15-451 Algorithms, Fall 2000
Lectures 1-5

Avrim Blum

August 29, 2000

Contents

Lecture 1

Introduction to Algorithms

1.1 Overview

The purpose of this lecture is to give a brief overview of the topic of Algorithms and the kind of thinking it involves: why we focus on the subjects that we do, and why we emphasize proving guarantees. We also go through an example of a problem that is easy to relate to (multiplying two numbers) in which the straightforward approach is surprisingly not necessarily the best one. This example leads naturally into the study of recurrences, which is the topic of the next lecture, and provides a forward pointer to topics such as the FFT later on in the course.

Material in this lecture:

- Administrivia (see handout from web page)
- What is the study of Algorithms all about?
- Why do we care about specification and proving guarantees?
- The Karatsuba multiplication algorithm.
- Strassen's matrix multiplication algorithm.

1.2 Introduction

This course is about the design and analysis of algorithms — how to design correct, efficient algorithms, and how to think clearly about analyzing their correctness and running time.

What is an algorithm? At its most basic, an algorithm is a method for solving a computational problem. Along with an algorithm comes a specification that says what the algorithm's guarantees are. For example, we might be able to say that our algorithm indeed correctly solves the problem in question and runs in time at most $f(n)$ on an input of size n. This course is about the whole package: the design of efficient algorithms, *and* proving that they meet desired specifications. For each of these parts, we will examine important techniques that have been developed, and with practice we will build up our ability to think clearly about the key issues that arise.

The main goal of this course is to provide students with intellectual tools for designing and analyzing their own algorithms for problems they need to solve in the future. Some tools we will discuss are Dynamic Programming, Divide-and-Conquer, Data Structure design principles, Randomization, Network Flows, Linear Programming, and the Fast Fourier Transform. Some analytical tools we will discuss and use are Recurrences, Probabilistic Analysis, Amortized Analysis, and Potential Functions.

There is also a dual to algorithm design: Complexity Theory. Complexity Theory looks at the intrinsic difficulty of computational problems — what kinds of specifications can we expect *not* to be able to achieve? In this course, we will delve a bit into complexity theory, focusing on the somewhat surprising and counter-intuitive notion of NP-completeness. We will also spend some time on cryptography. Cryptography is interesting from the point of view of algorithm design because one uses a problem that's assumed to be intrinsically hard in order to construct an algorithm (e.g., an encryption method) whose security rests on the difficulty of solving that hard problem.

1.3 On guarantees and specifications

One focus of this course is on proving correctness and running-time guarantees for algorithms. Why is having such a guarantee useful? Suppose we are talking about the problem of sorting a list of n numbers. It is pretty clear that it is nice to know at least that our algorithm produces correct solutions, so we don't have to worry about whether it has given us the right answer all the time. But, why analyze running time? Why not just code up our algorithm and test it on 100 random inputs and see what happens? Here are a few reasons that motivate our concern with this kind of analysis:

Composability. A guarantee on running time gives a type of "clean interface". It means that we can use the algorithm as a subroutine in some other algorithm, without needing to worry whether the kinds of inputs on which it is being used now necessarily match the kinds of inputs on which it was originally tested.

Scaling. The types of guarantees we will examine will tell us how the running time scales with the size of the problem instance. This is useful to know for a variety of reasons. For instance, it tells us roughly how large a problem size we can reasonably expect to handle.

Designing better algorithms. Analyzing the asymptotic running time of algorithms has shown itself to be a useful way of thinking about algorithms that often leads to nonobvious improvements.

Understanding. An analysis can tell us what parts of an algorithm are crucial for what kinds of inputs, and why. If we later get a different but related task, we can often use our analysis to quickly tell us if a small modification to our existing algorithm can be expected to give similar performance to the new problem.

Complexity-theoretic motivation. In Complexity Theory, we want to know: "how hard is fundamental problem X really?" For instance, we might know that no algorithm can possibly run in time $o(n \log n)$ (growing more slowly than $n \log n$ in the limit) and we have an algorithm that runs in time $O(n^{3/2})$. This tells us how well we understand the problem, and also how much room for improvement we have.

You can probably think of more reasons too.

1.4 An example: Karatsuba Multiplication

One of things that makes algorithm design "Computer Science" is that solving a problem in the most obvious way from its definitions is often not the best way to get a solution. A simple example of this is multiplication.

Say we want to multiply two n-bit numbers: for example, 41×42 (or, in binary, 101001×101010). According to the definition of what it means to multiply, what we are looking for is the result of adding 41 to itself 42 times (or vice versa). You could imagine actually computing the answer that way (i.e., performing 41 additions), which would be correct but not particularly efficient. If we used this approach to multiply two n-bit numbers, we would be making $\Theta(2^n)$ additions. This is exponential in n even without counting the number of steps needed to perform each addition. And, in general, exponential is bad.[1] A better way to multiply is to do what we learned in grade school:

```
       101001
x      101010
-------------
      1010010
    101001
+ 101001
-------------
  11010111010 = 1722
```

The running time here is $O(n^2)$, because we are performing n additions, each of which takes $O(n)$ time. So, this is a simple example where even though the problem is defined "algorithmically", using the definition is not the best way of solving the problem.

Is the above method the fastest way to multiply two numbers? It turns out it is not. Here is a faster method called Karatsuba Multiplication, discovered by Anatoli Karatsuba, in Russia, in 1962. In this approach, we take the two numbers X and Y and split them each into their most-significant half and their least-significant half.

```
                           ---------------
X = A*2^{n/2} + B          |  A   |   B  |
                           +-------------+
Y = C*2^{n/2} + D          |  C   |   D  |
                           ---------------
```

We can now write the product of X and Y as

$$XY = 2^n AC + 2^{n/2}BC + 2^{n/2}AD + BD. \tag{1.1}$$

[1] This is reminiscent of an exponential-time sorting algorithm I once saw in Prolog. The code just contains the definition of what it means to sort the input — namely, to produce a permutation of the input in which all elements are in ascending order. When handed directly to the interpreter, it results in an algorithm that examines all $n!$ permutations of the given input list until it finds one that is in the right order.

This does not yet seem so useful. If we use (1.1) as a recursive multiplication algorithm, we need to perform four $n/2$-bit multiplications, three shifts, and three $O(n)$-bit additions. If we use $T(n)$ to denote the running time to multiply two n-bit numbers by this method, this gives us a recurrence of

$$T(n) \quad = \quad 4T(n/2) + cn, \tag{1.2}$$

for some constant c. (The cn term reflects the time to perform the additions and shifts.) This recurrence solves to $O(n^2)$, so we do not seem to have made any progress. (In the next lecture we will go into the details of how to solve recurrences like this.)

However, we can take the formula in (1.1) and rewrite it as follows:

$$(2^n + 2^{n/2})AC - 2^{n/2}(A - B)(C - D) + (2^{n/2} + 1)BD. \tag{1.3}$$

It is not hard to see — you just need to multiply it out — that the formula in (1.3) is equivalent to the expression in (1.1). The new formula looks more complicated, but, it results in only *three* multiplications of size $n/2$, plus a constant number of shifts and additions. So, the resulting recurrence is

$$T(n) \quad = \quad 3T(n/2) + cn, \tag{1.4}$$

for some constant c. This recurrence solves to $O(n^{\log_2(3)}) \approx O(n^{1.585})$.

