the formula (makes it true).

3-CNF-SAT— Given a Boolean formula ϕ in 3-CNF form (an “and” of clauses, each of which is an “or” of at most three literals, each of which is a variable or its negation), is the formula *satisfiable* (that is, can one assign values to the variables to make the formula true).

For example, $\phi = (x \vee \bar{y} \vee z) \wedge (y \vee z) \wedge (\bar{y} \vee \bar{z})$ is satisfiable: take $x = T, y = T, z = F$.

We remark that SAT is a very “expressive” problem—computers are built from logic gates, so it is not too surprising that Boolean formulae can model computation. See Lemma 34.6 of the text [CLRS](#) and surrounding discussion for more details. Also, SAT reduces pretty easily to 3-CNF-SAT. So 3-CNF-SAT is just as expressive.

Lemma 9.3. $3\text{-CNF-SAT} \leq_p \text{Independent Set}$.

Proof. Here is the reduction. Let ϕ be any instance of 3-CNF-SAT. Construct the following instance $(G = (V, E), k)$ of Independent Set. For each variable x in ϕ , introduce a “variable gadget” consisting of two new “literal” vertices, labeled x and $\bar{x}$, with an edge between them.

(So far, if there are h variables in ϕ , then there will be 2^h independent sets of size h . Each independent set takes exactly one vertex from each variable gadget, and we have in mind that these 2^h independent sets correspond to the 2^h assignments true/false assignments to the variables of ϕ in the natural way.)

Next, for each clause in ϕ , introduce a “clause gadget” consisting of one new vertex for each literal in the clause, with edges between each pair of vertices within each clause gadget. For example, for the clause $x \vee \bar{y} \vee z$, the clause gadget would consist of three new vertices, labeled, respectively, $x, \bar{y}$, and z , connected by three edges.

(So far, if there are h variables and ℓ clauses in ϕ , each independent set of size $h + \ell$ must contain exactly one vertex from each gadget.)

Finally, for every vertex v that occurs in each clause gadget, introduce an edge between that vertex and the “opposite” vertex in the variable gadget. For example, if v is labeled with the variable x , then introduce an edge between v and the vertex labeled $\bar{x}$ in the variable gadget for x . Or, if v is labeled with $\bar{x}$, then introduce an edge between v and the vertex labeled x in the variable gadget for x .

This defines the graph G . Define the threshold k (the desired independent-set size) to be $h + \ell$ (the number of variables plus clauses, that is, the number of gadgets). This defines the reduction: $f(\phi) = (G, k)$, where G and k are defined from ϕ as described above.

Clearly the reduction can be computed in polynomial time. That is, given ϕ , one can construct G and k in time polynomial in $|\phi|$.

It remains to verify that the reduction f is correct. That is, we need to show

$$(\forall \phi) \quad \phi \text{ is satisfiable} \iff (G, k) \text{ has an independent set of size } k, \text{ where } (G, k) = f(\phi).$$

Above ϕ ranges over all instances of 3-CNF-SAT.

Consider an arbitrary instance ϕ of 3-CNF-SAT.

Let h be the number of variables, let ℓ be the number of clauses. Let $(G, k) = f(\phi)$, so that $k = h + \ell$.

We need to show that ϕ is satisfiable if and only if G has an independent set of size at least k .

Only if ($\implies$). Suppose that ϕ is satisfiable. We need to show that in this case G has an independent set I of size at least k . Fix any assignment A of the variables of ϕ that satisfies ϕ .

We will describe how to construct I from A . Make I contain one vertex from each gadget of G , as follows. For each variable x in ϕ , if A assigns x the value “true”, then make I contain the vertex labeled x from the variable gadget for x in G . Otherwise (A assigns x the value “false”), make I contain the vertex labeled $\bar{x}$ from the variable gadget. For each clause gadget in ϕ , make I contain the vertex from the clause gadget that corresponds to some literal in the clause that A makes true (there must be at least one such literal, since A satisfies the clause).

Clearly I has size $k = h + \ell$. Also, I is an independent set, because it takes only one vertex from each gadget, and (because of the way G is constructed) never takes a clause vertex labeled x and a variable vertex labeled $\bar{x}$, or vice versa.

If ($\Leftarrow$). Suppose that G has an independent set of size at least $k = h + \ell$. We need to show that in this case ϕ has some satisfying assignment A . Let I be any independent set of size k . Clearly I has exactly one vertex per gadget in G (it can't have more than one in any gadget, as each gadget is a clique, so it has to have one in each gadget in order to have size k).

Define assignment A from I as follows. For each variable x , consider the variable gadget for x . If I takes the vertex labeled x , then make A assign x the value “true”. Otherwise (I takes the vertex labeled $\bar{x}$) make A assign x the value “false”. This defines the assignment. To see that it is a satisfying assignment, consider any clause of ϕ . I contains some vertex v from the clause gadget, where v is labeled with a literal in the clause—either a variable x or its negation $\bar{x}$. Consider the case when v is labeled with a variable x (the other case is similar). There is an edge in G between v and the vertex (call it v') labeled $\bar{x}$ in x 's variable gadget. So v' cannot be in the independent set I . So the vertex labeled x in x 's variable gadget has to be in the independent set I . So the assignment assigns x the value “true”. So the clause is satisfied. $\square$

9.4 Review of key concepts

The previous reduction is perhaps surprising. 3-CNF-SAT reduces to (apparently unrelated) graph problems: Independent Set, Vertex Cover, Clique. What's going on? Let's take stock. Here's a review of the key concepts so far:

decision problem —A *decision problem* is a computational problem such that, for every input, the answer is either “yes” or “no”. In studying computational complexity, we work mostly with decision problems, just to keep things simple. Most computational problems can be cast as decision problems.

language —The *language* associated with a decision problem is the set of inputs for which the answer is “yes” (encoded in any standard binary encoding). We often identify the decision problem with its language.

An algorithm A for a decision problem is said to *accept* an input x if, on input x , A outputs “yes”. Otherwise A *rejects* x .

polynomial-time algorithm —An algorithm runs in *polynomial time* if there is some constant $c > 0$ such that the worst-case running time on any input of size n is $O(n^c)$. Note that this includes, for example algorithms that run in time $O(n \log n)$, because $O(n \log n)$ is $O(n^2)$.

the class P —Called *Polynomial Time*, this is the set of decision problems that have polynomial-time algorithms. This is one of many *complexity classes* studied by computer scientists. (A complexity class is a collection of computational problems, typically defined as containing all problems that have an algorithm that obeys a some given resource constraint.)

reduction —A *reduction* from a decision problem A to a decision problem B is a function f mapping each input x for A to some input $f(x)$ for B , such that the answer for x is “yes” if and only if the answer for $f(x)$ is yes. That is, $(\forall x) x \in L(A) \iff f(x) \in L(B)$. When there is a reduction from A to B , we say A *reduces to* B .

polynomial-time reduction —A reduction f is a *polynomial-time* reduction if there is a polynomial-time algorithm that, given any input x , outputs $f(x)$ in time polynomial in $|x|$. If there is such a reduction, we say A *reduces to* B *in polynomial time*, which we write as $A \leq_p B$.

Lemma 9.1. *If $A \leq_P B$ and $B \in P$, then $A \in P$.*

Lemma 9.2. *If $A \leq_P B$ and $B \leq_P C$, then $A \leq_P C$.*

Here are the decision problems we've just considered:

Clique— given graph G and integer k , does G have a clique of size k or more?

Independent Set— Given G and k , does G have an independent set of size k or more?

Vertex Cover— Given G and k , does G have a vertex cover of size k or less?

SAT —Given a Boolean formula ϕ , is there an assignment of values (true or false) to the variables of ϕ that *satisfies* the formula (makes it true).

3-CNF-SAT a.k.a. 3-SAT —SAT restricted to inputs ϕ that are in 3-CNF form (an “and” of clauses, where each clause is an “or” of at most three literals, where each literal is a variable or the negation of a variable).

Here are the reductions we’ve shown:

- $\text{CLIQUE} \leq_p \text{INDEPENDENT SET}$
- $\text{INDEPENDENT SET} \leq_p \text{CLIQUE}$
- $\text{INDEPENDENT SET} \leq_p \text{VERTEX COVER}$
- $\text{VERTEX COVER} \leq_p \text{INDEPENDENT SET}$
- $\text{3-CNF-SAT} \leq_p \text{INDEPENDENT SET (!!!)}$

9.5 The class NP—problems that have poly-time verifiers

These problems all have something in common: they belong to a class of problems called “NP” (for “non-deterministic polynomial time”). We’ll see that NP is defined as the class of decision problems that have polynomial-time *verifiers*. It is defined this way in order to capture a particular set of problems of interest. Intuitively, these are the problems for which, if you are given the correct solution w for a given input x , you can easily *verify* that the solution w is correct for x . (Even if *finding* a correct solution w for a given x is hard!)

Formally, a *verifier* is an algorithm V that is given two inputs: the problem instance x , and a *witness* w . The witness is also called a *certificate* or *proof* (of membership). The verifier V implicitly defines a language L_V , containing all inputs x for which there is a witness that makes the verifier accept:

$$L_V = \{x : (\exists w) V(x, w) \text{ outputs “yes”}\}.$$

The verifier is a *polynomial-time* verifier if $V(x, w)$ (the verifier run on input (x, w)) runs in time polynomial in the size of just x (regardless of the size of w)!

The class NP is defined as the set of decision problems whose languages have polynomial-time verifiers.

By definition, to show that a problem is in NP, you must show that it has a polynomial-time verifier. For example:

Clique —Given the instance $x = (G, k)$ and witness w , the verifier $V((G, k), w)$ checks that w encodes a subset S of the vertices, and that S is a clique in G of size k .

3-CNF-SAT —Given the instance ϕ and witness w , the verifier $V(\phi, w)$ checks that w encodes an assignment to the variables of ϕ , and that the assignment satisfies each clause in ϕ .

The class NP is defined in this way just because the definition captures many interesting problems, problems for which determining whether a solution exists may be hard, but verifying a solution (once you’ve found it) is easy. (Formally, “easy” means “in polynomial time”.)

Note that any problem in P is also in NP, because, given x and w , the verifier $V(x, w)$ can simply ignore the witness w and run the polynomial-time algorithm on x , and say “yes” to (x, w) if the algorithm says “yes” to x .

9.5.1 NP-hard problems; NP-complete problems

Definition 9.1. A decision problem B is NP-hard if, for every problem A in NP, A reduces in polynomial time to B . B is NP-complete if B is in NP and B is NP-hard.

If B is any NP-hard problem, and B has a polynomial-time algorithm, then *every problem in NP* has a polynomial-time algorithm (by reduction to B). So, we know for sure that either

1. no NP-hard problem has a polynomial-time algorithm ($P \neq NP$), or
2. every problem in NP has a polynomial-time algorithm ($P = NP$).

We don't know for sure which case holds—whether or not P equals NP is a main open problem in computer science. *After decades of study, nobody knows a polynomial-time algorithm for any NP-hard problem.* So we expect there are none. But we haven't been able to prove that!

Using NP-hardness. If you have trouble finding a polynomial-time algorithm for a problem, it might be because the problem is NP-hard. To show that a problem is NP-hard, you can reduce some other NP-hard problem to it.

The problems we've considered so far (Clique, Independent Set, Vertex Cover, 3-CNF-SAT) are all NP-complete (although we don't prove it here). So, you can try to reduce one of these problems to your problem.

To finish, we give some more examples of non-trivial reductions, from 3-CNF-SAT to Subset Sum, and from 3-CNF-SAT to a problem called Hamiltonian Path.

9.5.2 3-CNF-SAT reduces to Subset Sum

Recall Subset Sum:

Subset Sum —Given non-negative integers $x_1, x_2, \dots, x_n$ and a target T , is there a subset S of indices such that $\sum_{i \in S} x_i = T$?

(We've seen a dynamic-programming algorithm that runs in $O(nT)$ time. Note that that's not polynomial in the size of the input: the standard representation of the instance takes $O(n \log T)$ bits, so, when T is large (for example $T = 2^n$), time $O(nT)$ is not polynomial in the input size. To run in poly-time, the algorithm would have to run in time polynomial in $n \log T$.)

Let's say you've tried but failed to find a polynomial-time algorithm. So you decide to try to show the problem is NP-hard.

The first step is to choose an NP-hard problem to reduce to your problem. Let's try 3-CNF-SAT.

Given a 3-CNF formula ϕ , your goal is to construct an equivalent instance of Subset Sum: a sequence $x_1, x_2, \dots, x_n$ of numbers and a target T such that some subsequence of the numbers sums to T if and only if ϕ is satisfiable.

To get started, try to invent a “variable gadget” to model the notion that a variable v_i of ϕ has to be assigned “true” and “false”. Here's one way: take the sequence of numbers $x_1 = 1, x_2 = 1$, and total $T = 1$. For this small instance, you have to take exactly one of the two numbers to reach total T . We can interpret taking x_1 as setting variable v_i to true, and taking x_2 as setting variable v_i to false.

Next, try to somehow combine multiple variable gadgets into a single instance. For this, use numbers with multiple digits, using one digit for each variable. For example, combining the gadgets for three variables v_1, v_2, v_3 gives

$$\begin{array}{rcl}
 x_1 & = & 100 \quad (\text{for } v_1) \\
 x_2 & = & 100 \quad (\text{for } \bar{v}_1) \\
 x_3 & = & 010 \quad (\text{for } v_2) \\
 x_4 & = & 010 \quad (\text{for } \bar{v}_2) \\
 x_5 & = & 001 \quad (\text{for } v_3) \\
 x_6 & = & 001 \quad (\text{for } \bar{v}_3) \\
 \hline
 T & = & 111
 \end{array}$$

Above, the numbers are written in decimal. Because each column contains at most two 1's, when we add a subset of the numbers, there will be no carry. Hence, to reach a total of T , you have to take x_1 or x_2 , then x_3 or x_4 , then x_5 or x_6 : there are 8 subsets that sum to T , each representing one of the 8 assignments to the three variables v_1, v_2, v_3 .

Next, how can we model that each clause in ϕ must be satisfied? Consider clause $c_1 = v_1 \vee \bar{v}_2 \vee v_3$. We

can ensure that one literal in this clause will be true by adding a digit for c_1 as follows:

	v_1	v_2	v_3	c_1	
x_1	=	1	0	0	1 (v_1)
x_2	=	1	0	0	0 ($\bar{v}_1$)
x_3	=	0	1	0	0 (v_2)
x_4	=	0	1	0	1 ($\bar{v}_2$)
x_5	=	0	0	1	1 (v_3)
x_6	=	0	0	1	0 ($\bar{v}_3$)
T	=	1	1	1	1

Whereas there were 8 subsets that summed to T before, now only those that choose exactly one of x_1, x_4, x_5 will work. These subsets correspond to assignments that make exactly one of the three literals $v_1, \bar{v}_2, v_3$ true.

