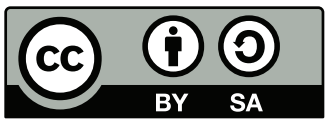
Algorithm
Design
In
Three
Acts

Version 0.9.11 Beta (August 14, 2020)

Kevin A. Wortman



Contents

Chapter 1

Introduction

1.1 The Story of Algorithm Design

This book is an introduction to the design and analysis of computer algorithms. We approach the subject of algorithm design through a dramatic lens, presenting the subject in three acts:

Act I. *Designing Algorithms from Patterns:* We survey six established frameworks for designing algorithms. We explore how to design efficient algorithms that solve a variety of practical computing problems. Times are good.

Act II. *Limitations of Algorithms:* Sadly, we see that our powers of algorithm design have definite limits. Some problems seem to have a "speed limit" (lower bounds), some can only be solved very slowly (NP-hard), and some are quite literally impossible to solve (undecidable). These are mathematical certainties, lending a certain amount of despair. Times are tough.

Act III. *Circumventing Limitations:* While we can't refute complexity results, we can sometimes sidestep them by being flexible in our expectations. Hope is restored.

1.2 Intended Audience and Background

This text is primarily intended to serve as the main textbook for a core undergraduate computer science course on algorithms. It is secondarily intended to be a readable survey for self-directed students.

Undergraduate computer science programs conventionally cover discrete mathematics, programming, and data structures in the first or second years, then require an introductory algorithms course in the second or third year. This book is designed to fit snugly into that sort of curriculum. It assumes that the reader has exposure to the aforementioned subjects, yet is still cultivating their mastery of mathematical reasoning, programming acumen, and technical reading and writing. Our

focus is the design and analysis of algorithms, but we hope that reading the book and doing the exercises will flex your problem solving, programming, and communication muscles.

Prospective students are sometimes anxious about their level of preparation, so we spell out the specific expectations we have of our readers. We expect that students are proficient, though not necessarily experienced, programmers, who know:

- the meanings of, and distinctions between, an algorithm, source code, runnable program, and runtime input;
- how to write a program in an object-oriented programming language such as C++, C#, Java, JavaScript, or Python;
- variables, primitive data types (integer, floating point, Boolean), arithmetic, and Boolean logic;
- control flow: if statements, loops, and variations; and
- fundamental data structures, and when to use them: array, string, linked list, binary search tree, and priority queue.

We expect fluency in discrete mathematics:

- the definitions, properties, and notation of tuples, sets, multisets, and sequences;
- set theory (membership, union, intersect, partition, etc.);
- combinatorial operations: Cartesian product, subsets, and permutations;
- probability theory: probability, dependence, expected value;
- functions: mapping, injection/surjection, associative/commutitive/distributive/transitive properties;
- formal logic: conjunction, disjunction, equivalence, implication, negation, and DeMorgan's law;
- proofs, writing proofs, and what constitutes a proof;
- proof techniques: direct argument, contradiction, induction, constructive proofs, the pigeonhole principle; and
- graphs: vertices, edges, labels, directedness, paths, cycles, and connectivity.

Finally, we expect familiarity with univariate derivative and integral calculus:

- real numbers and functions;
- continuous and discontinuous functions;

- limits;
- derivatives of a single variable; and
- routine integrals of a single variable.

Derivatives and integrals are used sparingly, so a reader who is rusty on these should be able to follow the text. The other subjects are essential.

There are open access texts that cover all of this material:

- Programming: *How to Think Like a Computer Scientist* [18] or *How to Design Programs* [21]
- Data structures: *Open Data Structures* [41]
- Discrete mathematics: *Mathematics for Computer Science* [20]
- Calculus: *Calculus* [48]

1.3 Code style

This is not a book about Python programming, but we do present pseudocode in a Python-like syntax, and algorithms implementations as Python functions. We use Python for this purpose because, as pseudocode, its syntax tends to be readable by programmers coming from just about any other contemporary language. As an implementation language, its conciseness allows us to express the key ideas of algorithms without getting bogged down in a lot of lower-level details. As a relatively simple language, Python code can be ported to other languages more easily than can code that makes inextricable use of non-universal language features (e.g. syntax sugar, laziness, or continuation-passing).

Since the goal of our code listings is to communicate algorithms to a broad audience, we prioritize clarity, simplicity, and ease of reimplementation over constant-factor efficiency and conformance to Python-specific coding conventions. Our code is not intended to be an exemplar of industrial-grade idiomatic Python, and should not be treated as such!

1.4 Navigating the book

This book covers material in an unconventional order. This is a deliberate design decision intended to "load-level" the cognitive difficulty of the material. Experience shows that students tend to struggle the most with the more abstract concepts of dynamic programming and NP-completeness. Conventionally these topics are covered toward the end of the text, as self-contained modules that introduce a lot of ideas all at once. In practice, material at the end of a class tends to be rushed, compounding the challenge.

Instead, this text refactors the material, front-loading some of the foundational ideas of dynamic programming and NP-completeness. The greedy method is the first algorithm pattern covered, allowing us to flesh out invariants, correctness proofs, and nontrivial time analyses in the context of algorithms that are otherwise straightforward. The shortest-path data structure used in Dijkstra's algorithm is a preview of the backtracking data structures used later with dynamic programming. Next we lavish quite a bit of time and space on exhaustive search, which allows us to solidify the notions of candidate solutions, verifier algorithms, and decision versions of optimization problems. Later, we cover reduction as a first-class algorithm design pattern, and explore how polynomial-time algorithms may reduce to each other (in particular sorting) as a practical design approach. Therefore, by the time we get to NP-completeness, students are already versed in the concepts of verification and reduction, and all that remains is the technical definition of NP-completeness and proving essential results.

1.5 Why Another Algorithms Textbook?

There are several excellent introductory algorithms on the market [16] [24] [38] [46] [49], so why did we go to the effort of creating another?

Simply put, none of the extant books are a great fit for the circumstances at CSU Fullerton. Existing books tend to assume that students have thoroughly mastered the mathematics of computer science, and then proceed to build upon that foundation. But, our students' mathematical preparation is heterogeneous, and many students are still developing their mathematical savvy concurrently with learning about algorithms. Proprietary textbook licenses are a barrier too, both because the cost of textbooks is a very real obstacle to access, and because our enrollment process encourages students to avoid investing in materials until several weeks deep into the semester. This is a difficult subject to learn and teach without a common text.

This book strives to solve these problems. As described in the previous sections, our pseudocode resembles code more than math notation in order to make it more legible for students who are more experienced with programming than they are with mathematics. We have permuted the ordering of the material so as to aid digestion. Our proofs and analyses are methodical, even verbose at first, so that they may be understood completely. The book is distributed digitally under a permissive Creative Commons license, so that students can obtain it instantly and free of charge.

1.6 Acknowledgements

We are indebted to the following generous souls who have contributed corrections, suggestions, and other improvements to this text: Doina Bein, Laurence Bernstein, Jeffrey Bohlin, Matthew Braun, Chantalle Bril, Jaime Cabrera, Elizabeth Chen, Shih-Min Annie Chen, David Dang, Ciaran Downey, Raul Esquivias, Andrew Gomez, Eric Guzman, Grace Hadiyanto, Patrick Henning, Dong Ho, Maygan Hooper, Yong Kim, Frida Kiriakos, Douglas Galm, Joe Greene, RJ Kretschmar, Cong Le, Hwa Sung Lee, Johnal Leifsson, Mariko Molodowitch, David Monson, Harry Mora, Noor Najjar,

Hympert Nguyen, Dion Pieterse, Jacob Pillai, Swati Sahoo, Anika Salhotra, Marek Sautter, Arshya Sharifian, Ryan Shim, Daniel Streb, Elizabeth Tsan, Chary Vielma, Kevin Vu, Nicholas Webster, Ryan Williams, and Sameera Yayavaram.

Act I

Designing Algorithms from Patterns

Chapter 2

Fundamentals

2.1 The Big Idea

Before we can get into designing and comparing algorithms, we need to define what an algorithm is, and establish a notation for defining algorithms. This chapter defines the technical terminology, pseudocode notation, and notion of algorithm patterns that are used throughout the text.

2.2 Terminology

This section introduces and defines fundamental technical terms pertinent to the study of computer algorithms. While it may seem pedantic to define these terms so specifically, please bear with us. This level of precision will come in handy later, especially in our treatment of the complexity of problems.

Data

Definition 1. *A* datum *(plural* data*) is a finite mathematical object that can be represented by a string of binary 0 and 1 digits.*

This should be a straightforward definition. By this definition, the primitive data types common to programming languages are types of data: integers (within a bounded range), floating point numbers, characters, Boolean values, and pointers (memory addresses). Data types composed of primitive types count as data too: strings, arrays of some data type, class types, and so on.

The "finite" qualifier is necessary because our computers have finite memory capacity. While it may be commonplace to reason about infinite objects in mathematics, our computers can only handle finite objects as data.

Note that our definition says that data is a specific kind of mathematical object. So it is appropriate to use mathematical notation and terminology to describe data.

Problems

Definition 2. *A* problem *is defined by an* input *and* output *specification, each of which specifies a type of data and possibly some constraints on that data.*

Intuitively, a problem is a kind of task that we'd like to solve with computers. Rendering web pages, evaluating spreadsheet cells, and decoding MP3 tracks are all examples of problems. We will define problems with the following notation:

problem name
input: input specification **output:** output specification

For example:

minimum problem
input: a non-empty list L of n orderable objects **output:** the least (minimum) element in L

To be clear, the input specifications states that the input's data type is a sequence of objects, and that the data type of each object must be *comparable*. A data type is comparable when it is possible to compare objects of that type to determine whether they are equal or unequal, for example with the `==` and `<` operators. In addition to defining the data type of an input, the specification also introduces the constraint that S must not be empty. This is a logical necessity since there is no such thing as the minimum element from an empty sequence. The output specification says that the data type of the output must be one element from S, and constrains the output to actually be the minimum instead of some other element.

A problem is defined in terms of input data and output data, each of which are mathematical objects. So a problem is a kind of mathematical object too.

Variables introduced in an input specification are visible in the same problem's output specification. That's why the output definition for the minimum problem doesn't need to define S; it is implicitly using S as defined in the input specification. Each problem definition exists in its own private scope, though. Suppose we define another, similar problem.

summation problem
input: a list X of numbers **output:** the sum (total) of all elements of X

Even though both the minimum problem and summation problem use S as a variable name, conceptually these are distinct objects. That's why it is valid to define them slightly differently.

Problem instances and solutions

Definition 3. *A* problem instance *for a specific problem is a concrete input object for that problem.*

For example, the sequence $\langle 7, 2, 1, 9 \rangle$ is a valid instance for the minimum problem. The empty sequence $\langle\rangle$ is not a valid instance for that problem.

Definition 4. *A* solution *for a specific problem and instance is a valid concrete output corresponding to the problem and instance.*

To continue our prior example, the solution for the instance $\langle 7, 2, 1, 9 \rangle$ to the minimum problem is 1.

Note that a solution is "a" correct output, not "the" correct output. This phrasing accommodates problems for which an instance might have more than one correct output. For example:

duplicate search problem
input: a list L of comparable objects **output:** an element of L that appears more than once in L, or `None` if no such element exists

The only solution for the instance $\langle 1, 2, 3, 3, 4 \rangle$ of the duplicate search problem is 3, and the only solution for $\langle 1, 2, 3 \rangle$ is `None`. However, the instance $\langle 1, 1, 2, 3, 3 \rangle$ has two correct solutions, 1 and 3.

Algorithms

Definition 5. *A* process *is a defined series of actions directed to some end.*

Driving directions, cooking recipes, and computer programs are all processes by this definition. An algorithm is a process that meets three distinguishing criteria.

Definition 6. *An* algorithm *is a process which, given an instance of a specific problem, produces a solution for that instance, and*

1. *may be described clearly enough to be implemented (clarity),*
2. *always produces a correct solution (correctness), and*

3. *takes a finite amount of time (termination).*

Algorithms are the entire subject of this text, so the definition of what constitutes an algorithm bears some scrutiny.

The *clarity* property means that a process only qualifies as an algorithm when it can be described clearly and specifically enough to be implemented, either by an executable computer program or by a human running it by hand. A phrase such as "render the HTML" hints at a process but is not clear enough to be coded up. By contrast the phrase "add x to y and divide by two" meets the clarity requirement (assuming that x and y have already been defined clearly).

The *correctness* property means that the process *always* succeeds at producing a correct output; there is no chance whatsoever of failure. This is a high bar. While real-world computer programs inevitably have bugs that cause them to fail in some cases, an algorithm must always operate perfectly.

The *termination* property means that the process takes a finite amount of time, or in other words never gets stuck in an infinite loop. As with correctness, this is an absolute requirement. In order to be an algorithm, a process must have no chance whatsoever of slipping into an infinite loop. As with the requirement of finite inputs and outputs, this is a practical requirement motivated by the finite nature of computer hardware (and human lifespans!). It also imposes a fundamental limitation on the power of algorithms, a subject we will explore in our later treatment of undecidable problems in chapter 13.

Pseudocode

An algorithm is a kind of process, and there are many ways of communicating processes, such as step-by-step instructions, flowcharts, diagrams, and recipes. We prefer conveying algorithms with pseudocode.

Definition 7. Pseudocode *is a human-readable format for communicating algorithms that may include code-like syntax, math notation, and prose.*

As its name implies, pseudocode is similar to program source code in a language such as Python or C, but is not required to be syntactically-perfect code. We relax the syntax requirements of true programming languages and allow authors to freely mix in snippets of math notation and English prose. For example, consider the following trivial problem.

arbitrary element problem
input: a non-empty sequence S of n elements **output:** any element of S, or `None` if S is empty

The following pseudocode describes an algorithm for solving the problem.

```
def arbitrary_element(S):
  if |S| = 0:
    return None
  else:
    return the first element of S
```

This pseudocode resembles Python code. However it includes the math notation $|S| = 0$, and the prose phrase `the first element of S`, neither of which would be accepted by a Python interpreter. Taking these liberties does not inhibit the clarity of the algorithm, though. It is still plain how to translate those excerpts into valid program source code and how to execute them by hand.

Taken to an extreme, the use of math notation or prose could impede clarity. This is particularly true when a sophisticated pseudocode author uses terminology that their audience may not understand. For example, the phrase "perform a bottom-up three-pivot randomized quick sort" would be perfectly clear to an algorithm researcher, but is likely unclear to an algorithm student reading this text for the first time. So clarity is a moving target; pseudocode may be clear for one audience and unclear for another. To resolve that ambiguity, we use the following litmus test to determine whether pseudocode is clear:

> *Pseudocode is clear when a member of its intended audience could translate it into running program code without any further explanation.*

Students should write for an intended audience of their peers. So, a student in an introductory algorithm course has produced clear pseudocode when a typical fellow classmate could translate it into a working computer program without any further explanation. Pseudocode in an academic research paper about algorithms is considered clear if it could hypothetically be implemented by a typical member of the paper's audience of algorithms researchers, many with Ph.D. degrees in the subject.

Implementations

Definition 8. *An* implementation *of an algorithm is executable computer code that follows the process defined by the algorithm.*

When a programmer develops code that carries out an algorithm, they have created an implementation of that algorithm. While algorithms and implementations of algorithms may be closely related, they are not the same thing. An algorithm is a timeless mathematical object that is intended to be read and understood by humans. An implementation of an algorithm is a computer program, written in a specific language, that runs in a specific environment and operating system, that is intended to be executed by computer hardware. Programming is difficult, and it is not uncommon for a correct algorithm to be implemented incorrectly. That means that, just because

an algorithm is correct (meaning that it unerringly produces valid outputs), does not mean that an implementation of that algorithm is correct.

Terminology Summary

In summary, the following are all mathematical objects:

- problems,
- algorithms, and
- pseudocode.

As mathematical objects, they exist in the minds of human beings and are intended for human consumption. We describe them verbally and in prose writing.

The following are digital artifacts:

- problem instances,
- solutions,
- code, and
- algorithm implementations.

These are intended for machine consumption, and are encoded in machine-readable formats such as computer code, binary executables, and data files.

2.3 Pseudocode Checklist

In order for a piece of pseudocode to be clear, correct, and terminate, it must satisfy all of the properties on the following list. This checklist is intended to help in *sanity-checking* whether pseudocode has any obvious deficiencies. If a piece of pseudocode fails any of these tests, it is not good enough to specify an algorithm and needs more work. If pseudocode passes all these tests it is probably, but not necessarily, at least adequate.

1. *Input and output:* Are the algorithm's inputs and produced outputs clear, and explicitly separated from other variables? Often it is best to write an algorithm as a Python-style function whose arguments correspond to problem inputs and whose return value corresponds to problem output.

2. *Undefined variables:* Are any variables used before they are defined or initialized? Every variable needs to be initialized before it is referenced.

3. *Variable meanings:* Is the intended meaning of every variable clear? Use conventional short names (e.g. `i` and `j` for array indices and `v` for a graph vertex), and long descriptive names such as `coins` or `matrix` for other values. Explain potentially-confusing variables with a comment.

4. *Defined return value:* Does every execution path have a defined `return` value? If not, the algorithm's output is undefined.

5. *Return value data type:* Does the data type of every returned value match the **output** in the problem definition? For example, an algorithm for the duplicate search problem must always return either an element of the sequence `S` or the special value `None`, not an integer index.

6. *Handles all cases:* Does your algorithm have the potential to return every kind of valid output? For example, an algorithm for duplicate search must have the capability to return `None` when appropriate. If a piece of pseudocode never mentions `None`, it cannot possibly solve that problem correctly.

7. *Loop termination:* Does every loop have a termination condition that prevents infinite loops? Remember that the body of every loop must update variables somehow so that the loop will eventually terminate.

8. *Base case:* Does every recursive function have a clearly defined base case that is inevitably triggered?

9. *Repetitive code:* Refactor repetitive code into a helper function or loop.

10. *Dead code:* Delete code that is never executed.

11. *Vagueness:* Are any steps vague? Check that every line of pseudocode could be translated into program code without elaboration.

2.4 Algorithm Patterns

This text covers the design of algorithms through the lens of *patterns.* As you might expect, an algorithm pattern is a problem-agnostic approach for getting started at designing an algorithm. It is something like a *template,* which provides a skeletal outline of an algorithm, but leaves problem-specific *blanks* unspecified. We will describe patterns using very-high-level pseudocode, with blanks written between angle brackets, e.g. `<BLANK>`. Algorithm patterns are inspired by *software patterns* [52], such as singletons and factories, which are themselves inspired by *architectural patterns* which were first identified and codified by the architect Christopher Alexander [10].

Designing algorithms is an iterative process. It is rarely feasible to sit down and write out clear, correct pseudocode without any prior contemplation. Rather, algorithm designers usually work

through many drafts of an algorithm, starting with a pattern with no blanks filled in, then revising through multiple drafts of increasing clarity, until finally arriving at polished and presentable pseudocode. Most of these drafts do not meet our strict definition of "algorithm," since parts of their pseudocode may be unclear, and may not initially be correct.

So, to design and analyze an algorithm using patterns:

1. Make sure you are clear about what problem your algorithm will solve.
2. Pick a pattern.
3. Make a copy of the pattern's pseudocode, blanks and all, and consider that your first draft.
4. Revise your draft by filling in blanks, repairing incorrect parts of the algorithm, eliminating potential infinite loops, or clarifying unclear passages. Repeat until your pseudocode is correct, terminates, and is clear.
5. You may need to write one final, clean draft so that your pseudocode will truly be clear to others and free of typographical errors.
6. Prove that your algorithm is correct, working from your final draft.
7. Prove the efficiency of your algorithm, working from your final draft.

This and the next few chapters introduce a toolbox of patterns to choose from.

Exercises

2-1. Describe, in your own words, the difference between a mathematical set and a mathematical sequence. Use proper math notation to write the set containing the numbers 1, 2, and 3, and also a sequence containing the same numbers.

2-2. Why is this section of the book titled "Exercises" and not "Problems"?

2-3. Give a) an example of a valid instance of the summation problem that is not a valid instance of the minimum problem, and b) vice-versa.

2-4. Each of the following snippets of pseudocode fails to live up to all of the clarity, correctness, and termination requirements of algorithms. In each case, describe the failing, and then rewrite the pseudocode as a proper algorithm. *Hint:* consult the checklist in section 2.3.

(a)
```
def contains_zero(S):
    for x in S:
        if x == 0:
            return True
        else:
            return False
```

(b)
```
def keep_positives(S):
  if len(S) == 0:
    return 0
  else:
    result = []
    for x in S:
      if x > 0:
        result.add(x)
    return result
```

(c)
```
for list:
  total = total + i
```

(d)
```
def long_division(numerator, denominator):
  quotient = numerator / denominator
  remainder = numerator % denominator
```

(e)
```
def does_alternate_01(S):
    i = 0
    while i < len(S):
        if S[i] == 0:
            if S[i+1] != 1:
                return False
        elif S[i] == 1:
            if S[i+1] != 0:
                return False
        else:
            return False

        i += 1

    return True
```

2-5. Write a problem definition for each of the following problems.

(a) computing a square root

(b) determining whether an integer is even or odd

(c) determining whether every element in a sequence is identical

(d) determining whether two strings are identical

(e) determining whether a sequence contains entirely positive numbers

(f) determining whether a sequence is in strictly increasing order (i.e. each number is greater than the last)

(g) concatenating two sequences into one larger sequence

2-6. Write pseudocode for an algorithm that solves each of the problems in the previous question.

2-7. Prove that it is possible for one algorithm to solve multiple problems. *Hint:* give a constructive proof including two problems, an algorithm, and an argument that the algorithm solves both problems correctly.

2-8. Do some research on the Internet to find a problem that computer scientists consider to be difficult for algorithms to solve. Write a definition of the problem in our format, complete with an **input** and **output** specification.

Chapter 3

Efficiency Analysis

3.1 The Big Idea

As algorithm designers, we endeavor to create algorithms of high quality. For the most part, "high quality" means "efficient." We are approaching algorithm design from a practical perspective. A first step in any design optimization is to establish a way of quantifying and measuring the feature that is being optimized, so that it is possible to reason objectively about whether design alternatives make things better or worse.

As we will see, computer scientists measure the efficiency of algorithms with *efficiency classes* such as $O(n)$ and $O(n^2)$. *Analyzing* an algorithm involves starting from an algorithm's pseudocode, and arriving at the efficiency class of that algorithm.

This chapter deals with efficiency classes, why we use them, and two different methods to algorithm analysis: mathematical and experimental. These methods have the predictive power of science; given a pseudocode description of an algorithm, we can accurately predict how efficient an implementation of that algorithm is likely to be.

3.2 Functions Measuring Resources

An algorithm is efficient when it consumes few resources.

Definition 9. *A* resource *is a quantifiable thing that is expended while executing an algorithm.*

This is a broad definition that may apply to a diverse array of computer resources, including but not limited to

- *time,* measured in units of seconds, CPU instructions, or generic *steps;*

- *space,* measured in units of bits, bytes, gigabytes, or generic *words;*
- *input/output bandwidth (I/O)*, measured in units of bytes or blocks;
- *cache misses,* measured in units of integers; or
- *energy,* measured in units of kilowatt-hours.

Of these, *time* tends to get the most attention. It was the first resource to be quantified formally and is often the most pressing bottleneck in practical applications. The usage of other resources tends to be closely correlated to the usage of time, so time is a reasonable measure for the overall efficiency of an algorithm. And, computer memory (space) is routinely garbage-collected and reused; energy, I/O, and cache misses may be supplied on demand; but time can never be reclaimed. Once you spend time on something there is no getting it back! So, for these reasons we will focus primarily on the time efficiency of algorithms, and give some secondary attention to their space efficiency.

3.2.1 Worst-case analysis

We use the word *complexity* to refer to the amount of a resource that is consumed by an algorithm.

Definition 10. *The* complexity *of an algorithm A, with respect to a specific instance I and resource R, is a non-negative real number representing the amount R that is consumed by A when run on I.*

For example, consider the following simple problem and algorithm.

summation problem
input: a list X of numbers **output:** the sum (total) of all elements of X

```
def naive_sum(S):
  total = 0
  for x in S:
    total += x
  return total
```

Suppose for the sake of discussion that an implementation of the `naive_sum` algorithm consumes 210 milliseconds of time and 100 bytes of memory when executed with the instance $S = \langle 3, -2, 8, 41 \rangle$. Then `naive_sum` 's complexity with respect to time is 210 milliseconds for that value of `L` , and its complexity with respect to space is 100 bytes with respect to that same S.

This sort of complexity factoid is not very interesting by itself. Suppose you were a software developer trying to decide whether to use `naive_sum` or a competing algorithm; would the fact

that it takes a certain number of milliseconds and bytes for the input $\langle 3, -2, 8, 41 \rangle$ help you make your decision? Not likely. We wish to characterize the complexity of algorithms more generally so that we can predict how they might perform on unforeseen instances. To that end, we often characterize the *worst case* complexity of algorithms. The worst case complexity is the worst (greatest) complexity of that algorithm, over all valid inputs to the algorithm.

Definition 11. *The* worst case complexity *of an algorithm with respect to a resource is the maximum amount of that resource that the algorithm may consume for any instance, as a function of the size of instances.*

As stated above we focus on time complexity.

Definition 12. *The* worst case time complexity*, or simply* complexity*, of an algorithm is the worst-case complexity of an algorithm with respect to time.*

Focusing on worst case complexity has an important benefit. By definition all complexity values are less than or equal to the maximum complexity value, so statements about worst-case complexities are both strong and conservative. Suppose you are concerned with the time complexity of an algorithm that generates driving directions for an in-car navigation system. Which statement would be most compelling:

1. the algorithm takes at least 2 seconds (and possibly longer);
2. the algorithm probably takes 2 seconds (sometimes more, sometimes less); or
3. the algorithm takes at most 2 seconds (and possibly fewer).

Clearly the third statement is strongest and most useful for users of the algorithm.

Example 1. *Suppose, again for the sake of discussion, that* `naive_sum` *always consumes at most* $2n$ *milliseconds. Then* `naive_sum` *'s worst case time complexity is* $2n$ *milliseconds.*

Focusing on worst case complexity instead of instance-specific complexity has a convenient side benefit: it simplifies mathematical analyses. Whenever an algorithm might consume two different amounts of a resource — due to an `if` statement, for example — we can quantify the two alternatives and take the maximum of the two expressions. While this does greatly simplify algorithm analysis, it is not the primary motivation for studying worst case complexity. Rather, we study worst case complexity since that is the most informative characterization of complexity that we could make.

3.2.2 Complexity functions

Common sense dictates that the complexity of an algorithm is related to the size of its input. In the case of `naive_sum` , the `for` loop iterates once per element of S, so it stands to reason that

the time consumed by that loop will be proportional to the length n of S. In mathematics we use *functions* to capture these sorts of relationships. Here, we use *complexity functions* that map *size measure* values to complexity values.

Definition 13. *A* size measure *is a real-valued variable corresponding to the size of a particular problem's instances.*

Many problems have only one natural size measure, and in these cases it is customary to use the variable n.

Example 2. *In the summation problem, a natural measure is the quantity $n \equiv |S|$, representing the number of elements in the list S.*

Some problems may have multiple size measures.

Example 3. *Graph problems typically have two size measures, the number of vertices $n \equiv |V|$ in the graph, and also the number of edges $m \equiv |E|$.*

Definition 14. *A* complexity function $f(n_0, n_1, \ldots, n_{k-1})$ *is a function that maps the $k > 0$ size measures $n_0, \ldots, n_{k-1}$ of a problem to a non-negative real number representing the complexity of an algorithm with respect to the size measures. A* univariate *complexity function is a complexity function of only $k = 1$ variable, e.g. $f(n_0)$ or $f(n)$. A* multivariate *complexity function is a complexity function of multiple ($k > 1$) variables.*

Note that complexity functions are *non-negative.* This is because they measure an amount of a resource that is consumed; resource consumption is never negative. There is no such thing as spending negative time, or negative space, on a computation, at least for our present purposes. If we ever arrive at a complexity function that produces negative values, we should take that as an indication that we have made some kind of error.

We adopt the convention of naming time complexity functions T.

Example 4. *The only size measure of the summation problem is n, the length of an input list. We already asserted that* `naive_sum` *takes at most $2n$ milliseconds and 200 bytes, so we could define a complexity function*

$$T(n) = 2n$$

representing the worst case time complexity of `naive_sum`*.*

3.3 Asymptotic Notation

So far we have simply asserted the complexity functions without any sort of evidence that those assertions are correct. Soon we will explore two evidence-based approaches to deriving complexity functions of algorithms:

1. *experimental analysis,* using the scientific method of hypothesis, experiment, and empirical data analysis; and
2. *mathematical analysis,* using the mathematical method of modeling, lemma, and proof.

The outcome of either kind of analysis is a complexity function. It is not actually feasible to define complexity functions that are precise down to the millisecond and byte, as we did in the preceding section, with either of these approaches. Practical considerations dictate that we instead establish a *big-picture trend* of complexity functions in a way that downplays small additive and multiplicative constants, and the complexity of "outlier" instances whose complexity is uncharacteristically small relative to the trend.

Computer scientists use the mathematical notion of an *efficiency class,* denoted with *asymptotic notation*, otherwise known as "big-O" notation, to deal with these practical considerations. An O expression defines a set of functions; its formal definition [33] is

$$O(f(n)) = \{g(n) \mid \exists\, c > 0, t \geq 0 \text{ such that } g(n) \leq c \cdot f(n) \ \ \forall n \geq t\}.$$

We do not expect this definition to be intuitive at this point. So, we now offer three observations about the nature of algorithmic complexity. Each observation relates to an obstacle to modeling complexity as a mathematical function. Each of these obstacles may be overcome by adding one stipulation to our definition of O.

For the purposes of exposition, this section will discuss only univariate complexity functions $f(n)$, but everything covered here generalizes naturally to multivariate complexity functions $f(n_0, \ldots, n_{k-1})$.

Observation: some instances' complexity is less than the worst case complexity

Our first observation is almost vacuously true: that if the *worst case* complexity of an algorithm on input size n is $f(n)$, then there may exist inputs of size n whose complexity x is less than $f(n)$. More concretely, if some function $T_p(n)$ measures the actual time of an algorithm for a *particular input,* and $T(n)$ measures the *worst case* time of the algorithm for *any input,* then it should be the case that $T_p(n) \leq T(n)$.

Takeaway: A complexity function $f(n)$ measures the worst case complexity of an algorithm when

$$g(n) \leq f(n)$$

for any $g(n)$ that measures the complexity of the algorithm for a particular instance.

Observation: sometimes small instances do not follow general trends

It is commonplace for complexity functions to be *discontinuous,* especially in the case of small input sizes. For example, we might optimize our `naive_sum` algorithm slightly to handle empty lists more quickly.

```
def special_case_sum(S):
  if len(S) == 0:
    return 0
  else:
    total = 0
    for x in S:
      total += x
    return total
```

We expect this version of the algorithm to be slightly faster in the case of an empty input list, since in that case the algorithm does not initialize any variables or incur any loop overhead. A precise time complexity function for the number of milliseconds taken by this algorithm might be something like

$$T(n) = \begin{cases} 2 & \text{if } n = 0 \\ 2n + 4 & \text{if } n > 0. \end{cases}$$

This is a piecewise and discontinuous function. It may be precise, but it is also unwieldy to state and manipulate mathematically. The precision regarding the algorithm's behavior around $n = 0$ does not really help us understand the big-picture trend of `special_case_sum` 's running time. It looks like this algorithm probably takes an insignificant amount of time when L is empty, and takes an amount of time proportional to n when L is long. Frankly, the simpler function $T(n) = 2n + 4$ also captures this "insignificant when n is small, proportional to n when n is large" trend, and the piecewise definition only distracts from that.

So, when analyzing an algorithm, we assume there is some *threshold* input size representing inputs that are large enough for big-picture trends to be established. Inputs that are smaller than that threshold are not germaine to big-picture trends, so we ignore them.

Takeaway: When analyzing the complexity of an algorithm, we assume there exists some threshold input size t, and only require that claims about complexity functions hold when $n \geq t$.

Observation: any resource measure involves a hidden constant

So far we have been measuring time in units of milliseconds. However, this is problematic; there is no straightforward way to quantify the time complexity of excerpts of pseudocode in real-world units such as milliseconds or days.

For one thing, computers have been manufactured with a wide array of clock speeds. The author first learned to program on a Commodore 64 computer with a clock speed of approximately 1 megahertz. The laptop used to write this book runs at approximately 2.4 gigahertz, about 2,400

times faster than the Commodore. If our laptop takes 1 unit of time to execute the pseudocode statement

```
  x = 0
```

then we should expect the Commodore to take approximately 2,400 units of time to execute the same statement. So, when analyzing the running time of that pseudocode, which number should we use, 1 or 2,400?

Actually that question is ill-posed. The purpose of algorithm analysis is to establish the efficiency of *algorithms;* this issue of CPU clock rates relates to the efficiency of *implementations* of algorithms, specifically the CPU used to run a software implementation of an algorithm. The specifications of the computer hardware used to execute an implementation of an algorithm are irrelevant to the efficiency of the algorithm itself.

One solution might be to measure time complexity in units of *instructions* instead of clock units such as seconds. Then we could analyze an algorithm's pseudocode by working out how many CPU instructions each part of the pseudocode corresponds to. However, once again we would be focusing on the efficiency of implementations of algorithms, not algorithms themselves. Different CPU architectures require different numbers of instructions to perform the same task.

For example, consider the following C function that computes the mean average of three integers.

```
int mean3(int a, int b, int c) {
  return (a + b + c) / 3;
}
```

The CLANG C compiler [2] generates the following AMD64-architecture assembly code for the function.

```
mean3:                                  # @mean3
        .cfi_startproc
# BB#0:
        pushq        %rbp
.Ltmp0:
        .cfi_def_cfa_offset 16
.Ltmp1:
        .cfi_offset %rbp, -16
        movq        %rsp, %rbp
.Ltmp2:
        .cfi_def_cfa_register %rbp
        movl        $3, %eax
        movl        %edi, -4(%rbp)
        movl        %esi, -8(%rbp)
        movl        %edx, -12(%rbp)
        movl        -4(%rbp), %edx
        addl        -8(%rbp), %edx
        addl        -12(%rbp), %edx
        movl        %eax, -16(%rbp)         # 4-byte Spill
        movl        %edx, %eax
        cltd
        movl        -16(%rbp), %esi         # 4-byte Reload
        idivl        %esi
        popq        %rbp
        retq
```

That compiler generates the following ARM assembly code for the same function.

```
mean3:                                  @ @mean3
        .fnstart
@ BB#0:
        push        {r4, lr}
        sub        sp, sp, #28
        mov        r3, r2
        mov        r12, r1
        mov        lr, r0
        str        r0, [sp, #24]
        str        r1, [sp, #20]
        str        r2, [sp, #16]
        ldr        r0, [sp, #24]
        ldr        r1, [sp, #20]
        add        r0, r0, r1
        add        r0, r0, r2
        ldr        r1, .LCPI0_0
        smull        r2, r4, r0, r1
        add        r0, r4, r4, lsr #31
        str        r3, [sp, #12]            @ 4-byte Spill
        str        r12, [sp, #8]            @ 4-byte Spill
        str        lr, [sp, #4]             @ 4-byte Spill
        str        r2, [sp]                 @ 4-byte Spill
        add        sp, sp, #28
        pop        {r4, lr}
        bx        lr
```

The AMD64 code is 20 lines long (not counting jump labels), while the ARM code is 23 lines long (again not counting labels). One reason for the discrepancy is that the two architectures pass arguments to functions differently. Another reason is that the ARM architecture does not have an instruction for integer division, so the compiler had to use a few addition and multiply instructions to divide by three.

The takeaway is that in this example, it's unclear whether this short function should cost 20 time units, 23 time units, or some other number. More generally, there is no set mapping from pseudocode operations to assembly operations. Even if we were to pick a specific CPU clock rate and architecture, we would still be unable to deduce a correspondence between pseudocode and instructions. Other details, such as the programming language, operating system, compiler version, and compiler settings would all need to be specified as well.

Trying to map pseudocode to specific units of time seems to be hopeless. Stepping back a bit, according to a unit analysis, the observed running time of an algorithm implementation, in units of seconds, is

$$\text{seconds} = \text{pseudocode steps} \cdot \frac{\text{CPU instructions}}{\text{pseudocode step}} \cdot \frac{\text{seconds}}{\text{CPU instruction}}.$$

Of these, the $\frac{\text{CPU instructions}}{\text{pseudocode step}}$ and $\frac{\text{seconds}}{\text{CPU instruction}}$ factors are actually aspects of an implementation and computing environment, and should be ignored in an analysis of the algorithm itself. The only factor relevant to the efficiency of the algorithm is the number of steps called for by the algorithm's pseudocode. So, we theorize that the observed running time of an algorithm may be modeled as

$$\text{seconds} = \text{pseudocode steps} \cdot c,$$

where

$$c = \frac{\text{CPU instructions}}{\text{pseudocode step}} \cdot \frac{\text{seconds}}{\text{CPU instruction}}$$

is some fixed constant that models the time factors that are specific to an algorithm's implementation.

Thus, when analyzing the time efficiency of an algorithm, we count a number of abstract *steps* performed by the algorithm. Each step conceptually corresponds to an amount of computation that an implementation can achieve in a fixed granule of time. In order for the correspondence between abstract steps and seconds of execution time to hold, it is important that one step only correspond to work that can be completed in some fixed number of CPU instructions. However, the duration of these time granules, and the precise number of CPU instructions per granule, are unimportant. No matter what they are, they amount to a fixed multiplicative constant c that is implicitly multiplied into any algorithm's running time.

Takeaway: For the purposes of analyzing algorithms' efficiency, two functions

$$T(n)$$

and

$$T'(n) = c \cdot T(n),$$

differing only by a constant factor $c > 0$, should be considered equivalent.

3.3.1 Definition of O

The previous section motivated measuring algorithmic complexity by mathematical functions $T(n)$ or $T(n_0, \ldots, n_{k-1})$, subject to three simplifying assumptions:

1. we analyze for worst case complexity functions $f(n)$ that form an *upper bound,* so that any observed complexity $g(n)$ obeys $g(n) \leq f(n)$;

2. we disregard input sizes n smaller than some algorithm-specific threshold t; and

3. we also disregard constant multiplicative factors in complexity functions; in other words, we may replace $c \cdot f(n)$ with $f(n)$ for any $c > 0$.

Intuitively, an efficiency class $O(f(n))$, for some specific complexity function $f(n)$, is a set of functions that are equivalent under those three assumptions.

For example,

- $O(n)$ is the set of all functions that are equivalent to $f(n) = n$ for the purposes of measuring algorithmic complexity;
- $O(n^2)$ is the set of all functions that are equivalent to $f(n) = n^2$ for the purposes of measuring algorithmic complexity;

and so on.

We now restate the definition of O, which ought to make more sense this time.

Definition 15. *If $f(n)$ is an univariate complexity function, then*

$$O(f(n)) = \{g(n) \mid \textit{ there exists some constants } c > 0 \textit{ and } t \geq 0 \textit{ such that } g(n) \leq c \cdot f(n) \textit{ whenever } n \geq t\}.$$

If $f(n_0, \ldots, n_{k-1})$ is a multivariate complexity function,

$$O(f(n_0, \ldots, n_{k-1})) = \{\, g(n_0, \ldots, n_{k-1}) \quad | \quad \begin{array}{l} \textit{there exists some constants } c > 0 \textit{ and } t_0 \geq 0, \ldots, t_{k-1} \geq 0, \\ \textit{such that } g(n_0, \ldots, n_{k-1}) \leq c \cdot f(n_0, \ldots, n_{k-1}) \\ \textit{whenever } n_0 \geq t_0, \ldots, n_{k-1} \geq t_{k-1} \,\}. \end{array}$$

It is important to remember that an efficiency class expression such as $O(n)$ or $O(n^2)$ defines a *set of functions.* Specifically, $O(n)$ is the set of functions resembling $f(n) = n$, and $O(n^2)$ is the set of functions resembling $f(n) = n^2$. So the mathematical notation relating to sets applies to efficiency classes.

Example 5. *The following are all well-typed, true mathematical statements.*

$$n \in O(n)$$

$$7n \in O(n)$$

$$2n + 3 \in O(n)$$

$$\frac{3}{2}n^2 \in O(n^2)$$

$$O(1) \subseteq O(n) \subseteq O(n^2)$$

3.4 The Big Eight Efficiency Classes

The following table lists the eight univariate efficiency classes that are encountered most frequently in the analysis of practical algorithms. For reasons that will become apparent, the time complexity of a well-designed algorithm is likely to fall into one of these classes.

Notation	Name	Example
$O(1)$	constant	evaluating one statement; `mean3`
$O(\log n)$	logarithmic	searching a balanced binary search tree
$O(n)$	linear	`for` loop; `naive_sum`
$O(n \log n)$	"n-log-n"	fast sorting algorithms such as merge sort
$O(n^2)$	quadratic	two nested `for` loops; initializing an $n \times n$ matrix
$O(n^3)$	cubic	three nested `for` loops; multiplying two $n \times n$ matrices
$O(c^n)$ for $c > 1$	exponential	all subsets of an n-element set
$O(n!)$	factorial	all permutations of an n-element sequence

3.5 Experimental Analysis

In this section we explore the *experimental,* or *empirical,* approach to establishing the efficiency class of an algorithm.

3.5.1 The Scientific Method

Scientific knowledge is established, challenged, and confirmed using the *scientific method* [7]. This method uses observable facts to lend support to, or challenge, theories. Using the scientific method involves five steps.

1. *Pose a question.*
2. *State a hypothesis.* A hypothesis is a conjectured answer to the question, based on our current understanding of how the world works.
3. *Make a prediction.* A prediction is a logical consequence of the hypothesis.
4. *Test.* Design an experiment that will measure observable facts. The result of the experiment should speak to whether the prediction came true or was falsified. Gather objective data.
5. *Analyze.* Analyze the data and assess whether it is consistent or inconsistent with the prediction. If consistent, the hypothesis is validated; if inconsistent, the hypothesis is rejected.

We can use the scientific method to explore whether gravity exists.

1. *Pose a question.* Does gravity exist?

2. *State a hypothesis.* Gravity does exist. That is, there exists a force that tends to propel objects toward the surface of the earth.

3. *Make a prediction.* We predict that if we push a textbook horizontally off the side of a table, it will be propelled by gravity toward the floor.

4. *Test.* Push a textbook off a table and observe whether it falls or not.

5. *Analyze.* If the book hovers in place, we reject the hypothesis that gravity exists; stated more succinctly, gravity does not exist. If it does fall, we have validated the hypothesis that gravity *does* exist.

We can also apply this approach to the analysis of algorithms.

1. *Pose a question.* What is the efficiency class of the `naive_sum` algorithm?

2. *State a hypothesis.* `naive_sum` runs in $O(n)$ time.

3. *Make a prediction.* We predict that if we implement `naive_sum` and measure its elapsed running time for various input sizes n, the points will fit a straight line $T(n) = c \cdot n$ for some constant slope c.

4. *Test.* Implement the algorithm in a real programming language such as Python or C, measure its performance on several inputs of varying sizes n, and draw the samples on a scatter plot.

5. *Analyze.* Analyze the scatter plot visually or statistically and decide whether the points fit, or do not fit, a straight line.

Experimental analyses always follow this same pattern: we pick a specific algorithm, implement it, observe its running time for many inputs under controlled conditions, and determine which of the common efficiency classes best fits the observed run times.

3.5.2 Measuring Elapsed Time

Measuring the elapsed run time of computer programs accurately and precisely is not easy. Time needs to be observed with a physical measurement instrument such as a stopwatch or the hardware clock built into a computer. Physical measurements always involve some *instrumental error,* which is inaccuracy introduced by imperfections in the measurement apparatus.

In order to minimize instrumental error, we should use a clock with a high enough resolution to capture variations in program execution time. Modern computers run at clock speeds measured in units of gigahertz, so they can execute billions of instructions per second. So a clock that measures units of whole seconds is not precise enough. We must insist on an instrument that measures time at

far finer gradations than whole seconds; milliseconds, microseconds, or even nanoseconds are preferable. The Python standard library includes the `time.perf_counter` function, which is designed specifically to measure the execution time of code precisely. Other programming environments typically provide a high resolution clock mechanism similar in function to `time.perf_counter`.

We must also do our best to make sure that our time measurements represent only the run time of our algorithm and not other computations. Modern operating systems provide time sharing, so elapsed wall clock time could include not only time spent on the measured code, but also time spent dealing with other programs running on the computer. We should do our best to perform timing experiments while running as few other programs as possible, and certainly not while multitasking with intensive operation such as playing a game.

Finally, we can mitigate measurement noise by performing several trials for each value n and taking the corresponding $T(n)$ to be the average of the observed times.

3.5.3 Benchmarks

In order to conduct these experiments we need many problem instances with known sizes.

For the experimental results to be relevant to real world situations, we need the instances to represent the kinds of inputs that will be encountered in reality. For instance, text processing algorithms will probably need to deal with things like blocks of English words, human names, and street addresses. They are unlikely to involve a lot of processing of gibberish, or strings full of one character, or other idiosyncratic inputs. So, some care needs to go into selecting inputs that are representative of reality.

Sometimes a community that is concerned with consistent timings will create a *benchmark* which is a documented, specific set of inputs and measurement settings that will be used to measure the performance of competing algorithms or programs. For example, the Computer Language Benchmarks Game [3] is a collection of algorithms that are intended to highlight the performance of different programming languages. That project has implemented the same collection of algorithms in several competing languages and measured the efficiency of those algorithms empirically, with the goal of characterizing the relative speed of programming languages.

3.5.4 Random Problem Instances

Creating a thorough and representative benchmark is a lot of work. An alternative is to create an algorithm that generates, for any input size n, a randomly generated problem instance of size n. While convenient, doing this well is not easy! Random problem instances may not correspond to the kinds of inputs your algorithm will see in practice. Real data tends to have patterns and special cases. For instance, a convenient way to generate random strings is to select printable characters from a uniform distribution. However, the letter e appears more frequently than any other in real English text. It is quite possible that algorithms would behave differently on strings whose characters come from a uniform distribution than they would given characters from a nonuniform

one. This is but one example of the challenges of generating realistic problem instances randomly.

3.6 Mathematical Analysis

Mathematical analysis is an alternative to empirical analysis. The key difference between the two approaches is the epistemology used for making claims. Empirical analyses use the scientific method of hypothesis, experiment, and analysis. By contrast mathematical analysis uses the mathematical method of lemma and proof.

Mathematical proofs are attractive since they are *permanent.* If you prove something once, it is true now and forever. So proofs seem superior to empirical results in terms of long-term relevance. However, we need to be careful to prove lemmas that will remain relevant forever. We don't want to prove fleeting claims such as "This algorithm runs in $O(n)$ time when implemented in Pascal and run on an Intel 486 CPU." We would rather prove enduring claims such as "any reasonable implementation of this algorithm runs in $O(n)$ time."

In order to prove the latter kind of lemma, we need a crisp and timeless mathematical definition of "reasonable implementation." A *computational model* is a mathematical model that specifies how the resources used by an algorithm are quantified for the purpose of analysis. Computational models are man-made concepts, so we have the flexibility to adapt them to accommodate differing goals.

As with an empirical analysis, a mathematical analysis may be broken down into a few steps:

1. Pick a computational model.
2. Examine the algorithm's pseudocode and derive a complexity function T for the algorithm's time complexity, and (optionally) S for its space complexity.
3. Prove which efficiency class T and S each belong to.

3.6.1 The Standard Model

The *Random Access Machine (RAM) model* is an attempt to model the CPU time and memory resources used by algorithms when implemented on modern, realistic computer hardware [6]. The model has become so popular that it is sometimes called the *standard model;* we prefer this term since the acronym "RAM" is widely understood to refer to Random Access Memory hardware. The standard model represents a computer as a CPU that executes discrete instructions coupled with a random-access memory divided into discrete *words.*

Definition 16. *In the standard model, a* computer *has the following properties:*

1. Memory. *A computer has a* memory *consisting of an array of* words. *Each word is identified by an* address, *which is a non-negative integer. The number of words is finite, but unbounded*

and presumed to be large enough to accommodate any input, output, and temporary storage used by our algorithms.

2. Words. *Each computer has a constant* word size $W > 1$*, and every word in memory is a string of W bits. Each word may be used to represent an address, W-bit integer, floating point number, Boolean value, or character.*

3. Instructions. *The computer executes programs by repeatedly reading an* instruction *from memory, decoding its meaning, and performing the corresponding computation. Any instruction may be encoded in a constant number of words.*

4. Complete instruction set. *The computer can execute instructions for: copying words; Boolean logic (* `and`*,* `or`*,* `not`*, etc.); arithmetic on single-word integer and floating point numbers; comparisons; branches (* `if`*,* `goto` *); and calling and returning from functions.*

5. Random access. *Reading or writing the word at address i takes a constant number of instructions regardless of the value of i.*

6. Primitive instructions. *Executing any instruction involves reading a constant number of words, then writing a constant number of words. The value of each written bit may be computed by a Boolean circuit that takes the read bits as inputs.*

Properties 1, 2, and 3 should be unsurprising. A computer in the standard model has a memory organized as an indexed array of fixed-size words. The word size W is intended to represent the native register width of the CPU architecture; a 64-bit AMD64 CPU would have $W = 64$, while a 32-bit ARM CPU would have $W = 32$. The computer operates by executing instructions one at a time. Each instruction involves reading words from memory, then writing some result to memory. Our computer follows the Von Neumann architecture, meaning that it stores instructions and data in the same memory.

Property 4 ensures that a computer's instruction set is sufficient to implement general purpose algorithms. It would be difficult, if not impossible, to implement practical algorithms on a machine lacking any of these kinds of instructions.

Property 5 is where Random Access comes in. It says that the number of instructions needed to read or write a word is unrelated to that word's address. This rules out linear access memory architectures, such as a tape that might need to be fast-forwarded or rewound i times to reach address i.

Property 6 is perhaps the most technical of our properties. Intuitively, it ensures that each instruction represents a computation that can actually be accomplished by digital circuitry. For this to be true, we require that an individual instruction involve reading and writing only a constant number of words. Otherwise, a single instruction could encompass a task such as initializing n words, a process which ought to take time proportional to n.

Further, we require that each individual bit of output may be computed by a *Boolean circuit.* As you may know, conventional CPUs are built from circuits involving transistors. The binary values 1 and 0 are represented by the presence or absence of electricity on microscopic wires. Any of the Boolean

operators (`and` , etc.) may be built out of a cluster of a few transistors wired together. Higher-level operations, such as arithmetic and branches, are built out of many instances of the fundamental Boolean operators. The details of all this constitute the subject of computer architecture, and are beyond the scope of this text. For now it suffices to say that, if each output bit of an instruction can be defined by a Boolean circuit, then that instruction can be executed by computer hardware. We will revisit this property of instructions in our later treatment of NP-complete problems.

Definition 17. *The unit of measure of time in the standard model is one* step. *Any constant number of CPU instructions may be counted as one step.*

Definition 18. *The unit of measure of space in the standard model is one* word.

These definitions make analyzing the time complexity of algorithms rather simple. Any piece of pseudocode that corresponds to any constant number of CPU instructions may be counted as a single step. So most single lines of pseudocode can count as 1 step. Loops and function calls are exceptions to this rule, since they may involve iteration and hence more than a constant number of steps.

3.6.2 Other Computational Models

All the analyses in this text will use the standard model; this section briefly surveys a few other computational models in the sake of breadth of knowledge.

Turing Machine

A *Turing machine* is an early computational model first defined by Alan Turing, a pioneer of computer science [51] [8]. The Turing machine is modeled after the kind of computer hardware that existed in Turing's time circa World War II. Its program is permanently hard-coded in a finite state machine; its data store is a separate, non-random-access *tape* which can only read, write, or advance one character at a time.

The standard model certainly captures the essence of modern computers better than does the Turing machine; but, the Turing machine model is significantly simpler. This simplicity facilitates proving facts about the capabilities and limitations of Turing machines, since there are fewer "moving parts" and special cases that need be considered. So Turing machines are the model of choice in the field of computational complexity, which deals with establishing and categorizing the limits of computational models.

Lambda Calculus

The *lambda calculus (λ calculus)* is a computational model defined entirely in terms of mathematical functions [14] [4]. It was developed by Alonzo Church concurrently with Turing's development of the Turing machine. In the lambda calculus, bits of data are represented by two distinguishable

functions, one for 1 and another for 0. Object constructors and getters can also be synthesized out of pure functions. Bits and objects may be used to implement linked list nodes, and with those it is possible to implement the full suite of conventional data structures including numbers, characters, strings, trees, and so on.

The lambda calculus is notable since it is a complete model that views all computational activity, including data storage, as a special case of function evaluation. This perspective stands apart from that of the standard model and Turing machine model, which define computation in terms of electronic machinery that store and manipulate those bits. The lambda calculus is the mathematical underpinning of functional-paradigm programming languages, such as those in the Lisp and ML families. This influence explains many aspects of those languages, such as why they have more affinity for linked lists than for arrays.

3.6.3 Chronological Step Counting

In order to analyze the efficiency of an algorithm in the standard model, we need to examine the algorithm's pseudocode and somehow derive a mathematical function $T(n)$, in terms of the relevant problem's size measure n, for the number of standard-model steps executed by the algorithm. (Or, similarly, a multivariate function $T(n_0, \ldots, n_{k-1})$ if the problem at hand has multiple size measures.)

A straightforward way of doing this is to mentally step through the execution of the algorithm, line by line, and keep a running tally of the number of steps that have been executed as a function of the problem's size measures. We call this method *chronological step counting* since it involves counting algorithm steps in the same order that the algorithm executes them.

Our Python-influenced pseudocode involves only a few broad categories of statements, each of which has a simple rule for the number of steps it entails, with the notable exception of recursive function calls. For now we will consider only non-recursive pseudocode; we will deal with recursive algorithms in Chapter 8.

Single-step statements

Each line of pseudocode that gets executed counts for at least 1 step. If the line involves a loop, function call, or list initialization, it may cost more than one step. Lines that do not involve any of these count as exactly 1 step.

For example, each line of pseudocode in the body of `arithmetic` counts as 1 step.

```
def arithmetic(lst):
    x = lst[0]
    y = lst[1]
    lst[2] = x + y
    return lst[0] / lst[1]
```

Since the body of this pseudocode has four lines of pseudocode, and each corresponds to one step, its time complexity is

$$T(n) = 4.$$

List initialization

The Python syntax to create a list of k copies of the value X is:

```
[X] * k
```

For example, the expression

```
[0] * 1000
```

creates a list filled with one thousand zeroes. The number of steps in this kind of statement is proportional to the length of the list, so the time complexity of the statement `[X] * k` is $T(k) = k$.

List slice

A *slice* of list `L` is a sub-list containing only a range of the indices of `L`. Creating a slice with k elements takes k steps, just like initializing a fresh list. Again we adopt the Python convention that `L[i:j]` denotes the sub-list starting at index `i`, and up to *but not including* `L[j]`. The start index `i` may be omitted, in which case the start defaults to 0. Likewise, the end index `j` may be omitted, in which case it defaults to the length of `L`. All this notation is summarized in Figure 3.1.

Notation	Slice Elements	# Elements	Time
`L[i:j]`	`[ L[i], L[i+1], ..., L[j-1] ]`	$j - i$	$O(j - i)$
`L[:j]`	`[ L[0], L[1], ..., L[j-1] ]`	j	$O(j)$
`L[i:]`	`[ L[i], L[i+1], ..., L[n-1] ]`	$n - i$	$O(n - i)$

Figure 3.1: List slice notation and time complexity.

`for` loops

Python's `for` syntax follows the pattern

```
for <VARIABLE> in <SEQUENCE>:
    <BODY>
```

For example, the following code prints every element of the list `lst`:

```
for x in lst:
    print(x)
```

Often the `range(n)` function is used to iterate through the sequence of integers $\langle 0, 1, \ldots, n-1 \rangle$, for instance

```
for i in range(n):
    print(i)

for i in range(len(lst)):
    print(lst[i])
```

The total number of steps executed by a `for` loop is the sum of the number of steps executed in each individual iteration. More formally,

$$T(n) = \sum_{x \in X} t_x$$

where X is the set of elements that are iterated over, $x \in S$ is an arbitrary element, t_x is the number of steps taken during the single loop iteration for x, and $T(n)$ is the total number of steps executed by the `for` loop.

For the sake of clarity, we will say that one loop iteration takes as many steps as are in the body of the loop, *plus 1 additional step for loop overhead.* The overhead step accounts for dealing with the loop counter and testing whether the loop has terminated, i.e. the `<VARIABLE> in <SEQUENCE>` part of the `for` syntax.

For instance, the number of steps executed by the pseudocode

```
total = 0
    for i in range(n):
        total += i
```

is

$$1 + \sum_{i=0}^{n-1} 2.$$

The 1+ term comes from the `total = 0` line. The `for` loop iterates over $i = 0, 1, \ldots, n-1$, which is why the summation goes from $i = 0$ through $i = n - 1$. The body of the loop involves the single step `total += i` , and each iteration involves one additional step to deal with the `i in range(n)` expression.

Non-recursive function calls

The cost of evaluating a function call is the number of steps executed by the body of that function. In order to arrive at this figure, the body of the called function must be analyzed separately using the usual process.

For example, consider the following pseudocode, which involves a function `setup` that calls a helper function `make_list` .

```
def setup():
    forward = make_list(20)
    backward = make_list(30)
    return forward, backward

def make_list(n):
    L = []  # an empty list
    for i in range(n):
      L.append(0)
    return L
```

The time complexity of `setup` is

$$T(n) = 1 + T_{ml}(20) + 1 + T_{ml}(30) + 1$$

where $T_{ml}(n)$ is the time complexity of a call to the `make_list` function. Each of the lines involving function calls costs 1 step as usual, plus the cost of the function call, whatever that is. The final +1 term accounts for the `return` statement.

In order to simplify this equation, we need to work out the definition of $T_{ml}(n)$. The `L = []` statement is 1 step. The `for` loop iterates exactly n times, and each iteration involves 2 steps. Finally the `return` statement is 1 step, so the total time complexity of `make_list` is

$$\begin{aligned} T_{ml}(n) &= 1 + 2n + 1 \\ &= 2n + 2 \end{aligned}$$

Putting these definitions together,

$$\begin{aligned} T(n) &= 1 + T_{ml}(20) + 1 + T_{ml}(30) + 1 \\ &= T_{ml}(20) + T_{ml}(30) + 3 \\ &= (2(20) + 2) + (2(30) + 2) + 3 \\ &= 107. \end{aligned}$$

`while` loops

The total number of steps taken by a `while` loop is the sum of the steps of each individual iteration, just like a `for` loop. However, the number of iterations and steps in each iteration are often more difficult to ascertain for `while` loops than they are with `for` loops. This is because every `while` loop uses its own distinctive termination condition.

Analyzing some `while` loops is straightforward, for example the following.

```
total = 0
while total < 100:
    total += 2
```

During the execution of the loop, `total` will take on the values `0, 2, 4, ..., 98`. This sequence has 50 elements (confirm that for yourself!), so the loop iterates exactly 50 times. Each iteration involves one step of overhead to evaluate `total < 100` and jump appropriately and another single-step statement `total += 2`, so the total time complexity of this pseudocode is

$$T(n) = 1 + \sum_{i=0}^{50} 2.$$

Note that the variable `total` in the pseudocode is related to, but not identical to, the variable i used in our math expression for the time complexity of the pseudocode. Our pseudocode, and the math expressions used to analyze it, exist in separate namespaces, so they are free to use different variable names to denote the same concept.

The following `while` loop is easy to write, but tricky to analyze.

```
def tricky_while_loop():
    total = 0
    i = 1
    j = 1
    while total < 100:
        i += j
        j += 1
        total += i + j
```

Deriving a summation expression for the number of steps executed by this pseudocode would not be straightforward!

Since every `while` loop iterates in its own way, there is no general-purpose rule for counting steps executed by arbitrary `while` loops. Instead, we must apply ingenuity to analyzing each `while` loop. Since `for` loops are simpler to analyze, we prefer `for` loops over `while` loops in our pseudocode.

`if` statements

Python includes only one kind of branch statement, `if` . An `if` statement may optionally include any number of `elif` blocks, and may include one final `else` block.

All of the following are valid Python `if` statements.

```
if n > 0:
    print('positive')

if n >= 0:
    print('nonnegative')
else:
    print('negative')

if n == 0:
    print('none')
elif n == 1:
    print('once')
elif n == 2:
    print('twice')
elif n == 3:
    print('thrice')
else:
    print(str(n) + ' times')
```

Branch statements complicate our analysis since some execution paths may be slower or faster than others. This ambiguity is resolved by our earlier decision to consider the *worst case* complexity of algorithms. Since we are analyzing for the worst case, we treat the number of steps used by a branch as the worst (greatest) number of steps among any of the alternatives. Mathematically,

$$T(n) = \max(t_{true}, t_{elif_0}, t_{elif_1}, \ldots, t_{false})$$

where t_{true} is the number of steps executed in the affirmative `if` case; each t_{elif_i} is the number of steps executed in `elif` case i, or zero if no such `elif` block is present; t_{false} is the number of steps executed in the `else` case, or zero if there is no `else` case; and t_{if} is the worst case number of steps executed by the entire `if` / `elif` / `else` construct. Evaluating an `if`, `elif`, or `else` line costs one step, to account for evaluating the conditional expression and possibly executing a jump instruction.

Example 6. *Suppose that a* `print` *function call takes 1 step when passed a literal string. The first* `if` *statement above (* `'positive'` *) takes two steps. The second* `if` *statement (* `'nonnegative'`, `'negative'` *) takes* $\max(2, 3) = 3$ *steps. The last* `if` *statement (* `'none'`*, etc.) takes* $\max(2, 3, 4, 5, 6) = 6$ *steps.*

Now consider the following.

```
if n == 0:
    total = 0
else:
    total = 0
    for x in lst:
        total += x
```

When n is zero, this pseudocode executes $t_{if} = 2$ steps; one to evaluate `if n == 0`, and a second to execute `total = 0`. In the `else` case, the pseudocode executes $t_{else} = 3 + 2n$ steps:

- testing whether $n == 0$ takes one step;
- jumping to the `else` line takes one step;
- executing `total = 0` takes one step;
- the `for` loop iterates exactly n times; and
- each `for` loop iteration involves one step to update `total += x`, and a second for loop overhead.

So the number of steps taken by the entire `if ... else` statement is

$$\begin{aligned} t_{in} &= \max(t_{true}, t_{false}) \\ &= \max(2, 3+2n) \\ &= 3+2n. \end{aligned}$$

3.6.4 Proving Efficiency Classes by Induction

We now know how to translate an algorithm's pseudocode into a time complexity function such as $T(n) = 2n + 2$ or $T(n) = 4n + 2$. This is half the battle of mathematical analysis; the other half is proving that such a complexity function fits into a particular efficiency class such as $O(n)$. Since we are using the mathematical method of argument, we must pose a lemma and provide a sound mathematical proof that the lemma is true.

The most fundamental way of proving that a complexity function fits into an efficiency class is by proving that the function is a member of that efficiency class, according to the definition of O and first principles. Recall that a univariate efficiency class is defined as

$O(f(n)) = \{g(n) \mid$ there exists some constants $c > 0$ and $t \geq 0$ such that $g(n) \leq c{\cdot}f(n)$ whenever $n \geq t\}$.

We can prove that our $T(n)$ is a member of a class $O(f(n))$ by carefully picking a particular pair of constants c and t, and showing that $T(n) \leq c \cdot f(n)$ for any $n \geq t$. The variable n is a size measure, so it is a non-negative integer; so we can argue that the inequality $T(n) \leq c \cdot f(n)$ holds for $n \geq t$ by induction on n.

To reiterate, the process of proving efficiency class membership by induction is

1. Make an informed guess about which efficiency class $O(f(n))$ to use.
2. Use algebra to solve for constant values c and t that seem likely to work.
3. Prove the base case, that $T(n) \leq c \cdot f(n)$ when $n = t$.
4. Prove the inductive step, that for any $n > t$, $T(n) \leq c \cdot f(n)$ implies $T(n+1) \leq c \cdot f(n+1)$.
5. Conclude that $T(n) \in O(f(n))$.

Example 7. *Let us prove an efficiency class for the time complexity function $T(n) = 2n + 2$ that we derived in the previous section. We will work through steps 1 through 4 as listed above.*

1. *$T(n) = 2n + 2$ seems to resemble the linear efficiency class $O(n)$ more than any other.*
2. *We want to find a c such that*

$$T(n) \leq c \cdot f(n)$$

for large n. We have $T(n) = 2n + 2$ and $f(n) = n$, so we need

$$2n + 2 \leq c \cdot n.$$

Solving for c,

$$c \geq \frac{2n+2}{n} = 2 + 2/n.$$

The term $2/n$ is undefined when $n = 0$, so already we can see that we need $t \geq 1$. We pick $t = 1$. The term $2/n$ decreases with n, and is at most $2/1 = 2$ when $n = t \equiv 1$. So the constants $c = 2 + 2 = 4$ and $t = 1$ seem like they should work.

