# 15-451 Algorithms, Fall 2007
# Lectures 1-20

**Avrim Blum**
Department of Computer Science
Carnegie Mellon University

October 22, 2007

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

```
       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:

$$X = 2^{n/2}A + B \qquad \boxed{\quad A \quad}\boxed{\quad B \quad}$$

$$Y = 2^{n/2}C + D \qquad \boxed{\quad C \quad}\boxed{\quad D \quad}$$

[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

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

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

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

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

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

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

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

$$T(n) = 3T(n/2) + c'n, \tag{1.4}$$

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

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

$i$th row of the first matrix and compute its dot-product with the $j$th column of the second matrix in order to produce the entry $ij$ in the output) takes time $O(n^3)$. If one breaks down each $n$ 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 $O$, $\Omega$, $\Theta$, and $o$ notation. We then turn to the topic of recurrences, discussing several methods for solving them. Recurrences will come up in many of the algorithms we study, so it is useful to get a good intuition for them right at the start. In particular, we focus on divide-and-conquer style recurrences, which are the most common ones we will see.

Material in this lecture:

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

## 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 $\leq$, $\Omega$ is like $\geq$, $\Theta$ is like $=$, and $o$ is like $<$. There is also a similar notation $\omega$ that corresponds to $>$.

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

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

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:

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

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:

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

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

$$\begin{aligned} T(n) &= 7T(n/7) + n, && (2.4)\\ T(1) &= 0. && (2.5) \end{aligned}$$

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

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

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

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

So, we have verified our guess.

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

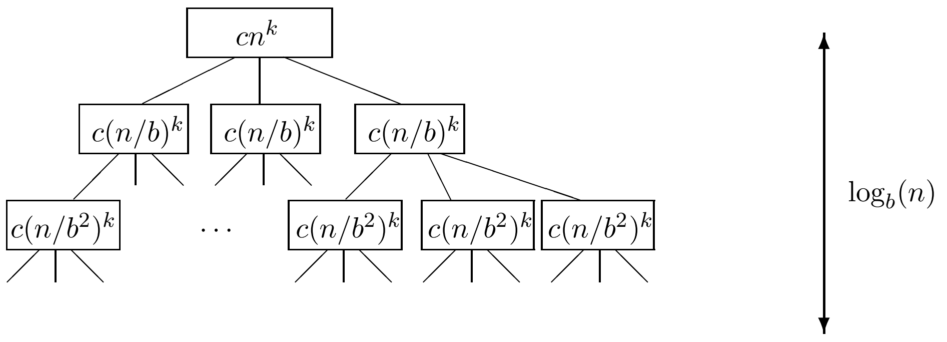
### 2.3.3 Recursion trees, stacking bricks, and a Master Formula

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

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

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

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

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

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

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".

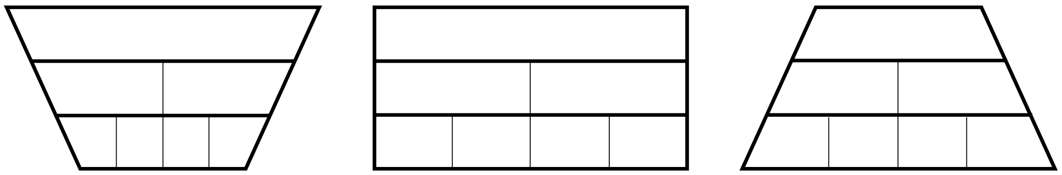
**Theorem 2.1** *The recurrence*

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

*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 do much better "on average" than in the worst case. Sometimes, we can fix this problem 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. There are 36 possible outcomes: it could be that the first die comes up 1 and the second comes up 2, or that the first comes up 2 and the second comes up 1, and so on. Each of these outcomes has probability 1/36 (assuming these are fair dice). Suppose we care about some quantity such as "what is the probability the sum of the

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

dice equals 7?" We can compute that by adding up the probabilities of all the outcomes satisfying this condition (there are six of them, for a total probability of 1/6).

In the language of probability theory, any probabilistic setting is defined by a *sample space* $S$ and a *probability measure* $p$. The points of the sample space are called *elementary events*. E.g., in 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 $i$th card ends in position $i$ and $X_i = 0$ otherwise. These $X_i$ are easy to analyze: $\Pr(X_i = 1) = 1/n$ where $n$ is the number of cards. $\Pr(x_i = 1)$ is also $\mathbf{E}[X_i]$. 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:

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

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 $i$th smallest and $j$th smallest elements in the course of sorting, and 0 if it does not. Let $X$ denote the total number of comparisons made by the algorithm. Since we never compare the same pair of elements twice, we have

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

and therefore,

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

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

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

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

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

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

## 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 $k$th 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 $k$th smallest element in an unsorted array. (Let's say all elements are distinct to avoid the question of what we mean by the $k$th smallest when we have equalities). One way to solve this problem is to sort and then output the $k$th 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 $k$th 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 $k$th smallest in an array of size $n$.*

**Proof:** Let $T(n,k)$ denote the worst-case time to find the $k$th 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.

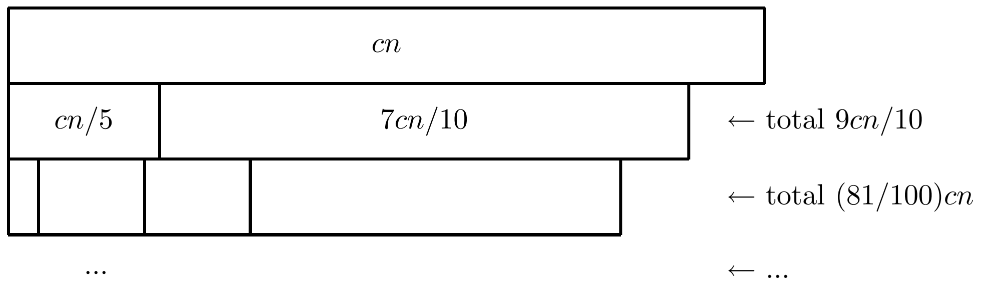


Figure 4.1: Stack of bricks view of recursions tree for recurrence 4.1.

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

# Concrete models and tight bounds I

## 5.1 Overview

In this lecture and the next, we discuss the notion of *lower bounds*: proving that any algorithm for some problem must take *at least* a certain amount of time to solve it. We will examine some simple concrete models of computation, each with a precise definition of what counts as a step, and will attempt to get tight upper and lower bounds for a number of problems. Unlike many of the other lectures, we will try to avoid using $O$, $\Theta$, and $\Omega$, and instead will examine exact quantities as much as possible.

In this lecture we focus on the *comparison model.* We will show that any deterministic comparison-based sorting algorithm must use at least $log_2(n!) \in \Omega(n \log n)$ comparisons to sort an array of $n$ elements in the worst case. We also consider the problem of finding the largest and second-largest element in an array, which has an interesting connection to tennis tournaments and Lewis Carroll.

## 5.2 The idea of lower bounds

So far we have been focusing on designing good algorithms for various problems like sorting and median-finding. A natural question that arises this context is: are these algorithms best possible? If not, how much better could one hope to get? Addressing these questions requires proving that any algorithm must take *at least* a certain amount of time to solve the problem at hand. Statements of this form are called *lower bounds* because they give a lower bound to the question: "how fast an algorithm can one hope to get for the given problem?" In this context, an algorithm with a performance guarantee would be considered an *upper bound.* 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.

Lower bounds are often difficult to show: you cannot necessarily assume, for instance, that the sorting algorithm is going to choose a pivot as in Quicksort, or that it will split the array into pieces as in Mergesort; you need to consider any possible algorithm. To make this task a bit more tractable (as well as more precise) we will look at *concrete computational models*, where we explicitly specify what operations are allowed, and how much they each 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. Then, within these models, we will see how close we can bring the upper and lower bounds together. In these next two lectures, we will consider doing this for several models and problems.

### 5.2.1 Terminology and setup

We will be focusing in this lecture on deterministic algorithms only (in the next lecture we will look at lower bounds for randomized algorithms). By a *worst-case 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 *worst-case lower bound* of $g(n)$, we mean that for any algorithm there exists an input on which it takes at least $g(n)$ steps. As mentioned above, 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 $f(n)$.

## 5.3 Sorting by comparisons

We begin by considering 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 definitions.

**Definition 5.1** *In the* **comparison model***, an input consists of an array* $[a_1, a_2, \ldots, a_n]$ *of* $n$ *items. Items can be examined only by comparing pairs of them. Each comparison ("is* $a_i > a_j$*?") returns YES or NO and counts a 1 time-step. All other computation, such as reordering items based on comparisons made, or incrementing counters, is free.*

**Definition 5.2** *A* comparison-based sorting algorithm *operates in the comparison model. It takes as input an array* $[a_1, a_2, \ldots, a_n]$ *and 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* $\lg(n!) \in \Omega(n \log n)$ *comparisons to sort* $n$ *elements in the worst case.*[1] *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* $\lg(n!)$ *comparisons to sort* $I$*.*

(Note: we will examine the quantity $\lg(n!)$ more carefully after proving the theorem.)

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*

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

(comparison-based) algorithm that might exist. The way we will do this is by showing that in order to sort its input, the sorting algorithm is implicitly playing a game of "20 questions" with the input, trying to figure out in what the order its elements are being given.

**Proof:** Let us fix a set of $n$ distinct input elements. The key to the argument is that (a) there are $n!$ different orders the input elements could be in, and (b) two different input orders cannot both be correctly sorted by the same permutation. Now, suppose that two different initial orderings of these inputs $I_1$, $I_2$, are consistent with all the comparisons the sorting algorithm has made so far. Then, the sorting algorithm cannot yet be done since any permutation it outputs at this point cannot be correct for both $I_1$ and $I_2$ (by observation (b) above). So, the sorting algorithm needs at least implicitly to have pinned down which ordering of the elements was given in the input.

Let $S$ be the set of input orderings consistent with the answers to all comparisons made so far (so, initially, $S$ is the set of all $n!$ possible orderings of the input). We can think of a new comparison as splitting $S$ into two groups: those input orderings for which the answer is YES and those for which the answer is NO. Now, if the answer to each comparison is always the one corresponding to the larger group, then each comparison cuts down the size of $S$ by at most a factor of 2. Since $S$ initially has size $n!$, and at the end the algorithm must have reduced $|S|$ down to 1, the algorithm will need to make at least $\log_2(n!)$ comparisons before it can halt. ■

Let's do an example with $n = 3$ and input elements $\{1, 2, 3\}$. In this case, there are six possible input orderings:

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

Suppose the sorting algorithm first compares $a_1$ with $a_2$. If the answer is that $a_2 > a_1$ then we have narrowed 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 $a_2 > a_3$, which removes just one ordering, leaving us with:

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

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

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

You can also perform the above proof by considering the possible *outputs* of the sorting algorithm. From this perspective, the two key facts we need are: (a) there are $n!$ possible outputs, and (b) for any permutation, there exists an input for which it is the only correct answer. Now we can consider the same 20-questions game where $S$ is the set of *outputs* consistent with all comparisons made so far. Again, each comparison breaks the set of possible outputs into two classes, and the response to the question says which class the correct output is in. By always giving the answer corresponding to the larger class, an adversary forces the algorithm to make at least $\lg(n!)$ comparisons.

Finally, let's take a look at the quantity $\lg(n!)$. We can expand this as: $\lg(n) + \lg(n-1) + \lg(n-2) + \ldots + 1$. The first $n/2$ terms are each at least $\lg(n/2)$, so this quantity is $\Omega(n \log n)$.

**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 two different input orderings can have the same correct answer. E.g., the identity permutation is a correct answer for both the inputs `[1 2 3 4]` and `[2 1 4 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.

### 5.3.1 How tight are our upper and lower bounds?

Let's consider how close our upper and lower bounds are for comparison-based sorting. First, 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:

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

In particular, no algorithm can sort with fewer than $n\lg n - 1.433n$ comparisons.[2]

Let's now consider our upper bounds. Assume $n$ is a power of 2 for simplicity. 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 $(n-1) + 2(n/2-1) + 4(n/4-1) + \ldots + n/2(2-1) = n\lg n - (n-1) < n\lg n$.

### 5.3.2 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 5.2 (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. ■

[2]Since $1.433n$ is a low-order term, sometimes people will write this these bounds on $\lg(n!)$ 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.

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 5.3 (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.

### 5.3.3 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 5.4 (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.

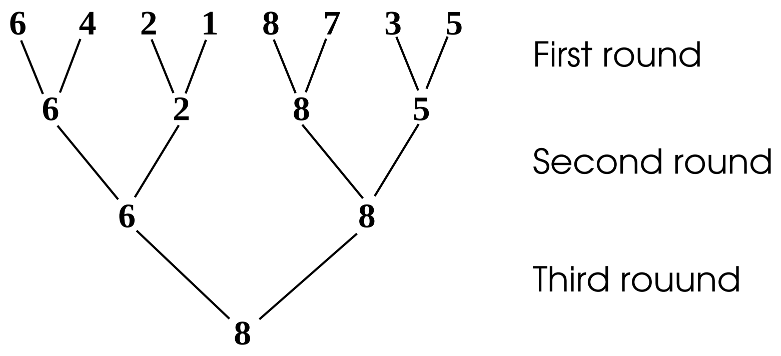
**Claim 5.5 (Upper bound #1)** *$2n-3$ comparisons are sufficient to find the second-largest of $n$ elements.*

**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 5.6 (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 basketball-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 team)? The answer is that the second-best must have been directly compared to the best, and lost.[3] This means there are only $\lg n$ possibilities for the second-highest number, and we can find the maximum of them making only $\lg(n) - 1$ more comparisons. ■

At this point, we have a lower bound of $n - 1$ and an upper bound of $n + \lg(n) - 2$, so they are nearly tight. It turns out that, in fact, the lower bound can be improved to exactly meet the upper bound.[4]

[3] Apparently 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.

[4] First shown by S.S. Kislitsyn "On the selection of the kth element of an ordered set by pairwise comparison," 1964 (in Russian).

# Lecture 6

# Review and Probability Practice

This (optional) lecture will be a review of material so far and practice with probabilistic reasoning. We will look at questions such as the expected number of times you need to flip a coin of bias $p$ until it comes up heads, and the *coupon collector's problem*, which asks what is the expected number of times you need to roll a fair $n$-sided die until you've seen all the sides at least once.

Lecture 7

# Concrete models and tight bounds II

## 7.1 Overview

In this lecture we continue our study of upper and lower bounds for simple concrete models of computation. In each model we give a precise definition of what counts as a step, and try to get tight upper and lower bounds for a number of problems. Specific models and problems examined in this lecture include:

- Sorting in the exchange model.
- The number of probes into a graph needed to determine if the graph is connected (the evasiveness of connectivity).
- Lower bounds for randomized algorithms, using the 2-player game view of algorithm design and analysis.

## 7.2 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 7.1** *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 7.1 (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

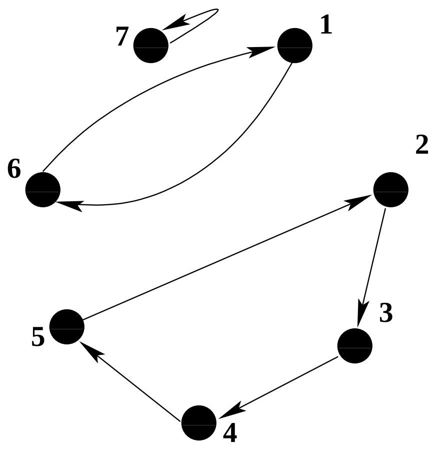


Figure 7.1: Graph in lower bound proof for input `[f c d e b a g]`

originally there. Then in step 2 it swaps the second-smallest item with whatever is currently in location 2, and so on (if in step $k$, the $k$th-smallest item is already in the correct position then we just do a no-op). No step ever undoes any of the previous work, so after $n-1$ steps, the first $n-1$ items are in the correct position. This means the $n$th item must be in the correct position too. ■

But are $n-1$ exchanges necessary in the worst-case? If $n$ is even, and no book is in its correct location, then $n/2$ exchanges are clearly necessary to "touch" all books. But can we show a better lower bound than that?

**Claim 7.2 (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 7.1. Note that this is a special kind of directed graph: it is a permutation — a set of cycles. In particular, every book points to *some* location, perhaps its own location, and every location is pointed to by exactly one book. Now consider the following points:

1. What is the effect of exchanging any two elements (books) that are in the same cycle?

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

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

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

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

   Answer: The final sorted array has $n$ cycles.

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

## 7.3 Query models, and the evasiveness of connectivity

For something totally different, let's now turn to the query complexity of determining whether a given 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 7.3 (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 7.4 (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 algorthm 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. The path $u', \ldots, u, v, \ldots, v'$ is an alternative path from $u'$ to $v'$ that is still possible given the fact that $(u, v)$ has not yet been revealed. 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.

## 7.4 Lower bounds for randomized algorithms

The lower bound proofs we have given so far have assumed the algorithm is *deterministic*. In particular, we used arguments in which the adversary did not commit to the actual instance until the end: e.g., in our lower bound for connectivity, the graph wasn't fully determined until after the algorithm finished; in our lower bound for sorting, we played "20 questions" without having a specific ordering of the input in mind until the end. This is fine for deterministic algorithms because the algorithm would behave in exactly the same way if we were to re-run it on the input that we produced. However, this argument is not legitimate for *randomized* algorithms. For randomized algorithms, an input that caused one sequence of random choices to behave badly might not necessarily cause the algorithm to behave badly in expectation. Instead, the way we will prove lower bounds for randomized algorithms is by giving a fixed randomized strategy for the *adversary*, such that no deterministic algorithm can do well in expectation. This will then imply that no randomized algorithm can do well in expectation either.

Let us see how this plays out with comparison-based sorting. In particular, the randomized strategy we will examine for the adversary is just to present the $n$ items in a random order. We then prove a stronger version of Theorem 5.1 from last time, showing:

**Theorem 7.5** *For any deterministic comparison-based sorting algorithm $A$, its* average-case number of comparisons *(the number of comparisons needed on average to sort a randomly chosen permutation of $n$ distinct elements) is at least $\lfloor \log_2(n!) \rfloor$.*

**Proof:** Let's build out the entire decision tree for algorithm $A$: the tree we get by looking at all possible series of answers that one might get from some ordering of the input. By the argument from last time, each leaf of this tree must correspond to a single input permutation (we can't have two permutations at the same leaf, else the algorithm would not be finished, and since we are proving a lower bound, we can assume that the algorithm does not make any additional comparisons once it has narrowed down the input to a single permutation). So there are $n!$ leaves. The depth of the leaf is the number of comparisons performed by the sorting algorithm on that input, so our goal is to analyze the average leaf depth.

If the tree is completely balanced, then each leaf is at depth at least $\lfloor \log_2(n!) \rfloor$ 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 one is a bit more clever in the proof, one can get rid of the floor in the bound.

