

Lecture Note 5

Greedy Algorithms

draft of September 29, 2026, by [N. Young](#)

© 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

5.1	Introduction	1
5.2	Making Change	1
5.2.1	Exercises	3
5.3	Activity Selection	4
5.3.1	Exercises	7
5.3.2	External sources on Activity Selection	8
5.4	Covering by Intervals	9
5.4.1	Exercises	11
5.4.2	External sources on Covering by Intervals	12
5.5	Huffman Coding	13
5.5.1	Exercises	16
5.5.2	External sources on Huffman Coding	16
5.6	Interval Partitioning	17
5.6.1	Exercises	18
5.6.2	External sources on Interval Partitioning	18
5.7	Minimizing Total Wait Time	19
5.7.1	Exercises	21
5.7.2	External sources on Minimizing Total Wait Time	21
5.8	Dijkstra’s algorithm for Single-Source Shortest Paths	22
5.8.1	Exercises	26
5.8.2	External sources on Dijkstra’s algorithm	27
5.9	Minimum Spanning Trees	28
5.9.1	Iterative approach (Prim’s algorithm)	28
5.9.2	A recursive view	32
5.9.3	Exercises	34
5.9.4	External sources on Minimum Spanning Tree	35

5.10	Additional Exercises	36
5.10.1	External exercises with solutions	36
5.10.2	Exercises without solutions	36
5.11	Python code	37

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. $\square$

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 $N = 0$, $\text{rchangeUS}(0)$ uses no coins, which is clearly the minimum possible.
- 3.1. Consider any $N \geq 1$, and assume that, for all $N' < N$, $\text{rchangeUS}(N')$ is correct.
- 3.2. Consider the execution of $\text{rchangeUS}(N)$. 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 $\text{rchangeUS}(N)$.
- 3.7. So $\text{rchangeUS}(N)$ outputs an optimal way to make change totaling N .
4. By Block 3, for all $N \geq 1$, $\text{rchangeUS}(N)$ is correct if $\text{rchangeUS}(N')$ is correct for all $N' < N$.
5. By Step 2, $\text{rchangeUS}(0)$ is correct.
6. By the preceding two steps and induction, for all $N \geq 0$, $\text{rchangeUS}(N)$ is correct. $\square$

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

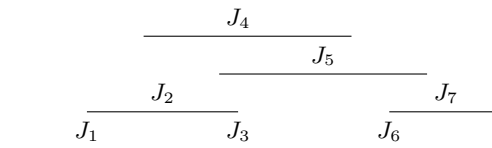
Here is another example where the algorithm makes one greedy choice, then recurses.

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).

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.

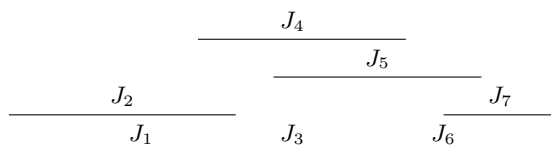
Exercise 5.2 asks you to find a counter-example for each of the first three rules (that is, an input for which the job chosen by the rule is not in any optimal solution). To build intuition, review that exercise and its solution. But what about the fourth rule—choose the *earliest-ending* job, J_1 ? 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.*

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?

²KT call this “greedy stays ahead”.

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:

```

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 $\text{rselect}(I)$ is correct in this case.

For the induction step, consider the execution of $\text{rselect}(I)$ 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 $\text{rselect}(D)$ is correct. So R is a correct solution for D . By Lemma 5.5, the solution $\{J_1\} \cup R$ returned by $\text{rselect}(I)$ 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( $I = \{J_1, J_2, \dots, J_n\}$ ):
```

