DSC 40B Course Notes

Justin Eldridge

Contents

1 Introduction **5**

1.1 The Wisdom of the Crowd . . . 6
1.2 The Components of a Data Science Problem . . . 8
1.2.1 The Computational Problem . . . 9
1.2.2 The Algorithm . . . 10
1.2.3 The Implementation and Beyond . . . 11
1.2.4 “Solving” a Data Science Problem . . . 12
1.3 Correctness . . . 12
1.3.1 Loop Invariants . . . 13
1.3.2 Proving a Loop Invariant . . . 14
1.3.3 A More Complicated Algorithm . . . 16
1.3.4 Nested Loop Invariants . . . 18
1.4 Efficiency . . . 20
1.4.1 Time Complexity Analysis . . . 20
1.4.2 The Time Complexity of Computing the Mean . . . 21
1.4.3 Scalability and Big-Theta . . . 23
1.4.4 Nested Loops . . . 23
1.4.5 Nested Loops with Dependent Ranges . . . 25
1.5 Big-Θ and Friends . . . 27
1.5.1 Big-Θ . . . 27
1.5.2 Big-O . . . 29
1.5.3 Big-Ω . . . 30
1.5.4 Properties . . . 30
1.6 Brute-Force Algorithms . . . 32
1.6.1 Search Problems and Optimization . . . 32
1.6.2 Wisdom of the Crowd, Revisited . . . 34
1.6.3 Exponential Time: Clustering . . . 37
1.6.4 Factorial Time: Doctor Orders . . . 39

2 Sorting **43**
2.1 Selection Sort . . . 44
2.1.1 Intuition . . . 44
2.1.2 An In-Place Selection Sort . . . 45
2.1.3 Time Complexity . . . 48
2.2 Digression on Recursion . . . 48
2.2.1 Recursive Algorithm Design . . . 49
2.2.2 Correctness . . . 50
2.2.3 Time Complexity . . . 51
2.3 Merge Sort . . . 55
2.3.1 Designing `mergesort` . . . 55
2.3.2 Understanding Merge Sort . . . 62
2.3.3 Correctness . . . 63
2.3.4 Time Complexity . . . 64
2.4 The Value of Sorted Data . . . 67
2.4.1 Searching . . . 67
2.4.2 Picking In-Flight Movies . . . 72

3 Graphs **77**
3.1 Definitions . . . 78
3.1.1 Directed Graphs . . . 79
3.1.2 Undirected Graphs . . . 81
3.1.3 Terminology Involving Edges . . . 82
3.1.4 Paths and Connectedness . . . 83
3.2 Computer Representations of Graphs . . . 86
3.2.1 Adjacency Matrices . . . 86
3.2.2 Adjacency Lists . . . 88
3.2.3 Dictionaries of Sets . . . 88
3.2.4 Node Attributes . . . 90
3.3 Breadth-First Search . . . 91
3.3.1 Search Strategies . . . 91
3.3.2 The Algorithm . . . 94
3.3.3 Properties of BFS . . . 98
3.3.4 Time Complexity of BFS . . . 99
3.3.5 Shortest Paths . . . 100
3.3.6 The Breadth-First Search Tree . . . 105
3.4 Depth-First Search . . . 107
3.4.1 The Algorithm . . . 109
3.4.2 Time Complexity . . . 112
3.4.3 Start and Finish Times . . . 113
3.4.4 DFS Forest . . . 119

3.4.5 Topological Sort 120
3.4.6 Edge Classification 122
3.5 Weighted Graphs and Minimum Spanning Trees 123
3.5.1 Distance Graphs 124
3.5.2 Minimum Spanning Trees 124
3.5.3 Clustering 126

Chapter 1

Introduction

Contents

1.1 The Wisdom of the Crowd **6**
1.2 The Components of a Data Science Problem **8**
1.2.1 The Computational Problem 9
1.2.2 The Algorithm 10
1.2.3 The Implementation and Beyond 11
1.2.4 “Solving” a Data Science Problem 12
1.3 Correctness **12**
1.3.1 Loop Invariants 13
1.3.2 Proving a Loop Invariant 14
1.3.3 A More Complicated Algorithm 16
1.3.4 Nested Loop Invariants 18
1.4 Efficiency **20**
1.4.1 Time Complexity Analysis 20
1.4.2 The Time Complexity of Computing the Mean 21
1.4.3 Scalability and Big-Theta 23
1.4.4 Nested Loops 23
1.4.5 Nested Loops with Dependent Ranges 25
1.5 Big-Θ and Friends **27**
1.5.1 Big-Θ 27
1.5.2 Big-O 29
1.5.3 Big-Ω 30
1.5.4 Properties 30
1.6 Brute-Force Algorithms **32**
1.6.1 Search Problems and Optimization 32

1.6.2 Wisdom of the Crowd, Revisited 34
1.6.3 Exponential Time: Clustering 37
1.6.4 Factorial Time: Doctor Orders 39

1.1 The Wisdom of the Crowd

Suppose that you are a contestant on a game show. In order to win the prize – free tuition at UCSD for one year – you must guess the height of the Geisel Library to within 10 feet. You're unsure of the answer, but, luckily, you're allowed to poll the television audience for their best guesses. Aggregating their responses is likely to produce a guess that is more accurate than your own – a phenomenon known as the "wisdom of the crowd".[1]

If you adopt this strategy, your next step is to decide how, exactly, to aggregate the audience's guesses. There are many approaches, but a natural one is to view aggregation as **compression**. That is, you wish to find a single number θ which is somehow representative of the data set. Of course, some information is lost by compressing a collection of numbers into a single number, θ; naturally, you wish to choose θ so as to minimize this loss.

More formally, the information lost when a data point x_i is represented by a different number θ can be quantified by choosing a **loss function**. You decide to use a popular choice, the **square loss**:

$$\ell_{\text{sq}}(x_i, \theta) = (x_i - \theta)^2.$$

The smaller the value of $\ell_{\text{sq}}(x_i, \theta)$, the better θ is thought to represent x_i. If you take the position that a good representative of the data set is one which is, on average, a good representative of each data point, then the goal of compression is to find a θ which minimizes the average square loss:

$$L_{\text{sq}}(\theta) = \frac{1}{n} \sum_{i=1}^{n} \ell_{\text{sq}}(x_i, \theta).$$

This is a very important step on your way towards winning a year's tuition: you have transformed the vague strategy of aggregating the crowd's guesses into a precise **computational problem**:

GIVEN: Real numbers $x_1, \ldots, x_n$.
COMPUTE: A real number θ minimizing the average square loss, L_{sq}.

Having stated the problem, your next task is to find an **algorithm** for producing the solution. And since the television audience consists of thousands, if not millions of people, each contributing one number to the data set, you will need to run your algorithm on the

[1] https://en.wikipedia.org/wiki/Wisdom_of_the_crowd

computer you have available to you. It should be able to produce its result in a reasonable amount of time.

It is not too hard to code up a Python function for computing the value of L_{sq} given some data and a particular θ:

```
def average_square_loss(theta, data):
    total_loss = 0
    for x in data:
        total_loss += (x - theta)**2
    return total_loss / len(data)
```

You can use this function to try several different values of θ in an effort to find that with the smallest average loss. But there are infinitely[2] many real numbers to try, and checking all of them is simply impossible. Your algorithm will have to be a bit more clever.

You remember that minimizing a function can often be done via calculus, and, coincidentally, the loss function you have chosen is differentiable. To find the minimizer of $L_{\text{sq}}(\theta)$, you first take the derivative with respect to θ:

$$\frac{dL_{\text{sq}}}{d\theta} = \frac{d}{d\theta}\left[\frac{1}{n}\sum_{i=1}^{n}(x_i - \theta)^2\right].$$

The derivative operator can be pushed inside of the summation:

$$= \frac{1}{n}\sum_{i=1}^{n}\frac{d}{d\theta}(x_i - \theta)^2.$$

A careful application of the chain rule results in:

$$= \frac{2}{n}\sum_{i=1}^{n}(\theta - x_i).$$

To minimize $L(\theta)$ over all possible θ, you set the derivative to zero and solve for θ:

$$\begin{aligned} &\frac{dL_{\text{sq}}}{d\theta} = 0 \\ \implies &\frac{2}{n}\sum_{i=1}^{n}(\theta - x_i) = 0. \end{aligned}$$

Dividing both sides by two:

$$\implies \frac{1}{n}\sum_{i=1}^{n}(\theta - x_i) = 0.$$

[2]An uncountable infinity.

The summation can be split into two summations:

$$\implies \frac{1}{n}\sum_{i=1}^{n}\theta - \frac{1}{n}\sum_{i=1}^{n}x_i = 0,$$
$$\implies \frac{1}{n}\sum_{i=1}^{n}\theta = \frac{1}{n}\sum_{i=1}^{n}x_i.$$

Since θ is a constant, you can pull it out of the first summation:

$$\implies \frac{\theta}{n}\sum_{i=1}^{n}1 = \frac{1}{n}\sum_{i=1}^{n}x_i,$$
$$\implies \frac{\theta}{n}\cdot n = \frac{1}{n}\sum_{i=1}^{n}x_i.$$
$$\implies \theta = \frac{1}{n}\sum_{i=1}^{n}x_i.$$

In other words, $L(\theta)$ is minimized by choosing θ to be the **mean**, $\frac{1}{n}\sum_{i=1}^{n}x_i$.

This little bit of calculus has turned a seemingly impossible computational problem into an easy one. Instead of computing $L(\theta)$ for every possible value of θ and returning that which produces the smallest value, we can compute the minimizer directly by computing the mean. That is, the earlier computational problem is equivalent to the following:

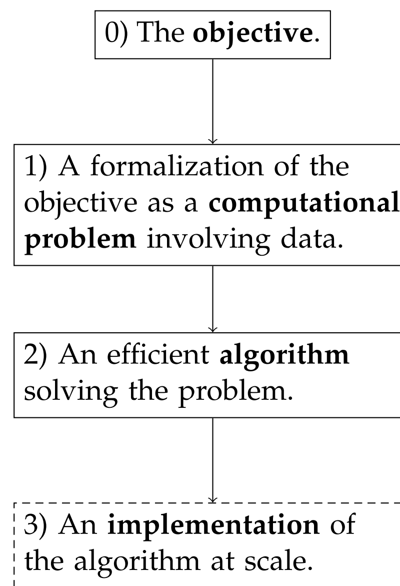
GIVEN: Real numbers $x_1, \ldots, x_n$.
COMPUTE: Their mean, $\frac{1}{n}\sum_{i=1}^{n}x_i$.

This computational problem has a straightforward algorithmic solution, coded in Python below:

```python
def mean(data):
    """Assume len(data) > 0."""
    total = 0
    for x in data:
        total += x
    return total / len(data)
```

With this practical algorithm in hand, you have everything you need in order to aggregate the audience's guesses and estimate your way to a year of free tuition.

1.2 The Components of a Data Science Problem



The game show example of the previous section, while simple, has all of the components of a typical data science problem. Such a problem starts with a high-level **objective**. Above, the objective was to predict the height of the Geisel Library as accurately as possible. Other objectives might be to determine whether an image contains a pedestrian, decide whether a new medical treatment is effective, or to discover trends in how people move around a city.

1.2.1 The Computational Problem

Once the objective has been stated, there are typically many approaches to reaching it. For instance, one person might try to determine the height of Geisel by imagining it next to a structure they do know the height of, while another might rely on luck and take a random guess. The strategy of the data scientist is to gather data and translate the objective into a precise **computational problem**. In the previous section, we opted to gather guesses from the audience and aggregate them by minimizing the average square loss.

There is typically no single way to go from a high-level objective to a precise computational problem, and the choices we make in this transition has major implications for the feasibility of finding a solution and the quality of the outcome. As such, this step often requires careful analysis. For instance, some thought shows that the square loss $\ell_{\text{sq}}(x_i, \theta)$ demands a high price for data points x_i which are far from the proposed θ. As such, the average square loss may be dominated by outliers. Practically, this means that if a member of the audience is a very bad guesser (or worse, is malicious in their response), the aggregated answer obtained by minimized the average square loss will be inaccurate. It might be better, therefore, to pick a loss function that is less sensitive to outliers.

Probability and statistics are often used to translate an objective into a computational problem. For example, we may assume that the guesses of the audience are random variables following some probability distribution whose peak is located at the true height of the Geisel Library. In this view, a natural computational problem is to use the data to estimate the distribution and recover this peak. Furthermore, framing the problem as a statistical one often allows us to compute confidence intervals – if the audience's guess comes with low confidence, we might decide to go with our own best guess instead.

Translating the objective to a computational problem is a crucial step in solving a data science problem, and it is a major focus of DSC 40A. This course will mostly be concerned with the components which follow.

1.2.2 The Algorithm

The next component of the data science problem roadmap is an efficient **algorithm** which solves the computation problem that has been posed. Making the leap from a problem to an algorithm can involve several steps in itself:

Analyze the problem. By analyzing the problem, we hope to understand the nature of solution in a way that helps us with designing or choosing an algorithm. The result of this step is often the transformation of the original problem into an equivalent one that is easier to solve. For instance, in the last section, we analyzed the problem of minimizing the average square loss using calculus. We found that this problem has the same solution as that of computing the mean of a data set.

Design an algorithm. Once we understand the nature of the problem's solution, we begin the process of designing an algorithm for computing it. For instance, knowing that the mean minimizes the average square loss, there is an obvious algorithmic solution: we loop over the numbers in the data set, cumulatively keeping track of their sum, and finally return the sum divided by the size of the set. This is the algorithm implemented by the Python code at the end of the previous section, but it should be noted that this is not the *only* algorithm for computing the mean. In fact, the next section will investigate a different algorithm which is in some ways superior.

The computational problem of computing the mean has a simple and straightforward algorithm associated with it, but this is not always the case. When an efficient algorithm is not so efficient, it is often helpful to try several **algorithm design patterns**. We will encounter some of these patterns, such as **divide-and-conquer**, in this course. Other strategies, like **dynamic programming**, are outside of the scope of DSC 40B, but are maybe worth studying independently – you might encounter them in an algorithms-focused job interview.

Prove the algorithm's correctness. The algorithm we listed above for computing the mean is evidently correct, but the correctness of many algorithms is not so straightforward to assess. For instance, algorithms for sorting a list of numbers can become quite complicated and their correctness can be difficult to prove. Case in point, Timsort, the sorting algorithm used by Python and Java, was found to contain a logical error by a team of computer scientists in 2015[3], over 13 years after it was first implemented. The Python implementation of the algorithm only triggered the error for inputs of a very, very large size (so large that such inputs were never seen), but the Java implementation of Timsort on Android was susceptible to failure for certain lists whose size was as small as 65,536.

[3] de Gouw, Stijn; Rot, Jurriaan; de Boer, Frank S.; Bubel, Richard; Hähnle, Reiner (July 2015). "OpenJDK's Java.utils.Collection.sort() Is Broken: The Good, the Bad and the Worst Case"

We will develop one tool for proving the correctness of algorithms in the next section – the **loop invariant**. Loop invariants are useful for not only proving that an iterative algorithm does what we expect it to do, but also for understanding how the algorithm works. This is important, since one of the most common approaches to designing an algorithm is to modify an existing one – a task which is made possible only if we understand the workings of the algorithm we're trying the modify.

Analyze the algorithm's efficiency. In practice, the correctness of an algorithm alone does not mean that it is useful – it must also be efficient. Therefore, once we have a correct algorithm, we must analyze its usage of computational resources, including processor time and memory. In the coming sections, we will introduce tools and a notation for assessing the efficiency of algorithms.

An algorithm which takes a minute to run on a data set of 1,000 images may take weeks (or longer) to run on a data set of 100,000 images. On such a large data set, this algorithm is effectively useless, and we must return to the algorithm design step in order to produce a faster algorithm. Improved performance often comes at the cost of using a more complicated algorithm design pattern.

But sometimes, no matter how clever we are, we cannot find an efficient algorithm for solving our computational problem. In fact, computer scientists have good reason to believe that some problems[4] are so hard that an efficient algorithm for solving them simply *does not exist*. Moreover, these problems aren't all that exotic – many of the computational problems we might pose in data science fall into this group.

If it turns out that the computational problem itself is too hard to solve, we have several options. First, we can go back and choose a different computational problem. It is sometimes the case that an intractable computational problem becomes efficiently solvable after it is modified slightly. Ideally, the solution of the modified problem retains the important properties of a solution to the original. Second, we may give up on trying to find an exact solution, and settle instead for an *approximate* one. Even then, some problems are so hard that it is thought to be impossible to efficiently *approximate* their solutions.

1.2.3 The Implementation and Beyond

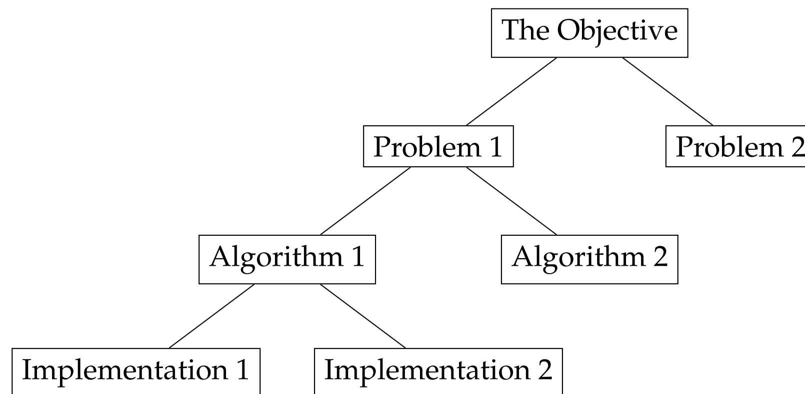
Once the algorithm has been designed an analyzed, it must be implemented. In this simplest case, this just means coding up the algorithm in a programming language, or even carrying it with pencil and paper. In other cases, such as when we have a truly massive data set, we must optimize our code and make infrastructure decisions which allow the implementation to scale. This is by no means an easy task, and it is largely outside of the scope of DSC 40B.

[4]For instance, the problems which are said to be "NP-Hard".

1.2.4 "Solving" a Data Science Problem

We have seen that a data science problem has several components: a high-level objective, a formal computational problem, an algorithm, and an implementation. Solving a data science problem consists in choosing each of these components. Of course, there are many choices to be made: there is more than one way to formalize the objective, each with its positives and negatives; any resulting computational problem may have many algorithms which solve it; and there are many possible implementations of any one algorithm.

As a result, the landscape of solutions takes the form of a many-branching tree:



If a solution to a data science problem is a path in this tree, a data scientist is someone who is comfortable in navigating it. They are able to formalize problems, design and assess algorithms, and implement them. Other disciplines have something to say about each of these steps, of course. Statisticians specialize in posing the computational problem and studying the nature of their solutions, computer scientists focus on designing and analyzing algorithms, and engineers concentrate on implementation. In this view, a data scientist is a generalist.

But such a generalist is necessary, because navigating this tree is rarely a linear process. After trying and failing to find an efficient solution for a particular computational problem, you might decide to solve a different problem altogether, moving back up the tree and down a different branch. This new problem may formalize the same overarching objective, but in a fundamentally different way. This is important to keep in mind, because while *solving* a data science problem involves choosing a path *down* the tree, *interpreting* the output of your solution involves walking back *up* the tree, back to the objective.

1.3 Correctness

Once we have formalized a computational problem involving data, we set about designing an algorithm to solve it. At a minimum, we want our algorithm to be *correct*. One

approach to assessing the correctness of an algorithm is to verify that is produces the correct output for each input in a small set of inputs called **test cases**. While certainly a good practice for testing the implementation of an algorithm, this approach has a big downside: it is always possible that *some* input results in the failure of your algorithm, but your test cases do not include it. Instead, we are interested in *proving* that the algorithm produces the correct output for every possible input.

Proving the correctness of an algorithm in this way has more than just academic benefits – it forces us to understand how the algorithm works at a fundamental level. This is important, since one of the main strategies we have for designing a new algorithm is to modify an existing one – a process that is much easier if we thoroughly understand the algorithm being modified.

1.3.1 Loop Invariants

Consider again the straightforward algorithm for computing the mean. The algorithm's Python implementation is shown below:

```
1 def mean(data):
2     """Assume len(data) > 0."""
3     total = 0
4     for x in data:
5         total += x
6     return total / len(data)
```

If we want to prove that this algorithm is correct, we need to show that its return value, `total / len(data)`, is indeed the mean of the numbers in `data`. We will assume that Python correctly computes the number of items in `data` when `len(data)` is executed, so proving the correctness of the return value boils down to proving that `total` does actually contain the sum of all elements of `data` when execution reaches the last line of the function. Of course, the value of `total` changes over the course of the function's execution, since it is computed iteratively inside a `for`-loop. Therefore, in order to prove that `total` is computed correctly, we need to prove the correctness of the loop.

The primary tool we have for proving the correctness of a loop is the **loop invariant**. As the name suggests, a loop invariant is a statement which is true before the loop executes and immediately after[5] every iteration. A useful loop invariant says something about the value of an important variable once the loop exits. In this case, a useful loop invariant tells us about the value of `total`:

Invariant 1. After the αth iteration, `total` contains the sum of the first α elements of `data`.

[5] Or immediately before every loop – either way.

Remember that a loop invariant should also say something about the status of the program *before* the loop executes, but the above invariant seems to only make a statement about the status of the program *after* each loop. We have a clever way of getting around this: Since the time immediately after iteration $\alpha - 1$ is also the time right before iteration α, we adopt the seemingly-strange convention that the time right before the first iteration of the loop is also the time "immediately after zeroth iteration". Therefore, plugging $\alpha = 0$ into the invariant results in a statement about the status `total` immediately before the loop begins execution.

You may think of a loop invariant as a whole collection of statements, one for each value of α, where α can be any iteration number ranging from 0 (i.e., before the first iteration) to the total number of iterations. For instance, replacing α by 42 yields the statement: After the 42nd iteration, `total` contains the sum of the first 42 elements of `data`. More useful is replacing α by the total number of iterations, n. In this case, the loop invariant tells us that "After all n iterations (i.e., immediately after the loop exits), `total` is the sum of the first n elements of `data` (i.e., all of the elements)".

Which iteration? When writing loop invariants (and discussing loops in general) we must be careful to distinguish the *mathematical* variables from the *implementation* variables; i.e., the ones that appear in the code. To see the potential for confusion, consider the following question: What is printed on the ith iteration of

```
for i in range(10): print(i)?
```

Observe that the question involves two instances of the 9th letter in the English alphabet: i and `i`. The first instance, i, is a *mathematical* variable, while `i` is an *implementation* variable. We typically start counting from 1 in mathematics, so the first iteration corresponds to $i = 1$. But on the $i = 1$ iteration, the *implementation* variable `i` $= 0$. Continuing on, the ith iteration prints the value of $i - 1$. On the other hand, if we confuse i and `i`, we are likely to say that the code prints i on the ith iteration. To avoid this confusion, we will use Greek letters (like α) as mathematical variables, since Greek symbols don't usually appear in code.

1.3.2 Proving a Loop Invariant

Simply stating a loop invariant does not make it true – we must prove that the invariant is indeed correct. And since we can think of a loop invariant as a collection of statements, one for each value of α, it may seem as though we will have to prove each statement in the collection individually. Luckily, we will soon learn a clever mathematical technique for proving all of the statements in one fell swoop.

But to begin, let's try proving all of the statements, one-by-one, starting with the statement that results from taking $\alpha = 0$. This statement reads: "After the zeroth iteration,

`total` contains the sum of the first zero elements of `data`". We understand that "the zeroth iteration" conventionally refers to the time immediately before the first iteration of the loop. At this point in time, `total` has been initialized to zero. Again by convention, the sum of zero elements is zero. Hence `total` is the sum of the first zero elements, and the statement corresponding to $\alpha = 0$ is true. Since the "zeroth iteration" refers to the point in time right before the initial iteration of the loop, proving this statement is referred to as **initialization**.

Next we prove the statement corresponding to $\alpha = 1$: "After the first iteration of the loop, `total` contains the (sum of the) first element of `data`." During the first iteration, `total` is updated by adding `x`, which takes the value of the first element of `data`. From proving the invariant for $\alpha = 0$, we know that `total` enters the iteration with a value of zero. Hence, during the first iteration, `total` becomes $0 + \texttt{x} = \texttt{x}$, which is the first element of `data`. This proves the statement corresponding to $\alpha = 1$.

Next we prove the statement corresponding to $\alpha = 2$: "After the second iteration of the loop, `total` contains the sum of the first two elements of `data`." On this iteration, `x` takes the value of the second entry of `data`. The statement for $\alpha = 1$ says that `total` enters the second iteration as the sum of the first element of `data`. During the second iteration, `total` becomes the sum of the first element of `data` and `x`, which is the second. Therefore, at the end of the iteration, `total` is the sum of the first two elements of `data`, and so the statement is proven.

We could continue on in this way, proving the statements for $\alpha = 3$, $\alpha = 4$, and so on. But notice that the argument for $\alpha = 1$ was nearly identical to the argument for $\alpha = 2$; the arguments for $\alpha = 3$ and above will be very similar, as well. Instead of writing each argument separately, we will write a single argument that works for any iteration number, α, saving ourselves a ton of work.

Observe that a common feature of our arguments for $\alpha = 1$ and $\alpha = 2$ is that, in each case, we use the value of `total` as it was at the end of the prior iteration. That is, we used the statement about the zeroth iteration to prove the statement for the first iteration, and we used the statement for the first iteration to prove the statement for the second. In general, we will use the statement for the $(\alpha - 1)$th iteration to prove the statement for the αth.

We now do something clever: we take α to be some arbitrary iteration number between one and the total number of iterations, and we *assume* that the statement for $\alpha - 1$ is true. We then use this assumption to prove that the statement for α is true, as well. This is process is called **maintenance**, since it proves that the truth of the loop invariant is maintained across an arbitrary iteration.

Here is what maintenance looks like for the current problem. First, we assume that after the $(\alpha - 1)$th iteration, `total` contains the sum of the first $\alpha - 1$ elements of `data`. On the αth iteration, `total` is updated by adding `x`, whose value is the αth element of `data`. Hence `total` becomes the result of adding the αth element of `data` to the sum of the first $(\alpha - 1)$ elements. This is just the sum of the first α elements. Therefore, at the end of

the iteration, `total` will contain the sum of the first α elements of `data`, and the claim is proven.

It may seem that we are being sloppy by *assuming* that the statement for the previous iteration is true, but we can do this precisely because we have proven that the statement for $\alpha = 0$ is true during the intialization step above. Since the statement is true for $\alpha = 0$, our maintenance argument implies that it is true for $\alpha = 1$. Since it is true for $\alpha = 1$, we can re-apply our maintenance argument to conclude that it is true for $\alpha = 2$, and so on. Once we prove initialization and maintenance, we set off a chain-reaction of proving, and all of the statements, for every value of α, fall like dominoes[6].

To recap, we can prove that a loop invariant is true via a two-step process:

1. **Initialize:** Show that the invariant holds before the first iteration of the loop.

2. **Maintain:** Prove that if the invariant holds after the $(\alpha - 1)$th iteration, it will still hold after the αth.

To then prove that the loop is correct, we then show that the loop **terminates**, and translate the loop invariant to a statement about the status of the algorithm after the loop ends. In this case, the loop clearly terminates after it has run through all n elements of `data`. Substituting $\alpha = n$ into the loop invariant, we find that "After the nth (last) iteration, `total` contains the sum of the first n elements of `data` (i.e., all of the elements)." Thus, the loop correctly calculates `total`.

1.3.3 A More Complicated Algorithm

We have shown that the above algorithm for computing the mean is logically correct: when implemented on an idealized computer, it will indeed compute exactly the mean for every possible input. But actual computers are not ideal; in particular, they can only represent real numbers to a finite number of decimal places. One of the implications of this finite precision is **roundoff error**, in which adding a large number and a small number results in a loss of accuracy. You can see this for yourself by opening a Python interpreter and writing

```
>>> 1e16 + 1 - 1e16
```

Mathematically, the result of this calculation should be 1, of course. But Python – and any other language using the standard implementation of floating point numbers – will give a result of 0.

It turns out that – while the above algorithm is logically correct – the implementation can run into precision issues. In particular, the operation on line 5, `total += x`, can incur roundoff error when `total` is very large and `x` is very small, or *vice versa*. In critical applications, a different algorithm is used for computing the mean – one which only

[6] You may have seen this style of proof before; it is known as (weak) **mathematical induction**.

adds numbers which are likely to be on the same order of magnitude, thereby avoiding roundoff error. The Python implementation of this algorithm is shown below:

```
1 def running_mean(data):
2     """Assume len(data) > 0."""
3     current_mean = data[0]
4     count = 1
5     for x in data[1:]: # data[1], data[2], ...
6         count += 1
7         current_mean += (x - current_mean)/count
8     return current_mean
```

The logical correctness of this algorithm is not quite as obvious as that of the straightforward algorithm. We wish to show that the value of `current_mean` is indeed the mean of the data when the algorithm ceases execution. But `current_mean` is iteratively computed by a `for`-loop. Therefore, we will state and prove a loop invariant which will be useful in proving the correctness of the algorithm as a whole.

Invariant 2. After the αth iteration of the loop, `current_mean` stores the mean of the first $\alpha + 1$ elements of `data`.

Assuming for the moment that this loop invariant is true, we can use it to prove the overall correctness of the algorithm; we simply need to argue that the loop terminates after `len(data)` $- 1$ iterations. Since the loop iterates through each of the `len(data)` $- 1$ elements of `data[1:]` in turn, this is indeed the case. Therefore, taking $\alpha =$ `len(data)` in the loop invariant and doing some translation from mathematics into English, we obtain the following: "After `len(data)` iterations of the loop, `current_mean` stores the mean of the first `len(data)` elements of `data` (i.e., all of the elements)."

We prove the loop invariant by first initializing, then maintaining. To initialize, we prove that the invariant is true before the execution of the loop begins. Plugging $\alpha = 0$ into the invariant, we obtain the statement: "After the 0th iteration of the loop, `current_mean` stores the mean of the first $0 + 1 = 1$ elements of `data`." Looking again at the algorithm, `current_mean` is initialized to be the first element of `data` in line 1. Since the mean of one number is just that number, `current_mean` is indeed the mean of the first element immediately prior to the first iteration of the loop. This proves initialization.

Next, we maintain. Let $\alpha \in \{1, 2, 3, \ldots\}$ be an arbitrary iteration number. We first assume that the invariant holds after the $(\alpha - 1)$th iteration (i.e., the previous iteration). Plugging $\alpha - 1$ into the loop invariant yields the statement: "After the $(\alpha - 1)$th iteration of the loop, `current_mean` stores the mean of the first $(\alpha - 1) + 1 = \alpha$ elements of `data`". Taking this statement as true, we wish to prove that after one more iteration, the value stored in `current_mean` will be the mean of the first $\alpha + 1$ elements of the list.

Two things happen on each iteration. First, `count` is incremented by one. It is easy to see that on the αth iteration, `count` is incremented from α to $\alpha + 1$; if we wished to be

rigorous, we could prove this with its own loop invariant. Next, `current_mean` is updated according to the code

$$\texttt{current_mean = current_mean + (x - current_mean) / count,} \tag{1.1}$$

where x takes on the value of the $(\alpha+1)$th element of the list.

We now introduce a bit of notation. For any $t \in \{0, 1, 2, \ldots\}$, let S_t denote the sum of the first t elements of `data`, where we define S_0 to be 0 by convention. Note that the mean of the first t elements of the list is then S_t/t. We have assumed that after the $(\alpha-1)$th iteration, `current_mean` holds the mean of the first α elements of the list. That is, `current_mean` $= S_\alpha/\alpha$. Using this piece of knowledge, along with the fact that `count` $= \alpha+1$, the right hand side of Equation 1.1 becomes:

$$\frac{S_\alpha}{\alpha} + \left(\texttt{x} - \frac{S_\alpha}{\alpha}\right) \cdot \frac{1}{\alpha+1},$$

Our goal is to show that this is equal to the mean of the first $\alpha+1$ elements of the list, $S_{\alpha+1}/(\alpha+1)$. Collecting the terms involving S_α, we find:

$$\left(1 - \frac{1}{\alpha+1}\right)\frac{S_\alpha}{\alpha} + \frac{\texttt{x}}{\alpha+1}.$$

Noting that $(1 - 1/(\alpha+1)) = \alpha/(\alpha+1)$, we see that the above is equal to:

$$\frac{\alpha}{\alpha+1} \cdot \frac{S_\alpha}{\alpha} + \frac{\texttt{x}}{\alpha+1} = \frac{S_{\alpha-1} + \texttt{x}}{\alpha+1}.$$

Since x is the $(\alpha+1)$th element of the list, $S_\alpha + \texttt{x} = S_{\alpha+1}$, and the above is simply $S_{\alpha+1}/(\alpha+1)$, as desired. This proves the maintenance step.

