## Contents

1 Introduction to Algorithms, and the Divide and Conquer Paradigm 3  
1.1 Overview ................................................................. 3  
1.2 Introduction ............................................................. 3  
1.3 On guarantees and specifications ...................................... 4  
1.4 An example: Karatsuba Multiplication ............................... 5  
1.5 Strassen's algorithm for matrix multiplication ..................... 7  
1.6 The Divide and Conquer Paradigm .................................... 7  

2 Concrete models and tight upper/lower bounds 9  
2.1 Overview ................................................................. 9  
2.2 Terminology and setup .................................................. 9  
2.3 Sorting in the comparison model ..................................... 10  
2.4 Sorting in the exchange model ....................................... 11  
2.5 The comparison model revisited ......................................... 12  
  2.5.1 Finding the maximum of \( n \) elements ....................... 12  
  2.5.2 Finding the second-largest of \( n \) elements ................... 13  
2.6 Query models, and the evasiveness of connectivity ................ 14  

A Asymptotic Analysis and Recurrences 17  
A.1 Overview ................................................................. 17  
A.2 Asymptotic analysis ...................................................... 17  
A.3 Recurrences .............................................................. 19  
  A.3.1 Solving by unrolling ................................................. 20  
  A.3.2 Solving by guess and inductive proof ........................... 20  
  A.3.3 Recursion trees, stacking bricks, and a Master Formula ........ 21  

B Probability basics 25
Lecture 1

Introduction to Algorithms, and the Divide and Conquer Paradigm

1.1 Overview

The goal 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 examples of some problems that are easy to relate to (multiplying two numbers and multiplying two matrices) in which the straightforward approach is surprisingly not the fastest one. These examples illustrate the power of a general approach called the Divide-and-Conquer paradigm.

Material in this lecture:

- What is the study of Algorithms all about?
- Why do we care about specifications and proving guarantees?
- The Karatsuba multiplication algorithm.
- Strassen’s matrix multiplication algorithm.
- The Divide and Conquer paradigm.
- Solving recurrences.

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 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 any input of size \( n \). This course is about the whole package: the design of efficient algorithms, \textit{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 the intellectual tools for designing and analyzing your own algorithms for problems you need to solve in the future. Some tools we will discuss are Divide-and-Conquer, Greedy Algorithms, Dynamic Programming, Network Flows, Linear Programming, and Randomization. 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 \textit{not} to be able to achieve? In this course, we will delve a bit into complexity theory, focusing on the somewhat surprising notion of NP-completeness. We will additionally discuss some approaches for dealing with NP-complete problems, including the notion of approximation algorithms.

Other problems may be challenging because they require decisions to be made without having full information, and we will discuss some paradigms for problems of this nature.

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 why we at least want to know that our algorithm is correct, 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 — you can probably think of more reasons too:

- **Composability.** A guarantee on running time gives a “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 given some amount of resources.

- **Designing better algorithms.** Analyzing the asymptotic running time of algorithms is 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.

It is often helpful when thinking about algorithms to imagine a game where one player is the algorithm designer, trying to come up with a good algorithm for the problem, and its opponent (the “adversary”) is trying to come up with an input that will cause the algorithm to run slowly. An algorithm with good worst-case guarantees is one that performs well no matter what input the adversary chooses. We will return to this view in a more formal way when we discuss game theory.

### 1.4 An example: Karatsuba Multiplication

One thing 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, \( 41 \times 42 \) (or, in binary, \( 101001 \times 101010 \)). 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:

\[
\begin{align*}
101001 & = 41 \\
x & 101010 = 42 \\
\hline
1010010 \\
101001 \\
+ 101001 \\
\hline
1101011010 = 1722
\end{align*}
\]

More formally, we scan the second number right to left, and every time we see a 1, we add a copy of the first number, shifted by the appropriate number of bits, to our total. Each addition takes \( O(n) \) time, and we perform at most \( n \) additions, which means the total running time here is \( O(n^2) \). 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.

\(^1\)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.
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:

\[
X = 2^{n/2} A + B \\
Y = 2^{n/2} C + D
\]

We can now write the product of \( X \) and \( Y \) as

\[
XY = 2^n AC + 2^{n/2} BC + 2^{n/2} AD + BD.
\] (1.1)

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

\[
T(n) = 4T(n/2) + cn,
\] (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 tutorial we will
review how to solve recurrences like this — see Appendix A.)

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

\[
(2^n - 2^{n/2})AC + 2^{n/2}(A + B)(C + D) + (1 - 2^{n/2})BD.
\] (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

\[
T(n) = 3T(n/2) + c'n,
\] (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 discov-
ered 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 was until recently the
asymptotically fastest algorithm known.\(^2\)

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.