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

**Theorem 7.6** *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. To make things easier, let us only consider algorithms that have some finite upper bound $B$ (like $n^2$) on the number of random coin-flips they make. This means we can think of $\mathcal{A}$ as having access to a special "random bit tape" with $B$ bits on it, and every time $\mathcal{A}$ wants to flip a coin, it just pulls the next bit off that tape. In that case, for any *given* string $s$ on that tape, the resulting algorithm $\mathcal{A}_s$ is deterministic, and we can think of $\mathcal{A}$ as just the uniform 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

$$\begin{aligned}
\operatorname*{avg}_{\text{inputs } I} \sum_s [\Pr(s)(\text{Running time of } \mathcal{A}_s \text{ on } I)] &= \sum_s \operatorname*{avg}_I [\Pr(s)(\text{Running time of } \mathcal{A}_s \text{ on } I)] \\
&= \sum_s \Pr(s) \operatorname*{avg}_I(\text{Running time of } \mathcal{A}_s \text{ on } I) \\
&\geq \sum_s \Pr(s) \lfloor \log_2(n!) \rfloor \\
&= \lfloor \log_2(n!) \rfloor. \quad \blacksquare
\end{aligned}$$

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 the $n$ 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 8

# Amortized Analysis

## 8.1 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.

## 8.2 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 8.1** *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.

## 8.3 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 $< 3$.

## 8.4 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 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 8.2** *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.

## 8.5 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 $0 \rightarrow 1$, 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 $1 \rightarrow 0$, 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 $1 \rightarrow 0$ flips, each increment has exactly one $0 \rightarrow 1$ 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 $2n$.

## 8.6 Example #3: What if it costs us $2^k$ to flip the $k$th 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.

## 8.7 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 8.1** *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 9

# Balanced search trees

## 9.1 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).

## 9.2 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 9.1** *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. iven 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.

## 9.3 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 E G I E'` (where `E'` > `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.

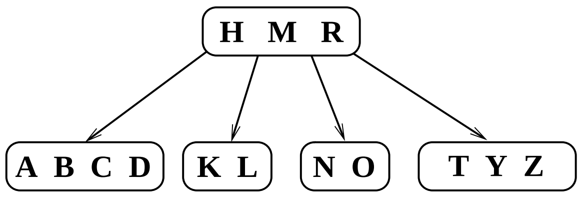
## 9.4 B-trees and 2-3-4 trees

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 $i$th child has items between the $(i-1)$st and $i$th 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:

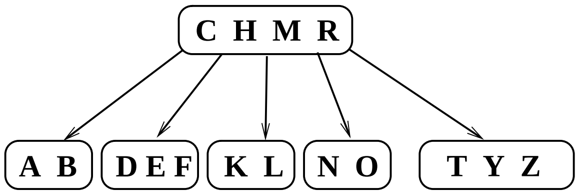
**Lookup:** Just do binary search in the array at the root. This will either return the item you are looking for (in which case you are done) or a pointer to the appropriate child, in which case you recurse on that child.

**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.

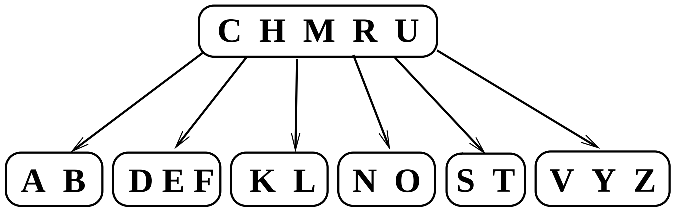
Let's consider the example above. If we insert an "**E**" then that will go into the leftmost leaf, making it full. If we now insert an "**F**", then in the process of walking down the tree we will split the full node, bringing the "**C**" up to the root. So, after inserting the "**F**" we will now have:



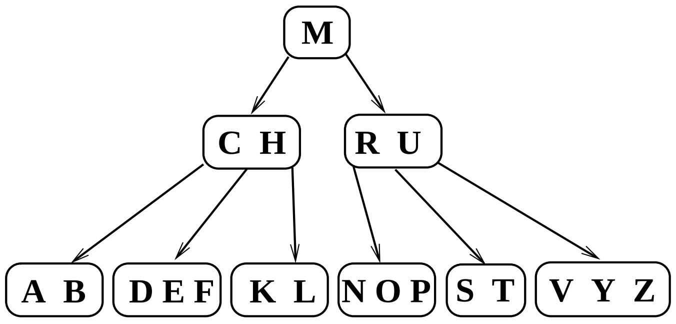
**Question:** We know that performing a split maintains the requirement of at least $t-1$ keys per non-root node (because we split at the median) but is it possible for a split to make the parent *over-full*?

**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:**

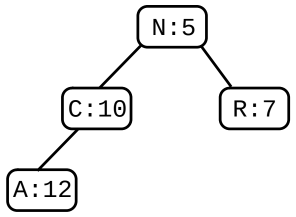
- B-trees are used a lot in databases applications because they fit in nicely with the memory heirarchy when you use a large value of $t$. For instance, if you have 1 billion items and use $t = 1,000$, then you can probably keep the top two levels in fast memory and only make one disk access at the bottom 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).

## 9.5 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*!

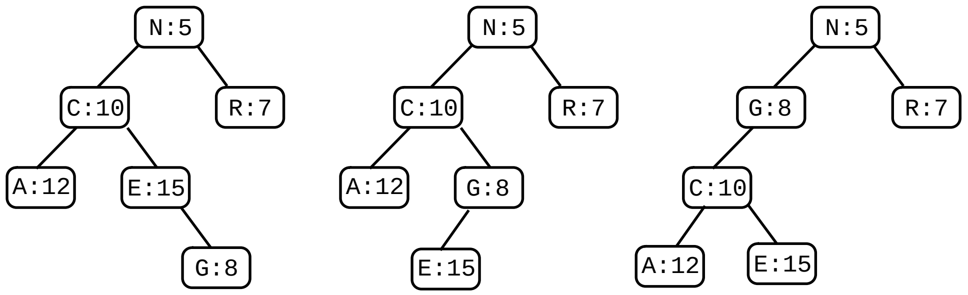
**Question:** Why must such a thing even exist? Given a set of (key, priority) pairs, how do we know it is even *possible* to design a tree so that the keys are in search-tree order and the priorities are in heap order? **Answer:** just sort by priority and run the standard BST algorithm. Moreover, notice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

```
      x                             y
     / \                           / \
   T1   y      <------>           x   T3
       / \                       / \
      T2  T3                    T1  T2
```

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 E'`, 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 10

# Universal and Perfect Hashing

## 10.1 Overview

Our previous data structures treated keys as abstract objects that could only be examined via comparisons. This lecture we will look at hashing, which uses the fact that keys are often objects you can compute a function on, e.g., to map them to an index in an array.

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:

$$\begin{array}{ccc} \text{h} & \text{x} & \text{h(x)} \\ \begin{bmatrix}1&0&0&0\\0&1&1&1\\1&1&1&0\end{bmatrix} & \begin{bmatrix}1\\0\\1\\0\end{bmatrix} = & \begin{bmatrix}1\\1\\0\end{bmatrix}\end{array}$$

**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 $i$th coordinate and for concreteness say $x_i = 0$ and $y_i = 1$. Imagine we first choose all of $h$ but the $i$th column. Over the remaining choices of $i$th column, $h(x)$ is fixed. However, each of the $2^b$ different settings of the $i$th 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* $\Pr_{h \sim H}$(*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 $i$th character (assuming our table size is at least 256) or the $i$th 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...).

# Lecture 11

# Digit-based sorting and data structures

## 11.1 Overview

In this short lecture we examine digit-based sorting and digit-based data structures. These are data structures that, like hashing, are designed for the case that keys are strings of characters or digits. 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.

## 11.2 Introduction

In this lecture we will be looking at algorithms for sorting and storing items when keys are strings of characters or digits.

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 $k$th 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.

## 11.3 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.

### 11.3.1 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 $O()$.

### 11.3.2 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 $i$th 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 $i$th pass, the items that differ in the $i$th 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 $i$th 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 $j$th 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.

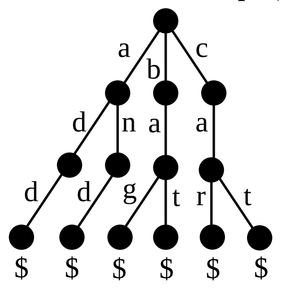
## 11.4 Tries

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:

$$\{\texttt{and, bat, add, bag, cat, car}\}$$

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.

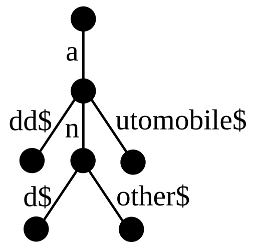
(In terms of implementation, each node has an array of child pointers, so when the picture shows an edge having a letter, like 'c', this really means that the child is the "cth" child.)

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 12

# Dynamic Programming

## 12.1 Overview

Dynamic Programming is a powerful technique that allows one to solve many different types of problems in time $O(n^2)$ or $O(n^3)$ for which a naive approach would take exponential time. In this lecture, we discuss this technique, and present a few key examples. Topics in this lecture include:

- The basic idea of Dynamic Programming.
- Example: Longest Common Subsequence.
- Example: Knapsack.
- Example: Matrix-chain multiplication.

## 12.2 Introduction

Dynamic Programming is a powerful technique that can be used to solve many problems in time $O(n^2)$ or $O(n^3)$ for which a naive approach would take exponential time. (Usually to get running time below that—if it is possible—one would need to add other ideas as well.) Dynamic Programming is a general approach to solving problems, much like "divide-and-conquer" is a general method, except that unlike divide-and-conquer, the subproblems will typically overlap. This lecture we will present two ways of thinking about Dynamic Programming as well as a few examples.

There are several ways of thinking about the basic idea.

**Basic Idea (version 1):** What we want to do is take our problem and somehow break it down into a reasonable number of subproblems (where "reasonable" might be something like $n^2$) in such a way that we can use optimal solutions to the smaller subproblems to give us optimal solutions to the larger ones. Unlike divide-and-conquer (as in mergesort or quicksort) it is OK if our subproblems overlap, so long as there are not too many of them.

## 12.3 Example 1: Longest Common Subsequence

**Definition 12.1** *The* **Longest Common Subsequence (LCS)** *problem is as follows. We are given two strings: string $S$ of length $n$, and string $T$ of length $m$. Our goal is to produce their longest common subsequence: the longest sequence of characters that appear left-to-right (but not necessarility in a contiguous block) in both strings.*

For example, consider:

$$S = \texttt{ABAZDC}$$
$$T = \texttt{BACBAD}$$

In this case, the LCS has length 4 and is the string `ABAD`. Another way to look at it is we are finding a 1-1 matching between some of the letters in $S$ and some of the letters in $T$ such that none of the edges in the matching cross each other.

For instance, this type of problem comes up all the time in genomics: given two DNA fragments, the LCS gives information about what they have in common and the best way to line them up.

Let's now solve the LCS problem using Dynamic Programming. As subproblems we will look at the LCS of a prefix of $S$ and a prefix of $T$, running over all pairs of prefixes. For simplicity, let's worry first about finding *length* of the LCS and then we can modify the algorithm to produce the actual sequence itself..

So, here is the question: say `LCS[i,j]` is the length of the LCS of $S$`[1..i]` with $T$`[1..j]`. How can we solve for `LCS[i,j]` in terms of the LCS's of the smaller problems?

**Case 1:** what if $S[i] \neq T[j]$? Then, the desired subsequence has to ignore one of $S[i]$ or $T[j]$ so we have:

$$\texttt{LCS[i,j]} = \texttt{max(LCS[i}-\texttt{1,j],LCS[i,j}-\texttt{1])}.$$

**Case 2:** what if $S[i] = T[j]$? Then the LCS of $S[1..i]$ and $T[1..j]$ might as well match them up. For instance, if I gave you a common subsequence that matched $S[i]$ to an earlier location in $T$, for instance, you could always match it to $T[j]$ instead. So, in this case we have:

$$\texttt{LCS[i,j]} = 1 + \texttt{LCS[i}-\texttt{1,j}-\texttt{1]}.$$

So, we can just do two loops (over values of $i$ and $j$) , filling in the LCS using these rules. Here's what it looks like pictorially for the example above, with $S$ along the leftmost column and $T$ along the top row.

| | B | A | C | B | A | D |
|---|---|---|---|---|---|---|
| A | 0 | 1 | 1 | 1 | 1 | 1 |
| B | 1 | 1 | 1 | 2 | 2 | 2 |
| A | 1 | 2 | 2 | 2 | 3 | 3 |
| Z | 1 | 2 | 2 | 2 | 3 | 3 |
| D | 1 | 2 | 2 | 2 | 3 | 4 |
| C | 1 | 2 | 3 | 3 | 3 | 4 |

We just fill out this matrix row by row, doing constant amount of work per entry, so this takes $O(mn)$ time overall. The final answer (the length of the LCS of $S$ and $T$) is in the lower-right corner.

**How can we now find the sequence?** To find the sequence, we just walk backwards through matrix starting the lower-right corner. If either the cell directly above or directly to the right contains a value equal to the value in the current cell, then move to that cell (if both to, then chose either one). If both such cells have values strictly less than the value in the current cell, then move diagonally up-left (this correspoints to applying Case 2), and output the associated character. This will output the characters in the LCS in reverse order. For instance, running on the matrix above, this outputs `DABA`.

## 12.4 More on the basic idea, and Example 1 revisited

We have been looking at what is called "bottom-up Dynamic Programming". Here is another way of thinking about Dynamic Programming, that also leads to basically the same algorithm, but viewed from the other direction. Sometimes this is called "top-down Dynamic Programming".

**Basic Idea (version 2)**: Suppose you have a recursive algorithm for some problem that gives you a really bad recurrence like $T(n) = 2T(n-1)+n$. However, suppose that many of the subproblems you reach as you go down the recursion tree are the *same*. Then you can hope to get a big savings if you store your computations so that you only compute each *different* subproblem once. You can store these solutions in an array or hash table. This view of Dynamic Programming is often called *memoizing*.

For example, for the LCS problem, using our analysis we had at the beginning we might have produced the following exponential-time recursive program (arrays start at 1):

```
LCS(S,n,T,m)
{
  if (n==0 || m==0) return 0;
  if (S[n] == T[m]) result = 1 + LCS(S,n-1,T,m-1); // no harm in matching up
  else result = max( LCS(S,n-1,T,m), LCS(S,n,T,m-1) );
  return result;
}
```

This algorithm runs in exponential time. In fact, if `S` and `T` use completely disjoint sets of characters (so that we never have `S[n]==T[m]`) then the number of times that `LCS(S,1,T,1)` is recursively called equals $\binom{n+m-2}{m-1}$.[1] In the memoized version, we store results in a matrix so that any given set of arguments to `LCS` only produces new work (new recursive calls) once. The memoized version begins by initializing `arr[i][j]` to `unknown` for all `i,j`, and then proceeds as follows:

```
LCS(S,n,T,m)
{
  if (n==0 || m==0) return 0;
  if (arr[n][m] != unknown) return arr[n][m];  // <- added this line (*)
  if (S[n] == T[m]) result = 1 + LCS(S,n-1,T,m-1);
  else result = max( LCS(S,n-1,T,m), LCS(S,n,T,m-1) );
  arr[n][m] = result;                          // <- and this line (**)
```

[1]This is the number of different "monotone walks" between the upper-left and lower-right corners of an $n$ by $m$ grid.

```
  return result;
}
```

All we have done is saved our work in line (**) and made sure that we only embark on new recursive calls if we haven't already computed the answer in line (*).

In this memoized version, our running time is now just $O(mn)$. One easy way to see this is as follows. First, notice that we reach line (**) at most $mn$ times (at most once for any given value of the parameters). This means we make at most $2mn$ recursive calls total (at most two calls for each time we reach that line). Any given call of `LCS` involves only $O(1)$ work (performing some equality checks and taking a max or adding 1), so overall the total running time is $O(mn)$.

Comparing bottom-up and top-down dynamic programming, both do almost the same work. The top-down (memoized) version pays a penalty in recursion overhead, but can potentially be faster than the bottom-up version in situations where some of the subproblems never get examined at all. These differences, however, are minor: you should use whichever version is easiest and most intuitive for you for the given problem at hand.

**More about LCS: Discussion and Extensions.** An equivalent problem to LCS is the "minimum edit distance" problem, where the legal operations are insert and delete. (E.g., the unix "diff" command, where $S$ and $T$ are files, and the elements of $S$ and $T$ are lines of text). The minimum edit distance to transform $S$ into $T$ is achieved by doing $|S| - \texttt{LCS}(S,T)$ deletes and $|T| - \texttt{LCS}(S,T)$ inserts.

In computational biology applications, often one has a more general notion of sequence alignment. Many of these different problems all allow for basically the same kind of Dynamic Programming solution.

## 12.5 Example #2: The Knapsack Problem

Imagine you have a homework assignment with different parts labeled A through G. Each part has a "value" (in points) and a "size" (time in hours to complete). For example, say the values and times for our assignment are:

| | A | B | C | D | E | F | G |
|---|---|---|---|---|---|---|---|
| value | 7 | 9 | 5 | 12 | 14 | 6 | 12 |
| time | 3 | 4 | 2 | 6 | 7 | 3 | 5 |

Say you have a total of 15 hours: which parts should you do? If there was partial credit that was proportional to the amount of work done (e.g., one hour spent on problem C earns you 2.5 points) then the best approach is to work on problems in order of points/hour (a greedy strategy). But, what if there is no partial credit? In that case, which parts should you do, and what is the best total value possible?[2]

The above is an instance of the *knapsack problem*, formally defined as follows:

[2]Answer: In this case, the optimal strategy is to do parts A, B, F, and G for a total of 34 points. Notice that this doesn't include doing part C which has the most points/hour!

**Definition 12.2** *In the* **knapsack problem** *we are given a set of $n$ items, where each item $i$ is specified by a size $s_i$ and a value $v_i$. We are also given a size bound $S$ (the size of our knapsack). The goal is to find the subset of items of maximum total value such that sum of their sizes is at most $S$ (they all fit into the knapsack).*

We can solve the knapsack problem in exponential time by trying all possible subsets. With Dynamic Programming, we can reduce this to time $O(nS)$.

Let's do this top down by starting with a simple recursive solution and then trying to memoize it. Let's start by just computing the best possible *total value*, and we afterwards can see how to actually extract the items needed.

```
// Recursive algorithm: either we use the last element or we don't.
Value(n,S)    // S = space left, n = # items still to choose from
{
  if (n == 0) return 0;
  if (s_n > S) result = Value(n-1,S); // can't use nth item
  else result = max{v_n + Value(n-1, S-s_n), Value(n-1, S)};
  return result;
}
```

Right now, this takes exponential time. But, notice that there are only $O(nS)$ *different* pairs of values the arguments can possibly take on, so this is perfect for memoizing. As with the LCS problem, let us initialize a 2-d array `arr[i][j]` to "unknown" for all `i,j`.

```
Value(n,S)
{
  if (n == 0) return 0;
  if (arr[n][S] != unknown) return arr[n][S];  // <- added this
  if (s_n > S) result = Value(n-1,S);
  else result = max{v_n + Value(n-1, S-s_n), Value(n-1, S)};
  arr[n][S] = result;                          // <- and this
  return result;
}
```

Since any given pair of arguments to Value can pass through the array check only once, and in doing so produces at most two recursive calls, we have at most $2n(S+1)$ recursive calls total, and the total time is $O(nS)$.

So far we have only discussed computing the *value* of the optimal solution. How can we get the items? As usual for Dynamic Programming, we can do this by just working backwards: if `arr[n][S]` = `arr[n-1][S]` then we *didn't* use the $n$th item so we just recursively work backwards from `arr[n-1][S]`. Otherwise, we *did* use that item, so we just output the $n$th item and recursively work backwards from `arr[n-1][S-s_n]`. One can also do bottom-up Dynamic Programming.