3. *Base case.*

   Lemma 1. *$T(n) \leq c \cdot f(n)$ when $n = t$.*

   Proof. By substitution

   $$t(n) = 2n + 2 = 2(1) + 2 = 4$$

   and

   $$c \cdot f(n) = (4)(n) = 4(1) = 4.$$

   $4 \leq 4$, so the lemma holds. □

4. *Inductive step.*

   Lemma 2. *If $n > t$ and $T(n) \leq c \cdot f(n)$, then $T(n+1) \leq c \cdot f(n+1)$.*

   Proof. By the inductive hypothesis

   $$\begin{aligned} T(n) &\leq c \cdot f(n) \\ (2n+2) &\leq (4)(n) \\ +2 &\quad +4 \\ 2n+4 &\leq 4n+4 \\ 2(n+1)+2 &\leq 4(n+1) \\ T(n+1) &\leq c \cdot f(n+1). \end{aligned}$$

   □

5. *Conclusion.*

   Corollary 1. *$2n + 2 \in O(n)$.*

   Proof. This follows from the definition of O and the previous two lemmas. □

3.6.5 Proving Efficiency Classes with Limits

While it may be possible to prove efficiency class membership using induction, that approach is rather tedious and error prone. The notion of growth trends that apply only to large input sizes is closely related to that of limits as input sizes approach infinity. It turns out that we can use limits directly to prove efficiency class containment, and that this approach is substantially simpler and easier than using induction. First, recall the definition of a limit approaching infinity.

Definition 19. *If $F(x)$ is a real-valued function, then* the limit of F as x approaches infinity is L, *written*

$$\lim_{x\to\infty} F(x) = L,$$

means that for any $\varepsilon > 0$, there exists k such that $|F(x) - L| < \varepsilon$ whenever $x > k$.

Theorem 1. *If T and f are univariate complexity functions, $f(n) > 0$, and*

$$\lim_{n\to\infty} \frac{T(n)}{f(n)} = L$$

where L is non-negative and constant with respect to n, then

$$T(n) \in O(f(n)).$$

Proof. Let $F(n) \equiv \frac{T(n)}{f(n)}$. By definition T and f are complexity functions, so $T(n)$ and $f(n)$ are each non-negative and real numbers. Further, $f(n) > 0$, so $\frac{T(n)}{f(n)}$ is defined and real, and our $F(n)$ is real-valued.

Choose $\varepsilon = L + 1$. By definition $L \geq 0$, and by construction $\varepsilon > L$, so $\varepsilon > 0$.

So for our choices of F and ε, we have

$$\lim_{n\to\infty} F(n) = L$$

with $\varepsilon > 0$. Thus by the definition of a limit approaching infinity, there exists some k such that $|F(n) - L| < \varepsilon$ for all $n > k$.

The expression $|F(n) - L|$ involves an absolute value, and of course either $\big(F(n) - L\big)$ is negative or it is non-negative. First suppose $\big(F(n) - L\big)$ is negative; then

$$\begin{aligned} F(n) - L &< 0 && \forall n > k \\ F(n) &< L && \forall n > k \end{aligned}$$

and substituting our definition for F,

$$\begin{aligned} \Big(\frac{T(n)}{f(n)}\Big) &< L && \forall n > k \\ T(n) &< L \cdot f(n) && \forall n > k \end{aligned}$$

or in terms of non-strict inequalities

$$T(n) \leq L \cdot f(n) \qquad \forall n \geq k+1.$$

Therefore choosing $c \equiv L$ and $t \equiv k+1$, we have

$$T(n) \leq c \cdot f(n) \qquad \forall n \geq t$$

and by the definition of O, $T(n) \in O(f(n))$.

Now suppose $\big(F(n)-L\big)$ is non-negative. Then $|F(n)-L| = F(n)-L$, and recall that $|F(n)-L| < \varepsilon$ when $n > k$. So

$$F(n) - L < \varepsilon \qquad \forall n > k.$$

Substituting our definitions for F and ε,

$$\begin{aligned} \Big(\frac{T(n)}{f(n)}\Big) - L &< (L+1) && \forall n > k \\ \frac{T(n)}{f(n)} &< 2L+1 && \forall n > k \\ T(n) &< (2L+1) \cdot f(n) && \forall n > k \end{aligned}$$

or in terms of non-strict inequalities,

$$T(n) \leq (2L+1) \cdot f(n) \qquad \forall n \geq k+1.$$

Therefore choosing $c \equiv 2L+1$ and $t \equiv k+1$, we have

$$T(n) \leq c \cdot f(n) \qquad \forall n \geq t$$

and again by the definition of O, $T(n) \in O(f(n))$. □

The process of proving which efficiency class a complexity function $f(n)$ belongs to, using limits, is

1. Make an informed guess about which efficiency class $O(f(n))$ to use, where $f(n) > 0$.
2. Take the limit
$$\lim_{n\to\infty} \frac{T(n)}{f(n)}.$$
3. If the limit is constant with respect to n and non-negative, conclude by Theorem 1 that $T(n) \in O(f(n))$.
4. Otherwise, try again with a different efficiency class.

Example 8. *We now reprove that $T(n) = 2n+2 \in O(n)$, this time using limits.*

Lemma 3. $2n+2 \in O(n)$.

Proof. The limit

$$\lim_{n\to\infty} \frac{T(n)}{f(n)} = \lim_{n\to\infty} \frac{(2n+2)}{(n)} = \lim_{n\to\infty} \frac{2n}{n} + \lim_{n\to\infty} \frac{2}{n} = 2 + 0 = 2$$

which is non-negative and constant with respect to n. Therefore $2n + 2 \in O(n)$. □

This proof is quite a bit shorter than the last one!

3.6.6 Proving Efficiency Classes with Properties of O

Theorem 1 can be used to establish several other properties of O that make efficiency classes easy to deal with.

Trivial efficiency classes

Lemma 4. *For any complexity functions $T(n)$ and $f(n)$ with $T(n) \leq f(n)$, $T(n) \in O(f(n))$.*

Proof. If $T(n) < f(n)$ then

$$\lim_{n\to\infty} \frac{T(n)}{f(n)} = 0;$$

otherwise $T(n) = f(n)$ and

$$\lim_{n\to\infty} \frac{T(n)}{f(n)} = 1.$$

In either case, the limit is a non-negative and constant with respect to n, so by Theorem 1, $T(n) \in O(f(n))$. □

Example 9. $n \in O(n)$.

Example 10. $3n \in O(4n)$.

Example 11. $17 \in O(18)$.

Example 12. $n \in O(n^2)$.

Example 13. $n^3 \in O(n^3)$.

Dropping multiplicative constants

We wish to prove that constant multiplicative factors, such as the factors of 4 and 7 in the expression $O(4n^2 + 7n)$, may be eliminated from efficiency class notation. This argument involves two degrees of freedom: first, that a constant c may be dropped regardless of whether it increases ($c > 1$) or decreases ($c < 1$) the function; and second, that constants may be either introduced or eliminated. The next lemma handles the first issue.

Lemma 5. *For any complexity function $f(n)$ and constant $k > 0$,*

$$O(f(n)) \subseteq O(k \cdot f(n)).$$

Proof. Let $g(n)$ be an arbitrary function in $O(f(n))$. Then by the definition of O, there exist constants c and t such that

$$g(n) \leq c \cdot f(n)$$

for any $n \geq t$.

Let $\varepsilon = \max(1, 1/k)$, and note that ε is a constant, $\varepsilon > 0$, and $(k \cdot \varepsilon) \geq 1$. So by substitution

$$\begin{aligned} g(n) &\leq c \cdot f(n) \\ &\leq (k \cdot \varepsilon) \cdot c \cdot f(n) \\ \frac{g(n)}{k \cdot f(n)} &\leq \varepsilon \cdot c \\ \lim_{n \to \infty} \left(\frac{g(n)}{k \cdot f(n)}\right) &\leq \varepsilon \cdot c \end{aligned}$$

and by Theorem 1, $g(n) \in O(k \cdot f(n))$. Thus any arbitrary element of $O(f(n))$ is also an element of $O(k \cdot f(n))$, and $O(f(n)) \subseteq O(k \cdot f(n))$. □

With that mathematical plumbing in place, we can prove that constants may be freely introduced or eliminated.

Lemma 6. *For any complexity function $f(n)$ and constant $k > 0$,*

$$O(k \cdot f(n)) = O(f(n)).$$

Proof. By direct application of Lemma 5,

$$O(f(n)) \subseteq O(k \cdot f(n)).$$

Now let $f'(n) = k \cdot f(n)$ and $k' = \frac{1}{k}$. Then, again invoking Lemma 5,

$$\begin{aligned} O(f'(n)) &\subseteq O(k' \cdot f'(n)) \\ O(k \cdot f(n)) &\subseteq O((\frac{1}{k})(k \cdot f(n))) \\ &\subseteq O(f(n)). \end{aligned}$$

So we have $O(f(n)) \subseteq O(k \cdot f(n))$ and $O(k \cdot f(n)) \subseteq O(f(n))$, so $O(k \cdot f(n)) = O(f(n))$. □

Example 14. $O(2n) = O(n)$.

Example 15. $O(8) = O(1)$.

Dropping dominated terms

Lemma 7. *For any complexity functions $f_0(n)$ and $f_1(n)$,*

$$O(f_0(n) + f_1(n)) = O(\max(f_0(n), f_1(n))).$$

Proof. By the definition of maximum,

$$f_0(n) \leq \max(f_0(n), f_1(n))$$

and

$$f_1(n) \leq \max(f_0(n), f_1(n)).$$

Adding these inequalities, we obtain

$$\begin{aligned} f_0(n) + f_1(n) &\leq \max(f_0(n), f_1(n)) + \max(f_0(n), f_1(n)) \\ &\leq 2 \cdot \max(f_0(n), f_1(n)). \end{aligned}$$

So by Lemmas 4 and 6,

$$O(f_0(n) + f_1(n)) = O(2 \cdot \max(f_0(n), f_1(n))) = O(\max(f_0(n), f_1(n))).$$

□

Example 16. $O(n + n^2) = O(\max(n, n^2)) = O(n^2)$.

Example 17. $O(2^n + n) = O(\max(2^n, n)) = O(2^n)$.

Corollary 2. *For any $k > 1$ complexity functions $f_0(n), \ldots, f_{k-1}(n)$,*

$$O(f_0(n) + \ldots + f_{k-1}(n)) = O(\max_{0 \leq i < k} f_i(n)).$$

Example 18. $O(3 + 2n^2 + 7n) = O(\max(3, 2n^2, 7n)) = O(2n^2)$.

Dropping additive constants

Corollary 3. *For any complexity function $f(n)$ and constant c,*

$$O(f(n) + c) = O(f(n)).$$

Example 19. $O(n + 3) = O(n)$.

Example 20. $O(n - 2) = O(n)$.

Dropping floor and ceiling operators

Lemma 8. *For any complexity function $f(n)$*

$$O(\lfloor f(n) \rfloor) = O(\lceil f(n) \rceil) = O(f(n)).$$

Example 21. $O(\lceil n/2 \rceil) = O(n/2)$.

A proof of this lemma is left as an exercise.

We now prove $2n + 2 \in O(n)$ one last time, this time with properties of O.

Lemma 9. $2n + 2 \in O(n)$.

Proof. By properties of O,

$$\begin{array}{rcll} 2n+2 & \in & O(2n+2) & \text{(trivial)} \\ & = & O(2n) & \text{(dominated term)} \\ & = & O(n) & \text{(constant factor).} \end{array}$$

□

Summary

In summary, the process of proving which efficiency class $O(f(n))$ a complexity function $T(n)$ belongs to, using properties of O, is

1. State that $T(n) \in O(T(n))$.
2. Use properties of O to manipulate the expression $O(T(n))$ until it becomes one of the common efficiency classes, or another simplified efficiency class.

Example 22.

Lemma 10. $7n^2 + 5n + 6 \in O(n^2)$.

Proof. By properties of O,

$$\begin{array}{rcll} 7n^2+5n+6 & \in & O(7n^2+5n+6) & \text{(trivial)} \\ & = & O(\max(7n^2, 5n, 6)) & \text{(dominated terms)} \\ & = & O(7n^2) & \\ & = & O(n^2) & \text{(constant factor).} \end{array}$$

□

3.7 Amortized Analysis

Chronological step counting works perfectly well for many algorithms. However, in some cases it may *overcount* the number of steps. This happens when our worst case mindset causes us to say that an expensive operation happens much more frequently than it actually does.

We deal with this situation using the concept of *amortization* borrowed from the discipline of financial accounting. Organizations often have "lumpy" overhead costs. For instance, suppose for the sake of discussion that a university classroom can be used for 8 years before it needs to be refurbished with paint and new furniture at a cost of $20,000. A large university might have hundreds of classrooms, and budgeting for each individual classroom's refurbishment on a case-by-case basis would be onerous. Instead, it is commonplace to calculate the *amortized cost* of improvements like this. In our example, the amortized cost of keeping a classroom in working order is

$$\begin{aligned} & \frac{\text{refurbishment cost}}{\text{period between refurbishments}} \\ = & \frac{\$20,000}{8 \text{ years}} \\ = & \ 2,500 \text{ dollars/year.} \end{aligned}$$

So if the university allocates $2,500 per year per classroom, they should have enough money to cover refurbishments in the long run.

This kind of "lumpy" cost arises in algorithm analysis when an algorithm involves a data structure that involves a one-time construction cost, or a periodic upkeep cost. For example, consider the following algorithm for maintaining a log of strings in a partially-filled array.

```
log_list = []
log_length = 0
def log_message(message):
    if log_list is []:
        log_list = [None] * 1000
    log_list[log_length] = message
    log_length += 1
```

Ordinarily each call to `log_message` involves assigning one array element and incrementing `log_length`, which takes only $2 \in O(1)$ steps. However the very first call is an exception; it creates a 1,000-element array, which takes 1,000 steps. Following our chronological step counting process, we analyze for the worst case when `log_list is []`, and conclude that `log_message` takes 1,002 steps. But that is a severe overcount.

To arrive at a more accurate step count, we observe that the slow array creation only ever happens once. The very first `log_message` call will indeed take 1,002 steps. But the algorithm's branch logic prevents that from ever happening again; every subsequent call will take only 2 steps. Since

the array's length is hard-coded to 1,000, we know that `log_message` may be called at most 1,000 times. When `log_message` is indeed called 1,000 times, the average number of steps is

$$\begin{aligned} & \frac{(\text{steps in first slow call}) + (\text{number of fast calls}) \times (\text{steps per fast call})}{\text{total steps}} \\ = & \frac{1,002 + (999) \times (2)}{1000} \\ = & \ 3.001 \end{aligned}$$

steps. This seems like a fairer appraisal of how long a call to `log_message` takes. Each call takes a little bit more than 2 steps; the extra .001 time accounts for the one-time cost of creating the list, averaged out over many calls. We call this 2.001 number the *amortized* time complexity of the algorithm, because it represents a per-operation cost that is averaged out over a mixture of fast and slow operations, instead of an absolute worst-case cost.

3.7.1 The Accounting Method

The *accounting method* is one approach to proving the amortized efficiency of one or more related algorithmic operations. In this method, we define a *token* as a unit of currency corresponding to $O(1)$ time, and envision a conceptual *bank account* of tokens. We carefully choose a *price,* in units of tokens, to charge for each kind of operation. This price is the advertised amortized time complexity of the operation. In truth, some operations may take fewer steps than their price; these operations earn a conceptual *profit* that accumulates in our bank account. Other operations may take more steps than their price; these operations run a *deficit*. Finally, some operations may break even with no net effect on the bank account. If we are able to show that our prices ensure that the bank account is always non-negative (i.e. we never go bankrupt), then those prices represent sustainable amortized costs for the operations in question.

Let us consider a more powerful log data structure that "grows" its array whenever it becomes over-full.

```
class Log:
    def __init__(self):
        self.array = [None, ]
        self.count = 0
    def record(self, message):
        if len(self.array) == self.count:
            new_array = [None] * (self.count * 2)
            for i in range(self.count):
                new_array[i] = self.array[i]
            self.array = new_array
        self.array[self.count] = message
        self.count += 1
```

The class constructor `__init__` creates an initial array of length 1, with a count of 0 valid messages. The `record` operation has an `if` statement that first determines whether the array is currently full. When full, the body of the `if` creates a new array that is twice as long as the current one, copies all the elements over to the new array, and makes the new array object take the place of the old one. Then, regardless of whether the array grew, the message is added to the array and the message count is incremented.

Aside from the `if` statement, the `record` algorithm takes $3 \in O(1)$ time. Rarely, the array is full and the algorithm takes an additional $4n + 1 \in O(n)$ steps to grow the array. Intuitively, since growing happens so rarely, it should add an insignificant amount to the amortized cost of the `record` operation. The following proof uses the accounting method to prove that conclusively.

Lemma 11. *The `Log` constructor and `record` operation each take $O(1)$ amortized time.*

Proof. Consider a conceptual time bank whose balance fluctuates as operations are performed. Let n represent the number of messages currently in the log at any given moment, and k be the number of fast `record` operations that have occurred since the last time a new array was created. We define a price of 2 tokens per constructor, 9 tokens per `record`, and a savings goal of maintaining a balance of at least $6k$ tokens in between operations.

Before the constructor our balance is zero. We charge 2 tokens and spend 2 tokens executing the constructor, leaving our balance at zero. The constructor creates a new array, so at this moment $k = 0$ and our balance is indeed at least $6k = 6(0) = 0$.

Next consider a `record` operation with a non-full array. The operation costs 9 tokens and we skip the body of the `if`, so we spend only 3 tokens executing the operation, leaving a profit of 6 tokens. After the operation finishes there have been $k' = k + 1$ fast operations since the last array creation. Before the operation we had a balance of at least $6k$ tokens; now we have at least

$$6k + 6 = 6(k + 1) = 6k'$$

tokens, so our savings goal is still met.

Finally consider a `record` operation with a full array. This time the body of the `if` statement is executed. Evaluating the `if` takes 1 step; creating the new array of length $2n$ takes $2n$ steps; copying n elements over to this new array takes n steps; reassigning `self.array` takes 1 step; and finally the two statements outside the `if` take 2 steps, for a total of $3n + 4$ steps. The price for this operation is only 9 time units, so this operation runs a deficit of $3n - 5$ tokens. Fortunately, our savings goal implies that we had a balance of at least $6k$ before the operation started.

We need to relate this surplus of $6k$ tokens (a function of k) to our deficit of $3n - 5$ tokens (a function of n). Immediately after the constructor the array is empty, and immediately after growth the array is exactly one-half full. Therefore, filling the array before this operation started required at least $\frac{1}{2}n$ fast `record` operations, and we know $k \geq \frac{1}{2}n$. So rewriting our $6k$ balance in terms of n, we have $6k \geq 6(\frac{1}{2}n) = 3n$. So withdrawing nearly all of our balance allows us to pay for the entire cost of this operation with at least 5 tokens left over. This is a slow list-creating operation, so afterward $k = 0$, and our balance of 5 tokens exceeds our goal of $5k = 0$ time units.

Therefore, charging a price of 2 tokens per constructor and 9 tokens per `record` guarantees that the time balance is always non-negative. $2 \in O(1)$ and $9 \in O(1)$ so each of these operations takes $O(1)$ amortized time. □

3.7.2 Amortized versus Worst-Case Efficiency Classes

We have seen two flavors of efficiency class: a worst-case efficiency class is true for each and every operation, while an amortized efficiency class is true on average. For example, when an operation takes $O(1)$ worst-case time, as is the case with adding a node to a linked list, *every* one of those operations take $O(1)$ time, without exception. By contrast an operation that takes $O(1)$ amortized time may sometimes take longer than $O(1)$ time, but on average takes only $O(1)$ time. For example, and as discussed shortly in Section 4.4, adding an element to the back of a vector occasionally takes $O(n)$ time, but this happens so rarely, that the amortized time works out to only $O(1)$. This time complexity unnevenness is certainly a drawback for vectors, but as we will see they have other redeeming qualities that make vectors an attractive data structure in many practical applications.

This kind of trade-off crops up frequently in the world of data structures. Often we have competitive alternatives, where one alternative features non-amortized worst-case efficiency classes, and another alternative offers only amortized efficiency classes but makes up for that with some other attractive feature. Sometimes the feature is preferable constant factors, sometimes it is a wider suite of operations, and sometimes it is conceptual simplicity and ease of understanding and implementation.

Exercises

3-1. *Matching socks.* Suppose that you have a mixture of red socks and blue socks in a jumbled drawer. You need to find a pair of matching socks, but it is so early in the morning that you can't see the color of a sock until you've pulled it out. In the worst case, how many socks must you pull in order to have two of the same color?

3-2. *Brute force password guessing.* Each of the following is a different kind of password. An attacker can always try to bypass password security by guessing every possible password that might exist. In the worst case (for the attacker), the correct password is the last one that they guess. So increasing the number of potential passwords makes password protection more effective. For each password type, compute the number of passwords of that type that exist. Justify your answers (show your work).

(a) Eight lower-case English letters.

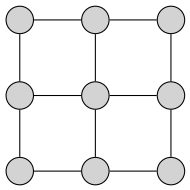
(b) Four base-10 digits.

(c) Eight lower-case letters, upper-case letters, or digits.

(d) Nine letters.

(e) A sequence of three distinct digits 0 through 39 (as seen in a combination lock).

(f) A path in the following graph that visits exactly four vertices.



(g) Based on this analysis, which of these password systems seems most secure?

3-3. Prove that `tricky_while_loop` on page 50 terminates.

3-4. Give definitions for three distinct functions $g_1(n), g_2(n),$ and $g_3(n)$ that are members of the set $O(n^3)$.

3-5. Explain why the statement "the running time of algorithm A is at least $O(n^2)$" is meaningless.

3-6. Let $g(n) = n$. Is it true that $g(n) \in O(n^2)$? Why or why not?

3-7. Evaluate each of the following complexity functions for $n = 2^3 = 8, n = 2^7 = 128$, and $n = 2^{10} = 1024$. The goal of this exercise is to get a feel for the relative rates of growth of some common efficiency classes.

(a) $\log_2 n$
(b) n
(c) n^2
(d) n^3
(e) 2^n
(f) $n!$

3-8. Compare the functions $f_1(n) = n^3$ and $f_2(n) = 3n^2 + 10$ for various values of n. Determine the approximae value of n at which $f_1(n)$ starts to become larger than $f_2(n)$. You will need to pick appropriate values of n to justify your answers.

3-9. Use induction to prove which efficiency class each of the following complexity functions belong to.

(a) $3n + 4$
(b) $n^2 + 1$
(c) $\sum_{i=1}^{n} i^2 + 100\, n \log n$ (hint: this function is in $O(n^3)$)

3-10. Use limits to prove which efficiency class each of the following complexity functions belongs to.

(a) $7n + 2$
(b) $(n + 1)(n + 2) + 2n$
(c) $(n + 2)^6$

3-11. Use properties of O to prove the efficiency class of each of the functions in the previous problem.

3-12. Prove that the following proposition is false: $50\,n + 2n^2 \in O(n)$.

3-13. Rank the following functions by their order of growth (i.e. asymptotic efficiency class), from slowest-growing to fastest-growing.

(a) n^2

(b) $\log n$

(c) $4^{\log n}$

(d) $n \log n$

(e) $\log \log n$

(f) n

(g) 2^n

(h) $n!$

(i) 2^{n+1}

(j) $n^{1.001}$

(k) $(\frac{3}{2})^n$

(l) $n \log \log n$

(m) $2^{n/2}$

3-14. Each of the following is a problem, and pseudocode for an algorithm solving that problem. For each algorithm, derive a complexity function for its running time $T(n)$, then prove the efficiency class $O(f(n))$ that $T(n)$ belongs to.

(a)

mean problem
input: a non-empty list L of n numbers **output:** the mean (average) of L

```
def mean(L):
    total = 0
    for x in L:
        total += x
    return total / len(L)
```

(b)

square matrix construction
input: a positive integer n and number x **output:** an $n \times n$ matrix with each element equal to x

```python
def construct_square_matrix(n, x):
    rows = []
    for r in range(n):
        rows.append([])
        for c in range(n):
            rows[r].append(x)
    return rows
```

(c)

list reversal problem
input: a list L **output:** a list containing the elements of L in reverse order

```python
def reverse(L):
      n = len(L)
      rev = []
      for i in range(n):
          rev.append(L[n-i-1])
      return rev
```

3-15. Prove Corollary 2 by induction on the number of terms k.

3-16. Write pseudocode for another algorithm whose time complexity is overcounted by a chronological step count. Derive an erroneous time complexity function using chronological step counting, and a more accurate complexity function with an amortized analysis.

3-17. Prove Lemma 8, which states $O(\lfloor f(n) \rfloor) = O(\lceil f(n) \rceil) = O(f(n))$.

Chapter 4

Essential Data Structures

4.1 The Big Idea

So far our algorithms have stored all of their input and input in primitive data types (integer, string, etc.) or arrays. These data types may suffice for simple algorithms and small data sets, but more sophisticated data structures are necessary for managing large volumes of data and designing efficient non-trivial algorithms.

As stated in Section 1.2, this book assumes that the reader already knows about these data structures. Therefore this chapter presents only a brief recap of data structures that will be used throughout the text. For a more in-depth treatment that explains the structures in detail, refer to the textbooks listed in Section 1.2.

4.2 Python Interfaces

Some data structures require that the elements stored within them conform to certain interfaces. Namely, binary search trees require that the key values used to locate nodes are *comparable*, meaning that code can determine, for any two keys x and y, whether $x < y, x = y$, or $x > y$.

In addition, programming practicalities often require that data structures are *iterable*, meaning that it is possible to loop through the elements of the structure using an *iterator* object that represents a location within the structure.

We aim for the data structure listings in this Chapter, and for the final drafts of our algorithm pseudocode, to do double duty as working Python implementations. Therefore we need to use Python's idioms for comparability and iteration. Admittedly, this syntax may be a bit surprising for those accustomed to different languages. We ask that you bear with us during this brief excursion into Python.

4.2.1 Classes and Interfaces

In object-oriented programming, a *class* is a kind of data type and a *class instance* (or simply *instance*) is an object belonging to a specific class type. A class defines the *data members* and *member functions* that each instance of that class contains. The defining characteristic of object-oriented programming is that objects have both data members and member functions.

For example, the following Python class defines a point in the Cartesian plane.

```
class Point:
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def quadrant(self):
        if self.x >= 0:
            if self.y >= 0:
                return 1
            else:
                return 4
        else:
            if self.y >= 0:
                return 2
            else:
                return 3
```

`Point` is the name of the class. `__init__` and `quadrant` are member functions. By convention, private member functions begin with `_`, member functions with special meanings begin and end with `__`, and all other member fuctions begin with letters. There are two data members, `x` and `y`. `__init__` is a special member function called the *constructor* which is implicitly called to initialize a new instance of the `Point` class. For example, the following line of code creates a new `Point` object bound to the variable `p`, which involves calling `__init__` to initialize `p`'s `.x` and `.y` data members.

```
p = Point(3, 4)
```

Once a class instance is created, its data members may be accessed with the pattern

```
<object>.<data member>
```

so `p.x` yields `3`. Member functions may be called with the pattern

```
<object>.<member function>(<arguments...>)
```

so `p.quadrant()` returns `1`.

Python uses *duck typing*, as in the idiom "if it looks like a duck, swims like a duck, and quacks like a duck, then it probably is a duck." In order for a class to conform to a particular interface, all it needs to do is contain the data members and member functions relevant to that interface. There is no need to explicitly declare that the class implements an interface, overrides an abstract class, or anything else of the kind.

4.2.2 Length Interface

Python has a special built-in function called `len` that returns the length of a container object. For example, `[1, 3, 3, 7]` is a literal list object containing four integers, so `len([1, 3, 3, 7])` returns `4`.

`len` works on the built-in data structures such as list, tuple, and so on. In order for `len` to work on a user-defined class instance, that class must implement a special `__len__` member function that returns the instance's length.

For example, we could augment our `Point` class to implement the length interface.

```
class Point:
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def __len__(self):
        return 2

    def quadrant(self):
        if self.x >= 0:
            if self.y >= 0:
                return 1
            else:
                return 4
        else:
            if self.y >= 0:
                return 2
            else:
                return 3
```

We always return `2` since a planar point always contains exactly two coordinates. Now that the

`Point` class has a defined `__len__` member function, calling `len` on a `Point` object works. For example, `len(p)` returns `2`.

This is duck typing at work; to make a class length-capable, all we need to do is implement one special member function related to length.

4.2.3 Comparable Interface

As you may know, some data structures maintain their elements in a defined order. Namely, each entry in a binary search tree contains a key, and the entries are maintained in left-to-right increasing order. Likewise, each entry in a priority queue contains a priority value, and the queue arranges the entries in a semi-sorted order according to their priorities. Numbers are often used as keys and priorities, but these structures can be generalized to use any kind of *comparable* data type as a key or priority.

The comparison operators `<`, `<=`, `==`, `>=`, and `>` correspond to the special class member functions `__lt__`, `__le__`, `__eq__`, `__ge__`, and `__gt__`. (The acronyms stand for "less than," "less-equal," and so on.) Any Python class that implements all these methods is comparable. For example, we could make our `Point` class comparable on the basis of x-coordinates.

```
class Point:
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def __len__(self):
        return 2

    def __lt__(self, other):
        return self.x < other.x
    def __le__(self, other):
        return self.x <= other.x
    def __eq__(self, other):
        return self.x == other.x
    def __ge__(self, other):
        return self.x >= other.x
    def __gt__(self, other):
        return self.x > other.x

    def quadrant(self):
        if self.x >= 0:
            if self.y >= 0:
                return 1
            else:
                return 4
        else:
            if self.y >= 0:
                return 2
            else:
                return 3
```

Allowing each comparison function to be defined separately supports a wide variety of ordering use-cases. For instance we could define a constant `INFINITY` representing ∞; the `__lt__` and `__gt__` functions would need special-case logic to ensure that non-infinite values are considered strictly less than ∞, but the `__eq__` function could stay as-is because ∞ is conceptually equal to itself. However, in routine use cases such as `Point` all five comparison functions are defined in terms of comparisons of the same field, and writing out those function definitions is tedious. To ameliorate this the Python standard library has a `functools.total_ordering` decorator that takes a class with a defined `__eq__` and any one of `__lt__`, `__le__`, `__ge__`, or `__gt__` and implicitly defines the missing comparison functions for you. So the following version of `Point` supports all five comparison operators, even though we only explicitly define two of them.

```
@total_ordering
class Point:
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def __len__(self):
        return 2

    def __lt__(self, other):
        return self.x < other.x
    def __eq__(self, other):
        return self.x == other.x

    def quadrant(self):
        if self.x >= 0:
            if self.y >= 0:
                return 1
            else:
                return 4
        else:
            if self.y >= 0:
                return 2
            else:
                return 3
```

Many of our data structures and problems require that certain inputs are comparable.

4.2.4 Iterator and Iterable Interfaces

An *iterator* (also known as a *cursor*, *generator*, and *enumeration*) is an object that supports stepping through the elements of a collection data structure one at a time. An iterator keeps track of its position within the structure, and can retrieve the structure element at the current position; move to the next position; and determine whether the iterator has reached the end of the structure. A data type is *iterable* when it can produce iterators.

An iterator type must support two special member functions: `__iter__` returns the iterator object itself, and is used to bootstrap loops; and `__next__` returns the next element and moves to the following position, or raises a `StopIteration` exception when that is impossible because the iterator has reached the end. The following iterator class produces the two coordinates of a `Point`.

```
POINT_POSITION_X = 0
POINT_POSITION_Y = 1
POINT_POSITION_END = 2

class PointIterator:
    def __init__(self, point):
        self.point = point
        self.position = POINT_POSITION_X
    def __iter__(self):
        return self
    def __next__(self):
        if self.position == POINT_POSITION_X:
            self.position = POINT_POSITION_Y
            return point.x
        elif self.position == POINT_POSITION_Y:
            self.position = POINT_POSITION_END
            return point.y
        else:
            raise StopIteration
```

The built-in function `iter` takes a container object of an iterable type and returns an iterator object for that container. In order to be iterable, a class must have a `__iter__` member function that returns a new iterator. We can easily make our `Point` iterable.

```
@total_ordering
class Point:
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def __iter__(self):
        return PointIterator(self)

    def __len__(self):
        return 2

    def __lt__(self, other):
        return self.x < other.x
    def __eq__(self, other):
        return self.x == other.x

    def quadrant(self):
        if self.x >= 0:
            if self.y >= 0:
                return 1
            else:
                return 4
        else:
            if self.y >= 0:
                return 2
            else:
                return 3
```

Now, for any `Point` instance `x`, `iter(x)` works and returns a `PointIterator` object at the x position.

The simplest way to step through an iterator is with a `for` loop. In the syntax pattern

```
for <element> in <iterator>:
    <body>
```

the `<iterator>` blank may be any expression that yields an iterator object. So we could print out the two coordinates of our `Point` object `p` with

```
for coord in iter(p):
    print(coord)
```

Executing that code with the `p` object created earlier would result in printing the following.

```
3
4
```

4.3 Array

An *array* is a fixed-length sequence of elements laid out in one contiguous block of memory. Each element is stored at a specific index i ranging from 0 to $n - 1$ inclusive, where n is the length of the array. An array can get or set the element at an arbitrary index i in only $O(1)$ time, by calculating the memory address of that element as a function of i. This is a crucial feature of computers under the random access machine model. The efficiency of many more sophisticated data structures and algorithms, such as hash tables and dynamic programming, hinge on the fast $O(1)$ access time of arrays. Arrays are iterable.

The Python language does not have an array data type built in; its fundamental "list" data type corresponds to what we call a vector, described in the next section. Our `Array` class is a thin object-oriented wrapper around a Python list.

In practically all programming languages in widespread use aside from C/C++, every element of an array must be initialized to a defined value when the array is created. Creating an n-element array where each element is initialized to some value x takes $O(n)$ time due to all the initializations. The Python syntax for this operation is `array = [x] * n`. Retrieving the element at index i takes $O(1)$ time, and its syntax is `array[i]`. Assigning the element at index i to a new value y also takes $O(1)$ time and its syntax is `array[i] = y`.

Once an array object is created its length is fixed for the lifetime of that object. If an array of a different length is needed later on, the only recourse is to create a new array object and discard the original one. This is inefficient and unwieldy, which is why vectors and linked lists were designed.

Array Operation	Pseudocode	Time Complexity
Create an array with n copies of x	`array = Array(n, x)`	$O(n)$
Return the length of an array	`len(array)`	$O(1)$
Return an iterator for all elements	`iter(array)`	$O(1)$
Get the element at index i	`array[i]`	$O(1)$
Set the element at index i to x	`array[i] = x`	$O(1)$

Table 4.1: Array Operations

```
class Array:
    def __init__(self, length, default):
        if length < 0:
            raise ValueError
        self.length = length
        self.elements = [default] * length
    def __len__(self):
        return self.length
    def __iter__(self):
        return iter(self.elements)
    def __getitem__(self, i):
        if i < 0 or i >= self.length:
            raise IndexError
        return self.elements[i]
    def __setitem__(self, i, x):
        if i < 0 or i >= self.length:
            raise IndexError
        self.elements[i] = x
        return x
```

4.4 Vector

A *vector* (also known as *arrayed list*, *dynamic array*, and *resizable array*) is essentially an array whose length is not fixed. A vector supports all the same operations as an array: retrieving by index, assigning by index, and iteration. In addition, the vector supports adding and removing elements dynamically. This makes vectors far more flexible and convenient than arrays, so vectors are overwhelmingly preferred in languages such as C++ and Java that provide both structures.

A vector is implemented with a partially-filled array; that is, an array whose actual length is n,

but is only storing legitimate elements in the first l indices for some $l \leq n$. Often $l < n$, meaning that the first l elements of the array are in use, while the following $n - l$ elements are conceptually vacant. (As stated above our array elements are always initialized, so vacant array elements hold some dummy placeholder value such as `None`). A vector implementation tracks the current value of l in a member variable.

The benefit of this design is that adding or removing elements at the back (highest index) is very fast, taking only $O(1)$ time. Removing an element from the back involves only decrementing the current length l and re-initializing the newly-vacant element to the dummy value. When $l < n$, adding an element x to the back involves assigning `array[l] = x` and incrementing l.

Things get more complicated when adding an element to a vector whose array is full, i.e. when $l = n$. In this case we *resize* the array. We create a new, substantially larger, array; copy all the existing elements from the old array into the new one; discard the old array; and finally add the new element x as usual. This process is relatively slow, taking $O(n)$ time to copy elements over. However, with some care it is possible to ensure that resizing only happens rarely. If we double the length of the array when it grows, and also resize the array when it is less than $1/3$ full, then it follows that a resize may only happen every $O(1/n)$ add or remove operations, and the amortized efficiency of those operations is $O(1)$. Actually, the shrink-ratio $1/3$ and growth-ratio 2 are not the only ratios that can work; any multiplicative grow-ratio $r > 1$ and shrink-ratio less than $1/r$ results in $O(1)$ amortized add/remove operations. These ratios may be chosen to tune the time and space efficiency of a vector; increasing r means that the array grows faster, making resizes less frequent, and the addition operation faster; but also means that a larger proportion of array elements may be vacant, decreasing the memory efficiency of the structure.

```
class Vector:
    def __init__(self, length, default):
        self.array = Array(length, default)
        self.in_use = 0
        for i in range(length):
            self.add_back(default)
    def __len__(self):
        return self.in_use
    def __iter__(self):
        return VectorIter(self)
    def __getitem__(self, i):
        if i < 0 or i >= self.in_use:
            raise IndexError
        return self.array[i]
    def __setitem__(self, i, x):
        if i < 0 or i >= self.in_use:
            raise IndexError
        self.array[i] = x
        return x
    def add_back(self, x):
        if self.in_use == len(self.array):
            self._resize(len(self.array) * 2)
        self.array[self.in_use] = x
        self.in_use += 1
    def remove_back(self):
        if self.in_use == 0:
            raise UnderflowError
        self.in_use -= 1
        if (3 * self.in_use) < len(self.array):
            self._resize(self._capacity() // 2)
    def _resize(self, new_capacity):
        new_array = Array(new_capacity, None)
        for i in range(self.in_use):
            new_array[i] = self.array[i]
        self.array = new_array
```

Vectors are iterable; the iterator simply keeps track of which index it is at.

Vector Operation	Pseudocode	Time Complexity
Create an empty vector	`vector = Vector()`	$O(1)$
Create a vector with n copies of x	`vector = Vector(n, x)`	$O(n)$
Return the length of a vector	`len(vector)`	$O(1)$
Return an iterator for all elements	`iter(vector)`	$O(1)$
Get the element at index i	`vector[i]`	$O(1)$
Set the element at index i to x	`vector[i] = x`	$O(1)$
Add element x to the back (highest index)	`vector.add_back(x)`	$O(1)$ amortized
Remove the element at the back (highest index)	`vector.remove_back()`	$O(1)$ amortized

Table 4.2: Vector Operations

```
class VectorIter:
    def __init__(self, vector):
        self.vector = vector
        self.index = 0
    def __iter__(self):
        return self
    def __next__(self):
        if self.index == len(self.vector):
            raise StopIteration
        else:
            self.index = self.index + 1
            return self.vector[self.index-1]
```

Vectors are widely used as a "go-to" sequence data structure; indeed, the built-in list structure in Python is a vector, and programmers in C++ and many other languages are often encouraged to use a vector unless there is a compelling reason to use some other data structure. One reason for this is that, like linked lists, building a vector one element at a time is fast at only $O(1)$ time per element (amortized in the case of the vector). But unlike linked lists, the contiguous memory layout of vectors facilitate locality of reference which makes the most of the memory cache available on contemporary processors. So vectors are both asymptotically efficienct for large data sets and tend to have good real-world constant factors in practice.

4.5 Linked List

A *doubly-linked list*, or simply *linked list* for short, is a sequence data structure that is similar in function to a vector, but markedly different in approach. Each element in a doubly-linked list is

stored within an enclosing *node* object. Each node contains one element; a pointer to the next node in the list; and a pointer to the previous node in the list.

```
class LinkedNode:
    def __init__(self, left, data, right):
        self.left = left
        self.data = data
        self.right = right
```

Our linked list data structure maintains a special *head* node that is a placeholder that comes before the nodes that store actual elements; and another similar *tail* node that comes after all of the nodes that store actual elements. The head and tail nodes make for simple data structure operations, since every data-containing element, including the first and last, has a valid node to its left and right. The linked list object contains a pointer to the head node, pointer to the tail node, and a count of how many elements are in the list.

```
class LinkedList:
    def __init__(self):
        self.head = LinkedNode(None, None, None)
        self.tail = LinkedNode(self.head, None, None)
        self.head.right = self.head.tail
        self.size = 0
    def __len__(self):
        return self.size
    def __iter__(self):
        return LinkedListIter(self)
    def _check_nonempty(self):
        if self.size == 0:
            raise UnderflowError
    def node_first(self):
        self._check_nonempty()
        return self.head.right
    def node_last(self):
        self._check_nonempty()
        return self.tail.left
    def first(self):
        return self.node_first().element
    def last(self):
        return self.node_last().element
```

This linked structure means that we can remove a node object in $O(1)$ time regardless of its index simply by adjusting the pointers of the node's neighbors so that the pointers "skip over" the removed node. Likewise, we can insert a new node beside a pre-existing node in $O(1)$ time regardless of its index. In particular, we can add an element to the front, add an element to the back, remove the front element, or remove the back element, in $O(1)$ time each. This is a big improvement over a vector; adding or removing the first element of a vector would take $O(n)$ time, because each element (aside from the first) needs to be moved one element over to make room. Likewise, removing an element at an arbitrary index from a vector takes $O(n)$ time because up to $n-1$ elements need to be relocated. It is true that an element may be added or removed to the back of a vector in $O(1)$ amortized time, but that only applies to the back position and no other position, and is an amortized time bound, not a worst-case time bound. So linked lists are superior to vectors in situations where elements need to be added and removed at locations aside from the back, or when amortized time bounds are unacceptable.

```
    def insert_before(self, position, x):
        new_node = LinkedNode(position.left, x, position)
        new_node.left.right = new_node
        new_node.right.left = new_node
        self.size += 1
        return new_node
    def insert_after(self, position, x):
        new_node = LinkedNode(position, x, position.right)
        new_node.left.right = new_node
        new_node.right.left = new_node
        self.size += 1
        return new_node
    def add_first(self, x):
        return self.insert_after(self.head, x)
    def add_last(self, x):
        return self.insert_before(self.tail, x)
    def remove(self, node):
        self._check_nonempty()
        assert( node is not self.head )
        assert( node is not self.tail )
        node.left.right = node.right
        node.right.left = node.left
        self.size -= 1
        return node.element
    def remove_first(self):
        return self.remove(self.node_first())
    def remove_last(self):
        return self.remove(self.node_last())
```

The drawback of linked lists is that seeking to the element at index i takes up to $O(n)$ time. The only way to find the element at an arbitrary index i is to start at one of the ends (the head or tail) and skip past nodes until we reach the node at index i. This seeking takes $O(n)$ time, which is inferior to the $O(1)$ lookup time of arrays and vectors. Also, since each node object is created separately, nodes cannot be expected to reside at contiguous addresses. Consequently navigating through linked lists by following left- and right-pointers tends to be significantly slower, by a constant factor, than navigating through an array, due to memory cache misses. So vectors are superior to linked lists in situations where looking up elements by index is essential, or when constant factors are important.

```
    def node_at(self, index):
        if index < 0 or index >= self.size:
            raise IndexError
        node = self.head.right
        for k in range(index):
            node = node.right
        return node
    def __getitem__(self, i):
        return node_at(self, i).element
    def __setitem__(self, i, x):
        node_at(self, i).element = x
```

Linked lists are iterable; an iterator maintains a pointer that starts out to the right of the head node, and moves to the right after visiting each node. When that pointer reaches the tail node, the iteration is over.

```
class LinkedListIter:
    def __init__(self, linked_list):
        self.linked_list = linked_list
        self.node = linked_list.head.right
    def __iter__(self):
        return self
    def __next__(self):
        if self.node is self.linked_list.tail:
            raise StopIteration
        else:
            x = self.node.element
            self.node = self.node.right
            return x
```

Linked List Operation	**Pseudocode**	**Time Complexity**
Create an empty linked list	`ll = LinkedList()`	$O(1)$
Get the length of a linked list	`len(ll)`	$O(1)$
Get an iterator for all elements	`iter(ll)`	$O(1)$
Get the first element of a non-empty list	`x = ll.first()`	$O(1)$
Get the last element of a non-empty list	`x = ll.back()`	$O(1)$
Get the first node of of a non-empty list	`node = ll.first_node()`	$O(1)$
Get the last node of of a non-empty list	`node = ll.last_node()`	$O(1)$
Get the node at index i	`node = ll.node_at(i)`	$O(n)$
Add element x to the front and return the new node	`node = ll.add_front(x)`	$O(1)$
Add element x to the back and return the new node	`node = ll.add_back(x)`	$O(1)$
Insert element x before node p and return the new node	`node=ll.insert_before(p,x)`	$O(1)$
Insert element x after node p and return the new node	`node=ll.insert_after(p,x)`	$O(1)$
Remove and return the first element of a non-empty list	`x = ll.remove_first()`	$O(1)$
Remove and return the last element of a non-empty list	`x = ll.remove_last()`	$O(1)$
Remove a node at an arbitrary position and return its element	`x = ll.remove(node)`	$O(1)$
Access a node's element	`node.element`	$O(1)$

Table 4.3: Linked List Operations

Entry Operation	Pseudocode	Time Complexity
Access an entry's key	`entry.key`	$O(1)$
Access an entry's value	`entry.value`	$O(1)$

Table 4.4: Entry Operations

4.6 Entry

An *Entry* is a simple container object that stores two objects: a *key* and an associated *value*. An entry key must be comparable (recall that the definition and notation for comparable objects was discussed back in Subsection 4.2.3).

```
class Entry:
    def __init__(self, key, value):
        self.key = key
        self.value = value
```

The idea of an entry is that its key is a concise identifier, and its value is the data associated with that key. For example, the key could be string containing a person's name, while the value is a `Person` class instance containing many data members such as address, age, and so on. Or the key could be an integer representing a product's serial number, while the value is a `Product` class with data members for price, shipping weight, etc. Data structures and algorithms may organize entries according to keys, so data types that are compact in memory and fast to compare are usually used for keys, such as integers or short strings.

The self-balancing binary search tree (Section 4.7), priority queue (Section 4.10), priority search queue (Section 10.4), and hash table (Section 10.3.1), data structures are all concerned with storing `Entry` objects efficiently.

4.7 Self-Balancing Binary Search Tree

A *binary search tree* is a data structure that stores entries (as described in Section 4.6) in a kind of sorted order. A binary search tree is comprised of *nodes*; each node contains one entry, a pointer to a left subtree, and a pointer to a right subtree. Each subtree pointer may be either a pointer to another node; or an empty subtree called a *leaf* which is represented in code by `None`. The topmost node is called the *root,* and the length of the longest path from the root to a leaf is called the *height* of a tree.

A binary search tree enforces the *binary search tree invariant* which dictates that, for any node p

containing entry key x, every key w in the left subtree of p obeys $w < x$; and symmetrically, every key y in the right subtree of p obeys $x < y$. This invariant makes it possible to search for a given key k quickly; starting from the root, we compare k to the key x in the current node's entry. When $k = x$, we have succesfully found the entry in question; when $k < x$, we continue searching in the node's left subtree; and when $k > x$, we continue searching in the node's right subtree. If we ever reach a leaf (`None`) in this process, then we can safely conclude that k does not exist in the tree.

Inserting a new entry involves searching for that entry according to the process above, then replacing the leaf node that is reached with a new node. Removing an entry involves searching for the node containing that entry, and removing that node. If the node has any children, its removal may leave a "hole" in the tree, which must be "patched" by finding a different node to take its place.

Each of the search, insertion, and removal algorithms involves traversing the tree from the root to a leaf, and spending $O(1)$ time at each node along the way. So if a binary search tree has height h, than these operations take $O(h)$ time each. Unfortunately it is possible for a binary search tree to become extremely *unbalanced*. When every right pointer is a leaf, so that every node is a left child (except the root), the tree has height $h = n$. A tree with all right children has the same height $h = n$, and more generally, any tree with the property that each node has at least one leaf child, has height $h = n$. In any of these degenerate cases the search, insertion, and removal operations take $O(n)$ worst-case time.

The solution is to redesign the tree structure to be *self-balancing*. A self-balancing binary search tree maintains, in addition to the binary search tree invariant, another structural invariant that prevents the tree from becoming badly imbalanced. It is impractical to keep a binary search tree *perfectly* balanced, but it turns out that there are many ways of keeping a tree *well* balanced, up to constant factors. In particular, there are many variations of self-balancing binary search trees that all guarantee that the height of the tree is $h \in O(\log n)$. Examples include AA trees [11], AVL trees [9], brother trees [26][42], finger trees [27], splay trees [47], and treaps [12].

Through intricate rebalancing algorithms, each of these data structures maintains a tree with $O(\log n)$ height, and therefore searching for, inserting, or removing an entry takes $O(\log n)$ time.

For the sake of brevity we will not present code for a self-balancing binary search tree here. Instead we will show the interface that we will use in our pseudocode that manipulates search trees.

```
class BST:
    def __init__(self) # create an empty tree
    def __iter__(self) # return an iterator for all entries
    def __len__(self) # return the number of entries

    # return true when key is present in the tree
    def contains(self, key)

    # return the Entry object for key if present; or None if no such entry
    # exists in the tree
    def search(self, key)

    # If the tree already contains an entry for key, overwrite its value and
    # return False. Otherwise insert a new entry and return True.
    def insert(self, key, value)

    # If the tree contains an entry for key, remove it and return True.
    # Otherwise leave the tree unchanged and return False.
    def remove(self, key)

    # Return the entry with the least key; throws UnderflowException when the
    # tree is empty.
    def min(self)

    # Return the entry with the greatest key; throws UnderflowException when
    # the tree is empty.
    def max(self)
```

4.8 Stack

A *stack* maintains elements in *last-in first-out (LIFO)* order. By convention, adding an element is called *pushing* it on to the *top* of the stack. When an object is pushed, all of the pre-existing elements "sink" down beneath the new top element. A stack can return the topmost element, but does not expose any other element. The *pop* operation removes and returns the top element, and the element immediately below it "rises" to become the new top. This behavior mimics a stack of papers that allows a sheet of paper to be pushed on top or pulled off the top, but cannot be otherwise rearranged conveniently. Since the most-recently pushed element would be the first to be popped, the order imposed by a stack is called last-in first-out.

Applications for last-in first-out orderings are admittedly narrow, but they do arise in computer

Balancing Binary Search Tree (BST) Operation	Pseudocode	Time Complexity
Create an empty tree	`bst = BST()`	$O(1)$
Get the number of keys in a tree	`len(bst)`	$O(1)$
Get an iterator for the entries in increasing order	`iter(bst)`	$O(1)$
Determine whether a tree contains key k	`boolean=bst.contains(k)`	$O(\log n)$
Get the entry associated with key k, or `None`	`entry = bst.search(k)`	$O(\log n)$
Associate key k with value v; return `True` if a new key entry was created	`boolean=bst.insert(k, v)`	$O(\log n)$
Remove the entry for k (if any); return `True` if an entry was actually removed	`bst.remove(k)`	$O(\log n)$
Get the entry with the least key	`entry = bst.min()`	$O(\log n)$
Get the entry with the greatest key	`entry = bst.max()`	$O(\log n)$

Table 4.5: Balancing Binary Search Tree (BST) Operations

Stack Operation	Pseudocode	Time Complexity
Create an empty stack	`stack = Stack()`	$O(1)$
Get the length of a stack	`len(stack)`	$O(1)$
Get an iterator for the elements in top-to-bottom order	`iter(stack)`	$O(1)$
Add element x to the top	`stack.push(x)`	$O(1)$
Get the element at the top	`x = stack.top()`	$O(1)$
Remove and return the element at the top	`x = stack.pop()`	$O(1)$

Table 4.6: Stack Operations

science applications. Last-in first-out order applies to situations where we process tasks that may interrupt each other, and once we finish with an interrupting task we need to resume to the task that got interrupted. Function calls work this way in many programming languages; when function a calls function b, the program's state in a is pushed onto an *activation record* on "the stack," the program executes function b, and then pops the activation record to resume running a. Stacks may also be used as part of the *depth-first search* graph traversal algorithm, the kind of parsers used in compilers, the *undo-redo* behavior common in many user interfaces, and iterators for binary search trees.

It is easy to implement a stack in terms of a doubly-linked list. The top, push, and pop operations each take $O(1)$ worst-case time.

```
class Stack:
    def __init__(self):
        self.linked_list = LinkedList()
    def __len__(self):
        return len(self.linked_list)
    def __iter__(self):
        return iter(self.linked_list)
    def top(self):
        return self.linked_list.first()
    def push(self, x):
        self.linked_list.add_first(x)
    def pop(self):
        x = self.top()
        self.linked_list.remove_first()
        return x
```

Since elements are only added and removed from one end of the linked list, a stack could be implemented in terms of a singly-linked list instead of a doubly-linked one. This would result in marginally better constant factors since the list would store and maintain fewer pointers. Or, a stack could be implemented in terms of a vector, which would have even better constant factors, but the push and pop operations would take $O(1)$ amortized time due to occasional resizing.

4.9 Queue

A *queue* maintains elements in *first-in first-out (FIFO)* order. In other words, elements are added to the "back of the line" and elements are removed from the "front." This operation mimics how people wait fairly for service, in a "line" in American English or "queue" in British English. Adding an element is called *enqueuing* and removing the front element is called *dequeueing.*

Applications of queue data structures abound. A queue is the most appropriate data structure for a playlist in a video player, download queue in a web browser, or build queue in a strategy game.

It is easy to implement a queue in terms of a doubly-linked list. As shown in Table 4.7, with this approach the front, enqueue, and dequeue operations each take $O(1)$ worst-case time.

```
class Queue:
    def __init__(self):
        self.linked_list = LinkedList()
    def __len__(self):
        return len(self.linked_list)
    def __iter__(self):
        return iter(self.linked_list)
    def front(self):
        return self.linked_list.first()
    def enqueue(self, x):
        self.linked_list.add_last(x)
    def dequeue(self):
        x = self.front()
        self.linked_list.remove_first()
        return x
```

Intriguingly, the only significant difference between our queue implementation and our stack implementation is that the stack adds element to the front of the list while the queue adds elements to the back.

It is possible to implement a queue in terms of a singly-linked list or vector, but unlike with a stack, this is rather complex and requires breaking encapsulation to manipulate the inner workings of the list/vector data structure. In order to implement a queue with a singly-linked list, the list needs an additional pointer to the last node, and this pointer must be updated every time a node is added or removed to the list. A queue implemented in terms of a vector is called a *circular array* or *ring buffer*. Such a queue needs to keep track of the range of element indices that are in-use, resize the vector when over- or under-full as usual, and preserve the in-use elements during the resize operation.

4.10 Priority Queue

A *priority queue* is similar in spirit to a plain queue, in that entries are enqueued and then dequeued in a fair proscribed order. The difference is that, in a priority queue, entries are ordered according to a numeric *priority* value that is stored as the keys of the entries. By convention, the least (lowest) priority correspond to the front of a priority queue while the greatest (highest) priority

Queue Operation	Pseudocode	Time Complexity
Create an empty queue	`queue = Queue()`	$O(1)$
Get the length of a queue	`len(queue)`	$O(1)$
Get an iterator for the elements in front-to-back order	`iter(queue)`	$O(1)$
Get the element at the front of a non-empty queue	`x = queue.front()`	$O(1)$
Add element x to the back	`queue.enqueue(x)`	$O(1)$
Remove and return the element at the front	`x = stack.dequeue()`	$O(1)$

Table 4.7: Queue Operations

corresponds to the back of a priority queue. Like a plain queue, a priority queue supports enqueue, front, and dequeue operations.

A priority queue can be implemented in terms of a data structure called a *binary heap*, and this approach is probably the most prevalent. Indeed, in some texts, the terms "priority queue" and "binary heap" are interchangeable. A binary heap is very similar to a binary search tree, except that a binary heap is more strict about the shape of the tree (a heap is always perfectly balanced) but less strict about ordering (a parent's key must be less than its childrens keys, but there is no other restriction on the keys of siblings). While we usually visualize a binary heap as a binary tree of nodes, with some clever indexing, it is possible to store the entries in a vector. The vector approach avoids allocating and freeing node objects, and has better locality of reference, so has better constant factors than the node approach.

Rather than delve into the details of the binary heap, we present a simple priority queue built in terms of a self-balancing binary search tree. Our priority queue keeps a search tree, and also a pointer to the entry whose key is least. Since our search tree requires that every key is unique, but a priority queue must accomodate tied priorities, each key in the tree corresponds to a priority and each value in the tree is a queue of entries whose priorities are all tied.

```
class PriorityQueue:
    def __init__(self):
        self.bst = BST()
        self.min_entry = None
        self.size = 0
    def __len__(self):
        return self.size
    def _check_nonempty(self):
        if self.size == 0:
            raise UnderflowError
    def front(self):
        self._check_nonempty()
        return self.min_entry.value.front()
    def enqueue(self, priority, value):
        pqueue_entry = Entry(priority, value)
        tree_entry = self.bst.search(priority)
        if tree_entry is None:
            new_queue = Queue()
            new_queue.enqueue(pqueue_entry)
            tree_entry = Entry(priority, new_queue)
            self.bst.insert(priority, tree_entry)
        else:
            tree_entry.value.enqueue(queue_entry)
        if self.min_entry is None or priority < self.min_entry.key:
            self.min_entry = tree_entry
        self.size += 1
        return queue_entry
    def dequeue(self):
        self._check_nonempty()
        front_entry = self.min_entry.value.dequeue()
        if len(self.min_entry.value) == 0:
            self.bst.remove(front_entry.key)
            if len(self.bst) == 0:
                self.min_entry = None
            else:
                self.min_entry = self.bst.min()
        self.size -= 1
        return front_entry
```

Our priority queue can find the front entry in $O(1)$ time simply by following the min-entry pointer. To enqueue an entry we insert the entry into the search tree, and if necessary update the min-entry pointer. This takes $O(\log n)$ time to insert into the search tree, plus $O(1)$ additional time for the

Priority Queue Operation	Pseudocode	Time Complexity
Create an empty priority queue	`pq = PriorityQueue()`	$O(1)$
Get the number of entries	`len(pq)`	$O(1)$
Get minimum-priority entry from a non-empty priority queue	`entry = pq.front()`	$O(1)$
Add an entry with priority p and value x	`pq.enqueue(p, x)`	$O(\log n)$
Remove and return the minimum-priority entry	`entry = pq.dequeue()`	$O(\log n)$

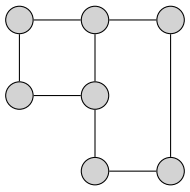
Table 4.8: Priority Queue Operations

min-pointer. To dequeue an entry we remove it from the search tree, then search the tree for the new min-entry (unless the priority queue just became empty). Each of those tree operations takes $O(\log n)$ time, so dequeueing takes $O(2 \log n) = O(\log n)$ total time.

In summary, our tree-based priority queue can find the minimum entry in $O(1)$ time; enqueue an entry in $O(\log n)$ time; and dequeue the next entry in $O(\log n)$ time. These efficiency classes match those of a binary heap, so our simple priority queue is equivalent in terms of asymptotic analysis. Though to be fair, a vector-based binary heap would be faster in terms of constant factors.

4.11 Graph

A *graph* is a mathematical object comprised of a set of *vertices* (singular *vertex*); and another set of *edges,* where each edge represents a connection between two vertices. The two vertices connected by an edge are called the *ends* of that edge. A graph may be *undirected* meaning that each edge defines a two-way connection. Or, a graph may be *directed,* in which case each edge defines a one-way connection from a *source* vertex into a *sink* index. For example, the following undirected graph has seven vertices and eight edges.



In formal math notation, an undirected graph G_U has the shape

$$G_U = (V, E) \text{ where } E = \{\{s, t\} \mid s, t \in V\}.$$

Each edge $e = \{s, t\}$ is a set of exactly two vertices. Note that mathematical sets have no inherent notion of ordering, so $\{s, t\}$ and $\{t, s\}$ denote the same set; this is deliberate and represents the fact that the two ends of an undirected edge work the same way.

By contrast a directed graph has the shape

$$G_D = (V, E) \text{ where } E = \{(s, t) \mid s, t \in V\}.$$

This definition is very similar, but here each edge is a *pair* $e = (s, t)$ (not a set). In a pair the order of the elements is distinguished, so (s, t) and (t, s) denote *different* pairs. This reflects the fact that the source and sink of a directed edge behave differently. By convention, in an edge $e = (s, t)$, the first vertex s is the source ("from") vertex while the second vertex t is the sink ("to") vertex; so e allows travel from s to t but not the other way.

There is a rich vocabulary and notation for graphs. A *mixed* graph may have directed or undirected edges. A graph is *empty* when V is empty; note that implies that the graph has no edges. By convention, $n = |V|$ is the number of vertices in a graph and $m = |E|$ is the number of edges. An edge or label may be annoted with another object called a *label*. A numeric label on an edge is called a *weight*, and when every edge in a graph has a weight, that graph is *weighted*. An edge e is *incident on* vertex v when v is one of e's ends (i.e. when e touches v). The *degree* of vertex v, written $deg(v)$, is the number of edges incident on v. For an edge e with ends s and t, t is the *opposite* of s and likewise s is the opposite of t. Edge e *connects* vertex a to vertex b b when e allows travel from a to b; either e is undirected and $a, b \in e$, or e is directed and $(a, b) = e$.

There are three leading data structures for graphs: the *edge list*, *adjacency matrix*, and *adjacency list*. Of the three structures, the adjacency list is the most complicated, but is also the most efficient for almost all applications.

In essence, an adjacency list maintains a linked list of vertex objects; a linked list of edge objects; and for each vertex v, a linked list of those edges incident on v. By virtue of using linked lists, an individual vertex or edge may be added in $O(1)$ time. Iterating through the list of edges incident on some vertex v takes only $O(deg(v))$ time, since the structure has a prepared linked list of precisely those edges. Deleting an edge involves deleting a node from a linked list and takes $O(1)$ time; deleting a vertex v takes all its incident edges with it, so takes $O(deg(v))$ time. To iterate through the list of all edges connecting vertex a to vertex b, we can iterate through whichever of a and b has a smaller degree, so this proces takes $O(\min(deg(a), deg(b)))$. Likewise determining whether two arbitrary vertices v and w are connected takes $O(\min(deg(v), deg(w)))$.

As with the self-balancing binary search tree, we do not present a complete implementation of an adjacency list structure, but only its interface. The data structure is spread across three related data types `Vertex`, `Edge`, and `Graph`. Our structure can store both directed and undirected edges, so the same data type is capable of representing directed, undirected, or mixed graphs.

A `Vertex` is an opaque class; users of the graph structure do not create `Vertex` objects, and instead access `Vertex` objects owned by a `Graph` object. Given a `Vertex v`, you may access its `v.label` field or call its `v.incident()` member function.

Vertex Operation	Pseudocode	Time Complexity
Access label	`v.label`	$O(1)$
Get an iterator for the edges incident on vertex v	`it = v.incident()`	$O(1)$

Table 4.9: Vertex Operations

```
class Vertex:
    def __init__(self, ...):
        self.label = label

    # Return an iterator for each Edge object incident on this vertex.
    def incident(self)
```

The `Edge` class is also opaque; it has publicly-visible fields `directed` (a Boolean), `s` and `t` (each a `Vertex`), and `label`. It also has member functions to determine whether an arbitrary vertex is an edge end; retrieve the opposite vertex of an edge end; and determine whether the edge connects two vertices.

```
class Edge:
    def __init__(self, directed, s, t, label, ...):
        self.directed = directed
        self.s = s
        self.t = t
        self.label = label

    # Return True when v is one of this edge's ends, or False otherwise.
    def is_end(self, v)

    # Assuming that v is one of this edge's ends, return the opposite end.
    def opposite(self, v)

    # Return True when this edge connects from v to w, or False otherwise.
    def connects(self, v, w)
```

Finally we have the over-arching `Graph` class. This class owns the collection of `Vertex` and `Edge` objects that make up a graph. It has member functions for adding, removing, iterating, and counting vertices and edges; and for determining whether the graph is directed, undirected, or

Edge Operation	Pseudocode	Time Complexity
Access directed flag (a Boolean)	`e.directed`	$O(1)$
Access s end vertex	`e.s`	$O(1)$
Access t end vertex	`e.t`	$O(1)$
Access label	`e.label`	$O(1)$
Determine whether vertex v is one of e's ends	`boolean = e.is_end(v)`	$O(1)$
Assuming v is one of e's ends, return the other end	`w = e.opposite(v)`	$O(1)$
Determine whether e connects from v to w	`boolean = e.connects(v, w)`	$O(1)$

Table 4.10: Edge Operations

mixed.

```
class Graph:
    # Create an empty graph.
    def __init__(self)

    # Add and return a new vertex with label l and no incident edges.
    def add_vertex(self, l)

    # Add an undirected edge between end vertices s and t and with label l.
    def add_undirected(self, s, t, l)

    # Add a directed edge from source vertex s to sink vertex t, with label l.
    def add_directed(self, s, t, l)

    # Remove Vertex v, and every Edge incident on v.
    def remove_vertex(self, v)

    # Remove Edge e.
    def remove_edge(self, e)

    # Return the number of vertices/edges.
    def vertex_count(self)
    def edge_count(self)

    # Return an iterator for every Vertex/Edge in the graph.
    def vertices(self)
    def edges(self)

    # Return True/False to indicate whether the graph is entirely
    # undirected, entirely directed, or mixed. When the graph is empty
    # these functions return False.
    def undirected(self)
    def directed(self)
    def mixed(self)
```

The `Graph` class also has member functions for obtaining an arbitrary vertex or edge, and for efficiently determining how a pair of vertices are connected (or not).

```
    # Return an arbitrary Vertex/Edge object. When the graph has none,
    # these functions throw ValueError
    def arbitrary_vertex(self)
    def arbitrary_edge(self)

    # Return True if there exists an Edge that connects from Vertex a to
    # Vertex b.
    def connected(self, a, b)

    # Return an iterator for every Edge that connects from Vertex a to Vertex b.
    def connections(self, a, b)

    # Return an arbitrary Edge that connects from Vertex a to Vertex b. If no
    # such Edge exists, throws ValueError.
    def connection(self, a, b)
```

Exercises

4-1. As discussed in Subsection 3.7.2 and Section 4.5, vectors and linked lists are natural competitors with some fundamental trade-offs. In general vectors are faster by a constant factor, but since adding an element takes $O(1)$ amortized time instead of worst-case time, linked lists may be preferable in engineering situations where a slow but predictable run time is preferable over a fast but unpredictable one. Describe a concrete programming use case for storing a sequence of objects where a vector would be a better choice, and another concrete use case where a linked list would be better.