Is *this* method the fastest possible? Again it turns out that one can do better. In fact, Karp discovered a way to use the Fast Fourier Transform to multiply two n-bit numbers in time $O(n \log^2 n)$. Schönhage and Strassen in 1971 improved this to $O(n \log n \log \log n)$, which is, asymptotically, the fastest algorithm known. We will discuss the FFT later on in this course.

Actually, the kind of analysis we have been doing really is meaningful only for very large numbers. On a computer, if you are multiplying numbers that fit into the word size, you would do this in hardware that has gates working in parallel. So instead of looking at sequential running time, in this case we would want to examine the size and depth of the circuit used, for instance. This points out that, in fact, there are different kinds of specifications that can be important in different settings.

1.5 Matrix multiplication

It turns out the same basic divide-and-conquer approach of Karatsuba's algorithm can be used to speed up matrix multiplication as well. To multiply two n-by-n matrices in the usual way takes time $O(n^3)$. If one breaks down each n by n matrix into four $n/2$ by $n/2$ matrices, then the standard method can be thought of as performing eight $n/2$-by-$n/2$ multiplications and four additions as follows:

$$\begin{array}{|c|c|}\hline A & B \\ \hline C & D \\ \hline\end{array} \times \begin{array}{|c|c|}\hline E & F \\ \hline G & H \\ \hline\end{array} = \begin{array}{|c|c|}\hline AE+BG & AF+BH \\ \hline CE+DG & CF+DH \\ \hline\end{array}$$

Strassen noticed that, as in Karatsuba's algorithm, one can cleverly rearrange the computation to involve only *seven* $n/2$-by-$n/2$ multiplications (and 14 additions).[2] This results in a recurrence of

$$T(n) \quad = \quad 7T(n/2) + cn^2. \tag{1.5}$$

[2]In particular, the quantities that one computes recursively are $q_1 = (A + D)(E + H)$, $q_2 = D(G - E)$, $q_3 = (B - D)(G + H)$, $q_4 = (A + B)H$, $q_5 = (C + D)E$, $q_6 = A(F - H)$, and $q_7 = (C - A)(E + F)$. The upper-left quadrant of the solution is $q_1 + q_2 + q_3 - q_4$, the upper-right is $q_4 + q_6$, the lower-left is $q_2 + q_5$, and the lower right is $q_1 - q_5 + q_6 + q_7$. (feel free to check!)

(Adding two n-by-n matrices takes time $O(n^2)$.) This recurrence solves to a running time of just $O(n^{\log_2 7}) \approx O(n^{2.81})$ for Strassen's algorithm.

Matrix multiplication is especially important in scientific computation. Strassen's algorithm has more overhead than standard method, but it is the preferred method on many modern computers for even modestly large matrices. Asymptotically, the best matrix multiply algorithm known is by Coppersmith and Winograd and has time $O(n^{2.376})$, but is not practical. Nobody knows if it is possible to do better — the FFT approach doesn't seem to carry over.

Lecture 2

Asymptotic Analysis and Recurrences

2.1 Overview

In this lecture we discuss the notion of asymptotic analysis and introduce O, Ω, Θ, and o notation. We then turn to the topic of recurrences, discussing several methods for solving them. Recurrences will come up in many of the algorithms we study, so it is useful to get a good intuition for them right at the start. In particular, we focus on divide-and-conquer style recurrences, which will be the most common ones we will see.

Material in this lecture:

- Asymptotic notation: O, Ω, Θ, and o.
- Recurrences and how to solve them.
  - Solving by unrolling.
  - Solving with a guess and inductive proof.
  - Solving using a recursion tree.
  - A master formula.

2.2 Asymptotic analysis

When we consider an algorithm for some problem, in addition to knowing that it produces a correct solution, we will be especially interested in analyzing its running time. There are several aspects of running time that one could focus on. Our focus will be primarily on the question: "how does the running time *scale* with the size of the input?" This is called *asymptotic analysis*, and the idea is that we will ignore low-order terms and constant factors, focusing on the shape of the running time curve. We will typically use n to denote the size of the input, and $T(n)$ to denote the running time of our algorithm on an input of size n.

We begin by presenting some convenient definitions for discussing this kind of analysis.

Definition 2.1 *$T(n) \in O(f(n))$ if there exist constants $c, n_0 > 0$ such that $T(n) \leq cf(n)$ for all $n > n_0$.*

Informally we can view this as "$T(n)$ is proportional to $f(n)$, or better, as n gets large." For example, $3n^2 + 17 \in O(n^2)$ and $3n^2 + 17 \in O(n^3)$. This notation is especially useful in discussing upper bounds on algorithms: for instance, we saw last time that Karatsuba multiplication took time $O(n^{\log_2 3})$.

Notice that $O(f(n))$ is a set of functions. Nonetheless, it is common practice to write $T(n) = O(f(n))$ to mean that $T(n) \in O(f(n))$: especially in conversation, it is more natural to say "$T(n)$ is $O(f(n))$" than to say "$T(n)$ is in $O(f(n))$". We will typically use this common practice, reverting to the correct set notation when this practice would cause confusion.

Definition 2.2 *$T(n) \in \Omega(f(n))$ if there exist constants $c, n_0 > 0$ such that $T(n) \geq cf(n)$ for all $n > n_0$.*

Informally we can view this as "$T(n)$ is proportional to $f(n)$, or worse, as n gets large." For example, $3n^2 - 2n \in \Omega(n^2)$. This notation is especially useful for lower bounds. In Chapter 5, for instance, we will prove that any comparison-based sorting algorithm must take time $\Omega(n \log n)$ in the worst case (or even on average).

Definition 2.3 *$T(n) \in \Theta(f(n))$ if $T(n) \in O(f(n))$ and $T(n) \in \Omega(f(n))$.*

Informally we can view this as "$T(n)$ is proportional to $f(n)$ as n gets large."

Definition 2.4 *$T(n) \in o(f(n))$ if for all constants $c > 0$, there exists $n_0 > 0$ such that $T(n) < cf(n)$ for all $n > n_0$.*

For example, last time we saw that we could indeed multiply two n-bit numbers in time $o(n^2)$ by the Karatsuba algorithm. Very informally, O is like $\leq$, Ω is like $\geq$, Θ is like $=$, and o is like $<$. There is also a similar notation ω that corresponds to $>$.

In terms of computing whether or not $T(n)$ belongs to one of these sets with respect to $f(n)$, a convenient way is to compute the limit:

$$\lim_{n \to \infty} \frac{T(n)}{f(n)}. \tag{2.1}$$

If the limit exists, then we can make the following statements:

- If the limit is 0, then $T(n) = o(f(n)$ and $T(n) = O(f(n))$.
- If the limit is a number greater than 0 (e.g., 17) then $T(n) = \Theta(f(n))$ (and $T(n) = O(f(n))$ and $T(n) = \Omega(f(n)))$
- If the limit is infinity, then $T(n) = \omega(f(n))$ and $T(n) = \Omega(f(n))$.

