15-451 Algorithms
Lectures 1-10

Author: **Avrim Blum**

Instructors: **Avrim Blum**
Manuel Blum

Department of Computer Science
Carnegie Mellon University

August 23, 2011

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 the fastest 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 handouts)
- 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.

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, *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 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 notion of NP-completeness. We will (may) also spend some time on cryptography. Cryptography is interesting from the point of view of algorithm design because it uses a problem that's assumed to be intrinsically hard to solve 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 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 randomized algorithms and lower bounds.

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, 41×42 (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 = 41
x       101010 = 42
-------------
       1010010
     101001
+  101001
-------------
   11010111010 = 1722
```

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.

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:

$X = 2^{n/2}A + B$ | A | B |

$Y = 2^{n/2}C + D$ | C | D |

[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.

We can now write the product of X and Y as

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

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

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

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:

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

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

$$T(n) = 3T(n/2) + c'n, \quad (1.4)$$

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 was until very recently the asymptotically fastest algorithm known.[2] 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 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-by-n matrices in the usual way (we take the

[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 discussed in Lecture 14. It remains unknown whether eliminating it completely and achieving running time $O(n \log n)$ is possible.

ith row of the first matrix and compute its dot-product with the jth 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 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).[3] Since adding two n-by-n matrices takes time $O(n^2)$, this results in a recurrence of

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

This recurrence solves to a running time of just $O(n^{\log_2 7}) \approx O(n^{2.81})$ for Strassen's algorithm.[4]

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.

[3] 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!)

[4] According to Manuel Blum, Strassen 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

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 are 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 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 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 $T(n)$ 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.

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:

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

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. 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 $[\frac{1}{6}n^6, \frac{1}{6}(n+1)^6]$.

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

$$T(n) = 7T(n/7) + n, \tag{2.4}$$
$$T(1) = 0. \tag{2.5}$$

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:

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

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. So, we have a new guess of

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

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

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

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!

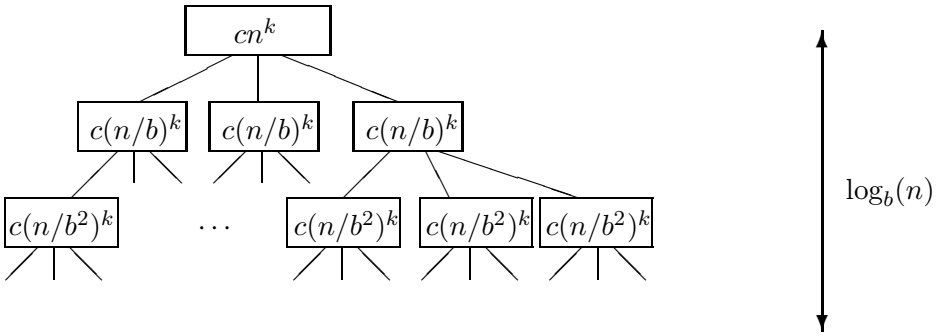
2.3.3 Recursion trees, stacking bricks, and a Master Formula

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:

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

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 (2.8) 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:

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

To help us understand this, let's define $r = a/b^k$. 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:

$$cn^k \left[1 + r + r^2 + r^3 + ... + r^{\log_b n}\right] \tag{2.10}$$

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 (2.9) 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 $\Theta(n^k)$.

[1] This tree has branching factor a.

Case 2: $r = 1$. 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: $r > 1$. In this case, the last term of the summation dominates. We can see this by pulling it out, giving us:

$$cn^k r^{\log_b n} \left[(1/r)^{\log_b n} + \ldots + 1/r + 1\right] \tag{2.11}$$

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

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

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

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

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

$$T(n) \quad \in \quad \Theta\left(n^{\log_b a}\right). \tag{2.13}$$

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".

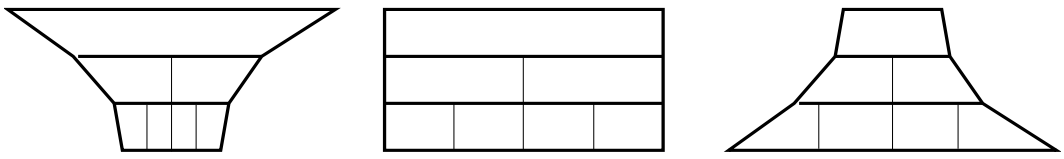
Theorem 2.1 *The recurrence*

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

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

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

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 introducing 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 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 $T(n) = \max\{T(I)\}_{\text{inputs } I \text{ of size } n}$. 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 $\mathbf{E}[T(I)]$ 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

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

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

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 $p(e)$.

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:

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

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.

3.3.1 Linearity of Expectation

An 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

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

random variables and then separately analyzing these simple RVs. Let's first prove this fact and then see how it can be used.

Theorem 3.1 (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).

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

3.3.2 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.

3.3.3 Example 2: Inversions in a random permutation

[hmm, lets leave this for homework]

3.4 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.

3.4.1 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 3.2 *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:

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

Formally, we are using the expression for Expectation given in (3.3), where the n different possible splits are the events A_i.[3] We can rewrite equation (3.4) by regrouping and getting rid of $T(0)$:

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

Now, we can solve this by the "guess and prove inductively" method. In order to do this, we 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

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

which we can see by drawing a graph of f and recalling that an integral represents the "area under the curve". 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:

$$\begin{aligned} T(n) &\leq (n-1) + \frac{2}{n}\sum_{i=1}^{n-1}(ci\ln i) \\ &\leq (n-1) + \frac{2}{n}\int_1^n (cx\ln x)dx \\ &\leq (n-1) + \frac{2}{n}\left((c/2)n^2\ln n - cn^2/4 + c/4\right) \\ &\leq cn\ln n, \quad \text{for } c = 2. \quad \blacksquare \end{aligned}$$

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] 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.

3.4.2 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 3.2): 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 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. 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:

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

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

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

3.5 Further Discussion

3.5.1 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,

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

Finally, no matter how many shares s of stock we hold at time t, $\mathbf{E}[X_t|s] = 0$. So, using (3.3), whatever probability distribution over s is induced by our strategy, $\mathbf{E}[X_t] = 0$. Since this holds for every t, we have $\mathbf{E}[X] = 0$.

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.

3.5.2 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 $\sum_k 2n/k = 2nH_n$.

Lecture 4

Selection (deterministic & randomized): 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 by (Manuel) Blum, Floyd, Pratt, Rivest, and Tarjan. In this lecture we describe two linear-time algorithms for this problem: one randomized and one deterministic. More generally, we solve the problem of finding the kth smallest out of an unsorted array of n elements.

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. For instance, if we are looking for the 87th-smallest element in our array, and after partitioning the "LESS" subarray (of elements less than the pivot) has size 200, then we just need to find the 87th smallest element in LESS. On the other hand, if the "LESS" subarray has size 40, then we just need to find the $87-40-1 = 46$th smallest element in GREATER. (And if the "LESS" subarray has size exactly 86 then we just return the pivot). One might at first think that allowing the algorithm to only recurse on one subarray rather than two would just cut down time by a factor of 2. However, since this is occuring recursively, it compounds the savings and we end up with $\Theta(n)$ rather than $\Theta(n \log n)$ time. 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 $O(n)$.*

Before giving a formal proof, let's first get some intuition. 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 $3n/4$ is only the *expected* size of the larger piece. That is, if i is the size of the larger piece, our expected cost to go is really $\mathbf{E}[T(i)]$ rather than $T(\mathbf{E}[i])$. However, 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). Let's now see this a bit more formally.

Proof (Theorem 4.1): 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 $T(n) < 4n$.

First of all, it takes $n-1$ comparisons to split into the array into two pieces in Step 2. These pieces are equally likely to 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. The piece we recurse on will depend on k, but since we are only giving an upper bound, we can 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 based on our intuition above. 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. ■

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.

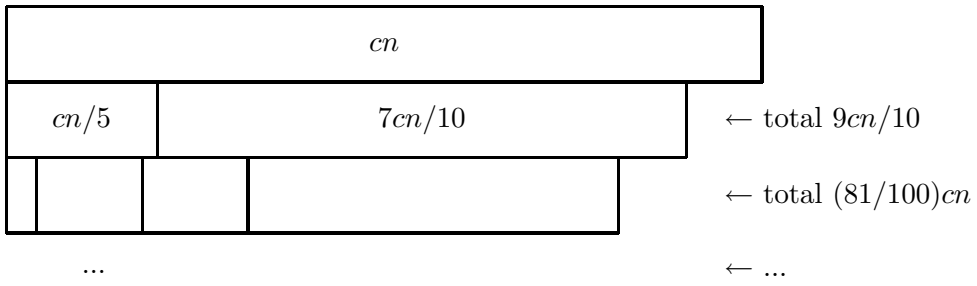


Figure 4.1: Stack of bricks view of recursions tree for recurrence 4.1.

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

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 time $O(f(n))$ 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, and our goal here is to make $f(n)$ as small as possible. In this lecture we examine the "lower bound problem." Here, the goal is to prove that *any* algorithm must take time $\Omega(g(n))$ time to solve the problem, where now our goal is to do this for $g(n)$ as large as possible. Lower bounds help us understand how close we are to the best possible solution to some problem: e.g., if we have an algorithm that runs in time $O(n \log^2 n)$ and a lower bound of $\Omega(n \log n)$, then we have a $\log(n)$ "gap": the maximum possible savings we could hope to achieve by improving our algorithm.

