15-451: Algorithms

Avrim Blum
Anupam Gupta
Danny Sleator

Lecture Notes
Spring 2016

Computer Science Department
Carnegie Mellon University
August 16, 2016

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, that of finding the median of a set of n elements. This is a problem for which there is a simple $O(n \log n)$ time algorithm, but we can do better, using randomization, and also a clever deterministic construction. These illustrate some of the ideas and tools we will be using (and building upon) in this course.

Material in this lecture:

- Administrivia (see course policies on the webpage)
- What is the study of Algorithms all about?
- Why do we care about specifications and proving guarantees?
- Finding the median: A Randomized Algorithm in expected linear time.
- A deterministic linear-time algorithm.

1 Goals of the Course

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. A recipe. 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, Hashing and other Data Structures, Randomization, Network Flows, and Linear Programming. 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 additionally discuss some approaches for dealing with NP-complete problems, including the notion of approximation algorithms.

Another goal will be to discuss models that go beyond the traditional input-output model. In the traditional model we consider the algorithm to be given the entire input in advance and it just has to perform the computation and give the output. This model is great when it applies, but it is

not always the right model. For instance, some problems may be challenging because they require decisions to be made without having full information, and we will discuss online algorithms and machine learning, which are two paradigms for problems of this nature. In other settings, we may have to deal with computing quantities of a "stream" of input data where the space we have is much smaller than the data. In yet other settings, the input is being held by a set of selfish agents who may or may not tell us the correct values.

2 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 lower bounds and game theory.

3 An example: Median Finding

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 median finding.

Recall the median. For a set of n elements, this is the element in this set that is the $n/2^{th}$ smallest, i.e., it has $n/2$ elements larger than it.[1] Given an unsorted array, how quickly can one find the median element? The definition gives us no clue: we can enumerate over all elements, and for each check if it is the median. This gives a $\Theta(n^2)$ time algorithm. Or one can sort, and then read off the median, which takes $O(n \log n)$ time using MergeSort or HeapSort (deterministic) or QuickSort (randomized).

Can one do it more quickly than by sorting? In this lecture we describe two linear-time algorithms for this problem: one randomized and one deterministic. More generally, we solve the problem of finding the kth smallest out of an unsorted array of n elements.

3.1 The problem and a randomized solution

Consider the problem of finding the kth smallest element in an unsorted array of size n. (Let's say all elements are distinct to avoid the question of what we mean by the kth smallest when we have equalities). One way to solve this problem is to sort and then output the kth element. We can do this in time $O(n \log n)$ if we sort using MergeSort, QuickSort, or HeapSort. 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 start with the Randomized QuickSort algorithm (choose a random element as "pivot", partition the array into two sets Less and Greater consisting of those elements less than and greater than the pivot respectively, and then recursively sort Less and Greater). Then notice that there is a simple speedup we can make if we just need to find the kth smallest element. In particular, after the partitioning step we can tell which of Less or Greater has the item we are looking for, just by looking at their sizes. So, we only need to recursively examine one of them, not both. For instance, if we are looking for the 87th-smallest element in our array, and suppose that after choosing the pivot and partitioning we find that Less has 200 elements, then we just need to find the 87th smallest element in Less. On the other hand, if we find Less has 40 elements, then we just need to find the $87 - 40 - 1 = 46$th smallest element in Greater. (And if Less has size exactly 86 then we can just return the pivot). One might at first think that allowing the algorithm to only recurse on one subset rather than both 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.

[1] We are deliberately ignoring what happens if n is odd, you can — and indeed, should (and will have to) — make this precise when you code it up, but for now it will be easier not to worry about this, since the ideas we get here can be all made perfectly precise.

QuickSelect: Given array A of size n and integer $1 \leq k \leq n$,

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 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 $\mathrm{E}[T(i)]$ rather than $T(\mathrm{E}[i])$. 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 1): Let $T(n,k)$ denote the expected time to find the kth smallest in an array of size n, and let $T(n) = \max_k T(n,k)$. We will show that $T(n) < 4n$.

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

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

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,

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

and we have verified our guess. ■

4 A deterministic linear-time algorithm

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

linear time algorithm was developed by Manuel Blum, Bob Floyd, Vaughan Pratt, Ron Rivest, and Bob Tarjan.[2]

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

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

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

$$T(n) \quad \leq \quad cn + T(n/5) + T(7n/10), \tag{1}$$

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? But first, let's do an example.

> **Example 1:** 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.

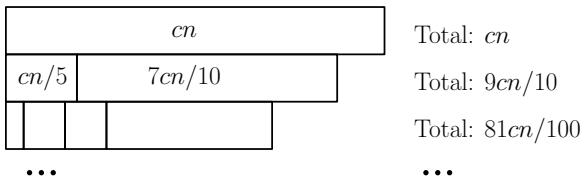
In general, what is the worst case? If there are $g = n/5$ groups, then we know that in at least $\lceil g/2 \rceil$ of them (those groups whose median is $\leq p$) at least three of the five elements are $\leq p$. Therefore, the total number of elements $\leq p$ is at least $3\lceil g/2 \rceil \geq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

[2] That's 4 Turing Award winners on that one paper!

answer is linear in n? One way to do that is to consider the "stack of bricks" view of the recursion tree discussed in the notes for Recitation #1.

In particular, let's build the recursion tree for the recurrence (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 (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 the notes to Recitation #1.

Theorem 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 $T(n) = O(n)$.

Exercise 1: Show that for constants c and $a_1, \ldots, a_k$ such that $a_1 + \ldots a_k = 1$ and each $a_i < 1$, the recurrence

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

solves to $T(n) = O(n \log n)$. Show that this is best possible by observing that $T(n) = T(n/2)+T(n/2)+n$ solves to $T(n) = \Theta(n \log n)$.

Exercise 2: What happens if we split the elements into $n/3$ groups of size 3? Or n/k groups of size k for larger odd values of k?

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

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

1 Terminology and setup

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

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

2 Sorting in the comparison model

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

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

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

Theorem 2 *Any deterministic comparison-based sorting algorithm must perform at least* $\lg(n!)$ *comparisons to sort n elements in the worst case.*[1] *Specifically, for any deterministic comparison-*

[1] As is common in CS, we will use "lg" to mean "$\log_2$".

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 $\lg(n!) = \Omega(n \log n)$ comparisons to sort I.

To prove this theorem, we cannot assume the sorting algorithm is going to necessarily choose a pivot as in Quicksort, or split the input as in Mergesort — we need to somehow analyze *any possible* (comparison-based) algorithm that might exist. We now present the proof, which uses a very nice information-theoretic argument. (This proof is deceptively short: it's worth thinking through each line and each assertion.)

Proof: Let us begin with a simple general claim. Suppose you have some problem where there are M possible different outputs the algorithm might produce: e.g., for sorting by comparisons where the output can be viewed as a specific permutation of the input, each possible permutation of the input is possible, and hence $M = n!$. Suppose furthermore that for each of these outputs, there exists some input under which it is the only correct answer. This is true for sorting. Then, we have a worst-case lower bound of $\lg M$. The reason is that the algorithm needs to find out which of these M outputs is the right one, and each YES/NO question could be answered in away that removes at most half of the possibilities remaining from consideration. So, in the worst case, it takes at least $\lg M$ steps to find the right answer. ■

The above is often called an "information theoretic" argument because we are in essence saying that we need at least $\lg(M) = \lg(n!)$ bits of information about the input before we can correctly decide what output we need to produce.

What does $\lg(n!)$ look like? We have: $\lg(n!) = \lg(n) + \lg(n-1) + \lg(n-2) + ... + \lg(1) < n \lg(n) = O(n \log n)$ and $\lg(n!) = \lg(n) + \lg(n-1) + \lg(n-2) + ... + \lg(1) > (n/2) \lg(n/2) = \Omega(n \log n)$. So, $\lg(n!) = \Theta(n \log n)$.

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

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

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

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

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

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

3 Sorting in the exchange model

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

are considering the problem of *sorting* in the *exchange model*.

Definition 3 *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 4 (Upper bound) *$n-1$ exchanges is sufficient.*

Proof: For this we just need to give an algorithm. For instance, consider the algorithm that in step 1 puts the smallest item in location 1, swapping it with whatever was originally there. Then in step 2 it swaps the second-smallest item with whatever is currently in location 2, and so on (if in step k, the kth-smallest item is already in the correct position then we just do a no-op). No step ever undoes any of the previous work, so after $n-1$ steps, the first $n-1$ items are in the correct position. This means the nth item must be in the correct position too. ■

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

Claim 5 (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. An example is given in Figure 1.

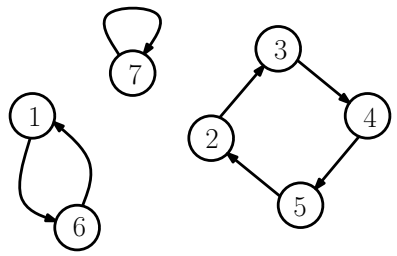


Figure 1: Graph for input `[f c d e b a g]`

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. ■

4 The comparison model revisited

4.1 Finding the maximum of n elements

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

Claim 6 (Upper bound) *$n-1$ comparisons are sufficient to find the maximum of n elements.*

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

Now, let's try for a lower bound. One simple lower bound is that since there are n possible answers for the location of the maximum element, our previous argument gives a lower bound of $\lg n$. But clearly this is not at all tight. Also, we have to look at all the elements (else the one not looked at may be larger than all the ones we look at). But looking at all n elements could be done using $n/2$ comparisons; not tight either. In fact, we can give a better lower bound of $n-1$.

Claim 7 (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, the bound of $n-1$ is tight.

Note that this argument was different from the "information theoretic" bound we used for sorting. Here we showed that if the algorithm makes "too few" comparisons on some input In and outputs *out*, we can give another input In' where the algorithms would do the same comparisons and receive the same answers to them, and hence also output *out*, but *out* is the incorrect output for input In'.

4.2 An Adversary Argument

A slightly different lower bound argument comes from showing that if an algorithm makes "too few" comparisons, then an adversary can fool it into giving the incorrect answer. Here is a little example.

We want to show that any deterministic sorting algorithm on 3 elements must perform at least 3 comparisons in the worst case. (This result follows from the information theoretic lower bound of $\lceil \lg 3! \rceil = 3$, but let's give a different proof.)

If the algorithm does fewer than two comparisons, some element has not been looked at, and the algorithm must be incorrect. So after the first comparison, the three elements are w the winner of the first query, l the loser, and z the other guy. If the second query is between w and z, the adversary replies $w > z$; if it is between l and z, the adversary replies $l < z$. Note that in either case, the algorithm must perform a third query to be able to sort correctly.

In this kind of argument the goal is to construct an adversary Ada who will answer the algorithm's comparisons in such a way that (a) all Ada's answers are consistent with some input In, and (b) her answers make the algorithm perform "many" comparisons.

4.3 Finding the second-largest of n elements

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

Claim 8 (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.

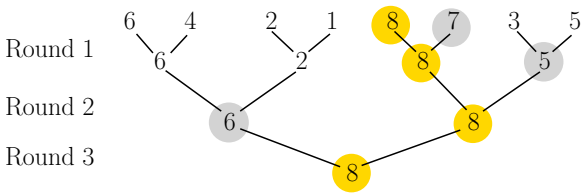
Claim 9 (Upper bound #1) *$2n-3$ comparisons are sufficient to find the second-largest of n elements.*

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

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



Now, given just what we know from comparisons so far, what can we say about possible locations for the second-highest number (i.e., the second-best player)? The answer is that the second-best

must have been directly compared to the best, and lost.[2] This means there are only $\lg n$ possibilities for the second-highest number, and we can find the maximum of them making only $\lg(n) - 1$ more comparisons. ■

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

5 Query models, and the evasiveness of connectivity (Optional)

This material is optional; you may find it interesting.

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

Claim 11 (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 12 (Lower bound) *$n(n-1)/2$ queries are necessary to determine connectivity in the worst case.*

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

> *Claim:* we maintain the invariant that for any un-asked pair (u, v), the graph revealed so far has no path from u to v.
>
> *Proof of claim:* If there was, consider the last edge (u', v') revealed on that path. We could have answered 0 for that and kept the same connectivity in the graph by having an edge (u, v). So, that contradicts the definition of our adversary strategy.

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

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

[3] First shown by Kislitsyn (1964).

Lecture 3

Amortized Analysis

3.1 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.

3.2 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 3.1 *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.

3.3 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.

3.4 Piggy banks and potential functions

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

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

Definition 3.2 *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.

3.5 Example #2: a binary counter

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

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

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

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

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

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

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

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

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

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

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

3.7 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 3.1 *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. ■

Notes on Amortization

D. Sleator

1. Introduction

A *data structure* is a way of representing information in a computer and a set of procedures for accessing and updating the information. These procedures for accessing and updating the information are called the *operations* on the data structure.

There are two fundamentally different ways in which data structures are used. They are used as part of an information retrieval system, and they are used as one component of an algorithm whose purpose is to solve some other problem.

Why do we have data structures? In the first application the purpose of the data structure is obvious, for the data structure itself is solving the problem that we want to solve. In the second application, the reason for having data structures is not so obvious. In this case our motivation for creating them is to aid in our conception of the algorithm. By using a data structure as a component of an algorithm, we split the problem of creating or expressing the algorithm into two parts. Once the interface between these two parts has been specified, these two components of the problem can be solved (or expressed) separately.

The interface between a data structure and the algorithm that uses it consists of two parts:

1. A set of operations that allow the algorithm to access and update the data structure.

2. Constraints that say how much time (or space) each operation is allowed to use in order for the algorithm to perform with the desired efficiency.

The structure of this interface implies that the data structure must perform in an on-line fashion, that is, it must perform the current operation before it knows what the future operations will be. Furthermore, no assumptions are made about the pattern of operations done by the algorithm. The data structure should have the desired performance for any sequence.

This note describes a technique that has been used to design improved data structures, and to analyze their performance. These advances were not made by changing the interface between the data structure and the algorithm that uses it. Rather, they were made by allowing the data structure to take full advantage of the flexibility of this interface.

The first observation is the following: Although the performance bounds on the data structure are specified by the interface, these bounds do not have to be satisfied by the data structure for every single operation. All that is actually needed is that the cost of sequence of operations be bounded by the sum of the specified bounds. An analysis of the worst case cost of a *sequence* of operations is called an *amortized analysis*.

For example suppose a sequence of operations $\sigma_1, \sigma_2, \cdots, \sigma_n$ is to be appled to a data structure. Say that the interface requires that the time taken by operation σ_i be at most $b(\sigma_i)$. In order for the data structure to satisfy the requirement of the interface, it is *not* necessary that:

$$t(\sigma_i) \leq b(\sigma_i),$$

rather, what is required is that

$$\sum_i^n t(\sigma_i) \leq \sum_i^n b(\sigma_i).$$

If the data structure has this property, then the operation σ_i is said to take *amortized time* $b(\sigma_i)$. (This definition of the amortized time of an operation is slightly more restricted than that used in the sequel. However, any bounds satisfying the above inequality certainly satisfy our more liberal definition given below.)

The second important observation is that althought the data structure cannot know the future operations, it is allowed to use the information it has about operations that were done in the past. It turns out that a useful technique in constructing data structures that are efficient in the amortized sense is to have the structure adjust itself based on past requests. Informally, we call such a data structure *self-adjusting*.

Amortized analysis and self-adjustment have been used to devise a variety of efficient new data structures. In the next section we illustrate the concepts of amortized analysis with a simple example.

2. Amortized analysis: an example

To illustrate the concepts of amortized analysis I shall use a simple example. Suppose that the cost of incrementing a binary number is the number of bits in it that change. What is the cost of incrementing a binary number from 0 to n?

It is easy to see that on each increment operation the low order bit changes. Thus, the number of times this bit changes is n. The 2's bit changes on the second increment operation, and on alternate subsequent operations, thus it changes a total of $\lfloor n/2 \rfloor \leq n/2$ times. Similarly, the ith bit changes at most $n/2^{i-1}$ times. Thus the total cost of this sequence of increments is at most

$$n + \frac{n}{2} + \frac{n}{4} + \cdots = 2n.$$

Thus, by the definition of amortized cost, the increment operation has an amortized cost of 2. Notice that some individual increment operation may cause $\lfloor \log n \rfloor$ bits to change. [1]

Another way to prove this result is my means of the *banker's view* of amortization. Suppose that each time a bit of the binary number changes it costs us one dollar. Also, suppose that we maintain a bank account such that for each bit of the binary

[1] The symbol "log" denotes the binary logarithm.

number that is a 1 we keep a dollar in the account. Initially there is no money in the account.

What happens when the binary number is incremented? A sequence of zero or more 1's all change to 0's, and then a 0 changes to a 1. For each bit that changes from a 1 to a 0, we take a dollar from the account to pay for it. For the bit that changes from a 0 to a 1, we must pay for changing the bit, and we must also put a dollar in the bank to maintain the relationship between the bank account and the number. Thus our out of pocket cost to pay for the increment is exactly two dollars, no matter how big the number is or how many carries occur. In a sequence of n increment operations, our total cost is $2n$ dollars and we are left with a non-negative bank account. Therefore the total cost of all the increments is at most $2n$.

This technique can be applied in a much more general way. The idea is to make a rule that says how much money must be kept in the bank as a function of the state of the data structure. Then a bound is obtained on how much money is required to pay for an operation and maintain the appropriate amount of money in the bank.

The *physicist's view* of amortization uses different terminology to describe the same idea. This is the formulation I shall use in this course. A *potential function* $\Phi(s)$ is a mapping from data state structure states to the reals. (This takes the place of the bank account in the banker's view.)

Consider a sequence of n operations $\sigma_1, \sigma_2, \ldots, \sigma_n$ the data structure. Let the sequence of states through which the data structure passes be $s_0, s_1, \ldots, s_n$. Notice that operation σ_i changes the state from s_{i-1} to s_i. Let the cost of operation σ_i be c_i. Define the amortized cost ac_i of operation σ_i by the following formula:

$$ac_i \;=\; c_i + \Phi(s_i) - \Phi(s_{i-1}), \tag{1}$$

or

$$\text{(amortized cost)} \;=\; \text{(actual cost)} + \text{(change in potential)}.$$

If we sum both sides of this equation over all the operations, we obtain the following formula:

$$\sum_i ac_i \;=\; \sum_i (c_i + \Phi(s_i) - \Phi(s_{i-1}) \;=\; \Phi(s_n) - \Phi(s_0) + \sum_i c_i.$$

Rearranging we get

$$\sum_i c_i \;=\; (\sum_i ac_i) + \Phi(s_0) - \Phi(s_n). \tag{2}$$

If $\Phi(s_0) \leq \Phi(s_n)$ (as will frequently be the case) we get

$$\sum_i c_i \leq \sum_i ac_i. \tag{3}$$

Thus, if we can bound the amortized cost of each of the operations, and the final potential is at least as large as the initial potential, then the bound we obtained for the amortized cost applies to the actual cost.

We can now apply this technique to the problem of computing the cost of binary counting. Let the potential Φ be the number of 1's in the current number. Our first

goal is to show that with this potential the amortized cost of an increment operation is 2.

Consider the ith increment operation that changes the number from $i-1$ to i. Let k be the number of carries that occur as a result of the increment. The cost of the operation is $k+1$. The change in potential caused by the operation is $-k+1$. (The number of bits that change from 1 to 0 is k and one bit changes from 0 to 1.) Therefore the amortized cost of the operation is

$$ac_i = k + 1 + (-k + 1) = 2.$$

Since the final potential is more than the initial potential, we can apply inequality (3) to obtain:

$$\sum_i c_i \leq \sum_i ac_i = 2n.$$

Notice that in this formulation, the definition of the amortized cost of an operation depends on the choice of the potential function. In fact, any choice of potential function whatsoever defines an amortized cost of each operation. However, these amortized bounds will not be useful unless $\Phi(s_0) - \Phi(s_n)$ is also bounded appropriately.

We have given two different definitions of amortized cost, one in Section 1, and the other in equation 1. Which definition applies in a discussion will depend on the context of the discussion. If we discuss amortized cost in the context of a potential function, then the amortized cost is that defined by equation 1. If it is outside the context of a potential function, then the meaning of amortized cost is that given in Section 1.

Most of the art of doing an amortized analysis is in choosing the right potential function. Once a potential function is chosen we must do two things:

1. Prove that with the chosen potential function, the amortized costs of the operations satisfy the desired bounds.

2. Bound the quantity $\Phi(s_0) - \Phi(s_n)$ appropriately.

In today's lecture, we will discuss:

- binary search trees in general
- definition of splay trees
- analysis of splay trees

The analysis of splay trees uses the potential function approach we discussed in the previous lecture. It seems to be required.

1 Binary Search Trees

These lecture notes assume that you have seen binary search trees (BSTs) before. They do not contain much expository or backtround material on the basics of BSTs.

Binary search trees is a class of data structures where:

1. Each node stores a piece of data
2. Each node has two pointers to two other binary search trees
3. The overall structure of the pointers is a tree (there's a root, it's acyclic, and every node is reachable from the root.)

Binary search trees are a way to store a update a set of items, where there is an ordering on the items. I know this is rather vague. But there is not a precise way to define the gamut of applications of search trees. In general, there are two classes of applications. Those where each item has a key value from a totally ordered universe, and those where the tree is used as an efficient way to represent an ordered list of items.

Some applications of binary search trees:

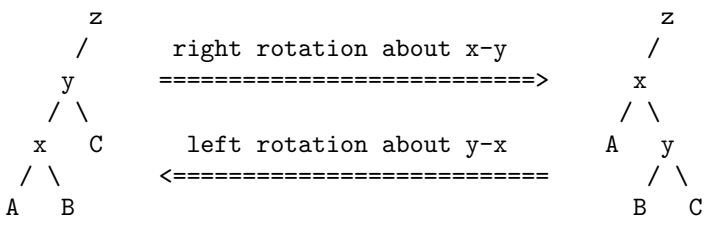
- Storing a set of names, and being able to lookup based on a prefix of the name. (Used in internet routers.)
- Storing a path in a graph, and being able to reverse any subsection of the path in $O(\log n)$ time. (Useful in travelling salesman problems).
- Being able to quickly determine the rank of a given node in a set.
- Being able to split a set S at a key value x into S_1 (the set of keys $< x$) and S_2 (those $> x$) in $O(\log n)$ time.

Given a tree[1] T an *in-order* traversal of the tree is defined recursively as follows. To traverse a tree rooted at a node x, first traverse the left child of x, then traverse x, then traverse the right child of x. When a tree is used to store a set of keys from a totally ordered universe, the in-order traversal will visit the keys in ascending order.

A *rotation* in binary search tree is a local restructuring operation that preserves the order of the nodes, but changes the depths of some of the nodes. Rotations are used by many BST algorithms to keep the tree "balanced", thus assuring that the depth is logarithmic.

[1] In this lecture "tree" is used synonymously with "BST".

A rotation involves a pair of nodes `x` and its parent `y`. After the rotation `x` is the parent of `y`. The update is done in such a way as to guarantee that the in-order traversal of the tree does not change. So if x is initially the left child of y, then after the rotation y is the right child of x. This is known as a *right rotation*. A *left rotation* is the mirror image variant.

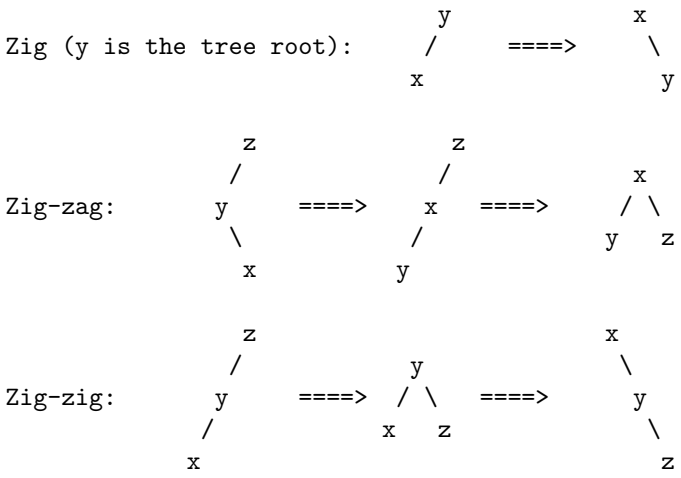


2 Splay Trees (self-adjusting search trees)

These notes just describe the *bottom-up splaying* algorithm, the proof of the access lemma, and a few applications. (See `http://www.link.cs.cmu.edu/splay/` for more information along with an interactive splay tree demo.)

At a high level, every time a node is accessed in a splay tree, it is moved to the root of the tree. We will show that the amortized cost of the operation is $O(\log n)$. Just moving the element to the root by rotating it up the tree does not have this property. (To see this, start with a tree of nodes labeled $0, 1, 2, \ldots, n-1$ that is just a long left path down from the root, with 0 being the leftmost node in the tree. If we sequentially rotate 0, then 1, then 2, etc. to the root, the tree that results is the same as the starting tree, but the total work is $\Omega(n^2)$, for an amortized lower bound of $\Omega(n)$ per operation.) Splay trees perform movements in a very special way that guarantees this logarithmic amortized bound.

I'll describe the algorithm by giving three rewrite rules in the form of pictures. In these pictures, `x` is the node that was accessed (that will eventually be at the root of the tree). By looking at the structure of the 3-node tree defined by `x`, `x`'s parent, and `x`'s grandparent we decide which of the following three rules to follow. We continue to apply the rules until `x` is at the root of the tree:



Some notes:

1. Each arrow in the diagram above represents a rotation. Zig case does one rotation, the other cases do two rotations.

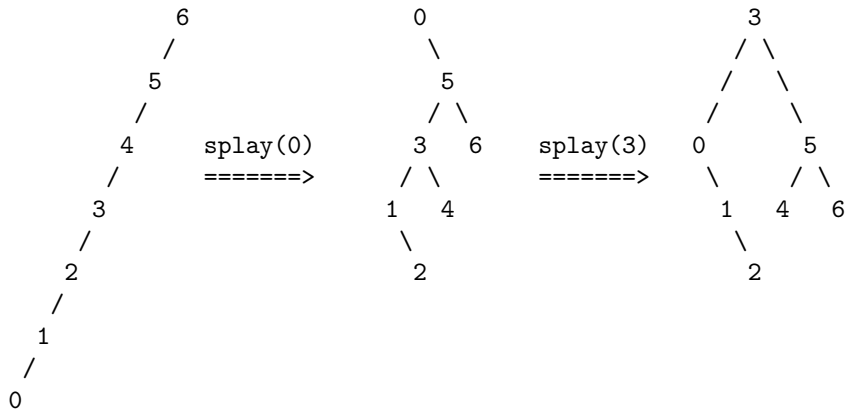
2. A *splay step* refers to applying one of these rewrite rules. Later we will be proving bounds on number of splay steps done. Each such step can be executed in constant time.

3. Each rule has a mirror image variant, which I have not shown for brevity. So for example, if `x` is the right child of `y` (the root), then the mirror image of the Zig case applies. In this case a left rotation would be done to put `x` above `y`, instead of the right rotation show above

4. The Zig-zig rule is the one that distinguishes splaying from just rotating `x` to the root of the tree.

5. There are a number of alternative versions of the algorithm, such as top-down splaying, which may be more efficient than the bottom-up version described here. Code for the top-down version is available here `http://www.link.cs.cmu.edu/splay/`.

Here are some examples:



To analyze the performance of splaying, we start by assuming that each node x has a weight $w(x) > 0$. These weights can be chosen arbitrarily. For each assignment of weights we will be able to derive a bound on the cost of a sequence of accesses. We can choose the weight assignment that gives the best bound. By giving the frequently accessed elements a high weight, we will be able to get tighter bounds on the running time. Note that the weights are only used in the analysis, and do not change the algorithm at all. (A commonly used case is to assign all the weights to be 1.)

2.1 Sizes and Ranks

Before we can state our performance lemma, we need to define two more quantities. Consider any tree T — think of this as a tree obtained at some point of the splay tree algorithm, but all these definitions hold for any tree. The *size of a node* x in T is the total weight of all the nodes in the subtree rooted at x. If we let $T(x)$ denote the subtree rooted at x, then the size of x is denoted by

$$s(x) = \sum_{y \in T(x)} w(y)$$

Now we define the *rank of a node* x as

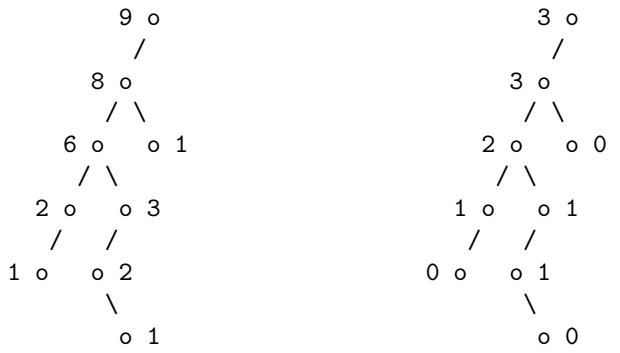
$$r(x) = \lfloor \log(s(x)) \rfloor$$

For each node x, we'll keep $r(x)$ tokens on that node if we are using the banker's method. (Alternatively, the *potential function* $\Phi(T)$ corresponding to the tree T is just the sums of the ranks of

all the nodes in the tree.)

$$\Phi(T) := \sum_{x \in T} r(x).$$

Here's an example to illustrate this: Here's a tree, labeled with sizes on the left and ranks on the right.



Notes about this potential Φ:

1. Doing a rotation between a pair of nodes x and y only effects the ranks of the nodes x and y, and no other nodes in the tree. Furthermore, if y was x's parent before the rotation, then the rank of y before the rotation equals the rank of x after the rotation.
2. Assuming all the weights are 1, the potential of a balanced tree is $O(n)$, and the potential of a long chain (most unbalanced tree) is $O(n \log n)$.
3. In the banker's view of amortized analysis, we can think of having $r(x)$ tokens on node x.

3 The Amortized Analysis

Lemma 1 (Access Lemma) *Take any tree T with root t, and any weights $w(\cdot)$ on the nodes. Suppose you splay node x; let T' be the new tree. Then*

$$\begin{aligned} \textit{amortized number of splaying steps} &= \textit{actual number of splaying steps} + (\Phi(T') - \Phi(T)) \\ &\leq 3(r(t) - r(x)) + 1. \end{aligned}$$

Proof: As we do the work, we must pay one token for each splay step we do. Furthermore we must make sure that we always leave $r(x)$ tokens on node x as we do our restructuring. We are going to allocate $3(r(t) - r(x)) + 1$ tokens to do the splay. Our job in the proof is to show that this is enough.

First we need the following observation about ranks, called the Rank Rule.

> **Rank Rule:** Suppose that two siblings have the same rank, r. Then the parent has rank at least $r + 1$.

This is because if the rank is r, then the size is at least 2^r. If both siblings have size at least 2^r, then the total size is at least 2^{r+1} and we conclude that the rank is at least $r + 1$. We can represent this with the following diagram:

```
  x
 / \       Then x >= r+1
r   r
```

Conversly, suppose we find a situation where a node and its parent have the same rank, r. Then the other sibling of the node must have rank $< r$. So if we have three nodes configured as follows, with these ranks:

```
  r
 / \       Then x < r
x   r
```

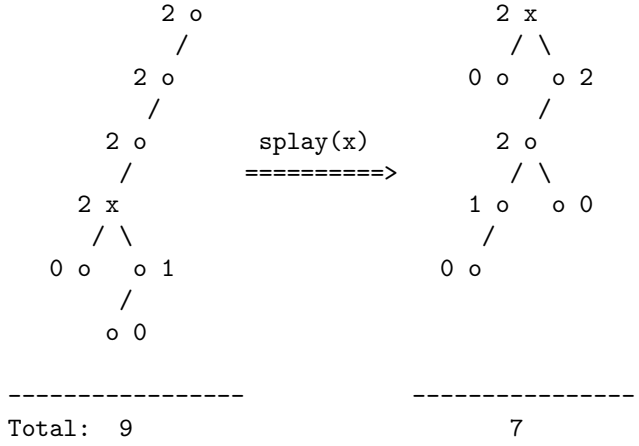
Now we can go back to proving the lemma. The approach we take is to show that $3(r(z) - r(x))$ tokens are sufficient to pay for a zig-zag or a zig-zig step. (Here z refers to x's grandparent, as in the definition of splaying.) And that $3(r(y) - r(x)) + 1$ is sufficient to pay for the zig step.

We can express this a little bit differently. Take the zig-zag amortized cost $3(r(z) - r(x))$ mentioned above. We can write this as $3(r'(x) - r(x))$, where $r'() = r_{T'}()$ is the rank function after the step, and $r() = r_T()$ represents the rank function before the step. This identity holds because the rank of x after the step is the same as the rank of z before the step.

Using this notation it becomes obvious that when we sum these costs to compute the amortized cost for the entire splay operation, they telescope to:

$$3(r(t) - r(x)) + 1.$$

Note that the +1 comes from the zig step, which can happen only once. Here's an example, the labels are ranks:

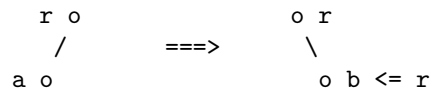


We allocated: 3(2-2)+1 = 1 tokens that are supposed to pay for the splay of x. There are two more tokens in the tree before than are needed at the end. (The potential decreases by two.) So we have a total of 3 tokens to spend on the splay steps of this splay.

But there are 2 splay steps. So $3 > 2$, and we have enough.

It remains to show these bounds for the individual steps. There are three cases, one for each of the types of splay steps.

The Zig Case: The following diagram shows the ranks of the two nodes that change as a result of the zig step. We've allocated $3(r - a) + 1$ to pay for the step. We'll show that this is sufficient.



The actual cost is 1, so we spend one token on this. We take the tokens on a and augment them with another $r - a$ (resulting in a pile of r tokens) and put them on b. Thus the total number of tokens needed is $1 + r - a$. This is at most $1 + 3(r - a)$.

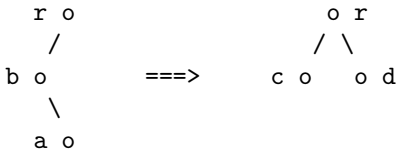
The Zig-zag Case: We'll split it further into 2 cases:

Case 1: The rank does not increase between the starting node and ending node of the step.

By the Rank Rule, one of a or b must be $< r$, so one token is released from the data structure. we use this to pay for the work.

Thus, our allocation of $3(r - r) = 0$ is sufficient to pay for this.

Case 2: The rank does increase between the starting node and ending node of the step.



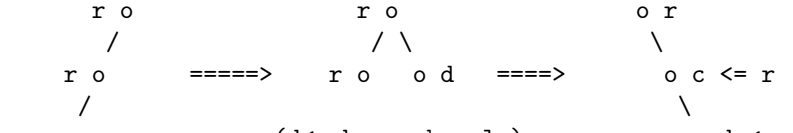
Note that $r - a > 0$. So use $r - a$ (which is at least 1) to pay for the work. Use $r - a$ to augment the tokens on a to create a pile of r tokens which can supply c. Finally use $r - b$ ($\leq r - a$) tokens to add to the pile on b to make a pile of r tokens, which are enough to cover d.

Thus $3(r - a)$ tokens are sufficient, which completes the zig-zag case.

(We actually only need $2(r - a)$ because it is easily shown that $c \leq b$.)

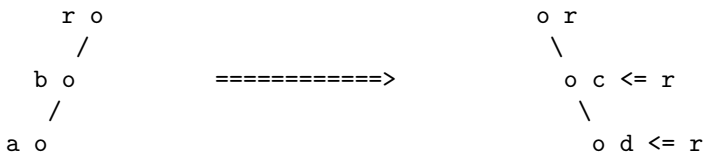
The Zig-zig Case: Again, we split into two cases.

Case 1: The rank does not increase between the starting node and the ending node of the step.



As in the zig-zag case, we use the token gained because of $d < r$ to pay for the step. Thus we have $3(r - r) = 0$ tokens and we need 0.

Case 2: The rank does increase between the starting node and the ending node of the step.



We can use $r - a$ (which is at least 1) to pay for the work, use $r - a$ to boost the tokens on a to cover those needed for d, and use $r - a$ to boost the tokens on b to cover those needed for c.

Summing these three gives $3(r - a)$, which completes the analysis of the zig-zig case.

So in every case we have shown that $3(r'(x) - r(x))$ is sufficient to pay for the splay step (modulo the extra $+1$ needed for the zig case). Thus these amortized costs telescope giving an amortized number of splay steps of $3(r(t) - r(x)) + 1$.

■

4 Balance Theorem

Using the Access Lemma we can prove the bound on the amortized cost of splaying: this is captured in the Balance Theorem.

Theorem 2 (Balance Theorem) *A sequence of m splays in a tree of n nodes takes time*

$$O(m \log n + n \log n).$$

Proof: Suppose all the weights equal 1. Then the access lemma says that if we splay node x in tree T to get the new tree T'

$$\begin{aligned} \text{actual number of splaying steps } + \big(\Phi(T') - \Phi(T)\big) &\leq 3(\log_2 n - \log_2 |T(x)|) + 1 \\ &\leq 3 \log_2 n + 1. \end{aligned}$$

Hence, if we start off with a tree T_0 of size at most n and perform any sequence of m splays to it

$$T_0 \xrightarrow{splay} T_1 \xrightarrow{splay} T_2 \xrightarrow{splay} \cdots \xrightarrow{splay} T_m,$$

repeatedly using this inequality m times shows:

$$\text{actual total number of splaying steps } + (\Phi(T_m) - \Phi(T_0)) \leq m(3 \log_2 n + 1).$$

In any tree T with unit weights, each $s(x) \leq n$ so each $r(x) \leq \log_2 n$ so $\Phi(T) \leq n \log_2 n$; also $\Phi(T) \geq 0$. Rearranging, we get

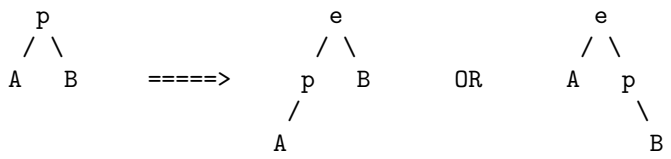
$$\begin{aligned} \text{actual total number of splaying steps } &\leq m(3 \log_2 n + 1) + (\Phi(T_0) - \Phi(T_m)) \\ &\leq O(m \log n) + O(n \log n). \end{aligned}$$

This proves the Balance Theorem. ■

5 Using Splaying with Searching and Updates

Searching: Since the keys in the splay tree are stored in in-order, the usual BST searching algorithm will suffice to find a node (or tell you that it is not there). However with splay trees it is necessary to splay the last node you touch in order to pay for the work of searching for the node. (This is clear if you think about what would happen if you repeatedly searched for the deepest node in a very unbalanced tree without every splaying.)

Insertion: Say we want to insert a new key `e` into the tree. Proceed like ordinary binary search tree insertion, but stop short of linking the new item `e` into the tree. Let the node you were about to link `e` to be called `p`. We splay `p` to the root. Now construct a new tree according to one of the following pictures:



The choice of which pattern to use on the right is determined by whether `e` is less than `p` or greater than it. Note that after this operation one of `p`'s children will be empty.

Let's analyze the amortized cost of this opertion when all the weights are 1 and the tree initially has n nodes in it. The amortized cost of the splay is $O(\log n)$ according to the Access Lemma. This also pays for the work of finding node `p`. But we're not done yet, because attaching `e` to the tree changes the potential, and we have to take this into account. The number of tokens on `p` can only decrease (its size can not increase). However we have to populate `e` with tokens, the number of which is at most $\lfloor \log(n+1) \rfloor = O(\log n)$. Thus the total amortized cost of the insertion is $O(\log n)$.

An alternative insertion algorithm is to do the ordinary binary tree insertion (link `e` to `p`), and then simply splay `e` to the root. This also has efficient amortized cost, but the analysis is a bit more complex.

Join: Here we have two trees, say, A, and B. We want to put them together with all the items in A to the left of all the items in B. This is called the *Join* operation. This is done as follows. Splay the rightmost node of A. Now make B the right child of this node. The amortized cost of the operation is easily seen to be $O(\log n)$ where n is the number of nodes in the tree after joining. (As in the case of insertion, we are using the uniform weight assignment to do this analysis.)

Delete: To delete a node `x`, simply splay it to the root and then Join its left and right subtrees together as described above. The amortized cost is easily seen to be $O(\log n)$ where n is the size of the tree before the deletion.

6 Additional Applications of the Access Lemms

One of the things that distinguishes splay trees from other forms of balanced trees is the fact that they are provably more efficient various natural access patterns. These theorems are proven by playing with the weight function in the definition of the potential function. In this section we describe some of these theorems.

The following theorem shows that splay trees perform within a constant factor of any static tree.

Theorem 3 (Static Optimality Theorem) *Let T be any static search tree with n nodes. Let t be the number of comparisons done when searching for all the nodes in a sequence s of accesses. (This sum of the depths of all the nodes). The cost of splaying that sequence of requests, starting with any initial splay tree is $O(n^2 + t)$.*

The following theorem says that if we measure the cost of an access by the log of the distance to a specific element called the finger, then this is a bound on the actual cost (modulo an additive term). There is a counter-intuitive aspect to this, which is that the algorithm achieves this without knowing which element you've picked to be the finger.

An alternative interpretation is that if the accesses have spatial locality (they cluster around a spacific place) then the sequence is cheap to process.

Theorem 4 (Static Finger Theorem) *Consider a sequence of m accesses in an n-node splay tree. Let $a[j]$ denote the jth element accessed. Let f be a specific element called the finger. Finally*

let $|e - f|$ denote the distance (in the in-order list of the tree) between elements e and f. Then the following is a bound on the cost of splaying the sequence:

$$O(m + n \log n + \sum_{1 \le j \le m} \log(|f - a[j]| + 1))$$

If the previous theorem addresses spatial locality in the sequence, the following one address temporal locality. If I access an element, then shortly after this, I access it again, then the second access is cheap. (It's the log of the number of different items accessed between the two.)

Theorem 5 (Working Set Theorem) *In a sequence of m accesses in an n-node splay tree, consider the jth access, which is to an item x. Let $t(j)$ be the number of different items accessed between the current access to x and the previous one. (If there is no previous access, then $t(j) = n$ the size of the tree.) Then the total cost of the access sequence is*

$$O(m + n \log n + \sum_{1 \le j \le m} \log(t(j) + 1)$$

The following two theorems also addresses spatial locality, in a different way from the Static Finger Theorem. Their proofs do not follow from the Access Lemma.

Theorem 6 (Sequential Access Theorem) *The cost of the access sequence that accesses each of the n items in the tree in left-to-right order (in-order) is $O(n)$.*

The following theorem generalizes the Sequential Access Theorem. It says that the cost of splaying an element can be tied to the log of the distance between the current element being splayed and the one that was just splayed.

Theorem 7 (Dynamic Finger Theorem) *Incorporation the notation from the Static Finger Theorem, the cost of an access sequence is bounded by*

$$O(m + n + \sum_{2 \le j \le n} \log(|a[j-1] - a[j]| + 1))$$

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

1 Motivation

To motivate the union-find problem, let's recall Kruskal's Algorithm for finding a minimum spanning tree (MST) in an undirected graph (see also Section 5). Remember that an MST is a tree that includes (i.e., spans) all the vertices and out of all such trees has the least total cost.

Kruskal's Algorithm (recap):

> Sort the edges in the given graph G 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 argue correctness in Section 5.2. 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.

2 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?".

3 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 (denoted by `x->head`). 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 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. So we could use this to get $O(m \log m)$ overall runtime for Kruskal.

4 Data Structure 2 (tree-based)

But even though the running time of the list-based data structure is pretty fast, let's think of ways we could make it even faster. How fast *can* we make a union-find data structure?

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 α 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.

(A) 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.

(B) 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.

(C) This means that when we do path compression, if x's parent changes, then the rank of x's new parent is *strictly more* than the rank of x's old parent.

(D) 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.

(E) 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 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 \$2 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 $2^r - 1$. *We'll denote the bucket as* $[r, 2^r - 1]$.

In total, we have $O(\lg^* n)$ buckets.

How many nodes have ranks in bucket $[r, 2^r - 1]$? All these nodes have ranks at least r, so by property (E) of ranks above, this is at most $\frac{n}{2^r} = \frac{n}{\text{upper bound of bucket} + 1}$.

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 last two steps, when we touch the root, or a child of the root, we'll charge to the walk-er again, but this is only \$2.

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 u. The argument is careful but simple.

1. The node u charged is not a root, so its rank is never going to change, by Property (D).

2. 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 (C).

3. One worry: the rank of u's parent could conceivably increase $\log n$ times, which to us is a "big" number. Hmm.

   But — and this is the crucial idea — 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 maximum charge any walk-ee node u gets is the range of his bucket.

4. A bucket $[r, 2^r - 1]$ has range $\leq 2^r$, and has at most $n/2^r$ elements in it, by Property (E). So, the total charge to all walk-ees in this bucket over the entire course of the algorithm is at most $(n/2^r)2^r = n$.

   (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.)

5. There are $O(\lg^* n)$ buckets, so the *total* charge of this kind summed *over all buckets* is at most n per bucket, times the number of buckets, which is $O(n \lg^* n)$.

Simple, and beautiful. Just as advertised. ■

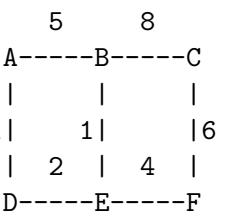
5 Appendix: MST Algorithms

Many of you have seen minimum spanning tree algorithms in previous courses, e.g., in 15-210 you saw Boruvka's algorithm, which is naturally parallel and runs in time $O(m \log n)$. But let us recap the basic definitions, and talk about two other algorithms: Prim's and Kruskal's.

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?



5.1 Prim's algorithm

Prim's algorithm is an MST algorithm that works much like Dijkstra's algorithm does for shortest path trees, if you are familiar with that. 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.

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 3 *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: To implement this efficiently, we can store the neighbors of the current tree in a priority-queue (pqueue), with priority-value equal to the length of the shortest edge between that node and the current tree. We add a new node into the tree using a remove-min operation (taking the node of smallest priority-value out of the pqueue); then, after adding this node, we examine all outgoing edges and for each one that points to a node not in the tree we either (a) add it into the pqueue if it is not there already, or (b) perform a "decrease-key" operation if it was in there already but the new edge is shorter. This will give us $O(m \log n)$ running time if we implement the pqueue using a standard heap, or $O(m + n \log n)$ running time if we use something called a Fibonacci heap.

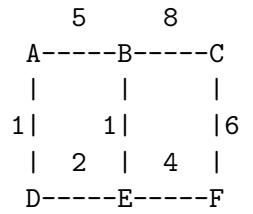
5.2 Kruskal's algorithm

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 4 *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? This is just the union-find data structure you saw in this lecture! It is so efficient that union/finds will actually will be a low-order cost compared to the sorting step.

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.

1 Maintaining a Dictionary

While describing the desired properties, let us keep one application in mind. We want to maintain a dictionary. We have a large universe of "keys" (say the set of all strings of length at most 80 using the Roman alphabet), denoted by U. The actual dictionary (say the set of all English words) is some subset S of this universe. S is typically much smaller than U. The operations we want to implement are:

- `add(x)`: add the key x to S.
- `query(q)`: is the key $q \in S$?
- `delete(x)`: remove the key x from S.

In some cases, we don't care about adding and removing keys, we just care about fast query times—e.g., the actual English dictionary does not change (or changes very gradually). This is called the *static case*. Another special case is when we just add keys: the *insertion-only case*. The general case is called the *dynamic case*.

For the static problem we could use a sorted array with binary search for lookups. For the dynamic 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.

2 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. (This is called "separate chaining".) 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 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?

2.1 A Key Idea

Here is a key idea: let's use randomization in our *construction* of h, in analogy to randomized quicksort. (The function 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".

3 Universal Hashing

Definition 2 *A randomized algorithm H for constructing hash functions $h : U \to \{0, \ldots, M-1\}$ is* **universal** *if for all $x \neq y$ in U, we have*

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

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.)*

Make sure you understand the definition! This condition must hold for *every pair* of distinct keys $x \neq y$, and the randomness is over the choice of the actual hash function h from the set H.

> Here's an equivalent way of looking at this. First, count the number of hash functions in H that cause a and b to collide. This is
>
> $$|\{h \in H | h(a) = h(b)\}|.$$
>
> Divide this number by $|H|$, the number of hash functions. This is the probability on the left hand side of (1). So, to show universality you want
>
> $$\frac{|\{h \in H \mid h(a) = h(b)\}|}{|H|} \leq \frac{1}{m}$$
>
> for every $a \neq b \in U$.

Here are some examples and exercises to help you become comfortable with the definition.