For example, suppose $T(n) = 2n^3 + 100n^2 \log_2 n + 17$ and $f(n) = n^3$. The ratio of these is $2 + (100 \log_2 n)/n + 17/n^3$. In this limit, this goes to 2. Therefore, $T(n) = \Theta(f(n))$. Of course, it is possible that the limit doesn't exist — for instance if $T(n) = n(2 + \sin n)$ and $f(n) = n$ then the ratio oscillates between 1 and 3. In this case we would go back to the definitions to say that $T(n) = \Theta(n)$.

One convenient fact to know (which we just used in the paragraph above and you can prove by taking derivatives) is that for any constant k, $\lim_{n\to\infty}(\log n)^k/n = 0$. This implies, for instance, that $n \log n = o(n^{1.5})$ because $\lim_{n\to\infty}(n \log n)/n^{1.5} = \lim_{n\to\infty}(\log n)/\sqrt{n} = \lim_{n\to\infty}\sqrt{(\log n)^2/n} = 0$.

So, this notation gives us a language for talking about desired or achievable specifications. A typical use might be "we can prove that *any* algorithm for problem X must take $\Omega(n \log n)$ time in the worst case. My fancy algorithm takes time $O(n \log n)$. Therefore, my algorithm is asymptotically optimal."

2.3 Recurrences

We often are interested in algorithms expressed in a recursive way. When we analyze them, we get a recurrence: a description of the running time on an input of size n as a function of n *and* the running time on inputs of smaller sizes. Here are some examples:

Mergesort: To sort an array of size n, we sort the left half, sort right half, and then merge the two results. We can do the merge in linear time. So, if $T(n)$ is running time on input of size n, we end up with the recurrence $T(n) = 2T(n/2) + cn$.

Selection sort: In selection sort, we run through the array to find the smallest element. We put this in the leftmost position, and then recursively sort the remainder of the array. This gives us a recurrence $T(n) = cn + T(n-1)$.

Multiplication: Here we split each number into its left and right halves. We saw in the last lecture that the straightforward way to solve the subproblems gave us $T(n) = 4T(n/2) + cn$. However, rearranging terms in a clever way improved this to $T(n) = 3T(n/2) + cn$.

What about the base cases? In general, once the problem size gets down to a small constant, we can just use a brute force approach that takes some other constant amount of time. So, almost always we can say the base case is that $T(n) \leq c$ for all $n \leq n_0$, where n_0 is a constant we get to choose (like 17) and c is some other constant that depends on n_0.

What about the "integrality" issue? For instance, what if we want to use mergesort on an array with an odd number of elements — then the recurrence above is not technically correct. Luckily, this issue turns out almost never to matter, so we can ignore it. In the case of mergesort we can argue formally by using the fact that $T(n)$ is sandwiched between $T(n')$ and $T(n'')$ where n' is the next smaller power of 2 and n'' is the next larger power of 2, both of which differ by at most a constant factor from each other.

We now describe four methods for solving recurrences that are useful to know.

2.3.1 Solving by unrolling

Many times, the easiest way to solve a recurrence is to unroll it to get a summation. For example, unrolling the recurrence for selection sort gives us:

$$T(n) = cn + c(n-1) + c(n-2) + \ldots + c. \tag{2.2}$$

Since there are n terms and each one is at most cn, we can see that this summation is at most cn^2. Since the first $n/2$ terms are each at least $cn/2$, we can see that this summation is at least $(n/2)(cn/2) = cn^2/4$. So, it is $\Theta(n^2)$. Similarly, a recurrence $T(n) = n^5 + T(n-1)$ unrolls to:

$$T(n) = n^5 + (n-1)^5 + (n-2)^5 + \ldots + 1^5, \tag{2.3}$$

which solves to $\Theta(n^6)$ using the same style of reasoning as before. Another convenient way to look at many summations of this form is to see them as an approximation to an integral. E.g., in this last case, the sum is at most the integral of $f(x) = x^5$ evaluated at $x = n+1$.

2.3.2 Solving by guess and inductive proof

Another good way to solve recurrences is to make a guess and then prove the guess correct inductively. Or if we get into trouble proving our guess correct (e.g., because it was wrong), often this will give us clues as to a better guess. For example, say we have the recurrence

$$\begin{aligned} T(n) &\leq 7T(n/7) + n, & (2.4)\\ T(1) &= 1. & (2.5) \end{aligned}$$

We might first try a solution of $T(n) = cn$, where $c \geq 1$ to handle the base case. We would then assume it holds true inductively and plug in to get:

$$\begin{aligned} T(n) &\leq 7(cn/7) + n \\ &= cn + n \\ &= (c+1)n. \end{aligned}$$

Unfortunately, this isn't what we wanted: our multiplier "c" went up by 1 when n went up by a factor of 7. In other words, our multiplier is acting like $\log_7(n)$. So, let's make a new guess using a multiplier of this form — or $\log_7(7n)$ to get the base case of $n = 1$ right. So, we have a new guess of

$$T(n) \leq n \log_7(7n). \tag{2.6}$$

If we assume this holds true inductively for $n' < n$, then we get:

$$\begin{aligned} T(n) &\leq 7[(n/7)\log_7(n)] + n \\ &= n\log_7(n) + n \\ &= n\log_7(7n). & (2.7) \end{aligned}$$

So, we have verified our guess.

It is important in this type of proof to be careful. For instance, one could be lulled into thinking that our initial guess of cn was correct by reasoning "we assumed $T(n/7)$ was $\Theta(n/7)$ and got $T(n) = \Theta(n)$". The problem is that the constants changed (c turned into $c+1$) so they really weren't constants after all.

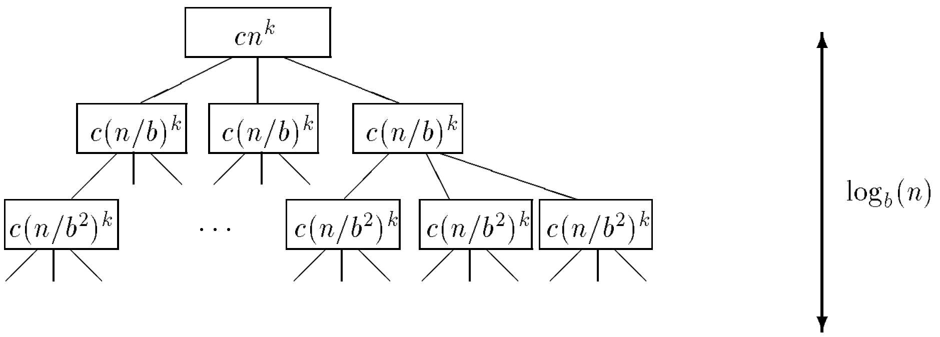
2.3.3 Recursion trees, stacking bricks, and a Master Formula

The final method we examine, which is especially good for divide-and-conquer style recurrences, is the use of a recursion tree. We will use this to method to produce a simple "master formula" that can be applied to many recurrences of this form.