Together, initialization and maintenance prove the loop invariant. We next show that the loop terminates. This simple `for`-loop clearly exits after `len(data) - 1` iterations. Therefore, we use the loop invariant to conclude that "After `len(data)` iterations of the loop, `current_mean` stores the mean of the first `len(data)` elements of `data` (i.e., all of the elements)." That is, the algorithm is correct.

1.3.4 Nested Loop Invariants

Consider Algorithm 1 which computes the mean of an $n \times n$ array of numbers in the straightforward way. In order to prove its correctness, we want to show that `total` contains the sum of all entries of the array at the end of the algorithm's execution. As before, `total` is computed iteratively by a `for`-loop, and so proving the correctness of the algorithm as a whole depends upon proving the correctness of a loop, which is precisely what we use loop invariants to do.

Algorithm 1 Computing the mean of an array.

```
def array_mean(arr):
    """Compute the mean of an n-by-n array of numbers."""
    total = 0
    n = len(arr) # the number of rows in the array
    for i in range(n):
        row_total = 0
        for j in range(n):
            row_total += arr[i, j]
        total += row_total
    return total / n**2
```

In this case, however, we have two loops: an inner loop over `j` and an outer loop[7] over `i`. As such, we will have *two* loop invariants, one for the inner loop and one for the outer. We begin by stating a loop invariant for the outer loop:

Invariant 3. After the αth iteration of the outer loop, `total` contains the sum of the elements of the first α rows of `arr`.

As before, proving the loop invariant involves an initialization step and a maintenance step. In the maintenance argument, we assume that after the $(\alpha - 1)$th iteration of the outer loop, `total` contains the sum of the elements of the first $\alpha - 1$ rows of `arr`, and we wish to show that after another iteration it contains the sum of the first α rows. Looking back at the algorithm's code, `total` is updated at the end of each iteration of the outer loop by adding `row_total` to it. Evidently, `row_total` should contain the sum of the αth row of the array. Since `row_total` is computed by the inner loop, we need to prove a second loop invariant:

Invariant 4. After the βth iteration of the inner loop, and during the αth iteration of the outer loop, `row_total` contains the sum of the first β elements of the αth row of the array.

To *initialize*, we argue that before execution of the inner loop, `row_total` is set to 0 and therefore contains the sum of the first β elements of the αth row (since the sum of an empty set of numbers is by convention 0). To *maintain*, we assume that after the $(\beta - 1)$th iteration of the inner loop during the αth iteration of the outer loop, `row_total` contains the sum of the first $\beta - 1$ elements of the αth row. On the βth iteration of the inner loop, `j` has value $\beta - 1$. Hence we add `arr[α-1, β-1]` to `row_total`, which we recognize as being the βth element of the αth row of the array. As a result, after this iteration, `row_total` contains the sum of the first β elements of the αth row.

[7]Remember that in this situation, the inner loop will iterate from 0 to $n - 1$ for *every* iteration of the outer loop.

At this point we have proven the loop invariant for the inner loop, now we need to use it. The second loop makes n executions. After the nth iteration, the loop exits and the loop invariant reads: "`row_total` contains the sum of the first n elements of the αth row of the array." In other words, it contains the sum of the αth row.

We can use this fact to prove the loop invariant for the outer loop. To *initialize*, we recognize that before the execution of the outer loop, `total` is 0 as desired. To *maintain*, we assume that after the $(\alpha - 1)$th iteration, `row_total` is the sum of the first $\alpha - 1$ rows of the array. The loop invariant for the inner loop tells us that at the end of the αth iteration, `row_total` is the sum of the elements in the αth row of the array. Since we add `row_total` to `total` at the end of each iteration of the outer loop, after the αth iteration `total` is the sum of the first α rows of the array.

This proves the outer loop invariant. Since the outer loop also runs n times, the loop invariant tells us that after the loop executes, `total` is the sum of the first n rows of the array, i.e., all of the rows of the array. Since we return this total divided by n^2, the correctness of the algorithm has been proven.

1.4 Efficiency

Once we have shown that an algorithm is correct, we must then analyze the computation resources it uses. An algorithm that is guaranteed to produce the correct result is useless if it requires an impractical amount of time or memory to do so. In this section, we will develop tools for assessing the efficiency of algorithms.

1.4.1 Time Complexity Analysis

Suppose that you and I both have our favorite algorithms for computing the mean, and I claim that my algorithm is faster. To settle the matter, we each implement our algorithms on our own computers and time their execution on the same benchmark data set. While a good first test of performance, this approach has some serious flaws. For instance, my computer might be faster or slower than yours, which makes any absolute timing hard to interpret. If we find that your algorithm takes 1 second on your laptop but my algorithm takes 0.1 seconds on my lab computer, it may just be that my computer is faster. We still don't know whether mine is a good, *efficient* algorithm.

Of course, it would be more fair to run both of the algorithms on the same computer. While better, this doesn't quite solve the problem. First, some algorithms are better suited to some hardware configurations than others. For example, maybe my algorithm is fast when run on the GPU in my computer, but significantly slower on computers without a GPU. Second, some algorithms are fast on certain inputs and slow on others. If we find that the your algorithm is slower, how can we be sure that it's slower in general, and not just on that particular input? Lastly, some algorithms have a lot of overhead and are slow

to start, but become relatively efficient when their input becomes bigger. Perhaps your algorithm is slower than mine when run on an input of size 100, but faster than mine when run on an input of size 1000.

In short, we want a method of assessing the speed of an algorithm that captures the properties of the algorithm itself, and not the computer on which it is run. Moreover, we are less concerned with the exact time it will take an algorithm to run, and instead wish to describe the performance in relative terms as a function of the size of the input. We do this by reasoning about the pseudocode of the algorithm in a process called **time complexity analysis**.

1.4.2 The Time Complexity of Computing the Mean

Recall the straightforward algorithm for computing the mean:

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

How long will this algorithm take to run when called on a list with n elements?

Our strategy will be to analyze the algorithm line-by-line to estimate the time each line takes to run; we will then add these estimates to obtain an estimate of the total time of execution. Consider, for example, the first line of the algorithm: `total = 0`. This operation of assigning zero to a variable takes different times on different computers, but note that the time it takes will not depend on n, the size of the input list. We don't have to decide on the *exact* time it takes for now; let's just assume that it takes a constant amount of time c_1. Later, when I run the algorithm on my computer or on your laptop, I can measure c_1 for that specific machine and fill it in here.

Let's do the same for `return total / len(data)`. Here we are dividing a number by the size of a list and returning the resulting number. While it may appear as though the time it takes to compute the length of the list becomes bigger the more elements the list has, this depends on the implementation of the data structure. Python's `list`s keep track of their size as they grow or shrink, so that asking for `len(numbers)` doesn't depend on how many numbers are actually stored in the list.[8] Because of this, we can say that this line takes time c_2 to execute, where it is understood that c_2 is a *constant* – it does not depend on the size of the input list.

Next, consider the `for`-loop portion of the algorithm:

```
for x in data:
    total += x
```

[8] The alternative, for example, is that Python would need to iterate through the list to count the number of entries, which would take more time the longer the list.

The big difference between these lines of code and the previous lines is, of course, that those lines ran only once, while these lines will run a number of times which depends on the size of the input list `data`.

Let's put aside the first line for a moment and start with `total += x`. This step does addition and assignment. The time it takes to run *once* does not depend on n, the size of the input list. Let's say that one execution of this line takes time c_4. The body of the `for`-loop runs once for each element in the list for a total of n times. Therefore, the *total* time contributed by this line to the run time of the algorithm as a whole, summed over all iterations, is $c_4 \cdot n$.

Now let's go back and consider the first line: `for x in data:`. Python does a lot of work in this line, and it isn't all visible to us. Namely, it asks for the next entry of the list (checking to make sure there is one) and assigns the result to `x`. Let's say that the time that it takes to do this process *once* is c_3. How many times does this process happen during the execution of the algorithm? One might guess that the answer is n, the size of the list. It actually happens $n+1$ times. This is due to how iteration works in Python (and indeed, in most languages): Python keeps asking for the next element of the list until the list responds that there are none left. On the nth ask, the list happily returns the nth element. It isn't until the $(n+1)$th ask that the list says that it has no more elements to give and the loop terminates. This issue of whether the line runs n times or $n+1$ may be rather subtle at first, but luckily it won't matter too much in the end.

The below table gathers the time it takes to run each line once as well as the number of executions of each line:

	Time per exec.	**Number of execs.**
`def mean(data):`		
`    total = 0`	c_1	1
`    n = len(data)`	c_2	1
`    for x in data:`	c_3	$n+1$
`        total += x`	c_4	n
`    return total / n`	c_5	1

To find the total time it takes to run the algorithm, we multiply the time it takes to run each line once by the number of times that line is run, and sum these up. That is, the time $T(n)$ to run on an input of size n is:

$$
\begin{aligned}
T(n) &= c_1 + c_2 + c_3(n+1) + c_4 \cdot n + c_5, \\
&= c_3 \cdot n + c_4 \cdot n + c_1 + c_2 + c_3 + c_5, \\
&= (c_3 + c_4)n + (c_1 + c_2 + c_4 + c_5), \\
&= cn + c',
\end{aligned}
$$

where in the last line we have defined $c = c_3 + c_4$ and $c' = c_1 + c_2 + c_4 + c_5$.

1.4.3 Scalability and Big-Theta

Now we could go measure $c_1, c_2, \ldots, c_5$ on an actual computer to come up with a formula for the algorithm's run time on that machine – but this isn't typically all that useful. What *is* useful, however, is the observation that the above formula is a *linear* function of n. Importantly, this tells us that – no matter what computer we use – the algorithm will take roughly twice as long if we double the input size (when the input is large enough). To see this mathematically, look at the ratio:

$$\begin{aligned} T(2n)/T(n) &= \frac{2cn + c'}{cn + c'}, \\ &= \frac{2cn}{cn + c'} + \frac{c'}{cn + c'}. \end{aligned}$$

When n is large, the first term is approximately two, whereas the second term is small. Hence $T(2n) \approx 2T(n)$ when n is big.

The **time scalability** of an algorithm refers to how the time it takes to run grows with the size of the input. Scalability is a major concern for data scientists because of the large size of the data sets we encounter. For small data sizes, even an algorithm that scales poorly will run in a reasonably amount of time. But as the size of the data set increases, that same algorithm may start to take too much time to be usable. Our algorithm to compute the mean scales **linearly**, which means that doubling the size of the input roughly doubles the time it takes to run. But we will soon see algorithms which do not scale linearly, and so that doubling the input size leads to larger increases in run time.

When comparing algorithms, we often care more about how each algorithm scales with the size of the input rather than the precise value of the constants involved in the calculation of its running time. The **time complexity** of an algorithm is an expression of its efficiency with constants and lower-order terms dropped. We often write the time complexity of an algorithm using the so-called **Θ-notation** (pronounced: "big-theta notation"). We will see the formal definition of Θ-notation shortly, but for now it suffices to understand it as a notation which ignores constant factors and lower order terms. In the case of the algorithm above, we saw that the run time was $T(n) = cn + c'$. Θ ignores the factor of c and the lower-order term c', so that we can simply write $T(n) = \Theta(n)$. If we had instead found that $T(n) = an^2 + bn + c$, we could have written $T(n) = \Theta(n^2)$.

1.4.4 Nested Loops

The algorithm above had a single `for`-loop which iterated over all of the n data points, resulting in a linear time complexity. Now consider the straightforward algorithm for finding the mean of an $n \times n$ array of numbers shown in Algorithm 1 and reproduced below for convenience:

```
def array_mean(arr):
    """Compute the mean of an n-by-n array of numbers."""
    total = 0
    n = len(arr) # the number of rows in the array
    for i in range(n):
        row_total = 0
        for j in range(n):
            row_total += arr[i, j]
        total += row_total
    return total / n**2
```

We would like to compute the time complexity of this algorithm. As before, we could construct a table detailing the time it takes for each line of the algorithm to run and the number of times it is executed. However, since we often do not care about the precise constants or lower-order terms involved in the expression for the running time, we can afford to be a little "sloppy". Intuitively, instead of analyzing each line of the algorithm, we need only find the segment of code which contributes the highest-order term to the running time calculation; this line determines the time complexity of the algorithm as a whole.

Looking at the above code, we see that each line takes only a constant amount of time per execution. Put another way, if we timed a single execution of `total = 0`, the time would not change if we made the input array `arr` bigger or smaller. This is true for every line of the algorithm – what differs between lines is how many times each is executed. Since Θ-notation ignores constant factors and lower-order terms, it suffices to find the line which is executed the greatest number of times.

Remember that `arr` is an $n \times n$ array. Let's calculate the number of executions of each line as function of n. Anything that isn't in a `for`-loop runs once, no matter what n is. The body of the outer `for`-loop runs n times. The outer `for`-loop's first line, `for i in range(n):` actually runs $n+1$ times, since on the $(n+1)$th iteration it discovers that there are no more entries and it must exit. The body of the inner loop runs n times for each iteration of the outer loop, and hence runs n^2 times in total. The inner loop's first line, `for j in range(n):`, runs $n+1$ times for each iteration of the outer loop, and hence runs $n(n+1)$ times in total. These results are summarized in the table below:

Code	Number of execs.
`def array_mean(arr):`	
`    total = 0`	1
`    n = len(data)`	1
`    for i in range(n)`	$n+1$
`        row_total = 0`	n
`        for j in range(n):`	$n(n+1)$
`            row_total += arr[i, j]`	n^2
`        total += row_total`	n
`    return total / n**2`	1

The line that runs the most number of times is the first line of the inner loop – it is executed $n(n+1) = n^2 + n$ times. Therefore, n^2 is the leading power of n in the total runtime of the algorithm, $T(n)$. Hence we say that $T(n) = \Theta(n^2)$, i.e., the algorithm has **quadratic** time complexity. Note that the line that is executed most runs $n(n+1)$ times, whereas the body of the inner loop runs n^2 times. Both are $\Theta(n^2)$, and so it doesn't actually matter that `for j in range(n):` runs n times more: that n is ignored by Θ.

A common mistake is to believe that any algorithm involving two loops, one nested inside the other, must necessarily take quadratic time, but this is not the case. For instance, consider the artificial example below:

```
def nested_algorithm(data):
    for x in data:
        for y in range(4):
            print(x * y)
```

Supposing that `data` has n elements, the above takes time $\Theta(n)$. This is because the body of the inner loop runs a fixed number of times per each iteration of the outer loop. The line which is executed the most is the second **for**-loop; it runs $(4+1)n = 5n$ times in total. After forgetting constants and lower order terms, this results in $T(n) = \Theta(n)$.

1.4.5 Nested Loops with Dependent Ranges

Recall that an $n \times n$ array is **symmetric** if its entries are mirrored over the diagonal; that is, `arr[i, j] = arr[j, i]`. For example, the below array is symmetric:

```
arr = [
    [0, 1, 2, 3],
    [1, 4, 5, 6],
    [2, 5, 7, 8],
    [3, 6, 8, 9],
]
```

To compute the mean of a symmetric array, we could, of course, use the function `array_mean` given above. But this function seems somewhat suboptimal: its nested loops total the entries of the array by iterating over *all* of its elements, whereas about half of the elements of a symmetric array are duplicates. A more efficient approach is to compute the total by iterating over only those elements of the array which are along or to the right of the diagonal. We count each diagonal element once, but an element `arr[i,j]` (`j` $>$ `i`) to the right of the diagonal should be counted twice in order to account for its "twin" element `arr[j,i]` which will not be iterated over. The below code implements this idea:

```
1 def symmetric_array_mean(arr):
2     """Compute the mean of an n-by-n symmetric array."""
3     total = 0
4     n = len(arr)
5     for i in range(n):
6         total += arr[i, i]
7         for j in range(i+1, n):
8             total += 2 * arr[i, j]
9     return total / n**2
```

This algorithm involves a nested loop, but one which differs in a significant way from the nested loop in `array_mean`: the range of the inner loop depends on the outer loop's progress. It is because of this that `symmetric_array_mean` is able to loop through fewer elements of the array as compared to `array_mean`, and is therefore a faster algorithm. But is `symmetric_array_mean` *significantly* faster in the sense that it has a smaller time complexity than `array_mean`?

To answer this question, we first observe that each line of the algorithm takes constant time per execution. Therefore, determining the time complexity of the algorithm as a whole boils down to counting the number of executions of the line which is executed the greatest number of times, asymptotically. Intuitively, this will be the body[9] of the inner loop, Line 8.

During different iterations of the outer loop, Line 8 runs a different number of times. For example, suppose `symmetric_array_mean` is run on a 4×4 array. On the first iteration of the outer loop, `i` $= 0$, and so the inner loop iterates over the range `range(0+1, 4)` $=$ `range(1,4)`. There are three elements in this range: `1`, `2`, and `3`. Hence Line 8 executes 3 times. However, on the second iteration of the outer loop, `i` $= 1$, and so the inner loop iterates over the range `range(1+1, 4) = range(2,4)`. There are only two elements in this range, and so Line 8 executes twice. If we continue on in this way, we will see that

[9]It turns out that Line 8 is not the line which is executed the greatest number of times; this distinction goes to Line 7. During any given iteration of the outer loop, Line 7 runs one more time than Line 8 in order to check the loop's exit condition. As mentioned previously, we can safely ignore this extra iteration when working with Θ-notation.

Line 8 executes once on the third iteration of the outer loop, and doesn't execute at all on the fourth iteration of the outer loop. In general, during the αth iteration of the outer loop, Line 8 runs $n - \alpha$ times.

The outer loop iterates n times. Therefore, the total number of executions of Line 8 is the sum of $n - \alpha$, for α ranging from 1 to n:

$$\underbrace{(n-1)}_{\alpha=1} + \underbrace{(n-2)}_{\alpha=2} + \ldots + \underbrace{(n-(n-2))}_{\alpha=n-2} + \underbrace{(n-(n-1))}_{\alpha=n-1} + \underbrace{(n-n)}_{\alpha=n}.$$

Simplifying:

$$\underbrace{(n-1)}_{\alpha=1} + \underbrace{(n-2)}_{\alpha=2} + \ldots + \underbrace{2}_{\alpha=n-2} + \underbrace{1}_{\alpha=n-1} + \underbrace{0}_{\alpha=n}.$$

That is, the total number of executions of Line 8 is the sum of the first $n-1$ natural numbers, starting with 1 and ending at $n-1$. The sequence $(1, 2, 3, \ldots)$ is called an **arithmetic sequence**, and the sum of the first n elements, S_n, can be computed via a famous formula:

$$S_n = \frac{n(n+1)}{2}.$$

Of course, we want the sum of the first $n-1$ elements. Substituting $n-1$ in for n, we find:

$$S_{n-1} = \frac{(n-1)((n-1)+1)}{2} = \frac{(n-1)n}{2},$$

and so Line 8 executes $n(n-1)/2$ times.

The time complexity of the algorithm as a whole is determined by the number of executions of Line 8. Observe that $n(n-1)/2 = \Theta(n^2)$. Therefore, the time complexity of `symmetric_array_mean` is $\Theta(n^2)$, which is the same as the time complexity of `array_mean`! This isn't to say that `symmetric_array_mean` isn't more efficient than `array_mean`; in fact, it is about twice as fast. Therefore, it is important to keep in mind that time complexity alone does not tell us everything there is to know about an algorithm's performance.

1.5 Big-Θ and Friends

We have so far used an informal definition of the Θ-notation. We now formalize Θ and introduce a few related notations that are frequently used to discuss the efficiency of algorithms.

1.5.1 Big-Θ

For the following definition, recall that $\mathbb{N}^+$ is the set of all positive natural numbers, i.e., $\mathbb{N} = \{1, 2, 3, \ldots\}$.

Definition 1. Let $g(n)$ be a function. We write $\Theta(g(n))$ to denote the set of all functions $f(n)$ for which there exist positive real numbers c_1, c_2 and an $n_0 \in \mathbb{N}^+$ such that for all natural numbers $n \geq n_0$,

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

There is a lot to unpack in this definition. Let's start with some examples:

Example 1. Suppose I am the gatekeeper of the set $\Theta(n^2)$, and you come to me with your favorite function, $f(n) = 3n^2 + 4n$. In order to gain membership into $\Theta(n^2)$, you will have to prove that $f(n)$ satisfies the above definition.

To begin, you note that the definition involves both an upper and lower bound. We'll start with the lower bound. You point out that for any $n \geq 0$, $3n^2 + 4n \geq 3n^2$. So, letting $c_1 = 3$, we have that $f(n) \geq c_1 n^2$ for any $n \geq 0$.

Now you tackle the upper bound. You write down:

$$3n^2 + 4n \leq c_2 n^2$$

and say that you will find a positive real number c_2 such that the above holds for large enough n. You start by dividing both sides by n, since we can assume that it is greater than zero:

$$3n + 4 \leq c_2 n.$$

Gathering like terms:

$$(c_2 - 3)n \geq 4.$$

You point out that if $c_2 = 4$, we have:

$$(c_2 - 3)n = (4 - 3)n = n.$$

Hence, for any $n \geq 4$, it is the case that $(c_2 - 3)n \geq 4$. This proves that $f(n) \leq 4n^2$ for all $n \geq 4$.

You have shown that $f(n) \geq 3n^2$ for all $n \geq 0$, and also that $f(n) \leq 4n^2$ for all $n \geq 4$. Putting these two together, you have proven that:

$$3n^2 \leq f(n) \leq 4n^2$$

for all $n \geq 4$. This proves that $f(n)$ belongs in $\Theta(n^2)$.

Example 2. It's a different day and you have a new favorite function: $f(n) = 3n^3 - 50n + 42$. Let's prove that it belongs to $\Theta(n^3)$.

First, we will find a lower bound. When we do this, we can get rid of any part of $f(n)$ that is clearly positive to simplify our work. For example, $f(n) = 3n^3 - 50n + 42 \geq 3n^3 - 50n$ for all $n \geq 0$. Now we want to find a constant c_1 such that $3n^3 - 50n \geq c_1 n^3$ for large enough n. Dividing both sides by n and collecting like terms, we get:

$$(3 - c_1)n^2 \geq 50.$$

Our task is to find c_1 and n satisfying this. We don't want c_1 to be bigger than 3, because then the left hand side will be negative for every $n \geq 0$, and it is very hard (impossible, even) to show that a negative number is greater than 50. So take c_1 to be your favorite number between 0 and 3, say, 1. Then the above is satisfied whenever

$$(3-1)n^2 \geq 50,$$

that is, when $2n^2 \geq 50$ or $n \geq \sqrt{25} = 5$. Hence $f(n) \geq n^3$ for all $n \geq 5$.

Now let's prove an upper bound. To make things easier, we first realize that $f(n) = 3n^3 - 50n + 42 \leq 3n^3 + 42$ for all $n \geq 0$, since $-50n$ is a non-positive quantity for all $n \geq 0$. Now we wish to find a constant c_2 such that

$$3n^3 + 42 \leq c_2 n^3$$

whenever n is large. Gathering like terms, we have:

$$(c_2 - 3)n^3 \geq 42.$$

We can take c_2 to be whatever we'd like, as long as the left hand side is positive. Let's pick 45, since it makes the math easier. Then:

$$(45-3)n^3 \geq 42,$$

i.e., $42n^3 \geq 42$. This is true for every $n \geq 1$. Hence $f(n) \leq 42n^3$ for all $n \geq 1$.

Putting the upper and lower bound together, we find that

$$n^3 \leq f(n) \leq 42n^3$$

for all $n \geq 42$. This shows that $f(n)$ is a member of $\Theta(n^3)$.

Example 3. Is $f(n) = 5n$ a member of $\Theta(n^2)$? It is not: we can prove the upper bound, but not the lower. To see this, suppose there is a constant c such that $5n \geq cn^2$ for large enough n. Dividing both sides by n, we get $5 \geq cn$ for all n large enough. But this cannot hold for large n, since the right hand side grows to ∞.

Curiously, while $\Theta(g(n))$ is a set of functions, we do not use set notation to denote that $f(n)$ is a member of $\Theta(g(n))$. That is, instead of writing $f(n) \in \Theta(g(n))$, we typically write $f(n) = \Theta(g(n))$.

1.5.2 Big-O

Writing $f(n) = \Theta(g(n))$ provides upper and lower asymptotic bounds for $f(n)$. However, in some cases, we may not have (or need) a lower bound. In such cases, we can use big-O notation:

Definition 2. Let $g(n)$ be a function. We write $O(g(n))$ to denote the set of all functions $f(n)$ for which there exists a positive real number c and a positive natural number n_0 such that for all $n \geq n_0$,

$$0 \leq f(n) \leq cg(n).$$

O-notation provides an upper bound, but not a lower bound. Compare this to Θ-notation, which provides both. As a result, if $f(n) = \Theta(g(n))$, then $f(n) = O(g(n))$, too.

Example 4. Let $f(n) = 3n^2 + 4n$. Then $f(n) = O(n^2)$. To prove this directly it is sufficient to use the upper bound from Example 1.

On the other hand, if $f(n) = O(g(n))$ it isn't necessarily $\Theta(g(n))$, as the following counterexample demonstrates.

Example 5. Let $f(n) = 5n$. Then $f(n) = O(n)$, but it is also true that $f(n) = O(n^2)$. In fact, $5n \leq 5n^2$ for all $n \geq 1$, which shows that $f(n) = O(n^2)$. But, as shown in Example 3, $f(n)$ is not a member of $\Theta(n^2)$.

1.5.3 Big-Ω

On the other hand, if we only want to provide a *lower* bound, we can use Ω-notation:

Definition 3. Let $g(n)$ be a function. We write $\Omega(g(n))$ to denote the set of all functions $f(n)$ for which there exists a positive real number c and a positive natural number n_0 such that for all $n \geq n_0$,

$$0 \leq cg(n) \leq f(n).$$

Example 6. $3n^2 + 4n = \Omega(n^2)$. To show this we can use the same lower bound as in Example 1. But, simultaneously, $3n^2 + 4n = \Omega(n)$, since $3n^2 + 4n \geq n$ for all $n \geq 0$.

1.5.4 Properties

The Θ-notation has a useful property that helps simplify the time complexity analysis of algorithms:

Theorem 1. *If $f_1(n) = \Theta(g_1(n))$ and $f_2(n) = \Theta(g_2(n))$ then $f_1(n) + f_2(n) = \Theta(g(n))$, where for every n the function g is defined by $g(n) = \max(g_1(n), g_2(n))$.*

Proof. Since $f_1(n) = \Theta(g_1(n))$, there exist positive real numbers α_1, β_1 and a positive natural number n_1 such that $\alpha_1 g_1(n) \leq f_1(n) \leq \beta_1 g_1(n)$ for all $n \geq n_1$. Similarly, there exist positive real numbers α_2, β_2 and a positive natural number n_2 such that $\alpha_2 g_2(n) \leq f_2(n) \leq \beta_2 g_2(n)$ for all $n \geq n_2$.

We wish to show that $f(n) = f_1(n) + f_2(n)$ is $\Theta(g(n))$, where $g(n) = \max\{g_1(n), g_2(n)\}$. For this, we must prove both an upper and a lower bound. We start with the upper bound.

Using the fact that $f_1(n) \leq \beta_1 g_1(n)$ for all $n \geq n_1$ and $f_2(n) \leq \beta_2 g_2(n)$ for all $n \geq n_2$, we have:

$$f_1(n) + f_2(n) \leq \beta_1 g_1(n) + \beta_2 g_2(n), \qquad \text{for all } n \geq \max\{n_1, n_2\}$$

Our goal is to get the right hand side to look like $c \max\{g_1(n), g_2(n)\}$, where c is some constant. A first step is to get it to look like $c(g_1(n) + g_2(n))$. To this end, define $\beta = \max\{\beta_1, \beta_2\}$. Now $\beta_1 \leq \beta$ and $\beta_2 \leq \beta$, so:

$$\begin{aligned} &\leq \beta g_1(n) + \beta g_2(n), \\ &= \beta(g_1(n) + g_2(n)), \end{aligned}$$

Now, recall that $g(n) = \max\{g_1(n), g_2(n)\}$. Hence $g_1(n) \leq g(n)$ and $g_2(n) \leq g(n)$, hence:

$$\begin{aligned} &\leq \beta(g(n) + g(n)), \\ &= 2\beta g(n). \end{aligned}$$

That is, $f_1(n) + f_2(n) \leq 2\beta g(n)$ for all $n \geq \max\{n_1, n_2\}$.

Now we need to prove the lower bound. We have from the fact that $f_1(n) = \Theta(g_1(n))$ and $f_2(n) = \Theta(g_2(n))$:

$$f_1(n) + f_2(n) \geq \alpha_1 g_1(n) + \alpha_2 g_2(n), \qquad \text{for all } n \geq \max\{n_1, n_2\}.$$

Define $\alpha = \min\{\alpha_1, \alpha_2\}$. Then:

$$\begin{aligned} &\geq \alpha g_1(n) + \alpha g_2(n), \\ &= \alpha(g_1(n) + g_2(n)). \end{aligned}$$

Can we say that $g_1(n) + g_2(n) \geq g(n) = \max\{g_1(n), g_2(n)\}$? Indeed we can. Observe that for any particular value of n, either $g_1(n)$ or $g_2(n)$ is the max, and we can replace it by $g(n)$. Suppose (without loss of generality) that $g_1(n) = g(n) = \max\{g_1(n), g_2(n)\}$. Then $g_1(n) + g_2(n) = g(n) + g_2(n) \geq g(n)$. Hence, in general, $\alpha(g_1(n) + g_2(n)) \geq \alpha g(n)$ for all $n \geq \max\{n_1, n_2\}$.

We have shown both lower and upper bounds. That is, for all $n \geq \max\{n_1, n_2\}$,

$$\alpha g(n) \leq f_1(n) + f_2(n) \leq 2\beta g(n).$$

Therefore $f_1(n) + f_2(n) = \Theta(g(n))$. □

How does this help us analyze the time complexity of algorithms? Suppose I have two functions, `foo(n)` and `bar(n)`. `foo(n)` takes $\Theta(n^2)$ time, whereas `bar(n)` takes $\Theta(n)$ time. A third function, `baz(n)`, simply calls `foo` and `bar` in series:

```python
def baz(n):
    foo(n)
    bar(n)
```

The time it takes for `baz` to run is clearly $T_{\text{baz}}(n) = T_{\text{foo}}(n) + T_{\text{bar}}(n)$. Since $T_{\text{foo}} = \Theta(n^2)$ and $T_{\text{bar}} = \Theta(n)$, the above theorem tells us that

$$T_{\text{baz}}(n) = \Theta(\max\{n^2, n\}) = \Theta(n^2).$$

In this situation, `foo` is often called the *bottleneck* of the algorithm, since it dominates the time complexity. If we could somehow get rid of `foo`, `baz` would immediately become a linear-time algorithm.

Similar results hold for O-notation and Ω-notation:

Theorem 2. *Let $g(n) = \max\{f_1(n), f_2(n)\}$. Then:*

1. *if $f_1(n) = O(g_1(n))$ and $f_2(n) = O(g_2(n))$, $f_1(n) + f_2(n) = O(g(n))$;*
2. *if $f_1(n) = \Omega(g_1(n))$ and $f_2(n) = \Omega(g_2(n))$, $f_1(n) + f_2(n) = \Omega(g(n))$.*

The proofs are similar to the last proof above.

1.6 Brute-Force Algorithms

Having so far developed the tools for assessing the correctness and efficiency of algorithms, we now turn to the problem of designing them. There are many algorithm design paradigms, and we will explore several in the coming chapters. In this section, we will consider what is arguably the most straightforward strategy: **brute-force search**. We will see that while the algorithms resulting from this strategy are typically easy to implement and obviously correct, they are often so inefficient that they are useless for all but the smallest and easiest problems.

1.6.1 Search Problems and Optimization

The brute force strategy can be used to design algorithms for solving what are known as **search problems**. In this type of problem, the goal is to find an object which satisfies a certain criterion. The object is assumed to come from a set of possible solutions called the **search space**. The search space may be finite or infinite; if it is finite, the brute-force approach to solving the problem is to systematically check each possible solution until a valid solutions is found. A textbook example of a brute force search is attempting to break into a safe by trying all of the possible keypad combinations. Clearly, this might take a while!