This doesn't quite capture the notion of satisfying a clause, because a clause is also satisfied if two or three of its literals are true. To fix this, we add two "filler" numbers for c_1 , and increase the digit in T from 1 to 3:

	v_1	v_2	v_3	c_1	
x_1	=	1	0	0	1 (v_1)
x_2	=	1	0	0	0 ($\bar{v}_1$)
x_3	=	0	1	0	0 (v_2)
x_4	=	0	1	0	1 ($\bar{v}_2$)
x_5	=	0	0	1	1 (v_3)
x_6	=	0	0	1	0 ($\bar{v}_3$)
x_7	=	0	0	0	1 (c_1)
x_8	=	0	0	0	1 (c_1)
T	=	1	1	1	3

We've now captured exactly those assignments that satisfy clause c_1 . For each of the remaining clauses, we add a digit (and a pair of filler numbers). For example, for the second clause $c_2 = \bar{v}_1 \vee v_2$, add a digit and two filler numbers to get

	v_1	v_2	v_3	c_1	c_2	
x_1	=	1	0	0	1	0 (v_1)
x_2	=	1	0	0	0	1 ($\bar{v}_1$)
x_3	=	0	1	0	0	1 (v_2)
x_4	=	0	1	0	1	0 ($\bar{v}_2$)
x_5	=	0	0	1	1	0 (v_3)
x_6	=	0	0	1	0	0 ($\bar{v}_3$)
x_7	=	0	0	0	1	0 (c_1)
x_8	=	0	0	0	1	0 (c_1)
x_9	=	0	0	0	0	1 (c_2)
x_{10}	=	0	0	0	0	1 (c_2)
T	=	1	1	1	3	3

With the ideas in place, we can now prove that 3-CNF-SAT reduces to Subset Sum.

Lemma 9.4. $3\text{-CNF-SAT} \leq_p \text{Subset Sum}$.

Proof. First we define the reduction, then we prove it correct.

The reduction. Consider any instance ϕ of 3-CNF-SAT. Let n be the number of variables in ϕ . Let m be the number of clauses. From ϕ , construct a Subset Sum instance $I = (x, T)$ as follows. Every number in the sequence will have $n + m$ decimal digits, each either 0 or 1. The target T will have $n + m$ decimal digits: n ones followed by m threes.

For each variable v_i , introduce two numbers into the instance I ; denote these numbers $x(v_i)$ and $x(\bar{v}_i)$. Make the i th digits of $x(v_i)$ and $x(\bar{v}_i)$ be 1. For each clause c_j , if literal v_i occurs in c_j , then make the $(n + j)$ 'th digit of $x(v_i)$ be 1; if literal $\bar{v}_i$ occurs in c_j , then make the $(n + j)$ 'th digit of $x(\bar{v}_i)$ be 1. Make the remaining digits 0.

For each clause c_j , introduce two copies of one “filler” number into the instance I ; denote the number $x(c_j)$. Make the $(n + j)$ ’th digit of this number be 1. Make the remaining digits 0.

This completes the description of the Subset Sum instance $I = (x, T)$. It’s easy to see that the reduction takes polynomial time (given ϕ , the collection of numbers and T can be constructed in polynomial time).

Proof of correctness. To finish we prove that the reduction is correct. That is, given any 3-CNF-SAT instance ϕ , the reduction outputs a Subset-Sum instance $(x, T) = f(\phi)$ such that some subsequence of the numbers in x sums to T if and only if ϕ is satisfiable:

$$(\forall \phi) \phi \in \text{3-CNF-SAT} \iff f(\phi) \in \text{Subset Sum}.$$

Consider any 3-CNF-SAT instance ϕ . Let $I = (x, T)$ be the Subset Sum instance produced by the reduction.

Only if ($\implies$). Assume ϕ is satisfiable. We need to show that in this case some subsequence S of x sums to T . Let A be the assignment that satisfies ϕ . Construct a subsequence S from A as follows. For each variable v_i , if A sets v_i to be “true”, then make S contain $x(v_i)$, and otherwise make S contain $x(\bar{v}_i)$.

By inspection of the reduction, the i th digit of the sum of numbers in S will be 1.

For each clause c_j , A gives c_j either one, two, or three distinct true literals. If A gives c_j one true literal, make S contain both copies of $x(c_j)$. If A gives c_j two true literals, make S contain one copy of $x(c_j)$. If A gives c_j three true literals, make S contain neither copy of $x(c_j)$. This guarantees that the $(n + j)$ th digit of the sum of numbers in S is 3. Hence, the sum of the numbers in S equals T .

By this part of the proof, ϕ is satisfiable only if there is a subsequence of x summing to T .

If ($\impliedby$). Suppose some subsequence S of the numbers sums to T . We need to show that ϕ is satisfiable. From S , construct an assignment A to the variables of ϕ as follows.

Each of the numbers in I has $n + m$ decimal digits (each 0 or 1), and, for each i , at most five numbers in I have non-zero i th digit. Hence, in adding the numbers in S , no carry occurs. The first n digits of T are 1’s, and the last m digits are 3’s, so, (i) *for each $i \leq n$, the subsequence S must have exactly one number whose i th digit is 1*, and (ii) *for $n < i \leq n + m$, the subsequence S must have exactly three numbers whose i th digit is 1*.

From (i) and the construction of each $x(v_i)$ and $x(\bar{v}_i)$, it follows that S must have exactly one of the two numbers $x(v_i)$ or $x(\bar{v}_i)$. Define assignment A by assigning v_i to be “true” if S contains $x(v_i)$, and “false” otherwise.

For each clause c_j , there are only two “filler” numbers (copies of $x(c_j)$) in I that have non-zero $(n + j)$ ’th digit. By (ii), there are three numbers in S that have non-zero $(n + j)$ th digit, so there must be some number $x(v_i)$ or $x(\bar{v}_i)$ in S that has non-zero $(n + j)$ ’th digit. In the case that some such $x(v_i)$ is in S , it must be that v_i occurs in clause c_j , so the assignment (which sets v_i true, because $x(v_i) \in S$) satisfies the clause. Otherwise some such $x(\bar{v}_i)$ is in S , and $\bar{v}_i$ occurs in clause c_j , so the assignment (which sets v_i false, because $x(\bar{v}_i) \in S$) satisfies the clause.

By this part of the proof, if there is a subsequence of x summing to T , then ϕ is satisfiable.

By the parts above, for any 3-CNF formula ϕ , the reduction, given ϕ , outputs a subset-sum instance (x, T) such that ϕ is satisfiable if and only if x has a subsequence summing to T . That is, the reduction is correct. $\square$

9.5.3 3-CNF-SAT reduces to Hamiltonian Path

To conclude, we’ll sketch one more reduction: from 3-CNF-SAT to Hamiltonian Path:

Hamiltonian Path —Given directed graph $G = (V, E)$ and $s, t \in V$, is there an s - t path in G that visits each vertex exactly once.

Lemma 9.5. $3\text{-CNF-SAT} \leq_p \text{Hamiltonian Path}$.

Proof sketch. First we describe the reduction, then we prove it is correct.

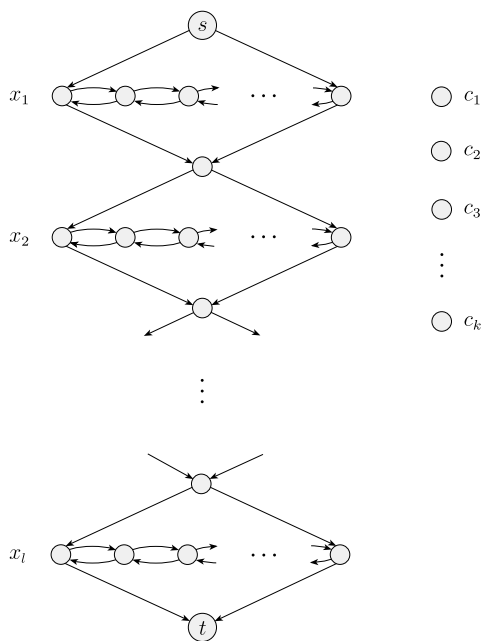


FIGURE 7.49
The high-level structure of G

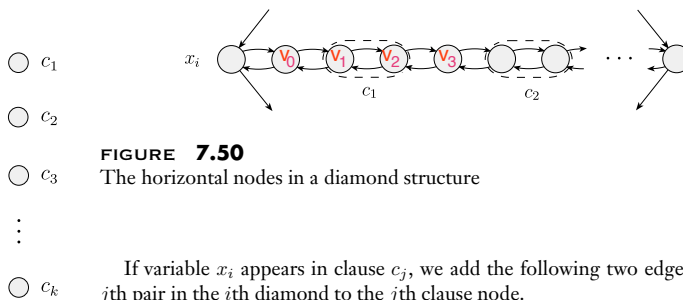


FIGURE 7.50
The horizontal nodes in a diamond structure

If variable x_i appears in clause c_j , we add the following two edges from the j th pair in the i th diamond to the j th clause node.

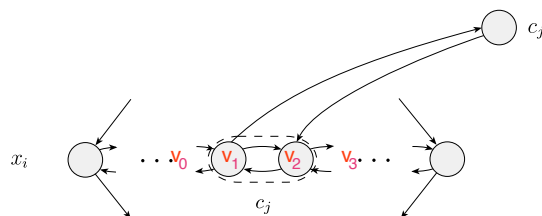


FIGURE 7.51
The additional edges when clause c_j contains x_i

Figure 9.1: Figures 7.49–7.51 of Sipser’s *Introduction to the Theory of Computation*, © Michael Sipser, reproduced here for commentary; they are not covered by these notes’ license.

The reduction. Given any instance ϕ of 3-CNF-SAT, the reduction $f(\phi)$ outputs (G, s, t) where digraph G , source s , and sink t are as shown in Figures 7.49–7.51. (We’ve borrowed these figures from the excellent text *Theory of Computation*, by Sipser.)

The global structure of G is shown in Sipser’s Fig. 7.49. Each variable x_i in ϕ has a variable gadget in G . Each clause in ϕ has a clause gadget. The variable gadget for x_i consists of a long, doubly linked list of vertices with edges from each endpoint of the list to an intermediate vertex below. (See Sipser’s Fig. 7.50.) The vertices of the list come in groups: for each clause c_j there is a pair of adjacent vertices, and between consecutive pairs, and at each end of the list, there is a *separator* vertex, whose only edges are those of the list. The gadgets for the variables $x_1, x_2, \dots, x_n$ are connected in sequence as shown in Fig. 7.49, ending with the sink t .

For each clause c_j , the clause gadget has a single new vertex labeled c_j , and edges as shown in Fig. 7.51. For each literal x or $\bar{x}$ in the clause, we select a unique adjacent pair of vertices (u, w) in the variable gadget for x (where u is to the left of w in the list). If the literal is a variable x , we add edges from u to the clause vertex c_j and from that vertex to w . Otherwise (the literal is a negation, $\bar{x}$), we add edges in the opposite direction: from w to the clause vertex and from the clause vertex to u . In either case, we call the path from u to w (or vice versa) formed by the two edges a *detour*.

This completes the reduction: given any 3-CNF-SAT formula ϕ , the reduction produces the Hamiltonian Path instance $f(\phi) = (G, s, t)$ consisting of the graph G , source s and sink t described above.

Clearly the reduction can be computed in time polynomial in $|\phi|$.

Proof of correctness. To finish, we need to prove that the reduction is correct. That is, we need to show

$$(\forall \phi) \phi \in \text{3-CNF-SAT} \iff f(\phi) \in \text{Hamiltonian Path}.$$

Consider any instance ϕ of 3-CNF-SAT. Let $(G, s, t) = f(\phi)$ as described above.

Only if ($\implies$). Suppose that ϕ is satisfiable. We need to show that, in this case, G has a Hamiltonian path P . Let A be the assignment that satisfies ϕ . Construct P from A as follows.

Start at s . If A assigns x_1 the value “true”, then traverse the variable gadget for x_1 from left to right. Along the way, for each clause c_j that contains the literal x_1 , we will encounter a pair of adjacent vertices u, w (in that order) such that u has an edge to the clause vertex for c_j , which in turn has an edge back to w . Take this “detour” (the two edges) to visit the clause vertex, returning to w and continuing to traverse from left to right.

Or, if A assigns x_1 the value “false”, then traverse the variable gadget for x_1 from *right to left*. Along the way, for each clause c_j that contains the literal $\bar{x}_1$, we will encounter a pair of adjacent vertices u, w (in that order) such that u has an edge to the clause vertex for c_j , which in turn has an edge back to w . Take this “detour” (the two edges) to visit the clause vertex, returning to w and continuing to traverse from right to left.

Continue by, traversing the variable gadgets for x_2, x_3 , etc. similarly: (traverse the gadget for x_i left-to-right if A assigns x_i the value “true”, and otherwise right-to-left; detour to visit clause vertices that contain the corresponding literal, unless the clause vertex has already been visited once). Stop when we reach t . In this way, we can traverse from s to t , and visit each vertex exactly once.

By this part of the proof, ϕ is satisfiable only if G has a Hamiltonian path.

If ($\Leftarrow$). Suppose that G has a Hamiltonian path P from s to t . We need to show that, in this case, ϕ has some satisfying assignment A . Construct A from P as follows.

First, we argue that the path P has to be *well-formed*, meaning that it corresponds to some assignment. Specifically, P must traverse the gadgets for variables $x_1, x_2, \dots, x_n$, in order, and, for each i , must traverse the variable gadget for x_i either left-to-right (with detours to clause gadgets of clauses that contain literal x_i) or right-to-left (with detours to clause gadgets of clauses that contain literal $\bar{x}_i$).

We claim that *if P takes any edge on some detour, then it has to take the other edge as well*.

Next we prove the claim. Note Figures 7.50 and 7.51. Consider any detour (v_1, c_j, v_2) from the gadget for a variable x_i . Assume v_1 is just to the left of v_2 and clause c_j contains literal x_i (the case when v_1 is just to the *right* of v_2 and clause c_j contains literal $\bar{x}_i$ is similar). Let v_0, v_1, v_2, v_3 be the sub-path in the variable gadget around v_1, v_2 .

Vertex v_3 has only one neighbor other than v_2 , so P contains exactly one of the two edges (v_2, v_3) or (v_3, v_2) . In the case that P contains (v_3, v_2) , it also contains (v_2, v_1) (because no other edge leaves v_2). So (i) *P contains either the edge (v_2, v_3) or the sub-path $P_3 = (v_3, v_2, v_1)$.*

Similarly, vertex v_0 has only one neighbor other than v_1 , so P contains exactly one of the two edges (v_0, v_1) or (v_1, v_0) . In the case that P contains (v_1, v_0) , it also contains (v_2, v_1) (because no other edge enters v_1). So (ii) *P contains either the edge (v_0, v_1) or the sub-path $P'_3 = (v_2, v_1, v_0)$.*

First consider the case that P contains (v_2, v_3) . Then P can't contain P'_3 (which has edge (v_2, v_1)), so by (ii) must contain (v_0, v_1) along with (v_2, v_3) . If P also happens to contain (v_1, v_2) , then P can't contain either edge of the detour. Otherwise (P does not contain (v_1, v_2)), P has to contain the first edge of the detour because that is the only way left out of v_1 , and it has to contain the second edge of the detour (because that is the only way left into v_2). So, if P contains (v_2, v_3) , then the claim holds.

Next consider the remaining case, by (i): P contains $P_3 = (v_3, v_2, v_1)$. Then P can't contain (v_0, v_1) (because P contains (v_2, v_1)), so by (ii) must contain $P'_3 = (v_2, v_1, v_0)$ as well as P_3 , so must contain sub-path (v_3, v_2, v_1, v_0) , and cannot contain either edge of the detour. So the claim holds in this case as well. This completes the proof of the claim.