Consider the following type of recurrence:

$$\begin{aligned} T(n) &\leq aT(n/b) + cn^k \\ T(1) &= c, \end{aligned} \tag{2.8}$$

for positive constants a, b, c, and k. This recurrence corresponds to the time spent by an algorithm that does cn^k work up front, and then divides the problem into a pieces of size n/b, solving each one recursively. For instance, mergesort, Karatsuba multiplication, and Strassen's algorithm all fit this mold. A *recursion tree* is just a tree that represents this process, where each node contains inside it the work done up front and then has one child for each recursive call. The leaves of the tree are the base cases of the recursion. A tree for the recurrence (2.9) is given below.[1]



To compute the result of the recurrence, we simply need to add up all the values in the tree. We can do this by adding them up level by level. The top level has value cn^k, the next level sums to $ca(n/b)^k$, the next level sums to $ca^2(n/b^2)^k$, and so on. The depth of the tree (the number of levels not including the root) is $\log_b(n)$. Therefore, we get a summation of:

$$cn^k\left[1 + a/b^k + (a/b^k)^2 + (a/b^k)^3 + ... + (a/b^k)^{\log_b n}\right] \tag{2.9}$$

We can now evaluate three cases (it might help to think of $k = 1$ if this gets confusing):

Case 1: $a < b^k$. In this case, the sum is a convergent series. For $r < 1$, the sum $1 + r + r^2 + \ldots = 1/(1-r)$. So, we can upper-bound formula (2.9) by $cn^k/(1 - a/b^k)$. Since a, b, and c are all constants, this solves to $\Theta(n^k)$.

Case 2: $a = b^k$. In this case, all terms in the summation (2.9) are equal to 1, so the result is $cn^k(\log_b n + 1) \in \Theta(n^k \log n)$.

Case 3: $a > b^k$. In this case, the last term of the summation dominates. We can see this by pulling it out, giving us:

$$cn^k(a/b^k)^{\log_b n}\left[(b^k/a)^{\log_b n} + \ldots + b^k/a + 1\right] \tag{2.10}$$

[1]This tree has branching factor a.

As in Case 1, the summation is a geometric series. Therefore, we have

$$T(n) \quad \in \quad \Theta\left(n^k (a/b^k)^{\log_b n}\right).$$

We can simplify this formula by noticing that $b^{k \log_b n} = n^k$, so we are left with

$$T(n) \quad \in \quad \Theta\left(a^{\log_b n}\right). \tag{2.11}$$

We can simplify this further by swapping the "a" and the "n" to get:

$$T(n) \quad \in \quad \Theta\left(n^{\log_b a}\right). \tag{2.12}$$

(Take $\log_b$ of (2.11) and (2.12) to convince yourself this is legal!)

Note that Case 3 is what we used for Karatsuba multiplication $(a = 3, b = 2, k = 1)$ and Strassen's algorithm $(a = 7, b = 2, k = 2)$.

Combining the three cases above gives us the following "master theorem".

Theorem 2.1 *The recurrence*

$$\begin{aligned} T(n) &\leq aT(n/b) + cn^k \\ T(1) &= c, \end{aligned}$$

where a, b, c, and k are all constants, solves to:

$$\begin{aligned} T(n) &\in \Theta(n^k) \textit{ if } a < b^k \\ T(n) &\in \Theta(n^k \log n) \textit{ if } a = b^k \\ T(n) &\in \Theta(n^{\log_b a}) \textit{ if } a > b^k \end{aligned}$$

A nice intuitive way to think of the computation above is to think of each node in the recursion tree as a brick of height 1 and width equal to the value inside it. Our goal is now to compute the area of the stack. Depending on whether we are in Case 1, 2, or 3, the picture then looks like one of the following:

```
+-----------------+   +---------------+      +------------+
 \_______________/    |_______________|     /______________\
  \______|______/     |_______|_______|    /_______|________\
   \__|__|__|__/      |___|___|___|___|   /____|___|___|_____\
```

In the first case, the area is dominated by the top brick; in the second case, all levels provide an equal contribution, and in the last case, the area is dominated by the bottom level.

Lecture 3

Probabilistic Analysis and Randomized Quicksort

3.1 Overview

In this lecture we begin by discussing the difference between worst-case and average-case behavior, and introduce randomized (probabilistic) algorithms and the notion of worst-case expected time bounds. We make this concrete with a discussion of a randomized version of the Quicksort sorting algorithm, which we prove has worst-case expected running time $O(n \log n)$. In the process, we discuss basic probabilistic concepts such as events, random variables, and linearity of expectation.

3.2 Worst-case, average-case, and randomized algorithms

The last lecture discussed the notions of O, Ω, and Θ bounds, and how to compute them using recurrences. We begin this lecture with a different issue: worst-case versus average case bounds.

Say I is some input and $T(I)$ is running time of our algorithm on input I. We can then define:

$$\begin{aligned} T_{\text{worstcase}}(n) &= \max_{\text{inputs } I \text{ of size } n} T(I) \\ T_{\text{averagecase}}(n) &= \operatorname*{avg}_{\text{inputs } I \text{ of size } n} T(I) \end{aligned}$$

For instance, Mergesort has both worst-case and average-case time $\Theta(n \log n)$. It doesn't really depend on the input at all. On the other hand, for some algorithms, the running time depends critically on the input. One example is Quicksort.

Quicksort: Given array of some length n,

1. Pick an element p of the array as the pivot (or halt if the array has size 0 or 1).
2. Split the array into sub-arrays LESS, EQUAL, and GREATER by comparing each element to the pivot. (LESS has all elements less than p, EQUAL has all elements equal to p, and GREATER has all elements greater than p).
3. recursively sort LESS and GREATER.

The Quicksort algorithm given above is not yet fully specified because we have not stated how we will pick the pivot element p. For the first version of the algorithm, we always choose the leftmost element.

Basic-Quicksort: Run the Quicksort algorithm as given above, always choosing the leftmost element in the array as the pivot.

What is worst-case running time of Basic-Quicksort? We can see that if the array is already sorted, then in Step 2, all the elements (except p) will go into the GREATER bucket. Furthermore, since the GREATER array is in sorted order,[1] this process will continue recursively, resulting in time $\Omega(n^2)$. We can also see that the running time is $O(n^2)$ on any array of n elements because Step 1 can be executed at most n times, and Step 2 takes at most n steps to perform. Thus, the worst-case running time is $\Theta(n^2)$.

On the other hand, it turns out (and we will prove) that the average-case running time for Basic-Quicksort (averaging over all different initial orderings of the n elements in the array) is $O(n \log n)$. So, Basic-Quicksort has good average case performance but not good worst-case performance.

The fact that this algorithm works well on *most* inputs may be small consolation if the inputs we are faced with are the bad ones (e.g., if our lists are nearly sorted already). One way we can try to get around this problem is to add randomization into the algorithm itself:

Randomized-Quicksort: Run the Quicksort algorithm as given above, each time picking a *random* element in the array as the pivot.

We will prove that for *any* given array input array I of n elements, the expected time of this algorithm $\mathbf{E}[T(I)]$ is $O(n \log n)$. This is called a Worst-case Expected-Time bound. Notice that this is better than an average-case bound because we are no longer assuming any special properties of the input. E.g., it could be that in our desired application, the input arrays tend to be mostly sorted or in some special order, and this does not affect our bound because it is a *worst-case* bound with respect to the input. It is a little peculiar: making the algorithm probabilistic gives us *more* control over the running time.