\(^2\)Fürer in 2007 improved this by replacing the \( \log \log n \) term with \( 2^{O(\log^* n)} \), where \( \log^* n \) is a very slowly growing
function.
1.5 Strassen’s algorithm for 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 be clear, we will now be considering a computational model where individual elements in the matrices are viewed as “small” and can be added or multiplied in constant time. In particular, to multiply two \( n \times n \) matrices in the usual way (we take the \( i \)th row of the first matrix and compute its dot-product with the \( j \)th column of the second matrix in order to produce the entry \( ij \) in the output) takes time \( O(n^3) \). If one breaks down each \( n \times n \) matrix into four \( n/2 \times n/2 \) matrices, then the standard method can be thought of as performing eight \( n/2 \times n/2 \) multiplications and four additions as follows:

\[
\begin{bmatrix}
A & B \\
C & D \\
\end{bmatrix} \times \begin{bmatrix}
E & F \\
G & H \\
\end{bmatrix} = \begin{bmatrix}
AE + BG & AF + BH \\
CE + DG & CF + DH \\
\end{bmatrix}
\]

Strassen noticed that, as in Karatsuba’s algorithm, one can cleverly rearrange the computation to involve only seven \( n/2 \times n/2 \) multiplications (and 14 additions). Since adding two \( n \times n \) matrices takes time \( O(n^2) \), this results in a recurrence of

\[
T(n) = 7T(n/2) + cn^2.
\]

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 has running time \( O(n^{2.37}) \), but is not practical. Nobody knows if it is possible to do better — the FFT approach doesn’t seem to carry over.

1.6 The Divide and Conquer Paradigm

Karatsuba’s algorithm and Strassen’s algorithm are both examples of a “divide and conquer” paradigm: break the problem into subproblems, solve each subproblem independently, and then argue that the solutions to the subproblems can be stitched together into a solution to the original problem. Merge-sort is probably the most iconic divide and conquer algorithm. Divide and conquer is a basic but important algorithmic technique, always worth thinking about when you have a new algorithmic problem to solve.

---

3In 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!)

4Strassen has said that when coming up with his algorithm, he first tried to solve the problem mod 2. Solving mod 2 makes the problem easier because you only need to keep track of the parity of each entry, and in particular, addition is the same as subtraction. One he figured out the solution mod 2, he was then able to make it work in general.
Lecture 2

Concrete models and tight upper/lower bounds

2.1 Overview

In this lecture, we will examine some simple, concrete models of computation, each with a precise definition of what counts as a step, and try to get tight upper and lower bounds for a number of problems. Specific models and problems examined in this lecture include:

- The number of comparisons needed to sort an array.
- The number of exchanges needed to sort an array.
- The number of comparisons needed to find the largest and second-largest elements in an array.
- The number of probes into a graph needed to determine if the graph is connected (the evasiveness of connectivity).

2.2 Terminology and setup

In this lecture, we will look at (worst-case) upper and lower bounds for a number of problems in several different concrete models. Each model will specify exactly what operations may be performed on the input, and how much they cost. Typically, each model will have some operations that cost 1 step (like performing a comparison, or swapping a pair of elements), some that are free, and some that are not allowed at all.

By an upper bound of $f(n)$ for some problem, we mean that there exists an algorithm that takes at most $f(n)$ steps on any input of size $n$. By a lower bound of $g(n)$, we mean that for any algorithm there exists an input on which it takes at least $g(n)$ steps. The reason for this terminology is that if we think of our goal as being to understand the “true complexity” of each problem, measured in terms of the best possible worst-case guarantee achievable by any algorithm, then an upper bound of $f(n)$ and lower bound of $g(n)$ means that the true complexity is somewhere between $g(n)$ and $f(n)$.
2.3 Sorting in the comparison model

One natural model for examining problems like sorting is what is known as the comparison model.

Definition 2.1 In the comparison model, we have an input consisting of \( n \) items (typically in some initial order). An algorithm may compare two items (asking is \( a_i > a_j \)?) at a cost of 1. Moving the items around is free. No other operations on the items are allowed (such as using them as indices, XORing them, etc).

For the problem of sorting in the comparison model, the input is an array \( a = [a_1, a_2, \ldots, a_n] \) and the output is a permutation of the input \( \pi(a) = [a_{\pi(1)}, a_{\pi(2)}, \ldots, a_{\pi(n)}] \) in which the elements are in increasing order. We begin this lecture by showing the following lower bound for comparison-based sorting.