4-2. Section 4.5 makes the claim that vectors have faster constant factors than do linked lists. Perform an experiment to test this claim. Write a program that adds n elements to the back of an empty vector, adds n elements to the back of an empty linked list, and measures the runtime of each step. Run your program for various large values of n (in the millions, say). According to this benchmark, which data structure is faster, and by how much?

4-3. Design an algorithm that reverses a doubly-linked list, *without creating any new node objects.*

4-4. Refactor (rewrite) our linked list class to avoid using the head and tail placeholder nodes. Your list structure should still have a head and tail pointer, but no placeholder nodes, so that the number of nodes in the list is exactly equal to the number of user-supplied elements. When the list is empty the head and tail pointers should both point to `None`. The time complexity of all operations in your list should match those in Table 4.3. Which variant do you prefer, and why?

Graph Operation	Pseudocode	Time Complexity
Create an empty graph	`g = Graph()`	$O(1)$
Create and return a vertex object with label ℓ	`v = g.add_vertex(l)`	$O(1)$
Create and return an undirected edge object between s and t with label ℓ	`e = g.add_undirected(s,t,l)`	$O(1)$
Create and return a directed edge object from s to t with label ℓ	`e = g.add_directed(s,t,l)`	$O(1)$
Remove vertex v and every edge incident on v	`g.remove_vertex(v)`	$O(deg(v))$
Remove edge e	`g.remove_edge(e)`	$O(1)$
Get the number of vertices	`n = g.vertex_count()`	$O(1)$
Get the number of edges	`m = g.edge_count()`	$O(1)$
Get an iterator for all vertices in arbitrary order	`it = g.vertices()`	$O(1)$
Get an iterator for all edges in arbitary order	`it = g.edges()`	$O(1)$
Return `True` when every edge is undirected	`boolean = g.undirected()`	$O(1)$
Return `True` when every edge is directed	`boolean = g.directed()`	$O(1)$
Return `True` when there is both a directed and undirected edge	`boolean = g.mixed()`	$O(1)$
Get an arbitrary vertex	`v = g.arbitrary_vertex()`	$O(1)$
Get an arbitrary edge	`e = g.arbitrary_edge()`	$O(1)$
Get an iterator over all edges connecting vertex a to b	`it=g.connections(a,b)`	$O(\min(deg(a), deg(b)))$
Get an arbitrary edge connecting a to b, or `None` if none exists	`g.connection(a, b)`	$O(\min(deg(a), deg(b)))$

Table 4.11: Graph Operations

4-5. Refactor (rewrite) our queue data structure so that, instead of using a doubly-linked list, it instead uses a singly-linked list and pointer to the last list node. The time complexity of all operations in your list should match those in Table 4.7. Which variant do you prefer, and why?

4-6. Refactor (rewrite) our queue data structure so that, instead of using a doubly-linked list, it instead uses two stacks. The time complexity of all operations in your list should match those in Table 4.7 (although some of the time bounds for your structure may be amortized). Which variant do you prefer, and why?

4-7. *Graph modeling.* For each of the following kinds of digital information, explain how that information could be stored in a graph data structure. Explain specifically what each vertex corresponds to, what each edge corresponds to, what each vertex label (if any) represents, what each edge label (if any) represents, and whether the graph is directed, undirected, or mixed. Draw a sketch of an example graph with a handful of vertices and edges.

(a) A street map including streets and intersections.

(b) A social network, such as the web of people and friendships present on a social network site such as Facebook or LinkedIn.

(c) The set of courses and prerequisite relationships for your major.

4-8. Section 4.11 lists three graph data structures, but only describes the adjacency list. Do some Internet research into the other two structures. How do they compare? In what applications should you use an adjacency matrix instead of an adjacency list.

4-9. Do some Internet research into the details of the list, stack, queue, and heap data structures available in the Python standard library. Do the time complexity bounds (i.e. big-O notation) of the Python operations match the ones listed in this Chapter? The API of the standard Python structures is different from the interfaces presented here; which API do you prefer, and why?

4-10. Repeat the previous question, but analyze the C++ standard template library (STL) vector, linked list, map, stack, queue, and priority queue.

4-11. Do some Internet research to find out which kind of self-balancing binary search tree (e.g. AA tree, AVL tree, etc.) is used in the C++ standard template library (STL) `map` and `set`. As far as you can tell, why is that kind of tree used over the alternatives listed in Section 4.7.

4-12. Do some Internet research into *pure-functional* data structures (sometimes called *immutable* data structures). Which data structures have pure-functional replacements, out of the vector, linked list, self-balancing binary search tree, stack, queue, priority queue, and graph? Which pure-functional structures are slower than their conventional counterparts (in terms of big-O efficiency classes)?

Chapter 5

The Naïve Pattern

5.1 The Big Idea

Now that we know how to analyze algorithms, we are finally ready to start designing them!

In general, something is "naïve" when it reflects a shallow understanding or lack of sophistication. A *naïve algorithm* is not based on any of the more elaborate patterns covered in later chapters. Instead, a naïve algorithm is one that you could design *ad hoc,* meaning that you could sketch out a first draft of pseudocode without developing a plan first. Naïve algorithms only involve the basic constructs of programming (if statements, for loops, and so on) used in conventional ways.

"Naïve" might sound like an insult, but that is not so. Many of the practical problems that arise in real-world software development can be solved efficiently by naïve algorithms. Naïve algorithms are an essential building block of practical software, so it is important for developers to master them. And, as we will see in Chapter 12, a naïve algorithm can actually be *optimal,* meaning that it is as fast as possible. In these cases, making a more complicated algorithm would be pointless since it would be slower, or at best equally efficient, as a simple naïve algorithm. Simpler algorithms and code are easier and cheaper to create and maintain, so it is best to keep things simple when we can.

5.2 Analyzing Loops

5.2.1 Properties of Summations

Naïve algorithms often involve loops, and sometimes nested loops. The complexity function for an algorithm with a loop usually involves a summation expression. This section covers some properties of summation expressions that are useful in simplifying summation expressions into closed form.

5.2.2 Definition of Summation Notation

We adopt the convention that summations start at index number F (for "first") and end at index L (for "last").

Definition 20. *A* summation *from F to L, where F and L are integers and $F \leq L$, over terms X_i, is*

$$\sum_{i=F}^{L} X_i \equiv X_F + X_{F+1} + X_{F+2} + \ldots + X_{L-1} + X_L.$$

Example 23.

$$\sum_{i=2}^{5} 2i = 2(2) + 2(3) + 2(4) + 2(5) = 28.$$

5.2.3 Splitting Summations

A summation from F to L may be split into two separate summations that together cover the same range of indices.

Lemma 12. *For integers $F, M,$ and L, with $F \leq M < L$,*

$$\sum_{i=F}^{L} X_i = (\sum_{i=F}^{M} X_i) + (\sum_{i=M+1}^{L} X_i).$$

Proof. By definition,

$$\begin{aligned}
\sum_{i=F}^{L} X_i &= X_F + \ldots + X_L \\
&= X_F + \ldots + X_M + X_{M+1} + \ldots + X_L \\
&= (X_F + \ldots + X_M) + (X_{M+1} + \ldots + X_L) \\
&= (\sum_{i=F}^{M} X_i) + (\sum_{M+1}^{L} X_i).
\end{aligned}$$

□

Example 24.

$$\sum_{i=1}^{10} i = \sum_{i=1}^{5} i + \sum_{i=6}^{10} i.$$

5.2.4 Factoring Out Constants

A factor that is present in all sum terms and constant with respect to the index i, may be factored out.

Lemma 13. *If c is constant with respect to i, then*

$$\sum_{i=F}^{L}(c \cdot X_i) = c\sum_{i=F}^{L} X_i.$$

Proof. By definition

$$\begin{aligned}\sum_{i=F}^{L}(c \cdot X_i) &= c \cdot X_0 + \ldots + c \cdot X_L \\ &= c(X_0 + \ldots + X_L) \\ &= c\sum_{i=F}^{L} X_i.\end{aligned}$$

□

Example 25.

$$\sum_{i=1}^{10} 7n = 7\sum_{i=1}^{10} n.$$

5.2.5 The Rectangular Sum Equation

Naïve algorithms tend to involve `for` loops whose body takes a constant number of steps. Our approach for analyzing loops is to write a sum

$$t_{loop} = \sum_{s \in S} x_s,$$

where each x_s is the number of steps made in step s. So the time complexity of naïve algorithms often involves these kinds of sums, where each x_s term is constant. The following Lemma gives us a way to reduce these summation expressions to a simple closed-form expression.

Lemma 14. *If F and L are integers with $F \leq L$, and c is constant with respect to i, then*

$$\sum_{i=F}^{L} c = [L - F + 1] \cdot c.$$

Proof. The definition of summation is

$$\sum_{i=F}^{L} X_i = X_F + X_{F+1} + \ldots + X_{L-1} + X_L.$$

The right hand side has exactly $L - F + 1$ terms.

$$\sum_{i=F}^{L} X_i = \underbrace{X_F + \ldots + X_L}_{L-F+1 \text{ terms}}.$$

By assumption each $X_i = c$ is constant with respect to i, so

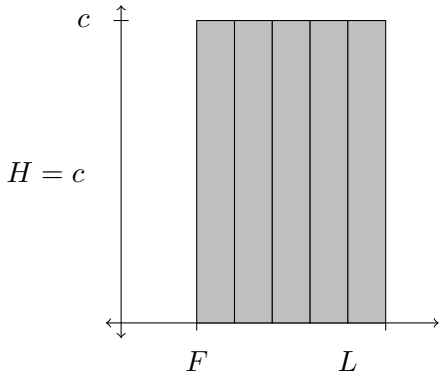
$$\sum_{i=F}^{L} X_i = \underbrace{c + \ldots + c}_{L-F+1 \text{ terms}}$$

or, combining like terms,

$$\sum_{i=F}^{L} X_i = [L - F + 1] \cdot c.$$

□

We call these summations *rectangular* because evaluating them corresponds to computing the area of a rectangle of width $[L - F + 1]$ and height c.



$$W = L - F + 1$$

$$\text{area of a rectangle } = W \cdot H = [L - F + 1] \cdot c$$

5.2.6 The Triangular Sum Equation

Another form of summation that arises from the analysis of algorithms is the *triangular sum*

$$\sum_{i=1}^{L} i.$$

Lemma 15. *If L is a positive integer, then*

$$\sum_{i=1}^{L} i = \frac{1}{2}L(L+1).$$

Proof. First suppose that L is even. By definition

$$\sum_{i=1}^{L} i = 1 + 2 + 3 + \ldots + (L-2) + (L-1) + L.$$

Suppose that we "pair up" the first term with the last term, to obtain $1 + L$; pair up the second term with the second-to-last term to obtain $2+(L-1)$; pair up the third term with the third-to-last term to obtain $(3 + (L-2))$; and so on. Every term may be paired up since L is even. Then we have

$$\sum_{i=1}^{L} i = (1+L) + (2 + (L-1)) + (3 + (L-2)) + \ldots.$$

Observe that each of these new terms is equal to $(L+1) : (1+L) = (2+(L-1)) = (3+(L-2)) = \ldots = (L+1)$, so we have

$$\sum_{i=1}^{L} i = (L+1) + (L+1) + \ldots.$$

The expression $(L+1)$ is constant with respect to i, so this is a rectangular sum. In order to use the rectangular sum formula, we need to know how many terms appear. Recall that each $(L+1)$ term corresponds to two terms from the original sum; so there are $L/2$ such terms ($L/2$ is an integer since L is even) and we have

$$\sum_{i=1}^{L} i = \underbrace{(L+1) + (L+1) + \ldots}_{L/2 \text{ terms}} = \sum_{i=1}^{L/2}(L+1) = [(L/2) - (1) + 1] \cdot (L+1) = \frac{1}{2}L(L+1).$$

Now suppose L is odd. We split the sum into the first $L-1$ terms, and the last term.

$$\begin{aligned} \sum_{i=1}^{L} i &= \sum_{i=1}^{L-1} i + \sum_{i=(L-1)+1}^{L} i \\ &= \sum_{i=1}^{L-1} i + L. \end{aligned}$$

Define $L' \equiv L - 1$; then we have

$$\sum_{i=1}^{L} i = \sum_{i=1}^{L'} i + L.$$

L is odd, so L' is even, and we may apply our equation from the even case to obtain

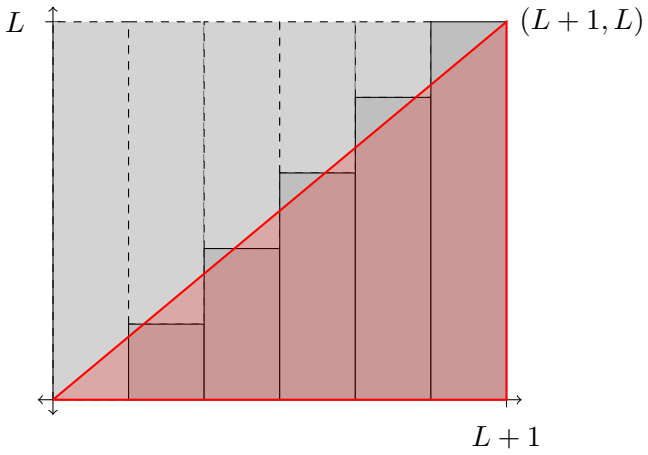
$$\sum_{i=1}^{L} i = \frac{1}{2}L'(L'+1) + L.$$

Substituting $L' = L - 1$ back in,

$$\begin{aligned}\sum_{i=1}^{L} i &= \frac{1}{2}(L-1)\big((L-1)+1\big)+L \\ &= \frac{1}{2}(L-1)L + \frac{1}{2}(2L) \\ &= \frac{1}{2}(L-1+2)L \\ &= \frac{1}{2}L(L+1).\end{aligned}$$

□

We call these summations *triangular* because evaluating them corresponds to computing the area of a triangle of base width $L+1$ and height L.



$$\text{area of a triangle } = \frac{1}{2}B \cdot H = \frac{1}{2}(L+1)L = \frac{1}{2}L(L+1)$$

5.3 Sequential Search

Now that we know how to analyze loops, we can tackle a foundational problem and the naïve algorithm that solves it. *Sequential search* involves finding an element of a list with a particular property.

sequential search
input: an iterator for a sequence S of n elements **output:** an element x of S containing `<A SPECIFIC PROPERTY>`, or `None` if no such element exists

This problem statement is itself a pattern, since it includes the blank `<A SPECIFIC PROPERTY>`. That blank can be filled in with any sort of predicate that can be applied to sequence elements, such as "is equal to 5" or "is odd" or "is a nonempty string."

A straightforward approach to solving this problem is to use a loop to check each element of S for the desired property. If we find a match, we return that element (immediately stopping the loop). However, if the loop ends without finding a match, we return `None` to indicate that S does not contain any match.

```
def sequential_search(S):
    for element in S:
        if element <SATISFIES THE SPECIFIC PROPERTY>:
            return element
    return None
```

This is clear pseudocode, aside from the intentional `<SATISFIES THE SPECIFIC PROPERTY>` blank. We can fill in the blank to obtain a sequential search for, say, the property of being the number zero.

```
def sequential_search_zero(S):
    for element in S:
        if element == 0:
            return element
    return None
```

Our presentation of the sequential search algorithm is not complete until we have proven its correctness and efficiency class.

Lemma 16. *The algorithm described by the* `sequential_search` *pseudocode correctly solves the sequential search problem.*

Proof. There are three cases of input S.

1. *S is empty.* In this case the `for` loop never iterates, and the algorithm returns `None`. This is a correct output in this case, since an empty $S = \langle\rangle$ can never contain a satisfactory $x \in S$.
2. *S is nonempty and contains no satisfactory x.* In this case the body of the `if` statement is never executed, so the algorithm returns `None`, which is a correct output.
3. *S is nonempty and does contain a satisfactory x.* In this case the body of the `if` statement is executed for the first element x such that the expression `<SATISFIES THE SPECIFIC PROPERTY>` evaluates to true. By the definition of the summation problem, that is a correct output for this case.

These three cases are exhaustive, so the algorithm produces a correct output for all problem instances. □

This formal proof is convincing, but it is also a bit tedious and pedantic. `sequential_search` 's pseudocode relates very closely to the definition of the sequential search problem. We would go so far as to say that `sequential_search` is clearly correct. In cases where pseudocode's correctness is readily apparent, we will claim its correctness without proof.

Lemma 17. *The sequential search algorithm runs in $O(n)$ time, provided that each element may be tested in $O(1)$ time.*

Proof. We count the number of steps $T(n)$ executed by the algorithm in chronological order. In the worst case, the `for` loop executes exactly n times. The `element in S` loop overhead takes 1 step per loop iteration. The body of the loop contains a single `if` statement. By assumption evaluating The Boolean expression for the `if` statement takes some constant number $c > 0$ of steps. When the expression is `True` , then the body `return element` takes 1 step. There is no `else` block, so when the expression is `False` , the `if` statement takes 1 step. Finally, the `return None` line takes 1 step. Altogether we have

$$\begin{aligned} T(n) &= \sum_{i=0}^{n-1} [1 + c + \max(1, 1)] + 1 \\ &= \sum_{i=0}^{n-1} (c + 2) + 1. \end{aligned}$$

The expression $c + 2$ is constant with respect to i, so this is a rectangular sum

$$\sum_{i=0}^{n-1} (c + 2) = [(n - 1) - (0) + 1] \cdot [c + 2] = (c + 2)n.$$

Substituting this into our equation for $T(n)$ gives

$$T(n) = (c + 2)n + 1.$$

By properties of O,

$$\begin{aligned} T(n) &= (c + 2)n + 1 \\ &\in O\big((c + 2)n + 1\big) && \text{(trivial)} \\ &= O\big((c + 2)n\big) && \text{(dominated term)} \\ &= O(n) && \text{(constant factor).} \end{aligned}$$

□

5.4 Sequential Optimization

In the context of algorithms, *optimization* means finding the best solution among competing alternatives. For example, the *traveling salesperson problem* is an optimization problem that involves

finding the fastest way of visiting a list of cities. The *knapsack problem* involves choosing the most valuable subset of items that can fit in a container. We will tackle both of those problems later in Chapters 7 and 13. For now, we consider the *sequential optimization problem,* which is a twist on the sequential search problem.

sequential optimization
input: an iterator for a sequence S of $n > 0$ elements **output:** the element x of S that is best according to `<A SPECIFIC MEASURE>`

There are two key differences between the sequential *optimization* problem, and the sequential *search* problem from the previous Section:

1. Instead of finding one element with a particular property, we are now searching for the best element according to a particular measure. (A measure is a way of deciding which of two elements is better).
2. The input requires $n > 0$, so we can be sure there is at least one element in S. This is necessary because the "best" element of an empty set is undefined. Since we can assume S is non-empty, we can always return some x, so there is no need to say that output could be `None` .

The traditional way to solve this problem is to use a variable `best` that holds the best element so far. To begin with, we make a guess about which element is best, and initialize `best` with any element of S. Then we loop through all the elements of S, and if we find an element x better than the current value of `best` , x becomes `best` .

```
def sequential_optimization(S):
    best = S[0]
    for element in S:
        if <element IS BETTER THAN best>:
            best = element
    return element
```

For example, the following algorithm finds the longest string in a non-empty sequence of strings.

```
def longest_string(S):
    best = S[0]
    for element in S:
        if len(element) > len(best):
            best = element
    return element
```

Note that the only differences between the `sequential_optimization` pattern and the clear

`longest_string` algorithm are the name, and that the `<element IS BETTER THAN best>` blank has been filled in with `len(element) > len(best)` .

`sequential_optimization` is a simple algorithm, but there are a few subtleties that are worth mention. The statement

```
    best = S[0]
```

looks suspiciously like a potential array-out-of-bounds error. However the assumption $n > 0$ guarantees that 0 is a valid index of `S` , so `S[0]` is safe. Also, recall that sequential search has a `return` statement inside the loop, so that loop may terminate before visiting every element. By contrast, sequential optimization never stops the loop early. It needs to consider each and every element of S. It is always possible that the very best element is at the end, and the algorithm has no way of knowing about that until it has examined all of the elements.

Lemma 18. *The sequential optimization algorithm runs in $O(n)$ time, provided that two elements can be compared in $O(1)$ time.*

Proof. Let $T(n)$ be the number of steps executed by the algorithm in the worst case. Initializing `best` takes 1 step. The `for` loop repeats exactly n times. Each iteration spends 1 step on loop overhead. By assumption the comparison takes $O(1)$ time, so the comparison takes c steps for some positive constant c. In the worst case, the `best = element` statement happens in *every* loop iteration, costing 1 more step per iteration. Finally, after the loop finishes, the return statement takes 1 more step. In total, the number of steps is

$$\begin{aligned} T(n) &= 1 + \sum_{i=0}^{n-1} (1 + c + 1) + 1 \\ &= \sum_{i=0}^{n-1} (c + 2) + 2 \end{aligned}$$

which, by the rectangular sum rule, simplifies to

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

By properties of O,

$$T(n) \in O\big((c + 2)n + 2\big) = O\big((c + 2)n\big) = O(n).$$

□

5.5 One-at-a-Time Sorting

The next problem that we consider is that of sorting a list. The conventional naïve approach is to sort one element at a time.

sorting problem
input: a list U of n comparable elements **output:** a list S containing the elements of U in non-decreasing order

Note that the input definition requires that elements of the vector be *comparable;* the notion of comparability, and its implementation in the Python language, were described in Section 4.2.3.

The output definition uses the term *non-decreasing order* which ought to be clarified.

Definition 21. *A sequence* $S = \langle s_0, s_1, \ldots, s_{n-1} \rangle$ *of comparable elements is in* non-decreasing order *when, for any adjacent* s_i *and* s_{i+1},

$$s_i \leq s_{i+1}.$$

We use the term "non-decreasing order" instead of the more concise "increasing order" since a sequence with duplicated elements can never be put into strictly increasing order, but may be put into non-decreasing order. The sequence $\langle 1, 3, 3, 7 \rangle$ is not in strict increasing order since $3 = 3$, but it is in non-decreasing order. If the sorting problem definition insisted that S be in strict increasing order, then any instance with duplicated elements would have no solution and the problem would be unsolvable.

Sorting is a well-studing problem in computer science, for both theoretical and practical reasons. As we will see in Chapter 12, the lower bound for sorting is a landmark result that lays the foundation for lower bounds of other problems. And as we will see in Chapter 10, a sorting algorithm can be used to solve many different problems that do not outwardly resemble the sorting problem. For this reason, software developers are expected to know how to use sorting to solve practical programming problems, most programming languages have a built-in sorting function, and these functions are heavily optimized.

You may have experience with sorting physical objects, such as alphabetizing books, or arranging a hand of playing cards into low-to-high order. A common naïve approach is to sort one element at a time: make sure that one element is sorted, and repeat that process until all of the elements are sorted.

```
def one_at_a_time_sort(U):
    S = []
    for element in U:
        <MOVE element INTO S, KEEPING S SORTED>
    return S
```

(Recall that `[]` means an empty list.)

This is a start, but is not a clear algorithm because the `<MOVE element INTO S, KEEPING S SORTED>` step is vague. Depending on how we fill in this blank, we will obtain a different sorting algorithm. In this Chapter we will consider one approach that results in the classical *insertion sort* algorithm.

Different approaches result in different classical algorithms: *insertion sort*, *bubble sort*, and *heap sort*.

We can say something about the correctness of efficiency of algorithms based upon this general pattern.

Lemma 19. *If, in the insertion step inside the loop of a one-at-a-time sorting algorithm, the algorithm correctly inserts an element x into a non-decreasing list S such that the new version of S is still in non-decreasing order, then the algorithm solves the sorting problem correctly.*

Proof. We argue by induction on the length $k = |S|$ of S.

Base case. In the base case $k = 0$; this is the case after `S` is created as an empty list and before the `for` loop begins. An empty list cannot possibly have two adjacent elements out of order, and so is in non-decreasing order (trivially).

Inductive case. Suppose S is in non-decreasing order and $|S| = k$. By assumption, an insertion step will modify S to become S' of length $|S'| = k + 1$, and this S' is also in non-decreasing order. Therefore S' of length $k' = k + 1$ is in non-decreasing order.

By induction, S is in non-decreasing order for any length of S. □

Lemma 20. *If a one-at-a-time sorting algorithm spends I worst-case time in each insertion step, then the worst case time efficiency of the entire algorithm is $O(n \cdot I)$.*

Proof. The one-at-a-time sorting algorithm pattern spends one step creating `S`. The `for` loop iterates exactly n times; each iteration involves 1 step of loop overhead, and by assumption another I steps moving x into S. Finally the `return` statement takes 1 step. So the total time spent by the algorithm is

$$T(n) = 1 + \sum_{i=0}^{n-1}(1 + I) + 1 = nI + n + 2$$

which is

$$T(n) \in O\big(nI + n + 2\big) = O\big(nI\big)$$

by properties of O. □

5.5.1 Invariants

An *invariant* is a property that a data structure or variable is obliged to maintain. Any operation that modifies that structure or variable is required to ensure that it either leaves the invariant undisturbed; or, if the operation may invalidate the invariant, it eventually restores it.

A familiar example is that of a binary search tree. As discussed in Section 4.7, a binary search tree maintains the invariant that, for any node p containing key k, all left descendants of p contains keys less than k, while all right descendants of p contain keys greater than k. Some binary search

tree operations, such as searching, do not jeopardize the invariant. The insertion operation is very careful to insert a new node in a position that maintains the invariant. Notoriously, the deletion operation may temporarily break the invariant while removing a parent node with two children; the deletion algorithm must take pains to restore the invariant in this case, which is a rather intricate process.

One-at-a-time sorting algorithms also maintain an invariant: that at all times, the output list `S` is in non-decreasing sorted order. This is implicit in the phrasing of the `<MOVE element INTO S, KEEPING S SORTED>` step. It is not acceptable to insert an element anywhere in `S`; the flow of the algorithm must ensure that, every time an element is added to `S`, the list remains sorted.

5.5.2 Basic Selection Sort

To turn the one-at-a-time sorting pattern into a clear algorithm, we need to determine how to handle the move step inside the `for` loop. The selection sort algorithm [34] approaches this issue by carefully *selecting* which element to insert in each step. At each step, the next element that should go into S is the *least* (smallest) element remaining in U. So selection sort:

- Finds the smallest element in U, removes it, and inserts it as the first and only element of S.
- Finds the smallest element still in U, which is the second-smallest element overall, removes it, and appends it to the end of S.
- Finds the smallest element still in U, which is the third-smallest element overall, removes it, and appends it to the end of S.
- ...
- Finds the smallest element still in U, which is the nth-smallest element overall (i.e. the largest element), removes it, and appends it to the end of S.

Selection sort gets its name from the fact that it dedicates a lot of time and attention to the matter of *selecting* which element to append next.

Our first rough draft of selection sort is the one-at-a-time sorting pattern pseudocode with minimal changes.

```
def selection_sort(U):
    S = []
    for element in U: # need to guarantee this is the least element still in L
        <MOVE element INTO S, KEEPING S SORTED>
    return S
```

The `<MOVE element INTO S, KEEPING S SORTED>` part can be handled succinctly since we know that `element` is always appended to the end of `S`.

```
def selection_sort(U):
    S = []
    for element in U: # need to guarantee this is the least element still in L
        S.add_back(element)
    return S
```

We still need to deal with the selection process that gives selection sort its name. This is awkward to express as a `for` loop, so we switch to a `while` loop. We also rename `element` to `least` for the sake of clarity.

```
def selection_sort(U):
    S = []
    while U is not empty:
        least = <FIND THE SMALLEST ELEMENT IN U>
        <REMOVE least FROM U>
        S.add_back(least)
    return S
```

The `<REMOVE least FROM U>` step is nontrivial because `least` could be at any index of `U`, our list data structure does not have an operation to remove an element at an arbitrary index; it is only capable of removing the back element at index -1 (in Python notation, index -1 is the last position i.e. $n-1$, index -2 is the second-to-last position i.e. $n-2$, and so on). The classic trick to solve this problem is to *swap* `least` with the element currently in `U[-1]`. That moves `least` to the back of the vector, where it may be removed with the `remove_back` operation, while the element that was previously in the back element is relocated to some other safer index. In order to swap the elements we need to determine the index of `least`.

```
def selection_sort(U):
    S = []
    while U is not empty:
        least = <FIND THE SMALLEST ELEMENT IN U>
        least_index = <INDEX OF least>

        # remove least from V
        swap(U[least_index], U[-1])
        U.remove_back()

        # add least to S
        S.add_back(least)
    return S
```

Finding `least` and `least_index` is a variation on sequential optimization. We initially guess that the least element is at index 0, then loop through the other indices, looking for lesser elements.

```
def selection_sort(U):
    S = []
    while U is not empty:
        # find the least unsorted element
        least = U[0]
        least_index = 0
        for i in range(1, len(U)):
            if U[i] < least:
                least = U[i]
                least_index = i

        # remove least from U
        swap(U[least_index], U[-1])
        U.remove_back()

        # add least to S
        S.add_back(least)
    return S
```

In this draft, the variable `least` is never actually used, so we might as well eliminate it. Figure 5.1 shows our final draft of the selection sort algorithm. This is clear, serviceable pseudocode for a selection sort algorithm. In the next section we will see how to improve it a bit. First, we analyze the efficiency of this version of the algorithm.

```
def selection_sort(U):
    S = []
    while U is not empty:
        # find the least unsorted element
        least_index = 0
        for i in range(1, len(U)):
            if U[i] < U[least_index]
                least_index = i

        # remove least from U
        swap(U[least_index], U[-1])
        U.remove_back()

        # add least to S
        S.add_back(U[least_index])
    return S
```

Figure 5.1: Basic selection sort.

Lemma 21. *The worst case time complexity of* `selection_sort` *is* $O(n^2)$.

Proof. We count the number of steps executed in each iteration of the outer `while` loop so that we may then apply Lemma 20. Initializing `least_index` takes 1 step. The inner `for` loop iterates at most $n-1$ times; each iteration involves 3 steps in the worst case. The swap and `remove_back` operations each take $O(1)$ time, so they account for 2 more steps. Finally the `add_back` operation takes $O(1)$ time which we count as 1 step. In total, each iteration of the loop takes

$$I = 1 + (n-1)(3) + 2 + 1 = 3n + 1$$

steps. So by Lemma 20, the total time complexity of `selection_sort` is

$$O(n \cdot I) = O(n \cdot (3n+1)) = O(3n^2 + n) = O(n^2)$$

by properties of O. □

5.5.3 In-Place Selection Sort

The pseudocode in Figure 5.1 is correct, and its $O(n^2)$ time complexity is acceptable. However, implementers (programmers) typically use a version of selection sort that differs in several ways. The differences stem from optimizations that make the algorithm faster. This Section describes this *in-place selection sort* algorithm.

The first difference, and the one that gives the algorithm its name, is that the optimized algorithm is *in-place*. By contrast, the selection sort algorithm from the previous section is *pure*.

Definition 22. *An algorithm is* pure *when it leaves its arguments and global variables unchanged.*

Observe that the `selection_sort` pseudocode takes `U` as an argument, and produces and returns a new list object `S`, leaving `U` completely unchanged. So that algorithm is pure.

Definition 23. *An algorithm is* in-place *when its output is stored in the same data structure as its input.*

That is, rather than allocating a new list object to store S, the original list object U is rearranged until it is in non-decreasing order. The benefit of this change is that it allows us to avoid the process of allocating a new list object, and eventually freeing the old one. Allocating and freeing memory is typically a relatively slow single-step operation.

How can this be done? Observe that each iteration of the `while` loop moves one element (`least`) out of U and into S. So at all times $|U| + |S| = n$. Rather than gradually shrinking U and growing S one element at a time, instead we can use a single list U. The list is divided into a *sorted zone* and *unsorted zone*, with a crisp boundary between them. The following invariant is a guide to how we can keep the two zones straight; shortly, we will write pseudocode that adheres to this invariant. (The invariant is written in terms of list slices, which were defined in Subsection 3.6.3.)

Invariant 1. *After k iterations of the `while` loop in in-place selection sort, `U[:k]` are in non-decreasing order, while `U[k:]` may be in any order.*

Figure 5.2 illustrates these zones at several stages of the execution of the algorithm.

With that invariant in place, we can start adapting the basic selection sort pseudocode from Figure 5.1 to be in-place. First we will simply replace all references to `U` and `S` with the placeholder texts `<UNSORTED ZONE>` and `<SORTED ZONE>`.

```
def in_place_selection_sort(U):
    <UNSORTED ZONE> = U
    <SORTED ZONE> = empty
    while <UNSORTED ZONE> is not empty:
        least_index = <INDEX OF SMALLEST ELEMENT IN UNSORTED ZONE>

        <SWAP U[least_index] TO THE END OF THE UNSORTED ZONE>
        <REMOVE THE LEAST ELEMENT FROM THE UNSORTED ZONE>

        <ADD THE LEAST ELEMENT TO THE SORTED ZONE>
    return <SORTED ZONE>
```

`<UNSORTED ZONE>` is an alias for the first k elements of `U`, and `<SORTED ZONE>` is an alias for the remaining elements of `U`. So the first two statements `<UNSORTED ZONE> = U` and `<SORTED ZONE> = empty`

1. Before the first iteration, when $k = 0$:

unsorted `U[0:n]`

2. When $k = 1$, after the first loop iteration:

sorted `U[0]`	unsorted `U[1:n]`

3. Early, when $k \approx \frac{1}{3}n$:

sorted `U[0:k]`	unsorted `U[k:n]`

4. Later, when $k \approx \frac{2}{3}n$:

sorted `U[0:k]`	unsorted `U[k:n]`

5. When $k = n - 1$, with only one element remaining in the unsorted zone:

sorted `U[0:n-1]`	unsorted `U[n-1]`

6. When $k = n$ and the entire list is in non-decreasing order:

sorted `U[0:n]`

Figure 5.2: The ordered and unordered zones of L throughout the execution of the in-place selection sort algorithm.

are unnecessary. As can be seen in Figure 5.2, before the loop starts, the entirety of `U` is considered unsorted and none of it is considered sorted. The algorithm need not take any explicit action to make this so, so we simply delete those statements. Also, at the end of the algorithm, `<SORTED ZONE>` is the entirety of `U`, so the statement `return <SORTED ZONE>` may be rewritten `return U`.

```
def in_place_selection_sort(U):
    while <UNSORTED ZONE> is not empty:
        least_index = <INDEX OF SMALLEST ELEMENT IN UNSORTED ZONE>

        <SWAP U[least_index] TO THE END OF THE UNSORTED ZONE>
        <REMOVE THE LEAST ELEMENT FROM THE UNSORTED ZONE>
        <ADD THE LEAST ELEMENT TO THE SORTED ZONE>
    return U
```

In order for the algorithm to know where the boundary between the zones is, we need to store the value k explicitly in a variable. We can do that by changing the main loop back to a `for` loop that uses a loop counter `k`.

```
def in_place_selection_sort(U):
    for k in range(len(U)):
        least_index = <INDEX OF SMALLEST ELEMENT IN UNSORTED ZONE>

        <SWAP U[least_index] TO THE END OF THE UNSORTED ZONE>
        <REMOVE THE LEAST ELEMENT FROM THE UNSORTED ZONE>
        <ADD THE LEAST ELEMENT TO THE SORTED ZONE>
    return U
```

According to Invariant 1, the sorted zone is in `U[0:k]`, while the unsorted zone is in `U[k:n]`. We can rewrite the `<INDEX OF SMALLEST ELEMENT IN UNSORTED ZONE>` step in terms of a loop through the unsorted zone.

```
def in_place_selection_sort(U):
    for k in range(len(U)):
        least_index = k
        for i in range(k+1, len(V)):
            if U[i] < U[least_index]:
                least_index = i

        <SWAP U[least_index] TO THE END OF THE UNSORTED ZONE>
        <REMOVE THE LEAST ELEMENT FROM THE UNSORTED ZONE>
        <ADD THE LEAST ELEMENT TO THE SORTED ZONE>
    return U
```

Now we have a problem: how can we remove the least element from the unsorted zone, and add it to the sorted zone? In principle we could delete an element out of `U`, shortening it, then insert an element back in, lengthening it again. But then the algorithm would not be in-place.

The solution is based on a subtle observation: according to our invariant, after the `<ADD THE LEAST ELEMENT TO THE SORTED ZONE>` step, `U[k]` must contain the least element. Once that is achieved, the sorted zone implicitly grows by one element to include index `k` and element `U[k]`. If we were to overwrite `U[k]` with `U[least_index]`, that would create two copies of `U[least_index]` and simultaneously destroy the element currently at `U[k]`. Instead we use the same elegant trick that worked in the pure version of selection sort: swap the two elements. We swap `U[least_index]` with `U[k]`, which simultaneously moves the least unsorted element out of the unsorted zone and into the sorted zone, and moves the old value `U[k]` into the unsorted zone, which is a valid place for it to stay.

```
def in_place_selection_sort(U):
    for k in range(len(U)):
        least_index = k
        for i in range(k+1, len(U)):
            if U[i] < U[least_index]:
                least_index = i

        swap(U[least_index], U[k])
    return U
```

There is one final optimization that we can make. Examine diagram 5 in Figure 5.2. At this point, after iteration $k = n - 1$ has finished, only one element remains in the unsorted zone. After that point our pseudocode will execute one last iteration of the outer loop, which will do a sequential search through the one element `U[n-1]`, then swap `U[n-1]` with itself. This last iteration is a waste of effort! If the first $n - 1$ elements of `U` are properly sorted, then by process of elimination

the only element that can possibly remain in `U[n-1]` is the *greatest* element in `U`, which is already in its proper ordered position. So we can leave it be, by skipping the last iteration of the outer loop.

Figure 5.3 shows our final draft of the in-place, optimized version of selection sort. This is the variant of selection sort which is most often implemented.

```
def in_place_selection_sort(U):
    for k in range(len(U) - 1):
        least_index = k
        for i in range(k+1, len(U)):
            if U[i] < U[least_index]:
                least_index = i

        swap(U[least_index], U[k])
    return U
```

Figure 5.3: In-place selection sort.

We now analyze our whizbang optimized algorithm.

Lemma 22. *The worst case time complexity of* `in_place_selection_sort` *is* $O(n^2)$.

Proof. The outer loop iterates k through the range $[0, n-2]$. Initializing `least_index` takes 1 step. The inner loop iterates i through the range $[k+1, n-1]$. Each iteration of the inner loop takes at most 3 steps. The swap takes one step, as does the `return` statement. So the total number of steps is

$$\begin{aligned} T(n) &= (\sum_{k=0}^{n-2}(1 + \sum_{i=k+1}^{n-1}(3) + 1)) + 1 \\ &= 1 + \sum_{k=0}^{n-2}(2 + \sum_{i=k+1}^{n-1} 3). \end{aligned}$$

We cannot simplify these nested sums all at once, so as is usual in algebra, we work our way outwards starting with the most deeply nested expression. That is the sum

$$\sum_{i=k+1}^{n-1} 3$$

which, by the rectangular sum formula (Lemma 14), simplifies to

$$\sum_{i=k+1}^{n-1} 3 = [(n-1) - (k+1) + 1] \cdot 3 = 3(n-k-1).$$

Substituting this back into our equation for $T(n)$,

$$\begin{aligned} T(n) &= 1 + \sum_{k=0}^{n-2} (2 + 3(n-k-1)) \\ &= 1 + \sum_{k=0}^{n-2} (3n - 3k - 1). \end{aligned}$$

The expression $(3n - 3k - 1)$ is *not* constant with respect to the sum counter variable k, so we cannot apply the rectangular sum formula to the entire sum. However, we can break it into two sum expressions, one of which can be simplified with the formula.

$$\begin{aligned} T(n) &= 1 + \sum_{k=0}^{n-2} (3n - 1) + \sum_{k=0}^{n-2} (-3k) \\ &= 1 + \sum_{k=0}^{n-2} (3n - 1) - 3 \sum_{k=0}^{n-2} k. \end{aligned}$$

Applying the rectangular sum formula to the left summation and the triangular sum formula (Lemma 15 to the right summation,

$$\begin{aligned} T(n) &= 1 + [(n-2) - (0) + 1] \cdot (3n - 1) - 3[\frac{1}{2}(n-2)((n-2)+1)] \\ &= 1 + [n-1] \cdot (3n-1) - \frac{3}{2}(n-2)(n-1) \\ &= 1 + (3n^2 - 4n + 1) - \frac{3}{2}(n^2 - 3n + 1) \\ &= \frac{3}{2}n^2 + \frac{1}{2}n + 1. \end{aligned}$$

By properties of O,

$$T(n) \in O(\frac{3}{2}n^2 + \frac{1}{2}n + 1) = O(\frac{3}{2}n^2) = O(n^2).$$

□

It may be counter-intuitive that our optimized in-place algorithm has the same time efficiency class $O(n^2)$ as the original unoptimized version of selection sort. As counter-intuitive as it may be, it is true. According to our mathematical analysis, our valiant efforts to speed up the selection sort algorithm did not make a significant difference. The big-picture trend for both algorithms is that they run in quadratic time. Now, the leading term of the time complexity of the unoptimized algorithm is $3n^2$, and the leading term of the optimized algorithm's time complexity is $\frac{3}{2}n^2$. This suggests that the optimized version would be faster than the unoptimized version by a factor of 2. However that constant-factor improvement is dwarfed by the $O(n^2)$ time complexity trend of both algorithms.

Exercises

5-1. Write pseudocode for a sequential search algorithm that searches through a list of strings to find a non-empty string.

5-2. Is the sequential search algorithm correct for the case of an input list containing multiple satisfactory elements? Justify your answer.

5-3. For each of the following summation expressions, either simplify the expression using the rectangular sum formula, or explain why the formula is not applicable.

(a)
$$\sum_{i=0}^{n-1} 5$$

(b)
$$\sum_{i=0}^{n-1} i$$

(c)
$$\sum_{i=0}^{n-1} n$$

(d)
$$\sum_{i=0}^{n-1} (2 + \sum_{j=0}^{n-1} 1)$$

(e)
$$\sum_{i=0}^{n-1} (1 + \sum_{j=0}^{i} 3)$$

5-4. Prove Lemma 15, that posits the triangular sum formula

$$\sum_{i=1}^{L} i = \frac{1}{2} L(L+1)$$

by induction.

5-5. Sort the characters of the string "SEQUENCE" using pure selection sort; show your work.

5-6. Sort the characters of the string "SEQUENCE" using in-place selection sort; show your work.

5-7. For each of the following problems: design a naïve algorithm that solves the problem; describe your algorithm with clear pseudocode; and prove the time efficiency class of your algorithm.

(a)

maximum problem
input: a non-empty sequence S of n orderable objects **output:** the greatest (maximum) element in S

(b)

list reversal
input: a list L of n elements **output:** a list containing the elements of L but in reversed order

(c)

reverse sorting problem
input: a list U of n comparable elements **output:** a list S containing the elements of U in non-increasing order

Hint: Base your algorithm on selection sort, and make as few changes as possible.

(d) If x, y are two adjacent elements in a sequence, with x before y, we say that the pair x, y is *in order* when $x \leq y$ and the pair is *out of order* when $x > y$. For example, in the string

"BEGGAR"

the pair G, A are out of order, but all the other pairs are in order.

out-of-order counting
input: a list L of n comparable elements **output:** the number of out-of-order pairs in L

Hint: An optimal algorithm for this problem takes $O(n)$ time.

5-8. Rewrite selection sort so that its input and output are linked lists instead of arrayed lists. What is the time complexity of this version of selection sort?

5-9. Suppose that selection sort is applied to a list that is already in sorted order. What is the time complexity of the algorithm this case?

5-10. Design a sorting algorithm whose time complexity is as follows:

(a) when the input happens to be already non-decreasing, the algorithm takes only $O(n)$ time;

(b) likewise, when the input happens to be already non-increasing (i.e. reverse-sorted), the algorithm takes only $O(n)$ time;

(c) but in any other situation, the algorithm may take $O(n^2)$ time.

Justify that your algorithm meets these efficiency goals.

Chapter 6

The Greedy Pattern

6.1 The Big Idea

The *greedy* pattern is perhaps the simplest and most straightforward algorithm pattern. It is inspired by the way that someone might tackle a menial, tedious real-world task such as digging a hole or washing a sink full of dishes. Essentially, the idea is to deal with one piece of input, and repeat that until the input has been handled completely.

```
def greedy_pattern(<INPUTS>):
    todo = <PIECES OF INPUTS>
    result = <INITIAL RESULT>
    for element in todo:
        <PROCESS element AND UPDATE result>
    return result
```

This pattern has three blanks: `<PIECES OF INPUTS>` is an expression for extracting a list of individual elements to deal with from the entire problem instance; `<INITIAL RESULT>`, which is often a trivial datum such as an empty list or the number zero; and `<PROCESS element AND UPDATE result>`, which is the step that handles an individual piece of input to move us closer to being done.

We can fill in these blanks to form a (non-computational) algorithm for washing dishes by hand:

```
def wash_dishes(vector_of_dishes):
    todo = vector_of_dishes
    result = empty vector
    for dish in todo:
        <WASH>(dish)
        result.add_back(dish)
    return result
```

The `<WASH>` function is still left unspecified, but hopefully you get the idea. We could also write a greedy, non-computational algorithm for digging a hole. Our first draft:

```
def dig_hole(depth):
    todo = depth
    result = <INITIAL RESULT>
    for inch in todo:
        <DIG ONE INCH DEEPER>
    return result
```

In writing this draft, it is evident that this algorithm doesn't really need to compute a result or return it. So, we can delete those parts of the algorithm. While we're at it, we rewrite the `inch in todo` expression, which is unclear since `todo` is an integer and not a list as required by Python syntax.

```
def dig_hole(depth):
    todo = depth
    for inch in range(todo):
        <DIG ONE INCH DEEPER>
```

These real-world examples are hopefully familiar, but we should get back to computational problems and algorithms.

summation problem
input: a list X of numbers **output:** the sum (total) of all elements of X

To design an algorithm for this problem, we first make a rote copy of the greedy, with only the name changed.

```
def sum(X):
    todo = <PIECES OF INPUTS>
    result = <INITIAL RESULT>
    for element in todo:
        <PROCESS element AND UPDATE result>
    return result
```

In our second draft, we fill in the blanks with some text that is specific to the summation problem, leaving only one unclear step.

```
def sum(X):
    todo = X
    result = 0
    for element in todo:
        <add element to result>
    return result
```

We know how to `<add element to result>`.

```
def sum(X):
  todo = X
  result = 0
  for element in todo:
    result += element
  return result
```

Finally, we observe that the `todo` variable is initialized, then only used once. This is a waste of computer time, and worse, makes our pseudocode overcomplicated. We remove `todo` and operate on `X` directly instead.

```
def sum(X):
  result = 0
  for element in X:
    result += element
  return result
```

This is our final draft of the `sum` algorithm.

6.2 American Change-Making

For our purposes, *change-making* involves choosing a collection of coins worth a specific monetary value. If you were to buy a sandwich for $3.55 and hand the cashier four dollar bills, they would need to do some change-making to figure out which coins to use to give you 45 cents.

In the USA, coins are widely distributed in denominations of 1 (penny), 5 (nickel), 10 (dime), and 25 (quarter) cents. Coins worth 50 and 100 cents exist, but are relatively uncommon, so we will ignore them. The cashier in the previous example could have made 45 cents out of four dimes and one nickel; one quarter and two dimes; 9 nickels; 45 pennies; and many other combinations. People tend to prefer counting out as few coins as possible, so we seek a solution that minimizes the number of coins used.

American change-making
input: a number of cents $k \geq 0$ **output:** a list V of coin values drawn from $\{1, 5, 10, 25\}$ such that $(\sum_{c \in V} c) = k$ and the length of V is minimal

Cashiers throughout the land seem to use the same algorithm:

1. Use as many quarters as you can.

2. If you still owe cents, use as many dimes as you can.

3. If you still owe cents, use as many nickels as you can.

4. If you still owe p cents, use p pennies.

We can think of this as a greedy algorithm, where `PROCESS an element` means using the largest coin that we can.

```python
def greedy_american_change_making(k):
    owe = k
    V = LinkedList()
    while owe > 0:
        # choose the largest denomination we can
        if owe >= 25:
            denomination = 25
        elif owe >= 10:
            denomination = 10
        elif owe >= 5:
            denomination = 5
        else:
            denomination = 1

        V.add_back(denomination)
        owe -= denomination

    return V
```

It should be clear that this algorithm correctly generates a vector V of denominations that add up to k. Intuitively the length of V is minimal, but in order to establish that we need to prove a lemma, such as the following.

Lemma 23. *Let $k \geq 0$ be an arbitrary number of cents, V be the sequence produced by the greedy change-making algorithm for k, and X be an arbitrary sequence of $1, 5, 10, 25$ such that $\sum X = k$; then $|V| \leq |X|$.*

Proving this turns out to be surprisingly challenging, and is left as an exercise.

6.3 Minimum Spanning Trees

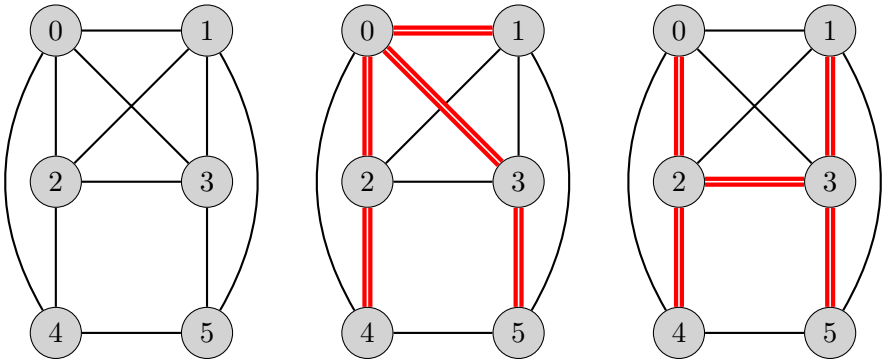
A *spanning tree* for a graph G is a subtree that includes all of G's vertices, and only enough edges to connect the vertices without creating a cycle (loop).

Definition 24. *In an undirected graph $G = (V, E)$, vertices $v, w \in V$ are* connected *if either $v = w$ or there exists a path between v and w. The entire graph G is connected if, for all $v, w \in V$, v and w are connected.*

Definition 25. *A* spanning tree *of a graph $G = (V, E)$ is a subgraph $T = (V, K)$ where $K \subseteq E$ is a subset of edges, such that T is connected and acyclic.*

A graph may have multiple valid spanning trees.

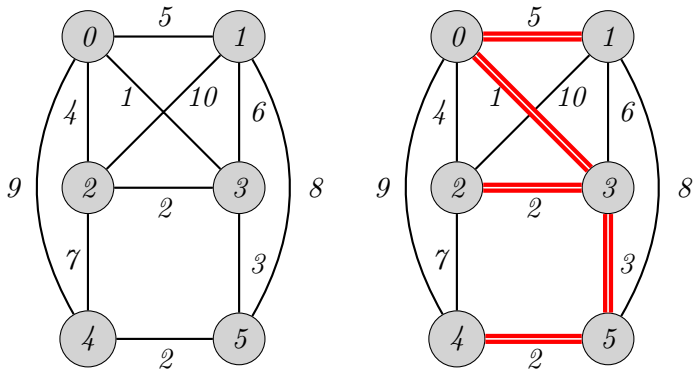
A graph G_6 of $n = 6$ vertices is shown on the left; two possible spanning trees are shown to the right, where tree edges are drawn as red double lines.



When a graph is weighted, any of its spanning trees have a defined weight, and some of those spanning trees may be of minimum weight.

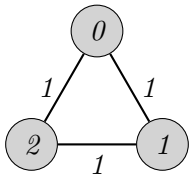
Definition 26. *A* minimum spanning tree *of a weighted graph G is a spanning tree T of minimal total edge weight.*

Example 26. *A weighted version of G_6, and its two minimum spanning tree, are shown below.*

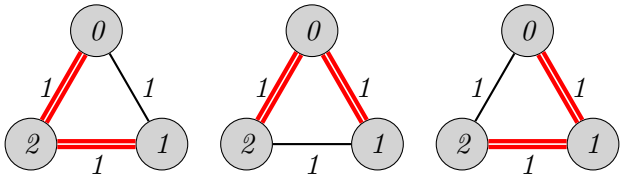


Our definition says *a* minimum spanning tree, not *the* minimum spanning tree. This is because edge weights may be duplicated, so a graph may have multiple spanning trees that are tied for minimum.

Example 27. *The 3-vertex graph*



has 3 distinct minimum spanning trees, all of total weight exactly 2.



minimum spanning tree problem
input: a connected, undirected, and weighted graph $G = (V, E)$ with $n = \|V\|$ vertices and $m = \|E\|$ edges **output:** a set of edges K such that $T = (V, K)$ is a minimum spanning tree for G

6.3.1 Greedy Spanning Trees

The output of the minimum spanning tree problem is a set of edges, so a greedy algorithm for the problem will involve choosing those edges greedily.

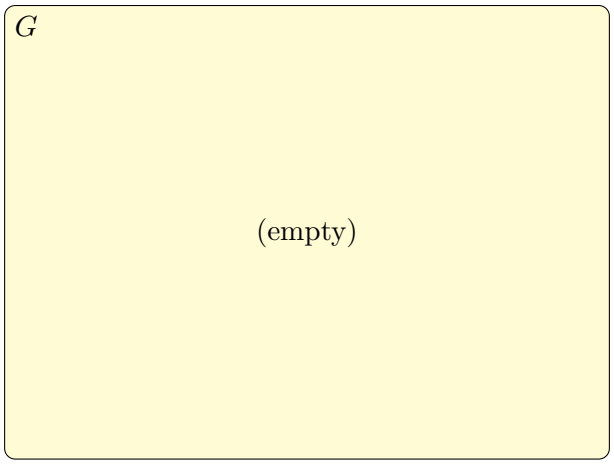
```
def greedy_mst(G):
  K = empty set
  while <K DOES NOT SPAN G>:
    e = <GREEDILY CHOOSE AN EDGE>
    <ADD e TO K>
  return K
```

The handling of the `<K DOES NOT SPAN G>` and `<GREEDILY CHOOSE AN EDGE>` steps each depend on somewhat esoteric facts about graphs. The first fact is that a spanning tree is certain to exist and contain precisely $n-1$ edges, which makes the `<K DOES NOT SPAN G>` step easy to implement.

Theorem 2. *For any connected graph $G = (V, E)$ with $n = |V|$ vertices, there exists a spanning tree $T = (V, K)$ for G. If $n > 0$ then any such T has exactly $|K| = n - 1$ edges.*

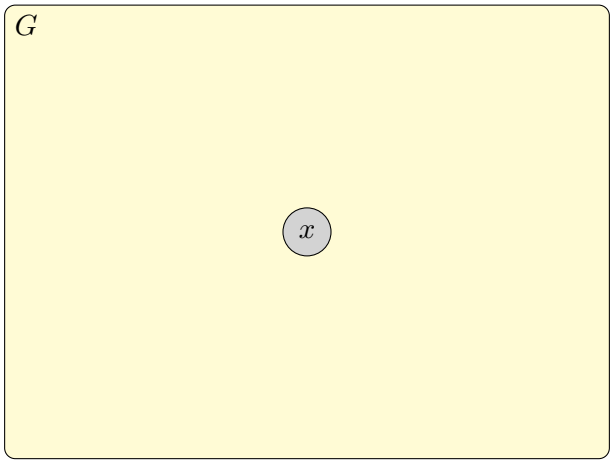
Proof. We argue by induction on n, the number of vertices in the graph.

Base cases. First suppose $n = 0$, i.e. that G contains no vertices or edges.



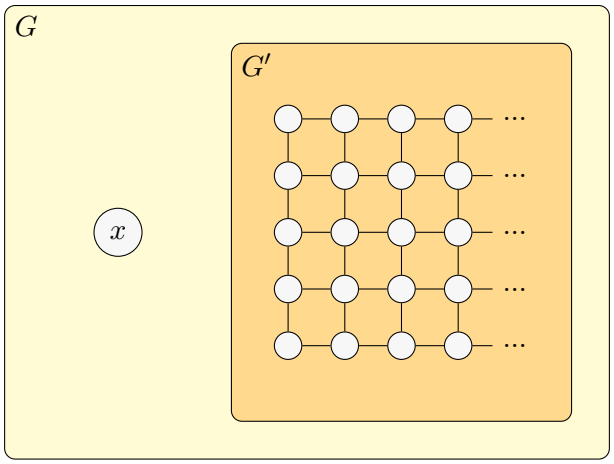
Since G contains no vertices it is (vacuously) connected, and $T = (V, K)$ with $K = \varnothing$ is a valid spanning tree for G. So there exists a spanning tree; in this case $n \not> 0$ so the constraint on the size of K need not hold.

Now suppose $n = 1$, so G contains exactly one vertex.

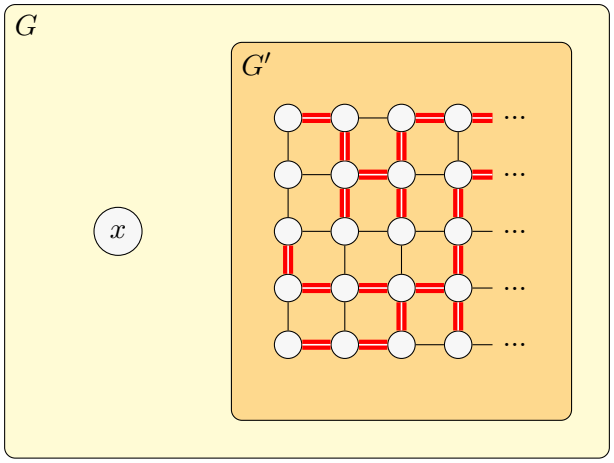


A vertex is considered connected to itself, so again $T = (V, K)$ is a valid spanning tree for G where $K = \varnothing$. So there exists a spanning tree with exactly $|K| = 0 = n - 1$ edges.

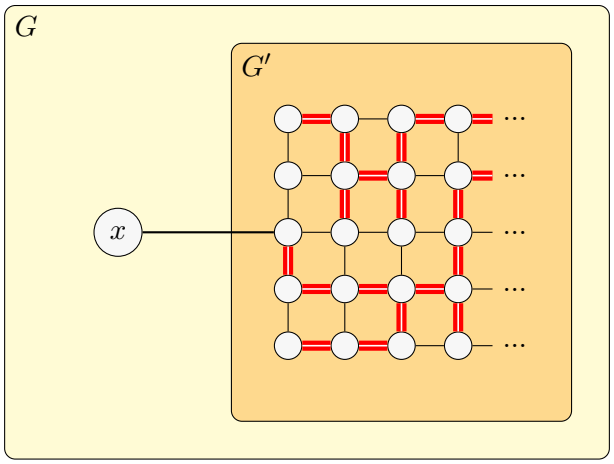
Inductive step. Suppose $n > 1$, and let $x \in V$ be an arbitrary vertex from G, $V' = V - \{x\}$ be the vertices of G aside from x, and $G' = (V', E')$ be the vertex-induced subgraph of G excluding x and any edges incident to x.



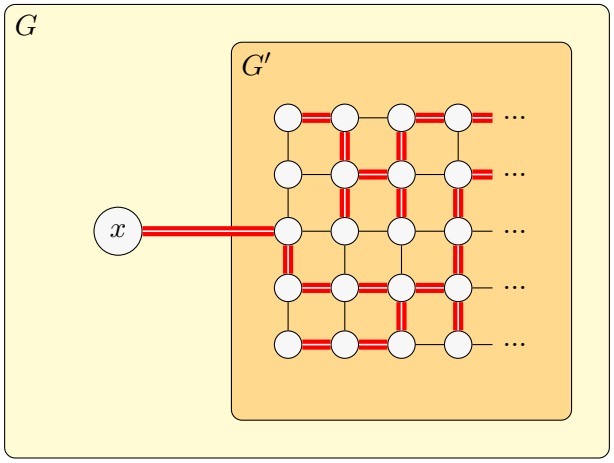
We suppose the inductive hypothesis that there exists a spanning tree $T' = (V', K')$ for G', and that any such T' has exactly $|K'| = |V'| - 1 = n - 2$ edges.



By assumption G is connected, so there exists at least one edge $e = \{x, y\}$, where $e \in E$ and $y \in V'$, that may connect x to G'.



If we let $K = K' \cup \{e\}$, then $T = (V, K)$ is a spanning tree for G.



T' spans G' and $T' \subseteq T$, so every vertex in V' is connected through T. x is connected to some vertex in V' through e, so x is connected to every vertex in G. Thus T is a spanning tree for all of G, and has size $|K| = |K'| + 1 = (n-2) + 1 = n - 1$. □

Spanning trees are right on the razor's edge between an unconnected subgraph and a cyclic subgraph. A subgraph with fewer than $n-1$ edges must leave some vertex unconnected; a subgraph with more than $n-1$ edges must contain a cycle. Only with exactly $n-1$ edges can a subgraph simultaneously span all vertices and remain acyclic.

This insight allows us to fill in the `<K DOES NOT SPAN G>` blank.

```
def greedy_mst(G):
  K = empty set
  while len(K) < (G.vertex_count() - 1):
    e = <GREEDILY CHOOSE AN EDGE>
    <ADD e TO K>
  return K
```

The second esoteric fact about graphs relates to the <GREEDILY CHOOSE AN EDGE> step. It turns out that all greedy algorithms for the minimum spanning tree problem follow a similar structure: each greedy step involves choosing a *minimum-weight edge that connects two not-yet-connected parts of the graph.* We can define this formally in terms of a *set partition.*

Definition 27. *For a universe set U, a* partition *of U is a family of at least two subsets $\mathcal{P} = \{X_0, X_1, \ldots X_{k-1}\}$ such that*

1. *each X_i in $\mathcal{P}$ is a proper subset of U;*
2. *the union of all subsets is equal to U, i.e.*

$$U = \bigcup_{X_i \in \mathcal{P}} X_i;$$

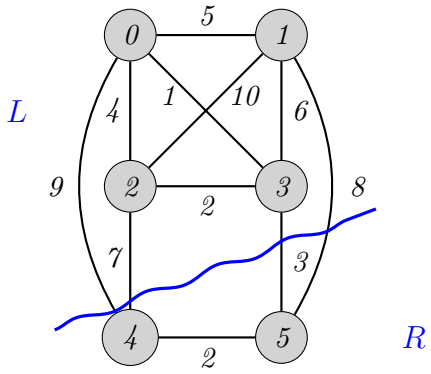
3. *every subset is nonempty, i.e. $X_i \neq \varnothing$ for all $X_i \in \mathcal{P}$; and*
4. *the subsets are pairwise disjoint, i.e. for any $X_i, X_j \in \mathcal{P}$ with $i \neq j$, $X_i \cap X_j = \varnothing$.*

Example 28. *All of the following are partitions of the set $U = \{1, 2, 3\}$:*

- $\{\{1\}, \{2, 3\}\}$
- $\{\{1, 2\}, \{3\}\}$
- $\{\{1\}, \{2\}, \{3\}\}$

Now suppose that our graph G's vertices are partitioned into two subsets $\{L, R\}$.

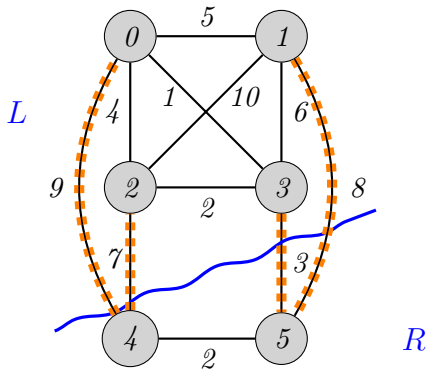
Example 29. *We could partition the vertices $\{0, \ldots, 5\}$ of G_6 into two parts $L = \{0, 1, 2, 3\}$ and $R = \{4, 5\}$.*



We define a *bridge edge* as any edge that crosses between the two parts L and R.

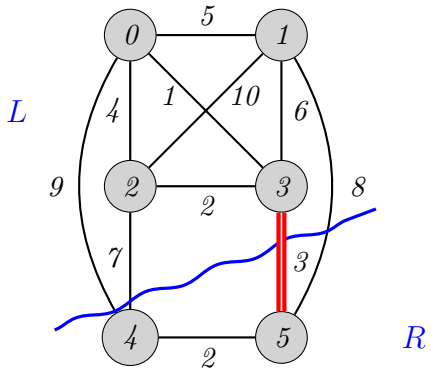
Definition 28. *For any weighted, undirected, connected graph $G = (V, E)$ and partition $\{L, R\}$ of V, the* bridge edges $B \subseteq E$ *are those edges with one end in L and the other in R.*

Example 30. *The bridge edges for the same partition $L = \{0, 1, 2, 3\}, R = \{4, 5\}$ are marked with orange dashes.*



It turns out that a least-weight bridge edge is always a valid choice to connect L and R in a minimum spanning tree [19]. If there are multiple edges tied for least, all of them are valid.

