15-451 / 15-651 Fall 2024

Algorithm Design and Analysis

Fall 2024 taught by

Daniel **Anderson**
Jason **Li**

Lecture notes by

Daniel **Anderson**
Avrim **Blum**
Anupam **Gupta**
Danny **Sleator**

Last updated October 10, 2024

Contents

Algorithm analysis and models of computation

1 Introduction and Linear-time Selection **1**
- 1.1 Goals of the Course . . . 2
- 1.2 Sorting and selection in the comparison model . . . 4
- 1.3 The Median and Selection Problems . . . 7

Exercises: Selection Algorithms & Recurrences **14**

2 Concrete models and lower bounds **15**
- 2.1 Terminology: Upper Bounds and Lower Bounds . . . 16
- 2.2 Selection in the comparison model . . . 17
- 2.3 Sorting in the comparison model . . . 19

3 Integer sorting **25**
- 3.1 Models of computation for integers . . . 26
- 3.2 Sorting small integers: Counting Sort . . . 28
- 3.3 A side quest: Tuple sorting . . . 30
- 3.4 Sorting bigger integers: Radix Sort . . . 32

Exercises: Integer Sorting **35**

Data structures and their analysis

4 Hashing: Universal and Perfect Hashing **37**
- 4.1 Dictionaries, hashing, and hashtables . . . 38
- 4.2 Universal Hashing . . . 40
- 4.3 More powerful hash families . . . 45
- 4.4 Perfect Hashing . . . 46

Exercises: Hashing **49**

5 Fingerprinting & String Matching **51**
- 5.1 How to Pick a Random Prime . . . 51
- 5.2 How Many Primes? . . . 52
- 5.3 The String Equality Problem . . . 53
- 5.4 The Karp-Rabin Algorithm (the “Fingerprint” method) . . . 56

Exercises: Fingerprinting **59**

6 Range query data structures **61**
6.1 Range queries . . . 61
6.2 Making range queries dynamic . . . 62
6.3 The data structure . . . 65
6.4 Speeding up algorithms with range queries . . . 67
6.5 Extensions of SegTrees . . . 68

7 Amortized Analysis **73**
7.1 The ubiquitous example: Dynamic arrays (lists) . . . 73
7.2 The Bankers Method . . . 76
7.3 The Potential Method . . . 77
7.4 Lists Revisited . . . 79
7.5 An Even-more-dynamic Array . . . 81

Exercises: Amortized Analysis **83**

8 Union-Find **85**
8.1 Motivation . . . 85
8.2 The Disjoint-Sets / Union-Find Problem . . . 86
8.3 The union-by-size optimization . . . 87
8.4 The path compression optimization . . . 88
8.5 Both optimizations at once . . . 93

Exercises: Union Find **94**

Algorithm design tools and techniques

9 Dynamic Programming I **95**
9.1 Introduction . . . 96
9.2 The Knapsack Problem . . . 98
9.3 Max-Weight Indep. Sets on Trees (Tree DP) . . . 100
9.4 Traveling Salesperson Problem (TSP) . . . 102

Exercises: Dynamic Programming **105**

10 Dynamic Programming II **107**
10.1 All-pairs Shortest Paths (APSP) . . . 107
10.2 Advanced Tree DP . . . 110
10.3 Longest Increasing Subsequence . . . 112

Exercises: Dynamic Programming II **114**

11 Network Flows I **115**
11.1 The Maximum Network Flow Problem . . . 115
11.2 The Ford-Fulkerson algorithm . . . 119
11.3 Bipartite Matching . . . 124

Exercises: Flow Fundamentals **126**

12 Network Flows II **127**
12.1 Network flow recap . . . 127
12.2 Shortest Augmenting Paths Algorithm (Edmonds-Karp) . . . 128
12.3 Dinic's Algorithm . . . 129
12.4 Dinic's algorithm for unit-capacity graphs . . . 133

13 Minimum-cost Flows **137**
13.1 Minimum-Cost Flows . . . 137
13.2 An augmenting path algorithm for minimum-cost flows . . . 139
13.3 An optimality criteria for minimum-cost flows . . . 141
13.4 Cycle canceling: Another algorithm for min-cost flow . . . 145

Contents

Lecture 1

Introduction and Linear-time Selection

The purpose of this lecture is to give a brief overview of the topic of algorithm analysis and the kind of thinking it involves. As a motivating problem, we consider the famous Quicksort algorithm and the problem of selecting a pivot. We review the complexity of Quicksort under various assumptions and use it to motivate the *selection* problem. This allows us to improve the Quicksort algorithm by deterministically finding the optimal pivot element (the median element) in linear worst-case complexity.

These problems illustrate some of the ideas and tools we will be using (and building upon) in this course. We will also practice writing and solving recurrence relations, which is a key tool for the analysis of algorithms.

Objectives of this lecture

In this lecture, we cover:

- Formal analysis of algorithms, models of computation,
- The analysis and complexity of the Quicksort algorithm,
- Finding the median: A randomized algorithm in expected linear time,
- Analyzing a randomized recursive algorithm,
- A deterministic linear-time algorithm for medians.

Recommended study resources

- CLRS, *Introduction to Algorithms*, Chapter 9, Medians and Order Statistics
- DPV, *Algorithms*, Chapter 2.4, Medians

1.1 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 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. 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. Algorithms that solve such problems are called *online* algorithms, which we will also discuss. 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.

1.1.1 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 another algorithm, without needing to worry whether the inputs on which it is being used now 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 non-obvious 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 for a given problem, 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 find an algorithm for the problem, and its opponent, the "adversary", is trying to find 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.

1.1.2 Formal analysis of algorithms

In this course we are interested in the *formal* analysis of algorithms. In your previous courses on algorithms and programming, you have hopefully studied the notion of the complexity of an algorithm, but you might have done it less formally. Most commonly when first learning about algorithm analysis, you learn to "count the number of operations" done by an algorithm. This is a bit vague and underspecified, but it does the job most of the time.

Our goal is to make this more precise to get a more rigorous view on what constitutes the complexity of an algorithm. To do this, we need the notion of a *model of computation*.

Definition: Model of computation

A model of computation consists of:

1. A set of allowed *operations* that an algorithm may perform, and
2. A cost for each operation, sometimes separately called the *cost model*.

To analyze the complexity of an algorithm, we consider it in a particular model of computation and add up the total costs of all the operations of that algorithm. Sometimes costs will be specified *concretely*, e.g., operation X costs 1 and operation Y costs 2. Other times we will only be interested in asymptotic bounds, so we may specify that an operation costs $O(1)$ time but be uninterested in constant factors in the analysis. We will see many examples in the first lectures.

1.2 Sorting and selection in the comparison model

For the first lecture we will consider algorithm in the *comparison model*. In the comparison model we assume that the input consists of some comparable elements (i.e., we can ask is $x < y$ for two elements x and y) but we can not assume anything else about their type. For example, we can not assume that they are integers or numbers at all, we can not assume that they are strings, or that we can hash them, etc. Comparing the elements is the *only* information we have about their values. We define the comparison model as follows.

Definition: Comparison Model

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

The comparison model is widely used to analyze sorting and selection (e.g., find the max, the second-max, the kth largest element, etc.) algorithms. On a practical level, sorting and selection algorithms designed for the comparison model are widely applicable because they make no assumptions about the types of the inputs, so they can be used to sort any types for which the problem of sorting makes sense (the elements must of course be comparable to define what sorting even means!) Conversely however, algorithms designed for the comparison model may be less efficient than algorithms which are specialized for specific types, so we get a tradeoff between generality and performance. We will see an example of this in a couple of lectures.

Since the problems of sorting and selection are fundamentally about ordering, defining the cost of the algorithms in terms of the number of comparisons performed is a natural metric. This number also *usually* (but not always) matches asymptotically the number of operations performed in a less concrete model of computation, so it provides a reasonable prediction of the algorithms' performance. The comparison model is also widely used for studying *lower bounds* on the complexity of sorting and selection problems, which we will study next lecture.

1.2.1 Revisiting a classic: The Quicksort algorithm

Quicksort is one of the most famous and well known algorithms in all of computer science.

Algorithm 1.1: Quicksort

Given array A of size n,

1. Pick an arbitrary pivot element p from A.
2. LESS $\leftarrow \{a_i$ such that $a_i < p\}$
3. GREATER $\leftarrow \{a_i$ such that $a_i > p\}$
4. **return** Quicksort(LESS) $+\{p\}+$ Quicksort(GREATER)

The step of splitting A into LESS and GREATER is called *partitioning*. The pseudocode given above gives the simplest version of the algorithm which copies/moves the elements into a new array rather than partitioning *in place*, which is more complicated, but the fundamental idea of the algorithm and its cost is the same.

We will use Quicksort as a starting point to review what we know about complexity analysis and to ensure that we do so rigorously. Then, we will spend the rest of the lecture figuring out how to improve the algorithm and make its complexity better! First, we need to make sure we remember how to measure complexity. Remember that there isn't a single measure of complexity, there are multiple, so we will review a bunch of them and see how they apply to Quicksort. Note that measures of complexity are orthogonal to the model of computation. We can consider any combination of both to obtain different ways of analyzing the same algorithm!

1.2.2 Worst-case complexity

Definition: Worst-case Cost

The *worst-case* complexity of an algorithm in a given model of computation is the largest cost that the algorithm could incur on any possible input, usually parameterized by the size of the input.

You might remember from your earlier classes that the worst-case cost of Quicksort occurs when the pivot selected happens to always be the smallest or largest element in the array.

Theorem: Worst-case cost of Quicksort

The worst-case cost of Quicksort is $O(n^2)$ comparisons.

1.2.3 Average-case complexity

The worst-case cost of Quicksort is bad, but it seems to perform well *most* of the time. This is not rigorous, but fortunately there is a formal way to state this using *average-case* complexity.

Definition: Average-case Cost

The *average-case* complexity of an algorithm in a given model of computation is the average of the costs of all possible inputs to the algorithm, usually parameterized by the size of the input.

Remark: Random interpretation of average-case cost

By definition, the average-case cost of an algorithm is equivalent to the *expected value* of its cost over a uniform distribution of possible inputs. You can therefore think of the average cost as the cost of the algorithm on a random input.

Computing average-case complexity tends to be quite a lot more work than computing worst-case complexity since we need a way to enumerate all possible inputs. This can be done for Quicksort using recurrence relations, and you may have seen this in a previous class. We won't repeat the analysis here.

Theorem: Average-case cost of Quicksort

The average-case cost of Quicksort is $O(n \log n)$ comparisons.

1.2.4 Randomized complexity

One interpretation of the average-case cost of Quicksort is that if we run it on a random input then the expected cost is just $O(n \log n)$. This is great... if the input to the algorithm is random. However, real life data is almost never random so it is a bad idea to design your algorithms with the assumption that the input is random! Thankfully, there is a simple yet powerful way to overcome this foolish assumption: put the randomess *in the algorithm* instead! This leads to *randomized Quicksort*, a variant of the algorithm that does not perform horribly for any particular input! The difference is just a single word compared to the vanilla variant.

Algorithm 1.2: RandomQuicksort

Given array A of size n,

1. Pick a **random** pivot element p from A.
2. LESS $\leftarrow \{a_i$ such that $a_i < p\}$
3. GREATER $\leftarrow \{a_i$ such that $a_i > p\}$
4. **return** RandomQuicksort(LESS) $+\{p\}+$ RandomQuicksort(GREATER)

Unlike the previous algorithms, this one now includes randomness, which means that its cost on a particular input is not fixed, it could vary from one run to the next! More formally, the runtime of the algorithm is now a probability distribution rather than a fixed number. When describing the complexity of a randomized algorithm, we usually give some property of the probability distribution, most commonly, we use the *expected value*. By default, unless otherwise specified, we still consider the *worst-case* input, i.e., we want to compute the maximum over all possible inputs, of the expected value of the cost of the algorithm over the random choices made by the algorithm. We refer to this as the *expected complexity* or *expected cost* of the algorithm.

Definition: Expected Cost

The *expected* complexity of a randomized algorithm in a given model of computation is the maximum cost over of all possible inputs to the algorithm, of the expected value of the cost of that input, where the expected value is over the distribution of random choices made by the algorithm.

Remark: Misconceptions of expected cost

The definition of expected complexity is a bit tricky, so it is important to not have misconceptions about it. Here are some important things to keep in mind:

1. Expected complexity by default considers *worst-case inputs*. We are not analyzing the algorithm for a random/average-case input.
2. We are also not considering worst-case random numbers. The input is worst case, and the randomness is well... random. The expected value is computed over the distribution of the random choices of the algorithm.

Theorem: Expected cost of Randomized Quicksort

The expected cost of Randomized Quicksort is $O(n \log n)$ comparisons.

The proof is almost identical to that of the average-case cost of (non-randomized) Quicksort since there is essentially a one-to-one mapping between random inputs and randomly selecting pivots. You have probably run across the proof previously so we again won't repeat it here.

1.2.5 Can we do better?

We have watched the analysis of Quicksort go from $O(n^2)$ worst-case to $O(n \log n)$ average-case, to expected $O(n \log n)$ by mixing randomization into the algorithm. Can we (theoretically at least) improve the algorithm even further or is this the end of the journey? Well, we know several other comparison-based sorting algorithms that run in $O(n \log n)$ comparisons, like Mergesort and Heapsort, and both of them are deterministic! It would be very cool if Quicksort could also be made to take just $O(n \log n)$ comparisons and still be deterministic too. To achieve this, we would need to find a balanced partition (i.e., pick a good pivot that splits the input into similarly sized halves). That motivates us to consider the **median-finding** problem. If we could find the median of an unsorted array in linear time, we could use that as the pivot and all of our dreams would come true! That will be the remainder of this lecture.

1.3 The Median and Selection Problems

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. An example of this is median finding. Recall the concept of the median of a set. For a set of n elements, this is the "middle" element in the set, i.e., there are exactly $\lfloor n/2 \rfloor$ elements larger than it. In computer sciencey terms, if the elements are zero-indexed, the median is the $\lfloor (n-1)/2 \rfloor^{\text{th}}$ element of the set when represented in sorted order.

Given an unsorted array, how quickly can one find the median element? Perhaps the simplest solution, one that can be described in a single sentence and implemented with one or two lines of code in your favorite programming language, is to sort the array and then read off the

element in position $\lfloor(n-1)/2\rfloor$, which takes $O(n \log n)$ comparisons using your favorite sorting algorithm, such as MergeSort or HeapSort (deterministic) or QuickSort (randomized).[1]

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

1.3.1 The problem and a randomized solution

Let's consider a problem that is slightly more general than median-finding:

Problem: Select-k / k^{th} Smallest

Find the kth smallest element in an unsorted array of size n.

To remove ambiguity, we will assume that our array is zero indexed, so the kth smallest element is the element that would be in position k if the array were sorted. Alternatively, it is the element such that exactly k other elements are smaller than it. Additionally, let's say all elements are distinct to avoid the question of what we mean by the kth smallest when we have duplicates.

The key idea to obtaining a linear time algorithm is to identify and eliminate redundant/wasted work in the "naive" algorithm that just sorts the input and outputs the kth element. Note that by sorting the array, we are not just finding the kth smallest element for the given value of k, we are actually finding the answer for *every possible* value of k. So instead of completely sorting the array, it would suffice to "partially sort" the array such that element k ends up in the correct position, but the remaining elements might still not be perfectly sorted. This description sounds suspiciously similar to Quicksort, which partitions the array by putting the pivot element into its correctly sorted position in the array (without guaranteeing yet that the rest of the array is sorted). In essence, this is partially sorting the array with respect to the pivot. Recursively sorting both sides then sorts the entire array.

So, what if we were to just run Quicksort, but skip some of the steps that do not matter? Specifically, note that if we run Quicksort and our goal is to output the kth element at the end, that after partitioning the array around the pivot, we know which of the two sides must contain the answer. Therefore instead of recursively sorting both sides, we ignore the side that can not contain the answer and recurse just once. That's it!

More specifically, the algorithm chooses a random pivot and then partitions the array into two sets LESS and GREATER consisting of those elements less than and greater than the pivot respectively. 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. 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

[1] Of course using a sorting algorithm to find the pivot for Quicksort would be rather useless, but it is a good start to get some intuition on the complexity of the median problem. We know that the problem is solvable in $O(n \log n)$, so our goal remains to bring this down to $O(n)$.

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.