We now finish the proof that ϕ has a satisfying assignment A . Since, for each detour, P contains either both edges of the detour or neither, replacing each detour (v_1, c_j, v_2) from P by the edge (v_1, v_2) gives a path P' that visits all the nodes except the clause nodes. Considering the global structure of G (Fig. 7.49), this path P' must traverse the gadgets for the variables $x_1, x_2, \dots, x_n$ in sequence, and must traverse each gadget either left-to-right or right-to-left. Let A assign the value “true” to each variable x_i whose gadget P' traverses left-to-right. Let A assign “false” to each variable x_i whose gadget P' traverses right-to-left. This is assignment assigns values to all the variables. To see that the assignment has to satisfy ϕ , consider any clause c_j . The original path P traversed this clause, and (by the claim) did so via some detour (v_1, c_j, v_2) from the gadget of a variable x_i such that x_i or $\bar{x}_i$ occurs in clause c_j . Consider the case that x_i occurs in clause c_j . (The case when $\bar{x}_i$ occurs in the clause is similar.) Then, by the construction of G , in the detour v_1 is just to the left of v_2 , which (considering P' and the claim) means that P' must traverse the gadget for x_i left-to-right, which by the way A is constructed that A assigns variable x_i value “true”, so A satisfies

clause c_j .

By this part of the proof, if G has a Hamiltonian path then ϕ is satisfiable.

By the parts of the proof above, for all 3-CNF formulas ϕ , ϕ is satisfiable if and only if $f(\phi)$ has a Hamiltonian path. Therefore, the reduction is correct. $\square$

9.6 External sources on P, NP, and NP-completeness

- DPV Chapter 8. JE Chapter 12. KT Chapter 8 (and KW slides). CLRS Chapter 34. MIT video:
 - <https://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-006-introduction-to-algorithms-fall-2011/lecture-videos/lecture-23-computational-complexity/>
- Chapter 7.3 of Sipser's *Theory of Computation*

Appendix A

Python code

This appendix lists the Python code for the lecture notes, grouped by lecture note. (Each lecture note, on its own, ends with a section listing its code.) Click on a file's name to open the file on GitHub (<https://github.com/nealeyoung/algs101/tree/main/src/code>). Copying code from GitHub is more reliable than copying it from this PDF, since some PDF viewers drop the leading spaces (the indentation) of copied lines.

- Stable Matching (Lecture Note 3)
 - `stable_matching.py`
- Divide and Conquer (Lecture Note 4)
 - `mergesort.py`
 - `closest_pair.py`
 - `selection.py`
 - `local_maximum.py`
- Greedy Algorithms (Lecture Note 5)
 - `library.py`
 - `activity_selection.py`
 - `Huffman.py`
 - `Dijkstra.py`
 - `min_spanning_tree.py`
- Graph Traversal (Lecture Note 6)
 - `graph_library.py`
 - `breadth_first_search.py`
 - `depth_first_search.py`
- Dynamic Programming (Lecture Note 7)
 - `graph_library.py`
 - `dynamic_programming_paths.py`
 - `making_change.py`
 - `subset_sum.py`
 - `knapsack.py`
 - `longest_increasing_subsequence.py`
 - `longest_common_subsequence.py`

- `edit_distance.py`
 - `longest_palindrome.py`
 - `paragraph_layout.py`
 - `min_wt_triangulation.py`
 - `optimal_binary_search_tree.py`
 - `Bellman_Ford.py`
 - `Floyd_Warshall.py`
- Max Flow, Reductions, and Linear Programming (Lecture Note 8)
 - `graph_library.py`
 - `max_flow.py`

A.1 Lecture Note 3: Stable Matching

`stable_matching.py`

```
#!/usr/bin/env python3

from collections.abc import Collection, Iterator, Sequence

# preferences[i] lists, in order of preference, the people that person i ranks
type Preferences = Sequence[Sequence[int]]
type Matching = Collection[tuple[int, int]]

class Person:
    def __init__(self, preferences: Sequence[int]) -> None:
        self.preferences = preferences
        self.ranks = [0] * len(preferences) # allocate array
        for r, p in enumerate(preferences):
            self.ranks[p] = r
        self.partner: int | None = None
        self.partner_iterator = iter(self.preferences)

    def prefers(self, p1: int, p2: int) -> bool:
        return self.ranks[p1] < self.ranks[p2]

def Gale_Shapley(
    hospitals_preferences: Preferences, doctors_preferences: Preferences
) -> set[tuple[int, int]]:
    """The Gale-Shapley algorithm for stable matching"""

    hospitals = tuple(Person(p) for p in hospitals_preferences)
    doctors = tuple(Person(p) for p in doctors_preferences)

    unmatched_hospitals = list(range(len(hospitals)))

    while unmatched_hospitals:
        h = unmatched_hospitals.pop()
        D = doctors[next(hospitals[h].partner_iterator)]

        if D.partner is None:
            D.partner = h
        elif D.prefers(D.partner, h):
            unmatched_hospitals.append(h)
        else:
            unmatched_hospitals.append(D.partner)
            D.partner = h

    return set((D.partner, d) for d, D in enumerate(doctors)) # type: ignore[misc] # all matched

def unstable_pairs(
    hospitals_preferences: Preferences, doctors_preferences: Preferences, M: Matching
) -> Iterator[tuple[int, int]]:
    hospitals = [Person(p) for p in hospitals_preferences]
    doctors = [Person(p) for p in doctors_preferences]

    for h, d in M:
        hospitals[h].partner = d
        doctors[d].partner = h
```

```

    for h, H in enumerate(hospitals):
        for d, D in enumerate(doctors):
            # M is perfect, so H.partner and D.partner aren't None
            if H.prefers(d, H.partner) and D.prefers(h, D.partner): # type: ignore[arg-type]
                yield (h, d)

def is_stable(
    hospitals_preferences: Preferences, doctors_preferences: Preferences, M: Matching
) -> bool:
    return not any(unstable_pairs(hospitals_preferences, doctors_preferences, M))

def exhaustive_search(
    hospitals_preferences: Preferences, doctors_preferences: Preferences
) -> set[tuple[int, int]] | None:
    """Exhaustive search to find a stable matching"""

    from itertools import permutations

    n = len(hospitals_preferences)

    for p in permutations(range(n)):
        M = {(i, p[i]) for i in range(n)}
        if is_stable(hospitals_preferences, doctors_preferences, M):
            return M

    return None

def old_algorithm(
    hospitals_preferences: Preferences, doctors_preferences: Preferences
) -> list[tuple[int, int]]:
    """Old algorithm from LN 3 Fig. 2"""

    n = len(hospitals_preferences)
    assert n == 3

    hospitals = [Person(p) for p in hospitals_preferences]
    doctors = [Person(p) for p in doctors_preferences]

    pairs_by_rank: list[list[list[tuple[int, int]]]] = [[] for _ in range(n)] for _ in range(n)]

    for h, H in enumerate(hospitals):
        for d, D in enumerate(doctors):
            i, j = H.ranks[d], D.ranks[h]
            pairs_by_rank[i][j].append((h, d))

    M: list[tuple[int, int]] = []

    for i, j in ((1, 1), (1, 2), (2, 1), (1, 3), (3, 1), (2, 2), (2, 3), (3, 2), (3, 3)):
        for h, d in pairs_by_rank[i - 1][j - 1]:
            H, D = hospitals[h], doctors[d]
            if H.partner is None and D.partner is None:
                M.append((h, d))
                H.partner = d
                D.partner = h

    assert len(M) == 3
    return M

```

```
def test_Gale_Shapley(n: int = 10) -> None:
    from random import sample

    prefs = [[sample(tuple(range(n)), n) for _ in range(n)] for _ in range(2)]

    M = Gale_Shapley(*prefs)
    assert is_stable(*prefs, M) # type: ignore[call-arg, arg-type] # prefs has 2 items

def test_exhaustive_search(n: int = 5) -> None:
    from random import sample

    prefs = [[sample(tuple(range(n)), n) for _ in range(n)] for _ in range(2)]

    M = exhaustive_search(*prefs)
    assert is_stable(*prefs, M) # type: ignore[call-arg, arg-type] # prefs has 2 items

def test_old_algorithm() -> bool:
    from random import sample

    n = 3

    prefs = [[sample(tuple(range(n)), n) for _ in range(n)] for _ in range(2)]

    M = old_algorithm(*prefs)
    if not is_stable(*prefs, M): # type: ignore[call-arg, arg-type] # prefs has 2 items
        return False
    return True

if __name__ == "__main__":
    print(
        "old_algorithm produced",
        len([stable for stable in [test_old_algorithm() for _ in range(100)] if stable]),
        "stable matchings out of 100",
    )

    for n in range(5, 20):
        test_Gale_Shapley(n)
        print(f"Gale_Shapley({n}) passed")

    for n in range(5, 9): # exhaustive search takes time n!, so keep n small
        test_exhaustive_search(n)
        print(f"exhaustive_search({n}) passed")
```

A.2 Lecture Note 4: Divide and Conquer

[divide_and_conquer/mergesort.py](#)

```
#!/usr/bin/env python3

from collections.abc import Iterator
from typing import Any, Protocol

class Ordered(Protocol):
    """a type whose values can be compared with <"""

    def __lt__(self, other: Any, /) -> bool: ...

def mergesort[T: Ordered](A: list[T]) -> list[T]:
    n = len(A)

    if n <= 1:
        return A

    m = int((n + 1) / 2)
    L = mergesort(A[:m])
    R = mergesort(A[m:])
    return list(merged(L, R))

def merged[T: Ordered](L: list[T], R: list[T]) -> Iterator[T]:
    i = j = 0
    while i < len(L) and j < len(R):
        if L[i] < R[j]:
            yield L[i]
            i += 1
        else:
            yield R[j]
            j += 1
    yield from L[i:] + R[j:]

def test(n: int = 1000) -> None:
    import random

    A = [random.randrange(n) for _ in range(n)]
    B = mergesort(A)
    C = sorted(A)
    assert B == C

for _ in range(50):
    test(200)

print("test passed")
```

[divide_and_conquer/closest_pair.py](#)

```
#!/usr/bin/env python3

from collections import namedtuple
```

```

from collections.abc import Iterable
from math import inf

def minf(*args: Iterable[float]) -> float:
    return min(*args, default=inf)

class Point(namedtuple("Point_", "x y")):
    def distance_squared(self, other: "Point") -> float:
        dx, dy = self.x - other.x, self.y - other.y
        return dx**2 + dy**2 # **0.5

def closest_pair(points: Iterable[tuple[float, float]]) -> float:
    """Return the square of the minimum Euclidean distance between any two of the given points"""

    points = list(points)
    P = set(Point(*p) for p in points)
    if len(P) < len(points): # two equal points
        return 0.0
    Px = sorted(P, key=lambda p: p.x)
    Py = sorted(P, key=lambda p: p.y)

    def cp(Px: list[Point], Py: list[Point]) -> float:
        if len(Px) <= 1:
            return inf

        mid = int(len(Px) / 2)
        median_x = Px[mid].x

        Lx, Rx = Px[:mid], Px[mid:]
        L = set(Lx)
        Ly, Ry = [p for p in Py if p in L], [p for p in Py if p not in L]

        delta_L, delta_R = cp(Lx, Ly), cp(Rx, Ry)
        delta = min(delta_L, delta_R)

        My = [p for p in Py if (median_x - p.x) ** 2 < delta]
        delta_M = minf(p.distance_squared(q) for i, p in enumerate(My) for q in My[i + 1 : i + 9])
        return min(delta, delta_M)

    return cp(Px, Py)

def test_closest_pair(n: int = 30) -> None:
    import random

    P = [(2 * i, 2 * j) for i in range(n) for j in range(n)]
    (a, b) = random.choice(P)
    P += [(a + 1, b + 1)]

    assert abs(closest_pair(P) - 2) <= 1e-100

def test_closest_pair_small(n: int = 8) -> None:
    """Compare with brute force on a few random points."""
    import random
    from itertools import combinations

    P = [(random.random(), random.random()) for _ in range(random.randint(2, n))]

```

```

    brute_force = min(Point(*p).distance_squared(Point(*q)) for p, q in combinations(P, 2))
    assert abs(closest_pair(P) - brute_force) <= 1e-9, P

if __name__ == "__main__":
    for _ in range(10):
        test_closest_pair()
    for _ in range(200):
        test_closest_pair_small()

    print("test passed")

```

[divide_and_conquer/selection.py](#)

```

#!/usr/bin/env python3

from typing import Any, Protocol

class Ordered(Protocol):
    """a type whose values can be compared with < and >"""

    def __lt__(self, other: Any, /) -> bool: ...

    def __gt__(self, other: Any, /) -> bool: ...

def median[T: Ordered](A: list[T]) -> T:
    """return the median of the values in A"""

    return select(A, int((len(A) + 1) / 2))

def select[T: Ordered](A: list[T], k: int, randomized: bool = False) -> T:
    """return the kth smallest of the values in A"""

    n = len(A)
    assert 1 <= k <= n, (k, n)

    if randomized:
        if n == 1:
            return A[0]

        import random

        pivot = random.choice(A)

    else:
        if n < 15:
            return sorted(A)[k - 1]

        medians = [median(A[i : i + 5]) for i in range(0, n, 5)]
        pivot = median(medians)

    A1 = [a for a in A if a < pivot]
    A2 = [a for a in A if a == pivot]
    A3 = [a for a in A if a > pivot]

    if k <= len(A1):
        return select(A1, k)

```

```

    k -= len(A1)

    if k <= len(A2):
        return pivot
    k -= len(A2)

    return select(A3, k)

def test(n: int = 1000) -> None:
    import random

    A = [random.randrange(n) for _ in range(n)]
    k = 1 + random.randrange(n)
    kth = select(A, k)
    kth_r = select(A, k, randomized=True)

    B = sorted(A)
    assert B[k - 1] == kth == kth_r

for _ in range(50):
    test(200)

print("test passed")

```

[divide_and_conquer/local_maximum.py](#)

```

#!/usr/bin/env python3

class Subarray:
    def __init__(self, array: list[list[float]], i_range: range, j_range: range) -> None:
        """array is a list of lists; i_range and j_range are ranges"""
        self.array = array
        self.i_range = i_range
        self.j_range = j_range

    def find_nbr_larger_than(self, i: int, j: int) -> tuple[int, int] | None:
        """
        return neighboring cell in subarray with larger value,
        or None if there is no such cell
        """
        for i2, j2 in ((i - 1, j), (i + 1, j), (i, j + 1), (i, j - 1)):
            if i2 in self.i_range and j2 in self.j_range and self.array[i2][j2] > self.array[i][j]:
                return i2, j2
        return None

    def find_local_max(self) -> tuple[int, int]:
        def partition(range_: range) -> tuple[range, int, range]:
            mid = int(len(range_) / 2)
            return range_[[:mid], range_[mid], range_[mid + 1 :]]

        top, mid_i, bottom = partition(self.i_range)
        left, mid_j, right = partition(self.j_range)

        middle_row = set((mid_i, j) for j in self.j_range)
        middle_col = set((i, mid_j) for i in self.i_range)
        cross = sorted(middle_row | middle_col)

```

```

        for i, j in cross:
            if self.find_nbr_larger_than(i, j) is None:
                return i, j

        i, j = max(cross, key=lambda ij: self.array[ij[0]][ij[1]])

        i2, j2 = self.find_nbr_larger_than(i, j)  # type: ignore[misc] # not None, see loop above
        i2_range = top if i2 in top else bottom
        j2_range = left if j2 in left else right

        return Subarray(self.array, i2_range, j2_range).find_local_max()

    @staticmethod
    def from_string(input_: str) -> "Subarray":
        """input_: array as string.  output: array as Subarray"""
        try:
            array = [
                [float(x.strip()) for x in line.strip().split() if x.strip()]
                for line in input_.split("\n")
                if line.strip()
            ]
            if not array:
                raise ValueError
            m, n = len(array), len(array[0])
            if not (m and n and all(len(row) == n for row in array)):
                raise ValueError
            return Subarray(array, range(m), range(n))

        except ValueError as e:
            raise AssertionError("MALFORMED INPUT") from e

if __name__ == "__main__":

    def main(input_: str) -> None:
        A = Subarray.from_string(input_)
        i, j = A.find_local_max()
        if A.find_nbr_larger_than(i, j) is None:
            print("local max:", i, j, "; value:", A.array[i][j])
        else:
            print("NOT LOCAL MAX:", i, j, "; value:", A.array[i][j])

    input_1 = """
    0 1 3
    3 2 3
    2 5 0
    """

    main(input_1)

    input_2 = """
    3 2 2
    2 2 1
    2 1 0
    """

    main(input_2)