Theorem 2.1 Any deterministic comparison-based sorting algorithm must perform at least \( \lg(n!) \) comparisons to sort \( n \) elements in the worst case.\(^1\) Specifically, for any deterministic comparison-based sorting algorithm \( A \), for all \( n \geq 2 \) there exists an input \( I \) of size \( n \) such that \( A \) makes at least \( \lg(n!) = \Omega(n \log n) \) 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. We now present the proof, which uses a very nice information-theoretic argument. (This proof is deceptively short: it’s worth thinking through each line and each assertion.)

Proof: First of all, for a deterministic algorithm, the permutation it outputs (e.g., \( [a_3, a_1, a_4, a_2] \)) is solely a function of the sequence of answers it receives to its comparisons. In particular, any two different input arrays that yield the same sequence of answers will cause the same permutation to be produced as output. So, if an algorithm always made at most \( k \) comparisons, then there would be at most \( 2^k \) different permutations of the input array that it can possibly output, since each comparison has a YES or NO answer. This implies that if \( k < \lg(n!) \), so \( 2^k < n! \), then there would be some permutation \( \pi \) of the input array that it can’t output. Let’s assume for contradiction that such a permutation exists. All that remains is to show that for any such permutation \( \pi \), there is some input array for which \( \pi \) is the only correct answer. This is easy. For example, the permutation \( [a_3, a_1, a_4, a_2] \) is the only correct answer for sorting the input \( [2, 4, 1, 3] \), and more generally, permutation \( \pi \) is the only correct answer for the input \( [\pi^{-1}(1), \pi^{-1}(2), \ldots, \pi^{-1}(n)] \). Thus we have our desired contradiction.\(\blacksquare\)

The above is often called an “information theoretic” argument because we are in essence saying that we need at least \( \lg_2(n!) \) bits of information about the input before we can correctly decide what output we need to produce. More generally, if we have some problem with \( M \) different outputs the algorithm might be required to produce, then we have a worst-case lower bound of \( \lg M \).

\(^1\)As is common in CS, we will use “\( \lg \)” to mean “\( \log_2 \)”.
What does $\lg(n!)$ look like? We have: $\lg(n!) = \lg(n) + \lg(n-1) + \lg(n-2) + \ldots + \lg(1) < n\lg(n) = O(n \log n)$ and $\lg(n!) = \lg(n) + \lg(n-1) + \lg(n-2) + \ldots + \lg(1) > (n/2)\lg(n/2) = \Omega(n \log n)$. So, $\lg(n!) = \Theta(n \log n)$.

However, since today’s theme is tight bounds, let’s be a little more precise. We can in particular use the fact that $n! \in [(n/e, n^n]$ to get:

$$n \lg n - n \log e < \lg(n!) < n \lg n$$
$$n \lg n - 1.443n < \lg(n!) < n \lg n.$$

Since 1.433$n$ is a low-order term, sometimes people will write this fact this as: $\lg(n!) = (n \lg n)(1 - o(1))$, meaning that the ratio between $\lg(n!)$ and $n \lg n$ goes to 1 as $n$ goes to infinity.

Assume $n$ is a power of 2 — in fact, let’s assume this for the entire rest of today’s lecture. Can you think of an algorithm that makes at most $n \lg n$ comparisons, and so is tight in the leading term? In fact, there are several algorithms, including:

**Binary insertion sort** If we perform insertion-sort, using binary search to insert each new element, then the number of comparisons made is at most $\sum_{k=2}^{n} \lceil \lg k \rceil \leq n \lg n$. Note that insertion-sort spends a lot in moving items in the array to make room for each new element, and so is not especially efficient if we count movement cost as well, but it does well in terms of comparisons.

**Mergesort** Merging two lists of $n/2$ elements each requires at most $n-1$ comparisons. So, unrolling the recurrence we get $(n-1)+2(n/2-1)+4(n/4-1)+\ldots+n/2(2-1) = n \lg n - (n-1) < n \lg n$.

### 2.4 Sorting in the exchange model

Consider a shelf containing $n$ unordered books to be arranged alphabetically. In each step, we can swap any pair of books we like. How many swaps do we need to sort all the books? Formally, we are considering the problem of sorting in the exchange model.

**Definition 2.2** In the exchange model, an input consists of an array of $n$ items, and the only operation allowed on the items is to swap a pair of them at a cost of 1 step. All other (planning) work is free: in particular, the items can be examined and compared to each other at no cost.