Algorithm 1.3: QuickSelect

Given array A of size n and integer $0 \le k \le n-1$,

1. Pick a pivot element p at random from A.
2. Split A into subarrays LESS and GREATER by comparing each element to p.
3. (a) If $|\text{LESS}| > k$, then return QuickSelect(LESS, k).
   (b) If $|\text{LESS}| < k$, then return QuickSelect(GREATER, $k-|\text{LESS}|-1$)
   (c) If $|\text{LESS}| = k$, then return p. [always happens when $n=1$]

Theorem 1.1

The expected number of comparisons for QuickSelect is at most $8n$.

Formally, let $T(n)$ denote the expected number of comparisons performed by QuickSelect on any (worst-case) input of size n. What we want is a recurrence relation that looks like

$$T(n) \le n-1+\mathbb{E}[T(X)],$$

where $n-1$ comparisons come from comparing the pivot to every other element and placing them into LESS and GREATER, and X is a random variable corresponding to the size of the subproblem that is solved recursively. We can't just go and solve this recurrence because we don't yet know what X or $\mathbb{E}[T(X)]$ look like yet.

Before giving a formal proof, here's some intuition. First of all, how large is X, the the size of the array given to the recursive call? It depends on two things: the value of k and the *randomly-chosen pivot*. After partitioning the input into LESS and GREATER, whose size adds up to $n-1$, the algorithm recurively calls QuickSelect on one of them, but which one? Since we are interested in the behavior for a *worst-case input*, we can assume pessimistically that the value of k will always make us choose the bigger of LESS and GREATER. Therefore the question becomes: if we choose a random pivot and split the input into LESS and GREATER, how large is the larger of the two of them? Well, possible sizes of the splits (ignoring rounding) are

$$(0,n-1),(1,n-2),(2,n-3),\ldots,(n/2-2,n/2+1),(n/2-1,n/2),$$

so we can see that the larger of the two is a random number from

$$n-1,n-2,n-3,\ldots,n/2+1,n/2.$$

So, the expected size of the larger half is about $3n/4$, again, ignoring rounding errors.

Another way to say this is that if we split a candy bar at random into two pieces, the expected size of the larger piece is 3/4 of the bar. Using this, we might be tempted to write the following recurrence relation, which is almost correct, but not quite.

$$T(n) \leq n - 1 + T(3n/4).$$

If we go through the motions of solving this recurrence, we get $T(n) = O(n)$, but unfortunately, this derrivation is not quite valid. The reasoning is that $3n/4$ is only the *expected* size of the larger piece. That is, if X is the size of the larger piece, we have written a recurrence where the cost of the recursive call is $T(\mathbb{E}[X])$, but it was supposed to be $\mathbb{E}[T(X)]$, and these are not the same thing! (The exercise below shows the two could differ by a lot.)[2] Let's now see this a bit more formally.

Proof Theorem 1.1. To correct the proof, we need to correctly analyze the *expected value of* $T(X)$, rather than the value of T for the expected value of X. To do so, we can consider with what probabilities does X take on certain values and analyze the corresponding behavior of T. So, with what probability is X at most 3/4? This happens when the smaller of LESS and GREATER is at least one quarter of the elements, i.e., when the pivot is not in the bottom quarter or top quarter[3]. This means the pivot needs to be in the middle half of the data, which happens with probability 1/2. The other half the time, the size of X will be larger, at most $n-1$. Although this might sound rather loose, this is good enough to write down a good upper bound recurrence!

$$\mathbb{E}[T(X)] \leq \frac{1}{2} T\left(\frac{3n}{4}\right) + \frac{1}{2} T(n)$$

This sounds like too loose of a bound, but we will see that it is good enough. Returning to our original recurrence, we can now correctly assert that

$$T(n) \leq n - 1 + \left(\frac{1}{2} T\left(\frac{3n}{4}\right) + \frac{1}{2} T(n)\right),$$

and multiplying both sides by 2 then subtracting $T(n)$, we obtain

$$T(n) \leq 2(n-1) + T\left(\frac{3n}{4}\right).$$

We can now use induction to prove that this recurrence relation satisfies $T(n) \leq 8n$. The base case is simple, when $n = 1$ there are no comparisons, so $T(1) = 0$. Now assume for the sake of

[2] It turns out that these two quantities are equal if T is linear, which it is in this particular case. However, we can not make this assumption in the proof, because that is we are trying to prove in the first place! Assuming that T is linear to prove that T is linear would be circular logic.

[3] Note that we are picking 3/4 here purely for convenience. You could use any constant and still prove a linear bound on the number of comparisons.

induction that $T(i) \leq 8i$ for all $i < k$. We want to show that $T(k) \leq 8k$. We have

$$\begin{aligned} T(k) &\leq 2(n-1) + T\left(\frac{3n}{4}\right), \\ &\leq 2(n-1) + 8\left(\frac{3n}{4}\right), && \textit{Use the inductive hypothesis} \\ &\leq 2n + 6n, \\ &= 8n, \end{aligned}$$

which proves our desired bound.

□

1.3.2 A deterministic linear-time selection 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.[4]

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 (i.e., some kind of "approximate median"). We will use a technique called the *median of medians* which takes the medians of a bunch of small groups of elements, and then finds the median of those medians. It has the wonderful guarantee that the selected element is greater than at least 30% of the elements of the array, and smaller than at least 30% of the elements of the array. This makes it work great as an approximate median and therefore a good pivot choice! The algorithm is as follows:

Algorithm 1.4: 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 in the same way as Quickselect.

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

Theorem 1.2

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

Proof of Theorem 1.2. Let $T(n)$ denote the worst-case number of comparisons performed by the DeterministicSelect algorithm on inputs of size n.

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:

$$T(n) \quad \leq \quad cn + T(n/5) + T(7n/10),$$

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. Suppose the array has 15 elements and breaks down into three groups of 5 like this:

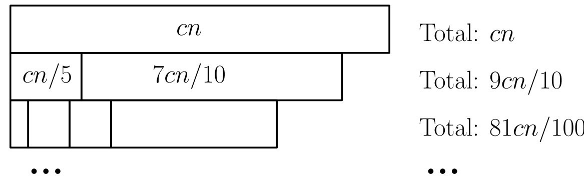
$$\{1,2,3,10,11\}, \ \{4,5,6,12,13\}, \ \{7,8,9,14,15\}.$$

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

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

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

In particular, let's build the recursion tree for the recurrence (1.1), 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. $\square$

Notice that in our analysis of the recurrence (1.1) 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.

Lemma 1.1

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

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

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

1.3.3 Optimal deterministic Quicksort

Armed with a deterministic linear-time median-finding algorithm, we now have the tools to create an $O(n \log n)$-cost deterministic Quicksort! Just use the linear-time median algorithm (DeterministicSelect) to select the pivot, then the divide-and-conquer subproblems are at most half the input, which gives us a cost of

$$T(n) = T\left(\frac{n}{2}\right) + T\left(\frac{n}{2}\right) + O(n).$$

Solving this using standard techniques (or remembering the solution from a previous class!) shows us that the cost is $O(n \log n)$ with no randomness required!

Exercises: Selection Algorithms & Recurrences

Problem 1. Recall the attempted analysis of randomized quickselect where we accidentally assumed that $\mathbb{E}[T(X)] = T(\mathbb{E}[X])$. Let X be a random variable, and find an increasing function $F : \mathbb{R}_+ \to \mathbb{R}_+$ such that $\mathbb{E}[F(i)] \gg F(\mathbb{E}[i])$.

Problem 2. 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

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

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 $T(n) = \Theta(n \log n)$.

Problem 3. Consider the median of medians algorithm. What happens if we split the elements into $n/3$ groups of size 3 instead? Or n/k groups of size k for larger odd values of k?

Problem 4. The rank of an element a with respect to a list A of n distinct elements is $|\{e \in A \mid e \leq a\}$ is the number of elements in A no greater than a. Hence the rank of the smallest element in A is 1, and the rank of the median is $n/2$. Given an unsorted list A and indices $i \leq j$, give an $O(n)$ time algorithm to output all elements in A with ranks lying in the interval $[i, j]$.

Lecture 2

Concrete models and lower bounds

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 find the largest item in an array,
- The number of comparisons needed to sort an array,

Objectives of this lecture

In this lecture, we want to:

- Understand some concrete models of computation (e.g., the comparison model)
- Understand the definition of a *lower bound* in a specific model
- See some examples of how to prove lower bounds in specific models, particularly for sorting and selection problems

Recommended study resources

- CLRS, *Introduction to Algorithms*, Chapter 8.1, Lower bounds for sorting
- DPV, *Algorithms*, Chapter 2.3, Mergesort (Page 59)

2.1 Terminology: Upper Bounds and Lower Bounds

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. Each model will have some operations that cost a certain amount (like performing a comparison, or swapping a pair of elements), some that are free, and some that are not allowed at all.

Definition: Upper bound

By an *upper bound* of U_n for some problem and some length n, we mean that *there exists an algorithm A that for every input x of length n costs at most* U_n.

A lower bound for some problem and some length n, is obtained by the negation of an upper bound for that n. It says that some upper bound is not possible (for that value of n). If we take the above statement (in italics) and negate it, we get the following. *for every algorithm A there exists an input x of length n such that A costs more than* U_n *on input x*. Rephrasing:

Definition: Lower bound

By a *lower bound* of L_n for some problem and some length n, we mean that *for any algorithm A there exists an input x of length n on which A costs at least* L_n *steps.*

These were definitions for a single value of n. Now a function $f : \mathbb{N} \to \mathbb{R}$ is an upper bound for a problem if $f(n)$ is an upper bound for this problem for every $n \in \mathbb{N}$. And a function $g(\cdot)$ is an lower bound for a problem if $g(n)$ is a lower bound for this problem for every n.

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 $f(n)$.

Finally, what is the *cost* of an algorithm? As we said before, that depends on the particular model of computation we're using. We will consider different models below, and show each has their own upper and lower bounds.

One natural model for examining problems like sorting and selection is the comparison model from last lecture, which we recall as follows.

Definition: Comparison Model

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

2.2 Selection in the comparison model

2.2.1 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: Upper bound on select-max in the comparison model

$n-1$ 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 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, so this is not tight. In fact, we can give a better lower bound:

Claim: Lower bound on select-max in the comparison model

$n-1$ comparisons are necessary for any deterministic algorithm in the worst-case to find the maximum of n elements.

Proof. The key claim is that every item that is not the maximum must lose at least one comparison (by lose, we mean it is compared to another element and is the lesser of the two). Why is this true? Suppose there were two elements a_i and a_j and neither lost a comparison. Suppose without loss of generality that $a_i > a_j$. If the algorithm outputs a_j it is incorrect. Otherwise, if it outputs a_i then we could construct another input that is the same except that a_j is now the maximum (we don't change the relative order of any other elements). On this new input, none of the results of any comparisons change since a_j never lost any comparisons in the first place, so the algorithm, being deterministic, must output the same answer. However, the algorithm is now incorrect. Therefore there must be $n-1$ elements that lose a comparison, and since only one element loses per comparison, a correct algorithm must perform $n-1$ comparisons. □

Since the upper and lower bounds are equal, the bound of $n-1$ is *tight*.

2.2.2 An alternate technique: decision trees

Our lower bound arguments so far have been based on an *adversary* technique. We argued that if an algorithm makes too few comparisons, then we can concoct an input such that it will produce the wrong answer. There are many techniques that can be used to prove lower bounds. Another powerful one are *decision trees*.

A decision tree is a binary tree that represents the behavior of a specific algorithm based on the outcomes of each comparison it makes. Specifically, each internal node corresponds to a comparison such that the left subtree corresponds to the outcome of the comparison being true and the right subtree corresponds to it being false. At a leaf node the algorithm performs no more comparisons and thus is finished and produces an output.

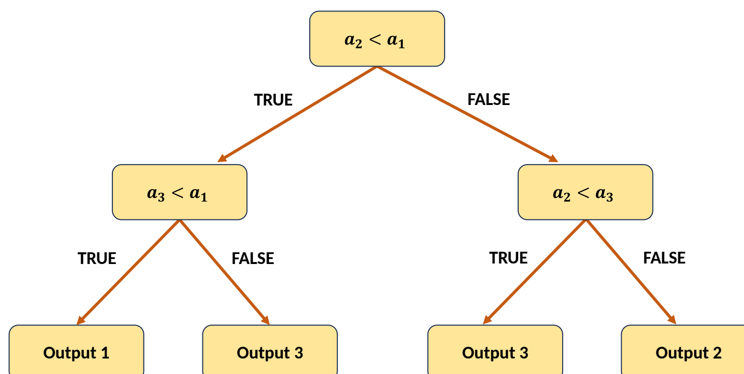
Remark: Decision trees are for particular algorithms

It is very important to remember that a decision tree encodes **a specific algorithm**. Different algorithms will have different decision trees. The decision tree does not however depend on the input to the algorithm, it encodes its behavior on any possible input. In some sense, you can think of the decision tree as a flow chart that tells you exactly what the algorithm does based on the results of the comparisons.

It turns out that decision trees can be a useful tool for analyzing lower bounds. Keeping in mind that a decision tree always represents a particular algorithm, to prove a lower bound, we must argue some property about the structure of *any possible decision tree* for the problem (if we make an argument about a specific decision tree, that is just like arguing about a specific algorithm, which does not help us derive a lower bound for the problem).

Since we are interested in the worst-case number of comparisons, we should observe that the number of comparisons performed by the algorithm on a particular input is the *depth* of the leaf node corresponding to that output. Therefore the worst-case cost (number of comparisons) of the algorithm corresponds exactly to the *longest root-to-leaf path*, i.e., the height of the tree. Therefore, if we can successfully argue about the height of any possible decision tree for a problem, we have an argument for a lower bound!

Here is a decision tree for some arbitrary algorithm that solves the select-max problem.



You can follow it just like a flowchart to determine for any input what index the algorithm will output! We can also use it to argue about lower bounds.

Proof of select-max lower bound via decision trees. At the root node of any decision tree for the select-max problem there are n possible outputs (positions $1 \dots n$). For each comparison, exactly one element loses, and hence the set of possible outputs at each node is one fewer than

at its parent node. Therefore all of the leaves of this decision tree have depth $n-1$ and hence $n-1$ comparisons are required to determine the maximum element. □

2.3 Sorting in the comparison model

For the problem of *sorting* in the comparison model, the input is an array $a = [a_1, a_2, \dots, a_n]$, and the output is a permutation of the input $\pi(a) = [a_{\pi(1)}, a_{\pi(2)}, \dots, a_{\pi(n)}]$ in which the elements are in increasing order.

Remark: Correctly defining the "output" of an algorithm

A surprisingly subtle aspect of proving lower bounds, and the source of many buggy or incorrect lower bound proofs is the seemingly simple step of defining what the *output* of the algorithm is supposed to be.

Remember, importantly, that the comparison model has no concept of "values" of the input elements. The *only* thing that an algorithms knows about them are the results of the comparisons. Therefore when a comparison-model sorting algorithm produces an output, it doesn't know the values of the elements, it only knows what order it rearranged them into – so its output can only be described as a permutation of the input elements. Many of our lower bound proofs will be combinatoric in nature, where we will count the number of *required outputs that an algorithm could need to produce.*

For example, suppose we ask an algorithm to sort $[c, a, b, d]$ and $[b, d, a, c]$. Both of these will become $[a, b, c, d]$ when sorted, so does this mean they were the same output? **No!** The former is sorted by outputting $[a_2, a_3, a_1, a_4]$, and the latter is sorted by outputting $[a_3, a_1, a_4, a_2]$, so these are *not the same output.*

On the other hand, suppose we ask to sort both $[c, a, b, d]$ and $[m, d, e, z]$. These will sort to $[a, b, c, d]$ and $[d, e, m, z]$, which are *both* the permutation $[a_2, a_3, a_1, a_4]$. So these are in fact **the same output**, because their elements are in the same relative permuted order, and the actions taken by a deterministic sorting algorithm on them would therefore be 100% identical (the algorithm could not tell the difference between those two inputs.)

When thinking about comparison-model lower bound proofs, be sure to keep this important distinction in mind – values do not matter at all because the algorithm does not know them. It can only deduce/know information about relative order!

Theorem 2.1: Lower bound for sorting in the comparison model

Any deterministic comparison-based sorting algorithm must perform at least $\lg(n!)$ comparisons to sort n distinct elements in the worst case.[a]

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

In other words, for any deterministic comparison-based sorting algorithm $\mathscr{A}$, for all $n \geq 2$ there exists an input I of size n such that $\mathscr{A}$ 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. This is a difficult task, and its not at all obvious how to even begin to do something like this! 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 of Theorem 2.1. First remember that we are dealing with *deterministic algorithms* here. Since the algorithm is deterministic, the first comparison it makes is always the same. Depending on the result of that comparison, the algorithm could take different actions, however, critically, **the result of all the previous comparisons always determines which comparison will be made next.** Therefore for any given input to the algorithm, we could write down the sequence of results of the comparisons (e.g., True, False, True, True, False, ...) and this sequence would entirely describe the behavior and hence the output of the algorithm on that input.

