1 The Seemingly Magical Power of Data Structures & Algorithms

Data structures and algorithms are the foundation of computing. Every software system relies heavily on them. This chapter offers a glimpse into what we are going to study in this book.

Every computer program—large and small—organizes and manipulates data in some form. The study of data structures (how to structure data) and algorithms (how to systematically work with them) is therefore immensely important and is the center piece of everyone's computing education. It is the foundation upon which dependable technologies, countries' infrastructure, and multi-million-dollar companies are built. Below are some examples where our lives constantly have to interact with data structures and algorithms:

Phone Contact Lookup. Gone are the days in which we have to remember phone numbers or keep a Rolodex bulged with important contacts. Our phones are capable of storing thousands of contacts, which can be instantaneously looked up—via a complete name match or even a partial match. Two technological advances have made this possible: First, modern devices—even the tiniest phones—have improved computational power. But more impressively, it is how the vast entries of contacts are organized—using efficient data structures that enable speedy look-up algorithms to sift through them.

When In Doubt, Search. Power users use complex queries to find what they want on the Internet and on our local computers. Amazingly, search engines can return the relevant results in a split second. This is no easy task considering how many web pages and files there are that the system has to go through. How are data organized to allow for such efficient searching?

Map Navigation. "Navigate to work," says an avid user of a popular maps platform—and the trip is guided by an all-knowing guru, who gives turn-by-turn directions of the best personalized route to the destination. Accomplishing this requires solving and coordinating many complex tasks: How is the map represented? How do we know where we are? What is the best route? Using such a service thus gives us a front-row seat to appreciate how data structures and algorithms come together to improve our quality of life.

Undoubtedly, data structures and algorithms are the fundamental building blocks of any software system. Their study provides a common vocabulary to discuss problem solving and lays out frameworks to systematically reason about and solve problems. This book aims to provide you with the skills and tools to confidently write efficient, maintainable, and correct programs. We will use Java in our examples and for this reason, the first several chapters will

briefly discuss Java concepts assuming the readers are familiar with another high-level programming language such as Python. However, the concepts and techniques here can be applied across problem domains, programming languages, and computer architectures.

1.1 Abstraction vs. Implementation

In this book, we will make a distinction between the description of a problem and the description of a solution. As an example, consider the following description of a problem:

- **Input:** a sequence of n arbitrarily-ordered integers.
- **Output:** the same sequence of integers now ordered from small to large.

This problem is typically known as the sorting problem. As a *problem* description, it merely specifies the expected input/output behavior—but does *not* prescribe how to accomplish it.

For a computational problem, an *algorithm* is a concrete, well-defined problem-solving steps that makes up a solution. There can be many ways to solve a problem. For example, the sorting problem (above) can be solved by the following algorithm:

> **Algorithm: Bubble Sort.** Repeatedly make a pass through the array comparing each adjacent pair of numbers and swapping them if they are out of order. Keep doing this until no swaps are made.

Using mathematical tools that we will discuss later in the book, this algorithm—known as bubble sort—turns out to need about n^2 steps on the most difficult array of length n. But there are other algorithms that can solve this problem much faster. For instance, algorithms such as quick sort and merge sort (discussed later in the book) can solve the same problem much more quickly, requiring only about $n \log n$ steps. The important point here is that a problem can have multiple solutions with varying efficiency.

Abstract Data Types and Data Structures

In the context of data organization, we have a specific term for problems in this arena: An *abstract data type* (ADT) specifies a set of operations and their behavior from the viewpoint of a user. For example, we can define a integer bag abstract data type keeping a collection of integers while supporting the following operations: (i) `add(num)` adds an integer to the bag; and (ii) `removeOne()` removes and returns an arbitrary integer from the bag.

Like a problem description, the description of an ADT merely specifies the expected behavior. It does *not* say how to concretely represent the data and support the operations.

A concrete implementation of an ADT is known as a *data structure*, which describes the organization and management of data—i.e., how the data will be stored and the operations will be supported. There can be many data structures implementing the same ADT often with different properties.

1.2 Correctness and Efficiency

When we set out to design data structures and algorithms, and implement them, two main metrics of consideration are key:

Correctness. Does the design/code behave according to the specification? When dealing with correctness, we can resort to mathematical reasoning to understand our design and implementation. Code testing (e.g., unit testing) complements mathematical reasoning, adding further confidence that the code indeed works correctly. We will look at both techniques in this book.

Efficiency. How fast and how much resource (e.g., main memory) does our solution require? This consideration can be tackled using mathematical tools, allowing us to, for example, make sense of the speed of our solution without writing real code. Empirical analysis can further our understanding, in many cases helping programmers fine-tune the implementation for even better performance. In subsequent chapters, this book will discuss both lines of techniques.

In practice, we generally require our solution to be correct on all cases that we care about. But we can be more relaxed about efficiency requirements: practical solutions only need to be good enough. In fact, we sometimes aim for a slightly less efficient solution in favor of simpler code that is hopefully less error-prone to implement.