## 12.6 Example #3: Matrix product parenthesization

Our final example for Dynamic Programming is the *matrix product parenthesization* problem.

Say we want to multiply three matrices $X$, $Y$, and $Z$. We could do it like $(XY)Z$ or like $X(YZ)$. Which way we do the multiplication doesn't affect the final outcome but it *can* affect the running time to compute it. For example, say $X$ is 100x20, $Y$ is 20x100, and $Z$ is 100x20. So, the end result will be a 100x20 matrix. If we multiply using the usual algorithm, then to multiply an $\ell$x$m$ matrix by an $m$x$n$ matrix takes time $O(\ell mn)$. So in this case, which is better, doing $(XY)Z$ or $X(YZ)$?

Answer: $X(YZ)$ is better because computing $YZ$ takes 20x100x20 steps, producing a 20x20 matrix, and then multiplying this by $X$ takes another 20x100x20 steps, for a total of 2x20x100x20. But, doing it the other way takes 100x20x100 steps to compute $XY$, and then multplying this with $Z$ takes another 100x20x100 steps, so overall this way takes 5 times longer. More generally, what if we want to multiply a series of $n$ matrices?

**Definition 12.3** *The* **Matrix Product Parenthesization** *problem is as follows. Suppose we need to multiply a series of matrices: $A_1 \times A_2 \times A_3 \times \ldots \times A_n$. Given the dimensions of these matrices, what is the best way to parenthesize them, assuming for simplicity that standard matrix multiplication is to be used (e.g., not Strassen)?*

There are an exponential number of different possible parenthesizations, in fact $\binom{2(n-1)}{n-1}/n$, so we don't want to search through all of them. Dynamic Programming gives us a better way.

As before, let's first think: how might we do this recursively? One way is that for each possible split for the final multiplication, recursively solve for the optimal parenthesization of the left and right sides, and calculate the total cost (the sum of the costs returned by the two recursive calls plus the $\ell mn$ cost of the final multiplication, where "$m$" depends on the location of that split). Then take the overall best top-level split.

For Dynamic Programming, the key question is now: in the above procedure, as you go through the recursion, what do the subproblems look like and how many are there? Answer: each subproblem looks like "what is the best way to multiply some sub-interval of the matrices $A_i \times \ldots \times A_j$?" So, there are only $O(n^2)$ *different* subproblems.

The second question is now: how long does it take to solve a given subproblem assuming you've already solved all the smaller subproblems (i.e., how much time is spent inside any *given* recursive call)? Answer: to figure out how to best multiply $A_i \times \ldots \times A_j$, we just consider all possible middle points $k$ and select the one that minimizes:

$$\begin{array}{lll} & \text{optimal cost to multiply } A_i \ldots A_k & \leftarrow \text{already computed} \\ + & \text{optimal cost to multiply } A_{k+1} \ldots A_j & \leftarrow \text{already computed} \\ + & \text{cost to multiply the results.} & \leftarrow \text{get this from the dimensions} \end{array}$$

This just takes $O(1)$ work for any given $k$, and there are at most $n$ different values $k$ to consider, so overall we just spend $O(n)$ time per subproblem. So, if we use Dynamic Programming to save our results in a lookup table, then since there are only $O(n^2)$ subproblems we will spend only $O(n^3)$ time overall.

If you want to do this using bottom-up Dynamic Programming, you would first solve for all subproblems with $j - i = 1$, then solve for all with $j - i = 2$, and so on, storing your results in an $n$ by $n$ matrix. The main difference between this problem and the two previous ones we have seen is that any *given* subproblem takes time $O(n)$ to solve rather than $O(1)$, which is why we get $O(n^3)$

total running time. It turns out that by being very clever you can actually reduce this to $O(1)$ amortized time per subproblem, producing an $O(n^2)$-time algorithm, but we won't get into that here.[3]

## 12.7 High-level discussion of Dynamic Programming

What kinds of problems can be solved using Dynamic Programming? One property these problems have is that if the optimal solution involves solving a subproblem, then it uses the *optimal* solution to that subproblem. For instance, say we want to find the shortest path from $A$ to $B$ in a graph, and say this shortest path goes through $C$. Then it must be using the shortest path from $C$ to $B$. Or, in the knapsack example, if the optimal solution does not use item $n$, then it is the optimal solution for the problem in which item $n$ does not exist. The other key property is that there should be only a polynomial number of different subproblems. These two properties together allow us to build the optimal solution to the final problem from optimal solutions to subproblems.

In the top-down view of dynamic programming, the first property above corresponds to being able to write down a recursive procedure for the problem we want to solve. The second property corresponds to making sure that this recursive procedure makes only a polynomial number of *different* recursive calls. In particular, one can often notice this second property by examining the arguments to the recursive procedure: e.g., if there are only two integer arguments that range between 1 and $n$, then there can be at most $n^2$ different recursive calls.

Sometimes you need to do a little work on the problem to get the optimal-subproblem-solution property. For instance, suppose we are trying to find paths between locations in a city, and some intersections have no-left-turn rules (this is particulatly bad in San Francisco). Then, just because the fastest way from $A$ to $B$ goes through intersection $C$, it doesn't necessarily use the fastest way to $C$ because you might need to be coming into $C$ in the correct direction. In fact, the right way to model that problem as a graph is not to have one node per intersection, but rather to have one node per $\langle$*intersection, direction*$\rangle$ pair. That way you recover the property you need.

[3]For details, see Knuth (insert ref).

# Lecture 13

# Graph Algorithms I

## 13.1 Overview

This is the first of several lectures on graph algorithms. We will see how simple algorithms like depth-first-search can be used in clever ways (for a problem known as *topological sorting*) and will see how Dynamic Programming can be used to solve problems of finding shortest paths. Topics in this lecture include:

- Basic notation and terminology for graphs.
- Depth-first-search for Topological Sorting.
- Dynamic-Programming algorithms for shortest path problems: Bellman-Ford (for single-source) and Floyd-Warshall (for all-pairs).

## 13.2 Introduction

Many algorithmic problems can be modeled as problems on graphs. Today we will talk about a few important ones and we will continue talking about graph algorithms for much of the rest of the course.

As a reminder of basic terminology: a graph is a set of *nodes* or *vertices*, with edges between some of the nodes. We will use $V$ to denote the set of vertices and $E$ to denote the set of edges. If there is an edge between two vertices, we call them *neighbors*. The *degree* of a vertex is the number of neighbors it has. Unless otherwise specified, we will not allow self-loops or multi-edges (multiple edges between the same pair of nodes). As is standard with discussing graphs, we will use $n = |V|$, and $m = |E|$, and we will let $V = \{1, \ldots, n\}$.

The above describes an *undirected* graph. In a *directed* graph, each edge now has a direction. For each node, we can now talk about out-neighbors (and out-degree) and in-neighbors (and in-degree). In a directed graph you may have both an edge from $i$ to $j$ and an edge from $j$ to $i$.

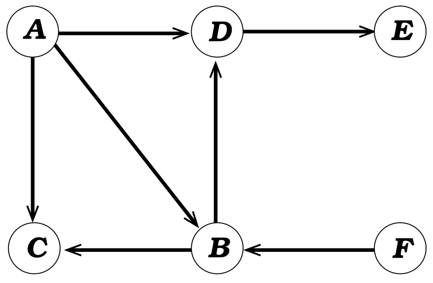
To make sure we are all on the same page, what is the maximum number of total edges in an *undirected* graph? Answer: $\binom{n}{2}$. What about a *directed* graph? Answer: $n(n-1)$.

There are two standard representations for graphs. The first is an *adjacency list*, which is an array of size $n$ where $A[i]$ is the list of out-neighbors of node $i$. The second is an *adjacency matrix*, which is an $n$ by $n$ matrix where $A[i, j] = 1$ iff there is an edge from $i$ to $j$. For an undirected graph, the adjacency matrix will be symmetric. Note that if the graph is reasonably sparse, then an adjacency list will be more compact than an adjacency matrix, because we are only implicitly representing the non-edges. In addition, an adjacency list allows us to access all edges out of some node $v$ in time proportional to the out-degree of $v$. In general, however, the most convenient representation for a graph will depend on what we want to do with it.

We will also talk about weighted graphs where edges may have weights or costs on them. The best notion of an adjacency matrix for such graphs (e.g., should non-edges have weight 0 or weight infinity) will again depend on what problem we are trying to model.

## 13.3 Topological sorting and Depth-first Search

A **Directed Acyclic Graph (DAG)** is a directed graph without any cycles.[1] E.g.,



$$\begin{tikzcd}
A \arrow[r] \arrow[d] \arrow[dr] & D \arrow[r] & E \\
C & B \arrow[l] \arrow[u] & F \arrow[l]
\end{tikzcd}$$

Given a DAG, the **topological sorting** problem is to find an ordering of the vertices such that all edges go forward in the ordering. A typical situation where this problem comes up is when you are given a set of tasks to do with precedence constraints (you need to do $A$ and $F$ before you can do $B$, etc.), and you want to find a legal ordering for performing the jobs. We will assume here that the graph is represented using an adjacency list.

One way to solve the topological sorting problem is to put all the nodes into a priority queue according to in-degree. You then repeatedly pull out the node of minimum in-degree (which should be zero — otherwise you output "graph is not acyclic") and then decrement the keys of each of its out-neighbors. Using a heap to implement the priority queue, this takes time $O(m \log n)$. However, it turns out there is a better algorithm: a simple but clever $O(m + n)$-time approach based on depth-first search.[2]

To be specific, by a *Depth-First Search (DFS) of a graph* we mean the following procedure. First, pick a node and perform a standard depth-first search from there. When that DFS returns, if the whole graph has not yet been visited, pick the next unvisited node and repeat the process.

[1]It would perhaps be more proper to call this an *acyclic directed graph*, but "DAG" is easier to say.

[2]You can also directly improve the first approach to $O(m + n)$ time by using the fact that the minimum always occurs at zero (think about how you might use that fact to speed up the algorithm). But we will instead examine the DFS-based algorithm because it is particularly elegant.

Continue until all vertices have been visited. Specifically, as pseudocode, we have:

```
DFSmain(G):
 For v=1 to n: if v is not yet visited, do DFS(v).

DFS(v):
  mark v as visited. // entering node v
  for each unmarked out-neighbor w of v: do DFS(w).
  return.            // exiting node v.
```

DFS takes time $O(1 + \text{out-degree}(v))$ per vertex $v$, for a total time of $O(m + n)$. Here is now how we can use this to perform a topological sorting:

1. Do depth-first search of $G$, outputting the nodes as you *exit* them.

2. Reverse the order of the list output in Step 1.

**Claim 13.1** *If there is an edge from $u$ to $v$, then $v$ is exited first. (This implies that when we reverse the order, all edges point forward and we have a topological sorting.)*

**Proof:** [In this proof, think of $u = B$, and $v = D$ in the previous picture.] The claim is easy to see if our DFS entered node $u$ before ever entering node $v$, because it will eventually enter $v$ and then exit $v$ before popping out of the recursion for DFS($u$). But, what if we entered $v$ first? In this case, we would exit $v$ before even entering $u$ since there cannot be a path from $v$ to $u$ (else the graph wouldn't be acyclic). So, that's it. ■

## 13.4 Shortest Paths

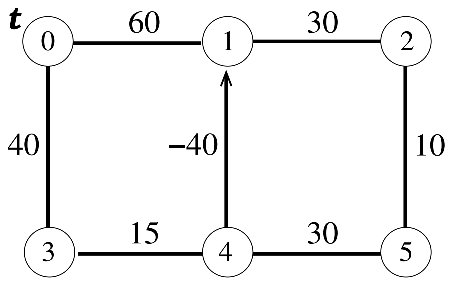
We are now going to turn to another basic graph problem: finding shortest paths in a weighted graph, and we will look at several algorithms based on Dynamic Programming. For an edge $(i, j)$ in our graph, let's use $len(i, j)$ to denote its length. The basic shortest-path problem is as follows:

**Definition 13.1** *Given a weighted, directed graph $G$, a start node $s$ and a destination node $t$, the* **s-t shortest path** *problem is to output the shortest path from $s$ to $t$. The* **single-source** *shortest path problem is to find shortest paths from $s$ to every node in $G$. The (algorithmically equivalent)* **single-sink** *shortest path problem is to find shortest paths from every node in $G$ to $t$.*

We will allow for negative-weight edges (we'll later see some problems where this comes up when using shortest-path algorithms as a subroutine) but will assume no negative-weight cycles (else the shortest path can wrap around such a cycle infinitely often and has length negative infinity). As a shorthand, if there is an edge of length $\ell$ from $i$ to $j$ and also an edge of length $\ell$ from $j$ to $i$, we will often just draw them together as a single undirected edge. So, all such edges must have positive weight.

### 13.4.1 The Bellman-Ford Algorithm

We will now look at a Dynamic Programming algorithm called the Bellman-Ford Algorithm for the single-sink (or single-source) shortest path problem.[3] Let us develop the algorithm using the following example:



How can we use Dyanamic Programming to find the shortest path from all nodes to $t$? First of all, as usual for Dynamic Programming, let's just compute the *lengths* of the shortest paths first, and afterwards we can easily reconstruct the paths themselves. The idea for the algorithm is as follows:

1. For each node $v$, find the length of the shortest path to $t$ that uses at most 1 edge, or write down $\infty$ if there is no such path.

   This is easy: if $v = t$ we get 0; if $(v, t) \in E$ then we get $len(v, t)$; else just put down $\infty$.

2. Now, suppose for all $v$ we have solved for length of the shortest path to $t$ that uses $i - 1$ or fewer edges. How can we use this to solve for the shortest path that uses $i$ or fewer edges?

   Answer: the shortest path from $v$ to $t$ that uses $i$ or fewer edges will first go to some neighbor $x$ of $v$, and then take the shortest path from $x$ to $t$ that uses $i-1$ or fewer edges, which we've already solved for! So, we just need to take the min over all neighbors $x$ of $v$.

3. How far do we need to go? Answer: at most $i = n - 1$ edges.

Specifically, here is pseudocode for the algorithm. We will use `d[v][i]` to denote the length of the shortest path from $v$ to $t$ that uses $i$ or fewer edges (if it exists) and infinity otherwise ("d" for "distance"). Also, for convenience we will use a base case of $i = 0$ rather than $i = 1$.

**Bellman-Ford pseudocode:**

```
initialize d[v][0] = infinity for v != t.  d[t][i]=0 for all i.
for i=1 to n-1:
   for each v != t:
      d[v][i] = $\min_{(v,x)\in E}$ (len(v,x) + d[x][i-1])
For each v, output d[v][n-1].
```

Try it on the above graph!

We already argued for correctness of the algorithm. What about running time? The min operation takes time proportional to the out-degree of $v$. So, the inner for-loop takes time proportional to the sum of the out-degrees of all the nodes, which is $O(m)$. Therefore, the total time is $O(mn)$.

[3]Bellman is credited for inventing Dynamic Programming, and even if the technique can be said to exist inside some algorithms before him, he was the first to distill it as an important technique.

So far we have only calculated the *lengths* of the shortest paths; how can we reconstruct the paths themselves? One easy way is (as usual for DP) to work backwards: if you're at vertex $v$ at distance $d[v]$ from $t$, move to the neighbor $x$ such that $d[v] = d[x] + len(v, x)$. This allows us to reconstruct the path in time $O(m + n)$ which is just a low-order term in the overall running time.

## 13.5 All-pairs Shortest Paths

Say we want to compute the length of the shortest path between *every* pair of vertices. This is called the **all-pairs** shortest path problem. If we use Bellman-Ford for all $n$ possible destinations $t$, this would take time $O(mn^2)$. We will now see two alternative Dynamic-Programming algorithms for this problem: the first uses the matrix representation of graphs and runs in time $O(n^3 \log n)$; the second, called the *Floyd-Warshall* algorithm uses a different way of breaking into subproblems and runs in time $O(n^3)$.

### 13.5.1 All-pairs Shortest Paths via Matrix Products

Given a weighted graph $G$, define the matrix $A = A(G)$ as follows:

- $A[i, i] = 0$ for all $i$.
- If there is an edge from $i$ to $j$, then $A[i, j] = len(i, j)$.
- Otherwise, $A[i, j] = \infty$.

I.e., $A[i, j]$ is the length of the shortest path from $i$ to $j$ using 1 or fewer edges. Now, following the basic Dynamic Programming idea, can we use this to produce a new matrix $B$ where $B[i, j]$ is the length of the shortest path from $i$ to $j$ using 2 or fewer edges?

Answer: yes. $B[i, j] = \min_k(A[i, k] + A[k, j])$. Think about why this is true!

I.e., what we want to do is compute a matrix product $B = A \times A$ except we change "*" to "+" and we change "+" to "min" in the definition. In other words, instead of computing the sum of products, we compute the min of sums.

What if we now want to get the shortest paths that use 4 or fewer edges? To do this, we just need to compute $C = B \times B$ (using our new definition of matrix product). I.e., to get from $i$ to $j$ using 4 or fewer edges, we need to go from $i$ to some intermediate node $k$ using 2 or fewer edges, and then from $k$ to $j$ using 2 or fewer edges.

So, to solve for all-pairs shortest paths we just need to keep squaring $O(\log n)$ times. Each matrix multiplication takes time $O(n^3)$ so the overall running time is $O(n^3 \log n)$.

### 13.5.2 All-pairs shortest paths via Floyd-Warshall

Here is an algorithm that shaves off the $O(\log n)$ and runs in time $O(n^3)$. The idea is that instead of increasing the number of edges in the path, we'll increase the set of vertices we allow as intermediate nodes in the path. In other words, starting from the same base case (the shortest path that uses no intermediate nodes), we'll then go on to considering the shortest path that's allowed to use node 1 as an intermediate node, the shortest path that's allowed to use $\{1, 2\}$ as intermediate nodes, and so on.

```
// After each iteration of the outside loop, A[i][j] = length of the
// shortest i->j path that's allowed to use vertices in the set 1..k
for k = 1 to n do:
  for each i,j do:
    A[i][j] = min( A[i][j], (A[i][k] + A[k][j]);
```

I.e., you either go through node $k$ or you don't. The total time for this algorithm is $O(n^3)$. What's amazing here is how compact and simple the code is!

# Lecture 14

# Graph Algorithms II

## 14.1 Overview

In this lecture we begin with one more algorithm for the shortest path problem, *Dijkstra's algorithm.* We then will see how the basic approach of this algorithm can be used to solve other problems including finding *maximum bottleneck paths* and the *minimum spanning tree* (MST) problem. We will then expand on the minimum spanning tree problem, giving one more algorithm, *Kruskal's algorithm*, which to implement efficiently requires an good data structure for something called the *union-find problem.* Topics in this lecture include:

- Dijkstra's algorithm for shortest paths when no edges have negative weight.
- The Maximum Bottleneck Path problem.
- Minimum Spanning Trees: Prim's algorithm and Kruskal's algorithm.

## 14.2 Shortest paths revisited: Dijkstra's algorithm

Recall the *single-source* shortest path problem: given a graph $G$, and a start node $s$, we want to find the shortest path from $s$ to all other nodes in $G$. These shortest paths can all be described by a tree called the *shortest path tree* from start node $s$.

