Chapter 2

Basics of Algorithm Analysis

Analyzing algorithms involves thinking about how their resource requirements—the amount of time and space they use—will scale with increasing input size. We begin this chapter by talking about how to put this notion on a concrete footing, as making it concrete opens the door to a rich understanding of computational tractability. Having done this, we develop the mathematical machinery needed to talk about the way in which different functions scale with increasing input size, making precise what it means for one function to grow faster than another.

We then develop running-time bounds for some basic algorithms, beginning with an implementation of the Gale-Shapley algorithm from Chapter 1 and continuing to a survey of many different running times and certain characteristic types of algorithms that achieve these running times. In some cases, obtaining a good running-time bound relies on the use of more sophisticated data structures, and we conclude this chapter with a very useful example of such a data structure: priority queues and their implementation using heaps.

2.1 Computational Tractability

A major focus of this book is to find efficient algorithms for computational problems. At this level of generality, our topic seems to encompass the whole of computer science; so what is specific to our approach here?

First, we will try to identify broad themes and design principles in the development of algorithms. We will look for paradigmatic problems and approaches that illustrate, with a minimum of irrelevant detail, the basic approaches to designing efficient algorithms. At the same time, it would be pointless to pursue these design principles in a vacuum, so the problems and

approaches we consider are drawn from fundamental issues that arise throughout computer science, and a general study of algorithms turns out to serve as a nice survey of computational ideas that arise in many areas.

Another property shared by many of the problems we study is their fundamentally *discrete* nature. That is, like the Stable Matching Problem, they will involve an implicit search over a large set of combinatorial possibilities; and the goal will be to efficiently find a solution that satisfies certain clearly delineated conditions.

As we seek to understand the general notion of computational efficiency, we will focus primarily on efficiency in running time: we want algorithms that run quickly. But it is important that algorithms be efficient in their use of other resources as well. In particular, the amount of *space* (or memory) used by an algorithm is an issue that will also arise at a number of points in the book, and we will see techniques for reducing the amount of space needed to perform a computation.

Some Initial Attempts at Defining Efficiency

The first major question we need to answer is the following: How should we turn the fuzzy notion of an "efficient" algorithm into something more concrete?

A first attempt at a working definition of *efficiency* is the following.

Proposed Definition of Efficiency (1): *An algorithm is efficient if, when implemented, it runs quickly on real input instances.*

Let's spend a little time considering this definition. At a certain level, it's hard to argue with: one of the goals at the bedrock of our study of algorithms is solving real problems quickly. And indeed, there is a significant area of research devoted to the careful implementation and profiling of different algorithms for discrete computational problems.

But there are some crucial things missing from this definition, even if our main goal is to solve real problem instances quickly on real computers. The first is the omission of *where*, and *how well*, we implement an algorithm. Even bad algorithms can run quickly when applied to small test cases on extremely fast processors; even good algorithms can run slowly when they are coded sloppily. Also, what is a "real" input instance? We don't know the full range of input instances that will be encountered in practice, and some input instances can be much harder than others. Finally, this proposed definition above does not consider how well, or badly, an algorithm may *scale* as problem sizes grow to unexpected levels. A common situation is that two very different algorithms will perform comparably on inputs of size 100; multiply the input size tenfold, and one will still run quickly while the other consumes a huge amount of time.

So what we could ask for is a concrete definition of efficiency that is platform-independent, instance-independent, and of predictive value with respect to increasing input sizes. Before focusing on any specific consequences of this claim, we can at least explore its implicit, high-level suggestion: that we need to take a more mathematical view of the situation.

We can use the Stable Matching Problem as an example to guide us. The input has a natural "size" parameter N; we could take this to be the total size of the representation of all preference lists, since this is what any algorithm for the problem will receive as input. N is closely related to the other natural parameter in this problem: n, the number of men and the number of women. Since there are $2n$ preference lists, each of length n, we can view $N = 2n^2$, suppressing more fine-grained details of how the data is represented. In considering the problem, we will seek to describe an algorithm at a high level, and then analyze its running time mathematically as a function of this input size N.

Worst-Case Running Times and Brute-Force Search

To begin with, we will focus on analyzing the *worst-case* running time: we will look for a bound on the largest possible running time the algorithm could have over all inputs of a given size N, and see how this scales with N. The focus on worst-case performance initially seems quite draconian: what if an algorithm performs well on most instances and just has a few pathological inputs on which it is very slow? This certainly is an issue in some cases, but in general the worst-case analysis of an algorithm has been found to do a reasonable job of capturing its efficiency in practice. Moreover, once we have decided to go the route of mathematical analysis, it is hard to find an effective alternative to worst-case analysis. Average-case analysis—the obvious appealing alternative, in which one studies the performance of an algorithm averaged over "random" instances—can sometimes provide considerable insight, but very often it can also become a quagmire. As we observed earlier, it's very hard to express the full range of input instances that arise in practice, and so attempts to study an algorithm's performance on "random" input instances can quickly devolve into debates over how a random input should be generated: the same algorithm can perform very well on one class of random inputs and very poorly on another. After all, real inputs to an algorithm are generally not being produced from a random distribution, and so average-case analysis risks telling us more about the means by which the random inputs were generated than about the algorithm itself.

So in general we will think about the worst-case analysis of an algorithm's running time. But what is a reasonable analytical benchmark that can tell us whether a running-time bound is impressive or weak? A first simple guide