Often, we will prove lower bounds in restricted models of computation, that specify what types of operations may be performed on the input and at what cost. So, a lower bound in such a model means that if we want to do better, we would need somehow to do something outside the model.

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. In particular, let us make the following definition.

Definition 5.1 *A* comparison-based sorting algorithm *takes as input an array $[a_1, a_2, \ldots, a_n]$ of n items, and can only gain information about the items by comparing pairs of them. Each comparison ("is $a_i > a_j$?") returns YES or NO and counts a 1 time-step. The algorithm may also for free*

reorder items based on the results of comparisons made. In the end, the algorithm must output a permutation of the input in which all items are in sorted order.

For instance, Quicksort, Mergesort, and Insertion-sort 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, and an adversary by responding correctly can force the algorithm to ask many questions before it can tell what is the correct permutation to output.

Proof: Recall that the sorting algorithm must output a permutation of the input $[a_1, a_2, \ldots, a_n]$. The key to the argument is that (a) there are $n!$ different possible permutations the algorithm might output, and (b) for each of these permutations, there exists an input for which that permutation is the only correct answer. For instance, the permutation $[a_3, a_1, a_4, a_2]$ is the only correct answer for sorting the input $[2, 4, 1, 3]$. In fact, if you fix a set of n distinct elements, then there will be a 1-1 correspondence between the different orderings the elements might be in and the permutations needed to sort them.

Given (a) and (b) above, this means we can fix some set of $n!$ inputs (e.g., all orderings of $\{1, 2, \ldots, n\}$), one for each of the $n!$ output permutations.

let S be the set of these inputs that are consistent with the answers to all comparisons made so far (so, initially, $|S| = n!$). We can think of a new comparison as splitting S into two groups: those inputs for which the answer would be YES and those for which the answer would be NO. Now, suppose an adversary always gives the answer to each comparison corresponding to the larger group. Then, each comparison will cut down the size of S by at most a factor of 2. Since S initially has size $n!$, and by construction, the algorithm at the end must have reduced $|S|$ down to 1 in order to know which output to produce, the algorithm must make at least $\log_2(n!)$ comparisons before it can halt. We can then solve:

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

Notice that our proof is like a game of 20 Questions in which the responder (the adversary) 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.

Let's do an example with $n = 3$, and S as initially consisting of the 6 possible orderings of $\{1, 2, 3\}$:

$$(123), (132), (213), (231), (312), (321).$$

Suppose the sorting algorithm initially compares the first two elements a_1 and a_2. Half of the possibilities have $a_1 > a_2$ and half have $a_2 > a_1$. So, the adversary can answer either way and let's say it answers that $a_2 > a_1$. This narrows down the input to the three possibilities:

$$(123), (132), (231).$$

Suppose the next comparison is between a_2 and a_3. In this case, the most popular answer is that $a_2 > a_3$, so the adversary returns that answer which removes just one ordering, leaving the algorithm with:

$$(132), (231).$$

It now takes one more comparison to finally isolate the input ordering and determine the correct permutation to output.

Alternative view of the proof: Another way of looking at the proof we gave above is as follows. For a deterministic algorithm, the permutation it outputs is solely a function of the series of answers it receives (any two inputs producing the same series of answers will cause the same permutation to be output). So, if an algorithm always made at most $k < \lg(n!)$ comparisons, then there are at most $2^k < n!$ different permutations it can possibly output. In other words, there is some permutation it *can't* output. So, the algorithm will fail on any input for which that permutation is the only correct answer.

Question: Suppose we consider the problem: "order the input array so that the smallest $n/2$ come before the largest $n/2$"? Does our lower bound still hold for that problem, or where does it break down? How fast can you solve that problem?

Answer: No, the proof does not still hold. It breaks down because any given input can have multiple correct answers. E.g., for input `[2 1 4 3]`, we could output any of $[a_1, a_2, a_3, a_4]$, $[a_2, a_1, a_3, a_4]$, $[a_1, a_2, a_4, a_3]$, or $[a_2, a_1, a_4, a_3]$. In fact, not only does the lower bound break down, but we can actually solve this problem in linear time: just run the linear-time median-finding algorithm and then make a second pass putting elements into the first half or second half based on how they compare to the median.

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 $\mathcal{A}$, the average-case number of comparisons (the number of comparisons on average on a randomly chosen permutation of n distinct elements) is at least $\lfloor \log_2(n!) \rfloor$.*

Proof: Let S be the set of all $n!$ possible orderings of n distinct elements. As noted in the previous argument, these each require a different permutation to be produced as output. Let's now build out the entire decision tree for algorithm $\mathcal{A}$ on S: the tree we get by looking at all the different question/answer paths we get by running algorithm $\mathcal{A}$ on the inputs in S. This tree has $n!$ leaves, where the depth of a leaf is the number of comparisons performed by the sorting algorithm on

that input. Our goal is to show that the *average* depth of the leaves must be at least $\lfloor \log_2(n!) \rfloor$ (previously, we only cared about the *maximum* depth).

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 $D-1$. Since any unbalanced tree can be modified to have a smaller average depth, such a tree cannot be one that *minimizes* average depth, and therefore the tree of smallest average depth must in fact be balanced. ■

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: The argument here is a bit subtle. The first step is to argue that with respect to counting comparisons, we can think of a randomized algorithm $\mathcal{A}$ as a probability distribution over deterministic algorithms. In particular, we can think of a randomized algorithm $\mathcal{A}$ as a deterministic algorithm with access to a special "random bit tape": every time $\mathcal{A}$ wants to flip a coin, it just pulls the next bit off that tape. In that case, for any *given* run of algorithm $\mathcal{A}$, say reading bit-string s from that tape, there is an equivalent deterministic algorithm $\mathcal{A}_s$ with those bits hardwired in. Algorithm $\mathcal{A}$ is then a probability distribution over all those deterministic algorithms $\mathcal{A}_s$.

This means that the expected number of comparisons made by randomized algorithm $\mathcal{A}$ on some input I is just

$$\sum_s \Pr(s)(\text{Running time of } \mathcal{A}_s \text{ on } I).$$

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 $\lfloor \log_2(n!) \rfloor$, any average over them must too. Formally, the average-case running time of the randomized algorithm is

$$\begin{aligned} \operatorname*{avg}_{\text{inputs } I} \sum_s [\Pr(s)(\text{Running time of } \mathcal{A}_s \text{ on } I)] &= \sum_s \operatorname*{avg}_I [\Pr(s)(\text{Running time of } \mathcal{A}_s \text{ on } I)] \\ &= \sum_s \Pr(s) \operatorname*{avg}_I (\text{Running time of } \mathcal{A}_s \text{ on } I) \end{aligned}$$

[1] Let us define a tree to be completely balanced if the deepest leaf is at most one level deeper than the shallowest leaf. Everything would be easier if we could somehow assume $n!$ was a power of 2....

$$\begin{aligned} &\geq \sum_s \Pr(s)\lfloor \log_2(n!)\rfloor \\ &= \lfloor \log_2(n!)\rfloor. \quad \blacksquare \end{aligned}$$

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 n given input elements (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" who gets to pick a row and an adversarial "input player" who gets to pick a column. Each player makes their choice and the entry in the matrix is the cost to the algorithm-player which we can think of as how much money the algorithm-player has to pay the input player. We have shown that there is a randomized 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.

Lecture 6

Concrete models and tight upper/lower bounds

6.1 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. Unlike many of the other lectures, in this one we will not be using O, Θ, and Ω, and we will instead try to examine exact quantities as much as possible. Specific models and problems examined in this lecture include:

- 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, and a more precise look at the number of comparisons needed to sort.
- The number of probes into a graph needed to determine if the graph is connected (the evasiveness of connectivity).

6.2 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 $f(n)$.

6.3 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 6.1 *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 6.1 (Upper bound) *$n-1$ exchanges is sufficient.*

Proof: To prove an upper bound of $n-1$ 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 kth-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 nth 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 6.2 (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 6.1. 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.

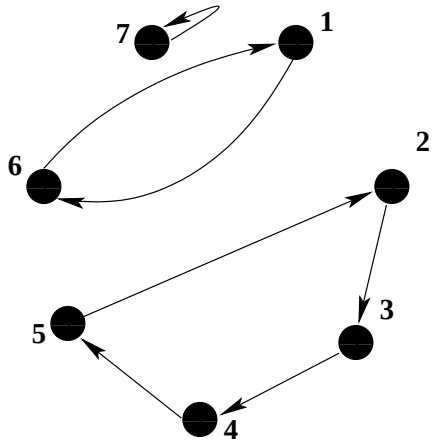


Figure 6.1: Graph for input `[f c d e b a g]`

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. ■

6.4 The comparison model

Let's now go back to the comparison model of computation we looked at last time.

Definition 6.2 *In the* comparison model*, we have an input containing n items, but the only information the algorithm can get about the items is by comparing pairs of them, where each comparison returns YES or NO. Each comparison costs 1 step. But exchanges and moves are free.*

In the last lecture, we looked at sorting in the comparison model, and gave a lower bound of $\lg(n!)$ on the number of comparisons needed.[1] Let us begin with a simple generalization. Suppose you have some problem where there are M possible different outputs the algorithm might be required to produce; e.g., for sorting, $M = n!$. Then, we have a worst-case lower bound of $\lg M$. The reason is that the algorithm needs to find out which of these M outputs is the right one, and each YES/NO question could be answered in a way that removes at most half of the possibilities remaining from consideration. So, in the worst case, it takes at least $\lg M$ steps to find the right answer.