Example 31. *The correct edge to connect L with R is $\{3, 5\}$ whose weight 2 is the least among all bridge edges.*



Theorem 3 (bridge theorem)**.** *Let G and B be defined as in Definition 28, and let $b \in B$ be a bridge edge of least weight; then there exists some minimum spanning tree $T = (V, K)$ such that $b \in K$.*

Proof. By definition G is connected, so by Theorem 2 there exists at least one spanning tree for G. Suppose for the sake of contradiction that, for any arbitrary minimum spanning tree $T = (V, K)$ for G, $b \notin K$. Let $l \in L$ and $r \in R$ be the endpoints of b. T is a spanning tree so there exists a path P between l and r through T. Since l and r are from disjoint parts, P must include some bridge edge $k \in B$, and by our supposition $k \neq b$. Suppose we form a new subgraph $T' = (V, K \cup \{b\})$; this subgraph contains the cycle $P \cup \langle b \rangle$. Now form another subgraph $T'' = (V, K \cup \{b\} - \{k\})$. Removing k from T'' breaks the cycle, so T'' is acyclic and has exactly as many edges as the spanning tree T, and T'' is a spanning tree. Let W and W'' be the weight of T and T'' respectively, and w_b and w_k be the weight of b and k respectively; then W'' is

$$W'' = W + w_b - w_e.$$

By assumption b is a minimum weight bridge edge, so $w_b \leq w_e$ and by substitution

$$\begin{aligned} W'' &\leq W + (w_e) - w_e \\ W'' &\leq W. \end{aligned}$$

If $W'' = W$ then both T and T'' are minimum spanning trees, which contradicts the supposition that no minimum spanning tree containing b exists. Alternatively if $W'' < W$ then we have a contradiction since that implies T is not a minimum spanning tree. Thus there must exist some minimum spanning tree that contains b. □

This result gives us a structure for making greedy choices about which edges to include in K. If our algorithm somehow maintains a partition of G's vertices, then in each loop iteration it need only identify a bridge edge of least weight. By the bridge theorem, any bridge edge of least weight is a valid minimum spanning tree edge.

```
def greedy_mst(G):
  K = empty set
  <ESTABLISH A PARTITION OF G's VERTICES>
  while len(K) < (G.vertex_count() - 1):
    e = <FIND A MINIMUM WEIGHT BRIDGE EDGE>
    <ADD e TO K>
    <UPDATE THE PARTITION>
  return K
```

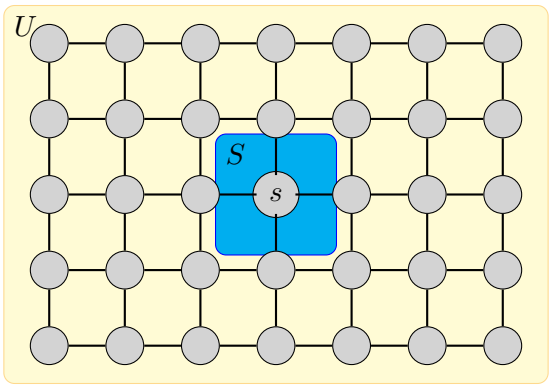
The three most widely known minimum spanning tree algorithms are called the Prim-Jarník algorithm [43], Kruskal's algorithm [36], and Borůvka's algorithm [13]. All these algorithms follow this same pattern. They differ principally in how they establish a vertex partition and identify bridge edges.

6.3.2 The Prim-Jarník algorithm

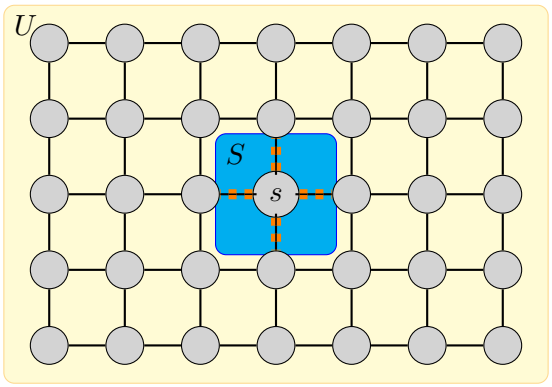
The Prim-Jarník algorithm was first discovered by the mathematician Vojtěch Jarník in 1930 and published in a Czech mathematics journal [30]. Later, in the computer age, it was independently rediscovered by the computer scientist Robert C. Prim and disseminated to the English-speaking computer science community [43]. The algorithm follows our `greedy_mst` pattern, where the vertices are partitioned into a *spanned part* and an *unspanned part.*

Invariant 2. *The Prim-Jarník algorithm maintains a set of tree edges K and vertex partition $\{S, U\}$ such that $T = (S, K)$ is a minimum spanning tree for the subgraph of G induced by S, and no edge in K is incident to any vertex in U.*

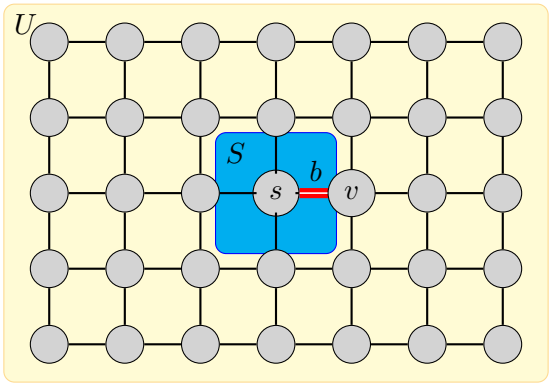
The part S is a subset of contiguous, connected vertices such that the edges in K are a correct minimum spanning tree for S. Meanwhile the other part U is entirely unspanned. At the very beginning of the algorithm, we can pick an arbitrary *start vertex* $s \in V$, and declare that $S = \{s\}, U = V - \{s\}$, and $K = \varnothing$. As discussed in the proof of Theorem 2, a subgraph of one vertex is trivially spanned by an empty list of edges.



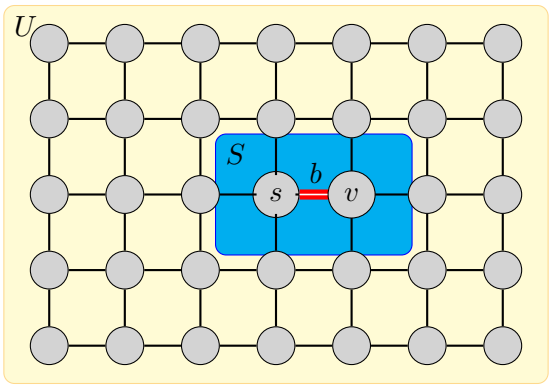
For the first greedy choice, all edges incident to the start vertex s are bridge edges.



The algorithm identifies which of these bridge edges b is lightest and adds it to K. Let $b = \{s, v\}$, so that v is the end of the chosen edge that is currently in the unspanned part U.

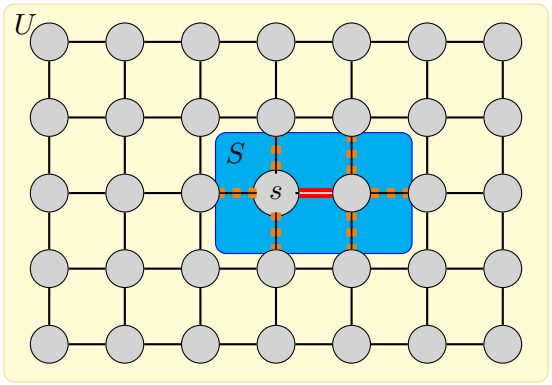


Vertex v is now spanned, so to maintain the correctness invariant we must insert v into S and delete v from U.

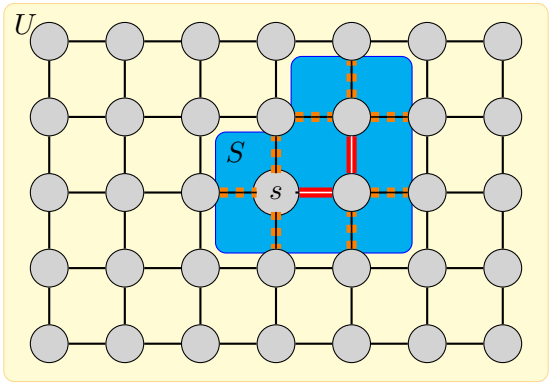


At this point we have completed one greedy choice. We went from the initial state of spanning $|S| = 1$ vertices using $|K| = 0$ edges, to spanning $|S| = 2$ vertices using $|K| = 1$ edges.

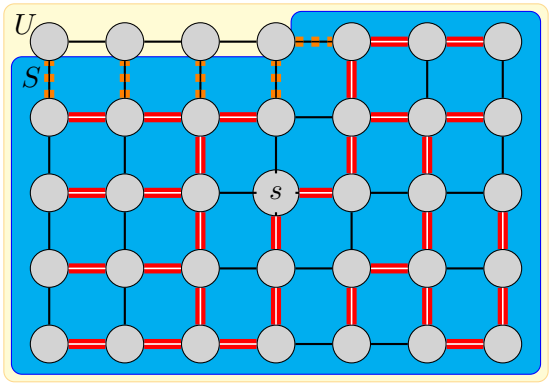
In the second greedy choice, the set of bridge edges includes the same edges as in the first choice (except for the edge b we added to the tree), as well as a few new edges.



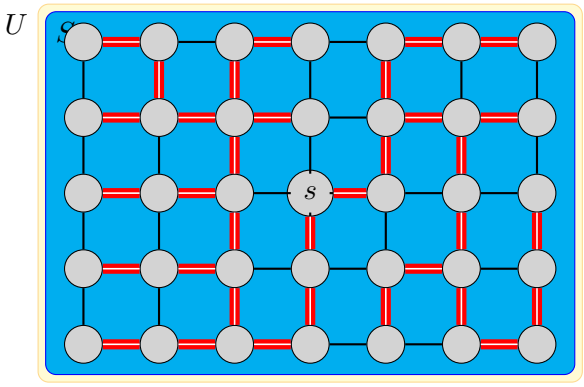
Again the algorithm determines which bridge edge is lightest, adds it to K, expands S by one vertex, and shrinks U by one vertex.



After a few more steps, most of the graph is spanned.



The algorithm terminates after all n vertices are in the spanned set S, at which point K contains exactly $n - 1$ edges. Observe that all vertices are spanned, and that adding one more edge to K would create a cycle.



Our first draft pseudocode for the Prim-Jarník algorithm is below.

```
def prim_jarnik(G):
    K = Vector()
    S = Vector()
    U = Vector()
    # initially one vertex is spanned and the rest are unspanned
    first = True
    for v in G.vertices():
        if first:
            S.add_back(v)
            first = False
        else:
            U.add_back(v)

    while len(K) < ( G.vertex_count() - 1 ):
        # sequential search for the minimum-weight bridge edge
        b = None
        for e in G.edges():
            # e is a bridge edge when one end is in S and the other is not
            if ((e.s in S) != (e.t in S)
                and
                (b is None or e.label < b.label)):
                    b = e

        K.add_back(b)

        # determine which end just got spanned
        if b.s in U:
            U.remove(b.s)
            S.add_back(b.s)
        else:
            U.remove(b.t)
            S.add_back(b.t)

    return K
```

Lemma 24. *The time complexity of this draft of the Prim-Jarník algorithm is* $O(n^2m)$.

Proof. Creating the `K`, `S`, and `U` data structures takes $O(1)$ time. The outer `for` loop that iterates through the vertices takes $O(n)$ time.

The main `while` loop iterates exactly $n-1$ times. The inner `for` loop implements a sequential search through the m edges of G. The time spent in each iteration is dominated by the `(v in S)` and `(w in S)` expressions which take $O(n)$ time each; the other steps in the body of the `for`

loop take $O(1)$ time. Updating K, S, and U takes $O(n)$ for the linear `U.remove` operation.

The final `return` statement takes 1 step.

The total time complexity for the algorithm is

$$O\Big(1 + n + (n-1)[1 + m(1+n) + n] + 1) = O(nmn) = O(n^2 m).$$

□

This time efficiency polynomial, so this algorithm is probably fast enough to be used in practice. However it could be a lot faster. The bottleneck is the `(v in S)` and `(w in S)` expressions, which determine whether the vertices v and w are each spanned or unspanned. Currently we are using an unordered vector of vertices to track S, so this step takes $O(n)$ time. However, graph data structures can index vertices with unique integers $\{0, 1, \ldots, n-1\}$. Rather than representing S by a vector of vertex objects, instead we can represent it with a lookup list of Booleans that we will call `spanned`.

Invariant 3. *A vertex with index* `i` *is in* S *part when* `spanned[i]` *is* `True`*, and is in* U *otherwise.*

```
def prim_jarnik(G):
    K = Vector()
    spanned = Vector(G.vertex_count(), False)
    spanned[0] = True

    while len(K) < ( G.vertex_count() - 1 ):
        # sequential search for the minimum-weight bridge edge
        b = None
        for e in G.edges():
            # e is a bridge edge when one end is in S and the other is not
            if (spanned[b.s.index] != spanned[b.t.index])
                and
                (b is None or e.label < b.label)):
                    b = e

        K.add_back(b)

        # mark both ends spanned (one already was)
        spanned[b.s.index] = True
        spanned[b.t.index] = True

    return K
```

Lemma 25. *The time complexity of this draft of the Prim-Jarník algorithm is* $O(nm)$.

Proof. The algorithm spends $O(n)$ time initializing K and `spanned`, then repeats the main `while` loop $n-1$ times. The inner `for` loop repeats m times and spends $O(1)$ time in each iteration. Updating K and `spanned` takes $O(1)$ time, as does the final `return` statement. So the total time complexity is

$$O(n + (n-1)(m+1) + 1) = O(nm).$$

□

This is a significant improvement over $O(n^2m)$, but we can do even better. This version of the Prim-Jarník algorithm searches through all edges E in every `while` loop iteration. Essentially, in searching for bridge edges, it starts from scratch every time. However the set B of bridge edges changes gradually, so we ought to be able to maintain that set in a data structure, and update it in less than $O(m)$ time per loop iteration. Later in Section 10.4 we will see how to use the *reduction to data structure operations* design pattern to maintain the set of bridge edges more efficiently.

Claim 1. *There is a version of the Prim-Jarník algorithm with time complexity $O(m + n \log n)$.*

6.4 Nonnegative Single-Source Shortest Paths and Dijkstra's Algorithm

Shortest path problems involve computing minimal-weight paths between vertices in weighted graphs. There are many variations on the problem, depending on exactly how the input graph and output path information is defined. *Single source* shortest path problems involve computing paths originating from a designated *source* (or *start*) vertex, while *all-pairs* shortest path problems involve computing paths between all pairs of distinct vertices. There are also varying definitions of the cost of edges in the path. Each edge may have a unit cost, or each may have a numeric weight; numeric weights may be unconstrained, or be required to be non-negative.

Let us define some formal notation related to shortest path problems.

Definition 29. *Let*

1. *$G = (V, E)$ be a weighted undirected graph,*
2. *V be the set of graph vertices,*
3. *E be the set of graph edges, where each element of E is a set of exactly two vertices,*
4. *a* path *be any non-empty sequence $\langle p_0, \ldots, p_{k-1} \rangle$ of vertices, such that each $p_i \in V$ and every pair of adjacent vertices p_i, p_{i+1} is connected in G, so $\{p_i, p_{i+1}\} \in E$;*
5. *w_e be the weight of any edge $e \in E$,*
6. *$W(X)$ be the total weight of a path X,*
7. *$L_{s,v} = \langle s, \ldots, v \rangle$ be a shortest path from start vertex $s \in V$ to end vertex $v \in V$,*

8. $d_{s,v}$ *be the shortest distance between* s *and* v*, i.e. the total weight of all edges visited by some* $L_{s,v}$*, and*

9. $p_{s,v}$ *be the* penultimate *(next to last) vertex on a shortest path from* s *to* v*, i.e.* $L_{s,v} = \langle \ldots, p_{s,v}, v \rangle$.

Note that, for a given $G, s,$ and v, $L_{s,v}$ and $d_{s,v}$ are only defined when s and v are connected, and $p_{s,v}$ only exists when s and v are connected by a shortest path that visits at least 2 vertices.

In this Section we consider the *single source shortest paths* problem, where edge weights are non-negative numbers. This is perhaps the most practical variant of shortest path problem, as it corresponds to planning fast routes in networks where each hop has a non-negative cost. Driving directions are a familiar instance of this problem, since a motorist has a known start location, and driving along a stretch of road has a non-negative cost, regardless of whether it is measured in units of time, fuel, or monetary cost.

non-negative single source shortest paths

input: a non-empty graph $G = (V, E)$ where each edge is labeled with a non-negative number; and start vertex $s \in V$

output: two vectors `distance` and `penultimate`, each of length exactly n, such that, for any $v \in V$,

$$\texttt{distance[v.index]} = \begin{cases} d_{s,v} & \text{if } s \text{ and } v \text{ are connected, or} \\ \texttt{None} & \text{otherwise} \end{cases}$$

and

$$\texttt{penultimate[v.index]} = \begin{cases} p_{s,v} & \text{if } s, v \text{ are connected and } s \neq v \\ \texttt{None} & \text{if } s = v, \text{ or} \\ \texttt{None} & \text{if } s, v \text{ are not connected .} \end{cases}$$

The definition of the output for this problem is rather complex. An algorithm for this problem produces two vectors `distance` and `penultimate` that together encode all of the shortest paths originating from vertex s. In spirit this problem involves computing a shortest path from a start vertex s to one specific end vertex t, but efficient algorithms for this problem can efficiently compute *all* shortest paths starting at s and ending at every other vertex including but not limited to t. Including the information for all these "bonus" paths does not degrade the time complexity of our algorithm, thus we include it since it is potentially useful.

When a vertex `v` is not connected to s, no $L_{s,v}$ exists and $d_{s,v}$ and $p_{s,v}$ are both undefined, represented with `distance[v.index]=None` and `penultimate[v.index]=None`. When `v` happens to be the source vertex s, the shortest path from s to itself is a single-vertex path $\langle s \rangle$, `distance[v.index]=0`, and `penultimate[v.index]` is `None`. In the more interesting case a

multi-vertex path $L_{s,v}$ exists, `distance[v.index]` stores the total weight (cost) of a shortest path from s to `v`, and `penultimate[v.index]` is the vertex object for the *penultimate* (second-to-last) vertex on a shortest path from s to `v`. In other words, if $L_{s,v} = \langle s, \ldots, p, v \rangle$ is a shortest path from s to v, the `penultimate[v.index]` stores vertex object p. The purpose of this somewhat curious data structure is to encode all of these shortest paths in only $O(n)$ total space. The alternative might be to store, for each vertex v, the full path $L_{s,v}$ as a vector of length $|L_{s,v}|$; but since a path may involve as many as n vertices, and there are n vertices in the graph, this approach could take $O(n^2)$ space. Instead the `penultimate` encoding takes a total of $O(n)$ space, and therefore can be initialized in $O(n)$ time.

Dijkstra's algorithm, discovered by Dutch computer scientist Edsger Dijkstra, is an efficient greedy algorithm for the non-negative single source shortest paths problem [17]. Like the Prim-Jarník algorithm, it maintains a partition $V = S \cup U$ of a graph's vertices into two disjoint and nonempty parts. One part S may be thought of as a *seen part,* such that the algorithm has computed correct shortest paths from s to any vertex in S. Meanwhile U is the *unseen part* of the graph, for which the algorithm has yet to compute a correct shortest path. Dijkstra's algorithm repeatedly picks an unseen vertex u according to a greedy heuristic, computes a valid shortest path from s to u, and updates its data structures accordingly. Each next vertex u is connected directly to a seen vertex by a bridge edge between S and U. This is analogous to how the Prim-Jarník algorithm grows the spanned part one vertex at a time.

```
def dijkstra(G, s):
    distance = Vector(G.vertex_count(), None)
    penultimate = Vector(G.vertex_count(), None)
    seen = Vector(G.vertex_count(), False)

    distance[s.index] = 0
    seen[s.index] = True

    done = False
    while not done:
        <FIND SOME EDGE b={v, u} SUCH THAT v IS SEEN AND u IS UNSEEN, AND
         [s, ..., v, u]
         IS A CORRECT SHORTEST PATH TO u>

        if <NO SUCH b EXISTS>:
            done = True
        else:
            distance[u.index] = distance[v.index] + b.label
            penultimate[u.index] = v
            seen[u.index] = True

    return distance, penultimate
```

The algorithm maintains the following correctness invariant.

Invariant 4. *If* `seen[u.index]` *is* `True` *then* `distance[u.index]` *stores the correct value of* $d_{s,u}$ *and* `penultimate[u.index]` *stores the correct value of* $p_{s,u}$.

Once again, we need some mathematical machinery to establish how to `<FIND SOME EDGE b...>` in a way that maintains our invariant.

Lemma 26. *Let* $s_0, s_1, \ldots, s_{n-1}$ *be the vertices of* V *in nondecreasing order by distance from* s*, so that* $s_0 \equiv s$ *and* $d_{s,s_0} \leq d_{s,s_1} \leq d_{s,s_2} \leq \ldots \leq d_{s,s_{n-1}}$*,* $S_k = \{s_0, \ldots, s_{k-1}\}$ *be the first* k *seen vertices,* $U = V - S_k$ *be the remaining unseen vertices,* $B \subseteq E$ *be the bridge edges with one seen and one unseen end, and* $b = \{s_i, u\}$ *be a bridge edge with* $s_i \in S$ *and* $u \in U$ *such that*

$$d_{s,s_i} + w_b \leq d_{s,s_{i'}} + w_{b'}$$

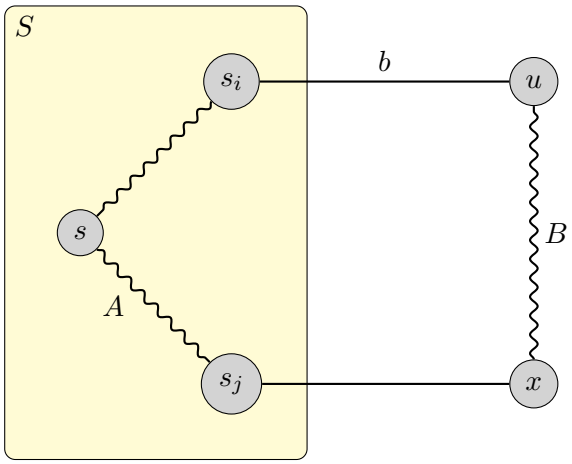
for any $s_{i'} \in S_k$ *and* $b' \in B$*; then*

$$\begin{aligned} L_{s,u} &= L_{s,s_i} + \langle u \rangle, \\ d_{s,u} &= d_{s,s_i} + w_b, \text{ and} \\ s_k &= u. \end{aligned}$$

Proof. By construction, the path $P = L_{s,s_i} + \langle u \rangle$ is a shortest path from s to u with the constraint that all vertices along the path, with the exception of u, are in S. We need that this is a shortest path overall, including paths that might stray outside of S into U. Suppose for the sake of contradiction that there exists a path $P' = \langle s, \ldots, s_j, x, \ldots, u \rangle$ from s to u that visits some $x \in U$ before reaching u, and that P' is a shorter path than P, or formally

$$W(P') < W(P).$$

Let A be the initial part of P' from s through x, and B be the remaining unseen part of P' from x to u, so $W(P') = W(A) + W(B)$.



By the definition of b, $W(A) \geq W(P)$; edge weights are non-negative so $W(B) \geq 0$, and we have

$$\begin{aligned} W(P') &= W(A) + W(B) \\ &\geq \big(W(P)\big) + \big(0\big) \\ &\geq W(P) \end{aligned}$$

which is a contradiction. Therefore no P' exists and P is a shortest path from s to u. □

According to this Lemma, we may greedily choose the bridge edge $b = \{s_i, u\}$ that minimizes the value of $d_{s,s_i} + w(b)$.

```
def dijkstra(G, s):
    distance = Vector(G.vertex_count(), None)
    penultimate = Vector(G.vertex_count(), None)
    seen = Vector(G.vertex_count(), False)

    distance[s.index] = 0
    seen[s.index] = True

    done = False
    while not done:
        least_weight = si = u = None
        for e in G.edges():
            x = e.s
            y = e.t
            # is e a bridge edge?
            if seen[x.index] != seen[y.index]:
                if not seen[x.index]:  # ensure x is the seen end
                    swap(x, y)
                w = distance[x.index] + e.label
                if least_weight is None or w < least_weight:
                    least_weight = w
                    si = x
                    u = y

        if least_weight is None:
            done = True
        else:
            distance[u.index] = least_weight
            penultimate[u.index] = si
            seen[u.index] = True

    return distance, penultimate
```

This version of Dijkstra's algorithm is straightforward to analyze.

Lemma 27. *The preceding version of Dijkstra's algorithm runs in $O(mn)$ time.*

Proof. The initializations before the main loop take $O(n)$ time. Before the outer `while` loop starts, exactly 1 vertex is seen and exactly $n-1$ are unseen, and each iteration of the outer loop changes exactly one unseen vertex to become seen, so the loop iterates at most $n-1$ times. Each iteration involves the inner `for` loop, which repeats m times and spends $O(1)$ time in each iteration. The `if...else` statement at the end of the loop body, and loop overhead, together take $O(1)$ time. So the total time complexity of the algorithm is at most

$$O(n + (n-1)(m+1)) = O(mn).$$

□

When G is *sparse* and $m \in O(n)$ this efficiency class simplifies to $O(mn) = O(nn) = O(n^2)$. Alternatively when G is *dense* and $m \in O(n^2)$, the class simplifies to $O(mn) = O(n^2 n) = O(n^3)$, which is a rather slow polynomial efficiency class. We can improve the algorithm's performance in the dense-graph case by changing how the inner `for` loop operates. Our current draft loops through all edges in the inner loop, which is the bottleneck in the $O(mn)$ efficiency class. With some care, it is possible to instead loop through all *vertices* in the inner loop, so that the bottleneck is more like $O(n^2)$.

The purpose of the inner `for` loop is to identify an unseen vertex u such that Lemma 26 guarantees that $L_{s,p_{s,u}} \cup \{u\}$ is a shortest path to u. In order to apply the Lemma we need that u is unseen, adjacent to a seen vertex, and that the quantity $L_{s,p_{s,u}} + w_{\{p_{s,u},u\}}$ is minimal. This is a search through a set of at most n vertices, so in principle it ought to be achievable $O(n)$, rather than $O(m)$, time. We still need methods for telling whether an unseen vertex is adjacent to a seen one, and for calculating the path cost to these vertices.

To do this, we reorder our algorithm to follow bridge edges to each unseen vertex u *before* we greedily choose u, instead of *while* choosing u. This process is called *relaxing* bridge edges, and we will codify it in a new sub-algorithm `relax`. We introduce a correctness invariant for the relaxation process, which is maintained in concert with the existing invariant.

Invariant 5. *If `seen[u]` is `True` then, for any bridge edge $e = \{u, v\}$ between u and an unseen v, `distance[v]` $\neq$ `None` and `distance[v]` $\leq d_{s,u} + w_e$; and if `distance[v]` $= d_{s,u} + w_e$ then `penultimate[v]` $= u$.*

This invariant allows us to simplify the inner `for` loop greatly. In order to maintain the invariant, we need to call `relax` at the two junctures where a vertex is marked seen: when accounting for the fact that s is seen before the outer loop starts, and after each `u` has been greedily chosen.

```
def dijkstra(G, s):
    distance = Vector(G.vertex_count(), None)
    penultimate = Vector(G.vertex_count(), None)
    seen = Vector(G.vertex_count(), False)

    distance[s.index] = 0
    relax(G, distance, penultimate, seen, s)

    done = False
    while not done:
        u = None
        for y in G.vertices():
            if (not seen[y.index]) and
               (distance[y.index] is not None) and
               (u is None or distance[y.index] < distance[u.index]):
                    u = y

        if u is None:
            done = True
        else:
            relax(G, distance, penultimate, seen, u)

    return distance, penultimate

def relax(G, distance, penultimate, seen, u):
    for e in u.incident():
        # either y==u or z==u
        y = e.s
        z = e.t
        if z == u: # ensure z != u
            swap(z, y)
        if not seen[z.index]:
            w = distance[u.index] + e.label
            if distance[z.index] is None or w < distance[z.index]:
                distance[z.index] = w
                penultimate[z.index] = u
    seen[u.index] = True
```

The `relax` algorithm uses the `G`, `distance`, `penultimate`, and `seen` data structures, and relaxes the edges incident to `u`, where Lemma 26 guarantees that `distance[u]` and `penultimate[u]` store correct values. The algorithm relaxes every edge `e` incident to `u`. The `for` loop iterates through each `z` incident to `u`, skipping already-seen vertices. If no path to `z` is known, or if the path to `u` and then through `e` is shorter than the current shortest-known path to `z`, then we

have found a new shorter (but not necessarily shortest) path to `z`. After the relaxation process is complete, `u` is marked seen, which is consistent with both correctness invariants.

At first this version may not appear to be any more efficient than the previous version. The outer loop repeats at most $n-1$ times; each iteration involves the inner `for` loop which clearly takes $O(n)$ time, a call to `relax`, and $O(1)$ other steps. The `for` loop in `relax` iterates through a list of edges and spends $O(1)$ time in each iteration, so each call to `relax` takes at most $O(m)$ time. So according to this cursory analysis, the algorithm takes at most

$$O(n + (n-1)(m+1)) = O(mn)$$

time, which is no better than the previous version.

This analysis is an example of an *overcounting.* While it is true that any individual call to `relax` may examine at most m edges, it is not possible for *every* call to involve all m edges, which is what the preceding analysis assumed.

Lemma 28. *During the execution of `dijkstra`, each edge `e` in E is examined by the body of the `for` loop in `relax` at most twice.*

Proof. `relax` is always called on a vertex `u` that is unseen before the call, and seen after the call finished. Once a vertex is seen it never reverts to being unseen, so `relaxed` is called at most once on each `u`. An edge may participate in the `for` loop at most once for each of its two ends, so each edge is examined at most twice by that loop. □

This insight allows for a tighter step count.

Lemma 29. *The time complexity of this version of Dijkstra's algorithm is $O(m + n^2)$.*

Proof. We analyze two parts of the algorithm's running time separately:

1. the time spent in the `for` loop inside `relax`, and
2. the time spent on all other operations.

For (1), Lemma 28 states that the total number of iterations of the `for` loop inside `relax` is at most $2m$. Each such loop iteration takes $O(1)$ time, so the total time spent on part (1) is $O(2m) = O(m)$.

For (2), `dijkstra` spends $O(n)$ time initializing data structures. The first call to `relax` takes $O(1)$ time, not counting the `for` loop in that algorithm. As usual the outer `while` loop iterates at most $n-1$ times. The inner `for` loop takes $O(n)$ time, and the remainder of each `while` loop iteration takes $O(1)$ time. Finally the `return` statement takes $O(1)$ time, so the total time for this concern is at most

$$O(n + 1 + (n-1)(n+1) + 1) = O(n^2).$$

The total time spent in both parts is at most

$$O(m + n^2).$$

□

In the case of a sparse graph, $O(mn) = O(m + n^2) = O(n^2)$, so both versions are equally efficient in the sparse case. However in the case of a dense graph, $O(m + n^2) = O(n^2 + n^2) = O(n^2)$, which is a significant improvement over the $O(n^3)$ efficiency of the previous version for the dense case.

Coincidentally, Dijkstra's algorithm can be sped up by using a priority search queue, just as the Prim-Jarník algorithm can be. The speedup comes from replacing the `for y in range(n)` sequential search, which takes $O(n)$ time, with a data structure operation that takes only $O(\log n)$ time.

Claim 2. *There is a version of Dijkstra's algorithm with time complexity* $O(m + n \log n)$.

Exercises

6-1. Analyze the greedy change-making algorithm on page 130.

(a) Prove that the algorithm runs in $O(k)$ time.

(b) Design a faster greedy algorithm, and prove that your algorithm runs in $O(|L|)$ time. *Hint:* try to eliminate the main `while` loop.

6-2. Each of the following proverbs may be thought of as a greedy heuristic for handling a real-life situation. For each, describe the proverb's advice in your own words, and then give an example of a scenario for which the outcome of following that advice would be sub-optimal.

(a) "The early bird catches the worm."

(b) "Practice makes perfect."

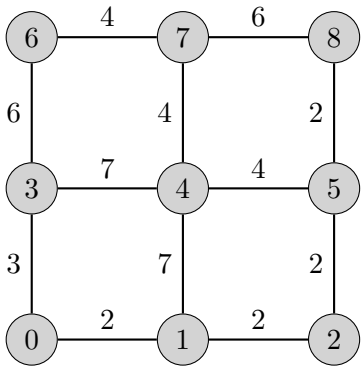
(c) "A bird in the hand is worth two in the bush."

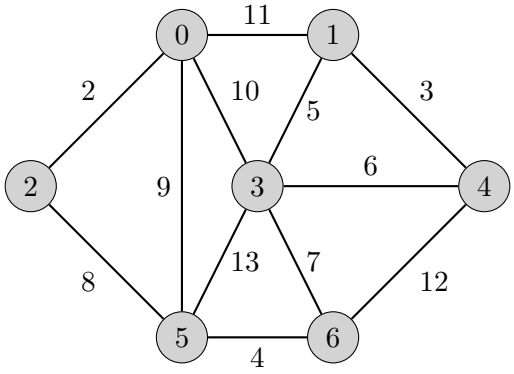
(d) "Let every man divide his money into three parts, and invest a third in land, a third in business and a third let him keep by him in reserve."

6-3. Execute the Prim-Jarník algorithm by hand on the following graphs. Show your work.

(a) The graph from Example 26.

(b) The following:





(c) The following:

6-4. Give an example of a graph with $n > 3$ vertices that has more than one minimum spanning tree.

6-5. What would happen if the `spanned[0] = True` statement were deleted from line 5 of the Prim-Jarník algorithm?

6-6. Execute Dijkstra's algorithm on each of the graphs in question R-6.3, starting from $s = 3$.

6-7. Prove that, for any connected graph G with distinct edge weights, G has exactly one minimum spanning tree.

6-8. Show an example of a graph with negative edge weights for which Dijkstra's algorithm produces incorrect output.

6-9. For each of the following problems: design a greedy algorithm that solves the problem; describe your algorithm with clear pseudocode; and prove the time efficiency class of your algorithm.

(a)

offline ski rental
input: a daily ski rental price $r > 0$, purchase price $p > 0$, and number of days $d > 0$ **output:** `True` if it is cheaper to rent skis for d days at r dollars per day, or `False` if it is cheaper to buy skis for p dollars

(b)

wiggle sort
input: a list L of n distinct comparable elements **output:** a list S containing the elements of L in wiggle order

A sequence $S = \langle s_0, s_1, \ldots \rangle$ is in *wiggle order* when the elements alternate between a less-than and greater-than relationship, so

$$s_0 < s_1 > s_2 < s_3 > s_4 < \ldots.$$

(c)

2-coloring verification
input: an undirected graph $G = (V, E)$, where each vertex v is labeled with a color `v.label` $\in \{1, 2\}$ **output:** `True` if no two adjacent vertices are labeled with the same color, or `False` otherwise

(d) [40]

sum swap problem
input: two non-empty lists of numbers L and R, each of length at most n **output:** (l, r) where $l \in L, r \in R$, and swapping l and r would makes L and R have the same sum

(e) Recall that a graph $G = (V, E)$ is defined by a set of vertices V and set of edges E. An edge $e \in E$ is *incident* to vertex $v \in V$ when v is one of e's endpoint vertices.

isolated vertex
input: a graph G with n vertices and m edges **output:** `True` if G contains a vertex with no incident edges, or `False` otherwise

The two most common data structures for graphs are the *adjacency lists* and *adjacency matrix*. You can choose whether the input to your algorithm is in adjacency lists or adjacency matrix format.

Hint: A decent algorithm for this problem runs in $O(n^2)$ time, and an optimal one takes $O(n)$ time.

(f)

Eulerian cycle decision
input: an adjacency lists data structure containing a connected undirected graph $G = (V, E)$ **output:** `True` iff G contains an Eulerian cycle

An *Eulerian cycle* is a cycle that visits every *edge* in G exactly once.

Hints: Review the vertex data structure operations in Table 4.9. An optimal algorithm for this problem has $O(n)$ time complexity.

(g)

2-coloring
input: an undirected graph $G = (V, E)$ **output:** a sequence `coloring` $= \langle (v, c) | v \in V$ and $c \in \{1, 2\}$ such that each $v \in V$ appears exactly once in `colors`, and no adjacent vertices are assigned the same color; or `None` if no such `coloring` exists

(h) [40] In a list of n elements, a *majority element* appears more than $\frac{n}{2}$ times.

majority element
input: a list L of n comparable elements **output:** a majority element m of L, or `None` if L has no majority element

An optimal algorithm for this problem has $O(n)$ time complexity.

Chapter 7

Exhaustive Search and Optimization

7.1 The Big Idea

The *exhaustive search* algorithm pattern is a generalization of sequential search. An exhaustive search algorithm iterates through objects in search for a particular kind of object, just as sequential search does. The difference is that, in exhaustive search, the list of objects to iterate through is not given as a part of a problem instance. Instead, the exhaustive search algorithm must synthesize those objects itself. Typically the list of these *candidate solutions* is substantially larger than the size of input, which is why exhaustive search algorithms are typically substantially slower than sequential search's $O(n)$ time complexity.

Like the greedy pattern, the exhaustive search pattern relates to a common sense approach for solving real-world problems. It corresponds to the problem of "finding a needle in a haystack." For example, suppose you wanted to break into a combination lock with four 10-digit wheels. One way of doing this is to work your way through all $10^4 = 10,000$ possible combinations until one of them works. Here the "input" to the problem is the 4-digit lock; you generate the 10,000 combinations yourself and simply try each in turn. This approach is clearly correct; if there is any combination that opens the lock, you will certainly come across it eventually. However it is, of course, very slow.

When writing an exhaustive search algorithm, it is natural to factor the algorithm into two parts: one part that enumerates all the objects to test, and another part that tests them. Using that approach means that the computer will need to store the list of all possible solutions in memory. These lists can be extremely large, so the space complexity of exhaustive search algorithms can easily become prohibitive. For that reason we will always analyze exhaustive search algorithms for space.

7.1.1 Terminology

The objects that an exhaustive search algorithm enumerates and then searches through are called *candidate solutions* (or simply *candidates*). An exhaustive search algorithm enumerates many candidates, then tests them until it finds an *acceptable* one which may serve as a correct output. We call the process of determining whether a candidate solution is acceptable or not *verification,* and when we write a sub-algorithm for that process, it is called a *verifier algorithm*.

Definition 30. *A* candidate solution *or simply* candidate *is an object of the same data type as a problem output, which may or may not be a correct output.*

Definition 31. *A candidate solution which comprises a correct output for a given problem instance is* acceptable*, and a candidate solution which is not a correct output is* unacceptable.

Definition 32. *A* verifier algorithm *or simply* verifier *is an algorithm that takes a problem instance and candidate solution as input, and returns* `True` *when the candidate is acceptable and* `False` *when the candidate is unacceptable.*

7.2 Proper Exhaustive Search

With that terminology in place, we can state the pseudocode pattern for exhaustive search algorithms.

```
def exhaustive_search(<INPUTS>):
  for candidate in <GENERATE CANDIDATES>(<INPUTS>):
    if <VERIFIER>(<INPUTS>, candidate):
      return candidate
  return None
```

This pseudocode leaves `<GENERATE CANDIDATES>` and `<VERIFIER>` as blanks; those sub-algorithms need to be defined in a problem-specific way. As we can see, this pseudocode bears a strong resemblance to the sequential search algorithm developed in Section 5.3. The only significant difference, aside from terminology changes, is that the exhaustive search pattern includes the `<GENERATE CANDIDATES>` step. Intuitively, exhaustive search is a kind of sequential search where the algorithm generates the objects to search through.

The pattern is certainly correct, provided that `<GENERATE CANDIDATES>` and `<VERIFIER>` work as advertised. If the main loop iterates through every `candidate` that could possibly exist, and checks each properly, then if any solution exists the algorithm will definitely find it. Also, note that the order of the elements in the candidate list does not affect the correctness of the algorithm. If an acceptable candidate exists it will be found regardless of whether that happens early in the loop, or late. However, if given the choice, we would prefer for acceptable candidates to be front-loaded near the beginning of the list so that the `for` loop can terminate as soon as possible.

We call this pattern *proper* exhaustive search because of its resemblance to sequential search. In proper exhaustive search we return the first acceptable candidate that we come across. This framework can be adapted to find an *optimal* candidate; the pattern is similar but not identical, and called *exhaustive optimization.*

7.3 Exhaustive Optimization

Instead of stopping with the first acceptable candidate that we encounter, we can instead search for the "best" candidate, where "best" is defined in a problem-specific way. For example, a proper exhaustive search algorithm for the combination lock problem returns the first combination it comes across that works. However, if the lock could accept multiple combinations, we might want to search for the "best" working combination; the "best" combination might be the one that requires the dials to be moved the least amount.

We affect this change by modifying the proper exhaustive search pattern to keep track of the `best` acceptable candidate that we have seen so far, and update `best` whenever a superior candidate is found. This is similar to how the greedy algorithm for the minimum problem worked, and should be familiar. We also add in a blank for prioritizing superior candidates in a problem-specific way.

```
def exhaustive_optimize(<INPUTS>):
  best = None
  for candidate in <GENERATE CANDIDATES>(<INPUTS>):
    if <VERIFIER>(<INPUTS>, candidate):
      if best is None or <COMPARE>(best, candidate):
        best = candidate
  return best
```

The new `<COMPARE>` sub-algorithm is responsible for determining which of `best` and `candidate` is a preferable solution. It must return `True` when `candidate` is superior, and `False` when `best` is superior (it may return either value if the two candidates are equally good).

Unfortunately the term "exhaustive search" is used alternatively to mean proper exhaustive search, and exhaustive optimization, in the literature. A reader must discern them from contextual clues.

7.4 Designing and Analyzing Exhaustive Algorithms

We recommend going about the following phases while designing an exhaustive search algorithm:

1. *Determine whether to use the proper exhaustive search or exhaustive optimization pattern.* When the problem definition specifies that a solution must be the "best" in some way, usually

maximizing or minimizing something, the optimization pattern is appropriate. On the other hand, when the problem definition says that the output is an object matching criteria that does not involve optimizing, proper exhaustive search is appropriate.

2. *Fill in the* `<INPUTS>` *blanks with the input for the problem at hand.*

3. *Clarify what each candidate is.* The proper exhaustive search pattern returns the first correct candidate object; the exhaustive optimization pattern stores a candidate in the `best` variable and eventually returns `best`. Therefore the data type of each candidate must match the data type of the problem's output. In this phase, write down a clear definition for what one candidate is, in terms of both a concrete data type (e.g. "sequence of vertices") and a concept (e.g. "path through the graph").

4. *Design the verification sub-algorithm.* Study the problem's output definition, and use it as a hint toward how to discern whether a particular candidate object is a correct solution, or not. Write pseudocode for the verifier as a function that takes a problem instance and candidate as inputs, and returns a `True` when the candidate is a correct solution.

5. *Design the comparison sub-algorithm (if this is an optimization).* This phase only applies to exhaustive optimization algorithms, and not to proper exhaustive search algorithms. Just like the verification sub-algorithm, the comparison sub-algorithm should be specified in pseudocode as its own function. The comparison sub-algorithm takes two correct candidates `best` and `candidate`, each of which passed the verification check, and returns `True` when `candidate` is the superior solution or `False` when `best` is superior.

6. *Analyze the algorithm.* Prove the time complexity of your complete algorithms.

Exhaustive search algorithms fit into a rather restrictive pattern, which means that analyzing them can often be a straightforward "plug-and-chug" process. The following lemmas show how to analyze a proper exhaustive search algorithm by analyzing its separate parts.

Lemma 30. *A proper exhaustive search algorithm takes $O(g + c \cdot v)$ time, where*

- *there are $O(c)$ candidates;*
- *generating all candidates takes a total of $O(g)$ time; and*
- *verifying one candidate takes $O(v)$ time.*

Proof. The `exhaustive_search` pattern involves a single `for` loop followed by a `return` statement. By assumption the loop repeats $O(c)$ times; each iteration takes $O(v)$ time to call `<VERIFIER>`, plus $O(1)$ additional steps for loop overhead and the `if` statement. The `return` statement takes $O(1)$ time. By assumption, the `<GENERATE CANDIDATES>` part of the algorithm takes $O(g)$ total time. So the total time complexity of the pattern is

$$O(c(v + 1) + 1 + g) = O(g + c \cdot v).$$

□

The next lemma makes a similar characterization of exhaustive optimization algorithms; it is essentially the same, but must also account for the time spent comparing candidates.

Lemma 31. *An exhaustive optimization algorithm takes $O(g + c(v + b))$ time, where $g, c,$ and v are defined as in Lemma 30, and comparing two candidates takes $O(b)$ time.*

Proof. The `exhaustive_optimize` algorithm pattern uses the same time as does the `exhaustive_search` pattern, and in addition uses

- 1 step to initialize `best` to `None`;
- $O(b)$ time each time `<COMPARE>` is called; and
- 1 step to execute or skip the body of the inner `if` statement.

The `<COMPARE>` function is called whenever a new acceptable candidate is found, except for the first. There are $O(c)$ candidates in total, so $O(c)$ acceptable candidates, and the total time spent in all comparisons is $O(c \cdot m)$. So by reasoning similar to that of the previous lemma, the total time complexity of an exhaustive optimization is

$$O(g + c(v + b)).$$

□

7.5 Generating Candidates

Designing an exhaustive search algorithm involves filling in the `<GENERATE CANDIDATES>` blank. Often this involves iterating through the elements of data structures that comprise the `<INPUTS>` for the problem at hand; or pairs of those elements, subsets, or permutations. This Section explores algorithms for generating these sorts of combinatorial collections.

7.5.1 Iterators and Generators

As discussed in Section 4.2.4, fundamental data structures such as the vector, linked list, and binary search tree all provide an iterator interface that makes it possible to loop through the individual elements in one of those structures. In our Python-based pseudocode, the built-in function `iter` maps an iterable data structure to an iterator object that yields the elements of the structure. So, for example, if `V` is a vector, then `iter(V)` produces an iterator that visits each of the elements of `V`, and that iterator may be used in loop syntax such as:

```
for element in iter(V):
    <DO SOMETHING WITH element>
```

In the context of exhaustive search algorithms, we need functions that produce an iterable sequence, such as the sequence of pairs or subsets of an input collection. We will use the Python concept of *generator functions* for this kind of function. A generator function resembles an ordinary function, except that it uses the `yield` statement to indicate output instead of `return`. Recall that a `return x` statement means that a function's execution ends, and the function outputs the value `x` to the caller. By contrast a `yield x` statement means that a generator function is temporarily paused, and the function outputs the value `x` which is used as a loop index value in one iteration of the loop.

An example should make this clear. Suppose we define the following generator, which yields the sequence $\langle 1, 2, 3 \rangle$.

```
def threecount():
    yield 1
    yield 2
    yield 3
```

Then the code

```
for i in threecount():
    print(i * 2)
```

prints

```
2
4
6
```

A generator function may take arguments, and use loops, like any other function. So the following function is a generator function that takes an iterable data structure and yields the first three elements in that structure.

```
def first3(obj):
    i = 0
    for x in iter(obj):
        if i < 3:
            yield x
        else:
            break
```

Note that the `for x in iter(obj)` loop will ordinarily iterate through each element of `obj`,

regardless of the length of `obj`. So when `obj` contains three or more elements, `first3` yields exactly three of them; but when `obj` contains fewer than three elements, `first3` yields all of them. The code

```
V = Vector()
V.push_back(10)
V.push_back(11)
V.push_back(12)
V.push_back(13)
V.push_back(14)
V.push_back(15)
for y in first3(V):
    print(y)
```

prints

```
10
11
12
```

In particular, only the first three elements of `V` are printed even though `V` contains six elements.

7.5.2 Generating Ranges of Integers

Some exhaustive search algorithms involve generating ranges of integers as candidates. This is a straightforward problem that can be solved with a straightforward greedy algorithm.

integer range generation
input: integers s, e with $s < e$ **output:** an iterator for the integers $\{s, s+1, s+2, \ldots, e-1\}$

Note that, to be consistent with Python range notation, the output does not include the value e.

This problem is not particularly difficult; the following greedy algorithm solves it. Note that this algorithm is a generator, as described in Subsection 7.5.1. It uses the `yield` statement to produce one element of the sequence at a time.

```
def integer_range(s, e):
    i = s
    while i < e:
        yield i
        i += 1
```

Claim 3. *The* `integer_range` *algorithm yields a sequence of* $n = e - s$ *elements, takes* $O(n)$ *total time, and uses* $O(1)$ *space.*

Python includes a built-in function called `range` that is practically equivalent to our `integer_range` algorithm. Calling the function

```
range(s, e)
```

is practically equivalent to calling `integer_range(s, e)`. The first argument `s` to `range` may be omitted, in which case it defaults to `0`. So `range(e)` is equivalent to `range(0, e)`.

7.5.3 Generating Pairs

In many cases the candidates in an exhaustive search algorithm are formed from *pairs* of elements drawn from a list.

A straightforward algorithm with two `for` loops solves this problem.

```
def pairs(L, R):
    for l in L:
        for r in R:
            pair = (l, r)
            yield pair
```

Example 32. *For example, if* `L=[1, 2, 3]` *and* `R=[7,8]` *then* `pairs(L, R)` *yields* `(1, 7), (1, 8), (2, 7), (2, 8), (3, 7), (3, 8)`.

Claim 4. *The* `pairs` *algorithm yields a sequence of* $|L| \cdot |R|$ *elements, takes* $O(|L| \cdot |R|)$ *total time, and uses* $O(1)$ *space.*

7.5.4 Generating Subsets

Some problems' candidate solutions may be formed by generating the *power set* of a set, which is all possible *subsets* of that set.

Definition 33. *The* power set

$$\mathcal{P}(U) = \{S \mid S \subseteq U\}$$

is the set of all subsets of a given set U.

A set U is called the *universe* defining the power set $\mathcal{P}(U)$.

Example 33. $\mathcal{P}(\{1,2,3\}) = \{\varnothing, \{1\}, \{2\}, \{3\}, \{1,2\}, \{2,3\}, \{1,3\}, \{1,2,3\}\}$.

$\mathcal{P}(\{1,2\}) = \{\varnothing, \{1\}, \{2\}, \{1,2\}\}$.

$\mathcal{P}(\{1\}) = \{\varnothing, \{1\}\}$.

Note that the empty set is a subset of any set, including the empty set itself. So the power set of $\varnothing$ is a set containing one sole element, $\varnothing$.

Example 34. $\mathcal{P}(\varnothing) = \{\varnothing\}$.

When the universe set U contains n elements, its powerset contains $|\mathcal{P}(U) = 2^n|$ subsets. Each subsets contains at most n elements.

Generating power sets is a computational problem.

subset generation problem
input: a set U of n distinct elements, represented by an iterable object **output:** an iterator for every subset S of U, where each subset is represented by a vector of distinct elements

As stated above, if n is the number of elements in U, then we need to generate 2^n subsets of U. We need some way to iterate through this many subsets. We cannot use the nested loop approach that we used for generating pairs; two nested loops iterate n^2 times, three nested loops iterate n^3 times, four nested loops iterate n^4 time, and in general k nested loops iterate n^k time. No value of k can make $n^k = 2^n$, since n^k is a polynomial function and 2^n is an exponential function. Instead, we will iterate through the range of integers $0, \ldots 2^n - 1$, then devise a way to yield a distinct subset in each loop iteration.

```
def subsets(U):
    n = len(U)
    # (2 ** n) evaluates to 2 to the n power
    for i in range(2 ** n):
        S = <SOMEHOW CREATE A VECTOR CONTAINING ONE SUBSET OF U>
        yield S
```

In order to fill in the `<SOMEHOW CREATE A VECTOR CONTAINING ONE SUBSET OF U>` blank, we need to devise a mapping from an integer $i \in [0, 2^n)$ to a specific subset $S_i \subseteq U$.

The 2^n expression should be familiar to computer scientists; it shows up frequently, since a collection of $n > 0$ bits is can represent 2^n different bit strings. Here we have an integer `i` that will take on 2^n different non-negative index values. We are looking for a way to convert a particular n-bit integer value `i` to a particular n-element subset S_i of U. So each n-bit integer value will map to one set S_i containing up to n elements. That means that each bit of `i` will map to one particular element of U.

Conventionally bit positions are indexed starting from 0 in the least-significant bit position, written toward the right. And, conveniently arrays and vectors are indexed starting from 0 with the first element. We can use these conventional indexing schemes to define a mapping from index values to subsets:

$$\text{bit } j \text{ of } \texttt{i} \overset{\text{corresponds to}}{\Longleftrightarrow} \text{element of } U \text{ at index } j$$

Each bit of `i` is assigned either 0 or 1; in order to turn this correspondence into clear pseudocode, we need to decide whether a bit of 0 corresponds to including, or not including, a particular element. It is conventional in programming for 0 to correspond to false and 1 to correspond to true, so we decide that 1 coresponds to inclusion, and 0 corresponds to exclusion.

$$\text{bit } j \text{ of } \texttt{i} \text{ is } 1 \overset{\text{corresponds to}}{\Longleftrightarrow} \texttt{U[j]} \text{ is included in this subset}$$
$$\text{bit } j \text{ of } \texttt{i} \text{ is } 0 \overset{\text{corresponds to}}{\Longleftrightarrow} \texttt{U[j]} \text{ is } \textit{not} \text{ included in this subset}$$

Example 35. *Let $U = \{a, b, c\}$. Then the elements of U may be indexed as follows.*

index	*0*	*1*	*2*
Element of U	a	b	c

We have the following correspondence between the integers $[0, 2^3)$ and the subsets in $\mathcal{P}(U)$.

Index `i` *(decimal)*	*Index* `i` *(binary)*	*subset of U*
0	*000*	$\{\}$
1	*001*	$\{a\}$
2	*010*	$\{b\}$
3	*011*	$\{a, b\}$
4	*100*	$\{c\}$
5	*101*	$\{a, c\}$
6	*110*	$\{b, c\}$
7	*111*	$\{a, b, c\}$

We can implement this correspondence in our pseudocode.

```
def subsets(U):
  elements = <VECTOR CONTAINING THE ELEMENTS OF U>
  for i in range(2 ** len(elements)):
      S = Vector()
      for j in range(n):
          if <bit j of i is 1>:
              S.add_back(elements[j])
      yield S
```

For those familiar with bitwise operations, this pseudocode is probably clear. To be on the safe side, we will detail how to test whether `bit j of i is 1`. The basic idea is to use the *bit shift* operation to move bit number `j` to position 0 (the least-significant bit position); then test whether the least-significant bit is 0 or 1. We can perform that kind of test by computing the bitwise-and with 1; the result of that operation is 0 when the least-significant bit was 0, and 1 otherwise. In most ALGOL-derived languages, including Python, C++, and Java, the right-shift operator is `>>` and the bitwise-and operator is `&`. So the expression `(i >> j) & 1` produces 0 when bit `j` of `i` is 0, or 1 otherwise. For the sake of type safety we compare the result to 1, so `((i >> j) & 1) == 1` produces a Boolean `True` when bit `j` of `i` is 1.

Again for the sake of clarity, we write out the details of the `<VECTOR CONTAINING THE ELEMENTS OF U>` step.

```
def subsets(U):
  elements = Vector()
  for x in U:
      elements.add_back(x)
  for i in range(2 ** len(elements)):
      S = Vector()
      for j in range(n):
          if ((i >> j) & 1) == 1:
              S.add_back(elements[j])
      yield S
```

Claim 5. *The* `subsets` *algorithm yields a sequence of* 2^n *vectors of length at most* n *and takes* $O(2^n \cdot n)$ *total time.*

7.5.5 Generating Permutations

Recall that a *permutation* of a sequence U is a rearrangement of the elements of U that preserves all its elements. Note that, unlike with subsets, U must be a sequence with a defined order (instead of a set with undefined element order), and every permutation of U has the same length as U.

permutation generation problem
input: a list U of n elements, represented by an iterable object **output:** an iterator for every permutation p of U, where each permutation is represented by a list

Example 36. *The permutations of the sequence* $\langle a, b, c \rangle$ *are*

$$\langle a, b, c \rangle, \langle a, c, b \rangle, \langle b, a, c \rangle, \langle b, c, a \rangle, \langle c, a, b \rangle, \text{ and } \langle c, b, a \rangle.$$

Algorithms for generating permutations have been studied for centuries, and there are several

known algorithms with subtle tradeoffs. Here we cover the *Johnson-Trotter* algorithm [31] [50], which can be derived by studying how one might enumerate permutations by hand.

Suppose we were to start with a list of the permutations of just the first element a; that list contains only one permutation:

1. $\langle a \rangle$

Now suppose we add the next element b to this list. The element b could go to the left of a, or to the right of a. These two alternatives yield the two permutations of $\langle a, b \rangle$.

1. $\langle b, a \rangle$ (b is left of a from permutation 1)
2. $\langle a, b \rangle$ (b is right of a from permutation 1)

Now suppose we add the next element c to this list. This step is more complicated. There are three positions where c could go: to the left, middle, or right. And, c could assume that position in either permutation 1 or permutation 2 from the previous step. There are three positions, and two permutations where that position could be used, for a total of 3×2 permutations.

1. $\langle c, b, a \rangle$ (c is left of a from permutation 1)
2. $\langle b, c, a \rangle$ (c is right of b from permutation 1)
3. $\langle b, a, c \rangle$ (c is right of b from permutation 1)
4. $\langle c, a, b \rangle$ (c is left of a from permutation 2)
5. $\langle a, c, b \rangle$ (c is right of b from permutation 2)
6. $\langle a, b, c \rangle$ (c is right of b from permutation 2)

Observe that this list of permutations is the same as the list in Example 36, albeit in a different order. This is acceptable since the definition of the permutation generation problem does not dictate the order of the permutations.

If we focus on the newly-added b in the second step, we see that b "moves" left-to-right. Focusing on c in the third step, we see that c also moves left-to-right, making one left-to-right sweep for each of the two permutations from the second step. We could describe the six permutations as being generated by the following process:

1. c moves until it is stuck
2. then b moves, and c starts over
3. a always stays still

4. this repeats until b and c are both stuck

We can generalize this algorithm to sequences with more than three elements.

```
while <THERE IS A NON-STUCK ELEMENT>
    <FIND THE NON-STUCK ELEMENT WITH THE LARGEST INDEX AND MOVE IT>
```

The Johnson-Trotter algorithm models this notion of motion and stuck-ness by assigning a *direction* to each element of the permutation. The direction may be either *positive* meaning moving toward a higher index, or *negative* meaning moving toward a negative index. Each element also has a *rank,* which is its index in the original position. An element x in the current permutation is *mobile* when another element y exists in the direction that x is looking (i.e. x is not looking past the end of the permutation) and the rank of x is greater than the rank of y.

```
class JohnsonTrotterElement:
    def __init__(self, value, rank, direction):
        self.value = value
        self.rank = rank
        self.direction = direction

def permutations(U):
    n = len(U)
    elements = Vector()
    i = 0
    for x in iter(U):
        elements.add_back(JohnsonTrotterElement(x, i, +1))
        i += 1
    S = Vector(n, None)
    while True:
        least_mobile = None
        for i in range(n):
            looking_at = i + elements[i].direction
            if ((looking_at >= 0) and
                (looking_at < n) and
                (elements[i].rank > elements[looking_at].rank) and
                (least_mobile is None or
                 elements[i].rank > elements[least_mobile].rank)):
                least_mobile = i
        if not least_mobile:
            break
        swap(elements[i], elements[least_mobile])
        for e in elements:
            if e.rank > elements[least_mobile].rank:
                e.direction = -e.direction
        for i in range(n):
            S[i] = elements[i].value
        yield S
```

Claim 6. *The Johnson-Trotter algorithm for generating subsets yields a sequence of $n!$ vectors of length exactly n and takes $O(n! \cdot n)$ total time.*

7.6 Minimum Spanning Trees by Exhaustive Search

Now that we have algorithm for generating pairs, subsets, and permutations, we can start designing exhaustive search algorithms. We begin with the familiar minimum spanning tree problem first explored in Section 6.3.

minimum spanning tree problem
input: a connected, undirected, and weighted graph $G = (V, E)$ with $n = \|V\|$ vertices and $m = \|E\|$ edges **output:** a set of edges K such that $T = (V, K)$ is a minimum spanning tree for G

Our first design decision: should we use proper exhaustive search, or exhaustive optimization? Recall that proper exhaustive search is appropriate when solving the problem involves searching for an object with a particular property, while exhaustive optimization is appropriate when the problem involves searching for the *best* object with a particular property. In this case the "minimum" part of the "is a minimum spanning tree for G" phrase says that we are not searching for any old spanning tree, but rather the tree of minimum weight. So we will proceed with the exhaustive optimization pattern.

```
def exhaustive_optimize(<INPUTS>):
  best = None
  for candidate in <GENERATE CANDIDATES>(<INPUTS>):
    if <VERIFIER>(<INPUTS>, candidate):
      if best is None or <COMPARE>(best, candidate):
        best = candidate
  return best
```

The <INPUTS> are the set of all inputs for the problem; in this case, the only input to the minimum spanning tree problem is the graph G. We can break out the <VERIFIER> and <COMPARE> steps as their own sub-algorithms.

```
def exhaustive_mst(G):
  best = None
  for candidate in <GENERATE CANDIDATES>(G):
    if verify_mst(G, candidate):
      if best is None or compare_mst(best, candidate):
        best = candidate
  return best

def verify_mst(G, candidate):
  # TODO

def compare_mst(best, candidate):
  # TODO
```

In order to move forward we need to clarify what precisely each candidate is. Observe that each candidate must have the same data type as the output of the algorithm. A solution for the minimum

spanning tree problem must be a vector of edges; therefore each candidate must be a vector of edges corresponding to an alleged spanning tree. This gives us a hint for how to fill in the `verify_mst` and `compare_mst` steps. As proved in Section 6.3, a set of edges is a spanning tree when it contains exactly $n - 1$ edges and no cycles. The definition for a minimum spanning tree implies that our `compare_mst` sub-algorithm should prioritize the candidate (tree) whose total edge weight is less.

```
def exhaustive_mst(G):
  best = None
  for candidate in <GENERATE CANDIDATES>(G):
    if verify_mst(G, candidate):
      if best is None or compare_mst(best, candidate):
        best = candidate
  return best

def verify_mst(G, candidate):
  return ((len(candidate) == G.vertex_count()-1) and
          <CANDIDATE IS ACYCLIC>)

def compare_mst(best, candidate):
  best_weight = 0
  for edge in best:
    best_weight += edge.label
  candidate_weight = 0
  for edge in candidate:
    candidate_weight += edge.label
  return (candidate_weight > best_weight)
```

Finally we need to fill in the `<GENERATE CANDIDATES>` part. We already decided that each candidate is a vector of edges. We might ask ourselves: is each candidate a pair of edges, subset of edges, permutation of edges, or something else? Actually the definition of spanning tree tells us the answer. A spanning tree is a subgraph of the original graph $G = (V, E)$, comprised of the same vertex set V, and a subset of the graph's edge set E. Therefore each candidate tree is a *subset* of edges, not a pair or permutation. Alternatively, we could have come to this conclusion by thinking about how many elements candidates ought to have, and whether or not the elements of a candidate have an inherent order. Recall that subsets range in size from 0 to n and are unordered, while every permutation has length exactly n and is a sequence with a defined element order. A valid spanning tree has fewer than n edges, and an edge set is unordered, so subsets are appropriate and permutations are inappropriate.

```
def exhaustive_mst(G):
  best = None
  for candidate in subsets(G.edges()):
    if verify_mst(G, candidate):
      if best is None or compare_mst(best, candidate):
        best = candidate
  return best

def verify_mst(G, candidate):
  return ((len(candidate) == G.vertex_count()-1) and
          <CANDIDATE IS ACYCLIC>)

def compare_mst(best, candidate):
  best_weight = 0
  for edge in best:
    best_weight += edge.label
  candidate_weight = 0
  for edge in candidate:
    candidate_weight += edge.label
  return (candidate_weight > best_weight)
```

This pseudocode is clear except for the `<CANDIDATE IS ACYCLIC>` step, which is left as an exercise.

Claim 7. *There exists an algorithm that determines whether a given graph contains a cycle in $O(n+m)$ time.*

Lemma 32. *The time complexity of* `exhaustive_mst` *is $O(2^m \cdot (n+m))$.*

Proof. We will apply Lemma 30, so we need to determine expressions for $g, c,$ and v.

The exhaustive optimization algorithm uses `subsets` to generate candidates as subsets of the graph's m edges, so by Claim 5, $c = 2^m$ and $g \in O(2^m \cdot m)$.

By Claim 7, the verification sub-algorithm takes $O(n+m)$ worst-case time, so $v \in O(n+m)$.