Now, in the comparison model, since values are unimportant and only order matters, there are $n!$ different possible input sequences that the algorithm needs to be capable of sorting correctly, one for each possible permutation of the elements. Furthermore, **every input permutation has a unique output permutation that correctly sorts it**. So, for a comparison-based sorting algorithm to be correct, it needs to be able to produce $n!$ different possible output permutations, because if there is an output it can not produce, then there is an input which it can not sort.

If the algorithm makes ℓ comparisons whose results are encoded by a sequence of binary outcomes (True or False) $b_1, b_2, \ldots, b_\ell$, then since each comparison has only two possible outcomes, the algorithm can only produce 2^ℓ different outputs. Since we argued that in order to be correct the algorithm must be capable of producing $n!$ different outputs, we need

$$2^\ell \geq n! \qquad \Longrightarrow \qquad \ell \geq \lg n!,$$

which proves the theorem. □

Key Idea: Information-theoretic lower bound

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 which of M outputs we need to produce. This technique generalizes: If we have some problem with M different outputs the algorithm needs to be able to produce, then in the comparison model we have a worst-case lower bound of $\lg M$.

What does $\lg(n!)$ look like? We have:

$$\lg(n!) = \lg(n) + \lg(n-1) + \lg(n-2) + \ldots + \lg(1) < n\lg(n) = O(n\log n),$$

and

$$\lg(n!) = \lg(n) + \lg(n-1) + \lg(n-2) + \ldots + \lg(1) > (n/2)\lg(n/2) = \Omega(n\log n).$$

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

How Tight is this Bound? Assume n is a power of 2, 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, we get $(n-1) + 2(n/2-1) + 4(n/4-1) + \ldots + n/2(2-1) = n\lg n - (n-1) < n\lg n$.

2.3.1 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.

2.3.2 Sorting with duplicates

The analysis of sorting with n distinct elements was surprisingly simple because we were able to characterize all of the possible inputs as all $n!$ permutations which all required a distinct output, and therefore argue that any correct algorithm therefore must be able to produce $n!$ distinct outputs. Most of the time it will not be this simple and we will need to take some extra steps. Here's a problem to demonstrate:

Problem: Sorting with D distinct elements

Suppose you have an array of n elements $a_1, \dots, a_n$ and a parameter D such that you are guaranteed that there are at most D distinct elements in the array (where $1 \leq D \leq n$.)

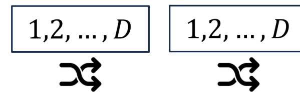
When $D = n$, it is the original sorting problem from before, which has a lower bound of $\Theta(n \log n)$, so this generalizes the previous problem by allowing duplicates in a constrained way. For $D = 1$, the array would consist of copies of a single element, which could be sorted in zero comparisons since it would be already sorted. For $D = 2$, we could sort the array in linear cost by scanning over the array and grouping the elements of the first value and second value. So it appears that the problem is cheaper as D gets smaller which makes sense; the fewer distinct elements, the more possible sorted orders there are so fewer outputs are required.

What makes this problem tricky is that it is very unclear how to count exactly how many required outputs there are. It is no longer true that each input element requires a distinct output; for example, both the arrays $[a, a, b]$ and $[a, b, b]$ are sorted by the identity permutation (they are already sorted). So it is **not** the case that we can just count the number of possible inputs and assume that it is equal to the number of required outputs, since a single output could sort multiple inputs.

Finding a hard subset of inputs A powerful technique that we will use to overcome this issue is to focus on a *specific subset of inputs* to the problem. If we find some family I of inputs and prove a lower bound on the cost to solve any input from I, then of course that lower bound also applies to solving the whole problem (all possible inputs). What properties do we want this family to have? Ideally two things:

- **It needs to require a lot of outputs**: The information-theoretic lower bound uses the number of required outputs, so we need a subset that requires a lot, otherwise we will get a very weak (low) lower bound.
- **It needs to be simple enough to count**: Since we are required to count the number of required outputs, our family of inputs should be simple enough that we can actually count that number! If our construction is too complicated, it will be too hard, so we usually try to construct something that is easy to describe and count with combinatorics tools that we have (factorials, binomials, powers, etc).

Constructing a family with the distinctness property The vanilla sorting problem was nice because every input required a distinct output, which made counting the number of outputs equivalent to counting the number of inputs. A common technique is therefore to try to construct our family of inputs so that it too has this property. We don't *always* have to do this, we could instead construct a family of inputs and then try to reason about *how many* outputs sort each input, and then divide the total number of outputs by that. For now, we will use the first technique. An important fact to keep in mind is that a permutation of distinct elements always has a unique inverse, i.e., it requires a distinct permutation to sort it. So, how can we generalize that idea to arrays with duplicates? What if we just glue two permutations together?



In other words, we take the elements $1,\ldots,D$ twice, then we randomly shuffle the first half and the second half independently. This gives us an array consisting of two copies each of $1,\ldots,D$, but with the extra property that there is only one 1 in the first half and one in the second half, and so on for each element.

An important fact about this construction is that given two different arrays generated by this process, **the same output can not sort both of them**. This is because if two elements on one side were in different positions, then the output permutation would sort those elements into the wrong order because they are unique!

Constructing our family of inputs Generalizing the above idea, given n and D, we can construct a family of inputs by taking n/D independently shuffled permutations of $1\ldots D$ and concatenating them together (if D does not divide n, the last group might stop early and not contain all of $1\ldots D$, that's fine). By the reasoning above, every input in this family **requires a distinct output**, i.e., no one output can correctly sort two of these inputs. So, the number of requires outputs to sort everything in this set is equal to the number of inputs in this family!

It remains to just count how many inputs are in this family. In each contiguous chunk containing $1\ldots D$, there are $D!$ possible orders, and there are n/D chunks. So, the total number of inputs in this family is

$$(D!)^{\frac{n}{D}}.$$

Obtaining the lower bound Applying the information-theoretic lower bound, any algorithm in the comparison model for solving this problem therefore requires

$$\log_2\left((D!)^{\frac{n}{D}}\right) = \frac{n}{D}\log_2(D!) = \frac{n}{D}\Theta(D\log D) = \Theta(n\log D)$$

comparisons! This intuitively makes sense, since if $D = n$ we get $\Theta(n\log n)$ which we should, since that is the problem from earlier of just sorting n distinct elements, and if D is smaller, the cost goes down. For example, if $D = 1$, then $\log_2(D) = 0$ which is correct since it takes no comparisons to sort an input consisting of entirely duplicates (it is already sorted).

As an exercise, try to come up with an algorithm that solves this problem in $\Theta(n\log D)$ comparisons, which proves that this bound is asymptotically tight.

Lecture 3
Integer sorting

Last lecture we saw a general lower bound which says that any *comparison model* sorting algorithm costs at least $\Omega(n \log n)$ in the worst case. In this lecture we will derive sorting algorithms that run in $O(n)$ cost! The catch is that to beat the comparison model lower bound we must of course leave the comparison model behind and explore other models of computation. Specifically, we will be looking at the problem of sorting integers, which gives us more power then the comparison model since we can use properties of integers to help us sort them faster. We will show two algorithms, Counting Sort and Radix Sort, which are capable of sorting integers in linear time, provided that those integers are not too large.

Objectives of this lecture

In this lecture, we want to:

- See the *Word RAM model of computation* for integer (non-comparison) algorithms
- Learn about the *counting sort* algorithm for sorting (small) integers
- Learn about the *radix sort* algorithm for sorting (slightly bigger) integers

Recommended study resources

- CLRS, *Introduction to Algorithms*, Chapter 8: Sorting in Linear Time

3.1 Models of computation for integers

In the first two lectures we have predominantly concerned ourselves with algorithms in the *comparison model* of computation, where the input to our algorithms consisted of an array of n comparable elements. In this model we didn't have any other information available to us about the elements. Were they integers, numbers, strings, who knows? This made the model very general since we could derive algorithms for sorting and selection that effectively work on *any type* because absolutely no assumptions about the type were made, only that they were comparable, which is essentially required by the definition of sorting!

Last lecture we saw a very cool and fundamental result which is that in the comparison model, sorting *can not* be done in less than $\Omega(n \log n)$ cost. We proved that it was mathematically impossible to invent a sorting algorithm that runs faster than this. Today, we want to invent some sorting algorithms than run in $O(n)$ cost! To do so without contradicting ourselves we are going to have to leave the comparison model behind and explore models of computation that give us more power. Specifically, we are going to work on the problem of sorting *integers*, which means that unlike in the comparison model where all we could do was compare two elements, we will now gain the power to do things to the input elements like add them, divide them, use them as indices into arrays, etc. These new found powers, and in particular the last one (using the items as indices) will be the tools to beating the comparison model. Our model of computation for today (and implicitly for much of the rest of the course) is the *word RAM model*.

3.1.1 The Word RAM model of computation

Definition: Word RAM model

In the word RAM model:

- We have unlimited constant-time addressable memory (called "registers"),
- Each register can store a w-bit integer (called a "word"),
- Reading/writing, arithmetic, logic, bitwise operations on a constant number of words takes constant time,
- With input size n, we need $w \geq \log n$.

The final assumption is needed because if our input contains n words, then to be capable of even reading the contents of the input, we are surely going to need to be able to write down the integer n as an index, and that requires $\log n$ bits. Since the word size is w, the maximum integer we can store in a single word $2^w - 1$.

Note that unlike some of our previous models such as the comparison model which had concrete costs, e.g., exactly 1 per comparison, in the word RAM we only care about asymptotic costs, so we ignore constant factors and just say that operations on a constant number of words takes constant time. This model is essentially just a more formal version of what you are probably used to from your previous classes when you analyzed an algorithm by counting "instruc-

tions". The only subtle part is the restriction on the word size w and the assumption that only operations on w-bit integers take constant time. Most of the time this is of no consequence, but there are some situations where it matters.

The importance of the word size Consider for example an algorithm that takes n integers as inputs each of which is written with w bits, and computes their product. How does such an algorithm work and what is its runtime? A "count the instructions" analysis would suggest that it just multiplies all the numbers together and takes $O(n)$ time!. However, remember that for n integers each w bits, their product is an integer containing nw bits, which requires n registers to store! Computing this product would therefore take much more than $\Theta(n)$ time since multiplying a super-constant number of integers can not be done in a single instruction and would instead require an algorithm for multiplying large integers[1].

This might seem odd at first but this model is really trying to help us match the behavior that such an algorithm would have on a real computer! Almost every modern processor operates on 64-bit words, such that all arithmetic, bitwise, comaprison operations, etc., can be done with a single machine instruction. What would happen if you tried to implement an algorithm that multiplies n integers on a real computer? In most languages other than Python, you will quickly find that the result will *overflow*, so you will just get a wrong answer (most likely you'll get the answer modulo 2^{64} or something similar.) In Python you will find that you do in fact get the right answer, but the computation will become very slow! Under the hood Python is actually happy to represent large numbers for you by decomposing them into an array of word-sized integers. Doing arithmetic on these big integers takes more than constant time. Python uses an algorithm called *Karatsuba multiplcation* for multiplying these big integers, which runs in $O(d^{\log_2 3}) \approx O(d^{1.58})$ operations for d-digit numbers.

But what if... we ignore the word size An alternative model is the *unit-cost RAM model* which does not place any restriction on the size of the integers. In this model we just say that all arithmetic operations on integers takes constant time regardless of how many bits it takes to represent them. This might seem like an unimportant and pedantic difference, but it turns out that this assumption allows you to implement some wild and crazy algorithms, such as being able to sort n integers in constant time![2] Perhaps even more ludicrous, a RAM with unlimited precision (no bound on w) can solve PSPACE-Complete problems in polynomial time![3]

3.1.2 Beating the comparison model

To beat the comparison model, we have to advantage of some power that it doesn't have. The major limitation of the comparison model is that every operation we pay for (a comparison) can only result in a *binary outcome*. This is fundamentally why our information theoretic lower bounds gave us $\log_2 \#$ outputs, because the best we could do was double/halve the possible outcomes each time we paid a cost of one. The source of untapped power of the word RAM is that we can use our input elements (integers) as indices into arrays! That is, if I have an array

[1]We might see an algorithm for this later in the course. It takes $\Theta(d \log d)$ instructions to multiply d-digit integers!
[2]Appendix A of Computing with arbitrary and random numbers, Michael Brand's PhD thesis, Monash University.
[3]See *A characterization of the class of functions computable in polynomial time on Random Access Machines*

of length n and some integer from 1 to n (or 0 to $n-1$ if zero-indexed), then I can access the value of the array in constant time, but depending on the value of the integer, there are now n possible outcomes, which is far more than the two outcomes of the comparison model!

Warmup example: constant-time static search Before we dive into sorting, lets demonstrate how the word RAM has more power than the comparison model. The same ideas will be used momentarily in our sorting algorithms. Consider the simpler problem of *static searching*, i.e., outputting the position of a given element in a given array if it exists, where arbitrary preprocessing is allowed. By arbitrary preprocessing, we mean that you can, for example, sort the elements or arrange them into a binary search tree, all for free to make the queries faster to answer. There are $n+1$ possible outcomes (each position and "it doesn't exist"), so an information theoretic lower bound in the comparison model immediately tells us that we can't do better than $\Omega(\log n)$ cost. This tells us that binary search (on a sorted array) and balanced binary search trees are *asymptotically optimal* in the comparison model as they match this bound.

But what could we do with the power of the word RAM when our elements are word-size integers? Given an array $a_1, \ldots, a_n$ of n integers which are in the range $\{0, 1, \ldots, u-1\}$, where $u \leq 2^w$ is the size of the *universe* of inputs we are considering, we could create an array T of size u, one slot for every possible value, then store $T[a_i] \leftarrow i$, i.e., store a lookup table of positions based on the values. To answer a search query for a value x we simply check $T[x]$ and output the index if there is one stored there. This solves the problem of static searching in constant time! We have defeated the confines of the comparison model, at the expense of making the assumption that our keys are integers rather than arbitrary comparable things. Furthermore this solution makes the assumption that u is a reasonable amount of space to use. If the universe of keys is very large, this solution is horribly space inefficient. That problem can be effectively eliminated by the use of *hashing* which we will study in great detail in the next few lectures!

3.2 Sorting small integers: Counting Sort

Lets start by setting up the problem precisely.

Problem: Integer sorting

The integer sorting problem consists of an input array of n elements $a_1, a_2, \ldots, a_n$, each identified by a (not necessarily unique) integer key called $\text{key}(a_i)$. The goal is to output an array containing a permutation of the input $a_{\pi_1}, \ldots, a_{\pi_n}$ such that

$$\text{key}(a_{\pi_1}) \leq \text{key}(a_{\pi_2}) \leq \ldots \text{key}(a_{\pi_n}).$$

An important feature of the problem is that we are not just assuming that the input is an array of nothing but integers. Rather, the input is an array of elements with associated integer *keys*. This of great practical importance since in real-life you are rarely going to want to sort an array that contains literally nothing but integers, but likely you want to sort some data by some integer property. For example, you may have a spreadsheet and you want to sort the rows by one of the columns which contains an integer value. When you sort the rows, you of course do not only

want to sort that one column containing the integer, but you want the entire row attached to that integer to come along for the ride, otherwise the spreadsheet would be messed up and the rows would no longer contain the right values.

Note that we allow there to be duplicate keys among the elements. Recall from your previous classes that a sorting algorithm is called *stable* if it preserves the relative order of duplicate keys. That is, if $\text{key}(a_i) = \text{key}(a_j)$ and $i < j$, then a_i appears before a_j in the output. It will be of importance to us in this lecture to design algorithms that are stable.

3.2.1 The algorithm

When sorting an array of elements, the main question we have for each element is "where should it go?" In the comparison model when dealing with black-box comparable things, the only way we can answer this question is by comparing the element to (often many) other elements. However, when our keys are integers, we already have a pretty good idea of where they go. Element x comes after element $x - 1$ and before element $x + 1$ (if they exist).

Warm-up problem: Sorting distinct keys $\{1, 2, \ldots, n\}$ As a warm up, lets say that the input only contains *distinct keys* in the range $\{1, 2, \ldots, n\}$, which means the keys are exactly the set $\{1, 2, \ldots, n\}$. In this case, we could sort them by simply creating an output array S of size n, then for each element a_i, just place a_i in $S[\text{key}(a_i)]$ directly!