Just to get a better handle on what exactly $\lg(n!)$ looks like, since today's theme is tight bounds, we can use the fact that $n! \in [(n/e)^n, n^n]$. So this means that:

$$\begin{array}{rcccl} n\lg n - n\lg e & < & \lg(n!) & < & n\lg n \\ n\lg n - 1.443n & < & \lg(n!) & < & n\lg n. \end{array}$$

Since $1.433n$ 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.

[1] We will use "lg" to mean "$\log_2$".

6.4.1 Almost-tight upper-bounds for comparison-based sorting

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 $(n-1) + 2(n/2-1) + 4(n/4-1) + \ldots + n/2(2-1) = n \lg n - (n-1) < n \lg n$.

6.4.2 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 6.3 (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 minimum element, our previous argument gives a lower bound of $\lg n$. But clearly this is not at all tight. In fact, we can give a better lower bound of $n-1$.

Claim 6.4 (Lower bound) *$n-1$ comparisons are needed in the worst-case to find the maximum of n elements.*

Proof: Suppose some algorithm $\mathcal{A}$ 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 $\mathcal{A}$. If fewer than $n-1$ comparisons are made, then this graph must have at least two components. Suppose now that algorithm $\mathcal{A}$ 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 $\mathcal{A}$, so on this new set of elements, algorithm $\mathcal{A}$ would still output u. Yet this now ensures that u is not the maximum, so $\mathcal{A}$ must be incorrect. ■

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

6.4.3 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 6.5 (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.

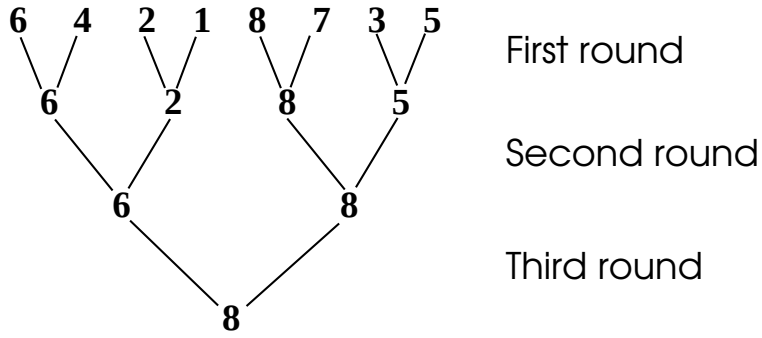
Claim 6.6 (Upper bound #1) *$2n-3$ comparisons are sufficient to find the second-largest of n elements.*

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 6.7 (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.,



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. ■

[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.

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]

6.5 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 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 6.8 (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 6.9 (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. ■

We'll see more arguments like this when we talk about spanning trees later on in the course.

[3] First shown by Kislitsyn (1964).

Lecture 7

Amortized Analysis

7.1 Overview

In this lecture we discuss a useful form of analysis, called *amortized analysis*, for problems in which one must perform a series of operations, and our goal is to analyze the time per operation. The motivation for amortized analysis is that looking at the worst-case time per operation can be too pessimistic if the only way to produce an expensive operation is to "set it up" with a large number of cheap operations beforehand.

We also introduce the notion of a *potential function* which can be a useful aid to performing this type of analysis. A potential function is much like a bank account: if we can take our cheap operations (those whose cost is less than our bound) and put our savings from them in a bank account, use our savings to pay for expensive operations (those whose cost is greater than our bound), and somehow guarantee that our account will never go negative, then we will have proven an *amortized* bound for our procedure.

As in the previous lecture, in this lecture we will avoid use of asymptotic notation as much as possible, and focus instead on concrete cost models and bounds.

7.2 Introduction

So far we have been looking at static problems where you are given an input (like an array of n objects) and the goal is to produce an output with some desired property (e.g., the same objects, but sorted). For next few lectures, we're going to turn to problems where we have a *series* of operations, and goal is to analyze the time taken per operation. For example, rather than being given a set of n items up front, we might have a series of n insert, lookup, and remove requests to some database, and we want these operations to be efficient.

Today, we will talk about a useful kind of analysis, called *amortized analysis* for problems of this sort. The definition of amortized cost is actually quite simple:

Definition 7.1 *The* **amortized cost per operation** *for a sequence of n operations is the total cost of the operations divided by n.*

For example, if we have 100 operations at cost 1, followed by one operation at cost 100, the

amortized cost per operation is $200/101 < 2$. The reason for considering amortized cost is that we will be interested in data structures that occasionally can incur a large cost as they perform some kind of rebalancing or improvement of their internal state, but where such operations cannot occur too frequently. In this case, amortized analysis can give a much tighter bound on the true cost of using the data structure than a standard worst-case-per-operation bound. Note that even though the definition of amortized cost is simple, analyzing it will often require some thought. We will illustrate how this can be done through several examples.

7.3 Example #1: implementing a stack as an array

Say we want to use an array to implement a stack. We have an array `A`, with a variable `top` that points to the top of the stack (so `A[top]` is the next free cell). This is pretty easy:

- To implement `push(x)`, we just need to perform:

  ```
  A[top] = x;
  top++;
  ```

- To implement `x=pop()`, we just need to perform:

  ```
  top--;
  x = A[top];
  ```

  (first checking to see if `top==0` of course...)

However, what if the array is full and we need to push a new element on? In that case we can allocate a new larger array, copy the old one over, and then go on from there. This is going to be an expensive operation, so a push that requires us to do this is going to cost a lot. But maybe we can "amortize" the cost over the previous cheap operations that got us to this point. So, on average over the sequence of operations, we're not paying too much. To be specific, let us define the following cost model.

Cost model: Let's say that inserting into the array costs 1, taking an element out of the array costs 1, and the cost of resizing the array is the number of elements moved. (Say that all other operations, like incrementing or decrementing "top", are free.)

Question 1: What if when we resize we just increase the size by 1? Is that a good idea?

Answer 1: Not really. If our n operations consist of n pushes then we will incur a total cost $1 + 2 + 3 + 4 + \ldots + n = n(n+1)/2$. That's an amortized cost of $(n+1)/2$ per operation.

Question 2: What if we instead decide to double the size of the array when we resize?

Answer 2: This is much better. Now, in any sequence of n operations, the total cost for resizing is $1 + 2 + 4 + 8 + \ldots + 2^i$ for some $2^i < n$ (if all operations are pushes then 2^i will be the largest power of 2 less than n). This sum is at most $2n - 1$. Adding in the additional cost of n for inserting/removing, we get a total cost $< 3n$, and so our amortized cost per operation is < 3.

7.4 Piggy banks and potential functions

Here is another way to analyze the process of doubling the array in the above example. Say that every time we perform a push operation, we pay \$1 to perform it, and we put \$2 into a piggy bank. So, our out-of-pocket cost per push is \$3. Any time we need to double the array, from size L to $2L$, we pay for it using money in the bank. How do we know there will be enough money (\$$L$) in the bank to pay for it? The reason is that after the last resizing, there were only $L/2$ elements in the array and so there must have been *at least* $L/2$ new pushes since then contributing \$2 each. So, we can pay for everything by using an out-of-pocket cost of at most \$3 per operation. Putting it another way, by spending \$3 per operation, we were able to pay for all the operations plus possibly still have money left over in the bank. This means our amortized cost is at most 3.[1]

This "piggy bank" method is often very useful for performing amortized analysis. The piggy bank is also called a *potential function*, since it is like potential energy that you can use later. The potential function is a guarantee on the amount of money in the bank. In the case above, the potential is twice the number of elements in the array after the midpoint. *Note that it is very important in this analysis to prove that the bank account doesn't go negative.* Otherwise, if the bank account can slowly drift off to negative infinity, the whole proof breaks down.

Definition 7.2 *A* **potential function** *is a function of the state of a system, that generally should be non-negative and start at 0, and is used to smooth out analysis of some algorithm or process.*

Observation: If the potential is non-negative and starts at 0, and at each step the actual cost of our algorithm plus the change in potential is at most c, then after n steps our total cost is at most cn. That is just the same thing we were saying about the piggy bank: our total cost for the n operations is just our total out of pocket cost minus the amount in the bank at the end.

Sometimes one may need in an analysis to "seed" the bank account with some initial positive amount for everything to go through. In that case, the kind of statement one would show is that the total cost for n operations is at most cn plus the initial seed amount.

Recap: The motivation for amortized analysis is that a worst-case-per-operation analysis can give overly pessimistic bound if the only way of having an expensive operation is to have a lot of cheap ones before it. Note that this is *different* from our usual notion of "average case analysis": we are not making any assumptions about the inputs being chosen at random, we are just averaging over time.

7.5 Example #2: a binary counter

Imagine we want to store a big binary counter in an array A. All the entries start at 0 and at each step we will be simply incrementing the counter. Let's say our cost model is: whenever we increment the counter, we pay \$1 for every bit we need to flip. (So, think of the counter as an

[1] In fact, if you think about it, we can pay for pop operations using money from the bank too, and even have \$1 left over. So as a more refined analysis, our amortized cost is \$3 per push and \$$-1$ per successful pop (a pop from a nonempty stack).