Moreover, from the point of view of software engineering, there are numerous patterns and tips for structuring code and components of an implementation. This area is beyond the scope of this book. We will only briefly touch on such design issues as they shape how some data structures and ADTs should be designed.

1.3 Preliminaries and Notation

This book assumes prior familiarity with a high-level programming language such as Python and knowledge of discrete mathematics equivalent to what is typically taught in US colleges at the undergraduate level. This means the readers are expected to have mastered or are studying concepts such as sets, functions, relations, summations, recurrences, counting, etc.

Throughout the book, the following standard sets are used:

$$\begin{aligned}\mathbb{Z} &= \{0, +1, -1, +2, -2, +3, -3, \dots\} \\ \mathbb{Z}_+ &= \{1, 2, 3, \dots\} \\ \mathbb{Z}_- &= \{-1, -2, -3, \dots\}\end{aligned}$$

Why do we use the letter $\mathbb{Z}$ for integers? The notation $\mathbb{Z}$ comes from the German word Zahlen, which means numbers.

In similar manners, we write $\mathbb{R}$ for the set of all real numbers and $\mathbb{R}_+$, for the positive reals.

Given a sequence $a_1, a_2, a_3, \dots$ of numbers, often denoted by $\{a_i\}_{i \geqslant 1}$, the summation $a_1 + a_2 + a_3 + \cdots + a_n$, where $n \in \mathbb{Z}_+ \cup \{0\}$ is a nonnegative

integer, is more concisely written in a *summation notation* as

$$\sum_{k=1}^{n} a_k.$$

This means when $n = 0$, the summation is empty and yields a value of 0. Extending this notation, we express the infinite sum as

$$\sum_{k=1}^{\infty} a_k = \lim_{n\to\infty} \sum_{k=1}^{n} a_k.$$

Commonly-used summations in this book include:

$$\sum_{k=1}^{n} k = 1+2+3+\cdots+n = \frac{n(n+1)}{2}$$

$$\sum_{k=1}^{n} k^2 = 1^2+2^2+3^2+\cdots+n^2 = \frac{n(n+1)(2n+1)}{6}$$

$$\sum_{k=0}^{n} a^k = 1+a+a^2+\cdots+a^n = \frac{1-a^{n+1}}{1-a} \text{ for } a \in \mathbb{R}, a \neq 1$$

$$\sum_{k=0}^{\infty} a^k = 1+a+a^2+\cdots = \frac{1}{1-a} \text{ for } -1 < a < 1, a \in \mathbb{R}.$$

1.4 Two Algorithmic Thinking Examples

We will look at two simple examples that will illustrate the kinds of questions we seek to investigate in this book and what their answers might look like. The discussion here will be high-level, but readers are encouraged to think about how we might program them in real code.

Maximum Finding: Who Has The Largest Number?

Imagine walking into a large lecture hall with hundreds of students, all seated waiting for the start of the first lecture. As part of an activity that is about to take place, everyone has a piece of paper and was instructed to write down their favorite integer. You are tasked with

finding the largest number in the room.

In more precise terms, the *input* is everyone's number, and the *expected output* is the highest number. Such input/output behaviors define a *problem*. In this view, we're just asking the question: find the largest number from a collection of numbers given as input.

We'll consider two natural strategies in turn.

Approach I: Basic Iteration. You will stand in front of the hall with an important task: keep track of the largest number announced so far (initially

$-\infty$) as you ask everyone to call out their number one by one. There is no doubt that if you don't make a mistake, the number you have at the end of this process is the largest number in the room. This simple process is akin to the following familiar code for computing the maximum number in a sequence:

```
maxSoFar = -INFINITY
for num in seq:
    if num > maxSoFar:
        maxSoFar = num
```

Because two people can't talk at the same time in this process, the number of rounds required will be as many as the number of people in the room. Hence, if n denotes the number of people, we say that the process takes about n rounds to complete.

Among others, this means that if we have 100 students, it will take around 100 rounds. For a larger population, if we have 1024 students, it will take about 1024 rounds!.

Is there a different strategy that uses fewer rounds?

Approach II: Pairing Up. The key idea will be to allow more of the necessary work to be completed in each round. This can be accomplished by taking a more social approach. Instead of you maintaining the maximum so far yourself, you will instruct student participants to involve more in the computation. Begin by asking everyone to stand up. We'll carry out the following steps until only one person is left standing:

(i) Everyone standing pairs up (if someone doesn't have a partner, just stand there for this round).

(ii) Each pair talks: the person with a smaller number will sit down (breaking ties arbitrarily) and the other person will continue to stand.

These two steps constitute one round. In a round, all the pairs can compare numbers and make the decision in parallel.

The example below illustrates this process:

```
Round #1:
  4 -- 2   11 -- 9   7 -- 53    9 -- 5
  4        11             53    9

Round #2:
  4   --   11             53 -- 9
           11             53

Round #3:
           11     --      53
                          53
```

Correctness is less straightforward than the previous approach. We claim that the last person standing has the largest number. Intuitively, this is like a tournament, where the argument is roughly that when someone sits down, that person cannot be the maximum as another person (his/her pair) has a larger number.