The comparison sub-algorithm involves two loops, each of which iterates through all of the elements of a candidate. Each candidate is a subset of edges, so contains at most m elements; so the comparison algorithm takes $O(m)$ worst-case time and $b \in O(m)$.

By Lemma 31, the total time complexity of the algorithm is

$$O(g + c(v+m)) = O((2^m \cdot m) + (2^m)((n+m) + (m))) = O(2^m(n+m)).$$

□

7.7 Circuit Satisfaction

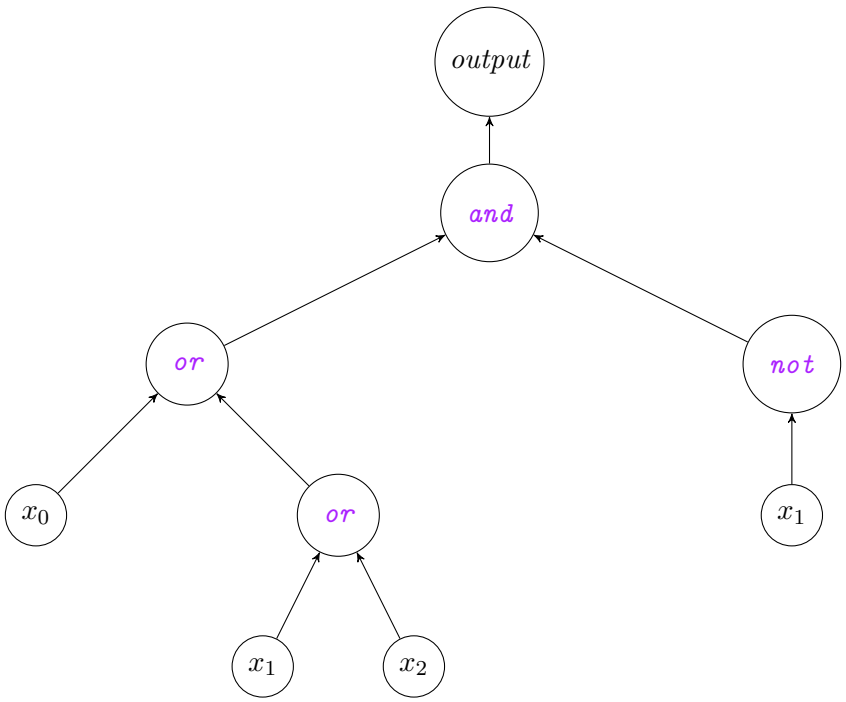
The *circuit satisfaction* problem is central to computational complexity. Its input is a *Boolean circuit* and its output is a *Boolean circuit assignment.*

Definition 34. *A* Boolean circuit $C = (T, X)$ *is a directed tree (a.k.a. directed acyclic graph)* $T = (V, E)$ *with* $n = |V|$ *vertices, and a set of* k *indexed variables* $X = \{x_0, x_1, \ldots, x_{k-1}\}$*, where* $k < n$ *and each vertex is one of*

1. *an* `and` *vertex with exactly two incoming edges and one outgoing edge;*
2. *an* `or` *vertex with exactly two incoming edges and one outgoing edge;*
3. *a* `not` *vertex with exactly one incoming and one outgoing edge;*
4. *an* output *vertex with exactly one incoming edge and no outgoing edges; or*
5. *a* literal *vertex labeled with an index* i *corresponding to variable* x_i*, no incoming edges, and exactly one outgoing edge.*

There is exactly one output vertex, and it is the root of the tree.

Example 37. *The following Boolean circuit has* $k = 3$ *variables and* $n = 9$ *vertices.*



Definition 35. *A* Boolean circuit assignment A *over* k *variables is a list of* k *Boolean* `True` *or* `False` *values* $A = \langle a_0, a_1, \ldots, a_{k-1} \rangle$*. Each* a_i *corresponds to a value that may be assigned to a variable* x_i *in a corresponding* k*-variable Boolean circuit.*

Example 38.

$$A = \langle \texttt{True}, \texttt{False}, \texttt{True} \rangle$$

is a Boolean assignment compatible with the preceding Boolean circuit.

A Boolean circuit may be evaluated given an appropriate Boolean circuit assignment.

Example 39. *The output of the circuit for assignment A is* `True`*. For* $A' = \langle$ `True`, `True`, `True` $\rangle$ *it is* `False`*.*

The task of evaluating a Boolean circuit can be framed as a computational problem.

Boolean circuit evaluation
input: A Boolean circuit $C = (X, T)$ with k variables and n vertices, and Boolean circuit assignment A with k variables **output:** the Boolean output of circuit C for assignment A

The following recursive algorithm evaluates Boolean circuits.

```
def eval_circuit(C, A):
  return recurse(C.get_output_vertex(), A)

def recurse(root, A):
  if root is a literal vertex:
    i = root.get_variable_index()
    return A[i]
  else if root is an and vertex:
    return (recurse(root.first_incoming_neighbor(), A) and
            recurse(root.second_incoming_neighbor(), A))
  else if root is an or vertex:
    return (recurse(root.first_incoming_neighbor(), A) or
            recurse(root.second_incoming_neighbor(), A))
  else: # a not vertex
    return not recurse(root.only_incoming_neighbor(), A)
```

Lemma 33. *The time and space complexity of* `eval_circuit` *are both $O(n)$.*

Proof. The time and space complexity of `eval_circuit` are dominated by the time and space complexity of the `recurse` function. We count the steps for two separate concerns:

1. Recursive calls to `recurse`, and
2. everything else.

First consider concern 2, everything in the body of `recurse` aside from recursive function calls. The body of the algorithm is an if/else branch with four cases. In each case the algorithm makes $O(1)$ steps (not counting recursive calls). So the total number of steps in concern 2 is $O(1)$.

Now consider concern 1. T is a tree, so contains no cycles, and `recurse` only calls itself recursively on incoming neighbors, so `eval_circuit(X, T, v, A)` is called at most once for each $v \in V$. As discussed each function call takes $O(1)$ time aside from recursive function calls. So the total time spent in recursion is at most $O(1 \cdot n) = O(n)$.

Thus the total time spent in `recurse`, and by extension in `eval_circuit`, is $O(n)$.

`recurse` uses only $O(1)$ space for local variables (`i`) and perhaps $O(1)$ time for call stack overhead. As discussed there are n calls to `recurse` for a total of $O(1 \cdot n) = O(n)$ space. □

Satisfying a Boolean circuit means feeding it an assignment that causes it to evaluate `True`.

Definition 36. *A Boolean assignment A* satisfies *a Boolean circuit C if A and C both have exactly k variables, and C evaluates to `True` when evaluated with A.*

circuit satisfaction problem
input: a Boolean circuit C with k variables and n vertices **output:** a Boolean assignment A that sastisfies C, or `None` if no such assignment exists

We now design an exhaustive search algorithm for circuit satisfaction. There is no notion of a "best" solution in the problem's output statement, so we start with the proper exhaustive search pattern.

```
def circuit_sat(C):
  for candidate in <GENERATE CANDIDATES>(C):
    if <VERIFIER>(C, candidate):
      return candidate
  return None
```

As stated in the problem's output definition, a non-`None` output must be an assignment A. So each candidate solution is a Boolean assignment.

```
def circuit_sat(C):
  for assignment in <GENERATE ASSIGNMENTS>(C):
    if <VERIFIER>(C, assignment):
      return assignment
  return None
```

The verifier is straightforward; it evaluates the circuit `C` with `assignment` and simply tests whether the output is `True`.

```
def circuit_sat(C):
  for assignment in <GENERATE ASSIGNMENTS>(C):
    if verify_circuit_sat(C, assignment):
      return assignment
  return None

def verify_circuit_sat(C, assignment):
  if eval_circuit(C, assignment) == True:
    return True
  else:
    return False
```

We do not have an algorithm for generating Boolean circuit assignments directly. However, we can generate assignments by using our subset generation algorithm. In an assignment each variable x_i is either assigned `True` or `False`. So a subset X_T of the variable set X is assigned `True`, and the remaining $X_F = X - X_T$ variables must be assigned `False`. So we can generate all the possible X_T sets of `True`-assigned variables as subsets of X, and form each assignment A from these sets X_T.

```
def circuit_sat(C):
  n = C.get_n()
  for assigned_true in subsets(range(n)):
    assignment = [None] * n
    for i in range(n):
      if i in assigned_true:
        assignment[i] = True
      else:
        assignment[i] = False

    if verify_circuit_sat(C, assignment):
      return assignment

  return None

def verify_circuit_sat(C, assignment):
  if eval_circuit(C, assignment) == True:
    return True
  else:
    return False
```

Lemma 34. *The time complexity of* `circuit_sat` *is* $O(2^n \cdot n)$.

Proof. `circuit_sat` is an exhaustive search algorithm that

- examines $c = 2^n$ candidates;
- verifying each assignment takes $v \in O(n)$ time;
- generating all the candidates takes time $g \in O(2^n \cdot n)$ time.

So by Lemma 30 `circuit_sat` takes

$$O(g + c \cdot v) = O((2^n \cdot n) + (2^n)(n)) = O(2^n \cdot n).$$

□

7.8 Traveling Salesperson

The *traveling salesperson problem* is a classic problem in graph theory [5].

traveling salesperson (TSP)
input: a graph $G = (V, E)$ where each edge $e \in E$ has a numeric weight w_e **output:** a Hamiltonian cycle in G of minimum total weight, or `None` if no such cycle exists

This definition references several terms from graph theory, which we briefly review.

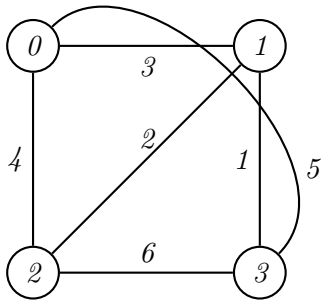
Definition 37. *A* path *in a graph $G = (V, E)$ is a nonempty sequence of vertices $L = \langle l_0, \ldots, l_{k-1} \rangle$ such that every $l_i \in V$, and for any pair of vertices l_i, l_{i+1} adjacent in L, G contains an edge from l_i to l_{i+1}.*

Definition 38. *A* Hamiltonian path *in a graph $G = (V, E)$ is a path in G such that every $v \in V$ appears in C exactly once.*

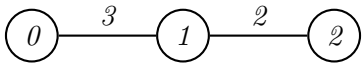
Definition 39. *A* cycle *is a path $C = \langle c_0, \ldots, c_{k-1} \rangle$ such that $c_0 = c_{k-1}$.*

Definition 40. *A cycle $C = \langle c_0, \ldots, c_{k-2}, c_0 = c_{k-1} \rangle$ is* Hamiltonian *when the path formed by removing one of the duplicated copies of c_0, e.g. $C' = \langle c_0, \ldots, c_{k-2} \rangle$ is a Hamiltonian path.*

Example 40. *The following graph G_1 has $n = 4$ vertices, and is complete (all pairs of distinct vertices are connected by edges), so has $n! = 4! = 24$ distinct Hamiltonian cycles. For example the cycle $C_1 = \langle 0, 1, 2, 3, 0 \rangle$ has total weight 16 and the cycle $C_2 = \langle 0, 1, 3, 2, 0 \rangle$ has total weight 14.*



Example 41. *The following graph G_2 has no cycles, so if it were uses as an instance to the traveling salesperson problem the output would be* `None` *.*



The traveling salesperson problem has received a lot of attention, in some part due to its puzzle-like character, and in large part due to its practical applicability. Of course, the problem, or slight variations of it, apply directly to the problem of planning routes for travelers that must visit a set itinerary such as salespeople, police patrol officers, or tourists. It also applies to operational business scenarios such as maximizing the throughput of a manufacturing assembly line, or minimizing the length of wires in a printed circuit.

The phrase "of minimum total weight" in the problem's output statement dictates that this is an optimization problem, so our first draft of an algorithm is based upon the exhaustive optimization pattern.

```
def tsp(G):
  best = None
  for candidate in <GENERATE CANDIDATES>(G):
    if <VERIFIER>(G, candidate):
      if best is None or <candidate IS BETTER THAN best>:
        best = candidate
  return best
```

We first consider the data type of each candidate solution. An output of the TSP problem is a cycle that has the Hamiltonian property. A cycle is a path (with the additional property that it starts and ends at the same vertex); a path is represented mathematically by a sequence of vertices. So the most appropriate data structure for each candidate is a *list of vertex objects.*

In general, graphs may have a huge number of paths. Actually any graph with a cycle has an infinite number of paths, since any path incident to the cycle can loop through the cycle an arbitrary

number of times. So it is impossible to design a terminating algorithm that generates every path in an arbitrary graph. However, in the case of the TSP, the only kinds of paths that are acceptable are Hamiltonian cycles. Since a Hamiltonian cycle visits every vertex exactly once, or twice in the case of the first-and-last vertex, every Hamiltonian cycle in a graph of n vertices has length exactly $n+1$. So our (finite) graph G has a finite number of Hamiltonian cycles, and it should be possible for a terminating algorithm to generate them all.

Recall that a Hamiltonian cycle is essentially a Hamiltonian path that is extended by one step to reconnect with the starting vertex. Let us focus for a moment on the problem of generating every Hamiltonian *path* in G. A Hamiltonian path is a sequence (list) in some order such that each vertex in G appears exactly once. So we can generate all candidate Hamiltonian paths in G *by generating all permutations of* G*'s vertices* V*.*

```
def tsp(G):
  best = None
  for path in permutations(G.get_vertices()):
    if <VERIFIER>(G, candidate):
      if best is None or <candidate IS BETTER THAN best>:
        best = candidate
  return best
```

A path of distinct vertices may be "closed off" and converted into a cycle by duplicating its first vertex.

```
def tsp(G):
  best = None
  for path in permutations(G.get_vertices()):
    cycle = path + [path[0]]
    if <VERIFIER>(G, cycle):
      if best is None or <cycle IS BETTER THAN best>:
        best = cycle
  return best
```

A cycle is optimal if its total weight is minimal.

```
def tsp(G):
  best = None
  for path in permutations(G.get_vertices()):
    cycle = path + [path[0]]
    if <VERIFIER>(G, cycle):
      if best is None or cycle_weight(cycle) < cycle_weight(best):
        best = cycle
  return best

def cycle_weight(G, cycle):
  total = 0
  for i in range(len(cycle)-1):
    # look up the cost of the edge from vertex cycle[i] to cycle[i+1]
    total += G.edge_weight(cycle[i], cycle[i+1])
  return total
```

The only blank left is `<VERIFIER>`, which needs to determine whether a given `cycle` is an acceptable Hamiltonian cycle. By construction each `cycle` value is indeed a cycle, visits the first-last vertex exactly twice, and all other vertices exactly once. The only other property that `cycle` must have is that the graph G does actually contain all the edges necessary to follow `cycle`. For example the cycle $C_3 = \langle 0, 1, 2, 0 \rangle$ meets the structural requirements of being a list of vertices that starts and ends at the same vertex. However C_3 is not acceptable for the graph G_2 in Example 41 since G_2 has no edge $\{2, 0\}$.

```
def tsp(G):
  best = None
  for path in permutations(G.get_vertices()):
    cycle = path + [path[0]]
    if verify_tsp(G, cycle):
      if best is None or cycle_weight(cycle) < cycle_weight(best):
        best = cycle
  return best

def cycle_weight(G, cycle):
  total = 0
  for i in range(len(cycle)-1):
    # look up the cost of the edge from vertex cycle[i] to cycle[i+1]
    total += G.edge_weight(cycle[i], cycle[i+1])
  return total

def verify_tsp(G, cycle):
  for i in range(len(cycle)-1):
    if not G.contains_edge(cycle[i], cycle[i+1]):
      return False
  return True
```

The complexity of `tsp` follows from Lemma 31.

Claim 8. *The time complexity of the* `tsp` *algorithm is* $O(n! \cdot n^2)$.

7.9 The Knapsack Problem

The *knapsack problem* is an optimization problem that deals with doing more with less [39]. The problem is conventionally described in terms of selecting a set of *items* to fit inside *knapsack*. Each item has an integer *weight* and real-number *value,* the knapsack has an integer weight capacity, and the goal is to choose a subset of items that maximizes the total value while fitting within the knapsack's weight capacity.

knapsack problem
input: a list of n item objects I, where each item `i` has a positive `i.value` and positive integer `i.weight` ; and a weight limit $W \geq 0$
output: a list K of items from I, such that the total weight of all items in K is at most W, and the total value of the items in K is maximized

This wording tells a story about storing physical objects into a bag, perhaps for a camping trip.

However the setup generalizes to any situation that involves selecting a subset of discrete options, so as to maximize some numeric utility measure, while subject to some kind of integer budget. The "items" could be purchases that must fit within a monetary household budget; components to add to an airplane that must fit within a weight limit; pieces of cargo to transport on a merchant ship that must fit within a cargo capacity; and so on.

In order to design an algorithm we must first decide whether to use the exhaustive search or exhaustive optimization pattern. The "total value of the items in K is maximized" phrase makes it clear that this is an optimization. We start by filling in a few obvious blanks in the exhaustive optimization pattern.

```
def exhaustive_knapsack(W, items):
    best = None
    for candidate in <GENERATE CANDIDATES>(items):
        if verify_knapsack(W, items, candidate):
            if best is None or total_value(candidate) > total_value(best)
                best = candidate
    return best
```

The problem involves choosing only some of the input items. Each item may be either chosen or not chosen, and the order of the choices does not matter, so `<GENERATE CANDIDATES>` corresponds to enumerating all subets of `items`. The problem definition says that we want the subset with the greatest possible value, which is why we introduce the `total_value` function intended to compute the total value of a given set of items. The `verify_knapsack` function needs to determine whether a given set of items is valid, which means computing whether the items' total weight is W or less. Fleshing out those details gives us a clear algorithm.

```
def exhaustive_knapsack(W, items):
    best = None
    for candidate_items in subsets(items):
        if verify_knapsack(W, items, candidate_items):
            if best is None or total_value(candidate_items) > total_value(best)
                best = candidate
    return best

def verify_knapsack(W, items, candidate_items):
    total_weight = 0
    for item in candidate_items:
        total_weight += item.weight
    if total_weight <= W:
        return True
    else:
        return False

def total_value(candidate_items):
    total_value = 0
    for item in candidate_items:
        total_value += item.value
    return total_value
```

This is a straightforward application of the exhaustive search pattern, so it should be clear that the algorithm is correct.

Claim 9. *The exhaustive search algorithm for the knapsack problem is correct.*

Analyzing the algorithm is straightforward as well.

Claim 10. *The exhaustive search algorithm for the knapsack problem runs in* $O(2^n \cdot n)$ *time.*

Exercises

7-1. Compute the power set of $\{a, b, c, d\}$ using the algorithm from Subsection 7.5.4. Show your work.

7-2. Compute the list of all permutations of $\langle a, b, c, d \rangle$ using the Johnson-Trotter algorithm from Subsection 7.5.5. Show your work.

7-3. Draw an unsatisfiable instance of the circuit satisfaction problem with at least $n = 5$ vertices.

7-4. Execute the `tsp` algorithm by hand on the graph G_2 from Example 40. What is the optimal Hamiltonian cycle for G_2, and what is that cycle's total weight? Show your work.

7-5. For each of the following problems: design an exhaustive search or optimization algorithm that solves the problem; describe your algorithm with clear pseudocode; and prove the time efficiency class of your algorithm. When writing your pseudocode, you can assume that your audience is familiar with the candidate generation algorithms in this chapter, so you can make statements like "for each subset X of S" without explaining the details of how to generate subsets.

(a)

subset sum problem
input: a list X of n distinct integers, and a target integer k **output:** a subset $S \subseteq X$ such that $k = \sum_{x \in S} x$, or `None` if no such S exists

(b)

set partition problem
input: list of X of n distinct positive integers **output:** a pair of lists (L, R) such that L and R form a set partition $X = L \cup R$ and $\sum_{l \in L} l = \sum_{r \in R} r$, or `None` if no such partition exists

(c)

Pythagorean triple problem
input: two positive integers a, b with $a < b$ **output:** a *Pythagorean triple* (x, y, z) such that x, y and z are positive integers, $a \leq x \leq y \leq z \leq b$, and $x^2 + y^2 = z^2$, or `None` if no such triple exists

(d) [40] An accomplished masseuse has received requests for appointments filling her entire work day. However, she needs a break of at least 1 minute between appointments. Which appointment-requests should she honor, and which should she decline?

the masseuse's problem
input: a list of requested appointment-lengths, in minutes (none overlap and none can be moved) **output:** a list of appointment-lengths that can be honored, such that no adjacent requests are included, and the total number of minutes is maximized

Example: For input `[30, 15, 15, 200, 30, 60]` the optimal list of appointments to honor is `[30, 200, 60]` for a total of 290 minutes.

7-6. The TSP algorithm described in Section 7.8 runs in $O(n! \cdot n^2)$ time, where n is the number of vertices in the input graph. Design a different exhaustive search algorithm that runs in $O(2^m(n+m))$ time. Which algorithm is faster for large n, and why?

7-7. *Code golf.* Re-implement each of the following algorithms with the shortest Python code you can manage. Whitespace doesn't count, so focus on meaningful simplifications rather than superficial formatting.

(a) `pairs`

(b) `subsets`

(c) `permutations`

Chapter 8

Decrease-by-Half (Divide and Conquer)

8.1 The Big Idea

The greedy and *decrease-by-half* patterns have many similarities. Both involve breaking a problem instance into pieces, performing a processing step that reduces the number of outstanding pieces, and repeating that processing step until all the pieces have been handled. In the greedy pattern each processing step reduces the number of input pieces by exactly 1, so a problem instance with n pieces requires n processing steps. By contrast, in the decrease-by-half pattern, each processing step reduces the number of input pieces by roughly *half*, so only about $\log_2 n$ processing steps are required. n grows substantially faster than $\log_2 n$ so this represents a very significant reduction in the number of processing steps.

Note that we must say "roughly" half since n may be odd, in which case $\frac{n}{2}$ is not an integer. But problem instances are usually comprised of discrete data structures that cannot be broken into fractional parts. For example a list of $n = 11$ elements cannot be divided into two lists of exactly $\frac{n}{2} = 5.5$ elements; there is no such thing as .5 elements of a list. In that case a decrease-by-half algorithm could divide the list into one list of 5 elements and another of 6. These lists have roughly, but not precisely, the same length. In spirit decrease-by-half algorithms divide each instance in half, but in actuality they divide each instance into two pieces of size $\lfloor \frac{n}{2} \rfloor$ and $\lceil \frac{n}{2} \rceil$ respectively. This is usually insignificant in terms of big-O efficiency classes and empirically-observed efficiency, but adds some clutter to our mathematical notation and analyses.

The decrease-by-half principle can be generalized to algorithms that divide their input into smaller fractional pieces than halves. A *decrease-by-third* algorithm reduces the number of input pieces to roughly $\frac{1}{3}n$ and requires about $\log_3 n$ steps; a *decrease-by-fourth* algorithm reduces the number of input pieces to roughly $\frac{1}{4}n$ and requires about $\log_4 n$ steps; and in general a *decrease-by-a-fraction* algorithm reduces the number of input pieces to roughly $\frac{1}{d}n$ and requires about $\log_d n$ steps, for some integer $d > 1$. Again we must qualify our statements with "roughly" since n may not be

evenly divisible by d. As we will see, decrease-by-half algorithms are usually simpler and just as efficient as other kinds of decrease-by-a-fraction algorithms, so we focus on the former. However the latter is a more natural way of solving certain problems.

Decrease-by-half algorithms can be implemented in a recursive or non-recursive fashion. In general the recursive version is more elegant and easier to analyze. Like any other recursive algorithm, a decrease-by-half algorithm first checks whether it is in a base case that can be solved without any recursive calls. In the recursive case it

1. *divides* the input into two smaller problem instances of roughly equal size, which we can call a *left half* and *right half;*
2. *recursively solves* the two smaller instances to obtain two solutions; then
3. *combines* the solutions to the smaller instances into one solution for the original undivided instance.

In pseudocode:

```
def decrease_by_half(<INPUTS>):
  if <THIS IS A BASE CASE>:
    return <BASE CASE SOLUTION>
  else:
    L, R = <DIVIDE instance INTO TWO INSTANCES OF ROUGHLY EQUAL SIZE>
    L_solution = decrease_by_half(L)
    R_solution = decrease_by_half(R)
    instance_solution = <COMBINE L_solution WITH R_solution>
    return instance_solution
```

Of the four blanks in this pattern, `<COMBINE L_solution WITH R_solution>` is typically the most challenging to fill in. Identifying and handling base cases is usually straightforward, and dividing instances in half is often as easy as slicing a vector down the middle. But efficiently combining two solutions to small problem instances into a correct solution for a larger instance often requires some insight.

8.2 Example: Summation

As a first concrete example, recall the summation problem.

summation problem
input: a list X of numbers **output:** the sum (total) of all elements of X

Of course there is a simple greedy algorithm that solves this problem in $O(n)$ time.

```
def greedy_sum(V):
    total = 0
    for x in V:
        total += x
    return total
```

That being said, our present exercise is to practice using the decrease-by-half pattern. We begin by filling in easy blanks.

```
def sum_dbh(V):
  if <THIS IS A BASE CASE>:
    return <BASE CASE SOLUTION>
  else:
    L, R = <DIVIDE V INTO TWO INSTANCES OF ROUGHLY EQUAL SIZE>
    L_solution = sum_dbh(L)
    R_solution = sum_dbh(R)
    instance_solution = <COMBINE L_solution WITH R_solution>
    return instance_solution
```

(`dbh` stands for Decrease-By-Half.)

This problem has two straightforward base cases: the sum of an empty vector is zero, and the sum of a single-element vector is simply the value of that element.

```
def sum_dbh(V):
  n = len(V)
  if n == 0:
    return 0
  elif n == 1:
    return V[0]
  else:
    L, R = <DIVIDE V INTO TWO INSTANCES OF ROUGHLY EQUAL SIZE>
    L_solution = sum_dbh(L)
    R_solution = sum_dbh(R)
    instance_solution = <COMBINE L_solution WITH R_solution>
    return instance_solution
```

What is the simplest way to `<DIVIDE instance INTO TWO INSTANCES OF ROUGHLY EQUAL SIZE>` ?

The first thing that comes to mind is to assign `L` the first $n/2$ elements of `V`, and assign the remaining elements to `R`.

```
def sum_dbh(V):
  n = len(V)
  if n == 0:
    return 0
  elif n == 1:
    return V[0]
  else:
    half = n / 2
    L = V[0] ... V[half-1]
    R = V[half] ... V[n-1]
    L_solution = sum_dbh(L)
    R_solution = sum_dbh(R)
    instance_solution = <COMBINE L_solution WITH R_solution>
    return instance_solution
```

The only remaining blank is `<COMBINE L_solution WITH R_solution>`. At that point of the algorithm,

- `L_solution` should be the sum of `L`, i.e. the sum of the first $\approx n/2$ elements of `V`;
- `R_solution` should be the sum of `R`, i.e. the sum of the remaining $\approx n/2$ elements of `V`; and
- we need to compute `instance_solution`, i.e. the sum of *all* elements of `V`.

How do we combine `L_solution` with `R_solution` to form a single correct sum for all of `V`? Simply add them together.

```
def sum_dbh(V):
  n = len(V)
  if n == 0:
    return 0
  elif n == 1:
    return V[0]
  else:
    half = n / 2
    L = V[0] ... V[half-1]
    R = V[half] ... V[n-1]
    L_solution = sum_dbh(L)
    R_solution = sum_dbh(R)
    instance_solution = L_solution + R_solution
    return instance_solution
```

This pseudocode is functional, but is longer than it needs to be since it initializes variables `L`, `R`, `L_solution`, `R_solution`, and `instance_solution` which are only referenced once each. We can make the pseudocode more concise by eliminating these single-use variables.

```
def sum_dbh(V):
  n = len(V)
  if n == 0:
    return 0
  elif n == 1:
    return V[0]
  else:
    half = n / 2
    L = V[0] ... V[half-1]
    R = V[half] ... V[n-1]
    return sum_dbh(L) + sum_dbh(R)
```

Analyzing this algorithm poses a problem. We already know how to account for the number of steps in every part of the algorithm except for the two recursive `sum_dbh` function calls. Let $T(n)$ be the number of steps executed by `sum_dbh` in the recursive (non-base) case; then a chronological step count yields

$$T(n) = 1 + \max(2, 3, 4 + |L| + |R| + T(|L|) + T(|R|).$$

By construction $|L| = \lfloor n/2 \rfloor$ and $|R| = n - |A| = n - \lfloor n/2 \rfloor = \lceil n/2 \rceil$, so

$$T(n) \quad = \quad T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) + n + 5 \tag{8.1}$$

which is a recursive definition! None of the properties of O established in Chapter 2 can help us

simplify away the $T(\lfloor n/2 \rfloor)$ or $T(\lceil n/2 \rceil)$ terms. We must establish a new technique for proving efficiency classes of recursively-defined complexity functions.

8.3 Analyzing Decrease-By-Fraction Algorithms

The not-so-humbly-named *master theorem* is a mathematical theorem that gives us a way to prove the efficiency classes of certain *recurrences* (recursively-defined functions) through a foolproof plug-and-chug process. The theorem only applies to recurrences in a form that arises from a complexity analysis of a decrease-by-a-fraction algorithm. Formalizing that form requires introducing several variables with peculiar definitions, so a statement of the master theorem can be an imposing visage. But applying the master theorem to prove efficiency classes is actually quite simple, and indeed analyzing decrease-by-a-fraction algorithms is often faster and easier than analyzing non-recursive algorithms.

8.3.1 The Master Theorem

Definition 41 (master form)**.** *A function is a* master-form recurrence *when it may be defined as*

$$\begin{array}{rcll} T(n) & = & r \cdot T\left(\frac{n}{d}\right) + c(n) & \forall n \geq t \\ T(n) & \in & O(1) & \forall n < t \end{array}$$

where

$$\begin{array}{rcl} c(n) & \in & O(n^k) \\ r, d, t, k & \in & O(1) \\ r, t & \geq & 1 \\ d & > & 1 \\ k & \geq & 0. \end{array}$$

As with the definition of O, this is an unwieldy definition, but its component parts make some intuitive sense. The base case of the recurrence

$$T(n) \quad \in \quad O(1) \quad \forall n < t$$

says that the the algorithm's time complexity is constant for a base case, or perhaps a few cases where $n < t$. So t is the boundary between base cases and recursive cases. The recursive case of the recurrence

$$T(n) \quad = \quad r \cdot T\left(\frac{n}{d}\right) + c(n) \quad \forall n \geq t$$

Says that, when $n \geq t$ so we are in a non-base case, the algorithm's time complexity is the sum of

1. r recursive calls, each on an input of size n/d, for a total of $r \cdot T\left(\frac{n}{d}\right)$ time spent in recursive calls; and

2. $c(n)$ time spent on everything aside from recursive calls, including dividing the input into pieces and combining solutions.

The requirement that $c(n) \in O(n^k)$ stipulates that the time complexity of the "everything else" steps is polynomial. This is a realistic assumption; it rules out algorithms that spend exponential or factorial time on the dividing and combining parts of the algorithm, which are not consistent with the spirit of the decrease-by-half pattern. This requirement also means that master-form recurrences are reasonably well-behaved mathematically, which makes it feasible to state and prove the master theorem relatively concisely.

The remaining requirements are mathematical technicalities that "sanity-check" that T is a legitimate complexity function in master-form. The parameters r, d, t, and k must all be constant with respect to n to rule out bizarre recurrences, which would never correspond to sincere decrease-by-fraction algorithms, such as $T(n) = n^3 \cdot T(\frac{n}{\sqrt{n}}) + n$. The $r \geq 1$ inequality ensures that the algorithm is indeed recursive (if $r = 0$ then it never makes a recursive call, and $r < 0$ is nonsense). The $t \geq 1$ inequality ensures that there exists at least one base case. The $d > 1$ inequality ensures that the size $\frac{n}{d}$ of each instance solved recursively is strictly smaller than n. And finally, the $k \geq 0$ inequality ensures that $c(n)$ is a non-decreasing function (otherwise its efficiency class could be e.g. $O(n^{-1}) = O(1/n)$, which is nonsense).

Theorem 4 (master theorem)**.** *Let T be a master-form recurrence; then*

$$T(n) \in \begin{cases} O(n^k) & \text{if } r < d^k \text{ (case 1),} \\ O(n^k \log n) & \text{if } r = d^k \text{ (case 2), or} \\ O(n^{\log_d r}) & \text{if } r > d^k \text{ (case 3).} \end{cases}$$

A rigorous proof of the master theorem is necessarily technical and long, so we will not present one here. See [35] or Appendix B of [38] for a full proof.

so we present only an informal sketch of a proof that illustrates where the three cases come from. A sound proof of the theorem may be found in [35] or Appendix B of [38].

8.3.2 The Master Method

The *master method* is the approach of applying the master theorem to prove the efficiency class of a time complexity function. The master method can be broken down into steps:

1. Identify the size t for which all $n < t$ is a base case.

2. Prove that the algorithm's time complexity is $O(1)$ when $n < t$.

3. Suppose the algorithm is given a non-base-case input ($n \geq t$) and perform another chronological step count to arrive at $T(n) = r \cdot T\left(\frac{n}{d}\right) + c(n)$. Make note of the parameters r and d.

4. Use properties of O to prove the efficiency class $c(n) \in O(n^k)$, and make note of the parameter k.

5. Check that $r, d, t, k \in O(1)$, and that $r \geq 1$, $t \geq 1$, $d > 1$, and $k \geq 0$.

6. Evaluate d^k, compare r to d^k, determine which of the three cases applies, and state the efficiency class of $T(n)$.

Example 42. *We prove the efficiency class of our most recent version of* `sum_dbh`*.*

Lemma 35. `sum_dbh` *runs in* $O(n \log n)$ *time.*

Proof. We apply the master method.

1. The algorithm has two base cases, when $n = 0$ or $n = 1$, so we define $t = 2$.

2. The time complexity of the base cases is

$$\begin{aligned} T(0) &= 3 \\ T(1) &= 4. \end{aligned}$$

   Therefore $T(n) \in O(1)$ when $n < t$.

3. When not in a base case (i.e. when $n \geq 2$), as shown in Equation (8.1), the algorithm's time complexity is

$$T(n) = 2 \cdot T(n/2) + n + 5,$$

   or

$$T(n) = 2 \cdot T(n/2) + c(n)$$

   where $c(n) \equiv n + 5$. So we have $r = 2$ and $d = 2$.

4. By properties of O, $c(n) = n + 5 \in O(n) = O(n^k)$ for $k = 1$.

5. Observe that our $r, d, t, k \in O(1)$, and our $r \geq 1$, $t \geq 1$, $d > 1$, and $k \geq 0$, so we may apply the master theorem.

6. $d^k = (2)^{(1)} = 2$ and $r = 2$, so $r = d^k$, we are in case 2, and by the master theorem

$$t(n) \in O(n^k \log n) = O(n^1 \log n) = O(n \log n).$$

□

We must concede at this point that `sum_dbh` is an unimpressive algorithm. The greedy alternative given earlier is simpler and takes only $O(n)$ time; our recursive algorithm is slower at $O(n \log n)$ time, and is more complex. While the decrease-by-half pattern guided us toward designing an acceptable algorithm, it is not the best pattern for this problem. This is our first example of a general truth about algorithm design.

Remark 1. *While it may be possible to design an algorithm for a particular problem using multiple different patterns, often one pattern works better than the others.*

The remainder of this Chapter covers decrease-by-half algorithms that manage to outperform decrease-by-one algorithms for the same problem.

8.4 Merge Sort

Merge sort [28] is a classical algorithm that solves the sorting problem.

sorting problem
input: a list U of n comparable elements **output:** a list S containing the elements of U in non-decreasing order

Merge sort is one of the oldest sorting algorithms to run in $O(n \log n)$ time, which is significantly faster, both in theory and in practice, than decrease-by-one sorting algorithms such as selection sort and insertion sort which run in $O(n^2)$ time. Merge sort follows the decrease-by-half pattern closely. To derive the algorithm we start by renaming identifiers in the decrease-by-half pattern.

```
def merge_sort(V):
  if <THIS IS A BASE CASE>:
    return <BASE CASE SOLUTION>
  else:
    L, R = <DIVIDE V INTO TWO INSTANCES OF ROUGHLY EQUAL SIZE>
    sorted_L = merge_sort(L)
    sorted_R = merge_sort(R)
    sorted_V = <COMBINE sorted_L WITH sorted_B>
    return sorted_V
```

The base case of a sorting problem instance is a vector `V` that is so small that it does not need to be sorted. You might recall from our discussion of in-place selection sort in Subsection 5.5.3 that sequences of length $n \leq 1$ are trivially sorted. That is because a sequence that is empty or that contains only one element cannot possibly have any pair of two elements out of order.

Claim 11. *Any sequence of length $n \leq 1$ is in sorted order.*

This observation allows us to fill in the blanks related to handling the base case.

```
def merge_sort(V):
  if len(V) <= 1:
    return V
  else:
    L, R = <DIVIDE V INTO TWO INSTANCES OF ROUGHLY EQUAL SIZE>
    sorted_L = merge_sort(L)
    sorted_R = merge_sort(R)
    sorted_V = <COMBINE sorted_L WITH sorted_R>
    return sorted_V
```

In `sum_dbh` we handled the `<DIVIDE...>` step by simply splitting `V` at index `len(V) / 2`. We see no reason not to use the same approach here.

```
def merge_sort(V):
  if len(V) <= 1:
    return V
  else:
    half = len(V) / 2
    L = V[0] ... V[half-1]
    R = V[half] ... V[n-1]
    sorted_L = merge_sort(L)
    sorted_R = merge_sort(R)
    sorted_V = <COMBINE sorted_L WITH sorted_R>
    return sorted_V
```

As stated above, typically the `<COMBINE...>` step is the most challenging to design, and that seems to be the case here. It is not obvious how to combine `sorted_L` and `sorted_R` into a single sorted vector of length exactly n. We cannot simple concatenate the vectors together since that approach would yield incorrect outputs whenever any element x in `sorted_L` should come after any element in `sorted_R`.

Example 43. *Suppose* `V = [3, 1, 5, 6, 2, 4]`. *Then*

$$A = [3, 1, 5]$$

and

$$B = [6, 2, 4] \ .$$

If the recursive calls `merge_sort(L)` *and* `merge_sort(R)` *work correctly, then*

L_sorted = [1, 3, 5]

R_sorted = [2, 4, 6] .

Unfortunately appending these vectors results in

[1, 3, 5, 2, 4, 6]

which is not in proper non-decreasing order.

This combining process is shaping up to be a challenge, so we factor it into its own sub-algorithm called `merge` .

```
def merge_sort(V):
  if len(V) <= 1:
    return V
  else:
    half = len(V) / 2
    L = V[0] ... V[half-1]
    R = V[half] ... V[n-1]
    sorted_L = merge_sort(L)
    sorted_R = merge_sort(R)
    sorted_V = merge(sorted_L, sorted_R)
    return sorted_V

def merge(L, R):
  return <COMBINE L WITH R>
```

We can now define the problem that `merge` needs to solve.

merge problem
input: two lists L and R in non-decreasing order, of length n_L and n_R respectively **output:** a list S in non-decreasing order containing all the elements of both L and R

The merge problem bears a very close resemblance to the sorting problem. Both problems involve producing an ordered vector. Of course the input to the sorting problem is a vector that is presumably unordered, while the input to the merge problem is two separate vectors that certainly are ordered. Let us consider using the greedy sorting pattern, introduced in Section 5.5 to design `merge` . Recall that the pattern is:

```
def greedy_sort(V):
    S = Vector()
    for element in V:
        <SOMEHOW INSERT element INTO S SO THAT S IS STILL IN ORDER>
    return S
```

After renaming identifiers to correspond to `merge` and initializing `result` to an empty list, we obtain:

```
def merge(A, B):
  S = Vector()
  for element in <A or B>:
    <SOMEHOW INSERT element INTO S SO THAT S IS STILL IN ORDER>
  return S
```

In Section 5.5.2 we saw how to develop this vague pseudocode into the selection sort algorithm. The selection sort algorithm handled the `<SOMEHOW INSERT...>` step by selecting the least unsorted element and then appending it to `S`. We can use the same approach here.

```
def merge(L, R):
  S = Vector()
  while <UNSORTED ELEMENTS REMAIN IN L OR R>:
    x = <FIND THE LEAST UNSORTED ELEMENT IN L AND R>
    <REMOVE x FROM L/R>
    S.add_back(x)
  return S
```

In selection sort the `<FIND THE LEAST UNSORTED ELEMENT...>` step took $O(n)$ time since the unsorted elements were stored in an unordered list. In this case the unsorted elements are stored in two *sorted* lists. While we could certainly perform a sequential search through `L` and then `R` in $O(n_L + n_R)$ time, that approach makes no use of the fact that `L` and `R` are already sorted, and it stands to reason that that fact can help us speed up the selection process.

Let us consider how we might `FIND THE LEAST UNSORTED ELEMENT...>` in faster than $O(n)$ time. Recall the input in Example 43. For the problem instance `V = [3, 1, 5, 6, 2, 4]`, `merge` would get inputs

```
L = [1, 3, 5]
```

and

R = [2, 4, 6].

Stare at those lists for a moment, and consider the question: *where might the least element among* `L` *and* `R` *be located?*

The answer: the least element of `L` and `R` is *either the first element of* `L` *or the first element of* `R`.

There are only two locations that might contain the least unsorted element `x`. Instead of $O(n)$ locations to search, as in sequential search, we have only $2 \in O(1)$ locations to search, so we ought to be able to search them in $O(1)$ time. We can "search" for the least element in `L[0]` and `R[0]` simply by comparing those elements to each other.

```
if L[0] <= R[0]:
  x = L[0]
else:
  x = R[0]
```

Going back to our example, `L[0]=1` and `R[0]=2` so `L[0] <= R[0]` and we would extract `x` from `L`. Then we have

L = [3, 5]

and

R = [2, 4, 6].

Once again, we ask: *where might the least element among* `L` *and* `R` *be located?* The answer is the same: at the beginning of the unprocessed part of either `L` or `R`. In this case the least element is at the front of `R`. Generalizing this process into a loop, we get:

```
def merge(L, R):
  S = Vector()
  while L and R are nonempty:
    if L[0] <= R[0]:
      S.add_back(L[0])
      L.remove_front()
    else:
      S.add_back(R[0])
      R.remove_front()
  return S
```

This pseudocode is correct. However it is rather inefficient. `L` is a vector, so the `L.remove_front` operation shuffles elements and takes $O(n_L)$ time. Similarly the `R.remove_front` operation takes $O(n_R)$ time. We can circumvent this problem by leaving `L` and `R` unchanged and using index variables to keep track of how far we have made it into each vector. We introduce the following invariant which formalizes the idea that `merge` does not duplicate or lose any elements.

Invariant 6. *The `merge` algorithm maintains `S` in non-decreasing order, and index variables `li` and `ri` such that, for any element of `L`,*

$$\texttt{L[i]} \begin{cases} \textit{has been copied into } \texttt{S} & \textit{when } \texttt{i} < \texttt{li} \\ \textit{has not been sorted yet} & \textit{when } \texttt{i} \geq \texttt{li} \end{cases}$$

and symmetrically

$$\texttt{R[i]} \begin{cases} \textit{has been copied into } \texttt{S} & \textit{when } \texttt{i} < \texttt{ri} \\ \textit{has not been sorted yet} & \textit{when } \texttt{i} \geq \texttt{ri}. \end{cases}$$

```
def merge(L, R):
  S = Vector()
  li = ri = 0
  while li < len(L) and ri < len(R):
    if L[li] <= R[ri]:
      S.add_back(L[li])
      li += 1
    else:
      S.add_back(R[ri])
      ri += 1
  return S
```

We are getting close, but unfortunately this algorithm may violate its invariant, and is not correct.

Example 44. *Suppose `L=[1, 2, 3]` and `R=[10, 11, 12]`. The `while` loop in `merge` will repeatedly pick `L[li]` until `li == len(L)`, at which point the loop terminates. So the algorithm as written would return `[1, 2, 3]` and fail to return any of the elements in `R`.*

The problem is that the loop condition `li < len(L) and ri < len(R)` means that the loop stops as soon as `li == len(L)` *or* `ri == len(R)`. Suppose for the moment that `li == len(L)`; then the loop stops, but `ri < len(R)` so there are still unsorted elements remaining in `R`. Those elements need to be copied into `S`. Symmetrically, if the loop ends because `ri == len(R)` then unsorted elements remain in `L`.

We can solve this problem by adding loops that copy any of these remaining "straggler" elements into `S`.

```
def merge(L, R):
  S = Vector()
  li = ri = 0
  while li < len(L) and ri < len(R):
    if L[li] <= R[ri]:
      S.add_back(L[li])
      li += 1
    else:
      S.add_back(R[ri])
      ri += 1
  for i in range(li, len(L)):
      S.add_back(L[i])
  for i in range(ri, len(R)):
      S.add_back(R[i])
  return S
```

It may appear odd to append elements from *both* `L` and `R` since only one of them contains elements. But, as discussed above, the `while` loop ends when *one* of `L` or `R` is depleted of elements; the other still has elements that need to go into `S`. Therefore one of the `for` loops will never iterate.

Lemma 36. *The time complexity of the* `merge` *algorithm is* $O(n)$.

Proof. Recall that $n = n_L + n_R$ is the number of elements collectively in `L` and `R`, or equivalently the length of the vector `S` returned by `merge`. The algorithm has three loops; observe that the total number of iterations of all the loops is n. If the `while` loop iterates w times, then either the first `for` loop iterates the remaining $(n - w)$ times, or else the second `for` loop repeats $(n - w)$ times. The body of every loop takes $O(1)$ time, so the total time spent in the loops is $O(n)$. The other parts of the algorithm take only $O(1)$ time, so the total time for the algorithm is $O(n)$. □

Now that we have designed and analyzed the `merge` sub-algorithm, we can give complete pseudocode for merge sort.

```
def merge_sort(V):
  if len(V) <= 1:
    return V
  else:
    half = len(V) / 2
    L = V[0] ... V[half-1]
    R = V[half] ... V[n-1]
    sorted_L = merge_sort(L)
    sorted_R = merge_sort(R)
    sorted_V = merge(sorted_L, sorted_R)
    return sorted_V

def merge(L, R):
  S = Vector()
  li = ri = 0
  while li < len(L) and ri < len(R):
    if L[li] <= R[ri]:
      S.add_back(L[li])
      li += 1
    else:
      S.add_back(R[ri])
      ri += 1
  for i in range(li, len(L)):
      S.add_back(L[i])
  for i in range(ri, len(R)):
      S.add_back(R[i])
  return S
```

The variables `L`, `R`, `sorted_L`, `sorted_R`, and `sorted_V` are only ever referenced once, so we can simplify the pseudocode by eliminating them.

```
def merge_sort(V):
  if len(V) <= 1:
    return V
  else:
    half = len(V) / 2
    L = V[0] ... V[half-1]
    R = V[half] ... V[n-1]
    return merge(merge_sort(L), merge_sort(R))

def merge(L, R):
  S = Vector()
  li = ri = 0
  while li < len(L) and ri < len(R):
    if L[li] <= R[ri]:
      S.add_back(L[li])
      li += 1
    else:
      S.add_back(R[ri])
      ri += 1
  for i in range(li, len(L)):
      S.add_back(L[i])
  for i in range(ri, len(R)):
      S.add_back(R[i])
  return S
```

Lemma 37. *The time complexity of* `merge_sort` *is* $O(n \log n)$.

Proof. Let $T(n)$ be the time complexity of `merge_sort`. The base case occurs when `len(V) <= 1`, i.e. when $n \leq 1$, and takes 2 steps, so

$$T(n) \leq c \text{ for all } n \leq b$$

for $b = 1$ and $c = 2$.

The recursive case spends 1 step on the `if`, one step computing `half`, $O(n)$ time copying `V` into two smaller vectors `L` and `R`, $O(n)$ time in `merge`, and finally makes one recursive call on each of `L` and `R`. Therefore the time complexity of the recursive case is

$$\begin{aligned} T(n) &= 1 + 1 + n + n + T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) \\ &\leq 2 \cdot T(\lceil n/2 \rceil) + 2n + 2. \end{aligned}$$

This recurrence is in master form with $a = 2, d = 2$, and $f(n) = 2n + 2$. We have $f(n) \in O(n) = O(n^1)$, so $k = 1$. Noting that $a \geq 1, b \geq 0, d \geq 2, c \geq 1,$ and $k \geq 0$ are constant with respect to n,

and that

$$\begin{aligned} a &\overset{?}{<} d^k \\ (2) &\overset{?}{<} (2)^{(1)} \\ 2 &= 2, \end{aligned}$$

by case 2 of the master theorem $T(n) \in O(n^k \log n) = O(n^1 \log n) = O(n \log n)$. □

We should briefly contemplate the significance of this result. Recall that our optimized selection sort algorithm runs in $O(n^2) = O(n \times n)$ time, while merge sort runs in $O(n \log n) = O(n \times \log n)$ time. According to this mathematical analysis, merge sort is significantly faster (for large n), since $n \gg \log n$. This may be counter-intuitive since merge sort performs several operations that are notorious for being slow. Merge sort is inherently recursive, and is non-in-place, and conventional wisdom states that recursion is slow, as is allocating and freeing large lists. Despite that conventional wisdom, merge sort is indeed much faster than selection sort (and other $O(n^2)$-time sorting algorithms), both in theory and in practice.

8.5 Binary Search

Binary search is an algorithm for searching for an element within an *ordered* vector of comparable elements. It is related to sequential search, in that the input is a query value q and a vector V that may or may not contain q. Binary search differs, however, because it is only applied to vectors that are already sorted, and that property allows for a faster algorithm than sequential search which runs in $O(n)$ time.

binary search
input: a vector V of n comparable elements in non-decreasing order, and a query value q **output:** the index i such that `V[i]` $= q$ or `None` if no such index exists

We consider how to solve this problem using a decrease-by-half style algorithm.

```
def binary_search(V, q):
  if <THIS IS A BASE CASE>:
    return <BASE CASE SOLUTION>
  else:
    L, R = <DIVIDE instance INTO TWO INSTANCES OF ROUGHLY EQUAL SIZE>
    solution_L = binary_search(L)
    solution_R = binary_search(R)
    instance_solution = <COMBINE solution_L WITH solution_R>
    return instance_solution
```

The base case is an empty vector, in which case the output is always `None`.

```
def binary_search(V, q):
  if len(V) == 0:
    return None
  else:
    L, R = <DIVIDE instance INTO TWO INSTANCES OF ROUGHLY EQUAL SIZE>
    solution_L = binary_search(L)
    solution_R = binary_search(R)
    instance_solution = <COMBINE solution_L WITH solution_R>
    return instance_solution
```

Once again we can `<DIVIDE instance INTO TWO INSTANCES OF ROUGHLY EQUAL SIZE>` by splitting V in half.

```
def binary_search(V, q):
  if len(V) == 0:
    return None
  else:
    half = len(V) / 2
    L = V[0] ... V[half-1]
    R = V[half] ... V[n-1]
    solution_L = binary_search(L)
    solution_R = binary_search(R)
    instance_solution = <COMBINE solution_L WITH solution_R>
    return instance_solution
```

What about the `<COMBINE solution_L WITH solution_R>` part? There are four cases:

1. `q` could happen to be equal to `V[half]`;
2. `binary_search(L)` may find a non-`None` match;
3. `binary_search(R)` may find a non-`None` match; or
4. `q` is nowhere to be found.

Our algorithm needs to work out which case we are in.

```
def binary_search(V, q):
  if len(V) == 0:
    return None
  else:
    half = len(V) / 2
    L = V[0] ... V[half-1]
    R = V[half] ... V[n-1]
    solution_L = binary_search(L)
    solution_R = binary_search(R)
    if q == V[half]:
      return half
    elif solution_L is not None:
      return solution_L
    elif solution_R is not None:
      return solution_R + half
    else:
      return None
```

Note that in the `solution_R is not None` case, we must return `solution_R + half` since i is relative to the start of `R`, which corresponds to the element of `V` at index `half`.

This algorithm works, but its efficiency is suspect. Intuitively the algorithm is looking for *one* element of `V`. Our current pseudocode searches in *both* halves `L` and `R` of `V`. If `V` were completely unstructured this would be sensible. However we know that `V` is already in non-decreasing order. So do we really need to search in both halves of `V`? If all the elements of `V` are distinct, the answer is *no*. The `V[i]` equal to `q` is either in `L` or `R`, but not both. If `V` does contain duplicates then we are free to return any index i such that `V[i]` is equal to `q`. So whether `V` contains duplicates or not, we can recursively search in *only one* of `L` or `R`; there is no need to search in both sub-lists.

```
def binary_search(V, q):
  if len(V) == 0:
    return None
  else:
    half = len(V) / 2
    L = V[0] ... V[half-1]
    R = V[half] ... V[n-1]
    if <q could be in L>:
      return binary_search(L)
    elif q == V[half]:
      return half
    else:
      # by process of elimination, if q is anywhere in V, it must be in R
      i = binary_search(R, q)
      if i is None:
        return None
      else:
        return i + half
```

Example 45. *Suppose* `V=[1, 3, 4, 5, 9, 13]`*; then* `binary_search` *would divide* `V` *into* `L=[1, 3, 4]` *and* `R=[5, 9, 13]`*. Observe that, for any query value* q *that exists in* `V`*,* q *is either in* `L` *or in* `R` *but not both.*

How can we determine whether `<q could be in L>` efficiently? Well, since `V` is in non-decreasing order and `L` is the first `half` elements of `V`, `L` is in nondecreasing order too. If $L = \langle a_0, a_1, \ldots, a_{n_A-1} \rangle$, then for any $0 \leq i < (n_A - 1)$, $a_i \leq a_{i+1}$. By transitivity any $a_i \leq a_{n_A-1}$. So any a_i such that $a_i = q$ must satisfy $x \leq a_{n_A-1}$. So when $q \leq a_{n_A-1}$, `L` may contain `q`; otherwise `L` certainly does not contain `q`.

```
def binary_search(V, q):
  if len(V) == 0:
    return None
  else:
    half = len(V) / 2
    L = V[0] ... V[half-1]
    R = V[half] ... V[n-1]
    if q < V[half]:
      return binary_search(L, q)
    elif q == V[half]:
      return half
    else:
      # by process of elimination, if q is anywhere in V, it must be in R
      i = binary_search(R, q)
      if i is None:
        return None
      else:
        return i + half
```

Let us analyze this draft of `binary_search`.

Lemma 38. *The time complexity of the* `binary_search` *algorithm shown above is* $O(n)$.

Proof. `binary_search` has one base case

$$T(0) = 2$$

so $T(n) \leq c$ for any $n \leq b$, where $c = 2$ and $b = 0$. In the non-recursive case the algorithm takes

- 2 steps getting to the `else` on line 4,
- 1 step computing `half`,
- $\lfloor n/2 \rfloor$ steps constructing `L`,
- $\lceil n/2 \rceil$ steps constructing `R`, and
- in the worst case the `else:` block is executed; it takes $1 + T(|R|) + 4$ steps.

So the total number of steps in the recursive case is

$$\begin{aligned} T(n) &= 2 + 1 + \lfloor n/2 \rfloor + \lceil n/2 \rceil + 1 + T(\lceil n/2 \rceil) + 4 \\ &= T(\lceil n/2 \rceil) + n + 8 \\ &\leq a \cdot T(\lceil n/d \rceil) + f(n) \end{aligned}$$

where $a = 1, d = 2$, and $f(n) = n + 8 \in O(n) = O(n^k)$ for $k = 1$. Observe that $a \geq 1, b \geq 0, c \geq 1, d \geq 2$, and $k \geq 0$, so we may apply the master theorem. $d^k = (2)^{(1)} = 2$ and $a = 1$, so $a < d^k$, and by case 1 of the master theorem $T(n) \in O(n^k) = O(n)$. □

This result says that our current draft of `binary_search` is not particularly efficient; it runs in $O(n)$ time, which is the same as the $O(n)$ time complexity of sequential search, which makes no use of the orderedness of `V`.

What is the bottleneck in our current algorithm? In the above proof we derived

$$T(n) \leq T(\lceil n/2 \rceil) + n + 8$$

where the dominating term n came from the construction of `L` and `R`. Our current algorithm copies `V` into two new vector objects, which takes $O(n)$ time and is the source of the $+n$ term above. However, we don't really need to do that; these vectors are never changed, so it should be possible to reuse the original vector `V`. We can do that by refactoring our recursive algorithm to solve a variant of the binary search problem.

binary search in range
input: a vector V of n elements, query element q, and range indices s, e such that $0 \leq s \leq e \leq n$ **output:** the index i such that $s \leq i < e$ and `V[i]` $= q$, or `None` if no such index exists

Note that the output index `i` is relative to the start index 0 of `V`, not relative to the start of the range s.

Our refactored pseudocode is below.

```
def binary_search_range(V, q, s, e):
  if s == e:
    return None
  else:
    half = (s + e) / 2
    if q < V[half]:
      return binary_search_range(V, q, s, half)
    elif q == V[half]:
      return half
    else:
      i = binary_search_range(V, q, half+1, e)
      if i is None:
        return None
      else:
        return i
```

The `if i is None` statement actually always returns the value of `i`, so we can eliminate that statement and the variable `i`. We can solve the original binary search problem by calling `binary_search_range` on the entire `range(0, n)`. The complete pair of algorithms is below.

```
def binary_search(V, q):
  return binary_search_range(V, q, 0, len(V))

def binary_search_range(V, q, s, e):
  if s == e:
    return None
  else:
    half = (s + e) / 2
    if q < V[half]:
      return binary_search_range(V, q, s, half)
    elif q == V[half]:
      return half
    else:
      return binary_search_range(V, q, half+1, e)
```

Lemma 39. *The time complexity of* `binary_search` *is* $O(\log n)$.

Proof. Clearly the time complexity of `binary_search` is dominated by that of `binary_search_range`. The base case of `binary_search_range` has time complexity

$$T(0) = 2$$

so $T(n) \leq c$ for all $n \leq b$ when $b = 0$ and $c = 2$. In the recursive case the algorithm spends 3 steps before the `if q < V[half]:` line. In the worst case the algorithm executes the `else:` case which takes at most $3 + T(\lceil n/2 \rceil)$ steps. So the total number of steps is

$$\begin{aligned} T(n) &= 3 + 3 + T(\lceil n/2 \rceil) \\ &= T(\lceil n/2 \rceil) + 6 \\ &= a \cdot T(\lceil n/d \rceil) + f(n) \end{aligned}$$

for $a = 1, d = 2$, and $f(n) = 6$. We have $f(n) = 6 \in O(1) = O(n^k)$ for $k = 0$. Observe that $a \geq 1, b \geq 0, c \geq 1, d \geq 2,$ and $k \geq 0$, so we may apply the master theorem. $d^k = (2)^{(0)} = 1$ and $a = 1$, so $a = d^k$, and by case 2 of the master theorem $T(n) \in O(n^k \log n) = O(n^{(0)} \log n) = O(\log n)$. □

This is our first example of a *sublinear time complexity*. The number of steps executed by the algorithm is much less than the length n of the list (up to constant factors). This is a very fast algorithm, though its speed depends on `V` arriving presorted, which we know would take $O(n \log n)$ time to achieve with merge sort.

8.6 Indivisible Problems

The decrease-by-a-fraction pattern cannot be used to solve all problems. Recall that the pattern is:

```
def decrease_by_half(<INPUTS>):
  if <THIS IS A BASE CASE>:
    return <BASE CASE SOLUTION>
  else:
    L, R = <DIVIDE instance INTO TWO INSTANCES OF ROUGHLY EQUAL SIZE>
    L_solution = decrease_by_half(L)
    R_solution = decrease_by_half(R)
    instance_solution = <COMBINE L_solution WITH R_solution>
    return instance_solution
```

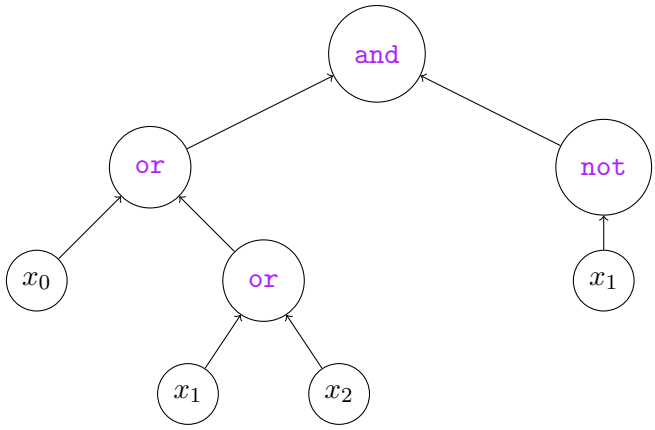
The viability of the pattern depends on two properties of the problem at hand:

1. it must always be possible to divide any instance into two sub-instances of roughly equal size; and

2. it must always be possible to combine the solutions to those sub-instances into a correct output for the entire original instance.

While the sorting and binary search problems may have these properties, that is not true of all problems. First, not all problem inputs may be divided cleanly into two coherent parts. For example, it is not always possible to divide a graph with n vertices and m edges into two smaller subgraphs with $\approx \frac{n}{2}$ vertices and $\approx \frac{m}{2}$ edges, since edges need not be distributed evenly throughout the graph, and some of the graph's edges may straddle the boundary between the two subgraphs. (As a matter of fact, actually there are algorithms for finding *separators* of graphs for use with the decrease-by-a-fraction pattern [44], but they are beyond the scope of this introductory text.)

Second, a problem's structure may make it impossible to compute `decrease_by_half(L)` and `decrease_by_half(R)` independently of one another. Some problems involve "global" inter-dependencies that necessitate processing an entire instance all at once rather than piecemeal. A good example is the circuit satisfaction problem. Recall that an instance of that problem is a Boolean circuit, such as the following.



Each `and` and `or` node has two inputs, and each `not` has one input. It is tempting to attempt to solve the problem with a decrease-by-half algorithm.

```
def circuit_sat_dbh(C):
  if C is a variable vertex:
    i = C.get_variable_index()
    return <assignment with x_i=True>
  elif C is a not vertex:
    return circuit_sat_dbh(C.get_only_input())
  elif C is an and vertex:
    L_assignment = circuit_sat_dbh(C.get_first_input())
    R_assignment = circuit_sat_dbh(C.get_second_input())
    return <COMBINE_AND>(L_assignment, R_assignment)
  else: # an or vertex
    L_assignment = circuit_sat_dbh(C.get_first_input())
    R_assignment = circuit_sat_dbh(C.get_second_input())
    return <COMBINE_OR>(L_assignment, R_assignment)
```

There is a huge obstacle to clarifying the `<COMBINE_AND>` and `<COMBINE_OR>` parts: the sub-assignments `L_assignment` and `R_assignment` may contain contradictory settings for some of the variables. In the Boolean circuit above, both the left subtree and right tree reference variable x_1. It is possible that `L_assignment` might set x_1 to `True` while `R_assignment` might set x_1 to `False` (or vice-versa). There is no straightforward way of resolving this contradiction, since toggling one of those settings would invalidate the satisfaction of the corresponding subtree. Simply put, there is no straightforward way of dividing an instance of the circuit satisfaction problem into two independently-solvable parts. Part of the problem's nature is that its instances are *indivisible*. This property seems to be shared by all of the so-called NP-complete problems, of which circuit satisfaction is an important example.

Exercises

8-1. Recall that the master theorem includes bounds on the constants $r \geq 1, d > 1, t \geq 1,$ and $k \geq 0$. For each of these inequalities, explain why the inequality helps prevent the master theorem from being applied to nonsensical recurrences, and given an example of a recurrence that violates the inequality and so is not in master form.

8-2. Sort the characters of the string "SEQUENCE" using merge sort; show your work.

8-3. *Decrease-by-third merge sort.* Design a version of merge sort that works by dividing V into three lists of length $\approx \frac{n}{3}$ instead of two lists of length $\approx \frac{n}{2}$. Prove the time efficiency class of your algorithm using the master method. How does your algorithm compare to the classical divided-by-half merge sort algorithm?

8-4. Design a variation of the binary search algorithm that does not use recursion. *Hint:* use a single `while` loop and index variables inspired by the `s` and `e` inputs to `binary_search_range`.

8-5. For each of the following problems: design decrease-by-half algorithm that solves the problem; describe your algorithm with clear pseudocode; and prove the time efficiency class of your algorithm.

(a)

min and max
input: a list L of $n > 0$ comparable objects **output:** a pair (l, g) where l and g are the least and greatest elements in L, respectively

Hint: An optimal algorithm for this problem has time complexity $O(n)$.

(b)

bottom left corner
input: a vector $V = \langle (x_0, y_0), (x_1, y_1), \ldots, (x_{n-1}, y_{n-1}) \rangle$ of n points $(x_i, y_i) \in \mathbb{R}^2$ **output:** a point (x_{min}, y_{min}) where $x_{min} \leq x_i$ for all $0 \leq i < n$ and $y_{min} \leq y_j$ for all $0 \leq j < n$

Hint: An optimal algorithm for this problem has time complexity $O(n)$.

(c) [40]

magic index
input: a list L of n distinct numbers in increasing order **output:** index i such that `L[i]==i`, or `None` if no such i exists

Hint: An optimal algorithm for this problem has time complexity $O(\log n)$.

(d) [40]

matrix binary search
input: an $m \times n$ matrix M, where each column is in non-decreasing order, and each row is in non-decreasing order; and a query value q **output:** (i, j) such that `M[i][j]==q`; or `None` if no such (i, j) exist

Hint: An optimal algorithm for this problem has time complexity $O(\log n)$.