**Question:** how many exchanges are necessary (lower bound) and sufficient (upper bound) in the exchange model to sort an array of $n$ items in the worst case?

**Claim 2.2 (Upper bound)** $n-1$ exchanges is sufficient.

**Proof:** For this we just need to give an algorithm. For instance, consider the algorithm that in step 1 puts the smallest item in location 1, swapping it with whatever was originally there. Then in step 2 it swaps the second-smallest item with whatever is currently in location 2, and so on (if in step $k$, the $k$th-smallest item is already in the correct position then we just do a no-op). No step ever undoes any of the previous work, so after $n-1$ steps, the first $n-1$ items are in the correct position. This means the $n$th item must be in the correct position too. ■
But are \( n - 1 \) exchanges necessary in the worst-case? If \( n \) is even, and no book is in its correct location, then \( n/2 \) exchanges are clearly necessary to “touch” all books. But can we show a better lower bound than that?

**Claim 2.3 (Lower bound)** *In fact, \( n - 1 \) exchanges are necessary, in the worst case.*

**Proof:** Here is how we can see it. Create a graph in which a directed edge \((i, j)\) means that that the book in location \( i \) must end up at location \( j \). For instance, consider the example in Figure 2.1. Note that this is a special kind of directed graph: it is a permutation — a set of cycles. In particular, every book points to some location, perhaps its own location, and every location is pointed to by exactly one book. Now consider the following points:

1. What is the effect of exchanging any two elements (books) that are in the same cycle?
   
   **Answer:** Suppose the graph had edges \((i_1, j_1)\) and \((i_2, j_2)\) and we swap the elements in locations \( i_1 \) and \( i_2 \). Then this causes those two edges to be replaced by edges \((i_2, j_1)\) and \((i_1, j_2)\) because now it is the element in location \( i_2 \) that needs to go to \( j_1 \) and the element in \( i_1 \) that needs to go to \( j_2 \). This means that if \( i_1 \) and \( i_2 \) were in the same cycle, that cycle now becomes two disjoint cycles.

2. What is the effect of exchanging any two elements that are in different cycles?

   **Answer:** If we swap elements \( i_1 \) and \( i_2 \) that are in different cycles, then the same argument as above shows that this merges those two cycles into one cycle.

3. How many cycles are in the final sorted array?

   **Answer:** The final sorted array has \( n \) cycles.

Putting the above 3 points together, suppose we begin with an array consisting of a single cycle, such as \([n, 1, 2, 3, 4, \ldots, n-1]\). Each operation at best increases the number of cycles by 1 and in the end we need to have \( n \) cycles. So, this input requires \( n - 1 \) operations.

### 2.5 The comparison model revisited

#### 2.5.1 Finding the maximum of \( n \) elements

How many comparisons are necessary and sufficient to find the maximum of \( n \) elements, in the comparison model of computation?

**Claim 2.4 (Upper bound)** *\( n - 1 \) comparisons are sufficient to find the maximum of \( n \) elements.*

**Proof:** Just scan left to right, keeping track of the largest element so far. This makes at most \( n - 1 \) comparisons.

Now, let’s try for a lower bound. One simple lower bound is that since there are \( n \) possible answers for the location of the maximum element, our previous argument gives a lower bound of \( \log n \). But clearly this is not at all tight. In fact, we can give a better lower bound of \( n - 1 \).
Claim 2.5 (Lower bound) \( n - 1 \) comparisons are needed in the worst-case to find the maximum of \( n \) elements.

**Proof:** Suppose some algorithm \( A \) claims to find the maximum of \( n \) elements using less than \( n - 1 \) comparisons. Consider an arbitrary input of \( n \) distinct elements, and construct a graph in which we join two elements by an edge if they are compared by \( A \). If fewer than \( n - 1 \) comparisons are made, then this graph must have at least two components. Suppose now that algorithm \( A \) outputs some element \( u \) as the maximum, where \( u \) is in some component \( C_1 \). In that case, pick a different component \( C_2 \) and add a large positive number (e.g., the value of \( u \)) to every element in \( C_2 \). This process does not change the result of any comparison made by \( A \), so on this new set of elements, algorithm \( A \) would still output \( u \). Yet this now ensures that \( u \) is not the maximum, so \( A \) must be incorrect. ■

Since the upper and lower bounds are equal, these bounds are tight.

### 2.5.2 Finding the second-largest of \( n \) elements

How many comparisons are necessary (lower bound) and sufficient (upper bound) to find the second largest of \( n \) elements? Again, let us assume that all elements are distinct.