> **Example 1:** The following three hash families with hash functions mapping the set $\{a, b\}$ to $\{0, 1\}$ are universal, because at most $1/M$ of the hash functions in them cause a and b to collide, were $M = |\{0, 1\}|$.
>
> | | a | b |
> |---|---|---|
> | h_1 | 0 | 0 |
> | h_2 | 0 | 1 |
>
> | | a | b |
> |---|---|---|
> | h_1 | 0 | 1 |
> | h_2 | 1 | 0 |
>
> | | a | b |
> |---|---|---|
> | h_1 | 0 | 0 |
> | h_2 | 1 | 0 |
> | h_3 | 0 | 1 |
>
> On the other hand, this next two hash families are not, since a and b collide with probability more than $1/M = 1/2$.
>
> | | a | b |
> |---|---|---|
> | h_1 | 0 | 0 |
> | h_3 | 1 | 1 |
>
> | | a | b | c |
> |---|---|---|---|
> | h_1 | 0 | 0 | 1 |
> | h_2 | 1 | 1 | 0 |
> | h_3 | 1 | 0 | 1 |

3.1 Using Universal Hashing

Theorem 3 *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 the random variable $C_{xy} = 1$ if x and y collide and 0 otherwise.
- Let C_x be the r.v. denoting the total number of collisions for x. So, $C_x = \sum_{y \in S, y \neq x} C_{xy}$.
- We know $\mathrm{E}[C_{xy}] = \Pr(x \text{ and } y \text{ collide}) \leq 1/M$.
- So, by linearity of expectation, $\mathrm{E}[C_x] = \sum_y \mathrm{E}[C_{xy}] < N/M$. ■

We now immediately get the following corollary.

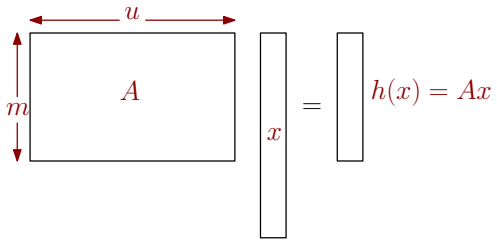
Corollary 4 *If H is universal then for any* **sequence** *of L insert, lookup, and delete operations in which there are at most M elements in the system at any one time, the expected total cost of the L operations for a random $h \in H$ is only $O(L)$ (viewing the time to compute h as constant).*

Proof: For any given operation in the sequence, its expected cost is constant by Theorem 3, so the expected total cost of the L operations is $O(L)$ by linearity of expectation. ■

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

3.2 Constructing a universal hash family: the matrix method

Let's say keys are u-bits long. Say the table size M is power of 2, so an index is b-bits long with $M = 2^b$. What we will do is pick h to be a random b-by-u 0/1 matrix, and define $h(x) = hx$, where we do addition mod 2. These matrices are short and fat. For instance:



Claim 5 *For $x \neq y$, $\Pr_h[h(x) = h(y)] = 1/M = 1/2^b$.*

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

Now, take an arbitrary pair of keys x, y such that $x \neq y$. They must differ someplace, so say they differ in the ith coordinate and for concreteness say $x_i = 0$ and $y_i = 1$. Imagine we first choose all of h but the ith column. Over the remaining choices of ith column, $h(x)$ is fixed. However, each

of the 2^b different settings of the ith column gives a different value of $h(y)$ (in particular, every time we flip a bit in that column, we flip the corresponding bit in $h(y)$). So there is exactly a $1/2^b$ chance that $h(x) = h(y)$. ■

There are other methods to construct universal hash families based on multiplication modulo primes as well (see Section 5.1).

One last note: there is a closely-related concept called an "ℓ-universal" or "ℓ-wise-independent" hash function. A hash function family is ℓ-universal if for every set of ℓ distinct keys $x_1, x_2, \ldots, x_\ell$ and every set of ℓ values $v_1, v_2, \ldots, v_\ell \in \{0, \ldots, M-1\}$, we have

$$\Pr_{h \leftarrow H}[h(x_1) = v_1 \text{ AND } h(x_2) = v_2 \text{ AND} \ldots \text{ AND } h(x_\ell) = v_\ell] = 1/M^\ell.$$

It is easy to see that if H is 2-universal then it is universal. Note that the above matrix-based hash family was not 2-universal since the hash functions all mapped 0 to 0.

> **Exercise 1:** Show that if we choose $A \in \{0,1\}^{b \times u}$ and $b \in \{0,1\}^b$ independently and uniformly at random, then the hash family $h(x) = Ax + b$ is 2-universal.

4 Perfect Hashing

The next question we consider is: if we fix the set S (the dictionary), 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*. 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.

4.1 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 6 *If H is universal and $M = N^2$, then $\Pr_{h \sim H}(\text{no collisions in } S) \geq 1/2$.*

Proof:

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

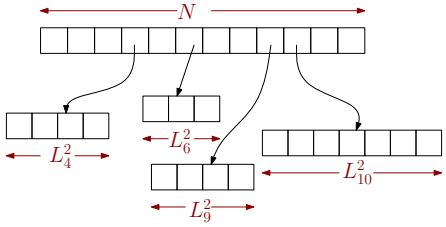
This is like the other side to the "birthday paradox". If the number of days is a lot *more* than the number of people squared, then there is a reasonable chance *no* pair has the same birthday.

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

4.2 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 L_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 $\sum_i (L_i)^2$. What remains is to show that we can find a first-level function h such that $\sum_i (L_i)^2 = O(N)$. In fact, we will show the following:

Theorem 7 *If we pick the initial h from a universal set H, then*

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

Proof: We will prove this by showing that $\mathrm{E}[\sum_i (L_i)^2] < 2N$. 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:

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

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

5 Further discussion

5.1 Another method for universal hashing

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

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

$$h(x) = r_1x_1 + r_2x_2 + \ldots + r_kx_k \mod M.$$

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 $h'(x) + r_i x_i = h'(y) + r_i y_i \mod M$, or equivalently when

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

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 $r_i = (h'(y) - h'(x))/(x_i - y_i) \mod M$. So, the probability of this occuring is exactly $1/M$.

15-451/651: Design & Analysis of Algorithms February 3, 2016
Lecture #7: Streaming Algorithms last changed: January 31, 2016

Today we'll talk about a topic that is both very old (as far as computer science goes), and very current. It's a model of computing where the amount of space we have is much less than the amount of data we examine, and hence we can store only a "summary" or "sketch" of the data. Back at the dawn of CS, data was stored on tapes and the amount of available RAM was very small, so people considered this model. And now, even when RAM is cheap and our machines have gigabytes of RAM and terabytes of disk space, it may be listening to an Ethernet cable carrying many gigabytes of data *per second*. *What can we compute in this model?*

- An introduction to the data streaming model, and its concerns.
- An algorithm for heavy hitters in the arrivals-only model.
- An algorithm for heavy hitters with both arrivals and departures.

---

1 Introduction

Today's lecture will be about a slightly different computational model called the *data streaming* model. In this model you see elements going past in a "stream", and you have very little space to store things. For example, you might be running a program on an Internet router, the elements might be IP Addresses, and you have limited space. You certainly don't have space to store all the elements in the stream. The question is: which functions of the input stream can you compute with what amount of time and space? (For this lecture, we will focus on space, but similar questions can be asked for update times.)

We will denote the stream elements by

$$a_1, a_2, a_3, \ldots, a_t, \ldots$$

We assume each stream element is from alphabet Σ and takes b bits to represent. For example, the elements might be 32-bit integers IP Addresses. We imagine we are given some function, and we want to compute it continually, on every prefix of the stream. Let us denote $a_{[1:t]} = \langle a_1, a_2, \ldots, a_t \rangle$.

Let us consider some examples. Suppose we have seen the integers

$$3, 1, 17, 4, -9, 32, 101, 3, -722, 3, 900, 4, 32, \ldots \qquad (\diamond)$$

- Computing the sum of all the integers seen so far? $F(a_{[1:t]}) = \sum_{i=1}^{t} a_i$. We want the outputs to be

  $$3, 4, 21, 25, 16, 48, 149, 152, -570, -567, 333, 337, 369, \ldots$$

  If we have seen T numbers so far, the sum is at most $T2^b$ and hence needs at most $O(b+\log T)$ space. So we can keep a counter, and when a new element comes in, we add it to the counter.

- How about the maximum of the elements so far? $F(a_{[1:t]}) = \max_{i=1}^{t} a_i$. Even easier. The outputs are:

  $$3, 1, 17, 17, 17, 32, 101, 101, 101, 101, 900, 900, 900$$

  We just need to store b bits.

- The median? The outputs on the various prefixes of $(\diamond)$ now are

  $$3, 1, 3, 3, 3, 3, 4, 3, \ldots$$

  And doing this will small space is a lot more tricky.

- ("distinct elements") Or the number of distinct numbers seen so far? You'd want to output:

$$1, 2, 3, 4, 5, 6, 7, 7, 8, 8, 9, 9, 9 \ldots$$

- ("heavy hitters") Or the elements that have appeared at least ε fraction of the times so far? Hmm...

You can imagine the applications of this model. An Internet router might see a lot of packets whiz by, and may want to figure out which data connections are using the most space? Or how many different connections have been initiated since midnight? Or the median (or the 90^{th} percentile) of the file sizes that have been transferred. Which IP connections are "elephants" (say the ones that have used more than 0.01% of your bandwidth)? Even if you are not working at "line speed",[1] but just looking over the server logs, you may not want to spend too much time to find out the answers, you may just want to read over the file in one quick pass and come up with an answer. Such an algorithm might also be cache-friendly. But how to do this? Two of the recurring themes will be:

- Approximate solutions: in several cases, it will be impossible to compute the function exactly using small space. Hence we'll explore the trade-offs between approximation and space.
- Hashing: this will be a very powerful technique.

2 Finding ε-Heavy Hitters

Let's formalize things a bit. We have a data stream with elements $a_1, a_2, \ldots, a_t$ seen by time t. Think of these elements as "arriving" and being added to a multiset S_t, with $S_0 = \emptyset$, $S_1 = \{a_1\}$, $\ldots, S_i = \{a_1, a_2, \ldots, a_i\}$, etc. Let

$$\texttt{count}_t(e) = \{i \in \{1, 2, \ldots, t\} \mid a_i = e\}$$

be the number of times element e has been seen in the stream so far. The *multiplicity* of e in S_t.

Call element $e \in \Sigma$ is called a *ε-heavy hitter* at time t if $\texttt{count}_t(e) > \varepsilon t$. That is, e constitutes *strictly more than* ε fraction of the elements that arrive by time t.

The goal is simple — given an threshold $\varepsilon \in [0, 1]$, maintain a data structure that is capable of outputting ε-heavy-hitters. At any point in time, we can query the data structure, and it outputs a set of at most $1/\varepsilon$ elements that contains all the ε-heavy hitters.[2]

> **Crucial note:** *it's OK to output "false positives" but we are not allowed "false negatives". I.e., we're not allowed to miss any heavy-hitters, but we could output non-heavy-hitters. (Since we only output $1/\varepsilon$ elements, there can be $\leq 1/\varepsilon$-false positives.)*

For example, if we're looking for $\frac{1}{3}$-heavy-hitters, and the stream is

$$E, D, B, D, D_{\ 5}, D, B, A, C, B_{\ 10}, B, E, E, E, E_{\ 15}, E \cdots \qquad (\star)$$

(the subscripts are not part of the stream, just to help you count) then

- at time 5, the element D is the only $\frac{1}{3}$-heavy-hitter,
- at time 11 both B and D are $\frac{1}{3}$-heavy-hitters, and

[1] Such a router might see tens of millions of packets per second.

[2] Clearly, at any moment in time, there are at most $1/\varepsilon$ elements that are ε-heavy-hitters, so this request is not unreasonable.

- at time 15, there is no $\frac{1}{3}$-heavy-hitter, and
- at time 16, only E is a $\frac{1}{3}$-heavy-hitter.

Note that as time passes, the set of frequent elements may change completely, so an algorithm would have to be adaptive. We cannot keep track of the counts for all the elements we've seen so far, there may be a lot of different elements that appear over time, and we have limited space.

Any ideas for how to find some "small" set containing all the ε-heavy-hitters?

Hmm, one trick that is useful in algorithm design, as in problem solving, is to try simple cases first. Can you find a 1-heavy-hitter? (Easy: there is no such element.) How about a 0.99-heavy-hitter? Random sampling will also work (maybe you'll see this in a homework or recitation), but let's focus on a deterministic algorithm for right now.

2.1 Finding a Majority Element

Let's first try to solve the problem of finding a 0.5-heavy-hitter, a.k.a. a majority element, an element that occurs (strictly) more than half the time. Keeping counts of all the elements is very wasteful! How can we find a majority element while using only a little space?

Here's an algorithm (due to R. Boyer and J.S. Moore) that keeps very little information. (One element from Σ in a variable called `memory`, and one counter.)

```
Initially, memory = empty and counter = 0.

When element a_t arrives
  if (counter == 0)
    set memory = a_t and counter = 1
  else
    if a_t == memory
      counter++
    else
      counter--
      discard a_t
```

At the end, return the element in `memory`.

- Suppose there is no majority element, then we'll output a false positive. As we said earlier, that's OK: we want to output a set of size 1 that contains the majority element, *if any*.
- If there is a majority element, we *will* indeed output it. Why? Observe: when we discard an element a_t, we also throw away another (different) element as well. (Decrementing the counter is like throwing away one copy of the element in `memory`.) So every time we throw away a copy of the majority element, we throw away another element too. Since there are fewer than half non-majority elements, we cannot throw away all the majority elements.

A different (albeit somewhat gruesome) way of looking at this is as a "gang war". Every element is a gang member. When we have two members from different gangs, they shoot each other in the standoff. If there is a gang with more than $n/2$ members (a majority gang), it will be the only one whose members survive.

2.2 Finding an ε-heavy-hitter

It's possible to extend this idea to finding ε-heavy-hitters.[3] Set $k = \lceil 1/\varepsilon \rceil - 1$; for $\varepsilon = 1/2$, we'd get $k = 1$.

> *Keep an array $T[1 \dots k]$, each location can hold one element from Σ.*
> *And an array $C[1 \dots k]$ each location can hold a non-negative integer.*
>
> *Initialize $C[i] = 0$ and $T[i] = \perp$ for all i.*
>
> *When element a_t arrives.*
> *If there exists $j \in \{1, 2, \dots, k\}$ such that $a_t == T[j]$, then $C[j]++$.*
> *Else if some counter $C[j] == 0$ then set $T[j] \leftarrow a_t$ and $C[j] \leftarrow 1$.*
> *Else decrement all the counters by 1 (and discard element a_t).*

Exercise: Check that this algorithm is identical to the one above for $\varepsilon = 1/2$.

At any point t, we will maintain estimates of the counts. Our estimates are:

$$\texttt{est}_t(e) = \begin{cases} C[j] & \text{if } e == T[j] \\ 0 & \text{otherwise} \end{cases} \tag{1}$$

The main claim naturally extends the proof for the case of majority.

Lemma 1 *The estimated counts satisfy:*

$$0 \leq \texttt{count}_t(e) - \texttt{est}_t(e) \leq \frac{t}{k+1} \leq \varepsilon t$$

Proof: The estimated count $\texttt{est}_t(e)$ is at most the actual count, since we never increase a counter for e unless we see e. So $\texttt{count}_t(e) - \texttt{est}_t(e) \geq 0$. (In other words, $\texttt{est}_t(e) \leq \texttt{count}_t(e)$, it is always an underestimate.)

To prove the other inequality, think of the arrays C and T saying that we have $C[1]$ copies of $T[1]$, $C[2]$ copies of $T[2]$, etc., in our hand. Look at some time when the difference between $\texttt{count}_t(e)$ and $\texttt{est}_t(e)$ increases by 1. Each time this happens, we decrement k different counters by 1 and discard an element (which is not currently present in the array T). This is like dropping $k+1$ distinct elements (one of which is e). We can drop at most t elements until time t. So the gap can be at most $t/(k+1)$. And since $k = \lceil 1/\varepsilon \rceil - 1 \geq 1/\varepsilon - 1$, we get that $t/(k+1) \leq \varepsilon t$. ■

Corollary 2 *For every n, the set of items in the array T contains all the ε-heavy-hitters.*

Proof: After we've seen n elements, the ε-heavy-hitters have occurred at least $\texttt{count}_t(e) > \varepsilon t$ times. If e is a ε-heavy-hitter, by Lemma 1, the estimate

$$\texttt{est}_t(e) \geq \texttt{count}_t(e) - \varepsilon t > 0.$$

So element e must be in the array T. ■

To summarize: if we set $k = \lceil 1/\varepsilon \rceil - 1$, we get an algorithm that gives us element counts on a stream of length t to within an additive error of at most $(\varepsilon \cdot t)$ with space $O(1/\varepsilon) \cdot (\log \Sigma + \log t)$ bits. As a corollary we get an algorithm to find a set containing the ε-heavy-hitters.

[3] The extension to finding ε-heavy-hitters was given by J. Misra and D. Gries, and independently by R.M. Karp, C.H. Papadimitriou and S. Shenker.

3 Heavy Hitters with Deletions

In the above problem, we assumed that the elements were only being added to the current set at each time. We maintained an approximate count for the elements (up to an error of εt). Now suppose we have a stream where elements can be both added and removed from the current set. How can we give estimates for the counts?

Formally, each time we get an *update*, it looks like $(\texttt{add}, e)$ or $(\texttt{del}, e)$. We will assume that for each element, the number of deletes we see for it is at most the number of adds we see — the running counts of each element is non-negative. As an example, suppose the stream looked like:

$$(\texttt{add}, A), (\texttt{add}, B), (\texttt{add}, A), (\texttt{del}, B), (\texttt{del}, A), (\texttt{add}, C)$$

then the sets at various times are

$$S_0 = \emptyset, S_1 = \{A\}, S_2 = \{A, B\}, S_3 = \{A, A, B\}, S_4 = \{A, A\}, S_5 = \{A\}, S_6 = \{A, C\}, \ldots$$

The counts of the element are defined in the natural way: $\texttt{count}_3(A) = 2$ and $\texttt{count}_5(A) = 1$. Observe that the "active" set S_t has size $|S_t| = \sum_{e \in \Sigma} \texttt{count}_t(e)$, and its size can grow and shrink.

What do we want want now? We want a data structure to answer count queries approximately. Specifically, at any point in time t, for any element, we should be able to ask "What is $\texttt{count}_t(e)$?" The data structure should respond with an estimate $\texttt{est}_t(e)$ such that

$$\Pr\left[\underbrace{\left|\texttt{est}_t(e) - \texttt{count}_t(e)\right|}_{\text{error}} \leq \underbrace{\varepsilon |S_t|}_{\text{is small}}\right] \geq \underbrace{1 - \delta}_{\text{with high probability}}.$$

Again, we would again like to use small space — perhaps even close to the

$$O(1/\varepsilon) \cdot (\log \Sigma + \log |S_t|) \text{ bits of space}$$

we used in the above algorithm. (As you'll see, we'll get close.)

3.1 A Hashing-Based Solution: First Cut

We're going to be using hashing for this approach, simple and effective. We'll worry about what properties we need for our hash functions later, for now assume we have a hash function $h : \Sigma \to \{0, 1, \ldots, k-1\}$ for some suitably large integer k. Maintain an array $A[1 \ldots k]$ capable of storing non-negative integers.

```
When update a_t arrives
    If (a_t == (add, e))
      then A[h(e)]++;
    else // a_t == (del, e)
      A[h(e)]--;
```

This was the update procedure. And what is our estimate for the number of copies of element e in our active set S_t? It is

$$\texttt{est}_t(e) := A(h(e)).$$

In words, we look at the location $h(e)$ where e gets mapped using the hash function h, and look at the count $A[h(x)]$ stored at that location. What does it contain? It contains the current count for

element e for sure. But added to it is the current count for any other element that gets mapped to that location. In math:

$$A(h(e)) = \sum_{e' \in \Sigma} \texttt{count}_t(e') \cdot \mathbf{1}(h(e') = h(e)),$$

where $\mathbf{1}$(some condition) is a function that evaluates to 1 when the condition in the parentheses is true, and 0 if it is false. We can rewrite this as

$$A(h(e)) = \texttt{count}_t(e) + \sum_{e' \neq e} \texttt{count}_t(e') \cdot \mathbf{1}(h(e') = h(e)), \tag{2}$$

or using the definition of the estimate, as

$$\texttt{est}_t(e) - \texttt{count}_t(e) = \sum_{e' \neq e} \texttt{count}_t(e') \cdot \mathbf{1}(h(e') = h(e)). \tag{3}$$

A great situation will be if no other elements $e' \neq e$ hashed to location $h(e)$ and the error (i.e., the summation on the right) evaluates to zero. But that may be unlikely. On the other hand, we can show that the expected error is not too much.

What's the expected error? Now we need to assume something good about the hash functions. Assume that the hash function h is a random draw from a universal family. Recall the definition from Lecture #5.

Definition 3 *A family H of hash functions from $\Sigma \to R$ is universal if for any pair of distinct keys $x_1, x_2 \in \Sigma$ with $x_1 \neq x_2$,*

$$\Pr[h(x_1) = h(x_2)] \leq \frac{1}{|R|}.$$

In other words, if we just look at two keys, the probability that they collide is no more than if we chose to map them randomly into the range R. We gave a construction where each hash function in the family used $(\lg k) \cdot (\lg |\Sigma|)$ bits to specify.

Good. So we drew a hash function h from this universal hash family H, and we used it to map elements to locations $\{0, 1, \ldots, k-1\}$. What is its expected error? From (3), it is

$$\begin{aligned} E\left[\sum_{e' \neq e} \texttt{count}_t(e') \cdot \mathbf{1}(h(e') = h(e))\right] &= \sum_{e' \neq e} \texttt{count}_t(e') \cdot E\left[\mathbf{1}(h(e') = h(e))\right] && (4) \\ &= \sum_{e' \neq e} \texttt{count}_t(e') \cdot \Pr[h(e') = h(e)] \\ &\leq \sum_{e' \neq e} \texttt{count}_t(e') \cdot (1/k) && (5) \\ &= \frac{|S_t| - \texttt{count}_t(e)}{k} \leq \frac{|S_t|}{k}. \end{aligned}$$

We used linearity of expectations in equality (4). To get (5) from the previous line, we used the definition of a universal hash family. Let's summarize:

Claim 4 *The estimator $\texttt{est}_t(e) = A(h(e))$ ensures that*

(a) $\texttt{est}_t(e) \geq \texttt{count}_t(e)$, and
(b) $E[\texttt{est}_t(e)] - \texttt{count}_t(e) \leq |S_t|/k$.

The space used is: k counters, and $O((\log k)(\log \Sigma))$ to store the hash function.

That's pretty awesome. (But perhaps not so surprising, once you think about it.) Note that if we were only doing arrivals and no deletions, the size of S_t would be exactly t. So the expected error would be at most t/k, which is about the same as we were getting in Section 2 using an array of size $(k-1)$. But now we can also handle deletions!

What's the disadvantage? We only have a bound on the *expected error* $E[\texttt{est}_t(e)] - \texttt{count}_t(e)$. It is no longer deterministic. Also, the expected error being small is weaker than saying: we have small error with high probability. So let's see how to improve things.

3.2 Amplification of the Success Probability

Any ideas how to *amplify* the probability that we are close to the expectation? The idea is simple: *independent repetitions*. Let us pick m hash functions $h_1, h_2, \ldots, h_m$. Each $h_i : \Sigma \to \{0, 1, \ldots, k-1\}$. How do we choose these hash functions? *Independently* from the universal hash family H.[4] We have m arrays $A_1, A_2, \ldots, A_m$, one for each hash function. The algorithm now just uses the i^{th} hash function to choose a location in the i^{th} array, and increments or decrements the same as before.

```
When update a_t arrives
  For each i from 1..m
    If (a_t == (add, e))
      then A_i[h_i(e)]++;
    else // a_t == (delete, e)
      A_i[h_i(e)]--;
```

And what is our new estimate for the number of copies of element e in our active set? It is

$$\texttt{best}_t(e) := \min_{i=1}^{m} A_i(h_i(e)).$$

In other words, each (h_i, A_i) pair gives us an estimate, and we take the least of these. It makes perfect sense — the estimates here are all *over*estimates, so taking the least of these is the right thing to do.

But how much better is this estimator? Let's do the math.

What is the chance that one single estimator has error more than $2|S_t|/k$? Remember the expected error is at most $|S_t|/k$. So by Markov's inequality

$$\Pr\left[error > 2 \cdot \frac{|S_t|}{k}\right] \leq \frac{1}{2}.$$

And what is the chance that all of the m copies have "large" (i.e., more than $2|S_t|/k$) error? The probability of m failures is

$$\begin{aligned}
&\Pr[\text{each of } m \text{ copies have large error}] \\
&= \prod_{i=1}^{m} \Pr[i^{th} \text{ copy had large error}] \\
&\leq (1/2)^m .
\end{aligned}$$

The first equality there used the independence of the hash function choices. (Only if events $\mathcal{A}, \mathcal{B}$ are independent you can use $\Pr[\mathcal{A} \wedge \mathcal{B}] = \Pr[\mathcal{A}] \cdot \Pr[\mathcal{B}]$.) And so the minimum of the estimates will have "small" error (i.e., at most $2|S_t|/k$) with probability at least $1 - (1/2)^m$.

[4] If we use the hash function construction given in Lecture #5 (Section 5.4.1), this means the $(\lg k) \cdot (\lg |\Sigma|)$-bit matrices for each hash function must be filled with independent random bits.

3.2.1 Final Bookkeeping

Let's set the parameters now. Set $k = 2/\varepsilon$, so that the error bound $2|S_t|/k = \varepsilon|S_t|$. And suppose we set $m = \lg 1/\delta$, then the failure probability is $(1/2)^m = \delta$, and our query will succeed with probability at least $1 - \delta$. [5]

Then on any particular estimate $\texttt{best}_t(e)$ we ensure

$$\Pr\left[\left|\texttt{best}_t(e) - \texttt{count}_t(e)\right| \leq \varepsilon|S_t|\right] \geq 1 - \delta.$$

Just as we wanted. And the total space usage is

$$m \cdot k \text{ counters} = O(\log 1/\delta) \cdot O(1/\varepsilon) = O(1/\varepsilon \log 1/\delta) \text{ counters.}$$

Each counter has to store at most $\lg T$-bit numbers after T time steps. [6]

Space for Hash Functions: We need to store the m hash functions as well. How much space does that use? The construction from Lecture #5 used $s := (\lg k) \cdot (\lg \Sigma)$ bits per hash function. Since $k = 1/\varepsilon$, the total space used for the functions is

$$m \cdot s = O(\log 1/\delta) \cdot (\lg 1/\varepsilon) \cdot (\lg \Sigma) \text{ bits.}$$

3.2.2 And in Summary...

Using about $1/\varepsilon \times$ poly-logarithmic factors space, and very simple hashing ideas, we could maintain the counts of elements in a data stream under both arrivals and departures (up to an error of $\varepsilon|S_t|$). As in the arrival-only case, these counts make sense only for very high-frequency elements.

[5] How small should you make δ? Depends on how many queries you want to do. Suppose you want to make a query a million times a day, then you could make $\delta = 1/10^9 \approx 1/2^{30}$ to get a 1-in-1000 chance that even one of your answers has high error. Our space varies linearly as $\lg 1/\delta$, so setting $\delta = 1/10^{18}$ instead of $1/10^9$ doubles the space usage, but drops the error probability by a factor of billion.

[6] So a 32-counter can handle a data stream of length 4 billion. If that is not enough, there are ways to reduce this space usage as well, look online for "approximate counters".

In today's lecture, we will talk about randomization and hashing in a slightly different way. In particular, we use arithmetic modulo prime numbers to (approximately) check if two strings are equal to each other. Building on that, we will get an randomized algorithm (called the *Karp-Rabin fingerprinting scheme*) for checking if a long text T contains a certain pattern string P as a substring.

1 How to Pick a Random Prime

In this lecture, we will often be picking random primes, so let's talk about that. (In fact, you do this when generating RSA public/private key pairs.)

How to pick a random prime in some range $\{1, \ldots, M\}$? The answer is easy.

- Pick a random integer X in the range $\{1, \ldots, M\}$.
- Check if X is a prime. If so, output it. Else go back to the first step.

How would you pick a random number in the prescribed range? Also easy. Pick a uniformly random bit string of length $\lfloor \log_2 M \rfloor + 1$. (We assume we have access to a source of random bits.) If it represents a number $\leq M$, output it, else repeat. The chance that you will get a number $\leq M$ is at least half, so in expectation you have to repeat this process at most twice.

How do you check if X is prime? You can use the Miller-Rabin randomized primality test (which may err, but it will only output "prime" when the number is composite with very low probability). There are other randomized primality tests as well, see the Wikipedia page. Or you can use the Agrawal-Kayal-Saxena primality test, which has a worse runtime, but is deterministic and hence guaranteed to be correct.

2 How Many Primes?

You have probably seen a proof that there are infinitely many primes. Here's a different question that we'll need for this lecture.

For positive integer n, how many primes are there in the set $\{1, 2, \ldots, n\}$?

Let there be $\pi(x)$ primes between 0 and x. One of the great theorems of the 20^{th} century was the Prime Number theorem which proved that

$$\lim_{n \to \infty} \frac{\pi(x)}{x/(\ln x)} = 1.$$

And while this is just a limiting statement, an older result of Chebyshev (from 1848) says that for $n \geq 2$,

$$\pi(n) \geq \frac{7}{8} \frac{n}{\ln n} = (1.262\ldots) \frac{n}{\lg n} > \frac{n}{\lg n}$$

As a bonus problem on HW#2, you can prove a slightly weaker version of this bound of $\frac{n}{2 \lg n}$. Here are two consequences of this theorem.

- A random integer between 1 and n is a prime number with probability at least $\frac{1}{\lg n}$.
- If we want at least $k \geq 4$ primes between 1 and n, it suffices to have $n \geq 2k \lg k$. [1]

The following even tighter set of bounds were proved by Pierre Dusart in 2010. For all $x \geq 60184$ we have:

$$\frac{x}{\ln x - 1.1} > \pi(x) > \frac{x}{\ln x - 1}$$

Because this is a two-sided bound, it allows us to deduce a lower bound on the number of primes in a range. For example, the number of 9-digit prime numbers (i.e. primes in the range $[10^8, 10^9 - 1]$) is

$$\pi(10^9 - 1) - \pi(10^8 - 1) > \frac{10^9 - 1}{\ln(10^9 - 1) - 1} - \frac{10^8 - 1}{\ln(10^8 - 1) - 1.1} = 44928097.3\ldots$$

From this we can infer that a randomly generated 9 digit number is prime with probability at least $0.049920\ldots$. Thus, the random sampling method would take at most 21 iterations in expectation to find a 9-digit prime.

3 The String Equality Problem

Here's a simple problem: we're sending a Mars lander. Alice, the captain of the Mars lander, receives an N-bit string x. Bob, back at mission control, receives an N-bit string y. Alice knows nothing about y, and Bob knows nothing about x. They want to check if the two strings are the same, i.e., if $x = y$. [2]

One way is for Alice to send the entire N-bit string to Bob. But N is very large. And communication is super-expensive between the two of them. So sending N bits will cost a lot. *Can Alice and Bob share less communication and check equality?*

If they want to be 100% sure that $x = y$, then one can show that fewer than N bits of communication between them will not suffice. But suppose we are OK with being correct with probability 0.9999. Formally, we want a way for Alice and Bob to send a message to Bob so that, at the end of the communication:

- If $x = y$, then Pr[Bob says `equal`] = 1.
- If $x \neq y$, then Pr[Bob says `unequal`] = $1 - \delta$.

Here's a protocol that does almost that.

1. Alice picks a random prime p from the set $\{1, 2, \ldots, M\}$ for $M = \lceil 2 \cdot (5N) \cdot \lg(5N) \rceil$.
2. She sends Bob the prime p, and also the value $h_p(x) := x \mod p$.
3. Bob checks if $h_p(x) = y \mod p$. If so, he says `equal` else he says `unequal`.

Let's see how this protocol performs.

[1] Indeed, we get $\pi(2k \ln k) \geq \frac{2k \ln k}{\lg(2k \lg k)} \geq \frac{2k \ln k}{\lg 2 + \lg k + \lg \lg k} \geq k$.

[2] E.g., this could be the latest update to the lander firmware, and we want to make pretty sure the file did not get corrupted in transition.

Lemma 1 *If $x = y$, then Bob always says* `equal`*.*

Proof: Indeed, if $x = y$, then $x \mod p = y \mod p$. So Bob's test will always succeed. ■

Lemma 2 *If $x \neq y$, then* Pr[*Bob says* `equal`] ≤ 0.2*.*

Proof: Consider x and y and N-bit binary numbers. So $x, y < 2^N$. Let $D = |x - y|$ be their difference. Bob says `equal` only when $x \mod p = y \mod p$, or equivalently $(x - y) = 0 \mod p$. This means p divides $D = |x - y|$. In words, the random prime p we picked happened to be a divider of D. What are the changes of that? Let's do the math.

The difference D is a N-bit integer, so $D \leq 2^N$. So D can be written (uniquely) as $D = p_1 p_2 \cdots p_k$, each p_i being a prime, where some of the primes might repeat[3]. Each prime $p_i \geq 2$, so $D = p_1 p_2 \cdots p_k \geq 2^k$. Hence $k \leq N$. The difference D has at most N divisors. The probability that p is one of them is

$$\frac{N}{\text{number of primes in } \{1, 2, \ldots, \lceil 10N \lg 5N \rceil\}}.$$

Finally, from the discussion in the previous section, the number of primes in $\{1, 2, \ldots, \lceil 10N \lg 5N \rceil\}$ is at least $5N$. So

$$\Pr[\text{ Bob says } \texttt{equal} \text{ and hence errs }] \leq \frac{1}{5}.$$

■

3.1 Reducing the Error Probability

If you don't like the probability of error being 20%, there're two ways to reduce the probability of error.

Approach #1: Have Alice choose a random prime from a larger set. For some integer $s \geq 1$, if we choose $M = 2 \cdot sN \lg(sN)$, then the arguments above show that the number of primes in $\{1, \ldots, M\}$ is at least sN. And hence the probability of error is $1/s$. Now choose any s large enough.

Approach #2: Just have Alice repeat this process 10 times independently, with Bob saying `equal` if and only if all in 10 repetitions, the test passes. The chance that he will make an error (i.e., say `equal` when $x \neq y$) is only

$$(1/5)^{10} = \frac{1024}{10^{10}} \leq 0.000001.$$

In general, if we repeat R times, we get the probability of error is at most

$$(1/5)^R.$$

3.2 Why did Alice not just send x over to Bob?

Naïvely, Alice could have sent x over to Bob. That would take N bits. Now she sends the prime p, and $x \mod p$. That's two numbers at most $M = 10N \lg 5N$. The number of bits required for that: $2 \log M = 2 \log(10N \lg 5N)) = O(\log N)$.

To put this in perspective, suppose x and y were two copies of all of Wikipedia. Say that's about 25 billion characters. Say 8 bits per character, so $N \approx 2 \cdot 10^{11} \approx 2^{38}$ bits. Whereas our approach, even with repeating things 10 times, sent over $10 \cdot 2 \log(10N \lg 5N)) \leq 1280$ bits. That's a lot less communication!

[3] This unique prime-factorization theorem is known as the fundamental theorem of arithmetic.

4 The Karp-Rabin Algorithm (a.k.a. the "Fingerprint" Method)

Let's use this idea to solve a different problem. In the *string matching* problem, we are given

- A *text* T, of length m.
- A *pattern* P, of length n.

The goal is to output all the occurrences of the pattern P inside the text T. E.g., if $T =$ `abracadabra` and $P =$ `ab` then the output should be $\{0, 7\}$.

There are many ways to solve this problem, but today we will use randomization to solve this problem. This solution is due to Karp and Rabin.[4] The idea is smart but simple, elegant and effective—like in many great algorithms.

4.1 The Karp-Rabin Idea

Think about the hash function $h_p(x) = x \mod p$, for $x \in \{0,1\}^n$. Now look at the string x' obtained by dropping the leftmost bit of x, and adding a bit to the right end. E.g., if $x =$ `0011001` then x' might be `0110010` or `0110011`. If I told you $h_p(x) = z$, can you compute $h_p(x')$ fast?

Suppose x'_{lb} is the lowest-order bit of x', and x_{hb} be the highest order bit of x. Then

$$\text{value of } x' = 2(\text{value of } x - x_{hb} \cdot 2^{n-1}) + x'_{lb}$$

Remember that $h_p(a+b) = (h_p(a) + h_p(b)) \mod p$, and $h_p(2a) = 2h_p(a) \mod p$. So

$$h_p(x') = (2h_p(x) - x_{hb} \cdot h_p(2^n) + x'_{lb}) \mod p.$$

That's two function computations to compute $h_p(x)$ and $h_p(2^n)$, three arithmetic operations modulo p, and one more residue modulo p.

4.2 How to use this idea for String Matching

To keep things short, let $T_{a\ldots b}$ denote the string from the a^{th} to the b^{th} positions of T, inclusive. So the string matching problem is: output all the locations $a \in \{0, 1, \ldots, m-n\}$ such that

$$T_{a\ldots a+(n-1)} = P.$$

Here's the algorithm:

1. Pick a random prime p in the range $\{1, \ldots, M\}$ for a $M = \lceil 2sn \lg(sn) \rceil$ for a value of s we'll choose later.
2. Compute $h_p(P)$ and $h_p(2^n)$, and store these results.
3. Compute $h_p(T_{0\ldots n-1})$, and check if it equals $h_p(P)$. If so, output `match at location 0`.

[4] Again, familiar names. Dick Karp is a professor of computer science at Berkeley, and won the Turing award in 1985. Among other things, he developed two of the max-flow algorithms you saw, and his 1972 paper showed that many natural algorithmic problems were NP complete. Michael Rabin is professor at Harvard; he won the Turing award in 1976 (jointly with CMU's Dana Scott). You may know him from the popular Miller-Rabin randomized primality test (the Miller there is our own Gary Miller); he's responsible for many algorithms in cryptography.

4. For each $i \in \{0, \ldots, m-n\}$, compute $h_p(T_{i+1\ldots i+n})$ using $h_p(T_{i\ldots i+n-1})$.
   If $h_p(T_{i+1\ldots i+n}) = h_p(P)$, output `match at location` $i+1$.

Clearly, we'll never have a false negative (i.e., miss a match) but we may erroneously output location that are not matches (have a false positive). Let's analyze the error probability, and the runtime.

4.2.1 Probability of Error

We do m different comparisons, each has a probability $1/s$ of failure. So, by a union bound, the probability of having at least one false positive is m/s. Hence, setting $s = 100m$ will make sure we have a $\frac{1}{100}$ chance of even a single mistake.

This means we set $M = (200 \cdot mn)\lg(200 \cdot mn)$. Which requires $\leq \lg M + 1 = O(\log m + \log n)$ bits to store. And hence our prime p is also at most $O(\log m + \log n)$ bits.[5]

4.2.2 Running Time

Let's say we can do arithmetic and comparisons on $O(\log M)$-bit numbers in constant time. (See the footnote above about why this is reasonable.) And let's not worry about the time to pick a random prime for now.

- Computing $h_p(x)$ for n-bit x can be done in $O(n)$ time. So each of the hash function computations in Steps 2 and 3 take $O(n)$ time.
- Now, using the idea in Section 4.1, we can compute each subsequent hash value in $O(1)$ time! So iterating over all the values of i takes $O(m)$ time.

That's a total of $O(m+n)$ time! And you can't do much faster, since the input itself is $O(m+n)$ bits long.

4.3 Extensions and Connections

There are other (deterministic) fast ways of solving the string matching problem we mentioned above. See, e.g., the Knuth-Morris-Pratt algorithm, and suffix trees. (See our 15-451 notes from another semester on these two topics.) The advantage of the Karp-Rabin approach is not only the simplicity, but also the extendability. You can, e.g., solve the following 2-dimensional problem using the same idea.

> **2-dimensional pattern matching.** Given a $m_1 \times m_2$-bit rectangular text T, and a $n_1 \times n_2$-bit pattern P (where $n_i \leq m_i$), find all occurrences of P inside T. Show that you do this in $O(m_1 m_2)$ time, where we assume that you can do modular arithmetic with integers of value at most poly$(m_1 m_2)$ in constant time.

[5] If we you do the math, and say $m, n \geq 10$, then $\lg M \leq 4(\lg m + \lg n)$. Now, just for perspective, if we were looking for a $n = 1024$-bit phrase in Wikipedia, this means the prime p is only $4(\lg 2^{38} + \lg 2^{10}) \leq 192$ bits long.

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: Independent Sets on Trees.
- (Optional) Example: Matrix-chain multiplication.

1 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.

2 Example 1: Longest Common Subsequence

Definition 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 necessarily in a contiguous block) in both strings.*

For example, consider:

$S =$ `ABAZDC`

$T =$ `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 the *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:

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

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 left contains a value equal to the value in the current cell, then move to that cell (if both do, 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 corresponts 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`.

3 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 (**)
  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.

[1] This is the number of different "monotone walks" between the upper-left and lower-right corners of an n by m grid. For this course you don't need to know this bound or how to prove it—it just suggests that brute-force is not the way to go. But if you are interested, look up Catalan numbers.

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.

4 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-per-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]

> **Exercise:** Give an example where using the greedy strategy will get you less than 1% of the optimal value (in the case there is no partial credit).

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

Definition 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)};
```

[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!

```
  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 nth item so we just recursively work backwards from `arr[n-1][S]`. Otherwise, we *did* use that item, so we just output the nth item and recursively work backwards from `arr[n-1][S-s_n]`. One can also do bottom-up Dynamic Programming.

> **Exercise:** The *fractional* knapsack problem is the one where you can add $\delta_i \in [0,1]$ fraction of task i to the knapsack, using $\delta_i s_i$ space and getting $\delta_i v_i$ value. The greedy algorithm adds items in decreasing order of v_i/s_i. Prove that this greedy algorithm produces the optimal solution for the *fractional* problem.

5 Example #3: Max-Weight Independent Sets on Trees

Given a graph G with vertices V and edges E, an *independent set* is a subset of vertices $S \subseteq V$ such that none of the edges have both of their endpoints in S. If each vertex v has a non-negative weight w_v, the goal of the *Max-Weight Independent Set* (MWIS) problem is to find an independent set with the maximum weight. We now give a Dynamic Programming solution for the case when the graph is a tree. Let us assume that the tree is rooted at some node v, which defines a notion of parents/children (and ancestors/descendents) for the tree.

Again, let us write down a naive resursive program to solve this problem. For a vertex v in the tree, let `MWIS(v)` be the weight of the maximum weight independent set in the subtree rooted at v. The naive recursive idea is simple: for each node v, either it is in the MWIS (in which case its children cannot be in the MWIS, and we look for the MWIS in its grandchildren. Or else v is not in the MWIS, and we look for the MWIS of its children.

```
MWIS(v) {
  if (v is a leaf) then result = w_v;
  else
    children_answer = sum_{u child of v} MWIS(u);
    grandchildren_answer = sum_{u grandchild of v} MWIS(u);
```

```
                // If no grandchildren, empty sum has value zero.
    result = max( w_v + grandchildren_answer, children_answer );
  return result;
}
```

Again, this is exponential-time in the worst-case. But fixing this using memoization is not difficult.

```
MWIS(v) {
  if (v is a leaf) then result = w_v;
  else
    if (arr[v] != unknown) return arr[v];  // <- added array check
    children_answer = sum_{u child of v} MWIS(u);
    grandchildren_answer = sum_{u grandchild of v} MWIS(u);
                // If no grandchildren, empty sum has value zero.
    result = max( w_v + grandchildren_answer, children_answer );
    arr[v] = result;                        // <- and this
  return result;
}
```

What is the runtime of this algorithm? Again, calls to the algorithm with any given vertex v can pass the array check only once. However, each recursive call no longer takes constant time. Indeed, we have recursive calls to all v's children and grand-children. There are n vertices, and there could be at most n such children/grand-children, so that is $O(n^2)$ runtime.

Actually, we can do a better calculation. For each v, `arr[v]` is unknown only once, and that particular recursive call takes time $O(1+$ number of v's children and grand-children$)$—the $O(1)$ accounts for other things done within the call. (Even if v is a leaf, the time taken is $\Theta(1)$.) When we make a call to v and the `arr[v]` is known, we take only $O(1)$ time; this can be charged to the parent recursive call. Summing over all v, the total time taken is

$$\sum_v O\Big(1 + \text{number of children of } v + \text{number of grand-children of } v\Big).$$

But this is $O(n)$, because each node u is counted at most twice in the sum, once for its parent and once for its grand-parent (if any).

> **Exercise:** We just showed how to find the weight of the MWIS. Show how to find the actual independent set as well, in $O(n)$ time.

> **Exercise:** Suppose you are given a tree T with vertex-weights w_v and also an integer $K \geq 1$. You want to find, over all independent sets of cardinality K, the one with maximum weight. (Or say "there exists no independent set of cardinality K".) Give a dynamic programming solution to this problem.

6 (Optional) Example #4: 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 ℓxm matrix by an mxn 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 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.[3] 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 ℓmn 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:

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

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.[4]

[3] For the purposes of this course, you need not know how to derive this bound — it's only to suggest the futility of brute-force. But if you are interested, refer to 15-251 notes on counting, or to Catalan numbers.

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

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 particularly 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.

7.1 Why "Dynamic Programming"?

The term dynamic programming comes from Richard Bellman, who also gave us the Bellman-Ford algorithm from the next lecture. In his autobiography he says

> *"Let's take a word that has an absolutely precise meaning, namely dynamic, in the classical physical sense. It also has a very interesting property as an adjective, and that is it's impossible to use the word, dynamic, in a pejorative sense. Try thinking of some combination that will possibly give it a pejorative meaning. [...] Thus, I thought dynamic programming was a good name. It was something not even a Congressman could object to. So I used it as an umbrella for my activities."*

Bellman clearly had a way with words, coining another super-well-known phrase, "*the curse of dimensionality*".

In this lecture we continue our discussion of dynamic programming, focusing on using it for a variety of path-finding problems in graphs. Topics in this lecture include:

- The Bellman-Ford algorithm for single-source (or single-sink) shortest paths.
- Matrix-product algorithms for all-pairs shortest paths.
- The Floyd-Warshall algorithm for all-pairs shortest paths.
- Dynamic programming for TSP.

1 Introduction

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 (and as we said earlier, we will sometimes call the edges in a directed graph *arcs*). 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 u to v and an edge from v to u.

We will be especially interested here in *weighted* graphs where edges have weights (which we will also call *costs* or *lengths*). For an edge (u, v) in our graph, let's use $len(u, v)$ to denote its weight. The basic shortest-path problem is as follows:

Definition 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 in our drawings, if there is an edge of length ℓ from u to v and also an edge of length ℓ from v to u, we will often just draw them together as a single undirected edge. So, all such edges must have positive weight.

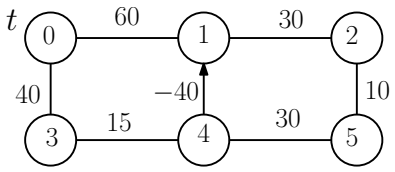
2 Dijkstra's Algorithm

Let's recall Dijkstra's algorithm: given a directed graph $G = (V, E)$ with positive edge weights, the single-source shortest path problem can be solved using Dijkstra's algorithm (you've seen this in 15-210) in time $O(m \log n)$. This uses a standard heap data-structure; using a fancier data structure called Fibonacci heaps, one can implement Dijkstra's algorithm in $O(m + n \log n)$ time. This will be a benchmark to keep in mind.

3 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. We will assume the graph is represented as an *adjacency list*: an array of size n where entry v is the list of arcs exiting from node v (so, given a node, we can find all its neighbors in time proportional to the number of neighbors).

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 ∞ 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 ∞.

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)∈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 $O(mn)$.

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.

4 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 $O(n^3)$.

4.1 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 ($i \neq j$ and there is no edge from i to j), $A[i, j] = \infty$.

I.e., $A[i, j]$ is the length of the shortest path from i to j using 1 or fewer edges.[1] 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 $O(n^3 \log n)$.

4.2 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:
```

[1] There are multiple ways to define an adjacency matrix for weighted graphs — e.g., what $A[i, i]$ should be and what $A[i, j]$ should be if there is no edge from i to j. The right definition will typically depend on the problem you want to solve.

```
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!

5 TSP

The NP-hard *Traveling Salesman Problem* (TSP) asks to find the shortest route that visits *all* vertices in the graph. To be precise, the TSP is the shortest tour that visits all vertices and returns back to the start.[2] Since the problem is NP-hard, we don't expect that Dynamic Programming will give us a polynomial-time algorithm, but perhaps it can still help.