Algorithm 2 Brute-force optimization algorithm.

```
def brute_force_minimize(objective_function, search_space):
    """
    Minimize an objective function through brute force.

    Arguments
    ---------
    objective_function
        A callable implementing the function to be minimized.
    search_space
        A finite iterable which will be searched for the minimizer.

    """
    min_value = float('inf') # Python for "∞"
    min_theta = None
    for theta in search_space:
        value = objective_function(theta)
        if value < min_value:
            min_value = value
            min_theta = theta
    return min_theta
```

An **optimization problem** is a special type of search problem which is often encountered in data science. In these problems, the goal is to find an element of the search space which minimizes (or maximizes) an **objective function** quantifying the quality of a solution. We have seen one optimization problem already: that of minimizing the average square loss, L_{sq}. In that problem, the search space was the set of all real numbers, and L_{sq} was the objective function. Other examples of optimization problems in data science include least squares regression (where the search space is the set of all linear functions, and the objective function is the sum-of-squared-errors) and k-Means clustering (where the search space is the set of all sets of k cluster centers, and the objective function is the sum of squared distances to the nearest cluster center).

If an optimization problem's search space is finite, we may once again use the brute force strategy to solve it. The first step is to design an algorithm for evaluating the objective function; what this algorithm looks like depends on the problem at hand. The next step, however, always looks the same: we evaluate the objective function on each element of the search space and return the minimizer (or maximizer). The algorithm for minimizing the objective function is shown in Algorithm 2. This algorithm is straightforward to

implement and is obviously correct[10]. But we can also see that the time it takes to complete the search is at least as long as it takes to iterate through the entire search space. More formally, if there are S elements in the search space, the time complexity of this algorithm is $\Omega(S)$. As we will see, the search space associated with many optimization problems can quickly become mind-bogglingly large, effectively ruling out brute-force search as an algorithm.

1.6.2 Wisdom of the Crowd, Revisited

At the beginning of this chapter, we considered the problem of aggregating guesses $x_1, \ldots, x_n$ into a single, accurate guess, θ^*. Our approach was to find the θ^* which minimizes the average square loss, L_{sq}, defined as follows:

$$L_{\text{sq}}(\theta) = \frac{1}{n}\sum_{i=1}^{n} \ell_{\text{sq}}(x_i, \theta) = \frac{1}{n}\sum_{i=1}^{n}(x_i - \theta)^2,$$

where $\ell_{\text{sq}}(x_i, \theta) = (x_i - \theta)^2$. A potential pitfall of using the square loss lies in its sensitivity to *outliers*. If the data are expected to be noisy[11], a better approach might be to minimize the **average absolute loss**, $L_{\text{abs}}(\theta)$, defined as

$$L_{\text{abs}}(\theta) = \frac{1}{n}\sum_{i=1}^{n} \ell_{\text{abs}}(x_i, \theta) = \frac{1}{n}\sum_{i=1}^{n} |x_i - \theta|,$$

where $\ell_{\text{abs}}(x_i, \theta) = |x_i - \theta|$. This leads to the following computational problem:

GIVEN: Real numbers $x_1, \ldots, x_n$.
COMPUTE: A real number θ^* minimizing the average absolute loss, L_{abs}.

The average absolute loss L_{abs} is not differentiable, and so we cannot solve the problem using calculus, as was the case with the average square loss. Moreover, this problem's search space is the infinite set of all real numbers, and so it seems that brute-force minimization cannot be used to solve it. However, we will soon see that L_{abs} achieves its minimum at one (or more) of the data points. As a result, the search space is reduced from the infinite set of all real numbers to the finite set of data points.

A good first step in solving an optimization problem is to visualize the objective function. In this case, the objective function, L_{abs}, is the sum of piecewise linear functions. More precisely, we have:

$$L_{\text{abs}}(\theta) = \frac{1}{n}\sum_{i=1}^{n} |x_i - \theta| = \sum_{i=1}^{n}\left(\frac{1}{n} \cdot |x_i - \theta|\right).$$

[10] Provided that the function implementing the objective function is correct!
[11] Or you have an enemy in the crowd who intentionally submits a bad guess.

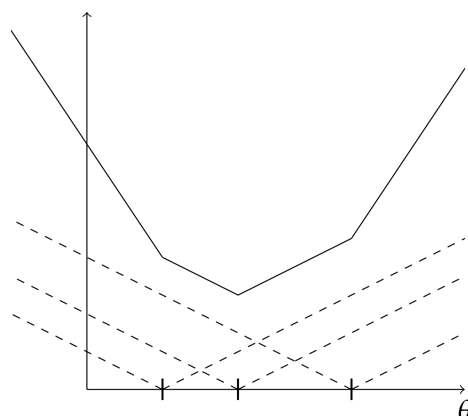


Figure 1.1: L_{abs} is a continuous, piecewise-linear function whose slope changes only at the data points.

Defining the piecewise-linear function $f_i(\theta) = \frac{1}{n} \cdot |x_i - \theta|$ for each $i \in \{1, \ldots, n\}$, we see that $L_{\text{abs}}(\theta) = \sum f_i(\theta)$. The plot of a given $f_i(\theta)$ makes a $\vee$-shape, centered around the data point x_i. Because $L_{\text{abs}}(\theta) = f_1(\theta) + \ldots + f_n(\theta)$, the plot of the objective function $L_{\text{abs}}(\theta)$ can be found by "stacking" the plots of $f_1, f_2, \ldots, f_n$, as is shown in Figure 1.1. The plot suggests that the following claim is true (its proof is left as an exercise):

Claim 1. *L_{abs} is a continuous, piecewise-linear function whose slope changes only at the data points, $x_1, \ldots, x_n$. Moreover, L_{abs} attains its minimum.*

Note that L_{abs} attaining its minimum value is not implied by its continuity or piecewise-linearity. For example, the function $f(x) = x$ is both continuous and linear (and therefore piecewise-linear), but never achieves its minimum value of $-\infty$.

We now wish to prove the following theorem:

Theorem 3. *L_{abs} achieves its minimum value at a data point.*

Note that the theorem does *not* say that a minimizer of L_{abs} *must* be a data point. If the preceding sentence seems contradictory, consider this: there may be several minimizers of L_{abs}. We only wish to show that at least one of the minimizers is a data point.

Proof. Our proof strategy is as follows. Let θ' be a minimizer of L_{abs}; Claim 1 tells us that such a minimizer exists. There are two cases: either θ' is a data point, or it is not. If it is a data point, great! We're done. If θ' is not a data point, we will show that the data point closest to θ' must be a minimizer[12].

[12] Actually, the closest data points on either side of θ' will turn out to be minimizers, but we need only one of them.

Assume in what follows that there are at least two data points (if $n = 1$, L_{abs} is minimized by taking $\theta = x_1$). If θ' is not a data point, then it must lie between two data points; call these x_- and x_+, with $x_- < x_+$. From Claim 1, the objective function L_{abs} is linear on the interval (x_-, x_+). It turns out that the slope of L_{abs} here must be zero: if the slope were non-zero, we would be able to decrease the value of L_{abs} by moving a little bit to the left or to the right of θ'. But θ' is a minimizer, so no smaller value of L_{abs} can exist! Hence the slope of L_{abs} on (x_-, x_+) is indeed zero. It follows from this and continuity that L_{abs} is constant on $[x_-, x_+]$. Since a minimizer $\theta' \in [x_-, x_+]$ is inside the interval, every point in the interval (x_- and x_+ in particular) is a minimizer. □

This proof used a technique called **proof by contradiction** that you may or may not have seen before. In a proof by contradiction, we show that a claim is true by assuming that it is false, and then showing that this leads to an obvious impossibility. The only way out of this embarrassing situation is to conclude that the assumption itself must be wrong, and that the claim must be true after all. This technique is very useful when we are working with objects that are supposedly special in some way (for instance, they claim to be minimizers or unique).

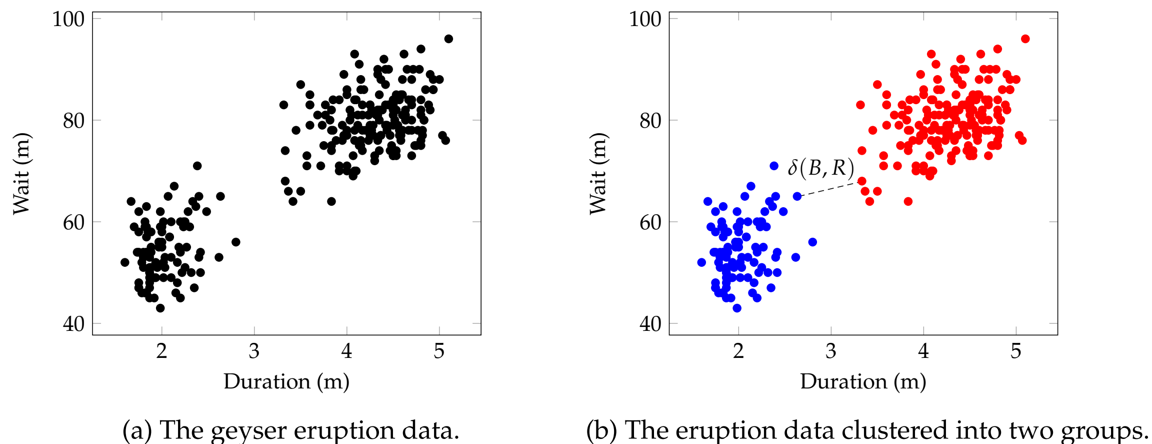
Proof by contradiction was used above to show that the slope within an interval containing a minimizer which isn't a data point must be zero. To make the technique clear, let's write this part of the proof as a separate claim:

Claim 2. *Let θ' be a minimizer of L_{abs} which is not a data point, and let x_- and x_+ be the closest data points such that $\theta' \in (x_-, x_+)$. Then the slope of L_{abs} is zero on the interval (x_-, x_+).*

Proof. Assume for a contradiction that the slope on (x_-, x_+) is not zero, and is instead some constant $m \neq 0$. If m is positive, then moving slightly to the left of θ' decreases the value of L_{abs}, contradicting the fact that θ' is a minimizer. If m is negative, then moving slightly to the right of θ' decreases the value of L_{abs}, again contradicting the fact that θ' is a minimizer. In either case, we reach a contradiction. Therefore the slope on the interval (x_-, x_+) must be zero. □

We now return to solving the computational problem stated at the beginning of this section. Theorem 3 tells us that L_{abs} is minimized at one of the data points, effectively reducing the search space from the set of all real numbers to the set of data points. This is a finite set, and so we can solve this problem with a brute-force search. We begin by implementing the objective function, L_{abs}:

```
def average_abs_loss(theta):
    total_loss = 0
    for x in DATA:
        total_loss += abs(x - theta)
    return total_loss / len(DATA)
```



(a) The geyser eruption data.

(b) The eruption data clustered into two groups.

Figure 1.2

Here it is has been assumed[13] that the data are stored in the module-level variable `DATA`. A quick analysis of this algorithm shows that it takes $\Theta(n)$ time to compute the value of the objective function at any point θ, where n is the size of the data set.

We then carry out the brute-force search by calling

```
brute_force_minimize(average_abs_loss, DATA).
```

The search calls `average_abs_loss` once for each data point, for a total of n calls. Each call takes time $\Theta(n)$. As a result, the brute-force search takes quadratic time, overall. While it turns out that there are faster (linear time) algorithms for solving this problem, a quadratic time algorithm is still practical for data sets of modest size.

1.6.3 Exponential Time: Clustering

We now turn our attention to the classic data science problem of **clustering**, in which the goal is to automatically discover groups (i.e., **clusters**) in the data. We will see that the brute-force approach to finding clusters takes time that is exponential in the number of data points, which is much too slow for most practical applications.

Figure 1.2a depicts data collected from the Old Faithful geyser in Yellowstone National Park. In particular, the time spent waiting for the geyser to erupt is plotted against the duration of the ensuing eruption. The plot suggests that the eruptions form two distinct groups: one group consisting of the eruptions which last longer but also take longer to form, and the other consisting of those which are short-lived but are quick to form.

[13] It would probably be a better practice to move `DATA` to a closure, but we'll opt for a module-level variable for now out of simplicity.

Suppose we wish to recover these groups automatically. The first step in doing so is to formalize the goal of clustering as a computational problem. There are many approaches to doing so, and we will adopt a view in which the goal is to separate the data into two groups which are well-separated. More formally, define a **partition** of the data set X to be a pair of sets, B and R, such that each point $x \in X$ belongs to exactly one of B and R. Intuitively, B and R are the **clusters**. In order to measure how well-separated the clusters are, we introduce the function $\delta(B, R)$ which returns the smallest distance between a point in B and a point in R; if one of the two sets is empty, we'll define $\delta(B, R) = \infty$ for convenience. Figure 1.2b shows an intuitively good clustering of the points into a blue cluster B and a red cluster, R, and the pair of points which determine the separation of the clusters, $\delta(B, R)$.

Of course, there are many different ways to partition the data. Our goal is to find the partition the data which maximizes the separation, $\delta(B, R)$. This leads to the following computational problem:

> GIVEN: Data vectors $X = \{x_1, \ldots, x_n\}$.
> COMPUTE: A partition (B, R) of the data with maximum separation, $\delta(B, R)$.

Note that the search space associated with this problem is the set of all partitions of the data. Since the data set is finite, the set of all partitions is finite, too. This means that we may apply the brute-force strategy in order to solve the problem. The resulting algorithm is simple to implement, and obviously correct.

But recall that the time it takes to execute a brute-force search is at least as long as it takes to iterate through all of the elements in the search space. How big is the search space in this problem? That is, how many partitions of n data points are there? For each point, we must make a decision: does it belong to cluster B or cluster R? Since there are two choices for each of n points, the total number of possible partitions is

$$\underbrace{2 \cdot 2 \cdots 2 \cdot 2}_{n} = 2^n.$$

Therefore, the size of the search space is 2^n, where n is the number of data points. This means that the brute-force search has a time complexity of $\Omega(2^n)$; that is, it takes **exponential time**.

Exponential time is impractical for all but the smallest problems. To demonstrate this, suppose that the time it takes for a computer to perform a single basic operation is one nanosecond[14]. A brute-force search solving the above problem performs at least 2^n operations, which takes a total of at least 2^n nanoseconds. The table below shows how long this is for various values of n:

[14] This is a fairly reasonable estimate. It is also a very short amount of time: it takes a photon (the fastest thing in the universe) one nanosecond to travel 30 centimeters.

n	Time
10	1 microsecond
20	1 millisecond
30	1 second
40	18 minutes
50	13 days
60	36 years
70	37,000 years

As the table shows, the brute-force approach will only be useful for small problem sizes, as it will take several days to cluster datasets as small as 50 points. In contrast, a quadratic-time algorithm would be able to cluster 10,000 points in one-tenth of a second. We will design such a quadratic algorithm for solving this problem in the coming chapters.

In general, computational problems which require us to search over all possible combinations or assignments of points into groups will have exponentially-large search spaces, making brute-force search impractical. We will see that many of these problems will turn out to have efficient, polynomial-time algorithms.

1.6.4 Factorial Time: Doctor Orders

We will now consider a problem whose search space grows at a rate that is even faster than exponential.

Suppose that it is graduation day at the medical school, and you are given the task of placing the new doctors in line to receive their diplomas. Each graduate is allowed to make a brief speech, and an adjustable microphone has been placed at center stage for that purpose. But some graduates are taller, and some are shorter – you are worried that if you do not line them up in an efficient way, the microphone will have to be constantly adjusted.

You decide that a good ordering is one in which no speaker has to adjust the microphone very far. To make this precise, you come up with an objective function L_{diff} which returns the greatest difference in height between consecutive speakers. For example, suppose there are four graduates this year; their names and heights are shown in the table below: If we place them in alphabetical order, as shown in the table, the value of the

Name	Height (inches)
Winona	62
Xanthippe	58
Yvonne	71
Zelda	68

objective function L_{diff} is:

$$\max\{|58 - 62|, |71 - 58|, |68 - 71|\} = \max\{4, 13, 3\} = 13.$$

On the other hand, the ordering of Xanthippe, Winona, Yvonne, and Zelda results in the objective function taking the value:

$$\max\{|62 - 58|, |71 - 62|, |68 - 71|\} = \max\{4, 9, 3\} = 9.$$

Intuitively, your goal is to find the ordering of speakers in which the biggest difference in heights is minimized, leading to the following computational problem:

GIVEN: n speakers and their heights.
COMPUTE: An ordering of the speakers which minimizes L_{diff}, the biggest absolute difference in height between consecutive speakers.

Since there are finitely-many speakers, there are finitely-many orders in which they can speak. How many are there, exactly? There are n choices for the first speaker, $n - 1$ for the second, $n - 2$ for the third, and so on, for a total of $n(n-1)(n-2)\cdots 3 \cdot 2 \cdot 1 = n!$ possible orderings (or **permutations**). A brute-force search to minimize L_{diff} will have a time complexity of $\Omega(n!)$; that is, it will take **factorial time**.

The factorial function grows much faster than any exponential function. As a result, a factorial time algorithm is even slower than an exponential time algorithm. To demonstrate this, assume once more that it takes one nanosecond for a computer to perform a basic operation. If there are n speakers, it will take a brute-force search at least $n!$ nanoseconds to compute the optimal ordering. The table below shows how long this is for various values of n:

n	Time
7	5 microseconds
10	3 milliseconds
15	21 minutes
20	77 years
27	25 times the age of the universe

Twenty-seven is not a lot of graduates – a large medical school could easily graduate 100 new doctors every year. But our algorithm clearly has no chance at working when n is this large! We need to do something more clever than simply checking all of the possible orderings.

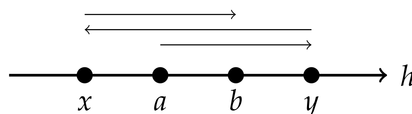
Recall that the objective function computes the greatest absolute difference in height between consecutive speakers. Intuitively, we want each speaker to speak between two people who are of similar height to their own. One way of doing this is to *sort* the speakers by height. It turns out that sorting the speakers by height indeed minimizes L_{diff}:

Theorem 4. *Sorting the speakers by height minimizes the maximum absolute difference in height between consecutive speakers.*

Proving the theorem will be made easier by using the following claim:

Claim 3. *Let a and b be speakers such that no other speaker is between a and b in height. Then in any ordering of the speakers in which a speaks first and b speaks last, the maximum absolute difference in height between consecutive speakers is at least the difference in height between a and b.*

Proof. We will prove this claim using a picture. Draw all of the speakers on a number line according to their height. Any ordering of the speakers can be drawn by making an arrow between consecutive people, where the length of the arrow is the difference in height between the people it connects. For example, the ordering (a, y, x, b) is depicted below:



The length of an arrow is the difference in height between the speakers. The length of the longest arrow is the biggest difference in height between consecutive speakers. In any sequence of arrows which starts at a and ends at b, there must be at least one arrow which crosses the gap between a and b. Since the width of this gap is the difference in height between a and b, the length of this crossing arrow (and thus the difference in height between the speakers it connects) must be larger than the difference in height between a and b. □

We will now prove the theorem. Our approach will be to use proof by contradiction, which was first introduced in the proof of Theorem 3 on page 35. Before we begin, note that value of the objective function, L_{diff}, is not changed if we reverse an ordering. This will be useful in this proof. For now, it means that it is safe to assume that by *sorted ordering* we mean sorted from shortest to tallest.

Proof of Theorem 4. In what follows, assume that the number of speakers is greater than two, as otherwise the proof is trivial.

Let ℓ be the maximum absolute difference in height between consecutive speakers in the sorted ordering. This means that there exists a pair of consecutive speakers – call them a and b – such that the difference in the height between a and b is ℓ. Since we have assumed that the graduates are in ascending order by height, we can assume that a is shorter than b.

Now suppose for a contradiction that there exists an ordering in which the maximum difference in height between consecutive speakers is *smaller* than ℓ. Without loss of generality, we can assume that a occurs before b in this ordering, too, since if this isn't the

case, we can simply reverse the ordering – this doesn't change the value of the objective function, and we are left with an ordering in which a comes before b.

Note that since a and b are consecutive speakers in the sorted order, there is no other speaker which is between a and b in height. Therefore, we may apply Claim 3 to conclude that the maximum absolute difference in height between consecutive speakers in the sub-ordering which starts with a and ends with b must be at least the difference in height between a and b, which is ℓ. That is, the maximum absolute difference in height between speakers in the new ordering is at least as large as it is in the sorted ordering. This contradicts the assumption that the new ordering is better. Hence this ordering cannot exist, and the sorted order is optimal. $\square$

Chapter 2

Sorting

Contents

2.1 Selection Sort 44
2.1.1 Intuition 44
2.1.2 An In-Place Selection Sort 45
2.1.3 Time Complexity 48
2.2 Digression on Recursion 48
2.2.1 Recursive Algorithm Design 49
2.2.2 Correctness 50
2.2.3 Time Complexity 51
2.3 Merge Sort 55
2.3.1 Designing `mergesort` 55
2.3.2 Understanding Merge Sort 62
2.3.3 Correctness 63
2.3.4 Time Complexity 64
2.4 The Value of Sorted Data 67
2.4.1 Searching 67
2.4.2 Picking In-Flight Movies 72

Sorting is a fundamental operation on data, and so the study of sorting algorithms naturally fits within the data science curriculum. Nevertheless, the fact remains that very few data scientists will ever have to implement a sorting algorithm as part of their job. Most every major programming language and numerical package provides highly optimized implementations of one or more sorting methods. So why do we study sorting, if not to know how to implement sorting algorithms?

The first reason is that sorting algorithms provide excellent opportunities to practice our skills in algorithm analysis. Over the next few sections, we will prove the correctness and analyze the time complexity of non-trivial sorting algorithms. We will also see recursive algorithms for the first time, and learn how to assess their efficiency.

Second, while you may never have to implement a sorting algorithm, you will certainly have to use them. Different sorting methods have their own advantages and disadvantages, and an algorithm which is the best for one use case might not be the best for another. Understanding how these methods work at a fundamental level will help you decide which to use, and when.

A third, more substantial reason is that sorting algorithms exhibit a diverse range of algorithm design strategies. It is true that you may never implement a sorting algorithm in your day-to-day job, but you will likely implement an algorithm that uses a similar strategy. You'll then find your understanding of sorting algorithms useful in applying the design strategy.

Fourth and finally, many computational problems become easier (or solutions become more efficient) it it can be assumed that the input is sorted. For instance, finding the median of a list of numbers can be done in constant time if the list is sorted. We will see several examples of such problems in this chapter. Moreover, we will get practice with designing algorithms that take advantage of assumptions in order to work faster[1].

2.1 Selection Sort

2.1.1 Intuition

Selection sort is an intuitive sorting algorithm that many people will come up with on their own if asked to sort a deck of cards or a stack of books. For example, suppose that the following cards are spread out in a row on the table in front of you:

7 2 1 -3 9

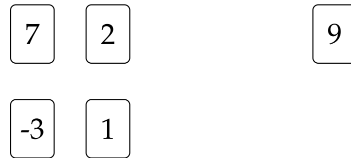
A simple procedure for sorting these cards is to iteratively remove the smallest. Here, -3 is the smallest card. We take this card from the top row and place it at the beginning of a new row of cards:

7 2 1 9

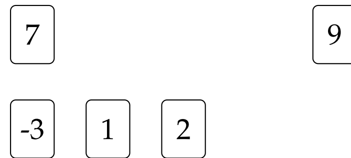
-3

[1]Fifth, sorting algorithms are popular subjects in job interviews.

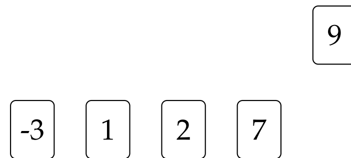
We then find the smallest remaining card in the top row and move it to the end of the new row. Here, the smallest remaining card is 1:



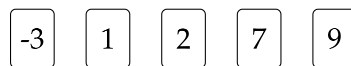
We proceed like this, at every step *selecting* the smallest card from the top row and placing it at the end of the bottom row. The next card to be moved is 2:



Then 7:



And finally 9:



At the end of the procedure – and at any step during – the bottom row of cards is sorted.

This informal description of the algorithm is easy to implement in code. The straightforward approach involves creating a new list for storing the sorted output. At each step, we pop the smallest element from the input list and append it to the output list. While this implementation is correct, it uses more memory than is necessary.

2.1.2 An In-Place Selection Sort

Instead, we will design a clever implementation of selection sort which modifies the input list, thereby avoiding the creation of a new list altogether. At any step of the algorithm,

the input list is divided into two parts. The first part, at the head of the list, contains the elements that have already been sorted; these are in their final positions. The second part, consisting of the tail of the list, contains the elements which have yet to be sorted. Separating these two parts of the list is the **barrier**. The barrier points to the first element of the tail of the list, which is also the location where the next selected element will be placed. At each iteration, the element at the barrier is swapped with the smallest element in the tail. Once this happens, the element at the barrier is in its final, sorted position, and so the barrier is moved to the right. The Python code below makes this precise.

```
def selection_sort(arr):
    """In-place selection sort."""
    n = len(arr)
    if n <= 1:
        return

    for barrier_ix in range(n-1):
        min_ix = find_minimum(arr, start=barrier_ix)
        arr[barrier_ix], arr[min_ix] = arr[min_ix], arr[barrier_ix] # swap

def find_minimum(arr, start):
    """Finds index of minimum. Assumes arr is non-empty."""
    n = len(arr)
    min_value = arr[start]
    min_ix = start
    for i in range(start + 1, n):
        if arr[i] < min_value:
            min_value = arr[i]
            min_ix = i
    return min_ix
```

Note that the index of the barrier starts at zero and ends at index `n-2`, which is not the last element of the list, but rather the *second to last*. We will see why this is in just a moment.

The correctness of the above algorithm follows from proving three loop invariants concerning the `for`-loop in the main `selection_sort` function:

Invariant 5. After the αth iteration, each of the first α elements of `arr` is less than or equal to each of the last `len(arr)` $- \alpha$ elements.

Invariant 6. After the αth iteration, the first α elements of `arr` are in sorted order.

Invariant 7. After the αth iteration, `arr` is a permutation of its original elements.

The first invariant is used to prove the second. It might seem like the third invariant is unnecessary, but consider this: an algorithm which simply overwrites all elements with zeroes satisfies the first two invariants, but it clearly doesn't correctly sort the list!

We will prove the first invariant now – the second and third are left as exercises. In proving these invariants, it is useful to adopt the convention that any universal statement about an empty list is **vacuously** true. For example, an empty list is considered to be sorted[2].

Proof of Invariant 1. We start with initialization. We wish to prove the statement: "After the zeroth iteration (i.e., before the first) each of the first zero elements of `arr` is less than or equal to each of the last `len(arr)` elements. Because `arr` is an empty list, this is vacuously true. We have therefore proven initialization.

Next, we prove maintenance. Assume that after the $(\alpha - 1)$th iteration, each of the first $\alpha - 1$ elements of `arr` are less than or equal to each of the last $\texttt{len(arr)} - (\alpha - 1)$ elements. We wish to show that after the αth iteration, each of the first α elements is less than or equal to the remaining $\texttt{len(arr)} - \alpha$ elements.

On the αth iteration, the αth element of `arr` is swapped with the smallest entry among the elements in the "tail" of the list, where the tail starts with the αth entry. Let x be the value of this smallest entry. Because x is a minimum, x is less than or equal to each element from the αth onwards. Therefore, after the swap, the αth element (which now has value x) is less than or equal to each element from the αth onwards; namely, it is less than or equal to each element from the $(\alpha + 1)$th onwards. From the assumption, each of the first $\alpha - 1$ elements is less than or equal to each element from $\alpha + 1$ onwards. Together, the previous two sentences say that each of the first α elements are less than or equal to each of the last $\texttt{len(arr)} - \alpha$ elements. This proves maintenance. □

Once we have stated and proved these invariants, we consider the termination of the loop. As we mentioned above, the loop only iterates $\texttt{len(arr)} - 1$ times, instead of `len(arr)` times as one might expect. Plugging this into each of the loop invariants, we see that after the loop exits:

1. each of the first $\texttt{len(arr)} - 1$ elements is less than or equal to the last $\texttt{len(arr)} - (\texttt{len(arr)} - 1) = 1$ elements.

2. the first $\texttt{len(arr)} - 1$ elements are sorted.

3. the array is a permutation of the original values.

The first statement tells us that the last element is the largest in the array; the second statement tells us that the preceding elements are sorted. Therefore the elements are in

[2]More exotically, the statement "Every element of `arr` is made of cheese." is a true statement, if `arr` is an empty list.

sorted order. Since the elements are the same elements that were given as input, we have proven that the algorithm is correct.

2.1.3 Time Complexity

We will now find the time complexity of selection sort. We start our analysis with `find_minimum`, since it is called in every iteration of the sort. How many times does the body of the `for`-loop in `find_minimum` run? The answer will depend on `start`, since the loop index `i` starts counting at `start + 1` and ends at `n - 1`. To come up with a formula for how many times the loop runs, a useful trick is to pick a nice value of `start` and run through the algorithm in your head. In this case, we might suppose that `start` is simply zero; then the loop variable starts at 1 and counts to `n - 1`, for a total of $n-1$ iterations. Similarly, if `start` is 1, then then loop starts counting at 2, and the body is therefore executed $n-2$ times. Picking up on the pattern, we see that, in general, the body is executed `n - start - 1` times.

Now we get back to analyzing `selection_sort`. The call to `find_minimum` which occurs in every iteration of `selection_sort` is just a nested loop in disguise, and we want to count the number of times the body of the inner loop runs in total. That is, how many times does the loop body in `find_minimum` run over the course of the sort? On the first iteration of the sort, `barrier_ix` is zero. Hence, from the above, the body of the loop in `find_minimum` runs $n-1$ times. On the second iteration, when `barrier_ix` is 1, it runs $n-2$ times. In general, for any particular value of `barrier_ix`, the body of the loop in `find_minimum` runs `n - barrier_ix - 1` times. In particular, on the last iteration, `barrier_ix` is $n-2$, and the body of `find_minimum` runs $n-(n-2)-1=1$ time. Therefore, the body of the loop in `find_minimum` runs one fewer time for every iteration of the loop in `selection_sort`, starting with $n-1$ times and ending at 1 time. Hence the total number of runs is:

$$(n-1)+(n-2)+\ldots+3+2+1.$$

This is the sum of the first $n-1$ natural numbers Recall (from the first homework) that the sum of the first n natural numbers is $n(n+1)/2$. Here we have the first $n-1$ numbers, and so we replace n by $n-1$ to obtain: $(n-1)n/2$ for the total number of times that the loop body in `find_minimum` runs over the course of selection sort. Since $n(n-1)/2 = \Theta(n^2)$, we say that the sort runs in *quadratic* time.

2.2 Digression on Recursion

The next sorting algorithm we will consider, merge sort, is not quite as intuitive as selection sort. Nevertheless, it is a classically-elegant algorithm and it is much faster than selection sort, to boot.

Merge sort is a **recursive** algorithm, meaning that it references itself. Recursive algorithms can be a little mind-bending at first, so we will take the opportunity to study a simple recursive algorithm before tackling merge sort.

2.2.1 Recursive Algorithm Design

Consider the problem of computing n-factorial. First recall that

$$n! = n \cdot (n-1) \cdot (n-2) \cdots 3 \cdot 2 \cdot 1.$$

Conventionally, we assume that $0! = 1$. Now, if we group the last $n-1$ terms, we will recognize the grouped quantity as $(n-1)!$:

$$\begin{aligned} &= n \cdot \underbrace{[(n-1) \cdot (n-2) \cdots 3 \cdot 2 \cdot 1]}_{(n-1)!} \\ &= n \cdot (n-1)! \end{aligned}$$

So if we *know* what $(n-1)!$ is, we can compute $n!$ by simply multiplying by n. For example, if we somehow know that 4! is 24, then we can compute 5! by calculating $5 \times 4! = 5 \times 24 = 120$.

Using this insight, we will now write a Python function, `factorial`, which takes one argument, `n`, and which computes `n`! by means of a recursive algorithm. When designing a recursive function such as this, we take a *leap of faith*. Here, we will *assume* while writing the body of `factorial` that calling `factorial(n-1)` (magically) results in the correct value. Under this assumption, `n`! is computed by carrying out `n * factorial(n-1)`, leading to the following code:

```
def factorial(n):
    """First attempt at recursive algorithm for computing n!"""
    return n * factorial(n-1)
```

If this seems too easy, you're right: the above code is incorrect, and will in fact never terminate. To see why, try running the function in your head with an input of `n` $= 3$. On the first call of the function, the computer tries to multiply `3` with the result of `factorial(3-1)`. This results in a second call to the function: `factorial(2)`, during which the computer tries to multiply `2` with the result of `factorial(2-1)`. This results in a third call to the function: `factorial(1)`. During this call, the computer makes a fourth call to the function: `factorial(0)`. During the fourth call, the computer makes a fifth: `factorial(-1)`. And so on. The algorithm will recurse forever, or at least until the computer runs out of memory[3].

[3] Or until you hit Python's **recursion limit**. Python is not optimized for recursion, and therefore only allows you to make several thousand recursive calls; you can see exactly how many by running `import sys; sys.getrecursionlimit()`.