Warmer-up: Sorting distinct keys in $\{0, 1, \ldots, u-1\}$ If we knew the input contained only distinct keys in the range $\{0, 1, \ldots, u-1\}$, we could sort them by creating an array S of size u instead of size n, and then similarly placing element a_i in slot $S[\text{key}(a_i)]$, then lastly filtering out the empty slots with a second pass over the output.

Counting Sort In general we might have duplicate keys, so we can fix this by instead storing a *list* of elements with key x at slot x. This gives us our first integer sorting algorithm.

Algorithm: Counting Sort

Suppose the input contains n elements $a_1, \ldots, a_n$ whose keys are integers in the range $0, 1, \ldots, u-1$. The `key` function takes an element and returns the associated integer key.

```
function CountingSort(a : array, key : element → int) {
  let L be an array of u empty lists
  for each element x in a do {
    L[key(x)].append(x)
  }
  let out = empty list
  for each integer k in range(0, u) do {
    out.extend(L[k])
  }
  return out
}
```

The correctness follows from the fact that the elements are stored in the array in key order.

Theorem: Complexity of Counting Sort

On n elements with integer keys in $\{0,1,\ldots,u-1\}$, Counting Sort runs in $O(n+u)$ time.

Proof. The algorithm just makes one pass over the input of length n, then one pass over the universe of keys u, and builds the output array of length n, so in total, we have an algorithm that runs in $O(n+u)$ time □

Finally, we should make the observation that Counting Sort is *stable* since it appends elements to the lists in order. This will be very important in the last section.

We wanted linear-time sorting, right? Our goal was to design an algorithm that sorts faster than a comparison sorting algorithm which we know can run in $\Theta(n \log n)$ time. Counting Sort achieves a runtime of $O(n+u)$, so how exactly does that stack up? Its not quite directly comparable since it depends on the size of u. Since our goal is to achieve linear-time sorting, we will describe the algorithm in terms of which inputs allow it to achieve this goal. So, in this case, as long as $u = O(n)$, Counting Sort runs in linear time! In other words, if our integers have at most $\log n + c$ bits for any constant c, Counting Sort is linear time. This is not too bad. If we have a data set of n elements and each has an index in $1\ldots n$ or even $1\ldots c\,n$ for some constant n, then we can sort on that index in linear time!

Our next goal will be to improve this and find an algorithm that can sort even larger integers still in linear time.

3.3 A side quest: Tuple sorting

A useful stepping stone towards our last algorithm will be a concept sometimes called *tuple sorting*. The problem is this:

Problem: Tuple sorting

Given an array of n elements where each element has a key which are equal-length tuples $(k_1, k_2, \ldots, k_d)$, we want to sort the array lexicographically by key. That is, the array should be sorted by the first element of the tuples, with ties broken by the second, ties on those broken by the third, and so.

We will describe three algorithms for this problem! All of them will work by using another ordinary sorting algorithm but in different ways.

Comparison tuple sorting The most straightforward (though not particularly useful for this lecture) algorithm for tuple sorting is to just apply any of your favourite comparison sorting algorithms (merge sort, quicksort, heapsort) that run in $O(n \log n)$ cost, comparing the tuples element by element. Note that comparing a tuple does **not** take constant time, so we need to

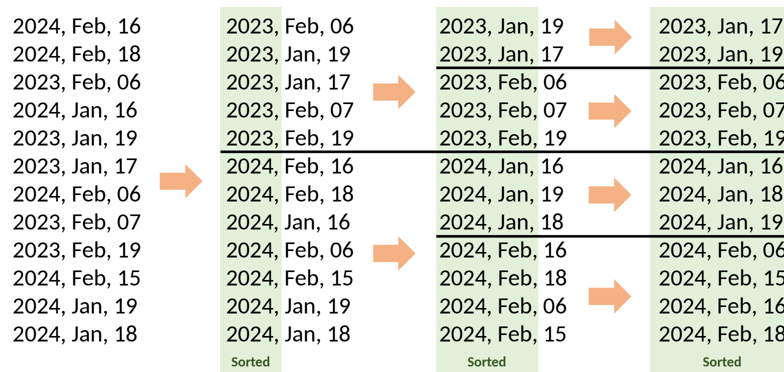
multiply the cost (number of comparisons) by the length of the tuples, i.e., d. This will give us an $O(d\,n\log n)$ algorithm. This works fine in the comparison model, but it doesn't generalize very well outside the comparison model, e.g., to tuples of integers and integer sorting.

The remaining two algorithms, instead of only performing one sort, will sort multiple times to achieve the same effect! This will also make them more generalizable and applicable outside the comparison model since each sort will only apply to one element of the tuple.

3.3.1 Top-down tuple sorting

A natural algorithm for tuple sorting is to essentially follow the definition. First, we can sort, using any sorting algorithm we like, using the first tuple element as the key. The array is now sorted correctly *except* that keys with equal first tuple elements are not tie-broken by the second tuple element yet. To resolve that, recursively sort each subarray of equal first elements, this time using the second tuple element as the key, and so on.

As an example, consider sorting a set of dates which are represented as (Year, Month, Day) tuples. A top-down tuple sort will first sort them by year, then recursively sort the dates that have equal years by their month, and then finally recursively sort the dates with equal years and months by their day.



The nice thing about this algorithm is that it can be applied using any sorting algorithm to perform each step, as long as that sorting algorithm can sort the individual elements of the tuples, e.g., we could use counting sort if every element of the tuples are integers!

3.3.2 Bottom-up tuple sorting

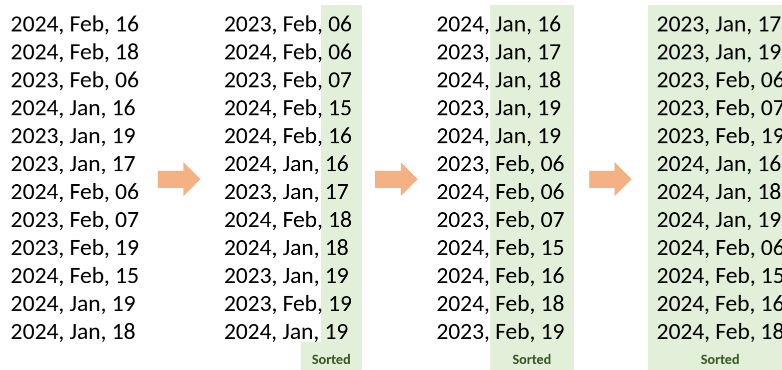
Like most recursive algorithms, we can also find a non-recursive alternative. What if instead of recursively sorting each subarray of equal tuple elements, we just sorted the entire array one tuple element at a time? Say we first sort the entire array using the first tuple element as the key, then the second, and so on. What would happen? Well, the array wouldn't actually be in sorted order because it would be sorted by the final tuple elements, not the first!

However, this algorithm almost works, with just one minor adjustment. The tuple element that we sort with respect to last is the one that ends up sorted, so we should actually sort in *reverse order* of the tuple elements, i.e., sort with respect to the final tuple element, then the second

last, and so on until we sort with respect to the first tuple element. This makes intuitive sense because the final sort is going to dictate that the elements are ordered by the first tuple element, which is exactly what we want.

Now the important observation: as long as the sorting algorithm that we use at each step is *stable*, we will end up with the correct answer. This is because before we sort on the first tuple element, the array is already sorted correctly with respect to the second tuple element because of the previous sort, so by using a stable sort, all ties between equal first tuple elements will be correctly tie-broken on the second tuple element! Applying this reasoning inductively should convince us that the array will be correctly sorted lexicographically with respect to the tuples!

Lets see the date sorting example again, this time using the bottom-up approach. Notice that the algorithm pretty much does the same thing but in reverse! First it sorts the days correctly, then it sorts the months so that the (Month, Day) pairs are in sorted order, then finally it sorts the years to bring everything into the correct order.



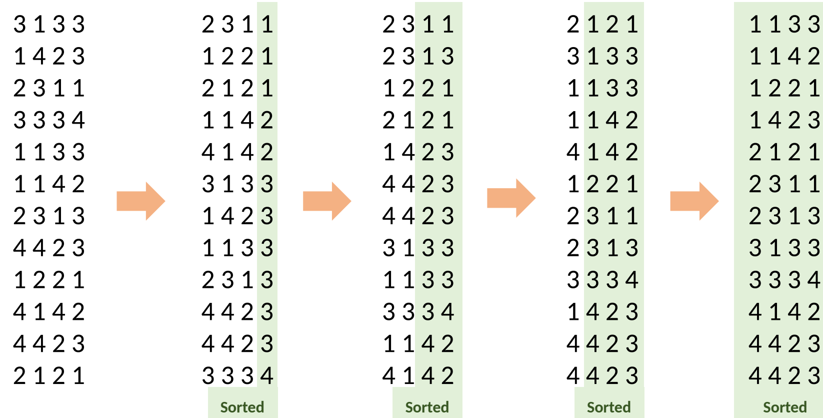
The key difference is that unlike the top-down approach, each round sorts the whole array, rather than recursively subdividing the array into smaller pieces that are sorted separately.

3.4 Sorting bigger integers: Radix Sort

We now have the ingredients to derive our last and best integer sorting algorithm. Counting Sort is great for sorting small integers but falls over once u becomes too large. However, what if we were to use *tuple sorting* instead on a tuple of small integers? As long as the tuple elements are small that sounds like it might be efficient! So the question is how can we convert a bigger integer into a tuple of smaller integers such that sorting on the tuples is equivalent to sorting on the original values? It turns out that we definitely already know the answer to this question, because it is just how we write down numbers every day! We can split numbers into their *digits*!

> ***Key Idea: Integer sorting is tuple sorting by digits***
>
> Sorting integers is equivalent to tuple-sorting them by their digits! If a number has a greater first digit than another, then it is greater. If two numbers have equal first digits, then they are tie-broken by their second digit and so on!

Here's the idea in picture form, just like bottom-up tuple sorting from a moment ago. We first sort the numbers by the least-significant digit, then by the second-least, and so on, until finally sorting by the most significant digit. This algorithm is called bottom-up Radix Sort or *Least-Significant-Digit (LSD) Radix Sort*, since it sorts in order from least to most significant.



Since it is stable and the digits are small integers, we use Counting Sort for each iteration!

Algorithm: LSD Radix Sort

Suppose the input contains n elements $a_1, \ldots, a_n$ whose keys are integers with `num_digits` digits, and we have a function `Digit(x, i)` which extracts the ith digit of x.

```
function RadixSort(a : array, key : element → int) {
  let out = copy of a
  for each i in range(0, num_digits) do {
    out = CountingSort(out, key = x → Digit(key(x), num_digits - i))
  }
  return out
}
```

We could similarly implement a recursive top-down / MSD Radix Sort as well, but we will stick with this for now since it allows us to make an optimization to improve the complexity in just a moment. In practice it would be more efficient to allocate the `out` array just once and reuse it (e.g., swapping it with `a` each iteration) for every iteration of Counting Sort rather than constantly allocating new arrays each time.

Analysis The last step is to analyze the algorithm. How fast is it? Well, it depends on the number of digits, right? If our numbers are again from a universe $\{0, 1, \ldots, u-1\}$, then they have $d = \log_b u$ digits when written in base b. Each digit will be a smaller integer from $\{0, 1, \ldots, b-1\}$. The algorithm performs d iterations, each of which performs a counting sort. Since the digits are in base b, Counting Sort takes $O(n+b)$ time. Therefore the total time for Radix Sort is

$$O((n+b)\log_b u).$$

Initially this doesn't look that great. If we pick any constant base b, then even if $u = O(n)$, the runtime becomes $O(n \log n)$, which is the same as comparison sorting and actually worse than Counting Sort! How can we make this better? Well, the higher the base, the fewer iterations the algorithm needs, so lets make it even higher! As long as we do not cause the Counting Sort to take more than linear time, we should be fine, so since the Counting Sort takes $O(n+b)$ time, our base can go up to n without trouble! Intuitively this makes sense because Counting Sort was good until the universe became bigger than linear in n, so we should pick that as the base for Radix Sort to get the most value out of each iteration of Counting Sort.

Theorem: Complexity of Radix Sort

On n elements with integer keys in $\{0, 1, \ldots, u-1\}$, Radix Sort runs in $O(n \log_n u)$ time.

This is a little hard to interpret and its not immediately obvious whether this is good, but the claim is that this is now good and can sort much larger integers than Counting Sort could!

Corollary: Radix Sort can sort bigger integers

Radix Sort can sort n elements with integers keys in the range $\{0, 1, \ldots, O(n^c)\}$ for any constant c, i.e., any integer keys that are *polynomial size* in n, **in linear time**.

Proof. Let $u = O(n^c)$, so $\lg_n u = O(\lg_n n^c) = O(c) = O(1)$ and hence the complexity is $O(n)$. □

Wow, that's a big improvement! We have gone from being able to only sort integer keys that were *linear* in n, i.e., keys up to $O(n)$, which is quite restrictive, to being able to sort any keys that are *polynomial* in n, i.e., keys up to $O(n^c)$ for any constant c.

Remark: Integer sorting is still an open problem!

One cool thing about comparison sorting is that it is (asymptotically) a solved problem. We know that $\Omega(n \log n)$ is a lower bound, but we also have algorithms that run in $O(n \log n)$ cost, so those algorithms are optimal up to constant factors.

The same however is **not true** for integer sorting! The algorithms we learned today can sort integers that are up to polynomial size in n, but what about sorting integers without any restriction on the size? This is still an open research problem. The best algorithms that have been discovered run in $O(n \log \log n)$ time (deterministically), or in $O(n\sqrt{\log \log n})$ expected time. We don't however know any lower bound for integer sorting other than the trivial $\Omega(n)$ lower bound! So, we still don't know whether there exists a linear-time sorting algorithm for integers that works regardless of their size, or whether there is a lower bound that shows that this is not possible. Either could be true!

Exercises: Integer Sorting

Problem 5. We wrote pseudocode that performs a bottom-up (LSD) Radix Sort. Write similar pseudocode for a top-down Most-Significant-Digit (MSD) Radix Sort instead.

Lecture 4

Hashing: Universal and Perfect Hashing

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* and *perfect hashing*.

Objectives of this lecture

In this lecture, we want to:

- Review dictionaries and understand the formal definition and general idea of hashing
- Define and analyze *universal hashing* and its properties
- Analyze an algorithm for *static perfect hashing*

Recommended study resources

- CLRS, *Introduction to Algorithms*, Chapter 11, Hash Tables
- DPV, *Algorithms*, Chapter 1.5, Universal Hashing

4.1 Dictionaries, hashing, and hashtables

4.1.1 The Dictionary problem

One of the main motivations behind the study and hashing and hash functions is the *Dictionary* problem. A dictionary D stores a set of *items*, each of which has an associated *key*. From an algorithmic point of view, items themselves are not typically important, they can be thought of as just data associated with a key. The key is the important part for us as algorithm designers. The operations we want to support with a dictionary are:

Definition: Dictionary data type

A dictionary supports:

- `insert(item)`: add the given item (associated with its key)
- `lookup(key)`: return the item with the given key (if it exists)
- `delete(key)`: delete the item with the given key

In some cases, we don't care about inserting and deleting keys, we just want fast query times—e.g., if we were storing a literal dictionary, the actual English dictionary does not change (or changes extremely rarely). This is called the *static case*. Another special case is when we only insert new keys but never delete: this is called the *incremental case*. The general case with insertions, lookups and deletes is called the *fully 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, *hashtables* are an alternative approach that is often the fastest and most convenient way to solve these problems. You should hopefully already be familiar with the main ideas of hashtables from your previous studies.

4.1.2 Hashing and hashtables

To design and analyze hashing and hashing-based algorithms, we need to formalize the setting that we will work in. We will continue to work in the word RAM model from last lecture, so operations on word-sized integers takes constant time, and we have access to constant time indirect addressing (looking up an element of an array by its index in constant time).

The key space (the universe): The *keys* are assumed to come from some large *universe* U. Most often, when analyzed on the word RAM model, we will assume that $U = 0, \ldots, u-1$, where $u = 2^w$ is the universe size, i.e., the keys are word-sized integers.

The hashtable: There is some set $S \subseteq U$ of keys that we are maintaining (which may be static or dynamic). Let $n = |S|$. Think of n as much smaller than the size of U. 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 item with key x, the idea of a hashtable is that we want to store it in $A[h(x)]$. Note that if U was small then you could just store the item in $A[x]$ directly, no need for hashing! Such a

data structure is often called a direct-access array or direct-address table. The problem is that U is assumed to be very big, so this would waste memory. That is why we employ hashing.