Specifically, the naive algorithm for the TSP is just to run brute-force over all $n!$ permutations of the n vertices and to compute the cost of each, choosing the shortest. (We can reduce this to $(n-1)!$ permutations by always using the same start vertex, but we still pay $\Theta(n)$ to compute the cost of each permutation, so the overall running time is $O(n!)$.) We're going to use Dynamic Programming to reduce this to "only" $O(n^2 2^n)$.

Any ideas? As usual, let's first just worry about computing the *cost* of the optimal solution, and then we'll later be able to add in some hooks to recover the path. Also, let's work with the shortest-path metric where we've already computed all-pairs-shortest paths (so we can view our graph as a complete graph with weights between any two vertices representing the shortest path between them). This is conveninent since it means a solution is really just a permutation. Finally, let's fix some start vertex s.

Now, here is one fact we can use. Suppose someone told you what the initial part of the solution should look like and we want to use this to figure out the rest. Then really all we need to know about it for the purpose of completing it into a tour is the *set* of vertices visited in this initial segment and the *last* vertex t visited in the set. We don't really need the whole ordering of the initial segment. This means there are "only" $n2^n$ subproblems (one for every set of vertices and ending vertex t in the set). Furthermore, we can compute the optimal solution to a subproblem in time $O(n)$ given solutions to smaller subproblems (just look at all possible vertices t' in the set we could have been at right before going to t and take the one that minimizes the cost so far (stored in our lookup table) plus the distance from t' to t).

Here is a top-down way of thinking about it: if we were writing a recursive piece of code to solve this problem, then we would have a bit-vector saying which vertices have been visited so far and a variable saying where we are now, and then we would mark the current location as visited and recursively call on all possible (i.e., not yet visited) next places we might go to. Naively this would take time $\Omega(n!)$. However, by storing the results of our computations we can use the fact that there are "only" $n2^n$ possible (bit-vector, current-location) pairs, and so only that many *different* recursive calls made. For each recursive call we do $O(n)$ work inside the call, for a total of $O(n^2 2^n)$ time.[3]

The last thing is we just need to recompute the paths, but this is easy to do from the computations stored in the same way as we did for shortest paths.

[2]Note that under this definition, it doesn't matter which vertex we select as the start. The *Traveling Salesman Path Problem* is the same thing but does not require returning to the start. Both problems are NP-hard.

[3]For more, see `http://xkcd.com/399/`.

In today's lecture, we'll talk about game theory and some of its connections to computer science. The topics we'll cover are:

- 2-player Zero-sum games, and the concept of (minimax) optimal strategies.
- The connection of 2-player zero-sum games to randomized algorithms.
- And time permitting, general sum-games, and the idea of a Nash equilibrium.

1 Introduction to Game Theory

Game theory is the study of how people behave in social and economic interactions, and how they make decisions in these settings. It is an area originally developed by economists, but given its general scope, it has applications to many other disciplines, including computer science.

A clarification at the very beginning: a *game* in game theory is not just what we traditionally think of as a game (chess, checkers, poker, tennis, or football), but is much more inclusive — a game is any interaction between parties, each with their own interests. And game theory studies how these parties make decisions during such interactions.[1]

Since we very often build large systems in computer science, which are used by multiple users, whose actions affect the performance of all the others, it is natural that game theory would play an important role in many CS problems. For example, game theory can be used to model routing in large networks, or the behavior of people on social networks, or auctions on Ebay, and then to make qualitative/quantitative predictions about how people would behave in these settings.

In fact, the two areas (game theory and computer science) have become increasingly closer to each other over the past two decades — the interaction being a two-way street — with game-theorists proving results of algorithmic interest, and computer scientists proving results of interest to game theory itself.

2 Some Definitions and Examples

In a game, we have

- A collection of participants, often called *players*.
- Each player has a set of choices, called *actions*, from which players choose about how to play (i.e., behave) in this game.
- Their combined behavior leads to *payoffs* (think of this as the "happiness" or "satisfaction level") for each of the players.

Let us look at some examples: all of these basic examples have only two players which will be easy to picture and reason about.

2.1 The Shooter-Goalie Game

This game abstracts what happens in a game of soccer, when some team has a penalty shot. There are two players in this game. One called the *shooter*, the other is called the *goalie*. Hence this is a

[1] Robert Aumann, Nobel prize winner, has suggested the term "interactive decision theory" instead of "game theory".

2-player game.

The shooter has two choices: either to shoot to her left, or shoot to her right. The goalie has two choices as well: either to dive to the shooter's left, or to the shooter's right. Hence, in this case, both the players have two actions, denoted by the set $\{\mathsf{L}, \mathsf{R}\}$.[2]

Now for the "payoffs". If both the shooter and the goalie choose the same strategy (say both choose L, or both choose R) then the goalie makes a save. Note this is an abstraction of the game: for now we assume that the goalie always makes the save when diving in the correct direction. This brings great satisfaction for the goalie, not so much for the shooter. On the other hand, if they choose different strategies, then the shooter scores the goal. (Again, we are modeling a perfect shooter.) This brings much happiness for the shooter, the goalie is disappointed.

Being mathematically-minded, suppose we say that the former two choices lead to a payoff of $+1$ for the goalie, and -1 for the shooter. And the latter two choices lead to a payoff of -1 for the goalie, and $+1$ for the shooter. We can write it in a matrix (called the *payoff matrix*) thus:

payoff matrix M	goalie L	goalie R
shooter L	$(-1, 1)$	$(1, -1)$
shooter R	$(1, -1)$	$(-1, 1)$

The rows of the game matrix are labeled with actions for one of the players (in this case the shooter), and the columns with the actions for the other player (in this case the goalie). The entries are pairs of numbers, indicating who wins how much: e.g., the L, L entry contains $(-1, 1)$, the first entry is the payoff to the row player (shooter), the second entry the payoff to the column player (goalie). In general, the payoff is (r, c) where r is the payoff to the row player, and c the payoff to the column player.

In this case, note that for each entry (r, c) in this matrix, the sum $r + c = 0$. Such a game is called a **zero-sum game**. The zero-sum-ness captures the fact that the game is "purely competitive". Note that being zero-sum does not mean that the game is "fair" in any sense—a game where the payoff matrix has $(1, -1)$ in all entries is also zero-sum, but is clearly unfair to the column player.

One more comment: for 2-player games, textbooks often define the *row-payoff matrix* R which consists of the payoffs to the row player, and the *column-payoff matrix* C consisting of the payoffs to the column player. The tuples in the payoff matrix M contain the same information, i.e.,

$$M_{ij} = (R_{ij}, C_{ij}).$$

The game being zero-sum now means that $R = -C$, or $R + C = 0$. In the example above, the matrix R is

payoff matrix M	goalie L	goalie R
shooter L	-1	1
shooter R	1	-1

[2]Note carefully: we have defined things so that left and right are with respect to the shooter. From now on, when we say the goalie dives left, it should be clear that the goalie is diving to *the shooter's left.*

2.2 Pure and Mixed Strategies

Now given a game with payoff matrix M, the two players have to choose strategies, i.e., decide how to play.

One strategy would be for the row player to decide on a row to play, and the column player to decide on a column to play. Say the strategy for the row player was to play row/ I and the column player's strategy was to play column J, then the payoffs would be given by the tuple (R_{IJ}, C_{IJ}) in location I, J:

$$\text{payoff } R_{IJ} \text{ to the row player, and } C_{IJ} \text{ to the column player}$$

In this case both players are playing deterministically. (E.g., the goalie decides to always go left, etc.) A strategy that decides to play a single action is called a *pure strategy*.

But very often pure strategies are not what we play. We are trying to compete with the worst adversary, and we may like to "hedge our bets". Hence we may use a randomized algorithm: e.g., the players dive/shoot left or right with some probability, or when playing the classic game of Rock-Paper-Scissors the player choose one of the options with some probability. This means the row player decides on a non-negative real p_i for each row, such that $\sum_i p_i = 1$ (this gives a probability distribution over the rows). Similarly, the column player decides on a $q_i > 0$ for each column, such that $\sum_i q_i = 1$. The probability distributions $\mathbf{p}, \mathbf{q}$ are called the *(mixed-strategy) mixed strategies* for the row (mixed-strategy) and column player, respectively. And then we look at the *expected payoff*

$$V_R(\mathbf{p}, \mathbf{q}) := \sum_{ij} \Pr[\text{row player plays } i, \text{column player plays } j] \cdot R_{ij} = \sum_{ij} p_i q_j R_{ij}$$

for the row player (where we used that the row and column player have independent randomness), and

$$V_C(\mathbf{p}, \mathbf{q}) := \sum_{ij} p_i q_j C_{ij}$$

for the column player. This being a two-player zero-sum game, we know that $V_R(\mathbf{p}, \mathbf{q}) = -V_C(\mathbf{p}, \mathbf{q})$, so we will just mention the payoff to one of the players (say the row player).

For instance, if $\mathbf{p} = (0.5, 0.5)$ and $\mathbf{q} = (0.5, 0.5)$ in the shooter-goalie game, then $V_R = 0$, whereas $\mathbf{p} = (0.75, 0.25)$ and $\mathbf{q} = (0.6, 0.4)$ gives $V_R = 0.45 - 0.55 = -0.1$.

2.3 Minimax-Optimal Strategies

What does the row player want to do? She wants to find a vector $\mathbf{p}^\star$ that maximizes the expected payoff to her, over all choices of the opponent's strategy $\mathbf{q}$. The mixed strategy that maximizes the minimum payoff. I.e., the row player wants to find

$$\mathsf{lb} := \max_{\mathbf{p}} \min_{\mathbf{q}} V_R(\mathbf{p}, \mathbf{q})$$

Make sure you parse this correctly:

$$\mathsf{lb} := \overbrace{\max_{\mathbf{p}} \underbrace{\min_{\mathbf{q}} V_R(\mathbf{p}, \mathbf{q})}_{\text{payoff when opponent plays her optimal strategy against our choice } \mathbf{p}}}^{\text{mixed strategy that maximizes the minimum expected payoff}}$$

Loosely, *this much payoff the row player can guarantee to herself no matter what the column player does.* The quantity lb is a **lower bound on the row-player's payoff**.

What about the column player? She wants to find some $\mathbf{q}^\star$ that maximizes her own expected payoff, over all choices of the opponent's strategy $\mathbf{p}$. She wants to optimize

$$\max_{\mathbf{q}} \min_{\mathbf{p}} V_C(\mathbf{p}, \mathbf{q})$$

But this is a zero-sum game, so this is the same as

$$\max_{\mathbf{q}} \min_{\mathbf{p}} (-V_R(\mathbf{p}, \mathbf{q}))$$

And pushing the negative sign through, we get the column player is trying to optimize her own worst-case payoff, which is

$$-\min_{\mathbf{q}} \max_{\mathbf{p}} V_R(\mathbf{p}, \mathbf{q})$$

So the payoff in this case to the row player is

$$\mathsf{ub} := \min_{\mathbf{q}} \max_{\mathbf{p}} V_R(\mathbf{p}, \mathbf{q})$$

The column player can guarantee that the row player does not get more than this much payoff, no matter what the row-player does. This is an **upper bound on the row player's payoff**.

We have two quantities: lb and ub. How do they compare? To figure this out, we first make a simple but important observation: suppose we want to find the row player's minimax-optimal strategy $\mathbf{p}^\star$. Then we can assume that the column player plays a pure strategy (a single column). Why? Once the row player fixes a mixed strategy $\mathbf{p}$, the column player then has no reason to randomize: her payoffs will be some average of the payoffs from playing the individual columns, so she can just pick the best column for her. In other words, we get that the quantity ub can be equivalently defined as

$$\mathsf{lb} = \max_{\mathbf{p}} \min_{j} \sum_{i} p_i R_{ij}.$$

Similarly, the column player wants to optimize

$$\mathsf{ub} = \min_{\mathbf{q}} \max_{i} \sum_{j} q_j R_{ij} = -\max_{\mathbf{q}} \min_{i} \sum_{j} q_j C_{ij}.$$

2.3.1 The Balanced Game Example

For the shooter-goalie game, we claim that the minimax-optimal strategies for both players is $(0.5, 0.5)$. How to calculate this?

> **Row Player:** For the row player (shooter), suppose $\mathbf{p} = (p_1, p_2)$ is the mixed strategy. Note that $p_1 \geq 0, p_2 \geq 0$ and $p_1 + p_2 = 1$. So it is easier to write the strategy as $\mathbf{p} = (p, 1-p)$ with $p \in [0, 1]$.
>
> OK. If the column player (goalie) plays L, then this strategy gets the shooter a payoff of
>
> $$p \cdot (-1) + (1-p) \cdot (1) = 1 - 2p.$$
>
> If the column player (goalie) plays R, then this strategy gets the shooter a payoff of
>
> $$p \cdot (1) + (1-p) \cdot (-1) = 2p - 1.$$

So we want to choose some value $p \in [0, 1]$ to maximize

$$\mathsf{lb} = \min(1 - 2p, 2p - 1)$$

In this case, this maximum is achieved at $p = 1/2$. (One way to see it is by drawing these two lines.) And the minimax-optimal expected payoff to the shooter is 0.

Column Player: The calculation for the column player (goalie) is very similar in this case. The minimax-optimal strategy for the goalie is also $(0.5, 0.5)$ and the guarantees that the shooter cannot make more than 0 payoff.

An observation: the shooter can guarantee a payoff of $\mathsf{lb} = 0$, and the goalie can guarantee that the shooter's payoff is never more than $\mathsf{ub} = 0$. Since $\mathsf{lb} = \mathsf{ub}$, in this case the *"value of the game"* is said to be $\mathsf{lb} = \mathsf{ub} = 0$.

2.3.2 The Asymmetric Goalie Example

Let's change the game slightly. Suppose the goalie is weaker on the left. What happens if the payoff matrix is now:

	L	R
shooter L	$(-\frac{1}{2}, \frac{1}{2})$	$(1, -1)$
R	$(1, -1)$	$(-1, 1)$

Row Player: For the row player (shooter), suppose $\mathbf{p} = (p, 1-p)$ is the mixed strategy, with $p \in [0, 1]$. If the column player (goalie) plays L, then this strategy gets the shooter a payoff of

$$p \cdot (-1/2) + (1 - p) \cdot (1) = 1 - (3/2)p.$$

If the column player (goalie) plays R, then this strategy gets the shooter a payoff of

$$p \cdot (1) + (1 - p) \cdot (-1) = 2p - 1.$$

So we want to choose some value $p \in [0, 1]$ to maximize

$$\mathsf{lb} = \min(1 - (3/2)p, 2p - 1)$$

In this case, this maximum is achieved at $p = 4/7$. And the minimax-optimal expected payoff to the shooter is $1/7$. Note that the goalie being weaker means the shooter's payoff increases.

Column Player: What about the calculation for the column player (goalie)? If her strategy is $\mathbf{q} = (q, 1 - q)$ with $q \in [0, 1]$, then if the shooter plays L then the shooter's payoff is

$$q \cdot (-1/2) + (1 - q) \cdot (1) = 1 - (3/2)q.$$

If she plays R, then it is $2q - 1$. So the goalie will try to minimize

$$\mathsf{ub} = \max(1 - (3/2)q, 2q - 1)$$

which will again give $(4/7, 3/7)$ and guarantees that the expected loss is never more than $1/7$.

Again, the shooter guarantees a payoff of $\mathsf{lb} = 1/7$, and the goalie can guarantee that the shooter's payoff is never more than $\mathsf{ub} = 1/7$. In this case the value of the game is said to be $\mathsf{lb} = \mathsf{ub} = 1/7$.

Exercise 1: What if both players have somewhat different weaknesses? What if the payoffs are:

```
(-1/2, 1/2) (3/4, -3/4)
(1, -1)     (-3/2, 3/2)
```

Show that minimax-optimal strategies are $\mathbf{p} = (2/3, 1/3), \mathbf{q} = (3/5, 2/5)$ and value of game is 0.

Exercise 2: For the game with payoffs:

```
(-1/2, 1/2) (3/4, -3/4)
(1, -1)     (-2/3, 2/3)
```

Show that minimax-optimal strategies are $\mathbf{p} = (\frac{4}{7}, \frac{3}{7}), \mathbf{q} = (\frac{17}{35}, \frac{18}{35})$ and value of game is $\frac{1}{7}$.

Exercise 3: For the game with payoffs:

```
(-1/2, 1/2)    (-1, 1)
(1, -1)        (2/3, -2/3)
```

Show that minimax-optimal strategies are $\mathbf{p} = (0, 1), \mathbf{q} = (0, 1)$ and value of game is $\frac{2}{3}$.

3 Von Neumann's Minimax Theorem

In all the above examples of 2-player zero-sum games, we saw that the row player has a strategy $\mathbf{p}^\star$ that guarantees some payoff lb for her, no matter what strategy $\mathbf{q}$ the column player plays. And the column player has a strategy $\mathbf{q}^\star$ that guarantees that the row player cannot get payoff more than ub, no matter what strategy $\mathbf{p}$ the row player plays. The remarkable fact in the examples was that $\mathsf{lb} = \mathsf{ub}$ in all these cases! Was this just a coincidence? No: a celebrated result of von Neumann[3] shows that we always have $\mathsf{lb} = \mathsf{ub}$ in (finite) 2-player zero-sum games.

Theorem 1 (Minimax Theorem (von Neumann, 1928)) *Given a finite 2-player zero-sum game with payoff matrices $R = -C$,*

$$\mathsf{lb} = \max_{\mathbf{p}} \min_{\mathbf{q}} V_R(\mathbf{p}, \mathbf{q}) = \min_{\mathbf{q}} \max_{\mathbf{p}} V_R(\mathbf{p}, \mathbf{q}) = \mathsf{ub}.$$

This common value is called the value of the game.

The theorem implies that in a zero-sum game, both the row and column players can even "publish" their minimax-optimal mixed strategies (i.e., tell the strategy to the other player), and it does not hurt their expected performance as long as they play optimally.[4]

Von Neumann's Minimax Theorem is an important result in game theory, but it has beautiful implications to computer science as well — as we see in the next section.

4 Lower Bounds for Randomized Algorithms

In order to prove lower bounds, we thought of us coming up with algorithms, and the adversary coming with some inputs on which our algorithm would perform poorly—take a long time, or make many comparisons, etc. We can encode this as a zero-sum game with row-player payoff matrix R. (Keep sorting as an example application in your mind.)

- The columns are various algorithms for the problem (for sorting n elements).

[3] John von Neumann, mathematician, physicist, and polymath.

[4] It's like telling your rock-paper-scissors opponent that you will play each action with equal probability, it does not buy them anything to know your strategy. This is not true in general non-zero-sum games; there if you tell your opponent the mixed-strategy you're playing, she may be able to do better. Note carefully that you are not telling them the actual random choice you will make, just the distribution from which you will choose.

- The rows are all the possible inputs (all $n!$ of them).
- The entry R_{ij} is the cost of the algorithm j on the input i (say the number of comparisons).

This may be a huge matrix, but we'll never write it down. It's just a conceptual guide. But what does the matrix tell us? A lot, as it turns out:

- A deterministic algorithm with good worst-case guarantee is a column that does well against all rows: all entries in this column are small.
- A randomized algorithm with good expected guarantee is a probability distribution $\mathbf{q}$ over columns, such that the expected cost for each row i is small. This is a mixed strategy for the column player. It gives an upper bound.
- Ideally we want to find the minimax-optimal distribution $\mathbf{q}^\star$ achieving the value of this game. This would be the best randomized algorithm.
- What is a lower bound for randomized algorithms? It is a mixed-strategy over rows (a probability distribution $\mathbf{p}$ over the inputs) such that for every column (i.e., deterministic algorithm j), the expected cost of j (under distribution $\mathbf{p}$) is high.

  So to prove a lower bound for randomized algorithms, if suffices to show that lb is high for this game. I.e., give a strategy for the row player (a distribution over inputs) such that every column (deterministic algorithm) incurs a high cost on it.

4.1 A Lower Bound for Sorting Algorithms

Recall from Lecture #2 we showed that any deterministic comparison-based sorting algorithm must perform $\log_2 n! = n \lg n - O(n)$ comparisons in the worst case. The next theorem extends this result to randomized algorithms.

Theorem 2 *Let $\mathcal{A}$ be any randomized comparison-based sorting algorithm (that always outputs the correct answer). Then there exist inputs on which $\mathcal{A}$ performs $\Omega(lgn!)$ comparisons in expectation.*

Proof: Suppose we construct a matrix R as above, where the columns are possible (deterministic) sorting algorithms for n elements, the rows are the $n!$ possible inputs, and entry R_{ij} is the number of comparisons algorithm j makes on input i. We claim that the value of this game is $\Omega(\lg n!)$: this implies that the best distribution over columns (i.e., the best randomized algorithm) must suffer at least this much cost on some column (i.e., input).

To show the value of the game is large, we show a probability distribution over the rows (i.e., inputs) such that the expected cost of every column (i.e., every deterministic algorithm) is $\Omega(\lg n!)$.

This probability distribution is the uniform distribution: each of the $n!$ inputs is equally likely. Now consider any deterministic algorithm: as in Lecture #2, this is a decision tree with at least $n!$ leaves. No two inputs go to the same leaf.

In this tree, how many leaves can have depth at most $(\lg n!) - 10$? At most the number of nodes at depth at most $(\lg n!) - 10$ in a binary tree. Which in turn is

$$1 + 2 + 4 + 8 + \ldots + 2^{(\lg n!)-10} \leq 1 + 2 + 4 + 8 + \ldots + \tfrac{n!}{1024} \leq \tfrac{n!}{512}$$

So $\frac{511}{512} \geq 0.99$ fraction of the leaves in this tree have depth more than $(\lg n!) - 10$. In other words, if we pick a random input, it will lead to a leaf at depth more than $(\lg n!) - 10$ with probability 0.99. Which gives the expected depth of a random input to be $0.99((\lg n!) - 10) = \Omega(\lg n!)$. ■

5 General-Sum Two-Player Games*

In general-sum games, we don't deal with purely competitive situations, but cases where there are win-win and lose-lose situations as well. For instance, the coordination game of "chicken", a.k.a. *what side of the street to drive on?* It has the payoff matrix:

payoff matrix M	Bob L	Bob R
Alice L	$(1,1)$	$(-1,-1)$
Alice R	$(-1,-1)$	$(1,1)$

Note that we are now using the convention that a player choosing L is driving on *their* left. Note that if both players choose the same side, then both win. And if they choose opposite sides, both crash and lose. (Both players can choose to drive on the left—like Britain, India, etc.—or both on the right, like the rest of the world, but they must coordinate. Both these are stable solutions and give a payoff of 1 to both parties.)

Consider another coordination game that we call "which movie?" Two friends are deciding what to do in the evening. One wants to see *Citizen Kane*, and the other *Dumb and Dumber*. They'd rather go to a movie together than separately (so the strategy profiles C, D and D, C have payoffs zero to both), but C, C has payoffs $(8, 2)$ and D, D has payoffs $(2, 8)$.

payoff matrix M	Bob C	Bob D
Alice C	$(8,2)$	$(0,0)$
Alice D	$(0,0)$	$(2,8)$

Finally, yet another game is "Prisoner's Dilemma" (or "to pollute or not?") with the payoff matrix:

payoff matrix M	Bob collude	Bob defect
Alice collude	$(2,2)$	$(-1,3)$
Alice defect	$(3,-1)$	$(0,0)$

5.1 Nash Equilibria

In this case, a good notion is to look for a *Nash Equilibrium*[5] which is a stable set of (mixed) strategies for the players. Stable here means that given strategies $(\mathbf{p}, \mathbf{q})$, neither player has any incentive to unilaterally switch to a different strategy. I.e., for any other mixed strategy $\mathbf{p}'$ for the row player

$$\text{row player's new payoff } = \sum_{ij} p'_i q_j R_{ij} \leq \sum_{ij} p_i q_j R_{ij} = \text{ row player's old payoff}$$

[5]Named after John Nash: mathematician and Nobel prize winner.

and for any other possible mixed strategy $\mathbf{q}'$ for the column player

$$\text{column player's new payoff } = \sum_{ij} p_i q'_j C_{ij} \leq \sum_{ij} p_i q_j C_{ij} = \text{ column player's old payoff.}$$

Here are some examples of Nash equilibria:

- In the chicken game, both $\{\mathbf{p} = (1,0), \mathbf{q} = (1,0)\}$ and $\{\mathbf{p} = (0,1), \mathbf{q} = (0,1)\}$ are Nash equilibria, as is $\{\mathbf{p} = (\frac{1}{2}, \frac{1}{2}), \mathbf{q} = (\frac{1}{2}, \frac{1}{2})\}$.
- In the movie game, the only Nash equilibria are $\{\mathbf{p} = (1,0), \mathbf{q} = (1,0)\}$ and $\{\mathbf{p} = (0,1), \mathbf{q} = (0,1)\}$.
- In prisoner's dilemma, the only Nash equilibrium is to defect (or pollute). So we need extra incentives for overall good behavior.

It is easy to come up with games where there are no stable *pure* strategies—this is even true for zero-sum games. But what about mixed-strategies? The main result in this area was proved by Nash in 1950 (which led to his name being attached to this concept)

Theorem 3 (Existence of Stable Strategies) *Every finite player game (with each player having a finite number of strategies) has at least one (mixed-strategy) Nash equilibrium.*

This theorem implies the Minimax Theorem (Theorem 1) as a corollary: indeed, take any two-player zero-sum game and consider any Nash equilibrium $(\mathbf{p}, \mathbf{q})$, with value $V = \sum_{ij} p_i q_j R_{ij} = -\sum_{ij} p_i q_j C_{ij}$. Since this is stable, neither player can do better by deviating, even knowing the other player's strategy. So they must be playing minimax-optimal.

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

1 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 1 below.

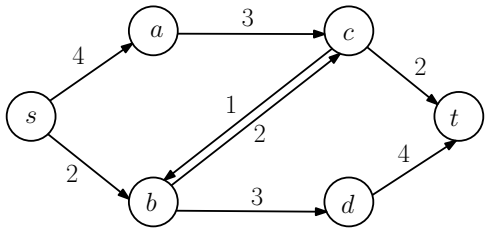


Figure 1: A network flow graph.

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 2. Note that the flow can split and rejoin itself.

How can you see that the above flow was really maximum? Notice, this flow saturates the $a \to c$ and $s \to b$ 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 sink from the source). 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.

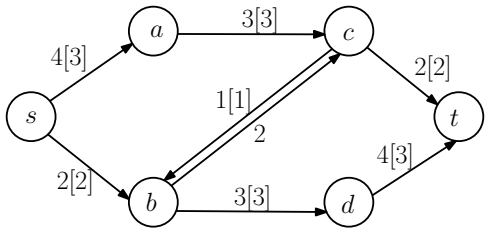


Figure 2: A network flow graph with positive flow shown using "capacity[flow]" notation.

We just argued that in general, the maximum s-t flow $\leq$ the capacity of the minimum s-t cut.[1] 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 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 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 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 a 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. In fact, let's now formally define the sum of two flows. If f and g are flows, then $h = f + g$ is defined as $h(u, v) = f(u, v) + g(u, v)$ for all pairs (u, v). Notice that if f and g satisfy flow-conservation and skew-symmetry, then h does too. In fact, using skew-symmetry, if we wanted we could rewrite the flow conservation condition as $\forall v \notin \{s, t\}, \sum_u f(u, v) = 0$, since the total flow *out* of a node will always be the negative of the total flow *into* a node.

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.

[1] This immediately implies the value of *any* s-t flow $\leq$ maximum s-t flow $\leq$ minimum s-t cut $\leq$ *any* s-t cut.

2 The Ford-Fulkerson algorithm

The Ford-Fulkerson algorithm[2] 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 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 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 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 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:

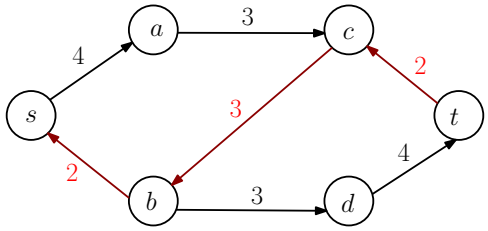


Figure 3: Residual graph resulting from pushing 2 units of flow along the path s-b-c-t in the graph in Figure 1. The red edges denote the changes.

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 4.

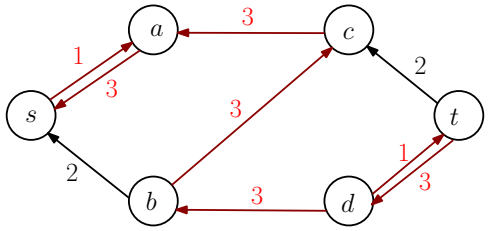


Figure 4: Residual graph resulting from pushing 3 units of flow along the path s-a-c-b-d-t in the graph in Figure 3. Again, the red edges denote the changes.

[2]The algorithm is due to Lester R. Ford Jr. and Delbert R. Fulkerson, then at the RAND corporation working on figuring out the capacity of the Soviet rail network. They didn't want to find the max-flow, they wanted the min-cut. As we will soon see, the two problems are inextricably entwined. They presented their algorithm in a RAND report of 1955.

At this point there is no 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, using the definition of adding flows we gave before. 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).

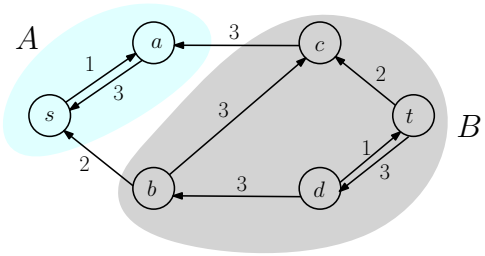
2.1 The Analysis

For now, let us assume that all the capacities are integers. If the maximum flow value 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 6 *If the given graph G has integer capacities, Ford-Fulkerson terminates in time $O(F(m+n))$ where F is the value of the maximum s-t flow.*

Theorem 7 *When it terminates, the Ford-Fulkerson algorithm outputs 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[3]; 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. ■

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

[3]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).

Theorem 8 *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.*

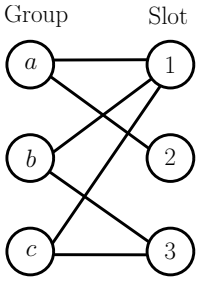
We have also proven the *integral-flow theorem*: if all capacities are integers, then there is a maximum flow in which all flows are integers. This seems obvious, but it turns out to have some nice and non-obvious implications.

> Technically, we just proved Theorem 8 only for integer capacities. What if the capacities are not integers? Firstly, if the capacities are rationals, then choose the smallest integer N such that $N \cdot c(u,v)$ is an integer for all edges (u,v). It is easy to see that at each step we send at least $1/N$ amount of flow, and hence the number of iterations is at most NF, where F is the value of the maximum s-t flow. (One can argue this by observing that scaling up all capacities by N will make all capacities integers, whence we can apply our above argument.) And hence we get Theorem 8 for rational capacities as well.
>
> What if the capacities are irrational? In this case Ford-Fulkerson may not terminate. And the solution it converges to (in the limit) may not even be the max-flow! (See here, here.) But the maxflow-mincut Theorem 8 still holds, even with irrational capacities. There are several ways to prove this; here's one. Suppose not, and suppose there is some flow network with the maxflow being $\epsilon > 0$ smaller than the mincut. Choose integer N such that $\frac{1}{N} \leq \frac{\epsilon}{2m}$, and round all capacities down to the nearest integer multiple of $1/N$. The mincut with these new edge capacities may have fallen by $m/N \leq \epsilon/2$, and the maxflow could be the same as the original flow network, but still there would be a gap of $\epsilon/2$ between maxflow and mincut in this rational-capacity network. But this is not possible, because used Ford-Fulkerson to prove maxflow-mincut for rational capacities in the previous paragraph.

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.

3 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.

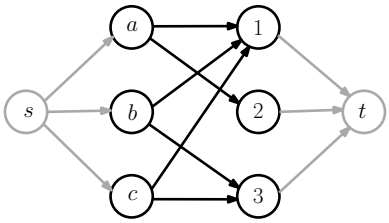
2. Find a max flow from s to t using Ford-Fulkerson.

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. We first build this flow network.



Then we use Ford-Fulkerson. Say we start by pushing flow on s-a-1-t and s-c-3-t, thereby matching a to 1 and c to 3. These are bad choices, since with these choices b cannot be matched. But the augmenting path s-b-1-a-2-t *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.

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

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 1.

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

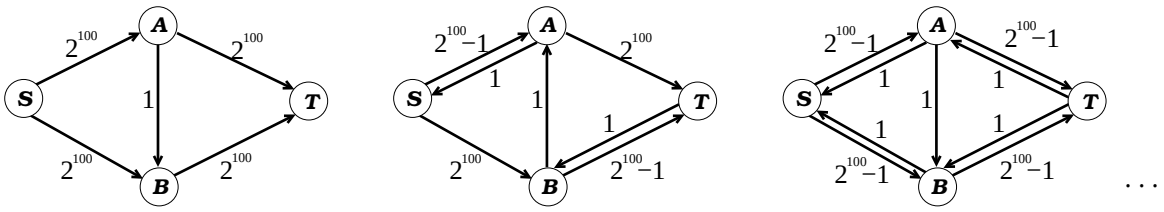


Figure 1: A bad case for Ford-Fulkerson. Starting with the graph on the left, we choose the path s-a-b-t, producing the residual graph shown in the middle. We then choose the path s-b-a-t, producing the residual graph on the right, and so on.

3 Edmonds-Karp #1

The first algorithm we study is due to Edmonds and Karp.[1] In fact, 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. (Such a path is called a "maximum bottleneck path." Dijkstra's algorithm can be trivially modified to find such paths. Instead of keeping tentative distance values in the priority queue, we keep maximum bottleneck values. The running time is the same as Dijkstra's.)

Claim 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 2 *Edmonds-Karp #1 makes at most $O(m \log F)$ iterations.*

Proof: By Claim 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 $O(m^2 \log F)$.[2]

[1] Jack Edmonds is another of the greats in algorithms—he did a lot of the pioneering work on flows and matchings, and is one of the first people to propose that efficient algorithms should (at least) run in polynomial-time, instead of just stopping "in finite time". We've already met Dick Karp when discussing Karp-Rabin fingerprinting.

[2] 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

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.

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 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.

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

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 $O(n^3)$.

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 2 still holds), but now it only takes us $O(m)$ time per iteration rather than $O(m \log n)$.

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 the level graph G_{level} which is a BFS layout from s containing only the forward edges, where we then remove any nodes that can't reach t using these edges (e.g., by doing a backwards BFS from t and removing unmarked nodes). Compute the capacities of all nodes, considering just edges in G_{level}.

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 $O(n^3)$. [3]

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

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?

[3] The idea of finding the blocking flow, i.e., maximum flow possible in the level graph G_{level}, and then iterating this as the distance from s to t gets bigger, is due to Yefim Dinitz. He gave this algorithm while getting his M.Sc. in Soviet Russia in 1969 under G. Adel'son Vel'sky (of AVL trees fame), in response to a exercise in an Algorithms class, and published it in 1970. The iron curtain meant this was not known in the west for a few more years, and then only as Dinic's [*sic*] algorithm. Dinitz talks about his algorithm and some history here. The algorithm we present here is due to V.M. Malhotra, M. Pramod Kumar, and S.N. Maheshwari (1978), and hence is called the *MPM Algorithm*. A similar idea was developed by Alexander Karzanov in Russia in 1974.

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,

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

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.

6.1 A Ford-Fulkerson-like Algorithm

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 show it does not create negative cycles, so we can still find shortest paths in them using the Bellman-Ford algorithm.

Why don't we get negative cycles in the residual graph? Remember that we assumed the initial graph G had no negative-cost cycles. For sake of a contradiction, suppose an augmenting path added in negative-cost back-edges into the residual graph, and we created a negative cycle. This cycle must contain one of these back-edges (v, u). This 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. (You should be careful in this argument when the newly-created negative cycle may contain multiple back edges.) This gives the contradiction, which proves we do not create any negative-cost cycles by augmenting along least-cost s-t paths.

Hence the final flow f found has the property that the residual graph G_f has no negative-cost cycles either. We claim this means that f is optimal. Suppose 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, and respecting arc capacities), since the arc capacities in G_f are $c(u, v) - f(u, v)$. Moreover, $g - f$ has negative cost since g costs less than f. Since this circulation can be broken into a collection of cycles, it must contain at least one 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, to get an algorithm whose running time is like Edmonds-Karp #1, or even one whose runtime just depends on m and n, but we won't discuss that in this course.

In this lecture we describe a very general problem called *linear programming* 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 linear programs (LPs). At the end, we will briefly describe some of the algorithms for solving LPs. 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.

1 Introduction

In recent lectures we have looked at the following problems:

— Bipartite maximum 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 (LPs). 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. Today we will mostly 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., $2S + E - 3P \geq 150$. In addition, our objective function is also linear. We're not allowed things like requiring $SE \geq 100$, since this wouldn't be a linear inequality.

2 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.)

An LP with an objective falls into three categories:

- **Infeasible** (there is no point satisfying the constraints)
- **Feasible and Bounded** (there is a feasible point of maximum objective function value)
- **Feasible and Unbounded** (there is a feasible point of arbitrarily large objective function value)

An algorithm for LP should classify the input LP into one of these categories, and find the optimum feasible point when the LP is feasible and bounded.

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

Variables: S, P, E.

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

Constraints:

$$\begin{aligned} S + P + E &= 168 \\ E &\geq 56 \\ S &\geq 60 \\ 2S + E - 3P &\geq 150 \\ P + E &\geq 70 \\ P &\geq 0 \end{aligned}$$

(can't spend negative time partying)

3 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. For problems where the numbers we are solving for are large (like here), it is usually not a very big deal because you can just round them down to get an almost-optimal solution. However, we will see problems later where it *is* a very big deal.

4 Modeling Network Flow

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

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

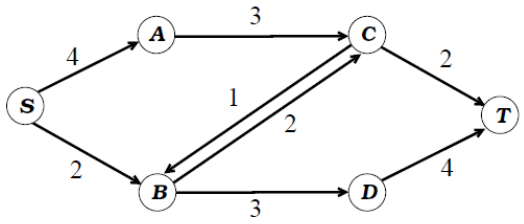
Objective: Maximize $\sum_u f_{ut} - \sum_u f_{tu}$. (maximize the flow into t minus any flow out of t)

Constraints:

- For all edges (u, v), $0 \leq f_{uv} \leq c(u, v)$. (capacity constraints)

– For all $v \notin \{s, t\}$, $\sum_u f_{uv} = \sum_u f_{vu}$. (flow conservation)

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



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

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

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

How about min cost max flow? In min-cost max flow, each edge (u, v) has both a capacity $c(u, v)$ and a cost $w(u, v)$. The goal is to find out of all possible maximum s-t flows the one of least total cost, where the cost of a flow f is defined as

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

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

5 2-Player Zero-Sum Games

Suppose we are given a 2-player zero-sum game with n rows and n columns, and we want to compute a minimax optimal strategy. For instance, perhaps a game like this (say payoffs are for the row player):

20	−10	5
5	10	−10
−5	0	10

Let's see how we can use linear programming to solve this game. 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 (assume our input is given as 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):

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 $\sum_i p_i m_{ij} \geq v$.

6 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 back to our 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 1.

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 $E = 88.5, P = 19.5$, as drawing the contours indicates (see Figure 2).

We can use this geometric view to motivate the algorithms.

The Simplex Algorithm: The earliest and most common algorithm in 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

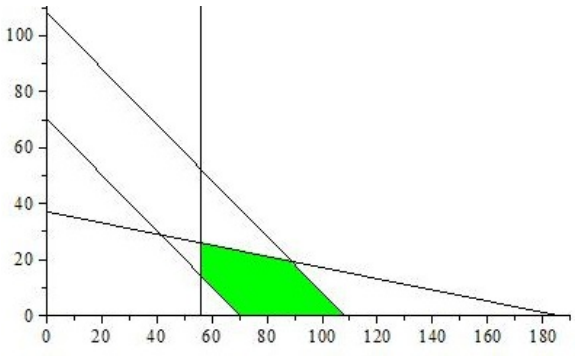


Figure 1: Feasible region for our time-planning problem. The constraints are: $E \geq 56$; $P + E \geq 70$; $P \geq 0$; $S \geq 60$ which means $168 - P - E \geq 60$ or $P + E \leq 108$; and finally $2S - 3P + E \geq 150$ which means $2(168 - P - E) - 3P + E \geq 150$ or $5P + E \leq 186$.

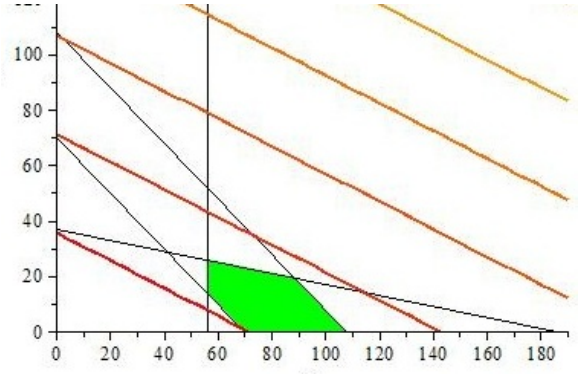


Figure 2: Contours for $2P + E$.

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.

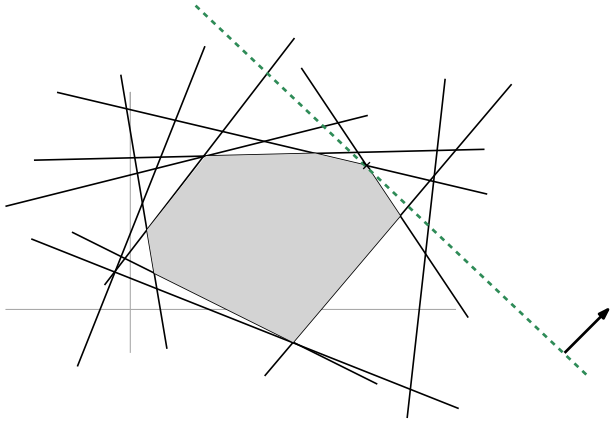
The development of better and better algorithms is a big ongoing area of research. In practice, for all of these algorithms, you get a lot of mileage by using good data structures to speed up the time needed for making each decision.

In this lecture we describe a very nice algorithm due to Seidel for Linear Programming in low-dimensional spaces. We then discuss the general notion of Linear Programming Duality, a powerful tool that you should definitely master.

1 Seidel's LP algorithm

We now describe a linear-programming algorithm due to Raimund Seidel that solves the 2-dimensional (i.e., 2-variable) LP problem in $O(m)$ time (recall, m is the number of constraints), and more generally solves the d-dimensional LP problem in time $O(d!m)$.

Setup: We have d variables $x_1, \ldots, x_d$. We are given m linear constraints in these variables $\mathbf{a}_1 \cdot \mathbf{x} \leq b_1, \ldots, \mathbf{a}_m \cdot \mathbf{x} \leq b_m$ along with an objective $\mathbf{c} \cdot \mathbf{x}$ to maximize. (Using boldface to denote vectors.) Our goal is to find a solution $\mathbf{x}$ satisfying the constraints that maximizes the objective. In the example above, the region satisfying all the constraints is given in gray, the arrow indicates the direction in which we want to maximize, and the cross indicates the $\mathbf{x}$ that maximizes the objective.



(You should think of sweeping the green dashed line, to which the vector $\mathbf{c}$ is normal (i.e., perpendicular), in the direction of $\mathbf{c}$, until you reach the last point that satisfies the constraints. This is the point you are seeking.)

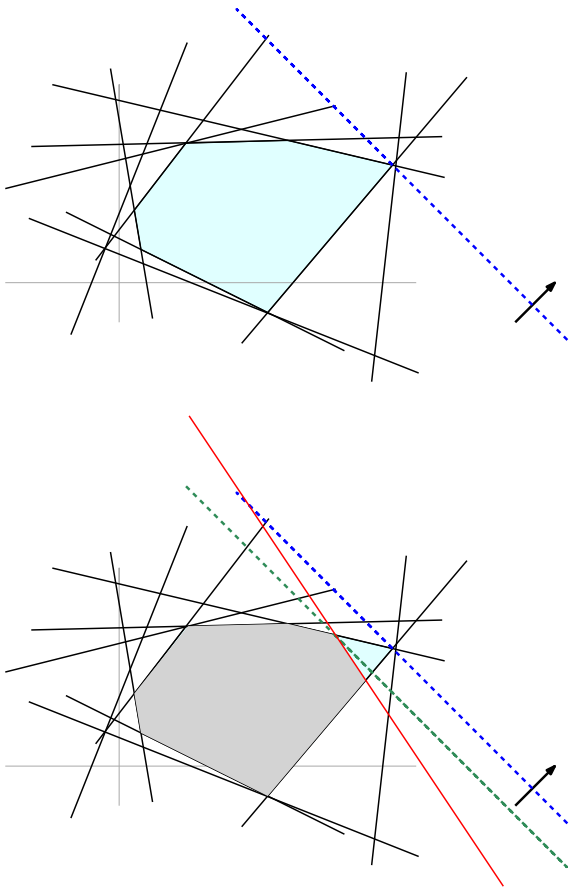
The idea: Here is the idea of Seidel's algorithm. [1] Let's add in the (real) constraints one at a time, and keep track of the optimal solution for the constraints so far. Suppose, for instance, we have found the optimal solution $\mathbf{x}^*$ for the first $m-1$ constraints, and now we add in the mth constraint $\mathbf{a}_m \cdot \mathbf{x} \leq b_m$. There are two cases to consider:

Case 1: If $\mathbf{x}^*$ satisfies the constraint, then $\mathbf{x}^*$ is still optimal. Time to perform this test: $O(d)$.

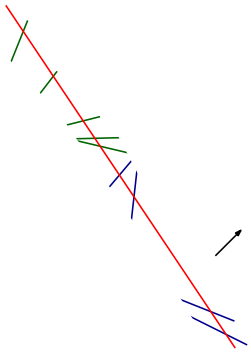
Case 2: If $\mathbf{x}^*$ doesn't satisfy the constraint, then the new optimal point will be on the $(d-1)$-dimensional hyperplane $\mathbf{a}_m \cdot \mathbf{x} = b_m$, or else there is no feasible point.

[1] To keep things simple, let's assume that we have inequalities of the form $-\lambda \leq x_i \leq \lambda$ for all i with sufficiently large λ which are separate from the "real" constraints, so that the starting optimal point is one of the corners of the box $[-\lambda, \lambda]^2$. See Section 1.1 for how to remove this assumption.

As an example, consider the situation below, before and after we add in the linear constraint $\mathbf{a}_m \cdot \mathbf{x} \leq b_m$ colored in red. This causes the feasible region to change from the light blue region to the smaller gray region, and the optimal point to move.



Let's now focus on the case $d = 2$ and consider the time it takes to handle Case 2 above. With $d = 2$, the hyperplane $\mathbf{a}_m \cdot \mathbf{x} = b_m$ is just a line, and let's call one direction "right" and the other "left". We can now scan through all the other constraints, and for each one, compute its intersection point with this line and whether it is "facing" right or left (i.e., which side of that point satisfies the constraint). We find the rightmost intersection point of all the constraints facing to the right and the leftmost intersection point of all that are facing left. If they cross, then there is no solution. Otherwise, the solution is whichever endpoint gives a better value of $\mathbf{c} \cdot \mathbf{x}$ (if they give the same value – i.e., the line $\mathbf{a}_m \cdot \mathbf{x} = b_m$ is perpendicular to $\mathbf{c}$ – then say let's take the rightmost point). In the example above, the 1-dimensional problem is the one in the figure below, with the green constraints "facing" one direction and the blue ones facing the other way. The direction of $\mathbf{c}$ means the optimal point is given by the "lowest" green constraint.



The total time taken here is $O(m)$ since we have $m-1$ constraints to scan through and it takes $O(1)$ time to process each one.

Right now, this looks like an $O(m^2)$-time algorithm for $d=2$, since we have potentially taken $O(m)$ time to add in a single new constraint if Case 2 occurs. But, suppose we add the constraints in *a random order*? What is the probability that constraint m goes to Case 2?

Notice that the optimal solution to all m constraints (assuming the LP is feasible and bounded) is at a corner of the feasible region, and this corner is defined by two constraints, namely the two sides of the polygon that meet at that point. If both of those two constraints have been seen already, then we are guaranteed to be in Case 1. So, if we are inserting constraints in a random order, the probability we are in Case 2 when we get to constraint m is at most $2/m$. This means that the *expected* cost of inserting the mth constraint is at most:

$$\text{E[cost of inserting } m\text{th constraint]} \leq (1-2/m)O(1) + (2/m)O(m) = O(1).$$

This is sometimes called "backwards analysis" since what we are saying is that if we go backwards and pluck out a random constraint from the m we have, the chance it was one of the constraints that mattered was at most $2/m$.