Claim 2.6 (Lower bound) \( n - 1 \) comparisons are needed in the worst-case to find the second-largest of \( n \) elements.

**Proof:** The same argument used in the lower bound for finding the maximum still holds. ■

Let us now work on finding an upper bound. Here is a simple one to start with.

Claim 2.7 (Upper bound #1) \( 2n - 3 \) comparisons are sufficient to find the second-largest of \( n \) elements.
**Proof:** Just find the largest using \( n - 1 \) comparisons, and then the largest of the remainder using \( n - 2 \) comparisons, for a total of \( 2n - 3 \) comparisons. □

We now have a gap: \( n - 1 \) versus \( 2n - 3 \). It is not a huge gap: both are \( \Theta(n) \), but remember today’s theme is tight bounds. So, which do you think is closer to the truth? It turns out, we can reduce the upper bound quite a bit:

**Claim 2.8 (Upper bound #2)** \( n + \lg n - 2 \) comparisons are sufficient to find the second-largest of \( n \) elements.

**Proof:** As a first step, let’s find the maximum element using \( n - 1 \) comparisons, but in a tennis-tournament or playoff structure. That is, we group elements into pairs, finding the maximum in each pair, and recurse on the maxima. E.g.,

![Diagram of a tennis tournament structure](image)

Now, given just what we know from comparisons so far, what can we say about possible locations for the second-highest number (i.e., the second-best player)? The answer is that the second-best must have been directly compared to the best, and lost.\(^2\) This means there are only \( \lg n \) possibilities for the second-highest number, and we can find the maximum of them making only \( \lg(n) - 1 \) more comparisons. □

At this point, we have a lower bound of \( n - 1 \) and an upper bound of \( n + \lg(n) - 2 \), so they are nearly tight. It turns out that, in fact, the lower bound can be improved to exactly meet the upper bound.\(^3\)

### 2.6 Query models, and the evasiveness of connectivity

To finish with something totally different, let’s look at the query complexity of determining if a graph is connected. Assume we are given the adjacency matrix \( G \) for some \( n \)-node graph. That is, \( G[i,j] = 1 \) if there is an edge between \( i \) and \( j \), and \( G[i,j] = 0 \) otherwise. We consider a model in which we can query any element of the matrix \( G \) in 1 step. All other computation is free. That is, imagine the graph matrix has values written on little slips of paper, face down. In one step we

\(^2\)Apparently the first person to have pointed this out was Charles Dodgson (better known as Lewis Carroll!), writing about the proper way to award prizes in lawn tennis tournaments.

\(^3\)First shown by Kislitsyn (1964).
can turn over any slip of paper. How many slips of paper do we need to turn over to tell if $G$ is connected?

**Claim 2.9 (Easy upper bound)** $n(n-1)/2$ queries are sufficient to determine if $G$ is connected.

**Proof:** This just corresponds to querying every pair $(i,j)$. Once we have done that, we know the entire graph and can just compute for free to see if it is connected. ■

Interestingly, it turns out the simple upper-bound of querying every edge is a lower bound too. Because of this, connectivity is called an “evasive” property of graphs.

**Theorem 2.10 (Lower bound)** $n(n - 1)/2$ queries are necessary to determine connectivity in the worst case.

**Proof:** Here is the strategy for the adversary: when the algorithm asks us to flip over a slip of paper, we return the answer 0 unless that would force the graph to be disconnected, in which case we answer 1. (It is not important to the argument, but we can figure this out by imagining that all un-turned slips of paper are 1 and seeing if that graph is connected.) Now, here is the key claim:

*Claim:* we maintain the invariant that for any un-asked pair $(u,v)$, the graph revealed so far has no path from $u$ to $v$.

*Proof of claim:* If there was, consider the last edge $(u',v')$ revealed on that path. We could have answered 0 for that and kept the same connectivity in the graph by having an edge $(u,v)$. So, that contradicts the definition of our adversary strategy.

Now, to finish the proof: Suppose an algorithm halts without examining every pair. Consider some unasked pair $(u,v)$. If the algorithm says “connected,” we reveal all-zeros for the remaining unasked edges and then there is no path from $u$ to $v$ (by the key claim) so the algorithm is wrong. If the algorithm says “disconnected,” we reveal all-ones for the remaining edges, and the algorithm is wrong by definition of our adversary strategy. So, the algorithm must ask for all edges. ■
Appendix A

Asymptotic Analysis and Recurrences

A.1 Overview

We discuss the notion of asymptotic analysis and introduce $O$, $\Omega$, $\Theta$, 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 are the most common ones we will see.