The way out of this situation is simple: we modify `factorial` so that, when `n` is small enough, it doesn't recurse: it calculates and returns the answer outright. Here, we know that 0! is one by convention, so we add a check to the beginning of the function which returns 1 if the argument is zero:

```
def factorial(n):
    """Computes n!, recursively. Assumes n is non-negative integer."""
    if n == 0:
        return 1
    return n * factorial(n-1)
```

An input for which the answer is computed without recursing is known as a **base case**. You can check that, for any integer value of $n \geq 1$, `factorial(n)` will only recurse *until* it hits the base case, and will therefore terminate in a finite amount of time. In fact, we will soon see how to assess the time complexity of a recursive algorithm such as this.

In the coming sections, we will design several more recursive algorithms. When we do so, we will keep in mind the two steps we performed above: First, we took a *leap of faith* in assuming that the function being written already works for smaller inputs, and second, we found a *base case* for which the algorithm can compute the answer directly.

2.2.2 Correctness

The leap of faith we took when designing the above recursive algorithm may have left you feeling uncertain about its correctness. We will now restore your confidence by *proving* that the above `factorial` algorithm works. To be precise, we wish to prove the following claim:

Theorem 5. *`factorial(n)` correctly computes $n!$ for all non-negative, integer inputs.*

Verifying the correctness of algorithms has so far been synonymous with proving loop invariants, but the recursive algorithm above has no loops to analyze. We will take a different, but related, approach instead.

First note that proving the correctness of `factorial(0)` is easy: In this case, the algorithm simply returns one without recursing, and 0! is indeed one. This proves the correctness of the algorithm in the **base case** – the challenge now is to prove the claim for the infinitely-many possible inputs which remain.

Now consider the case when $n = 1$. In this situation, the algorithm computes and returns `1 * factorial(1 - 1)`, resulting in a call to `factorial(0)`. This is will produce the correct value, provided that `factorial(0)` is correct. But we have just shown that `factorial(0)` is correct! So `factorial(1)` is correct, too.

Next, consider the case then the input $n = 2$. By similar logic, `factorial(2)` will be correct, as long as `factorial(1)` is correct. But we have just shown this! So `factorial(2)` is indeed correct. We could proceed like this indefinitely, showing that

each of `factorial(3)`, `factorial(4)`, `factorial(5)`, and so on are correct. Instead, we will write one generic argument that works for all positive integers $n \geq 1$.

Proof by Induction

Note that in proving the correctness of `factorial(1)` above, we used the fact that `factorial(0)` is correct. Likewise, we used the fact that `factorial(1)` is correct in order to prove the correctness of `factorial(2)`. In an argument that works for general non-negative, integer $n \geq 1$, we would show that, *if* `factorial(n-1)` is correct, then `factorial(n)` is correct; this is known as proving the **inductive step**.

Together, proving the **base case** and the **inductive step** are sufficient to prove Theorem 5. The base case shows that the algorithm is correct when given zero as input. We can then apply the inductive step with $n = 1$ to show that `factorial(1)` is correct. Applying the inductive step once more proves that `factorial(3)` is correct. Indeed, we can repeatedly apply the proof of the inductive step to show that `factorial(n)` is correct for any non-negative, integer value of `n`. A proof of this form is known as a **proof by induction**. We now give such a proof of Theorem 5:

Proof of Theorem 5. We will prove the theorem by induction.

BASE CASE (n = 0): When `n` is zero, the function simply returns one, which is indeed 0-factorial. So `factorial(0)` is correct.

INDUCTIVE STEP: Let $n \geq 1$ be an integer such that `factorial(n-1)` is correct. We will show that `factorial(n)` is correct. The call to `factorial(n)` returns `n * factorial(n-1)`. Note that `factorial(n-1)` $= (n-1)!$. Therefore, `factorial(n)` returns $n cdot (n-1)! = n!$, and so `factorial(n)` is correct. □

The **inductive step** should look familiar – it is very similar to the maintenance step of a loop invariant proof. Similarly, the base case is analogous to initialization. In fact, the proofs of loop invariants we have so far seen are special cases of inductive proofs.

2.2.3 Time Complexity

What is the time complexity of `factorial`? So far, we have only assessed the efficiency of iterative algorithms. We will need to develop a new approach to analyze recursive routines.

To be precise, let $T(n)$ be the time taken by `factorial(n)`. As of now, we do not know what $T(n)$ is in general, and our goal is to bound it asymptotically. We begin by observing that $T(0)$, the time taken by `factorial(0)`, is some (unknown) constant, c_0. Therefore, we may write:

$$T(0) = c_0,$$

Now consider what happens when `factorial(n)` is called, where $n \geq 1$ is an arbitrary integer. We can compute the time this call takes, $T(n)$, by adding up the time taken by each

line of code in the function body. To start, the function will take some constant amount of time d_1 to check whether `n` is zero. It will then spend some unknown amount of time $T(n-1)$ in the call to `factorial(n-1)`. Lastly, it will spend some constant amount of time d_2 multiplying the result of the recursive call with `n` and returning the result. Hence the total time taken is:

$$\begin{aligned} T(n) &= d_1 + T(n-1) + d_2 \\ &= T(n-1) + c, \end{aligned}$$

where $c = d_1 + d_2$. Note that the T on the right hand side is the same as the T on the left hand side; that is, we have a recursive formula for $T(n)$. In words, the above says that the time it takes to run `factorial(n)` is equal to the time it takes to run `factorial(n-1)`, plus a constant.

Recurrence Relations

We have found that the time it takes for `factorial(n)` to run is given by the following formula:

$$T(n) = \begin{cases} c_0, & n = 0 \\ T(n-1) + c, & n \geq 1 \end{cases}$$

A recursive formula such as this is known as a **recurrence relation**. The **base case** of the recurrence is the input for which $T(n)$ can be computed directly, without recursing – here, the base case is when $n = 0$.

The above recurrence tells us how to compute $T(n)$ for any integer $n \geq 0$. For instance, to compute $T(1)$, we calculate:

$$T(1) = T(1-1) + c = T(0) + c = c_0 + c.$$

Likewise, to compute $T(2)$, we have:

$$\begin{aligned} T(2) &= T(2-1) + c \\ &= T(1) + c \end{aligned}$$

We have just seen that $T(1) = c_0 + c$. Making this substitution, we find:

$$\begin{aligned} &= (c_0 + c) + c \\ &= 2c + c_0 \end{aligned}$$

In principle, we could follow this procedure to calculate $T(100)$, $T(1,000)$, or even $T(1,000,000)$, but doing so would be somewhat time consuming. Moreover, it is still unclear how fast $T(n)$ grows – is it $\Theta(n)$, $\Theta(n^2)$, or something else altogether? What we

would like is a formula for $T(n)$ which *isn't* recursive, and instead allows us to compute $T(n)$ directly. Finding a non-recursive formula for $T(n)$ is called **solving the recurrence**. There are several approaches for solving recurrences; we will take a look at one of them now.

Solving Recurrences by Unrolling Them

Consider again the recurrence:

$$T(n) = \begin{cases} c_0, & n = 0 \\ T(n-1) + c, & n \geq 1 \end{cases}. \tag{2.1}$$

Our goal is to get rid of the $T(n-1)$ on the right hand side of the equation. If we replace n by $n-1$ in Equation 2.1, we arrive at a formula for $T(n-1)$:

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

Using this expression for $T(n-1)$, we can rewrite the formula for $T(n)$:

$$\begin{aligned} T(n) &= T(n-1) + c \\ &= [T(n-2) + c] + c \\ &= T(n-2) + 2c \end{aligned}$$

We say that we have **unrolled** the recurrence. What have we gained by doing so? We still have a T on both sides of the equation, but notice that the argument of the T on the right hand side is smaller than it was before. What happens if we unroll for a second time? Observe:

$$T(n-2) = T((n-2)-1) + c = T(n-3) + c$$

so:

$$\begin{aligned} T(n) &= T(n-2) + 2c \\ &= [T(n-3) + c] + 2c \\ &= T(n-3) + 3c \end{aligned}$$

After each unrolling, the argument of T on the right hand side decreases. If we unroll sufficiently many times, eventually the argument will be zero. We can stop unrolling at that point, because we know that $T(0) = c_0$. We will have by then replaced all instances of T on the right hand side, obtaining a formula which allows us to compute $T(n)$ directly and thereby solving the recurrence.

More precisely, we can solve a recurrence by following the below procedure:

1. *Unroll the recurrence until a pattern emerges.* We have already done this above. Let's agree to call the "first step" of unrolling the point in time before we have unrolled the recurrence. Then at the first step we have the original recurrence: $T(n) = T(n-1) + c$. The "second step" of unrolling is then the point in time after unrolling once. At the second step we found: $T(n) = T(n-2) + 2c$. At the third step (after unrolling twice), we had: $T(n) = T(n-3) + 3c$. We could continue, but by now a pattern has emerged. Placing these results into a table can make it easier to identify the pattern:

Step	Formula
1	$T(n-1) + c$
2	$T(n-2) + 2c$
3	$T(n-3) + 3c$

2. *Find a general formula for $T(n)$ in the kth step of unrolling.* The table above makes it clear that on the kth step of unrolling we will obtain the formula $T(n) = T(n-k) + k \cdot c$. You can check your general formula by plugging in a particular value of k and verifying that the result matches what you have in the table above. For instance, taking $k = 3$ yields $T(n) = T(n-3) + 3c$, which is indeed what is in the table.

3. *Calculate the step number in which the base case is reached.* The argument to T decreases with each step until it eventually reaches zero, which is the base case of this recurrence. On what step does it reach zero, exactly? On the kth step, the argument is $n - k$. We therefore solve $n - k = 0$ for k, resulting in $k = n$. Hence the base case is reached on the nth step

4. *Replace k with this step number in the general formula for $T(n)$.* We found above that, on the kth step, $T(n) = T(n-k) + k \cdot c$. Replacing k by n (the number of steps needed to reach the base case) we arrive at a new formula:

$$\begin{aligned} T(n) &= T(n-n) + n \cdot c \\ &= T(0) + n \cdot c \end{aligned}$$

   We know from Equation 2.1 that $T(0) = c_0$. So:

$$= c_0 + n \cdot c$$

At this point, we've removed all T's from the right hand side of the equation. We have therefore *solved* the recurrence: its solution is $T(n) = c_0 + n \cdot c$. You can verify that this formula will produce the same result as using Equation 2.1 to recursively calculate $T(n)$, for all integer values of $n \geq 0$.

Therefore, the time it takes `factorial` to run on an input of size n is $T(n) = c_0 + c \cdot n = \Theta(n)$. That is, it takes linear time.

2.3 Merge Sort

We will now look at an elegant and efficient sorting algorithm called **merge sort**. Merge sort is a textbook example of a **divide and conquer** algorithm. In a divide and conquer approach, the problem is **divided** into smaller and smaller problems until they are small enough to solve easily (i.e., **conquer**); these solutions are then **recombined** to solve the original problem. Divide and conquer algorithms are often simple in the sense that they do not take up a lot of lines of code, but it can be difficult to see *why* they work at first glance.

Merge sort implements the divide and conquer strategy in the following way: First, it splits the input array into two smaller arrays which each have (roughly) half of the original elements. It then sorts these smaller arrays independently. Finally, these smaller, sorted arrays are merged into one large, sorted array.

2.3.1 Designing `mergesort`

We will implement merge sort as a recursive Python function, `mergesort`, which modifies the array itself instead of returning a sorted copy. Since our implementation will be recursive, we should think about the **base case**; i.e., an input for which the algorithm can return the correct answer without recursing. A natural base case in sorting is when the input array is of size one or less, as then the array is already sorted, and nothing needs to be done! Therefore, our function will immediately check the length of the array – if is is bigger than one, it will perform the work of sorting, otherwise it will simply return.

The high-level flow of the function is documented with comments in the code below:

```
def mergesort(arr):
    if len(arr) > 1:
        # split the array in half
        # TODO
        # sort the left half
        # TODO
        # sort the right half
        # TODO
        # merge the left and right halves
        # TODO
```

We will now fill in each part of the function in turn.

Splitting the Array in Half

Merge sort begins by splitting the array into two pieces, each of which will be sorted independently. We can split an array into two pieces in Python using the **slicing** syntax –

see the box below for a refresher. For instance, if `middle` is a variable storing the index of the middle of `arr`, then splitting the array is done by writing:

```
left = arr[:middle]
right = arr[middle:]
```

It should be noted that slicing an array produces a copy[4], so that modifying `left` or `right` will not change `arr`.

[4] A *shallow* copy, to be precise.

Array Slicing. If `arr` is a list (or NumPy array), the part of the list starting at index `start` and ending right before index `stop` can be obtained by writing `arr[start:stop]`; this feature is known as **slicing**. Note that `arr[stop]` is *not* included in the result! This, along with the fact that Python starts counting from zero, can take some getting used to – but it has its advantages. For example, to take the first three elements from an array, one can write:

```
>>> x = ['a', 'b', 'c', 'd', 'e']
>>> x[0:3]
['a', 'b', 'c']
```

In fact, the zero can be omitted. The below does the same as the above:

```
>>> x[:3]
['a', 'b', 'c']
```

To select everything from the third index to the end, we can write:

```
>>> x[3:len(x)]
['d', 'e']
```

But similarly, omitting the end of the slice defaults to selecting all elements up until the end of the array:

```
>>> x[3:]
['d', 'e']
```

Of course, we don't always have to start our selection at the beginning, nor do we have to end at the last element. We can select, for instance, the second, third, and fourth elements with a slice, too:

```
>>> x[1:4]
['b', 'c', 'd']
```

Observe that in each of these cases, whenever we write `x[start:stop]` the output is an array of size `stop - start`. This will be useful to keep in mind.

All that is left to do is to compute the index of the middle element of the array. Intuitively, the middle element is halfway through the array, and so the index should be approximately `len(arr)/2`. Of course, if `arr` has an odd number of elements, this will not be a whole number and thus not a valid index. In this case, we need to round – the question is, do we round up or down? To help decide, suppose `arr` has three elements, like below:

```
[0, 1, 2]
```

Here, $n/2 = 3/2 = 1.5$. If we set `middle` by rounding down to one, we split the array into `[0]` and `[1,2]`. On the other hand, if we split the array by rounding 1.5 up to two, we split the array into `[0,1]` and `[2]`. It turns out that either approach will work – we'll just pick one and stick with it to be consistent. So, in what follows, we will round down.

Rounding down in this way is called taking the **floor** of the number. More precisely, the floor of a real number x is the largest integer which is less than or equal to x. For example, the floor of 1.5 is 1; the floor of -2.3 is -3; and the floor of 4 is simply 4. In mathematical notation, we write $\lfloor x \rfloor$ to denote the floor of x. Python provides the `math.floor` function for computing the floor[5]. As you might have guessed, the *smallest* integer *greater* than or equal to x is called the **ceiling** of x, and is denoted by $\lceil x \rceil$.

We have therefore filled in the first part of the `mergesort` function:

```python
import math

def mergesort(arr):
    if len(arr) > 1:
        # split the array in half
        middle = math.floor(len(arr) / 2)
        left = arr[:middle]
        right = arr[middle:]

        # sort the left half
        # TODO
        # sort the right half
        # TODO
        # merge the left and right halves
        # TODO
```

[5]Python also provides a "flooring division" operation, `//`, which takes the floor after dividing. That is, `len(arr) // 2` will produce the same result as `math.floor(len(arr) / 2)`. We will use `math.floor` because it is more explicit.

Sorting the Halves

After splitting the array into two halves, merge sort next sorts each. In principle, we could do this using *any* sorting method, such as selection sort. While the resulting algorithm would work, it turns out that it would be no faster than just using selection sort to sort the whole list. Instead, merge sort takes a leap of faith and uses merge sort to sort each half:

```
import math

def mergesort(arr):
    if len(arr) > 1:
        # split the array in half
        middle = math.floor(len(arr) / 2)
        left = arr[:middle]
        right = arr[middle:]

        # sort the left half
        mergesort(left)
        # sort the right half
        mergesort(right)

        # merge the left and right halves
        # TODO
```

Note that `mergesort(left)` and `mergesort(right)` do not return new arrays – they modify `left` and `right`.

Merging the Sorted Halves

Our function `mergesort` recursively sorts each half of the input array `arr` by modifying `left` and `right`. Assuming for the moment that these recursive calls work as expected (i.e., assuming that `left` and `right` are sorted), there remains the work of combining `left` and `right` into a larger sorted array, which we will return by modifying `arr`. This is the job of the `merge` function, which we will now design.

Consider the problem of combining two sorted stacks of cards into one sorted stack. At any moment, you can only see the top of each stack. To merge these two stacks into one, you repeatedly take the smallest card off of either stack. You will eventually remove all of the cards from one of the stacks, but that's fine: you just continue drawing cards from the remaining stack until it, too, is depleted. Convince yourself that this procedure correctly merges two sorted stacks of cards into one sorted stack.

Our `merge` function will operate in the same way – the Python implementation is shown below. Several aspects are worth of explanation. First, the function's arguments

are the arrays to merge, `left` and `right`, but also `arr`; this is because we will store the result in `arr`. Second, our function appends `float('inf')` to the end of `left` and `right`[6]. This special value is known as a **sentinel**, and the reason for adding it will be clear in a moment. In short, its purpose is to make the algorithm cleaner and simpler[7] Lastly, we will keep track of the "front" of the left and right arrays with variables `left_ix` and `right_ix`; both initialized to zero.

```
def merge(left, right, arr):
    left.append(float('inf'))
    right.append(float('inf'))
    left_ix = 0
    right_ix = 0
    for ix in range(len(arr)):
        if left[left_ix] < right[right_ix]:
            arr[ix] = left[left_ix]
            left_ix += 1
        else:
            arr[ix] = right[right_ix]
            right_ix += 1
```

The core of the function is a loop ranging from 0 up to (and excluding) `len(arr)`. We can think of the loop variable, `ix`, as being the index of the element of `arr` whose value is currently being decided. At every iteration, the numbers at the "front" of `left` and `right` – that is, `left[left_ix]` and `right[right_ix]` – are compared. The smaller of the two is placed into `arr` at the index `ix`, and the "front" of the corresponding array is moved one element to the right by incrementing `left_ix` or `right_ix` as necessary. At the end of the αth iteration, the first α elements of `arr` will be in their final sorted order. Since the loop performs `len(arr)` iterations, `arr` will be sorted after the loop is finished.

The purpose of the `float('inf')` at the end of each half is to simplify the algorithm's handling of the situation where one "stack of cards" becomes empty. When this occurs in the above implementation, the "front" number on the "empty" array will be `float('inf')`. When the "front" of each array is compared, the other array will always win, since `float('inf')` is larger than any number. The other array will continue winning until it, too, is depleted. But we don't need to check that both arrays have been emptied; we know that we need to merge precisely `len(arr)` elements, and so we are done once we've made that many iterations.

[6] Recall that `float('inf')` is how ∞ is written in Python

[7] Appending to a Python list is (roughly-speaking) a constant-time operation, while appending to a NumPy array is a linear-time operation. We'll assume here that the input is a Python list (NumPy arrays don't have an `append` method, anyways). It should be stated that we can implement the `merge` function without appending an ∞; the code just becomes uglier.

Algorithm 3 Merge sort.

```
import math

def mergesort(arr):
    """Sorts array using merge sort."""
    if len(arr) > 1:
        # split the array in half
        middle = math.floor(len(arr) / 2)
        left = arr[:middle]
        right = arr[middle:]

        # sort the left half
        mergesort(left)
        # sort the right half
        mergesort(right)

        # merge the left and right halves
        merge(left, right, arr)

def merge(left, right, arr):
    left.append(float('inf'))
    right.append(float('inf'))
    left_ix = 0
    right_ix = 0
    for ix in range(len(arr)):
        if left[left_ix] < right[right_ix]:
            arr[ix] = left[left_ix]
            left_ix += 1
        else:
            arr[ix] = right[right_ix]
            right_ix += 1
```

Merging the left and right halves completes the algorithm. The full code is shown in Algorithm 3.

2.3.2 Understanding Merge Sort

Merge sort can seem like magic at first glance, and it isn't easy to see where the sorting actually occurs. We will prove that `mergesort` is correct in a moment, but first we will try to gain some intuition for how the algorithm works.

The Sorting Happens in `merge`

It is interesting to note that no actual sorting occurs in the main `mergesort` function. Instead, the sorting occurs in the `merge` subroutine.

To see this, consider what happens when we execute `mergesort([5, 2])`. The function splits this array into `left = [5]` and `right = [2]` and makes recursive calls to `mergesort([5])` and `mergesort([2])`. These recursive calls exit without doing anything, since their inputs are arrays of size one. Hence `mergesort` calls `merge([5], [2], [5, 2])`.

The `merge` function is given two "stacks of cards", each with one element. It will take the smaller of the two cards and place it into the first slot in `arr`, then it will place the other card into the second slot – this is where the elements are sorted! Before calling `merge`, `arr` is `[5, 2]`. After calling `merge`, `arr` is `[2, 5]`, and in sorted order.

Tracing a Merge Sort

Figure 2.1 depicts a merge sort on the array `[7, 3, 1, 6, 2, 5, 8, 4]`. The figure shows that the input array is recursively halved until arrays of size one are obtained. These are then merged to form sorted arrays of size two, which are in turn merged to form sorted arrays of size four, and so on.

While helpful, such a depiction does not fully capture the order in which the various calls to `mergesort` and `merge` are made. A helpful exercise is to imagine "bugging" `mergesort` and `merge` by placing `print` statements on the first and last lines of each[8]:

```
def mergesort(arr):
    args = arr.copy()
    print(f'Starting mergesort{args}...')
    ...
    print(f'Finishing mergesort{args}...')

def merge(left, right, arr):
    args = (left.copy(), right.copy(), arr.copy())
    print(f'Starting merge{args}...')
```

[8]We copy the arguments because we will later want to print them as they were when the function was called, but their values will change over the course of the function.

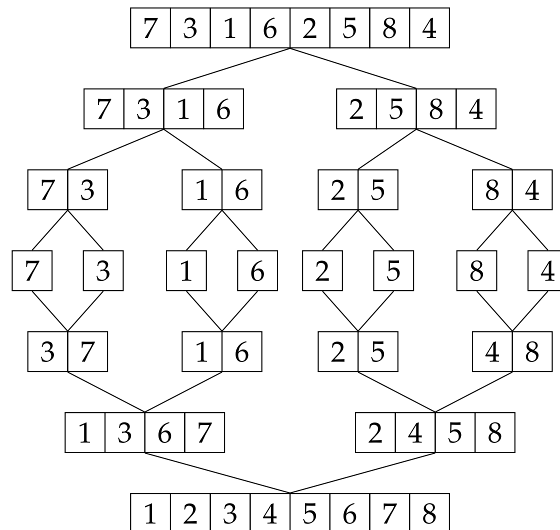


Figure 2.1: Merge sort operating on an array.

```
    ...
    print(f'Finishing merge{args}...')
```

For instance, when `mergesort` is called on the list above, the first few lines of output will be:

```
Starting mergesort([7, 3, 1, 6, 2, 5, 8, 4],)
Starting mergesort([7, 3, 1, 6],)
Starting mergesort([7, 3],)
Starting mergesort([7],)
Finishing mergesort([7],)
Starting mergesort([3],)
Finishing mergesort([3],)
Starting merge([7], [3], [7, 3])
Finishing merge([7], [3], [7, 3])
...
```

Note that there are many calls to `mergesort` before even the first call to `merge`! Tracing out the rest of the function's execution is left as an exercise.

2.3.3 Correctness

Merge sort takes a leap of faith whenever it recursively calls itself to sort each half of the original array. We now prove that this leap is warranted, and that merge sort indeed sorts correctly. More precisely, we will prove the following claim:

Theorem 6. *`mergesort` correctly sorts arrays of size n, for any $n \geq 0$.*

As with proving the correctness of our `factorial` function before, we will use the method of proof by induction. Recall that a proof by induction has two parts: the **base case** and the **inductive step**. Here, there are actually two bases cases: when the size of the input array is zero, and when it is one. In the inductive step, we show that if `mergesort` functions correctly on inputs of size less than k, it will work on inputs of size k.

Proof of Theorem 6. BASE CASE(S): $n = 0$ AND $n = 1$. An empty array and an array with one element are both considered to be trivially sorted. When $n = 0$ and when $n = 1$, `mergesort` simply returns without modifying the array. Hence in both cases, the algorithm is correct.

INDUCTIVE STEP: Let $k \geq 2$ be such that `mergesort` correctly sorts all arrays whose size is strictly less than k. We will show that `mergesort` correctly sorts an arbitrary array `arr` of size k. When `mergesort(arr)` is called on an array of size $k \geq 2$, the algorithm will make two recursive calls: one to `mergesort(left)` and another to `mergesort(right)`. The size of `left` is less than k, as is the size of `right`[9]. Hence, by assumption, `left` and `right` are correctly sorted by the recursive calls. Assuming that `merge` correctly merges two sorted arrays, `arr` will contain the elements it started with, but in sorted order. Hence the algorithm is correct.

□

2.3.4 Time Complexity

Merge sort is undoubtedly more clever than selection sort. This cleverness makes it harder to understand, but it also results in a huge performance boost. In fact, it turns out that merge sort achieves the best time complexity possible for sorting[10]!

Writing the Recurrence Relation

We will find the time complexity of `mergesort` in the same way that we analyzed `factorial`: by writing down and solving a recurrence relation. To begin, let $T(n)$ denote the time it takes for `mergesort` to run on an input of size n. We already know what $T(n)$ is when $n = 0$ or $n = 1$; in these cases, `mergesort` simply returns without recursing, and hence takes constant time. Therefore $T(0) = T(1) = \Theta(1)$.

On the other hand, a call to `mergesort` with an array of size two or larger spends its time in three different ways. First, it calculates the middle of the array and splits it

[9] A more careful argument would actually prove that this is true! Here, we will take it for granted.

[10] More specifically, it achieves the best time complexity possible for algorithms which sort by comparing elements to see which is smaller. Comparing elements is not always needed to sort. For instance, if is known that the input are the first ten natural numbers in shuffled order, we can immediately put each number in its correct position without comparing it to the others.

into two pieces. Splitting an array by slicing produces new arrays, and each of the n values must be copied from `arr` to either `left` or `right`, which takes $\Theta(n)$ time. Next, `mergesort` calls itself recursively on arrays which are roughly half the size. For simplicity, let's assume from here on that n is a perfect power of 2. Then each recursive call operates on an array of size $n/2$. Therefore, each of these recursive calls takes time $T(n/2)$, for a total time of $2 \cdot T(n/2)$. Lastly, we merge the two sorted sub-arrays; a quick analysis of the `merge` algorithm shows that this takes time $\Theta(n)$. Altogether, the time spent in one call to `mergesort` when $n \geq 2$ is

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

Since $\Theta(n) + \Theta(n)$ is just $\Theta(n)$ once again, we can simplify a little:

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

In total, we have:

$$T(n) = \begin{cases} \Theta(1), & n \in \{0, 1\}, \\ 2 \cdot T(n/2) + \Theta(n), & n \geq 2. \end{cases}$$

This is a recurrence relation, and we would like to "solve" it; that is, find a nice, closed form expression for $T(n)$ which doesn't refer to itself.

In the above, we have used the Θ-notation in a new way. When we write $T(n) = 2T(n/2) + \Theta(n)$, what we mean precisely is that there is an unknown function $f(n)$ such that $f(n) = \Theta(n)$ and the equality $T(n) = 2T(n/2) + f(n)$ holds for all n.

Solving the Recurrence Relation

Before we solve the above recurrence, we will perform one small modification that will make our lives easier. It turns out that we can replace the $\Theta(n)$ with n without changing the solution's *asymptotic* bounds; likewise, we can replace $\Theta(1)$ with 1. That is, if we are only interested in writing $T(n) = \Theta(\cdot)$, then we may instead solve the simpler recurrence:

$$T(n) = \begin{cases} 1, & n \in \{0, 1\}, \\ 2 \cdot T(n/2) + n, & n \geq 2. \end{cases}$$

We will solve this recurrence relation by "unrolling" it, as we did when we solved the recurrence for `factorial` on page 53. Recall that the first step of the procedure is to unroll the recurrence until a pattern emerges. To start, we have:

$$T(n) = 2 \cdot T(n/2) + n$$

Let's call this step #1 of our unrolling procedure. Now, if we plug in $n/2$ for every n, we obtain $T(n/2) = 2 \cdot T(n/4) + n/2$. Let's make that substitution for $T(n/2)$ in the above:

$$= 2\,[2 \cdot T(n/4) + n/2] + n,$$
$$= 4 \cdot T(n/4) + n + n,$$
$$= 4 \cdot T(n/4) + 2n.$$

That was step #2 of the procedure. Now we want to expand $T(n/4)$. Plugging in $n/4$ for every n in our original recurrence relation, we get $T(n/4) = 2 \cdot T(n/8) + n/4$. Making this substitution, we find that, after iteration $k = 3$:

$$= 4\,[2 \cdot T(n/8) + n/4] + 2n,$$
$$= 8 \cdot T(n/8) + 3n.$$

After the $k = 4$ iteration, we'll get:

$$= 16 \cdot T(n/16) + 4n.$$

Let's place these results into a table so that finding a pattern is easier:

Step	Formula
1	$T(n) = 2 \cdot T(n/2) + n$
2	$T(n) = 4 \cdot T(n/4) + 2n$
3	$T(n) = 8 \cdot T(n/8) + 3n$
4	$T(n) = 16 \cdot T(n/16) + 4n$

It seems that the coefficients on T and in the arguments to the functions are powers of two. In fact, a little thought shows that on the kth iteration of the procedure, we'll arrive at:

$$= 2^k \cdot T(n/2^k) + kn.$$

The next step of solving the recurrence is to find how many steps it takes to reach the base case. In this problem, the base case is when $n = 1$. Hence we want to find the step number k such that $n/2^k = 1$. Multiplying both sides by 2^k, we find that $2^k = n$. Taking $\log_2$ of both sides results in $k = \log_2 n$. That is, after $\log_2 n$ steps, we will find $T(1)$ on the right hand side of our formula for $T(n)$.

Lastly, we substitute this step number into our general formula. Replacing k with $\log_2 n$, we find:

$$T(n) = 2^{\log_2 n} \cdot T(n/2^{\log_2 n}) + n \log_2 n$$

Recall that $2^{\log_2 n}$ is simply n. Hence:

$$\begin{aligned}
&= n \cdot T(n/n) + n \log_2 n, \\
&= n \cdot T(1) + n \log_2 n.
\end{aligned}$$

Since $T(1) = 1$, we end up with:

$$\begin{aligned}
&= n + n \log_2 n, \\
&= \Theta(n \log_2 n).
\end{aligned}$$

This solves the recurrence, and the time complexity of merge sort is $\Theta(n \log_2 n)$.

> Note that $\log_2 n$ differs from $\ln n$ by a constant factor. In fact, if b is any valid logarithm base, $\log_2 n = (\log_b n)/(\log_b 2)$. Since Θ "ignores" constant factors, we could just as well have written $T(n) = \Theta(n \ln n)$, $T(n) = \Theta(n \log_{10} n)$, or even $T(n) = \Theta(n \log_\pi n)$. That is, the choice of base is arbitrary when using asymptotic notation. Because of this, we typically write $T(n) = \Theta(n \log n)$ without specifying a base.

Comparison to selection sort. We have seen that the time complexity of merge sort is $\Theta(n \log n)$. Compare this to the time complexity of selection sort, which is $\Theta(n^2)$. Is that a big improvement? Absolutely! Suppose it takes selection sort n^2 nanoseconds to sort a list, while it takes merge sort $n \log_2 n$ nanoseconds. To sort a list of 1 million elements, selection sort requires $(1 \text{ million})^2$ nanoseconds, or 16 and 2/3 minutes. Merge sort, however, requires only about 20 milliseconds. This is because $\log_2$ grows very slowly: in fact, $\log_2(1 \text{ million}) \approx 20$.

2.4 The Value of Sorted Data

Sorting is an important data science operation in its own right, but it turns out that many other apparently unrelated computational problems become easier to solve if the input is already sorted. We will see several of such problems in this section. We will use this opportunity to practice designing non-trivial algorithms which exploit the special structure of a problem in order to achieve impressive speedups – these types of problems are commonly posed in software engineering and data science interviews.

2.4.1 Searching

Suppose we are given a table with n rows, each of them containing information about a different movie. We wish to find the index of the row corresponding to the classic American comedy, "The Wicker Man". The straightforward approach to finding this index is

to loop over the table's rows and check, one-by-one, if each contains the desired movie. When we find the desired row, we stop and return its index.