To prove these bounds, we first detour into the basics of probabilistic analysis.

3.3 The Basics of Probabilistic Analysis (I)

Consider rolling two dice and observing the results. There are 36 possible outcomes: it could be that the first die comes up 1 and the second comes up 2, or that the first comes up 2 and the second comes up 1, and so on. Each of these outcomes has probability 1/36 (assuming these are fair dice). If we care about some quantity such as "what is the probability the sum of the dice equals 7?", we can compute that by adding up the probabilities of all the outcomes satisfying this condition (there are six of them, for a total probability of 1/6).

In the language of probability theory, any probabilistic setting is defined by a *sample space* S and a *probability measure* p. The points of the sample space are called *elementary events*. E.g., in

[1] Technically, this depends on how the partitioning step is implemented, but will be the case for any reasonable implementation.

our case, the elementary events are the 36 possible outcomes for the pair of dice. In a discrete probability distribution (as opposed to a continuous one), the probability measure is a function $p(e)$ over elementary events e such that $p(e) \geq 0$ for all $e \in S$, and $\sum_{e \in S} p(e) = 1$.

An *event* is a subset of the sample space. For instance, one event we might care about is the event that the first die comes up 1. Another is the event that the two dice sum to 7. The probability of an event is just the sum of the probabilities of the elementary events contained inside it (again, this is just for discrete distributions[2]).

A *random variable* is a function from elementary events to integers or reals. For instance, another way we can talk formally about these dice is to define the random variable X_1 representing the result of the first die, X_2 representing the result of the second die, and $X = X_1 + X_2$ representing the sum of the two. We could then ask: what is the probability that $X = 7$?

One property of a random variable we often care about is its *expectation*. For a discrete random variable X over sample space S, the expected value of X is:

$$\mathbf{E}[X] = \sum_{e \in S} \Pr(e) X(e). \tag{3.1}$$

In other words, the expectation of a random variable X is just its average value over S, where each elementary event e is weighted according to its probability. For instance, if we roll a single die and look at the outcome, the expected value is 3.5, because all six elementary events have equal probability. Often one groups together the elementary events according to the different values of the random variable and rewrites the definition like this:

$$\mathbf{E}[X] = \sum_{a} \Pr(X = a) a. \tag{3.2}$$

More generally, for any partition of the probability space into disjoint events $A_1, A_2, \ldots$, we can rewrite the expectation of random variable X as:

$$\mathbf{E}[X] = \sum_{i} \sum_{e \in A_i} \Pr(e) X(e) = \sum_{a} \Pr(A_i) \mathbf{E}[X|A_i], \tag{3.3}$$

where $\mathbf{E}[X|A_i]$ is the expected value of X given A_i, defined to be $\frac{1}{Pr(A_i)} \sum_{e \in A_i} \Pr(e) X(e)$. The formula (3.3) will be useful when we analyze Quicksort. In particular, note that the running time of Randomized Quicksort is a random variable, and our goal is to analyze its expectation.

3.4 Analysis of Randomized Quicksort

For simplicity, let us assume no two elements in the array are equal — when we are done with the analysis, it will be easy to look back and see that allowing equal keys could only improve performance. We now prove the following theorem.

Theorem 3.1 *The expected number of comparisons made by randomized quicksort on an array of size n is at most $2n \ln n + O(n)$.*

Proof: When we pick the random pivot, we might split the array into a LESS of size 0 and a GREATER of size $n - 1$, or into a LESS of size 1 and a GREATER of size $n - 2$, and so on, up

[2]For a continuous distribution, the probability would be an integral.

to a LESS of size $n-1$ and a GREATER of size 0. All of these are equally likely with probability $1/n$ each. Therefore, we can write a recurrence for the expected number of comparisons $T(n)$ as follows:

$$T(n) \quad = \quad n + \frac{1}{n}\sum_{i=0}^{n-1}(T(i) + T(n-i-1)). \tag{3.4}$$

Formally, we are using the expression for Expectation given in (3.3) here (the n different possible splits are the events A_i).[3]

We can now rewrite (3.4) as:

$$T(n) \quad = \quad n + \frac{2}{n}\sum_{i=0}^{n-1}T(i) \tag{3.5}$$

Now, we can solve this by the "guess and prove inductively" method. In order to do this, we will need to evaluate a summation of whatever it is we guess, and the easiest way to do this is to upper-bound the sum by an integral. In particular if $f(x)$ is an increasing function, then

$$\sum_{i=0}^{n-1} f(i) \quad \leq \quad \int_1^n f(x)dx,$$

which we can see by drawing a graph of f and recalling that an integral represents the "area under the curve". So, let's make a guess that is easy to integrate.

In particular, let's guess that for $x < n$, $T(x) \leq x + 2x\ln x$. This guess satisfies the base case, and its integral is $x^2 \ln x$. Arguing by induction we now have:

$$\begin{aligned} T(n) \quad &\leq \quad n + \frac{2}{n}\sum_{i=0}^{n-1}(i + 2i\ln i) \\ &\leq \quad n + \frac{2}{n}\left(n^2 \ln n\right) \\ &= \quad n + 2n\ln n. \quad \blacksquare \end{aligned}$$

The textbook has a less sneaky way of doing this (but it takes more work!) in Chapter 3.

In terms of the number of comparisons it makes, Randomized Quicksort is equivalent to randomly shuffling the input and then handing it off to Basic Quicksort. So, we have also proven that Basic Quicksort has $O(n \log n)$ *average-case* running time.

3.5 The Basics of Probabilistic Analysis (II)

Another important fact about expected values is Linearity of Expectation: for any two random variables X and Y, $\mathbf{E}[X+Y] = \mathbf{E}[X] + \mathbf{E}[Y]$. This fact is incredibly important for analysis of algorithms because it allows us to analyze a complicated random variable by writing it as a sum of simple random variables and then separately analyzing these simple RVs. Let's first prove this fact and then see how it can be used.

[3] Technically we are using a weak form of Linearity of Expectation too, to say that the expected time *given* one of these events can be written as the sum of two expectations. See the next section.

Theorem 3.2 (Linearity of Expectation) *For any two random variables X and Y,* $\mathbf{E}[X+Y] = \mathbf{E}[X] + \mathbf{E}[Y]$.

Proof (for discrete RVs): This follows directly from the definition as given in (3.1).

$$\mathbf{E}[X+Y] = \sum_{e \in S} \Pr(e)(X(e)+Y(e)) = \sum_{e \in S} \Pr(e)X(e) + \sum_{e \in S} \Pr(e)Y(e) = \mathbf{E}[X] + \mathbf{E}[Y]. \quad \blacksquare$$

3.5.1 Example 1: Card shuffling

Suppose we unwrap a fresh deck of cards and shuffle it until the cards are completely random. How many cards do we expect to be in the same position as they were at the start? To solve this, let's think formally about what we are asking. We are looking for the expected value of a random variable X denoting the number of cards that end in the same position as they started. We can write X as a sum of random variables X_i, one for each card, where $X_i = 1$ if the ith card ends in position i and $X_i = 0$ otherwise. These X_i are easy to analyze: $\Pr(X_i = 1) = 1/n$ where n is the number of cards. $\Pr(x_i = 1)$ is also $\mathbf{E}[X_i]$. Now we use linearity of expectation:

$$\mathbf{E}[X] = \mathbf{E}[X_1 + \ldots + X_n] = \mathbf{E}[X_1] + \ldots + \mathbf{E}[X_n] = 1.$$

So, this is interesting: no matter how large a deck we are considering, the expected number of cards that end in the same position as they started is 1.

