

Lecture Note 1

Preliminaries

These notes are for an undergraduate course on algorithms, covering the usual topics, but with more of an emphasis on learning to do proofs than is found in most courses. We take the perspective that being able to verify correctness (via detailed proofs) is an integral part of learning to design 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

1.1	External sources to complement these notes	1
1.2	The 4-step recipe for presenting an algorithm	2
1.2.1	How to define an algorithm	3
1.2.2	How to write pseudo-code	3
1.2.3	External sources on how to describe algorithms	5
1.3	Meta-cognition to the rescue	5
1.3.1	Learning algorithms $\equiv$ learning a new language and culture	5
1.3.2	Advice from previous students	8
1.4	Background topics to review	9
1.4.1	Big-O notation definitions and useful rules of thumb	9
1.4.2	Properties of logarithms that are good to know	10
1.4.3	External sources on background material	10
1.5	Exercises on background topics	11
1.5.1	Exercises with solutions	11
1.5.2	Exercises without solutions	11

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

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

1.1 External sources to complement these notes

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

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

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

1.2 The 4-step recipe for presenting an algorithm

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

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

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

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

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

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

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

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

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

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

1.2.1 How to define an algorithm

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

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

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

1.2.2 How to write pseudo-code

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

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

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

```

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

```

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

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

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

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

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

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

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

that come with the homework templates.

1.2.3 External sources on how to describe algorithms

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

1.3 Meta-cognition to the rescue

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

1.3.2 Advice from previous students

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

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

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

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

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

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

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

1.4 Background topics to review

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

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

1.4.1 Big-O notation definitions and useful rules of thumb

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

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

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

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

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

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

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

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

Two rules of thumb for easy cases:

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

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

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

(continued)

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

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

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

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

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

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

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

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

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

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

1.4.2 Properties of logarithms that are good to know

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

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

1.4.3 External sources on background material

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

[Chapter 1: Math Review](#)

[Chapter 3: Proofs](#)

[Chapter 11: Induction](#)

[Chapter 14: Big-O](#)

[Chapter 15: Algorithms](#)

1.5 Exercises on background topics

1.5.1 Exercises with solutions

External problems with solutions

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

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

Solution.

Proof (long form).

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

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

Proof (long form).

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

1.5.2 Exercises without solutions

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

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

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

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

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

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

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

External exercises without solutions²

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

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