(e) [40]

A *rotated-sorted list* is formed by sorting a list into non-decreasing order, dividing it into two sub-lists, and swapping the order of the sub-lists. Using list slice notation, if `S` is a

sorted list, and i is some "seam" index, then `S[i:] + S[:i]` is a rotated-sorted list. For example, the list `[3, 1, 2, 4, 5]` could be rotated-sorted into `[4, 5, 1, 2, 3]`, `[5, 1, 2, 3, 4]`, or even `[1, 2, 3, 4, 5]`.

search in a rotated-sorted list
input: a rotated-sorted list L, and query value q **output:** the index i such that `L[i]==q`, or `None` if no such i exists

Hint: An optimal algorithm for this problem has time complexity $O(\log n)$.

(f)

closest pair of sorted numbers
input: a list $X = \langle x_0, \ldots, x_{n-1} \rangle$ of $n \geq 2$ numbers in non-decreasing order **output:** a pair (p, q) such that $p, q \in X$ and $\lvert p - q \rvert \leq \lvert x_i - x_j \rvert$ for all $x_i, x_j \in X$

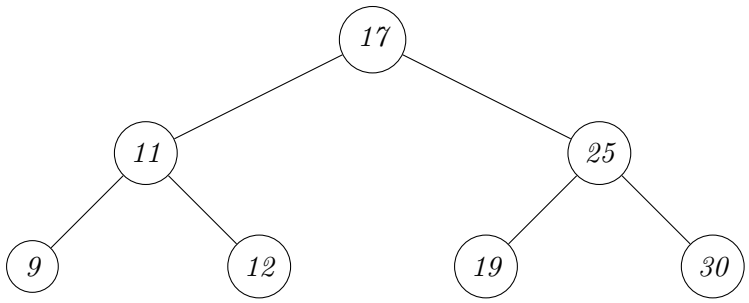
Example 46. *For input* $X = \langle -11, 5, 8, 12, 32 \rangle$ *there are two correct outputs,* $(5, 8)$ *or* $(8, 5)$.

Hints: An optimal algorithm for this problem has time complexity $O(n)$. If you divide `X` into two parts `L` and `R`, the closest pair could be in `L`, `R`, or straddling the boundary between `L` and `R`.

(g)

perfect binary search tree construction
input: a vector V of $n \geq 0$ orderable elements in strictly increasing order, where $n = 2^h - 1$ is one less than a power of 2 **output:** the root node of a balanced binary search tree of height $O(h) = O(\log n)$ containing the elements of V

Example 47. *For the input* $V = \langle 9, 11, 12, 17, 19, 25, 30 \rangle$*, a correct output is the following binary search tree:*



Note that $n = 7 = 2^3 - 1, h = 3$*, and the tree has height* $h - 1 = 2$.

Hint: An optimal algorithm for this problem has time complexity $O(n)$.

8-6. For each of the following problems: design decrease-by-fourth algorithm that solves the problem; describe your algorithm with clear pseudocode; and prove the time efficiency class of your algorithm.

(a)

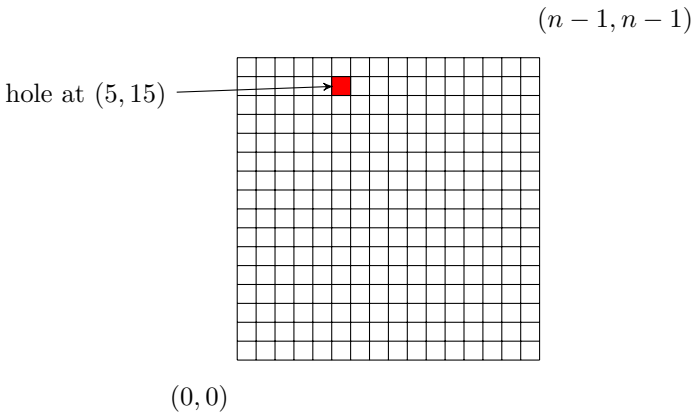
square matrix addition
input: two $n \times n$ matrices A and B **output:** an $n \times n$ matrix $C = A + B$

Recall that, in a matrix addition $C = A + B$, each element $c_{i,j}$ is the sum of the elements of A and B at the same row and column:

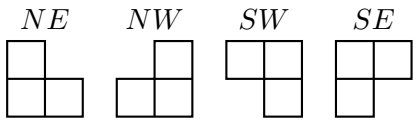
$$c_{i,j} = a_{i,j} + b_{i,j}.$$

(b) A *right tromino* (henceforth called *tromino*) is a shape formed by attaching three squares in an "L" shape.

Suppose you want to *tile* a square $n \times n$ grid, containing exactly one *hole* grid square, with trominoes. Such a tiling covers every grid square (except the hole) exactly once, so that there are no overlapping trominoes or gaps.



This may involve rotating trominoes into their four possible orientations.



right tromino tiling
input: an integer $n > 0$, which is a power of two representing the width and height of an $n \times n$ grid; and coordinates (x_h, y_h) of a *hole* in the grid, such that $0 \leq x_h < n$ and $0 \leq y_h < n$ **output:** a vector of k tromino objects $T = \langle (x_0, y_0, r_0), (x_1, y_1, r_1), \ldots, (x_{k-1}, y_{k-1}, r_{k-1}) \rangle$ where each (x_i, y_i) is the coordinate of the top-left corner of a tromino, and each r_i is one of NE, NW, SW, SE indicating the rotational orientation of the tromino

Writing detailed pseudocode to manipulate trominoes and coordinates could become tedious, so feel free to use high-level prose such as "Create a SE tromino in the top-left corner" or "Create a tromino covering the three open tiles around (x, y)."

8-7. *Code golf.* Re-implement each of the following algorithms with the shortest Python code you can manage. Whitespace doesn't count, so focus on meaningful simplifications rather than superficial formatting.

(a) `merge`

(b) `binary_search_range`

Chapter 9

Randomization

9.1 The Big Idea

Randomized algorithms intentionally make random choices. Non-randomized algorithms are *deterministic;* so far we have only studied deterministic algorithms. *Randomization* is the practice of designing randomized algorithms, and is most appropriate when

1. an algorithm repeatedly chooses among alternatives,
2. finding the optimal alternative would be expensive (slow), and
3. most of the alternatives are good (if not optimal).

A randomized algorithm handles this situation by making each choice *at random*. Intuitively, if most of the alternatives are good, then a random choice is likely to result in one of the good ones. When the algorithm is repeatedly choosing, occasional poor choice is very likely counterbalanced by the good ones. Under these circumstances a randomized algorithm can outperform a deterministic algorithm for the same problem, since the randomized algorithm makes fast random choices that are "good enough," whereas a similar deterministic algorithm would make slow perfect choices.

There are many examples of real-world processes that use randomness to avoid difficult decision-making processes. A *poll* is one example. The goal of a poll is to determine the sentiment of a large constituency, such as the entire population of the United States. The exhaustive approach to this problem would be to contact every single resident and ask them what they think. This would certainly be accurate, but would be prohibitively expensive, and probably impossible due to a myriad of practical obstacles. Instead, pollsters choose a small subset of the population randomly, called a *sample;* perhaps a few hundred individuals out of a nation of over 300 million. The field of statistics tells us, perhaps counter-intuitively, that the average of such a small sample is overwhelmingly likely to be representative of the average of the sample space.

When an algorithm makes random choices, some aspect of its behavior involves an element of

chance. So far we have studied deterministic algorithms, whose correctness and efficiency can be proven unequivocally. We cannot be so sure about randomized algorithms, whose correctness and/or efficiency may involve random variables. (We insist that randomized algorithms, like their deterministic counterparts, always terminate.) Randomized algorithms are categorized according to which aspect is uncertain; the categories are named after cities that are famous as gambling destinations.

1. A *Monte Carlo* algorithm is *certainly efficient* but only *probably correct.*

2. A *Las Vegas* algorithm is only *probably efficient* but is *certainly correct.*

Example 48. *John Wayne airport in Orange County, California has a curfew that prohibits airplanes from landing after 10 PM. As a rule, airplanes arriving after 10 PM are diverted to another nearby airport to land, and passengers must make arrangements to get back to Orange County using ground transportation. We could model this situation mathematically by saying that any flight scheduled to land around 10 PM lands at John Wayne with probability* 0.95 *(say), and lands at the nearby LAX airport with probability* .05*. The correctness of outcome is uncertain so we call this a* Monte Carlo *process.*

Example 49. *Many airlines publish statistics on how frequently each of their flights arrives on time. Suppose a flight from Los Angeles to Chicago is on time with probability* 0.97*, and always arrives at Chicago as expected. Here the outcome is certainly correct (travel from Los Angeles to Chicago) but the efficiency (travel time) is uncertain, so we call this a* Las Vegas *process.*

In most practical applications of computer algorithms we can be more flexible about efficiency than correctness. Consider the task of saving work in a text editor to a file on disk. A Monte Carlo algorithm for this problem would occasionally function improperly; every once in a while, it might corrupt some of the text, or fail to save any data, or otherwise misbehave. A Las Vegas algorithm would always work properly, but once in a while might take more resources than usual. The Las Vegas algorithm is clearly preferable for this application. End users are (understandably) intolerant of programs that occasionally destroy their work, and for better or worse are accustomed to occasional delays. So the scope of Monte Carlo algorithms is limited to the few problem domains where probabilistic correctness is acceptable, while Las Vegas algorithms are applicable to almost all problem domains.

Randomization is a trade-off: it can help make algorithms more efficient (either in terms of efficiency classes or constant factors) or simpler, but by definition makes their behavior somewhat unpredictable. Since the relative importance of efficiency, simplicity, and predictability vary from application to application, we welcome both kinds of algorithms. Randomized algorithms may be appropriate as general-purpose solutions, but may be inappropriate for situations where predictability is paramount, such as *real-time systems* which are subject to hard resource constraints.

Random Operation	Pseudocode	Time Complexity
Return a random integer $x, a \leq x \leq b$	`x = random.randint(a, b)`	$O(1)$
Return a random element from V	`random.choice(V)`	$O(1)$
Shuffle V to a random permutation in-place	`random.shuffle(V)`	$O(n)$

Table 9.1: Random Operations

9.2 Generating Random Numbers

A truly *random number* is entirely unpredictable. True random numbers can be obtained by observing chaotic physical processes, such as the roll of dice, or small fluctuations in ambient acoustic noise or temperature. There are computer hardware devices to produce true random number generators, but they are uncommon and produce randomness at a slow rate. In general purpose algorithms and software we must often satisfy ourselves with *pseudo-random numbers*, which are generated with a deterministic algorithm but are "random-enough" for practical applications. The issue of generating high-quality pseudo-random numbers is central to the field of cryptography which is beyond the scope of this text. For the present purpose of designing randomized algorithms, we will presume that high-quality pseudo-random number generator functions are available to us without peering into the details. For the sake of brevity, we will use the word "random" to refer to both truly random and pseudo-random variables.

We will introduce randomness into our algorithms by using the following sub-algorithms. These algorithms and their presumed time complexity are summarized in Table 9.1.

1. *Random integer:* Given integers a and b with $a < b$, return a random integer x in the range $a \leq x \leq b$ such that $0 \leq x < n$, where every outcome is equally likely.
2. *Random element:* Given a non-empty vector V, return an element x from V, where each element is equally likely to be selected.
3. *Random shuffle:* Given a vector V, re-arrange V in-place to a new random permutation, where every possible permutation is equally likely.

9.3 The Monte Carlo Pattern

The exhaustive search and optimization patterns of Chapter 7 can be modified slightly to become Monte Carlo patterns. The exhaustive search pattern involves generating and validating every candidate solution. Often the sheer number of candidates makes these algorithms impractically slow. However, rather than generating *every* candidate exhaustively, we can apply the idea of randomization and generate *only a few random candidates.*

```
def monte_carlo_search(instance):
  for i in range(<NUMBER OF ITERATIONS>):
    candidate = <GENERATE ONE RANDOM CANDIDATE>(instance)
    if <VERIFIER>(instance, candidate):
      return candidate
  return None
```

Here the algorithm samples a set number of random candidates rather than exhaustively generating all possible candidates. This pattern is easily *tunable*. If we select a large <NUMBER OF ITERATIONS> we get an algorithm that tries very many candidates, and is likely (though not guaranteed) to find a nearly optimal solution. Alternatively if we select a small <NUMBER OF ITERATIONS> , we get a faster algorithm that is less likely to produce good outputs.

We can make the same sort of change to the exhaustive optimization pattern.

```
def monte_carlo_optimize(instance):
  best = None
  for i in range(<NUMBER OF ITERATIONS>):
    candidate = <GENERATE ONE RANDOM CANDIDATE>(instance)
    if <VERIFIER>(instance, candidate):
      if best is None or <candidate IS BETTER THAN best>:
        best = candidate
  return best
```

For example, we could adapt this pattern to the traveling salesperson problem first introduced in Section 7.8.

```
def monte_carlo_tsp(G):
  best = None
  for i in range(1000):
    candidate = <GENERATE A RANDOM CYCLE>
    if verify_tsp(G, cycle):
      if best is None or total_weight(cycle) < total_weight(best):
        best = cycle
  return best
```

We chose an arbitrary value of 1000 for <NUMBER OF ITERATIONS> .

In this problem each candidate is a sequence of vertices with the potential to be a cycle. Recall that a cycle is a sequence of vertices that starts and ends with the same vertex. Further recall that

the traveling salesperson problem requires that a cycle visit each vertex in G exactly once. So we can `<GENERATE A RANDOM CYCLE>` by shuffling G's vertex list, and then adding the first vertex to the back of the sequence.

```
def monte_carlo_tsp(G):
  for i in range(1000):
    cycle = copy of G.vertex_list()
    random.shuffle(cycle)
    cycle.add_back(cycle[0])
    if verify_tsp(G, cycle):
      if best is None or total_weight(cycle) < total_weight(best):
        best = cycle
  return best
```

In some ways this algorithm is superior to the exhaustive search algorithm we developed earlier, and in other ways it is inferior. The Monte Carlo algorithm is certainly faster.

Lemma 40. *The time complexity of the* `monte_carlo_tsp` *algorithm is* $O(n)$.

Proof. The `for` loop repeats 1000 times, which is a constant, albeit a large one. The `copy of G.vertex_list()` operation takes $O(n)$ time. Shuffling `path` takes $O(n)$ time; adding `cycle[0]` takes $O(1)$ time; and as you may recall, `verify_tsp` and `total_weight` each take $O(n)$ time. So the total time complexity of `monte_carlo_tsp` is

$$t(n) = O(1000(n + n + 1 + n)) = O(n).$$

□

This is undeniably better than the exhaustive search algorithm's time complexity of $O(n! \cdot n^2)$. The drawback is that the Monte Carlo algorithm is not certain to produce a correct output. All we can say is that it tries 1000 random permutations and returns the lowest cost Hamiltonian cycle among those permutations. While unlikely, it is possible for the algorithm to be very unlucky and generate only the 1000 worst permutations. So, is the Monte Carlo algorithm certain to return an optimal output? *No.* Can we say that, if G contains at least one Hamiltonian cycle, the algorithm will find it? *No.* Can we say *anything* rigorous about the quality of the algorithm's output? Not really. All we can say is that it tries 1000 random permutations and returns the best cycle among them. While these results are rather unsatisfying, this algorithm may nonetheless be useful for applications where time is scarce and merely making a "best effort" is acceptable.

9.4 The Las Vegas Pattern

We could say that the Monte Carlo pattern gambles with the correctness of its output. By contrast, the Las Vegas pattern gambles with the resources (time and space) it consumes. An algorithm fits the Las Vegas pattern when it uses randomness to guide decisions that impact the efficiency of the algorithm, but not its correctness.

Unlike other patterns, the Las Vegas pattern is not a template that can be filled in to become a complete algorithm. Instead, it is a design approach that may be used to fill in particularly challenging parts of other patterns. So the Las Vegas pattern is always used in conjunction with some other pattern.

The time and space complexity of a Las Vegas algorithm may be defined probabilistically.

Definition 42. *The* expected time complexity *of a randomized algorithm is the expected value of the algorithm's worst case time complexity, taken over random variable(s) defined by the random choices made by the algorithm. The* (deterministic) worst case time complexity *of a randomized algorithm is the greatest worst case time complexity among all values its random variables might take. The* expected space complexity *and* (deterministic) worst case space complexity *of a randomized algorithm are defined analogously.*

To illustrate these definitions, consider the following contrived algorithm.

```
def slumber(n):
  x = 0
  for i in random.randint(1, 2*n):
    if n > 10:
      x = i
  return x
```

Let $X \in [1, 2n]$ be a random variable corresponding to the value returned by `random.randint(1, 2*n)`. In the best case $n \leq 10$, the `x = i` statement never executes, so the body of the loop takes 2 steps and the best case time complexity of the algorithm is

$$1 + X \cdot 2 + 1.$$

In the worst case $n > 10$ and the `x = i` statement executes every time, so the body of the loop takes 3 steps and the worst case time complexity becomes

$$t(n) = 1 + X \cdot 3 + 1 = 3X + 2.$$

Note that this expression is random since X is a random variable. The expected time complexity

of the algorithm is

$$\begin{aligned}
\mathbb{E}[t(n)] &= \mathbb{E}[3X + 2] \\
&= 3 \cdot \mathbb{E}[X] + \mathbb{E}[2] \text{ (linearity of expectation)} \\
&= 3 \cdot \mathbb{E}[X] + 2 \text{ (constant expectation)} \\
&= 3\big(\frac{1}{2}(1 + 2n)\big) + 2 \text{ (expected value of } \texttt{randint(1, 2*n)} \text{)} \\
&= 3n + \frac{7}{2}.
\end{aligned}$$

The worst case time complexity of the algorithm occurs when $X = 2n$, and is

$$t(n) = 1 + (2n) \cdot 3 + 1 = 6n + 2.$$

Observe that the worst case time complexity is greater than the expected time complexity.

Corollary 4. *The time and space complexity of any deterministic algorithm is non-random, so its expected time and space complexities are equal to its worst case time and space complexities, respectively.*

Example 50. *The sequential search algorithm is deterministic with $O(n)$ worst case time complexity, so its expected time complexity is also $O(n)$.*

9.5 Quick Sort

Quick sort [28] [45] is a fast algorithm that solves the sorting problem. It follows the decrease-by-half pattern, so bears some resemblance to merge sort. However it also incorporates randomization, which allows quick sort to be both simpler, and often faster, than merge sort. Quick sort is called "quick" because it combines the best efficiency-related features of both merge sort and in-place selection sort. Like merge sort, its time efficiency is best described as $O(n \log n)$; like selection sort, it can be implemented as an in-place algorithm that reuses the input list data structure rather than allocating fresh lists. The drawback is that quick sort is a Las Vegas algorithm, so while we can say that its expected time complexity is $O(n \log n)$, we must concede that its deterministic worst case time complexity is as much as $O(n^2)$.

9.5.1 Deterministic Quick Sort

We first derive a deterministic version of quick sort, then improve upon it through randomization. Recall our second draft of merge sort.

```
def merge_sort(V):
  if len(V) <= 1:
    return V
  else:
    L, R = <DIVIDE V INTO TWO INSTANCES OF ROUGHLY EQUAL SIZE>
    sorted_L = merge_sort(L)
    sorted_R = merge_sort(R)
    sorted_V = <COMBINE sorted_L WITH sorted_R>
    return sorted_V
```

At this point in our design of merge sort we had started with the decrease-by-half pattern and handled the base case based on the observation that lists of length $n \leq 1$ are trivially sorted. Our next move was to fill in the `<DIVIDE V INTO TWO INSTANCES OF ROUGHLY EQUAL SIZE>` blank by slicing `V` in half. The slicing operation was straightforward, but then the `<COMBINE sorted_L WITH sorted_R>` developed into the rather hairy merge sub-algorithm.

There has got to be an easier way! Is there an alternative approach to the `<DIVIDE...>` part, one that might make for a simpler and faster `<COMBINE...>` part? Recall that a vector `V` is made up of elements `V[i]`. Each such `V[i]` may be thought of as an association between two pieces of data, a position `i` and the list element `V[i]` stored at that position. In merge sort we divided `V` *by position;* all the low-index elements went into `L` and all the high-index elements went into `R`. Is there another way that we might have divided `V`?

The alternative approach is to divide `V` *by value.* Where merge sort divided `V` into low-index and high-index sub-lists, we now divide `V` into *low-value* and *high-value* sub-lists. This is the crucial distinction between merge sort and quick sort. The following pseudocode captures that idea, and identifiers are renamed accordingly.

```
def deterministic_quick_sort(V):
  if len(V) <= 1:
    return V
  else:
    L = <LOW-VALUED ELEMENTS OF V>
    R = <HIGH-VALUED ELEMENTS OF V>
    sorted_A = deterministic_quick_sort(L)
    sorted_B = deterministic_quick_sort(R)
    sorted_V = <COMBINE sorted_L WITH sorted_R>
    return sorted_V
```

How can we tell whether each element `x` of `V` is "low-valued" or "high-valued"? We cannot choose some arbitrary threshold such as 100, because an instance might be comprised entirely of

values greater than 100, in which case every element would be categorized as "high-valued" and the sub-lists would not be of roughly equal length. Our first solution to this obstacle is to pick an arbitrary element of `V`, such as `V[0]`, to serve as the dividing line between categories. When an element of the input vector is chosen as the boundary between sub-lists, it is called a *pivot* value. For any pivot p and arbitrary element x from `V`, either $x < p, x = p,$ or $x > p$, so it is natural to form *three* sub-lists corresponding to these categories.

Definition 43. *In the quick sort algorithm, a* pivot *is an element of* `V` *used to divide* `V` *into three sub-lists:*

1. *those elements strictly* less than *the pivot,*
2. *those elements* equal to *the pivot, and*
3. *those elements strictly* greater than *the pivot.*

We call these sub-lists `LT`, `EQ`, and `GT` respectively. If we divide the input into three sub-lists, then at first it seems that we have a decrease-by-third algorithm that should recur three times.

```
def deterministic_quick_sort(V):
  if len(V) <= 1:
    return V
  else:
    pivot = V[0]
    create empty vectors LT, EQ, GT
    for x in V:
        if x < pivot:
            LT.add_back(x)
        elif x == pivot:
            EQ.add_back(x)
        else:
            GT.add_back(x)

    sorted_LT = deterministic_quick_sort(LT)
    sorted_EQ = deterministic_quick_sort(EQ)
    sorted_GT = deterministic_quick_sort(GT)
    sorted_V = <COMBINE sorted_LT, sorted_EQ, AND sorted_GT>
    return sorted_V
```

Actually, by construction all the elements of `EQ` are equal to one another, and are trivially in non-decreasing order. So there is no need to recursively sort `EQ`.

```
def deterministic_quick_sort(V):
  if len(V) <= 1:
    return V
  else:
    pivot = V[0]
    create empty vectors LT, EQ, GT
    for x in V:
        if x < pivot:
            LT.add_back(x)
        elif x == pivot:
            EQ.add_back(x)
        else:
            GT.add_back(x)

    sorted_LT = deterministic_quick_sort(LT)
    sorted_GT = deterministic_quick_sort(GT)
    sorted_V = <COMBINE sorted_LT, EQ, AND sorted_GT>
    return sorted_V
```

All that remains is the `<COMBINE...>` step. We could certainly `merge sorted_LT` with `EQ` in $O(n)$ time, then merge the resulting list with `sorted_GT` in additional $O(n)$ time. However that would be overkill. By construction every element of `LT` is less than every element of `EQ`, and every element of `EQ` is less than every element of `GT`. So this time we may combine the lists simply by concatenating them together.

Example 51. *Suppose* `V=[4, 6, 3, 0, 1, 2, 5, 7]`*; then* `pivot = 4`, `LT = [3, 0, 1, 2]`, `EQ = [4]`, *and* `GT = [6, 5, 7]`. *Supposing that* `quick_sort` *were complete and correct, the recursive calls would produce* `sorted_LT = [0, 1, 2, 3]` *and* `sorted_GT = [5, 6, 7]`. *Observe that every element in* `LT` *is less than every element in* `EQ`*; and similarly that every element in* `EQ` *is less than every element in* `GT`. *Concatenating* `LT + EQ + GT` *produces the correct output* `[0, 1, 2, 3, 4, 5, 6, 7]`.

Lemma 41. *For arbitrary elements* $l \in$ `LT`, $e \in$ `EQ`, *and* $g \in$ `GT`,

$$l < e < g.$$

Proof. By construction, `LT` contains those elements of `L` that are less than `pivot`, `EQ` contains those equal to `pivot`, and `GT` contains those greater than `pivot`. So for arbitrary l

$$l < \texttt{pivot},$$

for arbitrary e

$$e = \texttt{pivot},$$

and for arbitrary g

$$\texttt{pivot} < g.$$

By the transitivity of the $<$ and $=$ relations,

$$l < \texttt{pivot} = e = \texttt{pivot} < g$$

or

$$l < e < g.$$

□

We can now clarify the `<COMBINE...>` step.

```
def deterministic_quick_sort(V):
  if len(V) <= 1:
    return V
  else:
    pivot = V[0]
    create empty vectors LT, EQ, GT
    for x in V:
        if x < pivot:
            LT.add_back(x)
        elif x == pivot:
            EQ.add_back(x)
        else:
            GT.add_back(x)

    sorted_LT = deterministic_quick_sort(LT)
    sorted_GT = deterministic_quick_sort(GT)
    sorted_V = concatenate sorted_LT + EQ + sorted_GT
    return sorted_V
```

Next we eliminate the single-use variables `sorted_LT`, `sorted_GT`, and `sorted_V`.

```
def deterministic_quick_sort(V):
  if len(V) <= 1:
    return V
  else:
    pivot = V[0]
    create empty vectors LT, EQ, GT
    for x in V:
        if x < pivot:
            LT.add_back(x)
        elif x == pivot:
            EQ.add_back(x)
        else:
            GT.add_back(x)
    return concatenation of (deterministic_quick_sort(LT) +
                             EQ +
                             deterministic_quick_sort(GT))
```

Claim 12. *The* `deterministic_quick_sort` *algorithm is correct.*

While `deterministic_quick_sort` may be correct, its worst case time efficiency is mediocre.

Lemma 42. *The worst case time complexity of* `deterministic_quick_sort` *is* $O(n^2)$.

Proof. In the base cases $n = 0$ and $n = 1$, the algorithm executes

$$\begin{aligned} T(1) &= 2 \\ T(0) &= 2 \end{aligned}$$

steps. In the recursive case, the `if` takes 1 step, initializing `pivot` takes 1 step, creating the empty vectors `LT`, `EQ`, and `GT` take 3 steps, the `for` loop takes $O(n)$ time, there are the recursive calls `deterministic_quick_sort(LT)` and `deterministic_quick_sort(GT)`, and concatenating the returned list together takes n steps, for a total of

$$\begin{aligned} T(n) &= 1 + 1 + 3 + n + T(|\texttt{ LT }|) + T(|\texttt{ GT }|) + n \\ &= T(|\texttt{ LT }|) + T(|\texttt{ GT }|) + 2n + 5. \end{aligned} \tag{9.1}$$

Suppose `V` is in increasing order. Then `V[0]` is the least element of `V`, `LT` is empty, `EQ` contains only `pivot`, and `GT` contains all $n - 1$ elements aside from `pivot`. In this scenario

$$\begin{aligned} T(n) &= T(|\texttt{ LT }|) + T(|\texttt{ GT }|) + 2n + 5 \\ &= T(0) + T(n-1) + 2n + 5 \\ &= (2) + T(n-1) + 2n + 5 \\ &= T(n-1) + 2n + 7 \\ T(1) &= 2 \\ T(0) &= 2. \end{aligned}$$

This recurrence may be rewritten in closed form as

$$\begin{aligned} T(n) &= \sum_{i=2}^{n}(2i+7) + T(1) + T(0) \\ &= 2\sum_{i=2}^{n} i + \sum_{i=2}^{n} 7 + (2) + (2) \\ &= 2(\sum_{i=1}^{n} i - 1) + \sum_{i=2}^{n} 7 + 4 \\ &= 2\sum_{i=1}^{n} i + \sum_{i=2}^{n} 7 + 2. \end{aligned}$$

Applying the linear and triangular sum formulas, we obtain

$$\begin{aligned} T(n) &= 2(\frac{1}{2}n(n+1)) + [(n) - (2) + 1] \cdot 7 + 2 \\ &= n^2 + n + 7n - 7 + 2 \\ &= n^2 + 8n - 5 \end{aligned}$$

and $T(n) \in O(n^2)$ by properties of O. □

This is surprising, since quick sort's name includes the word "quick," and yet merge sort's time complexity is significantly faster at $O(n \log n)$.

9.5.2 Randomized Quick Sort

Let us consider how we might make this $O(n^2)$-time scenario less likely to occur. Quick sort's weakness is that `pivot` might happen to be the very least element of `V`, in which case $|$ `LT` $| = 0$ and $|$ `GT` $| = n - 1$. There is also the symmetric case when `pivot` happens to be the greatest element of `V`, $|$ `LT` $| = n - 1$, and $|$ `GT` $| = 0$. In these cases our division process fails to divide `V` into two sub-lists of "roughly equal size;" the sub-lists are unbalanced to an extreme. The decrease-by-half pattern has devolved into a decrease-by-one pattern and lost its efficiency edge.

What would the optimal pivot value be? An optimal `pivot` element would yield

$$|\texttt{LT}| \approx |\texttt{GT}|.$$

Such a pivot would be near the "middle" of the sorted sequence, or more precisely would be the *median* of `V`. If `V` were guaranteed to be in order, the median would be around index $\lfloor n/2 \rfloor$ of `V`. However, that observation is not immediately useful since the whole point of this algorithm is to take an *unordered* list and bring it into order. If `V` were already in order we wouldn't need to solve the sorting problem.

We have here a dilemma: how can we pick a `pivot` that will divide `V` roughly in half? When we pick an arbitrary element such as `V[0]` or `V[n-1]`, the algorithm has a mediocre $O(n^2)$ time

complexity for some particular ordering of input list `V`. We could use merge sort to sort `V`, then use an element near the middle of the list as a pivot; but at that point, having invoked merge sort, we might as well forget about quick sort and just return the output from merge sort.

This is the moment where we call upon the mysterious powers of randomization. Rather than working hard to find an *optimal* pivot, which would be too slow, we simply pick a pivot *randomly,* yielding a non-deterministic algorithm.

```
def randomized_quick_sort(V):
  if len(V) <= 1:
    return V
  else:
    pivot = random.choice(V)
    create empty vectors LT, EQ, GT
    for x in V:
        if x < pivot:
            LT.add_back(x)
        elif x == pivot:
            EQ.add_back(x)
        else:
            GT.add_back(x)
    return concatenation of (randomized_quick_sort(LT) +
                             EQ +
                             randomized_quick_sort(GT))
```

The only difference between the pseudocode for `randomized_quick_sort` and our earlier `deterministic_quick_sort` is that the randomized algorithm picks `pivot = random.choice(V)`. This is our final draft of the randomized quick sort algorithm. Observe that picking `pivot` randomly instead of arbitrarily has no effect on the correctness of the algorithm.

Claim 13. *`randomized_quick_sort` is correct.*

9.5.3 Analysis of Randomized Quick Sort

This section works toward proving that quick sort takes $O(n \log n)$ expected time. This is not an easy result; we will first work out a precise definition for the expected value $\mathbb{E}T(n)$ of the worst-case time complexity of the algorithm, and once that is established, prove the complexity class of $\mathbb{E}T(N)$.

Lemma 43. *The expected time complexity* $\mathbb{E}[T(n)]$ *of quick sort is*

$$\begin{aligned}\mathbb{E}[T(n)] &= \frac{2}{n}\Big(\sum_{i=0}^{n-1} T(i)\Big) + 4n + 3\\ \mathbb{E}[T(1)] &= 2\\ \mathbb{E}[T(0)] &= 2.\end{aligned}$$

Proof. In Lemma 42 we derived the recurrence relation (9.1),

$$\begin{aligned}T(n) &= T(|\texttt{ LT }|) + T(|\texttt{ GT }|) + 2n + 5\\ T(1) &= 2\\ T(0) &= 2\end{aligned}$$

for the time complexity of quick sort. By constant expectation, the expected values of the base cases are

$$\begin{aligned}\mathbb{E}[T(1)] &= \mathbb{E}[2] = 2\\ \mathbb{E}[T(0)] &= \mathbb{E}[2] = 2.\end{aligned}$$

In the recursive case the lengths of `LT` and `GT` depend on the random choice of the pivot, which could be the least element in `V`, the greatest, or somewhere in between. Let r be a random variable representing the *rank* of the pivot, which is its index in a sorted permutation of `V`. So when the pivot happens to be the least element in `V`, $r = 0$; when the pivot happens to be the greatest element in `V`, $r = n - 1$; and when the pivot happens to be optimal, $r \approx \frac{n}{2}$. By definition r obeys $0 \leq r < n$.

Let p_i be the probability that $r = i$, and

$$T_r = T(|\texttt{ LT }|) + T(|\texttt{ GT }|) + 2n + 5$$

be the value of $T(n)$ for a particular value of r. Then by the definition of expected value,

$$\begin{aligned}\mathbb{E}[T(n)] &= p_0 \cdot T_0 + p_1 \cdot T_1 + \ldots + p_{n-1} \cdot T_{n-1}\\ &= \sum_{i=0}^{n-1} p_i \cdot T_i.\end{aligned}$$

Each element of `V` has an equal likelihood of being chosen as a pivot, so every $p_i = \frac{1}{n}$ and by substitution

$$\mathbb{E}[T(n)] = \sum_{i=0}^{n-1}(\frac{1}{n})T_i = \frac{1}{n}\sum_{i=0}^{n-1} T_i. \tag{9.2}$$

To simplify this equation further we need a closed-form definition for T_r. The number of elements in `LT`, `EQ`, and `GT` depend on the value of r and the number of elements in `V` that happen to

be equal to the pivot. In the worst case the pivot is distinct from all other elements and $|\,\texttt{EQ}\,| = 1$. So for any arbitrary random choice of r, in the worst case

$$\begin{aligned} |\,\texttt{LT}\,| &= r \\ |\,\texttt{GT}\,| &= n - r - 1 \end{aligned}$$

so

$$T_r = T(r) + T(n - r - 1) + 2n + 5.$$

Substituting this definition into Equation 9.2,

$$\mathbb{E}[t(n)] = \frac{1}{n}\sum_{i=0}^{n-1}[T(i) + T(n - i - 1) + 2n + 5],$$

and distributing the sum we obtain

$$\mathbb{E}[T(n)] \;=\; \Big(\frac{1}{n}\sum_{i=0}^{n-1}T(i)\Big) + \Big(\frac{1}{n}\sum_{i=0}^{n-1}T(n - i - 1)\Big) + \Big(\frac{1}{n}\sum_{i=0}^{n-1}(2n + 5)\Big).$$

The sum expression

$$\sum_{i=0}^{n-1}T(i) = T(0) + T(1) + \ldots + T(n - 2) + T(n - 1)$$

and similarly

$$\sum_{i=0}^{n-1}T(n - i - 1) = T(n - 1) + T(n - 2) + \ldots + T(1) + T(0),$$

so these expressions add up to the same sum; their only difference is that they add the terms in opposite orders. Collecting like terms,

$$\mathbb{E}[T(n)] = \Big(\frac{2}{n}\sum_{i=0}^{n-1}T(i)\Big) + \Big(\frac{1}{n}\sum_{i=0}^{n-1}(2n + 5)\Big).$$

The quantity $2n + 5$ is constant with respect to i, so by the linear sum formula

$$\frac{1}{n}\sum_{i=0}^{n-1}(2n + 5) = \frac{1}{n}[(n - 1) - (0) + 1] \cdot (2n + 5) = \frac{1}{n}[n](2n + 5) = 2n + 5.$$

Altogether we have

$$\mathbb{E}[T(n)] = \frac{2}{n}\Big(\sum_{i=0}^{n-1}T(i)\Big) + 2n + 5.$$

□

This recurrence is not in master form, so we cannot apply the master theorem. Instead we prove an efficiency bound for $T(n)$ by strong induction. To do that we will use the *approximation by definite integrals* method of bounding summations, which you may recall from the integral calculus.

Theorem 5 (Approximation by definite integrals)**.** *For any increasing function $f(x)$ and integers $a \leq b$,*

$$\sum_{i=a}^{b} f(i) \leq \int_{x=a}^{b+1} f(x)\,\partial x.$$

We are now ready to prove that our $\mathbb{E}[T(n)]$ is upper bounded by $c \cdot n \ln n$ for some constant c [29]. We use $\ln n \equiv \log_e n$ instead of the more familiar $\log_2 n$ to make it easier to take an integral.

Lemma 44. *Let $\mathbb{E}[T(n)]$ be defined as in Lemma 43; then there exists some constant $c > 1$ such that*

$$\mathbb{E}[T(n)] \leq c \cdot n \ln n + 4$$

for any integer $n \geq 2$.

Proof. We let $c = 7$ and argue by strong induction on n. (The constant 7 is the smallest whole number that allows the base case argument below to work properly.)

Base case. Suppose $n = 2$; then by the definition of $\mathbb{E}[T(n)]$,

$$\begin{aligned}
\mathbb{E}[T(n)] &= \frac{2}{n}\Big(\sum_{i=0}^{n-1} T(i)\Big) + 2n + 5 \\
&= \frac{2}{(2)}\Big(\sum_{i=0}^{(2)-1} T(i)\Big) + 2(2) + 5 \\
&= T(0) + T(1) + 9 \\
&= (2) + (2) + 9 \\
&= 13.
\end{aligned}$$

The quantity $c \cdot n \ln n + 4 = (7) \cdot (2) \ln(2) + 4 \approx 13.704$, which is greater than 13, so $\mathbb{E}[T(n)] < c \cdot n \ln n + 4$.

Inductive case. Suppose $n \geq 3$ and $T(n') \leq c \cdot n' \ln n'$ for all n' satisfying $2 \leq n' < n$. We need to prove that this implies $T(n) \leq c \cdot n \ln n$. By the definition of $\mathbb{E}[T(n)]$,

$$\begin{aligned}
\mathbb{E}[T(n)] &= \frac{2}{n}\Big(\sum_{i=0}^{n-1} T(i)\Big) + 2n + 5 \\
&= \frac{2}{n}\Big(\sum_{i=0}^{1} T(i)\Big) + \frac{2}{n}\Big(\sum_{i=2}^{n-1} T(i)\Big) + 2n + 5 \\
&= \frac{2}{n}T(0) + \frac{2}{n}T(1) + \frac{2}{n}\Big(\sum_{i=2}^{n-1} T(i)\Big) + 2n + 5 \\
&= \frac{2}{n}(2) + \frac{2}{n}(2) + \frac{2}{n}\Big(\sum_{i=2}^{n-1} T(i)\Big) + 2n + 5 \\
&= \frac{2}{n}\Big(\sum_{i=2}^{n-1} T(i)\Big) + 2n + 5 + \frac{8}{n}.
\end{aligned}$$

By assumption

$$\begin{aligned} n &\geq 3 \\ 1 &\geq \frac{3}{n} \\ 3 &> \frac{8}{n} \end{aligned}$$

so

$$\mathbb{E}[T(n)] < \frac{2}{n}\Big(\sum_{i=2}^{n-1} t(i)\Big) + 2n + 8.$$

The sum evaluates $T(i)$ for $i = 2, 3, \ldots, n-1$, so each $i < n$, by the inductive hypothesis each $T(i) \leq c \cdot i \ln i$, and

$$\begin{aligned} \mathbb{E}[T(n)] &< \frac{2}{n}\Big(\sum_{i=2}^{n-1} c \cdot i \ln i\Big) + 2n + 8 \\ &= \frac{2c}{n}\Big(\sum_{i=2}^{n-1} i \ln i\Big) + 2n + 8. \end{aligned}$$

The expression $i \ln i$ is an increasing function of i, so by Theorem 5

$$\sum_{i=2}^{n-1} i \ln i \leq \int_2^n i \ln i \, \partial i.$$

The integral of $x \ln x$ is[1]

$$\int x \ln x \, \partial x = \frac{1}{2}x^2 \ln x - \frac{1}{4}x^2$$

so

$$\begin{aligned} \sum_{i=2}^{n-1} i \ln i &\leq \frac{1}{2}i^2 \ln i - \frac{1}{4}i^2 \Big|_2^n \\ &= \Big(\frac{1}{2}(n)^2 \ln(n) - \frac{1}{4}(n)^2\Big) - \Big(\frac{1}{2}(2)^2 \ln(2) - \frac{1}{4}(2)^2\Big) \\ &= \frac{1}{2}n^2 \ln n - \frac{1}{4}n^2 - \big((2 \ln 2) - 1\big) \\ &< \frac{1}{2}n^2 \ln n - \frac{1}{4}n^2 \end{aligned}$$

since $\big((2 \ln 2) - 1\big) \approx 0.386 > 0$. By substitution

$$\begin{aligned} \mathbb{E}[T(n)] &< \frac{2c}{n}\Big(\frac{1}{2}n^2 \ln n - \frac{1}{4}n^2\Big) + 2n + 8 \\ &= \Big(c \cdot n \ln n - \frac{c}{2}n\Big) + 2n + 8 \\ &= c \cdot n \ln n - \frac{(12)}{2}n + 2n + 8 \\ &= c \cdot n \ln n - (4n - 8). \end{aligned}$$

[1] See the table of integrals in any calculus textbook.

Again by assumption

$$\begin{aligned} n &\geq 3 \\ 4n &\geq 12 \\ 4n - 8 &\geq 4 \end{aligned}$$

so

$$\mathbb{E}[t(n)] < c \cdot n \ln n + 4.$$

□

We summarize the efficiency of quick sort in a single theorem.

Theorem 6. *The expected time complexity of* `quick_sort` *is* $O(n \log n)$ *and its worst case time complexity is* $O(n^2)$.

Proof. The worst case complexity was proven in Lemma 42. Lemma 43 established a recurrence for the expected time complexity $\mathbb{E}[T(n)]$, and Lemma 44 proved that $\mathbb{E}[T(n)] \leq c \cdot n \ln n + 4$ for some constant $c > 0$. By properties of O we have $\mathbb{E}[T(n)] \in O(c \cdot n \ln n + 4) = O(n \ln n) = O(n \log n)$. □

We have seen two analyses of the randomized quick sort algorithm: we proved that the worst-case time complexity of the algorithm is $O(n^2)$, and also that its expected time complexity is $O(n \log n)$. These two results are not contradictory. The preceding theorem says that the expected value of the running time is $O(n \log n)$. So on average quick sort runs in $O(n \log n)$ time, where the average is taken out of all the possible pivot choices that the algorithm might make for a particular instance size of n. Of course that average includes some above-average and some below-average pivot choices, and therefore some above-average and below-average time complexity values. The worst of those below-average time complexities are all $O(n^2)$. In summary, quick sort typically takes $O(n \log n)$ time, but rarely could take as much as $O(n^2)$ time.

9.5.4 In-place Quick Sort

Like selection sort, quick sort can be modified to operate in-place, and thus avoid the cost of allocating fresh data structures to store its output. Converting our current non-in-place `quick_sort` algorithm to be in-place will involve ideas similar to those that we used to optimize selection sort, and interestingly, some of the ideas we used in developing binary search.

As with selection sort, our in-place quick sort will gradually rearrange the original input vector `V` until it is fully ordered. Since each recursive call is prohibited from returning a new vector object, instead it must ensure that a range of elements of `V` are properly sorted. As with binary search, we generalize our problem slightly to involve working over a specified range of indices rather than the entire list.

in-place range sorting
input: a vector `V` of n elements, start index s, and end index e, where $0 \leq s \leq e \leq n$
output: a guarantee that the elements `V[s]`...`V[e]` are in non-decreasing order

The following very-high-level pseudocode solves this problem and is based on non-in-place quick sort.

```
def inplace_quick_sort_range(V, s, e):
  n = e - s    # number of elements
  if n <= 1:
    pass # nothing needs to be done
  else:
    <MOVE LT INTO A RANGE OF INDICES>
    <MOVE EQ INTO A RANGE OF INDICES>
    <MOVE GT INTO A RANGE OF INDICES>
    inplace_quick_sort(V, <LT INDICES>)
    inplace_quick_sort(V, <GT INDICES>)
```

As with the non-in-place version of quick sort, there is no need to sort the `EQ` elements. Since the variable `n` is used only once, and the first branch of the `if` statement has no effect, we might as well eliminate them.

```
def inplace_quick_sort_range(L, s, e):
  if (e-s) > 1:
    <MOVE LT INTO A RANGE OF INDICES>
    <MOVE EQ INTO A RANGE OF INDICES>
    <MOVE GT INTO A RANGE OF INDICES>
    inplace_quick_sort(L, <LT INDICES>)
    inplace_quick_sort(L, <GT INDICES>)
```

In-Place Partitioning

Recall that the in-place selection sort algorithm maintained two conceptual zones within `V`: a sorted zone and an unsorted zone. We can take a similar approach here, and implement the `<MOVE LT...>` and `<MOVE GT...>` phases by establishing zones within `V` that correspond to the `LT`, `EQ`, and `GT` vectors in the non-in-place algorithm.

`LT` (< pivot)	`EQ`	`GT` (> pivot)

The process of rearranging a vector into distinct `LT`, `EQ`, and `GT` zones is called *partitioning*. This may be considered a problem of its own. The similarity of this problem's name to that of the set partition problem is an unfortunate coincidence.

in-place partition

input: `V`, `s`, and `e` as defined for in-place range sorting, and $(\texttt{e} - \texttt{s}) \geq 2$.
output: indices `lte` and `eqe` such that

1. $\texttt{s} \leq \texttt{lte} < \texttt{eqe} \leq \texttt{e}$;
2. every element in `V[s]`...`V[lte-1]` is less than `V[lte]`;
3. elements `V[lte]`...`V[eqe-1]` are all equal; and
4. every element in `V[eqe]`...`V[s-1]` is greater than `V[lte]`.

The variables `s` and `e` in the problem statement define a range of unsorted elements that are to be partitioned. We require this range to have at least two elements since ranges with fewer elements are trivially sorted, and these trivial cases would translate into tedious special cases for our partition algorithm. The return values `lte` and `eqe` represent the boundaries between the three zones; `lte` is the "less-than-end" index, and equivalently the start of the equal zone; and `eqe` is the "end-of-equal" zone, and also the start of the greater zone. `V[lte]` is the pivot element. Note that the index constraints allow for empty `LT` or `GT` zones, but ensure that the `EQ` zone is non-empty.

`LT` ($\leq$ `pivot`)	`EQ` ($=$ `pivot`)	`GT` ($>$ `pivot`)
`V[s]..V[lte-1]`	`V[lte]...V[eqe-1]`	`V[eqe]...V[e-1]`

Conceptually, solving this problem is not so hard: we need to find elements that are in the wrong zone, and move them into their proper zone. This is more complicated than was maintaining zones in selection sort, since now we have three zones to contend with, and their sizes are unpredictable. In-place selection sort grows the sorted zone by exactly one element in each loop iteration, so in that algorithm it is always clear where the next element ought to be located. Alas, in the in-place partition algorithm it is impossible to predict the precise size of the `LT`, `EQ`, and `GT` zones, since their sizes are unknowable until after a pivot has been selected and all the elements have been compared to that pivot. Since the sizes of the zones are unknown, we have no way of knowing where to position the `EQ` elements.

Partitioning into three zones all at once is awkward, but we know from our experience with selection sort that we *can* partition into two zones. So we can overcome the three-zone obstacle by computing the partition in *two passes.* In the first pass we rearrange the elements in our range into two zones: a correct `LT` zone containing elements less than the pivot, and a `GE` zone containing elements

greater than or equal to the pivot. After the first pass the `EQ` and `GT` elements may be intermingled in `GE`, so we are not done; but we have made progress by separating out the `LT` elements and determining their number.

LT (< pivot)	GE (≥ pivot)
V[s]...V[lte-1]	V[lte]...V[e-1]

At this point the algorithm has finished with the `LT` elements. In the second pass it separates `GE` into distinct `EQ` and `GT` zones.

	EQ (= pivot)	GT (> pivot)
	V[lte]...V[eqe-1]	V[eqe]...V[e-1]

Our first draft pseudocode is below.

```
def inplace_partition(V, s, e):
    pivot = V[random.randint(s, e-1)]
    < PARTITION V INTO LT AND GE, AND DETERMINE lte >
    < PARTITION GE INTO EQ AND GT, AND DETERMINE eqe >
    return lte, eqe
```

Each pass involves partitioning a range of elements into two distinct zones with a boundary between them; the boundary indices are `lte` and `eqe`. For now we focus on the first pass; the ideas we develop there will carry over to the second pass. As discussed above, at the onset we have no way of knowing what `lte` should be; we need to compute it while forming `LT` in the first pass.

We can do this with a greedy process that iteratively shrinks an *unpartitioned zone.* We introduce variables for `lte` and `ges` ("greater-equal-start"), so that `V[s]...V[lte-1]` is a correct `LT` zone and `V[ges]...V[e-1]` is a correct `GE` zone. At the beginning of the pass the entire range is unpartitioned, so `lte=s`, `ges=e`, and the ranges `V[s]...V[lte-1]` and `V[ges]...V[e-1]` are both empty.

unpartitioned V[s]...V[e-1]

After a few iterations, the first few elements of the range are established as a partial `LT` zone and the last few are established as a partial `GE` zone.

LT V[s]...V[lte-1]	unpartitioned V[lte]...V[ges-1]	GE V[ges]...V[e-1]

As the partitioning proceeds, the unpartitioned zone shrinks while the `LT` and `GE` zones grow.

`LT` `V[s]...V[lte-1]`	unpartitioned	`GE` `V[ges]...V[e-1]`

Eventually, when `lte = ges`, the unpartitioned zone is empty, `LT` and `GE` are correctly partitioned, and the first pass is complete.

`LT` `V[s]...V[lte-1]`	`GE` `V[ges]...V[e-1]`

We formalize these ideas as a correctness invariant.

Invariant 7. *At the beginning and end of each loop iteration in the first pass, every element in `V[s]...V[lte-1]` is less than `pivot`, and every element in `V[ges]...V[e-1]` is greater than or equal to `pivot`.*

Let us now hash out the details of how to shrink the unpartitioned zone one step at a time. Suppose we have picked a `pivot`, and the invariant holds so `V[s]...V[lte-1]` and `V[ges]...V[e-1]` are properly partitioned. Then

1. if `V[lte] < pivot` then `V[lte]` belongs in `LT`, so we can leave it alone and increment `lte`;

2. if `V[ges-1] >= pivot` then `V[ges-1]` belongs in `GE`, so we can leave it alone and decrement `ges`;

3. otherwise, `V[lte]` belongs in `GE` and `V[ges-1]` belongs in `LT`; both are on the wrong end of the range. We can solve both problems at once by swapping the two elements, and then may increment `lte`.

Our second draft pseudocode includes a clear first pass.

```
def inplace_partition(V, s, e):
    pivot = V[random.randint(s, e-1)]

    lte = s
    ges = e
    while lte < ges:
        if V[lte] < pivot:
            lte += 1
        elif V[ges-1] >= pivot:
            ges -= 1
        else:
            swap(V[lte], V[ges-1])
            lte += 1

    < PARTITION GE INTO EQ AND GT, AND DETERMINE eqe >

    return lte, eqe
```

The second pass is essentially the same process, except that we must rearrange `V[ges]...V[e-1]` into two distinct `EQ` and `GT` zones. The second pass is responsible for maintaining an invariant similar to that for the first pass.

Invariant 8. *At the beginning and end of each loop iteration in the second pass, every element in `V[lte]...V[eqe-1]` is equal to `pivot`, and every element in `V[gts]...V[e-1]` is greater than `pivot`.*

Our final, complete pseudocode is below.

```
def inplace_quick_sort(V):
    return inplace_quick_sort_range(V, 0, len(V))

def inplace_quick_sort_range(V, s, e):
    if (e-s) >= 2:
        lte, eqe = inplace_partition(V, s, e)
        inplace_quick_sort(V, s, lte)
        inplace_quick_sort(V, eqe, e)
    return V

def inplace_partition(V, s, e):
    pivot = V[random.randint(s, e-1)]

    lte = s
    ges = e
    while lte < ges:
        if V[lte] < pivot:
            lte += 1
        elif V[ges-1] >= pivot:
            ges -= 1
        else:
            swap(V[lte], V[ges-1])
            lte += 1

    eqe = lte
    for i in range(ges, e):
        if V[i] == pivot:
            swap(V[eqe], V[i])
            eqe += 1

    return lte, eqe
```

Lemma 45. *The expected time complexity of* `inplace_quick_sort` *is* $O(n \log n)$*, while its worst case time complexity is* $O(n^2)$*.*

Proof. Clearly the time complexity of `inplace_quick_sort` is dominated by that of `inplace_quick_sort_range`. Let $n = e - s$ be the number of elements sorted by `inplace_quick_sort_range`. When $n \leq 1$ the algorithm executes only 1 step deciding not to do anything else, so we have

$$\begin{aligned} T(1) &= 1 \\ T(0) &= 1. \end{aligned}$$

In the non-base case, `inplace_quick_sort_range` first spends 1 step on the `if` statement. It then calls `inplace_partition`.

The runtime of `inplace_partition` is dominated by its two loops. The `while` loop iterates at most n times: it repeats only as long as `lte` is less than `ges`, these variables start n apart, and each iteration of the loop brings them one step closer by either incrementing `lte` or decrementing `ges`. Each iteration of the `while` loop takes $O(1)$ time, so the loop takes a total of $O(n)$ time. Likewise the `for` loop takes $O(n)$ time because it loops through the `GE` zone, which has at most n elements in the worst case when `LT` is empty, and spends $O(1)$ time per iteration. Aside from these loops the other steps take $O(1)$ time. So the total time spent in `inplace_partition` is $O(n + n + 1) = O(n)$.

Returning to the analysis of `inplace_quick_sort_range`, after the partitioning phase, the algorithm calls itself recursively on the sub-ranges `LT` and `GT`. So the total number of steps in the recursive case is

$$\begin{aligned} T(n) &= 1 + n + T(|\,\texttt{LT}\,|) + T(|\,\texttt{GT}\,|) \\ &= T(|\,\texttt{LT}\,|) + T(|\,\texttt{GT}\,|) + n + 1. \end{aligned}$$

Aside from constant factors, this recurrence is the same as the one we derived for `quick_sort` in Lemma 42. Both algorithms pick pivots in the same manner, so by logic very similar to that of Lemmas 42 and 43, the expected time complexity of `inplace_quick_sort` is $O(n \log n)$ and its worst case time complexity is $O(n^2)$. □

9.5.5 Summary of Sorting Algorithms Covered so Far

The following table summarizes the properties of the three sorting algorithms we have studied.

algorithm	**expected time**	**worst case time**	**in-place?**
selection sort	$O(n^2)$	$O(n^2)$	yes
merge sort	$O(n \log n)$	$O(n \log n)$	no
quick sort	$O(n \log n)$	$O(n^2)$	yes

At this point selection sort seems obsolete. Of the three algorithms, merge sort has the best worst case time efficiency. Selection sort and quick sort both have $O(n^2)$ worst case complexity and are in-place, but quick sort is substantially faster in the expected case. So in applications where an ironclad worst case time bound is important, merge sort is the best choice; when practical efficiency or the in-place behavior is important, quick sort is the best choice.

Exercises

9-1. Sort the characters of the string "SEQUENCE" using non-in-place deterministic quick sort; show your work.

9-2. Sort the characters of the string "SEQUENCE" using in-place quick sort; show your work. Suppose that every pivot choice chooses the first element in the range, so that there will be one canonical solution.

9-3. Consider a version of quick sort that arbitrarily picks the middle element of `L` as a pivot instead of the first element or a random element, i.e. it assigns

```
pivot = V[floor(len(L) / 2)]
```

Give an example of an input `L` with $n = 7$ elements for which this version of quick sort experiences its worst case $O(n^2)$ time complexity. What is the general pattern of lists that make this version of quick sort perform poorly?

9-4. Do some Internet research into the "PHP hash table collision exploit." Describe this problem in your own words. How could it have been prevented?

9-5. Draw a diagram of a chained hash table with $m = 2^3 = 8$ chains, using the hash function

$$h(x) = (5x) \bmod 8,$$

after inserting each of the following integers: $1, 2, 3, 4, 5$.

9-6. Give an example of a programming scenario where a hash table is a better choice than a binary search tree, as well as an example of the opposite.

9-7. The C standard library includes a function called `qsort` that implements quick sort. Find and examine an open source implementation of `qsort`, such as the one in the BSD libc, GNU C Library, musl libc, or another. How closely does the implementation correspond with the pseudocode in this chapter? What significant changes, if any, are evident in the C implementation?

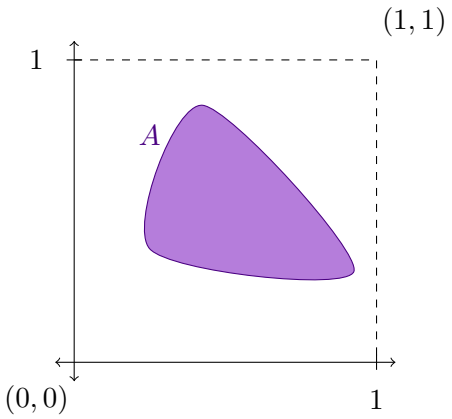
9-8. Design a collision-free hash function for strings containing the face value of a playing card: "2", "3", ..., "10", "J", "Q", "K", "A", "J".

9-9. For each of the following problems: design a Monte Carlo algorithm that solves the problem; describe your algorithm with clear pseudocode; and prove the time efficiency class of your algorithm. Since these algorithms are Monte Carlo, their output need not be perfectly correct, though you should strive to make them approximately correct.

(a)

median finding
input: a list L of $n > 0$ numbers **output:** the median m of `L`, i.e. a value m such that the number $n_L = \lvert\{x \in L \mid x < m\}\rvert$ of elements of L less than m, and the number $n_G = \lvert\{x \in L \mid x > m\}\rvert$ of elements of `L` greater than m, are equal

(b) *Area of an unknown region.* Suppose that you wish to approximate the area A of a *mystery region* in the Cartesian plane. You know that the region lies entirely within the unit square whose corners are $(0, 0)$ and $(1, 1)$, and have access to an `inside(x, y)` function that returns `True` if (x, y) is inside the region or `False` if (x, y) is outside. You have no other information about the location or shape of the region.



(c)

subset sum problem
input: a list X of n distinct integers, and a target integer k **output:** a subset $S \subseteq X$ such that $k = \sum_{x \in S} x$, or `None` if no such S exists

(d)

circuit satisfaction problem
input: a Boolean circuit C with k variables and n vertices **output:** a Boolean assignment A that sastisfies C, or `None` if no such assignment exists

9-10. *Three-way quick sort.* Design a variant of non-in-place quick sort that chooses *two* pivots and divides the input into *five* sub-lists. Write clear pseudocode for your algorithm. What is the expected length of each of the five lists, and what is the expected time complexity of your algorithm?

9-11.

mode finding
input: a list L of $n > 0$ orderable elements **output:** an element $x \in L$ and non-negative integer f, such that x is a most-frequently-appearing element of L that appears exactly f times

(a) Design a Las Vegas algorithm, based on non-in-place quick sort, for solving the mode finding problem. Analyze your algorithm. *Hints:* You can re-use the base case and partitioning parts of quick-sort, but need to rethink what happens when the recursive solutions get combined. Try to achieve an $O(n \log n)$ expected time complexity.

(b) Design a Las Vegas algorithm that uses hash tables to solve the same problem. Analyze your algorithm. *Hint:* Try to achieve a $O(n)$ expected time complexity.

9-12. *Code golf.* Re-implement each of the following algorithms with the shortest Python code you can manage. Whitespace doesn't count, so focus on meaningful simplifications rather than superficial formatting.

(a) `quick_sort`

(b) `inplace_partition`

Chapter 10

Reduction

10.1 The Big Idea

The *reduction* pattern involves using another known, pre-existing algorithm or data structure to do the "dirty work" of solving a problem. The idea is that, when we can, we ought to use known algorithms and data structures to solve problems rather than inventing new ones. Reduction is analogous to the idea of *reuse* in software engineering, which is the practice of writing and using modular library code rather than writing software from scratch.

10.1.1 Reduction to a Known Algorithm

One form of reduction involves using another known, off-the-shelf algorithm to solve a related problem. Some *pre-processing* may be necessary to prepare to use the known algorithm, as well as some *post-processing* necessary to convert its output. The spirit of the reduction pattern is that the pre- and post-processing phases are simple to design and efficient to execute; the hard work is done by the pre-existing algorithm.

```
def reduction_pattern(instance_of_A):
  instance_of_B = <PRE-PROCESS A>
  solution_to_B = solve_B(instance_of_B)
  solution_to_A = <POST-PROCESS solution_to_B>
  return solution_to_A
```

In many cases the pre-processing and post-processing steps are trivial. Consider the problem of finding the minimum and maximum element in a vector.

min and max
input: a list L of $n > 0$ comparable objects **output:** a pair (l, g) where l and g are the least and greatest elements in L, respectively

The following algorithm solves the minimum and maximum problem by reduction to sorting.

```
def min_and_max(V):
  sorted = merge_sort(V)
  l = sorted[0]
  g = sorted[len(sorted)-1]
  return l, g
```

The only post-processing is the two statements to initialize `l = sorted[0]` and `g = sorted[len(sorted) - 1]`.

These statements are based on the observation that the least element of a sorted vector is always at the first index, while the greatest element is always at the last index. This post-processing is necessary since the data type of the output of `merge_sort` is a vector, while the output of the minimum and maximum problem must be a pair of exactly two elements. This post-processing is trivial; no pre-processing is necessary in this example, since the data type of the input to both problems happens to be identical.

The algorithm clearly has $O(n \log n)$ time complexity since it executes merge sort and only $O(1)$ other steps. While not optimal (it is possible to solve minimum and maximum in $O(n)$ time), this algorithm is nonetheless reasonably efficient, and is remarkably concise.

Definition 44. *Problem A* reduces to *problem B if there exists an algorithm that solves A by pre-processing its input, solving B, then post-processing that output.*

Corollary 5. *Minimum-and-maximum reduces to sorting.*

Reducibility results establish a kind of *hardness* relationship between problems. Since minimum-and-maximum reduces to sorting, that means that sorting is the more versatile and general-purpose problem. In a sense, sorting is *harder* than minimum-and-maximum; or to be a bit more precise, minimum-and-maximum is no harder than sorting. These reduction-based hardness relationships are integral to the definition of NP-completeness, which we shall study soon.

10.1.2 Reduction to Data Structures

In some cases, the piece of technology that can solve an algorithmic obstacle is a pre-made data structure rather than a pre-made algorithm. In these cases we may replace a portion of the algorithm with data structure operations. The merge part of merge sort (Section 8.4) fits that mold. In each step of the merge operation the algorithm must find the least un-merged element. This could

certainly be accomplished in $O(n)$ time with a sequential search through both input vectors `L` and `R`. But instead the algorithm examines only the first element of each vector, which is correct since the vectors are surely sorted, and takes only $O(1)$ time which is significantly faster. We could say that the merge algorithm benefits from the sorted structure of the `L` and `R` vector.

We will see further examples of replacing algorithm steps with data structure operations in Sections 10.3 and 10.4.

10.2 Reduction to Sorting

One reason that we have studied sorting so comprehensively is because many problems reduce to sorting. This means that the efficiency of sorting algorithms dictates the efficiency of many other algorithms.

10.2.1 Median Finding

Consider the problem of finding the median of a collection of numbers.

median finding
input: a list L of $n > 0$ numbers **output:** the median m of `L`, i.e. a value m such that the number $n_L = \|\{x \in L \,\|\, x < m\}\|$ of elements of L less than m, and the number $n_G = \|\{x \in L \,\|\, x > m\}\|$ of elements of `L` greater than m, are equal

Intuitively, the median of a list of numbers in non-decreasing order is the middle element.

```
def median(V):
  sorted = merge_sort(V)
  return sorted[len(V) / 2]
```

This intuition is correct when n is odd, but is actually incorrect when n is even. When n is even, the median is defined as the mean of the two elements closest to the middle. So our reduction algorithm needs to decide which case we are in and handle each separately.

```
def median(V):
  sorted = merge_sort(V)
  n = len(V)
  mid = n / 2
  if n is odd:
    return sorted[mid]
  else:
    return (sorted[mid-1] + sorted[mid]) / 2
```

You may wish to confirm for yourself that the list indexing is correct in both cases. The algorithm clearly runs in $O(n \log n)$ time.

Corollary 6. *Median finding reduces to sorting.*

Once again we have obtained an algorithm that is concise, fast, and easy to reason about. This algorithm has no pre-processing steps; everything after the call to `merge` sort— is post-processing.

10.2.2 Set Intersection

Next we consider the problem of computing the intersection of two sets.

set intersection
input: two lists L and R, each representing a set of distinct elements **output:** a list S representing $L \cap R$

Reducing set intersection to sorting is more complex than the prior reductions. The result vector S needs to contain $L \cap R$, or in other words the elements that are present in both L and R. There is a straightforward algorithm that does this with two nested loops in $O(n^2)$ time, but we aim to do better.