array of heavy stone tablets, each with a "0" on one side and a "1" on the other.) For instance, here is a trace of the first few operations and their cost:

A[m]	A[m-1]	...	A[3]	A[2]	A[1]	A[0]	**cost**
0	0	...	0	0	0	0	
							$1
0	0	...	0	0	0	1	
							$2
0	0	...	0	0	1	0	
							$1
0	0	...	0	0	1	1	
							$3
0	0	...	0	1	0	0	
							$1
0	0	...	0	1	0	1	
							$2

In a sequence of n increments, the worst-case cost per increment is $O(\log n)$, since at worst we flip $\lg(n) + 1$ bits. But, what is our *amortized* cost per increment? The answer is it is at most 2. Here are two proofs.

Proof 1: Every time you flip $0 \to 1$, pay the actual cost of \$1, plus put \$1 into a piggy bank. So the total amount spent is \$2. In fact, think of each bit as having its own bank (so when you turn the stone tablet from 0 to 1, you put a \$1 coin on top of it). Now, every time you flip a $1 \to 0$, use the money in the bank (or on top of the tablet) to pay for the flip. Clearly, by design, our bank account cannot go negative. The key point now is that even though different increments can have different numbers of $1 \to 0$ flips, each increment has exactly one $0 \to 1$ flip. So, we just pay \$2 (amortized) per increment.

Equivalently, what we are doing in this proof is using a potential function that is equal to the number of 1-bits in the current count. Notice how the bank-account/potential-function allows us to smooth out our payments, making the cost easier to analyze.

Proof 2: Here is another way to analyze the amortized cost. First, how often do we flip `A[0]`? Answer: every time. How often do we flip `A[1]`? Answer: every other time. How often do we flip `A[2]`? Answer: every 4th time, and so on. So, the total cost spent on flipping `A[0]` is n, the total cost spent flipping `A[1]` is at most $n/2$, the total cost flipping `A[2]` is at most $n/4$, etc. Summing these up, the total cost spent flipping all the positions in our n increments is at most $2n$.

7.6 Example #3: What if it costs us 2^k to flip the kth bit?

Imagine a version of the counter we just discussed in which it costs 2^k to flip the bit `A[k]`. (Suspend disbelief for now — we'll see shortly why this is interesting to consider). Now, in a sequence of n increments, a single increment could cost as much as n (actually $2n - 1$), but the claim is the amortized cost is only $O(\log n)$ per increment. This is probably easiest to see by the method of "Proof 2" above: `A[0]` gets flipped every time for cost of \$1 each (a total of \$$n$). `A[1]` gets flipped

every other time for cost of \$2 each (a total of at most \$$n$). `A[2]` gets flipped every 4th time for cost of \$4 each (again, a total of at most \$$n$), and so on up to `A[`$\lfloor \lg n \rfloor$`]` which gets flipped once for a cost at most \$$n$. So, the total cost is at most $n(\lg n + 1)$, which is $O(\log n)$ amortized per increment.

7.7 Example #4: A simple amortized dictionary data structure

One of the most common classes of data structures are the "dictionary" data structures that support fast insert and lookup operations into a set of items. In the next lecture we will look at balanced-tree data structures for this problem in which both inserts and lookups can be done with cost only $O(\log n)$ each. Note that a sorted array is good for lookups (binary search takes time only $O(\log n)$) but bad for inserts (they can take linear time), and a linked list is good for inserts (can do them in constant time) but bad for lookups (they can take linear time). Here is a method that is very simple and *almost* as good as the ones in the next lecture. This method has $O(\log^2 n)$ search time and $O(\log n)$ amortized cost per insert.

The idea of this data structure is as follows. We will have a collection of arrays, where array i has size 2^i. Each array is either empty or full, and each is in sorted order. However, there will be no relationship between the items in different arrays. The issue of which arrays are full and which are empty is based on the binary representation of the number of items we are storing. For example, if we had 11 items (where $11 = 1 + 2 + 8$), then the arrays of size 1, 2, and 8 would be full and the rest empty, and the data structure might look like this:

```
A0:  [5]
A1:  [4,8]
A2:  empty
A3:  [2, 6, 9, 12, 13, 16, 20, 25]
```

To perform a lookup, we just do binary search in each occupied array. In the worst case, this takes time $O(\log(n) + \log(n/2) + \log(n/4) + \ldots + 1) = O(\log^2 n)$.

What about inserts? We'll do this like mergesort. To insert the number 10, we first create an array of size 1 that just has this single number in it. We now look to see if `A0` is empty. If so we make this be `A0`. If not (like in the above example) we merge our array with `A0` to create a new array (which in the above case would now be `[5, 10]`) and look to see if `A1` is empty. If `A1` is empty, we make this be `A1`. If not (like in the above example) we merge this with `A1` to create a new array and check to see if `A2` is empty, and so on. So, inserting 10 in the example above, we now have:

```
A0:  empty
A1:  empty
A2:  [4, 5, 8, 10]
A3:  [2, 6, 9, 12, 13, 16, 20, 25]
```

Cost model: To be clear about costs, let's say that creating the initial array of size 1 costs 1, and merging two arrays of size m costs $2m$ (remember, merging sorted arrays can be done in linear time). So, the above insert had cost $1 + 2 + 4$.

For instance, if we insert again, we just put the new item into `A0` at cost 1. If we insert again, we merge the new array with `A0` and put the result into `A1` at a cost of $1 + 2$.

Claim 7.1 *The above data structure has amortized cost $O(\log n)$ per insert.*

Proof: With the cost model defined above, it's exactly the same as the binary counter with cost 2^k for counter k. ■

Lecture 8

Balanced search trees

8.1 Overview

In this lecture we discuss search trees as a method for storing data in a way that supports fast insert, lookup, and delete operations. (Data structures handling these operations are often called *dictionary* data structures.) The key issue with search trees is that you want them to be *balanced* so that lookups can be performed quickly, and yet you don't want to require them to be perfect because that would be too expensive to maintain when a new element is inserted or deleted. In this lecture, we discuss *B-trees* and *treaps*, which are two methods that handle this tradeoff so that all desired operations can be performed in time $O(\log n)$.

Topics covered in this lecture:

- Simple binary search trees: like an on-the-fly version of quicksort.
- B-trees: a form of balanced search tree that uses flexibility in its node degrees to efficiently keep the tree balanced.
- Treaps: like an on-the-fly version of *randomized* quicksort, that uses randomization to keep the tree balanced with high probability.
- Tree-rotations: an important concept when talking about binary search trees, that is used inside many binary search tree data structures (including treaps).

8.2 Introduction

For the next few lectures we will be looking at several important data-structures. A data-structure is a method for storing data so that operations you care about can be performed quickly. Data structures are typically used as part of some larger algorithm or system, and good data structures are often crucial when especially fast performance is needed.

We will be focusing in particular on *dictionary* data structures, which support `insert` and `lookup` operations (and usually `delete` as well). Specificially,

Definition 8.1 *A* **dictionary data structure** *is a data structure supporting the following operations:*

- **insert(key, object)***: insert the (key, object) pair. For instance, this could be a word and its definition, a name and phone number, etc. The key is what will be used to access the object.*
- **lookup(key)***: return the associated object.*
- **delete(key)***: remove the key and its object from the data structure. We may or may not care about this operation.*

For example, perhaps we are the phone company and we have a database of people and their phone numbers (plus addresses, billing information and so on). As new people come in, we'd like to be able to insert them into our database. Given a name, we'd like to be able to quickly find their associated information.

One option is we could use a sorted array. Then, a lookup takes $O(\log n)$ time using binary search. However, an insert may take $\Omega(n)$ time in the worst case because we have to shift everything to the right in order to make room for the new key. Another option might be an unsorted list. In that case, inserting can be done in $O(1)$ time, but a lookup may take time $\Omega(n)$. In the last lecture we saw a data structure that consisted of an unsorted *set* of sorted arrays, where insert took $O(\log n)$ amortized time and lookup took time $O(\log^2 n)$. Today we will look at search tree methods that allow us to perform both operation in time $O(\log n)$.

A binary search tree is a binary tree in which each node stores a (key, object) pair such that all descendants to the left have smaller keys and all descendants to the right have larger keys (let's not worry about the case of multiple equal keys). To do a lookup operation you simply walk down from the root, going left or right depending on whether the query is smaller or larger than the key in the current node, until you get to the correct key or walk off the tree. We will also talk about non-binary search trees that potentially have more than one key in each node, and nodes may have more than two children.

For the rest of this discussion, we will ignore the "object" part of things. We will just worry about the keys since that is all that matters as far as understanding the data structures is concerned.

8.3 Simple binary search trees

The simplest way to maintain a binary search tree is to implement the insert operations as follows.

insert(x): If the tree is empty then put x in the root. Otherwise, compare it to the root: if x is smaller then recursively insert on the left; otherwise recursively insert on the right.

Equivalently: walk down the tree as if doing a lookup, and then insert x into a new leaf at the end.

Example: build a tree by inserting the sequence: `C A R N É G I E` (where `É` $>$ `E`).

Plusses and minuses: On the positive side, this is very easy to implement (though deletes are slightly painful — think about how you might handle them). On the negative side, this has very bad worst-case behavior. In fact, it behaves exactly like quicksort using the leftmost element as the pivot, and the search tree is the same as the quicksort recursion tree. In particular, if elements are in sorted order, this will produce a very unbalanced tree where all operations take time $\Omega(n)$.