Collisions: Recall that hashtables suffer from *collisions*, and we need a method for resolving them. A *collision* is when $h(x) = h(y)$ for two different keys x and y. For this lecture, we will assume that collisions are handled using the strategy of *separate chanining*, by having each entry in A be a list[1] containing all the colliding items. There are a number of other methods (e.g., open addressing aka probing), but for this lecture we are focusing primarily on the *hash function* itself, and separate chaining just happens to be the simplest method. To insert an item, we just add it to the list[2]. If h is a "good" hash function, then our hope is that the lists will be small. This lecture will focus on exploring what "good" hash functions look like.

The question we now turn to is: what properties are needed 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 items 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 $O(1)$ if we don't check for duplicates, or the same time again if we do. So, *our main goal will be to be able to analyze how big these lists get.*

Prehashing non-integer keys: One issue that we sweep under the rug in theory but that matters a lot in practice is dealing with non-integer keys. Hashtables in the real world are frequently used with data such as strings, so we want this to be applicable.

The way that we get around this in theory is to require non-integer key types to come equipped with a *pre-hash* function, i.e., a function that converts the keys reasonably uniformly into integers in the universe U. Then we can proceed as normal assuming integer keys.

Basic intuition: One way to spread items out nicely is to spread them *randomly*. Unfortunately, we can't just use a random number generator to decide where the next item 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.

[1] Historically, separate chaining was always described by using *linked lists* to store the set of colliding items. For most theoretical purposes however, the kind of list (e.g., linked list vs. dynamic array) is irrelevant so I just say list.

[2] This assumes that the user will never insert a duplicate key. To guard against this we could first scan the list and check for duplicates before inserting.

Claim: Bad news

For any hash function h, if $|U| \geq (n-1)m+1$, there exists a set S of n items that all hash to the same location.

Proof. By the pigeonhole principle. In particular, to consider the contrapositive, if every location had at most $n-1$ items 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? A common but unsatisfying answer is that there are a lot of simple hash functions that work well in practice for typical sets S. But we are not a practical class, we want theoretical guarantees! What can we do if we want to have good *worst-case* guarantees?

4.1.3 The key idea: Random hashing

In Lecture One we reviewed one of the holy grails of algorithm design. To foil an adversary from constructing worst-case inputs to your algorithm, *introduce randomness into the algorithm!* Specifically, let's use randomization in our *construction* of h. Importantly, we must remember that the function h itself will be a deterministic function, but we will use randomness to choose *which function* h we end up with.

What we will then show is that for *any* sequence of insert and lookup operations (remember, we won't assume the set S of items inserted is random since that would give us *average-case* complexity and we want *worst-case* bounds), if we pick h in this probabilistic way, the performance of h on this sequence will be good in expectation. We will come up with different kinds of hashing schemes depending on what we mean by "good". Intuitively, the goal is to make the hash appear *as if it was a totally random function*, even though it isn't.

We will first develop the idea of *universal hashing.* Then, we will use it for an especially nice application called "perfect hashing".

4.2 Universal Hashing

Definition: Universal Hashing

A set of hash functions $\mathscr{H}$ where each $h \in \mathscr{H}$ maps $U \to \{0,\ldots,m-1\}$ is called **universal** (or is called a *universal family*) if for all $x \neq y$ in U, we have

$$\Pr_{h \in \mathscr{H}}[h(x) = h(y)] \leq 1/m. \tag{4.1}$$

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

$\mathcal{H}$. Here's an equivalent way of looking at this. First, count the number of hash functions in $\mathcal{H}$ that cause x and y to collide. This is

$$|\{h \in \mathcal{H} | h(x) = h(y)\}|.$$

Divide this number by $|H|$, the number of hash functions. This is the probability on the left hand side of (4.1). So, to show universality you want

$$\frac{|\{h \in \mathcal{H} \mid h(x) = h(y)\}|}{|H|} \leq \frac{1}{m}$$

for every $x \neq y \in U$. Here are some examples to help you become comfortable with the definition.

Example

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, these 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

4.2.1 Using Universal Hashing

Theorem 4.1: Universal hashing

If $\mathcal{H}$ is universal, then for any set $S \subseteq U$ of size n, for any key $x \in S$ (e.g., that we might want to lookup), if h is drawn randomly from $\mathcal{H}$, the **expected** number of collisions between x and other keys in S is less than 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 random variable denoting the total number of collisions for x. So,

$$C_x = \sum_{\substack{y \in S \\ y \neq x}} C_{xy}.$$

We know $\mathbb{E}[C_{xy}] = \Pr(x \text{ and } y \text{ collide}) \leq 1/m$. Therefore, by linearity of expectation,

$$\mathbb{E}[C_x] = \sum_{\substack{y \in S \\ y \neq x}} \mathbb{E}[C_{xy}] \leq \frac{|S|-1}{m} = \frac{n-1}{m},$$

which is less than n/m. □

When a table is storing n items in m slots, this quantity n/m represents how "full" the table is and shows up frequently in the anaylsis of hashing, so it gets a name. $\alpha = n/m$ is called the *load factor* of the hashtable. We now immediately get the following theorem.

Theorem

Insert, lookup, and delete, on a hashtable using universal hashing with separate chaining cost $\Theta(1+\alpha)$ time in expectation.

Proof. The runtime for insert, lookup, and delete, for a key x is proportional to the number of items in the list at slot $A[h(x)]$ (plus a constant cost to compute the hash function). Suppose x is not currently in the hashtable, then by Theorem 4.1 the expected number of items in $A[h(x)]$ is the number of items in the hashtable that collide with x, which is less than $(n+1)/m = \Theta(1+\alpha)$. Similarly if x is currently in the table then then the number of items in $A[h(x)]$ is the the number of items in the hashtable that collide with x plus one (itself, since x is definitely at location $A[h(x)]$), so by Theorem 4.1 the cost is again at most $\Theta(1+n/m) = \Theta(1+\alpha)$. □

Corollary

For any sequence of L insert, lookup, and delete operations in which there are at most m keys in the hashtable at any one time, using separate chaining with universal hashing, the expected total cost of the L operations is only $O(L)$.

Proof. Since there are at most m keys in the table at any time, $\alpha \leq 1$, so every operation costs $\Theta(1+\alpha) = \Theta(1)$ in expectation. By linearity of expectation, the expected total cost is $O(L)$. □

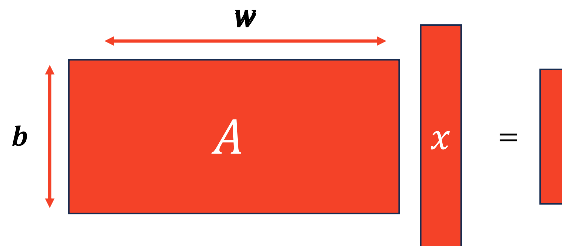
Can we construct a universal $\mathcal{H}$? If not, this this is all useless. Luckily, the answer is yes.

4.2.2 Constructing a universal hash family: the matrix method

Definition: The matrix method for universal hashing

Assume that keys are w-bits long, so $U = 0, \ldots, 2^w - 1$. We require that the table size m is a power of 2, so an index is b-bits long with $m = 2^b$. We pick a random b-by-w 0/1 matrix A, and define $h(x) = Ax$, where we do addition mod 2. x is interpreted as a 0/1 vector of length w, and $h(x)$ is a 0/1 vector of length b, denoting the bits of the result.

These matrices are short and fat. For instance, in picture form:



Claim: The matrix method is universal

Let $\mathscr{H}$ be the hash family generated by the matrix method. For all $x \neq y$ from U, we have

$$\Pr_{h \in \mathscr{H}}[h(x) = h(y)] = \frac{1}{2^b} = \frac{1}{m}$$

Proof. First of all, what does it mean to multiply A by x? We can think of it as adding some of the columns of A (doing vector addition mod 2) where the 1 bits in x indicate which ones to add. E.g., if $x = (1010\cdots)^\intercal$, Ax is the sum of the 1st and 3rd columns of A.

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 the entries of A but those in the ith column. Over the remaining choices of ith column, $h(x) = Ax$ is fixed, since $x_i = 0$ and so Ax does not depend on the ith column of A. 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 $h(x) = h(y)$.

More verbosely, let $y' = y$ but with the ith entry set to zero. So $Ay = Ay' +$ the ith column of A. Now Ay' is also fixed now since $y'_i = 0$. Now if we choose the entries of the ith column of A, we get $Ax = Ay$ exactly when the ith column of A equal $A(x - y')$, which has been fixed by the choices of all-but-the-i^{th}-column. Each of the b random bits in this i^{th} column must come out right, which happens with probability (1/2) each. These are independent choices, so we get probability $(1/2)^b = 1/m$. □

Efficiency Okay great, its universal! But how efficient is it? If we manually compute the matrix product, since it is an $b \times w$ matrix, this will take $O(bw) = O(w \lg m)$ time, which is not great, since this is actually worse than using a BST. However, this is assuming that we compute the result bit-by-bit. If we take advantage of the word RAM and use the fact that the key and rows are w-bit integers, we can compute each row-vector product in constant time with a single multiplication and improve the performance to $O(\lg m)$ time, which is about the same as a balanced BST since we assume $m = O(n)$. That means that the matrix method is not particularly practical as a hash family since we prefer hash functions that are constant time. It is mostly intended to serve as a proof that nontrivial universal families actually exist and a good first example of

how to prove universality. There does exist universal families that contain hash functions that can be evaluated in constant time, but their proofs of universality are more complicated.

4.2.3 Another universal family: the dot-product method

Definition: The dot-product method for universal hashing

We will first require m to be a prime number. 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 requirement being that each x_i is in the range $\{0, 1, \ldots, m-1\}$ and hence $k = \log_m u$. There is a very natural interpretation of this. Just think of the key being written in base m.

To select a hash function, we choose k random numbers $r_1, r_2, \ldots, r_k$ in $\{0, 1, \ldots, m-1\}$ and define:

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

Note that choosing $[r_1, r_2, \ldots, r_k]$ here is equivalent to just picking a single value r in the universe and then writing it in base m as well. The proof that this method is universal follows the exact same lines as the proof for the matrix method.

Claim: The dot-product method is universal

Let $\mathcal{H}$ be the hash family generated by the dot-product method. For all $x \neq y$ from U, we have

$$\Pr_{h \in \mathcal{H}}[h(x) = h(y)] = \frac{1}{m}$$

Proof. 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 occurring is exactly $1/m$. □

Efficiency Is this more efficient than the matrix method? Well, our keys consist of k pieces/digits, where $k = \log_m u$, so computing the hash function takes $O(k) = O(\log_m u)$ time, which is faster than the matrix method when m is large, but slower when m is small. If we pick $m = \Theta(n)$ like usual, then this is $O(\log_n u)$, which is $O(1)$ time if $u = O(n^c)$ (same as the analysis of Radix

Sort!), i.e., if the universe size is polynomial in n. So this hash family is constant time for reasonable universe sizes, but not for all universe sizes.

4.3 More powerful hash families

Recall that our overarching goal with universal hashing was to produce a hash function that behaved *as if it was totally random*. We can try to be more specific about what we mean. In the case of universal hashing, if we took any two distinct keys x, y from our universe, and then hashed them using our hash function from a universal family, then the probability of collision was at most $1/m$, which is the probability that we would get if the hash function was totally random! We can therefore think of universal hashing as hashing that appears to behave totally randomly if all we care about is pairwise collisions.

In some cases (for some algorithms), though, this is not good enough. Although universal hashing looks good if all we care about are collisions, there are scenarios where universal hashes appear totally not random. Lets consider an example. Suppose we are maintaining a hash table of size $m = 2$, and an evil adversary would like to cause a collision by inserting just two items. If our hash was totally random, then the adversary would have a 50/50 chance of success just by pure chance. Suppose that we use the following universal family for our hash table.

	a	b	c
h_1	0	0	1
h_2	1	0	1

In this case, the evil adversary can just first insert a, and now we are in trouble. If a goes into slot 0, then the adversary knows we have h_1 and can hence select b to insert next, causing a guaranteed collision. Otherwise, if a goes into slot 1, then the adversary can select c and cause a guaranteed collision. So, even though we used a universal hash family, it wasn't as good as a totally random hash, because the adversary was able to figure out which hash function had been selected by just knowing the hash of one key. The problem at a high level was that although this family makes collisions unlikely, it doesn't do anything to prevent the hashes of different keys from correlating. In this family, the adversary can deduce the hash values of b and c by just knowing the hash of a.

To fix this, there is a closely-related concept called pairwise independence.

Definition: Pairwise independence

A hash family $\mathcal{H}$ is *pairwise independent* if for all pairs of distinct keys $x_1, x_2 \in U$ and every pair of values $v_1, v_2 \in \{0, \ldots, m-1\}$, we have

$$\Pr_{h \in \mathcal{H}}[h(x_1) = v_1 \text{ and } h(x_2) = v_2] = \frac{1}{m^2}$$

Intuitively, pairwise independence guarantees that if we only ever look at pairs of keys in our universe, then their hash values appear to behave totally randomly! In other words, if the adversary ever learns the hash value of one key, it can not deduce any information about the hash

values of the other keys, they appear totally random. Of course, it is possible that by learning the hash values of *two* keys, the adversary may be able to deduce information about other keys. To improve this, we can generalize the definition of pairwise independence to arbitrary-size sets of keys.

Definition: k-wise independence

A hash family $\mathscr{H}$ is *k-wise independent* if for all k distinct keys $x_1, x_2, \ldots, x_k$ and every set of k values $v_1, v_2, \ldots, v_k \in \{0, \ldots, m-1\}$, we have

$$\Pr_{h \in \mathscr{H}}[h(x_1) = v_1 \text{ and } h(x_2) = v_2 \text{ and } \ldots \text{ and } h(x_k) = v_k] = \frac{1}{m^k}$$

Intuitively, if a hash family is k-wise independent, then the hash values of sets of k keys appear totally random, or, if an adversary learns the hash values of $k-1$ keys, it can not deduce any information about the hash values of any one other key.

4.4 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 $O(1)$ deterministic work-case cost (though lookup must be deterministic, randomization is still needed to actually construct the hash function). Here are now two methods for constructing perfect hash functions for a given set S.

4.4.1 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 $\mathscr{H}$ be universal and $m = n^2$. Then just pick a random h from $\mathscr{H}$ and try it out! The claim is there is at least a 50% chance it will have no collisions.

Claim

If $\mathscr{H}$ is universal and $m = n^2$, then

$$\Pr_{h \in \mathscr{H}}(\text{no collisions in } S) \geq 1/2.$$

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. Therefore,

$$\Pr(\text{exists a collision}) \leq \frac{\binom{n}{2}}{m} = \frac{n(n-1)}{2m} \leq \frac{n^2}{2n^2} = \frac{1}{2}$$

□

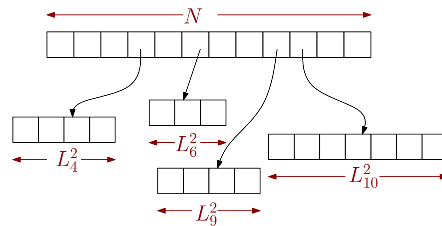
This is like the other side to the "birthday paradox". If the number of days is a lot *more* than the number of people squared, then there is a reasonable chance *no* pair has the same birthday.

So, we just try a random h from $\mathscr{H}$, 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?

4.4.2 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 a 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 a key x, we first compute $i = h(x)$ and then find the item 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 keys 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

If we pick the initial h from a universal family $\mathscr{H}$, then

$$\Pr\left[\sum_i (L_i)^2 > 4n\right] < \frac{1}{2}.$$

Proof. We will prove this by showing that $\mathbb{E}\left[\sum_i (L_i)^2\right] < 2n$. 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 a key 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:

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

□

So, we simply try random h from $\mathscr{H}$ 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.

Exercises: Hashing

Problem 6. Show that any pairwise independent hash family is also a universal hash family.

Problem 7. Show that the matrix method as defined above, which was universal, is **not** pairwise independent.

Lecture 5

Fingerprinting & String Matching

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. This technique is surprisingly elegant and extremely extensible!