Material in this lecture:

- Asymptotic notation: $O$, $\Omega$, $\Theta$, 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.

A.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 instead 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 performing this kind of analysis.
Definition A.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 A.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 Lecture 3, for instance, we will prove that any comparison-based sorting algorithm must take time $\Omega(n \log n)$ in the worst case.

Definition A.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 A.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 $\leq$, $\Omega$ is like $\geq$, $\Theta$ is like $=$, and $o$ is like $<$. There is also a similar notation $\omega$ 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:

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

18
For example, suppose $T(n) = 2n^3 + 100n^2 \log_2 n + 17$ and $f(n) = n^3$. The ratio of these is $2 + \frac{100 \log_2 n}{n} + \frac{17}{n^3}$. 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 $n \log n = \Theta(n^{1.5})$ 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.”

### A.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 $T(n)$ denotes the running time on an 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.
A.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:

$$T(n) = cn + c(n-1) + c(n-2) + \ldots + c.$$  \hfill (A.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:

$$T(n) = n^5 + (n-1)^5 + (n-2)^5 + \ldots + 1^5,$$  \hfill (A.3)

which solves to $\Theta(n^6)$ using the same style of reasoning as before. In particular, there are $n$ terms each of which is at most $n^5$ so the sum is at most $n^6$, and the top $n/2$ terms are each at least $(n/2)^5$ so the sum is at least $(n/2)^6$. Another convenient way to look at many summations of this form is to see them as approximations to an integral. E.g., in this last case, the sum is at least the integral of $f(x) = x^5$ evaluated from 0 to $n$, and at most the integral of $f(x) = x^5$ evaluated from 1 to $n+1$. So, the sum lies in the range $[\frac{1}{6}n^6, \frac{1}{5}(n+1)^6]$.

A.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

$$T(n) = 7T(n/7) + n,$$  \hfill (A.4)

$$T(1) = 0.$$  \hfill (A.5)

We might first try a solution of $T(n) \leq cn$ for some $c > 0$. We would then assume it holds true inductively for $n' < n$ (the base case is obviously true) and plug in to our recurrence (using $n' = n/7$) to get:

$$T(n) \leq 7(cn/7) + n = cn + n = (c+1)n.$$  \hfill (A.6)

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. So, we have a new guess of

$$T(n) \leq n \log_7(n).$$  \hfill (A.7)

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

$$T(n) \leq 7[(n/7) \log_7(n/7)] + n = n \log_7(n/7) + n = n \log_7(n) - n + n = n \log_7(n).$$  

20
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 constant after all!

### A.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 method to produce a simple “master formula” that can be applied to many recurrences of this form.

Consider the following type of recurrence:

\[
T(n) = aT(n/b) + cn^k \tag{A.8}
\]

\[
T(1) = c,
\]

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 (A.8) is given below.\(^1\)

```
  cn^k
 /       \
/         \n(c(n/b)^k) \(c(n/b)^k\) \(c(n/b)^k\)
 / \     / \        /  \
(\(c(n/b^2)^k\) \(c(n/b^2)^k\) \(c(n/b^2)^k\) \(c(n/b^2)^k\) \(c(n/b^2)^k\)
    \       \       \       \       \
        log_b(n)
```

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:

\[
\sum_{i=0}^{\log_b n} (a/b)^i \cdot (c/\ldots)(n/b^{\log_b n}) = cn^k \left[ 1 + (a/b)^1 + (a/b)^2 + \ldots + (a/b)^{\log_b n} \right] \tag{A.9}
\]

To help us understand this, let’s define \( r = \frac{a}{b} \). Notice that \( r \) is a constant, since \( a, b, \) and \( k \) are constants. For instance, for Strassen’s algorithm \( r = 7/2^2 \), and for mergesort \( r = 2/2 = 1 \). Using our definition of \( r \), our summation simplifies to:

\[
\sum_{i=0}^{\log_b n} (a/b)^i \cdot (c/\ldots)(n/b^{\log_b n}) = cn^k \left[ 1 + r + r^2 + \ldots + r^{\log_b n} \right] \tag{A.10}
\]

\(^1\)This tree has branching factor \( a \).
We can now evaluate three cases:

Case 1: $r < 1$. In this case, the sum is a convergent series. Even if we imagine the series going to infinity, we still get that the sum $1 + r + r^2 + \ldots = 1/(1 - r)$. So, we can upper-bound formula (A.9) by $cn^k/(1 - r)$, and lower bound it by just the first term $cn^k$. Since $r$ and $c$ are constants, this solves to $\Theta(n^k)$.