```

A.3 Lecture Note 5: Greedy Algorithms

[greedy_algorithms/library.py](#)

```
#!/usr/bin/env python3

from collections.abc import Iterable
from typing import TYPE_CHECKING, Any

if TYPE_CHECKING:
    from _typeshed import SupportsRichComparison

type Edge[Vertex] = tuple[Vertex, Vertex]

def out_neighbor_lists[Vertex](
    V: Iterable[Vertex], E: Iterable[Edge[Vertex]]
) -> dict[Vertex, list[Vertex]]:
    out_neighbors: dict[Vertex, list[Vertex]] = {v: [] for v in V}
    for u, w in E:
        out_neighbors[u].append(w)
    return out_neighbors

def in_neighbor_lists[Vertex](
    V: Iterable[Vertex], E: Iterable[Edge[Vertex]]
) -> dict[Vertex, list[Vertex]]:
    return out_neighbor_lists(V, [(w, u) for u, w in E])

class Heap[T: SupportsRichComparison]:
    """Heap for Dijkstra's and Prim's algorithms"""

    import heapq

    def __init__(self) -> None:
        self.array: list[T] = []

    def insert(self, item: T) -> None:
        self.heapq.heappush(self.array, item)

    def pop_min(self) -> T:
        return self.heapq.heappop(self.array)

    def empty(self) -> bool:
        return not self.array

class DisjointSet:
    """
    Disjoint-Set data structure for Kruskal's algorithm.
    (See https://en.wikipedia.org/wiki/Disjoint-set\_data\_structure.)
    """

    def __init__(self) -> None:
        self.parent: DisjointSet = self
        self.rank = 0

    def find(self) -> "DisjointSet":
        if self.parent == self:
            return self
```

```

        root = self.parent.find()
        self.parent = root
        return root

def union(self, other: "DisjointSet") -> None:
    r1, r2 = self.find(), other.find()
    if r1 == r2:
        return
    # union by rank: hang the root of smaller rank under the other
    if r1.rank < r2.rank:
        r1.parent = r2
    elif r1.rank > r2.rank:
        r2.parent = r1
    else:
        r2.parent = r1
        r1.rank += 1

def random_digraph(
    n: int = 10, m: int | None = None, acyclic: bool = False
) -> tuple[list[int], list[Edge[int]]]:
    """
    return (vertex list, edge set) for random graph
    with n vertices and m edges
    """

    import random

    vertices = list(range(n))

    n_pairs = n * (n - 1)

    if m is None:
        m = int(n_pairs / 2)
    else:
        assert 0 <= m <= n_pairs

    def random_edge_set(n_edges: int) -> set[Edge[int]]:
        if acyclic:
            priority = random.sample(range(n), n)

            edges: set[Edge[int]] = set()
            while len(edges) < n_edges:
                u, w = random.choice(vertices), random.choice(vertices)
                if u != w:
                    if acyclic and priority[u] < priority[w]:
                        u, w = w, u
                    edges.add((u, w))
            return edges

        if m <= n_pairs / 2:
            edges = random_edge_set(m)
        else:
            # if most edges are present, faster to generate the complement
            all_pairs = set((u, w) for u in vertices for w in vertices if u != w)
            assert len(all_pairs) == n_pairs
            edges = all_pairs - random_edge_set(n_pairs - m)

    assert len(vertices) == n
    assert len(edges) == m
    return vertices, list(edges)

```

```
def random_weighted_digraph(
    *args: Any, **kwargs: Any
) -> tuple[list[int], list[Edge[int]], dict[Edge[int], int]]:
    import random

    V, E = random_digraph(*args, **kwargs)
    weights = {e: random.choice(range(10000)) for e in E}
    return V, E, weights

def random_graph(*args: Any, **kwargs: Any) -> tuple[list[int], list[Edge[int]]]:
    # kwargs has no acyclic
    V, E = random_digraph(*args, **kwargs, acyclic=True) # type: ignore[misc]
    E += [(w, u) for u, w in E]
    return V, E

def random_weighted_graph(
    *args: Any, **kwargs: Any
) -> tuple[list[int], list[Edge[int]], dict[Edge[int], int]]:
    V, E, weights = random_weighted_digraph(*args, **kwargs, acyclic=True)
    E += [(w, u) for u, w in E]
    for u, w in E:
        weights[(w, u)] = weights[(u, w)]
    return V, E, weights
```

[greedy_algorithms/activity_selection.py](#)

```
#!/usr/bin/env python3

from collections.abc import Collection, Iterable, Sequence
from typing import NamedTuple

class Interval(NamedTuple):
    start: int
    finish: int

    def starts_after(self, other: "Interval") -> bool:
        return self.start > other.finish

    def empty(self) -> bool:
        return self.start > self.finish

    @staticmethod
    def random_interval(m: int) -> "Interval":
        import random

        return Interval(*sorted([random.randrange(m), random.randrange(m)]))

    @staticmethod
    def random_intervals(n: int, m: int = 0) -> list["Interval"]:
        if not n:
            return []

        if not m:
            m = 2 * n
        assert m > 0
```

```

        I: set[Interval] = set()
        while len(I) < n:
            I.add(Interval.random_interval(m))

        return sorted(I, key=lambda J: J.finish)

def rselect(I: list[Interval]) -> set[Interval]:
    if not I:
        return set()
    J1 = I[0]
    D = [J for J in I if J.starts_after(J1)]
    R = rselect(D)
    return {
        J1,
    } | R

def rselect_efficiently(I: list[Interval]) -> list[Interval]:
    result: list[Interval] = []

    def r(i: int) -> None:
        J1 = I[i]
        result.append(J1)
        for j in range(i + 1, len(I)):
            if I[j].starts_after(J1):
                r(j)
        return

    if I:
        r(0)

    return result

def iselect(I: list[Interval]) -> list[Interval]:
    X: list[Interval] = []
    for J in I:
        if not X or J.starts_after(X[-1]):
            X.append(J)

    return X

if __name__ == "__main__":

    def is_pairwise_disjoint(jobs: Iterable[Interval]) -> bool:
        jobs = sorted(jobs, key=lambda J: J.finish)
        return all(J2.starts_after(J1) for J1, J2 in zip(jobs, jobs[1:]))

    def is_sorted_by_finish_time(jobs: Sequence[Interval]) -> bool:
        return all(J1.finish <= J2.finish for J1, J2 in zip(jobs, jobs[1:]))

    def test() -> None:
        I = Interval.random_intervals(1000)
        assert is_sorted_by_finish_time(I)
        assert all(not J.empty() for J in I)

    solns: list[Collection[Interval]] = []
    for f in (rselect, rselect_efficiently, iselect):

```

```

        S = f(I)
        assert is_pairwise_disjoint(S), (f, S)
        solns.append(S)

    assert all(len(s) == len(solns[0]) for s in solns[1:])
    assert all(set(s) == set(solns[0]) for s in solns[1:])
    print("test passed")

# def test2():
#     I = Interval.random_intervals(10)
#     print(", ".join(f"[{j.start}, {j.finish}]" for j in I))

test()

```

greedy_algorithms/Huffman.py

```

#!/usr/bin/env python3

type Tree = InteriorNode | Leaf

class InteriorNode:
    def __init__(self, left: Tree, right: Tree) -> None:
        self.left = left
        self.right = right
        self.frequency: int = left.frequency + right.frequency
        self.cost: int = self.frequency + left.cost + right.cost

    def split(self, p1: int, p2: int) -> Tree:
        split_left = self.left.split(p1, p2)
        if split_left != self.left:
            return InteriorNode(split_left, self.right)

        split_right = self.right.split(p1, p2)
        if split_right != self.right:
            return InteriorNode(self.left, split_right)

        return self

class Leaf:
    def __init__(self, frequency: int) -> None:
        self.frequency = frequency
        self.cost = 0

    def split(self, p1: int, p2: int) -> Tree:
        if self.frequency == p1 + p2:
            return InteriorNode(Leaf(p1), Leaf(p2))
        else:
            return self

def Huffman(p: list[int]) -> Tree:
    if len(p) == 1:
        return Leaf(p[0])

    p1, p2 = p[:2]

    p_prime = sorted([p1 + p2] + p[2:])
    R = Huffman(p_prime)

```

```

    T = R.split(p1, p2)
    return T

def iHuffman(p: list[int]) -> Tree:
    from collections import deque

    assert all(p1 <= p2 for p1, p2 in zip(p, p[1:]))

    C1 = deque(Leaf(p_i) for p_i in p)
    C2: deque[InteriorNode] = deque()

    def get_next_tree() -> Tree:
        if not C2 or (C1 and C1[0].frequency < C2[0].frequency):
            return C1.popleft()
        return C2.popleft()

    for _ in range(len(p) - 1):
        t1, t2 = get_next_tree(), get_next_tree()
        t = InteriorNode(t1, t2)
        C2.append(t)

    return C2[0] if C2 else C1[0]  # just a leaf if len(p) == 1

if __name__ == "__main__":

    def random_p(n: int, m: int | None = None) -> list[int]:
        import random

        if n <= 0:
            return []

        if not m:
            m = 2 * n

        return sorted(random.randrange(m) for _ in range(n))

    def test() -> None:
        p = random_p(n=100)

        solns: list[Tree] = []
        for f in (Huffman, iHuffman):
            t = f(p)
            solns.append(t)

        assert all(t.cost == solns[0].cost for t in solns[1:])
        print("test passed")

    test()

```

greedy_algorithms/Dijkstra.py

```

#!/usr/bin/env python3

from collections.abc import Iterable, Mapping
from math import inf

from library import Edge, Heap, in_neighbor_lists, out_neighbor_lists

```

```

def Dijkstra[Vertex](
    V: Iterable[Vertex],
    E: Iterable[Edge[Vertex]],
    weights: Mapping[Edge[Vertex], float],
    source: Vertex,
) -> dict[Vertex, float]:
    assert all(w >= 0 for w in weights.values())

    neighbors = out_neighbor_lists(V, E)

    heap: Heap[tuple[float, tuple[Vertex | None, Vertex]]] = Heap() # heap of edges out of K
    d: dict[Vertex, float] = {}

    heap.insert((0, (None, source))) # first iteration: w = source

    while not heap.empty():
        key, (_, w) = heap.pop_min()
        if w not in d:
            d[w] = key
            for x in neighbors[w]:
                if x not in d:
                    key = d[w] + weights[(w, x)]
                    heap.insert((key, (w, x)))

    for v in V:
        if v not in d:
            d[v] = inf

    return d

def check_shortest_path_distances[Vertex](
    V: Iterable[Vertex],
    E: Iterable[Edge[Vertex]],
    weights: Mapping[Edge[Vertex], float],
    source: Vertex,
    d: Mapping[Vertex, float],
) -> bool:
    in_neighbors = in_neighbor_lists(V, E)

    return d[source] == 0 and all(
        d[w] == min((d[u] + weights[(u, w)] for u in in_neighbors[w]), default=inf)
        for w in V
        if w != source
    )

if __name__ == "__main__":
    from library import random_weighted_digraph

    def test(n: int = 10, m: int | None = None) -> None:
        V, E, weights = random_weighted_digraph(n, m)
        source = next(v for v in V)
        d = Dijkstra(V, E, weights, source)
        assert check_shortest_path_distances(V, E, weights, source, d)

    for _ in range(50):
        test(100)

    print("test passed")

```

greedy_algorithms/min_spanning_tree.py

```
#!/usr/bin/env python3

from collections.abc import Collection, Iterable, Mapping
from math import inf

from library import DisjointSet, Edge, Heap, out_neighbor_lists

def Prim[Vertex](
    V: Collection[Vertex], E: Iterable[Edge[Vertex]], weights: Mapping[Edge[Vertex], float]
) -> list[Edge[Vertex]] | None:
    F: list[Edge[Vertex]] = []
    F_vertices: set[Vertex] = set()
    heap: Heap[tuple[float, Vertex, Vertex]] = Heap() # heap of edges out of F

    neighbors = out_neighbor_lists(V, E)

    def add_to_F(u: Vertex) -> None:
        F_vertices.add(u)
        for w in neighbors[u]:
            if w not in F_vertices:
                key = weights[(u, w)]
                heap.insert((key, u, w))

    source = next(v for v in V)
    add_to_F(source)

    while not heap.empty():
        key, u, w = heap.pop_min()
        if w not in F_vertices:
            F.append((u, w))
            add_to_F(w)

    return F if len(F) == len(V) - 1 else None

def Kruskal[Vertex](
    V: Collection[Vertex], E: Iterable[Edge[Vertex]], weights: Mapping[Edge[Vertex], float]
) -> list[Edge[Vertex]] | None:
    F: list[Edge[Vertex]] = []
    subtrees = {v: DisjointSet() for v in V}

    for e in sorted(E, key=lambda e: weights[e]):
        u, w = e
        t_u, t_w = subtrees[u].find(), subtrees[w].find()
        if t_u != t_w:
            F.append(e)
            t_u.union(t_w)

    return F if len(F) == len(V) - 1 else None

if __name__ == "__main__":
    from library import random_weighted_graph

    def test(n: int = 10, m: int | None = None) -> None:
        V, E, weights = random_weighted_graph(n, m)
        F1 = Prim(V, E, weights)
        F2 = Kruskal(V, E, weights)
```

```
def weight(F: list[Edge[int]] | None) -> float:
    return inf if F is None else sum(weights[e] for e in F)

assert weight(F1) == weight(F2)

for _ in range(50):
    test(100)

print("test passed")
```

A.4 Lecture Note 6: Graph Traversal

[graph_traversal/graph_library.py](#)

```
#!/usr/bin/env python3

from collections.abc import Hashable, Iterable

def out_neighbor_lists[T: Hashable](V: Iterable[T], E: Iterable[tuple[T, T]]) -> dict[T, list[T]]:
    out_neighbors: dict[T, list[T]] = {v: [] for v in V}
    for u, w in E:
        out_neighbors[u].append(w)
    return out_neighbors

def in_neighbor_lists[T: Hashable](V: Iterable[T], E: Iterable[tuple[T, T]]) -> dict[T, list[T]]:
    return out_neighbor_lists(V, [(w, u) for u, w in E])

def random_digraph(
    n: int = 10, m: int | None = None, acyclic: bool = False
) -> tuple[list[int], list[tuple[int, int]]]:
    """
    return (vertex list, edge set) for random graph
    with n vertices and m edges
    """

    import random

    vertices = list(range(n))

    n_pairs = n * (n - 1)

    if m is None:
        m = int(n_pairs / 2)
    else:
        assert 0 <= m <= n_pairs

    def random_edge_set(n_edges: int) -> set[tuple[int, int]]:
        if acyclic:
            priority = random.sample(range(n), n)

            edges: set[tuple[int, int]] = set()
            while len(edges) < n_edges:
                u, w = random.choice(vertices), random.choice(vertices)
                if u != w:
                    if acyclic and priority[u] < priority[w]:
                        u, w = w, u
                    edges.add((u, w))
            return edges

        if m <= n_pairs / 2:
            edges = random_edge_set(m)
        else:
            # if most edges are present, faster to generate the complement
            all_pairs = set((u, w) for u in vertices for w in vertices if u != w)
            assert len(all_pairs) == n_pairs
            edges = all_pairs - random_edge_set(n_pairs - m)

    assert len(vertices) == n
```

```

    assert len(edges) == m
    return vertices, list(edges)

def random_graph(n: int = 10, m: int | None = None) -> tuple[list[int], list[tuple[int, int]]]:
    V, E = random_digraph(n, m, acyclic=True)
    E += [(w, u) for u, w in E]
    return V, E