Suppose for the sake of discussion that we *pre-sort* L and R before we try to intersect the sets.

```
def intersection(L, R):
  SL = merge_sort(L)
  SR = merge_sort(R)
  return <POST-PROCESS SL AND SR>
```

In the `<POST-PROCESS...>` phase we have two sorted vectors `SL` and `SR` that need to be combined into one vector `S` containing the intersection of `SL` and `SR`. This process is very similar to the merge phase of merge sort. The merge algorithm takes two sorted lists and produces a single sorted

list containing *all* of the elements among the input lists. Here we have two sorted lists and need to produce a single list containing the *elements common to both input lists.* This is very similar in spirit, but differs in a few key places.

```
def intersection(L, R):
  SL = merge_sort(L)
  SR = merge_sort(R)
  S = Vector()
  li = ri = 0
  while li < len(SL) and ri < len(SR):
    if SL[li] < SR[ri]:
      li += 1
    elif SR[ri] < SL[li]:
      ri += 1
    else:
      S.add_back(SL[li])
      li += 1
      ri += 1
  return S
```

Observe that this merge-like phase only retains elements that are common to both `SL` and `SR`; and that we may ignore any "straggler" elements left over after the `while` loop, since by definition they are not in $L \cap R$.

Claim 14. *The* `intersection` *algorithm is correct and its time complexity is* $O(n \log n)$.

Observe that, subjectively, most of the difficulty in achieving this $O(n \log n)$ efficiency lies in the `merge_sort` calls. Our set intersection algorithm benefits from all the hard work that went into that algorithm design, without duplicating that design effort. It is also true that those calls are responsible for most of the $O(n \log n)$ time complexity of the algorithm. By contrast, only $O(n)$ time, likely with a modest constant factor, is spent in the post-processing phase.

10.3 Reduction to Hash Table Operations

The *hash table* is a versatile and powerful data structure. Many problems reduce to hash table operations; in other words, sometimes a simple algorithm that dumps elements into a hash table and then reads them back out is actually highly efficient.

Hash Table Operation	Pseudocode	Time Complexity
Create an empty hash table	`ht = HashTable()`	$O(1)$
Get the number of entries in a tree	`len(ht)`	$O(1)$
Get an iterator for the entries in undefined order	`iter(ht)`	$O(1)$
Determine whether a hash table contains key k	`boolean=ht.contains(k)`	$O(1)$ expected
Get the entry associated with key k, or `None`	`entry = ht.search(k)`	$O(1)$ expected
Associate key k with value v; return `True` if a new key entry was created	`boolean=ht.insert(k,v)`	$O(1)$ exp. amort.
Remove the entry for k (if any); return `True` if an entry was actually removed	`ht.remove(k)`	$O(1)$ exp. amort.

Table 10.1: Hash Table Operations

10.3.1 Hash Table

Like the self-balancing binary search tree data structure (described in Section 4.7), a hash table stores a set of entry objects (described in Section 4.6). Each entry is a simple container object that stores a key and a value associated with that key. Each key in a hash table is unique. The essential hash table operations, their time complexity, and our pseudocode notation are given in Table 10.1.

A hash table may be thought of as a generalization of a vector. A vector of length n stores index-value associations, where each index must be between 0 and $n-1$ inclusive. We could think of each element of a vector as storing a key-value entry. In that case, a vector supports the same kind of operations as does a hash table, except that in the vector, keys are limited to integers in the range $[0, n)$. The vector data structure can search, insert, or remove the element at index/key i in $O(1)$ worst case time, but suffers from the severe restriction that indices/keys are limited to integers $0, \ldots, n-1$.

There are many practical situations in which we need to store key-value associations, but the keys do not happen to be well-behaved index integers. For example, a website might use username strings as keys and user account objects as values; an IPv6 Internet router uses 128-bit IP addresses as keys and network links as values; a Python interpreter uses class method names as keys and executable code as values; and so on. Obviously none of of these key types (username strings, 128-bit IP addresses, and Python method names) happen to be sets of vector indices, and yet still we need to use them as keys in a fast key-value data structure.

Just as there are several competing flavors of self-balancing binary search tree, there are several competing flavors of hash table. Perhaps the most widely known and implemented is the *chained hash table;* but there are many competitors to chaining including *linear probing, round-robin hashing, tabulation hashing,* and *cuckoo hashing.*

All hash table variants involve a large vector, where each element of the vector is a *bucket* which may contain an entry. In addition, the structure has a *hash function* `hash(k)` that takes a key object `k` and returns a bucket index `i`. In an ideal world, `hash` would map each key to its own distinct index, and that would be the end of it. Unfortunately it is not quite that easy; there is a distinct possibility that two different keys both map to the same index. Indeed, in the applications where hash tables are useful, the universe of keys is substantially larger than the set of bucket indices. For example, there exist 2^{128} IPv6 addresses, and yet an industrial-grade router might only dedicate 256 mb $= 2^{18}$ bytes to its routing hash table. In this case, the pigeonhole principle dictates that collisions exist, i.e. that two distinct keys might both have the same bucket index. So every hash table must somehow contend with these collisions.

The chained hash table manages collisions by storing a linked list of entry objects in each bucket. It maintains the invariant

`buckets[i]` = a linked list containing any entry whose key `k` has `hash(k)==i`.

With this invariant in place, the `search` operation involves computing an index with `hash(k)`, looking up the corresponding bucket, and a straightforward sequential search through the list elements.

```
def search(self, k):
    index = hash(k)
    for entry in self.buckets[index]:
        if entry.key == k:
            return entry
    return None
```

This operation clearly takes $O(\ell_i)$ time, where ℓ_i is the number of entries in bucket i. The `contains`, `insert`, and `remove` operations are similar. Each operation involves computing an index with `hash` and then looping through the entries in one bucket. `insert` also updates an entry or adds a new entry, and `remove` deletes one entry, but these operations take only $O(1)$ time so their time complexity is $O(\ell_i)$ each.

In order for these $O(\ell_i)$-time operations to be fast, we need each ℓ_i to be small. Let b be the number of buckets in the table; then the average number of entries in any bucket is

$$\cap\ell = \frac{n}{b}.$$

The data structure controls how many buckets exist, and has control over b; suppose we stipulate that the number of buckets is at least proportional to n, for example $b \geq 2n$. Then the average bucket length is

$$\widehat{\ell} = \frac{n}{b} \leq \frac{n}{(2n)} = \frac{1}{2}.$$

This fraction $\frac{1}{2}$ is in $O(1)$, so we are close to an efficient data structure. The problem is a small average bucket length by itself is not enough to ensure fast performance; the mean average says nothing about how the entries are distributed amongst buckets. Indeed, in the worst case the `hash` function maps every key to the same index i; in this case we have one bucket of length $l_i = n$, while each of the remaining $(n - 1)$ buckets have length 0. In this event the bucket vector is essentially useless; since only one bucket is in use, the hash table has devolved into an unsorted linked list with $O(n)$ access times.

Solving this problem is where randomization and the Las Vegas approach comes into play. Rather than using one hard-code `hash` function, instead we generate a random hash function every time a hash table is created. We need our hash function to do a reasonably good job of distributing entries evenly among bucket indices instead of piling them all into the same bucket. With care it is possible to generate a hash function with the property that, for any two distinct keys k_0 and k_1, the probability that `hash(` k_0 `) = hash(` k_1 `)` is in $O(\frac{1}{n})$. This implies that the expected value $\mathbb{E}[b_i]$ of the length of each bucket is only $O(1)$. It also implies that the probability of a worst-case scenario of some bucket having length $O(n)$ is vanishingly small.

Since a hash table's hash function is chosen randomly at runtime, entries are only *probably* evenly distributed, and we can only bound the maximum length of any bucket at $O(1)$ in a probabalistic, expected-value sense. That is why so many of the hash table operations have expected-time complexity classes. Any hash table operation that involves looping through the entries in a bucket takes $O(1)$ expected time.

There is one last detail to hash tables: maintaining the invariant that the number of buckets b is proportional to the current number of entries n. As entries are added and removed with the `insert` and `remove` operations, n fluctuates. Ordinarily a hash table will monitor the values of b and n, and if the table is over-full with $b < 2n$, *rehash* by creating a new vector twice as long as the current one, moving all the entries from the old smaller vector into the new larger vector, and then destroying the old vector. Likewise if the table is under-full with too many buckets b relative to the number of entries n, it rehashes and shrinks the bucket vector to half its current size. Just like the resizing operation for a vector, this operation takes $O(n)$ time but can only happen so infrequently that it adds only $O(1)$ amortized time to each `insert` or `remove` operation. This explains why some of the hash table operations' efficiency classes are qualified to take amortized time. Any operation that changes the number of entries may trigger a slow rehash, so those operations take $O(1)$ expected amortized time.

10.3.2 Set Intersection

To illustrate the usefulness of hash tables, we consider how to reduce the set intersection problem first introduced in Subsection 10.2.2 to hash table operations.

set intersection
input: two lists L and R, each representing a set of distinct elements **output:** a list S representing $L \cap R$

Recall that we already designed an algorithm that reduces set intersection to sorting and takes $O(n \log n)$ time.

Let us start with a dead-simple greedy algorithm for computing a set intersection. Suppose we loop through one of the two input sets L. The definition of set intersection says that, for any $x \in L$, x is in $X \cap L$ only when $x \in R$ as well.

```
def hash_intersection(L, R):
    S = Vector()
    for x in L:
        if x in R:
            S.add_back(x)
    return S
```

Evaluating the `x in R` expression inside the loop would involve a sequential search that takes $O(n)$ time, so this algorithm takes $O(n^2)$ total time, which is slow. How could we use a hash table to accelerate this step? This is precisely the kind of situation that hash tables are made for, we can simply `insert` each element of R into a hash table, then use the `contains` operation to test whether a given `x` is in the hash table. That way the $O(n)$-time sequential search through the vector R is replaced with a $O(1)$ expected time `contains` operation in a hash table.

```
def hash_intersection(L, R):
    R_hash = HashTable()
    for r in R:
        R_hash.insert(r, False) # the values are never used
    S = Vector()
    for x in L:
        if R_hash.contains(x):
            S.add_back(x)
    return S
```

Despite the facts that this pseudocode is longer and involves the sophisticated hash table data structure, this algorithm is actually very fast.

Lemma 46. *The `hash_intersection` algorithm takes $O(n)$ expected time.*

Proof. Let $n = |L| + |R|$ be the total number of elements between L and R. Creating the hash table takes $O(1)$ time. The first `for` loop repeats at most n times, each iteration performs one hash table `insert` operation which takes $O(1)$ expected time, so the total time complexity of this loop is $O(n \times 1) = O(n)$ expected time. Creating the vector takes only $O(1)$ time. The second `for` loop repeats at most n times, and each iteration performs one hash table `contains` operation which takes $O(1)$ expected time, so the loop also takes a total of $O(n \times 1) = O(n)$ expected time. Of

course the final `return` takes one step. So in total the algorithm takes $O(1+n+1+n+1) = O(n)$ expected time. □

10.4 Priority Search Queues; Revisiting Prim-Jarník and Dijkstra

In Chapter 6 we derived versions of the Prim-Jarník and Dijkstra algorithms with time complexities $O(mn)$ and $O(m+n^2)$, respectively. In both cases the bottleneck operation defining the algorithm's complexity was the search for a minimal edge. In the case of the Prim-Jarník algorithm, each loop iteration greedily chooses a bridge edge of minimum weight; in the case of Dijkstra's algorithm, each loop iteration chooses a bridge edge that completes a path of minimum weight. We can speed up these algorithms by using a special purpose *priority search queue* data structure to maintain the set of bridge edges; then these greedy choices reduce to a data structure operation to find the minimal bridge edge.

10.4.1 Priority Search Queue Operations

A priority search queue is an abstract data type that is a hybrid between a search tree and a priority queue. Each entry in a priority search queue associates an orderable key k with a numeric priority p. The data structure can manipulate key-priority entries according to their priority *or* their key. By contrast, regular priority queues (heaps) can only manipulate entries according to their priority. We describe two implementations of the priority search queue, one based on binary search trees, and another called a Fibonacci heap. The essential priority search queue operations, and the time complexities for each implementation, are shown below.

Priority search queue operation	**BSTs**	**Fib. Heap**
Create an empty priority search queue	$O(1)$	$O(1)$
Insert key k with priority p	$O(\log n)$	$O(1)$
Pop and return the key with minimum priority	$O(\log n)$	$O(\log n)$ amort.
Decrease priority of key k to p	$O(\log n)$	$O(1)$ amort.

10.4.2 Implementing a priority search queue with two search trees

Building a priority search queue out of self-balancing binary search trees is relatively simple, and results in a priority search queue data structure whose operations belong to good efficiency classes, with fair constant factors. This approach builds upon an existing self-balancing BST implementation with $O(\log n)$ find, insert, and remove operations. Each priority search queue includes two complete BSTs with a key-priority entry in each node. The first tree, called `by_priority` orders nodes by priority, while the second tree `by_key` orders nodes by key.

Creating a new queue involves creating two empty BSTs, which takes $O(1)$ time. Inserting a new key-priority entry involves two BST insertions, one into each tree, in $O(2\log n) = O(\log n)$ time.

The entry with least priority is at the leftmost node of `by_priority`, which may be found by following left pointers in that tree in $O(\log n)$ time; then deleting the entry from both trees involves two tree deletions, so the total time complexity of the pop-and-return operation is $O(3 \log n) = O(\log n)$. Decreasing the priority of k involves finding k's entry in `by_key` in $O(\log n)$ time, deleting the corresponding entries from both trees in $O(\log n)$ time each, and inserting a new entry in each tree, for a total of $O(5 \log n) = O(\log n)$ time.

10.4.3 Fibonacci Heaps

A *Fibonacci heap* [22] is a complex data structure that implements the priority search queue operations. Essentially, a Fibonacci heap is a forest (collection) of trees. Each tree is in heap order, which, among other things, means that the minimum-priority entry of each tree is at the root. The pointers to the root nodes are stored in a doubly linked list, so that a tree root may be inserted or deleted in $O(1)$ time. A lookup array is used to map an arbitrary key k to its corresponding node in the forest. Finally, there is a pointer to the overall minimum-priority root node.

Creating a new Fibonacci heap simply involves creating an empty doubly linked list and pointing the minimum-priority pointer to its root, which takes $O(1)$ time. To insert a new entry, a single-node tree is created for that entry and inserted into the forest list, and the minimum-entry pointer is updated if necessary; this takes $O(1)$ time. To decrease the priority of a key-priority entry, the lookup array is used to "jump" to the entry's node; the node is then deleted out of the tree, and a new key-priority entry is inserted following the usual procedure. Some additional work may be necessary to maintain the shape of the tree that the entry was deleted from, and the minimum-entry pointer. The pop-and-remove-min operation is the most complex. First, the minimum-entry pointer is followed and the key is saved so it can eventually be returned. Then that node's children are promoted to root status and inserted into the root list. If the root list is longer than $O(\log n)$, small trees are merged into larger trees so as to reduce the length of the root list back down to $O(\log n)$; the amortized time complexity of the merging process is $O(\log n)$. After the length of the root list is certain to be $O(\log n)$, it is searched again to update the minimum-priority node pointer. An elaborate analysis shows that the amortized time complexity of the decrease-key operation is $O(1)$, and that of the remove-minimum operation is $O(\log n)$.

10.4.4 Speeding up the Prim-Jarník Algorithm

Since this Section is all about reduction to data structures, we will treat the priority search queue as a "black box" and use its operations without pondering their implementation too deeply.

We first consider how to use a priority search queue to speed up the Prim-Jarník algorithm. We accomplish this by using such a queue, called `bridges`, to keep track of the bridge edges. Non-bridge edges are assigned a priority of `INFINITY`, while the priority of each bridge edge is its true weight. The algorithm now needs to initialize `bridges` by inserting each edge with priority `INFINITY`. Also, when a vertex u is marked as spanned, every edge e incident to u must have its priority decreased to w_e. The upside is that the minimum weight bridge edge may be found with

a pop-minimum operation instead of a slow sequential search.

```
def prim_jarnik(G):
    K = Vector()
    spanned = Vector(G.vertex_count(), False)
    bridges = PrioritySearchQueue()
    for e in G.edges():
        bridges.insert(e.index, INFINITY)
    span(G, spanned, bridges, 0)
    while len(K) < ( G.vertex_count() - 1 ):
        b = bridges.pop_minimum()
        K.add_back(b)
        span(G, spanned, bridges, b.v)
        span(G, spanned, bridges, b.w)
    return K

def span(G, spanned, bridges, s):
    if not spanned[s.index]:
        for e in s.incident():
            if spanned[e.v] != spanned[e.w]:
                bridges.decrease_priority(e.index, e.label)
        spanned[s.index] = True
```

Lemma 47. *The time complexity of this version of the Prim-Jarník algorithm is $O(m \log n)$ if using a BST-based priority search queue, or $O(m + n \log n)$ if using a Fibonacci heap.*

Proof. Let I, P and D be the amortized time complexity of each insert, pop-minimum and decrease-key operation, respectively.

The algorithm includes a sub-algorithm `span` that, like the `relax` sub-algorithm of Dijkstra's algorithm, needs to be analyzed carefully. As discussed in the analysis of the previous version of this algorithm, each vertex becomes spanned exactly once, so each edge may be considered in the `for` loop in the `span` algorithm at most once per each of its endpoints, so the number of iterations of that loop is at most $2m \in O(m)$. The body of the loop involves one decrease-key operation and $O(1)$ overhead, so the total time spent in all iterations of the loop is $O(mD)$. The rest of `span` takes only $O(1)$ time, so the total time spent in `span` is $O(mD)$.

We now count the time spent in all parts of the algorithm aside from `span`. Initializing `K` and `spanned` takes $O(n)$ time. Inserting the m edges into `bridges` takes $O(mI)$ time. The `while` loop repeats $(n-1)$ times, performs a pop-minimum operation that takes $O(P)$ time, and involves only $O(1)$ other steps aside from calling `span`. So the total time spent in the non-`span` parts of the algorithm is

$$O(n + mI + (n-1)P) = O(mI + nP).$$

Adding the `span` time to the non-`span` time we obtain

$$O(mD + mI + nP) = O(m(D + I) + nP).$$

If we use a BST-based priority queue then $D, I, P \in O(\log n)$ so the total runtime becomes

$$O(m(\log n + \log n) + n(\log n)) = O((m + n) \log n) = O(m \log n)$$

since the input graph is always connected and $m \geq n - 1$.

Alternatively, if we use a Fibonacci heap, then $D, I \in O(1)$ and $P \in O(\log n)$ and the total runtime is

$$O(m(1 + 1) + n(\log n)) = O(m + n \log n).$$

□

10.4.5 Speeding up Dijkstra's Algorithm

A similar optimization will allow us to speed up Dijkstra's algorithm. We introduce a priority search queue `unseen`, where each unseen vertex is assigned its priority based on the shortest known distance from s to that vertex. Then the costly sequential search for such a `u` may be replaced with a fast pop-minimum operation.

```
def dijkstra(G, s):
    distance = Vector(G.vertex_count(), None)
    penultimate = Vector(G.vertex_count(), None)
    seen = Vector(G.vertex_count(), False)

    distance[s.index] = 0

    unseen = PrioritySearchQueue()
    for v in G.vertices():
        if v is s:
            unseen.insert(v, 0)
        else:
            unseen.insert(v, INFINITY)

    while ((len(unseen) > 0) and
           (distance[unseen.minimum().index] != INFINITY)):
        u = unseen.pop_minimum()
        relax(G, distance, penultimate, seen, unseen, u)

    return distance, penultimate

def relax(G, distance, penultimate, seen, unseen, u):
    for e in u.incident():
        # either y==u or z==u
        y = e.s
        z = e.t
        if z == u: # ensure z != u
            swap(z, y)
        if not seen[z.index]:
            w = distance[u.index] + e.label
            if distance[z.index] is None or w < distance[z.index]:
                distance[z.index] = w
                penultimate[z.index] = u
                unseen.decrease_priority(z.index, w)
    seen[u.index] = True
```

Note that the start vertex `s` is inserted into the `unseen` queue, so it is certainly chosen as `u` in the first iteration of the `while` loop. Therefore we do not bother to relax `s` before the loop; it gets relaxed in the first iteration of the loop.

The main `while` loop repeats as long as the queue contains at least one vertex with a non-infinite distance. We need to safeguard against an infinite distance because, when the graph is unconnected, eventually all of the reachable vertices will be removed from the queue, but the unreachable vertices

will remain all with infinite priorities. The body of the loop is massively simplified since the entire process of searching for a closest unseen vertex reduces to a single priority search queue pop-minimum operation. The `relax` sub-algorithm is essentially the same, except that when it is updating the `distance` and `penultimate` data structures with a new shortest path to vertex z, it must be careful to keep the priority search queue in sync by decreasing the priority for z's entry.

Lemma 48. *The time complexity of this version of Dijkstra's algorithm is $O(m \log n)$ if using a BST-based priority search queue, or $O(m + n \log n)$ if using a Fibonacci heap.*

Proof. Let I, P and D be defined as in Lemma 47. Just like that Lemma, we first analyze count the time spent in `relax`, and then count the time spent in all other parts of the algorithm.

As established in Lemma 29, the runtime of `relax` is dominated by the time spent in the `for` loop. That loop iterates at most twice per graph edge, so in the worst cast it iterates $2m$ times. The body of the loop executes one decrease-priority operation which takes D steps, and $O(1)$ other steps. So the total time spent in `relax` is $O(mD)$.

Initializing the `distance`, `penultimate`, and `seen` vectors takes $O(n)$ time. Creating the `unseen` priority search queue and inserting all of the vertices takes $O(nI)$ time. The main `while` loop iterates up to n times, and aside from `relax`, it performs one pop-minimum operation and $O(1)$ additional steps. The `return` statement costs one step. So the total amount of time spent aside from relaxation is $O(n + nI + n(P + 1)) = O(n(I + P))$.

The total time spent in both relaxing and non-relaxing work is $O(mD + n(I + P))$. If the priority search queue is based upon binary search trees, this is $O(m(\log n) + n(\log n + \log n)) = O((m + n) \log n) = O(m \log n)$. If the priority search queue is a Fibonacci heap, then this is $O(mD + n(I + P)) = O(m(1) + n(1 + \log n)) = O(m + n \log n)$. □

Exercises

10-1. All of the algorithms that reduce to sorting in this Chapter use merge sort and have an $O(n \log n)$ deterministic time efficiency. What would be the consequence of reducing to quick sort instead of merge sort?

10-2. For each of the following problems: design a reduction algorithm that solves the problem; describe your algorithm with clear pseudocode; and prove the time efficiency class of your algorithm. *Hint:* your algorithm should use a hash table or sorting algorithm.

(a)

set union
input: two lists L and R, each representing a set of distinct elements **output:** a list S representing $L \cup R$

(b)

set difference
input: two lists L and R, each representing a set of distinct elements **output:** a list S representing $L - R$

(c)

reverse sorting problem
input: a list U of n comparable elements **output:** a list S containing the elements of U in non-increasing order

(d)

mode finding
input: a list L of $n > 0$ orderable elements **output:** an element $x \in L$ and non-negative integer f, such that x is a most-frequently-appearing element of L that appears exactly f times

(e)

duplicate search problem
input: a list L of comparable objects **output:** an element of L that appears more than once in L, or `None` if no such element exists

(f) [40] Suppose that we have a collection of cardboard boxes, and want to arrange them into the tallest stack possible. A box can only be placed on top of another if the top box is smaller in both width and depth.

stacking boxes
input: a list B of box dimensions, where each element is a tuple (w, d, h) representing the positive width, depth, and height of a box **output:** a list S of boxes from B, where each `B[i].w < B[i+1].w`, `B[i].d < B[i+1].d`, and the sum of all heights in S is maximized

(g) [40] Define a *peak* to be an element that is greater than or equal to its adjacent element(s), and a *valley* to be an element that is less than or equal to its adjacent element(s). For example, in the list `[8, 6, 7, 5, 3, 0, 9]`, the peaks are 8, 7, and 9; and the valleys are 6 and 0.

peaks and valleys
input: a list L of numbers **output:** a list PV containing the elements of L, such that PV is an alternating sequence of peaks and valleys

10-3. For each of the following problems: design a reduction algorithm that solves the problem; describe your algorithm with clear pseudocode; and prove the time efficiency class of your algorithm.

(a)

maximum spanning tree
input: a connected, undirected, and weighted graph $G = (V, E)$ **output:** a list of edges K such that $T = (V, K)$ is a maximum spanning tree for G

A *maximum spanning tree* is a spanning tree of maximal total weight.

(b)

unweighted single source shortest paths
input: an unweighted graph $G = (V, E)$, and a start vertex $s \in V$ **output:** two lists `distance` and `penultimate`, as defined for the non-negative single source shortest paths problem

10-4. Design an algorithm that solves set intersection by reduction to sorting, but only performs *one* sort, not two. What is the efficiency class of your algorithm?

10-5. Implement an in-place version of heap sort. You may need to do some research into the details of Python's `heapq` operations.

Chapter 11

Dynamic Programming

11.1 The Big Idea

Dynamic programming is an algorithm pattern that is applicable to only a narrow range of circumstances, but offers impressive efficiency gains in those circumstances. At a very high level, the pattern is suited to problems with a divide-and-conquer structure (e.g. decrease-by-one or decrease-by-half) where there is "overlap" between the sub-instances. A Naïve recursive algorithm would recompute solutions for the overlapping parts of the sub-instances, which would be redundant and wasteful. In the dynamic programming pattern, an array `A` is used as a *cache* for sub-instance solutions. The first time a particular sub-instance is solved, its solution is stored in an element of `A`; subsequently, the same sub-instance's solution may be found with a fast array lookup rather than being recomputed from scratch.

We shall categorize dynamic programming algorithms by *dimension,* which is the number of nested `for` loops needed to fully initialize `A` and solve the full-sized problem. The following pattern illustrates the *1D (one-dimensional)* dynamic programming pattern.

```
def dynamic_programming_1D(instance):
  A = Array(<SIZE OF T>, None)
  <INITIALIZE BASE CASE ELEMENTS OF A>
  for i in range(<LEAST NON-BASE CASE>, <GREATEST INSTANCE> + 1):
    <INITIALIZE A[i] USING ONLY ELEMENTS OF A THAT HAVE ALREADY BEEN
     INITIALIZED>
  return A[<SOLUTION INDEX>]
```

There are many blanks, and it is often impossible to fill them in without some advance planning. We suggest designing a dynamic programming algorithm in phases:

1. Define an *invariant* that defines the contents of each element `A[i]` of the solution array `A`. Since the output of the algorithm is `A[<SOLUTION INDEX>]`, the data type of each `A[i]` must match the data type of the problem's output; and, the invariant must guarantee that some easily-identifiable array element will contain the problem's solution. This is analogous to how, in the exhaustive search pattern, the data type of a candidate must match the data type of the problem's output.

2. Identify some base-case elements of `A` that may be initialized, trivially, before the main `for` loop.

3. Thinking in a *top-down* way, design algorithm steps for initializing a general, non-base `A[i]`. Here "top-down" means thinking in terms of a divide-and-conquer recursive algorithm that combines smaller solutions into one larger solution. These steps may reference other array elements `A[j]`, provided `j` $<$ `i`, since indices coming before `i` would be initialized earlier in the loop, but indices coming after `i` would still be uninitialized.

4. Transform your top-down steps into a *bottom-up* algorithm that starts out initializing the low-index elements of `A`, and works its way up. You could think of this as turning the algorithm "upside down."

5. Fill in any remaining blanks, which should be straightforward at this point, to produce a final draft.

11.2 1D Dynamic Programming and Fibonacci Numbers

Our first example involves a computation which is likely to be familiar.

Definition 45. *The nth element of the sequence of Fibonacci sequence, for* $n \geq 0$, *is*

$$\begin{aligned} F(0) &= 0 \\ F(1) &= 1 \\ F(n) &= F(n-2) + F(n-1). \end{aligned}$$

Example 52. *The first ten Fibonacci numbers are* $0, 1, 1, 2, 3, 5, 8, 13, 21,$ *and* 34.

We can formulate the computation of a Fibonacci number as a computational problem.

Fibonacci number
input: a non-negative integer n **output:** the nth Fibonacci number

The Fibonacci number problem does not seem to be particularly relevant to practical programming; but it is simple, and makes for a clear illustration of the dynamic programming pattern.

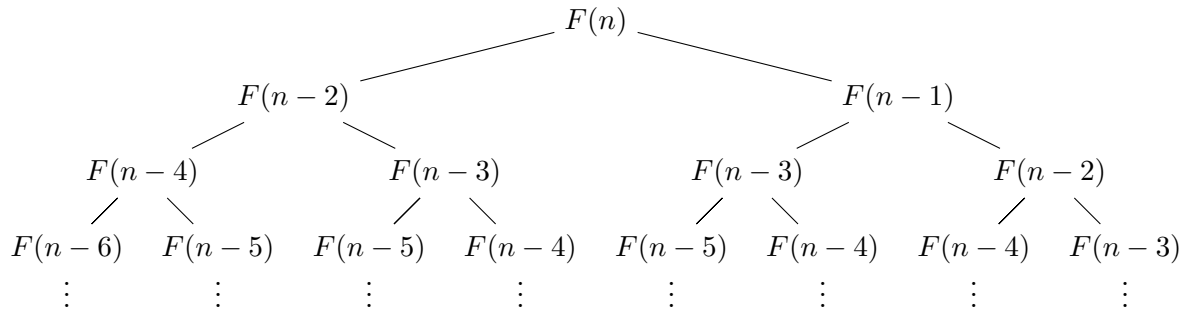
The recurrence defining the Fibonacci sequence may be translated directly into a straightforward recursive algorithm.

```
def fibonacci_recursive(n):
    if n == 0:
        return 0
    elif n == 1:
        return 1
    else:
        return fibonacci_recursive(n-1) + fibonacci_recursive(n-2)
```

This algorithm is sometimes used as an example for programming students first learning about recursion, despite being an extremely inefficient algorithm.

The time complexity of the `fibonacci_recursive` algorithm is $O(2^n)$.

To gain some intuition into how such a simple algorithm can be so time-intensive, we draw the computation tree evaluated by a call to `fibonacci_recursive`. Each $F(i)$ node represents a call to `fibonacci_recursive(i)`, and its two child nodes are the function's two recursive calls.



Upon close examination, this computation tree involves a *lot* of redundancy. For example, there are two complete subtrees rooted at a $F(n-2)$ node; one is the left child of $F(n)$, and the other is the right child of $F(n-1)$. Both of these subtrees are completely identical, which means that the algorithm computes $F(n-2)$ from scratch *twice.* Each of those computations involves many other redundant computations; for example $F(n-4)$ appears four times in our diagram. These redundant subtrees are what we mean by "overlapping sub-problems."

We now set to designing a dynamic programming algorithm for the Fibonacci number problem, using the phases outlined above. We first define an invariant clarifying what exactly is represented by each array element `A[i]`.

Invariant 9. *After being initialized,*

$$\texttt{A[i]} \;=\; \textit{the nth Fibonacci number.}$$

The definition of Fibonacci numbers says that $F(0) = 0$ and $F(1) = 1$, so we have two straightforward base cases, `A[0] = 0` and `A[1] = 1`. We use these statements to fill in the `<INITIALIZE BASE CASE ELEMENTS OF A>` blank.

```
def fibonacci_dynamic(n):
  A = Array(<SIZE OF A>, None)
  A[0] = 0
  A[1] = 1
  for i in range(<LEAST NON-BASE CASE>, <GREATEST INSTANCE> + 1):
    <INITIALIZE A[i] USING ONLY ELEMENTS OF T THAT HAVE ALREADY BEEN
     INITIALIZED>
  return A[<SOLUTION INDEX>]
```

Next, we need to determine how to initialize each non-base-case element `A[i]`. Again looking to the definition of Fibonacci numbers, we see the inductive definition $F(n) = F(n-2) + F(n-1)$, or equivalently $F(i) = F(i-2) + F(i-1)$. According to our invariant, $F(i-2)$ is stored at `A[i-2]` and $F(i-1)$ is stored at `A[i-1]`. So we may initialize each `A[i]` with the quantity `A[i-2] + A[i-1]`. Note that, while this expression does reference other array elements, they are at indices less than `i` which should be initialized prior to iteration `i` of the `for` loop.

```
def fibonacci_dynamic(n):
  A = Array(<SIZE OF A>, None)
  A[0] = 0
  A[1] = 1
  for i in range(<LEAST NON-BASE CASE>, <GREATEST INSTANCE> + 1):
    A[i] = A[i-2] + A[i-1]
  return A[<SOLUTION INDEX>]
```

Our algorithm must `return` the nth Fibonacci number; the invariant says it should be at `T[n]`. We fill in the `return T[<SOLUTION INDEX>]` blank accordingly.

```
def fibonacci_dynamic(n):
  A = Array(<SIZE OF A>, None)
  A[0] = 0
  A[1] = 1
  for i in range(<LEAST NON-BASE CASE>, <GREATEST INSTANCE> + 1):
    A[i] = A[i-2] + A[i-1]
  return A[n]
```

That means that the `<GREATEST INSTANCE>` is `i=n`. The greatest base case is $i = 1$, so `<LEAST NON-BASE CASE>` is $i = 2$.

```
def fibonacci_dynamic(n):
  A = Array(<SIZE OF A>, None)
  A[0] = 0
  A[1] = 1
  for i in range(2, n+1):
    A[i] = A[i-2] + A[i-1]
  return A[n]
```

Note that `range(2, n+1)` is a rather peculiar range of indices for a `for` loop dealing with an array. Dynamic programming algorithms often do quite a bit of peculiar, yet correct, array indexing.

The only remaining blank is `<SIZE OF A>`. We need `A[n]` to be a valid array element, so we need `A`'s size to be $n + 1$. Note, again, that this is one more than the more typical list length n.

```
def fibonacci_dynamic(n):
  A = Array(n+1, None)
  A[0] = 0
  A[1] = 1
  for i in range(2, n+1):
    A[i] = A[i-2] + A[i-1]
  return A[n]
```

This algorithm is *almost* correct, but not quite; it contains a subtle bug. The bug manifests when $n = 0$, which is allowable according to the Fibonacci number problem's definition. If $n = 0$ then $(n + 1) = 1$, and `A` is created as a length-1 array. That is fine for the statement `A[0] = 0`, but then the assignment `A[1] = 1` goes out of bounds and crashes.

We could handle this with an `if` statement at the start of the algorithm, that handles $n = 0$ as a special case; or we could simply ensure that `A` is always large enough. We suggest using the second approach of always making the solution array `A` large enough to accomodate all base cases. That approach tends to make for substantially less code in situations where a problem might have many special cases.

Our final draft of the dynamic programming algorithm is below. It creates `A` with `max(n+1, 2)` elements, so that `A` is guaranteed to always have at least two elements, and the base cases never go out-of-bounds.

```
def fibonacci_dynamic(n):
  A = Array(max(n+1, 2), None)
  A[0] = 0
  A[1] = 1
  for i in range(2, n+1):
    A[i] = A[i-2] + A[i-1]
  return A[n]
```

Claim 15. *The time complexity of* `fibonacci_dynamic` *is* $O(n)$.

This is a huge improvement: all the way from $O(2^n)$ to $O(n)$. The pseudocode is remarkably concise and rudimentary, too; it involves only arrays, arithmetic, and a `for` loop, and could probably be implemented by a novice programmer. A novice programmer would be unlikely to come up with this algorithm themselves, however.

This algorithm is typical of dynamic programming algorithms in many ways:

- the dynamic programming algorithm is much faster than alternatives;
- the algorithm's correctness depends on many fine subtleties (e.g. `range(2, n+1)` and `max(n+1, 2)`) which might be easily overlooked;
- the final pseudocode is short and simple, and easy to analyze and implement;
- it is unclear how the pseudocode corresponds to the problem, and could be described as "alien."

Dynamic programming algorithms have a reputation for being difficult to design and understand. One potential source of difficulty is that these algorithms require a great deal of attention to detail. One could be forgiven for overlooking the nuances of initializing `A` to the proper length `max(n+1, 2)` ; or stopping the `for` loop at `i=n` instead of `i=n-1` ; or returning `A[n-1]` instead of `A[n]` . But any of these oversights would render the algorithm incorrect.

Another source of difficulty is the fact that the dynamic programming pattern involves deliberately mixing array indices with the values provided as input. Ordinarily we exercise great discipline in distinguishing the concept of an array index `i` from a meaningful value `A[i]` . We usually consider an index `i` and element `A[i]` to be different data types, and regard an expression that mixes them, such as `A[i - B[j]]` to be suspicious and most likely wrong. But the dynamic programming algorithm makes critical use of the fact that computers are capable of mixing indices with integer values. The pattern forces us to mix indices with values, which can feel unsettling, yet be correct and indeed efficient.

11.3 American Change-Making

Recall that our first example of a greedy algorithm solved the American change-making problem.

American change-making
input: a number of cents $k \geq 0$ **output:** a list V of coin values drawn from $\{1, 5, 10, 25\}$ such that $(\sum_{c \in V} c) = k$ and the length of V is minimal

Our greedy algorithm is quite simple and solves the problem in $O(|V|)$ time. The problem can also be solved by a dynamic programming algorithm in $O(|V|)$ time; this algorithm is a bit more complicated, but is easier to generalize to denomination sets besides $\{1, 5, 10, 25\}$. We follow the five phases outlined in the introduction.

1. *Invariant.*

   Invariant 10. *After being initialized,*

   `A[i]` = *a minimal-length vector of coin values that adds up to* `i`.

2. *Base cases.* We can always make zero cents by using an empty list of coins.

   ```
   A[0] = Vector()
   ```

   Note that this assignment is consistent with our correctness invariant. The minimal-length list that adds up to exactly `i` $= 0$ is the empty vector.

3. *Inductive case.* Here we need to work out pseudocode to initialize `A[i]` for a general index `i` > 0. At first this seems daunting, so for the sake of discussion we focus on using only whether or not to pick a 25 cent piece. If we commit to using a quarter, then so far we have a vector `q` (for "quarter") that contains only a single element `25`.

   ```
   q = Vector()
   q.add_back(25)
   ```

   The vector `q` sums to 25, so we still need to account for the remaining (`i` $- 25$) cents. According to our invariant,

   `A[i]` = a minimal-length vector of coin values that adds up to `i`.

   so the vector of the best combination of coins to make (`i` $- 25$) should be available at

   `A[i-25]`.

Appending these lists together, we obtain the statement

```
q = copy of A[i-25] with 25 added
```

By construction `q` is a vector of valid coin denominations, whose sum is exactly `i`, and is of minimal length, *provided that the last coin added is a quarter.*

While it is possible that this `q` is correct, we also need to consider the possibility that picking a different coin at this point produces a superior (shorter) vector. By the same sort of reasoning that we used for `q`, we can form alternative vectors `d`, `n`, and `p` (for "dime," "nickel," and "penny" respectively).

```
    q = copy of A[i-25] with 25 added
    d = copy of A[i-10] with 10 added
    n = copy of A[i-5] with 5 added
    p = copy of A[i-1] with 1 added
```

Each of these four vectors is a valid list of denominations that sums to exactly `i`. How do we choose one as the value for `A[i]`? The definition of the American change-making problem says that we should favor the vector of minimal length.

```
    q = copy of A[i-25] with 25 added
    d = copy of A[i-10] with 10 added
    n = copy of A[i-5] with 5 added
    p = copy of A[i-1] with 1 added
    A[i] = the shortest vector among q, d, n, p
```

4. *Bottom-up pseudocode.* Next we pull all these ideas together into a first draft of some pseudocode. Our algorithm creates an array `A` of adequate length, initializes the base case `A[0]`, uses a `for` loop to initialize all the remaining elements of `A`, and finally returns `A[k]`, which according to our invariant stores the solution to the original problem.

```
def american_change(k):
    A = Array(None, k+1)
    A[0] = Vector()
    for i in range(1, k+1):
        q = copy of A[i-25] with 25 added
        d = copy of A[i-10] with 10 added
        n = copy of A[i-5] with 5 added
        p = copy of A[i-1] with 1 added
        A[i] = the shortest vector among q, d, n, p
    return A[k]
```

5. *Final draft.* Once again, our draft pseudocode contains a bug. This time the base case initializations are safe, and the bug cannot be found there. We know from the problem

definition that `k` ≥ 0, and `A` is initialized to length $(k+1)$, so `A` always contains at least one element and the statement

```
A[0] = Vector()
```

is safe, even when $k = 0$. Instead, the bug is inside the `for` loop. The index variable `i` starts at 1 and works upwards, but the body of the loop contains the expressions `A[i-25]`, `A[i-10]`, and `A[i-5]`, each of which could be an out-of-bounds array index. Note that the expression `A[i-1]` that considers using a penny is always safe since `i` ≥ 1. We can fix the bug by adding an `if` around each of the potentially-unsafe array references.

```
def american_change(k):
    A = Array(None, k+1)
    A[0] = Vector()
    for i in range(1, k+1):
        A[i] = copy of A[i-1] with 1 added
        if i >= 25:
            q = copy of A[i-25] with 25 added
            if q is shorter than A[i]:
                A[i] = q
        if i >= 10:
            d = copy of A[i-10] with 10 added
            if d is shorter than A[i]:
                A[i] = d
        if i >= 5:
            n = copy of A[i-5] with 5 added
            if n is shorter than A[i]:
                A[i] = n
    return A[k]
```

This is the final draft of our algorithm.

Lemma 49. *The time and space complexity of* `american_change` *are each* $O(k \cdot |V|)$.

Proof. The algorithm's space complexity is dominated by the vector `A`, which has exactly $(k+1)$ elements. Each element of `A` is a vector that contains at most `V` elements, so the total space complexity is $O((k+1) \cdot$ `V` $) = O(k \cdot$ `V` $)$.

The algorithm's time complexity is dominated by the `for` loop, which iterates $O(k)$ times. The time complexity of the body of the loop is dominated by the vector-construction expressions, such as `q = copy of A[i-25] with 25 added` and variations of same. Each of these expressions builds a new vector object of length $O(|V|)$, so takes $O(|V|)$ time. The rest of the loop body takes $O(1)$ time, so the total time complexity of the loop is $O(k \cdot |V|)$. Initializing `A` takes $O(k)$ time, and the rest of the algorithm takes $O(1)$ time, so its total time complexity is $O(k \cdot |V|)$. □

11.4 2D Dynamic Programming and Universal Change-Making

The part of the `american_change` algorithm that handles the denominations 1, 5, 10, and 25 is highly repetitive, which begs the question of whether this algorithm may be generalized to coin systems other than the familiar $\{1, 5, 10, 25\}$. This corresponds to a more general variant of the change making problem.

universal change-making
input: a vector C of distinct positive coin denominations; and a number of cents $k \geq 0$ **output:** a vector V of coin values drawn from C such that $(\sum_{c \in V} c) = k$ and the length of V is minimal; or `None` if no such V exists

The American change making problem is essentially a special case of the universal problem, where the coin set is hard-coded as $C = \{1, 5, 10, 25\}$. However, there is one additional difference. The general problem's definition does not guarantee that $1 \in C$, which means that some cent totals may be unachievable.

Example 53. *It is impossible to make $k = 3$ cents using the denomination set $C = \{2, 7\}$.*

We now adapt the previous dynamic programming algorithm to handle this more general problem. Our algorithm will end up having two loops, making it a 2D dynamic programming algorithm.

1. *Invariant.* We use the same invariant as before, except that we accomodate unachievable totals.

   `A[i]` = a minimal-length vector of coin values that adds up to `i`, or `None` if no such vector exists.

2. The *base case* may be re-used, unchanged.

   ```
   A[0] = Vector()
   ```

3. *Inductive case.* Recall that our final algorithm for the American problem had the following inductive case:

```
        A[i] = copy of A[i-1] with 1 added
        if i >= 25:
            q = copy of A[i-25] with 25 added
            if q is shorter than A[i]:
                A[i] = q
        if i >= 10:
            d = copy of A[i-10] with 10 added
            if d is shorter than A[i]:
                A[i] = d
        if i >= 5:
            n = copy of A[i-5] with 5 added
            if n is shorter than A[i]:
                A[i] = n
```

This pseudocode is hard-coded to consider using the alternatives of a 1, 5, 10, or 25 cent piece. Now that our coin set C is part of the input, we need to use a loop to iterate through C's elements. Here is a first attempt:

```
        for denom in C:
            if i >= denom:
                candidate = copy of A[i - denom] with denom added
                if candidate is shorter than A[i]:
                    A[i] = candidate
```

This loop captures the idea of the repeated `if` blocks, where `denom` corresponds to the values 1, 5, 10, or 25. It would be correct, except for the fact that some cent totals are unachievable, so `A[i]` may be `None`, even after being initialized.

```
        for denom in C:
            if i >= denom:
                candidate = copy of A[i - denom] with denom added
                if A[i] is None or candidate is shorter than A[i]:
                    A[i] = candidate
```

Finally, we must also contend with the fact that `A[i - denom]` may also be `None`. We can handle this by skipping over `denom` values for which `A[i - denom]` is `None`.

```
        for denom in C:
            if i >= denom and A[i - denom] is not None:
                candidate = copy of A[i - denom] with denom added
                if A[i] is None or candidate is shorter than A[i]:
                    A[i] = candidate
```

4. *Bottom-up pseudocode.* Combining the previous three steps, we obtain the following pseudocode.

```
def universal_change(C, k):
    A = Array(k+1, None)
    A[0] = Vector()
    for i in range(1, k+1):
        for denom in C:
            if i >= denom and A[i - denom] is not None:
                candidate = copy of A[i - denom] with denom added
                if A[i] is None or candidate is shorter than A[i]:
                    A[i] = candidate
    return A[k]
```

Claim 16. *The time and space complexity of* `universal_change` *are each* $O(k \cdot |C| \cdot |V|)$.

Exercises

11-1. Compute the 5th Fibonacci number, $F(5)$, using the dynamic programming algorithm. Show your work, including the contents of the dynamic programming array `A`.

11-2. Our final draft of `fibonacci_dynamic` used $O(n)$ time and $O(n)$ space. Design a version that uses $O(n)$ time but only $O(1)$ space.

Hint: Create `A` with exactly 3 elements.

11-3. For each of the following problems: design a dynamic programming algorithm that solves the problem; describe your algorithm with clear pseudocode; and prove the time efficiency class of your algorithm.

(a) The tribonacci sequence is a generalization of the Fibonacci sequence which looks back at the previous *three* elements instead of just two. The nth tribonacci number may be defined by the recurrence

$$\begin{aligned} T(n) &= T(n-3) + T(n-2) + T(n-1) \\ T(2) &= 1 \\ T(1) &= 0 \\ T(0) &= 0. \end{aligned}$$

tribonacci number
input: an integer $n \geq 0$ **output:** the nth tribonacci number

(b) Section 11.3 discusses the American change-making problem, where the goal is to arrive at a list of coins worth k cents. What if, instead, we wanted to know how many different ways there are of making k cents?

American change-counting problem
input: a number of cents $k \geq 0$ **output:** an integer n, where n is the number of distinct lists of coin values drawn from $\{1, 5, 10, 25\}$ that add up to exactly k

(c) (If you are unfamiliar with this problem, review its introduction in Section 7.9.)

the masseuse's problem
input: a list of requested appointment-lengths, in minutes (none overlap and none can be moved) **output:** a list of appointment-lengths that can be honored, such that no adjacent requests are included, and the total number of minutes is maximized

(d) Suppose you harvest berries along a forest path. There are b kinds of berries, but you have only two berry baskets, and you can only sell the contents of a basket if all its berries are of the same kind. Where should you start and stop picking?

the berry picker's problem
input: a positive integer b, and non-empty list P of n integers in the range $[1, b]$ **output:** integers i, k such that k is maximized, and the slice `P[i:i+k]` contains only one or two distinct values

(e) This problem captures the spirit of dividing up a resource into pieces, where there are alternative kinds of pieces with different values. For example, a real estate developer might need to decide how to divide a tract of land into a combination of small lots for houses and large lots for apartment buildings. We model the resource as a list of bits (0s or 1s) and the two kinds of resource as a `[1, 1, 0]` or `[0, 1]` subsequence.

the land monger's problem
input: a list L of n bits; and positive real numbers a and b **output:** a list of non-overlapping `[1, 1, 0]` or `[0, 1]` subsequences of L of maximum value, where each `[1, 1, 0]` is worth a and each `[0, 1]` is worth b

(f) This problem involves finding the most profitable way of cutting a log into shorter timbers for sale. Note that there is no constraint on the `price` list. In particular, there is no guarantee that longer timbers are more valuable than shorter ones. This means that the greedy pattern is unlikely to work. (This is realistic; for example, the United States has standardized on 96-inch dimensional lumber, so 96-inch timbers are more valuable than 97-inch timbers that would probably need to be trimmed.)

the lumberjacks's problem
input: the length n of an uncut log in cm, and a list of non-negative market prices such that `price[i]` = the value of a log `i` cm long for any $0 \leq$ `i` $\leq n$ **output:** a vector of cm-lengths that the log should be cut into, such that the total of all lengths is at most n, and the total value of all pieces is maximized

(g) This problem involves computing the cheapest way of buying packs of chicken nuggets, to obtain exactly n nuggets. Note that there is no constraint on the pricing scheme. In particular, there is no guarantee that larger packs are offered at a lower unit price. This means that the greedy pattern is unlikely to work.

precise chicken nugget problem
input: a desired number of nuggets n; and a non-empty list of *packs,* where each pack is a pair (k, c) indicating that a certain fast food restaurant will sell a pack of k (a positive integer) nuggets for a cost of c (a positive real number) **output:** a list of packs, not necessarily distinct, accounting for exactly n nuggets, and of minimal total cost; or `None` if it is impossible to buy exactly n nuggets

(h) In this variation of the previous problem, the goal is to buy *at least* n nuggets instead of exactly n nuggets. Note that this relaxation means that it is always possible to find a solution, so the output may never be `None` .

generous chicken nugget problem
input: a desired number of nuggets n; and a non-empty list of *packs,* where each pack is a pair (k, c) indicating that a certain fast food restaurant will sell a pack of k (a positive integer) nuggets for a cost of c (a positive real number) **output:** a list of packs, not necessarily distinct, accounting for at least n nuggets, and of minimal total cost

(i) In this problem, you are planning the path of a robot arm through a rectangular grid. A grid cell may be empty, represented by 0; or may contain one gold coin, represented by 1. The goal is to collect as many coins as possible. The path always starts at the top-left corner at row 0 and column 0. The robot *can only move right or down;* the robot cannot move left or up.

robot arm motion
input: an $n \times m$ matrix M, with each $M[i][j] \in \{0, 1\}$ **output:** a list P, where each element is R (right) or D (down); and P is a path through M where the sum of the elements of M that P visits is maximized

(j) [40] Suppose we have a black-and-white image, and need to find the largest square in the image, where every pixel along the borders of the square is black (zero).

largest black square problem
input: a $n \times n$ square matrix M, where each `M[i][j]` $\in \{0, 1\}$ **output:** (i, j, d) where i and j are the top-left row and column of a largest black square in M, and d is the width/height of that square; or `None` if M contains no black pixel

Act II

Limitations of Algorithms

Chapter 12

Lower Bounds

12.1 The Big Idea

This chapter introduces the idea of a lower bound of a problem: that is, for a particular problem, the best possible time complexity of any algorithm solving the problem. A lower bound is a kind of a "speed limit" for a particular problem.

Notably, the sorting problem's lower bound is $\Omega(n \log n)$, which means that every algorithm that solves the sorting problem has time complexity $O(n \log n)$ or slower. We have seen two algorithms with $O(n \log n)$ time complexities (merge sort and quick sort), and one with a slower $O(n^2)$ time complexity (selection sort). This $\Omega(n \log n)$ lower bound implies that we will never see a sorting algorithm faster than $O(n \log n)$; $O(n)$, for example, is out of the question.

A lower bound is a mixed blessing. On the one hand, it is bad news, because it means that faster algorithms are hopeless; no matter how clever we are at designing algorithms, we will never be able to break the "speed limit" represented by a lower bound. On the other hand, a lower bound can be liberating because it determines an end-goal for algorithm design. If we know the lower bound for a given problem, and succeed in designing an algorithm whose complexity matches that lower bound, then we are effectively done. We have designed a fastest-possible algorithm for the problem, no one could ever improve upon it, so the problem is a closed case and we may turn to different problems. In the case of sorting, the $\Omega(n \log n)$ lower bound means that we need not toil at designing an algorithm that is asymptotically faster than merge sort and quick sort.

12.2 Notation and Terminology

Up until now all of our asymptotic efficiency classes have involved big-O. Recall the definition of O for univariate complexity functions,

$O(f(n)) = \{g(n) \mid$ there exists some constants $c > 0$ and $t \geq 0$ such that $g(n) \leq c \cdot f(n)$ whenever $n \geq t\}$.

Intuitively, $O(f(n))$ is the set of functions that are upper-bounded by $f(n)$. $O(n)$ is the set of functions that are at most c·n for some constant c, while $O(n^2)$ is the set of functions that are at most $c \cdot n^2$ for some (independent) constant c. This O notation is designed to notate upper bounds. When we design and analyze an algorithm, we establish an upper bound for how slow an algorithm for that problem might be.

Definition 46. *If algorithm A solves problem X in $O(f(n))$ worst-case time, then A establishes an upper bound of $O(f(n))$ for X.*

For instance, our first sorting algorithm was selection sort whose time complexity is $O(n^2)$. Selection sort proves that it is possible to solve the sorting problem in $O(n^2)$ time. We call this kind of result an upper bound, because it establishes that sorting takes $O(n^2)$ time or possibly less. Later we derived merge sort which takes only $O(nlogn)$ time. This proved a more insightful upper bound, that sorting takes $O(nlogn)$ time or less. Note that these results are not contradictory. Selection sort shows that sorting "costs" up to $O(n^2)$ time, or possibly less; merge sort refines that budget to say that sorting costs $O(nlogn)$ time, or possibly less. By contrast a lower bound for a problem establishes the fastest possible worst-case time complexity of an algorithm for that problem.

Definition 47. *If each algorithm A that solves problem X has worst-case time complexity $O(f(n))$ or slower, then X has a lower bound of $\Omega(f(n))$.*

This definition references big-Ω, which is a variation on big-O intended for lower bounds.

$\Omega(f(n)) = \{g(n) \mid$ there exists some constants $c > 0$ and $t \geq 0$ such that $g(n) \geq c \cdot f(n)$ whenever $n \geq t\}$.

Observe that the only difference between the definition of O and Ω is that O uses a $\leq$ inequality while Ω uses $\geq$. $\Omega(n^2)$ is the set of functions that grow proportional to n^2, or faster, so $n^2, n^3, n^4, 2^n \in \Omega(n^2)$.

Earlier we claimed that the sorting problem has a lower bound of $\Omega(n \log n)$. That lower bound implies that every correct sorting algorithm must have a time complexity of $O(n \log n)$ or possibly slower. We have seen sorting algorithms with $O(n \log n)$ and $O(n^2)$ time complexities, which are compatible with this lower bound. Even slower sorts, which take $O(n^3)$ time, say, are also conceivable and consistent with this lower bound. When a problem's upper and lower bounds match, we say that the problem has a *tight bound.*

Definition 48. *When problem X has an upper bound of $O(f(n))$ and matching lower bound $\Omega(f(n))$, we say that X has a tight bound of $\Theta(f(n))$.*

Analogous to O and Ω, Θ is is a kind of asymptotic notation that is intended to notate tight bounds.

When a problem has a tight bound, we not only know the fastest possible efficiency class of an algorithm for that problem, but also have at least one algorithm that achieves that level of speed. Algorithm designers tend to view this as a thoroughly satisfying answer to the question of how efficiently that problem could be solved. When a problem has a tight bound, it is often considered a closed case with respect to algorithm design; there will never be a faster algorithm, at least in

an asymptotic sense. On the other hand, when there is a gap between the best known lower and upper bounds for a problem, there is an open question as to which bound can be improved on.

Before moving on, we emphasize that upper bounds and lower bounds are properties of *problems,* not *algorithms.* Up until now our analyses and proofs have all centered on the features of individual algorithms. We are now changing gears to consider the inherent difficulty of solving a problem. A given problem can have many competing algorithms, but all of those algorithms must contend with the same intrinsic obstacles while solving the problem.

12.3 Proving Negatives

Establishing a lower bound is fundamentally challenging because it involves proving a negative.

An upper bound, such as "there exists an algorithm that solves the sorting problem in $O(n \log n)$ time," can be proven by example. To prove this claim, we only need to show one example of an algorithm that solves the sorting problem in $O(n \log n)$ time. This is a "there exists" condition which can indeed be proven with only one example. So our design and analysis of merge sort could be considered a proof of an upper bound of $O(n \log n)$ for sorting.

Proving a lower bound is more difficult because that kind of proof involves a "for all" condition. In order to prove that sorting takes at least $\Omega(n \log n)$ time, we need to prove that, for every sorting algorithm every designed, and every sotring algorithm that might be invented in the future, that algorithm's time complexity is $\Omega(n \log n)$ or slower. Our argument must somehow address the fact that other algorithm designers might have insight we lack, and that future computer scientists could develop new algorithm design patterns. We must be careful to avoid the *personal incredulity fallacy*, which is the mistaken belief that, when something does not make sense to us, it cannot be true. In particular, if we try to design a fast algorithm and fail, that does not prove that no fast algorithm exists. While it might be the case that no fast algorithm exists, it could also be the case that we lack the expertise or insight necessary to devise such an algorithm. Instead, our argument for a lower bound must be built upon some kind of intrinsic property of the problem at hand that forces correct algorithms to exert a certain amount of effort. We will explore three proof techniques for establishing lower bounds: trivial arguments, information-theoretic arguments, and reduction arguments.

12.4 Trivial Lower Bounds

A trivial lower bound is implicit in the definition of the problem. There are two kinds of trivial lower bounds. The first is based upon the fact that a correct algorithm must, at a minimum, initialize the data structures that hold its output.

Claim 17. *If the space complexity of the output for problem X is $O(f(n))$, then X has a lower bound of $\Omega(f(n))$.*

Recall the definition of the sorting problem.

sorting problem
input: a list U of n comparable elements **output:** a list S containing the elements of U in non-decreasing order

The output of this problem is a vector S of length n. This data structure has space complexity $O(n)$; therefore, there is a trivial lower bound of $\Omega(n)$ time for the sorting problem. Intuitively, every correct sorting algorithm must spend at least $\Omega(n)$ time just to create S. In all likelihood any sorting algorithm does quite a bit more than just initialize the elements of S. But, at a minimum, we can be certain that at least $\Omega(n)$ time is spent solely on creating the output data structure.

We may also define a trivial lower bound corresponding to the amount of input data that any correct algorithm must interact with.

Claim 18. *If every correct algorithm for problem X must observe $O(f(n))$ data, then X has a lower bound of $\Omega(f(n))$.*

For example, consider the sequential search problem.

sequential search
input: an iterator for a sequence S of n elements **output:** an element x of S containing `<A SPECIFIC PROPERTY>`, or `None` if no such element exists

The output x is one datum whose space complexity is only $O(1)$, so the trivial lower bound based on the size of output is $\Omega(1)$. This is an unhelpful lower bound since every problem takes at least constant time to solve and has a truly-trivial lower bound of $\Omega(1)$. However, any correct sequential search algorithm must be capable of examining all n elements. The mere act of observing n elements takes $O(n)$ time, so we have a lower bound of $\Omega(n)$ for sequential search.

Lemma 50. *The sequential search problem has a lower bound of $\Omega(n)$ time.*

Proof. In order to be a correct algorithm, any sequential search algorithm must be capable of returning `None`. In particular it must return `None` when none of the input elements in S has the desired property. In order to be certain of this, and avoid false negatives, a correct algorithm must examine each of the n elements of S. Therefore every correct sequential search algorithm must have a worst-case time complexity of $O(n)$ or slower. □

This logic is sound, but we must be careful to only apply it to situations when we can be absolutely sure that an algorithm necessarily examines some of its input. Some algorithms are clever about examining only a small subset of their input which is relevant. The binary search problem is the prime example.

binary search
input: a vector V of n comparable elements in non-decreasing order, and a query value q **output:** the index i such that `V[i]` $= q$ or `None` if no such index exists

It might be tempting to argue that every binary search algorithm must examine all n elements of its input, the same as for the sequential search problem. However this is actually untrue. Indeed the central idea of the classical binary search algorithm is to rule out half of the input at every step, and thereby ignore almost all of the input. The binary search algorithm examines only $O(\log n)$ input elements, avoiding the other $n - O(\log n)$ elements entirely. The binary search algorithm has worst-case time complexity of $O(\log n)$, which establishes an upper bound of $O(\log n)$ time for that problem, and rules out the possibility of a $\Omega(n)$ lower bound. The lower bound for the binary search problem is actually $\Omega(\log n)$, so we have a tight $\Theta(\log n)$ bound for this problem.

Claim 19. *The binary search problem has a tight bound of* $\Theta(logn)$.

12.5 The Tight Bound for Sorting

Earlier we alluded to the fact that there is a tight $\Theta(n \log n)$ bound for the sorting problem. We derived the merge sort algorithm, with its $O(n \log n)$ worst-case time complexity, back in Section 8.4. So we have an upper bound of $O(n \log n)$ for sorting; in order to establish this tight bound we need to prove a corresponding $\Omega(n \log n)$ lower bound.

This tight bound is important for three reasons. First, as we have argued elsewhere, the sorting problem is fundamental and applies both directly to many practical programming problems. In addition, as we saw in Section 10.2, many other practical problems reduce to sorting. Second, as described in the introduction to this chapter, when we establish a tight bound for a problem it means that algorithm design for that problem has reached a satisfying conclusion, and we would like to attain that closure for the sorting problem. Third, as we will see in Section 12.6, we can leverage this lower bound for sorting to establish lower bounds for other, different problems.

As discussed in Section 12.3, proving a lower bound for sorting is daunting because our proof arguments can only use properties that are shared by all conceivable sorting algorithms. Our approach is to focus on the information that any sort must "learn" in order to solve the problem.

Every correct sort must be capable of starting from a (likely-unordered) vector V and "learning" which permutation will re-order V into non-decreasing order. We deliberately avoid peering into all the other work that a sort must do, such as actually moving elements around and forming the output vector. Instead we focus only on the act of "learning" how to permute the unsorted sequence V into a sorted sequence stored in S.

We introduce one assumption, that a sort learns about this permutation by comparing pairs of elements. In other words, that the algorithm makes a series of comparisons $a < b$ for elements $a, b \in V$, and uses the outcome of these comparisons to guide the sorting process. This is true of all of the sorting algorithms we have studied. Selection sort makes these comparisons while searching

for the least unsorted element. Merge sort makes these comparisons during the merge phase. Quick sort makes these comparisons during the partition phase. Nearly all practical sorting algorithms are comparison-based although *radix sort* is one notable exception.

Theorem 7. *The sorting problem has a lower bound of* $\Omega(n \log n)$ *that applies to all comparison-based sorting algorithms.*

Proof. Let A be an arbitrary decision-based sorting algorithm. By assumption A is a correct algorithm, so it must re-order the elements of V into a sorted vector S in non-decreasing order. To achieve this ordering A must necessarily decide upon a permutation p of V, such that when V is permuted according to p, the resulting sequence is in proper non-decreasing order.

V has n elements, so there are $n!$ candidate permutations that might possibly be a valid p. The best case for A is when A must make few comparisons and the worst case is when A must make very many comparisons. In the best case all elements of V are equal, so all $n!$ permutations are valid. In a more moderate case a few elements of V are tied, and there are several valid permutations. In the worst case for A, all elements of V are distinct, and there exists one and only one permutation that is valid. We are analyzing for the worst-case time complexity of an arbitrary A, so henceforth we assume that all elements of V are distinct and there exists exactly one valid permutation p.

Let a and b be the first two elements of V that are compared by A. As discussed above $a \neq b$ so either $a < b$ or $a > b$. When $a < b$, we know that a must precede b in S, so S has the form

$$S = \ldots a \ldots b \ldots .$$

Of the $n!$ permutations of V, exactly half fit this pattern where a comes before b. Symmetrically, when $a > b, S$ has the form

$$S = \ldots b \ldots a \ldots$$

and only the other half of the permutations fit this pattern. This comparison between a and b gives A enough information to decrease the size of the set of candidate permutations by half. If A makes best use of this information, it must now consider only $\frac{n!}{2}$ of the remaining permutations.

This decrease-by-half pattern continues. The second comparison of elements could provide A enough information to halve the candidate set again to $\frac{n!}{4}$ permutations, the third comparison could halve the candidate set to $\frac{n!}{8}$ permutations, and so on. As we know from the master theorem and analysis of binary search, a decrease-by-half decision process narrows a field of k candidates down to one result after only $O(\log k)$ decisions. Therefore A makes $O(\log(n!))$ comparisons.

To complete the proof we must simplify the expression $\log(n!)$. By the definition of factorial,

$$\log(n!) = \log(n \times (n-1) \times (n-2) \times \ldots \times (1)).$$

Recall the log rule

$$\log_b(x \times y) = (\log_b x) + (\log_b y)$$

(yes! log rules are useful!) so

$$\log(n!) = (\log n) + (\log(n-1)) + (\log(n-2)) + \ldots + (\log(1)).$$

We have n terms each as large as $\log n$, so by arguments symmetrical to those of the triangular sum rule in Lemma 15,

$$\log(n!) \in \Omega(n \log n).$$

In summary, any arbitrary decision-based sorting algorithm A must perform enough comparisons to be able to pick one correct permutation out of a field of $n!$ candidate permutations. A may perform many other steps aside from these comparisons, but the time spent comparing elements constitutes a lower bound for the total amount of time spent by the algorithm. If A uses the outcomes of its comparisons to greatest effect in a binary search-like process, it performs at least $\Omega(\log(n!))$ comparisons. By properties of asymptotic notation, $\Omega(\log(n!)) = \Omega(n \log n)$. □

In Section 8.4 we derived merge sort and proved that its time complexity is $O(n \log n)$, which together with Theorem 7 establishes a tight bound for the sorting problem.