So, Seidel's algorithm is as follows. Place the constraints in a random order and insert them one at a time, keeping track of the best solution so far as above. We just showed that the expected cost of the ith insert is $O(1)$ (or if you prefer, we showed $T(m) = O(1) + T(m-1)$ where $T(i)$ is the expected cost of a problem with i constraints), so the overall expected cost is $O(m)$.

1.1 Handling Special Cases, and Extension to Higher Dimensions*

(We will not be testing you on this part, but you should try to understand it all the same.)

What if the LP is infeasible? There are two ways we can analyze this. One is that if the LP is infeasible, then it turns out this is determined by at most 3 constraints. So we get the same as above with $2/m$ replaced by $3/m$. Another way to analyze this is imagine we have a separate account we can use to pay for the event that we get to Case 2 and find that the LP is infeasible. Since that can only happen once in the entire process (once we determine the LP is infeasible, we stop), this just provides an additive $O(m)$ term. To put it another way, if the system is infeasible, then there will be two cases for the final constraint: (a) it was feasible until then, in which case we pay $O(m)$ out of the extra budget (but the above analysis applies to the the (feasible) first $m-1$ constraints), or (b) it was infeasible already in which case we already halted so we pay 0.

What about unboundedness? We had said for simplicity we could put everything inside a bounding box $-\lambda \leq x_i \leq \lambda$. E.g., if all c_i are positive then the initial $\mathbf{x}^* = (\lambda, \ldots, \lambda)$. However, what value

of λ should we choose? We could actually do the calculations viewing λ symbolically as a limiting quantity which is arbitrarily large. For example, in 2-dimensions, if $\mathbf{c} = (0, 1)$ and we have a constraint like $2x_1 + x_2 \leq 8$, then we would see it is not satisfied by (λ, λ), and hence intersect the contraint with the box and update to $\mathbf{x}^* = (4 - \lambda/2, \lambda)$.

So far we have shown that for $d = 2$, the expected running time of the algorithm is $O(m)$. For general values of d, there are two main changes. First, the probability that constraint m enters Case 2 is now d/m rather than $2/m$. Second, we need to compute the time to perform the update in Case 2. Notice, however, that this is a $(d-1)$-dimensional linear programming problem, and so we can use the same algorithm recursively, after we have spent $O(dm)$ time to project each of the $m - 1$ constraints onto the $(d-1)$-dimensional hyperplane $\mathbf{a}_m \cdot \mathbf{x} = b_m$. Putting this together we have a recurrence for expected running time:

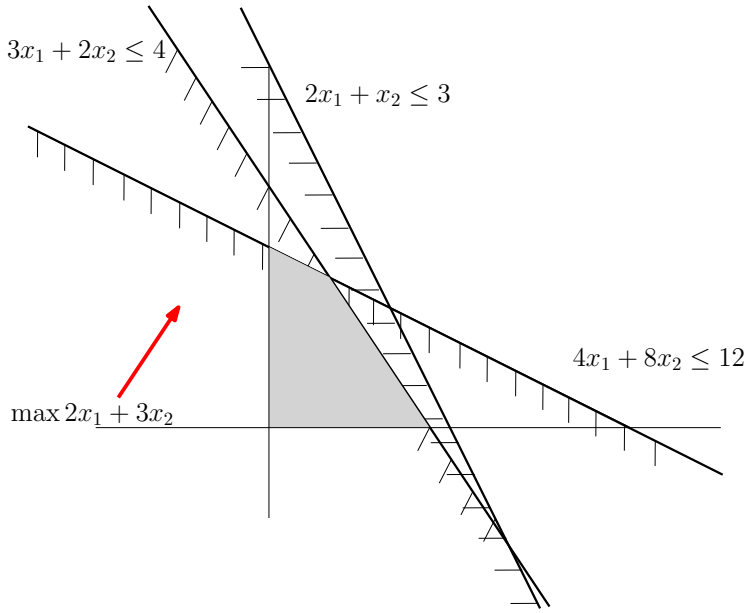
$$T(d, m) \;\leq\; T(d, m-1) + O(d) + \frac{d}{m}[O(dm) + T(d-1, m-1)].$$

This then solves to $T(d, m) = O(d!m)$.

2 Linear Programming Duality

Consider the following LP

$$\begin{aligned} P = \max(2x_1 + 3x_2) \\ \text{s.t.} \quad 4x_1 + 8x_2 &\leq 12 \\ 2x_1 + x_2 &\leq 3 \\ 3x_1 + 2x_2 &\leq 4 \\ x_1, x_2 &\geq 0 \end{aligned} \tag{1}$$



In an attempt to solve P we can produce upper bounds on its optimal value.

- Since $2x_1 + 3x_2 \leq 4x_1 + 8x_2 \leq 12$, we know $\text{OPT}(P) \leq 12$. (The first inequality uses that $2x_1 \leq 4x_1$ because $x_1 \geq 0$, and similarly $3x_2 \leq 8x_2$ because $x_2 \geq 0$.)

- Since $2x_1 + 3x_2 \leq \frac{1}{2}(4x_1 + 8x_2) \leq 6$, we know $\text{OPT}(P) \leq 6$.
- Since $2x_1 + 3x_2 \leq \frac{1}{3}((4x_1 + 8x_2) + (2x_1 + x_2)) \leq 5$, we know $\text{OPT}(P) \leq 5$.

In each of these cases we take a positive[2] linear combination of the constraints, looking for better and better bounds on the maximum possible value of $2x_1 + 3x_2$.

How do we find the "best" lower bound that can be achieved as a linear combination of the constraints? This is just another algorithmic problem, and we can systematically solve it, by letting y_1, y_2, y_3 be the (unknown) coefficients of our linear combination. Then we must have

$$\begin{aligned} 4y_1 + 2y_2 + 3y_2 &\geq 2 \\ 8y_1 + y_2 + 2y_3 &\geq 3 \\ y_1, y_2, y_3 &\geq 0 \\ \text{and we seek } \min(12y_1 + 3y_2 + 4y_3) & \end{aligned} \tag{2}$$

This too is an LP! We refer to this LP (2) as the "dual" and the original LP 1 as the "primal". We designed the dual to serve as a method of constructing an upper bound on the optimal value of the primal, so if y is a feasible solution for the dual and x is a feasible solution for the primal, then $2x_1 + 3x_2 \leq 12y_1 + 3y_2 + 4y_3$. If we can find two feasible solutions that make these equal, then we know we have found the optimal values of these LP.

In this case the feasible solutions $x_1 = \frac{1}{2}, x_2 = \frac{5}{4}$ and $y_1 = \frac{5}{16}, y_2 = 0, y_3 = \frac{1}{4}$ give the same value 4.75, which therefore must be the optimal value.

> **Exercise:** The dual LP is a *minimization* LP, where the constraints are of the form $lhs_i \geq rhs_i$. You can try to give *lower* bounds on the optimal value of this LP by taking positive linear combinations of these constraints. E.g., argue that
>
> $$12y_1 + 3y_2 + 4y_3 \geq 4y_1 + 2y_2 + 3y_2 \geq 2$$
>
> (since $y_i \geq 0$ for all i) and
>
> $$12y_1 + 3y_2 + 4y_3 \geq 8y_1 + y_2 + 2y_3 \geq 3$$
>
> and
>
> $$12y_1 + 3y_2 + 4y_3 \geq \frac{2}{3}(4y_1 + 2y_2 + 3y_2) + (8y_1 + y_2 + 2y_3) \geq \frac{4}{3} + 3 = 4\frac{1}{3}.$$
>
> Formulate the problem of finding the best lower bound obtained by linear combinations of the given inequalities as an LP. Show that the resulting LP is the same as the primal LP 1.

> **Exercise:** Consider the "primal" LP below on the left:
>
> $$\begin{aligned} P = \max(7x_1 - x_2 + 5x_3) & \\ \text{s.t.} \quad x_1 + x_2 + 4x_3 &\leq 8 \\ 3x_1 - x_2 + 2x_3 &\leq 3 \\ 2x_1 + 5x_2 - x_3 &\leq -7 \\ x_1, x_2, x_3 &\geq 0 \end{aligned} \qquad \begin{aligned} D = \min(8y_1 + 3y_2 - 7y_3) & \\ \text{s.t.} \quad y_1 + 3y_2 + 2y_3 &\geq 7 \\ y_1 - y_2 + 5y_3 &\geq -1 \\ 4y_1 + 2y_2 - y_3 &\geq 5 \\ y_1, y_2, y_3 &\geq 0 \end{aligned}$$
>
> Show that the problem of finding the best upper bound obtained using linear combinations of the constraints can be written as the LP above on the right (the "dual" LP). Also, now formulate the problem of finding a lower bound for the dual LP. Show this lower-bounding LP is just the primal (P).

[2] Why positive? If you multiply by a negative value, the sign of the inequality changes.

Exercise: In the examples above, the maximization LPs had constraints of the form $lhs_i \leq rhs_i$, and the rhs were all scalars, so taking positive linear combinations gave us $blah \leq number$, i.e., an *upper* bound as we wanted. However, suppose the primal LP has some "nice" constraints $lhs_i \leq rhs_i$ and others are "not nice" $lhs_i \geq rhs_i$, e.g., like the left one below. Show that the dual has non-positive variables for the non-nice constraints. For example,

$$\begin{aligned} P = \max(7x_1 - x_2 + 5x_3) \\ \text{s.t.} \quad x_1 + x_2 + 4x_3 \leq 8 \\ 3x_1 - x_2 + 2x_3 \geq 3 \\ x_1, x_2, x_3 \geq 0 \end{aligned} \qquad \begin{aligned} D = \min(8y_1 + 3y_2) \\ \text{s.t.} \quad y_1 + 3y_2 \geq 7 \\ y_1 - y_2 \geq -1 \\ 4y_1 + 2y_2 \geq 5 \\ y_1 \geq 0, y_2 \leq 0 \end{aligned}$$

Another way is to replace $lhs_i \geq rhs_i$ in P by the equivalent constraint $(-lhs_i) \leq (-rhs_i)$ and get to an LP P' with only nice constraints. Show that the dual D' for P' is equivalent to the dual D for P.

2.1 The Method

Consider the examples/exercises above. In all of them, we started off with a "primal" maximization LP:

$$\begin{aligned} &\text{maximize } \mathbf{c}^T\mathbf{x} \\ &\text{subject to } A\mathbf{x} \leq \mathbf{b} \\ &\qquad\qquad \mathbf{x} \geq \mathbf{0}, \end{aligned} \tag{3}$$

The constraint $\mathbf{x} \geq \mathbf{0}$ is just short-hand for saying that the $\mathbf{x}$ variables are constrained to be non-negative.[3] And to get the best lower bound we generated a "dual" minimization LP:

$$\begin{aligned} &\text{minimize } \mathbf{r}^T\mathbf{y} \\ &\text{subject to } P\mathbf{y} \geq \mathbf{q} \\ &\qquad\qquad \mathbf{y} \geq \mathbf{0}, \end{aligned} \tag{4}$$

The important thing is: this matrix P, and vectors $\mathbf{q}, \mathbf{r}$ are not just any vectors. Look carefully: $P = A^T$. $\mathbf{q} = \mathbf{c}$ and $\mathbf{r} = \mathbf{b}$. The dual is in fact:

$$\begin{aligned} &\text{minimize } \mathbf{y}^T\mathbf{b} \\ &\text{subject to } \mathbf{y}^T A \geq \mathbf{c}^T \\ &\qquad\qquad \mathbf{y} \geq \mathbf{0}, \end{aligned} \tag{5}$$

And if you take the dual of (5) to try to get the best lower bound on this LP, you'll get (4). *The dual of the dual is the primal.* The dual and the primal are best upper/lower bounds you can obtain as linear combinations of the inputs.

The natural question is: maybe we can obtain better bounds if we combine the inequalities in more complicated ways, not just using linear combinations. Or do we obtain optimal bounds using just linear combinations? In fact, we get optimal bounds using just linear combinations, as the next theorems show.

[3] We use the convention that vectors like $\mathbf{c}$ and $\mathbf{x}$ are column vectors. So $\mathbf{c}^T$ is a row vector, and thus $\mathbf{c}^T\mathbf{x}$ is the same as the inner product $\mathbf{c} \cdot \mathbf{x} = \sum_i c_i x_i$. We often use $\mathbf{c}^T\mathbf{x}$ and $\mathbf{c} \cdot \mathbf{x}$ interchangeably. Also, $\mathbf{a} \leq \mathbf{b}$ means component-wise inequality, i.e., $a_i \leq b_i$ for all i.

2.2 The Theorems

It is easy to show that the dual (5) provides an upper bound on the value of the primal (4):

Theorem 1 (Weak Duality) *If* $\mathbf{x}$ *is a feasible solution to the primal LP (4) and* $\mathbf{y}$ *is a feasible solution to the dual LP (5) then*

$$\mathbf{c}^T\mathbf{x} \leq \mathbf{y}^T\mathbf{b}.$$

Proof: This is just a sequence of trivial inequalities that follow from the LPs above:

$$\mathbf{c}^T\mathbf{x} \leq (y^T A)\mathbf{x} = y^T(A\mathbf{x}) \leq y^T b.$$

■

The amazing (and deep) result here is to show that the dual actually gives a perfect upper bound on the primal (assuming some mild conditions).

Theorem 2 (Strong Duality Theorem) *Suppose the primal LP (4) is feasible (i.e., it has at least one solution) and bounded (i.e., the optimal value is not* ∞*). Then the dual LP (5) is also feasible and bounded. Moreover, if* $\mathbf{x}^*$ *is the optimal primal solution, and* $\mathbf{y}^*$ *is the optimal dual solution, then*

$$\mathbf{c}^T\mathbf{x}^* = (\mathbf{y}^*)^T\mathbf{b}.$$

In other words, the maximum of the primal equals the minimum of the dual.

Why is this useful? If I wanted to prove to you that $\mathbf{x}^*$ was an optimal solution to the primal, I could give you the solution $\mathbf{y}^*$, and you could check that $\mathbf{x}^*$ was feasible for the primal, $\mathbf{y}^*$ feasible for the dual, and they have equal objective function values.

This min-max relationship is like in the case of s-t flows: the maximum of the flow equals the minimum of the cut. Or like in the case of zero-sum games: the payoff for the maxmin-optimum strategy of the row player equals the (negative) of the payoff of the maxmin-optimal strategy of the column player. Indeed, both these things are just special cases of strong duality!

We will not prove Theorem 2 in this course, though the proof is not difficult. But let's give a geometric intuition of why this is true in the next section.

2.3 The Geometric Intuition for Strong Duality

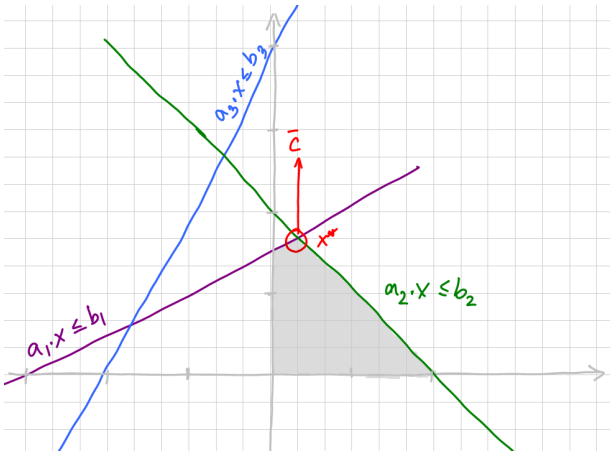
To give a geometric view of the strong duality theorem, consider an LP of the following form:

$$\begin{aligned} \text{maximize } & \mathbf{c}^T\mathbf{x} \\ \text{subject to } & A\mathbf{x} \leq \mathbf{b} \\ & \mathbf{x} \geq 0 \end{aligned} \tag{6}$$

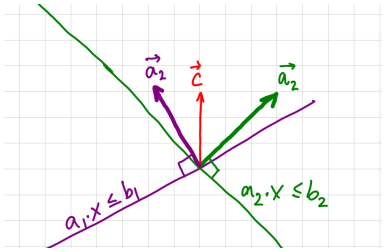
For concreteness, let's take the following 2-dimensional LP:

$$\begin{aligned} \text{maximize } & x_2 \\ \text{subject to } & -x_1 + 2x_2 \leq 3 \\ & x_1 + x_2 \leq 2 \\ & -2x_1 + x_2 \leq 4 \\ & x_1, x_2 \geq 0 \end{aligned}$$

If $\mathbf{c} := (0, 1)$, then the objective function wants to maximize $\mathbf{c} \cdot \mathbf{x}$, i.e., to go as far up in the vertical direction as possible. As we have already argued before, the optimal point $\mathbf{x}^*$ must be obtained at the intersection of two constraints for this 2-dimensional problem (n tight constraints for n dimensions). In this case, these happen to be the first two constraints.



If $\mathbf{a}_1 = (-1, 2), b_1 = 3$ and $\mathbf{a}_2 = (1, 1), b_2 = 2$, then $\mathbf{x}^*$ is the (unique) point $\mathbf{x}$ satisfying both $\mathbf{a}_1 \cdot \mathbf{x} = b_1$ and $\mathbf{a}_2 \cdot \mathbf{x} = b_2$. Indeed, we're being held down by these two constraints. Geometrically, this means that $\mathbf{c} = (0, 1)$ lies "between" these the vectors $\mathbf{a}_1$ and $\mathbf{a}_2$ that are normal (perpendicular) to these constraints.



Consequently, $\mathbf{c}$ can be written as a positive linear combination of $\mathbf{a}_1$ and $\mathbf{a}_2$. (It "lies in the cone formed by $\mathbf{a}_1$ and $\mathbf{a}_2$.") I.e., for some positive values y_1 and y_2,

$$\mathbf{c} = y_1\mathbf{a}_1 + y_2\mathbf{a}_2.$$

Great. Now, take dot products on both sides with $\mathbf{x}^*$. We get

$$\begin{aligned} \mathbf{c} \cdot \mathbf{x}^* &= (y_1\,\mathbf{a}_1 + y_2\,\mathbf{a}_2) \cdot \mathbf{x}^* \\ &= y_1(\mathbf{a}_1 \cdot \mathbf{x}^*) + y_2(\mathbf{a}_2 \cdot \mathbf{x}^*) \\ &= y_1 b_1 + y_2 b_2 \end{aligned}$$

Defining $\mathbf{y} = (y_1, y_2, 0, \ldots, 0)$, we get

$$\text{optimal value of primal} = \mathbf{c} \cdot \mathbf{x}^* = \mathbf{b} \cdot \mathbf{y} \geq \text{value of dual solution } \mathbf{y}.$$

The last inequality follows because

- the $\mathbf{y}$ we found satisfies $\mathbf{c} = y_1\mathbf{a}_1 + y_2\mathbf{a}_2 = \sum_i y_i\mathbf{a}_i = A^T\mathbf{y}$, and hence $\mathbf{y}$ satisfies the dual constraints $\mathbf{y}^T A \geq \mathbf{c}^T$ by construction.

In other words, $\mathbf{y}$ is a feasible solution to the dual, has value $\mathbf{b} \cdot \mathbf{y} \leq \mathbf{c} \cdot \mathbf{x}^*$. So the *optimal* dual value cannot be less. Combined with weak duality (which says that $\mathbf{c} \cdot \mathbf{x}^* \leq \mathbf{b} \cdot \mathbf{y}$), we get strong duality

$$\mathbf{c} \cdot \mathbf{x}^* = \mathbf{b} \cdot \mathbf{y}.$$

Above, we used that the optimal point was constrained by two of the inequalities (and that these were not the non-negativity constraints). The general proof is similar: for n dimensions, we just use that the optimal point is constrained by n tight inequalities, and hence $\mathbf{c}$ can be written as a positive combination of n of the constraints (possibly some of the non-negativity constraints too).

3 Example #1: Zero-Sum Games

Consider a 2-player zero-sum game defined by an n-by-m payoff matrix R for the row player. That is, if the row player plays row i and the column player plays column j then the row player gets payoff R_{ij} and the column player gets $-R_{ij}$. To make this easier on ourselves (it will allow us to simplify things a bit), let's assume that all entries in R are positive (this is really without loss of generality since as pre-processing one can always translate values by a constant and this will just change the game's value to the row player by that constant). We saw we could write this as an LP:

- Variables: $v, p_1, p_2, \ldots, p_n$.
- Maximize v,
- Subject to:

  $p_i \geq 0$ for all rows i,

  $\sum_i p_i = 1$,

  $\sum_i p_i R_{ij} \geq v$, for all columns j.

To put this into the form of (4), we can replace $\sum_i p_i = 1$ with $\sum_i p_i \leq 1$ since we said that all entries in R are positive, so the maximum will occur with $\sum_i p_i = 1$, and we can also safely add in the constraint $v \geq 0$. We can also rewrite the third set of constraints as $v - \sum_i p_i R_{ij} \leq 0$. This then gives us an LP in the form of (4) with

$$\mathbf{x} = \begin{bmatrix} v \\ p_1 \\ p_2 \\ \ldots \\ p_n \end{bmatrix}, \mathbf{c} = \begin{bmatrix} 1 \\ 0 \\ 0 \\ \ldots \\ 0 \end{bmatrix}, \mathbf{b} = \begin{bmatrix} 0 \\ 0 \\ \ldots \\ 0 \\ 1 \end{bmatrix}, \text{and } A = \left[\begin{array}{c|ccc} 1 & & & \\ 1 & & -R^T & \\ \ldots & & & \\ 1 & & & \\ \hline 0 & 1 & \ldots & 1 \end{array}\right].$$

I.e., maximizing $\mathbf{c}^T\mathbf{x}$ subject to $A\mathbf{x} \leq \mathbf{b}$ and $\mathbf{x} \geq \mathbf{0}$.

We can now write the dual, following (5). Let $\mathbf{y}^T = (y_1, y_2, \ldots, y_{m+1})$. We now are asking to minimize $\mathbf{y}^T\mathbf{b}$ subject to $\mathbf{y}^T A \geq c^T$ and $\mathbf{y} \geq \mathbf{0}$. In other words, we want to:

- Minimize y_{m+1},

- Subject to:

  $y_1 + \ldots + y_m \geq 1$,

  $-y_1 R_{i1} - y_2 R_{i2} - \ldots - y_m R_{im} + y_{m+1} \geq 0$ for all rows i,

  or equivalently,

  $y_1 R_{i1} + y_2 R_{i2} + \ldots + y_m R_{im} \leq y_{m+1}$ for all rows i.

So, we can interpret y_{m+1} as the value to the row player, and $y_1, \ldots, y_m$ as the randomized strategy of the column player, and we want to find a randomized strategy for the column player that minimizes y_{m+1} subject to the constraint that the row player gets *at most* y_{m+1} no matter what row he plays. Now notice that we've only required $y_1 + \ldots + y_m \geq 1$, but since we're minimizing and the R_{ij}'s are positive, the minimum will happen at equality.

Notice that the fact that the maximum value of v in the primal is equal to the minimum value of y_{m+1} in the dual follows from strong duality. Therefore, the minimax theorem is a corollary to the strong duality theorem.

4 Example #2: Shortest Paths

Duality allows us to write problems in multiple ways, which often gives us power and flexibility. For instance, let us see two ways of writing the shortest s-t path problem, and why they are equal.

Here is an LP for computing an s-t shortest path with respect to the edge lengths $\ell(u, v) \geq 0$:

$$\begin{aligned} \max \quad & d_t \\ \text{subject to} \quad & d_s = 0 \\ & d_v - d_u \leq \ell(u, v) \qquad \forall (u, v) \in E \end{aligned} \tag{7}$$

The constaints are the natural ones: the shortest distance from s to s is zero. And if the s-u distance is d_u, the s-v distance is at most $d_u + \ell(u, v)$ — i.e., $d_v \leq d_u + \ell(u, v)$. It's like putting strings of length $\ell(u, v)$ between u, v and then trying to send t as far from s as possible—the farthest you can send t from s is when the shortest s-t path becomes tight.

Here is another LP that also computes the s-t shortest path:

$$\begin{aligned} \min & \textstyle\sum_e \ell(e)\, y_e \\ \text{subject to} \quad & \textstyle\sum_{w:(s,w)\in E} y_{sw} = 1 \\ & \textstyle\sum_{v:(v,t)\in E} y_{vt} = 1 \\ & \textstyle\sum_{v:(u,v)\in E} y_{uv} = \sum_{v:(v,w)\in E} y_{vw} \qquad \forall w \in V \setminus \{s, t\} \\ & y_e \geq 0. \end{aligned} \tag{8}$$

In this one we're sending one unit of flow from s to t, where the cost of sending a unit of flow on an edge equals its length ℓ_e. Naturally the cheapest way to send this flow is along a shortest s-t path length. So both the LPs should compute the same value. Let's see how this follows from duality.

4.1 Duals of Each Other

Take the first LP. Since we're setting d_s to zero, we could hard-wire this fact into the LP. So we could rewrite (7) as

$$\begin{aligned} & \max \quad d_t & \\ \text{subject to} \quad & d_v - d_u \le \ell(u,v) & \forall (u,v) \in E, s \notin \{u,v\} \\ & d_v \le \ell(s,v) & \forall (s,v) \in E \\ & -d_u \le \ell(u,s) & \forall (u,s) \in E \end{aligned} \tag{9}$$

Moreover, the distances are never negative for $\ell(u,v) \ge 0$, so we can add in the constraint $d_v \ge 0$ for all $v \in V$.

How to find an upper bound on the value of this LP? The LP is in the standard form, so we can do this mechanically. But let us do this from starting from the definition of the dual as the "best upper bound".

Let us define $E_s^{out} := \{(s,v) \in E\}$, $E_s^{in} := \{(u,s) \in E\}$, and $E^{rest} := E \setminus (E_s^{out} \cup E_s^{in})$. For every arc $e = (u,v)$ we will have a variable $y_e \ge 0$. We want to get the best upper bound on d_t by linear combinations of the the constraints, so we should find a solution to

$$\sum_{e \in E^{rest}} y_{uv} \, (d_v - d_u) + \sum_{e \in E_s^{out}} y_{sv} \, d_v - \sum_{e \in E_s^{in}} y_{us} \, d_u \ge d_t \tag{10}$$

(this is like $\mathbf{y}^T A \ge c$) and the objective function is to

$$\text{minimize} \quad \sum_{(u,v) \in E} y_{uv} \, \ell(u,v). \tag{11}$$

(This is like $\min \mathbf{y}^T \mathbf{b}$.) Great, the objective function (11) is exactly what we want, but what about the craziness in (10)? Just collect all copies of each of the variables d_v, and it now says

$$\sum_{v \ne s} d_v \left(\sum_{u:(u,v) \in E} y_{uv} - \sum_{w:(v,w) \in E} y_{vw} \right) \ge d_t.$$

First, this must be an equality at optimality (since otherwise we could reduce the y values). Moreover, these equalities must hold regardless of the d_v values, so this is really the same as

$$\begin{aligned} \sum_{u:(u,v) \in E} y_{uv} - \sum_{w:(v,w) \in E} y_{vw} = 0 \qquad & \forall v \notin \{s,t\}. \\ \sum_{u:(u,t) \in E} y_{ut} - \sum_{w:(t,w) \in E} y_{tw} = 1. & \end{aligned} \tag{12}$$

Summing all these inequalities for all nodes $v \in V \setminus \{s\}$ gives us the missing equality:

$$\sum_{w:(s,w) \in E} y_{sw} - \sum_{u:(u,s) \in E} y_{us} = 1.$$

Finally, observe that since there's flow conservation at all nodes, and the net unit flow leaving s and reaching t, this means we must have a possibly-empty circulation (i.e., flow going around in circles) plus one unit of s-t flow. Removing the circulation can only lower the objective function, so at optimality we're left with one unit of flow from s to t. This is precisely the LP (8), showing that the dual of LP (7) is LP (8), after a small amount of algebra.

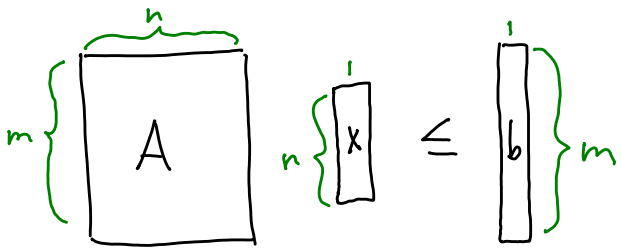
15-451/651: Algorithms **CMU, Spring 2015**
Lecture #19: Linear Programming in Two Dimensions March 30, 2015
Lecturer: Danny Sleator

1 Standard Linear Programming Terminology and Notation

A linear program (LP) written in the following form is said to be in *Standard Form*:

$$\begin{aligned} &\text{maximize } c^T x \\ &\text{subject to } Ax \le b \\ &x \ge 0 \end{aligned}$$

Here there are n non-negative variables $x_1, x_1, \ldots, x_n$, and m linear constraints encapsulated in the $m \times n$ matrix A and the $m \times 1$ matrix (vector) b. The objective function to be maximized is represented by the $n \times 1$ matrix (vector) c.



Any LP can be expressed in standard form. Example 1: if we are given an LP with some linear equalities, we can split each equality into two inequalities. Example 2: if we need a variable x_i to be allowed to be positive or negative, we replace it by the difference between two variables x_i' and x_i''. Let $x_i = x_i' - x_i''$, and eliminate all occurrences of x in the LP by this substitution.

An LP is *feasible* if there exists a point x satisfying the constraints, and *infeasible* otherwise.

An LP is *unbounded* if $\forall B\ \exists x$ such that x is feasible and $c^T x > B$. Otherwise it is *bounded*.

An LP has an *optimal solution* iff it is feasible and bounded.

The job of an LP solver is to classify a given LP into these categories, and if it is feasible and bounded, it should return a point with the optimum value of the objective function.

2 LP in Two Dimensions

In this lecture I present an algorithm of Raimund Seidel to solve LPs with two variables and m constraints in expected $O(m)$ time.

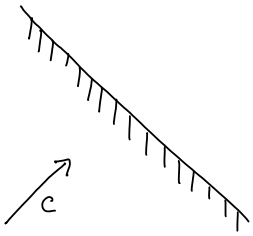
The input to the problem is an LP in the following form:

$$\begin{aligned} &\text{maximize } c^T x \\ &\text{subject to } Ax \leq b \end{aligned}$$

Note that here we don't require that the variables be non-negative. This is strictly more general than standard form. If we do wish the variables to be non-negative, we can simply add those constraints to A (increasing m by two).

We will think about the algorithm geometrically. Each constraint is a half-space. We number the constraints $C_1, C_2, \ldots C_m$. The objective function is a 2D vector, and we are trying to find a point which is farthest in that direction.

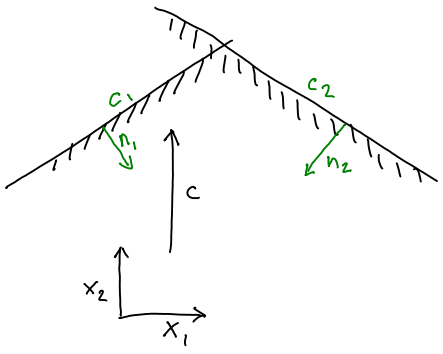
To simplify the explanation of the algorithm we will make an additional assumption about the linear system. We assume that there is no constraint which is perpendicular to the objective function. In other words, we don't allow this kind of situation:



Seidel's 2D LP Algorithm:

Part 1: Determine Boundedness

For the purposes of this part, rotate the entire system so that the c vector points in the positive x_2 direction. (Such a rotation has no effect on the boundedness of the LP.) Now search through all of the constraints, looking for ones whose normal (in the direction of the satisfying side of the constraint) are down and to the right and down and to the left. If there exists one or more of each type, then the system is bounded. If not then the system is unbounded. In this case the algorithm returns "unbounded".



Now reorder the constraints so that the two bounding ones that we just found are numbered C_1 and C_2. The remaining constraints are $C_3, \ldots, C_m$.

Part 2: Finding the Optimum Solution

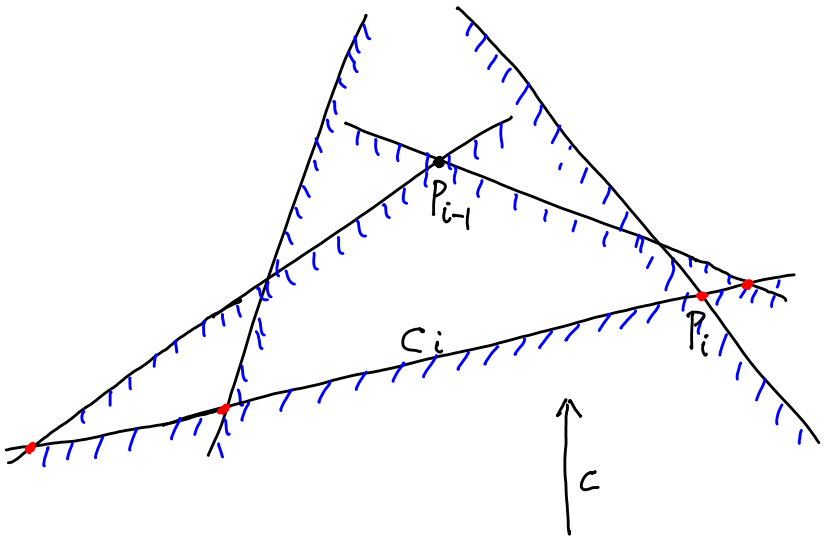
Now randomly permute the remaining constraints $C_3, \ldots, C_m$. We're going to process them in that order. We're going to generate a sequence of points $P_2, P_3, \ldots, P_m$ such that P_i is the optimum solution to the first i constraints. We can immediately compute P_2, which is the intersection point between C_1 and C_2.

For each i from 3 to m do:

Test if P_{i-1} satisfies constraint C_i. If it does, let $P_i = P_{i-1}$ and continue the loop.

So P_{i-1} violates the constraint C_i. Now we try to generate a new point P_i that satisfies all the constraints $C_1, \ldots, C_i$.

Let L_i be the line of the boundary of the constraint C_i. Each of the constraints $C_1, \ldots, C_{i-1}$ that is not parallel to L_i maps to a 1D constraint inside the line L_i. (The ones that are parallel to L_1 are irrelevant.) In addition, the objective function also maps to a direction inside of L_i that optimizes it. (Here again we use the assumption that c is not perpendicular to L_i.) This 1D LP problem is easily solved by constructing the feasible interval. If it contains no points then our LP is infeasible, so return "infeasible". Otherwise take the end of the feasible interval that has the maximum objective function value. This is our point P_i. In the figure below, the intersection between all the prior constraints and L_i are shown in red.



If the loop completes then the optimum solution is P_m. (If it does not complete, then it must have returned "infeasible" already.)

Theorem: Seidel's algorithm runs in expected $O(m)$ time.

Proof: Note that at any point in time we have a point P_i which is the optimum solution to the constraints $C_1, \ldots, C_i$. We also have two constraints among these, call them C_k and C_l, which are the ones that prove the bound on the objective function, and whose intersection point is P_i.

The time it takes to go to compute P_i from P_{i-1} depends on whether or not P_{i-1} satisfies C_i. If it does, then the step is $O(1)$ time. Otherwise the algorithm must look at all the previous constraints and takes $O(i)$ time.

So let's use backward analysis. Suppose we are at a point in time when we just computed P_i as the optimum solution to $C_1, \ldots, C_i$. We now randomly remove one of the constraints $C_3, \ldots, C_i$. What is the probability that P_i differs from P_{i-1}? In order for this to happen we must remove one of the two constraints that constrain the current P_i. What is an upper bound on the probability of this happening?

The probability of this happening is at most $2/(i-2)$. This happens when just two constraints go through P_i, and those are from among $C_3, \ldots, C_i$. In which case the probability of removing one of those two is at most $2/(i-2)$. In all other cases the probability is lower. For example if several constraints conspire to bound P_i. This only lowers the probability that P_i changes. It's also lowered if C_1 and/or C_2 are involved in constraining P_i, cause these will not be removed.

So the cost of computing P_i is at most $2i$ with probability $2/(i-2)$, and 1 with the remaining probability. So the expected cost of the step is at most $2i/(i-2) \leq 6$ because $i \geq 3$. So the expected running time of the algorithm is $O(m)$. QED.

Today we discuss an algorithm for solving linear programming feasibility problems of the form $A\mathbf{w} \geq \mathbf{1}$, where $\mathbf{w}$ is our vector of variables. It will in fact do more than this, as we will see.[1]

1 Online linear classification

Consider the following problem. You have a sequence of email messages arriving one at a time $\mathbf{a}_1, \mathbf{a}_2, \mathbf{a}_3, \ldots$, and for each one you (the algorithm) have to decide whether to mark it "important" or "not important". After marking it, you are then told (by your user) if you were correct or if you made a mistake.

Let's assume that email messages are represented as vectors in some space $\mathbb{R}^n$, such as indicating how many times each word in the dictionary appears in the email message (in this case, n is the number of words in the dictionary, and this would be called a "bag of words" model). Let's furthermore suppose that there exists some unknown weight vector $\mathbf{w}^*$ such that $\mathbf{a}_i \cdot \mathbf{w}^* \geq 1$ for the important emails (the *positive* examples) and $\mathbf{a}_i \cdot \mathbf{w}^* \leq -1$ for the non-important emails (the *negative* examples). Our goal is to give an algorithm for performing this task that makes as few mistakes as possible.

2 The Perceptron Algorithm

The Perceptron Algorithm is one such algorithm for this problem. It maintains a weight vector $\mathbf{w}$ that it uses for prediction (predicting positive on $\mathbf{a}_i$ if $\mathbf{a}_i \cdot \mathbf{w} > 0$ and predicting negative if $\mathbf{a}_i \cdot \mathbf{w} < 0$) and then updates it when it makes a mistake (let's say it says "I don't know" if $\mathbf{a}_i \cdot \mathbf{w} = 0$ and so makes a mistake either way). What we will prove about the algorithm is the following theorem.

Theorem 1 *[Block ('62), Novikoff ('62), Minsky-Papert ('69)] On any sequence of examples $\mathbf{a}_1, \mathbf{a}_2, \ldots$, if there exists a consistent $\mathbf{w}^*$, i.e., $\mathbf{a}_i \cdot \mathbf{w}^* \geq 1$ for the positive examples and $\mathbf{a}_i \cdot w^* \leq -1$ for the negative examples, then the Perceptron algorithm makes at most $R^2||\mathbf{w}^*||^2$ mistakes, where $R = \max_i ||\mathbf{a}_i||$.*

To get a feel for this statement, notice that if we multiply all entries in all the $\mathbf{a}_i$ by 100, we can divide all entries in $\mathbf{w}^*$ by 100 and it will still be consistent. So the bound is invariant to this kind of scaling (i.e., what our "units" are).

Secondly, if we rewrite $\mathbf{a}_i \cdot \mathbf{w}^* \geq 1$ as $\mathbf{a}_i \cdot \mathbf{w}^*/||\mathbf{w}^*|| \geq 1/||\mathbf{w}^*||$, and we think of the $\mathbf{a}_i$ as points in $\mathbb{R}^n$, then we can think of the LHS as the distance of $\mathbf{a}_i$ to the hyperplane $\mathbf{a} \cdot \mathbf{w}^* = 0$. So, we can think of w^* as defining a *linear separator* that separates the positive examples from the negative examples, and what the theorem is saying is that if there exists a hyperplane that correctly separates the positive examples from the negative examples by a large *margin*, then the total number of mistakes will be small.

2.1 Perceptron for Solving Linear Programs

Before giving the algorithm and analysis, note that by the above theorem, we can use it to solve "feasibility" linear programs of the form $A\mathbf{w} \geq \mathbf{1}$. I.e., these are LPs where we just want to find

[1] Note the change in notation, using $\mathbf{w}$ instead of $\mathbf{x}$; this will avoid confusion: traditionally in our machine learning motivation, the data is called $\mathbf{x}_1, \mathbf{x}_2, \ldots$, whereas in LPs $\mathbf{x}$ is the solution. So we'll just not use $\mathbf{x}$ at all today.

$\mathbf{w} \in \mathbb{R}^n$ such that $\mathbf{a}_i \cdot \mathbf{w} \geq 1$ for all $i \in [m]$. (Since A may contain both positive and negative values, finding such a $\mathbf{w}$ is not trivial. Such LPs are called "cone" LPs.) In fact, given any LP of the form $A\mathbf{w} \geq \mathbf{b}$, we could multiply each entry on the i^{th} row $\mathbf{a}_i$ of A by $1/b_i$, and transform it into $A'\mathbf{w} \geq \mathbf{1}$ where $a'_{ij} = a_{ij}/b_i$.

How to solve this feasibility LP? Just cycle through the constraints (viewing each one as a positive example) until the algorithm stops making any more mistakes. At this point we have $A\mathbf{w} > \mathbf{0}$, and we can just scale up $\mathbf{w}$ to be large enough so that $A\mathbf{w} \geq \mathbf{1}$.

2.2 The algorithm

The Perceptron algorithm (due to Frank Rosenblatt) is simply the following:

1. Begin with $\mathbf{w} = \mathbf{0}$.
2. Given $\mathbf{a}_i$, predict positive if $\mathbf{a}_i \cdot \mathbf{w} > 0$ and predict negative if $\mathbf{a}_i \cdot \mathbf{w} < 0$, else say "I don't know".
3. If a mistake was made (or the prediction was "I don't know"):
   - If the correct answer was "positive", update: $\mathbf{w} \leftarrow \mathbf{w} + \mathbf{a}_i$.
   - If the correct answer was "negative", update: $\mathbf{w} \leftarrow \mathbf{w} - \mathbf{a}_i$.

2.3 The analysis

Proof (Theorem 1): Fix some consistent $\mathbf{w}^*$. We will keep track of two quantities, $\mathbf{w} \cdot \mathbf{w}^*$ and $||\mathbf{w}||^2$.

- First of all, each time we make a mistake, $\mathbf{w} \cdot \mathbf{w}^*$ increases by at least 1. That is because if $\mathbf{a}_i$ is a positive example, then

  $$(\mathbf{w} + \mathbf{a}_i) \cdot \mathbf{w}^* = \mathbf{w} \cdot \mathbf{w}^* + \mathbf{a}_i \cdot \mathbf{w}^* \geq \mathbf{w} \cdot \mathbf{w}^* + 1,$$

  by definition of $\mathbf{w}^*$. Similarly, if $\mathbf{a}_i$ is a negative example, then

  $$(\mathbf{w} - \mathbf{a}_i) \cdot \mathbf{w}^* = \mathbf{w} \cdot \mathbf{w}^* - \mathbf{a}_i \cdot \mathbf{w}^* \geq \mathbf{w} \cdot \mathbf{w}^* + 1.$$

- Next, on each mistake, we claim that $||\mathbf{w}||^2$ increases by at most R^2. Let us first consider mistakes on positive examples. If we make a mistake on a positive example $\mathbf{a}_i$ then we have

  $$(\mathbf{w} + \mathbf{a}_i) \cdot (\mathbf{w} + \mathbf{a}_i) = ||\mathbf{w}||^2 + 2\mathbf{w} \cdot \mathbf{a}_i + ||\mathbf{a}_i||^2 \leq ||\mathbf{w}||^2 + ||\mathbf{a}_i||^2 \leq ||\mathbf{w}||^2 + R^2,$$

  where the middle inequality comes from the fact that we made a mistake, which means that $\mathbf{w} \cdot \mathbf{a}_i \leq 0$. Similarly, if we make a mistake on a negative example $\mathbf{a}_i$ then we have

  $$(\mathbf{w} - \mathbf{a}_i) \cdot (\mathbf{w} - \mathbf{a}_i) = ||\mathbf{w}||^2 - 2\mathbf{w} \cdot \mathbf{a}_i + ||\mathbf{a}_i||^2 \leq ||\mathbf{w}||^2 + ||\mathbf{a}_i||^2 \leq ||\mathbf{w}||^2 + R^2.$$

  Note that it is important here that we only update on a mistake.

So, if we make M mistakes, then $\mathbf{w} \cdot \mathbf{w}^* \geq M$ (by the first bullet point). Also, $||\mathbf{w}||^2 \leq MR^2$, or equivalently, $||\mathbf{w}|| \leq R\sqrt{M}$ (by the second).

Finally, we use the fact that $\mathbf{w} \cdot \mathbf{w}^*/||\mathbf{w}^*|| \leq ||\mathbf{w}||$ which is just saying that the projection of $\mathbf{w}$ in the direction of $\mathbf{w}^*$ cannot be larger than the length of $\mathbf{w}$. This gives us:

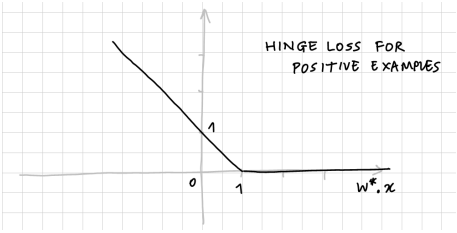
$$\begin{aligned} M/||\mathbf{w}^*|| &\leq R\sqrt{M} \\ \sqrt{M} &\leq R||\mathbf{w}^*|| \\ M &\leq R^2||\mathbf{w}^*||^2 \end{aligned}$$

as desired. ■

2.4 Extensions and hinge-loss

We assumed above that there existed a perfect $\mathbf{w}^*$ that correctly classified all the examples, i.e., correctly classified all the emails into important versus non-important. This is rarely the case in real-life data. What if $\mathbf{w}^*$ isn't quite perfect? We can see what this does to the above proof: if there is an example that $\mathbf{w}^*$ doesn't correctly classify, then while the second part of the proof still holds, the first part (the dot product of $\mathbf{w}$ with $\mathbf{w}^*$ increasing) breaks down. However, if this doesn't happen too often, and also $\mathbf{a}_i \cdot \mathbf{w}^*$ is just a "little bit wrong" then this just means we will make a few more mistakes.

To make this formal, define the *hinge-loss* of $\mathbf{w}^*$ on a positive example $\mathbf{a}_i$ as $\max(0, 1 - \mathbf{a}_i \cdot \mathbf{w}^*)$. In other words, if $\mathbf{a}_i \cdot \mathbf{w}^* \geq 1$ as desired then the hinge-loss is zero; else, the hinge-loss is the amount the LHS is less than the RHS.[2]



Similarly, the hinge-loss of $\mathbf{w}^*$ on a negative example $\mathbf{a}_i$ is $\max(0, 1 + \mathbf{a}_i \cdot \mathbf{w}^*)$. Given a sequence of labeled examples A, define the total hinge-loss $L_{hinge}(\mathbf{w}^*, A)$ as the sum of hinge-losses of $\mathbf{w}^*$ on all examples in A. We now get the following extended theorem.

Theorem 2 *On any sequence of examples $A = \mathbf{a}_1, \mathbf{a}_2, \ldots$, the Perceptron algorithm makes at most*

$$\min_{\mathbf{w}^*} \left(R^2||\mathbf{w}^*||^2 + 2L_{hinge}(\mathbf{w}^*, A) \right)$$

mistakes, where $R = \max_i ||\mathbf{a}_i||$.

Proof: As before, each update of the Perceptron algorithm increases $||\mathbf{w}||^2$ by at most R^2, so if the algorithm makes M mistakes, we have $||\mathbf{w}||^2 \leq MR^2$.

What we can no longer say is that each update of the algorithm increases $\mathbf{w} \cdot \mathbf{w}^*$ by at least 1. Instead, on a positive example we are "increasing" $\mathbf{w} \cdot \mathbf{w}^*$ by $\mathbf{a}_i \cdot \mathbf{w}^*$ (it could be negative), which is at least $1 - L_{hinge}(\mathbf{w}^*, \mathbf{a}_i)$. Similarly, on a negative example we "increase" $\mathbf{w} \cdot \mathbf{w}^*$ by $-\mathbf{a}_i \cdot \mathbf{w}^*$, which is also at least $1 - L_{hinge}(\mathbf{w}^*, \mathbf{a}_i)$. If we sum this up over all mistakes, we get that at the end we have $\mathbf{w} \cdot \mathbf{w}^* \geq M - L_{hinge}(\mathbf{w}^*, A)$, where we are using here the fact that hinge-loss is never negative so summing over all of A is only larger than summing over the mistakes that $\mathbf{w}$ made.

[2]This is called "hinge-loss" because as a function of $\mathbf{a}_i \cdot \mathbf{w}^*$ it looks like a hinge.