— assumes jobs are ordered by end time

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

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( $I = \{J_1, J_2, \dots, J_n\}$ ):
```

— assumes jobs are ordered by end time

1. initialize $X = \emptyset$ and $\ell = -\infty$ *— maintain ℓ as max end time of any job in X*
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

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 5.11).

5.3.1 Exercises

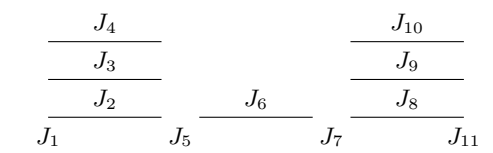
Exercise with solution

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

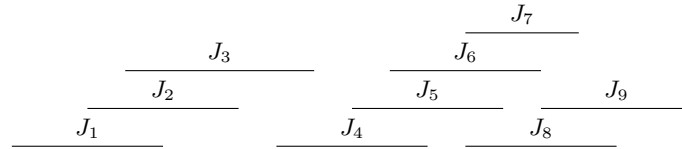
$$S = \{J_1, J_5, J_7, J_{11}\} = \{[0, 2], [3, 6], [7, 10], [11, 13]\},$$

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

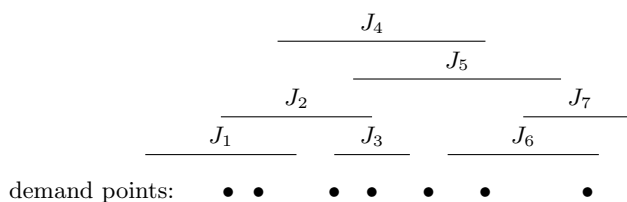
Here is another example where the algorithm makes one greedy choice, then recurses.

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:

`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) . $\square$

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.

$\text{icover}(J = (J_1, J_2, \dots, J_m), X = (x_1, x_2, \dots, x_n))$: — 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.)

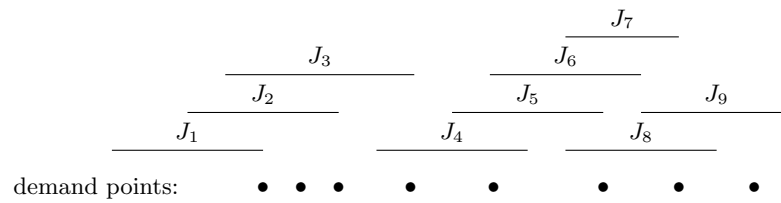
³Assume that each point x_t is in at least one interval. (Otherwise there is no solution.) For each point x_t , define L_t to be the index of the latest-ending job that contains x_t . Note that $L_t \leq L_{t+1}$ for all t , because the points are in increasing order and the jobs are ordered by increasing 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 .

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)

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 5.11).

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

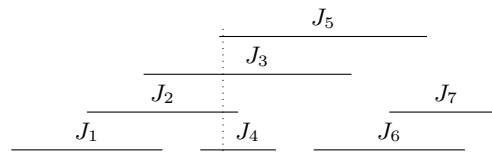
This example does *not* follow the usual “make one choice then recurse” pattern, nor is the design motivated by maintaining the greedy invariant. The algorithm does build the solution iteratively, but we prove optimality of the solution by a combinatorial argument (a matching lower bound).

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. $\square$

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:

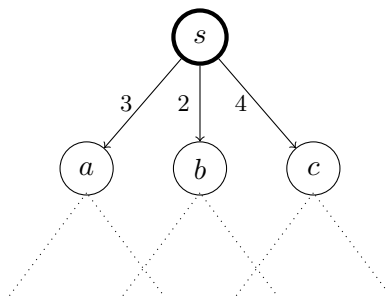
input: Directed graph $G = (V, E)$ with *non-negative* edge weights and source vertex $s \in V$.

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:

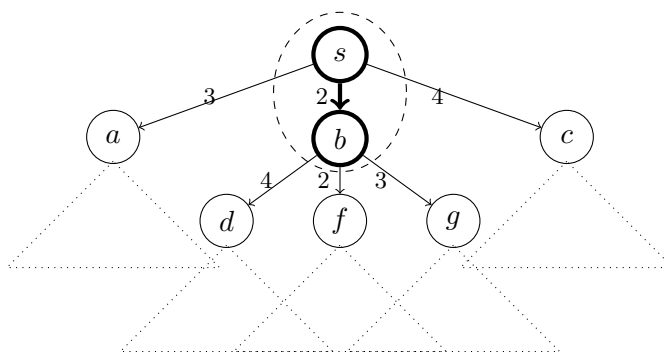
```
greedyShortestPaths( $G = (V, E), s$ ):
1. initialize  $K \leftarrow \{s\}$  and  $d[s] = 0$  ...  $K$  will be the set of vertices whose distances are known
2. until the distance from  $s$  to every vertex is known:
3.1. somehow select a vertex  $w' \in V \setminus K$  note:  $V \setminus K$  is 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. $\square$

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. $\square$

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 $O((m + n) \log n)$. $\square$

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

$$\delta(w) = \min\{d[u] + \text{wt}(u, w) : u \in K, (u, w) \in E\}.$$

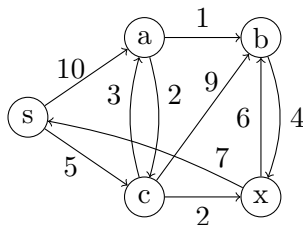
That is, for each $w \notin K$, the algorithm groups all edges into w into a single group, and considers only the best edge (u, w) into w . Then, the implementation maintains a heap of the *vertices* in $V \setminus K$, where each vertex $w \in V \setminus K$ is keyed by $\delta(w)$. In each iteration, 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 5.11).

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 $d[x]$.

⁴a.k.a. a priority queue, see [KT 2.5](#)

Exercise 5.9. (*Dijkstra counter-example*)

- (a) 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.)
- (b) 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.

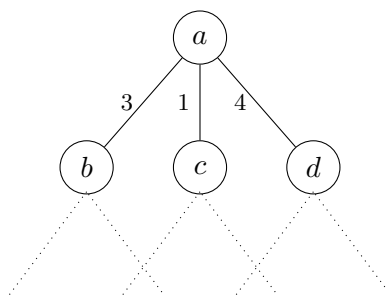
Intuitively, the problem is to connect all vertices in the graph at minimum cost.

[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* be 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') ?