Objectives of this lecture

In this lecture, we will:

- Cover some facts about prime numbers that are useful for randomized hashing schemes
- See a new application of hashing to string equality checking
- Look at the Karp-Rabin pattern matching algorithm

Recommended study resources

- CLRS, *Introduction to Algorithms*, Section 32.2, The Rabin-Karp algorithm
- DPV, *Algorithms*, Section 1.3.1, Generating random primes

5.1 How to Pick a Random Prime

In this lecture, we will often be picking random primes, so let's talk about that. This is used in many widely used algorithms, for example, you do this when generating RSA public/private key pairs. So, how do we actually pick a random prime in some range $\{1, \ldots, M\}$? Here's a straightforward approach:

Algorithm: Random prime generation

- 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.

Okay this is not quite complete, we have to fill in some details. How would you pick a random number in the prescribed range? 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[1] (which may produce false positives, 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[2] primality test, which has a worse runtime, but is deterministic and hence guaranteed to be correct. We won't cover those algorithms in this course, so for now, just know that they exist, we can use them, and know that they run in $O(\text{polylog}\, M)$ time.

5.2 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 $\pi(n)$ primes between 0 and n. One of the great theorems of the 20th century was the Prime Number theorem:

Theorem 5.1: The prime number theorem

The prime counting function $\pi(n)$ satisfies

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

And while this is just a limiting statement, an older result of Chebyshev (from 1848) says that

Theorem 5.2: Chebyshev

For $n \geq 2$, the prime counting function $\pi(n)$ satisfies

$$\pi(n) \geq \frac{7}{8} \frac{n}{\ln n} = (1.262\ldots) \frac{n}{\log_2 n} > \frac{n}{\log_2 n}$$

Here are two consequences of this theorem. The first is that a random integer between 1 and n is a prime number with probability at least $1/\log_2 n$. We also get the following fact:

Corollary 5.1: Density of primes

If we want at least $k \geq 4$ primes between 1 and n, it suffices to have $n \geq 2k \log_2 k$.

[1] http://en.wikipedia.org/wiki/Miller-Rabin_primality_test
[2] http://en.wikipedia.org/wiki/AKS_primality_test

Proof. Just plugging in to Theorem 5.2, we get

$$\pi(2k\log_2 k) \geq \frac{2k\log_2 k}{\log_2(2k\log_2 k)} \geq \frac{2k\log_2 k}{\log_2 2 + \log_2 k + \log_2\log_2 k} \geq k.$$

□

5.2.1 Tighter Bounds

The following even tighter set of bounds were proved by Pierre Dusart in 2010.

Theorem 5.3: Dusart

For all $n \geq 60184$ we have:

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

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.

5.3 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 $x = y$. [3] 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 $\Pr[\text{ Bob says } \textbf{equal} \,] = 1$.
- If $x \neq y$, then $\Pr[\text{ Bob says } \textbf{equal} \,] \leq \delta$.

[3] E.g., this could be an update to the lander firmware, and we want to make sure the file did not get corrupted

Here's a protocol that does almost that. We will *hash* the strings using the hash function $h_p(x) = (x \bmod p)$ for a random prime p, then check whether the hashes are equal.

Algorithm: Randomized string equality test

1. Alice picks a random prime p from the set $\{1, 2, \ldots, M\}$ for $M = \lceil (200N) \cdot \log_2(100N) \rceil$.
2. She sends Bob the prime p, and also the value $h_p(x) := (x \bmod p)$.
3. Bob checks if $h_p(x) \equiv y \mod p$. If so, he says **equal** else he says **not equal**.

For now, let's not worry about where the particular value of M came from: it will arise naturally. Let's see how this protocol performs.

Lemma 5.1

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 5.2

If $x \neq y$, then Pr[Bob says **equal**] $\leq \frac{1}{100}$.

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 chances 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[4]. 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 prime divisors. The probability that the randomly chosen prime p is one of them is

$$\frac{N}{\text{number of primes in } \{1, 2, \ldots, M\}}.$$

We want this to be at most $1/100$, i.e., we would like that the number of primes in $\{1, 2, \ldots, M\}$ is at least $100N$. But Corollary 5.1 says that choosing $M \geq 200N \log_2 100N$ will give us at least $100N$ primes. Hence

$$\Pr[\text{ Bob falsely says } \textbf{equal} \text{ }] \leq \frac{N}{\text{number of primes in } \{1, 2, \ldots, M\}} \leq \frac{N}{100N} \leq \frac{1}{100}.$$

□

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

5.3.1 Communication cost

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 $M = 200N\log_2 100N$. The number of bits required for that:

$$2\log_2 M = 2\log_2(200N\log_2 100N)) = O(\log N).$$

To put this in perspective, suppose x and y were two copies of all of Wikipedia. Say that's about 25 billion characters (25 GB of data!). Say 8 bits per character, so $N \approx 2 \cdot 10^{11}$ bits. With the new approach, Alice sends over $2\log_2(200N\log_2 100N)) \approx 100$ bits, or 13 bytes of data. That's a lot less communication!

5.3.2 Reducing the Error Probability

If you don't like the probability of error being 20%, here are two ways to reduce it.

Approach #1 Have Alice repeat this process multiple times independently with different random primes, with Bob saying **equal** if and only if in all repetitions, the test passes. For example, for 5 repetitions, the chance that he will make an error (i.e., say **equal** when $x \neq y$) is only

$$(1/100)^5 = 10^{-10}$$

That's a 99.999999999% chance of success! In general, if we repeat R times, we get the probability of error is at most $(1/100)^R$, so if we desire an error probability of δ, we should do $R = \log_{100}(1/\delta)$ repetitions. Since each round requires communicating $O(\log N)$ bits, the total number of bits that Alice must communicate is

$$O(\log(1/\delta)\log N).$$

Can we do better than this?

Approach #2 Have Alice choose a random prime from a larger set. For some integer $s \geq 1$, if we choose $M = 2 \cdot sN\log_2(sN)$, 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 $1/s$. If we desire an error probability of δ, then we must choose $s = 1/\delta$. For example, to obtain 99.999% chance of success, we would pick $s = 1/10^{-6} = 10^6$. Now Alice is communicating two integers at most $2 \cdot sN\log_2(sN)$, so the number of bits is

$$\begin{aligned} 2\log_2(2 \cdot sN\log_2(sN)) &= 2\log_2 s + 2\log_2 N + 2\log_2(\log_2(sN)) + 2, \\ &= O(\log s + \log N), \\ &= O(\log(1/\delta) + \log N). \end{aligned}$$

This is much better than Approach #1!

5.4 The Karp-Rabin Algorithm (the "Fingerprint" method)

Let's use this idea to solve a different problem.

Problem: Pattern matching

In the *pattern matching* problem, we are given, over some alphabet Σ,

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

The goal is to output the locations of all the occurrences of the pattern P inside the text T. E.g., if $T = \texttt{abracadabra}$ and $P = \texttt{ab}$ 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. The idea is smart but simple, elegant and effective—like in many great algorithms. To simplify the presentation, we will start by assuming that $\Sigma = \{0, 1\}$, i.e., all of our strings are written in binary, but the ideas generalize to larger alphabets.

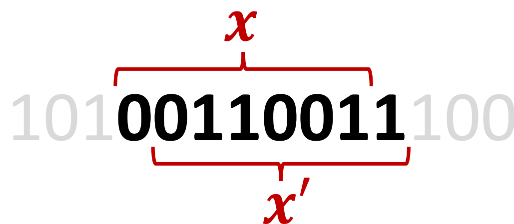
5.4.1 The Karp-Rabin Idea: "Rolling the hash"

As in the last section, interpret the string written in binary *as an integer* and use the hash

$$h_p(x) = (x \bmod p)$$

for some randomly chosen prime p.

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 $x = \texttt{0011001}$ then x' might be $\texttt{0110010}$ or $\texttt{0110011}$. If I told you $h_p(x) = z$, can you compute $h_p(x')$ fast?



Let x'_l be the lowest-order (rightmost) bit of x', and x_h be the highest order (leftmost) bit of x. Now observe that

- removing the high-order bit (x_h) is just equivalent to subtracting $x_h \cdot 2^{m-1}$,
- shifting all of the remaining bits to one higher position is equivalent to multiplying by 2,
- appending the low-order bit x'_l is equivalent to just adding x'_l.

Therefore, we can write

$$x' = 2(x - x_h \cdot 2^{m-1}) + x'_l$$

Since $h_p(a+b) = (h_p(a) + h_p(b)) \mod p$, and $h_p(2a) = 2h_p(a) \mod p$, we then have

$$h_p(x') = (2h_p(x) - x_h \cdot h_p(2^m) + x'_l) \mod p.$$

Take a moment to understand the significance of this fact. Given the hash $h_p(x)$ for the substring x and the value $h_p(2^m)$, we can compute the hash of the next adjacent substring $h_p(x')$ in just a constant number of arithmetic operations modulo p. This is an enormous speedup compared to computing $h_p(x')$ from scratch which would take $O(m)$ arithmetic operations.

5.4.2 The pattern matching algorithm

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

$$T_{i...i+(m-1)} = P.$$

Here's the algorithm.

Algorithm: Karp-Rabin pattern matching

1. Pick a random prime p in $\{1, \ldots, M\}$ for $M = \lceil 2sm\log_2(sm)\rceil$ (we'll choose s later.)
2. Compute $h_p(P)$ and $h_p(2^m)$, and store these results.
3. Compute $h_p(T_{0...m-1})$, and check if it equals $h_p(P)$. If so, output **match at position 0**.
4. For each $i \in \{0, \ldots, n-m-1\}$
   (i) compute $h_p(T_{i+1...i+m})$ using $h_p(T_{i...i+m-1})$
   (ii) If $h_p(T_{i+1...i+m}) = h_p(P)$, output **match at position i+1**.

Notice that 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) if we get a hash collision! Let's analyze the error probability, and the runtime.

Probability of Error Since the Karp-Rabin algorithm can encounter false positives, we should analyze the probability of them occuring. This will also show us how large of a prime number of we need to pick in order to achieve a desired false positive probability.

Theorem: Error probability of Karp-Rabin

We can achieve an error probability of δ using the Karp-Rabin algorithm with a prime of $O\left(\log\left(\frac{1}{\delta}\right) + \log m + \log n\right)$ bits.

Proof. We do $n-m$ 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 at most n/s. Hence, setting $s = 100n$ will make sure we have at most a $\frac{1}{100}$ chance of even a single mistake.

This means we set $M = (200 \cdot mn)\log_2(100 \cdot mn)$, which requires $\log_2 M + 1 = O(\log m + \log n)$ bits to store. Hence our prime p is also at most $O(\log m + \log n)$ bits.[5] Generalizing this, picking a prime from the range $M = (2/\delta \cdot mn)\log_2(1/\delta \cdot mn)$ achieves a failure probability of δ with a prime of $O\left(\log\left(\frac{1}{\delta}\right) + \log m + \log n\right)$ bits. □

Running Time Continuing to work in the word-RAM model, note that since the inputs consists of an n-length string and an m-length string, our word-RAM must have $w \geq \log(n)$ and $w \geq \log(m)$ to be able to index into the input. It is common in randomized algorithms to seek *polynomial* failure probability, i.e., the probability of failure should be proportional to

$$\delta = \frac{1}{O\big(\text{poly}(n, m)\big)},$$

where the notation $\text{poly}(n, m)$ means any polynomial expression in n and m. Since $p < M$, we have that $\log M = O(\log \text{poly}(n, m)) = O(\log n + \log m)$ and hence all arithmetic on integers mod p can be done in constant time!

Theorem: Running time of Karp-Rabin

The Karp-Rabin pattern matching algorithm runs in $O(m + n)$ time.

Proof. Since $p < M$ and all of our calcuations are done mod p, each individual arithmetic operation takes constant time.

- Computing $h_p(x)$ for m-bit x can be done in $O(m)$ time. So each of the hash function computations in Steps 2 and 3 take $O(m)$ time.
- Now, using the idea in Section 5.4.1, we can compute each subsequent hash value in $O(1)$ time! So iterating over all the values of i takes $O(n)$ time.

That's a total of $O(m + n)$ time! You can't do much faster, since the input is $m + n$ bits long. We did not talk about the complexity of picking the prime since we did not cover the complexity of the best primality testing algorithms, but this step can also be shown to run in $O(\text{polylog}(n, m))$ time, which is dominated by $O(n + m)$. □

[5] If we do the math, and say $m, n \geq 10$, then $\log_2 M \leq 4(\log_2 m + \log_2 n)$. Now, just for perspective, if we were looking for a $n = 1024$-bit phrase in Wikipedia, this means the prime p is only $4(\log_2 2^{38} + \log_2 2^{10}) \leq 192$ bits long.

Exercises: Fingerprinting

General alphabets For simplicity, we looked at the case where $\Sigma = \{0, 1\}$. The Karp-Rabin algorithm generalizes naturally to larger alphabets. Instead of treating the input as a number in binary, treat it as base-$|\Sigma|$. For example, if the text contains only lower-case English words, we would use base-26. The formula for rolling the hash still works, except we replace 2 with $|\Sigma|$, and the range $\{1, \ldots, M\}$ from which we should select our prime becomes slightly larger.

Problem 8. Suppose we use an alphabet Σ which has size $|\Sigma|$ for Karp-Rabin. What should we use as our new value of M, and how does this affect the number of bits required to store the prime p?

Other pattern matching algorithms and problems Though there are many algorithms for pattern matching, the advantage of the Karp-Rabin approach is not only the simplicity, but also the extensibility. You can, for example, solve the following 2-dimensional problem using the same idea.

Problem 9. 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 can 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.

Lecture 6

Range query data structures

Today we are going to explore a class of data structures for performing *range queries* and see how they can be applied to speed up algorithms. Our focus is therefore twofold. First, we would like to design and analyze a specific data structure, that we will refer to as a SegTree, and explore its power. Then, we will see how it can be used to improve other algorithms.

Objectives of this lecture

In this lecture, we will:

- Introduce the SegTree data structure
- See what kinds of problems SegTrees are capable of solving
- See how SegTrees and related data structures can be used to speed up algorithms

6.1 Range queries

Definition: Range queries

Given a sequence (but not strictly necessarily integers), a *range query* on that sequence asks about a property of some contiguous range of the data i to j.

For example, suppose we have a sequence of n integers $a[0], a[1], \ldots, a[n-1]$, and we want to support querying the sum between positions i to j. We will use the convention that the left index is inclusive, and the right index is exclusive. Here are two approaches to get warmed up.

Approach #1 For each query, just loop over positions i to $j-1$ and compute the sum. This takes $\Theta(j-i) = O(n)$ time. This is very simple, but not at all efficient if the sequence is long.

Approach #2 Start by *precomputing* the prefix sums

$$p[j] = \sum_{0 \le i < j} a[i],$$

then answer a query for the sum between i and j by returning $p[j] - p[i]$. This takes $O(n)$ preprocessing time, which seems reasonable, then each query can be answered in $O(1)$ time!

Approach #2 is basically optimal if we never plan to modify the elements of the array, but range queries become so much more useful if we allow modifications. So, our goal is to support an API that enables fast modifications *and* fast queries. Specifically, lets try to design a data structure that maintains an array of n integers and implements:

- **Assign**(i, x): Assign $a[i] \leftarrow x$,
- **RangeSum**(i, j): Return $\sum_{i \le k < j} a[k]$.

How would approaches #1 and #2 above fare now that we want to support modifications?

1. Approach #1 can implement **Assign** in $O(1)$ time by just assigning x to $a[i]$, then **RangeSum**s are still the same as before and take $O(n)$ time.
2. Approach #2 would require us to re-compute the prefix sums $p[j]$ for all $j < i$ whenever we perform **Assign**(i, x), which requires $\Theta(n - i) = O(n)$ time. Queries however still take $O(1)$ time which is nice.

So, in both cases we have one operation that takes $O(1)$ time, and the other which takes $O(n)$ time. If in some particular application, one of the operations is extraordinarily rare, maybe this is a good solution, but if we perform roughly half and half updates and queries, then both solutions are taking $O(n)$ time on average for each operation. This is not great at all. Can we design a data structure that makes both operations fast?

You may have already seen a data structure that can do this. In fact, we've already seen it in this course! Augmented balanced binary search trees (e.g., Splay Trees) can be used to solve this problem in just $O(\log n)$ time per operation and $O(n)$ words of space. However, they are tricky to implement, and often in practice the constant factor is quite high, making them somewhat less practical. Splay Trees also give amortized bounds rather than worst case, though you can improve this by using AVL trees or Red-Black trees.