```

graph_traversal/breadth_first_search.py

```

#!/usr/bin/env python3

from collections import deque
from collections.abc import Hashable, Iterable, Mapping
from math import inf

from graph_library import in_neighbor_lists, out_neighbor_lists, random_digraph

def minf(collection: Iterable[float]) -> float:
    return min(collection, default=inf)

def breadth_first_search[T: Hashable](
    V: Iterable[T], E: Iterable[tuple[T, T]], source: T | None = None
) -> dict[T, float]:
    if source is None:
        source = next(v for v in V)

    neighbors = out_neighbor_lists(V, E)

    queue: deque[T] = deque()
    d: dict[T, float] = {source: 0}
    queue.append(source)

    while queue:
        u = queue.popleft()
        for w in neighbors[u]:
            if w not in d:
                d[w] = d[u] + 1
                queue.append(w)

    for v in V:
        if v not in d:
            d[v] = inf

    return d

def check_unweighted_shortest_path_distances[T: Hashable](
    V: Iterable[T], E: Iterable[tuple[T, T]], source: T, d: Mapping[T, float]
) -> bool:
    in_neighbors = in_neighbor_lists(V, E)

    return d[source] == 0 and all(
        d[w] == minf(d[u] + 1 for u in in_neighbors[w]) for w in V if w != source
    )

```

```

if __name__ == "__main__":

    def test_BFS(n: int = 10, m: int | None = None) -> None:
        V, E = random_digraph(n, m)
        source = next(v for v in V)
        d = breadth_first_search(V, E, source)
        assert check_unweighted_shortest_path_distances(V, E, source, d)

    for _ in range(50):
        test_BFS(100)

    print("test passed")

```

[graph_traversal/depth_first_search.py](#)

```

#!/usr/bin/env python3

from collections.abc import Callable, Hashable, Iterable

from graph_library import out_neighbor_lists, random_digraph, random_graph

type Visit[T] = Callable[[T, T | None], object]

def depth_first_search[T: Hashable](
    V: Iterable[T],
    E: Iterable[tuple[T, T]],
    previsit: Visit[T] | None = None,
    postvisit: Visit[T] | None = None,
) -> None:
    neighbors = out_neighbor_lists(V, E)
    visited: set[T] = set()

    def DFS1(u: T, parent: T | None = None) -> None:
        visited.add(u)
        if previsit:
            previsit(u, parent)
        for w in neighbors[u]:
            if w not in visited:
                DFS1(w, u)
        if postvisit:
            postvisit(u, parent)

    for v in V:
        if v not in visited:
            DFS1(v)

def topological_sort[T: Hashable](V: Iterable[T], E: Iterable[tuple[T, T]]) -> list[T] | None:
    """Return the vertices in topological order, or None if the graph has a cycle."""
    E = list(E)
    finished: list[T] = []

    def postvisit(u: T, parent: T | None) -> None:
        finished.append(u)

    depth_first_search(V, E, postvisit=postvisit)

    # G has a cycle iff the DFS has a back edge, i.e., an edge (u, w) with w finishing no earlier
    # than u (for tree, forward and cross edges, w finishes before u; a self-loop is a back edge)

```

```

    finish_time = {u: t for t, u in enumerate(finished)}
    if any(finish_time[w] >= finish_time[u] for u, w in E):
        return None

    return list(reversed(finished))

def test_topological_sort(n: int = 10, m: int | None = None) -> None:
    from random import shuffle

    V, E = random_digraph(n, m, acyclic=True)

    shuffle(V)
    order = topological_sort(V, E)
    assert order is not None

    rank = dict((v, i) for i, v in enumerate(order))

    assert all(rank[u] < rank[w] for u, w in E)

    # adding an edge that closes a cycle (or a self-loop) must be detected
    if E:
        u, w = E[0]
        assert topological_sort(V, E + [(w, u)]) is None
        assert topological_sort(V, E + [(V[0], V[0])]) is None

def count_forward_edges[T: Hashable](V: Iterable[T], E: Iterable[tuple[T, T]]) -> int:
    neighbors = out_neighbor_lists(V, E)
    visited: set[T] = set()
    being_visited: set[T] = set()
    n_forward_edges = 0

    def DFS1(u: T) -> None:
        nonlocal n_forward_edges

        visited.add(u)
        being_visited.add(u)

        for w in neighbors[u]:
            if w not in visited:
                DFS1(w)
            elif w not in being_visited:
                n_forward_edges += 1

        being_visited.remove(u)

    for v in V:
        if v not in visited:
            DFS1(v)

    return n_forward_edges

def count_connected_components[T: Hashable](V: Iterable[T], E: Iterable[tuple[T, T]]) -> int:
    # (V, E) is undirected graph

    n_components = 0

    def previsit(u: T, parent: T | None) -> None:
        nonlocal n_components

```

```

        if parent is None:
            n_components += 1

    depth_first_search(V, E, previsit=previsit)

    return n_components

def test_count_forward_edges(n: int = 10, m: int | None = None) -> None:
    V, E = random_graph(n, m)
    assert n == len(V)

    n_components = count_connected_components(V, E)
    n_tree_edges = n - n_components
    n_forward_edges = count_forward_edges(V, E)

    assert n_tree_edges + n_forward_edges == len(E) / 2

# def strong_components(G):
#     assert isinstance(G, DiGraph)

#     finished = []
#     depth_first_search(G, postvisit=lambda u, _: finished.append(u))

#     G_reversed = DiGraph(vertices=list(reversed(finished)),
#                           edges=set((w, u) for (u, w) in G.edges))

#     components = []

#     def previsit(u, tree_edge):
#         if tree_edge is None:
#             components.append([])
#         components[-1].append(u)

#     depth_first_search(G_reversed, previsit)

#     return components

# class DFS_edge_classifier:
#     def __init__(self, G):
#         self.G = G
#         self.start_times = {}
#         self.finish_times = {}

#         counter = 0

#         def time():
#             nonlocal counter
#             counter += 1
#             return counter

#         self.tree_edges = set()

#         def previsit(u, tree_edge):
#             if tree_edge is not None:
#                 self.tree_edges.add(tree_edge)
#                 self.start_times[u] = time()

#         def postvisit(u, _):

```

```
#         self.finish_times[u] = time()

#         depth_first_search(G, previsit, postvisit)

#         self.forward_edges = set()
#         self.back_edges = set()
#         self.cross_edges = set()

#         for e in G.edges:
#             if e not in self.tree_edges:
#                 u, w = e
#                 s_u, f_u = self.start_times[u], self.finish_times[u]
#                 s_w, f_w = self.start_times[w], self.finish_times[w]
#                 if e == {u, w}:      # undirected edge
#                     assert s_w < s_u < f_u < f_w or s_u < s_w < f_w < f_u
#                     self.forward_edges.add(e)
#                 elif s_u < s_w < f_w < f_u:
#                     self.forward_edges.add(e)
#                 elif s_w < s_u < f_u < f_w:
#                     self.back_edges.add(e)
#                 elif s_w < f_w < s_u < f_u:
#                     self.cross_edges.add(e)
#                 else:
#                     assert False

if __name__ == "__main__":
    for _ in range(5):
        test_topological_sort(10)

    for _ in range(5):
        test_count_forward_edges(10)

    print("test passed")
```

A.5 Lecture Note 7: Dynamic Programming

[dynamic_programming/graph_library.py](#)

```
#!/usr/bin/env python3

from collections.abc import Callable, Hashable, Iterable, Iterator
from typing import Any

type Edge = tuple[int, int]

def out_neighbor_lists[Vertex: Hashable](
    V: Iterable[Vertex], E: Iterable[tuple[Vertex, Vertex]]
) -> dict[Vertex, list[Vertex]]:
    out_neighbors: dict[Vertex, list[Vertex]] = {v: [] for v in V}
    for u, w in E:
        out_neighbors[u].append(w)
    return out_neighbors

def in_neighbor_lists[Vertex: Hashable](
    V: Iterable[Vertex], E: Iterable[tuple[Vertex, Vertex]]
) -> dict[Vertex, list[Vertex]]:
    return out_neighbor_lists(V, [(w, u) for u, w in E])

def random_digraph(
    n: int = 10, m: int | None = None, acyclic: bool = False
) -> tuple[list[int], list[Edge]]:
    """
    return (vertex list, edge set) for random graph
    with n vertices and m edges
    """

    import random

    vertices = list(range(n))

    n_pairs = n * (n - 1)

    if m is None:
        m = int(n_pairs / 2)
    else:
        assert 0 <= m <= n_pairs

    def random_edge_set(n_edges: int) -> set[Edge]:
        if acyclic:
            priority = random.sample(range(n), n)

            edges: set[Edge] = set()
            while len(edges) < n_edges:
                u, w = random.choice(vertices), random.choice(vertices)
                if u != w:
                    if acyclic and priority[u] < priority[w]:
                        u, w = w, u
                    edges.add((u, w))
            return edges

    if m <= n_pairs / 2:
        edges = random_edge_set(m)
```

```

    else:
        # if most edges are present, faster to generate the complement
        all_pairs = set((u, w) for u in vertices for w in vertices if u != w)
        assert len(all_pairs) == n_pairs
        edges = all_pairs - random_edge_set(n_pairs - m)

    assert len(vertices) == n
    assert len(edges) == m
    return vertices, list(edges)

def random_graph(n: int = 10, m: int | None = None) -> tuple[list[int], list[Edge]]:
    V, E = random_digraph(n, m, acyclic=True)
    E += [(w, u) for u, w in E]
    return V, E

def random_weighted_digraph(
    *args: Any, **kwargs: Any
) -> tuple[list[int], list[Edge], dict[Edge, int]]:
    import random

    V, E = random_digraph(*args, **kwargs)
    weights = {e: random.choice(range(10000)) for e in E}
    return V, E, weights

def random_weighted_graph(
    *args: Any, **kwargs: Any
) -> tuple[list[int], list[Edge], dict[Edge, int]]:
    V, E, weights = random_weighted_digraph(*args, **kwargs, acyclic=True)
    E += [(w, u) for u, w in E]
    for u, w in E:
        weights[(w, u)] = weights[(u, w)]
    return V, E, weights

def depth_first_search[Vertex: Hashable](
    V: Iterable[Vertex],
    E: Iterable[tuple[Vertex, Vertex]],
    previsit: Callable[[Vertex, Vertex | None], object] | None = None,
    postvisit: Callable[[Vertex, Vertex | None], object] | None = None,
    source: Vertex | None = None,
) -> None:
    neighbors = out_neighbor_lists(V, E)
    visited: set[Vertex] = set()

    def DFS1(u: Vertex, parent: Vertex | None = None) -> None:
        visited.add(u)
        if previsit:
            previsit(u, parent)
        for w in neighbors[u]:
            if w not in visited:
                DFS1(w, u)
        if postvisit:
            postvisit(u, parent)

    if source is not None:
        DFS1(source)
    else:
        for v in V:

```

```

        if v not in visited:
            DFS1(v)

def breadth_first_search_parents[Vertex: Hashable](
    V: Iterable[Vertex], E: Iterable[tuple[Vertex, Vertex]], source: Vertex | None = None
) -> Iterator[tuple[Vertex, Vertex]]:
    from collections import deque

    if source is None:
        source = next(v for v in V)

    neighbors = out_neighbor_lists(V, E)

    queue: deque[Vertex] = deque()
    queue.append(source)
    visited = {source}

    while queue:
        u = queue.popleft()
        for w in neighbors[u]:
            if w not in visited:
                visited.add(w)
                queue.append(w)
                yield (w, u)

    # for v in V:
    #     if v not in d:
    #         d[v] = inf

    # return d, parents

def topological_sort[Vertex: Hashable](
    V: Iterable[Vertex], E: Iterable[tuple[Vertex, Vertex]]
) -> list[Vertex]:
    finished: list[Vertex] = []

    def postvisit(u: Vertex, parent: Vertex | None) -> None:
        finished.append(u)

    depth_first_search(V, E, postvisit=postvisit)

    return list(reversed(finished))

```

[dynamic_programming/dynamic_programming_paths.py](#)

```

#!/usr/bin/env python3

from collections.abc import Collection, Hashable, Mapping
from functools import cache as memoize
from math import inf
from typing import Any

from graph_library import (
    in_neighbor_lists,
    random_digraph,
    random_weighted_digraph,
    topological_sort,
)

```

```

def count_paths[Vertex: Hashable](
    V: Collection[Vertex],
    E: Collection[tuple[Vertex, Vertex]],
    s: Vertex,
    t: Vertex,
    recursively: bool = False,
) -> int:
    in_neighbors = in_neighbor_lists(V, E)

    N: Any # a memoized function if recursively, else a table (dict)
    if recursively:

        @memoize
        def N(w: Vertex) -> int:
            return 1 if w == s else sum(N(u) for u in in_neighbors[w])

        return N(t)

    else:
        N = {}
        for w in topological_sort(V, E):
            N[w] = 1 if w == s else sum(N[u] for u in in_neighbors[w])
        return N[t]

def test_count_paths(n: int = 10, m: int | None = None) -> None:
    V, E = random_digraph(n, m, acyclic=True)
    n1 = count_paths(V, E, 0, n - 1)
    n2 = count_paths(V, E, 0, n - 1, recursively=True)
    assert n1 == n2

def shortest_path[Vertex: Hashable](
    V: Collection[Vertex],
    E: Collection[tuple[Vertex, Vertex]],
    weights: Mapping[tuple[Vertex, Vertex], float],
    s: Vertex,
    t: Vertex,
    recursively: bool = False,
) -> float:
    in_neighbors = in_neighbor_lists(V, E)

    D: Any # a memoized function if recursively, else a table (dict)
    if recursively:

        @memoize
        def D(w: Vertex) -> float:
            return (
                0 if w == s else min((D(u) + weights[u, w] for u in in_neighbors[w]), default=inf)
            )

        return D(t)

    else:
        D = {}
        for w in topological_sort(V, E):
            D[w] = (
                0 if w == s else min((D[u] + weights[u, w] for u in in_neighbors[w]), default=inf)
            )