**Definition 14.1** *A* **Shortest Path Tree** *in $G$ from start node $s$ is a tree (directed outward from $s$ if $G$ is a directed graph) such that the shortest path in $G$ from $s$ to any destination vertex $t$ is the path from $s$ to $t$ in the tree.*

Why must such a tree exist? The reason is that if the shortest path from $s$ to $t$ goes through some intermediate vertex $v$, then it must use a shortest path from $s$ to $v$. Thus, every vertex $t \neq s$ can be assigned a "parent", namely the second-to-last vertex in this path (if there are multiple equally-short paths, pick one arbitrarily), creating a tree. In fact, the Bellman-Ford Dynamic-Programming algorithm from the last class was based on this "optimal subproblem" property.

The first algorithm for today, *Dijkstra's algorithm*, builds the tree outward from $s$ in a greedy fashion. Dijkstra's algorithm is faster than Bellman-Ford. However, it requires that all edge lengths be non-negative. *See if you can figure out where the proof of correctness of this algorithm requires non-negativity.*

We will describe the algorithm the way one views it conceptually, rather than the way one would code it up (we will discuss that after proving correctness).

**Dijkstra's Algorithm:**

Input: Graph $G$, with each edge $e$ having a length $len(e)$, and a start node $s$.

Initialize: tree = $\{s\}$, no edges. Label $s$ as having distance 0 to itself.

Invariant: nodes in the tree are labeled with the correct distance to $s$.

Repeat:

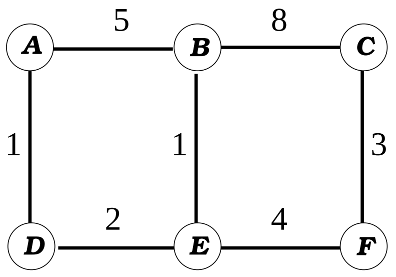
1. For each neighbor $x$ of the tree, compute an (over)-estimate of its distance to $s$:

$$\text{distance}(x) = \min_{e=(v,x):v\in\text{tree}} [\text{distance}(v) + len(e)] \tag{14.1}$$

   In other words, by our invariant, this is the length of the shortest path to $x$ whose only edge not in the tree is the very last edge.

2. Insert the node $x$ of minimum distance into tree, connecting it via the argmin (the edge $e$ used to get distance($x$) in the expression (14.1)).

Let us run the algorithm on the following example starting from vertex $A$:



**Theorem 14.1** *Dijkstra's algorithm correctly produces a shortest path tree from start node s. Specifically, even if some of distances in step 1 are too large, the* minimum *one is correct.*

**Proof:** Say $x$ is the neighbor of the tree of smallest distance($x$). Let $P$ denote the *true* shortest path from $s$ to $x$, choosing the one with the fewest non-tree edges if there are ties. What we need to argue is that the last edge in $P$ must come directly from the tree. Let's argue this by contradiction. Suppose instead the first non-tree vertex in $P$ is some node $y \neq x$. Then, the length of $P$ must be at least distance($y$), and by definition, distance($x$) is smaller (or at least as small if there is a tie). This contradicts the definition of $P$. ■

Did you catch where "non-negativity" came in in the proof? Can you find an example with negative-weight directed edges where Dijkstra's algorithm actually fails?

**Running time:** To implement this efficiently, rather than recomputing the distances every time in step 1, you simply want to update the ones that actually are affected when a new node is added to the tree in step 2, namely the neighbors of the node added. If you use a heap data structure to store the neighbors of the tree, you can get a running time of $O(m \log n)$. In particular, you can start by giving all nodes a distance of infinity except for the start with a distance of 0, and putting all nodes into a min-heap. Then, repeatedly pull off the minimum and update its neighbors, tentatively assigning parents whenever the distance of some node is lowered. It takes linear time to initialize the heap, and then we perform $m$ updates at a cost of $O(\log n)$ each for a total time of $O(m \log n)$.

If you use something called a "Fibonacci heap" (that we're not going to talk about) you can actually get the running time down to $O(m + n \log n)$. The key point about the Fibonacci heap is that while it takes $O(\log n)$ time to remove the minimum element just like a standard heap (an operation we perform $n$ times), it takes only amortized $O(1)$ time to decrease the value of any given key (an operation we perform $m$ times).

## 14.3 Maximum-bottleneck path

Here is another problem you can solve with this type of algorithm, called the "maximum bottleneck path" problem. Imagine the edge weights represent capacities of the edges ("widths" rather than "lengths") and you want the path between two nodes whose minimum width is largest. How could you modify Dijkstra's algorithm to solve this?

To be clear, define the *width* of a path to be the minimum width of any edge on the path, and for a vertex $v$, define widthto($v$) to be the width of the widest path from $s$ to $v$ (say that widthto($s$) $= \infty$). To modify Dijkstra's algorithm, we just need to change the update rule to:

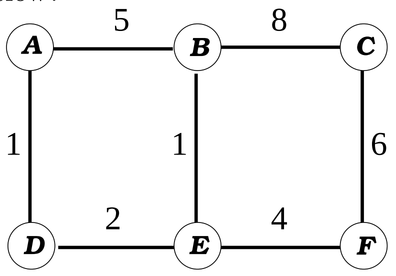
$$\text{widthto}(x) = \max_{e=(v,x):v \in \text{tree}} [\min(\text{widthto}(v), \text{width(e)})]$$

and now put the node $x$ of *maximum* "widthto" into tree, connecting it via the argmax. We'll actually use this later in the course.

## 14.4 Minimum Spanning Trees

A **spanning tree** of a graph is a tree that touches all the vertices (so, it only makes sense in a connected graph). A **minimum spanning tree** (MST) is a spanning tree whose sum of edge lengths is as short as possible (there may be more than one). We will sometimes call the sum of edge lengths in a tree the *size* of the tree. For instance, imagine you are setting up a communication network among a set of sites and you want to use the least amount of wire possible. *Note:* our definition is only for *undirected* graphs.

What is the MST in the graph below?



### 14.4.1 Prim's algorithm

Prim's algorithm is an MST algorithm that works much like Dijkstra's algorithm does for shortest path trees. In fact, it's even simpler (though the correctness proof is a bit trickier).

**Prim's Algorithm:**

1. Pick some arbitrary start node $s$. Initialize tree $T = \{s\}$.
2. Repeatedly add the shortest edge incident to $T$ (the shortest edge having one vertex in $T$ and one vertex not in $T$) until the tree spans all the nodes.

So the algorithm is the same as Dijsktra's algorithm, except you don't add distance($v$) to the length of the edge when deciding which edge to put in next. For instance, what does Prim's algorithm do on the above graph?

Before proving correctness for the algorithm, we first need a useful fact about spanning trees: if you take any spanning tree and add a new edge to it, this creates a cycle. The reason is that there already was one path between the endpoints (since it's a *spanning* tree), and now there are two. If you then remove any edge in the cycle, you get back a spanning tree (removing one edge from a cycle cannot disconnect a graph).

**Theorem 14.2** *Prim's algorithm correctly finds a minimum spanning tree of the given graph.*

**Proof:** We will prove correctness by induction. Let $G$ be the given graph. Our inductive hypothesis will be that the tree $T$ constructed so far is consistent with (is a subtree of) some minimum spanning tree $M$ of $G$. This is certainly true at the start. Now, let $e$ be the edge chosen by the algorithm. We need to argue that the new tree, $T \cup \{e\}$ is also consistent with some minimum spanning tree $M'$ of $G$. If $e \in M$ then we are done ($M' = M$). Else, we argue as follows.

Consider adding $e$ to $M$. As noted above, this creates a cycle. Since $e$ has one endpoint in $T$ and one outside $T$, if we trace around this cycle we must eventually get to an edge $e'$ that goes back in to $T$. We know $len(e') \geq len(e)$ by definition of the algorithm. So, if we add $e$ to $M$ and remove $e'$, we get a new tree $M'$ that is no larger than $M$ was and contains $T \cup \{e\}$, maintaining our induction and proving the theorem. ■

**Running time:** We can implement this in the same was as Dijkstra's algorithm, getting an $O(m \log n)$ running time if we use a standard heap, or $O(m + n \log n)$ running time if we use a Fibonacci heap. The only difference with Dijkstra's algorithm is that when we store the neighbors of $T$ in a heap, we use priority values equal to the shortest edge connecting them to $T$ (rather than the smallest sum of "edge length plus distance of endpoint to $s$").

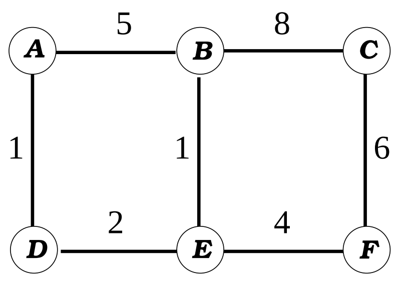
### 14.4.2 Kruskal's algorithm

Here is another algorithm for finding minimum spanning trees called Kruskal's algorithm. It is also greedy but works in a different way.

**Kruskal's Algorithm:**

Sort edges by length and examine them from shortest to longest. Put each edge into the current forest (a forest is just a set of trees) if it doesn't form a cycle with the edges chosen so far.

E.g., let's look at how it behaves in the graph below:



Kruskal's algorithm sorts the edges and then puts them in one at a time so long as they don't form a cycle. So, first the AD and BE edges will be added, then the DE edge, and then the EF edge. The AB edge will be skipped over because it forms a cycle, and finally the CF edge will be added (at that point you can either notice that you have included $n-1$ edges and therefore are done, or else keep going, skipping over all the remaining edges one at a time).

**Theorem 14.3** *Kruskal's algorithm correctly finds a minimum spanning tree of the given graph.*

**Proof:** We can use a similar argument to the one we used for Prim's algorithm. Let $G$ be the given graph, and let $F$ be the forest we have constructed so far (initially, $F$ consists of $n$ trees of 1 node each, and at each step two trees get merged until finally $F$ is just a single tree at the end). Assume by induction that there exists an MST $M$ of $G$ that is consistent with $F$, i.e., all edges in $F$ are also in $M$; this is clearly true at the start when $F$ has no edges. Let $e$ be the next edge added by the algorithm. Our goal is to show that there exists an MST $M'$ of $G$ consistent with $F \cup \{e\}$.

If $e \in M$ then we are done ($M' = M$). Else add $e$ into $M$, creating a cycle. Since the two endpoints of $e$ were in different trees of $F$, if you follow around the cycle you must eventually traverse some edge $e' \neq e$ whose endpoints are also in two different trees of $F$ (because you eventually have to get back to the node you started from). Now, both $e$ and $e'$ were eligible to be added into $F$, which by definition of our algorithm means that $len(e) \leq len(e')$. So, adding $e$ and removing $e'$ from $M$ creates a tree $M'$ that is also a MST and contains $F \cup \{e\}$, as desired. ■

**Running time:** The first step is sorting the edges by length which takes time $O(m \log m)$. Then, for each edge we need to test if it connects two different components. This seems like it should be a real pain: how can we tell if an edge has both endpoints in the same component? It turns out there's a nice data structure called the *Union-Find*' data structure for doing this operation. It is so efficient that it actually will be a low-order cost compared to the sorting step.

We will talk about the union-find problem in the next class, but just as a preview, the *simpler* version of that data structure takes time $O(m + n \log n)$ for our series of operations. This is already good enough for us, since it is low-order compared to the sorting time. There is also a more sophisticated version, however, whose total time is $O(m \lg^* n)$, in fact $O(m\alpha(n))$, where $\alpha(n)$ is the inverse-Ackermann function that grows even more slowly than $\lg^*$.

What is $\lg^*$? $\lg^*(n)$ is the number of times you need to take $\log_2$ until you get down to 1. So,

$$\begin{aligned}
\lg^*(2) &= 1 \\
\lg^*(2^2 = 4) &= 2 \\
\lg^*(2^4 = 16) &= 3 \\
\lg^*(2^{16} = 65536) &= 4 \\
\lg^*(2^{65536}) &= 5.
\end{aligned}$$

I won't define Ackerman, but to get $\alpha(n)$ up to 5, you need $n$ to be at least a stack of 256 2's.

# Lecture 15

# Graph Algorithms III: Union-Find

## 15.1 Overview

In this lecture we describe the *union-find* problem. This is a problem that captures the key task we had to solve in order to efficiently implement Kruskal's algorithm. We then give two data structures for it with good amortized running time.

## 15.2 Motivation

To motivate the union-find problem, let's recall Kruskal's minimum spanning tree algorithm.

**Kruskal's Algorithm (recap):**
Sort edges by length and examine them from shortest to longest. Put each edge into the current forest if it doesn't form a cycle with the edges chosen so far.

We argued correctness last time. Today, our concern is running time. The initial step takes time $O(|E| \log |E|)$ to sort. Then, for each edge, we need to test if it connects two different components. If it does, we will insert the edge, merging the two components into one; if it doesn't (the two endpoints are in the same component), then we will skip this edge and go on to the next edge. So, to do this efficiently we need a data structure that can support the basic operations of (a) determining if two nodes are in the same component, and (b) merging two components together. This is the union-find problem.

## 15.3 The Union-Find Problem

The general setting for the union-find problem is that we are maintaining a collection of disjoint sets $\{S_1, S_2, \ldots, S_k\}$ over some universe, with the following operations:

**MakeSet**$(x)$**:** create the set $\{x\}$.

**Union**$(x, y)$**:** replace the set $x$ is in (let's call it $S$) and the set $y$ is in (let's call it $S'$) with the single set $S \cup S'$.

**Find**($x$)**:** return the unique ID for the set containing $x$ (this is just some representative element of this set).

Given these operations, we can implement Kruskal's algorithm as follows. The sets $S_i$ will be the sets of vertices in the different trees in our forest. We begin with MakeSet($v$) for all vertices $v$ (every vertex is in its own tree). When we consider some edge $(v, w)$ in the algorithm, we just test whether Find($v$) equals Find($w$). If they are equal, it means that $v$ and $w$ are already in the same tree so we skip over the edge. If they are not equal, we insert the edge into our forest and perform a Union($v, w$) operation. All together we will do $|V|$ MakeSet operations, $|V| - 1$ Unions, and $2|E|$ Find operations.

**Notation and Preliminaries:** in the discussion below, it will be convenient to define $n$ as the number of MakeSet operations and $m$ as the total number of operations (this matches the number of vertices and edges in the graph up to constant factors, and so is a reasonable use of $n$ and $m$). Also, it is easiest to think conceptually of these data structures as adding fields to the items themselves, so there is never an issue of "how do I locate a given element $v$ in the structure?".

## 15.4 Data Structure 1 (list-based)

Our first data structure is a simple one with a very cute analysis. The total cost for the operations will be $O(m + n \log n)$.

In this data structure, the sets will be just represented as linked lists: each element has a pointer to the next element in its list. However, we will augment the list so that each element also has a pointer directly to head of its list. The head of the list is the representative element. We can now implement the operations as follows:

**MakeSet**($x$)**:** just set `x->head=x`. This takes constant time.

**Find**($x$)**:** just return `x->head`. Also takes constant time.

**Union**($x, y$)**:** To perform a union operation we merge the two lists together, and reset the head pointers on one of the lists to point to the head of the other.

Let $A$ be the list containing $x$ and $B$ be the list containing $y$, with lengths $L_A$ and $L_B$ respectively. Then we can do this in time $O(L_A + L_B)$ by appending $B$ onto the end of $A$ as follows. We first walk down $A$ to the end, and set the final `next` pointer to point to `y->head`. This takes time $O(L_A)$. Next we go to `y->head` and walk down $B$, resetting head pointers of elements in $B$ to point to `x->head`. This takes time $O(L_B)$.

Can we reduce this to just $O(L_B)$? Yes. Instead of appending $B$ onto the end of $A$, we can just splice $B$ into the middle of $A$, at $x$. I.e., let `z=x->next`, set `x->next=y->head`, then walk down $B$ as above, and finally set the final `next` pointer of $B$ to `z`.

Can we reduce this to $O(\min(L_A, L_B))$? Yes. Just store the length of each list in the head. Then compare and insert the shorter list into the middle of the longer one. Then update the length count to $L_A + L_B$.

We now prove this simple data structure has the running time we wanted.

**Theorem 15.1** *The above algorithm has total running time $O(m + n \log n)$.*

**Proof:** The Find and MakeSet operations are constant time so they are covered by the $O(m)$ term. Each Union operation has cost proportional to the length of the list whose head pointers get updated. So, we need to find some way of analyzing the total cost of the Union operations.

Here is the key idea: we can pay for the union operation by charging $O(1)$ to each element whose head pointer is updated. So, all we need to do is sum up the costs charged to all the elements over the entire course of the algorithm. Let's do this by looking from the point of view of some lowly element $x$. Over time, how many times does $x$ get walked on and have its head pointer updated? The answer is that its head pointer is updated at most $\log n$ times. The reason is that we only update head pointers on the *smaller* of the two lists being joined, so every time $x$ gets updated, the size of the list it is in at least doubles, and this can happen at most $\log n$ times. So, we were able to pay for unions by charging the elements whose head pointers are updated, and no element gets charged more than $O(\log n)$ total, so the total cost for unions is $O(n \log n)$, or $O(m + n \log n)$ for all the operations together. ■

Recall that this is already low-order compared to the $O(m \log m)$ sorting time for Kruskal's algorithm.

## 15.5 Data Structure 2 (tree-based)

Even though the running time of the list-based data structure was pretty fast, let's think of ways we could make it even faster. One idea is that instead of updating all the head pointers in list $B$ (or whichever was shorter) when we perform a Union, we could do this in a lazy way, just pointing the head of $B$ to the head of $A$ and then waiting until we actually perform a find operation on some item $x$ before updating its pointer. This will decrease the cost of the Union operations but will increase the cost of Find operations because we may have to take multiple hops. Notice that by doing this we no longer need the downward pointers: what we have in general is a collection of trees, with all links pointing *up*. Another idea is that rather than deciding which of the two heads (or roots) should be the new one based on the *size* of their sets, perhaps there is some other quantity that would give us better performance. In particular, it turns out we can do better by setting the new root based on which tree has larger *rank*, which we will define in a minute.

We will prove that by implementing the two optimizations described above (lazy updates and union-by-rank), the total cost is bounded above by $O(m \lg^* n)$, where recall that $\lg^* n$ is the number of times you need to take $\log_2$ until you get down to 1. For instance,

$$\lg^*(2^{65536}) = 1 + \lg^*(65536) = 2 + \lg^*(16) = 3 + \lg^*(4) = 4 + \lg^*(2) = 5.$$

So, basically, $\lg^* n$ is never bigger than 5. Technically, the running time of this algorithm is even better: $O(m\alpha(m, n))$ where $\alpha$ is the inverse-Ackermann function which grows even more slowly than $\lg^*$. But the $\lg^* n$ bound is hard enough to prove — let's not go completely overboard!

We now describe the procedure more specifically. Each element (node) will have two fields: a *parent* pointer that points to its parent in its tree (or itself if it is the root) and a rank, which is an integer used to determine which node becomes the new root in a Union operation. The operations are as follows.

**MakeSet**$(x)$**:** set $x$'s rank to 0 and its parent pointer to itself. This takes constant time.

**Find**$(x)$**:** starting from $x$, follow the parent pointers until you reach the root, updating $x$ and all the nodes we pass over to point to the root. This is called *path compression.*

The running time for Find$(x)$ is proportional to (original) distance of $x$ to its root.