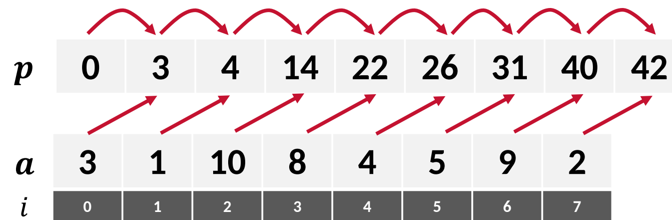
Today, we are going to design a data structure for this problem with the same bounds, $O(\log n)$ *worst-case* time per operation and $O(n)$ words of space, but that is much simpler to implement, and much faster in practice due to smaller constants hidden by the big-O. We will refer to this data structure as a SegTree[1]. We will also discuss how to generalize SegTrees to handle a much wider class of problems, where the $\sum$ operation is replaced by an arbitrary associative operator, enabling us to perform many different kinds of range queries with just one data structure!

6.2 Making range queries dynamic

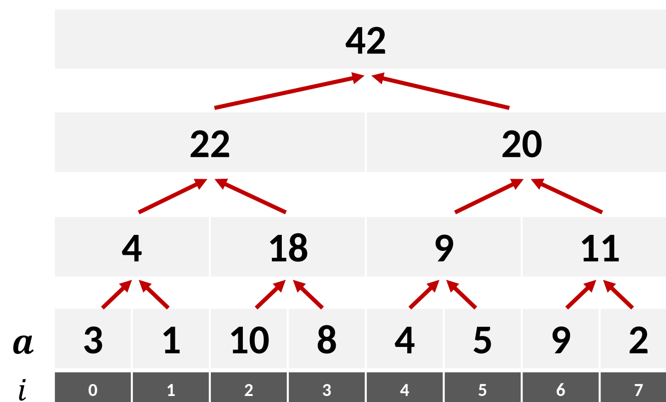
Let's take a step back and have a closer look at Approach #2 from earlier and see if we can find inspiration for a better algorithm. What the algorithm from Approach #2 is really doing is computing the sum from 1 to n in a sequential loop and just saving the results along the way.

[1] "SegTree" has become the traditional name for this data structure in 15-451, though you might not find it called that in other places. In the competition programming literature they are called "segment trees". However, this name conflicts with another specifically augmented binary search tree that represents a set of line segments in the plane. To remove this ambiguity, Danny Sleator coined the name "SegTree" and it has stuck.

The inefficiency of performing updates was due to the fact that if we edit element 0, there are n values in p that depend on it, and hence we do $O(n)$ work in updating everything.



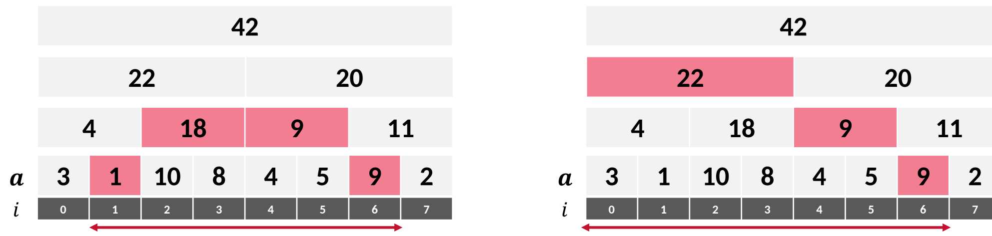
The *dependencies* here are what make the update algorithm slow. Is there instead an alternative way to break up the computation such that most of the intermediate calculations depends on fewer of the numbers? You may or may not have seen this idea before when seeking a *parallel algorithm* for computing the sum (or in general, any reduction) of a range of values. The left-to-right sequential sum is completely not parallel because of the dependencies, but a *divide-and-conquer* algorithm avoids this problem.



The divide-and-conquer sum has fewer dependencies because each element of the input only affects $\log_2 n$ intermediate vales produced by the computation. This means that if we update an element of the input, the output could be updated efficiently[2]. Its not clear yet though how we can actually answer **RangeSum** queries using this information, so lets figure that out now.

Doing queries The key idea is in figuring out how to build any interval $[i, j)$ that we might want to query out of some combination of the intervals represented by the divide-and-conquer tree. For example, if we wanted to query the interval $[1,7) = [1,6]$, we could add up the intervals $[1,1], [2,3], [4,5], [6,6]$ as shown below. Similarly, we can query $[0,7)$ as shown on the right.

[2]What we've actually made here is an extremely cool and powerful observation! It turns out that *parallel algorithms* are usually much easier to convert into dynamic algorithms / data structures than sequential ones, because they both share a common feature—both of them rely on having shallow dependence chains. An algorithm where all of the computations are dependent on the previous ones is hard to parallelize, and also hard to dynamize (make updatable) because changing a small amount of the input may change a large amount of the computation.

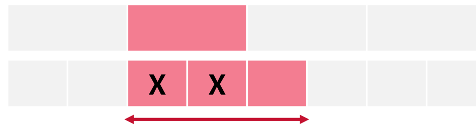


We need to somehow prove that we don't need too many intervals to make up any query interval. Because of course, we could always answer any query $[i, j)$ by summing up all of the intervals of size 1 from i to j, but that's just the first algorithm which takes $O(n)$ time.

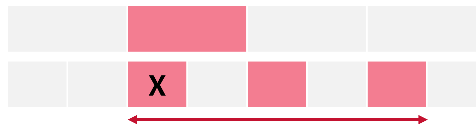
Lemma 6.1: Few blocks per level

Any interval $[i, j)$ can be made up of a set of disjoint intervals/blocks from the tree such that we use at most two intervals from any level.

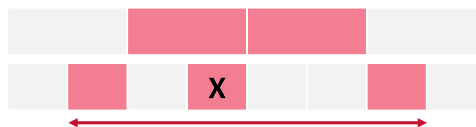
Proof. Suppose for the sake of contradiction that we use three *adjacent* blocks on the same level. Since each block is half the size of its parent, it must be the case that one of the two pairs of adjacent blocks could be replaced by its parent to cover the same interval, yielding a construction that uses fewer blocks.



Now suppose instead that we use three non-adjacent blocks on the same level. We can make a similar argument. Since the union of all the intervals gives one contiguous interval, it must be the case that the leftmost block is a right child of its parent (and symmetrically, that the rightmost block is a left child of its parent). If this were not the case, we could replace it with its parent to cover the same interval, yielding a construction that uses fewer blocks on this level.



It must be the case that the leftmost block is a right child and the rightmost block is a left child. So, the third block somewhere in between them could instead be covered by some intervals between the leftmost and rightmost block, yielding a construction that uses fewer blocks.



Applying this argument up the tree, there is a construction with at most two blocks per level. □

Since there is a construction with at most two blocks per level, we get the following corollary.

Corollary: $O(\log_2 n)$ blocks per query

Any interval $[i, j)$ can be made up of a set of at most $2\log_2 n$ disjoint blocks from the tree.

6.3 The data structure

Now that we have the key ingredients, we can put together the data structure. For the moment let's assume that n is a power of two. If it is not, we can always round it up to the next one by at most doubling it, so it won't affect our asymptotic bounds. We will talk about some fairly low level implementation details today, more than we might often do so in this course.

One of the things that make common tree data structures inefficient is that traversing them requires chasing down pointers throughout the tree, each of which points to a node that might reside far away from the former in memory. To make SegTrees more efficient, we will apply the same ordering trick from the *binary heap* data structure that you may have seen before. It works because the SegTree data structure is a so-called *perfect binary tree* (every internal node has two children, and all leaves are on the same depth).

Definition: The binary heap ordering trick

Given a perfect binary tree on N nodes, we can lay it out in memory using an array of size N using the following scheme:

- Place the root node at position 0
- Given the node at position i, place its left child in position $2i+1$
- Given the node at position i, place its right child in position $2i+2$

	0							
	1				2			
	3		4		5		6	
a	7	8	9	10	11	12	13	14

Notice that the number of nodes $N = 2n-1$, and that when using this trick, the n input elements of a are all stored in the *last* n elements of the array. We can use these ideas to implement the data structure. We will make use of the following convenience functions:

- $\text{Parent}(i) := \lfloor (i-1)/2 \rfloor$. Returns the parent of node i
- $\text{LeftChild}(i) := 2i+1$. Returns the left child of i

- RightChild(i) := $2i+2$. Returns the right child of i

Building the tree To build the tree, we start with the input $a[0\ldots(n-1)]$. We then perform the divide-and-conquer to compute all of the block values for the intermediate sums. The leaves of the tree using the binary heap indexing trick are the indices $n-1+i$ for each i, so we start by copying the input into those blocks before recursively computing all of the composite sums.

Algorithm: Building a SegTree

```
n : int
A : list⟨int⟩

SegTree(a : list⟨int⟩) {
  n := size(a)
  A := array⟨int⟩(2*n-1)
  for i in 0 to (n-1) do
    A[n + i - 1] = a[i]
  build(0, 0, n)
}

build : (u : int, L : int, R : int) -> () = {
  mid := (L + R)/2
  if LeftChild(u) < n-1 then
    build(LeftChild(u), L, mid)
  if RightChild(u) < n-1 then
    build(RightChild(u), mid, R)
  // reduce the value of the children
  A[u] = A[LeftChild(u)] + A[RightChild(u)]
}
```

Implementing updates To implement **Assign**(i, x), we just have to update the element at position i and all of its ancestor blocks in the tree. This takes $O(\log n)$ time and looks like this.

Algorithm: Updating a SegTree

```
Assign : (i : int, x : int) -> () = {
  u := i + n - 1
  A[u] = x
  while u > 0 do {
    u = Parent(u)
    // reduce the value of the children
    A[u] = A[LeftChild(u)] + A[RightChild(u)]
  }
}
```

Implementing queries To implement **RangeSum**(i, j), we need to figure out how to identify the set of $O(\log n)$ blocks that make up $[i, j)$. Fortunately, we can do this very naturally with recursion. What we will do is essentially the same as the original divide-and-conquer sum,

except we only need to recurse when we are on a block that contains some elements from $[i, j)$ and some elements not from $[i, j)$.

Algorithm: Querying a SegTree

```
RangeSum : (i : int, j : int) -> int = {
  return compute_sum(0, i, j, 0, n)
}

compute_sum = (u : int, i : int, j : int, L : int, R : int) -> int {
  if (i <= L && R <= j) then
    return A[u]
  else {
    mid := (L + R)/2
    if i >= mid then
      return compute_sum(RightChild(u), i, j, mid, R)
    else if j <= mid then
      return compute_sum(LeftChild(u), i, j, L, mid)
    else {
      left_sum := compute_sum(LeftChild(u), i, j, L, mid)
      right_sum := compute_sum(RightChild(u), i, j, mid, R)
      return left_sum + right_sum // reduce the value of the children
    }
  }
}
```

6.4 Speeding up algorithms with range queries

One thing that we want to practice in this course is choosing the right data structure for the job when designing an algorithm. We've seen in recent lectures that applying hashing can often drastically reduce the running time of algorithms. How can range queries help us design better algorithms? If we find ourselves designing an algorithm that requires summing over, or taking the minimum or maximum of a set of numbers in a loop, then we may be able to improve it by substituting that code with a range query. Lets see an example.

Problem: Inversion count

Given a permutation p of 0 through $n-1$, the number of *inversions* in the permutation is the number of pairs i, j such that $i < j$ but $p[i] > p[j]$.

For example, the inversion count of the sorted permutation is 0, because everything is in order. The inversion count of the reverse sorted permutation is $\binom{n}{2}$ since every pair is out of order. Lets start by designing an inefficient algorithm. Per the definition, we can just loop over all pairs $i < j$ and check whether $p[i] > p[j]$.

```
inversions : (p : list⟨int⟩) -> int {
  n := size(p)
  result : int = 0
  for j in 0 to n - 1 do {
    for i in 0 to j - 1 do {
      if p[i] > p[j] then
        result = result + 1
    }
  }
  return result
}
```

This will take $O(n^2)$ time, but can we improve it somehow using range queries? Lets try to decompose what the loops of the algorithm are doing. The first one is considering each index j in order, straightforward enough. The second loop is considering all $i < j$ and counting the number of such i's that have been seen previously such that $p[i]$ is larger than $p[j]$. Okay, this is sounding like some kind of range query now because we are counting the number of things in a range... How exactly do we express this using a SegTree?

We need a range of values to correspond to the **count** of the number of elements that we have seen that are greater than a certain value. Let's store an indicator variable for each element x that contains a 1 if we have seen that element, or otherwise 0. To count the number of elements that are greater than $p[j]$ will therefore correspond to a range query of **RangeSum**$(\boldsymbol{p[j], n})$. The optimized code for inversion counting therefore looks something like this.

Algorithm: Optimized inversion count using a SegTree

```
inversions : (p : list⟨int⟩) -> int {
  n := size(p)
  counts : SegTree = (list⟨int⟩(n, 0)) // SegTree containing n zeros
  result : int = 0
  for j in 0 to n - 1 do {
    result = result + counts.RangeSum(p[j], n)
    counts.Assign(p[j], 1)
  }
  return result
}
```

Since each **Assign** and each **RangeSum** cost $O(\log n)$, the total cost of this algorithm is just $O(n \log n)$, which is a great improvement over the earlier $O(n^2)$ one!

6.5 Extensions of SegTrees

6.5.1 Other range queries

We just figured out how to implement SegTrees that support an **Assign** and **RangeSum** API. What makes SegTrees so versatile, though, is that they are not limited to only performing sums

of integers. There was nothing particularly special about summing integers, except that it made for a good motivating example. Note that nowhere in our algorithm did we ever need to perform subtraction, which means we never made the assumption that the operation was invertible, which actually makes it even more general than the original "Approach #2" at the beginning! The exact same algorithm that we have just discussed can therefore also be used to implement range queries over **any associative operation**. In the code presented above, there are three lines commented with ***reduce the value of the children***, which add the values computed at the child nodes. Replacing just these three lines with any other associative operation yields a correct data structure for performing range queries over that operator.

For example, we can also compute the maximum or minimum over a range by replacing $X+Y$ in the three commented lines above with $\max(X,Y)$ or $\min(X,Y)$. There's also no reason to restrict ourselves to integers. We can use any value type, such as floating-point values, or tuples of multiple values, as long as we are able to provide the corresponding associative operator.

> ***Key Idea: SegTrees with any associative operation***
>
> SegTrees can be used with any associative operation, such as sum, min, max, but also even more complicated ones that you will see in recitation!

6.5.2 Other update operations

Our update operation for our vanilla SegTree is **Assign**(i,x), which sets the value of $a[i]$ to x. In some applications, instead of directly setting the value, we might want something slightly different. For example, we might want to *add* to the value instead of overwriting it. Fortunately, this can be supported with a combination of **Assign** and **RangeSum**. Note that we can always get the current value of $a[i]$ by performing **RangeSum**$(i,i+1)$, and then use that in an **Assign**. So to implement a new operation **Add**(i,x), which adds x to $a[i]$, we can just write

- **Add**(i,x): **Assign**$(i,x+$**RangeSum**$(i,i+1))$

The **Add** operation calls a constant number of SegTree operations, and hence it also runs in $O(\log n)$ time. This operation will be convenient for the next extension.

6.5.3 Flipping the operations

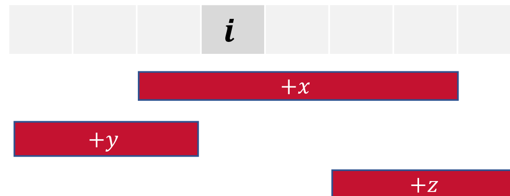
Our vanilla SegTree supports *point updates* and *range queries*, that is, we can edit the value of one element of the sequence and then query for properties (e.g., sum, min, max) of a range of values in $O(\log n)$ time. What if we want to do the opposite? Lets imagine that we want to support the following API over a sequence of n integers:

- **RangeAdd**(i,j,x): Add x to all elements $a[i],\ldots,a[j-1]$
- **GetValue**(i): Return the value of $a[i]$

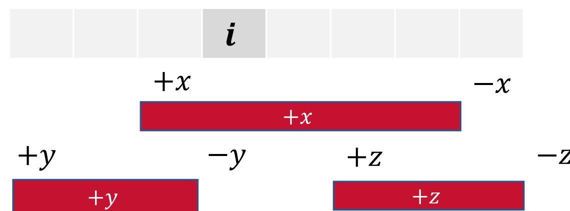
Rather than come up with a brand new data structure, lets try to use our original SegTree as a black box and reduce this new problem to the old one. This should always be your first choice

when designing a new data structure or algorithm (can I reduce to something that I already know how to solve? This is almost always easier than designing something new from scratch!)