3.5.2 Example 2: Random Walk Stock Market

Suppose there is a stock with the property that each day, it has a 50:50 chance of going either up or down by \$1, unless the stock is at 0 in which case it stays there. You start with \$m. Each day you can buy or sell as much as you want, until at the end of the year all your money is converted back into cash. What is the best strategy for maximizing your expected gain?

The answer is that no matter what strategy you choose, your expected gain by the end of the year is 0 (i.e., you expect to end with the same amount of money as you started). Let's prove that this is the case.

Define random variable X_t to be the gain of our algorithm on day t. Let X be the overall gain at the end of the year. Then,

$$X = X_1 + \ldots + X_{365}.$$

Notice that the X_t's can be highly dependent, based on our strategy. For instance, if our strategy is to pull all our money out of the stock market the moment that our wealth exceeds \$m, then X_2 depends strongly on the outcome of X_1. Nonetheless, by linearity of expectation,

$$\mathbf{E}[X] = \mathbf{E}[X_1] + \ldots + \mathbf{E}[X_{365}].$$

Finally, no matter how many shares s of stock we hold at time t, $\mathbf{E}[X_t|s] = 0$. So, $\mathbf{E}[X_t] = 0$ (for all t) and therefore $\mathbf{E}[X] = 0$.

3.6 A different way to analyze Randomized Quicksort

Here is a neat alternative way to analyze randomized quicksort that is basically just like what we did with the previous examples. This gives us even a slightly better bound than the last method.

Theorem 3.3 *For Randomized Quicksort, the expected number of comparisons is at most* $2n \ln n$.

Proof: As before, let's assume no two elements in the array are equal since it is the worst case and will make our notation simpler. As with the previous examples, the trick is to write the quantity we care about (the total number of comparisons) as a sum of simpler random variables, and then just analyze the simpler ones.

Define random variable X_{ij} to be 1 if the algorithm *does* compare the ith smallest and jth smallest elements in the course of sorting, and 0 if it does not. Let X denote the total number of comparisons made by the algorithm. Since we never compare the same pair of elements twice, we have

$$X = \sum_{i=1}^{n} \sum_{j=i+1}^{n} X_{ij},$$

and therefore,

$$\mathbf{E}[X] = \sum_{i=1}^{n} \sum_{j=i+1}^{n} \mathbf{E}[X_{ij}].$$

Let us consider one of these X_{ij}'s for $i < j$. Denote the ith smallest element in the array by e_i and the jth smallest element by e_j, and conceptually imagine lining up the elements in sorted order. If the pivot we choose is between e_i and e_j then these two end up in different buckets and we will never compare them to each other. If the pivot we choose *is* either e_i or e_j then we *do* compare them. If the pivot is less than e_i or greater than e_j then both e_i and e_j end up in the same bucket and we have to pick another pivot. So, we can think of this like a dart game: we throw a dart at random into the array: if we hit e_i or e_j then X_{ij} becomes 1, if we hit between e_i and e_j then X_{ij} becomes 0, and otherwise we throw another dart. Therefore, the probability that $X_{ij} = 1$ is exactly $2/(j - i + 1)$. So, we have:

$$\begin{aligned}
\mathbf{E}[X] &= \sum_{i=1}^{n} \sum_{j=i+1}^{n} 2/(j-i+1) \\
&= 2\sum_{i=1}^{n} \sum_{d=2}^{n-i+1} 1/d \\
&< 2n \sum_{d=2}^{n} 1/d.
\end{aligned}$$

The quantity $1 + 1/2 + 1/3 + \ldots + 1/n$, denoted H_n, is called the "nth harmonic number" and is in the range $[\ln n, 1 + \ln n]$ (this can be seen by considering the integral of $f(x) = 1/x$). Therefore,

$$\mathbf{E}[X] < 2n(H_n - 1) \leq 2n \ln n. \quad \blacksquare$$

3.7 Yet another way to analyze it: run it backwards

Here's another way to analyze quicksort — run the algorithm backwards. Actually, to do this analysis, it is better to think of a version of Quicksort that instead of being recursive, at each step it picks a random bucket in proportion to its size to work on next. The reason this version is nice is that if you imagine watching the pivots get chosen and where they would be on a sorted array, they are coming in completely at random. Looking at the algorithm run backwards, at a generic point in time, we have k pivots (producing $k+1$ buckets) and we "undo" one of our pivot choices at random, merging the two adjoining buckets. [The tricky part here is showing that this is really a legitimate way of looking at Quicksort in reverse.] The cost for an undo operation is the sum of the sizes of the two buckets joined (since this was the number of comparisons needed to split them). Notice that for each undo operation, if you sum the costs over all of the k possible pivot choices, you count each bucket twice (or once if it is the leftmost or rightmost) and get a total of $< 2n$. Since we are picking one of these k possibilities at random, the *expected* cost is $2n/k$. So, we get $\sum_k 2n/k = 2nH_n$.

Lecture 4

Deterministic and Randomized Selection: finding the median in linear time

4.1 Overview

Given an unsorted array, how quickly can one find the median element? Can one do it more quickly than by sorting? This was an open question for some time, solved affirmatively in 1972. In this lecture we describe two linear-time algorithms for this problem (more generally, the problem of finding the kth smallest out of n elements): one randomized and one deterministic.

4.2 The problem and a randomized solution

A related problem to sorting is the problem of finding the kth smallest element in an unsorted array. (Let's say all elements are distinct to avoid the question of what we mean by the kth smallest when we have equalities). One way to solve this problem is to sort and then output the kth element. Is there something faster – a linear-time algorithm? The answer is yes. We will explore both a simple randomized solution and a more complicated deterministic one.

The idea for the randomized algorithm is to notice that in Randomized-Quicksort, after the partitioning step we can tell which subarray has the item we are looking for, just by looking at their sizes. So, we only need to recursively examine one subarray, not two. This algorithm is often called Randomized-Select, or QuickSelect.

QuickSelect: Given array A of size n and integer $k \leq n$,

1. Pick a pivot element p at random from A.
2. Split A into subarrays LESS and GREATER by comparing each element to p as in Quicksort. While we are at it, count the number L of elements going in to LESS.
3. (a) If $L = k - 1$, then output p.

   (b) If $L > k - 1$, output QuickSelect(LESS, k).

(c) If $L < k-1$, output QuickSelect(GREATER, $k-L-1$)

Theorem 4.1 *The expected number of comparisons for QuickSelect is $O(n)$.*

Proof: Let $T(n,k)$ denote the expected time to find the kth smallest in an array of size n, and let $T(n) = \max_k T(n,k)$. We will show that $T(n) \leq 4n$.