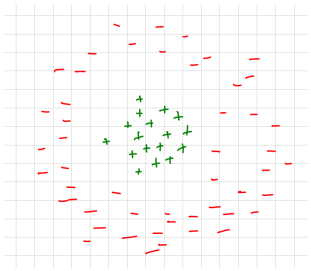
Finally, we just do some algebra. Let $L = L_{hinge}(\mathbf{w}^*, A)$. If $M \leq L$, then the theorem is clearly true. Else we can assume that $M > L$, and hence $0 < M - L \leq \mathbf{w} \cdot \mathbf{w}^*$. Now we have:

$$\begin{aligned}
\mathbf{w} \cdot \mathbf{w}^* / ||\mathbf{w}^*|| &\leq ||\mathbf{w}|| \\
\Rightarrow (\mathbf{w} \cdot \mathbf{w}^*)^2 &\leq ||\mathbf{w}||^2 ||\mathbf{w}^*||^2 \\
\Rightarrow (M - L)^2 &\leq M R^2 ||\mathbf{w}^*||^2 \\
\Rightarrow M^2 - 2ML + L^2 &\leq M R^2 ||\mathbf{w}^*||^2 \\
\Rightarrow M - 2L + L^2/M &\leq R^2 ||\mathbf{w}^*||^2 \\
\Rightarrow M &\leq R^2 ||\mathbf{w}^*||^2 + 2L - L^2/M \;\leq\; R^2 ||\mathbf{w}^*||^2 + 2L
\end{aligned}$$

as desired. ■

2.5 Kernel functions

What if even the best $\mathbf{w}^*$ has high hinge-loss? E.g., maybe instead of a linear separator decision boundary, the boundary between important emails and unimportant emails looks more like a circle?



A neat idea for addressing situations like this is to use what are called *kernel functions*, or sometimes the "kernel trick". Here is the idea. Suppose you have a function K (called a "kernel") over pairs of data points such that for some (doesn't even have to be known) function $\phi : \mathbb{R}^n \to \mathbb{R}^N$ (where perhaps $N \gg n$) we have $K(\mathbf{a}_i, \mathbf{a}_j) = \phi(\mathbf{a}_i) \cdot \phi(\mathbf{a}_j)$.

In that case, if we can write the Perceptron algorithm so that it only interacts with the data via dot products, and then replace every dot-product with an invocation of K, then we can act as if we had performed the function ϕ explicitly without having to actually compute ϕ. For example, consider $K(\mathbf{a}_i, \mathbf{a}_j) = (1 + \mathbf{a}_i \cdot \mathbf{a}_j)^d$. It turns out this corresponds to a mapping ϕ into a space of dimension $N \approx n^d$ (try doing it with $d = 2$), and perhaps in this higher-dimensional space there is a $\mathbf{w}^*$ such that the bound of Theorem 2 is small. But the nice thing is we didn't have to computationally perform the mapping ϕ!

So, how can we view the Perceptron algorithm as only interacting with data via dot-products? Notice that $\mathbf{w}$ is always a linear combination of data points, e.g., we might have $\mathbf{w} = \mathbf{a}_1 + \mathbf{a}_2 - \mathbf{a}_5$. So if we keep track of it this way, and need to predict on a new example $\mathbf{a}_6$, we can write $\mathbf{w} \cdot \mathbf{a}_6 = \mathbf{a}_1 \cdot \mathbf{a}_6 + \mathbf{a}_2 \cdot \mathbf{a}_6 - \mathbf{a}_5 \cdot \mathbf{a}_6$. So if we just replace each of these dot-products with "K", we are running the algorithm as if we had explicitly performed the ϕ mapping. This is called "kernelizing" the algorithm.

3 Other Notes

3.1 Perceptron's Worst-Case Runtime

The runtime of Perceptron may be exponential in the number of bits required to write down the input. Here's an example:

(Take the powerpoint example from today's lecture. The positive points are at $(1, 1)$, $(1, 0)$, and a negative point at $(0, 1)$.) Now add in another positive point at $(1/K, 1)$ for some large integer $K > 0$. Note that the length of each vector $\mathbf{a}_i$ is no more than $\sqrt{2}$ so the R^2 term is only 2.

Moreover, the total length of the input is $O(\log K)$ bits. This is because the number of bits to write down a fraction $1/K$ is only $O(\log K)$. And numbers 0 and 1 take $O(1)$ bits.

But if you run the Perceptron algorithm, you'll see that you will run for at least $\Omega(K)$ steps, maybe even more, before you find a linear separator for this data. And $\Omega(K)$ is exponential in the length of the input.

3.2 Affine Linear Separators

Suppose we want a linear separator that did not necessarily pass through the origin. I.e., we want a vector $\mathbf{w}^*$ and real b, such that $\mathbf{w}^* \cdot \mathbf{a}_i > b$ for the positive examples and $\mathbf{w}^* \cdot \mathbf{a}_i < b$ for the negative ones. Call the pair $(\mathbf{w}^*, b)$ an affine separator. Here's an easy way to do this using Perceptron.

Add a new coordinate to each data point $\mathbf{a}_i \in \mathbb{R}^n$ and put value 1 in it—call the new "lifted" points $\mathbf{a}'_i \in \mathbb{R}^{n+1}$. Then for any $\mathbf{w}^*, b$ as above, you can define vector $\mathbf{v}^* \in \mathbb{R}^{n+1}$ to be the same as $\mathbf{w}^*$ in the first n coordinates, and with value $-b$ in the $n + 1^{st}$ coordinate. Observe $\mathbf{v}^* \cdot \mathbf{a}'_i = \mathbf{w}^* \cdot \mathbf{a}_i - b$ which is > 0 for the positive examples and < 0 for the negative ones. So there exists a linear separator $\mathbf{v}^*$ for the lifted points if and only if there exists an affine separator for the original points.

Now, you could just run Perceptron on the lifted points, find the vector $\mathbf{v} = (v_1, v_2, \ldots, v_{n+1}) \in \mathbb{R}^{n+1}$, and be able to infer that $\mathbf{w} = (v_1, \ldots, v_n), b = -v_{n+1}$ is a good affine separator.

In the past few lectures we have looked at increasingly more expressive problems solvable 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
- Formal definitions: decision problems, **P** and **NP**.
- Circuit-SAT and 3-SAT
- Examples of showing **NP**-completeness.

1 Reductions and Expressiveness

In the last few lectures we 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 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.

> **Example:** Think about why 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 2 *A problem A is* **poly-time reducible** *to problem B (written as $A \leq_p B$) if we can solve problem A in polynomial time given a polynomial time black-box algorithm for problem B.*[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).

1.1 Decision Problems and Karp Reductions

We consider *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 split all instances into two categories: YES-instances (whose correct answer is YES)

[1] You can loosely think of $A \leq_p B$ as saying "A is no harder than B, up to polynomial factors."

and NO-instances (whose correct answer is NO). We put any ill-formed instances into the NO category.

In this lecture, we seek reductions (called *Karp reductions*) that are of a special form:

Many-one reduction (a.k.a. Karp reduction) from problem A to problem B: To reduce problem A to problem B we want a function f that maps 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 $f(x)$. [2]

2 Definitions: P, NP, and NP-Completeness

We can now define the complexity classes **P** and **NP**. These are both classes of decision problems.

Definition 3 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**.

But there are other problems we don't know how to efficiently solve. Some of these problems may be really uncomputable (like the HALTING PROBLEM that you probably saw in 15-251). Others, like the TRAVELING SALESMAN PROBLEM, have algorithms that run in $2^{O(n)}$ time. Yet others, like FACTORING, have algorithms that run in $2^{O(n^{1/3})}$ time. How to refine the landscape of these problems, to make sense of it all?

Here's one way. 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 don't know of any polynomial-time algorithms for solving the problem but we could easily check a proposed solution if someone gave one to us. The class of problems of this type — namely, if the answer is YES, then there exists a polynomial-length proof that can be checked in polynomial time — is called **NP**.

Definition 4 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, $V(I, X) = NO$.*

[2] Why Karp reductions? Why not reductions that, e.g., map YES-instances of A to NO-instances of B? Or solve two instances of B and use that answer to solve an instance of A? Two reasons. Firstly, Karp reductions give a stronger result. Secondly, using general reductions (called Turing reductions) no longer allows us to differentiate between **NP** and co-**NP**, say; see Section 8 for a discussion.

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.

Loosely speaking, **NP**-complete problems are the "hardest" problems in **NP**, if you can solve them in polynomial time then you can solve any other problem in **NP** in polynomial time. Formally,

Definition 5 (NP-complete) *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 Definition 5, then it's called **NP**-hard.

3 Circuit-SAT and 3-SAT

The definition of **NP**-completeness would be useless if there were no **NP**-complete problems. Thankfully that is not the case. Let's consider now what would be a problem *so expressive* that if we could solve it, we could solve any problem in **NP**. Moreover, we must define the problem so that it is in **NP** itself. Here is a natural candidate: CIRCUIT-SAT.

Definition 6 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 7 *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 reduce any problem in NP to CIRCUIT-SAT. By definition, this problem has a verifier program V that given an instance I and a witness W correctly outputs YES/NO in $b := p(|I|)$ time for a fixed polynomial $p()$. We can assume it only uses b bits of memory too. We 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. ■

We now have one **NP**-complete problem. And it looks complicated. However, now we will show that a much simpler-looking problem, 3-SAT has the property that CIRCUIT-SAT $\leq_p$ 3-SAT.

Definition 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.*

Theorem 9 *CIRCUIT-SAT $\leq_p$ 3-SAT. Hence 3-SAT is* **NP***-complete.*

Proof: First of all, 3-SAT is clearly in **NP**, again you can guess the input and try it. Now, we give a Karp reduction from Circuit-SAT to it: i.e., a function f from 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:

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

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. ■

4 Search versus Decision

Technically, a polynomial-time algorithm for CIRCUIT-SAT or 3-SAT just tells us if a solution exists, but doesn't actually produce it. How could we use an algorithm that just answers the YES/NO question of CIRCUIT-SAT to actually find a satisfying assignment? If we can do this, then we can use it to actually *find* the coloring or *find* the tour, not just smugly tell us that there is one. The problem of actually finding a solution is often called the *search* version of the problem, as opposed to the *decision* version that just asks whether or not the solution exists. That is, we are asking: can we reduce the search version of the CIRCUIT-SAT to the decision version?

It turns out that in fact we can, by essentially performing binary search. In particular, once we know that a solution x exists, we want to ask: "how about a solution whose first bit is 0?" If, say, the answer to that is YES, then we will ask: "how about a solution whose first two bits are 00?" If, say, the answer to that is NO (so there must exist a solution whose first two bits are 01) we will then ask: "how about a solution whose first three bits are 010?" And so on. The key point is that we can do this using a black-box algorithm for the decision version of CIRCUIT-SAT as follows: we can just set the first few inputs of the circuit to whatever we want, and feed the resulting circuit to the algorithm. This way, using at most n calls to the decision algorithm, we can solve the search problem too.

5 On to Other Problems: CLIQUE

We now use the **NP**-completeness of 3-SAT to show that another problem, a natural graph problem called CLIQUE is **NP**-complete.

Definition 10 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 $\geq k$?"*

Theorem 11 CLIQUE *is* **NP***-Complete.*

Proof: Note that CLIQUE is clearly in **NP**; the witness is the set of vertices that are all connected to each other via edges. Next, we reduce 3-SAT to 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:

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

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. We claim that the maximum size is m if and only if the original formula has a satisfying assignment.

Suppose 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).

For the other direction, we can either show that if there *isn't* a satisfying assignment to F then the maximum clique in the graph has size $< m$, or argue the contrapositive and show that if there is a m-clique in the graph, then there is a satisfying assignment for the formula. 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.

Finally, to complete the proof, we note that 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 CLIQUE is **NP**-complete. ■

5.1 Independent Set and Vertex Cover

Now that we know that CLIQUE is **NP**-complete, we can use that to show other problems are **NP**-complete.

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 12 Independent Set *is* **NP***-complete.*

Proof: We reduce from Clique. Given an instance (G, k) of the 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 13 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. ■

6 Summary: Proving NP-completeness in 2 Easy Steps

If you want to prove that problem Q is NP-complete, you need to do two things:

1. Show that Q is in NP.
2. Choose some NP-hard problem P to reduce from. This problem could be 3-SAT or Clique or Independent Set or Vertex Cover or any of the zillions of NP-hard problems known. Most of the time in this course we will suggest a problem to reduce from (but you can choose another one if you like). In the real world you will have to figure out this problem P yourself.

   Now you want to reduce **from** P **to** Q. In other words, given any instance I of P, show how to transform it into an instance $f(I)$ of Q, such that

   $$I \text{ is a YES-instance of } P \iff f(I) \text{ is a YES-instance of } Q.$$

   Note the " $\iff$ " in the middle—*you need to show both directions.* You also need to show that the mapping $f(\cdot)$ can be done in polynomial time (and hence $f(I)$ has size polynomial in the size of the original instance I).

A common mistake is reducing from the problem Q to the hard problem P. Think about what this means. It means you can model your problem as a hard problem. Just because you can model the problem of adding two numbers as a linear program or as 3-SAT does not make addition complicated. You want to reduce the hard problem P to your problem Q, this shows Q is "at least as hard" as P.

7 Bonus: A Non-Trivial Proof of Membership in NP*

Most of the time the proofs of a problem belonging to **NP** are trivial: you can use the solution as the witness X. Here's a non-trivial example that we alluded to in lecture, a proof that Primes is in **NP**. Recall that Primes is the decision problem: given a number N, is it prime? To show this

is in **NP**, we need to show a poly-time verifier algorithm $V(N, X)$ that N is a prime if and only if there exists a short witness X which will make the verifier say YES.

Today we know that PRIMES is in **P**, so we could just use that algorithm as a verifier. In fact we could use the fact that testing primality has a randomized algorithm with one-sided error to also show that PRIMES is in **NP**. But those result are somewhat advanced, can we use something simpler?

Here is a proof due to Vaughan Pratt[3] from 1975 that uses basic number theory. He uses the following theorem of Édouard Lucas[4] which we present without proof:

Theorem 14 *A number p is a prime if and only if there exists some $g \in \{0, 1, \ldots, p-1\}$ such that*

$$g^{p-1} \equiv 1 \text{ mod } p \qquad \text{and} \qquad g^{(p-1)/q} \not\equiv 1 \text{ mod } p \quad \text{for all primes } q|(p-1).$$

Great. So if N was indeed a prime, the witness could be this number g corresponding to N that Lucas promises. We could check for that g to the appropriate powers was either equivalent to 1 or not. (By repeating squaring we can compute those powers in time $O(\log N)$.)

Hmm, we need to check the condition for all primes q that divide $N-1$. No problem: we can write down the prime factorization of $N-1$ as part of the witness. It can say: $N-1 = q_1 \cdot q_2 \cdot \ldots \cdot q_k$. Note that there are at most $\log_2(N-1)$ many distinct primes in this list (since each of them is at least 2), and each of them takes $O(\log N)$ bits to write down. And what about their primality? This is the clever part: we recursively write down witnesses for their primality. (The base case is 3 or smaller, then we stop.)

How long is this witness? Let's just look at the number of numbers we write down, each number will be $O(\log N)$ bits.

Note we wrote down g, and then k numbers q_i. That's a total of $k+1$ numbers. And then we recurse. Say each q_i required $c(\log_2 q_i) - 2$ numbers to write down a witness of primality. Then we need to ensure that

$$(k+1) + \sum_{i=1}^{k} (c \log_2 q_i - 3) \leq c \log_2 N - 3$$

But $\sum_i^k \log_2 q_i = \log_2(N-1)$. So we get that the LHS is $(k+1) + c\log_2(N-1) - 3k = c\log_2(N-1) - 2k + 1$. And finally, we use the fact that $N-1$ cannot be prime if N is (except for $N=3$), so $k \geq 2$ and thus $c\log_2(N-1) - 2k + 1 \leq c\log_2 N - 3$. Finally, looking at the base case shows that $c \geq 4$ suffices.

To summarize, the witness used at most $O(\log N)$ numbers, each of $O(\log N)$ bits. This completes the proof that PRIMES is in **NP**.

8 Bonus: Co-NP*

Just like we defined NP as the class of problems for which there were short proofs for YES-instances, we can define a class of problems for which there are short proofs/witnesses for NO-instances. Specifically, $Q \in$ **co-NP** if there exists a verifier $V(I, X)$ such that:

[3] Computer Science Professor at Stanford. He was one of the inventors of the deterministic linear-time median-finding algorithm, and also the Knuth-Morris-Pratt string matching algorithm. Also designed the logo for Sun Microsystems.

[4] French mathematician (1842-1891), worked on number theory, and on Fibonacci numbers and the Lucas numbers named after him. Apparently invented the Tower of Hanoi puzzle (or at least popularized it). Died when cut by a piece of glass from a broken plate at a banquet.

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

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 $C_1(x) \neq C_2(x)$). Or the problem CO-CLIQUE: "Given a graph G and a number K, is every clique in the graph of size at least K?" If the answer is NO, there is a short witness (a clique in G with fewer than K vertices). CO-3-SAT: "Given a 3-CNF formula, does it have no satisfying assignments?" If the answer is NO, there is a short witness (a satisfying assignment).

It is commonly believed that **NP** does not equal **co-NP**. Note that if **P** = **NP**, then **NP** = **co-NP**, but the other implication is not known to be true. Again, one can define **co-NP**-complete problems. This is there using Karp reductions (as opposed to Turing reductions) becomes important.

In the past few lectures we have looked at increasingly more expressive problems solvable 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
- Formal definitions: decision problems, **P** and **NP**.
- Circuit-SAT and 3-SAT
- Examples of showing **NP**-completeness.

1 Reductions and Expressiveness

In the last few lectures we 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 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.

> **Example:** Think about why 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 2 *A problem A is* **poly-time reducible** *to problem B (written as $A \leq_p B$) if we can solve problem A in polynomial time given a polynomial time black-box algorithm for problem B.*[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).

1.1 Decision Problems and Karp Reductions

We consider *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 split all instances into two categories: YES-instances (whose correct answer is YES)

[1] You can loosely think of $A \leq_p B$ as saying "A is no harder than B, up to polynomial factors."

and NO-instances (whose correct answer is NO). We put any ill-formed instances into the NO category.

In this lecture, we seek reductions (called *Karp reductions*) that are of a special form:

Many-one reduction (a.k.a. Karp reduction) from problem A to problem B: To reduce problem A to problem B we want a function f that maps 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 $f(x)$. [2]

2 Definitions: P, NP, and NP-Completeness

We can now define the complexity classes **P** and **NP**. These are both classes of decision problems.

Definition 3 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**.

But there are other problems we don't know how to efficiently solve. Some of these problems may be really uncomputable (like the HALTING PROBLEM that you probably saw in 15-251). Others, like the TRAVELING SALESMAN PROBLEM, have algorithms that run in $2^{O(n)}$ time. Yet others, like FACTORING, have algorithms that run in $2^{O(n^{1/3})}$ time. How to refine the landscape of these problems, to make sense of it all?

Here's one way. 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 don't know of any polynomial-time algorithms for solving the problem but we could easily check a proposed solution if someone gave one to us. The class of problems of this type — namely, if the answer is YES, then there exists a polynomial-length proof that can be checked in polynomial time — is called **NP**.

Definition 4 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, $V(I, X) = NO$.*

[2] Why Karp reductions? Why not reductions that, e.g., map YES-instances of A to NO-instances of B? Or solve two instances of B and use that answer to solve an instance of A? Two reasons. Firstly, Karp reductions give a stronger result. Secondly, using general reductions (called Turing reductions) no longer allows us to differentiate between **NP** and co-**NP**, say; see Section 8 for a discussion.

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.

Loosely speaking, **NP**-complete problems are the "hardest" problems in **NP**, if you can solve them in polynomial time then you can solve any other problem in **NP** in polynomial time. Formally,

Definition 5 (NP-complete) *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 Definition 5, then it's called **NP**-hard.

3 Circuit-SAT and 3-SAT

The definition of **NP**-completeness would be useless if there were no **NP**-complete problems. Thankfully that is not the case. Let's consider now what would be a problem *so expressive* that if we could solve it, we could solve any problem in **NP**. Moreover, we must define the problem so that it is in **NP** itself. Here is a natural candidate: CIRCUIT-SAT.

Definition 6 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 7 *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 reduce any problem in NP to CIRCUIT-SAT. By definition, this problem has a verifier program V that given an instance I and a witness W correctly outputs YES/NO in $b := p(|I|)$ time for a fixed polynomial $p()$. We can assume it only uses b bits of memory too. We 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. ■

We now have one **NP**-complete problem. And it looks complicated. However, now we will show that a much simpler-looking problem, 3-SAT has the property that CIRCUIT-SAT $\leq_p$ 3-SAT.

Definition 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.*

Theorem 9 *CIRCUIT-SAT $\leq_p$ 3-SAT. Hence 3-SAT is* **NP***-complete.*

Proof: First of all, 3-SAT is clearly in **NP**, again you can guess the input and try it. Now, we give a Karp reduction from Circuit-SAT to it: i.e., a function f from 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:

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

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. ■

4 Search versus Decision

Technically, a polynomial-time algorithm for CIRCUIT-SAT or 3-SAT just tells us if a solution exists, but doesn't actually produce it. How could we use an algorithm that just answers the YES/NO question of CIRCUIT-SAT to actually find a satisfying assignment? If we can do this, then we can use it to actually *find* the coloring or *find* the tour, not just smugly tell us that there is one. The problem of actually finding a solution is often called the *search* version of the problem, as opposed to the *decision* version that just asks whether or not the solution exists. That is, we are asking: can we reduce the search version of the CIRCUIT-SAT to the decision version?

It turns out that in fact we can, by essentially performing binary search. In particular, once we know that a solution x exists, we want to ask: "how about a solution whose first bit is 0?" If, say, the answer to that is YES, then we will ask: "how about a solution whose first two bits are 00?" If, say, the answer to that is NO (so there must exist a solution whose first two bits are 01) we will then ask: "how about a solution whose first three bits are 010?" And so on. The key point is that we can do this using a black-box algorithm for the decision version of CIRCUIT-SAT as follows: we can just set the first few inputs of the circuit to whatever we want, and feed the resulting circuit to the algorithm. This way, using at most n calls to the decision algorithm, we can solve the search problem too.

5 On to Other Problems: CLIQUE

We now use the **NP**-completeness of 3-SAT to show that another problem, a natural graph problem called CLIQUE is **NP**-complete.

Definition 10 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 $\geq k$?"*

Theorem 11 CLIQUE *is* **NP***-Complete.*

Proof: Note that CLIQUE is clearly in **NP**; the witness is the set of vertices that are all connected to each other via edges. Next, we reduce 3-SAT to 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:

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

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. We claim that the maximum size is m if and only if the original formula has a satisfying assignment.

Suppose 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).

For the other direction, we can either show that if there *isn't* a satisfying assignment to F then the maximum clique in the graph has size $< m$, or argue the contrapositive and show that if there is a m-clique in the graph, then there is a satisfying assignment for the formula. 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.

Finally, to complete the proof, we note that 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 CLIQUE is **NP**-complete. ■

5.1 Independent Set and Vertex Cover

Now that we know that CLIQUE is **NP**-complete, we can use that to show other problems are **NP**-complete.

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 12 Independent Set *is* **NP***-complete.*

Proof: We reduce from Clique. Given an instance (G, k) of the 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 13 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. ■

6 Summary: Proving NP-completeness in 2 Easy Steps

If you want to prove that problem Q is NP-complete, you need to do two things:

1. Show that Q is in NP.
2. Choose some NP-hard problem P to reduce from. This problem could be 3-SAT or Clique or Independent Set or Vertex Cover or any of the zillions of NP-hard problems known. Most of the time in this course we will suggest a problem to reduce from (but you can choose another one if you like). In the real world you will have to figure out this problem P yourself.

   Now you want to reduce **from** P **to** Q. In other words, given any instance I of P, show how to transform it into an instance $f(I)$ of Q, such that

   $$I \text{ is a YES-instance of } P \iff f(I) \text{ is a YES-instance of } Q.$$

   Note the " $\iff$ " in the middle—*you need to show both directions.* You also need to show that the mapping $f(\cdot)$ can be done in polynomial time (and hence $f(I)$ has size polynomial in the size of the original instance I).

A common mistake is reducing from the problem Q to the hard problem P. Think about what this means. It means you can model your problem as a hard problem. Just because you can model the problem of adding two numbers as a linear program or as 3-SAT does not make addition complicated. You want to reduce the hard problem P to your problem Q, this shows Q is "at least as hard" as P.

7 Bonus: A Non-Trivial Proof of Membership in NP*

Most of the time the proofs of a problem belonging to **NP** are trivial: you can use the solution as the witness X. Here's a non-trivial example that we alluded to in lecture, a proof that Primes is in **NP**. Recall that Primes is the decision problem: given a number N, is it prime? To show this

is in **NP**, we need to show a poly-time verifier algorithm $V(N, X)$ that N is a prime if and only if there exists a short witness X which will make the verifier say YES.

Today we know that PRIMES is in **P**, so we could just use that algorithm as a verifier. In fact we could use the fact that testing primality has a randomized algorithm with one-sided error to also show that PRIMES is in **NP**. But those result are somewhat advanced, can we use something simpler?

Here is a proof due to Vaughan Pratt[3] from 1975 that uses basic number theory. He uses the following theorem of Édouard Lucas[4] which we present without proof:

Theorem 14 *A number p is a prime if and only if there exists some $g \in \{0, 1, \ldots, p-1\}$ such that*

$$g^{p-1} \equiv 1 \text{ mod } p \qquad \text{and} \qquad g^{(p-1)/q} \not\equiv 1 \text{ mod } p \quad \text{for all primes } q|(p-1).$$

Great. So if N was indeed a prime, the witness could be this number g corresponding to N that Lucas promises. We could check for that g to the appropriate powers was either equivalent to 1 or not. (By repeating squaring we can compute those powers in time $O(\log N)$.)

Hmm, we need to check the condition for all primes q that divide $N-1$. No problem: we can write down the prime factorization of $N-1$ as part of the witness. It can say: $N-1 = q_1 \cdot q_2 \cdot \ldots \cdot q_k$. Note that there are at most $\log_2(N-1)$ many distinct primes in this list (since each of them is at least 2), and each of them takes $O(\log N)$ bits to write down. And what about their primality? This is the clever part: we recursively write down witnesses for their primality. (The base case is 3 or smaller, then we stop.)

How long is this witness? Let's just look at the number of numbers we write down, each number will be $O(\log N)$ bits.

Note we wrote down g, and then k numbers q_i. That's a total of $k+1$ numbers. And then we recurse. Say each q_i required $c(\log_2 q_i) - 2$ numbers to write down a witness of primality. Then we need to ensure that

$$(k+1) + \sum_{i=1}^{k} (c \log_2 q_i - 3) \leq c \log_2 N - 3$$

But $\sum_i^k \log_2 q_i = \log_2(N-1)$. So we get that the LHS is $(k+1) + c\log_2(N-1) - 3k = c\log_2(N-1) - 2k + 1$. And finally, we use the fact that $N-1$ cannot be prime if N is (except for $N=3$), so $k \geq 2$ and thus $c\log_2(N-1) - 2k + 1 \leq c\log_2 N - 3$. Finally, looking at the base case shows that $c \geq 4$ suffices.

To summarize, the witness used at most $O(\log N)$ numbers, each of $O(\log N)$ bits. This completes the proof that PRIMES is in **NP**.

8 Bonus: Co-NP*

Just like we defined NP as the class of problems for which there were short proofs for YES-instances, we can define a class of problems for which there are short proofs/witnesses for NO-instances. Specifically, $Q \in$ **co-NP** if there exists a verifier $V(I, X)$ such that:

[3] Computer Science Professor at Stanford. He was one of the inventors of the deterministic linear-time median-finding algorithm, and also the Knuth-Morris-Pratt string matching algorithm. Also designed the logo for Sun Microsystems.

[4] French mathematician (1842-1891), worked on number theory, and on Fibonacci numbers and the Lucas numbers named after him. Apparently invented the Tower of Hanoi puzzle (or at least popularized it). Died when cut by a piece of glass from a broken plate at a banquet.

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

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 $C_1(x) \neq C_2(x)$). Or the problem CO-CLIQUE: "Given a graph G and a number K, is every clique in the graph of size at least K?" If the answer is NO, there is a short witness (a clique in G with fewer than K vertices). CO-3-SAT: "Given a 3-CNF formula, does it have no satisfying assignments?" If the answer is NO, there is a short witness (a satisfying assignment).

It is commonly believed that **NP** does not equal **co-NP**. Note that if **P** = **NP**, then **NP** = **co-NP**, but the other implication is not known to be true. Again, one can define **co-NP**-complete problems. This is there using Karp reductions (as opposed to Turing reductions) becomes important.

While we have good algorithms for many optimization problems, the previous lecture showed that many others we'd like to solve are **NP**-hard. What do we do? Suppose we are given an **NP**-complete problem to solve. Assuming $\mathbf{P} \neq \mathbf{NP}$, we can't hope for a polynomial-time algorithm for these problems. But can we

(a) get polynomial-time algorithms that always produce a "pretty good" solution? (A.k.a. *approximation algorithms*)
(b) get faster (though still exponential-time) algorithms than the naïve solutions?

Today we consider approaches to combat intractability for several problems. Specific topics in this lecture include:

- 2-approximation (and better) for scheduling to minimize makespan.
- 2-approximation for vertex cover via greedy matchings.
- 2-approximation for vertex cover via LP rounding.
- Greedy $O(\log n)$ approximation for set-cover.
- A faster-than-$O(n^k)$ time algorithm for exact vertex cover when k is small. (optional)

1 Introduction

Given an **NP**-hard problem, we don't hope for a fast algorithm that always gets the optimal solution. But we can't just throw our hands in the air and say "We can't do anything!" Or even, "We can't do anything other than the trivial solution!". There are (at least) two ways of being smarter.

- First approach: find a poly-time algorithm that guarantees to get at least a "pretty good" solution? E.g., can we guarantee to find a solution that's within 10% of optimal? If not that, then how about within a factor of 2 of optimal? Or, anything non-trivial?

- Second approach: find a faster algorithm than the naïve one. There are a bunch of ideas that you can use for this, often related to dynamic programming — today we'll talk about one of them.[1]

As seen in the last two lectures, the class of **NP**-complete problems are all equivalent in the sense that a polynomial-time algorithm to solve any one of them would imply a polynomial-time algorithm to solve all of them (and, moreover, to solve any problem in **NP**). However, the difficulty of getting faster exact algorithms, or good approximations to these problems varies quite a bit. In this lecture we will examine some important **NP**-complete problems and look at to what extent we can achieve both goals above.

2 Scheduling Jobs to Minimize Load

Here's a scheduling problem. You have m identical machines on which you want to schedule some n jobs. Each job $j \in \{1, 2, \ldots, n\}$ has a processing time $p_j > 0$. You want to partition the jobs among the machines to minimize the load of the most-loaded machine. In other words, if S_i is the

[1] You already saw an example: in Lecture II on Dynamic programming, we gave an algorithm for TSP that ran in time $O(n^2 2^n)$, which is much faster than the trivial $O(n \cdot n!) \geq n^{n/2}$ time algorithm.

set of jobs assigned to machine i, define the *makespan* of the solution to be $\max_{\text{machines } i}(\sum_{j \in S_i} p_j)$. You want to minimize the makespan of the solution you output.

Approach #1: Greedy. Pick any unassigned job, and assign it to the machine with the least current load.

Theorem 1 *The greedy approach outputs a solution with makespan at most* 2 *times the optimum.*

Proof: Let us first get some bounds on the optimal value OPT. Clearly, the optimal makespan cannot be lower than the average load $\frac{1}{m}\sum_j p_j$. And also, the optimal makespan must be at least the largest job $\max_j p_j$.

Now look at our makespan. Suppose the most-loaded machine is i^*. Look at the last job we assigned on i^*, call this job j^*. If the load on i^* was L just before j^* was assigned to it, the makespan of our solution is $L + p_{j^*}$. By the greedy rule, i^* had the least load among all machines at this point, so L must be at most $\frac{1}{m}\sum_{j \text{ assigned before } j^*} p_j$. Hence, also $L \leq \frac{1}{m}\sum_j p_j \leq OPT$. Also, $p_{j^*} \leq OPT$. So our solution has makespan $L + p_{j^*} \leq 2OPT$.

Being careful, we notice that $L \leq \frac{1}{m}\sum_{j \neq j^*} p_j \leq OPT - \frac{p_j^*}{m}$, and hence our makespan is at most $L + p_{j^*} \leq (2 - \frac{1}{m})OPT$, which is slightly better. ■

Is this analysis tight? Sadly, yes. Suppose we have $m(m-1)$ jobs of size 1, and 1 job of size m, and we schedule the small jobs before the large jobs. The greedy algorithm will spread the small jobs equally over all the machines, and then the large job will stick out, giving a makespan of $(m-1)+m$, whereas the right thing to do is to spread the small jobs over $m-1$ machines and get a makespan of m. The gap is $\frac{2m-1}{m} = (2 - \frac{1}{m})$, hence this analysis is tight.

Hmm. Can we get a better algorithm? The bad example suggests one such algorithm.

Approach #2: Sorted Greedy. Pick the *largest* unassigned job, and assign it to the machine with the least current load.

Theorem 2 *The sorted greedy approach outputs a solution with makespan at most* 1.5 *times the optimum.*

Proof: Again, suppose the most-loaded machine is i^*, the last job assigned to it is j^*, and the load on i^* just before j^* was assigned to it is L. So $OPT = L + p_{j^*}$.

Suppose $L = 0$, then we are optimal, since $p_{j^*} \leq OPT$, so assume that at least one job was already on i^* before j^* was assigned. This means that each machine had at least one job on it when we assigned j^*. But by the sorted property, each such job has size at least p_{j^*}. So there are $m+1$ jobs of size at least p_{j^*}, and by the pigeonhole principle, OPT is at least $2p_{j^*}$. In other words, $p_{j^*} \leq \frac{OPT}{2}$.

By the same arguments as above, we know that our makespan is $L + p_{j^*} \leq OPT + \frac{OPT}{2} = 1.5\,OPT$. (And you can show a slightly better bound of $(1.5 - \frac{1}{2m})OPT$.) ■

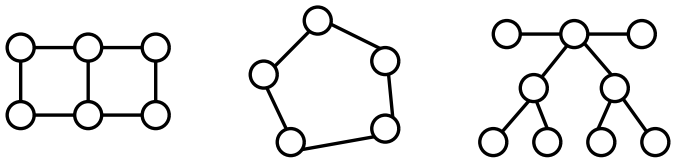
It is possible to show that the makespan of Sorted Greedy is at most $\frac{4}{3}OPT$. (Try it!)

3 Vertex Cover

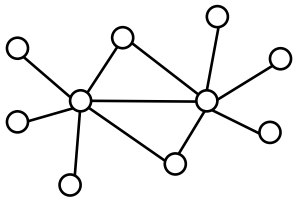
Recall that a *vertex cover* in a graph is a set of vertices such that every edge is incident to (touches) at least one of them. The vertex cover problem is to find the smallest such set of vertices.

Definition 3 VERTEX-COVER*: Given a graph G, find the smallest set of vertices such that every edge is incident to at least one of them. Decision problem: "Given G and integer k, does G contain a vertex cover of size $\leq k$?"*

For instance, this problem is like asking: what is the fewest number of guards we need to place in a museum in order to cover all the corridors.



Exercise: Find a vertex cover in the graphs above of size 3. Show that there is no vertex cover of size 2 in them.



Exercise: Find a vertex cover in the graph above of size 2. Show that there is no vertex cover of size 1 in this graph.

As we saw last time (via a reduction from INDEPENDENT SET), this problem is **NP**-hard.

3.1 Poly-time Algorithms that Output Approximate Solutions

OK, we don't expect to find the optimal solution in poly-time. However, any graph G we *can* get within a factor of 2 (or better). That is, if the graph G has a vertex cover of size k^*, we can return a vertex cover of size at most $2k^*$.

Let's start first, though, with some strategies that *don't* work.

Strawman Alg #1: Pick an arbitrary vertex with at least one uncovered edge incident to it, put it into the cover, and repeat.

What would be a bad example for this algorithm? [Answer: how about a star graph]

Strawman Alg #2: How about picking the vertex that covers the *most* uncovered edges. This is very natural, but unfortunately it turns out this doesn't work either, and it can produce a solution $\Omega(\log n)$ times larger than optimal.[2]

How can we get factor of 2? It turns out there are actually several ways. We will discuss here two quite different algorithms. Interestingly, while we have several algorithms for achieving a factor of 2, nobody knows if it is possible to efficiently achieve a factor 1.99.

[2]The bad examples for this algorithm are a bit more complicated however. One such example is as follows. Create a bipartite graph with a set S_L of t nodes on the left, and then a collection of sets $S_{R,1}, S_{R,2}, \ldots$ of nodes on the right, where set $S_{R,i}$ has $\lfloor t/i \rfloor$ nodes in it. So, overall there are $n = \Theta(t \log t)$ nodes. We now connect each set $S_{R,i}$ to S_L so that each node $v \in S_{R,i}$ has i neighbors in S_L and no two vertices in $S_{R,i}$ share any neighbors in common (we can do that since $S_{R,i}$ has at most t/i nodes). Now, the optimal vertex cover is simply the set S_L of size t, but this greedy algorithm might first choose $S_{R,t}$ then $S_{R,t-1}$, and so on down to $S_{R,1}$, finding a cover of total size $n - t$. Of course, the fact that the bad cases are complicated means this algorithm might not be so bad in practice.

Algorithm 1: Pick an arbitrary edge. We know any vertex cover must have at least 1 endpoint of it, so let's take *both* endpoints. Then, throw out all edges covered and repeat. Keep going until there are no uncovered edges left.

Theorem 4 *The above algorithm is a factor 2 approximation to* Vertex-Cover.

Proof: What Algorithm 1 finds in the end is a matching (a set of edges no two of which share an endpoint) that is "maximal" (meaning that you can't add any more edges to it and keep it a matching). This means if we take both endpoints of those edges. we must have a vertex cover. In particular, if the algorithm picked k edges, the vertex cover found has size $2k$. But, *any* vertex cover must have size at least k since it needs to have at least one endpoint of each of these edges, and since these edges don't touch, these are k *different* vertices. So the algorithm is a 2-approximation as desired. ■

Here is now another 2-approximation algorithm for Vertex Cover:

Algorithm 2: First, solve a *fractional* version of the problem. Have a variable x_i for each vertex with constraint $0 \leq x_i \leq 1$. Think of $x_i = 1$ as picking the vertex, and $x_i = 0$ as not picking it, and in-between as "partially picking it". Then for each edge (i, j), add the constraint that it should be covered in that we require $x_i + x_j \geq 1$. Then our goal is to minimize $\sum_i x_i$.

We can solve this using linear programming. This is called an "LP relaxation" because any true vertex cover is a feasible solution, but we've made the problem easier by allowing fractional solutions too. So, the value of the optimal solution now will be at least as good as the smallest vertex cover, maybe even better (i.e., smaller), but it just might not be legal any more. [Give examples of triangle-graph and star-graph]

Now that we have a super-optimal fractional solution, we need to somehow convert that into a legal integral solution. We can do that here by just picking each vertex i such that $x_i \geq 1/2$. This step is called *rounding* of the linear program (which literally is what we are doing here by rounding the fraction to the nearest integer — for other problems, the "rounding" step might not be so simple).

Theorem 5 *The above algorithm is a factor 2 approximation to* Vertex-Cover.

Proof: Claim 1: the output of the algorithm is a legal vertex cover. Why? [get at least 1 endpt of each edge]

Claim 2: The size of the vertex cover found is at most twice the size of the optimal vertex cover. Why? Let OPT_{frac} be the value of the optimal fractional solution, and let OPT_{VC} be the size of the smallest vertex cover. First, as we noted above, $OPT_{frac} \leq OPT_{VC}$. Second, our solution has cost at most $2 \cdot OPT_{frac}$ since it's no worse than doubling and rounding down. So, put together, our solution has cost at most $2 \cdot OPT_{VC}$. ■

Interesting fact: nobody knows any algorithm with approximation ratio 1.9. Best known is $2 - O(1/\sqrt{\log n})$, which is $2 - o(1)$.

3.2 Hardness of Approximation

There are results showing that a good-enough approximation algorithm will end up showing that P=NP. Clearly, a 1-approximation would find the exact vertex cover, and show this. Håstad showed that if you get a 7/6-approximation, you would prove P=NP. This 7/6 was improved to 1.361 by Dinur and Safra. Beating $2 - \varepsilon$ has been related to some other open problems (it is "unique games hard"), but is not known to be NP-hard.

4 Set Cover

The Set-Cover problem is defined as follows:

Definition 6 Set-Cover*: Given a domain X of n points, and m subsets $S_1, S_2, \ldots, S_m$ of these points. Goal: find the fewest number of these subsets needed to cover all the points. The decision problem also provides a number k and asks whether it is possible to cover all the points using k or fewer sets.*

Set-Cover is NP-Complete. However, there is a simple algorithm that gets an approximation ratio of $\ln n$ (i.e., that finds a cover using at most a factor $\ln n$ more sets than the optimal solution).

Greedy Algorithm (Set-Cover): Pick the set that covers the most points. Throw out all the points covered. Repeat.

What's an example where this algorithm *doesn't* find the best solution?

Theorem 7 *If the optimal solution uses k sets, the greedy algorithm finds a solution with at most $O(k \ln n)$ sets.*

Proof: Since the optimal solution uses k sets, there must some set that covers at least a $1/k$ fraction of the points. The algorithm chooses the set that covers the most points, so it covers at least that many. Therefore, after the first iteration of the algorithm, there are at most $n(1-1/k)$ points left. Again, since the optimal solution uses k sets, there must some set that covers at least a $1/k$ fraction of the remainder (if we got lucky we might have chosen one of the sets used by the optimal solution and so there are actually $k-1$ sets covering the remainder, but we can't count on that necessarily happening). So, again, since we choose the set that covers the most points remaining, after the second iteration, there are at most $n(1-1/k)^2$ points left. More generally, after t rounds, there are at most $n(1-1/k)^t$ points left. After $t = k \ln n$ rounds, there are at most $n(1-1/k)^{k \ln n} < n(1/e)^{\ln n} = 1$ points left, which means we must be done.[3] ■

Also, you can get a slightly better bound by using the fact that after $k \ln(n/k)$ rounds, there are at most $n(1/e)^{\ln(n/k)} = k$ points left, and (since each new set covers at least one point) you only need to go k more steps. This gives the somewhat better bound of $k \ln(n/k) + k$. So, we have:

Theorem 8 *If the optimal solution uses k sets, the greedy algorithm finds a solution with at most $k \ln(n/k) + k$ sets.*

For set cover, this may be about the best you can do. Dinur and Steurer (improving on results of Feige) showed that if you get a $(1-\epsilon)\ln(n)$-approximation algorithm for any constant $\epsilon > 0$, you will prove that P=NP.

5 Faster Exact Algorithms for Vertex Cover*

Let's see if we can solve the vertex cover problem faster than the obvious solution. To solve the decision version of the vertex cover problem, here's the trivial algorithm taking about n^k time.

Iterate over all subsets $S \subseteq V$ of size k.
    If S is a vertex cover, output `YES` and stop.
output `NO`.

[3] Notice how the above analysis is similar to the analysis we used of Edmonds-Karp #1.

How can you do better? Here's one cool way to do it:

Theorem 9 *Given $G = (V, E)$ and $k \leq n = |V|$, we can output (in time poly(n)) another graph H with at most $2k^2$ vertices, and an integer $k_H \leq k$ such that:*

$$(G, k) \text{ is a YES-instance } \iff (H, k_H) \text{ is a YES-instance .}$$

Since H is so much smaller, we can use the trivial algorithm above on (H, k') to run in time

$$|V(H)|^{k'} \leq (2k^2)^k \leq 2^{k(2 \lg k + 1)}.$$

Hence solve the problem on (G, k) in the time it takes to produce H (which is $poly(n)$), plus about $2^{2k \lg k}$ time. Much faster than n^k when $k \ll \sqrt{n}$.

It remains to prove Theorem 9.

Proof: Here's a simple observation: suppose G has an isolated vertex v, i.e., there are no edges hitting v. Then

$$(G, k) \text{ is a YES-instance } \iff (G - \{v\}, k) \text{ is a YES-instance .}$$

The second observation: suppose G has a vertex v of degree at least $k + 1$. Then v *must be* in any vertex cover of size k. Why? If not, these $k + 1$ edges incident to v would have to be covered by their other endpoints, which would require $k + 1$ vertices. So

$$(G, k) \text{ is a YES-instance } \iff (G - \{v\}, k - 1) \text{ is a YES-instance .}$$

(In this step, $G - \{v\}$ removes these edges incident to v as well.)

And the final observation: if G has maximum degree k and has a vertex cover of size at most k, G has at most k^2 edges. Why? Each of the vertices in the cover can cover at most k edges (their degree is at most k).

So now the construction of H is easy: start with G. Keep dropping isolated vertices (without changing k), or dropping high-degree vertices (and decrementing k) until you have some graph G' and parameter k'. By the first two observations,

$$(G, k) \text{ is a YES-instance } \iff (G', k') \text{ is a YES-instance .}$$

Now either G' has more than $(k')^2$ edges, then by the final observation (G', k') is a NO-instance, so then let H be a single edge and $k_H = 0$ (which is also a NO-instance.) Else return $H = G'$ and $k_H = k'$. Since H has $(k')^2 \leq k^2$ edges, it has at most $2k^2$ nodes.

Finally, the time to produce H? You can do all the above in linear time in G. ■

1 Prediction with Expert Advice

Today we'll study the problem of making predictions based on expert advice. There are n "experts" (who are just people with opinions, the quotes suggesting that they may or may not have any expertise in the matter). Each day the following sequence of events happens:

- We see the n experts' predictions of the outcome.
- We make our own prediction about the outcome.
- In parallel, the actual outcome is revealed.
- We are correct if we made the right prediction, and make a mistake otherwise.

This process goes on indefinitely. Our goal: at any time (say after some T days), if the best of the experts so far has made only few mistakes, we want to have made "not too many more" mistakes. We want to bound the number of mistakes, and hence this is called the "mistake-bound" model.

2 Warmup: Simple Strategies

To start off, say we have just "Up/Down" predictions. (If the market will go up or down, or if the weather will be fair or not.)

Suppose we know the best expert makes no mistakes. Can we hope to make only a few mistakes? Here's a strategy that makes only $\lceil \log_2 n \rceil$ mistakes. Just predict what the majority of the remaining experts predicts. When an expert is incorrect, discard him. So each time we make a mistake, we reduce by the number of experts by at least 1/2. This means after $\log_2 n$ mistakes we will be left with the perfect expert.

Suppose the best expert makes at most M mistakes on some sequence. Can we hope to make only a few mistakes? Here's a strategy that makes only $(M+1)(\log_2 n + 1)$ mistakes.[1] Run the above majority-and-halving strategy, but when you have discarded all the experts, bring them all back (call this the beginning of a new "phase"), and continue. Note that in each phase each expert makes at least one mistake, and you made $\log_2 n + 1$ mistakes. Hence, if the best expert makes only M mistakes, there would be at most M finished phases (plus the last unfinished one), and hence at most $(M+1)(\log_2 n + 1)$ mistakes in all.

> **Exercise:** Suppose the predictions belonged to some set of K items. (E.g., youx could say "cloudy", "sunny", "rain", "snow".) Give algorithms showing the bound of $O(M \log n)$ mistakes still holds.

3 Better: The Weighted Majority/Multiplicative Weights Algorithm

Throwing away an expert when he makes a mistake seems too drastic. Suppose we instead assign weights w_j to the experts, sum the weights of the expert saying "Up", sum the weights of the of

[1] Let's assume n is a power of 2.

the expert saying "Down", and predict the outcome with greater weight. Then once we see the outcome, we can reduce the weight of the experts who were wrong. In the above algorithm, we were zeroing out the weight, but suppose we are gentler?

The *(basic) deterministic weighted majority* algorithm does the following:

> Start with each expert having weight 1. Each time an expert makes a mistake, half his weight.

Remarkably, we can get a much stronger result now.