Today we will examine two ways to fix this problem, *B-trees* and *treaps*. B-trees are a particularly nice method used in database applications, and treaps are a lot like *randomized* quicksort, but trickier since the keys are coming in one at a time.

An important idea: the problem with the basic binary search tree was that we were not maintaining balance. On the other hand, if we try to maintain a perfectly balanced tree, we will spend too much time rearranging things. So, we want to be balanced but also give ourselves some slack. It's a bit like how in the median-finding algorithm, we gave ourselves slack by allowing the pivot to be "near" the middle. For B-trees, we will make the tree perfectly balanced, but give ourselves slack by allowing some nodes to have more children than others.

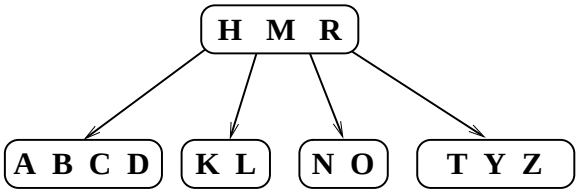
8.4 B-trees and 2-3-4 trees

A **B-tree** is a search tree where for some pre-specified $t \geq 2$ (think of $t = 2$ or $t = 3$):

- Each node has between $t-1$ and $2t-1$ keys in it (except the root has between 1 and $2t-1$ keys). Keys in a node are stored in a sorted array.
- Each non-leaf has degree (number of children) equal to the number of keys in it plus 1. So, node degrees are in the range $[t, 2t]$ except the root has degree in the range $[2, 2t]$. The semantics are that the ith child has items between the $(i-1)$st and ith keys. E.g., if the keys are $[a_1, a_2, \ldots, a_{10}]$ then there is one child for keys less than a_1, one child for keys between a_1 and a_2, and so on, until the rightmost child has keys greater than a_{10}.
- All leaves are at the same depth.

The idea is that by using flexibility in the sizes and degrees of nodes, we will be able to keep trees perfectly balanced (in the sense of all leaves being at the same level) while still being able to do inserts cheaply. Note that the case of $t = 2$ is called a **2-3-4 tree** since degrees are 2, 3, or 4.

Example: here is a tree for $t = 3$ (so, non-leaves have between 3 and 6 children—though the root can have fewer—and the maximum size of any node is 5).



Now, the rules for lookup and insert turn out to be pretty easy:

Lookup: Just do binary search in the array at the root. This will either return the item you are looking for (in which case you are done) or a pointer to the appropriate child, in which case you recurse on that child.

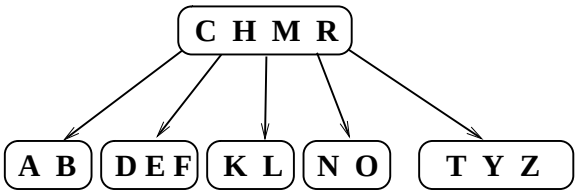
Insert: To insert, walk down the tree as if you are doing a lookup, but if you ever encounter a *full* node (a node with the maximum $2t-1$ keys in it), perform a **split** operation on it (described below) before continuing.

Finally, insert the new key into the leaf reached.

Split: To split a node, pull the median of its keys up to its parent and then split the remaining $2t-2$ keys into two nodes of $t-1$ keys each (one with the elements less than the median and

one with the elements greater than the median). Then connect these nodes to their parent in the appropriate way (one as the child to the left of the median and one as the child to its right). If the node being split is the root, then create a fresh new root node to put the median in.

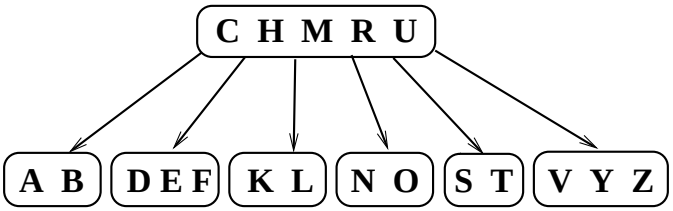
Let's consider the example above. If we insert an "**E**" then that will go into the leftmost leaf, making it full. If we now insert an "**F**", then in the process of walking down the tree we will split the full node, bringing the "**C**" up to the root. So, after inserting the "**F**" we will now have:



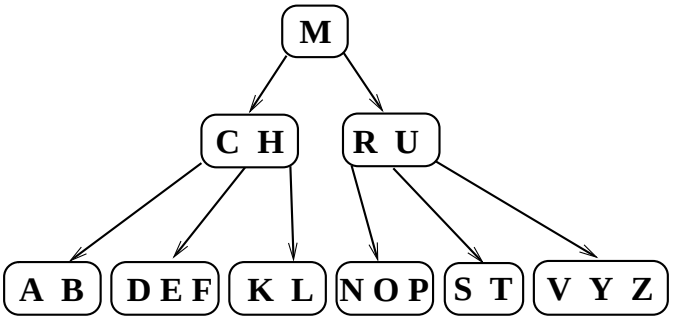
Question: We know that performing a split maintains the requirement of at least $t - 1$ keys per non-root node (because we split at the median) but is it possible for a split to make the parent *over-full*?

Answer: No, since if the parent was full we would have already split it on the way down.

Let's now continue the above example, inserting "**S**", "**U**", and "**V**" (which causes a split):



Now, suppose we insert "**P**". Doing this will bring "**M**" up to a new root, and then we finally insert "**P**" in the appropriate leaf node:



Question: is the tree always height-balanced (all leaves at the same depth)?

Answer: yes, since we only grow the tree *up*.

So, we have maintained our desired properties. What about running time? To perform a lookup, we perform binary search in each node we pass through, so the total time for a lookup is $O(\text{depth} \times \log t)$. What is the depth of the tree? Since at each level we have a branching factor of at least t (except possibly at the root), the depth is $O(\log_t n)$. Combining these together, we see that the "t" cancels out in the expression for lookup time:

$$\text{Time for lookup } = O(\log_t n \times \log t) = O(\log n).$$

Inserts are similar to lookups except for two issues. First, we may need to split nodes on the way down, and secondly we need to insert the element into the leaf. So, we have:

$$\text{Time for insert } = \text{ lookup-time } + \text{ splitting-time } + \text{ time-to-insert-into-leaf.}$$

The time to insert into a leaf is $O(t)$. The splitting time is $O(t)$ per split, which could happen at each step on the way down. So, if t is a *constant*, then we still get total time $O(\log n)$.

What if we don't want to think of t as a constant, though. The interesting thing is that even if t is large, amortized analysis comes to our rescue. In particular, if we create a tree from n inserts, we can have made at most $O(n/t)$ splits *total* in the process. **Why?** Because each split creates a new node, and there are $O(n/t)$ nodes total. So the *total* time spent on splits over all n inserts is $O(n)$, which means that we are only spending $O(1)$ time on average on splits per insert. So, the amortized time per insert is:

$$O(\log n) + O(1) + O(t) = O(\log n + t).$$

More facts about B-trees:

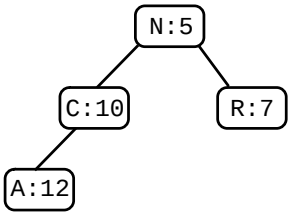
- B-trees are used a lot in databases applications because they fit in nicely with the memory heirarchy when you use a large value of t. For instance, if you have 1 billion items and use $t = 1,000$, then you can probably keep the top two levels in fast memory and only make one disk access at the bottom level. The savings in disk accesses more than makes up for the additive $O(t)$ cost for the insert.

- If you use $t = 2$, you have what is known as a 2-3-4 tree. What is special about 2-3-4 trees is that they can be implemented efficiently as binary trees using an idea called "red-black-trees". We will not discuss these in detail, but they use the same notion of tree rotation as treaps (which are discussed below).

8.5 Treaps

Going back to binary search trees, we saw how a standard binary search tree is like quicksort using the leftmost element as a pivot, with all of its worst-case problems. A natural question is: can we come up with a method that is instead like *randomized* quicksort? The problem is that we don't have all the elements at the start, so it's not so obvious (we can't just say "let the root be some element we are *going* to see in the future"). However, it turns out we *can* come up with an analog to randomized quicksort, and the data structure based on this is called a "treap".

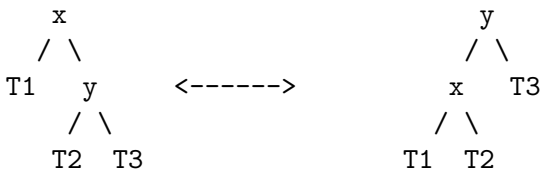
The idea for a treap is that when an element x is inserted, we also give it a random *priority* value. Think of the priorities as giving the order in which they are supposed to be chosen as pivots. (Also, think of priorities as real numbers so we don't get any ties). In particular, the property we will require is that if v is a child of u, then priority(v) > priority(u). For example:



So, the keys are *search-tree ordered* and the priorities are *heap ordered*, which is why this is called a *treap*!