```

```

        return D[t]

def longest_path[Vertex: Hashable](
    V: Collection[Vertex],
    E: Collection[tuple[Vertex, Vertex]],
    weights: Mapping[tuple[Vertex, Vertex], float],
    s: Vertex,
    t: Vertex,
    recursively: bool = False,
) -> float:
    in_neighbors = in_neighbor_lists(V, E)

    D: Any # a memoized function if recursively, else a table (dict)
    if recursively:

        @memoize
        def D(w: Vertex) -> float:
            return (
                0 if w == s else max((D(u) + weights[u, w] for u in in_neighbors[w]), default=-inf)
            )

        return D(t)

    else:
        D = {}
        for w in topological_sort(V, E):
            D[w] = (
                0 if w == s else max((D[u] + weights[u, w] for u in in_neighbors[w]), default=-inf)
            )

        return D[t]

def test_shortest_path(n: int = 10, m: int | None = None) -> None:
    V, E, weights = random_weighted_digraph(n, m, acyclic=True)
    d3 = shortest_path(V, E, weights, 0, n - 1)
    d4 = shortest_path(V, E, weights, 0, n - 1, recursively=True)
    assert d3 == d4

def test_longest_path(n: int = 10, m: int | None = None) -> None:
    V, E, weights = random_weighted_digraph(n, m, acyclic=True)
    d1 = longest_path(V, E, weights, 0, n - 1)
    d2 = longest_path(V, E, weights, 0, n - 1, recursively=True)

    weights = {e: -weights[e] for e in weights}
    d3 = shortest_path(V, E, weights, 0, n - 1)
    d4 = shortest_path(V, E, weights, 0, n - 1, recursively=True)

    assert d1 == d2 == -d3 == -d4

def make_change(denominations: Collection[int], target: int, recursively: bool = False) -> float:
    assert isinstance(target, int) and target >= 0
    assert all(isinstance(d, int) and d >= 1 for d in denominations)

    M: Any # a memoized function if recursively, else a table (dict)
    if recursively:

```

```

@memoize
def M(i: int) -> float:
    return (
        0 if i == 0 else min((1 + M(i - d) for d in denominations if d <= i), default=inf)
    )

    return M(target)

else:
    M = {}
    for i in range(target + 1):
        M[i] = (
            0 if i == 0 else min((1 + M[i - d] for d in denominations if d <= i), default=inf)
        )
    return M[target]

def make_change_via_shortest_paths(denominations: Collection[int], target: int) -> float:
    assert isinstance(target, int) and target >= 0
    assert all(isinstance(d, int) and d >= 1 for d in denominations)

    V = range(target + 1)
    E = [(i, i + d) for i in V for d in denominations if i + d in V]
    weights = {e: 1 for e in E}
    return shortest_path(V, E, weights, 0, target)

def test_making_change(n: int = 10) -> None:
    import random

    assert n >= 1
    denominations = set(random.randrange(1, 25) for _ in range(5))
    target = random.randrange(n, 2 * n)
    d1 = make_change(denominations, target)
    d2 = make_change(denominations, target, recursively=True)
    d3 = make_change_via_shortest_paths(denominations, target)

    assert d1 == d2 == d3

if __name__ == "__main__":
    for _ in range(20):
        test_count_paths(50)
        test_shortest_path(50)
        test_longest_path(50)
        test_making_change()

    print("test passed")

```

[dynamic_programming/making_change.py](#)

```

#!/usr/bin/env python3

from collections.abc import Collection
from functools import cache as memoize
from math import inf
from typing import Any

def make_change(D: Collection[int], T: int, recursively: bool = False) -> float:

```

```

assert all(d > 0 and isinstance(d, int) for d in D)
assert T >= 0 and isinstance(T, int)

m: Any # a memoized function if recursively, else a table (dict)
if recursively:

    @memoize
    def m(t: int) -> float:
        """min coins to make change for t"""
        return 0 if t == 0 else min((1 + m(t - d) for d in D if d <= t), default=inf)

    return m(T)
else:
    m = {}
    for t in range(T + 1):
        m[t] = 0 if t == 0 else min((1 + m[t - d] for d in D if d <= t), default=inf)
    return m[T]

def test_make_change(n: int = 10, T: int = 100) -> None:
    import random

    D = [random.randrange(1, 100) for _ in range(n)]
    m1 = make_change(D, T)
    m2 = make_change(D, T, recursively=True)
    assert m1 == m2

if __name__ == "__main__":
    for _ in range(20):
        test_make_change()

    print("test passed")

```

[dynamic_programming/subset_sum.py](#)

```

#!/usr/bin/env python3

from collections.abc import Collection
from functools import cache as memoize
from typing import Any

def subset_sum(X: Collection[int], T: int, recursively: bool = False) -> bool:
    assert all(x >= 0 and isinstance(x, int) for x in X)
    assert T >= 0 and isinstance(T, int)

    X = list(X)
    n = len(X)

    S: Any # a memoized function if recursively, else a table (dict)
    if recursively:

        @memoize
        def S(i: int, t: int) -> bool:
            """is there a subset of the first j items summing to t ?"""
            return (
                True
                if t == 0 and i == 0
                else False
            )
    
```

```

        if t > 0 and i == 0
        else S(i - 1, t) or S(i - 1, t - X[i - 1])
        if i > 0 and X[i - 1] <= t
        else S(i - 1, t)
    )

    return S(n, T)
else:
    S = {}
    for i in range(n + 1):
        for t in range(T + 1):
            S[i, t] = (
                True
                if t == 0 and i == 0
                else False
                if t > 0 and i == 0
                else S[i - 1, t] or S[i - 1, t - X[i - 1]]
                if i > 0 and X[i - 1] <= t
                else S[i - 1, t]
            )
    return S[n, T]

def test_subset_sum(n: int = 10, T: int = 100) -> None:
    import random

    X = [random.randrange(1, 100) for _ in range(n)]
    s1 = subset_sum(X, T)
    s2 = subset_sum(X, T, recursively=True)
    assert s1 == s2

if __name__ == "__main__":
    for _ in range(20):
        test_subset_sum()

    print("test passed")

```

[dynamic_programming/knapsack.py](#)

```

#!/usr/bin/env python3

from collections.abc import Sequence
from functools import cache as memoize
from typing import Any

def knapsack(w: Sequence[int], v: Sequence[float], L: int, recursively: bool = False) -> float:
    assert all(wi > 0 and isinstance(wi, int) for wi in w)
    assert all(vi >= 0 for vi in v)
    assert L >= 0 and isinstance(L, int)
    assert len(w) == len(v)

    n = len(w)

    V: Any # a memoized function if recursively, else a table (dict)
    if recursively:

        @memoize
        def V(i: int, ell: int) -> float:

```

```

        """max value achievable
        using first items {0,1,2,...,i-1}
        and total weight <= ell"""
        return (
            0
            if i == 0 or ell == 0
            else V(i - 1, ell)
            if w[i - 1] > ell
            else max(V(i - 1, ell), V(i - 1, ell - w[i - 1]) + v[i - 1])
        )

    return V(n, L)
else:
    V = {}
    for i in range(n + 1):
        for ell in range(L + 1):
            V[i, ell] = (
                0
                if i == 0 or ell == 0
                else V[i - 1, ell]
                if w[i - 1] > ell
                else max(V[i - 1, ell], V[i - 1, ell - w[i - 1]] + v[i - 1])
            )
    return V[n, L]

def test_knapsack(n: int = 10, L: int = 100) -> None:
    import random

    w = [random.randrange(1, 100) for _ in range(n)]
    v = [random.randrange(0, 100) for _ in range(n)]
    k1 = knapsack(w, v, L)
    k2 = knapsack(w, v, L, recursively=True)
    assert k1 == k2, (w, v, L, k1, k2)

if __name__ == "__main__":
    for _ in range(20):
        test_knapsack()

    print("test passed")

```

[dynamic_programming/longest_increasing_subsequence.py](#)

```

#!/usr/bin/env python3

from collections.abc import Sequence
from functools import cache as memoize

from longest_common_subsequence import longest_common_subsequence

def longest_increasing_subsequence(A: Sequence[float], recursively: bool = False) -> int:
    n = len(A)

    if recursively:
        @memoize
        def LIS(j: int) -> int:

```

```

        return 1 + max((LIS(i) for i in range(j) if A[i] < A[j]), default=0)

    return max((LIS(i) for i in range(n)), default=0)

else:
    L: dict[int, int] = {}
    for j in range(n):
        L[j] = 1 + max((L[i] for i in range(j) if A[i] < A[j]), default=0)
    return max((L[i] for i in range(n)), default=0)

def LIS_via_LCS(A: Sequence[float]) -> float:
    B = sorted(set(A)) # distinct values, so common subsequences are strictly increasing
    return longest_common_subsequence(A, B)

if __name__ == "__main__":

    def test_LIS(n: int = 10) -> None:
        import random

        A = [random.randrange(n) for _ in range(n)]
        L1 = longest_increasing_subsequence(A)
        L2 = longest_increasing_subsequence(A, recursively=True)
        L3 = LIS_via_LCS(A)
        assert L1 == L2 == L3

    for _ in range(20):
        test_LIS(50)

    print("test passed")

```

[dynamic_programming/longest_common_subsequence.py](#)

```

#!/usr/bin/env python3
# mypy: allow-redefinition

from collections.abc import Sequence
from functools import cache as memoize

from dynamic_programming_paths import longest_path

def longest_common_subsequence(
    A: Sequence[object], B: Sequence[object], recursively: bool = False
) -> int:
    m, n = len(A), len(B)

    if recursively:

        @memoize
        def LCS(i: int, j: int) -> int:
            if 0 in {i, j}:
                return 0
            elif A[i - 1] == B[j - 1]:
                return 1 + LCS(i - 1, j - 1)
            else:
                return max(LCS(i, j - 1), LCS(i - 1, j))

```

```

        return LCS(m, n)

    else:
        L: dict[tuple[int, int], int] = {}
        for i in range(m + 1):
            for j in range(n + 1):
                if 0 in {i, j}:
                    L[i, j] = 0
                elif A[i - 1] == B[j - 1]:
                    L[i, j] = 1 + L[i - 1, j - 1]
                else:
                    L[i, j] = max(L[i, j - 1], L[i - 1, j])
        return L[m, n]

def LCS_via_longest_path(A: Sequence[object], B: Sequence[object]) -> float:
    m, n = len(A), len(B)

    vertices = [(i, j) for i in range(m + 1) for j in range(n + 1)]
    edges = (
        [((i - 1, j), (i, j), 0) for i, j in vertices]
        + [((i, j - 1), (i, j), 0) for i, j in vertices]
        + [
            ((i - 1, j - 1), (i, j), 1)
            for i, j in vertices
            if i > 0 and j > 0 and A[i - 1] == B[j - 1]
        ]
    )
    weights = {(u, v): wt for u, v, wt in edges if min(u) >= 0}
    edges = weights.keys()
    return longest_path(vertices, edges, weights, (0, 0), (m, n))

if __name__ == "__main__":

    def test_LCS(m: int = 10, n: int = 10) -> None:
        import random

        A = [random.randrange(m) for _ in range(m)]
        B = [random.randrange(n) for _ in range(n)]
        L1 = longest_common_subsequence(A, B)
        L2 = longest_common_subsequence(A, B, recursively=True)
        L3 = LCS_via_longest_path(A, B)
        assert L1 == L2 == L3

    for _ in range(20):
        test_LCS(50, 40)

    print("test passed")

```

[dynamic_programming/edit_distance.py](#)

```

#!/usr/bin/env python3
# mypy: allow-redefinition

from collections.abc import Sequence
from functools import cache as memoize
from typing import Any

from dynamic_programming_paths import shortest_path

```

```

def edit_distance(A: Sequence[object], B: Sequence[object], recursively: bool = False) -> int:
    m, n = len(A), len(B)

    D: Any # a memoized function if recursively, else a table (dict)
    if recursively:

        @memoize
        def D(i: int, j: int) -> int:
            return (
                max(i, j)
                if i == 0 or j == 0
                else D(i - 1, j - 1)
                if A[i - 1] == B[j - 1]
                else 1 + min(D(i, j - 1), D(i - 1, j), D(i - 1, j - 1))
            )

        return D(m, n)

    else:
        D = {}
        for i in range(m + 1):
            for j in range(n + 1):
                D[i, j] = (
                    max(i, j)
                    if i == 0 or j == 0
                    else D[i - 1, j - 1]
                    if A[i - 1] == B[j - 1]
                    else 1 + min(D[i, j - 1], D[i - 1, j], D[i - 1, j - 1])
                )
        return D[m, n]

def edit_distance_via_shortest_path(A: Sequence[object], B: Sequence[object]) -> float:
    m, n = len(A), len(B)

    vertices = [(i, j) for i in range(m + 1) for j in range(n + 1)]

    def match(i: int, j: int) -> bool:
        return i > 0 and j > 0 and A[i - 1] == B[j - 1]

    edges = (
        [((i - 1, j - 1), (i, j), int(not match(i, j))) for i, j in vertices]
        + [((i - 1, j), (i, j), 1) for i, j in vertices if not match(i, j)]
        + [((i, j - 1), (i, j), 1) for i, j in vertices if not match(i, j)]
    )
    weights = {(u, v): wt for u, v, wt in edges if min(u) >= 0}
    edges = weights.keys()
    return shortest_path(vertices, edges, weights, (0, 0), (m, n))

if __name__ == "__main__":

    def test_edit_distance(m: int = 10, n: int = 10) -> None:
        import random

        A = [random.randrange(m) for _ in range(m)]
        B = [random.randrange(n) for _ in range(n)]
        D1 = edit_distance(A, B)
        D2 = edit_distance(A, B, recursively=True)

```

```

D3 = edit_distance_via_shortest_path(A, B)
assert D1 == D2 == D3, (D1, D2, D3)

for _ in range(15):
    test_edit_distance(30, 40)

A, B = "SNOWY", "SUNNY"
print("edit distance between", A, "and", B, "is", edit_distance(A, B))
print("test passed")

```

dynamic_programming/longest_palindrome.py

```

#!/usr/bin/env python3

from collections.abc import Sequence
from functools import cache as memoize
from typing import Any

def longest_palindromic_subsequence(X: Sequence[object], recursively: bool = False) -> int:
    n = len(X)

    M: Any # a memoized function if recursively, else a table (dict)
    if recursively:

        @memoize
        def M(i: int, j: int) -> int:
            return (
                0
                if j < i
                else 1
                if j == i
                else 2 + M(i + 1, j - 1)
                if j > i and X[i - 1] == X[j - 1]
                else max(M(i, j - 1), M(i + 1, j))
            )

        return M(1, n)

    else:
        M = {}
        for j in range(-1, n + 1):
            for i in range(j + 1, 0, -1):
                M[i, j] = (
                    0
                    if j < i
                    else 1
                    if j == i
                    else 2 + M[i + 1, j - 1]
                    if j > i and X[i - 1] == X[j - 1]
                    else max(M[i, j - 1], M[i + 1, j])
                )
        return M[1, n]

if __name__ == "__main__":

    def test(n: int = 10) -> None:
        import random

```

```

X = [random.randrange(10) for _ in range(n)]
l1 = longest_palindromic_subsequence(X)
l2 = longest_palindromic_subsequence(X, recursively=True)
assert l1 == l2

def brute_force(X: Sequence[object]) -> int:
    from itertools import combinations

    return max(len(c) for r in range(len(X) + 1) for c in combinations(X, r) if c == c[::-1])

def test_small(n: int = 8) -> None:
    import random

    X = [random.randrange(3) for _ in range(random.randrange(n + 1))]
    assert longest_palindromic_subsequence(X) == brute_force(X), X

for _ in range(15):
    test(30)
for _ in range(100):
    test_small()

print("test passed")

```

dynamic_programming/paragraph_layout.py

```

#!/usr/bin/env python3

from collections.abc import Iterator, Sequence
from functools import cache as memoize
from typing import Any

def paragraph_layout(N: int, w: Sequence[str], recursively: bool = False) -> int:
    """return min cost of any width-N layout of w"""

    # note: w is indexed from 0..(n-1) instead of 1..n

    assert all(len(wi) <= N for wi in w)

    n = len(w)

    @memoize
    def width(i: int, j: int) -> int:
        """return width of line with words w[i..j]"""
        assert 0 <= i <= j <= n - 1
        return len(w[i]) + (0 if i == j else 1 + width(i + 1, j))

    def i_range(j: int) -> Iterator[int]:
        """return range of i's s.t. width(i, j) <= N, in time O(j-i)"""
        assert 0 <= j <= n - 1
        i = j
        while i >= 0 and width(i, j) <= N:
            yield i
            i -= 1

    M: Any # a memoized function if recursively, else a table (dict)
    if recursively:

        @memoize
        def M(j: int) -> int:

```

```

        """min cost of layout of w[0...j]"""
        return 0 if j == -1 else min(M(i - 1) + (N - width(i, j)) ** 2 for i in i_range(j))

    return M(n - 1)
else:
    M = {}
    M[-1] = 0
    for j in range(n):
        M[j] = min(M[i - 1] + (N - width(i, j)) ** 2 for i in i_range(j))
    return M[n - 1]

def test_paragraph_layout(N: int = 15, n: int = 10) -> None:
    import random

    alphabet = "abcdefghijklmnopqrstuvwxyz"
    words = [random.choice(alphabet) * random.randrange(2, 6) for _ in range(n)]
    c1 = paragraph_layout(N, words)
    c2 = paragraph_layout(N, words, recursively=True)
    assert c1 == c2

    c1 = paragraph_layout(6, ["aaa", "bb", "cc", "dddd"])
    assert c1 == 14, c1

if __name__ == "__main__":
    for _ in range(20):
        test_paragraph_layout()

    print("test passed")

```

dynamic_programming/min_wt_triangulation.py

```

#!/usr/bin/env python3

from collections.abc import Sequence
from functools import cache as memoize
from typing import Any

def min_wt_triangulation(p: Sequence[tuple[float, float]], recursively: bool = False) -> float:
    # note: p is indexed from 0..(n-1) instead of 1..n

    n = len(p)

    @memoize
    def d(i: int, j: int) -> float:
        return sum((p[i][k] - p[j][k]) ** 2 for k in (0, 1)) ** 0.5

    T: Any # a memoized function if recursively, else a table (dict)
    if recursively:
        @memoize
        def T(i: int, j: int) -> float:
            return (
                d(i, j) if j == i + 1 else min(T(i, k) + T(k, j) + d(i, j) for k in range(i + 1, j))
            )

    return T(0, n - 1)
else:

```

```

T = {}
for j_i in range(1, n):
    for i in range(n - j_i):
        j = i + j_i
        T[i, j] = (
            d(i, j)
            if j == i + 1
            else min(T[i, k] + T[k, j] + d(i, j) for k in range(i + 1, j))
        )
    return T[0, n - 1]

def test_MWT() -> None:
    p1 = ((0, 0), (11, 1), (9, 8), (2, 6))
    p2 = ((0, 0), (1, 0), (1, 1), (0, 1))

    for p in (p1, p2):
        t1 = min_wt_triangulation(p)
        t2 = min_wt_triangulation(p, recursively=True)
        assert t1 == t2

    assert abs((t1 - 4) ** 2 - 2) <= 0.001

if __name__ == "__main__":
    for _ in range(1):
        test_MWT()

    print("test passed")

```

[dynamic_programming/optimal_binary_search_tree.py](#)

```

#!/usr/bin/env python3

from collections.abc import Sequence
from functools import cache as memoize
from typing import Any

def optimal_BST(p: Sequence[float], recursively: bool = False) -> float:
    # note: p is indexed from 0..(n-1) instead of 1..n

    n = len(p)

    @memoize
    def wt(i: int, k: int) -> float:
        """return p[i-1] + p[i] + ... + p[k-1]"""
        return 0 if i > k else p[k - 1] + wt(i, k - 1)

    M: Any # a memoized function if recursively, else a table (dict)
    if recursively:

        @memoize
        def M(i: int, k: int) -> float:
            return (
                0 if i > k else min(wt(i, k) + M(i, j - 1) + M(j + 1, k) for j in range(i, k + 1))
            )

        return M(1, n)
    else:

```

```

    M = {}
    for k in range(n + 1):
        for i in range(k + 1, 0, -1):
            M[i, k] = (
                0
                if i > k
                else min(wt(i, k) + M[i, j - 1] + M[j + 1, k] for j in range(i, k + 1))
            )
    return M[1, n]

def test_BST(n: int = 10) -> None:
    import random

    p = [random.randrange(n) for _ in range(n)]
    t1 = optimal_BST(p)
    t2 = optimal_BST(p, recursively=True)
    assert t1 == t2

if __name__ == "__main__":
    for _ in range(40):
        test_BST()

    print("test passed")

```

[dynamic_programming/Bellman_Ford.py](#)

```

#!/usr/bin/env python3

from collections.abc import Collection, Hashable, Mapping
from functools import cache as memoize
from math import inf
from typing import Any

from graph_library import in_neighbor_lists

def negative_cycles[Vertex: Hashable](
    V: Collection[Vertex],
    E: Collection[tuple[Vertex, Vertex]],
    weights: Mapping[tuple[Vertex, Vertex], float],
    d: Mapping[Vertex, float],
) -> bool:
    return any(d[w] > d[u] + weights[u, w] for (u, w) in E if d[u] != inf)

def check_negative_cycles[Vertex: Hashable](
    V: Collection[Vertex],
    E: Collection[tuple[Vertex, Vertex]],
    weights: Mapping[tuple[Vertex, Vertex], float],
    d: dict[Vertex, float],
) -> dict[Vertex, float] | None:
    return None if negative_cycles(V, E, weights, d) else d

def shortest_paths[Vertex: Hashable](
    V: Collection[Vertex],
    E: Collection[tuple[Vertex, Vertex]],
    weights: Mapping[tuple[Vertex, Vertex], float],

```

```

    s: Vertex,
    recursively: bool = False,
) -> dict[Vertex, float] | None:
    n = len(V)
    in_neighbors = in_neighbor_lists(V, E)

    M: Any # a memoized function if recursively, else a table (dict)
    if recursively:

        @memoize
        def M(w: Vertex, i: int) -> float:
            return (
                0
                if i == 0 and w == s
                else inf
                if i == 0 and w != s
                else min((M(u, i - 1) + weights[u, w] for u in in_neighbors[w]), default=inf)
            )

        d = {w: min(M(w, i) for i in range(n)) for w in V}
        return check_negative_cycles(V, E, weights, d)

    else:
        M = {}
        for i in range(n):
            for w in V:
                M[w, i] = (
                    0
                    if i == 0 and w == s
                    else inf
                    if i == 0 and w != s
                    else min(
                        (M[u, i - 1] + weights[u, w] for u in in_neighbors[w]),
                        default=inf,
                    )
                )

        d = {w: min(M[w, i] for i in range(n)) for w in V}
        return check_negative_cycles(V, E, weights, d)

def Bellman_Ford[Vertex: Hashable](
    V: Collection[Vertex],
    E: Collection[tuple[Vertex, Vertex]],
    weights: Mapping[tuple[Vertex, Vertex], float],
    s: Vertex,
) -> dict[Vertex, float] | None:
    n = len(V)
    in_neighbors = in_neighbor_lists(V, E)

    d: dict[Vertex, float] = {v: inf for v in V}
    d[s] = 0

    for _ in range(n):
        for w in V:
            d[w] = min(d[w], min((d[u] + weights[u, w] for u in in_neighbors[w]), default=inf))

    return check_negative_cycles(V, E, weights, d)

def check_shortest_path_distances[Vertex: Hashable](
    V: Collection[Vertex],

```

```

    E: Collection[tuple[Vertex, Vertex]],
    weights: Mapping[tuple[Vertex, Vertex], float],
    source: Vertex,
    d: Mapping[Vertex, float],
) -> bool:
    in_neighbors = in_neighbor_lists(V, E)

    return d[source] == 0 and all(
        d[w] == min((d[u] + weights[(u, w)] for u in in_neighbors[w]), default=inf)
        for w in V
        if w != source
    )

def test_shortest_paths(n: int, m: int, m_neg: int) -> int:
    import random

    from graph_library import random_weighted_digraph

    V, E, weights = random_weighted_digraph(n, m)
    for e in random.sample(E, m_neg):
        weights[e] *= -1
    source = 0

    d1 = Bellman_Ford(V, E, weights, source)
    d2 = shortest_paths(V, E, weights, source)
    d3 = shortest_paths(V, E, weights, source, recursively=True)

    assert d1 == d2 == d3, (d1, d2, d3)

    if d1:
        assert check_shortest_path_distances(V, E, weights, source, d1)
        return 1
    else:
        return 0

if __name__ == "__main__":
    n = 200
    m = int(n**1.25)
    T = 10
    count = 0
    for t in range(T):
        count += test_shortest_paths(n, m, int(0.15 * n * t / T))

    print("test passed, with", count, "of", T, "having no negative cycle")

```

[dynamic_programming/Floyd_Warshall.py](#)

```

#!/usr/bin/env python3

from collections.abc import Collection, Hashable, Mapping
from functools import cache as memoize
from math import inf
from typing import Any

def Floyd_Warshall[Vertex: Hashable](
    V: Collection[Vertex],
    E: Collection[tuple[Vertex, Vertex]],

```

```

    weights: Mapping[tuple[Vertex, Vertex], float],
    recursively: bool = False,
) -> dict[tuple[Vertex, Vertex], float]:
    n = len(V)
    V = list(V)
    E = set(E)

    M: Any # a memoized function if recursively, else a table (dict)
    if recursively:

        @memoize
        def M(u: Vertex, w: Vertex, i: int) -> float:
            return (
                0
                if u == w
                else weights[u, w]
                if i == 0 and (u, w) in E
                else inf
                if i == 0
                else min(M(u, w, i - 1), M(u, V[i - 1], i - 1) + M(V[i - 1], w, i - 1))
            )

        return {(u, w): M(u, w, n) for u in V for w in V}

    else:
        M = {}

        for i in range(n + 1):
            for u in V:
                for w in V:
                    M[u, w, i] = (
                        0
                        if u == w
                        else weights[u, w]
                        if i == 0 and (u, w) in E
                        else inf
                        if i == 0
                        else min(M[u, w, i - 1], M[u, V[i - 1], i - 1] + M[V[i - 1], w, i - 1])
                    )
        return {(u, w): M[u, w, n] for u in V for w in V}

def test_FW(n: int = 10, m: int | None = None) -> None:
    from graph_library import random_weighted_digraph

    V, E, weights = random_weighted_digraph(n, m)
    d1 = Floyd_Warshall(V, E, weights)
    d2 = Floyd_Warshall(V, E, weights, recursively=True)
    assert d1 == d2

if __name__ == "__main__":
    for _ in range(20):
        test_FW(50)

    print("test passed")

```

A.6 Lecture Note 8: Max Flow, Reductions, and Linear Programming

[graph_library.py](#)

```
#!/usr/bin/env python3

from collections.abc import Callable, Hashable, Iterable, Iterator
from typing import Any

def out_neighbor_lists[T: Hashable](V: Iterable[T], E: Iterable[tuple[T, T]]) -> dict[T, list[T]]:
    out_neighbors: dict[T, list[T]] = {v: [] for v in V}
    for u, w in E:
        out_neighbors[u].append(w)
    return out_neighbors

def in_neighbor_lists[T: Hashable](V: Iterable[T], E: Iterable[tuple[T, T]]) -> dict[T, list[T]]:
    return out_neighbor_lists(V, [(w, u) for u, w in E])

def random_digraph(
    n: int = 10, m: int | None = None, acyclic: bool = False
) -> tuple[list[int], list[tuple[int, int]]]:
    """
    return (vertex list, edge set) for random graph
    with n vertices and m edges
    """

    import random

    vertices = list(range(n))

    n_pairs = n * (n - 1)

    if m is None:
        m = int(n_pairs / 2)
    else:
        assert 0 <= m <= n_pairs

    def random_edge_set(n_edges: int) -> set[tuple[int, int]]:
        if acyclic:
            priority = random.sample(range(n), n)

            edges: set[tuple[int, int]] = set()
            while len(edges) < n_edges:
                u, w = random.choice(vertices), random.choice(vertices)
                if u != w:
                    if acyclic and priority[u] < priority[w]:
                        u, w = w, u
                    edges.add((u, w))
            return edges

    if m <= n_pairs / 2:
        edges = random_edge_set(m)
    else:
        # if most edges are present, faster to generate the complement
        all_pairs = set((u, w) for u in vertices for w in vertices if u != w)
        assert len(all_pairs) == n_pairs
```

```

        edges = all_pairs - random_edge_set(n_pairs - m)

    assert len(vertices) == n
    assert len(edges) == m
    return vertices, list(edges)

def random_graph(n: int = 10, m: int | None = None) -> tuple[list[int], list[tuple[int, int]]]:
    V, E = random_digraph(n, m, acyclic=True)
    E += [(w, u) for u, w in E]
    return V, E

def random_weighted_digraph(
    *args: Any, **kwargs: Any
) -> tuple[list[int], list[tuple[int, int]], dict[tuple[int, int], int]]:
    import random

    V, E = random_digraph(*args, **kwargs)
    weights = {e: random.choice(range(10000)) for e in E}
    return V, E, weights

def random_weighted_graph(
    *args: Any, **kwargs: Any
) -> tuple[list[int], list[tuple[int, int]], dict[tuple[int, int], int]]:
    V, E, weights = random_weighted_digraph(*args, **kwargs, acyclic=True)
    E += [(w, u) for u, w in E]
    for u, w in E:
        weights[(w, u)] = weights[(u, w)]
    return V, E, weights

type Visit[T] = Callable[[T, T | None], object]

def depth_first_search[T: Hashable](
    V: Iterable[T],
    E: Iterable[tuple[T, T]],
    previsit: Visit[T] | None = None,
    postvisit: Visit[T] | None = None,
    source: T | None = None,
) -> None:
    neighbors = out_neighbor_lists(V, E)
    visited: set[T] = set()

    def DFS1(u: T, parent: T | None = None) -> None:
        visited.add(u)
        if previsit:
            previsit(u, parent)
        for w in neighbors[u]:
            if w not in visited:
                DFS1(w, u)
        if postvisit:
            postvisit(u, parent)

    if source is not None:
        DFS1(source)
    else:
        for v in V:
            if v not in visited:

```

```

DFS1(v)

def breadth_first_search_parents[T: Hashable](
    V: Iterable[T], E: Iterable[tuple[T, T]], source: T | None = None
) -> Iterator[tuple[T, T]]:
    from collections import deque

    if source is None:
        source = next(v for v in V)

    neighbors = out_neighbor_lists(V, E)

    queue: deque[T] = deque()
    queue.append(source)
    visited = {source}

    while queue:
        u = queue.popleft()
        for w in neighbors[u]:
            if w not in visited:
                visited.add(w)
                queue.append(w)
                yield (w, u)

    # for v in V:
    #     if v not in d:
    #         d[v] = inf

    # return d, parents

def topological_sort[T: Hashable](V: Iterable[T], E: Iterable[tuple[T, T]]) -> list[T]:
    finished: list[T] = []

    def postvisit(u: T, parent: T | None) -> None:
        finished.append(u)

    depth_first_search(V, E, postvisit=postvisit)

    return list(reversed(finished))

```

max_flow.py

```

#!/usr/bin/env python3

from collections.abc import Hashable, Mapping, Set
from typing import Any

from graph_library import (
    breadth_first_search_parents,
    depth_first_search,
    in_neighbor_lists,
    out_neighbor_lists,
)

def Ford_Fulkerson[T: Hashable](
    V: Set[T],
    E: Set[tuple[T, T]],