**Union**$(x, y)$**:** Let Union$(x, y)$ = Link(Find$(x)$, Find$(y)$), where Link(root1,root2) behaves as follows. If the one of the roots has larger rank than the other, then that one becomes the new root, and the other (smaller rank) root has its parent pointer updated to point to it. If the two roots have *equal* rank, then one of them (arbitrarily) is picked to be the new root *and its rank is increased by 1.* This procedure is called *union by rank.*

**Properties of ranks:** To help us understand this procedure, let's first develop some properties of ranks.

1. The rank of a node is the same as what the height of its subtree would be if we didn't do path compression. This is easy to see: if you take two trees of *different* heights and join them by making the root of the shorter tree into a child of the root of the taller tree, the heights do not change, but if the trees were the *same* height, then the final tree will have its height increase by 1.
2. If $x$ is not a root, then rank$(x)$ is strictly less than the rank of $x$'s parent. We can see this by induction: the Union operation maintains this property, and the Find operation only increases the difference between the ranks of nodes and their parents.
3. The rank of a node $x$ can only change if $x$ is a root. Furthermore, once a node becomes a non-root, it is never a root again. These are immediate from the algorithm.
4. There are at most $n/2^r$ nodes of rank $\geq r$. The reason is that when a (root) node first reaches rank $r$, its tree must have at least $2^r$ nodes (this is easy to see by induction). Furthermore, by property 2, all the nodes in its tree (except for itself) have rank $< r$, and their ranks are never going to change by property 3. This means that (a) for each node $x$ of rank $\geq r$, we can identify a set $S_x$ of at least $2^r - 1$ nodes of smaller rank, and (b) for any two nodes $x$ and $y$ of rank $\geq r$, the sets $S_x$ and $S_y$ are disjoint. Since there are $n$ nodes total, this implies there can be at most $n/2^r$ nodes of rank $\geq r$.

We're now ready to prove the following theorem.

**Theorem 15.2** *The above tree-based algorithm has total running time $O(m \lg^* n)$.*

**Proof:** Let's begin with the easy parts. First of all, the Union does two Find operations plus a constant amount of extra work. So, we only need to worry about the time for the (at most $2m$) Find operations. Second, we can count the cost of a Find operation by charging \$1 for each parent pointer examined. So, when we do a Find$(x)$, if $x$ was a root then pay \$1 (just a constant, so that's ok). If $x$ was a child of a root we pay \$2 (also just a constant, so that's ok also). If $x$ was lower, then the *very rough* idea is that (except for the last \$2) every dollar we spend is shrinking the tree because of our path compression, so we'll be able to amortize this cost somehow. For the remaining part of the proof, we're only going to worry about the steps taken in a Find$(x)$ operation up until we reach the child of the root, since the remainder is just constant time per operation. We'll analyze this using the ranks, and the properties we figured out above.

**Step 1:** let's imagine putting non-root nodes into buckets according to their rank. Bucket 0 contains all non-root nodes of rank 0, bucket 1 has all of rank 1, bucket 2 has ranks 2 through $2^2 - 1$, bucket 3 has ranks $2^2$ through $2^{2^2} - 1$, bucket 4 has ranks $2^{2^2}$ through $2^{2^{2^2}} - 1$, etc. In general, a bucket has ranks $r$ through $2^r - 1$. In total, we have $O(\lg^* n)$ buckets.

The point of this definition is that the number of nodes with rank in any given bucket is at most $n/(\text{upper bound of bucket} + 1)$, by property (4) of ranks above.

**Step 2:** When we walk up the tree in the Find($x$) operation, we need to charge our steps to something. Here's the rule we will use: if the step we take moves us from a node $u$ to a node $v$ such that $v$ is in the *same* bucket as $u$, then we charge it to node $u$ (the walk-ee). But if $v$ is in a higher bucket, we charge that step to $x$ (the walk-er).

The easy part of this is the charge to $x$. We can move up in buckets at most $O(\lg^* n)$ times, since there are only $O(\lg^* n)$ different buckets. So, the total cost charged to the walk-ers (adding up over the $m$ Find operations) is at most $O(m \lg^* n)$.

The harder part is the charge to the walk-ee. The point here is that first of all, the node $u$ charged is not a root, so its rank is never going to change. Secondly, every time we charge this node $u$, the rank of its new parent (after path compression) is at least 1 larger than the rank of its previous parent by property 2. Now, it is true that increasing the rank of $u$'s parent could conceivably happen $\log n$ times, which to us is a "big" number. *But*, once its parent's rank becomes large enough that it is in the next bucket, we never charge node $u$ again as walk-ee. So, the bottom line is that the maximum charge any node $u$ gets is the range of his bucket.

To finish the proof, we just said that a bucket of upper bound $B - 1$ has at most $n/B$ elements in it, and its range is $\leq B$. So, the total charge to all walk-ees in this bucket over the entire course of the algorithm is at most $(n/B)B = n$. (Remember that the only elements being charged are non-roots, so once they start getting charged, their rank is fixed so they can't jump to some other bucket and start getting charged there too.) This means the *total* charge of this kind summed *over all buckets* is at most $n$ times the number of buckets which is $O(n \lg^* n)$. ■ (Whew!)

# Lecture 16

# Network Flow I

## 16.1 Overview

In these next two lectures we are going to talk about an important algorithmic problem called the *Network Flow Problem*. Network flow is important because it can be used to express a wide variety of different kinds of problems. So, by developing good algorithms for solving network flow, we immediately will get algorithms for solving many other problems as well. In Operations Research there are entire courses devoted to network flow and its variants. Topics in today's lecture include:

- The definition of the network flow problem
- The basic Ford-Fulkerson algorithm
- The maxflow-mincut theorem
- The bipartite matching problem

## 16.2 The Network Flow Problem

We begin with a definition of the problem. We are given a directed graph $G$, a start node $s$, and a sink node $t$. Each edge $e$ in $G$ has an associated non-negative *capacity* $c(e)$, where for all non-edges it is implicitly assumed that the capacity is 0. For example, consider the graph in Figure 16.1 below.

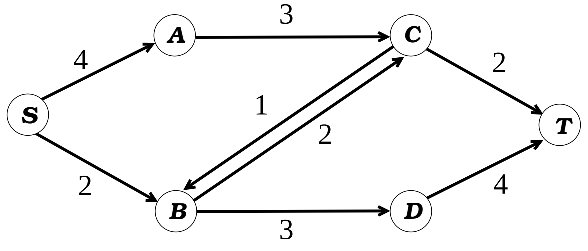


Figure 16.1: A network flow graph.

Our goal is to push as much *flow* as possible from $s$ to $t$ in the graph. The rules are that no edge can have flow exceeding its capacity, and for any vertex except for $s$ and $t$, the flow *in* to the vertex must equal the flow *out* from the vertex. That is,

**Capacity constraint:** On any edge $e$ we have $f(e) \leq c(e)$.

**Flow conservation:** For any vertex $v \notin \{s,t\}$, flow in equals flow out: $\sum_u f(u,v) = \sum_u f(v,u)$.

Subject to these constraints, we want to maximize the total flow into $t$. For instance, imagine we want to route message traffic from the source to the sink, and the capacities tell us how much bandwidth we're allowed on each edge.

E.g., in the above graph, what is the maximum flow from $s$ to $t$? Answer: 5. Using "capacity[flow]" notation, the positive flow looks as in Figure 16.2. Note that the flow can split and rejoin itself.

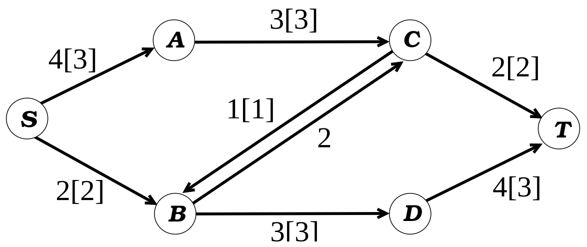


Figure 16.2: A network flow graph with positive flow shown using "capacity[flow]" notation.

How can you see that the above flow was really maximum? Notice, this flow saturates the $a \rightarrow c$ and $s \rightarrow b$ edges, and, if you remove these, you disconnect $t$ from $s$. In other words, the graph has an "$s$-$t$ cut" of size 5 (a set of edges of total capacity 5 such that if you remove them, this disconnects the source from the sink). The point is that any unit of flow going from $s$ to $t$ must take up at least 1 unit of capacity in these pipes. So, we know we're optimal.

We just argued that in general, the maximum $s$-$t$ flow $\leq$ the capacity of the minimum $s$-$t$ cut. An important property of flows, that we will prove as a byproduct of analyzing an algorithm for finding them, is that the maximum $s$-$t$ flow is in fact *equal to* the capacity of the minimum $s$-$t$ cut. This is called the *Maxflow-Mincut Theorem.* In fact, the algorithm will find a flow of some value $k$ and a cut of capacity $k$, which will be proofs that both are optimal!

To describe the algorithm and analysis, it will help to be a bit more formal about a few of these quantities.

**Definition 16.1** *An $s$-$t$* **cut** *is a set of edges whose removal disconnects $t$ from $s$. Or, formally, a cut is a partition of the vertex set into two pieces $A$ and $B$ where $s \in A$ and $t \in B$. (The edges of the cut are then all edges going from $A$ to $B$).*

**Definition 16.2** *The* **capacity** *of a cut $(A,B)$ is the sum of capacities of edges in the cut. Or, in the formal viewpoint, it is the sum of capacities of all edges going from $A$ to $B$. (Don't include the edges from $B$ to $A$.)*

**Definition 16.3** *It will also be mathematically convenient for any edge $(u,v)$ to define $f(v,u) = -f(u,v)$. This is called* **skew-symmetry**. *(We will think of flowing 1 unit on the edge from $u$ to $v$ as equivalently flowing $-1$ units on the back-edge from $v$ to $u$.)*

The skew-symmetry convention makes it especially easy to add two flows together. For instance, if we have one flow with 1 unit on the edge $(c, b)$ and another flow with 2 units on the edge $(b, c)$, then adding them edge by edge does the right thing, resulting in a net flow of 1 unit from $b$ to $c$. Also, using skew-symmetry, the total flow *out* of a node will always be the negative of the total flow *into* a node, so if we wanted we could rewrite the flow conservation condition as $\sum_u f(u, v) = 0$.

How can we find a maximum flow and prove it is correct? Here's a very natural strategy: find a path from $s$ to $t$ and push as much flow on it as possible. Then look at the leftover capacities (an important issue will be how exactly we define this, but we will get to it in a minute) and repeat. Continue until there is no longer any path with capacity left to push any additional flow on. Of course, we need to *prove* that this works: that we can't somehow end up at a suboptimal solution by making bad choices along the way. This approach, with the correct definition of "leftover capacity", is called the Ford-Fulkerson algorithm.

## 16.3 The Ford-Fulkerson algorithm

The Ford-Fulkerson algorithm is simply the following: while there exists an $s \to t$ path $P$ of positive *residual capacity* (defined below), push the maximum possible flow along $P$. By the way, these paths $P$ are called *augmenting paths*, because you use them to augment the existing flow.

Residual capacity is just the capacity left over given the existing flow, where we will use skew-symmetry to capture the notion that if we push $f$ units of flow on an edge $(u, v)$, this *increases* our ability to push flow on the back-edge $(v, u)$ by $f$.

**Definition 16.4** *Given a flow $f$ in graph $G$, the* **residual capacity** *$c_f(u, v)$ is defined as $c_f(u, v) = c(u, v) - f(u, v)$, where recall that by skew-symmetry we have $f(v, u) = -f(u, v)$.*

For example, given the flow in Figure 16.2, the edge $(s, a)$ has residual capacity 1. The back-edge $(a, s)$ has residual capacity 3, because its original capacity was 0 and we have $f(a, s) = -3$.

**Definition 16.5** *Given a flow $f$ in graph $G$, the* **residual graph** *$G_f$ is the directed graph with all edges of positive residual capacity, each one labeled by its residual capacity. Note: this may include back-edges of the original graph $G$.*

Let's do an example. Consider the graph in Figure 16.1 and suppose we push two units of flow on the path $s \to b \to c \to t$. We then end up with the following residual graph:

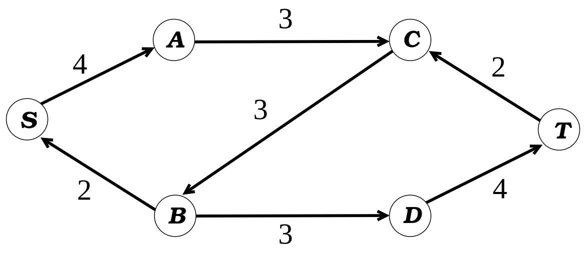


Figure 16.3: Residual graph resulting from pushing 2 units of flow along the path $s$-$b$-$c$-$t$ in the graph in Figure 16.1.

If we continue running Ford-Fulkerson, we see that in this graph the only path we can use to augment the existing flow is the path $s \to a \to c \to b \to d \to t$. Pushing the maximum 3 units on this path we then get the next residual graph, shown in Figure 16.4. At this point there is no

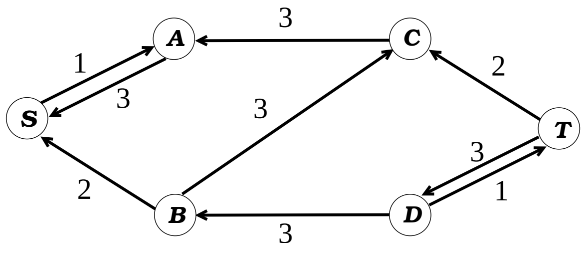


Figure 16.4: Residual graph resulting from pushing 3 units of flow along the path *s-a-c-b-d-t* in the graph in Figure 16.3.

longer a path from $s$ to $t$ so we are done.

We can think of Ford-Fulkerson as at each step finding a new flow (along the augmenting path) and adding it to the existing flow, where by *adding* two flows we mean adding them edge by edge. Notice that the sum of two flows continues to satisfy flow-conservation and skew-symmetry. The definition of residual capacity ensures that the flow found by Ford-Fulkerson is *legal* (doesn't exceed the capacity constraints in the original graph). We now need to prove that in fact it is *maximum.* We'll worry about the number of iterations it takes and how to improve that later.

Note that one nice property of the residual graph is that it means that at each step we are left with same type of problem we started with. So, to implement Ford-Fulkerson, we can use any black-box path-finding method (e.g., DFS).

**Theorem 16.1** *The Ford-Fulkerson algorithm finds a maximum flow.*

**Proof:** Let's look at the final residual graph. This graph must have $s$ and $t$ disconnected by definition of the algorithm. Let $A$ be the component containing $s$ and $B$ be the rest. Let $c$ be the capacity of the $(A, B)$ cut in the *original* graph — so we know we can't do better than $c$.

The claim is that we in fact *did* find a flow of value $c$ (which therefore implies it is maximum). Here's why: let's look at what happens to the residual capacity of the $(A, B)$ cut after each iteration of the algorithm. Say in some iteration we found a path with $k$ units of flow. Then, even if the path zig-zagged between $A$ and $B$, every time we went from $A$ to $B$ we added $k$ to the flow from A to B and subtracted $k$ from the residual capacity of the $(A, B)$ cut, and every time we went from $B$ to $A$ we took away $k$ from this flow and added $k$ to the residual capacity of the cut[1]; moreover, we must have gone from $A$ to $B$ *exactly* one more time than we went from $B$ to $A$. So, the residual capacity of this cut went down by exactly $k$. So, the drop in capacity is equal to the increase in flow. Since at the end the residual capacity is zero (remember how we defined $A$ and $B$) this means the total flow is equal to $c$.

So, we've found a flow of value *equal* to the capacity of this cut. We know we can't do better, so this must be a max flow, and $(A, B)$ must be a minimum cut. ■

[1]This is where we use the fact that if we flow $k$ units on the edge $(u, v)$, then in addition to reducing the residual capacity of the $(u, v)$ edge by $k$ we also *add* $k$ to the residual capacity of the back-edge $(v, u)$.

Notice that in the above argument we actually proved the nonobvious *maxflow-mincut* theorem:

**Theorem 16.2** *In any graph $G$, for any two vertices $s$ and $t$, the maximum flow from $s$ to $t$ equals the capacity of the minimum $(s, t)$-cut.*

We have also proven the *integral-flow theorem*: if all capacities are integers, then there is a maximum flow in which all flows are integers. This seems obvious, but you'll use it to show something that's not at all obvious on homework 5!

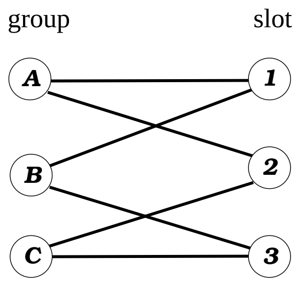
In terms of running time, if all capacities are integers and the maximum flow is $F$, then the algorithm makes at most $F$ iterations (since each iteration pushes at least one more unit of flow from $s$ to $t$). We can implement each iteration in time $O(m+n)$ using DFS. So we get the following result.

**Theorem 16.3** *If the given graph $G$ has integer capacities, Ford-Fulkerson runs in time $O(F(m+n))$ where $F$ is the value of the maximum $s$-$t$ flow.*

In the next lecture we will look at methods for reducing the number of iterations the algorithm can take. For now, let's see how we can use an algorithm for the max flow problem to solve other problems as well: that is, how we can *reduce* other problems to the one we now know how to solve.

## 16.4 Bipartite Matching

Say we wanted to be more sophisticated about assigning groups to homework presentation slots. We could ask each group to list the slots acceptable to them, and then write this as a bipartite graph by drawing an edge between a group and a slot if that slot is acceptable to that group. For example:



This is an example of a **bipartite graph**: a graph with two sides $L$ and $R$ such that all edges go between $L$ and $R$. A **matching** is a set of edges with no endpoints in common. What we want here in assigning groups to time slots is a **perfect matching**: a matching that connects every point in $L$ with a point in $R$. For example, what is a perfect matching in the bipartite graph above?

More generally (say there is no perfect matching) we want a **maximum matching**: a matching with the maximum possible number of edges. We can solve this as follows:

**Bipartite Matching:**

1. Set up a fake "start" node $s$ connected to all vertices in $L$. Connect all vertices in $R$ to a fake "sink" node $T$. Orient all edges left-to-right and give each a capacity of 1.

2. Find a max flow from $s$ to $t$ using Ford-Fulkerson.

3. Output the edges between $L$ and $R$ containing nonzero flow as the desired matching.

This finds a legal matching because edges from $R$ to $t$ have capacity 1, so the flow can't use two edges *into* the same node, and similarly the edges from $s$ to $L$ have capacity 1, so you can't have flow on two edges *leaving* the same node in $L$. It's a *maximum* matching because any matching gives you a flow of the same value: just connect $s$ to the heads of those edges and connect the tails of those edges to $t$. (So if there was a better matching, we wouldn't be at a maximum flow).

What about the number of iterations of path-finding? This is at most the number of edges in the matching since each augmenting path gives us one new edge.

Let's run the algorithm on the above example. Notice a neat fact: say we start by matching $A$ to 1 and $C$ to 3. These are bad choices, but the augmenting path *automatically* undoes them as it improves the flow!

Matchings come up in many different problems like matching up suppliers to customers, or cell-phones to cell-stations when you have overlapping cells. They are also a basic part of other algorithmic problems.