Theorem 1 *If on some sequence of days, the best expert makes M mistakes. The basic deterministic weighted majority algorithm makes $\leq 2.41(M + \log_2 n)$ mistakes.*

Proof: Let $\Phi := \sum_{i=1}^{n} w_i$ be the sum of weights of the n experts. Note that initially $\Phi = n$. Moreover, we claim that each time we make a mistake

$$\Phi_{new} \leq \frac{3}{4}\Phi_{old}.$$

Indeed, at least half the weight (which was making the majority prediction) gets halved (because it made a mistake), so we lose at least a quarter of the weight with each mistake. (Also if we don't make a mistake, Φ does not increase.) So if we've made m mistakes at some point, the total weight is at most

$$\Phi_{final} \leq (3/4)^m \cdot \Phi_{init} = (3/4)^m \cdot n.$$

Moreover, if the best expert i^* has made M mistakes, then $\Phi_{final} \geq w_{i^*} = (1/2)^M$. So

$$(1/2)^M \leq (3/4)^m \cdot n \quad \Rightarrow \quad (4/3)^m \leq n2^M.$$

Taking logs (base 2) and noting that $\frac{1}{\log_2(4/3)} = 2.41\ldots$ completes the proof. ■

This is pretty cool! If the best expert makes mistakes 10% of the time we err about 24% of the time (plus $O(\log n)$, but since this depends only on the number of experts, it will be a negligible fraction as $M \gg 2^n$. We can improve the 2.41 factor down to as close to 2 as we want, as the next exercise shows.

> **Exercise*:** Show that if we reduce the weight not by 1/2 but by $(1 - \epsilon)$ for some $\epsilon \leq 1/2$, then the number of mistakes is at most $2(1+\epsilon)M + O(\frac{\log n}{\epsilon})$. (Hint: check out the approximations we use in the proof of Theorem 2.)

However, you can show that any deterministic algorithm cannot make fewer than $2M$ mistakes. How? Two experts: one says "Up" all the time, the other "Down" all the time. Fix any prediction algorithm. Since this algorithm is deterministic, you know what prediction it will make on any day (given what happened on all previous days). You can now make the "real" outcome on this day be the opposite of the algorithm's prediction for this day. So algorithm makes mistake on all days. But one of the experts must be right at least 50% of the days.

4 Randomized Weighted Majority

Given the lower bound above for deterministic algorithms, randomization seems like a natural source of help. (At least that example would fail.) Let's define the *Randomized Weighted Majority* algorithm as follows:

> Start with unit weights. At each time, predict Up with probability $\frac{\sum_{j \text{ says Up}} w_j}{\sum_j w_j}$, and Down otherwise. And each time an expert makes a mistake, multiply its weight by $(1-\epsilon)$.

Note that the weights are pretty much like in the basic MW algorithm, just reduced more gently. (Moreover, note that the prediction we make does not alter the weight reduction step.) A slightly different, but equivalent view is the following:

> Start with unit weights. At each time, pick a random expert, where expert i is picked with probability $\frac{w_i}{\sum_j w_j}$, and predict that picked expert's prediction. And each time an expert makes a mistake, multiply its weight by $(1-\epsilon)$.

The analysis very similar to Theorem 1, we just need to handle expectations.

Theorem 2 *Let $\epsilon \leq 1/2$. If on some sequence of days, the best expert makes M mistakes. The expected number of mistakes the randomized weighted majority algorithm makes is at most*

$$(1+\epsilon)M + O\left(\frac{\log n}{\epsilon}\right).$$

Proof: Again, let us look at the potential $\Phi = \sum_j w_j$, the total weight in the system. Having fixed the days, this potential varies deterministically.

Let F_t be the fraction of the total weight on the t^{th} day that is on the experts who make a mistake on that day. Hence

- The expected number of mistakes we make is $\sum_t F_t$.
- On the t^{th} day, $\Phi_{new} = \Phi_{old} \cdot (1 - \epsilon F_t)$. So,

  $$\Phi_{final} = n \cdot \prod_t (1 - \epsilon F_t) \leq n \cdot e^{-\epsilon \sum_t F_t},$$

  where we used that $(1+x) \leq e^x$ for all $x \in \mathbb{R}$.

Again, $\Phi_{final} \geq (1-\epsilon)^M$, so

$$(1-\epsilon)^M \leq n \cdot e^{-\epsilon \sum_t F_t} \quad \Rightarrow \quad \epsilon \sum_t F_t \leq M \ln \frac{1}{(1-\epsilon)} + \ln n.$$

We can now use that $\ln \frac{1}{(1-\epsilon)} = -\ln(1-\epsilon) \leq \epsilon + \epsilon^2$ for $\epsilon \in [0, \frac{1}{2}]$ to get

$$\text{the expected number of mistakes } = \sum_t F_t \leq M(1+\epsilon) + \frac{\ln n}{\epsilon}.$$

■

Pretty sweet, right? The number of mistakes we make is almost the same as the best expert, plus this additive term that just depends on n and ϵ, but is independent of the input sequence. Since the optimal number of mistakes M is at most T, the number of days, we can say

$$\Rightarrow \text{our \# of errors } \leq \text{ optimal \# of errors } + \epsilon T + \frac{\ln n}{\epsilon} \tag{1}$$

So dividing both sides by T, we get:

$$\Rightarrow \text{our error rate } \leq \text{ optimal error rate } + \epsilon + \frac{\ln n}{\epsilon T} \tag{2}$$

And now setting $\epsilon = \sqrt{\frac{\ln n}{T}}$:

$$\Rightarrow \text{our error rate } \leq \text{ optimal error rate } + 2\sqrt{\frac{\ln n}{T}}. \tag{3}$$

This last term is called the "regret". So as we do the prediction for longer and longer (i.e., $T \to \infty$), our error rate gets as close as we want to the optimal error rate. (We have "vanishing regret".)

4.1 Some Extensions

Repeated Zero-Sum Game. The interpretation of choosing a random expert (according to the probability distribution $\frac{w_i}{\sum_j w_j}$) allows us to view the process as playing a two-player zero-sum game repeatedly. Let the cost matrix contain only 0s and 1s. Each column is an expert. Each day the adversary plays a row (which defines the costs for us), we play a column (which is like choosing an expert). Theorem 2 says that we can play almost as best as the best column in hindsight.

Fractional Costs. Suppose each day t, instead of costs that are zero (no mistake) or one (mistake), we have costs c_i^t in $[0, 1]$. We can extend the result to that setting with a very slight change: update the weight of an expert i by $w_i \leftarrow w_i(1 - \epsilon c_i^t)$. With small changes in the analysis, the guarantee of Theorem 2 goes through unchanged! (Exercise: Show this!)

4.2 A Proof of the Minimax Theorem

Recall the minimax theorem: it says that in any zero-sum game (given by the row player's payoff matrix M), if we define the row player's guaranteed payoff (which is a lower bound on what she's guaranteed to earn regardless of what the column player does):

$$V_R = \max_{\mathbf{p}} \min_j \mathbf{p}^T M e_j = \max_{\mathbf{p}} \min_{\mathbf{q}} \mathbf{p}^T M \mathbf{q}$$

and the column player's guaranteed payoff (which is an upper bound on what the row player can earn, regardless of what he does):

$$V_C = \min_{\mathbf{q}} \max_i e_i^T M \mathbf{q} = \min_{\mathbf{q}} \max_{\mathbf{p}} \mathbf{p}^T M \mathbf{q},$$

then $V_C = V_R$. Clearly, $V_R \leq V_C$. The tricky part is the other direction, which we now use the experts theorem to prove.

Suppose not. There is some zero-sum game where $V_R = V_C - \delta$ for $\delta > 0$, with a strict inequality. Remember,

- If the column player commits to some strategy first, there is some row that the row player can play to get payoff at least V_C.
- On the other hand, if the row player commits first, then the column the column player can play so that row player get at most $V_R = V_C - \delta$.

By scaling the payoffs, imagine they are in $[0, 1]$.

Now imagine we use Randomized Weighted Majority (RWM) to play the columns. Since the distribution from which RWM draws is updated in a deterministic fashion, let the row player play optimally against this distribution at each step.

- In T steps, RWM has cost at most that of the best column (expert) in hindsight $+\epsilon T + \frac{\ln n}{\epsilon}$.
- The best column in hindsight can ensure cost at most $V_R \cdot T$, since it's like the row player has played first.
- But at each time, the row player knows your strategy, so his expected payoff is at least $V_C \cdot T$.

Putting these together

$$V_C \cdot T \leq \text{ row player's payoff } \leq V_R \cdot T + \epsilon T + \frac{\ln n}{\epsilon},$$

or

$$\delta T \leq \epsilon T + \frac{\ln n}{\epsilon}.$$

But we were free to choose ϵ and T as we want, so if we choose $\epsilon = \delta/2$ and $T \geq \frac{\ln n}{\epsilon^2}$, then we get a contradiction. Hence, there cannot be any gap δ, and $V_R = V_C$.

1 Mechanism Design (incentive-aware algorithms, inverse game theory)

- How to give away a printer
- The Vickrey Auction
- Social Welfare, incentive-compatibility
- The VCG (Vickrey-Clarke-Groves) mechanism

Today we're going to talk about an area called *mechanism design*, also sometimes called *incentive-aware algorithms* or *inverse game theory*. But rather than start with the big picture, let's start with an example.

1.1 How to give away a printer

Say the department has a spare printer, and wants to give it to whoever can make the most use of it. To make this formal, let's say there are n people, and assume each person i has some value $v_i \geq 0$ (called their *private value*) on getting the printer, and 0 on not getting it. We'll assume everyone knows their own v_i.

What we want to do is to give the printer to the person with the highest v_i. So, one option is we ask each person for their own v_i and we then compute the argmax (the i for which v_i is maximum) and give it to that person.

Can anyone see any potential problems with this? The problem is people might lie (*misreport* is the formal term) because they want the printer.

So, let's assume one more thing, which is that we (the department) have the ability to charge people money, and that the utility of person i for getting the item and paying p is $v_i - p$. (The definition of *utility* for our purposes is: the thing that people/players want to maximize. If there are probabilities, then we assume they want to maximize expected utility. But everything today will be deterministic.) Let's use $u_i(x)$ to denote the utility of player i for outcome x. For instance, $u_i(x) = 0$ for the outcome "get nothing, pay nothing".

To be clear: while we are giving the department the ability to charge money, it's goal isn't to make money but just to use this to help in getting the printer to the right person.

What about asking people how much they would pay (ask each person i to write down a bid b_i) and then give the printer to the person who bids the highest, charging them that amount. Any potential problems with this? Would you write down the amount that you value the printer as your bid? No. Because even if you win, you'd get zero utility.

Here is something interesting we can do called a Vickrey auction.

1.2 The Vickrey auction

The Vickrey auction:

- Ask everyone each person i to report the value of the printer to them (let's call this their "bid" b_i).
- Give the printer to $i = arg\max_j b_j$ (the person of highest reported value).
- Charge that person the *second-highest* bid.

Claim: Vickrey is *dominant-strategy truthful*, aka *incentive-compatible*. Specifically, for any valuations $v_1, \ldots, v_n$, for any player i, for any vector of bids of the other players (call this b_{-i}), we have:

$$u_i(\text{Vickrey}(v_i, b_{-i})) \geq u_i(\text{Vickrey}(v_i', b_{-i})).$$

Notation: given a vector v, will write v_{-i} as the vector removing the ith component, and "(x, v_{-i})" as the vector v with the ith component replaced by x.

In other words: even if you knew what all the other bids were, and got to choose what to bid based on those, you would still be best off (have highest utility) bidding your true value on the printer.

Can anyone see why?

Proof: Consider player i and let p be the highest bid among everyone else.

Case 1: $v_i > p$. In this case, if player i announces truthfully then it gets the item and pays p and has positive utility. Any other v_i' either will produce the same outcome or will result in someone else getting the item, for a utility of 0.

Case 2: $v_i = p$. Then it doesn't matter. Utility is 0 no matter what.

Case 3: $v_i < p$. In this case, announcing truthfully, player i doesn't get the item and has utility 0. Any other v_i' will either have the same outcome or else (if $v_i' > p$) will give him the item at a cost of p, yielding negative utility.

■

Another way to think of it: Vickrey is like a system that bids for you, up to a maximum bid of whatever you tell it, in an ascending auction where prices go up by tiny epsilons. Initially everyone is in the game and then they drop out as their maximum bids are reached. In this game, you would want to give v_i as your maximum bid (there is no advantage to dropping out early, and no reason to continue past v_i).

So, Vickrey is incentive-compatible and (assuming everyone bids their valuations, which they should because of IC) gives the printer to the person of highest value for it. This is called "maximizing social welfare".[1]

[1] Economists will call this "efficient". E.g., an "efficient market" is one that gets goods to the people who value them the most. We won't use that terminology because it clashes with the "runs quickly" meaning of "efficient".

1.3 What about two printers?

What if the department has two (equal quality) printers?

One option is you could do one Vickrey auction after another. Equivalently, take in the bids, give one printer to the highest bidder at a price equal to the 2nd-highest bid, and then give the other printer to the 2nd-highest bidder at a price equal to the 3rd highest bid. Does this work (is it incentive-compatible)? No. Why not?

How about giving the printers to the top 2 bidders at the 3rd-highest price? Does this work? Yes! Why?

> If you don't get a printer, would you regret your decision of bidding your true value v_i and want to raise your bid? No. If you do get a printer, you have no way of lowering the price you paid, and are happier than (or at least as happy as) if you lowered your bid below the 3rd highest and didn't get the printer.

So, this procedure (a) is incentive-compatible and (b) gives the printers to the two people who value them the most—i.e., it maximizes social welfare—if everyone bids truthfully (which they might as well do, by (a)).

So, you can think of this as inverse game theory, because we are designing the rules of the game so that if people act in their own interest, an outcome that we want will occur.

1.4 More general scenarios: the formal setup

What if the department has two printers but one is nicer than the other? Or maybe some things that go together (like bagels and cream cheese) or even dorm rooms where you might care not only about the room you get but maybe you also would prefer a room near to someone else in the same classes you could study with? The amazing thing is the Vickrey auction can be generalized to essentially any setting where you have payments and the players have what are called "quasi-linear utilities". This will be the Vickrey-Clarke-Groves, or VCG, mechanism. Let's now define the general setup formally.

We have n players and a set of alternatives A (will also call them "allocations"), such as who gets the printers or what the assignment of students to dorm rooms is. It can be arbitrarily complicated (we're not going to be worried about running time here). Each player i has a valuation *function* $v_i : A \to \mathbb{R}$.

We assume **quasi-linear utilities**: The utility for alternative $a \in A$ and paying a payment p is $v_i(a) - p$. It's called "quasi-linear" because it is linear in money, even if it might be some weird function over the alternatives. E.g., if we have multiple items we are allocating, the utility does not have to be additive over the items you get (maybe you need several together to build a product or maybe two printers isn't much better than one, and it even can depend on what other people get!) but you are assumed to be linear in money.

The **social welfare** of an allocation a is $SW(a) = \sum_i v_i(a)$. Notice that it involves the values, not the utilities. However, we can think of it as the sum of utilities if we also put the utility of the center (the department in this case) in the picture, so the money cancels out.

A **direct revelation mechanism** is a function that takes in a sequence $v = (v_1, ..., v_n)$ of valuation functions, and selects an alternative $a \in A$, along with a vector p of payments. It will be convenient

to split it into two functions: $f(v) = a$ and $p(v) =$ the vector of payments. We will use $p_i(v)$ to denote the payment of player i.

A direct revelation mechanism (f, p) is **incentive-compatible** if for every $v = (v_1, ..., v_n)$, every i, every v_i', we have:

$$v_i(f(v)) - p_i(v) \geq v_i(f(v_i', v_{-i})) - p_i(v_i', v_{-i}).$$

I.e., misreporting can never help.

Here is the amazing thing: there exists a mechanism, called VCG, that in this very general setting is both (a) incentive compatible and (b) produces the alternative that maximizes social welfare if everyone reports truthfully (which they should, due to (a)).

1.5 The Vickrey-Clarke-Groves (VCG) mechanism

The basic idea is to design payments so that everyone wants to optimize what *we* want to optimize, namely social welfare. There are a couple versions. Let's start with the simplest to analyze:

VCG version 1: Given a vector of reported valuation functions v,

- Let $f(v)$ be the allocation that maximizes social welfare with respect to v.
  I.e., $f(v) = arg\max_{a \in A} \sum_j v_j(a)$.
- Pay each player i an amount equal to the sum of everyone else's reported valuations.
  I.e., $p_i(v) = -\sum_{j \neq i} v_j(f(v))$.

Analysis: Suppose player i reports truthfully. Then its utility will be

$$v_i(f(v)) + \sum_{j \neq i} v_j(f(v)) \;\; = \;\; \sum_j v_j(f(v)) \;\; = \;\; \max_a \sum_j v_j(a) \quad \text{(by the definition of } f(v))$$

Suppose instead player i reports v_i'. Call the resulting vector v'. Then player i's utility will be:

$$v_i(f(v')) + \sum_{j \neq i} v_j(f(v')) \;\; = \;\; \sum_j v_j(f(v')) \;\; \leq \;\; \max_a \sum_j v_j(a).$$

So, misreporting can only hurt. This means that the mechanism is incentive-compatible, and by design it maximizes social welfare when everyone reports truthfully.

Problems with version 1: If you think of this as an auction, a big problem is this requires the auctioneer to give money to the bidders! E.g, in the case of the printer, it corresponds to giving the top guy the printer for free, and pay everyone else the amount the top guy valued it. That way everyone gets a utility equal to what the top guy got.

However, notice that if we add to each $p_i(v)$ something that depends on v_{-i} only (and not influenced at all by v_i) then it is just a constant as far as player i is concerned and so still incentive-compatible. This suggests the following generalization:

VCG - general version: Let h_i be any function over v_{-i} for each $i = 1 \ldots n$. Now, given a vector of reported valuation functions v,

- Let $f(v)$ be the allocation that maximizes social welfare with respect to v. I.e., $f(v) = arg\max_{a \in A} \sum_j v_j(a)$.
- Let $p_i(v) = h_i(v_{-i}) - \sum_{j \neq i} v_j(f(v))$.

As we just argued, this is incentive-compatible too.

Now, there is a specific set of h_i's that have the nice properties that (a) the center is never paying the bidders/players, and (b) on the other hand, assuming the v_i's themselves are non-negative, no player gets negative utility: this is called "individual rationality" (e.g., if we're allocating goods and people's valuations depend only on what they get, then among other things this implies that people who don't get anything don't have to pay anything). This set of h_i's is called the "Clarke pivot rule": $h_i(v_{-i}) = \max_a \sum_{j \neq i} v_j(a)$. This gives us the following:

VCG - standard version: Given a vector of reported valuation functions v,

- Let $f(v)$ be the allocation that maximizes social welfare with respect to v. I.e., $f(v) = arg\max_{a \in A} \sum_j v_j(a)$.
- Let $p_i(v) = \max_a \sum_{j \neq i} v_j(a) - \sum_{j \neq i} v_j(f(v))$.

In other words, you charge each player i an amount equal to how much less happy they make everyone else. This is often called "charging them their externality".

Why does this satisfy $p_i(v) \geq 0$? Answer: because the 1st term is a max.

Why does this satisfy individual rationality? Think of it this way: if you got value 3 but hurt everyone else's total value by more than 3, then this couldn't have been the maximum social welfare allocation since a better allocation would have been to use the the optimal allocation without you and give you nothing.

What does this look like for the case of the single printer? Everyone who doesn't get the printer pays nothing (both terms are equal to the maximum guy's value). The person who gets the printer pays the second-highest valuation (since the sum of everyone else's valuations went from the second-highest valuation down to zero.) So it reduces to the Vickrey auction in that case.

Last time we started talking about mechanism design: how to allocate an item to the person who has the maximum value for it, so that we are "incentive compatible"—no one has an incentive to misreport their valuations (a.k.a., to lie). This Vickery auction could be implemented as a natural ascending-price auction.

Then we generalized this to the VCG mechanism which extends to arbitrary allocation problems, with the same incentive-compatibility property. But VCG was a sealed-bid auction, people had to submit their entire valuation functions to the center, which would then output an allocation and some prices. In this lecture, we consider a special kind of market called a matching market, where we want a matching between buyers and goods. For this market we give a "natural" ascending-price auction that generalizes the single-item Vickery auction.[1] Along the way this gives a simple algorithm for finding max-weight matchings in bipartite graphs.

1 Matching Markets

Think of the setting of assigning dorm rooms to students, houses to buyers, etc. Each person wants one of each item. Each item can be given to at most one person. Formally, consider the setting where there is a set I of n items on sale, and a set B of n buyers who may buy some of them. The variables j will refer to a generic buyer and variables i will refer to items. Each buyer wants to buy at most one item, and has *valuation* $v_j(i) \in \mathbb{Z}$ for being allocated item i.[2] We assume that $v_j(i) \geq 0$. An example with $n = 3$, items on the left, buyers on the right, buyer a values items A, B, C at $10, 9, 0$ respectively.

If the price for item i is p_i, the utility that j gets from being assigned item i is

$$u_j(i) := v_j(i) - p_i.$$

Naturally, the buyer seeks to maximize his utility. Moreover, we want to ensure *individual rationality*, that the assignment results in every buyer achieving non-negative utility; hence if item i is assigned to buyer j at price p_i, it must be that $v_j(i) \geq p_i$.

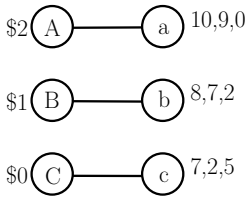
The *social welfare* achieved by an assignment of items to buyers is the sum of *valuations* of the buyers for their allocations. Not the sum of the utilities, the actual valuations. (If you sum utilities, you must include the utility of the sellers in the summation.) In the example above, at prices

[1] CLRS and DPV do not cover auctions and mechanism design. But a gentle and very readable introduction to matching markets (with more examples) can be found in Chapter 10 of the Easley-Kleinberg book: *Networks, Crowds, and Markets: Reasoning About a Highly Connected World*.

[2] In this model, the value of person j getting some room i does not depend on who else is assigned which other rooms. This model may not be accurate in some cases: e.g., you may value your assigned dorm room more or less depending on who has the neighboring rooms. We are not modeling such effects. In today's lecture, the value of j for some assignment of items to buyers depends only on the item that j gets in this assignment.

$p_A = 6, p_B = 5, p_C = 3$, assigning $A-a, B-b, C-c$ would achieve social welfare $10+7+5 = 22$. The utility of the players under this assignment would be $10-6 = 4, 7-5 = 2, 5-3 = 2$ respectively.

We want to find an allocation that maximizes the social welfare. Last time we saw the VGC mechanism that finds an allocation (and a pricing scheme) which maximizes the social welfare, and also gives incentive compatibility. That is, no player has any reason to misreport their valuations. And what would VCG do in this setting of a matching market? It would compute the max-weight matching in the bipartite graph where the weight of an edge (i, j) is the valuation $v_j(i)$. This would give us the allocation. Then the payment that buyer j makes is the "externality" that j causes: the difference between the valuations of all the other players when i participates, and when it does not. In the example above, a max-weight matching is $M_{VCG} = \{A-a, B-b, C-c\}$. The corresponding VCG prices are given below.



(What is the price VCG charges to a? The valuations of the other players under M_{VCG} is $7+5 = 12$. Now if a does not bid, then the optimal allocation for the others is $B-b, A-c$ with total valuation 14. Hence the "externality" caused is $14-12 = 2$, which is the price p_a VCG sets for a.)

But the VCG mechanism required us to submit our valuations to the center, which then finds the allocation and prices in this centralized way. Today we will give a different, distributed mechanism that acts more like real markets seem to do: it will gradually raise prices until they "stabilize", and each buyer will buy their favorite item (which maximizes their own utility), and that will be the optimal allocation, maximizing social welfare.

> *Important note: for all of Section 1, we will assume everyone acts truthfully. It will be just an algorithm design problem. We come back to incentive-compatibility in Section 2.*

1.1 Preferred Items and the Preferred Graph

Given prices $p_1, p_2, \ldots, p_n$ for the items, the preferred items for buyer j are all the items that maximize his utility (and for which the utility is non-negative). I.e., if

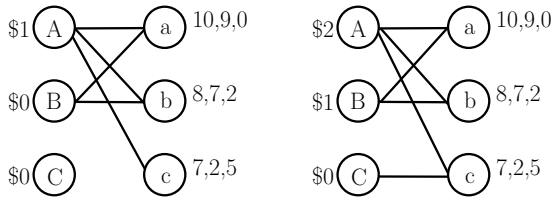
$$u_j^* := \max_{i \in I}(v_j(i) - p_i) \tag{1}$$

is the maximum utility that j can achieve at these price, and if $u_j^* \geq 0$, the preferred items (at these prices) are

$$S_j := \{i \in I \mid v_j(i) - p_i = u_j^*\}. \tag{2}$$

Else if $u_j^* < 0$ then buyer j has no preferred items (and $S_j = \emptyset$).

Now you can create the *preferred graph* H by having a node for each item $i \in I$, one for each buyer $j \in [n]$, and an edge (i, j) if item i is preferred item for j, i.e., $i \in S_j$. (All this is with respect to the current set of prices, of course.) Here are the preferred graphs with respect to two different price settings.



1.2 Market-Clearing Prices

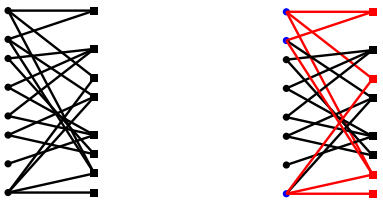
A set of prices $p_1, p_2, \ldots, p_n$, one for each item, is called *market-clearing* if

- There is a matching in the preferred graph H (at these prices) that matches *all* the buyers to items, and
- if an item i is not matched, then its price $p_i = 0$. (This is irrelevant when the number of items equals the number of buyers, since if all buyers are matched, so are the items. But if you had more items, you need this condition.)

This means at these prices, each buyer can come by and pick some item that maximizes her utility[3], and the market will "clear". (Of course, items that are not desirable might be left behind, but they have no value in this market and hence have zero price.) The matching given by the definition of market-clearing prices gives us the corresponding allocation.

In the previous figure on the right $p_A = 2, p_B = 1, p_C = 0$ were market-clearing prices, the graph has a perfect matching.

But $p_A = 1, p_B = 0, p_C = 0$ are not market-clearing prices. Why not? There is no perfect matching in the preferred graph H (on the left). How do you prove the absence of a perfect matching? Look at the set $S = \{a, b, c\}$. The neighborhood has size $|N(S)| = |\{A, B\}| = 2$ whereas this set has size $|S| = 3$. There is no way we can match S to its neighbors. Hall's theorem[4] says that a bipartite graph with equal number of vertices on either side has no perfect matching if and only if there is a set $S \subseteq B$ with neighborhood size $|N(S)| < |S|$. The set $N(S)$ is called a *constricted set* or a *Hall set*.[5] Here is another example of a bipartite graph with $n = 8$ where there is no perfect matching, as shown by a constricted set, the three (blue) neighbors on the left of the set of four red vertices on the right.



Take away message: if there is no perfect matching in a bipartite graph, we can always find a constricted set. This will be useful later.

[3] Due to ties, they can't pick an arbitrary item that maximizes their utility; we might have to give them a utility-maximizing item breaking ties carefully

[4] You may have seen it in 15-251; we also saw this in Recitation #6.

[5] A constricted set is a set $C \subseteq A$ such that there exists $S \subseteq R$ with $N(S) = C$, and $|S| > |C|$.

1.3 Market-Clearing Prices and Social Welfare Maximization

Why are we interested in market-clearing prices, apart from them seeming "natural"? And do they always exist? The first answer is short and sweet.

Theorem 1 *If there are market-clearing prices, then the corresponding allocation maximizes social-welfare.*

Proof: Fix any market-clearing prices p_i. Let M be the matching from items to buyers. By the definition, each player is given a preferred item, and hence is (individually) maximizing her utility. So the maximum utility that can be achieved (by any matching) with respect to these prices is precisely

$$\sum_{(i,j)\in M} u_j(i) = \sum_{(i,j)\in M} (v_j(i) - p_i).$$

Now look at the items. Some of them are matched to buyers and their prices are included in the sum. Others are not matched, and by the definition again, their prices must be zero. Hence the above expressions are equal to

$$\sum_{(i,j)\in M} v_j(i) - \sum_i p_i.$$

But the latter sum does not depend on the matching, so in finding the matching maximizing this expression, we have found a matching that maximizes the social welfare. QED. ■

Cool: if we have market-clearing prices, they automatically maximize the value to the buyers.

And why should these market-clearing prices exist? Here's an algorithm that will terminate with these prices.

1.4 An Algorithm for Market-Clearing Prices

We want to find market-clearing prices. The idea will be a natural one: if a set of k items has more than k people preferring it at the current prices, it seems likely that the prices of these items should rise. And that is almost exactly what we will do.

Start with all prices $p_i = 0$. Our prices will always be integers. Recall that the valuations $v_j(i)$ are all non-negative integers. Now do the following:

1. Check if the current prices are market-clearing. I.e., build the preferred graph, check if there is a matching that matches all the buyers to items.[6] If so, stop and output prices/matching.

2. If there is no such matching, Hall's theorem says there exists some set S of buyers such that the set of neighbors of S in the graph H (denoted by $N(S)$) has strictly smaller cardinality. I.e., $|S| > |N(S)|$. (Recall we called $N(S)$ a *constricted set*).

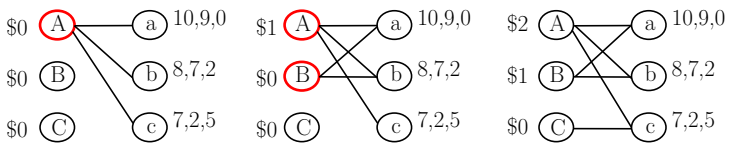
   Such a set is "over-subscribed", the demand for it (i.e., $|S|$) is more than the supply (i.e., $|N(S)|$). So such a set seems like a natural candidate to see price increases. For each item $i \in N(S)$, increment the prices $p_i \leftarrow p_i + 1$.

3. If this causes the minimum price to become non-zero (i.e., it must be 1), subtract 1 from each price so that the minimum price becomes zero,

4. Go back to Step 1.

[6] Since the number of items equals the number of buyers, if there is a matching that no items are left over, and we don't have to check the zero-price condition for unassigned items.

Here's a run of the algorithm (with the constricted sets shown in red).



If the algorithm terminates it will output market-clearing prices. So we show that the algorithm terminates. First, a simple lemma that justifies the price-lowering step at the very end.

Fact 2 *Each buyer has degree at least* 1 *in the preferred graph (at all times).*

Proof: By construction, at least one item stays at price 0. The utility of every buyer for this item remains non-negative, and hence the preferred set for any buyer can never be empty. ■

Theorem 3 *The algorithm terminates in finite time.*

Proof: We use a potential function. Given prices $p_1, p_2, \ldots, p_n$, define the potential to be

$$\Phi = \sum_{\text{items } i} p_i + \sum_{\text{buyers } j} u_j^*. \tag{3}$$

where u_j^* is defined as in (1). Note that all the terms here are non-negative. At the beginning, the prices are zero, and the $u_j^* = \max_{i \in I} v_j(i)$. So the initial potential $\Phi_0 = \sum_j \max_{i \in I} v_j(i)$.

Now, every time we execute Step 2 of the algorithm, the potential decreases. Why? By Fact 2, each buyer has at least one edge out of it, so $|N(S)| \geq 1$. Raising the prices on $|N(S)|$ nodes increases the potential by $|N(S)|$. However, the value of u_j^* decreases for all nodes in S—this decreases the potential by $|S|$. Since $|S| > |N(S)|$, the potential decreases by at least 1. So the algorithm stops in at most Φ_0 steps. ■

BTW, this algorithm is not polynomial-time, if the maximum valuation is $V_{\max}$ this process could take $\Omega(V_{\max})$ time to finish. You can change the algorithm slightly to take "bigger" steps, and get a poly-time algorithm out of it. (That algorithm is called the *Hungarian Algorithm*, due to Kuhn's reinterpretation of a theorem of Egervary.)

1.5 The Ascending-Price Mechanism*

You may wonder if the price-lowering step was actually required. What if we just wanted to have "ascending prices", which is more natural. One can get such an algorithm as well, with a little more care. The new algorithm is the following:

1. Check if the current prices are market-clearing. (I.e., build the preferred graph, check if there is a matching that matches all the buyers to items.) If so, stop and output prices/matching.

2. Else, there must be some set S of buyers such that the set of neighbors of S in the graph H (denoted by $N(S)$) have strictly smaller cardinality. I.e., $|S| > |N(S)|$. (Recall we called $N(S)$ a *constricted set*). Such a set is "over-subscribed", the demand for it (i.e., $|S|$) is more than the supply (i.e., $|N(S)|$). So such a set seems like a natural candidate to see price increases.

   There may be multiple constricted sets $N(S)$. Choose a *minimal* constricted set: i.e., $N(S)$ does not contain another constricted set $N(T)$. For each item $i \in N(S)$, increment the prices $p_i \leftarrow p_i + 1$. Go back to Step 1.

The choice of raising prices on a minimal constricted set gives us that at least one zero-price element remains, as the following lemma shows.

Lemma 4 *At most $n-1$ items will have strictly positive prices.*

Proof: Just for the sake of analysis, maintain a tentative matching M around, where M contains a subset of edges in the preferred graph. We maintain the invariant that if an item has non-zero cost, that item is tentatively matched to some buyer. In math, $p_i > 0 \implies \exists j : (i,j) \in M$. Initally M is empty, which satisfies the invariant.

Suppose you have prices and a tentative matching, and now you raise prices for items in $N(S)$. Tentatively match all the items in $N(S)$ to buyers in S. (If either these items, or the buyers they are matched to, were tentatively matched earlier, drop those tentative matching edges.) Since $|S| > |N(S)|$ and we chose a minimal $N(S)$, we know that this tentative matching can be done. (Why? Think about it — it is Hall's theorem again!) So the items in $N(S)$, they have non-zero prices and are tentatively matched. And the items outside $N(S)$ must be tentatively matched to buyers outside S (since all neighbors of S are in $N(S)$), and we did not drop any such edge. Hence, the invariant is maintained.

For sake of contradiction, if we ever want to raise the price for the n^{th} item, we would get a tentative matching of size n, which would mean we would have had a matching of all buyers to items, and hence would not want to raise prices at this step. Hence, at most $n-1$ items would every have their prices raised. ■

Using this Lemma, one can prove Fact 2 and hence Theorem 3 for this ascending-price algorithm.

2 Relationship to Vickery and VCG

The auctions we just saw are a direct generalization of the Vickery auction from the previous lecture. In the Vickery auction there was a single item we were auctioning among n buyers. To make it fit our model here, just introduce $n-1$ dummy items (numbered 2 through n), and number the real item 1. And if the buyer j had value v_j for the real item, define $v_j(1) = v_j$ and $v_j(i) = 0$ for $i \geq 2$. Assuming that each $v_j \geq 0$, the preferred graph starts off with edges from item 1 to all buyers. It is easy to check that $\{1\}$ is the unique constricted set (make sure you see why!), and we raise its price. Each time we raise its price, the preferred graph changes, but no other element can enter the constricted set. Finally, all except one buyer has a preferred edge to at least one item from $\{2, 3, \ldots, n\}$, and there is no more constricted set. This will happen precisely when p_1 reaches the second-highest valuation, and the person matched to the real item will be the one with the highest valuation.

Also, for the case when we have K identical items to sell, the same argument shows that our algorithm above will sell to the K highest bidders, at a price equal to the valuation of the $K+1^{st}$-highest bidder.

Is this just a coincidence? Or could it be the case that the matching and prices found by our algorithm above are the VCG allocation and prices? That would be great, because this would say that the ascending-price mechanism is also incentive compatible, that it is a dominant strategy for the players to play truthfully. Well, clearly the matching we found is the max-weight matching (by Theorem 1), and hence the same as the VCG allocation. But what about the prices? This is a little more tricky, but it is true: *the prices we find by the ascending-price auction are exactly the VCG prices!* We'll not cover this here, see Chapter 15.9 of the Easley-Kleinberg book for more details.

3 Walrasian equilibrium*

The same ideas work for more general markets where the players need not just want one item, they may be interested in buying multiple items. The valuation functions may have different characteristics. For instance, you may want a hotel room and a flight ticket and a car rental (and any strict subset of these might be useless to you). Or you may want multiple computers, but each additional computer gets less valuable to you the more you have ("diminishing returns"). Or you may value set A at \$50, or set B at \$100, but all other sets are worthless to you. One can imagine very general valuation functions. [7]

Formally, suppose you have a set of m items I, a set of n buyers B. Each player $j \in B$ now has valuations $v_j(S)$ for each subset $S \subseteq I$ of items. Let us define some useful shorthand: given prices $p_1, p_2, \ldots, p_m$, define

$$p(S) := \sum_{i \in S} p_i. \tag{4}$$

The utility of a set S for buyer j is $u_j(S) := v_j(S) - p(S)$. We imagine that $v_j(\emptyset) = 0$, and hence $u_j(\emptyset) = 0 - p(\emptyset) = 0$.

Definition 5 (Preferred Set) *At prices $p_1, p_2, \ldots, p_m$, a set S is called* preferred *for player j if $u_j(S) \geq 0$, and also for all $T \subseteq I$, $u_j(T) \leq u_j(S)$.*

Definition 6 (Walrasian Equilibrium) *The prices $p_1, p_2, \ldots, p_m$ and allocations $S_1, S_2, \ldots, S_n$ are at Walrasian equilibrium if (a) for each j the set S_j is preferred for player j, and (b) if item i is not allocated (i.e., $i \notin \cup_j S_j$), the price $p_i = 0$.*

This is the natural way to extend market-clearing prices to the more general setting, and a proof similar to Theorem 1 shows that if $\{p_i\}_{i \in I}, \{S_j\}_{j \in B}$ are a Walrasian equilibrium, then the allocation $\{S_j\}$ maximizes social welfare.

3.1 Existence No Longer Guaranteed

Unfortunately, Walrasian equilibria don't always exist, when the valuation functions are allowed to be so general.

> There are 2 items $\{A, B\}$ and 2 buyers $\{a, b\}$. Buyer a values the entire set at 3, and all other sets have value 0. b values every non-empty subset at 2. The welfare maximizing allocation is to give both items to a, with social welfare 3. For this to be a Walrasian equilibrium, what prices should we choose? The prices of each of the items must be at least 2 (for b's preferred set to be empty). But then the price of the entire set is at least 4, so a's preferred set cannot be $\{A, B\}$.

3.2 An Ascending-Price Mechanism for "Nice" Valuations

The problem in the example above comes from "complementarity". Items A and B "complement" each other, and the whole $\{A, B\}$ set is worth more to a than the sum of its parts. For example, think of A as being a left shoe and B the right shoe. Or a hotel room and a flight ticket. Or bread and butter.

But if the valuation functions are "nice", an ascending-price auction find an (approximate) Walrasian equilibrium. What are these "nice" valuations? These are called "substitutes", and is one

[7] For more details on this section's material, see the survey by Blumrosen and Nisan from the *Algorithmic Game Theory* book, which has loads of other cool topics. (Link to download entire book, 5Mb.)

way of ensuring the lack of complements. Loosely, it says that if you raise the price of a set of items, your preference or demand for some other item does not go down. (This clearly excludes complements, since if you were to raise the price of left shoes, not only would the demand for left shoes go down, but the demand for right shoes would also go down despite their prices not changing.)

Definition 7 (Substitutes) *Formally, valuation v_j satisfies the* substitutes *property if the following holds. For any two prices $p_1, p_2, \ldots, p_m$ and $q_1, q_2, \ldots, q_m$ where $p_i \leq q_i$, if P is a preferred set for buyer j at prices p, and $P^{=} := \{i \in P \mid p_i = q_i\}$ is the subset of P containing those items whose prices did not change, then there exists a preferred set Q for j at prices q with $P^{=} \subseteq Q$.*

Start with tentative assignments $T_j = \emptyset$, and prices $p_i = 0$.

1. Look at the preferred set P_j for buyer j under the following prices: each item $i \in T_j$ costs p_i, and each item $i \notin T_j$ costs $p_i + \epsilon$. (Think of this as saying that acquiring an item not in our tentative set has some ϵ extra cost, you have to work for it.)

2. If each such preferred set $P_j = T_j$, stop and return the prices p_i.

3. Else, pick some buyer with $P_j \neq T_j$, who is not satisfied. Set $T_j = P_j$, that is, give him his preferred set.

   Now increase the prices of all items newly assiged to j, namely those in $P_j \setminus T_j$, to $p_i + \epsilon$. Finally, remove these items in P_j from other tentative sets: $T_{j'} \leftarrow T_{j'} \setminus P_j$. Go back to Step 1.

Again, a natural ascending-price mechanism. If the valuations satisfy the substitutes property, this gives us an ϵ-approximate Walrasian equilibrium. The precise definition of this approximate equilibrium is not important, what is interesting is how this simple algorithm can give market-clearing prices. (Sadly, now the prices are no longer VCG prices, and players have an incentives to lie. One can get around this by pricing sets of items, but that is a story for another day.)

Last time we looked at algorithms for finding approximately-optimal solutions for NP-hard problems. Today we'll be looking at finding approximately-optimal solutions for problems where the difficulty is not that the problem is necessarily *computationally* hard but rather that the algorithm doesn't have all the *information* it needs to solve the problem up front.

Specifically, we will be looking at *online algorithms*, which are algorithms for settings where inputs or data is arriving over time, and we need to make decisions on the fly, without knowing what will happen in the future. This is as opposed to standard problems like sorting where you have all inputs at the start. Data structures are one example of online algorithms (they need to handle sequences of requests, and to do well without knowing in advance which requests will be arriving in the future). We'll talk about some other kinds of examples today.

1 Rent or buy?

Here is a simple online problem that captures a common issue in online decision-making, called the rent-or-buy problem. Say you are just starting to go skiing. You can either rent skis for \$50 or buy them for \$500. You don't know if you'll enjoy skiing, so you decide to rent. Then you decide to go again, and again, and after a while you realize you have shelled out a lot of money renting and you wish you had bought right at the start.[1] The optimal strategy is: if you know you're going to end up skiing more than 10 times, you should buy right at the beginning. If you know you're going to go fewer than 10 times, you should just always rent. (If you know you're going to go exactly 10 then either way is equally good.) But, what if you don't know?

To talk about the quality of an online algorithm, we will look at what's called its *competitive ratio*:

Definition 1 *The* competitive ratio *of an online algorithm ALG is the worst case (i.e., maximum) over possible futures σ of the ratio:*

$$\frac{ALG(\sigma)}{OPT(\sigma)},$$

where $ALG(\sigma)$ represents the cost of ALG on σ and $OPT(\sigma)$ is the least possible cost on σ.

E.g., what is competitive ratio of the algorithm that says "buy right away"? The worst case is we only go skiing once. Here the ratio is $500/50 = 10$.

What about the algorithm that says "Rent forever"? Now the worst case is that we keep going skiing. So the competitive ratio of this algorithm is unbounded.

Here's a nice strategy: rent until you realize you should have bought, then buy. (In our case: rent 9 times, then buy). Let's call this algorithm *better-late-than-never*. Formally, if the rental cost is r and the purchase cost is p then the algorithm is to rent $\lceil p/r \rceil - 1$ times and then buy.

Theorem 2 *The algorithm* better-late-than-never *has competitive ratio ≤ 2. If the purchase cost p is an integer multiple of the rental cost r, then the competitive ratio is $2 - r/p$.*

Proof: We consider two cases. **Case 1:** if you went skiing fewer than $\lceil p/r \rceil$ times (e.g., 9 or fewer times in the case of $p = 500, r = 50$) then you are optimal. The algorithm never purchased and

[1] We are ignoring practical issues such as the type of ski you want depending on your ability level, etc.

OPT doesn't purchase either. **Case 2:** If you went skiing $\lceil p/r \rceil$ or more times, then the optimal solution would have been to buy at the start, so OPT $= p$. The algorithm paid $r(\lceil p/r \rceil - 1) + p$ (e.g., \$450 + \$500 in our specific case). This is always less than $2p$, and equals $2p - r$ if p is a multiple of r. In Case 1, the ratio of the algorithm's cost to OPT was 1, and in Case 2, the ratio of the algorithm's cost to OPT was less than 2 ($(2p - r)/p = 2 - r/p$ if p was a multiple of r). The worst of these is Case 2, and gives the claimed competitive ratio. ■

Theorem 3 *Algorithm* better-late-than-never *has the best possible competitive ratio for the ski-rental problem for deterministic algorithms when p is a multiple of r.*

Proof: Consider the event that the day you purchase is the last day you go skiing (this is a legitimate event, since (a) if the algorithm never purchases, we already know its competitive ratio is unbounded, so we may assume a purchase occurs, and (b) the algorithm is deterministic so this occurs after some specific number of rentals). Now, if you rent longer than *better-late-than-never*, then the numerator in Case 2 goes up (the algorithm's cost is larger) but the denominator stays the same, so your ratio is strictly worse. If you rent fewer times (say you rent k fewer times than *better-late-than-never* for some $k \geq 1$), then the numerator in Case 2 goes down by kr but so does the denominator, so again the ratio is worse. ■

2 The elevator problem

You go up to the elevator and press the button. But who knows how long it's going to take to come, if ever? How long should you wait until you give up and take the stairs?

Say it takes time E to get to your floor by elevator (once it comes) and it takes time S by stairs. E.g, maybe $E = 15$ sec, and $S = 45$ sec. How long should you wait until you give up? What strategy has the best competitive ratio?

Answer: wait 30 sec, then take the stairs (in general, wait for $S - E$ time). This is exactly the *better-late-than-never* strategy since we are taking the stairs once we realize we should have taken them at the start. If elevator comes in less than 30 sec, we're optimal. Otherwise, OPT = 45. We took 30+45 sec, so the ratio is $(30 + 45)/45 = 5/3$. Or, in general, the ratio is $(S - E + S)/S = 2 - E/S$.

You may have noticed this is really the *same* as rent-or-buy where stairs=buy, waiting for E time steps is like renting, and the elevator arriving is like the last time you ever ski. So, this algorithm is optimal for the same reason. Other problems like this: whether it's worth optimizing code, when your laptop should stop spinning the disk between accesses, and many others.

When $E \ll S$, this is very close to being 2-competitive. As you saw in HW#4, you can do better by randomization. Indeed, even in the case where $E \ll S$, you can get close to $\frac{e}{e-1}$-competitiveness using the zero-sum game approach from the problem of waiting for the bus.

3 An aside

Interesting article in NYT Sept 29, 2007: Talking about a book by Jason Zweig on how people's emotions affect their investing, called "Your money and your brain":

> "There is a story in the book about Harry Markowitz, Mr. Zweig said the other day. He was referring to the renowned economist who shared a Nobel for helping found modern portfolio theory and proving the importance of diversification.... Mr. Markowitz was then working at the RAND Corporation and trying to figure out how to allocate his

> retirement account. He knew what he should do: I should have computed the historical co-variances of the asset classes and drawn an efficient frontier. (That's efficient-market talk for draining as much risk as possible out of his portfolio.)
>
> But, he said, I visualized my grief if the stock market went way up and I wasn't in it or if it went way down and I was completely in it. So I split my contributions 50/50 between stocks and bonds. As Mr. Zweig notes dryly, Mr. Markowitz had proved incapable of applying his breakthrough theory to his own money."

So, he wasn't applying his own theory but he *was* using competitive analysis: 50/50 guarantees you end up with at least half as much as if you had known in advance what would happen, which is best possible Competitive Ratio you can achieve.

4 List Update

This is a nice problem that illustrates some of the ideas one can use to analyze online algorithms. Here are the ground rules for the problem:

- There's a list of n items. (For simplicity we fix n). The positions in the list are numbered $1, 2, \ldots, n$. Position 1 is the front of the list. The initial ordering of the items in the list is the same for any algorithm.
- An item x can be accessed. The operation is called `Access`(x). The cost of the operation is the position i of x in the list.
- After doing an access, the algorithm is allowed to rearrange the list by doing swaps of adjacent elements. The cost of a swap is 1.

So an on-line algorithm is specified by describing which swaps are done after an element is accessed. (Without loss of generality we can give off-line optimum algorithm the power to do its sawps any time it wants, and not associate them with any particular access.)

The goal is to devise and analyze an on-line algorithm for doing all the accesses `Access`(σ_1), `Access`(σ_2), `Access`$(\sigma_3), \ldots$ with a small competitive factor.

Here are several algorithms to consider.

- **Do Nothing:** Don't reorder the list.
- **Single Exchange:** After accessing x, if x is not at the front of the list, swap it with its neighbor toward the front.
- **Frequency Count:** Maintain a frequency of access for each item. Keep the list ordered by non-increasing frequency from front to back.
- **Move To Front (MTF):** After an access to an element x, do a series of swaps to move x to the front of the list.

It's easy to construct sample sequences to show that Do Nothing, Single Exchange and Frequency Count have competitive factors that are $\Omega(n)$.

Theorem 4 *MTF is a 4-competitive algorithm for the list-update problem.*

Proof: We'll use the potential function method. There will be a potential function that depends on the state of the MTF algorithm and the state of the "opponent" algorithm B, which can be any algorithm, even one which can see the future. Using this potential, we'll show that the amortized cost to MTF of an access is at most 4 times the cost of that access to B.