Case 2: $r = 1$. In this case, all terms in the summation (A.9) are equal to 1, so the result is $cn^k(\log_b n + 1) \in \Theta(n^k \log n)$.

Case 3: $r > 1$. In this case, the last term of the summation dominates. We can see this by pulling it out, giving us:

$$cn^kr^{\log_b n} \left[(1/r)^{\log_b n} + \ldots + 1/r + 1\right] \tag{A.11}$$

Since $1/r < 1$, we can now use the same reasoning as in Case 1: the summation is at most $1/(1 - 1/r)$ which is a constant. Therefore, we have

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

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

$$T(n) \in \Theta\left(n^{\log_b a}\right). \tag{A.12}$$

We can simplify this further using $a^{\log_b n} = b^{(\log_b a)(\log_b n)} = n^{\log_b a}$ to get:

$$T(n) \in \Theta\left(n^{\log_b a}\right). \tag{A.13}$$

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 A.1** The recurrence

$$T(n) = aT(n/b) + cn^k$$

$$T(1) = c,$$

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

$$T(n) \in \Theta(n^k) \text{ if } a < b^k$$

$$T(n) \in \Theta(n^k \log n) \text{ if } a = b^k$$

$$T(n) \in \Theta(n^{\log_b a}) \text{ if } a > b^k$$

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.

More generally, this approach can be used to solve recurrences with uneven sizes as well. For example, the following theorem immediately follows from considering the “stack of bricks” view, since each level is a constant-factor smaller than the level above it.

**Theorem A.2** For constants $c$ and $a_1, \ldots, a_k$ such that $a_1 + \ldots a_k < 1$, the recurrence

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

solves to $T(n) = O(n)$.  

Appendix B

Probability basics

B.1 The notion of randomized algorithms

As we have discussed previously, we are interested in how the running time of an algorithm scales with the size of the input. In addition, we will usually be interested in worst-case running time, meaning the worst-case over all inputs of a given size. That is, if $I$ is some input and $T(I)$ is running time of our algorithm on input $I$, then $T(n) = \max\{T(I) : \text{size}(I) = n\}$. One can also look at notions of average-case running time, where we are concerned with our performance on “typical” inputs $I$. However, one difficulty with average-case bounds is that it is often unclear in advance what typical inputs for some problem will really look like, and furthermore this gets more difficult if our algorithm is being used as a subroutine inside some larger computation. In particular, if we have a bound on the worst-case running time of an algorithm for some problem $A$, it means that we can now consider solving other problems $B$ by somehow converting instances of $B$ to instances of problem $A$. We will see many examples of this later when we talk about network flow and linear programming as well as in our discussions of NP-completeness.

On the other hand, there are algorithms that have a large gap between their performance “on average” and their performance in the worst case. Sometimes, in this case we can improve the worst-case performance by actually adding randomization into the algorithm itself. One classic example of this is the Quicksort sorting algorithm.

**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, let’s 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 $O(n \log n)$. This fact 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 $E[T(I)]$ 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.

### B.2 The Basics of Probabilistic Analysis

Consider rolling two dice and observing the results. We call this an *experiment*, and it has 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). Suppose 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, such a probabilistic setting is defined by a *sample space* $S$ and a *probability measure* $p$. The points of the sample space are the possible outcomes of the experiment and are called *elementary events*. E.g., in 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 $\sum_{e \in S} p(e) = 1$. We will also use $\Pr(e)$ interchangeably with $p(e)$.

---

\(^1\)Technically, this depends on how the partitioning step is implemented, but will be the case for any reasonable implementation.
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\textsuperscript{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:

$$E[X] = \sum_{e \in S} \text{Pr}(e)X(e). \quad (B.1)$$

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:

$$E[X] = \sum_a \text{Pr}(X = a)a. \quad (B.2)$$

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:

$$E[X] = \sum_i \sum_{e \in A_i} \Pr(e)X(e) = \sum_i \Pr(A_i)E[X|A_i], \quad (B.3)$$

where $E[X|A_i]$ 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 (B.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.

### B.2.1 Linearity of Expectation

An important fact about expected values is Linearity of Expectation: for any two random variables $X$ and $Y$, $E[X + Y] = E[X] + E[Y]$. 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.


**Proof** (for discrete RVs): This follows directly from the definition as given in (B.1).

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

\textsuperscript{2}For a continuous distribution, the probability would be an integral over a density function.
B.2.2 Example: 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 $i$th 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 $E[X_i]$. Now we use linearity of expectation:

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

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.