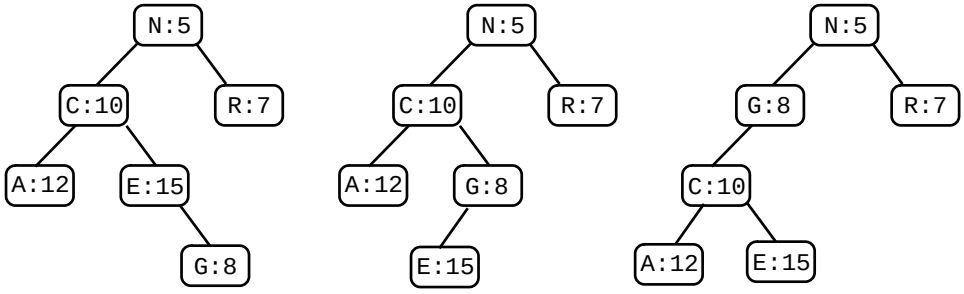
Question: Why must such a thing even exist? Given a set of (key, priority) pairs, how do we know it is even *possible* to design a tree so that the keys are in search-tree order and the priorities are in heap order? **Answer:** just sort by priority and run the standard BST insertion algorithm. Moreover, notice that if we choose priorities at random, the tree is exactly the same as the recursion tree of a randomized quicksort that chooses the pivots in the same random order as the priorities.

The big question now is: how can we perform inserts to maintain the treap property? It turns out it is not too difficult. To insert a new element into a treap, just do the usual binary search tree insert (walking down and inserting at a leaf) and then *rotate* the new item up the tree so long as its parent has a larger priority value. A *tree rotation* is the following operation (which can be done in either direction) that maintains search tree order:



Here, T_1, T_2, T_3 represent subtrees. This rotation is legal (maintains search tree order) because both trees correspond to the statement $T_1 < x < T_2 < y < T_3$. In the above picture, in the left-to-right direction, we will call this "rotating y above x" (or "rotating x above y" in the right-to-left direction).

Let's do an example of inserting into a treap. Suppose we are inserting the letters `C A R N E G I É`, and so far we have the tree with 4 letters above. If we now insert `E` with priority 15 (so no rotations) and then `G` with priority 8, we would do:



We now need to prove this maintains the treap property. First, the search-tree property on keys is maintained since that is not affected by rotations. We can analyze the heap property as follows. Initially, all descendant relations are satisfied (if y is descendant of x then priority(y) > priority(x)) *except* for case that y is the new node. Now, suppose the new node y does violate the heap property. Then it must do so with its parent x, and we will do a rotation. Without loss of generality, assume

it is left-to-right in the generic picture above. Notice now that the only new descendant relation we add is that x and T_1 become descendants of y. But since priority(x) > priority(y), and priority(T_1) > priority(x) by assumption, these are all satisfied. So, we maintain our invariant. Finally when new node y has priority greater than its parent, all descendant relations are satisfied and we are done.

So, insert can be done in time proportional to the search time, since at worst the number of rotations equals the number of steps on the way down. (One can actually furthermore show that the *expected* number of rotations per insert is $O(1)$.)

Depth analysis. Inserts and searches both take time proportional to the depth of the tree, so all that remains is to analyze depth. When we analyzed randomized quicksort, we showed that in expectation, the total number of comparisons is $O(n \log n)$. This means that in expectation, the *sum* of all node depths is $O(n \log n)$, or equivalently, in expectation, the *average* node depth is $O(\log n)$. However, we can actually show something a lot stronger: in fact, with high probability, the *maximum* node depth is $O(\log n)$. (This also implies that the quicksort $O(n \log n)$ bound holds with high probability.) Here is a sketch of one way to prove it:

Proof: let's go back to our "dart-throwing" argument for quicksort. Let's line up the elements in sorted order, and pick one element X that we care about. We can think about the depth of this node as follows: we throw a dart at random into this sorted list, representing whatever element is at the root of the tree, and whereever it lands, it cuts the list and the part that X is not on disappears. We do this again, representing the next node on the path to X, and keep going until only X is left. Now, if you think about it, whether the dart lands to the left or right of X, it has a 50% chance of deleting at least half of that side of the list. This can happen at most $\lg n$ times to the left of X, and at most $\lg n$ times to the right.

So, we can think of it like this: each dart/pivot is like a coin toss with a 50% chance of heads. Each heads cuts off at least half of that side of the list. We have to stop sometime before getting $2 \lg n$ heads. There's a nice bound called "Hoeffding's inequality" that says that if you flip a coin t times, the chance of getting less than $t/4$ heads is at most $e^{-t/8}$. So, if we flip $8 \lg n$ times, the chance of getting at most $2 \lg n$ heads is at most $e^{-\lg n} = e^{-(\ln n)/(\ln 2)} = 1/n^{1.44}$. Even if you multiply this by n to account for the fact that we want this to be true for *every* node X, you still get that it's unlikely *any* element has depth more than $8 \lg n$.[1]

[1] Hoeffding bounds also say that the chance you get fewer than $t/8$ heads in t flips is at most $e^{-9t/32}$. So in $16 \lg n$ flips, the chance of failure is at most $n^{-6.49}$. This means the chance that *any* X has depth greater than $16 \lg n$ is at most $1/n^{5.49}$.

Lecture 9

Digit-based sorting and data structures

9.1 Overview

In this short lecture we examine digit-based sorting and digit-based data structures. Our previous data structures treated keys as abstract objects that could only be examined via comparisons. The methods discussed in this lecture will instead treat them as a sequence of digits or a sequence of characters. Material in this lecture includes:

- Radix sort: a method for sorting strings or integers.
- Tries: a data structure that can be viewed as an on-the-fly version of radix sort.

9.2 Introduction

So far, we have looked at sorting and storing items whose keys can be anything at all, so long as we have a notion of "less than". Today we will look at methods for the case that keys are natural objects like strings or integers.

To start, say we have n objects to sort whose keys are all integers in a small range: $1, \ldots, r$ (e.g., say I have a stack of homeworks to sort by section). In this case, what would be a fast way to sort? One easy method is bucket-sort:

Bucket-sort:

- Make an array A of size r, of "buckets" (perhaps implemented as linked lists).
- Make one pass through the n objects. Insert object with key k into the kth bucket $A[k]$.
- Finally, make a pass through the array of buckets, concatenating as you go.

The first step of bucket-sort takes time $O(r)$, the second step takes time $O(n)$, and the last step takes time $O(r)$ or $O(r+n)$ depending on how the buckets are implemented. In any case, the total time is $O(n+r)$.

So, bucket-sort is linear time if $r = O(n)$. Notice one thing interesting about it is that by using the magic of indirect addressing, we are making an r-way decision at each step (something you can't do in the comparison model). Unfortunately, bucket-sort is only good if the range of keys is small. This leads us to our next method, radix-sort.

9.3 Radix Sort

Suppose our keys are numbers, or strings, that can be viewed as a *sequence* of digits (or characters) each in a range of size r. For example, we might have $r = 10$, 26, or 128. In that case, there is a natural sorting algorithm called Radix Sort we can use to sort them. ("radix" is the base r). Actually there are two versions: one that's conceptually easier called Most-Significant-First radix sort, where we go top-down digit by digit (or byte by byte), and another that's trickier to think of but easy to code called Least-Significant-First radix sort where we go in the other direction.

9.3.1 Most-significant-first (MSF) radix sort

Most-significant-first radix sort begins by sorting keys into buckets according to their most significant character or digit. For instance, if we are sorting strings, we would get a bucket for 'a', a bucket for 'b', a bucket for 'c' and so on. So, now the strings are roughly sorted in that any two strings that begin with different letters are in the correct order.[1] Now we just recursively sort each bucket that has more than one element using the same procedure (sorting into buckets by the next most significant character and so on) and then concatenate the buckets.

If we ignore time spent scanning empty buckets, then to sort n strings we just spend $O(n)$ time at each level. So, if strings are length L, then the total time is $O(nL)$. The time spent scanning empty buckets could be a problem if r is large, but if we view r as a constant, then just goes into the $O()$.

9.3.2 Least-significant-first (LSF) radix sort

Here is another idea that is easier to implement but trickier to see why it works. Let's switch from strings to numbers since that will make the method a little cleaner.

In this algorithm, we first perform a bucketsort using only the *least* significant digit. That is, we place the keys into buckets according to the ones digit and then concatenate the buckets. Next we bucketsort by the tens digit, then the hundreds and so on. This sounds weird, but the claim is that if we perform each bucketsort in a *stable* manner that doesn't rearrange the order of items that go into the same bucket (a sorting method is called **stable** if it doesn't rearrange equal keys) then this will end up correctly sorting the items. Let's see what happens on an example:

Example: Suppose our input is `[28, 15, 24, 19, 13, 22]`.

- We first sort by the ones digit, producing: `[22, 13, 24, 15, 28, 19]`.
- Now we sort by the tens digit using a stable bucketsort (so the relative order of items with the same tens digit remains unchanged), producing: `[13, 15, 19, 22, 24, 28]`.

[1] If we are sorting numbers, we need to pad to the left with zeros to have the correct semantics.

Why does the algorithm work? Let's prove by induction that after the ith pass, the items are correctly sorted according to the least i digits. This is clearly true for the base case $i = 1$. Now, let's do the general case. We know that after the ith pass, the items that differ in the ith digit will be in the desired order with respect to each other (because we just sorted them by that digit!) but what about the items that are equal in this digit? Well, by induction, these were in the desired order *before* we began the ith pass, and since our bucketsort is *stable*, they remain in the correct order afterwards. So we are done.[2]

If numbers have L digits, then running time is $O(L(r + n)) = O(Ln)$ if $r = O(n)$.