```

```

caps: Mapping[tuple[T, T], int],
s: T,
t: T,
use_BFS: bool = False,
) -> dict[tuple[T, T], int]:
    assert all((w, u) not in E for (u, w) in E)

    f = {(u, w): 0 for (u, w) in E}

    def cap(u: T, w: T) -> int:
        return caps[u, w]

    def residual_cap(u: T, w: T) -> int:
        return cap(u, w) - f[u, w] if (u, w) in E else f[w, u]

    while True:
        Ef = set((u, w) for (u, w) in E if f[u, w] < cap(u, w)) | set(
            (w, u) for (u, w) in E if f[u, w] > 0
        )

        if not use_BFS:
            parents: dict[T, T | None] = {}

            def previsit(w: T, u: T | None) -> None:
                if u is not None:
                    parents[w] = u # noqa: B023 (previsit is used only in this iteration)

            depth_first_search(V, Ef, previsit=previsit, source=s)
        else:
            parents = dict(breadth_first_search_parents(V, Ef, s))

        if t not in parents:
            break

        path: list[Any] = [t] # a list of vertices, then (below) of edges
        while path[-1] in parents:
            path.append(parents[path[-1]])
        assert path[-1] == s

        path = list(reversed(path))
        path = list(zip(path, path[1:])) # list of vertices -> list of edges

        delta = min(residual_cap(u, w) for (u, w) in path)
        for u, w in path:
            if (u, w) in E:
                f[u, w] += delta
            else:
                f[w, u] -= delta

        assert delta > 0

# done. verify then return f

# verify flow value is cut capacity

in_neighbors = in_neighbor_lists(V, E)
out_neighbors = out_neighbor_lists(V, E)

parents[s] = None

value = sum(f[s, w] for w in out_neighbors[s])

```

```
cut_capacity = sum(cap(u, w) for (u, w) in E if u in parents and w not in parents)
assert value == cut_capacity

# verify capacity constraints

assert all(0 <= f[u, w] <= cap(u, w) for u, w in E)

# verify conservation constraints

assert all(
    sum(f[u, v] for u in in_neighbors[v]) == sum(f[v, w] for w in out_neighbors[v])
    for v in V - {s, t}
)
return f

def Edmonds_Karp[T: Hashable](
    V: Set[T], E: Set[tuple[T, T]], capacities: Mapping[tuple[T, T], int], s: T, t: T
) -> dict[tuple[T, T], int]:
    return Ford_Fulkerson(V, E, capacities, s, t, use_BFS=True)

def test_max_flow() -> None:
    s, a, b, t = "sabt"
    V = {s, a, b, t}
    capacities = {(s, a): 9, (s, b): 5, (a, b): 5, (a, t): 6, (b, t): 7}
    E = capacities.keys()

    f1 = Ford_Fulkerson(V, E, capacities, s, t)
    f2 = Edmonds_Karp(V, E, capacities, s, t)
    value1 = sum(f1[s, w] for (u, w) in E if u == s)
    value2 = sum(f2[s, w] for (u, w) in E if u == s)

    assert value1 == value2 == 13

if __name__ == "__main__":
    test_max_flow()

    print("test passed")
```

Appendix B

Homework assignments

The homework assignments are available at <https://nealeyoung.github.io/algs101/>. For how to use LaTeX to prepare the homeworks, see Section B.12, at the end of this appendix. For each homework, listed below are the topics (lecture notes) it covers and the topic of each of its problems, then links to the assignment (PDF), to a LaTeX template for writing up solutions (a zip file), and to the template opened in Overleaf.

B.1 Preliminaries, Long-Form Proofs, Stable Matching

1. Asymptotic (Θ) notation
2. Proof about big-O
3. Proof by induction (a mysterious recursive function)
4. An instance with only one stable matching
5. Checking a proof about the 1951 algorithm

[PDF](#), [template](#), [Overleaf](#)

B.2 Preliminaries, Stable Matching, Divide and Conquer

1. Ordering functions by growth rate (big-O)
2. Bounding sums (geometric sums)
3. Recursion trees
4. Recursion trees: a variant of mergesort
5. Busy hospitals (prove or disprove)
6. Lonely doctor (prove or disprove)

[PDF](#), [template](#), [Overleaf](#)

B.3 Divide and Conquer

1. Recurrences
2. 2D Local Maximum: an incorrect algorithm
3. Bus routes

[PDF](#), [template](#), [Overleaf](#)

B.4 Greedy Algorithms

1. Activity Selection (simulating rselect and iselect)
2. Making Change: when is the greedy algorithm optimal?
3. Puncturing Intervals: the greedy step

[PDF](#), [template](#), [Overleaf](#)

B.5 Greedy Algorithms, Graph Traversal

1. Puncturing Intervals: the full algorithm
2. Breadth-first search
3. Depth-first search

[PDF](#), [template](#), [Overleaf](#)

B.6 Greedy Algorithms, Graph Traversal

1. Depth-first search and topological sort
2. Prim's algorithm (Minimum Spanning Tree)
3. Minimum Spanning Tree: checking a conjecture and proof
4. Dijkstra's algorithm
5. Street sweeping (a DFS-based algorithm; HackerRank)

[PDF](#), [template](#), [Overleaf](#)

B.7 Dynamic Programming

1. Shortest path in a grid
2. Making Change
3. Counting Ways to Make Change
4. Dijkstra's algorithm: a conjecture

[PDF](#), [template](#), [Overleaf](#)

B.8 Dynamic Programming, Greedy Algorithms

1. Scheduling light and heavy jobs
2. Counting valid subsequences
3. The Game of Piles
4. Minimum Spanning Tree: the reverse-delete (cycleBreaker) algorithm

[PDF](#), [template](#), [Overleaf](#)

B.9 Dynamic Programming, Flow

1. Counting Subset Sums
2. Smallest Subset Sum
3. Three-Way LCS
4. Finding a maximum flow and a minimum cut

[PDF](#), [template](#), [Overleaf](#)

B.10 Flow, Reductions, LP

1. Tiling by bars (via Bipartite Matching)
2. Solving small linear programs
3. Critical edges, or a conjecture about maximum flows

[PDF](#), [template](#), [Overleaf](#)

B.11 Flow, Reductions, P and NP

1. Family seating (via Max Flow)
2. K-Path

[PDF](#), [template](#), [Overleaf](#)

B.12 Using LaTeX for homeworks

LaTeX is a language for making documents (e.g., these notes). We recommend that you prepare each homework in LaTeX, using the template provided with that homework. The template provides most of the LaTeX source that you'll need (you'll just need to replace some placeholders by your answers) as well as guidance and formatting requirements for your solutions.

The homework assignments and their templates are available online (see <https://nealeyoung.github.io/algs101/>). Each template is provided in a zip file (e.g. `homework_1.zip`). The zip archive also contains the macros file and a README file with instructions for that homework. Edit and compile the LaTeX document to produce the PDF.

Compile the LaTeX document as described next (Section B.12.1). You'll need to learn a little bit about how to write your answers in LaTeX, as described in Section B.12.2.

B.12.1 Compiling LaTeX

You have two options for compiling the LaTeX document to produce the PDF document:

Option 1: Online. This is the easiest way to start. Go to <https://www.overleaf.com>, register an account and create a new project by uploading the `homework_1.zip` file (see https://www.overleaf.com/learn/how-to/Uploading_a_project), or use the homework's "Open in Overleaf" link. Edit the document (replace the placeholders by your answers), then compile the document and download the PDF.

Option 2: On your computer

Mac: Install MacTeX (<https://tug.org/mactex/mactex-download.html>). Download the `homework_1.zip` file and unzip it into a directory of your choosing. If you like, use a text editor to edit the files and the shell to compile them (e.g. with command "pdflatex homework_1.tex"). Alternatively, install and use a GUI frontend such as TeXShop (<https://pages.uoregon.edu/koch/texshop/>).

Windows: 1. Install a LaTeX compiler (e.g. MiKTeX, <https://miktex.org/howto/install-miktex>).
 2. Install the GUI called TeXnic (<https://www.texniccenter.org>)
 3. Download the `homework_1.zip` file and unzip it into a directory of your choosing.
 4. Start the TeXnic application. (The first time you start it, the configuration wizard will open. Click on Yes, Configure TeXnicCenter for use with MiKTeX, then finish the configuration. Immediately select the Output Profile dropdown in the toolbar and change to LaTeX → PDF.)
 5. Open the LaTeX file(s) in TeXnic and edit them per the instructions in the README file.
 6. Under Build, click on Build, Current File, Build and View. Or key in Ctrl + Shift + F5.
 7. The PDF will show up on the screen when the build is complete.
 8. Find the PDF in the folder with your tex file.

See also <https://texniccenter.sourceforge.net/configuration.html>.

Linux: 1. Get `install-tl-unx.tar.gz` at <https://www.tug.org/texlive/acquire-netinstall.html>.
 2. Untar/gunzip the file to a directory of your choice. Cd to the directory in your terminal.
 3. Run the command "sudo ./install-tl" in that directory.
 4. The default installation requires 6 GB. If you have that, install it by responding "I" to the "Enter Command" prompt, otherwise you can enter "C" to configure which packages you want. You need at least the packages that the macros file uses (and those that these packages depend on).
 5. Use an editor of your choice (e.g. Emacs, perhaps with the emacs auctex package) to edit the LaTeX source file (.tex) file, then compile to PDF using e.g. the command "pdflatex homework_1.tex".

You can find more about installing LaTeX here: <https://www.tug.org/texlive/>

LaTeX source	result
<pre> \documentclass[11pt]{article} \input{macros} % load file named "macros.tex" \begin{document} % start the body of the document This is the first paragraph. % a comment It has two sentences. Here is some inline math: \((e^{i\pi} = 1)\), followed by some displayed math: \[\sum_{i=1}^n i = n(n+1)/2 .\] and some math symbols: \\ \(\forall, \exists, \leq, \geq, <, >, =\) . \end{document} </pre>	This is the first paragraph. It has two sentences. Here is some inline math: $e^{i\pi} - 1 = 0$, followed by some displayed math: $\sum_{i=1}^n i = n(n+1)/2.$ and some math symbols: $\forall, \exists, \leq, \geq, <, >, =$.

Figure B.1: A minimal LaTeX document with preamble and body. The homework LaTeX template will already have the preamble and most of the body, with placeholders for your answers. You just need to replace those placeholders with your answers. To do that you'll need to know how to do math notation in LaTeX, and how to use the macros for pseudo-code and long-form proofs.

B.12.2 Learning how to write LaTeX

For a quick start, download and compile the homework template document (see Section B.12.1) and replace the placeholders there with your answers. You'll need to learn how to do math notation:

<https://artofproblemsolving.com/wiki/index.php/LaTeX:Math>

<https://artofproblemsolving.com/wiki/index.php/LaTeX:Symbols>

Read Section B.12.3 below and Section 2.4 about using the LaTeX macros to write pseudo-code and long-form proofs. In case you are interested, Figure B.1 shows a minimal LaTeX document, and more detailed LaTeX documentation is available (see footnote).¹

B.12.3 Writing pseudo-code in LaTeX

For the homeworks, you'll need to write pseudo-code when you are defining an algorithm. You can do this using macros that are predefined for you in the homework template, as shown in Figure B.2 (the LaTeX source for the pseudo-code in Figure 1.3), and Figure B.4 (Figure B.3 shows the result). That example includes a while loop with a nested code block. LaTeX numbers the lines for you, which makes it easy to refer to specific steps in the algorithm. You can refer to these in your explanation and analysis of the algorithm, and readers can refer to them in their feedback.

B.12.4 Writing proofs in LaTeX (see Lecture Note 2)

Lecture Note 2 describes long-form proofs and how to write them in LaTeX using the macros.

¹<https://mirrors.ctan.org/info/latex-essential/ess2e.pdf> (17 pages)
<https://www.andy-roberts.net/writing/latex>
<https://tug.ctan.org/info/lshort/english/lshort.pdf> (139 pages)
<https://www.latex-project.org/help/documentation>

```

\begin{myframed}
\begin{steps}

\item[] \textsf{closest-pair}$(P=(p_1, p_2, \ldots, p_n))$
\comment{($p_i=(x_i, y_i)\in \mathbb{R}^2$, with $x_1 < x_2 < \cdots < x_n$)}

\step if $n \le 1$: return $\infty$
\step if $n = 2$: return $d(p_1, p_2)$

\step let $m = \lfloor n/2 \rfloor$

\step let $L = (p_1, p_2, \ldots, p_m)$; let $R = (p_{m+1}, \ldots, p_n)$
\comment{(partition $P$ into left and right halves by x-coordinate)}

\step let $\delta_L = \textsf{closest-pair}(L)$;
let $\delta_R = \textsf{closest-pair}(R)$

\step let $\delta = \min(\delta_L, \delta_R)$

\step let $M = \{p_i : |x_i - x_m| \le \delta\}$
\comment{(compute this in $O(n)$ time)}

\step let $\delta_M =
\min\{d(p_i, p_j) : p_i, p_j \in M, y_i \le y_j \le y_j + \delta\}$,
computed as described in the text

\step return $\min(\delta, \delta_M)$
\end{steps}
\end{myframed}

```

Figure B.2: The LaTeX source (using the provided macros file) for the pseudo-code in Figure 1.3.

Gale-Shapley(n, preference lists) 1. Maintain a matching, initially empty. 2. While there is an unmatched man who has not <i>proposed</i> (as defined below) to every woman: 3.1. Choose any such man m. 3.2. Among women that man m has not yet proposed to, let w be the one he prefers most. 3.3. Man m <i>proposes</i> to woman w, with the following outcome: 3.4. If woman w is unmatched, she <i>accepts</i> the proposal, and m and w become matched. 3.5. Else, if w prefers her current partner, say m', to m, she <i>rejects</i> m, and stays matched to m'. 3.6. Else, w <i>accepts</i>: she rejects m', and becomes matched to m instead of m'. 4. Finally, return the (possibly incomplete) matching formed by the currently matched pairs.	— <i>algorithm for the Stable Matching problem</i>
---	---

Figure B.3: Pseudo-code for the Gale-Shapley algorithm. Figure B.4 shows the LaTeX source.

```

\begin{myframed}
  \begin{steps}

    \item[] {\sf Gale-Shapley}$(n, \text{preference lists})$
    \comment{--- algorithm for the Stable Matching problem}

    \step Maintain a matching, initially empty.
    \step While there is an unmatched man
    who has not \emph{proposed} (as defined below) to every woman:

    \begin{block}

      \step Choose any such man $m$.

      \step\label{S0} Among women that man $m$ has not yet proposed to,
      let $w$ be the one he prefers most.

      \step Man $m$ \emph{proposes} to woman $w$, with the following outcome:

      \step~~~If woman $w$ is unmatched,
      she \emph{accepts} the proposal, and $m$ and $w$ become matched.

      \step~~~Else, if $w$ prefers her current partner, say $m'$, to $m$,
      she \emph{rejects} $m$, and stays matched to $m'$.

      \step~~~Else, $w$ accepts:
      she rejects $m'$, and becomes matched to $m$ instead of $m'$.

    \end{block}

    \step Finally, return the (possibly incomplete) matching
    formed by the currently matched pairs.

  \end{steps}
\end{myframed}

```

Figure B.4: The LaTeX source (using macros from the homework macros file) for the pseudo-code in Figure B.3. References to line numbers can be saved and referred to using the LaTeX `\label` and `\ref` commands. For example, after the `\label{S0}` in the second step of the inner block, the command “`\ref{S0}`” will result in “3.2”.