Corollary 7. *The sorting problem has a tight bound of* $\Theta(n \log n)$ *that applies to all comparison-based sorting algorithms.*

There we have it! The fastest possible efficiency class for sorting is $O(n \log n)$, we know about an $O(n \log n)$ sorting algorithm, so the quest for faster sorting algorithms is officially over.

12.6 Reduction Arguments

The information-based argument for the lower bound for sorting may be sound, but it does not give us much of a hint for how to prove additional lower bounds. The sorting problem has a very specific structure wherein solving the problem involves choosing a permutation of the problem input and practically every sorting algorithm makes its choice through a series of binary decisions. Few other interesting problems have that structure. It is not clear at all how we might apply this kind of logic to prove a lower bound for the minimum spanning tree problem or change making problem, for example. The fundamental challenge is that it is difficult to pinpoint steps that *every* correct algorithm for a given problem (aside from sorting) *must* necessarily perform. As argued in Section 12.3, it is difficult to prove a negative.

Our workaround is to "flip the script" and use a different line of logic that involves proving a positive "there exists" condition. Act I of this text taught us several patterns for designing algorithms, so we are well-equipped to prove "there exists" a particular kind of algorithm. A *reduction argument* is a kind of proof that involves only one piece of creative problem solving, which is to show that there exists a specific kind of reduction algorithm.

A reduction argument uses the *proof by contradiction* proof technique. Recall the steps of a proof by contradiction:

1. Let p be the proposition that we wish to prove true.

2. Suppose for the sake of contradiction that p is false, or in terms of Boolean logic, suppose that $\neg p$ is true.
3. Prove that $\neg p$ implies another proposition q is true.
4. Prove that $\neg p$ implies that $\neg q$ is also true.
5. Since q and $\neg q$ cannot be simultaneously true, we know that $\neg p$ is false, and p is true.

To make this more concrete, suppose that we wish to prove that the problem of converting an unordered vector into a balanced binary search tree has a lower bound of $\Omega(n \log n)$, the same lower bound as for sorting.

vector-to-binary search tree conversion
input: a vector V of $n \geq 0$ orderable elements **output:** the root of a balanced binary search tree of height $O(\log n)$ containing the elements of V

To fit this into the proof by contradiction template, we define propositions

$$p = \text{“tree conversion has a lower bound } \Omega(n \log n)\text{”}$$

and

$$q = \text{“sorting has a lower bound } \Omega(n \log n)\text{”}.$$

We already know q is true; what remains is to prove that $\neg p$ implies $\neg q$. We may rephrase p more precisely as

$$p = \forall \text{ tree conversion algorithm } A, \text{ the time complexity of } A \text{ is } O(n \log n) \text{ or slower .}$$

The negation of p is

$$\begin{aligned} \neg p &= \neg(\forall \text{ tree conversion algorithm } A, \text{ the time complexity of } A \text{ is } O(n \log n) \text{ or slower }) \\ &= \exists \text{ tree conversion algorithm } A \text{ such that } \neg(\text{ the time complexity of } A \text{ is } O(n \log n) \text{ or slower }) \\ &= \exists \text{ tree conversion algorithm } A \text{ such that the time complexity of } A \text{ is faster than } O(n \log n) \end{aligned}$$

Our proof by contradiction will begin by supposing $\neg p$, that there exists some tree conversion algorithm A with time complexity faster than $O(n \log n)$. This algorithm A is opaque; we know that it is an algorithm, that solves the tree conversion problem faster than $O(n \log n)$, and that we can run it. However we do not know anything more specific about A.

The goal of the proof is to show $\neg q$, which, analogous to p, may be rephrased

$$\exists \text{ sorting algorithm } S \text{ such that the time complexity of } S \text{ is faster than } O(n \log n).$$

This is the part of the proof that involves some creative problem solving. We need to show that “there exists” a sorting algorithm faster than $O(n \log n)$, in order to illustrate a contradiction. The mathematical fact that we have at our disposal is the supposition that A exists. We can use this fact by designing and explaining a reduction algorithm that solves the sorting problem by reduction to tree conversion. If the time complexity of this algorithm is faster than $O(n \log n)$, then we have a sorting algorithm faster than $O(n \log n)$, which is precisely the kind of contradiction that we seek.

Lemma 51. *The tree construction problem has a lower bound of* $\Omega(n \log n)$.

Proof. Suppose for the sake of contradiction that there exists some tree construction algorithm A that is faster than $O(n \log n)$. We may use A to design the following sorting algorithm.

```
def S(V):
    tree = A(V)
    sorted = new Vector()
    for each element x in an inorder traversal of tree:
        sorted.add_back(x)
    return sorted
```

Recall that an inorder traversal of a binary search tree visits elements in least-to-greatest order; therefore `sorted` is in non-decreasing order and S is a correct sorting algorithm.

The construction of `sorted` with the `for` loop takes $O(n)$ time. Let $T(n)$ be the time complexity of A; then the total time complexity of S is $O(T(n) + n)$. We know that $T(n)$ is not in $O(n \log n)$, so regardless of whether $T(n)$ or n is the dominating term, the time complexity of S is faster than $O(n \log n)$. Therefore S is a sorting algorithm that is faster than $O(n \log n)$, which contradicts the lower bound of $\Omega(n \log n)$ for sorting. □

While this proof is not exactly easy to write, it is significantly more straightforward than is the proof for sorting. As promised, the trickiest part of the proof is showing that "there exists" an algorithm. In particular that, if we have an impossibly-fast algorithm for tree construction, that would allow for an impossibly-fast sorting algorithm, which is contradictory.

This proof technique can be generalized to novel problems aside from tree conversion. To prove a lower bound of $\Omega(n \log n)$ for a problem X :

1. Suppose for the sake of contradiction that there exists an algorithm A that solves X faster than $O(n \log n)$ time.
2. Show pseudocode for an example of a sorting algorithm that sorts correctly by calling A and spending less than $O(n \log n)$ time on "overhead" steps outside of A.
3. Observe that your algorithm is a sorting algorithm that is faster than $O(n \log n)$, which is a contradiction.

This technique can be generalized even further, to establish lower bounds aside from $\Omega(n \log n)$. Let $T(n)$ be the lower bound that you want to prove, and let K be a problem with a known lower bound of $\Omega(T(n))$. (K stands for "known.") To prove a lower bound of $\Omega(T(n))$ for a problem X :

1. Suppose for the sake of contradiction that there exists an algorithm A that solves X faster than $O(T(n))$.

2. Show pseudocode for an example of an algorithm that solves K by calling A and spending less than $O(T(n))$ time on overhead steps outside of A.

3. Observe that your algorithm solves the known problem K in time faster than K's established lower bound of $\Omega(T(n))$, which is a contradiction.

We must emphasize that the direction of reduction is both crucially important, and counter-intuitive. When we used reduction as an algorithm design pattern in Chapter 10, we designed an algorithm that solves a new problem by calling into an already-known algorithm. However, the example algorithm in a reduction proof goes in the *opposite* direction: it solves a problem with an already-known lower bound (e.g. sorting) by calling into an algorithm for the problem whose lower bound we are in the process of proving (e.g. tree conversion). This opposite-ness is a consequence of the negation $\neg p$ that we deliberately introduced as part of the proof-by-contradiction approach. In order to show a contradiction, we need to design an algorithm according to backwards principles. Keeping the direction of the reduction straight may be a challenge, but on balance this is less of a challenge than the alternative of trying to prove every non-trivial lower bound with an information-theoretical argument.

Exercises

12-1. Prove that the square matrix construction problem has a trivial lower bound of $\Omega(n^2)$.

square matrix construction
input: a positive integer n and number x **output:** an $n \times n$ matrix with each element equal to x

12-2. Prove that the summation problem has a trivial lower bound of $\Omega(n)$.

summation problem
input: a list X of numbers **output:** the sum (total) of all elements of X

12-3. Prove that a priority search queue may have either an insert operation that is faster than $O(\log n)$, or a pop-and-remove-min operation that is faster than $O(\log n)$, but not both. Recall that priority search queues were described in Section 10.4, and that the Fibonacci heap structure's insert operation takes $O(1)$ time while its pop-and-remove-min operation takes $O(\log n)$. *Hint:* Use a reduction argument involving a sorting algorithm that inserts elements into a priority search queue and then pops the elements back out.

12-4. Do some research into state of the art bounds for the *matrix multiplication problem.* What is the trivial lower bound for this problem? What is the upper bound established by the Naïve algorithm for the problem? What is the currently best-known upper bound for the problem? Are experts optimistic or pessimistic about finding a tight bound, and why?

Chapter 13

Intractable Problems

13.1 The Big Idea

So far we have been unilaterally successful at designing algorithms. Every time we have considered a problem, we have been able to adapt one of the algorithm patterns resulting in a correct algorithm. We have been able to use multiple patterns for some problems, such as sorting. Our algorithms have been reasonably efficient, aside from a few of our exhaustive search algorithms.

We will now shift gears and consider problems that *cannot* be solved efficiently by algorithms. We must face an unsettling fact: that some problems cannot be solved efficiently, and some problems cannot be solved at all. We can separate problems into three broad categories:

1. "Easy:" Problems that can be solved reasonably efficiently. The technical term for this category is P.
2. "Hard:" Problems that can be solved, but apparently only by extremely slow algorithms. The hardest of these problems are called *NP-complete*.
3. "Impossible:" Problems that cannot be solved by algorithms. The term for this category is *undecidable*.

Complexity theory is fascinating as a purely academic subject, but it is also of practical relevance. Sometimes a requested software feature corresponds to an NP-complete or undecidable problem. Implementing such features is, for all practical purposes, impossible, so it behooves us to recognize this as early in the development process as possible. Software developers ought to cultivate an intuition for when computational problems might be NP-complete or undecidable, and an ability to prove conclusively when this is in fact the case.

It is convenient to think of a problem, defined by an input and output specification, as a mathematical object. We may then use set terminology and notation to describe categories of problems.

Definition 49. *A* complexity class *is a set of problems.*

13.2 Efficiently-Solvable Problems and P

We first characterize the complexity class (set) of problems which may be solved reasonably efficiently. We choose to use polynomial time complexity as a working definition of "reasonable efficiency."

Definition 50. *An algorithm is* polynomial *if its time complexity is $O(n^k)$ for some constant $k \geq 0$.*

Example 54. *If an algorithm's time complexity is $O(1), O(\sqrt{n}) = O(n^{.5}), O(n), O(n^2)$, or $O(n^3)$, the algorithm is polynomial.*

Example 55. *If an algorithm's time complexity is exponential e.g. $O(2^n)$, or factorial e.g. $O(n!)$, the algorithm is* not *polynomial.*

This definition coheres with our empirical results. We have implemented and empirically analyzed algorithms, and those with $O(1), O(n)$, and $O(n^2)$ time complexities ran in a "reasonable" amount of time. Meanwhile those with $O(2^n), O(n!)$, or slower ran in an "unreasonable" amount of time.

Note that $\log n$ factors are polynomial according to our definitions.

Lemma 52. *If an algorithm's time complexity is $O(n^k \log n)$, the algorithm is polynomial.*

Proof. For sufficiently large n,

$$\begin{aligned} \log n &< n \\ (n) \times \log n &< (n) \times n \\ n \log n &< n^2 \\ O(n \log n) &\subseteq O(n^2). \end{aligned}$$

More generally, $O(n^k \log n) \subseteq O(n^{k+1})$, which is a polynomial time complexity of order $(k+1)$, which is constant. □

So the fast sorting algorithms with time complexity $O(n \log n)$, and the optimal versions of the Prim-Jarník and Dijkstra's algorithms with time complexity $O(m + n \log n)$, are polynomial.

It may seem curious that the order k of the polynomial $O(n^k)$ may be arbitrarily large; for example, $O(n^{100})$ is a very slow time complexity, yet counts as polynomial and hence "reasonably efficient." It turns out that polynomial running times slower than $O(n^3)$ are rarely seen in practical algorithms. While it is possible to synthesize an algorithm with 100 nested `for` loops and time complexity $O(n^{100})$, algorithms designed according to our patterns that run in polynomial time rarely run any slower than $O(n^3)$. It is difficult to prove this fact rigorously at this point, but intuitively, when an algorithm has a polynomial time complexity slower than $O(n^3)$, there is almost always a way to speed it up to $O(n^3)$ or faster by reduction to data structure operations, or by using a better pattern. We saw this happen with the Prim-Jarník algorithm, which started out at $O(n^2 m) \approx O(n^4)$ time,

and improved to $O(m + n \log n) \approx O(n^2)$ time with the clever use of the Fibonacci heap data structure.

Another reason why polynomial algorithms tend to have $O(n^3)$ time complexity or better, is that, if a problem can be solved efficiently, it probably reduces to one of the following problems:

1. sorting, which may be solved in $O(n \log n)$ time;
2. matrix multiplication, which may be solved in $O(n^3)$ time (or even faster); or
3. maximum flow, which may be solved in $O(n^3)$ time.

The complexity class P is the set of all problems that may be solved by polynomial algorithms.

Definition 51 (P)**.** *The complexity class P is the set of problems*

$$P = \{X \mid \textit{ there exists a polynomial algorithm that solves } X\}.$$

13.2.1 Proving That a Problem Is In P

According to this definition, proving that a problem is in P is straightforward.

Lemma 53. *For any problem X, if there exists some polynomial algorithm* `Alg` *that solves X, then $X \in P$.*

Proof. The Lemma follows from Definition 51. □

So we may prove that any problem X is in P according to the following process:

1. Design an algorithm `Alg` that solves X.
2. Prove that `Alg` runs in polynomial time.
3. Conclude that $X \in P$.

This means that every algorithm analysis we have performed, that resulted in a polynomial running time, may be seen as a proof that the corresponding problem is in P.

Corollary 8. *The following problems are in P:*

1. *summation,*
2. *sequential search,*
3. *sorting,*

4. *binary search,*

5. *minimum spanning tree,*

6. *non-negative single source shortest paths,*

7. *median finding, and*

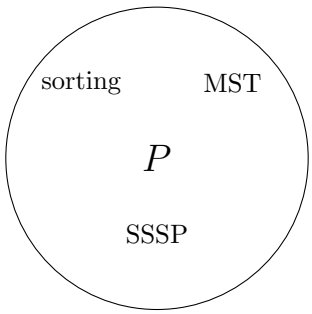
8. *set intersection.*

Our algorithms for circuit satisfaction and traveling salesperson did not run in polynomial time, so we cannot say that those problems are in P. However, we cannot say that those problems are certainly outside of P either. Just because we tried to design a polynomial algorithm for these problems and failed to do so, does not mean that no such algorithm exists.

Designing an algorithm in order to prove that a problem is in P is a different goal from that of designing an algorithm to implement and execute. When designing algorithms in the context of complexity proofs, we seek an algorithm that is easy prove to be correct and polynomial. The order of its polynomial time complexity is inconsequential, so there is no point in exerting time and energy toward improving polynomial time complexities. So, for example, if we were out to prove that the sorting problem is in P, we could stop at the first draft of selection sort that is clearly correct and has $O(n^2)$ time complexity; there is no need to develop faster and more complex algorithms such as in-place selection sort or merge sort. Likewise, for the purposes of proving that the minimum spanning tree problem is in P, our first $O(n^2m)$ algorithm suffices, and there is no need to delve further into the vagaries of Fibonacci heaps.

13.2.2 Venn Diagram

Since complexity classes are sets, it is customary to visualize them using Venn diagrams. The following diagram illustrates the complexity relationships that we have discussed up to this point. So far, we have seen that the class P exists and contains such problems as sorting, minimum spanning tree (MST), single source shortest paths (SSSP), and others.



13.3 Unsolvable Problems and Decidability

If problems lie on a spectrum where one end corresponds to being easy to solve, then other end naturally corresponds to problems that are *impossible* to solve. The technical term for "unsolvable" is *undecidable.*

Definition 52. *Problem X is* decidable *if there exists an algorithm that solves X, or is* undecidable *if there does not exist an algorithm that solves X.*

The terms "decidable" and "undecidable" are perhaps a bit awkward; they are artifacts of the tradition within complexity theory of focusing on *decision problems*. A decision problem is a problem whose output is a Boolean; intuitively this kind of problem corresponds to making a yes-no decision. When a decision problem is solvable, it is "decidable," otherwise it is "undecidable." "Decidable" is almost a synonym for "solvable."

Recall that, in order to qualify as an algorithm, a process must satisfy the clarity, correctness, and termination properties. Problems are undecidable when there cannot exist any process that solves the problem while satisfying all these properties. Most often, an undecideable problem could either be solved by a process that produces correct output, but may fall into an infinite loop; or by a process that always terminates, but is sometimes incorrect; but not both.

13.3.1 Proving That a Problem Is Decidable

Proving that a problem is decidable is even simpler than proving that a problem is in P. By the definition of decidability, all we need to show is that there exists a correct algorithm that solves the problem; we do not even need to analyze its efficiency.

Corollary 9. *Every problem in P is decidable.*

While our exhaustive search algorithms for circuit satisfaction and traveling salesperson did not prove those problems to be in P, they are correct algorithms, and so proved those problems to be decidable.

Corollary 10. *The circuit satisfaction and traveling salesperson problems are decidable.*

13.3.2 The Halting Problem

Perhaps the most widely known undecidable problem is the *halting problem* first posed by Alan Turing [51].

halting
input: a procedure `proc` and input `I`
output: `True` if `proc` halts when run with input `I`, or `False` otherwise

The `proc` part of the input is a digital data structure representing a procedure, such as a string containing Python source code, or an array of bytes storing compiled and executable machine code. We would certainly appreciate an algorithm that solves this problem, since it would allow compilers to detect infinite loops rather than allowing them to go unnoticed. Unfortunately, no such algorithm can exist.

Our proof that the halting problem is undecidable relies on the fact that a program may be fed into itself as its own input. For example, suppose the file `script.py` contains the following executable Python script.

```
#! /usr/bin/env python3

import sys

print('my input was: ' + sys.stdin.read())
```

This process echoes its input back as output.

```
$ ./script.py
kowabunga
my input was: kowabunga
```

It is perfectly legal to execute `script.py` and redirect the contents of `script.py` to standard input.

```
$ ./script.py < script.py
my input was: #! /usr/bin/env python3

import sys

print('my input was: ' + sys.stdin.read())
```

Another example of using a program as input to itself is seen in the bootstrapping process of compiling the GCC compiler from source code. The build involves three phases:

1. The existing system compiler, called `cc` and presumed to be inferior to GCC, is used to compile GCC. This results in an executable binary we call `gcc1`, which does a better job of generating optimized code than does `cc`.
2. `gcc1` is used to compile GCC's source code again to produce a second version of the compiler we call `gcc2`. `gcc2` implements the same algorithms as `gcc1`, but was compiled with a superior compiler (`gcc1`), so `gcc2` should produce the same output as `gcc1` and do so more efficiently.

3. Finally, `gcc2` compiles the source code one last time to produce a third binary `gcc3`, which is compared with `gcc2` as a correctness check.

So, it is fair to expect that a procedure can be run on itself as input.

Theorem 8. *The halting problem is undecidable.*

Proof. We argue by contradiction, so suppose that there exists an algorithm `halts` that solves the halting problem. We may summarize the behavior of `halts` as

$$\texttt{halts(proc, I)} = \begin{cases} \texttt{True} & \text{if } \texttt{proc} \text{ halts on input } \texttt{I} \\ \texttt{False} & \text{if } \texttt{proc} \text{ loops forever on input } \texttt{I}\,. \end{cases}$$

We may create a new process `self_halts` that uses `halts` to determine whether a procedure halts when run on itself.

```
def self_halts(proc):
  return halts(proc, proc)
```

We may summarize `self_halt` 's behavior as

$$\texttt{self\_halts(proc)} = \begin{cases} \texttt{True} & \text{if } \texttt{proc} \text{ halts on input } \texttt{proc} \\ \texttt{False} & \text{if } \texttt{proc} \text{ loops forever on input } \texttt{proc}\,. \end{cases}$$

We introduce another procedure `contrary` that calls `self_halts` and then does the opposite of what `proc` would do if run on itself.

```
def contrary(proc):
  if self_halts(proc):
    # infinite loop
    while True:
      pass
  else:
    # halt
    return
```

In general, `contrary` 's behavior is

$$\texttt{contrary(proc)} = \begin{cases} \text{loops forever} & \text{if } \texttt{proc} \text{ halts on input } \texttt{proc} \\ \text{halts} & \text{if } \texttt{proc} \text{ loops forever on input } \texttt{proc}\,. \end{cases}$$

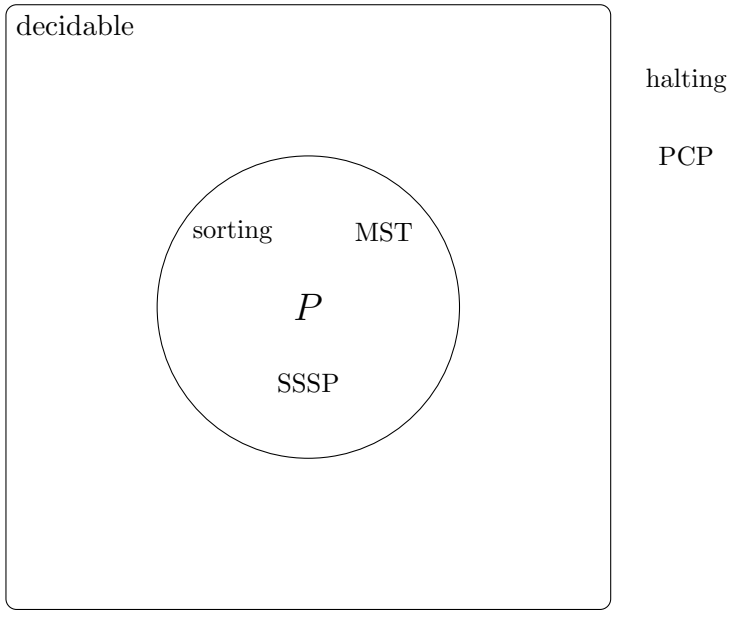
Now consider the case when `contrary` is run on itself. By substitution, its behavior is

$$\texttt{contrary(contrary)} = \begin{cases} \text{loops forever} & \text{if } \texttt{contrary} \text{ halts on input } \texttt{contrary} \\ \text{halts} & \text{if } \texttt{contrary} \text{ loops forever on input } \texttt{contrary}\,. \end{cases}$$

Both cases are clear contradictions! So our supposition that `halts` exists is unsound, and there does not exist an algorithm solving the halting problem. □

13.3.3 Updated Venn Diagram

We now know that all problems in P are decidable, while the halting problem and Post's correspondence problem are not.



13.4 Verifiable Problems and NP

The third major category of problem are the NP-complete problems, which are easy enough to be solvable, yet apparently too hard to be solved efficiently. Their definition is reminiscent of the

fable of Goldilocks, who found one bowl of porridge to be "too hot," a second to be "too cold," and a third to be "just right." The definition of NP-completeness involves two properties, one that says that an NP-complete problem is "not too hard," and another that says that an NP-complete problem is "not too easy." The complexity class NP corresponds to the set of problems that are "not too hard." Specifically, NP is the set of problems for which verifier algorithms have polynomial time and space complexity.

Definition 53 (NP)**.** *For any problem X, $X \in NP$ if there exists an exhaustive search algorithm that solves X, such that each candidate solution fits in polynomial space and the verification algorithm is polynomial.*

Recall that we designed an exhaustive search algorithm for the circuit satisfaction problem; each candidate is a set of at most n variable names, and the verification algorithm has time complexity $O(n)$. Likewise, we have an exhaustive search algorithm for the traveling salesperson problem, where each candidate is a permutation of n vertices using $O(n)$ space, and the verification algorithm takes $O(n)$ time.

Corollary 11. *The circuit satisfaction and traveling salesperson problems are in NP.*

13.4.1 Proving That a Problem Is In NP

We may use the following process to prove that a problem X is in NP.

1. Define the data type of a candidate solution to the problem.
2. Prove that each candidate fits in polynomial space.
3. Design a verification algorithm for the problem.
4. Prove that the verification algorithm is polynomial.
5. Conclude that $X \in NP$.

13.4.2 P Is a Subset of NP

NP is the set of problems that may be verified, but not necessarily fully solved, in polynomial time, while P is the set of problems that may be completely solved in polynomial time. Intuitively, if we can solve a problem in polynomial time, we ought to be able to complete the more limited task of verifying the problem in polynomial time too. This intuition is correct.

Lemma 54. *$X \in P$ implies $X \in NP$.*

Proof. By the definition of P, there exists some algorithm `solve_X` that returns a correct solution to X in polynomial time. We can create a verifier algorithm by reduction to `solve_x`. If every input to X has a unique correct solution, we define `verify_x` as

```
def verify_x(instance, candidate):
  if candidate == solve_X(instance):
    return True
  else:
    return False
```

Otherwise, we use `solve_x` to generate a correct solution and determine whether that solution is better than `candidate`.

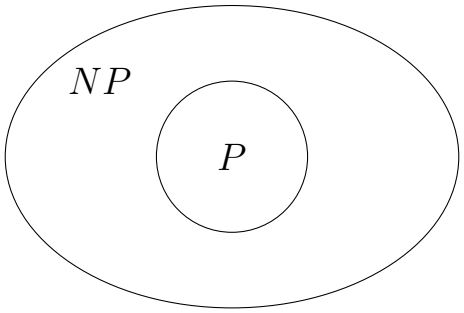
```
def verify_x(instance, candidate):
  solution = solve_x(instance)
  if <solution IS BETTER THAN candidate>:
    return False
  else:
    return True
```

Under the modest assumption that the `<solution IS BETTER THAN candidate>` part takes polynomial time, both versions of `verify_x` are correct and polynomial. Hence there exists a polynomial verification algorithm for X, and by the definition of NP, $X \in NP$. □

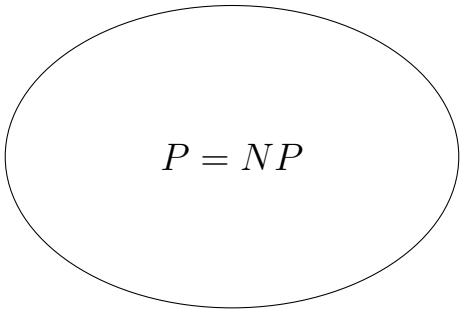
Corollary 12. $P \subseteq NP$.

13.4.3 Is P Equal to NP?

Corollary 12 established that $P \subseteq NP$, or in other words, P is a *subset or equal* to NP. It could be that $P \subset NP$, or that $P = NP$. Surprisingly, we do not presently know which is the case! Both of the Venn diagrams below are consistent with the fact that $P \subseteq NP$. Either:



or:



The current state of the art is that there is no proof of which diagram is correct.

We have reason to believe the $P \subset NP$ scenario would be less surprising, and is perhaps more likely. So for the purpose of discussion we shall draw our Venn diagrams as if $P \subset NP$. However we reiterate that $P = NP$ may be the correct scenario. We shall return to this issue and the implications of $P \subset NP$ and $P = NP$ in Section 13.8.

13.5 Hard Problems and NP-Hardness

Recall that when a problem is in NP, it is "not too hard." A problem is "not too easy" when it is at least as hard as any problem in NP. We say that problem H is at least as hard as problem E when E reduces to H.

Definition 54. *Problem E is* polynomial time reducible *to problem H, written*

$$E \overset{P}{\to} H,$$

if there exists an algorithm that solves E by reduction to an algorithm that solves H, where any pre-processing and post-processing takes polynomial time.

The problems that are at least as hard as any in NP are called *NP-hard.*

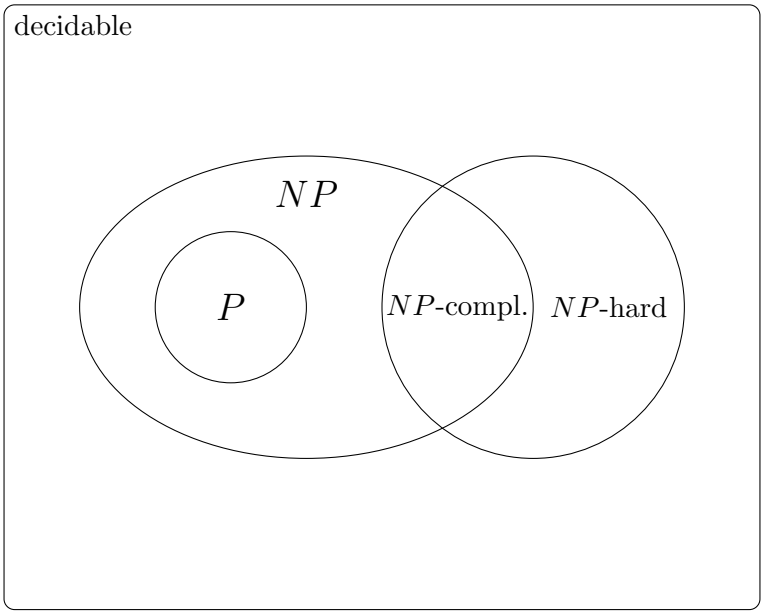
Definition 55 (NP-hard)**.** *Problem H is* NP-hard *if, for any problem $E \in NP$, $E \overset{P}{\to} H$.*

When a problem is both "not too easy" and "not too hard," it is NP-complete.

Definition 56 (NP-complete)**.** *Problem H is NP-*complete *if*

1. *$H \in NP$, and*
2. *H is NP-hard.*

13.5.1 Updated Venn diagram



13.6 A First NP-Complete Problem

Now that we have defined what it means to be NP-complete, we establish that at least one NP-complete problem actually exists. The Cook-Levin theorem is a landmark result in computer science that establishes the existence of such a problem. The original theorem proved that the *Boolean satisfaction* problem is NP-complete; instead we use the same form of argument to prove that the related circuit satisfaction problem is NP-complete [24].

circuit satisfaction problem
input: a Boolean circuit C with k variables and n vertices **output:** a Boolean assignment A that sastisfies C, or `None` if no such assignment exists

Recall that ordinarily we prove a lower bound with a reduction argument. But a reduction argument says that the fact that one problem is already known to be hard implies that a novel problem is hard as well. Before we can use reduction arguments, we need to prove that at least one problem is

hard from first principles. That is the role of the Cook-Levin theorem [15] [37]. The theorem proves that circuit satisfaction is NP-complete, by direct argument to the definition of NP-completeness. Once we have that result, we can use more straightforward reduction arguments to prove that additional problems are NP-complete [23].

Theorem 9 (Cook-Levin)**.** *The circuit satisfaction problem is NP-complete.*

Proof (sketch). We must show that

1. circuit satisfaction $\in NP$, and
2. circuit satisfaction is NP-hard: for any $E \in NP, E \xrightarrow{P}$ circuit satisfaction.

circuit satisfaction $\in NP$

A candidate solution for circuit satisfaction is an assignment of a Boolean value to each of the k variables in the circuit. This list of k Booleans takes $O(k)$ space. A circuit with n vertices has at most n variables (and probably far fewer), so $k \leq n$ and $O(k) \subseteq O(n)$. So each candidate takes polynomial space. As discussed in the context of exhaustive search, a Boolean circuit may be evaluated in $O(n^2)$ or $O(n)$ time. An assignment may be verified by evaluating the input circuit and testing whether the output is `True` ; so each candidate may be verified in $O(n)$ time, which is polynomial.

circuit satisfaction is NP-hard

This part of the proof is both deep and technical, so we first outline our argument, then present it again in detail.

Outline: By assumption $E \in NP$, so there must be some verifier algorithm `verify_E` that runs in polynomial time. This algorithm takes polynomial bits of input and outputs one Boolean, corresponding to whether or not the input bits represent a correct solution for E. We can build a massive Boolean circuit that *emulates* the execution of `verify_E` : the circuit shall have one variable for each bit of input to `verify_E` , and the circuit's output shall be equivalent to that computed by `verify_E` . We construct the circuit by "compiling" `verify_E` into Boolean circuitry. Any polynomial algorithm involves a polynomial number of steps; every step involves executing a constant number of CPU instructions; and every CPU instruction can be executed by evaluating a constant number of circuits each of constant size. So the circuit implementing an entire run of `verify_E` has polynomial size — a very large polynomial, to be sure, but a polynomial nonetheless. To solve E, we find a satisfying assignment for our constructed circuit. A satisfying assignment is equivalent to an input to `verify_E` that is verified to be a correct solution, or more simply, an assignment satisfying the circuit is a solution to E.

The following high-level pseudocode outlines our reduction algorithm.

```
def solve_E(instance_E):
  emulator = <CONVERT verify_E INTO A BOOLEAN CIRCUIT>
  assignment = solve_CSAT(emulator)
  solution_E = <CONVERT assignment INTO A CANDIDATE FOR E>
  return solution_E
```

Argument: E is in NP, so there exists an algorithm `verify_E` that takes a candidate solution C for E, and returns `True` when C is a correct solution or `False` otherwise. By the definition of NP, `verify_E` runs in polynomial time, and C occupies polynomial space. Suppose that we may inspect the pseudocode of `verify_E`, and know the specific time complexity $O(n^t)$ of `verify_E` and space complexity $O(n^s)$ of C. (Technically the definition of NP only guarantees that a verifier exists and has polynomial complexity; the content of the algorithm and polynomials need not be known, they must merely exist. We ignore this technicality, which is one reason this is a "sketchy" proof.)

Our reduction algorithm inspects `verify_E` and its complexity bounds, and produces a Boolean circuit B with the properties that

- each variable x_i in B corresponds to one bit of memory storing the input to `verify_E`;
- the output of B for a given assignment is identical to the output of `verify_E` on the corresponding input;
- the total size of B is polynomial; and
- B may be constructed in polynomial time.

To generate B, we model the execution of the steps specified by `verify_E`'s pseudocode. By the definition of algorithms, `verify_E` must be specified clearly enough to be implemented. By the definition of the standard model of computation, any algorithm may be executed by performing a sequence of individual steps, where each step involves $O(1)$ individual instructions. Our definition of an instruction is that it is a computation that reads $O(1)$ words from memory, then overwrites $O(1)$ words of memory, and the value of any written bit may be computed by a Boolean circuit with one variable per read word and $O(1)$ total size. A word is comprised of $W \in O(1)$ bits. Therefore, for every bit of memory, we can construct a Boolean circuit of $O(1)$ size that computes the new state of the bit after executing one CPU instruction. By assumption `verify_E` makes at most $O(n^t)$ steps, and so can modify at most $O(n^t)$ distinct memory addresses, and its space complexity is at most $O(n^t)$. So we can construct a family of $O(n^t)$ instruction-bit circuits; each computes the new state of one bit of memory after executing one step of `verify_E`'s execution, in "parallel" so to speak.

The collective behavior of the family of circuits described in the previous paragraph is to read the $O(n^t)$ bits of memory used by `verify_E`, and compute the new state of those bits after executing

one CPU instruction. We may copy the family into a second instance, and define the inputs to the second instance as the outputs of the first instance. The resulting super-family of circuits is twice as large, and computes the memory configuration resulting from executing *two* CPU instructions in sequence. A third copy yields an even larger emulator for three instructions. If we chain a total of $O(n^t)$ copies together, we obtain an emulator for the entire execution of `verify_E`. Note that this process depends critically on the fact that `verify_E` terminates after polynomial steps.

We consider the size of the `emulator` circuit. It is constructed out of $O(n^t)$ layers, where each layer corresponds to circuitry executing one CPU instruction. Each layer includes an $O(1)$-sized circuit that for each of $O(n^t)$ memory bits. So the total size of the circuit is $O(n^t \times n^t) = O(n^{2t})$ which is polynomial. Converting a CPU instruction to these circuits may be done in polynomial time, so the process of creating `emulator` takes polynomial time.

Our reduction algorithm uses this circuit as a circuit satisfaction instance, and solves it. The result is a circuit assignment that causes `verify_E` to return `True`, or `None` if no such assignment exists. By construction this assignment is equivalent to a candidate for E. So we may return the candidate (or `None`) as a solution for E.

Thus, for any arbitrary problem E, there exists an algorithm that solves E by reduction to circuit satisfaction. The pre-processing and post-processing time of the algorithm is polynomial, so $E \overset{P}{\rightarrow}$ circuit satisfaction, and circuit satisfaction is NP-hard. □

13.7 Proving NP-Completeness By Reduction

The Cook-Levin theorem is quite long and technical, and the thought of reproducing something like that every time we prove a problem to be NP-complete is daunting. Fortunately our definitions are crafted in such a way that more direct and concise proofs of NP-completeness are possible. Intuitively, a problem is NP-complete if it is at least as hard as any other problem in NP. If some novel problem H is at least as hard as an existing problem E known to be NP-complete, then H is at least as hard as E, and by the transitivity of the "at least as hard as" relation, H is NP-complete too.

Lemma 55. *If $H \in NP$, E is NP-complete and $E \overset{P}{\rightarrow} H$, then H is NP-complete.*

Proof. In order to prove that H is NP-complete, we need to show that

1. $H \in NP$, and
2. H is NP-hard.

By assumption $H \in NP$. For H to be NP-hard, it must be the case that, for any arbitrary problem $X \in NP$, $X \overset{P}{\rightarrow} H$. Any such reduction may be achieved through a two-step process: first reduce X to E, then reduce E to H.

```
def solve_X(instance_X):
  instance_E = <CONVERT instance_X INTO AN INSTANCE OF E>
  instance_H = <CONVERT instance_E INTO AN INSTANCE OF H>
  solution_H = solve_H(instance_H)
  solution_E = <CONVERT solution_H INTO A SOLUTION FOR E>
  solution_X = <CONVERT solution_E INTO A SOLUTION FOR X>
  return solution_X
```

By assumption E is NP-complete, so E is NP-hard, $X \overset{P}{\to} E$, and the reduction from arbitrary problem X to E is possible and polynomial. By assumption $E \overset{P}{\to} H$, so the reduction from E to H is also possible and polynomial. So `solve_X` is correct, its total pre-processing and post-processing time is polynomial. □

This Lemma establishes a framework for proving that a novel problem H is NP-complete:

1. Prove that $H \in NP$, by showing that a candidate solution for H is of polynomial size and may be verified in polynomial time.
2. Find an existing NP-complete problem E whose structure is similar to that of H.
3. Prove that $E \overset{P}{\to} H$, by showing that E may be solved by reducing an instance of E to an instance of H, then calling a hypothetical algorithm that solves H.
4. Conclude that H is NP-complete.

This kind of reduction proof depends on some problem E already being NP-complete, so we could not have used this form of argument to prove the Cook-Levin theorem; that would involve circular logic. However, since we proved that circuit satisfaction is NP-complete using only definitions and first principles, the circuit satisfaction problem may now take the role of E in reduction proofs. We may prove that another problem is NP-complete by reduction to circuit satisfaction, and then we will have two known NP-complete problems. Either of those two problems may then take the role of E, making future reduction proofs easier. Every time we add a problem to the portfolio of known NP-complete problems, later proofs of NP-completeness become that much easier.

13.7.1 3-SAT

We now introduce a new problem, $3-SAT$, and prove it NP-complete by reduction.

Definition 57. *A* 3-CNF Boolean formula *over n variables $X = \{x_0, x_1, \ldots, x_{n-1}\}$ is a Boolean expression of the form*

$$C_0 \ \textit{and}\ C_1 \ \textit{and}\ \ldots \ \textit{and}\ C_m$$

where each C_i *is a* clause *of the form*

$$(L_0 \ \texttt{or}\ L_1 \ \texttt{or}\ L_2)$$

and each L_j *is a* literal *of the form*

$$x_k$$

or

$$\texttt{not}\ x_k.$$

Example 56. *The following is a 3-CNF formula with five variables and three clauses:*

$$(x_0 \ \texttt{or} \ \texttt{not}\ x_1 \ \texttt{or}\ x_2) \ \texttt{and}\ (x_0 \ \texttt{or}\ x_3 \ \texttt{or}\ x_4) \ \texttt{and}\ (\ \texttt{not}\ x_2 \ \texttt{or} \ \texttt{not}\ x_3 \ \texttt{or}\ x_4).$$

The $3-SAT$ problem [32] is analogous to the circuit satisfaction problem.

$3-SAT$
input: a string S of length n containing a 3-CNF Boolean formula F **output:** an assignment A that satisfies F, or `None` if no such A exists

Lemma 56. $3-SAT$ *is* NP*-complete.*

Proof. We will prove that $3-SAT$ is NP complete by reducing circuit satisfaction to $3-SAT$. So we must show that

1. $3-SAT \in NP$, and
2. circuit satisfaction $\overset{P}{\rightarrow} 3-SAT$.

$\underline{3-SAT \in NP}$

A candidate solution to $3-SAT$ is an assignment for each of the m variables of F. As you may recall from our exhaustive search algorithm for circuit satisfaction, an assignment may be represented by a list `assignment` of m Boolean values, where `assignment[i]` = `True` if and only if x_i is assigned `True` . Such a list has length $O(m)$. Each literal encoded in S uses at least one character, so $m < n$ (actually m is likely to be much smaller than n), so the size of each candidate is $O(n)$ which is polynomial.

The following high-level pseudocode describes a verification algorithm for $3-SAT$. It parses the string S into a data structure representing F, then verifies that every clause is satisfied. A clause is satisfied if any of its literals evaluate to true, which is the case when it is negated and its variable is assigned `False` , or when it is not negated and its variable is assigned `True` .

```
def verify_3SAT(S, assignment):
  clauses = parse_clauses(S)
  for clause in clauses:
    clause_satisfied = False
    for literal in clause.literals:
      if literal.is_negated != assignment[literal.index]:
        clause_satisfied = True

    if not clause_satisfied:
      return False

  return True
```

The `parse_clauses` function is responsible for translating the characters of S into a list of clause objects, where each clause object has exactly three literal objects; and each literal object has an integer for its index and a Boolean for whether it is negated. We shall not describe `parse_clauses` in detail, but standard lexing techniques from the field of compiler design may be used to implement `parse_clauses` elegantly in $O(n)$ time. Or, since 3-CNF syntax does not allow for nested parenthesis, S may be parsed with routine string manipulation techniques: split S into clauses delimited by `and` , split each clause into literals delimited by `or` , and search each literal token for a `not` character and integer index. The second approach takes $O(n^2)$ time, or $O(n)$ with some care. After parsing S, the remainder of `verify_3SAT` takes $O(n)$ time. So `verify_3SAT` takes polynomial time, and $3-SAT \in NP$.

circuit satisfaction $\xrightarrow{P}$ $3-SAT$

We must describe an algorithm that solves circuit satisfaction by reducing that problem to $3-SAT$. This algorithm must use the following structure:

```
def solve_CSAT(circuit):
  S = <CONVERT circuit INTO AN EQUIVALENT 3-CNF FORMULA>
  formula_assignment = solve_3SAT(S)
  circuit_assignment = <CONVERT formula_assignment INTO AN EQUIVALENT CIRCUIT
                        ASSIGNMENT>
  return circuit_assignment
```

In addition to using this structure, we need that the `<CONVERT circuit...>` and `<CONVERT formula_assignment..>` steps take polynomial total time, and that the size of `formula_assignment` is polynomial.

Now let n to be the number of vertices in `circuit` . We convert `circuit` into an equivalent

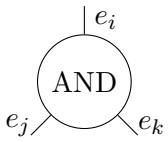
$3-CNF$ formula in three phases:

1. Convert each vertex of `circuit` into one or two *improper clause* objects, for a total of at most $2n \in O(n)$ improper clauses.
2. Convert each improper clause object into at most 8 proper 3-CNF clause objects, producing a valid 3-CNF formula with $O(8n) = O(n)$ proper clauses.
3. Convert the list of proper clause objects into a string of length $O(n)$.

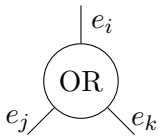
An improper clause is a Boolean expression of three variables that may involve nested parenthesis and operators aside from `or`, while a proper clause is in the strict $(L_0$ `or` L_1 `or` $L_2)$ format required for $3-SAT$.

In phase 1, we create a variable y_i for each *edge* e_i in `circuit`. We also create a *pad variable* p. We then convert each internal vertex of `circuit` into one or two improper clauses. The clauses are all considered to be connected by the `and` operator.

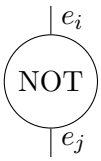
- An AND vertex

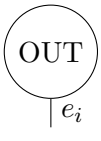
  converts to a clause $(e_i$ `==` $(e_j$ `and` $e_k))$.
- An OR vertex

  converts to a clause $(e_i$ `==` $(e_j$ `or` $e_k))$.
- A NOT vertex

  converts to two clauses $(e_i$ `==` `not` e_j `or` $p)$ `and` $(e_i$ `==` `not` e_j `or` `not` $p)$.
- The OUT vertex

  converts to two clauses $(e_i$ `or` p `or` $p)$ `and` $(e_i$ `or` `not` p `or` `not` $p)$.

We forced the clauses in the NOT and OUT cases to involve the pad variable p so that every improper clause references exactly three variables, a property which will be useful shortly. Either p is assigned `True`, in which case the left clause is always true and the right clause is meaningful; or vice-versa.

Observe that every variable vertex x_i in the circuit has an outgoing edge; let $y_{L(i)}$ be the variable on that edge, and the *leaf variables* be the set of all such variables. An assignment to the $Y = \{y_0, \ldots\}$ variables in the formula must include an assignment for each leaf variable, and also assignments for all non-leaf variables. By construction, a formula assignment satisfies our *formula* if and only if that assignment is consistent with the logical relationships defined by the original *circuit.* So this phase of our conversion preserved the satisfiability of the input circuit.

In phase 2 we convert each of the improper clauses from the previous phase into $O(1)$ proper 3-CNF clauses, again while preserving satisfiability. For each clause, we evaluate the truth table for that clause. As previously noted, each clause references exactly three variables, so each truth table has exactly 3 inputs and 8 rows.

Example 57. *The truth table for an AND clause* $(e_i \texttt{ == } (e_j \texttt{ and } e_k))$ *is:*

e_i	e_j	e_k	$(e_i \texttt{ == } (e_j \texttt{ and } e_k))$
False	False	False	True
False	False	True	True
False	True	False	True
False	True	True	False
True	False	False	False
True	False	True	False
True	True	False	False
True	True	True	True

Observe that, logically,

"the original improper clause is satisfied"

is equivalent to the predicate

"we are in any of the cases that evaluate to `True`"

which is equivalent to

"we are not in any of the cases that evaluate to `False`"

which is equivalent to

`not` ("we are in any of the cases that evaluate to `False`").

We can form a Boolean expression for the predicate "we are in any of the cases that evaluate to `False`" by forming a clause for each such case, and connecting them with the `or` operator.

Example 58. *The clauses equivalent to "we are in any of the cases that evaluate to `False`" for an AND vertex are*

$$(\ \texttt{not}\ e_i\ \texttt{and}\ e_j\ \texttt{and}\ e_k)\ \texttt{or}\ (e_i\ \texttt{and}\ \texttt{not}\ e_j\ \texttt{and}\ \texttt{not}\ e_k)\ \texttt{or}\ (e_i\ \texttt{and}\ \texttt{not}\ e_j\ \texttt{and}\ e_k)\ \texttt{or}\ (e_i\ \texttt{and}\ e_j\ \texttt{and}\ \texttt{no}$$

Observe that the clauses of this expression are connected with `or`, and the literals in each clause are connected with `and`. Since the truth table has 8 rows, there are at most 8 clauses.

We may obtain a Boolean expression corresponding to `not` ("we are in any of the cases that evaluate to `False`") by negating the above expression. We eliminate the `not` operator by invoking *DeMorgan's Law,* which states

$$\begin{aligned} \texttt{not}\ (A\ \texttt{and}\ B) &= (\ \texttt{not}\ A)\ \texttt{or}\ (\ \texttt{not}\ B) \\ \texttt{not}\ (A\ \texttt{or}\ B) &= (\ \texttt{not}\ A)\ \texttt{and}\ (\ \texttt{not}\ B). \end{aligned}$$

This results in clauses connected with `and`, where the literals in each clause are connected with `or`. There are still at most 8 clauses.

Example 59. *The clauses equivalent to "we are in any of the cases that evaluate to `False`" for an AND vertex are*

$$\texttt{not}\ \Big((\ \texttt{not}\ e_i\ \texttt{and}\ e_j\ \texttt{and}\ e_k)\ \texttt{or}\ (e_i\ \texttt{and}\ \texttt{not}\ e_j\ \texttt{and}\ \texttt{not}\ e_k)\ \texttt{or}\ (e_i\ \texttt{and}\ \texttt{not}\ e_j\ \texttt{and}\ e_k)\ \texttt{or}\ (e_i\ \texttt{and}\ e_j\ \texttt{a}$$

$$= \texttt{not}\ (\ \texttt{not}\ e_i\ \texttt{and}\ e_j\ \texttt{and}\ e_k)\ \texttt{and}\ \texttt{not}\ (e_i\ \texttt{and}\ \texttt{not}\ e_j\ \texttt{and}\ \texttt{not}\ e_k)\ \texttt{and}\ \texttt{not}\ (e_i\ \texttt{and}\ \texttt{not}\ e_j\ \texttt{and}\ e_k)\ \texttt{a}$$

$$= (e_i\ \texttt{or}\ \texttt{not}\ e_j\ \texttt{or}\ \texttt{not}\ e_k)\ \texttt{and}\ (\ \texttt{not}\ e_i\ \texttt{or}\ e_j\ \texttt{or}\ e_k)\ \texttt{and}\ (\ \texttt{not}\ e_i\ \texttt{or}\ e_j\ \texttt{or}\ \texttt{not}\ e_k)\ \texttt{and}\ (\ \texttt{not}\ e_i\ \texttt{or}\ \texttt{n}$$

Observe that these clauses are now in 3-CNF form, and are logically equivalent to the improper clause that we started with.

Phase 2 converts each of the $O(n)$ improper clauses from phase 1 into at most 8 proper clauses, so after phase 2 we have $O(n)$ proper clauses. Phase 3 need only print these proper clause objects, connected with `and`, into a string representation; this takes $O(n)$ time. So, the size of S is polynomial, and the time spent pre-processing is polynomial.

In the post-processing phase we must convert an assignment A_F in the 3-CNF formula into an equivalent assignment A_C in the input circuit. If `solve_3SAT` returns `None`, then `solve_CSAT` returns `None` too. Otherwise, we convert A_F into an equivalent A_C, which is easy: for each variable x_i in the circuit, we assign x_i the same value assigned to variable $y_{L(i)}$ in A_F.

conclusion

We have shown that circuit satisfaction $\stackrel{P}{\rightarrow}\ 3-SAT$; circuit satisfaction is NP-complete, thus $3-SAT$ is NP-hard. We also showed that $3-SAT \in NP$, hence $3-SAT$ is NP-complete. □

13.7.2 Other NP-Complete Problems

It is possible to prove, by reduction, that the traveling salesperson, longest common subsequence, clique, and vertex cover problems are all NP-complete.

13.8 P Versus NP Revisited

Any NP-complete problem H is NP-hard, so $E \xrightarrow{P} H$ for any $E \in NP$. Any H is also in NP; so any NP-complete problem may serve the role of E or H in the expression $E \xrightarrow{P} H$. As a consequence, every NP-complete problem is polynomial time reducible to every other NP-complete problem.

Lemma 57. *For any NP-complete problems X and Y,*

$$X \xrightarrow{P} Y$$

and

$$Y \xrightarrow{P} X.$$

Consequently, if *any* NP-complete problem may be solved in polynomial time, then *all* NP-complete problems may be solved in polynomial time.

Lemma 58. *If there exists an NP-complete problem X such that $X \in P$, then every NP-complete problem is in P.*

This gives us an angle of attack for proving that $P = NP$: if we design a polynomial algorithm for any one of the hundreds of known NP-complete problems, then $P = NP$. Recall that to show $P \subset NP$, we must prove that, for all algorithms that could ever exist, none solves an NP-complete problem in polynomial time. To be a sound proof, this must be true not only for the algorithms and data structures that are currently known, but also for those that have yet to be discovered.

At present we have no proof for $P = NP$ and no proof for $P \subset NP$, so we cannot say either is true with any certainty. However we could try to make a Bayesian inference [1] into which alternative might be *more likely* based upon empirical evidence.

In a world where $P = NP$,

- all of the NP-complete problems are solvable in polynomial time;
- computer scientists have so far failed to prove $P = NP$ due to a massive collective oversight: despite all of the algorithm patterns and data structures at our disposal, the long list of NP-complete problems, and the fact that practically all computer science students entertain the notion of solving one of them in polynomial time, everyone has failed to do so;
- any exhaustive search algorithm with a polynomial verifier can be solved in polynomial time; in essence, the loop that iterates through candidate solutions, and the number of candidate solutions, is of no significance to the algorithm's time complexity; and

- breaking widely used cryptography ciphers reduces to integer factorization, and integer factorization is in NP, so cryptography can be broken by efficient algorithms.

In a world where $P \subset NP$,

- none of the NP-complete problems are solvable in polynomial time;
- computer scientists have so far failed to prove $P \subset NP$ because the "for all algorithms which could ever exist" quantifier makes it a very difficult thing to prove, perhaps at the limits of human intelligence or requiring mathematical results that do not yet exist;
- exponential time algorithms, such as the exhaustive search algorithms we developed earlier, are the best that we can expect for NP-complete problems, regardless of any advances in algorithm design; and
- the security of cryptography is unaffected.

The author posits that the $P \subset NP$ scenario sounds more realistic, and we *believe* that it is the more *likely* reality. A 2002 poll of 100 complexity theory researchers [25] corroborates that position.

conjectured answer to $P \stackrel{?}{=} NP$	**votes**
$P \subset NP$	61
$P = NP$	9
other	8
no opinion	22

So 61% of respondents, and 78% of those who registered an opinion, feel that $P \subset NP$, which is a majority. However, there is clearly significant doubt around this issue on the part of experts; there is no scientific consensus.

Our reasoned opinion is that, as a practical matter, we should go about developing software as if $P \subset NP$, and consider solving NP-complete problems to be impractical; but, we should simultaneously remain open-minded to the possibility, even if remote, that $P = NP$.

Exercises

13-1. For each of the following efficiency classes, state whether the class is polynomial or not. Justify your answers.

(a) $O(n^3)$
(b) $O(\sqrt[3]{n^2})$
(c) $O(\frac{n^2}{\log \log n})$

13-2. Visit the website A compendium of NP optimization problems. Find three NP-complete problems that interest you, aside from the ones we have already studied. For each problem, state

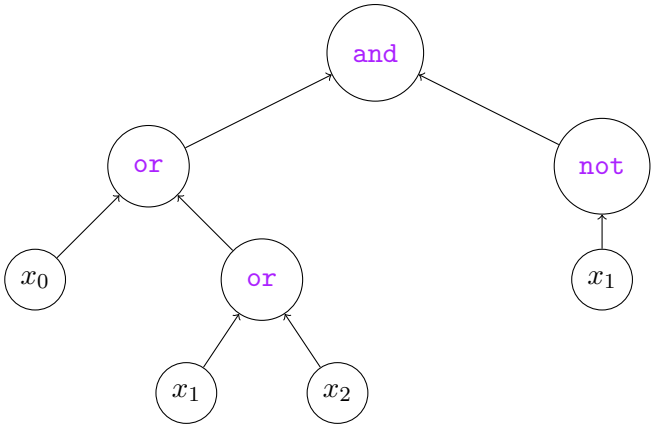
(a) the **input** and **output** of the problem;

(b) the data type of a candidate solution; and

(c) a hypothetical real-world application of the problem.

13-3. Suppose that we removed the requirement that, in order for a problem to be in NP, the size complexity of one of its candidate solutions is polynomial. What problem would that pose for a proof of the Cook-Levin theorem?

13-4. Convert the following circuit satisfaction instance into an equivalent $3-SAT$ instance using the algorithm outlined in the proof of Lemma 56. Show your work, including all the improper clauses and proper 3-CNF clauses.



13-5. Implement the `parse_clauses` function for $3-SAT$.

13-6. Prove that each of the following problems is in NP.

(a)

traveling salesperson (TSP)
input: a graph $G = (V, E)$ where each edge $e \in E$ has a numeric weight w_e **output:** a Hamiltonian cycle in G of minimum total weight, or `None` if no such cycle exists

(b)

subset sum problem
input: a list X of n distinct integers, and a target integer k **output:** a subset $S \subseteq X$ such that $k = \sum_{x \in S} x$, or `None` if no such S exists

(c)

set partition problem
input: list of X of n distinct positive integers **output:** a pair of lists (L, R) such that L and R form a set partition $X = L \cup R$ and $\sum_{l \in L} l = \sum_{r \in R} r$, or `None` if no such partition exists

(d)

knapsack problem
input: a list of n item objects I, where each item `i` has a positive `i.value` and positive integer `i.weight` ; and a weight limit $W \geq 0$ **output:** a list K of items from I, such that the total weight of all items in K is at most W, and the total value of the items in K is maximized

(e)

maximum clique
input: an $n \times n$ adjacency matrix M representing an undirected graph $G = (V, E)$ with n vertices **output:** a k-clique C such that k is maximum

(f)

minimum vertex cover
input: an $n \times n$ adjacency matrix M representing an undirected graph $G = (V, E)$ with n vertices **output:** the smallest C such that C is a vertex cover for G

13-7. The $2 - SAT$ problem is the same as $3 - SAT$, except that each clause contains exactly two literals (L_0 `or` L_1) instead of three. Prove that $2 - SAT \in P$.

13-8. The $4 - SAT$ problem is the same as $3 - SAT$, except that each clause contains exactly four literals (L_0 `or` L_1 `or` L_2 `or` L_3) instead of three. Prove that $4 - SAT$ is NP-complete. Is it true that $k - SAT$ is NP-complete for any $k \geq 3$?

13-9. Suppose that set partition is NP-complete; prove that subset sum is NP-complete.

set partition problem
input: list of X of n distinct positive integers **output:** a pair of lists (L, R) such that L and R form a set partition $X = L \cup R$ and $\sum_{l \in L} l = \sum_{r \in R} r$, or `None` if no such partition exists

subset sum problem
input: a list X of n distinct integers, and a target integer k **output:** a subset $S \subseteq X$ such that $k = \sum_{x \in S} x$, or `None` if no such S exists

13-10. Suppose that vertex cover is NP-complete; prove that maximum clique is NP-complete.

minimum vertex cover
input: an $n \times n$ adjacency matrix M representing an undirected graph $G = (V, E)$ with n vertices **output:** the smallest C such that C is a vertex cover for G

maximum clique
input: an $n \times n$ adjacency matrix M representing an undirected graph $G = (V, E)$ with n vertices **output:** a k-clique C such that k is maximum

13-11. Prove that the halting problem is NP-hard. Is the halting problem NP-complete?

Act III

Appendices

Appendix A

Bibliography

[1] Bayesian inference. http://en.wikipedia.org/wiki/Bayesian_inference.

[2] Clang: a c language family frontend for llvm. https://clang.llvm.org/.

[3] The computer language benchmarks game. http://benchmarksgame.alioth.debian.org/.

[4] Lambda calculus. https://en.wikipedia.org/wiki/Lambda_calculus.

[5] Lambda calculus. https://en.wikipedia.org/wiki/Travelling_salesman_problem.

[6] Random-access machine. https://en.wikipedia.org/wiki/Random-access_machine.

[7] Scientific method. https://en.wikipedia.org/wiki/Scientific_method.

[8] Turing machine. https://en.wikipedia.org/wiki/Turing_machine.

[9] Adel'son-Vel'skii, G., and Landis, E. M. An algorithm for the organization of information. *Sov. Math. Dokl. 3* (1962), 1259—1262.

[10] Alexander, C., Ishikawa, S., Silverstein, M., i Ramió, J. R., Jacobson, M., and Fiksdahl-King, I. *A pattern language*. Gustavo Gili, 1977.

[11] Andersson, A. Balanced search trees made simple. *Algorithms and Data Structures* (1993), 60–71.

[12] Aragon, C. R., and Seidel, R. G. Randomized search trees. In *Foundations of Computer Science, 1989., 30th Annual Symposium on* (1989), IEEE, pp. 540–545.

[13] Borůvka, O. O jistém problému minimálním.

[14] Church, A. A set of postulates for the foundation of logic. *Annals of mathematics* (1932), 346–366.

[15] Cook, S. A. The complexity of theorem-proving procedures. In *Proceedings of the third annual ACM symposium on Theory of computing* (1971), pp. 151–158.

[16] Cormen, T. H., Leiserson, C. E., Rivest, R. L., and Stein, C. *Introduction to algorithms*. MIT press, 2009.

[17] Dijkstra, E. W., et al. A note on two problems in connexion with graphs. *Numerische mathematik 1*, 1 (1959), 269–271.

[18] Elkner, J., Downey, A. B., and Meyers, C. How to Think Like a Computer Scientist: Interactive Edition. http://interactivepython.org/courselib/static/thinkcspy/index.html.

[19] Eppstein, D. Design and analysis of algorithms: Lecture notes for february 6, 1996. www.ics.uci.edu/~eppstein/161/960206.html.

[20] Eric Lehman, F. Thomas Leighton, A. R. M. Mathematics for Computer Science. https://courses.csail.mit.edu/6.042/spring17/mcs.pdf.

[21] Felleisen, M., Findler, R. B., Flatt, M., and Krishnamurthi, S. How to Design Programs. http://homedirs.ccs.neu.edu/matthias/HtDP2e/.

[22] Fredman, M. L., and Tarjan, R. E. Fibonacci heaps and their uses in improved network optimization algorithms. *Journal of the ACM (JACM) 34*, 3 (1987), 596–615.

[23] Garey, M. R., and Johnson, D. S. *Computers and intractability*, vol. 174. freeman San Francisco, 1979.

[24] Goodrich, M. T., and Tamassia, R. *Algorithm design: foundation, analysis and internet examples*. John Wiley & Sons, 2006.

[25] Hemaspaandra, L. A. Sigact news complexity theory column 36. *ACM SIGACT News 33*, 2 (2002), 34–47.

[26] Hinze, R. Purely functional 1-2 brother trees. *Journal of Functional Programming 19*, 6 (2009), 633–644.

[27] Hinze, R., and Paterson, R. Finger trees: a simple general-purpose data structure. *Journal of functional programming 16*, 2 (2006), 197–217.

[28] Hoare, C. A. R. Algorithm 64: Quicksort. *Communications of the ACM 4*, 7 (1961), 321.

[29] Hugue, M. M. Lecture 15: Quicksort. http://www.cs.umd.edu/~meesh/351/mount/lectures/lec15-quicksort.pdf.

[30] Jarník, V. O jistém problému minimálním. *Práca Moravské Prírodovedecké Spolecnosti 6* (1930), 57–63.

[31] Johnson, S. M. Generation of permutations by adjacent transposition. *Mathematics of Computation 17*, 83 (1963), 282–285.

[32] Karp, R. M. Reducibility among combinatorial problems. In *Complexity of computer computations*. Springer, 1972, pp. 85–103.

[33] Knuth, D. E. Big omicron and big omega and big theta. *ACM Sigact News 8*, 2 (1976), 18–24.

[34] Knuth, D. E. *The art of computer programming: Volume 3: Sorting and Searching*. Addison-Wesley Professional, 1998.

[35] Kozen, D. Proof of the master method. http://www.cs.cornell.edu/courses/cs3110/2011sp/lectures/lec19-master/mm-proof.pdf.

[36] Kruskal, J. B. On the shortest spanning subtree of a graph and the traveling salesman problem. *Proceedings of the American Mathematical society 7*, 1 (1956), 48–50.

[37] Levin, L. A. Universal sequential search problems. *Problemy peredachi informatsii 9*, 3 (1973), 115–116.

[38] Levitin, A. *Introduction to the design & analysis of algorithms*. Boston: Pearson,, 2012.

[39] Mathews, G. B. On the partition of numbers. *Proceedings of the London Mathematical Society 1*, 1 (1896), 486–490.

[40] McDowell, G. L. *Cracking the coding interview: 189 programming questions and solutions*. CareerCup, 2019.

[41] Morin, P. Open Data Structures. `http://opendatastructures.org/`.

[42] Ottmann, T., and Wood, D. 1-2 brother trees or avl trees revisited. *The computer journal 23*, 3 (1980), 248–255.

[43] Prim, R. C. Shortest connection networks and some generalizations. *The Bell System Technical Journal 36*, 6 (1957), 1389–1401.

[44] Rosenberg, A. L., and Heath, L. S. *Graph separators, with applications*. Springer Science & Business Media, 2001.

[45] Sedgewick, R. *Stanford University Ph. D.* PhD thesis, Dissertation, 1975.

[46] Skiena, S. S. *The algorithm design manual: Text*, vol. 1. Springer Science & Business Media, 1998.

[47] Sleator, D. D., and Tarjan, R. E. Self-adjusting binary search trees. *J. ACM 32*, 3 (July 1985), 652–686.

[48] Strang, G. Calculus. `https://ocw.mit.edu/resources/res-18-001-calculus-online-textbook-spring-2005/textbook/`.

[49] Tardos, E., and Kleinberg, J. Algorithm design, 2005.

[50] Trotter, H. F. Algorithm 115: Perm. *Commun. ACM 5*, 8 (Aug. 1962), 434–435.

[51] Turing, A. M. On computable numbers, with an application to the entscheidungsproblem. *J. of Math 58*, 345-363 (1936), 5.

[52] Wolfgang, P. Design patterns for object-oriented software development. *Reading Mass* (1994).

Appendix B

Index