The algorithm we have just described is called **linear search**. The below Python code implements the algorithm in the more general setting in which we wish to find the index of an object (called `target`) in a list (called `things`). If the object is not in the list, the function returns `None`.

```
def linear_search(things, target):
    for ix, thing in enumerate(things): # see below
        if thing == target:
            return ix
    return None
```

Python's `enumerate`. Python's usual `for`-loop just produces the elements in a container, such as a list. But what if we want not only the elements, but their indices, too? That is where `enumerate` comes in. An example should help make its behavior clear. Let `things = ['a', 'b', 'c']`. Then the output of:

```
for ix, thing in enumerate(things):
    print(ix, thing)
```

will be: `0 'a'`, then: `1 'b'`, and finally: `2 'c'`.

Best and Worst Case Time Complexities

What is the time complexity of `linear_search`? As usual, this will depend upon the number of iterations of the loop, but here we see something new: the number of iterations depends not only on the size of `things`, but also on the particular `target` that is provided. If the `target` happens to be the first thing in `things`, the loop only iterates once – this is the **best case** out of all possible inputs. In the **worst case**, the target isn't in the list at all, and the loop goes through `len(things)` iterations.

More formally, let $T_{\text{best}}(n)$ be the time taken by `linear_search` on a list of size n in the best case. That is, to compute $T_{\text{best}}(n)$, we time `linear_search` on all targets and all lists of size n and return the *smallest* time. Of course, $T_{\text{best}}(n)$ is a function of n, and we can express the growth of this function using asymptotic notation: $T_{\text{best}}(n) = \Theta(1)$. This is the **best case time complexity** of `linear_search`. Likewise, let $T_{\text{worst}}(n)$ be the time taken by `linear_search` on a list of size n in the worst case. By analogy, to compute $T_{\text{worst}}(n)$, we time the function on all targets and all lists of size n and return the *largest* time. Here, the worst case takes time $T_{\text{worst}}(n) = \Theta(n)$. This is the **worst case time complexity** of `linear_search`.

We tend to focus on the worst case time complexity when discussing the efficiency of algorithms[11]. In particular, we often pick algorithms based on which has better performance in the worst case. For example, **insertion sort** is a sorting algorithm with $\Theta(n)$ best case time complexity and $\Theta(n^2)$ worst case time complexity. Merge sort, on the other hand, has matching best case and worst case time complexities of $\Theta(n \log n)$. While the best case time complexity of insertion sort actually beats that of merge sort, we tend to prefer merge sort because it is has a much faster worst case time complexity[12].

Moreover, we can sometimes lower bound the worst case time complexity for a particular problem. For example, if we assume nothing about the order of the list to be searched, any algorithm must loop through the entire list in the worst case, taking at least linear time. Since `linear_search` takes $\Theta(n)$ in the worst case, it optimal, in the sense that no other algorithm has a better worst case time complexity. Similarly, it can be shown that any sorting algorithm[13] takes at least $\Omega(n \log n)$ time in the worst case. Since merge sort takes $\Theta(n \log n)$ in the worst case, it is – in a sense – an optimal sorting algorithm.

Lastly, when stating the time complexity of `linear_search`, we should report *both* the worst case and best case time complexities. We should *not* say that the "time complexity is $O(n)$ and $\Omega(1)$", because this is needlessly imprecise. For instance, the previous statement also holds for an algorithm which takes time $\Theta(\sqrt{n})$ in both the best and worst cases.

Binary Search

Linear search has the best possible worst case time complexity if we make no assumption on the input list. But what if the input is sorted? It turns out that we can do *much* better.

To motivate the algorithm, remember the following game (you have probably played it before while on a long road trip). I am thinking of a number between 1 and 100. Your task is to guess the number in as few guesses as possible. After each guess, I tell you whether your guess is too high, too low, or that it is correct.

If you were a fan of `linear_search`, you might try guessing 1 first, and then 2, and then 3, and so on, until you eventually come to the correct number. But this strategy feels very inefficient. Instead, you might try guessing 50 first. I tell you that 50 is too high of a number, and to guess again. With this information, you can immediately rule out not only 50, but also every number greater than 50 – half of the possibilities! In fact, the only valid guesses remaining are 1 through 49. You try the same strategy again, and guess 25. I tell you that this is too low. Now you know that the number is between 26 and 49.

---

[11] But not always! We may also use the **average case time complexity** of an algorithm, which is potentially much better than its worst case performance. Computing the average case time complexity can be difficult.

[12] We are leaving out many details of the comparison. It is conceivable that insertion sort's worst case occurs on a very rare input, and that the average case time complexity is much better. It turns out, however, that insertion sort's average case time complexity is also quadratic. In general, however, describing the performance of algorithm based on its worst case complexity can be misleading.

[13] To be more precise, any *comparison* sorting algorithm which works by comparing the list's elements to one another.

Algorithm 4 Binary search

```python
import math

def binary_search(things, target, start, stop):
    """
    Searches things[start:stop] for the target.
    Assumes that `things` is sorted.
    """
    if start >= stop:
        return None

    middle = math.floor((start + stop) / 2)
    if things[middle] == target:
        return middle
    elif things[middle] > target:
        return binary_search(things, target, start, middle)
    else:
        return binary_search(things, target, middle + 1, stop)
```

You continue on, at each step halving the number of possibilities, until you find that the number I was thinking of was 42. This is the principle idea behind **binary search**, a very fast search algorithm whose code is shown in Algorithm 4.

Binary search shares some similarities with merge short. At every step, it breaks the array into two pieces[14]. Unlike merge sort, it recursively searches only one of the pieces, and not both. This is possible only because `things` is assumed to be sorted. Once we compare the target to the middle element of the array, we know that we only need to check either the left half or the right half, precisely because the list is sorted.

Time Complexity of Binary Search

Like linear search, binary search will also have a *best case* time complexity and a *worst case* time complexity. In the best case, the thing we are looking for is right at the middle of the list, and the answer is returned without making any recursive calls. Hence the best case time complexity is $\Theta(1)$. The worst case input for linear search is also the worst case for binary search: the item we are looking for isn't in the list. Computing the time com-

[14]We do not physically break the array into two pieces by slicing, as we did in merge sort. That is because slicing produces a copy, and copying takes linear time. Copying was necessary in merge sort, but it is not necessary here – we instead keep track of the index of the middle of the list.

plexity in this situation is similar to the analysis of merge sort in that it requires solving a recurrence relation.

Let $T_{\text{worst}}(n)$ be the time binary search takes whenever $n = \texttt{stop} - \texttt{start}$, but `target` is not in `things`. Apart from the recursive call, everything else – computing the floor, checking whether the middle element is the target, etc. – takes constant time. Although there are two appearances of `binary_search` within the function body, only one of them will run on a given call. Whichever call is made, it operates on a sub-array of `things` that has at most $n/2$ elements. Therefore, the time it takes is at most $T_{\text{worst}}(n/2)$. We therefore arrive at the following recurrence relation:

$$T_{\text{worst}}(n) = \begin{cases} \Theta(1), & n = 1 \\ T_{\text{worst}}(n/2) + \Theta(1), & n \geq 2 \end{cases}$$

As mentioned in the previous section, we can replace $\Theta(1)$ by 1 as long as we are OK with a final answer in Θ-notation. Making this substitution:

$$= \begin{cases} 1, & n = 1 \\ T_{\text{worst}}(n/2) + 1, & n \geq 2 \end{cases}$$

Now we solve the recurrence by unrolling it. Since $T_{\text{worst}}(n/2) = T_{\text{worst}}(n/4) + 1$, we have:

$$\begin{aligned} &= (T_{\text{worst}}(n/4) + 1) + 1, \\ &= T_{\text{worst}}(n/4) + 2. \end{aligned}$$

And since $T_{\text{worst}}(n/4) = T_{\text{worst}}(n/8) + 1$, we also have:

$$\begin{aligned} &= (T_{\text{worst}}(n/8) + 1) + 2, \\ &= T_{\text{worst}}(n/8) + 3. \end{aligned}$$

Continuing in this way, we see that on the kth step:

$$= T_{\text{worst}}(n/2^k) + k.$$

We know that $T_{\text{worst}}(1)$ takes constant time, since this is the base case of the recursion. Therefore we iterate this process until $2^k = n$. Assuming that n is a perfect power of two, this occurs when $k = \log_2 n$. Substituting this value of k into the above, we find:

$$\begin{aligned}
&= T_{\text{worst}}(n/2^{\log_2 n}) + 1 \log_2 n, \\
&= T_{\text{worst}}(n/n) + \log_2 n, \\
&= T_{\text{worst}}(1) + \log_2 n, \\
&= 1 + \log_2 n, \\
&= \Theta(\log n).
\end{aligned}$$

We have just shown that the worst case time complexity of binary search is $\Theta(\log n)$. To get an idea of just how fast this is, consider the following: Suppose that `linear_search` takes n nanoseconds to find an element in the worst case if the input is a list of size n, while `binary_search` takes $\log n$ nanoseconds in the worst case. It is estimated that there are something like 10^{19} grains of sand on Earth. Suppose we assign each a unique number and line them up from least to greatest. In the worst case, linear search would take roughly 10^{19} nanoseconds, or roughly 317 years, to search for a particular grain of sand. Binary search, on the other hand, would take roughly 63 nanoseconds in the worst case. If you're still not impressed, consider the same problem – but replace "grains of sand on Earth" with "atoms in the known universe". A loose estimate of this latter quantity is 10^{80}. In the worst case, binary search can find a single atom – out of all of the atoms in the universe – in roughly 265 nanoseconds.

If this seems like it is too fast to be true, remember that binary search makes an important assumption that linear search does not: it requires the input to be sorted. It turns out that sorting a collection of n objects takes $\Omega(n \log n)$ time in the worst case, which is just slightly worse than $\Omega(n)$. Therefore, if you are given unsorted data and you're doing a single search, linear search is actually better. But if you are doing many subsequent searches you might consider sorting the data and using binary search thereafter.

2.4.2 Picking In-Flight Movies

Suppose you are on a long flight to Hawaii. The airline lost your baggage the last time you flew with them, and so they've generously given you a voucher for *two* free in-flight movies. In order to minimize your boredom, you decide to find two movies such that the sum of their durations is as close as possible to the duration of your flight. That is, we wish to solve the following computational problem:

> GIVEN: A list of n positive movie durations $d_0, \ldots, d_{n-1}$ and a positive flight duration, t.
> COMPUTE: A pair of indices $a, b \in \{0, \ldots n-1\}$ such that $|(d_a + d_b) - t|$ is minimized.

This is an optimization problem. The objective function is $L(a,b) = |(d_a + d_b) - t|$, and the search space is the set of all distinct pairs of numbers (a,b) taken from $\{1,\ldots,n\}$. The size of the search space is $\Theta(n^2)$, and so the brute force algorithm solves the problem in quadratic time.

Shrinking the Search Space

Suppose now that the input durations are sorted; that is, $d_0 \leq d_1 \leq \ldots \leq d_{n-1}$. We will design an algorithm which solves this problem in linear time[15]. While a brute force search works by computing the objective function L on every element of the search space and returning the minimizer, the new algorithm will eliminate large parts of the search space on every iteration, shrinking it more and more until eventually nothing remains.

To see how the search space can be shrunk, consider the following concrete example. The movies we can choose from have durations of $10, 20, 40, 50$, and 100 minutes, respectively, and our flight length, t, is 65 minutes long (short flight!). Suppose we first try pairing the shortest movie (10 minutes) with the longest (100 minutes). In total, these two movies will run for 110 minutes. This is longer than the flight by 45 minutes! We now need to check if there is a better pair, but must we check *all* of them? No! Note that any other pair involving the 100 minute movie will have *at least* as long of a total run time, since the 10 minute movie was the shortest! In other words, out of any pair of movies containing the 100 minute movie, this pair was the best possible. Hence we do not need to check these other pairs: we can effectively eliminate any containing the 100 minute movie from the search space.

Next, we again consider pairing the shortest remaining movie with the longest. The shortest movie is still the 10 minute movie, but the longest remaining movie is the 50 minute movie. This pair has total duration of 60 minutes, which is 5 minutes *less* than the flight duration. This is much better than the previous pairing, and is the best we have seen so far. We still need to check that there isn't a better pairing, however. We do not need to consider pairings which involve the 100 minute movie – these have already been eliminated. But we also no longer need to consider pairings containing the 10 minute movie. To see why, note that pairing any other remaining movie with the 10 minute movie can only *decrease* the total run time, moving it further from the target of 65 minutes. Hence we no longer need to consider pairs involving the 10 minute movie.

We continue on like this, at each step considering the pair of the shortest and longest remaining movies. If the total run time is longer than the target of 65 minutes, we eliminate the longer movie; otherwise we eliminate the shorter of the two. We keep track of the best pair seen so far, and return it once we have eliminated all pairs.

[15] If the input is not sorted, we can use merge sort to first sort them in $\Theta(n \log n)$ time, resulting in an overall time complexity of $\Theta(n \log n) + \Theta(n) = \Theta(n \log n)$.

Algorithm 5 Find the pair of movies whose total duration is closest to the target.

```
def optimize_entertainment(durations, target):
    shortest = 0
    longest = len(durations) - 1

    best_pair = None
    best_objective = float('inf')

    for i in range(len(durations) - 1):
        total_duration = durations[shortest] + durations[longest]

        if abs(total_duration - target) < best_objective:
            best_objective = abs(total_duration - target)
            best_pair = (shortest, longest)

        if total_duration == target:
            return (shortest, longest)
        elif total_duration < target:
            shortest += 1
        else: # total_duration > target
            longest -= 1

    return best_pair
```

Implementation

We now turn the intuition we've gained from this concrete example into the algorithm shown in Algorithm 5. We are given as input a list `durations` which contains the movie durations in ascending order, along with a floating-point number `target` which represents the target duration (i.e., the duration of the flight). We first introduce variables `shortest` and `longest` which will hold the indices of the shortest and longest remaining movies in `durations`, respectively. At the beginning of the algorithm, no movie pairs have been eliminated, so `shortest = 0` and `longest = len(durations) - 1`. Also introduce a variable `best_objective`, which contains the smallest value of the objective seen so far, and a variable `best_pair`, which tracks the corresponding pair of movies.

We then begin iterating. At every iteration, we will check the pair containing the shortest remaining movie and the longest remaining movie. We compare this pair to the best pair seen so far; if it is better, we update `best_objective` and `best_pair`. Having done this, we next shrink the search space for the next iteration. If the total duration of

the pair is exactly equal to the target, we can simply return the pair without proceeding to the next iteration, since it is not possible to do better than this. On the other hand, if the total duration of this pair is longer than the target, we can eliminate the longer movie following the logic from the previous part. We must then update `longest` so that it is the index of the longest movie remaining. Because `durations` is sorted, this index is simply `longest - 1`, and we therefore update `longest` by decrementing it. Likewise, if the total duration of the pair is shorter than the target, we eliminate the shorter of the pair by incrementing `shortest`.

We iterate until `shortest` and `longest` are equal to each other, as at that point all pairs consisting of unique movies have been eliminated. We could implement this with a `while`-loop, but we in fact know exactly how many iterations should be performed. Before the loop begins, the difference between `shortest` and `longest` is $n-1$, where n is the number of movies. With each iteration, the difference between the two decreases by one. After $n-1$ iterations, the difference will be zero, and we should terminate the loop. Therefore, we iterate $n-1$ times.

Chapter 3

Graphs

Contents

3.1 Definitions . . . 78
3.1.1 Directed Graphs . . . 79
3.1.2 Undirected Graphs . . . 81
3.1.3 Terminology Involving Edges . . . 82
3.1.4 Paths and Connectedness . . . 83
3.2 Computer Representations of Graphs . . . 86
3.2.1 Adjacency Matrices . . . 86
3.2.2 Adjacency Lists . . . 88
3.2.3 Dictionaries of Sets . . . 88
3.2.4 Node Attributes . . . 90
3.3 Breadth-First Search . . . 91
3.3.1 Search Strategies . . . 91
3.3.2 The Algorithm . . . 94
3.3.3 Properties of BFS . . . 98
3.3.4 Time Complexity of BFS . . . 99
3.3.5 Shortest Paths . . . 100
3.3.6 The Breadth-First Search Tree . . . 105
3.4 Depth-First Search . . . 107
3.4.1 The Algorithm . . . 109
3.4.2 Time Complexity . . . 112
3.4.3 Start and Finish Times . . . 113
3.4.4 DFS Forest . . . 119
3.4.5 Topological Sort . . . 120

3.4.6 Edge Classification 122
3.5 Weighted Graphs and Minimum Spanning Trees 123
3.5.1 Distance Graphs 124
3.5.2 Minimum Spanning Trees 124
3.5.3 Clustering 126

The first step in answering a data science question is to formulate it as a mathematical problem involving data. But raw data by itself may not be expressed in the language of mathematics – it could be, for instance, a video recorded by a camera at the front of a car, a collection of text documents, or a list of the friendships between students at a school. In order to frame the question mathematically, we must choose a way of representing these entities (i.e., a video, a document, a list of friendships) as mathematical objects.

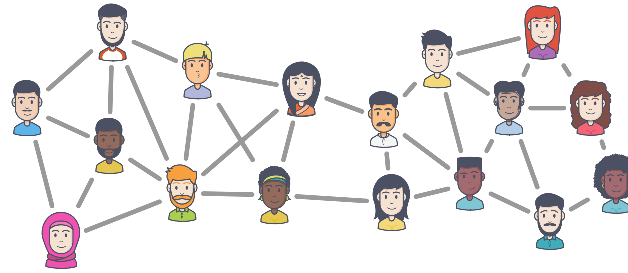
Two types of mathematical objects are most frequently used to represent data: **vectors** and **graphs**. Vector representations are especially useful when *attributes* (or **features**) are of primary importance. For instance, suppose we would like to predict the probability that a person will develop heart disease. Modern medicine tells us that certain risk factors, such as the individual's age, cholesterol level, and sugar consumption, are most predictive. It is therefore natural to represent a person as a list (or **vector**) of three numbers, each quantifying the extent to which a different risk factor is present in the individual. We will revisit vector representations in later chapters.

Graph representations, on the other hand, are used when the *relationship* between entities is of primary importance. For example, suppose that a highly-contagious virus has broken out in San Diego, and we wish to understand how it will spread. Naturally, the virus is likely to spread quickly among communities of people who interact frequently with one another. Therefore, an individual's attributes are not quite as important as their relationships with other individuals. A **graph** is a natural mathematical object for encoding such relationships.

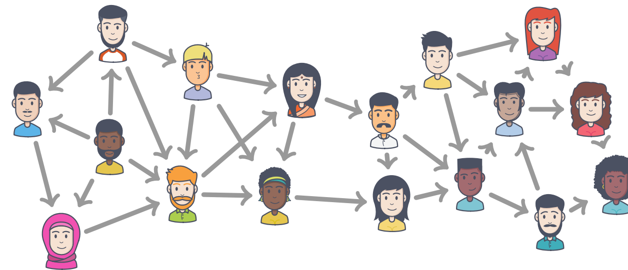
This chapter introduces graphs as mathematical objects and develops their theory. We will then see how to represent graphs on a computer, and study several key algorithms for working with graph data. We will conclude by developing graph-based methods for performing fundamental data science tasks, such as clustering.

3.1 Definitions

A **graph** is a mathematical structure that encodes relationships between pairs of objects. In data science, we may use a graph to represent, for example the relationships between pairs of users in a social network – say, Facebook. In Facebook, users are the **nodes** of the network. If we draw a line – or an **edge** – between any two users who are "friends", then we obtain something like the below picture:



We can do something similar to represent Twitter, too. But note that Twitter's notion of a relationship between users is different than Facebook's in that it is *directional*. That is, a Facebook "friendship" is mutual – if I am Facebook friends with you, then you are Facebook friends with me. But Twitter's relationships are not mutual: I may follow you on Twitter, but you may not follow me back. Therefore, when we draw a picture depicting Twitter, our edges have arrows attached which capture who follows whom:



While we refer to both social networks as graphs, we distinguish them by the kinds of edges they contain. A graph like Facebook whose edges have no direction (i.e., they are lines instead of arrows) is called an **undirected graph**. A graph like Twitter whose edges have a direction (i.e., they are arrows instead of lines) is called a **directed graph**.

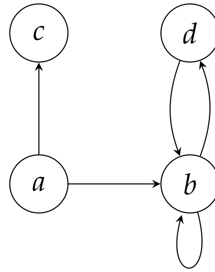
3.1.1 Directed Graphs

We begin by formally defining a directed graph:

Definition 4. A **directed graph** (or **digraph**) G is a pair (V, E), where V is a finite set of objects (called the **nodes** or **vertices**) and E is a set of ordered pairs of objects in V (called the **edges**).

In the case of Twitter, the node set V is the set of users, and the edge set E consists of ordered pairs of users. We will find a pair (a, b) in E if (and only if) user a follows user b.

Oftentimes when we work with graphs, we assume nothing about the elements of V; that is, they are not people or users, but rather abstract "objects". It is helpful to number them or give them names. For instance, the below depicts an abstract graph whose nodes are named a, b, c, and d:

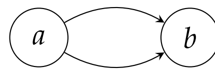


The node set of this graph is $V = \{a, b, c, d\}$. The edge set is:

$$E = \{(a, c), (a, b), (d, b), (b, d), (b, b)\}.$$

Some of the edges in the above graph deserve special mention. First, notice that there is a **self-loop** from node b to itself, which we have written as (b, b). These edges are permissible according to Definition 4, which simply states that E is a "set of ordered pairs". Since (b, b) is an ordered pair, it is a valid edge. Self-loops cannot occur in Twitter, since users cannot follow themselves. On the other hand, consider a graph of Reddit in which an edge occurs between two users a and b if users a has liked a post by user b. Since Reddit users can like their own posts, this graph will have self-loops.

Second, notice that there are two edges between b and d in the graph above: one in each direction. This is valid. What is not permissible, however, is to have more than one edge going in the same direction. For instance, the below picture does *not* depict a valid directed graph:



In fact, Definition 4 rules this situation out, since E is a "*set* of ordered pairs". Recall that sets cannot contain duplicate objects, so the same ordered pair (a, b) cannot appear twice in E. Note that (a, b) and (b, a) are *different* ordered pairs, and so it is possible for both to be in E.

The set of possible edges in a directed graph is the set of all possible ordered pairs. We write this set as $V \times V$; it is called the **Cartesian product** of V with itself. For instance, if $V = \{a, b, c, d\}$, then:

$$\begin{aligned}
V \times V = \{ & \\
& (a, a), (a, b), (a, c), (a, d), \\
& (b, a), (b, b), (b, c), (b, d), \\
& (c, a), (c, b), (c, c), (c, d), \\
& (d, a), (d, b), (d, c), (d, d) \\
\} &
\end{aligned}$$

The edge set E of any graph with these nodes can contain any (or all) of the elements listed above.

3.1.2 Undirected Graphs

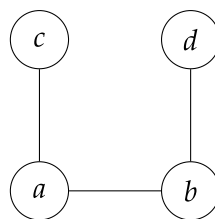
We will now formally define the notion of an **undirected graph** – i.e., the type of graph we might use to represent Facebook. An undirected graph differs from a directed graph in that its edges have no direction. While edges in a directed graph are *ordered* pairs, edges in an undirected graph will naturally be *unordered* pairs:

Definition 5. A **undirected graph** G is a pair (V, E), where V is a finite set of objects (called **nodes**) and E is a set of unordered pairs of objects in V (called **edges**).

We should pause for a moment to elaborate on what is meant by "unordered pair". For the present purposes, we will define an **unordered pair** to be a set of two elements. This is natural, since sets are unordered. That is, $\{a, b\}$ is the same mathematical object as $\{b, a\}$. One consequence of this definition – and a difference between directed and undirected graphs – is that there are no **self-loops** in undirected graphs. This is because a self loop between a node a and itself would be represented by the "set" $\{a, a\}$, but this is not a set, as sets cannot contain duplicates.

While the edges of an undirected graph are unordered pairs and therefore sets, we will not use set notation to write them. That is, we will write (a, b) instead of $\{a, b\}$ to denote an edge between nodes a and b. We should understand, however, that (a, b) is actually the set $\{a, b\}$. To be clear, in an undirected graph, (a, b) and (b, a) are *the same edge*, while they are *two different edges* if G is directed. The advantage of this uniformity in notation will become clear shortly when we make statements about both undirected and directed graphs, as it will allow us to write one sentence that holds for both types of graph.

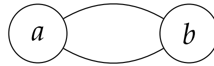
As a concrete example of an undirected graph, consider the picture below:



The node set is $V = \{a, b, c, d\}$. The edge set is:

$$E = \{(a, c), (a, b), (d, b)\}.$$

Note that we cannot have two edges between the same pair of nodes. That is, the below does *not* a depiction of a valid undirected graph:



This is because the set E is a *set* of unordered pairs. Note that we cannot find both (a, b) and (b, a) in E, since (a, b) and (b, a) are considered to be the same unordered pair, and sets cannot contain duplicates.

3.1.3 Terminology Involving Edges

The terminology used in graph theory can differ from author to author. In this course, we will stick to a standard set of definitions.

First are terms which concern the relationship between an edge and the nodes it involves. In a directed graph, the edge (u, v) is said to **leave** u and **enter** v. In an undirected graph, the edge (u, v) is said to be **incident upon** both u and v. We may sometimes talk about an edge "from" u "to" v in an undirected graph, but it should be understood that such an edge has no actual direction, and we are simply using these words to suggest a certain interpretation.

In an undirected graph, the set of **neighbors** of a node u consists of all of the nodes which share an edge with u. In the Facebook graph, for instance, the set of a user's neighbors is simply the set of their friends. In a directed graph, the set of **predecessors** of a node u is the set of nodes on the other end of edges entering u. In the Twitter graph, the predecessors of a user u are all users who follow u. On the other hand, the set of successors of a node u consists of all nodes which are on the other end of edges leaving u. In the case of Twitter, the successors of a user u are the users who are followed by u. Authors differ in defining the **neighbors** of a node in a directed graph, with some preferring to leave the term undefined altogether. In this course, we define a node's neighbors to be its successors – this will be useful when writing algorithms, as we will see in the coming sections.

The **in-degree** of a node u in a directed graph is the number of edges which enter u, and the **out-degree** of u is the number of edges which leave u. For example, in the Twitter graph, where an edge (u, v) denotes the fact that user u follows user v, the in-degree of a user's node is the number of followers they have, while the out-degree is the number of people they follow. The **degree** (or **total degree**) of a node in a directed graph is the sum of the node's in-degree and the node's out-degree. Because edges in undirected graphs have no direction, there is no concept of in-degree or out-degree. Rather, the **degree** of a node in an undirected graph is simply the number of edges which are incident upon it. The degree of a node in Facebook represents the number of friends the user has.

For instance, consider node b in the directed graph above. Its out-degree is 2, since there are two edges leaving b (the edge (b, b) leaves b). Its in-degree is 3, since there are three edges entering b (the edge (b, b) also enters b). Its total degree is therefore 5. In the undirected graph show above, the degree of b is 2.

A sometimes useful result about the degree of a graph is the following:

Claim 4. *Let $G = (V, E)$ be a graph (directed or undirected). Then*

$$\sum_{v \in V} \text{degree}(v) = 2|E|.$$

Proof. Consider removing all of the edges from the graph, then adding them back one-by-one. Adding each edge increases the degree of each of its members by one, and the total degree by two (a self-loop increases the degree of its node by two). □

3.1.4 Paths and Connectedness

The edges in a graph represent direct relationships between pairs of objects, but we are often interested in higher-order relationships. For instance, consider an undirected graph encoding all domestic flights. Each airport is a node in this graph, and there is an edge[1] between two nodes (i.e., airports) if there is a direct flight from one to the other. There are unfortunately no direct flights from San Diego to Columbus, Ohio, and so this graph has no edge between these nodes. Still, I can get to Columbus from San Diego by taking a series of one (or more) connecting flights. In other words, there is a **path** from San Diego to Columbus. For instance, one such path is to fly from San Diego to Los Angeles to Chicago to Columbus.

We will now capture the notion of a path in a formal definition. We will represent a path by the sequence of nodes visited while traveling it. But not all sequences of nodes are valid paths! For instance, if there is no direct flight from Phoenix to Columbus, then there is no path from San Diego, to Phoenix, to Columbus. We can take this into account by requiring that there is a direct flight between any consecutive pair of airports in the sequence; that is, there is an edge between every consecutive pair of nodes in the sequence of nodes. The definition we arrive at is as follows:

Definition 6. Let $G = (V, E)$ be a graph (either directed or undirected), and let $u, u' \in V$ be nodes. A **path** of length k from u to u' is a sequence $(v_0, v_1, \ldots, v_{k-1}, v_k)$ of $k + 1$ nodes in V (with $k \geq 0$) such that $v_0 = u$, $v_k = u'$, and for each $i \in \{1, \ldots, k\}$, $(v_{i-1}, v_i) \in E$.

There are several things about the above definition which should be noted. The first is that it does not stipulate that the nodes in the path must be unique. In the context of our example, this means that flying from San Diego to Denver to Houston, and then back to Denver, and then onwards to Columbus, is in fact a valid path. Instead, we will say that a path is a **simple path** if the nodes are unique.

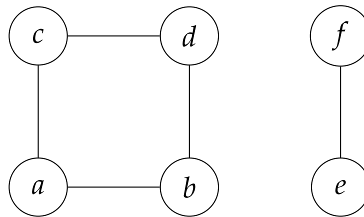
---

[1]One might argue for using a *directed* graph here, because flights have a direction. On the other hand, if there is a flight from one airport to another, there is almost certainly a flight back. If every edge in an directed graph is accompanied by an edge in the reverse direction, there is nothing gained by using a directed graph, and we can simply collapse each pair of edges into a single, undirected edge.

Second, we have defined the **length** of a path to be one less than the number of nodes contained within. In other words, the length of a path is the number of "hops" it makes. In the context of our example, the length of a path between two airports is the number of flights taken. Note that we might also say that the length of a path is the number of edges along it, but we should be careful here: an edge can appear multiple times in a path, and the length takes into account each appearance. That is, the number of unique edges in a path can be smaller than the length of the path.

Thirdly, the above definition allows for paths of length zero. In fact, any node $u \in V$ has a path of length zero which starts and ends at itself, with zero edges along the way. This is a consequence of the above definition, and in no way depends upon there being self-loops or whether the graph is directed or undirected.

For example, consider the undirected graph below.



First, recall that not every sequence of nodes is a valid path. For instance, the sequence (c, b, d, c) is not a path, since the edge (c, b) does not exist. Likewise, it is easy to see that there is no path from c to f.

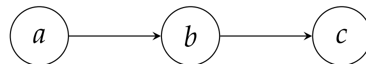
On the other hand, there is a path (of length 1) from c to d, namely, (c, d). But there is also another path (of length 4) from c to d: (c, a, b, d). In fact, since the definition above doesn't require the nodes along the path to be unique, (c, a, b, d, c, a, b, d) is *also* a valid path of length 7 between c and d. In fact, there are *infinitely many* paths from c to d under the definition given above, since each loop around gives us another (longer) path. To exclude these paths, we define a **simple path** to be a path in which each node along the path is unique. There are exactly two simple paths from c to d in the graph shown: (c, d) and (c, a, b, d).

The path (c, a, b, d, c) forms what we might call a cycle. More formally, a **simple cycle** is a path $(v_0, v_1, \ldots v_k)$ such that the length of the path is at least three and each node in the path is unique, apart from the first and last node which must be the same (i.e., $v_0 = v_k$). (c, a, b, d, c) is a simple cycle, but (c, a, b, d, c, a, b, d) is not. Definitions of a cycle vary from author to author, so it is important to verify the definition whenever you see the word used formally. Under our definition, the path of flights from San Diego to Denver to Columbus to Houston and back to San Diego is a simple cycle.

We say that a node v is **reachable** from a node u if there is a path from u to v. Intuitively, an airport is reachable from San Diego if there is a sequence of flights starting at San Diego which takes me there. In the example above, b is reachable from c, as are both a and d. But nodes e and f are *not* reachable from c. Perhaps somewhat surprisingly, we

also say that node c is reachable from itself. In fact, in any graph, every node is reachable from itself. This follows from our definition of a path, since there is always a path of length zero from a node to itself.

Note also that while the *definition* of a path is the same for directed and undirected graphs, directedness still has consequences for reachability. For instance, if v is reachable from u in an undirected graph, then u is reachable from v; we can simply follow the path from u to v back to u. But this is not true in a directed graph! Since edges have a direction, it may be that we can get to a node v starting at u, but we can't get back to u from v. For a simple example, consider the directed graph shown below:



There is a path from a to c, and so c is reachable from a. But there is no path from c to a: all of the edges are one-way streets. So a is not reachable from c.