# Lecture 17

# Network Flow II

## 17.1 Overview

The Ford-Fulkerson algorithm discussed in the last class takes time $O(F(n+m))$, where $F$ is the value of the maximum flow, when all capacities are integral. This is fine if all edge capacities are small, but if they are large numbers written in binary then this could even be exponentially large in the description size of the problem. In this lecture we examine some improvements to the Ford-Fulkerson algorithm that produce much better (polynomial) running times. We then consider a generalization of max-flow called the min-cost max flow problem. Specific topics covered include:

- Edmonds-Karp Algorithm #1
- Edmonds-Karp Algorithm #2
- Further improvements
- Min-cost max flow

## 17.2 Network flow recap

Recall that in the network flow problem we are given a directed graph $G$, a source $s$, and a sink $t$. Each edge $(u,v)$ has some capacity $c(u,v)$, and our goal is to find the maximum flow possible from $s$ to $t$.

Last time we looked at the Ford-Fulkerson algorithm, which we used to prove the maxflow-mincut theorem, as well as the integral flow theorem. The Ford-Fulkerson algorithm is a greedy algorithm: we find a path from $s$ to $t$ of positive capacity and we push as much flow as we can on it (saturating at least one edge on the path). We then describe the capacities left over in a "residual graph" and repeat the process, continuing until there are no more paths of positive residual capacity left between $s$ and $t$. Remember, one of the key but subtle points here is how we define the residual graph: if we push $f$ units of flow on an edge $(u,v)$, then the residual capacity of $(u,v)$ goes down by $f$ but also the residual capacity of $(v,u)$ goes *up* by $f$ (since pushing flow in the opposite direction is the same as reducing the flow in the forward direction). We then proved that this in fact finds the maximum flow.

Assuming capacities are integers, the basic Ford-Fulkerson algorithm could make up to $F$ iterations, where $F$ is the value of the maximum flow. Each iteration takes $O(m)$ time to find a path using DFS or BFS and to compute the residual graph. (To reduce notation, let's assume we have pre-processed the graph to delete any disconnected parts so that $m \geq n-1$.) So, the overall total time is $O(mF)$.

This is fine if $F$ is small, like in the case of bipartite matching (where $F \leq n$). However, it's not good if capacities are in binary and $F$ could be very large. In fact, it's not hard to construct an example where a series of bad choices of which path to augment on could make the algorithm take a very long time: see Figure 17.1.

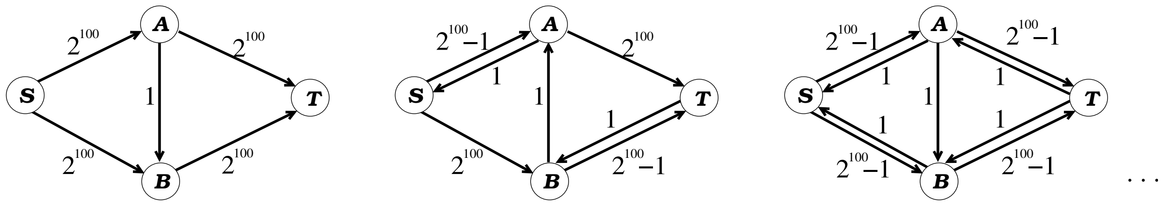


Figure 17.1: A bad case for Ford-Fulkerson. Starting with the graph on the left, we choose the path $s$-$a$-$b$-$t$, producing the residual graph shown in the middle. We then choose the path $s$-$b$-$a$-$t$, producing the residual graph on the right, and so on.

Can anyone think of some ideas on how we could speed up the algorithm? Here are two we can prove something about.

## 17.3 Edmonds-Karp #1

Edmonds-Karp #1 is probably the most natural idea that one could think of. Instead of picking an *arbitrary* path in the residual graph, let's pick the one of largest capacity (we called this the "maximum bottleneck path" a few lectures ago).

**Claim 17.1** *In a graph with maximum $s$-$t$ flow $F$, there must exist a path from $s$ to $t$ with capacity at least $F/m$.*

Can anyone think of a proof?

**Proof:** Suppose we delete all edges of capacity less than $F/m$. This can't disconnect $t$ from $s$ since if it did we would have produced a cut of value less than $F$. So, the graph left over must have a path from $s$ to $t$, and since all edges on it have capacity at least $F/m$, the path itself has capacity at least $F/m$. ■

**Claim 17.2** *Edmonds-Karp #1 makes at most $O(m \log F)$ iterations.*

**Proof:** By Claim 17.1, each iteration adds least a $1/m$ fraction of the "flow still to go" (the maximum flow in the current residual graph) to the flow found so far. Or, equivalently, after each iteration, the "flow still to go" gets reduced by a $(1-1/m)$ factor. So, the question about number of iterations just boils down to: given some number $F$, how many times can you remove a

$1/m$ fraction of the amount remaining until you get down below 1 (which means you are at zero since everything is integral)? Mathematically, for what number $x$ do we have $F(1-1/m)^x < 1$? Notice that $(1-1/m)^m$ is approximately (and always less than) $1/e$. So, $x = m \ln F$ is sufficient: $F(1-1/m)^x < F(1/e)^{\ln F} = 1$. ■

Now, remember we can find the maximum bottleneck path in time $O(m \log n)$, so the overall time used is $O(m^2 \log n \log F)$. You can actually get rid of the "$\log n$" by being a little tricky, bringing this down to $O(m^2 \log F)$.[1]

So, using this strategy, the dependence on $F$ has gone from linear to logarithmic. In particular, this means that even if edge capacities are large integers written in binary, running time is polynomial in the number of bits in the description size of the input.

We might ask, though, can we remove dependence on $F$ completely? It turns out we *can*, using the second Edmonds-Karp algorithm.

## 17.4 Edmonds-Karp #2

The Edmonds-Karp #2 algorithm works by always picking the *shortest* path in the residual graph (the one with the fewest number of edges), rather than the path of maximum capacity. This sounds a little funny but the claim is that by doing so, the algorithm makes at most $mn$ iterations. So, the running time is $O(nm^2)$ since we can use BFS in each iteration. The proof is pretty neat too.

**Claim 17.3** *Edmonds-Karp #2 makes at most $mn$ iterations.*

**Proof:** Let $d$ be the distance from $s$ to $t$ in the current residual graph. We'll prove the result by showing that (a) $d$ never decreases, and (b) every $m$ iterations, $d$ has to increase by at least 1 (which can happen at most $n$ times).

Let's lay out $G$ in levels according to a BFS from $s$. That is, nodes at level $i$ are distance $i$ away from $s$, and $t$ is at level $d$. Now, keeping this layout fixed, let us observe the sequence of paths found and residual graphs produced. Notice that so long as the paths found use only forward edges in this layout, each iteration will cause at least one forward edge to be saturated and removed from the residual graph, and it will add only backward edges. This means first of all that $d$ does not decrease, and secondly that so long as $d$ has not changed (so the paths *do* use only forward edges), at least one forward edge in this layout gets removed. We can remove forward edges at most $m$ times, so within $m$ iterations either $t$ becomes disconnected (and $d = \infty$) or else we must have used a non-forward edge, implying that $d$ has gone up by 1. We can then re-layout the current residual graph and apply the same argument again, showing that the distance between $s$ and $t$ never decreases, and there can be a gap of size at most $m$ between successive increases.

[1]This works as follows. First, let's find the largest power of 2 (let's call it $c = 2^i$) such that there exists a path from $s$ to $t$ in $G$ of capacity at least $c$. We can do this in time $O(m \log F)$ by guessing and doubling (starting with $c = 1$, throw out all edges of capacity less than $c$, and use DFS to check if there is a path from $s$ to $t$; if a path exists, then double the value of $c$ and repeat). Now, instead of looking for *maximum* capacity paths in the Edmonds-Karp algorithm, we just look for $s$-$t$ paths of residual capacity $\geq c$. The advantage of this is we can do this in linear time with DFS. If no such path exists, divide $c$ by 2 and try again. The result is we are always finding a path that is within a factor of 2 of having the maximum capacity (so the bound in Claim 17.2 still holds), but now it only takes us $O(m)$ time per iteration rather than $O(m \log n)$.

Since the distance between $s$ and $t$ can increase at most $n$ times, this implies that in total we have at most $nm$ iterations. ■

## 17.5 Further discussion: Dinic and MPM

Can we do this a little faster? (We may skip this depending on time, and in any case the details here are not so important for this course, but the high level idea is nice.)

The previous algorithm used $O(mn)$ iterations at $O(m)$ time each for $O(m^2n)$ time total. We'll now see how we can reduce to $O(mn^2)$ time and finally to time $O(n^3)$. Here is the idea: given the current BFS layout used in the Edmonds-Karp argument (also called the "level graph"), we'll try in $O(n^2)$ time all at once to find the maximum flow that only uses forward edges. This is sometimes called a "blocking flow". Just as in the analysis above, such a flow guarantees that when we take the residual graph, there is no longer an augmenting path using only forward edges and so the distance to $t$ will have gone up by 1. So, there will be at most $n$ iterations for a total time of $O(n^3)$.

To describe the algorithm, define the *capacity of a vertex* $v$ as $c(v) = \min[c_{in}(v), c_{out}(v)]$, where $c_{in}(v)$ is the sum of capacities of the in-edges to $v$ and $c_{out}(v)$ is the sum of capacities of the out-edges from $v$. The algorithm is now as follows:

1. In $O(m)$ time, create a BFS layout from $s$, and intersect it with a backwards BFS from $t$. Compute the capacities of all nodes.

2. Find the vertex $v$ of minimum capacity $c$. If it's zero, that's great: we can remove $v$ and incident edges, updating capacities of neighboring nodes.

3. If $c$ is not zero, then we greedily pull $c$ units of flow from $s$ to $v$, and then push that flow along to $t$. We then update the capacities, delete $v$ and repeat. Unfortunately, this seems like just another version of the original problem! But, there are two points here:

   (a) Because $v$ had *minimum* capacity, we can do the pulling and pushing in any greedy way and we won't get stuck. E.g., we can do a BFS backward from $v$, pulling the flow level-by-level, so we never examine any edge twice (do an example here), and then a separate BFS forward from $v$ doing the same thing to push the $c$ units forward to $t$.

   This right away gives us an $O(mn)$ algorithm for saturating the level graph ($O(m)$ per node, $n$ nodes), for an overall running time of $O(mn^2)$. So, we're half-way there.

   (b) To improve the running time further, the way we will do the BFS is to examine the in-edges (or out-edges) one at a time, fully saturating the edge before going on to the next one. This means we can allocate our time into two parts: (a) time spent pushing/pulling through edges that get saturated, and (b) time spent on edges that we didn't quite saturate (at most one of these per node). In the BFS we only take time $O(n)$ on type-b operations since we do at most one of these per vertex. We may spend more time on type-a operations, but those result in deleting the edge, so the edge won't be used again in the current level graph. So over *all* vertices $v$ used in this process, the *total* time of these type-a operations is $O(m)$. That means we can saturate the level graph in time $O(n^2 + m) = O(n^2)$.

Our running time to saturate the level graph is $O(n^2 + m) = O(n^2)$. Once we've saturated the level graph, we recompute it (in $O(m)$ time), and re-solve. The total time is $O(n^3)$.

By the way, even this is not the best running time known: the currently best algorithms have performance of almost $O(mn)$.

## 17.6 Min-cost Matchings, Min-cost Max Flows

We talked about the problem of assigning groups to time-slots where each group had a list of acceptable versus unacceptable slots. A natural generalization is to ask: what about preferences? E.g, maybe group $A$ prefers slot 1 so it costs only \$1 to match to there, their second choice is slot 2 so it costs us \$2 to match the group here, and it can't make slot 3 so it costs \$infinity to match the group to there. And, so on with the other groups. Then we could ask for the *minimum cost* perfect matching. This is a perfect matching that, out of all perfect matchings, has the least total cost.

The generalization of this problem to flows is called the *min-cost max flow* problem. Formally, the min-cost max flow problem is defined as follows. We are given a graph $G$ where each edge has a *cost* $w(e)$ in as well as a capacity $c(e)$. The cost of a flow is the sum over all edges of the positive flow on that edge times the cost of the edge. That is,

$$\mathit{cost}(f) = \sum_e w(e) f^+(e),$$

where $f^+(e) = \max[f(e), 0]$. Our goal is to find, out of all possible maximum flows, the one with the least total cost. We can have negative costs (or benefits) on edges too, but let's assume just for simplicity that the graph has no negative-cost cycles. (Otherwise, the min-cost max flow will have little disconnected cycles in it; one can solve the problem without this assumption, but it's conceptually easier with it.)

Min-cost max flow is more general than plain max flow so it can model more things. For example, it can model the min-cost matching problem described above.

There are several ways to solve the min-cost max-flow problem. One way is we can run Ford-Fulkerson, where each time we choose the *least cost* path from $s$ to $t$. In other words, we find the shortest path but using the costs as distances. To do this correctly, when we add a back-edge to some edge $e$ into the residual graph, we give it a cost of $-w(e)$, representing that we get our money back if we undo the flow on it. So, this procedure will create residual graphs with negative-weight edges, but we can still find shortest paths in them using the Bellman-Ford algorithm.

We can argue correctness for this procedure as follows. Remember that we assumed the initial graph $G$ had no negative-cost cycles. Now, even though each augmenting path might add new negative-cost *edges*, it will never create a negative-cost *cycle* in the residual graph. That is because if it did, say containing some back-edge $(v, u)$, that means the path using $(u, v)$ wasn't really shortest since a better way to get from $u$ to $v$ would have been to travel around the cycle instead. So, this implies that the final flow $f$ found has the property that $G_f$ has no negative-cost cycles either. This means that $f$ is optimal: if $g$ was a maximum flow of lower cost, then $g - f$ is a legal circulation in $G_f$ (a flow satisfying flow-in = flow-out at *all* nodes including $s$ and $t$) with negative cost, which means it would have to contain a negative-cost cycle, a contradiction.

The running time for this algorithm is similar to Ford-Fulkerson, except using Bellman-Ford instead of Dijkstra. It is possible to speed it up, but we won't discuss that in this course.

Lecture 18

# Linear Programming

## 18.1 Overview

In this lecture we describe a very general problem called *linear programming* (LP) that can be used to express a wide variety of different kinds of problems. We can use algorithms for linear programming to solve the max-flow problem, solve the min-cost max-flow problem, find minimax-optimal strategies in games, and many other things. We will primarily discuss the setting and how to code up various problems as LPs. At the end, we will briefly describe some of the algorithms for solving linear programming problems. Specific topics include:

- The definition of linear programming and simple examples.
- Using linear programming to solve max flow and min-cost max flow.
- Using linear programming to solve for minimax-optimal strategies in games.
- Algorithms for linear programming.

## 18.2 Introduction

In the last two lectures we looked at:

— Bipartite matching: given a bipartite graph, find the largest set of edges with no endpoints in common.

— Network flow. (More general than bipartite matching)

— Min-Cost Max-flow (even more general than plain max flow).

Today, we'll look at something even more general that we can solve algorithmically: **linear programming**. (Except we won't necessarily be able to get integer solutions, even when the specification of the problem is integral).

Linear Programming is important because it is so expressive: many, *many* problems can be coded up as linear programs. This especially includes problems of allocating resources and business supply-chain applications. In business schools and Operations Research departments there are entire courses devoted to linear programming. We're only going to have time for 1 lecture. So, we will just have time to say what they are, and give examples of encoding problems as LPs. We will only say a tiny bit about algorithms for solving them.

Before defining the problem, let's motivate it with an example:

**Example:** There are 168 hours in a week. Say we want to allocate our time between classes and studying ($S$), fun activities and going to parties ($P$), and everything else ($E$) (eating, sleeping, taking showers, etc). Suppose that to survive we need to spend at least 56 hours on $E$ (8 hours/day). To maintain sanity we need $P + E \geq 70$. To pass our courses, we need $S \geq 60$, but more if don't sleep enough or spend too much time partying: $2S + E - 3P \geq 150$. (E.g., if don't go to parties at all then this isn't a problem, but if we spend more time on P then need to sleep more or study more).

**Q1:** Can we do this? Formally, is there a *feasible* solution?

**A:** Yes. For instance, one feasible solution is: $S = 80, P = 20, E = 68$.

**Q2:** Suppose our notion of happiness is expressed by $2P + E$. What is a feasible solution such that this is maximized? The formula "$2P + E$" is called an *objective function.*

The above is an example of a *linear program.* What makes it linear is that all our constraints are linear inequalities in our variables. E.g., $2S + E - 3P \geq 150$. And, our objective function is also linear. We're not allowed things like requiring $SE \geq 100$, since this wouldn't be a linear inequality.

## 18.3 Definition of Linear Programming

More formally, a linear programming problem is specified as follows.

**Given:**

- $n$ variables $x_1, \ldots, x_n$.

- $m$ linear inequalities in these variables (equalities OK too).

  E.g., $3x_1 + 4x_2 \leq 6$, $0 \leq x_1 \leq 3$, etc.

- We may also have a linear objective function. E.g., $2x_1 + 3x_2 + x_3$.

**Goal:**

- Find values for the $x_i$'s that satisfy the constraints and maximize the objective. (In the "feasibility problem" there is no objective function: we just want to satisfy the constraints.)

For instance, let's write out our time allocation problem this way.

**Variables:** $S$, $P$, $E$.

**Objective:** maximize $2P + E$, subject to

**Constraints:**

$$\begin{aligned} S + P + E &= 168 \\ E &\geq 56 \\ S &\geq 60 \\ 2S + E - 3P &\geq 150 \\ P + E &\geq 70 \\ P &\geq 0 \qquad \text{(can't spend negative time partying)} \end{aligned}$$

## 18.4 Modeling problems as Linear Programs

Here is a typical Operations-Research kind of problem (stolen from Mike Trick's course notes): Suppose you have 4 production plants for making cars. Each works a little differently in terms of labor needed, materials, and pollution produced per car:

| | labor | materials | pollution |
|---|---|---|---|
| plant 1 | 2 | 3 | 15 |
| plant 2 | 3 | 4 | 10 |
| plant 3 | 4 | 5 | 9 |
| plant 4 | 5 | 6 | 7 |

Suppose we need to produce at least 400 cars at plant 3 according to a labor agreement. We have 3300 hours of labor and 4000 units of material available. We are allowed to produce 12000 units of pollution, and we want to maximize the number of cars produced. How can we model this?

To model a problem like this, it helps to ask the following three questions in order: (1) what are the variables, (2) what is our objective in terms of these variables, and (3) what are the constaints. Let's go through these questions for this problem.

1. What are the variables? $x_1, x_2, x_3, x_4$, where $x_i$ denotes the number of cars at plant $i$.
2. What is our objective? maximize $x_1 + x_2 + x_3 + x_4$.
3. What are the constraints?

$$\begin{aligned} x_i &\geq 0 \quad \text{(for all } i\text{)} \\ x_3 &\geq 400 \\ 2x_1 + 3x_2 + 4x_3 + 5x_4 &\leq 3300 \\ 3x_1 + 4x_2 + 5x_3 + 6x_4 &\leq 4000 \\ 15x_1 + 10x_2 + 9x_3 + 7x_4 &\leq 12000 \end{aligned}$$

Note that we are not guaranteed the solution produced by linear programming will be integral, but for problems where the numbers are large like here, it is not a very big deal (we will see problems later, however, where it *is* a very big deal).

## 18.5 Modeling Network Flow

We can model the max flow problem as a linear program too.