What is the potential function Φ? Define

$$\Phi_t = 2 \cdot (\text{The number of inversions between } B\text{'s list and MTF's list at time } t)$$

Recall that an inversion is a pair of distinct elements (x, y) that appear in one order in B's list and in a different order in MTF's list. It's a measure of the similarity of the lists.

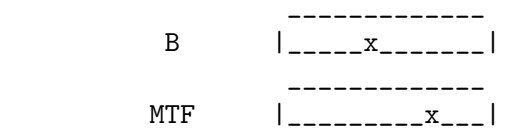
We can first analyze the amortized cost to MTF of `Access`(x) (where it pays for the list traversal and its swaps, but B only does its access). Then we separately analyze the amortized cost to MTF that is incurred when B does any swaps. (Note that in the latter case MTF incurs zero cost, but it will have a non-zero amortized cost, since the potential function may change. To be complete the analysis must take this into account.). In each case we'll show that the amortized cost to MTF (which is the actual cost, plus the increase in the potential) is at most 4 times the cost to B.

For any particular step, let C_{MTF} and C_B be the actual costs of MTF and B on this step, and $AC_{MTF} = C_{MTF} + \Delta\Phi$ be the amortized cost. Here $\Delta\Phi = \Phi_{new} - \Phi_{old}$ is the increase in Φ. Hence observe that $\Delta\Phi$ may be negative, and the amortized cost may be less than the actual cost. We want to show that

$$AC_{MTF} \leq 4 \cdot C_B$$

We can then sum the amortized costs, which would equal the actual cost of the entire sequence of accesses to MTF plus the final potential (non-negative) minus the initial potential (zero). This would be the four times total cost of B, which would give the result.

Analysis of `Access`(x). First look at what happens when MTS accesses x and brings it to the front of its list. Say the picture looks like this:



Look at the elements that lie before x in MTF's list, and partition them as follows:

$$S = \{y \mid y \text{ is before } x \text{ in MTF and } y \text{ is before } x \text{ in } B\}$$
$$T = \{y \mid y \text{ is before } x \text{ in MTF and } y \text{ is after } x \text{ in } B\}$$

What is the cost of the access to MTF in terms of these sets?

$$C_{MTF} = 1 + \underbrace{|S| + |T|}_{\text{find cost}} + \underbrace{|S| + |T|}_{\text{swap cost}} = 1 + 2(|S| + |T|).$$

On the other hand, since all of S lies before x in B, the cost of the algorithm B is at least

$$C_B \geq |S| + 1.$$

What happens to the potential as a result of this operation? Well, here's MTF after the operation:

```
        -------------
MTF    |x____________|
```

The only changes in the inversions involve element x, because all other pairs stay in the same relative order. Hence, for every element of S the the number of inversions increases by 1, and for every element of T the number of inversions decreases by 1. Hence the increase in Φ is precisely:

$$\Delta(\Phi) = 2 \times (|S| - |T|)$$

Now the amortized cost is

$$\begin{aligned} AC_{MTF} &= C_{MTF} + \Delta(\Phi) = 2(|S| - |T|) + 1 + 2(|S| + |T|) \\ &= 1 + 4|S| \leq 4(1 + |S|) \leq 4\, C_B \end{aligned}$$

This completes the amortized analysis of `Access`(x).

Analysis of B swapping. Now we perform all the swaps that B does. We do the analysis one swap at a time. For each such swap, observe that $C_{MTF} = 0$ and $C_B = 1$. Moreover, $\Delta(\Phi) \leq 2$, since the swap may introduce at most one new inversion. Hence,

$$AC_{MTF} \leq 2\, C_B \leq 4\, C_B$$

Putting the parts together. Summing the amortized costs we get:

$$\text{Total Cost to MTF } + \Phi_{final} - \Phi_{init} \leq 4(\text{Total Cost to } B)$$

But $\Phi_{init} = 0$, since we start off with the same list as B. And $\Phi_{final} \geq 0$. Hence $\Phi_{final} - \Phi_{init} \geq 0$. Hence,

$$\text{Total Cost to MTF} \leq 4 \times (\text{Total Cost to } B).$$

Hence the MTF algorithm is 4-competitive. ■

5 Paging

In paging, we have a disk with N pages, and fast memory with space for $k < N$ pages. When a memory request is made, if the page isn't in the fast memory, we have a page fault. We then need to bring the page into the fast memory and throw something else out if our space is full. Our goal is to minimize the number of misses. The algorithmic question is: what should we throw out? E.g., say $k = 3$ and the request sequence is 1,2,3,2,4,3,4,1,2,3,4. What would be the right thing to do in this case if we knew the future? Answer: throw out the thing whose next request is farthest in the future.

A standard online algorithm is LRU: "throw out the least recently used page". E.g., what would it do on above case? What's a bad case for LRU? 1,2,3,4,1,2,3,4,1,2,3,4... Notice that in this case, the algorithm makes a page fault every time and yet if we knew the future we could have thrown out a page whose next request was 3 time steps ahead. More generally, this type of example shows that the competitive ratio of LRU is at least k. In fact, you can show this is actually the worst-case for LRU, so the competitive ratio of LRU is exactly k (it's not hard to show but we won't prove it here).

In fact, it's not hard to show that you can't do better than a competitive ratio of k with a deterministic algorithm: you just set $N = k + 1$ and consider a request sequence that always

requests whichever page the algorithm *doesn't* have in its fast memory. By design, this will cause the algorithm to have a page fault every time. However, if we knew the future, every time we had a page fault we could always throw out the item whose next request is farthest in the future. Since there are k pages in our fast memory, for one of them, this next request has to be at least k time steps in the future, and since $N = k + 1$, this means we won't have a page fault for at least $k - 1$ more steps (until that one is requested). So, the algorithm that knows the future has a page fault at most once every k steps, and the ratio is k.

Here is a neat *randomized* algorithm with a competitive ratio of $O(\log k)$. Specifically, for any request sequence σ, we have $\mathrm{E}[ALG(\sigma)]/OPT(\sigma) = O(\log k)$.

Algorithm "Marking":

- Assume the initial state is pages $1, \ldots, k$ in fast memory. Start with all pages unmarked.
- When a page is requested,
  - if it's in fast memory already, mark it.
  - if it's not, then throw out a random unmarked page. (If all pages in fast memory are marked, unmark everything first. For analysis purposes, call this the end of a "phase"). Then bring in the page and mark it.

We can think of this as a 1-bit randomized LRU, where marks represent "recently used" vs "not recently used".

We will show the proof for the special case of $N = k + 1$. For general N, the proof follows similar lines but just is a bit more complicated.

Proof:(for $N = k + 1$). In a phase, you have $k + 1$ different pages requested so OPT has at least one page fault. For the algorithm, the page not in fast memory is a *random* unmarked page. Every time a page is requested: if it was already marked, then there is no page fault for sure. If it wasn't marked, then the probability of a page fault is $1/i$ where i is the number of unmarked pages. So, within a phase, the expected total cost is $1/k + 1/(k-1) + ... + 1/2 + 1 = O(\log k)$. ■

15-451/651: Algorithms **CMU, Spring 2016**
Lecture #24: Computational Geometry Introduciton April 13, 2015
Lecturer: Danny Sleator

1 Introduction

Computational geometry is the design and analysis of algorithms for geometric problems that arise on low dimensions, typically two or three dimensions. Many elegant algorithmic design and analysis techniques have been devised to attack geometric problems. This is why I've included this topic in this course.

Some applications of CG:

```
Computer Graphics
  images creation
  hidden surface removal
  illumination

Robotics
  motion planning

Geographic Information Systems
  Height of mountains
  vegetation
  population
  cities, roads, electric lines

CAD/CAM computer aided design/computer aided manufacturing

Computer chip design and simulations

Scientific Computation
  Blood flow simulations
  Molecular modeling and simulations
```

Basic algorithmic design approaches:

- Divide-and-Conquer
- Line-Sweep (typically in 2D)
- Random Incremental

In this course there will be three lectures on computational geometry covering the following topics:

- Geometric primitives
- Convex hull in 2D
- Random incremental algorithm for closest pair
- Divide and conquer algorithm for closest pair
- Sweep line algorithm for intersecting a set of segments
- An expected linear time algorithm for 2D linear programming.

1.1 Representations

The basic approach used by computers to handle complex geometric objects is to decompose the object into a large number of very simple objects. Examples:

- An image might be a 2D array of dots.
- An integrated circuit is a planar triangulation.
- Mickey Mouse is a surface of triangles

It is traditional to discuss geometric algorithms assuming that computing can be done on ideal objects, such as real valued points in the plane. The following chart gives some typical examples of representations.

Abstract Object	**Representation**
Real Number	Floating Point Number, Big Number
Point	Pair of Reals
Line	Pair of Points, An Equation
Line Segment	Pair of Endpoints
Triangle	Triple of points
Etc	

1.2 Using Points to Generate Objects

Suppose $P_1, P_2, \ldots P_k \in \mathbb{R}^d$. Below are several ways to use these points to generatel more complex objects.

Linear Combination

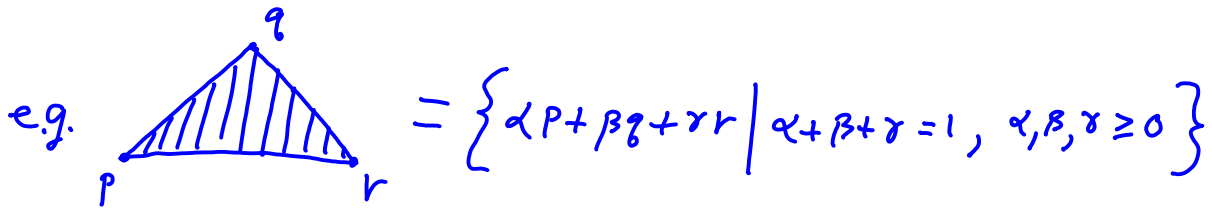
$$\text{Subspace} = \sum \alpha_i P_i \qquad \text{where} \qquad \alpha_i \in \mathbb{R}$$

Affine Combination

$$\text{Plane} = \sum \alpha_i P_i \qquad \text{where} \qquad \sum \alpha_i = 1, \;\; \alpha_i \in \mathbb{R}$$

Convex Combination

$$\text{Body} = \sum \alpha_i P_i \qquad \text{where} \qquad \sum \alpha_i = 1, \;\; \alpha_i \geq 0 \;\; \alpha_i \in \mathbb{R}$$



2 Primitive Operations

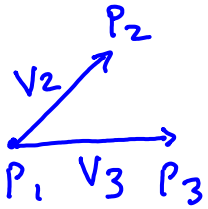
I'll be giving integer implementations of these primitives in ocaml. Let's start with some basic operations on vectors in 2D. The code below defines vector addition subtraction, cross product, dot product and the sign of a number.

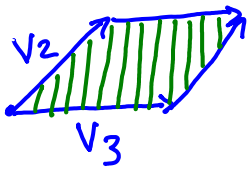
```
let (--) (x1,y1) (x2,y2) = (x1-x2, y1-y2)
let (++) (x1,y1) (x2,y2) = (x1+x2, y1+y2)
let cross (x1,y1) (x2,y2) = (x1*y2) - (y1*x2)
let dot (x1,y1) (x2,y2) = (x1*x2) + (y1*y2)
let sign x = compare x 0
  (* returns -1 if x<0, 0 if x=0 and 1 if x>0 *)
```

Line Side Test

Given three points P_1, P_2, P_3, the output of the *line side test* is "LEFT" if the point P_3 is to the left of ray $P_1 \to P_2$, "RIGHT" if the point P_3 is to the right of ray $P_1 \to P_2$, and "ON" if it is on that ray.

The algorithm is to construct vectors V_2 and V_3 by subtracting P_1 from P_2 and P_3 respectively. Then take the cross product of V_2 and V_2 and look at its value compared to 0.





$V_2 \times V_3$ = Signed area of parallelogram

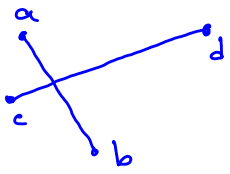
(in this case negative by the right hand rule.)

Here is an implementation of this test in ocaml which returns 1 if p_3 is LEFT of ray $p_1 \to p_2$, -1 if RIGHT, and 0 if ON.

```
let line_side_test p1 p2 p3 = sign (cross (p2--p1) (p3--p1))
```

Line segment intersection testing

Here we are given two line segments (a, b) and (c, d) (where a, b, c, d are points), and we have to determine if they cross. We can do this using four line-side tests, as illustrated here.



Segments cross iff

c & d on opp. sides of a→b

a & b on opp. sides of c→d

```
let segments_intersect (a,b) (c,d) =
  (line_side_test a b d) * (line_side_test a b c) <= 0 &&
  (line_side_test c d a) * (line_side_test c d b) <= 0
```

By changing the `<=` into a `<`, this can be changed into a strict intersection test, which would require the segments to intersect at a point interior to both of them.

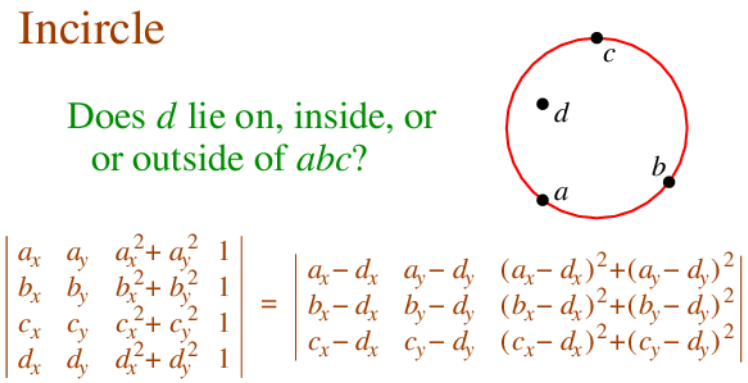
In-circle test

Three non-colinear points determine a circle. The *in-circle test* will tell us the relationship of a fourth point to the circle determined by the other three points. So the test takes points a, b, c, and d as inputs, and returns 1, 0, or -1 as follows:

> This returns 0 if the four points are on the same circle (or straight line.) Suppose I walk forward around the circle with my right hand on the circle from $a \to b \to c$. It returns 1 if d is on the same side of the circle as my body, and -1 otherwise. It's a fourth degree function in the given coordinates.

```
let incircle (ax,ay) (bx,by) (cx,cy) (dx,dy) =
  let det ((a,b,c),(d,e,f),(g,h,i)) =
    a*(e*i - f*h) - b*(d*i - f*g) + c*(d*h - e*g)
  in

  let row ax dx ay dy =
    let a = ax - dx in
    let b = ay - dy in
      (a, b, (a*a) + (b*b))
  in
    sign (det (row ax dx ay dy, row bx dx by dy, row cx dx cy dy))
```



The picture above illustrates a case when the incircle test would return 1. (This figure was taken from `http://www.cs.cmu.edu/~quake/robust.html`)

3 Computing the Convex Hull

This is the "sorting problem" of computational geometry. There are many algorithms for the problem, and there are often analogous to well-known sorting algorithms.

A point set $A \subseteq \mathbb{R}^d$ is *convex* if it is closed under convex combinations. That is, if we take any convex combination of any two points in A, the result is a point in A. In other words if when we walk along the straight line between any pair of points in A we find that the entire path is also inside of A, then the set A is convex.

We saw convex sets before when we talked about linear programming. One observation we used at that time is that the intersection of any two convex sets is convex.

Definition: ConvexClosure(A) = smallest convex set containing A

This is well-defined and unique for any point set A. (We won't prove this.) Assuming that the set A is a closed set of points we can define the convex hull of A as follows:

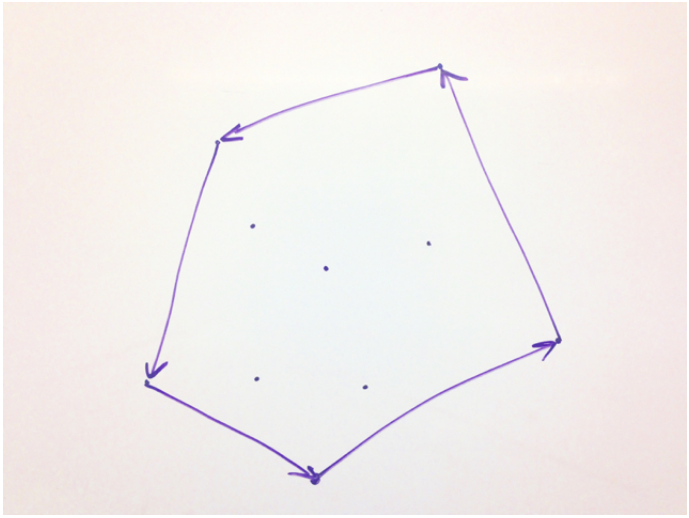
Definition: ConvexHull(A) = boundary of ConvexClosure(A). (A point p is on the boundary of S if for any $\epsilon > 0$ there exists a point within ϵ of p that is inside S and also another point with ϵ of p that is outside of S.)

These definitions are general and apply to any closed set of points.

For our purposes we're only interested in the ConvexClosure(A) and ConvexHull(A) when A is a finite set of points. In this case the ConvexClosure will be a closed polyhedron.

A computer representation of a convex hull must include the combinatorial structure. In two dimensions, this just means a simple polygon in, say counter-clockwise order. (In three dimensions it's a planar graph of vertices edges and faces) The vertices are a subset of the input points.

So in this context, a 2D convex hull algorithm takes as input a finite set of n points $A \in \mathbb{R}^2$, and produces a list L of points from A which are the vertices of the ConvexHull(A) in counter-clockwise order.



This figure shows the convex hull of 10 points.

Today we're going to focus on algorithms for convex hulls in 2-dimensions. We first present an $O(n^2)$ algorithm, than refine it to run in $O(n \log n)$. To slightly simplify the exposition we're going to assume that no three points of the input are colinear.

3.1 An $O(n^2)$ Algorithm for 2D Convex Hulls

First we give a trivial $O(n^3)$ algorithm for convex hull. The idea is that a directed segment between a pair of points (P_i, P_j) is on the convex hull iff all other points P_k are to the left of the ray from P_i to P_j. Note that no point to the right of the ray can be in the convex hull because that entire half-plane is devoid of points from the input set. And the points on the segment (P_i, P_j) are in the ConvexClosure of the input points. Therefore the segment is on the boundary of the ConvexClosure. Therefore it is on the convex hull.

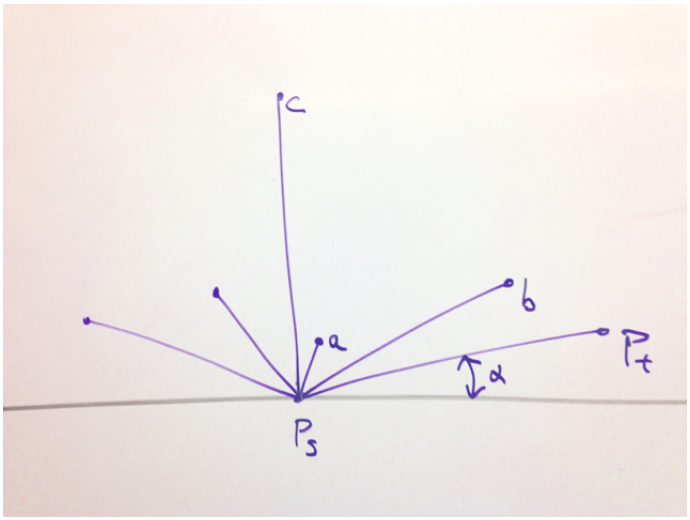
Here's the pseudo-code for this algorithm.

Slow-Hull$(P_1, P_2, \ldots, P_n)$:
    For each distinct pair of indices (i, j) do
        if for all $1 \leq k \leq n$ and $k \neq i$ and $k \neq j$
        it is the case that P_k is to the left of segment (P_i, P_j)
        Then output (i, j).
    done

(To make this into a proper convex hull algorithm, a final pass would be required to turn this list of pairs of indices into an ordered list of points in counterclockwise order.)

To get this to run in $O(n^2)$ time we just have to be a bit more organized.

The first observation is that if we take the point with the lowest y-coordinate, this point must be on the contex hull. Call it P_s. Suppose we now measure the angle from P_s to all the other points. These angles range from 0 to π. If we take the point P_t with the smallest such angle, then we know that (P_s, P_t) is on the convex hull. The following figure illustrates this.



All the other points must be to the left of segment (P_s, P_t). We can continue this process to find the point P_u which is the one with the smallest angle with respect to (P_s, P_t). This process is continued until all the points are exhausted. The running time is $O(n)$ to find each segment. There are $O(n)$ segments, so the algorithm is $O(n^2)$.

Actually we don't need to compute angles. The line-side-test can be used for this instead. For example look at what happens after we've found P_s and P_t. We process possibilities for the next point in any order. Say we start from a in the figure. Then we try b, and note that b is on the right side of segment (P_t, a) so we jettison a and continue with (P_t, b). But then we then throw out b in

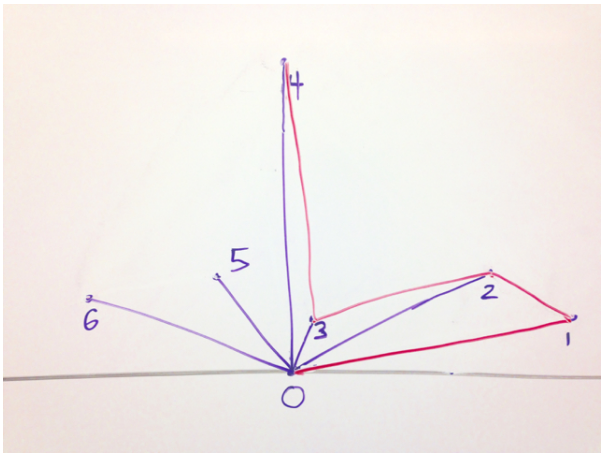
favor of c. It turns out that the remaining points are all to the left of segment (P_t, c). Thus $c = P_u$ is the next point on the convex hull.

3.2 Graham Scan, an $O(n \log n)$ Algorithm for 2D Convex Hulls

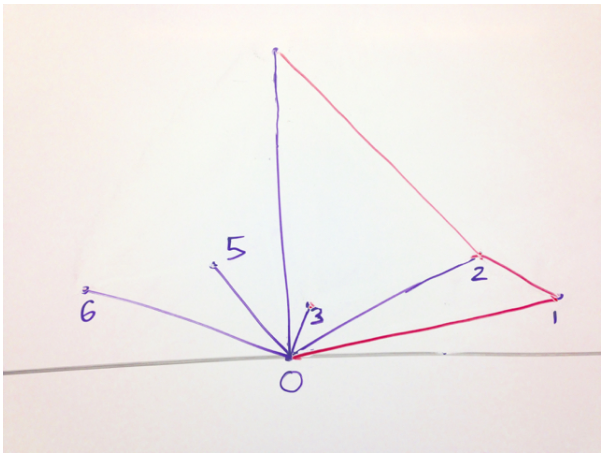
We can convert this into an $O(n \log n)$ algorithm with a slight tweek. Instead of processing the points in an arbitrary order, we process them in order of increasing angle with respect to point p_s.

Let's relabel the points so that P_0 is the starting point, and $P_1, P_2 \ldots$ are the remaining points in order of increasing angle with respect to P_0. From the discussion above we know that (P_0, P_1) is an edge of the convex hull.

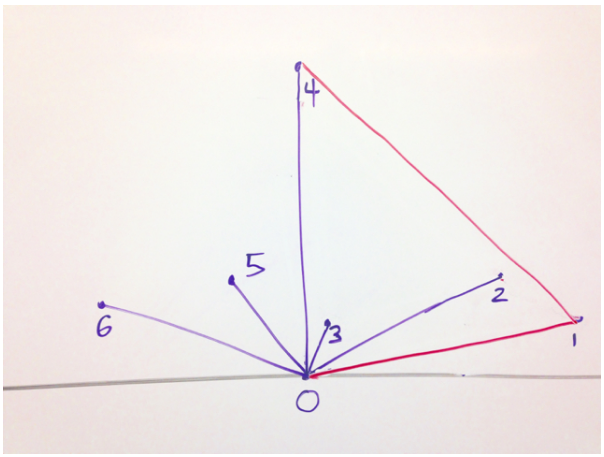
The Graham Scan works as follows. We main a "chain" of points that starts with $P_0, P_1, \ldots$. This chain has the property that each step is always a left turn with respect to the previous element of the chain. We try to extend the chain by taking the next point in the sorted order. If this has a left turn with respect to the current chain, we keep it. Otherwise we remove the last element of the chain (repeatedly) until the chain is again restored to be all left turns. Here's an example of the algorithm.



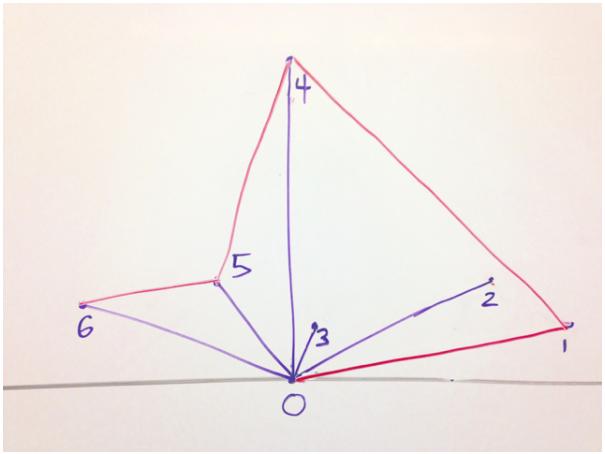
At this point we've formed the chain P_0, P_1, P_2, P_3, P_4. But the last step (from P_3 to P_4) is a right turn. So we delete P_3 from the chain. Now we have:

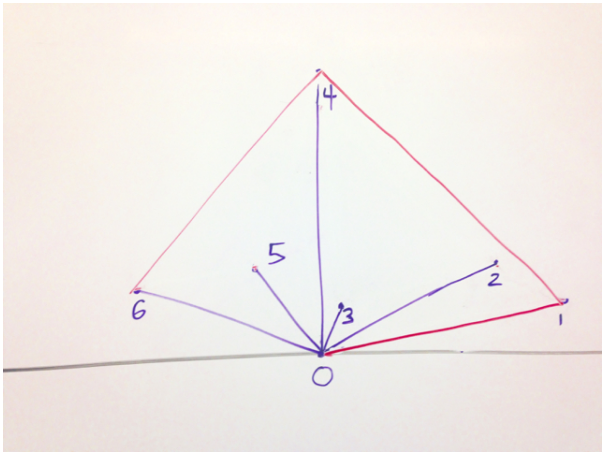


Now at P_2 we have a right turn, so we remove it, giving:



Now the process continues with points P_5 and P_6. When P_6 is added, P_5 becomes a right turn, so it's removed.





After all the points are processed in this way, we can just add the last segment from P_{n-1} to P_0, to close the polygon, which will be the convex hull.

Each point can be added at most once to the sequence of vertices, and each point can be removed at most once. Thus the running time of the scan is $O(n)$. But remember we already paid $O(n \log n)$ for sorting the points at the beginning of the algorithm, which makes the overall running time of the algorithm $O(n \log n)$.

The reason this algorithm works is because whenever we delete a point we have implicitely shown that it is a convex combination of other points. For example when we deleted P_3 we know that it is inside of the triangle formed by P_0, P_2 and P_4. Because of the presorting P_3 is to the left of (P_0, P_2), and to the right of (P_0, P_4). And because (P_2, P_3, P_4) is a right turn, P_3 is to the left of (P_2, P_4).

At the end the chain is all left turns, with nothing outside of it. Therefore it must be the convex hull.

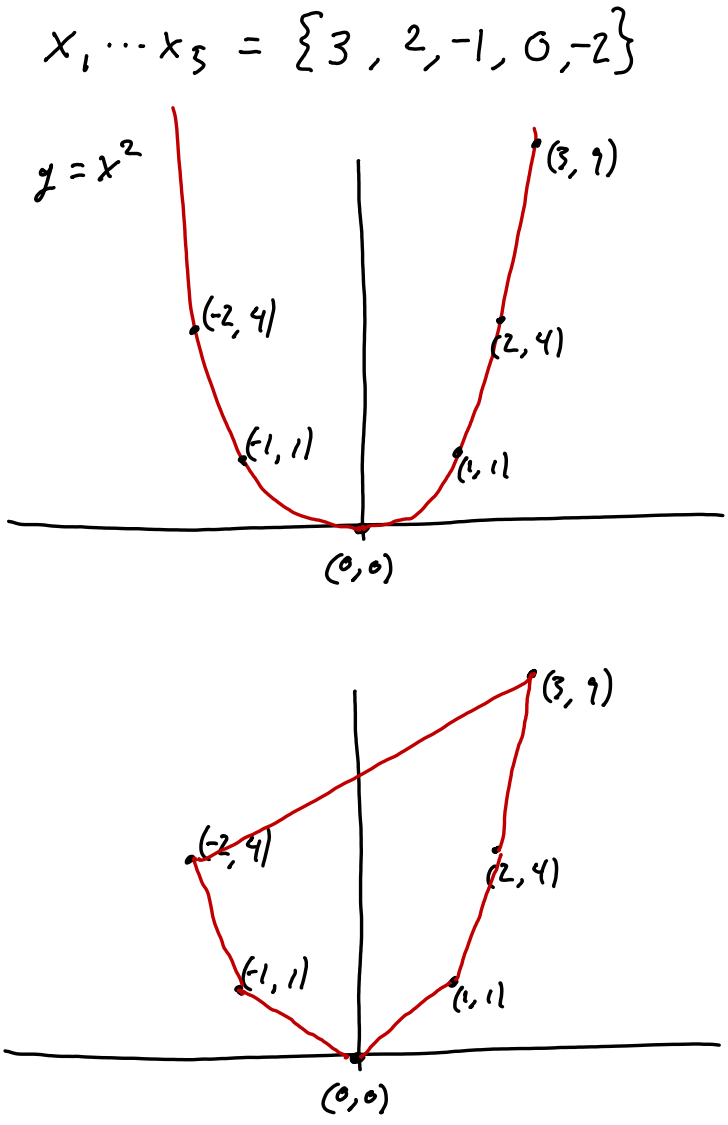
Complete ocaml code for the graham scan is at the end of these notes.

3.3 Lower bound for computing the convex hull

Suppose the input to a sorting problem is $X_1, \ldots, X_n$. Consider computing the convex hull of the following set of points:

$$(X_1, X_1^2), \ldots, (X_n, X_n^2)$$

All of these points are on the convex hull (they're on a parabola). Thus they are returned in the order they appear along the parabola. No matter which convex hull algorithm is used, the points can be reflected and/or cyclically shifted so that their x coordinates are in sorted order. Thus, they can be sorted by computing a convex hull followed by $O(n)$ additional work. Thus any comparison based convex hull algorithm must make $\Omega(n \log n)$ comparisons. The figure below illustrates this phenomenon.



3.4 Ocaml code for the Graham Scan convex hull algorithm

```
let (--) (x1,y1) (x2,y2) = (x1-x2, y1-y2)
let (++) (x1,y1) (x2,y2) = (x1+x2, y1+y2)
let cross (x1,y1) (x2,y2) = (x1*y2) - (y1*x2)
let dot (x1,y1) (x2,y2) = (x1*x2) + (y1*y2)
let sign x = compare x 0

let line_side_test p1 p2 p3 = sign (cross (p2--p1) (p3--p1))
(* Which side of ray p1-->p2 is p3 on?  This returns 1 for
   "LEFT", 0 for "ON" and -1 for "RIGHT". *)

let len (x,y) =
  let sq a = a*.a in
  let (x,y) = (float x, float y) in
  sqrt ((sq x) +. (sq y))

let graham_convex_hull points =
  let inf = max_int in
  let base = List.fold_left min (inf,inf) points in

  let points = List.sort (
    fun pi pj ->
      if pi=pj then 0
      else if pi=base then 1
      else if pj=base then -1
      else line_side_test base pj pi
  ) points in

  (* now the list starts at p1, and base is at the end of the list *)
  let rec scan chain points =
    let (c1,c2,chainx) = match chain with
      | c1::((c2::_) as chainx) -> (c1,c2,chainx)
      | _ -> failwith "chain must have length at least 2"
    in
    match points with [] -> chain
      | pt::tail ->
        match line_side_test c2 c1 pt with
          |  1 -> scan (pt::chain) tail
          | -1 -> scan chainx points
          | _  ->
            if len (pt--c2) > len (c1--c2)
            then scan (pt::chainx) tail
            else scan chain tail
  in
  match points with
    | (p1::((_::(_::_)) as rest)) -> List.tl(scan [p1;base] rest);
    | _ -> points (* do nothing if < 3 points *)

let print_list l =
  List.iter (fun (x,y) -> Printf.printf "(%d,%d) " x y) l;
  print_newline()
let () = print_list (graham_convex_hull [(0,0);(0,2);(2,2);(2,0);(1,1)])

-------------------------
output:
(0,2) (2,2) (2,0) (0,0)
```

1 Preliminaries

We'll give two algorithms for the following *closest pair* proglem:

> Given n points in the plane, find the pair of points that is the closest together.

The first algorithm is a deterministic divide and conquer and runs in $O(n \log n)$. The second one is random incremental and runs in expected time $O(n)$. In this lecture we make the following assumptions:

> We assume the points are presented as real number pairs (x, y).
> We assume arithmetic on reals is accurate and runs in $O(1)$ time.
> We will assume that we can take the floor function of a real.
> We also assume that hashing is $O(1)$ time.

These assumptions (in this context) are resonable, because the algorithms will not abuse this power.

2 $O(n \log n)$ Divide and Conquer Algorithm

First we present the algorithm. Then we prove that it works. Then we analyze its running time.

ClosestPair $(p_1, p_2, \ldots, p_n)$:
// The points are in sorted order by x.
// We also have the points in a different list ordered by y

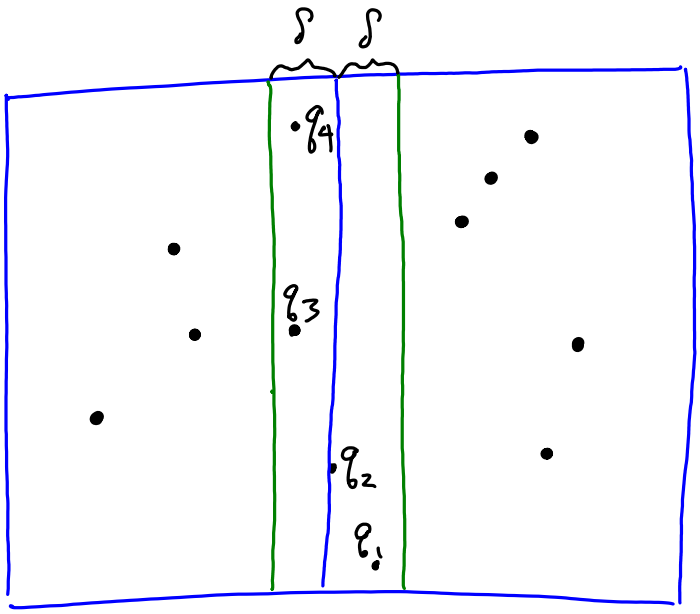
if $n \leq 3$ then solve and return the answer.
let $m = \lfloor n/2 \rfloor$
let $\delta = \min(\text{ClosestPair}(p_1, \ldots, p_m), \text{ClosestPair}(p_{m+1}, \ldots, p_n))$

form a list q of the points (sorted by increasing y) that are
within δ of the x coordinate of p_m. Call these points $q_1, q_2 \ldots q_k$

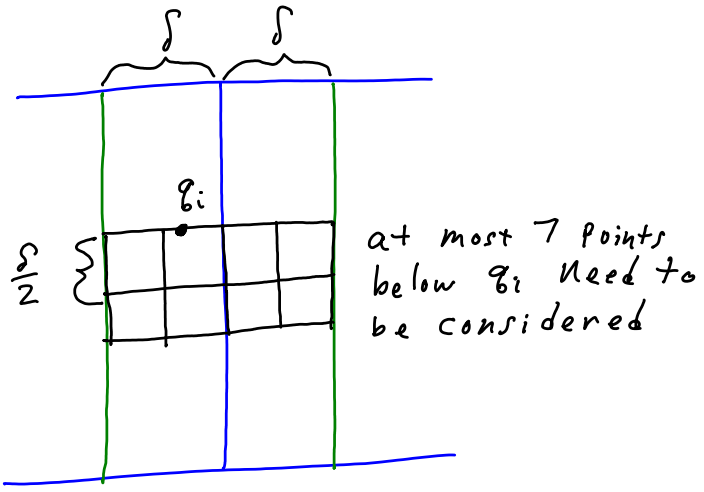
Now we compute d_i, the minimum distance between q_i and
all the qs below it. We do this for all $1 \leq i \leq k$
d_i = min distance from q_i to $q_{i-1}, q_{i-2}, \ldots, q_j$
where we stop when j gets to 1, or the y coordinate gets
too small: $q_j.y < q_i.y - \delta$

Return $\min(d_1, \ldots, d_k, \delta)$

The divide and conquer approach is obvious. The closest pair is either within the left half, within the right half, or it has one endpoint in the left half and one in the right half.



We need to determine if there is a pair that straddles the dividing line that has a distance less than δ. This is accomplished by the inner loop. It's obviously correct, because any pairs that are not considered are too far apart. (Their y coordinates differ by at least δ.) The only tricky part is proving that the inner loop (trying $q_{i-1}, \ldots, q_j$) finds a stopping value of j in $O(1)$ time.



In this figure there is a four (across) by two (down) grid of squares of size $\delta/2$. The top of the grid goes through q_i. Note that each of these squares can contain at most one of the points (interior or on its boundary), because any two points in the square are at most $\delta/\sqrt{2} < \delta$ apart. Therefore any q that is below the bottom of this grid is at least δ away from q_i, and thus has no effect on the closest pair distance. This proves that there are at most 7 points besides below q_i that are tested before the loop terminates. So for each i this search is $O(1)$ time.

The initiation phase sorts by x and also by y. This is $O(n \log n)$. Subsequent phases just filter these sorted lists, and no further sorting is done.

So the algorithm partitions the points ($O(n)$), does two recursive calls of $n/2$ in size, scans the points again to form the list q ($O(n)$), then scans the list q looking for the closest side-crossing pair ($O(n)$).

So we get the classic divide and conquer recurrence:

$$T(n) = 2T(n/2) + n$$

which solves to $O(n \log n)$.

3 Sariel Har-Peled's Randomized $O(n)$ Algorithm for closest pair

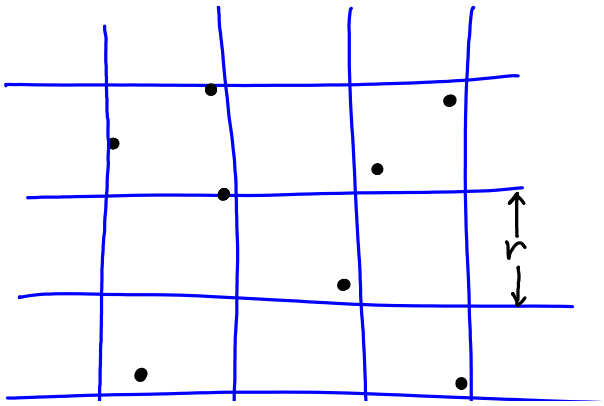
For any set of points P, let CP(P) be the closest pair distance in P.

We're going to define a "grid" data structure, and an API for it. The grid (denoted G) stores a set of points, (we'll call P) and also stores the closest pair distance r for those points. The number r is called "the grid size of G". Here's the API:

MakeGrid(p, q): Make and return a new grid containing points p and q using $r = |p - q|$ as the initial grid size.

Lookup(G, p): p is a point, G is a grid. This returns two types of answers. Let r' be the closest distance from p to a point in P. If $r' < r$ then return r'. If $r' > r$ return "Not Closest". Note that Lookup() does not need to compute r' if $r' \geq r$.

Insert(G, p): G is a grid. p is a new point not in the grid. This inserts p into the grid. It returns the current grid size.



The figure above shows a portion of the grid. Suppose a new point p lands in the middle square of this grid. Now to determine if it is closer to another point than the grid size, all we have to do is examine the points in each of the nine boxes shown. (In this case it is nine points.) Since CP(P) is the grid size, this means that each box has at most four points in it, and we only have to examine a total of at most 36 points.

Here's how we implement these. First define a function Boxify(x, y, r). This returns the integer point $(\lfloor x/r \rfloor, \lfloor y/r \rfloor)$.

The data structure maintains a hash table whose keys are integer pairs. The values in the hash table are lists of points from P. So the key (i, j) (also called a *box*) in the hash table stores all the points of P whose Boxify() value is (i, j). Also, it is inductively maintained that the grid size r is always equal to $\mathrm{CP}(P)$, the closest pair distance for the set of points being stored.

MakeGrid(p, q) is trivial. Just insert p and q into a new table with $r = |p - q|$.

Lookup(G, p) computes Boxify(p, r). It then looks in that box, and the 8 surrounding ones, and computes the distance between p and the closest one of these. Call this number r'. If $r' < r$, then we return r'. If $r' > r$ then return "Not Closest". This works because we know that if there is a point closer to p than r, it must be in one of the 9 boxes that are searched by this function. Also note that the running time of this is O(1) because it does 9 lookups in the hash table, and the total number of points it has to consider is at most 36. (A box contains at most 4 points.)

Insert(G, p) works as follows. It first does a Lookup(G, p). If the result is "Not Closest" it just inserts p into the data structure into box Boxify(p). This is $O(1)$ time. On the other hand if the Lookup() returns $r' < r$, then the algorithm rehashes every point into a new hash table based on the grid size being r'. This takes $O(i)$ time if there are i points now being stored in the data structure.

These algorithm are clearly correct, simply by virtue of the fact that at any point in time the grid size r is equal to the $\mathrm{CP}(P)$ where P is the set of points in the data structure. This is preserved by all the operations.

We can now complete the description of the algorithm.

Randomized-CP(P):
    Randomly permute the points. Call the new ordering $p_1, p_2, \ldots, p_n$.
    $G =$ Makegrid(p_1, p_2)
    for $i = 3$ to n do
        $r =$ Insert(G, p_i)
    done
    return r

Claim: This algorithm computes $\mathrm{CP}(P)$.

Proof: Follows from the definition of the API. QED.

Claim: The algorithm runs in expected $O(n)$ time.

Proof: Recall the time to do Insert() is $O(1)$ if the grid size does not change, and $O(i)$ ($i =$ the number of points in the grid) if the grid size does change.

Consider running the algorithm backwards. Here we are deleting points in order $p_n, p_{n-1}, \ldots, p_3$. When deleting point i, the operation is $O(1)$ if the closest pair distance does not change, and $O(i)$ if it does. In general if you remove a random point from a set of i points, the probability that the closest pair distance changes is at most $2/i$. (Because if there is just one pair with that closest distance, then deleting one of them is the only way to increase the closest pair distance. In other configurations, the probability is lower.)

So the removal is costly (i.e. $O(i)$) with probability at most $2/i$, and cheap $O(1)$ with the remaining probability. Therefore the expected cost of a step is $O(1)$. Thus the expected cost of the entire algorithm is $O(n)$. QED.

Below is code implementing this algorithm in Ocaml.

```
(*  Sariel Har-Peled's linear time algorithm for closest pairs.
    Danny Sleator, Nov 2014
*)

let sq x = x *. x

(* The function below takes an array of points (float*float), and returns
   the distance between the closest pair of points.  *)
let closest_pair p =
  let n = Array.length p in

  let dist i j =
    let (xi,yi) = p.(i) in
    let (xj,yj) = p.(j) in
    sqrt ((sq (xi -. xj)) +. (sq (yi -. yj)))
  in

  let truncate x r = int_of_float (floor (x /. r)) in
  let boxify (x,y) r = (truncate x r, truncate y r) in

  let getbox h box = try Hashtbl.find h box with Not_found -> [] in

  let add_to_h h i box =
    Hashtbl.replace h box (i::(getbox h box))
  in

  let make_grid i r =
    (* put points p.(0) ... p.(i) into a new grid of size r.
       it has already been established that the closest pair
       in that point set has distance r *)
    let h = Hashtbl.create 10 in

    for j=0 to i do
      add_to_h h j (boxify p.(j) r)
    done;
    h
  in

  Random.self_init ();

  let swap i j =
    let (pi,pj) = (p.(i),p.(j)) in
    p.(i) <- pj;
    p.(j) <- pi
  in

  for i=0 to n-2 do
    let r = Random.int (n-i) in
    swap i (i+r)
  done;
```

```
  let rec loop h i r =
    (* already built the table for points 0...i, and they have dist r *)

    if i=n-1 then r else
      let i = i+1 in
      let (ix,iy) = boxify p.(i) r in
      let li = ref [] in
      for x = ix-1 to ix+1 do
        for y = iy-1 to iy+1 do
          li := (getbox h (x,y)) @ !li
        done
      done;

      let r' = List.fold_left (
        fun ac j -> min (dist i j) ac
      ) max_float !li in

      if r' < r then (
        loop (make_grid i r') i r'
      ) else (
        add_to_h h i (ix,iy);
        loop h i r
      )
  in

  let r0 = dist 0 1 in
  loop (make_grid 1 r0) 1 r0

let () =
  let p = [|(0.0,0.0); (3.0,4.0); (20.0,15.0); (15.0,20.0)|] in
  let answer = closest_pair p in
  Printf.printf "%f\n" answer;
```

1 Preliminaries

The *sweep-line* paradigm is a very powerful algorithmic design technique. It's particularly useful for solving geometric problems, but it has other applications as well. We'll illustrate this by presenting algorithms for two problems involving intersecting collections of line segments in 2-D.

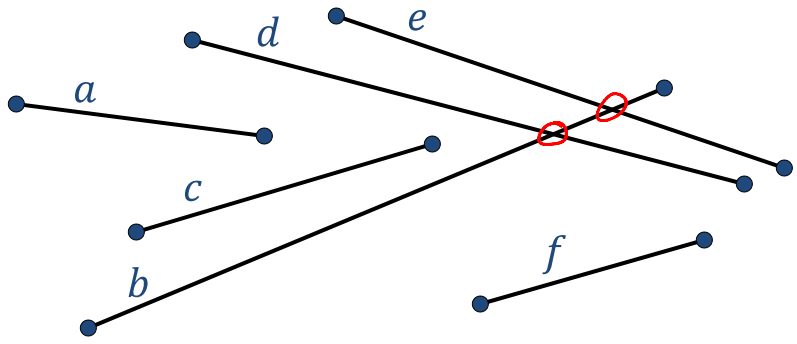
Generally speaking sweepline means that you are processing the data in some order (e.g. left to right order). A data structure is maintained that keeps the information gleened from the part of the data currently to the left of the sweepline. The sweepline moves across absorbing new pieces of the input and incorporating them into its data structure.

Obviously this is very vague. So let's get concrete and solve some problems.

2 Computing all Intersections of a Set of Segments

The input is a set S of line segments in the plane (each defined by a pair of points). The output is a list of all the places where these line segments intersect.

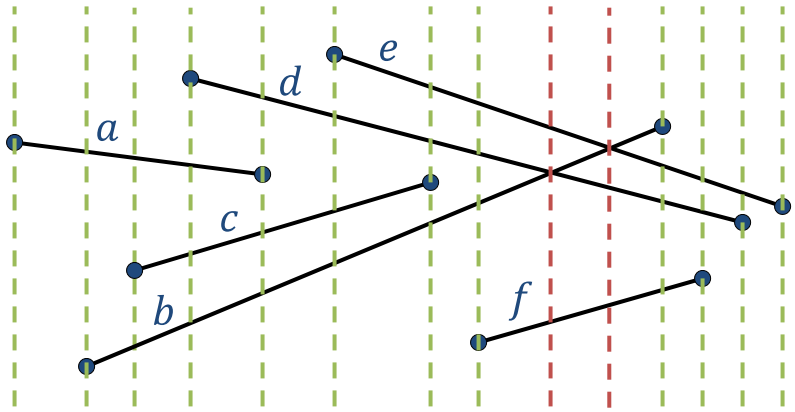
- Input: n line segments in 2D
- Goal: Find the k intersections



As usual, we're going to make our lives easier by making some geometric assumptions. We're going to assume that none of the segments is vertical. We'll also assume that no three segments intersect at the same point. We'll also assume that no segment has an endpoint that is part of another segment. (These assumptions can be avoided by adding some extra cases to the algorithm that do not change the running time bounds.)