Consider again the undirected graph shown above. In terms of reachability, it appears as though nodes fall into two distinct groups. The nodes a, b, c, and d are all mutually reachable from one another. Likewise, the nodes e and f are reachable from each other. However, no node in $\{a, b, c, d\}$ is reachable from $\{e, f\}$, and *vice versa*. If we imagine each node in the graph to be an island, and each edge to be a bridge, then we can drive between all of the islands a, b, c, and d, and we can drive between e and f, but we can't drive from any of a, b, c, and d to either of e and f. That is, the graph has two "connected components". The definition below formalizes this notion for undirected graphs.

Definition 7. Let $G = (V, E)$ be an undirected graph. A **connected component** of G is a set $C \subset V$ such that

1. any pair of nodes $u, u' \in C$ are reachable from one another; and
2. if $u \in C$ and $v \notin C$ then u and v are not reachable from one another.

From the definition, the above graph has two connected components: $\{a, b, c, d\}$ and $\{e, f\}$. Note that a consequence of the definition is that connected components must be *maximal*, in that you can't split up the nodes of a connected component and obtain another connected component. That is, $\{a, b\}$ is *not* a connected component of the above graph, because it violates the second part of the definition: the node c is not in $\{a, b\}$, but a, for instance, is reachable from c. It also follows from the definition that connected components of a graph are disjoint; a node u can only be in one connected components. Additionally, any given node u must in *some* connected component, even if that component only contains u. As a result, the connected components of the graph partition the nodes into disjoint sets.

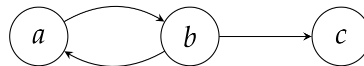
An undirected graph with exactly one connected component is said to be **connected**; otherwise, it is **disconnected**. The undirected graph above has two connected components, and is therefore disconnected.

The above definition of a connected component is for an undirected graph, but we can define connected components for directed graphs, too. In fact, directed graphs have two types: **weakly connected components** and **strongly connected components**. The weakly connected components of a digraph are obtained by replacing every directed edge with an undirected edge (removing duplicates and self-loops) and finding the connected components of the resulting undirected graph. The strongly connected components, on the other hand, are the maximal sets of mutually-reachable nodes. More formally, we say that two nodes u and u' are **mutually reachable** if u can be reached from u' and u' can be reached from u. We define strongly connected components as follows:

Definition 8. Let $G = (V, E)$ be an undirected graph. A **strongly connected component** of G is a set $C \subset V$ such that

1. any pair of nodes $u, u' \in C$ are mutually-reachable from one another; and
2. if $u \in C$ and $v \notin C$ then u and v are not mutually-reachable from one another.

For example, consider the following directed graph:



There is only one weakly connected component of the above graph: $\{a, b, c\}$. However, there are two strongly connected components: $\{a, b\}$ and $\{c\}$.

3.2 Computer Representations of Graphs

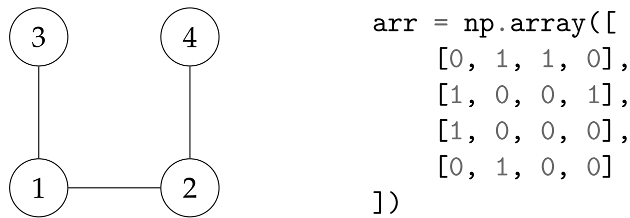
Having defined graphs mathematically, we now turn to working with graphs algorithmically. The first step towards this is representing a graph on a computer. In this section, we'll briefly cover two popular representations: **adjacency matrices** and **adjacency lists**. We'll then introduce a simple graph data structure written in Python.

3.2.1 Adjacency Matrices

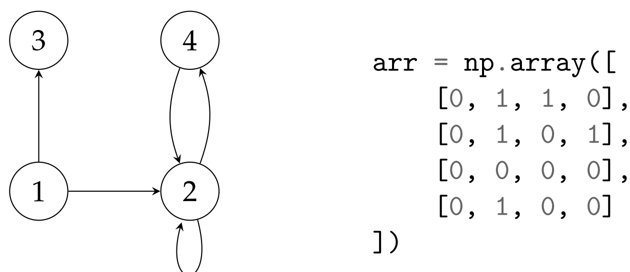
An **adjacency matrix** represents a graph $G = (V, E)$ as array of size $|V| \times |V|$. To encode a graph as an adjacency matrix, we first number its nodes from $0, 1, 2, \ldots, |V|$. We then allocate a square array, `arr`. If the edge (i, j) exists in G, we set `arr[i,j]` $= 1$; otherwise, we set `arr[i,j]` $= 0$.

This representation can be used for both undirected and directed graphs. In the undirected case, the array `arr` is symmetric; that is, `arr[i,j] == arr[j,i]`. This is because (i, j) and (j, i) are the same edge, and so we set `arr[i,j]` $=$ `arr[j,i]` $= 1$. An example of an undirected graph and its corresponding adjacency array[2] is shown below.

[2]We are using Numpy to create the array. You can also using lists of lists, but indexing an element is then done by writing `arr[i][j]` instead of `arr[i,j]`.

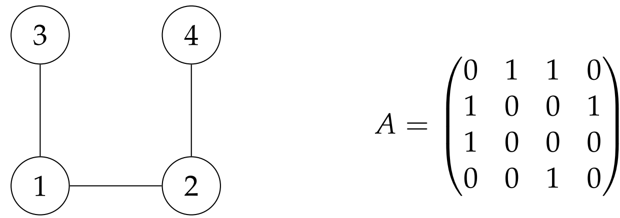


On the other hand, the adjacency matrix representing a directed graph is not necessarily symmetric, since the edge (i, j) results in `arr[i,j] == 1`, but not `arr[i,j]`. An example of a directed graph and its corresponding adjacency matrix is shown below.



Adjacency matrices have the advantage of simplicity, but they have a major disadvantage that prevents them from being used in many practical applications: they can be very wasteful of memory. An adjacency matrix stores one bit for each of the $\Theta(|V|^2)$ *possible edges*, whether the edge actually exists or not. But most real-world graphs are **sparse** in the sense that the number of edges they contain is far fewer than the number of possible edges. For instance, Facebook is estimated to have over 2 billion users, and so there are almost (2 billion)^2 possible edges. However, the average number of friends per Facebook user is only around 200, resulting in around "only" $200 \cdot$ (2 billion) edges – far fewer than the possible number of edges! Instead of using an adjacency matrix, we should rather store only the edges that exist; we will see such an approach in a moment. Nevertheless, if the graph is **dense** – that is, it has close to the greatest number of edges possible – then adjacency matrices are not so wasteful.

While we have introduced adjacency matrices as a way of representing graphs on computers, they are also useful mathematical objects. That is, no matter how we choose to represent a graph in a computer's memory, we can still think of the graph's adjacency matrix when designing algorithms. The mathematical definition of an adjacency matrix is nearly identical, apart from assuming that nodes are labeled starting with one, instead of zero. Then an adjacency matrix is a $|V| \times |V|$ matrix A whose (i, j) entry A_{ij} is one if and only if edge (i, j) exists, and otherwise zero. An example of a mathematical adjacency matrix is shown below:

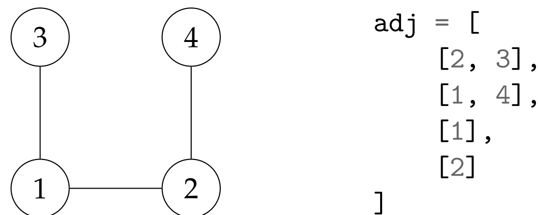


$$A = \begin{pmatrix} 0 & 1 & 1 & 0 \\ 1 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{pmatrix}$$

3.2.2 Adjacency Lists

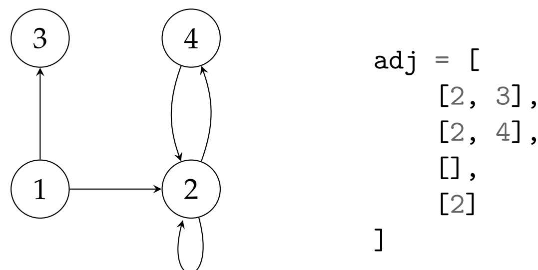
The major deficiency of the adjacency matrix representation is that it is wasteful of memory if the graph is sparse. **Adjacency lists** remedy this by storing only the edges which exist in the graph.

To encode a graph as an adjacency list, we once again start by numbering the nodes from zero to $|V| - 1$. We then create a list of $|V|$ lists, where the ith list contains all of the neighbors of node i. An example might help clarify. Consider the same undirected graph as above. The first entry of the adjacency list is a list of all of node 1's neighbors: nodes 2 and 3. The second entry is a list of node 2's neighbors: nodes 1 and 4. And so on:



```
adj = [
    [2, 3],
    [1, 4],
    [1],
    [2]
]
```

Notice that if node i appears in node j's list, then node j appears in node i's list. This is only the case if the graph is undirected. An example of a directed graph represented as an adjacency list is shown below:



```
adj = [
    [2, 3],
    [2, 4],
    [],
    [2]
]
```

3.2.3 Dictionaries of Sets

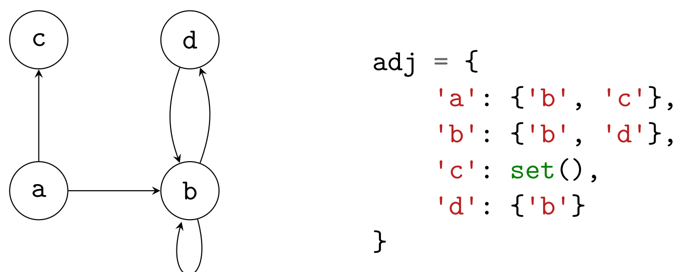
The time complexity of performing various types of queries and operations on graphs depends on the representation being used. For instance, querying whether or not an edge

(u, v) exists takes constant time if the graph is represented using an adjacency matrix, since we need only return `arr[u,j]`. With adjacency lists, on the other hand, it can take $\Theta(V)$ time in the worst case, since we must search for v in a list of all of u's neighbors, which can potentially contain all of the nodes in the graph.

Instead of using a list to store a node's neighbors, we may instead use a **hash table**. A hash table is a container data structure which supports fast queries, additions, and deletions. In particular, these operations take constant time *on average* – though they may take linear time in the worst case. If the hash table is well-implemented and suitable for the data at hand, then encountering the worst case is sufficiently rare that we can effectively assume that these operations run in constant time. Additionally, if the set of objects in the container is fixed (i.e., we aren't adding or removing objects) and known ahead of time, it is easy to design so that the *worst* case time complexity of querying is $\Theta(1)$.

Python provides two data types which implement hash tables: `set` and `dict` (i.e., a dictionary). The difference between the two is that the elements in a `dict` (i.e., the **keys**) have associated **values**, while the elements in a `set` have no associated values. Both support constant time addition, deletion, and querying on average.

We can implement a fast graph representation using a **dictionary of sets**. The approach is very similar to that of the adjacency list representation, instead we replace the outer list with a `dict` and the inner lists with `set`s. The principal advantage of using a `dict` for the outer container is that it allows us to label nodes as something other than $0, 1, \ldots, |V| - 1$; for instance, we can label them with strings. The figure below shows a graph whose labels are letters, and the corresponding `dict` of `set` representation:



Here we have written `set()` to denote the empty set, since writing `{}` results in an empty dictionary.

Using this representation, querying for the existence of an edge is a $\Theta(1)$ operation on average, just as with adjacency matrices. To be precise, to determine whether the edge $(\texttt{u}, \texttt{v})$ is in the graph, we first find `u` in the outer `dict`, which takes constant time on average. This returns the `set` of `u`'s neighbors; we can check whether `v` is in this set in constant average time. Hence in total the query takes constant time on average.

We noted above that if the elements to be stored in a hash table are fixed and known ahead of time, then we can always design the table so that it supports constant time

queries in the worst case. Most of the graph algorithms we will analyze will not modify the graph, and so we may assume that the hash tables used in its representation are well-designed[3], and that querying for a node or edge has constant time complexity in the worst case.

For now it is not necessary to understand how such a representation is implemented in Python[4]. Rather, we will simply state the average case time complexity of the frequent graph operations. The below table shows these time complexities for undirected graphs – those for directed graphs are similar:

Operation	Average Case
Adding a node	$\Theta(1)$
Removing a node with m neighbors	$\Theta(m)$
Adding an edge	$\Theta(1)$
Removing an edge	$\Theta(1)$
Querying for a node	$\Theta(1)$
Querying for an edge	$\Theta(1)$
Retrieving number of neighbors	$\Theta(1)$
Iterating over m neighbors	$\Theta(m)$

In fact, if a node has m neighbors, iterating over them can be done in $\Theta(m)$ time in the *worst* case. This will be important to remember when analyzing the algorithms to come.

In what follows, we will assume that a dictionary of sets is used to represent graphs unless stated otherwise, and we will analyze the time complexity of graph algorithms accordingly.

3.2.4 Node Attributes

It is often useful to store attributes alongside the nodes of a graph. For instance, we might wish to associate each airport in a graph of domestic flights with a rating of its restaurant choices. These attributes may be used by a graph algorithm – for example, one which finds a path from San Diego to Columbus consisting of airports with restaurants above a 4-star rating.

There are several ways of storing node attributes. In what follows, we will use hash tables to map a node's label to its attribute values. For instance, to keep track of the airport ratings, we will use a Python `dict` named `ratings`, so that if `u` is a node in the graph, `ratings[u]` returns the rating. From the previous discussion regarding hash tables, we

[3]Actual Python `dict`s and `set`s are mutable, and so they do not support constant time querying in the worst case. But since we *could* replace `dict` and `set` with data structures which do support constant time access, we'll ignore this technicality and assume that the built-ins support constant time access, too. Practically-speaking, the built-ins are plenty efficient.

[4]A full implementation is provided on the course website, if you are curious.

will assume that looking up the attribute in this way takes constant time even in the worst case[5].

Another common approach is to store the attribute directly on the node object itself, in which case it can be retrieved by writing `u.rating`. This is not an option if we are using the `dict`-of-`set`s representation above, since nodes are referenced only by their label (which is a `str`. number, or other hashable thing). As such, we have no node object on which to store attributes.

3.3 Breadth-First Search

Consider again the graph of all domestic flights. Each node in the graph is an airport, and there is an edge between two nodes (airports) if and only if there is a direct flight between them. Perhaps the simplest question one can ask about this graph is whether or not there is an edge (i.e., a direct flight) between two nodes (i.e., two cities). This is a simple **edge query**, and it is easily answered by either looking at the appropriate entry of the adjacency matrix or searching the adjacency list (or set, as it may be).

Other questions, however, are more difficult to answer. For instance: is Columbus reachable from San Diego? That is, is there a sequence of flights starting from one place and ending at the other? This cannot be answered by querying for the existence of a single edge. Instead, the answer is found by exploring (or **searching**) the graph's structure. A **graph search algorithm** is a strategy for exploring the structure of a graph in a meaningful way. In this section and the next, we will develop two search algorithms: breadth-first search and depth-first search. We will see that, while similar, the two strategies have different applications.

3.3.1 Search Strategies

In a basic sense, a graph search algorithm is just a particular strategy for visiting the nodes of a graph, one-by-one, in a way that respects the graph's *edge structure*. Of course, we *could* visit nodes in whatever order you'd choose – say, in ascending order of their label, so that we visit node 1, then node 2, then node 3, and so on, irrespective of whatever edges exist or do not exist. But visiting the nodes in this way completely disregards the graph's structure, and is not all that useful. In particular, visiting the nodes in this way tells us nothing about the connectivity of the graph. At a minimum, we expect that a search started inside a connected component only visits nodes inside that component. Even if the graph has only a single connected component, visiting nodes according to their labels (or in any other arbitrary order) is not useful, since the fact that one node is visited before another gives us no meaningful information about the relationship between these two nodes.

[5] Remember that this is not technically true, except in specific cases.

Instead, we will design two graph search strategies which *do* provide meaningful information about the relationship between nodes. When describing these strategies, it will be helpful to distinguish between *visiting* a node, which we can think of as physically standing on it, and *discovering* a node, which is like seeing it from a distance, but not visiting it just yet. We will say that nodes which have been discovered but not yet visited are **pending**. At any point during the search, a given node is in exactly one of three states: visited, pending, or undiscovered.

The two strategies in this section adopt the following rules which serve to constrain the search:

1. At every iteration, the newly-visited node is chosen among those that are pending (i.e., discovered but unvisited).

2. When a node is visited, all of its undiscovered neighbors are marked as pending before the node itself is marked as visited.

Note that the first rule is fairly general. It doesn't say, for instance, that the next node to be visited must be a *neighbor* of the last visited node, but merely that it must have been marked as pending at some point.

These two rules together ensure that nodes are visited in a meaningful order, as the following theorem shows:

Theorem 7. *Suppose a graph search obeying the above rules starts at node u, and after some number of iterations t visits node a node v. Then a path exists between u and v.*

Proof. Consider marking each node with the iteration number on which it was found, so that node u is marked with a time of 1 and node v is marked with time of t. Since node v was visited, it must have been marked as pending by a node which was visited at some iteration between 1 and $t-1$. Likewise, that node must have been marked as pending by a node visited at some iteration between 1 and $t-2$. Iteratively applying this logic, we obtain a chain of predecessor nodes, each marked with progressively earlier times, necessarily ending with the node discovered at time 1, that is, node u. □

Note that this proof works for directed and undirected graphs, and indeed the theorem is true for both. Its converse is also true[6]:

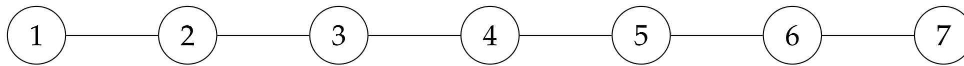
Theorem 8. *If two nodes u and v are connected by a path, then a search starting at u and satisfying the above rules will eventually reach v.*

Together, these two theorems mean that any search algorithm satisfying the above rules will never leave the connected component it starts in, and it will visit all of the nodes within. As a result, such an algorithm can be used to answer questions about the

[6] The proof is left to the reader. Hint: try proof by contradiction.

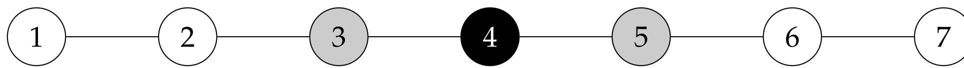
graph's connectivity, such as whether two nodes are connected by a path or if the graph is connected or disconnected.

We will now develop two different strategies which implement the above rules: breadth-first search and depth-first search. To motivate these, consider searching the graph below:

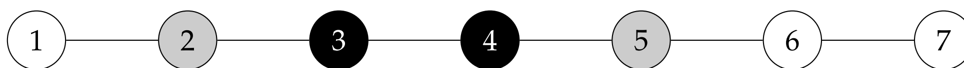


Suppose we start our search by visiting node 4. We will depict the fact that this node has been visiting by filling it as black:

The rules above state that after visiting node 4, we must mark its undiscovered neighbors as pending. In this case, that means marking nodes 3 and 5, which we do by filling them with gray:



We now have to pick the next node to visit. The first rule above says that we must choose a pending node; that is, either node 3 or node 5. The choice is arbitrary; suppose we pick node 3 to visit next, remembering that we will must visit node 5 at some point in the future. We then mark its neighbor, node 2, as pending:



We must now pick the next node to visit. It is here that we reach a fork in the road. There are two nodes marked as pending: node 2 and node 5. Which do we choose? There are two natural answers. The first is to continue exploring the most recently discovered node, which in this case is node 2. This approach is known as **depth-first search** (DFS), since it will tend to move away from the starting node, deeper into the graph. The second approach is to explore the node which was discovered first, which in this case is node 5. This is known as **breadth-first search** (BFS), since this strategy will search the graph around the starting node, iteratively expanding outwards. We will now turn to developing BFS; depth-first search will be discussed in the sequel.

3.3.2 The Algorithm

We will now implement the breadth-first search strategy. In a breadth-first search of a graph, we visit nodes in the order they were marked as pending. As a result, the natural data structure for storing the pending nodes is a first-in, first-out **queue**. As we will see, the use of the queue is what separates BFS from DFS; the latter will use a last-in, first-out **stack**.

The Python standard library's `collections` module provides a data structure called a **deque**, which stands for "double-ended queue". A deque supports constant-time appends and pops from either side. It can therefore not only be used as an efficient queue, but also as a stack. Note that while Python `list`s also support appending and popping from either side, using a `list` as a queue is costly[7].

When using a deque as a queue, we treat the front of the deque as the front of the queue, and the end of the deque as the end of the queue. To get an element from the front of the queue, we use the `deque.popleft()` method, and to push an element to the back of the queue we use the `deque.append()` method.

As mentioned above, at any point in time during the search, a given node exists in one of three states: visited, pending, or undiscovered. To keep track of this attribute, we will use a simple dictionary mapping nodes to statuses. Before beginning the search, we set each node to `'undiscovered'`. We then mark the starting node as `'pending'` and add it to the queue.

At each iteration, a breadth-first search removes a node from the queue. It visits this node, marks all of its undiscovered neighbors as pending, adds them to the queue, then marks the current node as visited and moves onto the next iteration. This process continues until there are no nodes left in the queue. The code implementing this strategy is shown in Algorithm 6.

Note that the above function will work for both directed and undirected graphs, assuming that `graph.neighbors(u)` returns the successors of `u`. This is the motivation for defining the neighbors of a node in a directed graph to be the successors, as we did in the previous section.

As written, `bfs(graph, source)` returns nothing. The above code should be considered a "foundation" upon which other graph algorithms can be built. For instance, we will soon see that, with a slight modification, the above can be used to find the *shortest* path between two nodes.

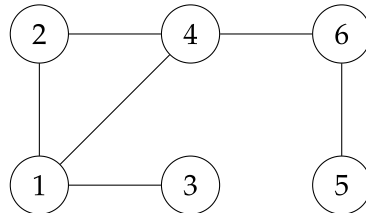
Example

Consider performing a breadth-first search on the below graph, starting with node 1:

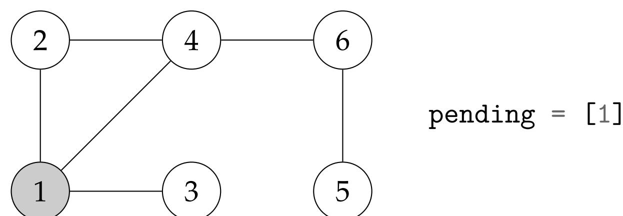
[7] See `https://wiki.python.org/moin/TimeComplexity` for a list of time complexities of various `list` operations. Popping from the front takes time linear in the number of elements.

Algorithm 6 Breadth-First Search

```
from collections import deque

def bfs(graph, source):
    """Start a BFS at `source`."""
    status = {node: 'undiscovered' for node in graph.nodes}

    status[source] = 'pending'
    pending = deque([source]) # will use as a queue

    # while there are still pending nodes
    while pending:
        u = pending.popleft() # pop from left (front of queue)
        for v in graph.neighbors(u):
            # explore edge (u,v)
            if status[v] == 'undiscovered':
                status[v] = 'pending'
                # append to right
                pending.append(v) # append to right (back of queue)
        status[u] = 'visited'
```



We begin by marking each node as `'undiscovered'`, which we will represent by filling a coloring it white. Before the first iteration of the loop, we add node 1 to the queue and mark it as `'pending'`. We will color pending nodes gray. Therefore, going into the first iteration, we have:

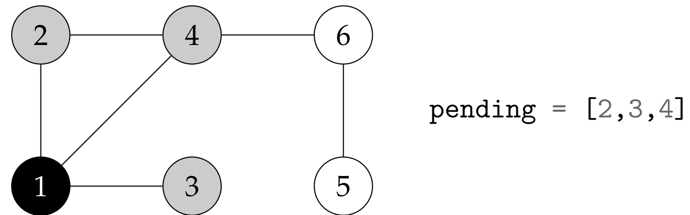


During the first iteration, we pop node 1 from the queue and visit. We mark its neighbors

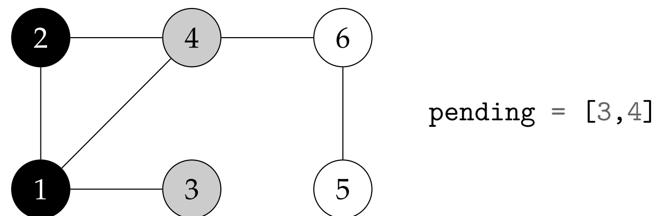
(nodes 2, 3, and 4) as `'pending'` and add them to the queue.

> We have not yet said anything about the order in which `graph.neighbors(u)` returns the neighbors of node `u`. In fact, this method can return the neighbors in any order without affecting the correctness of the search. Nevertheless, the order in which neighbors are returned directly affects the order in which they are added to the queue, and therefore the order in which they are visited. For uniformity, we'll adopt the convention that `graph.neighbors()` generates nodes in ascending order of label. Hence in this case, node 2 is added to the queue first, then node 3, followed by node 4.

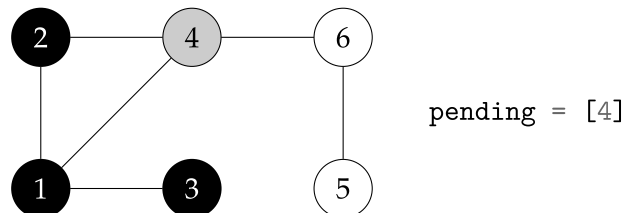
Once all neighbors have been processed, we finish by marking the popped node as `'visited'`. After the first iteration, we have:



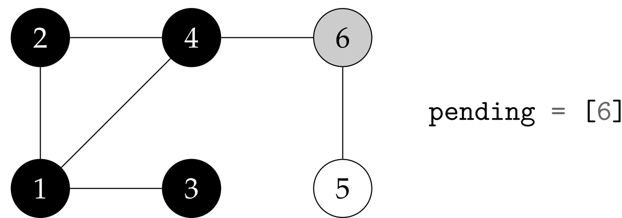
On the next iteration, we will visit node 2, since it is at the front of the queue. All of node 2's neighbors have been discovered, so we append nothing to the queue. After the iteration, the situation is as depicted below:



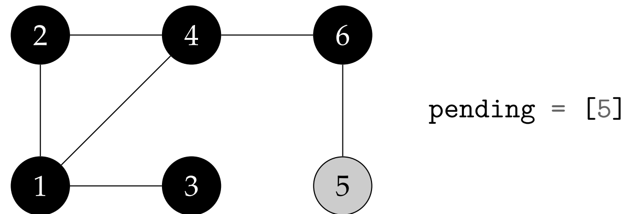
We visit node 3 on the next iteration:



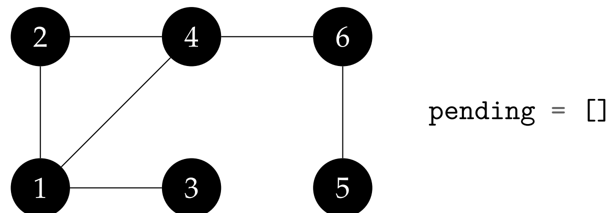
And then node 4:



Then node 6:



And finally node 5:



At this point, `pending` is empty, and so the search terminates.

"Full BFS"

As written, `bfs` will visit only those nodes which are reachable from the source; this is typically the desired behavior. Sometimes, however, we with to continue on and visit all of the graph's nodes. We can perform a search of the entire graph by looping through each of the graph's nodes and calling `bfs` every time we discover a node which has not been discovered by a previous BFS. For this to be efficient, we must have a way of passing the dictionary of node statuses into `bfs` so that it does not create a new one every time it is called. We can do so by adding `status` as a keyword argument to `bfs`. If this argument is not provided, `bfs` will populate the dictionary as before. This change is implemented in Algorithm 7.

With this change, we can run a "full" breadth-first search using the following code:

```
def full_bfs(graph):
    status = {node: 'undiscovered' for node in graph.nodes}
    for node in graph.nodes:
        if status[node] == 'undiscovered'
            bfs(graph, node, status)
```

Algorithm 7 Breadth-First Search with `status` Argument

```
from collections import deque

def bfs(graph, source, status=None):
    """Start a BFS at `source`."""
    if status is None:
        status = {node: 'undiscovered' for node in graph.nodes}

    status[source] = 'pending'
    pending = deque([source]) # will use as a queue

    # while there are still pending nodes
    while pending:
        u = pending.popleft() # pop from left (front of queue)
        for v in graph.neighbors(u):
            # explore edge (u,v)
            if status[v] == 'undiscovered':
                status[v] = 'pending'
                pending.append(v) # append to right (back of queue)
        status[u] = 'visited'
```

3.3.3 Properties of BFS

There are several key properties of breadth-first search which will help in understanding how it works and in assessing its efficiency. Both of the properties below will pertain to the "full" version of BFS, i.e., the version implemented by `full_bfs` above, in which all of the graph are explored. Similar statements can be made about `bfs`.

Each node is added to the queue exactly once.

To see that a node cannot be added more than once, note that only `'undiscovered'` nodes are added to the queue. But once a node is added to the queue, it is immediately marked `'pending'` – and it can never become `'undiscovered'` again. To see that a node must be added at least once, observe that the only way a node can become anything other than `'undiscovered'` is to append it to the queue. The `for`-loop in `full_bfs` iterates over all of the nodes, and calls `bfs` on any which are `'undiscovered'`; `bfs` immediately adds the node to the queue. Hence every node is added to the queue at least once. Together, these imply that each node is added exactly once.

Each edge is explored exactly once if the graph is directed, and twice if it is undirected.

During each iteration of the `for`-loop in `bfs` a single edge in the graph is "explored"; in particular, the edge from the popped node `u` to its neighbor, `v`. From the previous property, node `u` is popped from the queue exactly once. As a result, the directed edge $(\texttt{u}, \texttt{v})$ is explored exactly once. On the other hand, the undirected edge $(\texttt{u}, \texttt{v})$ is the same edge as $(\texttt{v}, \texttt{u})$, and so it is explored not only when `u` is popped, but also when `v` is popped, for a total of twice.

3.3.4 Time Complexity of BFS

BFS is an iterative algorithm with a `for`-loop nested inside a `while`-loop. Intuitively, the algorithm's complexity will depend upon how many times the inner `while`-loop runs in total. Our usual approach to calculating this is to a formula for the number of times that the inner loop runs as a function of the outer iteration number. In this case, however, this is not feasible: the number of iterations of the `for`-loop in `bfs` depends on how many neighbors the node `u` has, and has no direct relationship to the outer loop iteration number.

Instead, we will compute the time complexity of the BFS algorithm using an approach called **aggregate analysis**. While we do not know how many times the inner loop runs on a particular iteration of the outer loop, it turns out that we can easily determine exactly how many times it runs in **aggregate** over the course of the entire algorithm – and this is all we need to know.

For simplicity, we will analyze the "full" version of BFS, `full_bfs`, which visits the whole graph. Intuitively, this will upper-bound the run time of `bfs`, which may only visit part of the graph. Before the search begins, `full_bfs` initializes the status of each node in $\Theta(V)$ time. It then loops over every node in the graph, checking whether it is `'undiscovered'` or not. These two lines are executed $\Theta(V)$ times, taking $\Theta(1)$ time per execution for a total of $\Theta(V)$ time.

Switching our attention to `bfs`, we focus on the number of iterations of the `for`-loop over the course of a single call to `full_bfs`. In the previous section, we observed that – during the execution of `full_bfs(graph)` – each edge is explored exactly once if the graph is directed, and exactly twice if the graph is undirected. Each iteration of the `for`-loop in `bfs_visit` explores a single edge. Therefore, the inner loop iterates exactly $|E|$ times in total if the graph is directed, and exactly $2|E|$ times in total if the graph is undirected. On each iteration, the loop spends constant time checking the status of nodes and appending them to the queue, if need be. As a result, the inner `for`-loop takes $\Theta(E)$ time in total over the course of the entire search.

Note that we haven't said how many times `bfs_visit` is called! This depends on the input graph, the source, and the order in which neighboring nodes are added to the queue. Rather, we've said that in total, over *all* calls to `bfs_visit`, a total of $\Theta(|E|)$ time is

spent in the `for`-loop.

Since initializing the node statuses takes $\Theta(V)$ time, and exploring each edge takes $\Theta(E)$ time, the time complexity of the algorithm as a whole is $\Theta(V + E)$. If we know that one of $|V|$ or $|E|$ is larger than the other, we can simplify this expression. For instance, if the graph is fully connected (meaning, it has every possible edge), then $|E| = \Theta(|V|^2)$, and the run-time simplifies to $\Theta(|V|^2)$. On the other hand, if the graph has no edges, the algorithm must still perform initialization (as well as add each node to the queue and pop each node off), taking time $\Theta(|V|)$.

Lastly, since the time complexity of the "full" BFS is an upper-bound on the time complexity of `bfs`, we have that the time complexity of the latter is $O(V + E)$.