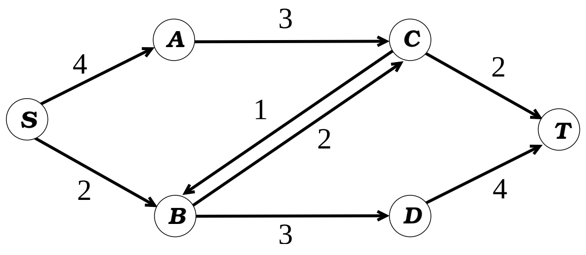
**Variables:** Set up one variable $x_{uv}$ for each edge $(u, v)$. Let's just represent the positive flow since it will be a little easier with fewer constraints.

**Objective:** Maximize $\sum_u x_{ut} - \sum_u x_{tu}$. (maximize the flow into $t$ minus any flow out of $t$)

**Constraints:**

- For all edges $(u, v)$, $0 \leq x_{uv} \leq c(u, v)$. (capacity constraints)
- For all $v \notin \{s, t\}$, $\sum_u x_{uv} = \sum_u x_{vu}$. (flow conservation)

For instance, consider the example from the network-flow lecture:



In this case, our LP is: maximize $x_{ct} + x_{dt}$ subject to the constraints:

$0 \leq x_{sa} \leq 4$, $0 \leq x_{ac} \leq 3$, etc.

$x_{sa} = x_{ac}$, $x_{sb} + x_{cb} = x_{bc} + x_{bd}$, $x_{ac} + x_{bc} = x_{cb} + x_{ct}$, $x_{bd} = x_{dt}$.

**How about min cost max flow?** We can do this in two different ways. One way is to first solve for the maximum flow $f$, ignoring costs. Then, add a *constraint* that flow must equal $f$, and subject to that constraint (plus the original capacity and flow conservation constraints), minimize the linear cost function

$$\sum_{(u,v)\in E} w(u, v)x_{uv},$$

where $w(u, v)$ is the cost of edge $(u, v)$. Alternatively, you can solve this all in one step by adding an edge of infinite capacity and very negative cost from $t$ to $s$, and then just minimizing cost (which will automatically maximize flow).

## 18.6 2-Player Zero-Sum Games

Recall back to homework 1, we looked at a 2-player zero-sum game. We had Alice hiding a nickel or quarter behind her back and Bob guessing, and then based on whether the guess was right or wrong, some money changed hands. This is called a "zero-sum game" because no money is entering or leaving the system (it's all going between Alice and Bob).

A **Minimax Optimal** strategy for a player is a (possibly randomized) strategy with the best guarantee on its expected gain over strategies the opponent could play in response — i.e., it is the strategy you would want to play if you imagine that your opponent knows you well.

Here is another game: Suppose a kicker is shooting a penalty kick against a goalie who is a bit weaker on one side. Let's say the kicker can kick left or right, the goalie can dive left or right, and the payoff matrix for the kicker (the chance of getting a goal) looks as follows:

| Kicker \ Goalie | *left* | *right* |
|---|---|---|
| *left* | 0 | 1 |
| *right* | 1 | 0.5 |

Notice that even though the goalie is weaker on the right, the minimax optimal strategy for the kicker is not to always kick to the right since a goalie who knows the kicker well could then stop half the shots. Instead, the minimax optimal strategy for the kicker is to kick right with probability 2/3 and to kick left with probability 1/3. This guarantees an expected gain of 2/3 no matter how the goalie decides to dive.

How about solving an $n$ by $n$ game? For instance, perhaps a game like this:

| | | |
|---|---|---|
| 20 | −10 | 5 |
| 5 | 10 | −10 |
| −5 | 0 | 10 |

Let's see if we can use linear programming to compute a minimax optimal strategy (say, for the row player). Assume we are given a input an array $M$ where $m_{ij}$ represents the payoff to the row player when the row player plays $i$ and the column player plays $j$.

Informally, we want the variables to be the things we want to figure out, which in this case are the probabilities to put on our different choices $p_1, \ldots, p_n$. These have to form a legal probability distribution, and we can describe this using linear inequalities: namely, $p_1 + \ldots + p_n = 1$ and $p_i \geq 0$ for all $i$.

Our goal is to maximize the worst case (minimum), over all columns our opponent can play, of our expected gain. This is a little confusing because we are maximizing a minimum. However, we can use a trick: we will add one new variable $v$ (representing the minimum), put in *constraints* that our expected gain has to be at least $v$ for every column, and then define our objective to be to maximize $v$. Putting this all together we have:

**Variables:** $p_1, \ldots, p_n$ and $v$.

**Objective:** Maximize $v$.

**Constraints:**

- $p_i \geq 0$ for all $i$, and $\sum_i p_i = 1$. (the $p_i$ form a probability distribution)
- for all columns $j$, we have $\sum_i p_i m_{ij} \geq v$.

## 18.7 Algorithms for Linear Programming

How can we solve linear programs? The standard algorithm for solving LPs is the Simplex Algorithm, developed in the 1940s. It's *not* guaranteed to run in polynomial time, and you *can* come up with bad examples for it, but in general the algorithm runs pretty fast. Only much later in 1980 was it shown that linear programming could be done in polynomial time by something called the Ellipsoid Algorithm (but it tends to be fairly slow in practice). Later on, a faster polynomial-time algorithm called Karmarkar's Algorithm was developed, which is competitive with Simplex. There are many commercial LP packages, for instance LINDO, CPLEX, Solver (in Excel) and others.

We won't have time to describe any of these algorithms in detail. Instead, we will just give some intuition and the high-level idea of how they work by viewing linear programming as a geometrical problem.

Think of an $n$-dimensional space with one coordinate per variable. A solution is a point in this space. An inequality, like $x_1 + x_2 \leq 6$ is saying that we need the solution to be on a specified side of a certain hyperplane. The feasible region is the convex region in space defined by these constraints. Then we want to find the feasible point that is farthest in the "objective" direction.

Let's go to first example with $S$, $P$, and $E$. To make this easier to draw, we can use our first constraint that $S + P + E = 168$ to replace $S$ with $168 - P - E$. This means we can just draw in 2 dimensions, $P$ and $E$. See Figure 18.1.

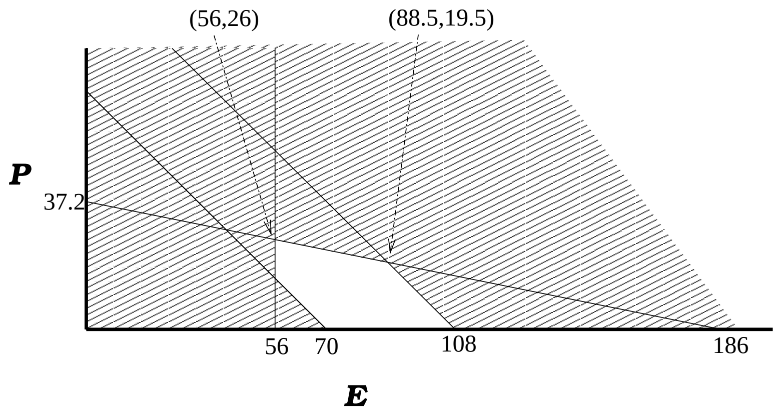


Figure 18.1: Feasible region for our time-planning problem. The constraints are: $E \geq 56$; $P + E \geq 70$; $P \geq 0$; $S \geq 60$ which means $168 - P - E \geq 60$ or $P + E \leq 108$; and finally $2S - 3P + E \geq 150$ which means $2(168 - P - E) - 3P + E \geq 150$ or $5P + E \leq 186$.

We can see from the figure that for the objective of maximizing $P$, the optimum happens at $E = 56, P = 26$. For the objective of maximizing $2P + E$, the optimum happens at $E = 88.5, P = 19.5$.

We can use this geometric view to motivate the algorithms.

**The Simplex Algorithm:** The earliest and most common algorithm use is called the Simplex method. The idea is to start at some "corner" of the feasible region (to make this easier, we can add in so-called "slack variables" that will drop out when we do our optimization). Then we repeatedly do the following step: look at all neighboring corners of our current position and go to the best one (the one for which the objective function is greatest) if it is better than our current position. Stop when we get to a corner where no neighbor has a higher objective value than we currently have.

The key fact here is that (a) since the objective is *linear*, the optimal solution will be at a corner (or maybe multiple corners). Furthermore, (b) there are no local maxima: if you're *not* optimal, then some neighbor of you must have a strictly larger objective value than you have. That's because the feasible region is *convex*. So, the Simplex method is guaranteed to halt at the best solution. The problem is that it is possible for there to be an exponential number of corners and it is possible for Simplex to take an exponential number of steps to converge. But, in practice this usually works well.

**The Ellipsoid Algorithm:** The Ellipsoid Algorithm was invented by Khachiyan in 1980 in Russia.

This algorithms solves just the "feasibility problem," but you can then do binary search with the objective function to solve the optimization problem. The idea is to start with a big ellipse (called an ellipsoid in higher dimensions) that we can be sure contains the feasible region. Then, try the center of the ellipse to see if it violates any constraints. If not, you're done. If it does, then look at some constraint violated. So we know the solution (if any) is contained in the remaining at-most-half-ellipse. Now, find a new smaller ellipse that contains that half of our initial ellipse. We then repeat with the new smaller ellipse. One can show that in each step, you can always create a new smaller ellipse whose volume is smaller, by at least a $(1 - 1/n)$ factor, than the original ellipse. So, every $n$ steps, the volume has dropped by about a factor of $1/e$. One can then show that if you ever get *too* small a volume, as a function of the number of bits used in the coefficients of the constraints, then that means there is no solution after all.

One nice thing about the Ellipsoid Algorithm is you just need to tell if the current solution violates any constraints or not, and if so, to produce one. You don't need to explicitly write them all down. There are some problems that you can write as a linear program with an exponential number of constraints if you had to write them down explicitly, but where there is an fast algorithm to determine if a proposed solution violates any constraints and if so to produce one. For these kinds of problems, the Ellipsoid Algorithm is a good one.

**Karmarkar's Algorithm:** Karmarkar's Algorithms sort of has aspects of both. It works with feasible points but doesn't go from corner to corner. Instead it moves inside the interior of the feasible region. It was one of first of a whole class of so-called "interior-point" methods.

The development of better and better algorithms is a big ongoing area of research. In practice, for all of these algorithms, you get a lot of mileage by using good data structures to speed up the time needed for making each decision.

# Lecture 19

# NP-Completeness I

## 19.1 Overview

In the past few lectures we have looked at increasingly more expressive problems that we were able to solve using efficient algorithms. In this lecture we introduce a class of problems that are so expressive — they are able to model *any* problem in an extremely large class called **NP** — that we believe them to be *intrinsically unsolvable by polynomial-time algorithms.* These are the **NP-complete** problems. What is particularly surprising about this class is that they include many problems that at first glance appear to be quite benign. Specific topics in this lecture include:

- Reductions and expressiveness
- Informal definitions and the ESP problem
- Formal definitions: decision problems, P and NP.
- Circuit-SAT and 3-SAT

## 19.2 Introduction: Reduction and Expressiveness

In the last few lectures have seen a series of increasingly more expressive problems: network flow, min cost max flow, and finally linear programming. These problems have the property that you can code up a lot of different problems in their "language". So, by solving these well, we end up with important tools we can use to solve other problems.

To talk about this a little more precisely, it is helpful to make the following definitions:

**Definition 19.1** *We say that an algorithm runs in* **Polynomial Time** *if, for some constant $c$, its running time is $O(n^c)$, where $n$ is the size of the input.*

In the above definition, "size of input" means "number of bits it takes to write the input down". So, to be precise, when defining a problem and asking whether or not a certain algorithm runs in polynomial time, it is important to say how the input is given. For instance, the basic Ford-Fulkerson algorithm is *not* a polynomial-time algorithm for network flow when edge capacities are written in binary, but both of the Edmonds-Karp algorithms *are* polynomial-time.

**Definition 19.2** *A Problem $A$ is* **poly-time reducible** *to problem $B$ (written as $A \leq_p B$) if, given a polynomial-time algorithm for $B$, we can use it to produce a polynomial-time algorithm for $A$.*[1] *Problem $A$ is* **poly-time equivalent** *to problem $B$ ($A =_p B$) if $A \leq_p B$ and $B \leq_p A$.*

For instance, we gave an efficient algorithm for Bipartite Matching by showing it was poly-time reducible to Max Flow. Notice that it could be that $A \leq_p B$ and yet our fastest algorithm for solving problem $A$ might be slower than our fastest algorithm for solving problem $B$ (because our reduction might involve several calls to the algorithm for problem $B$, or might involve blowing up the input size by a polynomial but still nontrivial amount).

## 19.3 Our first NP-Complete Problem: ESP

Many of the problems we would like to solve have the property that if someone handed us a solution, we could at least check if the solution was correct. For instance the Traveling Salesman Problem asks: "Given a weighted graph $G$ and an integer $k$, does $G$ have a tour that visits all the vertices and has total length at most $k$?" We may not know how to find such a tour quickly, but if someone gave such a tour to us, we could easily check if it satisfied the desired conditions (visited all the vertices and had total length at most $k$). Similarly, for the 3-Coloring problem: "Given a graph $G$, can vertices be assigned colors red, blue, and green so that no two neighbors have the same color?" we could easily check a proposed solution if we had one. Problems of this kind are called **NP** problems (we define the class **NP** formally in Section 19.4).

Let's consider now what would be a problem *so expressive* that if we could solve it, we could solve any problem of this kind. Here is a natural candidate:

**Definition 19.3** **Existence of a verifiable Solution Problem (ESP)***: The input to this problem is in three parts. The first part is a program $V(I, X)$, written in some standard programming language, that has two arguments and outputs a binary YES or NO. The second part is a string $I$ intended as a first argument, and the third part is a bound $b$ written in unary ($b$ bits). Question: does there exist a string $X$, $|X| \leq b$, such that $V(I, X)$ halts in at most $b$ steps and outputs YES?*[2]

Why are we calling this the "existence of a verifiable solution problem" (besides the ESP acronym)?[3] Consider some problem we might want to solve like 3-COLORING: Given a graph $G$ we want to know if there is an assignment of colors to the vertices such that at most three different colors are used and no two neighbors have the same color. We don't know any fast ways of solving that problem, but we can easily write a program $V$ that given inputs $I = G$ and $X =$ an assignment of colors to the vertices, verifies whether $X$ indeed satisfies our requirements (uses at most three colors and gives no two adjacent vertices the same color). Furthermore, this solution-verifier is linear time. So, if we had an algorithm to solve the ESP, we could feed in this $V$, feed in the graph $G$, feed in a bound $b$ that is linear in the size of $G$, and solve the 3-COLORING problem.

[1]Formally, $A \leq_p B$ if problem $A$ is solvable in polynomial time given access to an $O(n^c)$-time black-box oracle for solving arbitrary instances of problem $B$.

[2]The reason for $b$ is that we will want the problem to have the property that if we "had ESP" and could guess the string $X$ (if it exists), we could at least check our guess in polynomial time (by running $V$ for $b$ steps).

[3]Thanks to Manuel Blum for suggesting the acronym.

Technically, the above procedure just tells us if a coloring exists, but doesn't actually produce it. How could we use an algorithm that just answers the YES/NO question to find a coloring? One way is we can pick some vertex $v$ and start adding edges from $v$ to the rest of $G$ one at a time, at each step re-running the above procedure. If the answer is "NO" then we take the edge back out. At the end of this procedure, we can give $v$ and all nodes $v$ is not connected to one color (say, red) and then use BFS to 2-color the remainder.

Let's try another problem. Say we wanted to factor a large integer $N$ written in binary. We don't know any polynomial-time algorithms for solving that problem. But, we can easily write a verifier that given $N$ and a proposed factor $X$, tells us if $X$ is a solution to the problem. In fact, let's modify this slightly (you'll see why in a second) so the verifier takes in an additional integer $k$ (so $I = (N, k)$) and outputs YES iff $X$ divides $N$ *and* $1 < X < k$.

So, if we can solve the ESP, we can tell if $N$ has a factor between 2 and $k - 1$ by feeding $V$ and $I = (N, k)$ into our solver. Then, if we want to actually *find* a factor, we can do binary search on $k$. (That's why we needed the extra input $k$.)

In fact, by design, we could use a polynomial-time algorithm for the ESP to solve *any* problem for which the size of a solution is polynomial in the size of the input, and we have a polynomial-time method for *verifying* if a proposed solution is correct: we just write down the verifier $V$ and then make the length bound $b$ big enough to handle the running time of $V$.

Interestingly, the ESP also belongs to this same category (as pointed out in Footnote 2). Namely if someone hands us a proposed solution $X$, we can check it by just running $V$.

### 19.3.1 ESP, NP, and NP-completeness

This class of problems — problems for which we can efficiently verify a proposed solution — is called **NP**. A problem $Q$ is said to be NP-complete if (a) $Q$ is in **NP** and (2) you could use a polynomial-time algorithm for $Q$ to solve *any* problem in **NP** in polynomial time. That is, for any $Q'$ in **NP**, we have $Q' \leq_p Q$. And we have just proven our first **NP**-complete problem, namely the ESP.

This ESP problem seems pretty stylized. But we can now show that other simpler-looking problems have the property that if you could solve them in polynomial-time, then you could solve the ESP in polynomial time as well, so they too are **NP**-complete.

So, the way to think of **NP**-completeness is that an **NP**-complete problem is highly expressive. It is so expressive, that we believe there are no polynomial-time algorithms for solving them. In particular, if we *could* solve an **NP**-complete problem in polynomial-time, then it would mean that for any problem where we could *check* a proposed solution efficiently, we could also *find* such a solution efficiently. Now, onto formal definitions.

## 19.4 Formal definitions: P, NP, and NP-Completeness

We will formally be considering decision problems: problems whose answer is YES or NO. E.g., "Does the given network have a flow of value at least $k$?" or "Does the given graph have a 3-coloring?" For such problems, we can split all possible instances into two categories: YES-instances (whose correct answer is YES) and NO-instances (whose correct answer is NO). We can also put any ill-formed instances into the NO category. We now define the complexity classes **P** and **NP**.

**Definition 19.4** **P** *is the set of decision problems solvable in polynomial time.*

E.g., the decision version of the network flow problem: "Given a network $G$ and a flow value $k$, does there exist a flow $\geq k$?" belongs to **P**.

**Definition 19.5** **NP** *is the set of decision problems that have polynomial-time* verifiers. *Specificially, problem $Q$ is in* **NP** *if there is a polynomial-time algorithm $V(I, X)$ such that:*

- *If $I$ is a YES-instance, then there exists $X$ such that $V(I, X) =$ YES.*
- *If $I$ is a NO-instance, then for all $X$, $V(I, X) =$ NO.*

*Furthermore, $X$ should have length polynomial in size of $I$ (since we are really only giving $V$ time polynomial in the size of the instance, not the combined size of the instance and solution).*

The second input $X$ to the verifier $V$ is often called a *witness.* E.g., for 3-coloring, the witness that an answer is YES is the coloring. For factoring, the witness that $N$ has a factor between 2 and $k$ is a factor. For the TRAVELING SALESMAN PROBLEM: "Given a weighted graph $G$ and an integer $k$, does $G$ have a tour that visits all the vertices and has total length at most $k$?" the witness is the tour. All these problems belong to **NP**. Of course, any problem in **P** is also in **NP**, since $V$ could just ignore $X$ and directly solve $I$. So, $\mathbf{P} \subseteq \mathbf{NP}$.