Advantages: this method is easy to implement since there is no need to keep buckets separate or even to do recursion: We just have a loop that for `j = L` down to `1` calls `bucketsort(A,j)` which does a bucketsort using the jth character of each string for sorting.

Relation to bounds on comparison-based sorting: If we have n different numbers, then their length is at least $\log_r n$. If all have length $O(\log_r n)$, then the running time is $O(n \log_r n)$. The reason we get this instead of $n \log_2 n$ is we are using indirect-addressing to make an r-way decision in 1 time step. On the negative side, if some keys are much longer than others, then L could be a lot bigger than $\log_r n$. On the positive side, each operation is just an operation on a single digit.

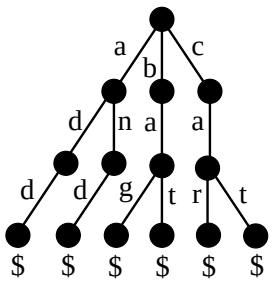
9.4 Tries

The word *trie* comes from re*trie*val. These are also called *digital search trees.* Tries are to MSF radix sort like binary search trees are to quicksort.

In a trie, you put letters on the edges and you walk down the tree reading off the word. In particular, each node will have an array of size r (e.g., $r = 26$ or 128 or 256) of child pointers. To store a string, you just follow down the associated child for each letter in the string from first to last. For instance, say we wanted to store the words:

{`and, bat, add, bag, cat, car`}

When doing an insert, we end each string with "$" (a special end character) in case some strings are substrings of others. To do a lookup, we just walk down the tree letter-by-letter and then see if the node we get to at the end has a "$" in it. (If we ever get to a null pointer, the we know the key is not there — e.g., if we look up "apple" then we will notice in the 2nd step that it can't possibly be in the trie). For instance, in the above example, we would have:



[2]If keys are strings, and they have different lengths, then to match the usual notion of what we mean by "sorted in alphabetical order", we should pad them to the right with blanks that are defined to be less than 'a'. E.g., {car, card} should be viewed {car␣, card}. This is the flip-side of the previous footnote.

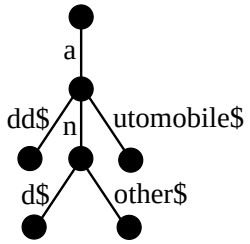
(In terms of implementation, each node has an array of child pointers, so when the picture shows an edge having a letter, like 'c', this really means that the child is the "cth" child.)

The time to do a search is only O(length of key). The same is true for doing an insert, if we view r as a constant. (If r is really big then we should worry about the time spent allocating the arrays of child pointers). So, this is really nice. Also, prefix searching is especially easy (e.g., if you wanted to make a text editor that did word-completion). The main drawback is that the overhead is high because you have so much pointer-following to do. E.g., what if we added "automobile" to the above trie? Also, you have a factor of r extra space since you need to have an array of size r in each node.

Since the design is so nice conceptually, people have thought about ways of making it more practical. In particular, some things you can do are:

- Compress paths that don't branch into a single edge.
- Only split when needed.

So, for instance, the set {add, automobile, and, another} would look like:



This is called a "Patricia tree" (practical algorithm to retrieve information coded in alphanumeric).

Lecture 10

Universal and Perfect Hashing

10.1 Overview

Hashing is a great practical tool, with an interesting and subtle theory too. In addition to its use as a dictionary data structure, hashing also comes up in many different areas, including cryptography and complexity theory. In this lecture we describe two important notions: *universal hashing* (also known as *universal hash function families*) and *perfect hashing*.

Material covered in this lecture includes:

- The formal setting and general idea of hashing.
- Universal hashing.
- Perfect hashing.

10.2 Introduction

We will be looking at the basic dictionary problem we have been discussing so far and will consider two versions, static and dynamic:

- Static: Given a set S of items, we want to store them so that we can do lookups quickly. E.g., a fixed dictionary.
- Dynamic: here we have a sequence of insert, lookup, and perhaps delete requests. We want to do these all efficiently.

For the first problem we could use a sorted array with binary search for lookups. For the second we could use a balanced search tree. However, hashing gives an alternative approach that is often the fastest and most convenient way to solve these problems. For example, suppose you are writing an AI-search program, and you want to store situations that you've already solved (board positions or elements of state-space) so that you don't redo the same computation when you encounter them again. Hashing provides a simple way of storing such information. There are also many other uses in cryptography, networks, complexity theory.

10.3 Hashing basics

The formal setup for hashing is as follows.

- Keys come from some large universe U. (E.g, think of U as the set of all strings of at most 80 ascii characters.)

- There is some set S in U of keys we actually care about (which may be static or dynamic). Let $N = |S|$. Think of N as much smaller than the size of U. For instance, perhaps S is the set of names of students in this class, which is much smaller than 128^{80}.

- We will perform inserts and lookups by having an array A of some size M, and a **hash function** $h : U \to \{0, \ldots, M-1\}$. Given an element x, the idea of hashing is we want to store it in $A[h(x)]$. Note that if U was small (like 2-character strings) then you could just store x in $A[x]$ like in bucketsort. The problem is that U is big: that is why we need the hash function.

- We need a method for resolving collisions. A *collision* is when $h(x) = h(y)$ for two different keys x and y. For this lecture, we will handle collisions by having each entry in A be a linked list. There are a number of other methods, but for the issues we will be focusing on here, this is the cleanest. This method is called *separate chaining*. To insert an element, we just put it at the top of the list. If h is a good hash function, then our hope is that the lists will be small.

One great property of hashing is that all the dictionary operations are incredibly easy to implement. To perform a lookup of a key x, simply compute the index $i = h(x)$ and then walk down the list at $A[i]$ until you find it (or walk off the list). To insert, just place the new element at the top of its list. To delete, one simply has to perform a delete operation on the associated linked list. The question we now turn to is: what do we need for a hashing scheme to achieve good performance?

Desired properties: The main desired properties for a good hashing scheme are:

1. The keys are nicely spread out so that we do not have too many collisions, since collisions affect the time to perform lookups and deletes.

2. $M = O(N)$: in particular, we would like our scheme to achieve property (1) without needing the table size M to be much larger than the number of elements N.

3. The function h is fast to compute. In our analysis today we will be viewing the time to compute $h(x)$ as a constant. However, it is worth remembering in the back of our heads that h shouldn't be too complicated, because that affects the overall runtime.

Given this, the time to lookup an item x is $O(\text{length of list } A[h(x)])$. The same is true for deletes. Inserts take time $O(1)$ no matter what the lengths of the lists. So, we want to be able to analyze how big these lists get.

Basic intuition: One way to spread elements out nicely is to spread them *randomly*. Unfortunately, we can't just use a random number generator to decide where the next element goes because then we would never be able to find it again. So, we want h to be something "pseudorandom" in some formal sense.

We now present some bad news, and then some good news.

Claim 10.1 (Bad news) *For any hash function h, if $|U| \geq (N-1)M+1$, there exists a set S of N elements that all hash to the same location.*

Proof: by the pigeon-hole principle. In particular, to consider the contrapositive, if every location had at most $N-1$ elements of U hashing to it, then U could have size at most $M(N-1)$. ■

So, this is partly why hashing seems so mysterious — how can one claim hashing is good if for any hash function you can come up with ways of foiling it? One answer is that there are a lot of simple hash functions that work well in practice for typical sets S. But what if we want to have a good *worst-case* guarantee?

Here is a key idea: let's use randomization in our *construction* of h, in analogy to randomized quicksort. (h itself will be a deterministic function, of course). What we will show is that for *any* sequence of insert and lookup operations (we won't need to assume the set S of elements inserted is random), if we pick h in this probabilistic way, the performance of h on this sequence will be good in expectation. So, this is the same kind of guarantee as in randomized quicksort or treaps. In particular, this is idea of **universal hashing**.

Once we develop this idea, we will use it for an especially nice application called "perfect hashing".

10.4 Universal Hashing

Definition 10.1 *A randomized algorithm H for constructing hash functions $h : U \to \{1, \ldots, M\}$ is* **universal** *if for all $x \neq y$ in U, we have*

$$\Pr_{h \leftarrow H}[h(x) = h(y)] \leq 1/M.$$

We also say that a set H of hash functions is a **universal hash function family** *if the procedure "choose $h \in H$ at random" is universal. (Here we are identifying the set of functions with the uniform distribution over the set.)*

Theorem 10.2 *If H is universal, then for any set $S \subseteq U$ of size N, for any $x \in U$ (e.g., that we might want to lookup), if we construct h at random according to H, the* **expected** *number of collisions between x and other elements in S is at most N/M.*

Proof: Each $y \in S$ ($y \neq x$) has at most a $1/M$ chance of colliding with x by the definition of "universal". So,

- Let $C_{xy} = 1$ if x and y collide and 0 otherwise.
- Let C_x denote the total number of collisions for x. So, $C_x = \sum_{y \in S, y \neq x} C_{xy}$.

- We know $\mathbf{E}[C_{xy}] = \Pr(x \text{ and } y \text{ collide}) \leq 1/M$.
- So, by linearity of expectation, $\mathbf{E}[C_x] = \sum_y \mathbf{E}[C_{xy}] < N/M$. ■

We now immediately get the following corollary.