First of all, it takes $n-1$ comparisons to split into the array into two pieces in Step 2. As with Quicksort, these pieces might have size 0 and $n-1$, or 1 and $n-2$, or 2 and $n-3$, and so on up to $n-1$ and 0. Each of these is equally likely. Now, the piece we recurse on will depend on k, but since we are only giving an upper bound, we can be pessimistic and imagine that we always recurse on the larger piece. Therefore we have:

$$\begin{aligned} T(n) &\leq (n-1) + \frac{2}{n}\sum_{i=n/2}^{n-1} T(i) \\ &= (n-1) + \text{avg}\left[T(n/2), \ldots, T(n-1)\right]. \end{aligned}$$

We can solve this using the "guess and check" method. Assume inductively that $T(i) \leq 4i$ for $i < n$. Then,

$$\begin{aligned} T(n) &\leq (n-1) + \text{avg}\left[4(n/2), 4(n/2+1), \ldots, 4(n-1)\right] \\ &\leq (n-1) + 4(3n/4) \\ &< 4n, \end{aligned}$$

and we have verified our guess. ■

One way to think intuitively about this bound is that if we split a candy bar at random into two pieces, then the expected size of the larger piece is 3/4 of the bar. If the size of the larger subarray after our partition was always 3/4 of the array, then we would have a recurrence $T(n) \leq (n-1) + T(3n/4)$ which solves to $T(n) < 4n$. Now, this is not quite the case for our algorithm because 3/4 is only an expected value, but because the answer is linear in n, the average of the $T(i)$'s turns out to be the same as T(average of the i's).

4.3 A deterministic linear-time algorithm

What about a deterministic linear-time algorithm? For a long time it was thought this was impossible – that there was no method faster than first sorting the array. In the process of trying to prove this claim it was discovered that this thinking was incorrect, and in 1972 a deterministic linear time algorithm was developed.

The idea of the algorithm is that one would like to pick a pivot deterministically in a way that produces a good split. Ideally, we would like the pivot to be the median element so that the two sides are the same size. But, this is the same problem we are trying to solve in the first place! So, instead, we will give ourselves leeway by allowing the pivot to be any element that is "roughly" in the middle: at least 3/10 of the array below the pivot and at least 3/10 of the array above. The algorithm is as follows:

DeterministicSelect: Given array A of size n and integer $k \leq n$,

1. Group the array into $n/5$ groups of size 5 and find the median of each group. (For simplicity, we will ignore integrality issues.)
2. Recursively, find the true median of the medians. Call this p.
3. Use p as a pivot to split the array into subarrays LESS and GREATER.
4. Recurse on the appropriate piece.

Theorem 4.2 *DeterministicSelect makes $O(n)$ comparisons to find the kth smallest in an array of size n.*

Proof: Let $T(n, k)$ denote the worst-case time to find the kth smallest out of n, and $T(n) = \max_k T(n, k)$ as before.

Step 1 takes time $O(n)$, since it takes just constant time to find the median of 5 elements. Step 2 takes time at most $T(n/5)$. Step 3 again takes time $O(n)$. Now, we claim that at least $3/10$ of the array is $\leq p$, and at least $3/10$ of the array is $\geq p$. Assuming for the moment that this claim is true, Step 4 takes time at most $T(7n/10)$, and we have the recurrence:

$$T(n) \quad \leq \quad cn + T(n/5) + T(7n/10), \tag{4.1}$$

for some constant c. Before solving this recurrence, lets prove the claim we made that the pivot will be roughly near the middle of the array. So, the question is: how bad can the median of medians be?

Let's first do an example. Suppose the array has 15 elements and breaks down into three groups of 5 like this:

$$\{1, 2, 3, 10, 11\}, \quad \{4, 5, 6, 12, 13\}, \quad \{7, 8, 9, 14, 15\}.$$

In this case, the medians are 3, 6, and 9, and the median of the medians p is 6. There are five elements less than p and nine elements greater.

In general, what is the worst case? If there are $g = n/5$ groups, then we know that in at least $\lceil g/2 \rceil$ of them (those groups whose median is $\leq p$) at least three of the five elements are $\leq p$. Therefore, the total number of elements $\leq p$ is at least $3\lceil g/2 \rceil \geq 3n/10$. Similarly, the total number of elements $\geq p$ is also at least $3\lceil g/2 \rceil \geq 3n/10$.

Now, finally, let's solve the recurrence. We have been solving a lot of recurrences by the "guess and check" method, which works here too, but how could we just stare at this and *know* that the answer is linear in n? One way to do that is to consider the "stack of bricks" view of the recursion tree discussed in Lecture 2.

In particular, let's build the recursion tree for the recurrence (4.1), making each node as wide as the quantity inside it:

```
+--------------------------------------+
|                 cn                   |
+---------+-----------------------+----+
|  cn/5   |       7cn/10          |    <- total 9cn/10
+--+----+-+-----+-------------+---+
|  |    |       |             |        <- total (81/100)cn
+--+----+-------+-------------+
...                                   ...
```

Notice that even if this stack-of-bricks continues downward forever, the total sum is at most

$$cn(1 + (9/10) + (9/10)^2 + (9/10)^3 + \ldots),$$

which is at most $10cn$. This proves the theorem. ∎

Notice that in our analysis of the recurrence (4.1) the key property we used was that $n/5 + 7n/10 < n$. More generally, we see here that if we have a problem of size n that we can solve by performing recursive calls on pieces whose total size is at most $(1-\epsilon)n$ for some constant $\epsilon > 0$ (plus some additional $O(n)$ work), then the total time spent will be just linear in n. This gives us a nice extension to our "Master theorem" from Lecture 2.

Theorem 4.3 *For constants c and $a_1, \ldots, a_k$ such that $a_1 + \ldots a_k < 1$, the recurrence*

$$T(n) \quad \leq \quad T(a_1 n) + T(a_2 n) + \ldots T(a_k n) + cn$$

solves to $T(n) = \Theta(n)$.

Lecture 5

Comparison-based Lower Bounds for Sorting

5.1 Overview

In this lecture we discuss the notion of *lower bounds*, in particular for the problem of sorting. We show that any deterministic comparison-based sorting algorithm must take $\Omega(n \log n)$ time to sort an array of n elements in the worst case. We then extend this result to average case performance, and to randomized algorithms. In the process, we introduce the 2-player game view of algorithm design and analysis.

5.2 Sorting lower bounds

So far we have been focusing on the question: "given some problem X, can we construct an algorithm that runs in $O(f(n))$ time on inputs of size n?" This is often called an upper bound problem because we are determining an upper bound on the inherent difficulty of problem X. In this lecture we examine the "lower bound problem." Here, the goal is to prove that *any* algorithm must take time $\Omega(f(n))$ time to solve the problem.

Today we consider the class of comparison-based sorting algorithms. These are sorting algorithms that only operate on the input array by comparing pairs of elements, and moving elements around based on the results of these comparisons. For instance, Quicksort, Mergesort, and Insertionsort are all comparison-based sorting algorithms. What we will show is the following theorem.