3.3.5 Shortest Paths

Breadth-first search gets its name from the fact that it searches broadly around the source before moving deeper into the graph. We now make this key fact more precise. In what follows, we will say that a node v is **distance** k **away from** u if the length of the *shortest* path between u and v is k. The following important claim will not be proven, but is intuitively true:

Claim 5. *Suppose a breadth-first search is started from a node u, and let k be any natural number. Then all nodes which are distance k from u are discovered and added to the queue before any node whose distance is larger than k is added to the queue.*

This claim has a very important consequence:

Theorem 9. *Suppose that there is a path from node u to node v ($u \neq v$). Furthermore, let d be the length of the shortest path from u to v. Then during a breadth-first search started at node u, node v is discovered while visiting a node which is distance $d - 1$ from u.*

Proof. Suppose that node v is discovered during the BFS while visiting node v_0. We want to show that v_0 is necessarily at distance $d - 1$ from node u.

When node v is discovered it is immediately added to the queue. The claim above therefore implies that node v_0 is at most distance d from u.

Node v_0 cannot be less than $d - 1$ away from u, as then the path constructed by going from u to v_0 and then immediately to v would have length less than d, contradicting the fact that the shortest path between u and v has length d.

Likewise, v_0 cannot be distance d away from u, since then node v would have been discovered prior to visiting v_0. To see this, consider any shortest path from u to v, and let v_0^* be the node right before v on this path. Note that v_0^* must be distance $d - 1$ from node u. If it were the case that v_0 is distance d away from u, then v_0^* would be added to the queue before v_0 as a consequence of the above claim. Hence if v were still undiscovered when v_0^* was visited, it would have been discovered then. This contradicts the fact that v is discovered while visiting node v_0, and so v_0 cannot be distance d from u.

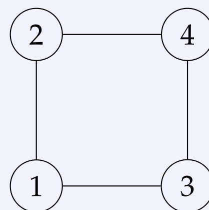
The above three paragraphs imply that node v_0 cannot be less than distance $d - 1$ away from u, and it cannot be more than distance $d - 1$ away. Therefore it must be distance d away from u. □

This theorem motivates us to modify `bfs` so that it returns a shortest path[8] between `source` and every node in the graph, as well as the shortest path distance between `source` and every other node. While the latter is easily obtained from the former by computing the length of the path, returning the distance can be done with little extra overhead and is oftentimes more convenient.

Suppose that during a BFS started at node `u`, we discover a node `v` while scanning the neighbors of a node `v_0`. We will say that `v_0` is the **predecessor** of `v`. Of course, unless `v_0` is the source of the search, it will have its own predecessor. Following this chain of predecessors back to the source node produces a path from the source node `u` to node `v`; moreover, this path will have the fewest number of edges out of any path from `source` to `v`. We will keep track of every node's predecessor with an attribute dictionary, initialized so that the predecessor of every node is initially `None`.

Recall that the order in which we process the neighbors of `v_0` is arbitrary, and we have chosen the convention that nodes are processed in increasing order by label for reasons of simplicity and uniformity of presentation. But two different BFSs which adopt different conventions for the order in which neighbors are processed will in general produce different predecessors. This does not break the algorithm, but you should be aware of this fact.

For instance, suppose we run a BFS on the graph below, starting at node 1:



You can check that if the BFS adopts the convention of processing the neighbors in increasing order by label, then the predecessor of node 4 will be node 2. But if the BFS adopts the convention of processing neighbors in *decreasing* order by label (which is an equally-valid convention), the predecessor of node 4 will be node 3.

We will also keep track of each node's distance from the `source` with an attribute dictionary. Suppose once more that we discover a node `v` while scanning the neighbors

[8] Note the careful wording here: the algorithm will return *a* shortest path. We do not say that it will return *the* shortest path, because there may be more than one.

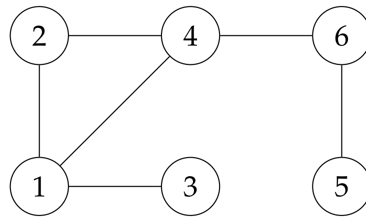
Algorithm 8 Breadth-First Search with `status` Argument

```
from collections import deque

def bfs_shortest_paths(graph, source):
    """Start a BFS at `source`."""
    status = {node: 'undiscovered' for node in graph.nodes}
    distance = {node: float('inf') for node in graph.nodes}
    predecessor = {node: None for node in graph.nodes}

    status[source] = 'pending'
    distance[source] = 0
    pending = deque([source]) # use as a queue

    # while there are still pending nodes
    while pending:
        u = pending.popleft() # pop from left (front of queue)
        for v in graph.neighbors(u):
            # explore edge (u,v)
            if status[v] == 'undiscovered':
                status[v] = 'pending'
                distance[v] = distance[u] + 1
                predecessor[v] = u
                pending.append(v) # append to right (back of queue)
        status[u] = 'visited'

    return predecessor, distance
```

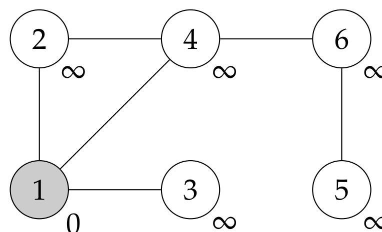
of a node `v_0`. If d is the distance from `source` to `v_0`, then the distance from `source` to `v` is simply $d + 1$. Before searching, we set each distance to ∞; this represents the fact that we have yet to discover a path of any length between the source and the node. We set the distance from the source to itself to zero, since there is a path of length zero between any node and itself.

Algorithm 8 adds these attribute dictionaries to `bfs` and updates them accordingly. Note that if a node is not visited during the search – if it is in another connected component, for instance – then its distance will remain at ∞ and its predecessor will be `None`.

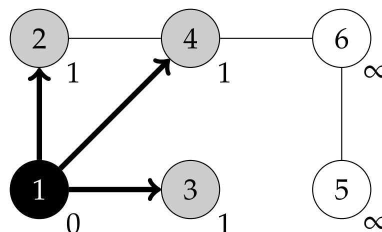
For example, consider searching the graph below, starting at node 1:



Before the loop begins, we set `distances[u]` to ∞ for every node `u` in the graph. We then set `distances[1]` to zero, since it is the source. We will depict the value of `distance[u]` for each node `u` by writing it below the node. For instance, after the "zeroth" iteration the situation is as follows:



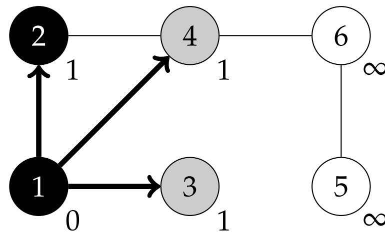
On the first iteration, nodes 2, 3, and 4 are discovered. These nodes are therefore assigned a distance of one. The predecessor of each of these nodes is set to node 1; we depict this by drawing a thick arrow from node 1 to each of its successors in the search. Hence, after the first iteration we have:



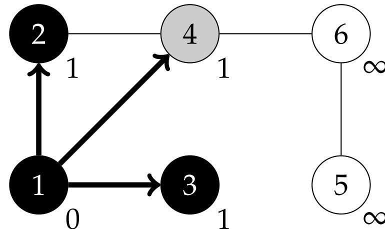
The dictionary of predecessors will be:

```
predecessor = {
    1: None,
    2: 1,
    3: 1,
    4: 1,
    5: None,
    6: None
}
```

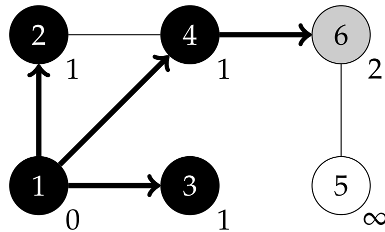
On the second iteration, node 2 is visited, and no nodes are discovered:



Likewise, no new nodes are discovered when we visit node 3 on the third iteration:



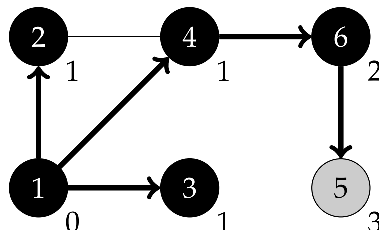
On the fourth iteration, we visit node 4. It is then that we discover node 6; it is assigned a distance of two and its predecessor is updated to be node 4:



The dictionary of predecessors will be:

```
predecessor = {
    1: None,
    2: 1,
    3: 1,
    4: 1,
    5: None,
    6: 4
}
```

On the next iteration, node 6 is visited and node 5 is discovered. It is given a distance of three, and its predecessor is updated to be node 5:



All node distances have been assigned at this point. You can verify that these are indeed the number of edges in the shortest path from the source node. Finally, the dictionary of predecessors will read:

```
predecessor = {
    1: None,
    2: 1,
    3: 1,
    4: 1,
    5: 6,
    6: 4
}
```

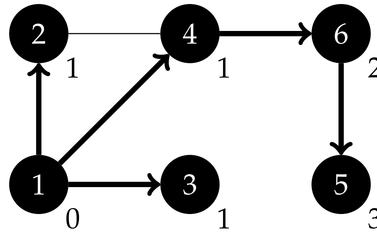
We can read off the shortest path between the source and any node in the graph by recursively looking up predecessors. For instance, consider finding the shortest path between node 1 and node 5. The node immediately before node 5 in this path must be node 6, since node 6 is node 5's predecessor. Likewise, node 4 must come immediately before node 6, since node 4 is node 6's predecessor, and so on, such that the shortest path is $(1,4,6,5)$.

The following function prints this list of predecessors in order, starting at the source:

```
def print_shortest_path(predecessor, source, dest):
    if source == dest:
        print(source)
    elif predecessor[dest] is None:
        print('There is no path.')
    else:
        print_shortest_path(predecessor, source, predecessor[dest])
        print(dest)
```

3.3.6 The Breadth-First Search Tree

We have seen that a key application of breadth-first search is in finding the shortest path between a source node and all other nodes in the graph. These shortest paths are encoded in the `predecessor` dictionary which maps each node in the graph to its predecessor in the search. We graphically represented the fact that a node v_0 is the predecessor of a node v by drawing an arrow from u to v, as in the figure below:



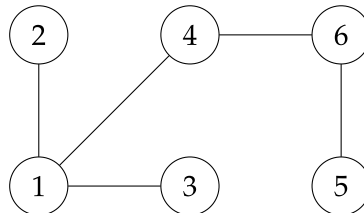
Note that these arrows mark a distinguished subset of the edges of the graph – these are the edges that the breadth-first traveled along. As a result, they effectively encode the result of the search. For instance, to find the shortest path from node 1 to node 5, we follow the arrows backwards from node 5 until we reach the source.

It is often convenient to work with a graph which is constructed from these distinguished edges alone. In particular, suppose we perform a BFS on a graph $G = (V, E)$ starting at node s, resulting in a `predecessor` dictionary. We define the **breadth-first search subgraph** G_{BFS} to be the undirected graph whose nodes are those which were reached by the BFS, and in which edge (u, v) exists if and only if u is the predecessor of v; i.e., `predecessor[v] == u`. More formally, $G_{\text{BFS}} = (V_{\text{BFS}}, E_{\text{BFS}})$, where:

$$V_{\text{BFS}} = \{v \in V : \texttt{predecessor[v] is not None}\} \cup \{s\},$$
$$E_{\text{BFS}} = \{(u, v) \in E : \texttt{predecessor[v] == u}\}.$$

This is *set-builder notation*, and it is very commonly used wherever discrete math is applicable. The first line should be read as "V_{BFS} is the set consisting of all nodes v in V such that the predecessor of v is not `None`, unioned with the singleton set consisting of just the source node, s. The reason for the union is that the source node s has no predecessor and would otherwise be omitted, but it is visited by the search and should therefore be included in the subgraph.

For example, the breadth-first search subgraph for the above graph will consist of all nodes and every edge, except for the edge $(2, 4)$. That is, it will be the graph below:



The breadth-first search subgraph is truly a "subgraph" of G in this sense that its nodes form a subset of G's nodes and its edges form a subset of E. Moreover, note that the graph

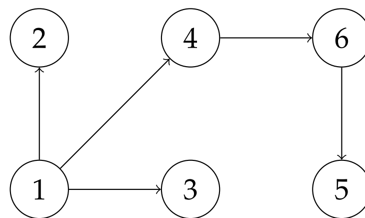
is connected, and that the number of edges in the graph is precisely one fewer than the number of nodes. We call such a graph a **tree**:

Definition 9. An undirected graph $T = (V, E)$ is a **tree** if 1) it is connected; and 2) $|E| = |V| - 1$.

Because of this, we say that an element of E_{BFS} is a **breadth-first search tree edge**.

An important property of a tree is that there is exactly one path between any given pair of nodes.

One node in the breadth-first search tree is special: the source node, s. We call a tree with a distinguished node s a **rooted** tree, and s is called the **root**. A rooted tree is technically an undirected graph, but we can unambiguously assign a direction to each edge by pointing it away from the root. For example, the natural direction of each edge in the breadth-first tree rooted at node 1 is depicted below:



The breadth-first tree is an important object because it effectively encodes the result of a breadth-first search. For instance, we can recover a shortest path from the source of the search to any other node in the graph by following the edges in the BFS tree. Note, however, that the edges in the breadth-first search tree encode the predecessor of each node, and this depends on the convention adopted by the BFS as to the order in which neighbors are processed and added to the pending queue. As a result, the breadth-first search tree is in general *not* uniquely determined.

3.4 Depth-First Search

Suppose you are an incoming DSC student at UCSD, and you are very excited to take CSE 151: Machine Learning. Unfortunately, you cannot enroll in CSE 151 right away, since it has prerequisites of DSC 40B and DSC 80. Of course, these courses have their own prerequisites, which in turn have prerequisites, and so on. Undeterred, you find a list of the required courses and their prerequisites online. You sit down with a pencil and paper and try to work out which classes need to be taken and in what order so that you can meet all of the prerequisites to take CSE 151.

Even without yet knowing any graph theory, you might draw something similar to the directed graph shown in Figure 3.1. Here, each node is a course, and there is an edge

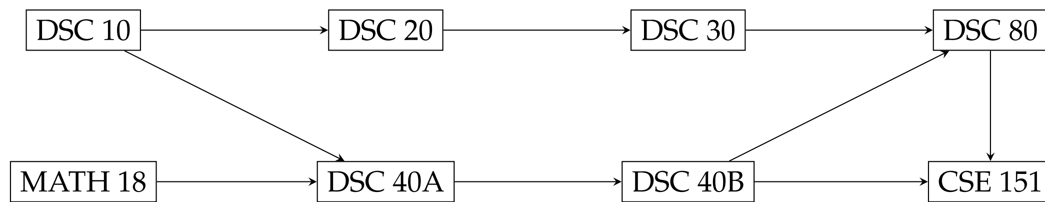


Figure 3.1: DSC Prerequisites

from course u to course v if course u is a direct prerequisite of course v; for instance, there is an edge from "DSC 10" to both "DSC 20" and "DSC 40A", since DSC 10 is a prerequisite for both.

By drawing the course prerequisites as a graph, we can state the problem more formally using the language of graph theory. More precisely, we aim to find an order in which to take the courses (i.e., an ordering of the nodes) such that if there is an edge from u to v in the graph above, then u is taken some time before v (though not necessarily immediately before). Such an ordering of the nodes of a directed graph is called a **topological sort**. One[9] topological sort is the following:

1. MATH 18
2. DSC 10
3. DSC 40A
4. DSC 40B
5. DSC 20
6. DSC 30
7. DSC 80
8. CSE 151

Knowing what you now know about graphs, you might guess that such a topological sort can be produced using a graph search algorithm, and you'd be right. But it turns out that breadth-first search is not the best tool for the job. Instead, we will use a **depth-first search**. Recall that on each iteration of a breath-first search, the node which has been pending the longest is visited. In a depth-first search, on the other hand, the node which has been pending for the least amount of time – i.e., the most recently discovered node – is visited. As we will see, this has the effect of searching deeper into the graph on each iteration, and will help us find a topological sort of the graph's nodes.

[9]There are generally multiple topological sorts of the graph's nodes. For instance, the order (DSC 10, DSC 20, DSC 30, MATH 18, DSC 40A, DSC 40B, DSC 80, CSE 151) is equally valid.

Algorithm 9 Depth-First Search with `status` Argument

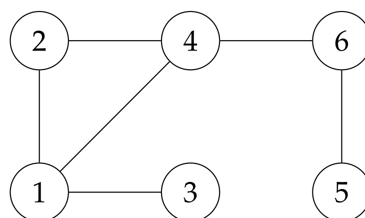
```
def dfs(graph, u, status=None):
    """Start a DFS at `u`."""
    # initialize status if it was not passed
    if status is None:
        status = {node: 'undiscovered' for node in graph.nodes}

    status[u] = 'pending'
    for v in graph.neighbors(u): # explore edge (u, v)
        if status[v] == 'undiscovered':
            dfs(graph, v, status)
    status[u] = 'visited'
```

3.4.1 The Algorithm

Depth-first search is most naturally and elegantly stated recursively, as is shown in Algorithm 9. Like BFS, depth-first search iterates through a node's neighbors in order to discover them. When an undiscovered neighbor is found, `dfs` recursively calls itself to explore the corresponding edge. This recursive call drills deeper and deeper into the graph until it reaches a node with no undiscovered neighbors to visit. It then "unrolls" by "backtracking" until it returns to a node which has undiscovered neighbors remaining. The process of drilling deeper into the graph and unrolling is repeated until, eventually, the search visits all of the source's neighbors and unrolls back to the source.

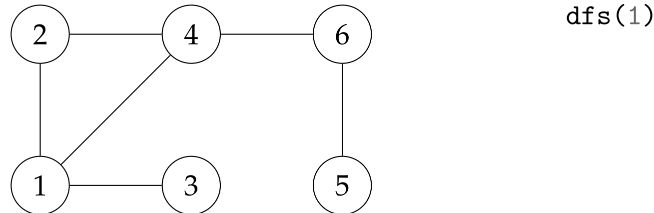
Example

Consider running a depth-first search on the graph below, starting at node 1.

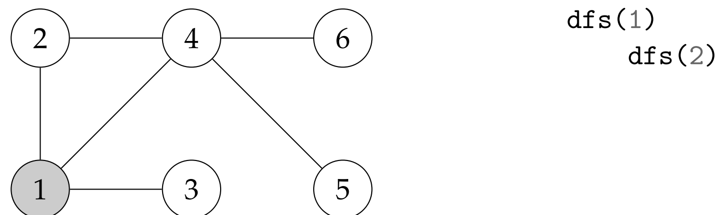


As before, we will depict nodes with a status of `'visited'` by filling them with black, nodes which are `'pending'` will be marked gray, and `'undiscovered'` nodes will be blank. We will show the current call stack to the right of the graph, with the level of indentation denoting the nesting of calls. For conciseness, we will write `dfs(u)` as shorthand for the call `dfs(graph, u)`.

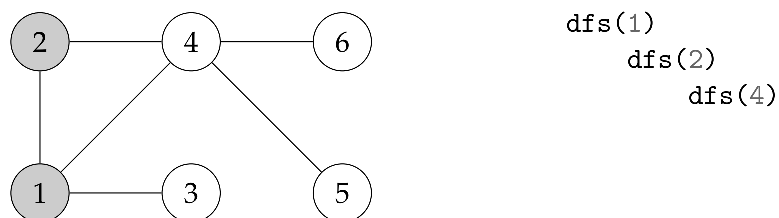
At the time of the first call to `dfs`, each node status is `'undiscovered'`, and hence each node is blank:[10]



Next, the first call enters the `for`-loop iterating over node 1's neighbors. Using the convention that `.neighbors()` produces nodes in ascending order by label, a recursive call is made to `dfs(2)`. When this call is made, the parent call to `dfs(1)` is suspended in the middle of the `for`-loop, to be resumed later. At the time of the call to `dfs(2)`, node 1 has been marked as pending:

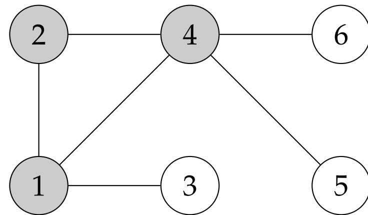


While executing `dfs(2)`, node 2 is marked as pending. Next, node 4 is discovered and a recursive call to `dfs(4)` is made. At this time, the status of the search is as shown:



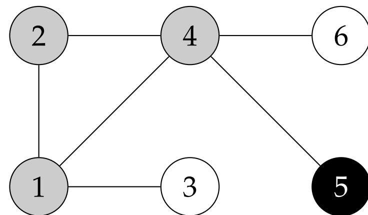
During the call to `dfs(4)`, node 5 is discovered and a recursive call to `dfs(5)` is made. At the time of this call, the situation is as depicted:

[10]Technically, each node's status is set immediately after the `dfs` call, but can imagine that the already-initialized `status` has been passed into the function.



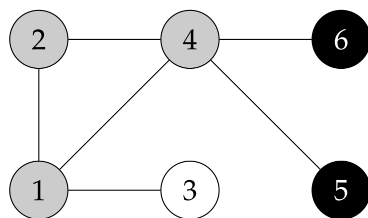

```
dfs(1)
    dfs(2)
        dfs(4)
            dfs(5)
```

During the execution of `dfs(5)`, node 5 is marked as pending. Next, its neighbors are iterated through, but none of them are `'undiscovered'`. Hence no recursive calls are made. The `for`-loop ends, node 5 is marked as `'visited'`, and the call to `dfs(5)` returns. The search "backtracks" to `dfs(4)`, finding itself in the following situation:




```
dfs(1)
    dfs(2)
        dfs(4)
```

The call to `dfs(4)` was suspended in the first iteration of the `for`-loop in order to make a recursive call to `dfs(5)`. When the search "backtracks" to `dfs(4)`, the call resumes exactly where it left off. It proceeds to the next iteration of the `for`-loop and discovers node 6. A recursive call to `dfs(6)` is made. As with the call to `dfs(5)`, this recursive call finds no undiscovered nodes and terminates after marking node 6 as `'visited'`. Immediately after the call to `dfs(6)` terminates, the status of the search is as shown:

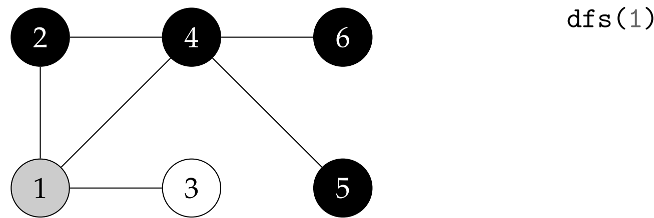



```
dfs(1)
    dfs(2)
        dfs(4)
```

The search now finds itself back in the call to `dfs(4)`. At this point, there are no neighbors of node 4 left to discover. Node 4 is marked as `'visited'`, and the search "backtracks" to node 2. Again, there are no new nodes to discover, and so the search "backtracks" all the way to the source call, `dfs(1)`. At the moment this call is resumed, the situation is as shown below:

Algorithm 10 "Full" DFS

```
def full_dfs(graph):
    status = {node: 'undiscovered' for node in graph.nodes}
    for node in graph.nodes:
        if status[node] == 'undiscovered'
            dfs(graph, node, status)
```



The call to `dfs(1)` was suspended during the first iteration of the `for`-loop. Continuing on to the next iteration, node 3 is discovered and a recursive call is made to `dfs(3)`. This call terminates after marking node 3 as visited, returning to `dfs(1)`. The root call then has no more work left to do, and it too terminates after marking node 1 as `'visited'`.

"Full DFS"

Like `bfs` from the previous section, a call to `dfs` will only visit those nodes which are reachable from the source. As with BFS, we can implement a "full" version of DFS which visits all of the nodes of the graph. In fact, the code for `full_dfs` is nearly identical to that of `full_bfs`; we simply change `bfs` for `dfs`. The result is shown in Algorithm 10.

3.4.2 Time Complexity

As with breadth-first search, we will find the time complexity of the "full" DFS implemented in `full_dfs`. This will obviously upper-bound the time complexity of `dfs`.

Our implementation of DFS is recursive, and so we might think to write and solve a recurrence relation in order to find its time complexity. However, writing a recurrence relation for `dfs` is not easy, since we do not know how many recursive calls will be made from each parent call. Instead, we will use aggregate analysis as we did with `bfs` in the previous section.

First, note that the initialization of `status` in `full_dfs` will take $\Theta(V)$ time, as will iterating over the nodes of the graph and checking whether each is undiscovered. What remains is to determine the time taken in aggregate by all calls to `dfs`. Intuitively, this

will depend on the total number of executions of the `for`-loop in Line 22 of `dfs`. As the comment in Line 22 suggests, each iteration of this `for`-loop explores an edge in the graph. The question thus becomes: how many time is each edge explored?

To calculate this, first note that `dfs` is called at most once for each node, since it is only called on nodes which are marked as `'undiscovered'`, and it immediately marks such nodes as `'pending'`. In fact, `dfs` is called *at least* once for every node, due to the `for`-loop in `full_dfs`. As a result, `dfs` is called exactly once for each node in the graph.

Now consider an arbitrary edge, (u, v). If the graph is directed, this edge is explored exactly once: when `dfs` is called on node u. On the other hand, if the graph is undirected the edge is explored exactly twice: when `dfs` is called on node u, and again when it is called on node v. As a result, each edge is explored $\Theta(1)$ times, and the total number of executions of Line 22 is $\Theta(E)$. In sum, the total time taken by `full_dfs` is therefore $\Theta(V + E)$ – the same as breadth-first search.

Since the time complexity of `full_dfs` upper-bounds that of `dfs`, the time complexity of the latter is $O(V + E)$.

3.4.3 Start and Finish Times

As with breadth-first search, the order in which nodes are visited in a depth-first search reveals meaningful information about the structure of the graph. But as we saw above, depth-first search does not finish visiting one node before visiting the next; instead, the visit to the first node is suspended while the neighboring nodes are visited. It is only after all neighbors have been visited that the search finishes visiting the first node. It turns out that the order in which the search finishes visiting nodes is also meaningful. As such, we will keep track of two "times" for each node: the time at which we begin visiting the node – i.e., the **start time** – and the time at which we finish visiting the node – i.e., the **finish time**.

The code for the modified depth-first search is shown in Algorithm 11. We use dictionaries `start` and `finish` to keep track of the start and finish times of each node. In addition, we use a `clock` variable keep track of the "time"; in this case, `clock` is an integer variable that is incremented whenever we start visiting a node and whenever we finish. These all must be passed between calls to `dfs_times`. In principle, this can be done by making each an argument to the function. Instead, we will use Python's `dataclasses` module, added in Python 3.7, to create a `Times` dataclass which has three attributes: `clock`, `start`, and `finish`. This allows us to pass one object (an instance of `Times`) instead of three, making the code somewhat cleaner.

In addition to computing the start and finish times for each node, we will also keep track of each node's predecessor, as was done in `bfs_shortest_paths` above.

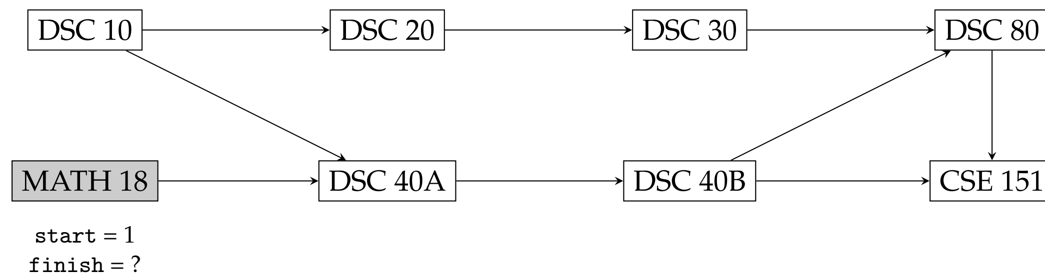
Algorithm 11 Depth-First Search with Start and Finish Times

```
from dataclasses import dataclass

@dataclass
class Times:
    clock: int
    start: dict
    finish: dict

def full_dfs_times(graph):
    status = {node: 'undiscovered' for node in graph.nodes}
    predecessor = {node: None for node in graph.nodes}
    times = Times(clock=0, start={}, finish={})
    for u in graph.nodes:
        if status[u] == 'undiscovered':
            dfs_times(graph, u, status, times)
    return times, predecessor

def dfs_times(graph, u, status, predecessor, times):
    times.clock += 1
    times.start[u] = times.clock
    status[u] = 'pending'
    for v in graph.neighbors(u): # explore edge (u, v)
        if status[v] == 'undiscovered':
            predecessor[v] = u
            dfs_times(graph, v, status, times)
    status[u] = 'visited'
    times.clock += 1
    times.finish[u] = times.clock
```

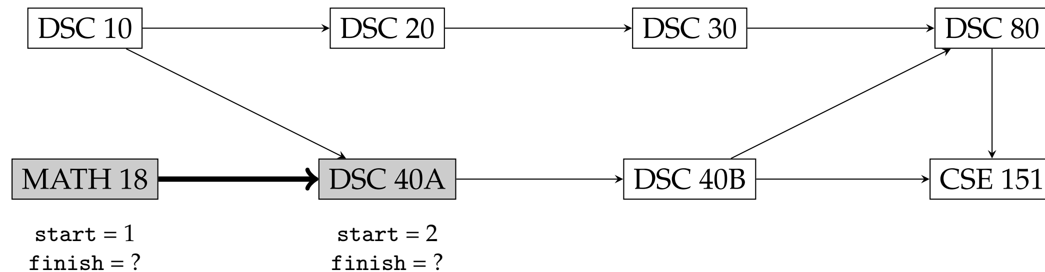
Example

Consider once again the directed graph of DSC prerequisites. We will perform a depth-first search on this graph, recording start and finish times along the way.