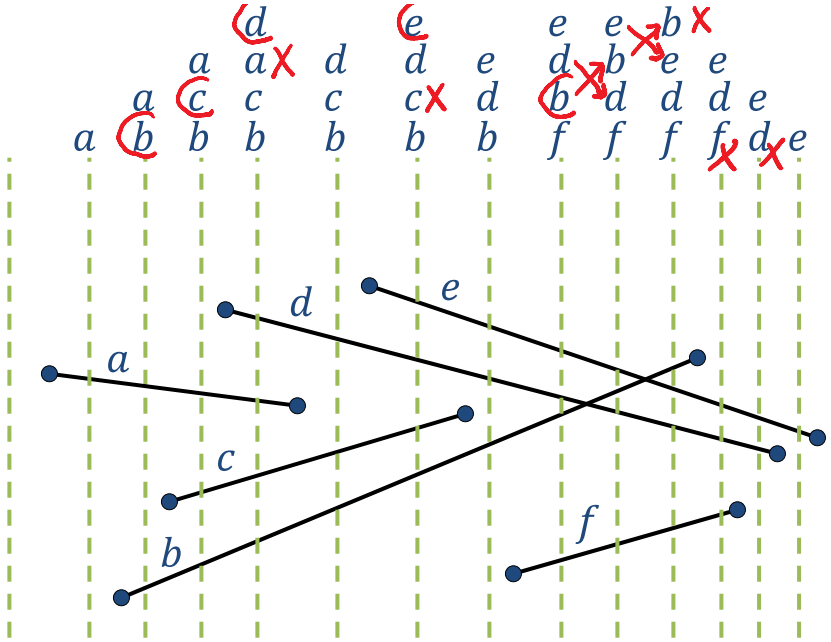
There's a trivial $O(n^2)$ algorithm: Just apply segment intersection to all pairs of segments. The solution we give here will be $O((n+k)\log n)$, where k is the number of segment intersections found by the algorithm.

To get some intuition for what's going to happen, consider the following figure.



Here we have six segments, called a, b, c, d, e, f. There are also two segment intersections. The "interesting" x-coordinates are the ones where something happens: either a segment begins, or a segment ends, or two segments cross.

The following figure shows a vertical dashed line before each interesting x-coordinate. Above the dashed line is a list of the segments in the same order they appear in the diagram, along that dashed line.



Between any two neighboring events, the segments (in that range of x) are in some order by y from bottom to top. This order (and which segments are there) does not change between a pair of neighboring interesting x coordinates.

We're going to maintain a data structure that represents the list shown at the top of the diagram. Let's call this a "segment list". Look at how this list changes as we move across the interesting x values. Only one of three things can happen. (1) A new segment is inserted into the list. (2) A segment is removed from the list. (3) Two neighboring segments in the list cross.

Our goal is to compute the intersections among these segments. The key observation that leads to a good algorithm is that right before two segments cross, they must be neighbors in the segment list. So the key idea of the algorithm is that as our segment list evolves (as we process the interesting x coordinates from left to right), we only need to consider the possible intersections between segments that are neighbors, at some point in time, in the segment list.

So our algorithm is going to maintain two data structures. A segment list (SL) and an event queue (EQ). Each event in the EQ will be labeled with its type ("segment start", "segment end", "segments cross"), as well as the x value where this happens. The EQ is initialized with all the segment starts and segment ends.

The EQ data structure must support `insert`, `findmin`, and `deletemin`. It's just a standard priority queue.

Although a segment is defined by its two endpoints, it's going to be useful to also use a "slope-intercept" representation for the line containing the segment. So segment i will have a left and right endpoint as well as a pair (m_i, b_i) where m_i is the slope of the line and b_i is the y-intercept.

Consider what we need for the SL data structure. The items being stored are a set of segments. The ordering used is the value of $m_i \cdot x + b_i$, where x is the current x value in the ongoing sweep-line algorithm. Here's what it needs to be able to do:

Requirements for the SL data structure:
- Insert a new segment into the data structure.
- Delete a segment from the data structure.
- Find the successor of a segment in the data structure.
- Find the predecessor of a segment in the data structure.

All of these operations can be done in $O(\log n)$ time using any standard search tree data structure (e.g. splay trees, or AVL trees).

Now we can write the complete algorithm. Note that whenever it says "take a pair of segments into consideration" what this means is that this pair of segments are now neighbors in the SL. It's possible that they intersect. So this function tests if they do intersect to the right of the current x coordinate. If they do, this event is added to the EQ.

```
FindSegmentIntersections(S):
    Create an empty EQ.
    For each segment in S, insert its start and end events into EQ.
    Create an empty SL.
    While the EQ is not empty do:
        let E = deletemin(EQ)
        if E is segment start of a segment s then
            Insert s into SL.
            Take the pair (s, the successor of s) into consideration.
            Take the pair (s, the predecessor of s) into consideration.
        else if E is a segment end of a segment s then
            Delete s from SL.
            Take the two former neighbors of s into consideration.
        else if E is a segments cross event involving segments s1 and s2 then
            Output the discovery of the segment intersection between s1 and s2.
            remove s1 and s2 from SL, and reinsert them in the opposite order.
            Take the two new neighbors of s1 and s2 into consderation.
    done.
```

It's easy to see that the running time of the algorithm is $O((n+k)\log n)$ using the data structures described. Because there are $O(n+k)$ events, and each one involves a constant number of operations on the SL and EQ data structures.

3 Counting Intersections of Horizontal and Vertical Segments

Note: this section will be updated for clarity eventually.

Now we switch gears and consider the case when the segments are only vertical or horizontal. Here however, we want to count the total number of intersections (not enumerate them) and we want our algorithm to run in $O(n \log n)$ time. Let V be the set of vertical segments and let H be the set of horizontal segments. We assume that no two vertical segments intersect and no two horizontal segments intersect.

First we form sorted lists of the "interesting" x and y values. A y is interesting of the segment endpoints has that y value. x is analogous. Label these values $y_1 < y_2 < \cdots < y_m$, and $x_1 < x_2 < \cdots < x_n$.

We sweep from left to right with a vertical sweep-line. The events are the interesting x coordinates.

There are three types of events:

We come to a vertical segment.
We come to the beginning of a horizontal segment
We come to the end of a horizontal segment.

At any point in time the vertical sweep-line crosses some number of horizontal segments. We define a variable a_i for each interesting y value y_i. $a_i = 1$ if the current sweep line crosses a horizontal segment with y coordinate equal to y_i. Otherwise it is zero. We maintain these a_i values using a segment tree (which you saw in recitation).

CountSegmentIntersetions(V, H):

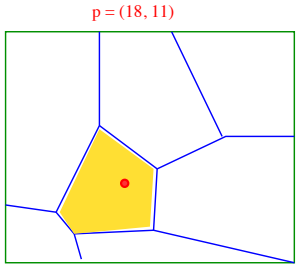
Construct the interesting x and y lists.
Create a segment tree ST using the interesting y values.
For each x_i in increasing order do:
If x_i is the right end of a horizontal segment s then
let the y value of s be y_j. then $a_j \leftarrow 0$
else if x_i is the left end of a horizontal segment s then
let the y value of s be y_j. then $a_j \leftarrow 1$
else if x_i is a vertical segment s then
let y_l be the bottom s and y_h be the top of s.
compute `sumRange`(l, r) and add it to a running total.
done

Note that if several events happen at the same x coordinate, then left ends should be processed first, then the vertical segments, then the right ends. This order means that we will not miss an intersection where the left end of the horizontal segment lies on a vertical segment.

The running time is $O(n \log n)$ where n is the total number of segments. This is because all the work can be assigned to the segment trees, which take $O(\log n)$ per opration.

Point Location

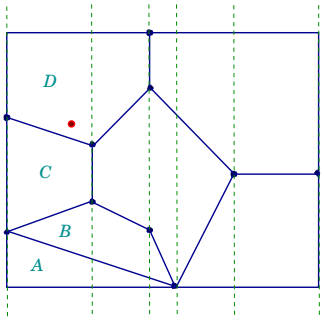
- Preprocess a planar, polygonal subdivision for point location queries.



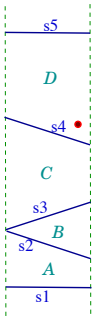
- Input is a subdivision S of complexity n, say, number of edges.
- Build a data structure on S so that for a query point $p = (x, y)$, we can find the face containing p fast.
- Important metrics: space and query complexity.

The Slab Method

- Draw a vertical line through each vertex. This decomposes the plane into slabs.
- In each slab, the vertical order of line segments remains constant.



Partition into slabs

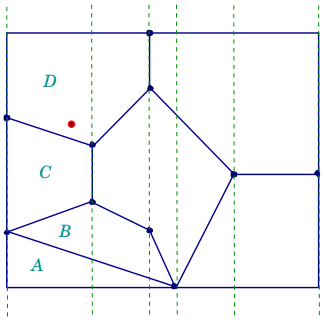


Slab 1

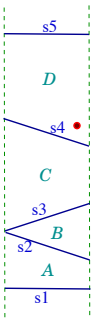
- If we know which slab $p = (x, y)$ lies, we can perform a binary search, using the sorted order of segments.

The Slab Method

- To find which slab contains p, we perform a binary search on x, among slab boundaries.
- A second binary search in the slab determines the face containing p.



Partition into slabs



Slab 1

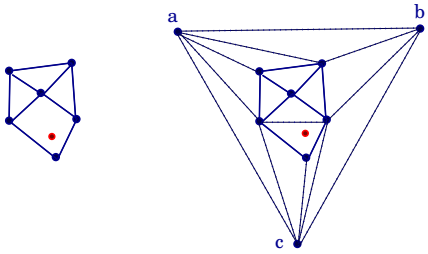
- Thus, the search complexity is $O(\log n)$.
- But the space complexity is $\Theta(n^2)$.

Optimal Schemes

- There are other schemes (kd-tree, quad-trees) that can perform point location reasonably well, they lack theoretical guarantees. Most have very bad worst-case performance.
- Finding an optimal scheme was challenging. Several schemes were developed in 70's that did either $O(\log n)$ query, but with $O(n \log n)$ space, or $O(\log^2 n)$ query with $O(n)$ space.
- Today, we will discuss an elegant and simple method that achieved optimality, $O(\log n)$ time and $O(n)$ space [D. Kirkpatrick '83].
- Kirkpatrick's scheme however involves large constant factors, which make it less attractive in practice.
- Later we will discuss the use of persistent data structures to obtain a practical and almost optimal solution.

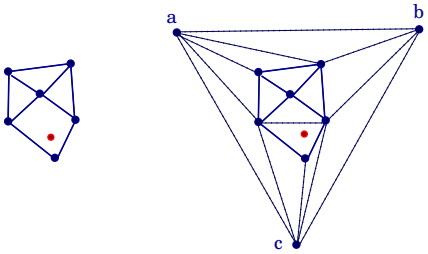
Kirkpatrick's Algorithm

- Start with the assumption that planar subdivision is a triangulation.
- If not, triangulate each face, and label each triangular face with the same label as the original containing face.
- If the outer face is not a triangle, compute the convex hull, and triangulate the pockets between the subdivision and CH.
- Now put a large triangle abc around the subdivision, and triangulate the space between the two.



Modifying Subdivision

- By Euler'e formula, the final size of this triangulated subdivision is still $O(n)$.
- This transformation from S to triangulation can be performed in $O(n \log n)$ time.



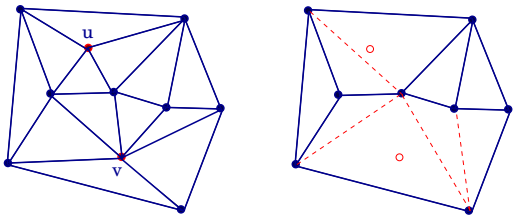
- If we can find the triangle containing p, we will know the original subdivision face containing p.

Hierarchical Method

- Kirkpatrick's method is hierarchical: produce a sequence of increasingly coarser triangulations, so that the last one has $O(1)$ size.
- Sequence of triangulations $T_0, T_1, \ldots, T_k$, with following properties:
  1. T_0 is the initial triangulation, and T_k is just the outer triangle abc.
  2. k is $O(\log n)$.
  3. Each triangle in T_{i+1} overlaps $O(1)$ triangles of T_i.
- Let us first discuss how to construct this sequence of triangulations.

Building the Sequence

- Main idea is to delete some vertices of T_i.
- Their deletion creates holes, which we re-triangulate.



Vertex deletion and re–triangulation

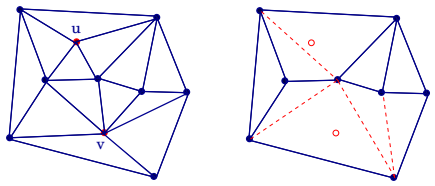
- We want to go from $O(n)$ size subdivision T_0 to $O(1)$ size subdivision T_k in $O(\log n)$ steps.
- Thus, we need to delete a constant fraction of vertices from T_i.
- A critical condition is to ensure each new triangle in T_{i+1} overlaps with $O(1)$ triangles of T_i.

Independent Sets

- Suppose we want to go from T_i to T_{i+1}, by deleting some points.
- Kirkpatrick's choice of points to be deleted had the following two properties:

[Constant Degree] Each deletion candidate has $O(1)$ degree in graph T_i.

- If p has degree d, then deleting p leaves a hole that can be filled with $d-2$ triangles.
- When we re-triangulate the hole, each new triangle can overlap at most d original triangles in T_i.

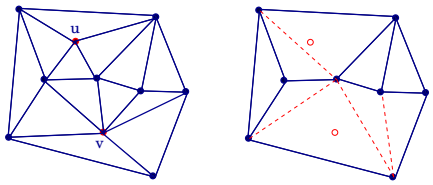


Vertex deletion and re–triangulation

Independent Sets

[Independent Sets] No two deletion candidates are adjacent.

- This makes re-triangulation easier; each hole handled independently.



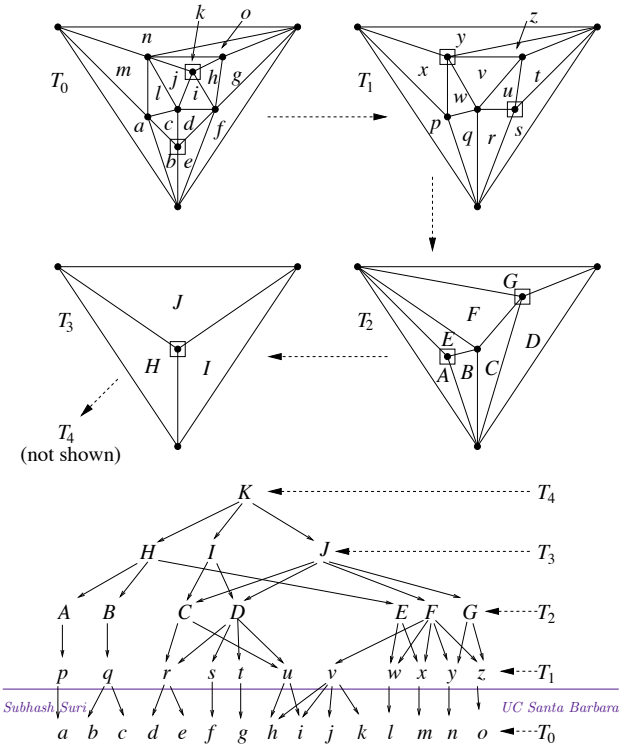
Vertex deletion and re–triangulation

I.S. Lemma

Lemma: Every planar graph on n vertices contains an independent vertex set of size $n/18$ in which each vertex has degree at most 8. The set can be found in $O(n)$ time.

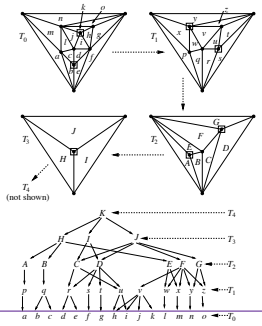
- We prove this later. Let's use this now to build the triangle hierarchy, and show how to perform point location.
- Start with T_0. Select an ind set S_0 of size $n/18$, with max degree 8. Never pick a, b, c, the outer triangle's vertices.
- Remove the vertices of S_0, and re-triangulate the holes.
- Label the new triangulation T_1. It has at most $\frac{17}{18}n$ vertices. Recursively build the hierarchy, until T_k is reduced to abc.
- The number of vertices drops by $17/18$ each time, so the depth of hierarchy is $k = \log_{18/17} n \approx 12 \log n$

Illustration

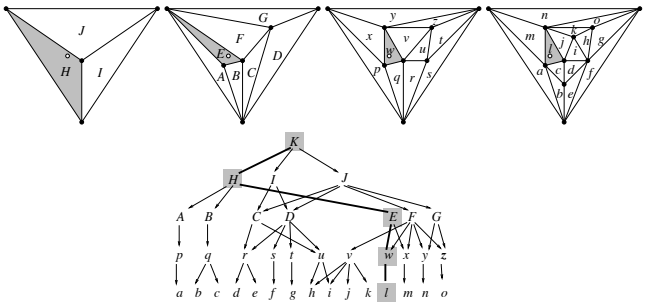


The Data Structure

- Modeled as a DAG: the root corresponds to single triangle T_k.
- The nodes at next level are triangles of T_{k-1}.
- Each node for a triangle in T_{i+1} has pointers to all triangles of T_i that it overlaps.
- To locate a point p, start at the root. If p outside T_k, we are done (exterior face). Otherwise, set $t = T_k$, as the triangle at current level containing p.

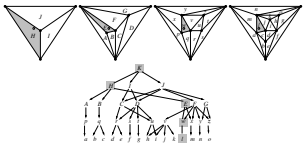


The Search



- Check each triangle of T_{k-1} that overlaps with t—at most 6 such triangles. Update t, and descend the structure until we reach T_0.
- Output t.

Analysis



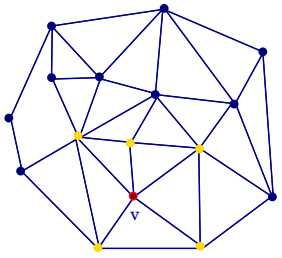
- Search time is $O(\log n)$—there are $O(\log n)$ levels, and it takes $O(1)$ time to move from level i to level $i-1$.
- Space complexity requires summing up the sizes of all the triangulations.
- Since each triangulation is a planar graph, it is sufficient to count the number of vertices.
- The total number of vertices in all triangulations is

$$n\left(1 + (17/18) + (17/18)^2 + (17/18)^3 + \cdots\right) \leq 18n.$$

- Kirkpatrick structure has $O(n)$ space and $O(\log n)$ query time.

Finding I.S.

- We describe an algorithm for finding the independent set with desired properties.
- Mark all nodes of degree ≥ 9.
- While there is an unmarked node, do
  1. Choose an unmarked node v.
  2. Add v to IS.
  3. Mark v and all its neighbors.
- Algorithm can be implemented in $O(n)$ time—keep unmarked vertices in list, and representing T so that neighbors can be found in $O(1)$ time.



I.S. Analysis

- **Existence of large size, low degree IS follows from Euler's formula for planar graphs.**
- **A triangulated planar graph on n vertices has $e = 3n - 6$ edges.**
- **Summing over the vertex degrees, we get**

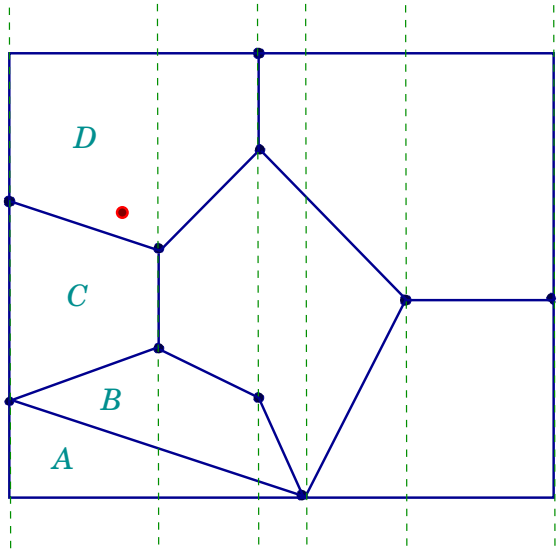
$$\sum_v deg(v) = 2e = 6n - 12 < 6n.$$

- **We now claim that at least $n/2$ vertices have degree ≤ 8.**
  - **Suppose otherwise. Then $n/2$ vertices all have degree ≥ 9. The remaining have degree at least 3. (Why?)**
  - **Thus, the sum of degrees will be at least $9\frac{n}{2} + 3\frac{n}{2} = 6n$, which contradicts the degree bound above.**
  - **So, in the beginning, at least $n/2$ nodes are unmarked. Each chosen v marks at most 8 other nodes (total 9 counting itself.)**
  - **Thus, the node selection step can be repeated at least $n/18$ times.**
  - **So, there is a I.S. of size $\geq n/18$, where each node has degree ≤ 8.**

1 Point Location using Persistent Search Trees

In this section we describe another approach to the point-location problem, based on a fully-functional, or "persistent", representation of sets. We obtain an $O(\log n)$ query time and $O(n \log n)$ space solution. So it is not optimal in terms of space. This can be made optimal by a more sophisticated way to make data structures persistent.

Let's go back to the approach of dividing the polygonal subdivision into slabs. With the representation described earlier, doing a point location query took only $O(\log n)$ time. The problem was that the data structure took too much space.



Partition into slabs

Let's examine this problem a little more closely. The reason that the space is potentially quadratic in n is that there is the possibility that a long horizontal segment can be divided up into many pieces, one for each slab. For example, the segment separating regions A and B is divided into three parts.

If there were some way that we could avoid having to store that redundantly in multiple slabs, we have a chance of controlling the space blow-up. This is the approach we take here.

With this in mind, let's examine the difference between consecutive slabs. Call the two slabs S_i and S_{i+1}. The differences between the two slabs occur when there is a vertex v of the subdivision along their boundary. To convert S_i into S_{i+1} we delete from S_i the segments incident on v from the left, and we insert into S_i the segments incident on v from the right. We do this for all the vertices on the boundary between the two slabs.

The total number of insertions and deletions that occur in the entire diagram is twice the number of segments, or $O(n)$. It turns out that using functional programming we can incurr a space coast of $O(\log n)$ per insertion or deletion.

1.1 Fully Functional Ordered Sets

A slab is represented by a set of non-vertical segments. For each non-vertical we're going to keep the equation of its line. So for segment i we keep $m_i x + b_i$, where m_i is the slope and b_i is the intercept. This representation will allow us to determine which of two segments is higher (at a particular x value).

A slab is a set of segments. We're going to store this in a balance binary search tree. It will be a fully functional implementation. This means, for example, that if we insert a segment a into a set S, it returns a new set S'. The original set S remains the same.

Our set data structure will support the following operations:

Insert(S, a):	Insert a segment a into a set S. Return the result.
Delete(S, a):	Delete a segment a from the set S. Return the result.
Lookup(S, p):	Given a point p and a set of segments S, return a segment from S that neighbors the region containing p.

Fully functional implementations of sets based on balanced binary search trees are standard features of functional programming languages such as Haskell, Ocaml, and SML. Furthermore, it's easy to create fully functional implementations in any language. You just have to maintain the discipline of never modifying any fields of a note. Instead you create a new node initialized with the values you want in each field.

Because the sets are stored as a tree, the way lookup works is that it returns the last segment touched when searching down the tree for a segment containing point p.

The space used by Insert and Delete is $O(\log n)$. This happens automatically in a functional implementation of sets using balanced binary search trees. (You can also think of it as copying the path from the root to a node that changes.)

1.2 Building the Data Structure

Start out with an empty slab S_0. Process the vertices $v_0, \ldots, v_k$ in left-to-right order. To process vertex v_i we take slab S_i and delete the segments connecting to it on the left and insert the ones that connect to it on the right. After this, we have slab S_{i+1}. The space used by this is $O(n \log n)$.

As we do this we build a an array A, sorted by x, with pointers to the slabs that we constructed in this scan. So given x we can find in $O(\log n)$ time the slab containing x.

We're also going to build a dictionary D which keeps, for each non-vertical segment in the diagram, the name of the region above it and below it.

1.3 Doing Point Location

Given a point $p = (x, y)$ here is how we find the region containing p.

First we find the slab containing p by doing binary search in the array A. Say it's in slab S_i. Now we do a lookup of p in S_i. This gives us a segment s bounding the region containing point p. Now we determine if the point p is above or below s. So we can then lookup in the dictionary D the name of the region containing the point p.

This process is $O(\log n)$ time.

1.4 Achieving $O(n)$ Space

Optimal space can be achieved by using the "fat node" method of making data structures persistent. It's described in this paper by Sarnak and Tarjan.

`www.link.cs.cmu.edu/15859-f07/papers/point-location.pdf`

The details are beyond the scope of this course, but feel free to talk to me about how it works.

In this lecture, we will see some of the power of polynomials in algorithm design.

You've probably all seen polynomials before: e.g., $3x^2 - 5x + 17$, or $2x - 3$, or $-x^5 + 38x^3 - 1$, or x, or even a constant 3. These are all polynomials over a single variable (here, x). The degree of a polynomial is the highest exponent of x that appears: hence the degrees of the above polynomials are $2, 1, 5, 1, 0$ respectively.

In general, a polynomial over the variable x of degree at most d looks like:

$$P(x) = c_d x^d + c_{d-1} x^{d-1} + .. + c_1 x + c_0$$

Note that the sequence of $d+1$ coefficients $\langle c_d, c_{d-1}, \ldots, c_0 \rangle$ completely describes $P(x)$.

Hence, if the coefficients were all drawn from the set $\mathbb{Z}_p$, then we have exactly p^{d+1} possible different polynomials. This includes the zero polynomial $0 = 0x^d + 0x^{d-1} + .. + 0x + 0$.

In this lecture, we will use properties of polynomials, to construct error correcting codes, and do other cool things with them.

1 Operations on Polynomials

Before we study properties of polynomials, recall the following simple operations on polynomials:

- Given two polynomials $P(x)$ and $Q(x)$, we can add them to get another polynomial $R(x) = P(x) + Q(x)$. Note that the degree of $R(x)$ is at most the maximum of the degrees of P and Q. (Q: Why is it not equal to the maximum?)

  $$(x^2 + 2x - 1) + (3x^3 + 7x) = 3x^3 + x^2 + 9x - 1$$

  The same holds for the difference of two polynomials $P(x) - Q(x)$, which is the same as $P(x) + (-Q(x))$.

- Given two polynomials $P(x)$ and $Q(x)$, we can multiply them to get another polynomial $S(x) = P(x) \times Q(x)$.

  $$(x^2 + 2x - 1) \times (3x^3 + 7x) = 3x^5 + 4x^3 + 6x^4 + 14x^2 - 7x$$

  The degree of $S(x)$ is equal to the sum of the degrees of P and Q.

- Note that $P(x)/Q(x)$ may not be a polynomial.

- We can also *evaluate* polynomials. Given a polynomial $P(x)$ and a value a, $P(a) := c_d \cdot a^d + c_{d-1} \cdot a^{d-1} + .. + c_1 \cdot a + c_0$. For example, if $P(x) = 3x^5 + 4x^3 + 6x^4 + 14x^2 - 7x$, then

  $$P(2) = 3 \cdot 2^5 + 4 \cdot 2^3 + 6 \cdot 2^4 + 14 \cdot 2^2 - 7 \cdot 2 = 266$$

  Of course, the multiplication between the c_i's and a must be well-defined, as should the meaning of a^i for all i. E.g., if the c_i's and a are reals, this is immediate.

  But also, if c_i's and a all belonged to $\mathbb{Z}_p$ for prime p, evaluation would still be well-defined. For instance, if we were working in $\mathbb{Z}_{17}$, then

  $$P(2) = 266 \pmod{17} = 11$$

- A *root* of a polynomial $P(x)$ is a value r such that $P(r) = 0$. For example, $P(x)$ above has three real roots $0, -1 + \sqrt{2}, -1 - \sqrt{2}$, and two complex roots.

Here, we were implictly working over $\mathbb{R}$, the field of real numbers. But almost everything today holds true for any field, say a finite field of integers modulo a prime ($\mathbb{F}_p$ for prime p).

2 How Many Roots?

Let's start with the following super-important theorem.

Theorem 1 (Few-Roots Theorem) *Any non-zero polynomial of degree at most d has at most d roots.*

This holds true, regardless of what field we are working over. When we are working over the reals (i.e., the coeffcients are reals, and we are allowed to plus in arbitrary reals for x), this theorem is a corollary of the fundamental theorem of Algebra. But it holds even if we are working over some other field (say $\mathbb{Z}_p$ for prime p).

Let's relate this to what we know. Consider polynomials of degree 1, also known as linear polynomials. Say they have real coefficients, this gives a straight line when we plot it. Such a polynomial has at most one root: it crosses the x-axis at most once. And in fact, any degree-1 polynomial looks like $c_1 x + c_0$, and hence setting $x = -c_0/c_1$ gives us a root. So, in fact, a polynomial of degree exactly 1 has exactly one root.

What about degree 2, the quadratics? Things get a little more tricky now, as you probably remember from high school. E.g., the polynomial $x^2 + 1$ has no real roots, but it has two complex roots. However, you might remember that if it has one real root, then both roots are real. But anyways, a quadratic crosses the x-axis at most twice. At most two roots.

And in general, Theorem 1 says, any polynomial of degree at most d has at most d roots.

3 A New Representation for degree-d Polynomials

Let's prove a simple corollary of Theorem 1, which says that if we plot two polynomials of degree at most d, then they can intersect in at most d points—*unless they are the same polynomial* (and hence intersect everywhere)! Remember, two distinct lines intersect at most once, two distinct quadratics intersect at most twice, etc. Same principle.

Corollary 2 *Given $d+1$ pairs $(a_0, b_0), (a_1, b_1), \ldots, (a_d, b_d)$, there is* at most *one polynomial $P(x)$ of degree at most d, such that $P(a_i) = b_i$ for all $i = 0, 1, \ldots, d$.*

Proof: First, note that if $a_i = a_j$, then b_i better equal b_j—else no polynomial can equal both b_i and b_j when evaluated at $a_i = a_j$.

For a contradiction, suppose there are two distinct polynomials $P(x)$ and $Q(x)$ of degree at most d such that for all i,

$$P(a_i) = Q(a_i) = b_i.$$

Then consider the polynomial $R(x) = P(x) - Q(x)$. It has degree at most d, since it is the difference of two polynomials of degree at most d. Moreover,

$$R(a_i) = P(a_i) - Q(a_i) = 0$$

for all the $d+1$ settings of $i = 0, 1, \ldots, d$. Once again, R is a polynomial of degree at most d, with $d+1$ roots. By the contrapositive of Theorem 1, $R(x)$ must be the zero polynomial. And hence $P(x) = Q(x)$, which gives us the contradiction. ■

To paraphrase the theorem differently, given two (i.e., $1+1$) points there is at most one linear (i.e., degree-1) polynomial that passes through them, given three (i.e., $2+1$) points there is at most one quadratic (i.e., degree-2) polynomial that passes through them, etc.

Can it be the case that for some $d+1$ pairs $(a_0, b_0), (a_1, b_1), \ldots, (a_d, b_d)$, there is *no* polynomial of degree at most d that passes through them? Well, clearly if $a_i = a_j$ but $b_i \neq b_j$. But what if all the a_i's are distinct?

Theorem 3 (Unique Reconstruction Theorem) *Given $d+1$ pairs $(a_0, b_0), (a_1, b_1), \ldots, (a_d, b_d)$ with $a_i \neq a_j$ for all $i \neq j$, there always exists a polynomial $P(x)$ of degree at most d, such that $P(a_i) = b_i$ for all $i = 0, 1, \ldots, d$.*

We will prove this theorem soon, but before that note some implcations. Given $d+1$ pairs $(a_0, b_0), (a_1, b_1), \ldots, (a_d, b_d)$ with distinct a's, this means there is a *unique* polynomial of degree at most d that passes through these points. Exactly one.

In fact, given $d+1$ numbers $b_0, b_1, \ldots, b_d$, there is a unique polynomial $P(x)$ of degree at most d such that $P(i) = b_i$. (We're just using the theorem with $a_i = i$.) Earlier we saw how to represent any polynomial of degree at most d by $d+1$ numbers, the coefficients. Now we are saying that we can represent the polynomial of degree at most d by a different sequence of $d+1$ numbers: its values at $0, 1, \ldots d$.

Two different representations for the same thing, cool! Surely there must be a use for this new representation. We will give at least two uses for this, but first let's see the proof of Theorem 3. (If you are impatient, you can skip over the proof, but do come back and read it—it is very elegant.)

3.1 The Proof of Theorem 3*

OK, now the proof. We are given $d+1$ pairs (a_i, b_i), and the a's are all distinct. The proof will actually give an algorithm to find this polynomial $P(x)$ with degree at most d, and where $P(a_i) = b_i$.

Let's start easy: suppose all the $d+1$ values b_i's were zero. Then $P(x)$ has $d+1$ roots, and now Theorem 1 tells us that $P(x) = 0$, the zero polynomial!

OK, next step. Suppose $b_0 = 1$, but all the d other b_i's are zero. Do we know a degree-d polynomial which has roots at d places $a_1, a_2, \ldots, a_d$. Sure, we do—it is just

$$Q_0(x) = (x - a_1)(x - a_2) \cdots (x - a_d).$$

So are we done? Not necessarily: $Q_0(a_0)$ might not equal $b_0 = 1$. But that is easy to fix! Just scale the polynomial by $1/Q_0(a_0)$. I.e., what we wanted was

$$\begin{aligned} R_0(x) &= (x - a_1)(x - a_2) \cdots (x - a_d) \cdot \frac{1}{Q_0(a_0)} \\ &= \frac{(x - a_1)(x - a_2) \cdots (x - a_d)}{(a_0 - a_1)(a_0 - a_2) \cdots (a_0 - a_d)} \quad . \end{aligned}$$

Again, $R_0(x)$ has degree d by construction, and satisfies what we wanted! (We'll call $R_0(x)$ the 0^{th} "switch" polynomial.)

Next, what if b_0 was not 1 but some other value. Easy again: just take $b_0 \times R_0(x)$. This has value $b_0 \times 1$ at a_0, and $b_0 \times 0 = 0$ at all other a_i's.

Similarly, one can define switch polynomials $R_i(x)$ of degree d that have $R_i(a_i) = 1$ and $R_i(a_j) = 0$ for all $i \neq j$. Indeed, this is

$$R_i(x) = \frac{(x-a_0)\cdots(x-a_{i-1})\cdot(x-a_{i-1})\cdots(x-a_d)}{(a_i-a_0)\cdots(a_i-a_{i-1})\cdot(a_i-a_{i-1})\cdots(a_i-a_d)} \quad .$$

So the polynomial we wanted after all is just a linear combination of these switch polynomials:

$$P(x) = b_0R_0(x) + b_1R_1(x) + \ldots + b_dR_d(x)$$

Since it is a sum of degree-d polynomials, $P(x)$ has degree at most d. And what is $P(a_i)$? Since $R_j(a_i) = 0$ for all $j \neq i$, we get $P(a_i) = b_iR_i(a_i)$. Now $R_i(a_i) = 1$, so this is b_i. All done.[1][2]

> **Example:** Consider the tuples $(5,1),(6,2),(7,9)$: we want the unique degree-2 polynomial that passes through these points. So first we find $R_0(x)$, which evaluates to 1 at $x = 5$, and has roots at 6 and 7. This is
>
> $$R_0(x) = \frac{(x-6)(x-7)}{(5-6)(5-7)} = \frac{1}{2}(x-6)(x-7)$$
>
> Similarly
>
> $$R_1(x) = \frac{(x-5)(x-7)}{(6-5)(6-7)} = -(x-5)(x-7)$$
>
> and
>
> $$R_2(x) = \frac{(x-5)(x-6)}{(7-5)(7-6)} = \frac{1}{2}(x-5)(x-6)$$
>
> Hence, the polynomial we want is
>
> $$P(x) = 1 \cdot R_0(x) + 2 \cdot R_1(x) + 9 \cdot R_2(x) = 3x^2 - 32x + 86$$
>
> Let's check our answer:
>
> $$P(5) = 1, P(6) = 2, P(7) = 9.$$

Running Time: Note that constructing the polynomial $P(x)$ takes $O(d^2)$ time. (Can you find the simplified version in this time as well?) If you chose the points $a_0, \ldots, a_d$ carefully, you could do this in $O(d \log d)$ time. For this, you choose 2^r roots of unity for $r = 2^{\lceil \log_2(d+1) \rceil}$, and then use a divide-and-conquer idea: the resulting algorithm is called the Fast Fourier Transform.

4 Application: Error Correcting Codes

Consider the situation: I want to send you a sequence of $d+1$ numbers $\langle c_d, c_{d-1}, \ldots, c_1, c_0 \rangle$ over a noisy channel. I can't just send you these numbers in a message, because I know that whatever message I send you, the channel will corrupt up to k of the numbers in that message. For the

[1] This algorithm is called Lagrange interpolation, after Joseph-Louis Lagrange, or Giuseppe Lodovico Lagrangia, depending on whether you ask the French or the Italians. (He was born in Turin, and both countries claim him for their own.) And to muddy things even further, much of his work was done at Berlin. He later moved to Paris, where he survived the French revolution—though Lavoisier was sent to the guillotine because he intervened on behalf of Lagrange. Among much other great research, he also gave the first known proof of Wilson's theorem, that n is a prime if and only if $n|(n-1)! + 1$—apparently Wilson only conjectured Wilson's theorem.

[2] You can use an idea very similar to Lagrange interpolation to prove the Chinese Remainder Theorem.

current example, assume that the corruption is very simple: whenver a number is corrupted, it is replaced by a $\star$. Hence, if I send the sequence

$$\langle 5, 19, 2, 3, 2\rangle$$

and the channel decides to corrupt the third and fourth numbers, you would get

$$\langle 5, 19, \star, \star, 2\rangle.$$

On the other hand, if I decided to delete the fourth and fifth elements, you would get

$$\langle 5, 19, 2, \star, \star\rangle.$$

Since the channel is "erasing" some of the entries and replacing them with $\star$'s, the codes we will develop will be called *erasure* codes. The question then is: how can we send $d+1$ numbers so that the receiver can get back these $d+1$ numbers even if up to k numbers in the message are erased (replaced by $\star$s)? (Assume that both you and the receiver know d and k.)

A simple case: if $d = 0$, then one number is sent. Since the channel can erase k numbers, the best we can do is to repeat this single number $k+1$ times, and send these $k+1$ copies across. At least one of these copies will survive, and the receiver will know the number.

This suggests a strategy: no matter how many numbers you want to send, repeat each number $k+1$ times. So to send the message $\langle 5, 19, 2, 3, 2\rangle$ with $k = 2$, you would send

$$\langle 5, 5, 5, 19, 19, 19, 2, 2, 2, 3, 3, 3, 2, 2, 2\rangle$$

This takes $(d+1)(k+1)$ numbers, approximately dk. Can we do better?

Indeed we can! We view our sequence $\langle c_d, c_{d-1}, \ldots, c_1, c_0\rangle$ as the $d+1$ coefficients of a polynomial of degree at most d, namely $P(x) = c_d x^d + c_{d-1}x^{d-1} + .. + c_1 x + c_0$. Now we evaluate P at some $d+k+1$ points, say $0, 1, 2, \ldots, d+k$, and send these $d+k+1$ numbers

$$P(0), P(1), \ldots, P(d+k)$$

across. The receiver will get back at least $d+1$ of these numbers, which by Theorem 3 uniquely specifies $P(x)$. Moreover, the receiver can also reconstruct $P(x)$ using, say, Langrange interpolation.

Example: Here is an example: Suppose we want to send $\langle 5, 19, 2, 3, 2\rangle$ with $k = 2$. Hence $P(x) = 5x^4 + 19x^3 + 2x^2 + 3x + 2$. Now we'll evaluate $P(x)$ at $0, 1, 2, \ldots d+k = 6$. This gives

$$P(0) = 2, P(1) = 31, P(2) = 248, P(3) = 947, P(4) = 2542, P(5) = 5567, P(6) = 10676$$

So we send across the "encoded message":

$$\langle 2, 31, 248, 947, 2542, 5567, 10676\rangle$$

Now suppose the third and fifth entries get erased. the receiver gets:

$$\langle 2, 31, \star, 947, \star, 5567, 10676\rangle$$

So she wants to reconstruct a polynomial $R(x)$ of degree at most 4 such that $R(0) = 2, R(1) = 31, R(3) = 947, R(5) = 5567, R(6) = 10676$. (That is, she wants to "decode" the message.) By Langrange interpolation, we get that

$$R(x) = \frac{1}{45}(x-1)(x-3)(x-5)(x-6) - \frac{31}{40}x(x-3)(x-5)(x-6) + \frac{947}{36}x(x-1)(x-5)(x-6)$$
$$-\frac{5567}{40}x(x-1)(x-3)(x-6) + \frac{5338}{45}x(x-1)(x-3)(x-5)$$

which simplifies to $P(x) = 5x^4 + 19x^3 + 2x^2 + 3x + 2$!

Note on the Running Time: The numbers can get large, so you may want work in the field $\mathbb{F}_p$, as long as the size of the field is large enough to encode the numbers you want to send across. (Of course, if you are working modulo a prime p, both the sender and the receiver must know the prime p.)

> **Example:** Since we want to send numbers as large as 19, let's work in $\mathbb{Z}_{23}$. Then you'd send the numbers modulo 23, which would be
>
> $$\langle 2, 8, 18, 4, 12, 1, 4\rangle$$
>
> Now suppose you get
>
> $$\langle 2, 8, \star, 4, \star, 1, 4\rangle$$
>
> Interpolate to get
>
> $$R(x) = 45^{-1}(x-1)(x-3)(x-5)(x-6) - 5^{-1}x(x-3)(x-5)(x-6) + 9^{-1}x(x-1)(x-5)(x-6)$$
> $$-40^{-1}x(x-1)(x-3)(x-6) + 2\cdot 45^{-1}x(x-1)(x-3)(x-5)$$
>
> where the multiplicative inverses are modulo 23, of course. Simplifying, we get $P(x) = 5x^4 + 19x^3 + 2x^2 + 3x + 2$ again.

4.1 Error Correction

One can imagine that the channel is more malicious: it decides to *replace* some k of the numbers not by stars but by other numbers, so the same encoding/decoding strategy cannot be used! Indeed, the receiver now has no clue which numbers were altered, and which ones were part of the original message! In fact, even for the $d = 0$ case of a single number, we need to send $2k + 1$ numbers across, so that the receiver knows that the majority number must be the correct one. And indeed, if you evaluate $P(x)$ at $n = d + 2k + 1$ locations and send those values across, even if the channel alters k of those numbers, there is a unique degree-d polynomial that agrees with $d+k+1$ of these numbers (and this must be $P(x)$)—this is not hard to show. What is more interesting is that the receiver can also reconstruct $P(x)$ fast: this is known as the *Berlekamp-Welch* algorithm. We will cover this if we have time.

4.1.1 The Berlekamp-Welch Error-Correction Algorithm*

Let $[n] := \{0, 1, \ldots, n-1\}$. Suppose we send over the n numbers $s_0, s_1, \ldots, s_{n-1}$, where $s_i = P(i)$. We receive numbers $r_0, r_1, \ldots, r_{n-1}$, where at most k of these r_is are not the same as the s_is. Define $Z := \{i \mid s_i \neq r_i\}$. Suppose we know $j := |Z|$ (you can always try all values $j = 0, 1, \ldots, k$).

Define a degree-j polynomial $E(x)$ such that

$$E(x) = \prod_{a \in S}(x - a).$$

Observe that

$$P(x) \cdot E(x) = b_i \cdot E(x) \qquad \forall x \in [n].$$

Indeed, $E(x) = 0$ for all $x \in S$ and $P(x) = r_i$ for all $x \in [n] \setminus S$. Of course, we just received the r_is, and don't know either $P(x)$ nor S, so we can't construct $E(x)$.

But we know that $E(x)$ looks like

$$E(x) = x^j + e_{j-1}x^{j-1} + \ldots + e_1 x + e_0. \tag{1}$$

for some values $e_{j-1}, e_{j-2}, \ldots, e_0$. Moreover, we know that $P(x) \cdot E(x)$ has degree $d+j$, so looks like

$$P(x) \cdot E(x) = x^{d+j} + f_{d+j-1}x^{j-1} + \ldots + f_1 x + f_0. \tag{2}$$

So we get $n = d + 2k + 1$ equalities that look like

$$x^j + e_{j-1}x^{j-1} + \ldots + e_1 x + e_0 = x^{d+j} + f_{d+j-1}x^{j-1} + \ldots + f_1 x + f_0, \tag{3}$$

one for each $x \in [n]$. The unknown are e_i and f_i values—there are $j + (d + j + 1) = d + 2j + 1 \leq d + 2k + 1$ unknowns. So we can solve for these using Gaussian elimination, and get $E(x)$ and $P(x) \cdot E(x)$. Dividing the latter by the former gives back $P(x)$. It's like magic.

Exercise: Show that there is a unique solution to the system of n inequalities in at most n variables.

5 Multivariate Polynomials and Matchings

Here's a very different application of polynomials in algorithm design. Now we'll consider multivariate polynomials, and use the fact that they also have "few" roots (where "few" is interpreted suitably) to get an unusual algorithm for finding matchings in graphs.

We can rephrase Theorem 1 as follows: if we fix a set S of values in the field we are working over (e.g., $\mathbb{R}$, or $\mathbb{F}_p$), and pick a random $X \in S$, the probability that $P(X) = 0$ is at most $d/|S|$. One can extend this to the following theorem for multivariate polynomials $P(\mathbf{x}) = P(x_1, x_2, \ldots, x_m)$. Recall that the degree of a monomial is the sum of exponents of the variables in it, and the degree of a polynomial is the maximum degree of any non-zero monomial in it.

Theorem 4 (Schwartz (1980), Zippel (1979)) *For any non-zero degree-d polynomial $P(\mathbf{x})$ and any subset S of values from the underlying field, if each X_i is chosen independently and uniformly from S, then*

$$\Pr[P(X_1, \ldots, S_m) = 0] \leq \frac{d}{|S|}.$$

This theorem is useful in many contexts. E.g., we get yet another algorithm for perfect matchings.

Definition 5 *For any graph $G = (V, E)$ with vertices $v_1, v_2, \ldots, v_n$, the* Tutte matrix[3] *and is a $|V| \times |V|$ matrix $M(G)$:*

$$M(G)_{i,j} = \begin{cases} x_{i,j} & \text{if } \{v_i, v_j\} \in E \text{ and } i < j \\ -x_{j,i} & \text{if } \{v_i, v_j\} \in E \text{ and } i > j \\ 0 & \text{if } (v_i, v_j) \notin E \end{cases}$$

This is a square matrix of variables $x_{i,j}$. And like any matrix, we can take its determinant, which is a (multivariate) polynomial $P_G(\mathbf{x})$ in the variables $\{x_{i,j}\}_{\{i,j\} \in E}$. The degree of this polynomial is at most $n = |V|$, the dimension of the matrix. Here is a surprising (and not difficult) fact:[4]

[3] Named after William T. (Bill) Tutte, pioneering graph theorist and algorithm designer. Recently it was discovered that he was one of the influential code-breakers in WWII, making crucial insights in breaking the Lorenz cipher.

[4] For bipartite graphs you can define the "Edmonds" matrix of $G = (U, V, E)$ is an $|U| \times |V|$ matrix $M_b(G)$ with the entry at row i and column j,

$$M_b(G)_{i,j} = \begin{cases} 0 & \text{if } (u_i, v_j) \notin E \\ x_{i,j} & \text{if } (u_i, v_j) \in E \end{cases}$$

Theorem 6 is easy to prove for this case. The proof for general graphs requires a little more care.

Theorem 6 (Tutte (1947)) *A graph G has a perfect matching if and only if $P_G(\mathbf{x})$, the determinant of the Tutte matrix, is not the zero polynomial.*

How do we check if $P_G(\mathbf{x})$ is zero or not? That's the problem: since we're taking a determinant of a matrix of variables, the usual way of computing determinants may lead to $n!$ terms, which eventually may all cancel out!

However, we can combine Theorems 4 and 6 together: take G, construct $M(\mathbf{x})$, and replace each variable by an independently uniform random value in some set S, and then compute the determinant of the resulting matrix of random numbers. This is exactly like plugging in the random numbers into $P_G(\mathbf{x})$. So if $P_G(\mathbf{x})$ was zero, the answer is zero for sure. And else, the answer is zero with probability at most $n/|S|$, which we can make as small as we want by choosing S large enough.

> **Exercise:** Think about how you would use this algorithm to find perfect matching (with high probability) in a graph, if one exists.
>
> **Exercise:** Given an algorithm to find perfect matchings in a graph (if one exists), use it to find maximum cardinality matchings in graphs.