But how many rounds does the new approach need? We will give a crude analysis. Using n to denote the number of students like before, we'll observe that there are about $n/2$ pairs and hence after one round, we are left with $n/2$ students. The same reasoning shows that

- After 2 rounds, we are left with $n/4 = n/2^2$ students.
- After 3 rounds, we are left with $n/8 = n/2^3$ students.
- ...
- After r rounds, we are left with $n/2^r$ students.

The question is then after how many rounds would the number of students become 1? The answer* can be found by solving for r in the equation $n/2^r = 1$. This means this approach takes about $r = \log_2(n)$ rounds.

To appreciate the difference between the two approaches, for $n = 1024$, Approach I will require about 1024 rounds and Approach II will require about $\log_2(1024) = 10$ rounds. The gap grows even wider for larger n.

Remarks. As it turns out, the pairing up approach requires far fewer rounds than the standard basic iteration approach. But how about the number of "comparisons"? In this case, a comparison is when two numbers are compared—either directly in the case of `maxSoFar` or when two people in a pair compare numbers. You should prove this to yourself, but as it turns out, both approaches end up with the same† $n - 1$ comparisons!

Searching: Where Is That Number?

As another example, consider a problem where the input is a sequence of numbers `a`, together with another number `k`, and we have to determine where the number `k` appears in `a`—or whether it doesn't exist. More concretely, we are to write a function `find(a, k)` to return an index `i` such that `a[i] == k` or otherwise return `i = -1`. Below are some example calls for concreteness:

```
find({3, 5, 2, 7, 9, 3}, 7) // ==> 3
find({3, 5, 2, 7, 9, 3}, 1) // ==> -1
find({3, 5, 2, 7, 9, 3}, 3) // ==> 0 (or equally correct: 5)
```

where `{3, 5, 2, 7, 9, 3}` is an array (sequence/list).

One natural idea is to look at every element of the array in turn and each time compare it with the number `k` that we are looking for. This idea is known as *linear search* and is easy to implement. But how fast is this approach? It takes time proportional to the length of the array in the worst case, because we will have to look through the whole array before we know that the number we're looking for really is not there. This means if it takes 1 µs to search in an array of length 10,000, it will take roughly 1000 µs in an array of length 10,000,000—a 1000x increase. For long sequences, the approach might not always be practical.

Is there an approach that takes time less than the length of the array? In other words, can we avoid looking at every element of the array? At first glance,

*To be precise, we will need to be careful about rounds in which someone doesn't have a pair.
†A smarter version of basic iteration should not need to compare $-\infty$ with the first number, saving one comparison.

this is impossible: intuitively, we cannot possibly know where the number `k` might be hiding in the input array without checking every spot.

However, we can make the situation more favorable. If the numbers in the input array is ordered from small to large—we say it's sorted—there is much room to play. First but crucially, comparing `k` with `a[i]` can help narrow down where to look in the array. For ease, let us assume that `k` appears in `a`.

- If `k > a[i]`, we know precisely that `k` will have to be before `a[i]`.
- The converse is also true: if `k < a[i]`, then `k` will have to be after `a[i]`.

Hence, after such a comparison, the other side of the array is irrelevant.

To make the idea concrete, consider finding `19` in the array

```
{1, 7, 10, 14, 19, 23, 24, 28, 33, 34, 35, 48, 51, 57}
```

For reasons that will be clear soon, we will always compare with the middle element of the remaining array.

Remaining Array	Mid Element	Comparison
{1,7,10,14,19,23,24,28,33,34,35,48,51,57}	28	19 < 28
{1,7,10,14,19,23,24}	14	19 > 14
{19,23,24}	23	19 < 23
{19}	19	19 == 19

This new approach is known as *binary search*. Notice that by always comparing against the middle element, after each comparison, half of the remaining array is no longer relevant. Therefore, if we start with an array size n, after one comparison, the portion of the array that remains relevant has size roughly $n/2$, which will become $n/4$ after another comparison. How many times can we divide n by 2 until it becomes at most 1? The answer is about $\log_2 n$ times. This means, binary search only needs to make about $\log_2 n$ comparisons in the worst case provided that the input array is sorted.

What does this mean compared to linear search? Say n is $10,000,000$. Linear search will need compare with roughly $10,000,000$ numbers whereas binary search will use about $\log_2(10,000,000) \approx 23$ comparisons. This is a marked improvement.

But how can we make the array sorted in the first place and would that be worth it? We will discuss sorting in Chapter 8 and delve into both searching ideas in greater detail in Chapter 10.

Chapter Notes

As you go through this book, complementary references will be useful—to seek an alternative explanation or to delve deeper into some topics. Excellent references on introductory data structures and algorithms include Goodrich et al. [GTG14], Sedgewick and Wayne [SW11], and Morin [Mor13]. For a deeper dive, check out books by Cormen el al. [Cor+09], Kleinberg and Tardos [KT06], Dasgupta et al. [DPV08], and Erickson [Eri19]. To pick up Java, we recommend starting out with *Java: A Beginner's Guide* by Schildt [Sch18a] and *Java for Python Programmers* by Miller [Mil08].