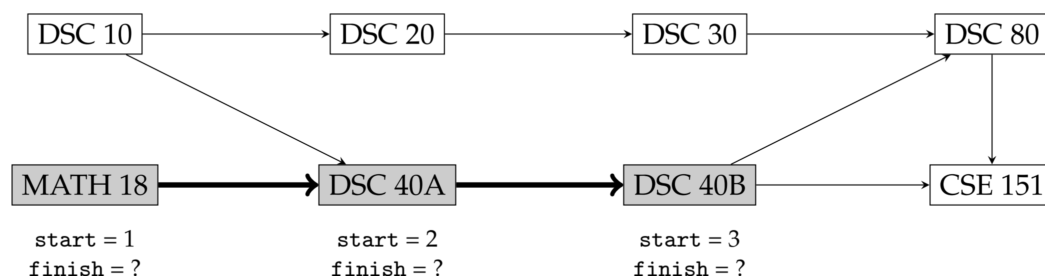
When we call `full_dfs_times`, it starts the depth-first search by iterating through `graph.nodes`. The order in which these nodes are produced is arbitrary, and so the search may start at any node. Suppose for this example that the first node produced is “MATH 18”. Then this node is visited and assigned a start time of 1:



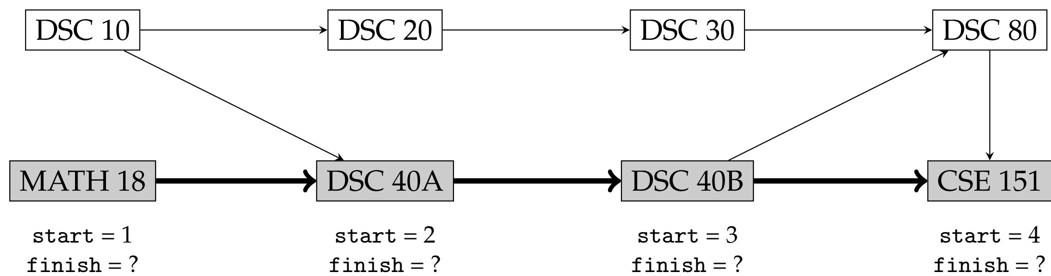
While visiting “MATH 18”, we discover “DSC 40A” and immediately perform a recursive call to visit it. When we do, the clock is incremented to 2; this becomes “DSC 40A”’s start time. Notice, too, that “MATH 18” is the predecessor of “DSC 40A” in the search. We denote this by the bold arrow below:



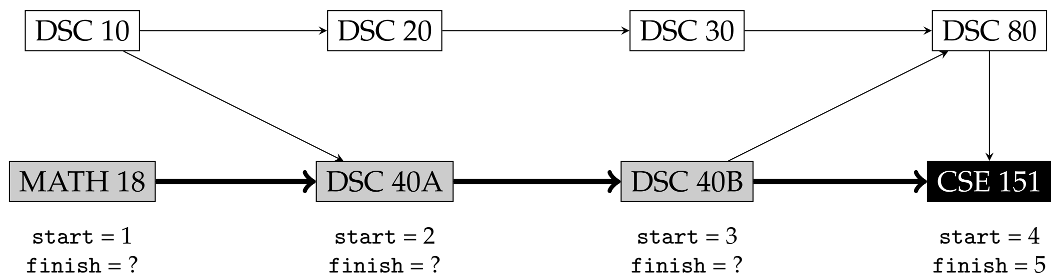
While visiting “DSC 40A”, node “DSC 40B” is discovered. A recursive call is made, and this node is assigned a start time of 3:



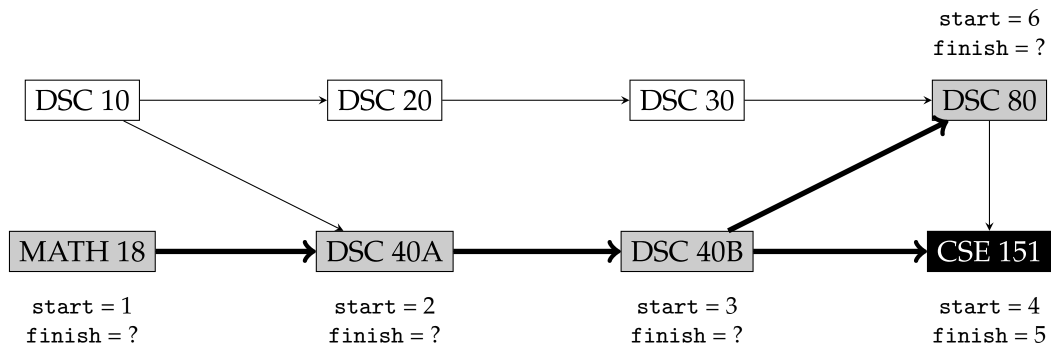
When we visit “DSC 40B”, the node has two undiscovered neighbors. Using the convention that `.neighbors()` returns neighbors in ascending order by label, “CSE 151” is discovered first. A recursive call is made, and “CSE 151” is assigned a start time of 4:



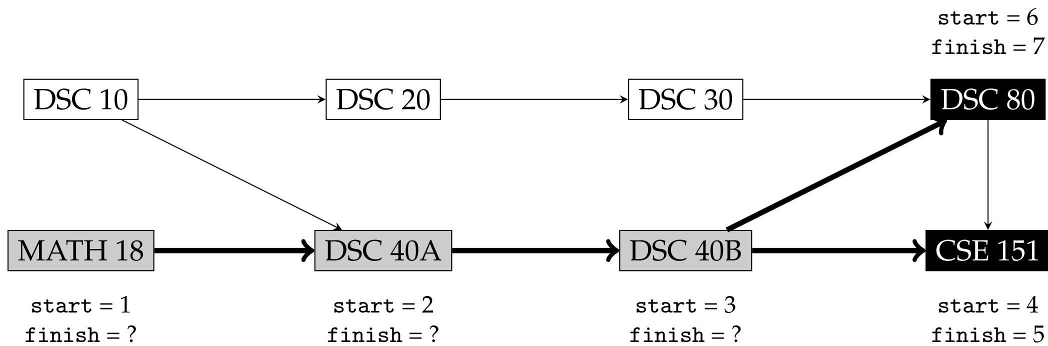
"CSE 151" has no undiscovered neighbors, and so the recursive call terminates. Before it does, however, the clock is incremented to 5. This becomes the node's finish time:



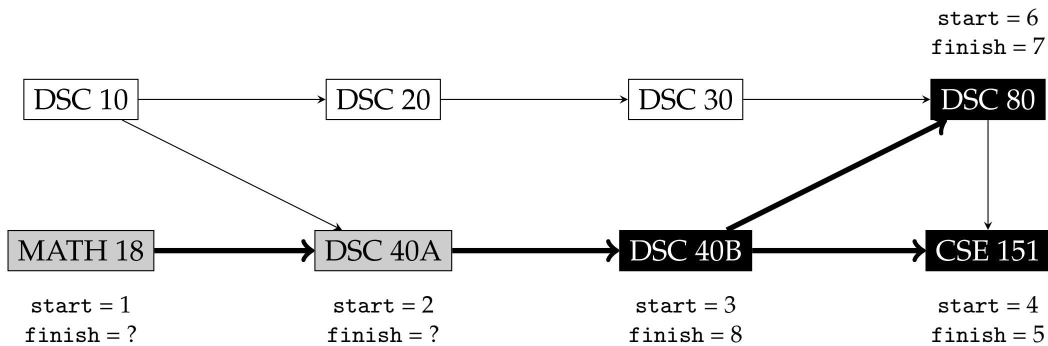
Now the search resumes the visit to "DSC 40B". This node has only one undiscovered neighbor remaining: "DSC 80". A recursive call is made to this node, the clock is incremented to 6, and this becomes its start time:



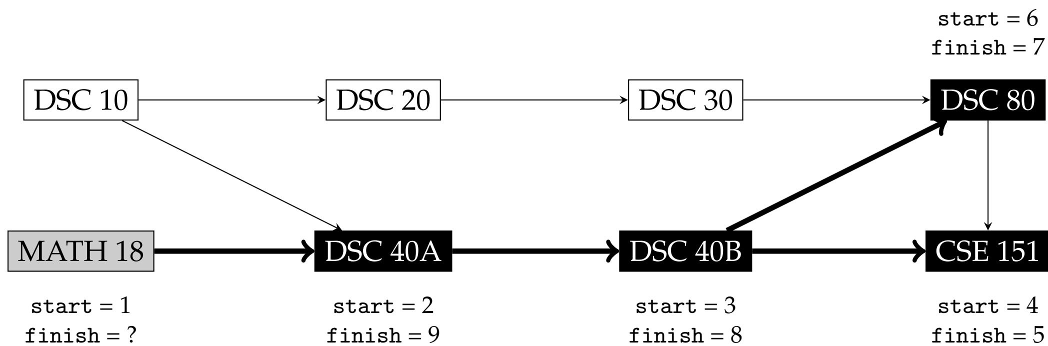
"DSC 80" has no undiscovered neighbors, so the visit terminates by incrementing to clock to 7. This becomes the node's finish time:



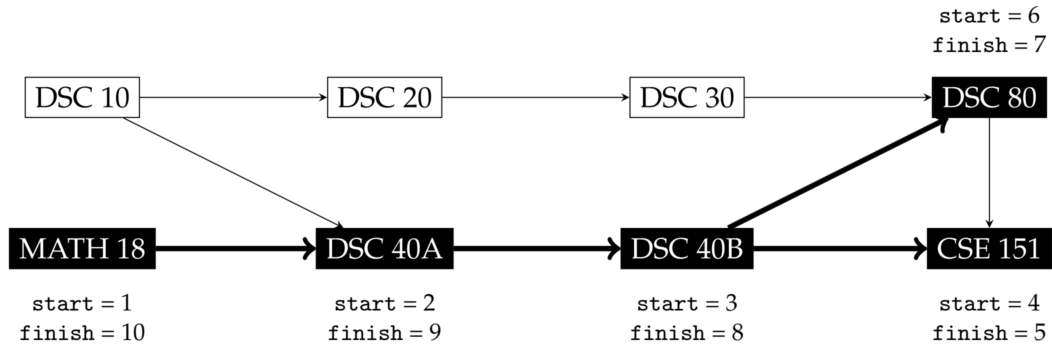
The search resumes at "DSC 40B", but now there are no undiscovered neighbors remaining. At this point we are finished with the node: we increment the clock to 8 and save this as the node's finish time:



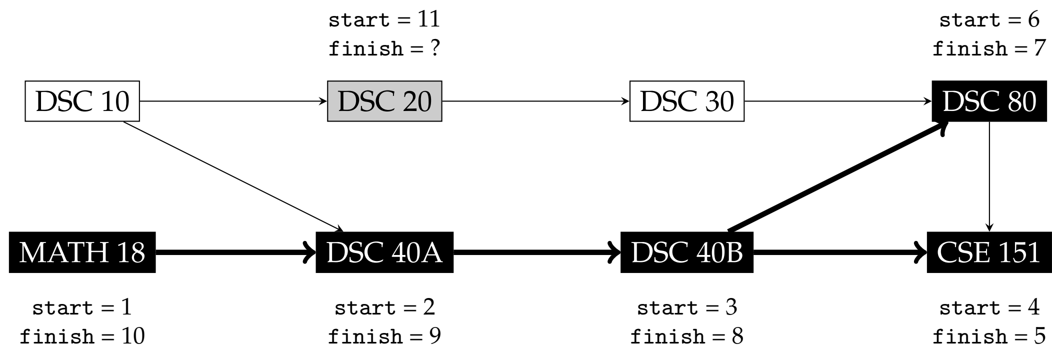
The search then "backtracks" to "DSC 40A". Again, there are no new nodes to discover, and so we are finished with the node. It is assigned a finish time of 9:



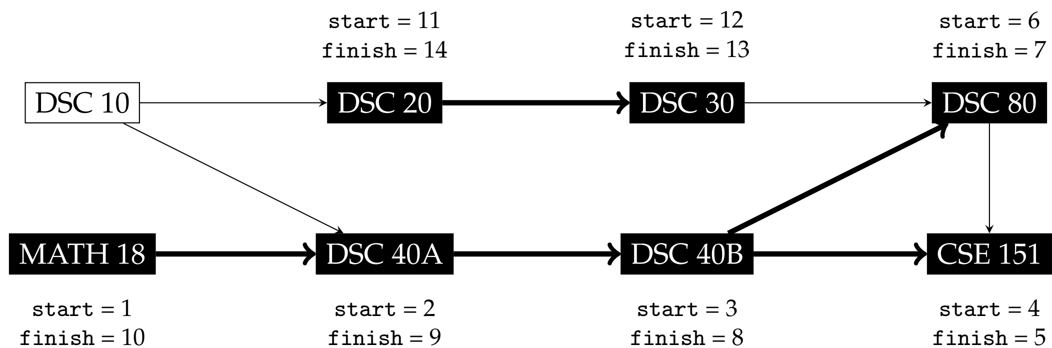
The search finally returns to where it started: "MATH 18". At this point there are no undiscovered neighbors to visit, and so we are finished with the node. It is assigned a finish time of 10:



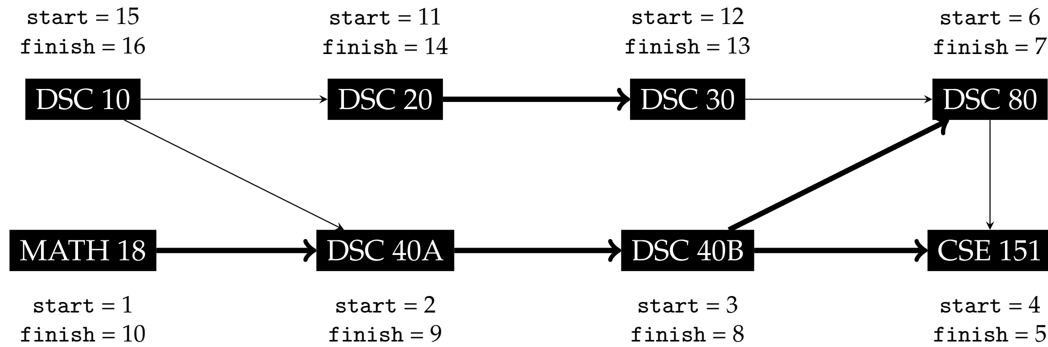
Execution of the program then returns to the `for`-loop in `full_dfs_times`. The loop will iterate until it finds the next undiscovered node remaining in the graph, at which point `dfs_times` will be called on that node. The order in which the nodes are produced is arbitrary, but let's say that the next undiscovered node is "DSC 20". At the time of this call, the clock is at 10; it is incremented, and 11 is assigned as the start time of "DSC 20":



Next, "DSC 30" is discovered and given a start time of 12. This node has no undiscovered neighbors, so it immediately finishes at time 13. The search returns to visiting "DSC 20" which at this point has no undiscovered neighbors. As a result, it too is marked as visited and assigned a finish time of 14:



Execution returns to the `for`-loop in `full_dfs_times`, which iterates until "DSC 10" is encountered. At this point, `dfs_times` is called on the node. Since it has no undiscovered neighbors, the start and finish times are 15 and 16, respectively:



Observe that the graph has 8 nodes and the largest finish time is 16. This is not a coincidence: the largest finish time will always be twice the number of nodes.

3.4.4 DFS Forest

As in the case of breadth-first search, it is often useful to construct a graph encoding the predecessor relationship between nodes. In particular, suppose we call `full_dfs_times` on a graph $G = (V, E)$, resulting in a `predecessor` dictionary. As we did with BFS in the previous section, we define the **depth-first search subgraph** G_{DFS} to be the undirected graph whose nodes are those which were reached by the DFS, and in which edge (u, v) exists if and only if u is the predecessor of v; i.e., `predecessor[v] == u`. More formally, $G_{\text{DFS}} = (V_{\text{DFS}}, E_{\text{DFS}})$, where:

$$V_{\text{DFS}} = \{v \in V : \texttt{predecessor[v] is not None}\} \cup \{s\},$$
$$E_{\text{DFS}} = \{(u, v) \in E : \texttt{predecessor[v] == u}\}.$$

The elements of E_{DFS} are called **depth-first search tree edges**.

The depth-first search subgraph is an undirected graph whose connected components are trees. We call such a graph a **forest**. The depth-search forest resulting from the depth-first search performed in the previous section is shown in Figure 3.2, with the root of each subtree drawn in red.

Since each edge in a DFS forest has a natural direction (pointing away from the root of its connected component), we can think of it as a directed graph, F. We will say that node v is a **descendent** of a node u in this tree if v is reachable from u in F. Likewise, we say that v is an **ancestor** of u if u is reachable from v in T. For example, the descendents of "DSC 40A" in the DFS forest above are "DSC 40B', "DSC 80", and "CSE 151". Node "DSC 40A" has only one ancestor: "MATH 18".

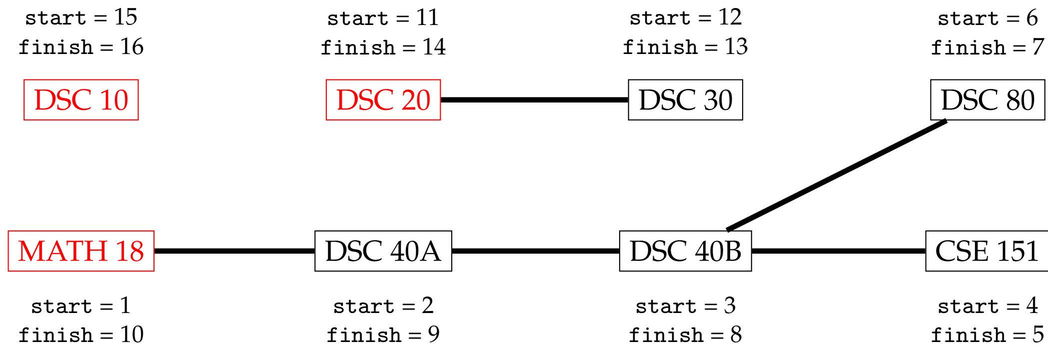


Figure 3.2: Start and finish times for the prerequisite graph.

The start and finish times of nodes and their descendants are related. Intuitively, we do not finish visiting a node in a DFS until we have finished visiting all of its descendants. As a result, we have the following claim:

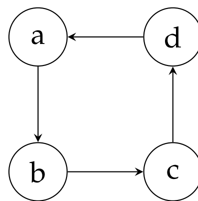
Claim 6. *Let v be a proper descendent of a node u in a DFS forest. Then:*

$$\texttt{times.start[u]} < \texttt{times.start[v]} < \texttt{times.finish[v]} < \texttt{times.finish[u]}$$

3.4.5 Topological Sort

Consider once again the graph of prerequisites shown in Figure 3.1. We wish to find a **topological sort** of its nodes in order to create a schedule of classes in which each course is taken after all of its prerequisites have been satisfied. In the language of graph theory, node u should appear before all nodes which are reachable from u in the prerequisite graph.

Intuitively, whether or not a node v is reachable from u is related to the finish times of both nodes. To make this precise, we must first define the concept of a directed acyclic graph. A **directed cycle** is a path from a node to itself which contains at least one edge. For instance, (a, b, c, d, a) is directed cycle in the graph below:



A **directed acyclic graph** (DAG) is one which contains no cycles.

Our prerequisite graph above is a directed acyclic graph, and it had better be: if there were a cycle in the graph there would be no way to satisfy all prerequisites. For instance, suppose there were a cycle:

This would mean that, in order to take DSC 30, one must take DSC 20. In order to take DSC 20, one must take DSC 10. But in order to take DSC 10, one must take DSC 30! Clearly this isn't possible; there is a *circular dependency*. In general, a topological sort of a directed graph is possible if and only if the graph is acyclic.

Armed with the definition of a DAG we are able to make the key intuition precise:

Claim 7. *Suppose v is reachable from $u (\neq v)$ in the directed acyclic graph G. Then the finish time of v is less than the finish time of u.*

We will prove this claim informally. There are two cases: either v is a descendant of u in the DFS forest, or it is not. If it is a descendant of u in the DFS forest, then Claim 6 implies that its finish time comes before u's finish time. If it is not a descendant of u in the DFS forest, then v must have been discovered by a previous search started at a different source. In this case, v's finish time again comes before u's finish time. Therefore, in both cases, v's finish time comes before u's finish time.

The above claim suggests an algorithm for performing a topological sort. If a node v has a finish time which comes before node u, it should be placed *after* node u in the sorted order. That is, we should sort the nodes by decreasing order of their finish time. Algorithm 12 provides Python code for this approach.

Algorithm 12 Topological sort.

```
def by_finish_time(item):
    return item[1]

def topological_sort(graph):
    times, predecessor = full_dfs_times(graph)
    # sort nodes in decreasing order by finish time
    return sorted(times.finish.items(), key=by_finish_time, reverse=True)
```

Consulting Figure **??**, which shows the start and finish times of a DFS on the data science prerequisite graph, this strategy results in a topological sort of:

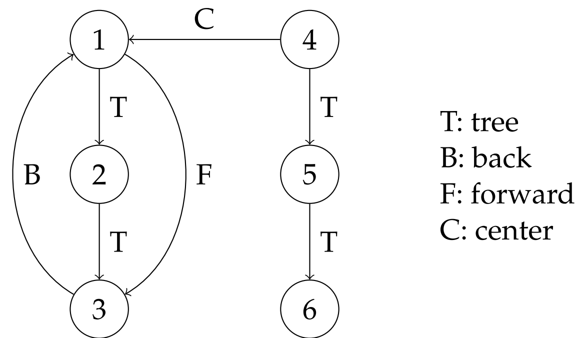
1. DSC 10
2. DSC 20
3. DSC 30
4. MATH 18
5. DSC 40A

6. DSC 40B
7. DSC 80
8. CSE 151

3.4.6 Edge Classification

Recall that an edge (u, v) in a graph $G = (V, E)$ is a DFS tree edge is v is discovered while visiting the node u during a DFS. Only a select few of the edges in the graph are tree edges, but we can classify the non-tree edges into three distinct groups based upon how they figure into the DFS forest. It will turn out that labeling the edges in this way holds to key to detecting the presence of cycles in the graph.

Suppose that a full DFS is run on the directed graph below, and assume that the first source node was node 1. The tree edges found during this DFS are labeled with a T. Suppose that the second source used by the full DFS is node 4; the tree edges found during this search are also labeled with T.



Now consider how we might label the rest of the edges in the graph, starting with $(3, 4)$. Because this edge points from a node 3 *back* to a higher node in the tree, we will call it a **back edge**. On the other hand, edge $(1, 3)$ points *forward* from a node high on the tree to another node which is *lower* on the tree (and which isn't a direct successor); as such, we will call it a **forward edge**. Edge $(4, 1)$, points from a node in one tree of the forest to a node in another tree; we will therefore call it a **cross edge**.

We can use node statuses and start/finish times to define the edge classifications more precisely. Suppose that when the edge (u, v) is encountered during the visit to node u:

- v is `'undiscovered'`. Then (u, v) is a tree edge;
- v is `'visited'`. Then (u, v) is a cross edge;
- v is `'pending'` and `start[v] > start[u]`. Then (u, v) is a forward edge.
- v is `'pending'` and `start[v] < start[u]`. Then (u, v) is a back edge.

It can be shown that a directed graph has a directed cycle if and only if a DFS finds a back edge.

3.5 Weighted Graphs and Minimum Spanning Trees

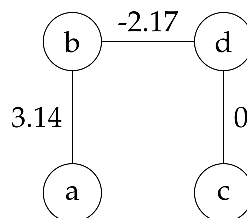
Consider again the flight graph discussed in previous sections in which there is an edge between two airports if there is a direct flight between them. Of course, some flights are cheaper than others, and we may wish to encode the cost of each flight within the graph itself. This is one motivation for introducing the concept of an **edge-weighted graph**, in which each edge is associated with a real number, known as a **weight**. More formally:

Definition 10. An **edge-weighted graph** is a graph $G = (V, E)$ together with an associated edge weight function, $\omega : E \to \mathbb{R}$.

Arrow notation. The notation $\omega : E \to \mathbb{R}$ means that the function ω maps *every* element of the set E to an element in the set $\mathbb{R}$ (the set of real numbers). In particular, this means that every edge must have an associated weight.

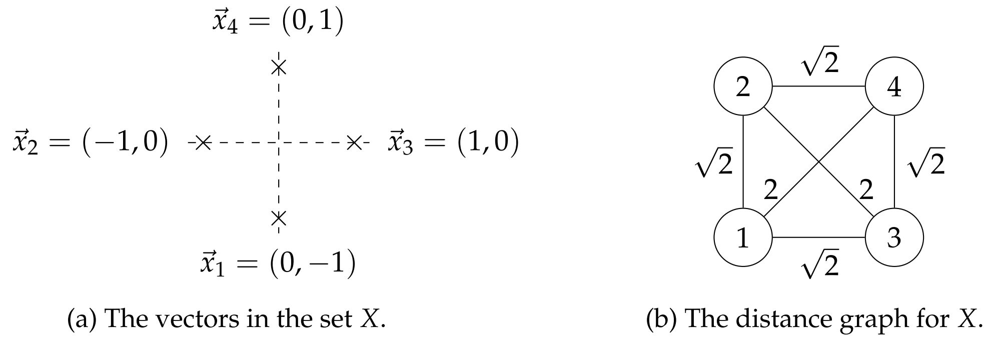
There are also **node-weighted graphs**, whose nodes are associated with scalar values, as well as graphs whose edges *and* nodes have weights. In practice, edge-weighted graphs seem to be most common. Therefore, when we use the term **weighted graph** without qualification, we are referring to an edge-weighted graph.

An example of a weighted graph is shown below.



Note that every edge in the graph has been assigned a weight. Weights may be positive or negative. This example shows a weighted undirected graph, but weighted directed graphs are also permissible.

Many problems can be framed in terms of weighted graphs. For instance, suppose we wish to find the sequence of flights from San Diego to Columbus with the smallest total cost. That is, we wish to find the **shortest weighted path**. While breadth-first search finds the shortest path in unweighted graphs, it does not solve the problem for weighted graphs. Instead, we must use a different algorithm which is specifically designed to take edge weights into account, such as Dijkstra's algorithm, or the Floyd-Warshall algorithm. We will not study these algorithms in this class.



(a) The vectors in the set X.

(b) The distance graph for X.

Figure 3.3

3.5.1 Distance Graphs

Data often comes to us as a set X of vectors in d-dimensional space, $\mathbb{R}^d$. There is a natural weighted graph associated with such a set of vectors: we make a node for each vector, and an edge between every pair of nodes. We then define the weight of the edge between nodes $\vec{x}$ and $\vec{y}$ to be the distance between them, $\|\vec{x} - \vec{y}\|$. We call the resulting graph the **distance graph** representing X. More formally:

Definition 11. Let $X = \{\vec{x}_1, \ldots, \vec{x}_n\}$ be a finite set of vectors in $\mathbb{R}^d$. The **distance graph** associated with X is the weighted, undirected graph $G = (V, E, \omega)$, where:

$$\begin{aligned}
&V = \{1, \ldots, n\}, \\
&E = \{(u, v) : u, v \in V \text{ and } u \neq v\}, \\
&\omega((u, v)) = \|\vec{x}_u - \vec{x}_v\|.
\end{aligned}$$

For example, suppose we have a set of vectors $X = \{\vec{x}_1, \vec{x}_2, \vec{x}_3, \vec{x}_4\}$, where

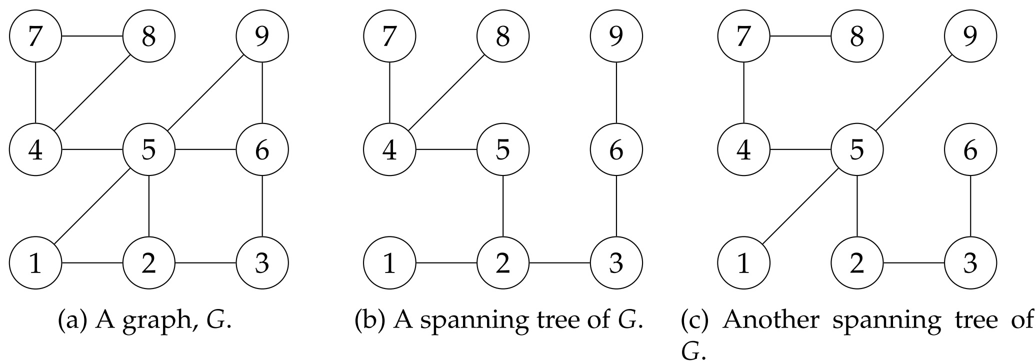
$$\begin{aligned}
\vec{x}_1 &= (0, -1)^\intercal, \\
\vec{x}_2 &= (-1, 0)^\intercal, \\
\vec{x}_3 &= (1, 0)^\intercal, \\
\vec{x}_4 &= (0, 1)^\intercal.
\end{aligned}$$

These vectors are depicted in Figure 3.3a. Then the distance graph for X is as shown in Figure 3.3b.

3.5.2 Minimum Spanning Trees

An important problem concerning weighted graphs is that of computing a **minimum spanning tree** (MST), that is, a tree[11] which spans all of the nodes of the graph and whose

[11] Recall that a tree is a connected graph which has one fewer edge than node.



(a) A graph, G. (b) A spanning tree of G. (c) Another spanning tree of G.

Figure 3.4

edges have the smallest total weight. Computing the minimum spanning tree is a crucial step in many other algorithms, and we will soon see that it is used in a simple clustering algorithm.

We start by defining a **spanning tree**:

Definition 12. Let $G = (V, E)$ be a connected, undirected graph (weighted or unweighted). A **spanning tree** is a tree $T = (V, E_T)$ which contains all of the nodes in G, and whose edge set E_T is a subset of E.

There may be many spanning trees of the same graph. For instance, Figure 3.4 shows two spanning trees of the same graph G. Note that each edge in the spanning tree must appear in the graph, G.

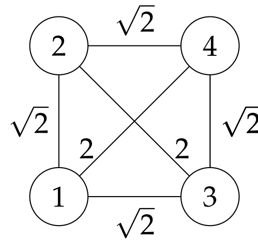
Intuitively, a **minimum spanning tree** of a weighted graph is a spanning tree whose total edge weight is minimized. More formally:

Definition 13. Let $G = (V, E, \omega)$ be a connected, weighted, undirected graph. Let $\mathcal{T}$ be the set of spanning trees of G. For any spanning tree $T \in \mathcal{T}$, define the total edge weight $\omega(T)$ to be:

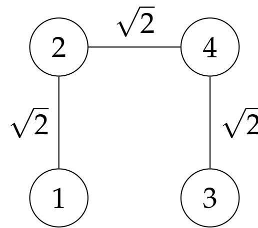
$$\omega(T) = \sum_{(u,v) \in T} \omega((u,v)).$$

A **minimum spanning tree** is a spanning tree $T^* \in \mathcal{T}$ such that for any spanning tree $T \in \mathcal{T}$, $\omega(T^*) \leq \omega(T)$.

For instance, consider the weighted graph shown below:



One can verify that a minimum spanning tree of this graph is the tree below:



This tree has total weight $3\sqrt{2}$. Of course, this is not the only spanning three with this total weight; there are in fact four minimum spanning trees of this graph.

3.5.3 Clustering

Section 1.6.3 introduced the problem of clustering a data set into two groups. Figure 3.5a shows the geyser eruption data set that was considered in that section. The geyser apparently erupts in one of two "modes": either the geyser erupts for a small duration after a small waiting period, or it erupts for a long duration after a lengthy wait time. We wish to design an algorithm for automatically discovering this structure in the data.

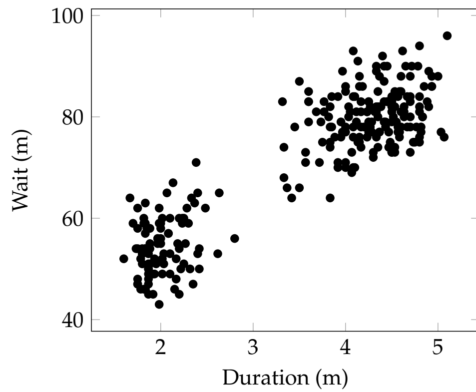
We formalized this problem as one of partitioning the data into clusters B and R which are maximally separated, as in Figure 3.5b. More precisely, we defined the **separation** $\delta(B, R)$ of two sets B and R to be the smallest distance between any two points, one of them in B and the other in R. That is:

$$\delta(B, R) = \min_{\vec{b} \in B, \vec{r} \in R} \|\vec{b} - \vec{r}\|.$$

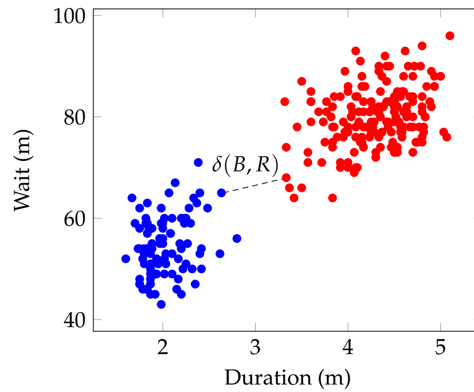
The formal computational problem then becomes:

GIVEN: Data vectors $X = \{x_1, \ldots, x_n\}$.
COMPUTE: A partition (B, R) of the data with maximum separation, $\delta(B, R)$.

This is an optimization problem, and the brute-force approach of trying every possible partitioning of the data takes exponential time. It turns out, however, that this problem can be efficiently solved by computing a minimum spanning tree of the data.



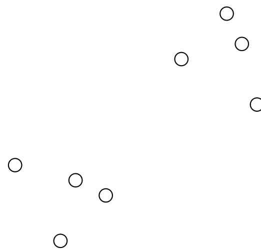
(a) The geyser eruption data.



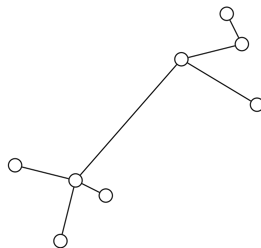
(b) The eruption data clustered into two groups.

Figure 3.5

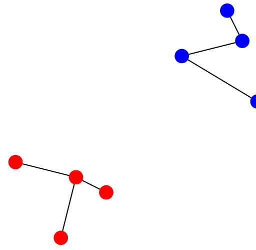
To see how we may obtain a clustering from a minimum spanning tree, suppose we are given the following data:



Consider the distance graph of this data. A possible minimum spanning tree of this graph is shown below, where the graph's nodes are placed in the same positions as the data points they represent:



A minimum spanning tree is a connected graph, so deleting a single edge necessarily results in two connected components. Suppose we remove the largest edge in the tree. The connected components obtained are colored in red and blue below:



We will interpret these connected components as clusters. We claim that clustering the data in this way solves the above computational problem; that is, it finds the partition of the data which maximizes the separation, δ.

Theorem 10. *Let X be a set of data vectors. Let B^* and R^* be the connected components of the graph obtained by deleting the largest edge from a minimum spanning tree T^* of the distance graph of X. Then B^* and R^* are maximally-separated; that is, $\delta(B^*, R^*) \geq \delta(B', R')$, where (B', R') is any other partition of X.*

In proving this theorem, we will use the following claim without proof:

Claim 8. *Let Δ be the weight of the longest edge in the minimum spanning tree T^*. Then the separation between B and R is Δ. That is, $\delta(B, R) = \Delta$.*

We now prove the theorem:

Proof. Let Δ be the weight of the longest edge in the minimum spanning tree. Let (B', R') be any partition of X. There must be an edge in T^* which starts in B' and ends in R', as otherwise T^* would be disconnected; call this edge (u, v). By the second claim, $\delta(B', R') \leq \omega(u, v)$. But the length of this edge is at most Δ, since Δ is the length of the largest edge in T^*; that is, $\omega(u, v) \leq \Delta$. Therefore $\delta(B', R') \leq \Delta$. By the above claim, $\delta(B^*, R^*) = \Delta$, and so $\delta(B', R') \leq \delta(B^*, R^*)$. Since (B', R') was an arbitrary partition, this shows that (B^*, R^*) has maximal separation. $\square$