B.3 Analysis of Randomized Quicksort

We now give two methods for analyzing randomized quicksort. The first is more intuitive but the details are messier. The second is a neat tricky way using the power of linearity of expectation: this will be a bit less intuitive but the details come out nicer.

B.3.1 Method 1

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 B.2** The expected number of comparisons made by randomized quicksort on an array of size $n$ is at most $2n \ln n$.

**Proof:** First of all, when we pick the pivot, we perform $n - 1$ comparisons (comparing all other elements to it) in order to split the array. Now, depending on the 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 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:

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

(B.4)

Formally, we are using the expression for Expectation given in (B.3), where the $n$ different possible splits are the events $A_i$.\(^3\) We can rewrite equation (B.4) by regrouping and getting rid of $T(0)$:

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

(B.5)

\(^3\)In addition, we are using Linearity of Expectation to say that the expected time given one of these events can be written as the sum of two expectations.
Now, we can solve this by the “guess and prove inductively” method. In order to do this, we first need a good guess. Intuitively, most pivots should split their array “roughly” in the middle, which suggests a guess of the form $cn \ln n$ for some constant $c$. Once we’ve made our guess, we will need to evaluate the resulting summation. One of the easiest ways of doing this is to upper-bound the sum by an integral. In particular if $f(x)$ is an increasing function, then

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

which we can see by drawing a graph of $f$ and recalling that an integral represents the “area under the curve”. In our case, we will be using the fact that $\int (cx \ln x)dx = (c/2)x^2 \ln x - cx^2/4$.

So, let’s now do the analysis. We are guessing that $T(i) \leq ci \ln i$ for $i \leq n - 1$. This guess works for the base case $T(1) = 0$ (if there is only one element, then there are no comparisons). Arguing by induction we have:

$$T(n) \leq (n - 1) + \frac{2}{n} \sum_{i=1}^{n-1} (ci \ln i)$$

$$\leq (n - 1) + \frac{2}{n} \int_1^{n} (cx \ln x)dx$$

$$\leq (n - 1) + \frac{2}{n} \left((c/2)n^2 \ln n - cn^2/4 + c/4\right)$$

$$\leq cn \ln n, \text{ for } c = 2. \quad \square$$

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.

### B.3.2 Method 2

Here is a neat alternative way to analyze randomized quicksort that is very similar to how we analyzed the card-shuffling example.

**Alternative proof (Theorem B.2):** 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. The trick will be 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 $i$th smallest and $j$th 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,

$$\mathbb{E}[X] = \sum_{i=1}^{n} \sum_{j=i+1}^{n} \mathbb{E}[X_{ij}].$$
Let us consider one of these $X_{ij}$’s for $i < j$. Denote the $i$th smallest element in the array by $e_i$ and the $j$th 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. At each step, the probability that $X_{ij} = 1$ conditioned on the event that the game ends in that step is exactly $2/(j-i+1)$. Therefore, overall, the probability that $X_{ij} = 1$ is $2/(j-i+1)$.

In other words, for a given element $i$, it is compared to element $i+1$ with probability 1, to element $i+2$ with probability $2/3$, to element $i+3$ with probability $2/4$, to element $i+4$ with probability $2/5$ and so on. So, we have:

$$E[X] = \sum_{i=1}^{n} 2 \left( \frac{1}{2} + \frac{1}{3} + \frac{1}{4} + \frac{1}{5} + \ldots + \frac{1}{n-i+1} \right).$$

The quantity $1 + 1/2 + 1/3 + \ldots + 1/n$, denoted $H_n$, is called the “$n$th 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,

$$E[X] < 2n(H_n - 1) \leq 2n \ln n.$$

### B.4 Further Discussion

#### B.4.1 More linearity of expectation: a 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,

$$E[X] = E[X_1] + \ldots + E[X_{365}].$$
Finally, no matter how many shares $s$ of stock we hold at time $t$, $E[X_t|s] = 0$. So, using (B.3), whatever probability distribution over $s$ is induced by our strategy, $E[X_t] = 0$. Since this holds for every $t$, we have $E[X] = 0$.

This analysis can be generalized to the case of gambling in a “fair casino”. In a fair casino, there are a number of games with different kinds of payoffs, but each one has the property that your expected gain for playing it is zero. E.g., there might be a game where with probability $99/100$ you lose but with probability $1/100$ you win $99$ times your bet. In that case, no matter what strategy you use for which game to play and how much to bet, the expected amount of money you will have at the end of the day is the same as the amount you had going in.

### B.4.2 Yet another way to analyze quicksort: 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 $\sum_k 2n/k = 2nH_n$. 

31