A huge open question in complexity theory is whether $\mathbf{P} = \mathbf{NP}$. It would be quite strange if they were equal since that would mean that any problem for which a solution can be easily *verified* also has the property that a solution can be easily *found.* So most people believe $\mathbf{P} \neq \mathbf{NP}$. But, it's very hard to prove that a fast algorithm for something does *not* exist. So, it's still an open problem.

**Definition 19.6** *Problem $Q$ is* **NP***-complete if:*

1. *$Q$ is in* **NP***, and*
2. *For any other problem $Q'$ in* **NP***, $Q' \leq_p Q$.*

So if $Q$ is **NP**-complete and you could solve $Q$ in polynomial time, you could solve *any* problem in **NP** in polynomial time. If $Q$ just satisfies part (2) of the definition, then it's called **NP**-hard.

As we showed above, the ESP is **NP**-complete: it belongs to **NP** (that was the reason for including the bound $b$ in unary, so that running the verifier for $b$ steps counts as being polynomial-time), and we saw that we could use a polynomial-time algorithm for the ESP to solve any other problem in **NP** in polynomial-time.

## 19.5 Circuit-SAT and 3-SAT

Though the ESP is **NP**-complete, is a bit unweildy. We will now develop two more natural problems that also are **NP**-complete: CIRCUIT-SAT and 3-SAT. Both of them will be obviously in **NP**. To show they are **NP**-complete, we will show that ESP $\leq_p$ CIRCUIT-SAT, and then that CIRCUIT-SAT $\leq_p$ 3-SAT. Notice that this is enough: it means that if you had a polynomial-time algorithm for 3-SAT then you would also have a polynomial-time algorithm for CIRCUIT-SAT; and if you had a polynomial-time algorithm for CIRCUIT-SAT, then you would also have a polynomial-time algorithm for ESP; and we already know that if you have a polynomial-time algorithm for ESP, you can solve any problem in **NP** in polynomial-time. In other words, to show that a problem $Q$ is **NP**-complete, we just need to show that $Q' \leq_p Q$ for *some* **NP**-complete problem $Q'$ (plus show that $Q \in \mathbf{NP}$).

**Definition 19.7** CIRCUIT-SAT: *Given a circuit of NAND gates with a single output and no loops (some of the inputs may be hardwired). Question: is there a setting of the inputs that causes the circuit to output 1?*

**Theorem 19.1** *CIRCUIT-SAT is* **NP***-complete.*

**Proof Sketch:** First of all, CIRCUIT-SAT is clearly in **NP**, since you can just guess the input and try it. To show it is **NP**-complete, we need to show that if we could solve this, then we could solve the ESP. Say we are given $V$, $I$, and $b$, and want to tell if there exists $X$ such that $V(I, X)$ halts and outputs YES within at most than $b$ steps. Since we only care about running $V$ for $b$ steps we can assume it uses at most $b$ bits of memory, including the space for its arguments. We will now use the fact that one can construct a RAM with $b$ bits of memory (including its stored program) and a standard instruction set using only $O(b \log b)$ NAND gates and a clock. By unrolling this design for $b$ levels, we can remove loops and create a circuit that simulates what $V$ computes within $b$ time steps. We then hardwire the inputs corresponding to $I$ and feed this into our CIRCUIT-SAT solver. ■

So, we now have one more **NP**-complete problem. Still, CIRCUIT-SAT looks complicated: we weren't expecting to be able to solve it in polynomial-time. However, now we will show that a much simpler-looking problem, 3-SAT has the property that CIRCUIT-SAT $\leq_p$ 3-SAT.

**Definition 19.8** 3-SAT: *Given: a CNF formula (AND of ORs) over $n$ variables $x_1, \ldots, x_n$, where each clause has at most 3 variables in it. E.g., $(x_1 \vee x_2 \vee \bar{x}_3) \wedge (\bar{x}_2 \vee x_3) \wedge (x_1 \vee x_3) \wedge \ldots$. Goal: find an assignment to the variables that satisfies the formula if one exists.*

We'll save the proof that 3-SAT is **NP**-complete for the next lecture, but before we end, here is formally how we are going to do our reductions. Say we have some problem $A$ that we know is **NP**-complete. We want to show problem $B$ is **NP**-complete too. Well, first we show $B$ is in **NP** but that is usually the easy part. The main thing we need to do is show that $A \leq_p B$; that is, any polynomial-time algorithm for $B$ would give a polynomial-time algorithm for $A$. We will do this through the following method called a *many-one* or *Karp* reduction:

**Many-one (Karp) reduction from problem $A$ to problem $B$:** To reduce problem $A$ to problem $B$ we want a function $f$ that takes arbitrary instances of $A$ to instances of $B$ such that:

1. if $x$ is a YES-instance of $A$ then $f(x)$ is a YES-instance of $B$.
2. if $x$ is a NO-instance of $A$ then $f(x)$ is a NO-instance of $B$.
3. $f$ can be computed in polynomial time.

So, if we had an algorithm for $B$, and a function $f$ with the above properties, we could using it to solve $A$ on any instance $x$ by running it on $f(x)$.

# Lecture 20

# NP-Completeness II

## 20.1 Overview

In the last lecture, we defined the class **NP** and the notion of **NP**-completeness, and proved that the Circuit-SAT problem is **NP**-complete. In this lecture we continue our discussion of NP-Completeness, showing the following results:

- CIRCUIT-SAT $\leq_p$ 3-SAT (proving 3-SAT is NP-complete)
- 3-SAT $\leq_p$ CLIQUE (proving CLIQUE is NP-complete)
- NP-completeness of Independent Set and Vertex Cover

## 20.2 Introduction

Let us begin with a quick recap of our discussion in the last lecture. First of all, to be clear in our terminology, a *problem* means something like 3-coloring or network flow, and an *instance* means a specific instance of that problem: the graph to color, or the network and distinguished nodes $s$ and $t$ we want to find the flow between. A *decision problem* is just a problem where each instance is either a YES-instance or a NO-instance, and the goal is to decide which type your given instance is. E.g., for 3-coloring, $G$ is a YES-instance if it has a 3-coloring and is a NO-instance if not. For the Traveling Salesman Problem, an instance consists of a graph $G$ together with an integer $k$, and the pair $(G, k)$ is a YES-instance iff $G$ has a TSP tour of total length at most $k$.

We now define our key problem classes of interest.

**P:** The class of decision problems $Q$ that have polynomial-time algorithms. $Q \in \mathbf{P}$ if there exists a polynomial-time algorithm $A$ such that $A(I) = \text{YES}$ iff $I$ is a YES-instance of $Q$.

**NP:** The class of decision problems where at least the YES-instances have short proofs (that can be checked in polynomial-time). $Q \in \mathbf{NP}$ if there exists a verifier $V(I, X)$ such that:

- If $I$ is a YES-instance, then there exists $X$ such that $V(I, X) = \text{YES}$,
- If $I$ is a NO-instance, then for all $X$, $V(I, X) = \text{NO}$,

and furthermore the length of $X$ and the running time of V are polynomial in $|I|$.

**co-NP:** vice-versa — there are short proofs for NO-instances. Specifically, $Q \in$ **co-NP** if there exists a verifier $V(I, X)$ such that:

- If $I$ is a YES-instance, for all $X$, $V(I, X) =$ YES,
- If $I$ is a NO-instance, then there exists $X$ such that $V(I, X) =$ NO,

and furthermore the length of $X$ and the running time of V are polynomial in $|I|$.

For example, the problem Circuit-Equivalence: "Given two circuits $C_1, C_2$, do they compute the same function?" is in **co-NP**, because if the answer is NO, then there is a short, easily verified proof (an input $x$ such that $C_1(x) \neq C_2(x)$).

A problem $Q$ is **NP**-complete if:

1. $Q \in$ **NP**, and
2. Any other $Q'$ in **NP** is polynomial-time reducible to $Q$; that is, $Q' \leq_p Q$.

If Q just satisfies (2) then it's called **NP**-hard. Last time we showed that the following problem is **NP**-complete:

> **Circuit-SAT**: Given a circuit of NAND gates with a single output and no loops (some of the inputs may be hardwired). Question: is there a setting of the inputs that causes the circuit to output 1?

**Aside:** we could define the *search*-version of a problem in **NP** as: "...and furthermore, if $I$ is a YES-instance, then *produce* $X$ such that $V(I, X) =$ YES." If we can solve any **NP**-complete decision problem in polynomial time then we can actually solve search-version of any problem in **NP** in polynomial-time too. (Will talk about this in recitation.)

Unfortunately, Circuit-SAT is a little unweildy. What's *especially interesting* about NP-completeness is not just that such problems *exist*, but that a lot of very innocuous-looking problems are NP-complete. To show results of this form, we will first reduce Circuit-SAT to the much simpler-looking 3-SAT problem (i.e., show Circuit-SAT $\leq_p$ 3-SAT). Recall the definition of 3-SAT from last time:

**Definition 20.1** 3-SAT: *Given: a CNF formula (AND of ORs) over $n$ variables $x_1, \ldots, x_n$, where each clause has at most 3 variables in it. Goal: find an assignment to the variables that satisfies the formula if one exists.*

**Theorem 20.1** *CIRCUIT-SAT $\leq_p$ 3-SAT. I.e., if we can solve 3-SAT in polynomial time, then we can solve CIRCUIT-SAT in polynomial time (and thus all of* **NP***).*

**Proof:** We need to define a function $f$ that converts instances $C$ of Circuit-SAT to instances of 3-SAT such that the formula $f(C)$ produced is satisfiable iff the circuit $C$ had an input $x$ such that $C(x) = 1$. Moreover, $f(C)$ should be computable in polynomial time, which among other things means we cannot blow up the size of $C$ by more than a polynomial factor.

First of all, let's assume our input is given as a list of gates, where for each gate $g_i$ we are told what its inputs are connected to. For example, such a list might look like: $g_1 = \mathsf{NAND}(x_1, x_3)$;

$g_2 == \mathsf{NAND}(g_1, x_4)$; $g_3 = \mathsf{NAND}(x_1, 1)$; $g_4 = \mathsf{NAND}(g_1, g_2)$; .... In addition we are told which gate $g_m$ is the output of the circuit.

We will now compile this into an instance of 3-SAT as follows. We will make one variable for each input $x_i$ of the circuit, and one for every gate $g_i$. We now write each NAND as a conjunction of 4 clauses. In particular, we just replace each statement of the form "$y_3 = \mathsf{NAND}(y_1, y_2)$" with:

$$\begin{array}{lll} & (y_1 \text{ OR } y_2 \text{ OR } y_3) & \leftarrow \text{if } y_1 = 0 \text{ and } y_2 = 0 \text{ then we must have } y_3 = 1 \\ \text{AND} & (y_1 \text{ OR } \overline{y}_2 \text{ OR } y_3) & \leftarrow \text{if } y_1 = 0 \text{ and } y_2 = 1 \text{ then we must have } y_3 = 1 \\ \text{AND} & (\overline{y}_1 \text{ OR } y_2 \text{ OR } y_3) & \leftarrow \text{if } y_1 = 1 \text{ and } y_2 = 0 \text{ then we must have } y_3 = 1 \\ \text{AND} & (\overline{y}_1 \text{ OR } \overline{y}_2 \text{ OR } \overline{y}_3). & \leftarrow \text{if } y_1 = 1 \text{ and } y_2 = 1 \text{ we must have } y_3 = 0 \end{array}$$

Finally, we add the clause $(g_m)$, requiring the circuit to output 1. In other words, we are asking: is there an input to the circuit *and* a setting of all the gates such that the output of the circuit is equal to 1, *and* each gate is doing what it's supposed to? So, the 3-CNF formula produced is satisfiable if and only if the circuit has a setting of inputs that causes it to output 1. The size of the formula is linear in the size of the circuit. Moreover, the construction can be done in polynomial (actually, linear) time. So, if we had a polynomial-time algorithm to solve 3-SAT, then we could solve circuit-SAT in polynomial time too. ∎

**Important note:** Now that we know 3-SAT is **NP**-complete, in order to prove some other **NP** problem $Q$ is **NP**-complete, we just need to reduce 3-SAT to $Q$; i.e., to show that 3-SAT $\leq_p Q$. In particular, we want to construct a (polynomial-time computable) function $f$ that converts instances of 3-SAT to instances of $Q$ that preserves the YES/NO answer. This means that if we could solve $Q$ efficiently then we could solve 3-SAT efficiently. *Make sure you understand this reasoning — a lot of people make the mistake of doing the reduction the other way around.* Doing the reduction the wrong way is just as much work but does not prove the result you want to prove!

## 20.3 CLIQUE

We will now use the fact that 3-SAT is **NP**-complete to prove that a natural graph problem called the MAX-CLIQUE problem is **NP**-complete.

**Definition 20.2** MAX-CLIQUE*: Given a graph G, find the largest clique (set of nodes such that all pairs in the set are neighbors). Decision problem: "Given G and integer k, does G contain a clique of size $\geq k$?"*

Note that MAX-CLIQUE is clearly in **NP**.

**Theorem 20.2** MAX-CLIQUE *is* **NP**-*Complete.*

**Proof:** We will reduce 3-SAT to MAX-CLIQUE. Specifically, given a 3-CNF formula $F$ of $m$ clauses over $n$ variables, we construct a graph as follows. First, for each clause $c$ of $F$ we create one node for every assignment to variables in $c$ that satisfies $c$. E.g., say we have:

$$F = (x_1 \vee x_2 \vee \overline{x}_4) \wedge (\overline{x}_3 \vee x_4) \wedge (\overline{x}_2 \vee \overline{x}_3) \wedge \ldots$$

Then in this case we would create nodes like this:

$$\begin{array}{llll}
(x_1=0,x_2=0,x_4=0) & (x_3=0,x_4=0) & (x_2=0,x_3=0) & \ldots \\
(x_1=0,x_2=1,x_4=0) & (x_3=0,x_4=1) & (x_2=0,x_3=1) & \\
(x_1=0,x_2=1,x_4=1) & (x_3=1,x_4=1) & (x_2=1,x_3=0) & \\
(x_1=1,x_2=0,x_4=0) & & & \\
(x_1=1,x_2=0,x_4=1) & & & \\
(x_1=1,x_2=1,x_4=0) & & & \\
(x_1=1,x_2=1,x_4=1) & & &
\end{array}$$

We then put an edge between two nodes if the partial assignments are consistent. Notice that the maximum possible clique size is $m$ because there are no edges between any two nodes that correspond to the same clause $c$. Moreover, if the 3-SAT problem *does* have a satisfying assignment, then in fact there *is* an $m$-clique (just pick some satisfying assignment and take the $m$ nodes consistent with that assignment). So, to prove that this reduction (with $k = m$) is correct we need to show that if there *isn't* a satisfying assignment to $F$ then the maximum clique in the graph has size $< m$. We can argue this by looking at the contrapositive. Specifically, if the graph has an $m$-clique, then this clique must contain one node per clause $c$. So, just read off the assignment given in the nodes of the clique: this by construction will satisfy all the clauses. So, we have shown this graph has a clique of size $m$ iff $F$ was satisfiable. Also, our reduction is polynomial time since the graph produced has total size at most quadratic in the size of the formula $F$ ($O(m)$ nodes, $O(m^2)$ edges). Therefore MAX-CLIQUE is **NP**-complete. ■

## 20.4 Independent Set and Vertex Cover

An Independent Set in a graph is a set of nodes no two of which have an edge. E.g., in a 7-cycle, the largest independent set has size 3, and in the graph coloring problem, the set of nodes colored red is an independent set. The INDEPENDENT SET problem is: given a graph $G$ and an integer $k$, does $G$ have an independent set of size $\geq k$?

**Theorem 20.3** INDEPENDENT SET *is* **NP**-*complete.*

**Proof:** We reduce from MAX-CLIQUE. Given an instance $(G, k)$ of the MAX-CLIQUE problem, we output the instance $(H, k)$ of the INDEPENDENT SET problem where $H$ is the complement of $G$. That is, $H$ has edge $(u, v)$ iff $G$ does *not* have edge $(u, v)$. Then $H$ has an independent set of size $k$ iff $G$ has a $k$-clique. ■

A *vertex cover* in a graph is a set of nodes such that every edge is incident to at least one of them. For instance, if the graph represents rooms and corridors in a museum, then a vertex cover is a set of rooms we can put security guards in such that every corridor is observed by at least one guard. In this case we want the smallest cover possible. The VERTEX COVER problem is: given a graph $G$ and an integer $k$, does $G$ have a vertex cover of size $\leq k$?

**Theorem 20.4** VERTEX COVER *is* **NP**-*complete.*

**Proof:** If $C$ is a vertex cover in a graph $G$ with vertex set $V$, then $V - C$ is an independent set. Also if $S$ is an independent set, then $V - S$ is a vertex cover. So, the reduction from Independent Set to Vertex Cover is very simple: given an instance $(G, k)$ for Independent Set, produce the instance $(G, n - k)$ for Vertex Cover, where $n = |V|$. In other words, to solve the question "is there an independent set of size at least $k$" just solve the question "is there a vertex cover of size $\leq n - k$?" So, Vertex Cover is **NP**-Complete too. ■

## 20.5 Beyond NP

As mentioned earlier, it is an open problem whether $\mathbf{P} \neq \mathbf{NP}$ (though everyone believes they are different). It is also an open problem whether $\mathbf{NP} \neq \mathbf{co\text{-}NP}$ (though again, everyone believes they are different). One can also define even more expressive classes. For instance, **PSPACE** is the class of all problems solvable by an algorithm that uses a polynomial amount of memory. Any problem in **NP** is also in **PSPACE**, because one way to solve the problem is to take the given instance $I$ and then simply run the verifier $V(I, X)$ on all possible proof strings $X$, halting with YES if any of the runs outputs YES, and halting with NO otherwise. (Remember, we have a polynomial upper bound on $|X|$, so this uses only a polynomial amount of space.) Similarly, any problem in **co-NP** is also in **PSPACE**. Unfortunately, it is not even known for certain that $\mathbf{P} \neq \mathbf{PSPACE}$ (though all the classes mentioned above are believed to be different). One can also define classes that are *provably* larger than **P** and **NP**. For instance, **EXPtime** is the class of all problems solvable in time $O(2^{n^c})$ for some constant $c$. This class is known to *strictly* contain **P**. The class **NEXPtime** is the class of all problems that have a *verifier* which runs in time $O(2^{n^c})$ for some constant $c$. This class is known to strictly contain **NP**. The class of all Turing-computable problems is known to strictly contain all of the above, and some problems such as the Halting problem (given a program $\mathcal{A}$ and an input $x$, determine whether or not $\mathcal{A}(x)$ halts) are not even contained in that!

## 20.6 NP-Completeness summary

**NP**-complete problems have the dual property that they belong to **NP** and they capture the essence of the entire class in that a polynomial-time algorithm to solve one of them would let you solve anything in **NP**.

We proved that 3-SAT is **NP**-complete by reduction from Circuit-SAT. Given a circuit $C$, we showed how to compile it into a 3-CNF formula by using extra variables for each gate, such that the formula produced is satisfiable if and only if there exists $x$ such that $C(x) = 1$. This means that a polynomial-time algorithm for 3-SAT could solve any problem in **NP** in polynomial-time, even factoring. Moreover, 3-SAT is a simple-looking enough problem that we can use it to show that many other problems are **NP**-complete as well, including Max-Clique, Independent Set, and Vertex Cover.