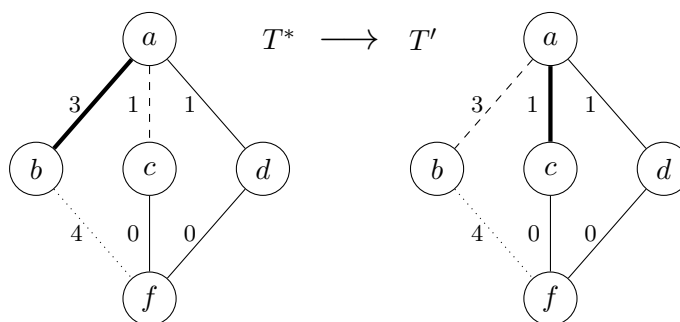
Because T^* is connected and every vertex is in T^* , there has to be at least one edge in T^* into u' . Let (u', w^*) be an edge in T^* incident to u' . What if we construct T' from T^* by removing (u', w^*) and replacing it by (u', w') ? That is, let $T' = \{(u', w')\} \cup T^* \setminus \{(u', w^*)\}$.

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?

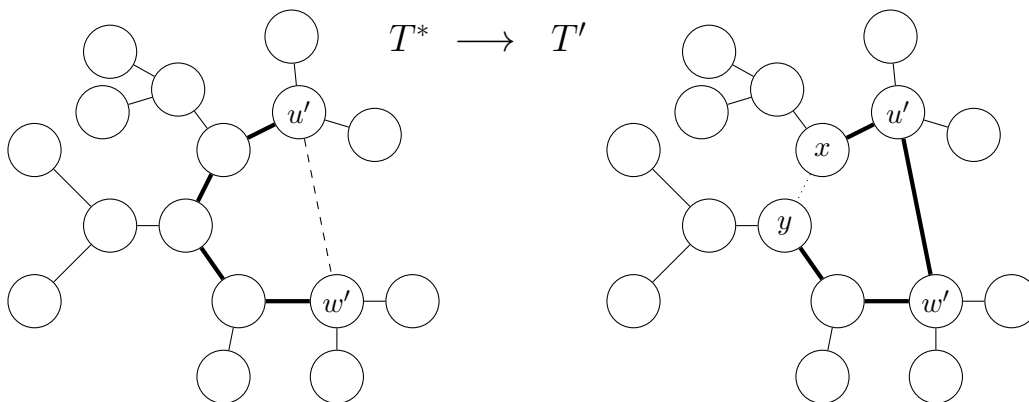
No! Our reasoning has a gap that hides an error. As we've defined T' , it might not be a spanning tree! Consider the 5-vertex graph G shown on the left below. Consider the minimum spanning tree T^* formed by the four solid edges in the graph on the left.



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.

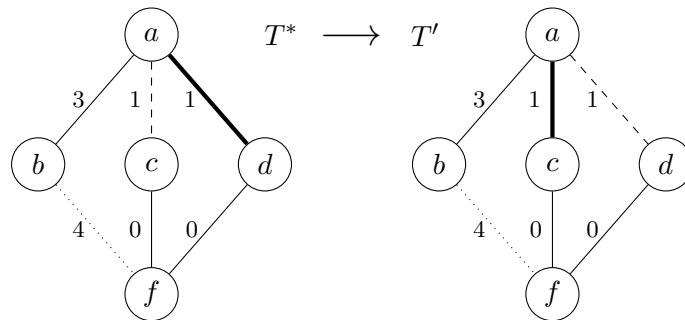
Observation 5.8. Let T^* be any spanning tree of G . Let (u', w') be any edge not in T^* , and let (x, y) be any edge that lies on the path in T^* between u' and w' . Then adding (u', w') to T^* and removing (x, y) gives a spanning tree $T' = \{(u', w')\} \cup T^* \setminus \{(x, y)\}$.



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') . $\square$

⁵And so, in fact, it must be that $\text{wt}(T') = \text{wt}(T^*)$, and also $\text{wt}(u', w') = \text{wt}(u', w^*)$.

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:

Prim($G = (V, E)$): — G is a connected, edge-weighted graph

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

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. □

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 5.11).

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:

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 $d[u] + \text{wt}(u, w)$.

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($G = (V, E)$): — G is a connected, edge-weighted graph

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 $\{(u', w')\} \cup R$

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.

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 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($G = (V, E)$):	— <i>Kruskal's algorithm</i> —
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	<i>but how do we check this?</i>
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 5.11).

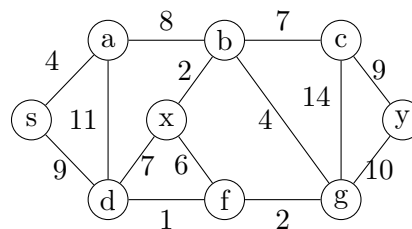
5.9.3 Exercises

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 solutions⁷

- [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.

5.11 Python code

This section lists the Python code for this lecture note. Click on a file's name to open the file on GitHub. 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.

[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()

```

```
#!/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")
```