Corollary 10.3 *If H is universal then for any* **sequence** *of L insert, lookup, and delete operations in which there are at most M elements in the system at any one time, the expected total cost of the L operations for a random $h \in H$ is only $O(L)$ (viewing the time to compute h as constant).*

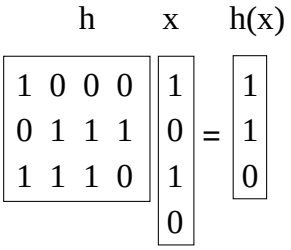
Proof: For any given operation in the sequence, its expected cost is constant by Theorem 10.2, so the expected total cost of the L operations is $O(L)$ by linearity of expectation. ■

Question: can we actually construct a universal H? If not, this this is all pretty vacuous. Luckily, the answer is yes.

10.4.1 Constructing a universal hash family: the matrix method

Let's say keys are u-bits long. Say the table size M is power of 2, so an index is b-bits long with $M = 2^b$.

What we will do is pick h to be a random b-by-u 0/1 matrix, and define $h(x) = hx$, where we do addition mod 2. These matrices are short and fat. For instance:



Claim 10.4 *For $x \neq y$, $\Pr_h[h(x) = h(y)] = 1/M = 1/2^b$.*

Proof: First of all, what does it mean to multiply h by x? We can think of it as adding some of the columns of h (doing vector addition mod 2) where the 1 bits in x indicate which ones to add. (e.g., we added the 1st and 3rd columns of h above)

Now, take an arbitrary pair of keys x, y such that $x \neq y$. They must differ someplace, so say they differ in the ith coordinate and for concreteness say $x_i = 0$ and $y_i = 1$. Imagine we first choose all of h but the ith column. Over the remaining choices of ith column, $h(x)$ is fixed. However, each of the 2^b different settings of the ith column gives a different value of $h(y)$ (in particular, every time we flip a bit in that column, we flip the corresponding bit in $h(y)$). So there is exactly a $1/2^b$ chance that $h(x) = h(y)$. ■

There are other methods to construct universal hash families based on multiplication modulo primes as well (see Section 10.6.1).

The next question we consider is: if we fix the set S, can we find a hash function h such that *all* lookups are constant-time? The answer is *yes*, and this leads to the topic of *perfect hashing*.

10.5 Perfect Hashing

We say a hash function is **perfect** for S if all lookups involve $O(1)$ work. Here are now two methods for constructing perfect hash functions for a given set S.

10.5.1 Method 1: an $O(N^2)$-space solution

Say we are willing to have a table whose size is quadratic in the size N of our dictionary S. Then, here is an easy method for constructing a perfect hash function. Let H be universal and $M = N^2$. Then just pick a random h from H and try it out! The claim is there is at least a 50% chance it will have no collisions.

Claim 10.5 *If H is universal and $M = N^2$, then $\Pr_{h \sim H}(\text{no collisions in } S) \geq 1/2$.*

Proof:

- How many pairs (x, y) in S are there? **Answer:** $\binom{N}{2}$
- For each pair, the chance they collide is $\leq 1/M$ by definition of "universal".
- So, Pr(exists a collision) $\leq \binom{N}{2}/M < 1/2$. ■

This is like the other side to the "birthday paradox". If the number of days is a lot *more* than the number of people squared, then there is a reasonable chance *no* pair has the same birthday.

So, we just try a random h from H, and if we got any collisions, we just pick a new h. On average, we will only need to do this twice. Now, what if we want to use just $O(N)$ space?

10.5.2 Method 2: an $O(N)$-space solution

The question of whether one could achieve perfect hashing in $O(N)$ space was a big open question for some time, posed as "should tables be sorted?" That is, for a fixed set, can you get constant lookup time with only linear space? There was a series of more and more complicated attempts, until finally it was solved using the nice idea of universal hash functions in 2-level scheme.

The method is as follows. We will first hash into a table of size N using universal hashing. This will produce some collisions (unless we are extraordinarily lucky). However, we will then rehash each bin using Method 1, squaring the size of the bin to get zero collisions. So, the way to think of this scheme is that we have a first-level hash function h and first-level table A, and then N second-level hash functions $h_1, \ldots, h_N$ and N second-level tables $A_1, \ldots, A_N$. To lookup an element x, we first compute $i = h(x)$ and then find the element in $A_i[h_i(x)]$. (If you were doing this in practice, you might set a flag so that you only do the second step if there actually were collisions at index i, and otherwise just put x itself into $A[i]$, but let's not worry about that here.)

Say hash function h hashes n_i elements of S to location i. We already argued (in analyzing Method 1) that we can find $h_1, \ldots, h_N$ so that the total space used in the secondary tables is $\sum_i (n_i)^2$. What remains is to show that we can find a first-level function h such that $\sum_i (n_i)^2 = O(N)$. In fact, we will show the following:

Theorem 10.6 *If we pick the initial h from a universal set H, then*

$$\Pr[\sum_i (n_i)^2 > 4N] < 1/2.$$

Proof: We will prove this by showing that $\mathbf{E}[\sum_i (n_i)^2] < 2N$. This implies what we want by Markov's inequality. (If there was even a $1/2$ chance that the sum could be larger than $4N$ then that fact by itself would imply that the expectation had to be larger than $2N$. So, if the expectation is less than $2N$, the failure probability must be less than $1/2$.)

Now, the neat trick is that one way to count this quantity is to count the number of ordered pairs that collide, including an element colliding with itself. E.g, if a bucket has {`d,e,f`}, then `d` collides with each of {`d,e,f`}, `e` collides with each of {`d,e,f`}, and `f` collides with each of {`d,e,f`}, so we get 9. So, we have:

$$\begin{aligned}
\mathbf{E}[\sum_i (n_i)^2] &= \mathbf{E}[\sum_x \sum_y C_{xy}] \quad (C_{xy} = 1 \text{ if } x \text{ and } y \text{ collide, else } C_{xy} = 0) \\
&= N + \sum_x \sum_{y \neq x} \mathbf{E}[C_{xy}] \\
&\leq N + N(N-1)/M \quad \text{(where the } 1/M \text{ comes from the definition of universal)} \\
&< 2N. \quad (\text{since } M = N) \quad \blacksquare
\end{aligned}$$

So, we simply try random h from H until we find one such that $\sum_i n_i^2 < 4N$, and then fixing that function h we find the N secondary hash functions $h_1, \ldots, h_N$ as in method 1.

10.6 Further discussion

10.6.1 Another method for universal hashing

Here is another method for constructing universal hash functions that is a bit more efficient than the matrix method given earlier.

In the matrix method, we viewed the key as a vector of bits. In this method, we will instead view the key x as a vector of integers $[x_1, x_2, \ldots, x_k]$ with the only requirement being that each x_i is in the range $\{0, 1, \ldots, M-1\}$. For example, if we are hashing strings of length k, then x_i could be the ith character (assuming our table size is at least 256) or the ith pair of characters (assuming our table size is at least 65536). Furthermore, we will require our table size M to be a prime number. To select a hash function h we choose k random numbers $r_1, r_2, \ldots, r_k$ from $\{0, 1, \ldots, M-1\}$ and define:

$$h(x) = r_1 x_1 + r_2 x_2 + \ldots + r_k x_k \bmod M.$$

The proof that this method is universal follows the exact same lines as the proof for the matrix method. Let x and y be two distinct keys. We want to show that $\Pr_h(h(x) = h(y)) \leq 1/M$. Since $x \neq y$, it must be the case that there exists some index i such that $x_i \neq y_i$. Now imagine choosing all the random numbers r_j for $j \neq i$ first. Let $h'(x) = \sum_{j \neq i} r_j x_j$. So, once we pick r_i we will have $h(x) = h'(x) + r_i x_i$. This means that we have a collision between x and y exactly when $h'(x) + r_i x_i = h'(y) + r_i y_i \bmod M$, or equivalently when

$$r_i(x_i - y_i) = h'(y) - h'(x) \bmod M.$$

Since M is prime, division by a non-zero value mod M is legal (every integer between 1 and $M-1$ has a multiplicative inverse modulo M), which means there is exactly one value of r_i modulo M for which the above equation holds true, namely $r_i = (h'(y) - h'(x))/(x_i - y_i) \bmod M$. So, the probability of this occuring is exactly $1/M$.

10.6.2 Other uses of hashing

Suppose we have a long sequence of items and we want to see how many *different* items are in the list. What is a good way of doing that?

One way is we can create a hash table, and then make a single pass through our sequence, for each element doing a lookup and then inserting if it is not in the table already. The number of distinct elements is just the number of inserts.

Now what if the list is really huge, so we don't have space to store them all, but we are OK with just an approximate answer. E.g., imagine we are a router and watching a lot of packets go by, and we want to see (roughly) how many different source IP addresses there are.

Here is a neat idea: say we have a hash function h that behaves like a random function, and let's think of $h(x)$ as a real number between 0 and 1. One thing we can do is just keep track of the *minimum* hash value produced so far (so we won't have a table at all). E.g., if keys are 3,10,3,3,12,10,12 and $h(3) = 0.4, h(10) = 0.2, h(12) = 0.7$, then we get 0.2.

The point is: if we pick N random numbers in $[0, 1]$, the expected value of the minimum is $1/(N+1)$. Furthermore, there's a good chance it is fairly close (we can improve our estimate by running several hash functions and taking the median of the minimums).

Question: why use a hash function rather than just picking a random number each time? That is because we care about the number of *different* items, not just the total number of items (that problem is a lot easier: just keep a counter...).