Theorem 5.1 *Any deterministic comparison-based sorting algorithm must perform $\Omega(n \log n)$ comparisons to sort n elements in the worst case. Specifically, for any deterministic comparison-based sorting algorithm $\mathcal{A}$, for all $n \geq 2$ there exists an input I of size n such that $\mathcal{A}$ makes at least $\log_2(n!) = \Omega(n \log n)$ comparisons to sort I.*

To prove this theorem, we cannot assume the sorting algorithm is going to necessarily choose a pivot as in Quicksort, or split the input as in Mergesort — we need to somehow analyze *any possible* (comparison-based) algorithm that might exist. The way we will do this is by showing that in order to sort its input, the sorting algorithm is implicitly playing a game of 20 questions with the input, trying to figure out in what the order its elements are being given.

Proof: Let's say the input array contains the elements $a_1, a_2, \ldots, a_n$ in some unknown order (assume the a_i are all distinct). There are $n!$ different orders these elements could be in. Now, suppose that two different initial orderings of these numbers I_1, I_2, are consistent with all the comparisons the sorting algorithm has made so far. In this case, we claim that the sorting algorithm cannot yet be done. The reason it cannot yet be done is that the output of a comparison-based sorting algorithm has to be a permutation of the input (the sorter can't make up new numbers to output since all it knows about the inputs a_i is how they relate to each other). Furthermore, whatever permutation of the input it produces as output cannot be correct for both I_1 and I_2.

So, the sorting algorithm needs at least implicitly to have pinned down which ordering of $\{a_1, \ldots, a_n\}$ was given in the input. There are $n!$ different possible input orderings, and each comparison is a binary yes/no question that removes some of these orderings from consideration. If the answer to this question is always the one consistent with the majority of the orderings still remaining, the algorithm will need to make $\log_2(n!)$ questions to narrow down the input to a single possibility. We can then solve:

$$\begin{aligned} \log_2(n!) &= \log_2(n) + \log_2(n-1) + \ldots + \log_2(2) \\ &= \Omega(n \log n). \quad \blacksquare \end{aligned}$$

Let's do an example with $n = 3$. In this case, there are six possible input orderings:

$$\{123\}, \{132\}, \{213\}, \{231\}, \{312\}, \{321\}.$$

Suppose the sorting algorithm first compares $A[0]$ with $A[1]$. If the answer is that $A[1] > A[0]$ then we have narrowed down the input to the three possibilities:

$$\{123\}, \{132\}, \{231\}.$$

Suppose the next comparison is between $A[1]$ and $A[2]$. In this case, the most popular answer is that $A[1] > A[2]$, which removes just one ordering, leaving us with:

$$\{132\}, \{231\}.$$

It now takes one more comparison to finally isolate the input ordering.

Notice that our proof is like a game of 20 questions in which the responder doesn't actually decide what he is thinking of until there is only one option left. This is legitimate because we just need to show that there is *some* input that would cause the algorithm to take a long time. In other words, since the sorting algorithm is deterministic, we can take that final remaining option and then re-run the algorithm on that specific input, and the algorithm will make the same exact sequence of operations.

5.3 Average-case lower bounds

In fact, we can generalize the above theorem to show that any comparison-based sorting algorithm must take $\Omega(n \log n)$ time *on average*, not just in the worst case.

Theorem 5.2 *For any deterministic comparison-based sorting algorithm, the Average-Case number of comparisons (the number of comparisons on average on a randomly chosen input permutation) is at least* $\lfloor \log_2(n!) \rfloor$.

Proof: Let's build out the entire decision tree: the tree we get by looking at all possible series of answers that one might get from some ordering of the input. By the previous argument, each leaf of this tree corresponds to a single input permutation (we can't have two permutations at a leaf, else the algorithm would not be finished). The depth of the leaf is the number of comparisons performed by the sorting algorithm on that input.

If the tree is completely balanced, then each leaf is at depth $\lceil \log_2(n!) \rceil$ or $\lfloor \log_2(n!) \rfloor$ and we are done.[1] To prove the theorem, we just need to show that out of all binary trees on a given number of leaves, the one that minimizes their average depth is a completely balanced tree. This is not too hard to see: given some unbalanced tree, we take two sibling leaves at largest depth and move them to be children of the leaf of smallest depth. Since the difference between the largest depth and the smallest depth is at least 2 (otherwise the tree would be balanced), this operation reduces the average depth of the leaves. Specifically, if the smaller depth is d and the larger depth is D, we have removed two leaves of depth D and one of depth d, and we have added two leaves of depth $d+1$ and one of depth $D-1$. ■

In fact, if we are a bit more clever in the proof, we can get rid of the floor in the bound.

5.4 Lower bounds for randomized algorithms

Theorem 5.3 *The above bound holds for randomized algorithms too.*

Proof: We can think of a randomized algorithm as a probability distribution over deterministic algorithms. E.g., with some probability it makes sequence of choices s_1, with some probability it makes sequence of choices s_2, and so on. The expected running time of the randomized algorithm on some input I is just

$$\sum_{\text{choice sequences } s} \Pr(s)(\text{Running time on } I \text{ given } s).$$

If you recall the definition of expectation, the running time of the randomized algorithm is a random variable and the sequences s correspond to the elementary events.

So, the expected running time of the randomized algorithm is just an average over deterministic algorithms. Since each deterministic algorithm has average-case running time at least $\lfloor \log_2(n!) \rfloor$, their average does too, completing the proof.

A more formal way to make this statement is that the average-case running time of the randomized algorithm is

$$\begin{aligned}
\operatorname*{avg}_{\text{inputs } I} \sum_{\text{choices } s} \Pr(s)(\text{Running time on } I \text{ given } s) &= \sum_s \Pr(s) \operatorname*{avg}_I(\text{Running time on } I \text{ given } s) \\
&\geq \sum_s \Pr(s) \lfloor \log_2(n!) \rfloor \\
&= \lfloor \log_2(n!) \rfloor. \quad ■
\end{aligned}$$

[1] Everything would be easier if we could somehow assume $n!$ was a power of 2....

One way to think of the kinds of bounds we have been proving is to think of a matrix with one row for every possible deterministic comparison-based sorting algorithm (there could be a lot of rows!) and one column for every possible permutation of the n inputs (there are a lot of columns too). Entry (i, j) in this matrix contains the running time of algorithm i on input j. The worst-case deterministic lower bound tells us that for each row i there exists a column j_i such that the entry (i, j_i) is large. The average-case deterministic lower bound tells us that for each row i, the average of the elements in the row is large. The randomized lower bound says "well, since the above statement holds for every row, it must also hold for any weighted average of the rows." In the language of game-theory, one could think of this as a two-player game (much like rock-paper-scissors) between an "algorithm player" and an "input player". Each player chooses their strategy (a row for the algorithm-player and a column for the input-player) and the entry in the matrix is how much money the algorithm-player has to pay the input player. We have shown that there is a strategy for the input player (namely, pick a column at random) that guarantees it an expected gain of $\Omega(n \log n)$ no matter what strategy the algorithm-player chooses.