The idea Somehow we need to convert range additions into just a single update, and range sums into the ability to get a specific value. How might we do that? Well, notice that at a particular location i, the value $a[i]$ is equal to the *sum* of all of the **RangeAdd**s that have touched location i. That sounds like **RangeSum** should be able to help us then...



Consider the diagram above. The value of **Get**(i) is affected only by $+x$ since the ranges $+y$ and $+z$ do not touch i. Notice that more specifically, the value at i is the sum of all of the ranges whose starting point is at most i, but whose ending point is at least i. We can represent this using *prefix sums*.



Notice that the value of i is just the sum of the $+x$'s that occur at or before i, then subtract the $-x$'s that occur before at or before i (since the interval ended before it reached i). This is exactly just the prefix sums of all of these $+x$'s and $-x$'s up to position i. Therefore, we can use the **RangeSum** method to compute this prefix sum and hence implement **Get**.

Algorithm: RangeAdd and Get

We can implement the **RangeAdd** and **Get** API in terms of **Add** and **RangeSum** as follows.

- **RangeAdd**(i, j, x): **Add**(i, x); **Add**($j, -x$)
- **Get**(i): **return RangeSum**($0, i+1$)

Both **RangeAdd** and **Get** call a constant number of SegTree operations, and hence they both run in $O(\log n)$ time as well.

Note that in this algorithm we had to make use of *subtraction*, which means that it isn't applicable to any arbitrary associative operation anymore, since not every associative operator has an inverse. This algorithm is therefore only applicable to invertible associative operations.

6.5.4 Range queries *and* range updates

Optional content — Not required knowledge for the exams

We have now seen how to implement point updates with range queries (i.e., **Assign** and **RangeSum**) and the opposite set of operations, range updates with point queries (i.e., **RangeAdd** and **Get**). Wouldn't it be amazing if we could get the best of both worlds and have *both*? It would be amazing if there existed a data structure that supported the following API:

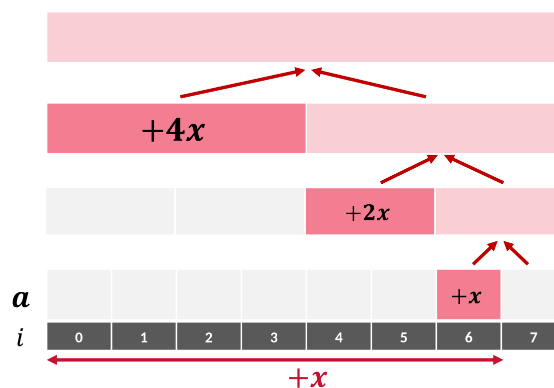
- **RangeAdd**(i, j, x): Add x to all elements $a[i], \ldots, a[j-1]$
- **RangeSum**(i, j): Return $\sum_{i \leq k < j} a[k]$.

This almost sounds too good to be true, but it turns out that there is a very clever reduction that involves using *two SegTrees* as black boxes that implements both of these operations in $O(\log n)$ time. You *might* see this in recitation, or as a practice problem later!

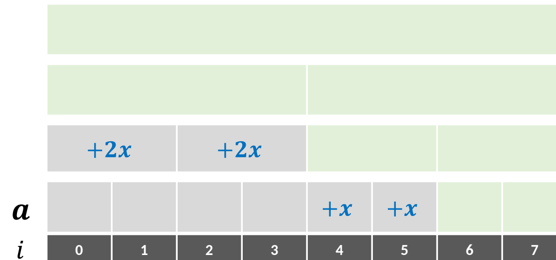
Similar to before, this extension specifically works for sum and addition because of the fact that addition is invertible, and hence it wouldn't work for querying the max or min of a range.

Can we do even better? At this point maybe we are starting to ask for too much, but would it be possible to support range queries for more general associative operations again, while also supporting range update operations too? Surely we are dreaming at this point, but it turns out to still be possible! We won't delve into the full implementation details, but it is super cool to know that this is possible, so lets just sketch a high-level overview of the how it works. The key technique is called **lazy propagation**.

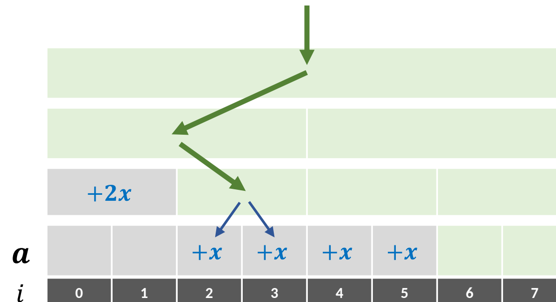
Lazy propagation Suppose we want to perform **RangeAdd**(i, j, x). The naive way would be to manually add x to all elements $a[i], \ldots, a[j-1]$ and then reduce the ancestors of these nodes in the tree, but this would take $O(n)$ time. Instead, lets use the same idea that we used for querying and start by identifying a set of at most $2\log_2 n$ disjoint blocks that make up the interval $[i, j)$. What if we only performed the update operation $+x$ on these blocks? Well, for each block that was affected, if the block contained s elements, then the sum would increase by $s \cdot x$. Then, we could reduce the ancestors of the affected blocks to update their values, too. This would leave us with a partially updated tree of blocks like so...



All of the red blocks have the most up-to-date value, but the descendants of the dark red blocks are all *out of date*, they are missing the update! To fix this, we remember for each child block of the updated blocks, *a "pending" value*, which indicates that an update is still waiting to be applied to that block and its descendants. In the following diagram, green blocks are up-to-date, and gray blocks are out of date. The blue values indicate the pending updates that are waiting to be applied.



So, when do we actually apply the pending updates? Well, if we never traverse to those blocks then we never need to, because we don't care about their values! Only when we actually traverse to that block in the future as part of a subsequent update or query do when then *apply* the update and then *propagate* the update to the block's children. Hence the name lazy propagation. We only propagate the update when we actually need it, instead of when we first perform the operation!



In the diagram above, a future query visits a block with a pending update $+2x$, so it applies it to the value of the block and then *propagates* the update, which creates pending updates of $+x$ on both of the children. That's it! With lazy propagation, you can have range updates and range queries, and you don't need your operation to be invertible since we didn't make use of subtraction this time.

Lecture 7

Amortized Analysis

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

Objectives of this lecture

In this lecture, we want to:

- Understand amortized analysis, how it differs from worst- and average-case analysis.
- See different methods for amortized analysis (aggregate, bankers, potential)
- Practice using the method of *potential functions* for amortized analysis
- See some examples of data structures and their performance using amortized analysis

Recommended study resources

- CLRS, *Introduction to Algorithms*, Chapter 17, Amortized Analysis

7.1 The ubiquitous example: Dynamic arrays (lists)

Every (good) programming language has an *array* type. An array is usually defined to be a *fixed-size* contiguous sequence of elements. However, it is extremely common for programmers to not know exactly how large an array needs to be up front. Instead, they need something more

general than this, an array where you can increase the size and add new elements over time. Luckily, most programming languages also supply such a container in their standard libraries!

Interface: List

The List API consists of:

- `initialize()`: Create an empty list
- `append(`x`)`: Insert x at the end of the list
- `get(`i`)`: Read and return the ith element of the list

`C++` provides this with `vector`, Java has `ArrayList`, and Python has `list` (the type you get when you write something between square brackets, `[...]`). In this lecture, to avoid ambiguity between this data structure, and a fixed-size array, we'll borrow from Python and refer to this data structure as a *list*. When we say array, we will mean a fixed-size array. So, how would we implement a list type from scratch if all we have access to are regular arrays? One option would be to represent a list of size n as an array of size n, then allocate a new array of size $n+1$ and move over all of the existing elements whenever we want to append a new element, but this would make append cost $O(n)$ time! We'd like to be much much faster than this.

The standard solution is to use *doubling* resizing. We maintain an array of some *capacity* $c \geq 1$ in which the first n of the slots are the actual list elements, and the remaining slots are free space for future elements, so $c \geq n$. When we want to append to the list, we check whether $c > n$, and if so, we can just insert the new element in the next available position, increment n, and we are done. If $c = n$, i.e., the current array is full, we allocate a new array of size $2c$, i.e., double the size! We can then move all of the existing elements over to the new array, throw away the old one, and insert the new element. To make this concrete, lets add another method[1] to our data structure:

- `grow()`: Double the current capacity of the array, moving the elements from the old array into the new array. This costs n since we move n elements.

So, now we can concretely write down our algorithm in terms of this method.

Algorithm 7.1: Doubling List

We implement the API of List as follows:

- `initialize()`: Create an array with capacity 1. ($n = 0, c = 1$)
- `append(`x`)`: If $c = n$ then `grow()`. Write x at position n and increment n.
- `get(`i`)`: Return the ith element of the array
- `grow()`: Double c and create a new array with capacity c, move every element from the old into the new array.

[1] If you like thinking in terms of Object-Oriented Programming, this is a *private* method, its not available to the user, but we will use it internally and it will help our analysis to separate it out from the rest of the operations.

How fast is this solution? Well, in the best case when $c > n$, we can just insert the new element and be done in $O(1)$ operations. That's pretty great. But in the worst case, we have $c = n$, and we have to spend $O(n)$ time moving all of the n elements over to the new array, which costs $O(n)$. So, in the worst case, this new algorithm still takes $O(n)$ time. Does that mean this is really a bad algorithm, though? Maybe in this case it is not our algorithm that is bad, but our analysis. By considering the worst-case performance for every append, we are really being too pessimistic, because it is impossible for every append to trigger the worst case. After triggering the worst case, it is impossible to trigger another one until we have performed $c/2$ more appends, so we can make our analysis better by considering *an entire sequence* of operations, rather than just thinking about one operation at a time. This is the key idea behind amortized analysis.

Key Idea: Amortized analysis

The key idea of amortized analysis is to consider the worst-case cost of a **sequence of operations** on a data structure, rather than the worst-case individual cost of any particular operation.

This means that amortized analysis is not applicable an isolated run of a single algorithm, e.g., it wouldn't make logical sense to ask about the amortized cost of Quicksort. Rather, we always talk about the amortized cost of a sequence of operations on a data structure.

So now that we have the idea of amortized analysis, how do we actually do it? It turns out that there are several ways, and we will see many of them in this lecture. You may have already seen some of them in your previous classes. The first method that we will analyze is probably the simplest, but the least general. It is called the *aggregate method*, or *total cost method*.

Definition: The aggregate method for amortized analysis

In this method, we will simply add up the **total cost** of performing a sequence of m operations on the data structure, then divide this by the number of operations m, yielding an average cost per operation, which we will define as our amortized cost!

This is the simplest and least general method of analysis because we will end up assigning each operation the same amortized cost. If there are multiple kinds of operations, you could choose to split the cost unevenly, giving different amortized costs to each operation. It is extremely important to not confuse this kind of analysis with *average-case* analysis, even though both involve averages!

Okay, we have to do one last thing before we analyze our algorithm, which is to decide on a cost model. When we were looking at sorting and selection, we used the comparison model which just counts the number of comparisons performed by the algorithms. This new algorithm performs no comparisons, so that would be a rather bad choice for this problem! We will stress that the cost model is always going to be somewhat arbitrary and there is no one correct choice. A good cost model should try to capture the costs of the most important/expensive steps of the algorithm so that it gives as realistic of a prediction of its performance as possible. It should also ideally be as simple as possible so that we don't make our analysis more difficult

than it needs to be. Lets go with the following simple cost model:

- Writing a value to any location in an array costs 1
- Moving a value from one array to another costs 1
- All other operations are free

Now we are ready! Lets use this model and the aggregate method to analyze the algorithm.

Lemma

The cost of a sequence of m append operations using array-doubling is at most $3m$.

Proof. After performing m appends, the number of elements in the list is $n = m$. First, lets count the cost of all of the append operations not including the growing operations (we will account for those separately). The cost of inserting elements is just 1, so in total this costs m.

Now we consider the cost of growing. After m appends, the capacity of the array will be $c = [[m]]$ (this notation means the smallest power of two that is at least m, also called the "super ceiling of m"). Also, notice that $[[m]] \le 2m$ for any value of m. Now, since the array is capacity c, the most recent grow must have incurred a cost of $c/2$ since the previous capacity was $c/2$ and that many elements were moved from the old array to the new array. Furthermore, the grow operation before that must have cost $c/4$, and so on. So, the costs of the grow operations is

$$1+2+4+\ldots+\frac{[[m]]}{2} \le 1+2+4+\cdots+m \le 2m.$$

Therefore in total, the cost of a sequence of m append operations is at most $m+2m = 3m$. □

The aggregate method for amortized analysis therefore gives us the following result.

Theorem: The amortized cost of array-doubling lists

The amortized cost (using the aggregate method) of `append` using the array-doubling algorithm is 3

Proof. The lemma tells us that a sequence of m append operations costs at most $3m$, hence each append in the sequence costs at most 3 on average, which is the amortized cost according to the aggregate method. □

7.2 The Bankers Method

So far so good, but we're going to need a better way to keep track of things for more complex problems. You may have previously heard of the *bankers method* (also known as the *accounting method*, or the *charging method*) for amortized analysis. This is a more general and powerful method of amortized analysis than the aggregate method since it allows us to assign different costs to different operations, but in exchange, it requires more creativity to use.

Definition: The bankers method of amortized analysis

In the bankers method, each operation has an actual cost as specified by the cost model, and additionally may choose to pay an extra cost (a **credit**) to store in the data structure for later use. A future operation may use this credit to offset the cost of an expensive operation. The amortized cost of an operation is its actual cost in the cost model, plus any credit it pays for, minus the value of any previous credit that it consumes.

Now lets see an example of using the bankers method to analyze our list.

Theorem: The amortized cost of array-doubling lists using the bankers method

The amortized cost (using the bankers method) of `append` using the array-doubling algorithm is 3

Proof. We will use the following charging scheme: Whenever we insert a new element into the list, we will pay a credit of 2 and leave it on the list element. Therefore, the cost of an append, not counting the grow operation is 3.

Now consider what happens when a grow operation is triggered. The array must be full ($c = n$), and the final half of the elements ($n/2$ of them) will each possess an unused credit of 2. Therefore, there is a credit of n in the data structure. We consume this credit to pay for the cost of moving the n elements to the freshly allocated array (which costs exactly n), so the amortized cost of the resizing procedure is zero. Therefore, the amortized cost of any append operation, whether it triggers a resizing or not is always 3. □

Thankfully, we get the same amortized cost that we got from the aggregate method, which suggests that the method also makes sense. An important thing that we have to be careful of and keep in mind when using the bankers method is that we must never spend credit that doesn't exist. We must be able to argue that the necessary amount of credit has been placed in data structure by previous operations, and has not already been consumed by a previous operation. In the argument above, we note that we only consume credit when growing occurs, and hence when a grow triggers, the final $n/2$ element must each possess 2 unused credits.

7.3 The Potential Method

The bankers method is a nice improvement over the aggregate method since it can be used to analyze trickier data structures, or data structures with different costs per operation. We can do even better with an even more general method called the *potential method*. The potential method is the most general, but therefore most difficult-to-use method that we will see for amortized analysis, so we will spend the rest of this lecture working with it.

In the potential method, we define a *potential function* Φ, which maps a *data structure state* S to a real number $\Phi(S)$. We will often use the notation S_i to denote the state of the data structure after applying the ith operation, and S_0 to denote the initial state of the data structure. The

potential function can be thought of as a generalization of the credit in the data structure in the bankers method.

Definition: The potential method for amortized analysis

Consider a sequence of m operations $\sigma_1, \sigma_2, \ldots, \sigma_m$ on the data structure. Let the sequence of states through which the data structure passes be $S_0, S_1, \ldots, S_m$. Notice that operation σ_i changes the state from S_{i-1} to S_i. Let the actual cost of operation σ_i in the cost model be c_i. Given a potential function Φ, we then define the amortized cost ac_i of operation σ_i by the following formula:

$$ac_i = c_i + \Phi(S_i) - \Phi(S_{i-1}),$$

In plain English, we can describe the amortized cost as

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

Lets compare this to the bankers method for a moment. If an operation pays a credit of p, we can consider that as equivalent to increasing the potential by p. If an operation consumes p credits, we can consider that as the same as decreasing the potential by p. With this correspondence, we can observe that the amortized cost defined by the potential method is the same as the amortized cost defined by the bankers method. This shows that the potential method is indeed a generalization of the bankers method.

Returning to the general potential method, if we sum up the amortized costs of a sequence of operations, we obtain the following formula:

$$\sum_i ac_i = \sum_i (c_i + \Phi(S_i) - \Phi(S_{i-1})) = \Phi(S_m) - \Phi(s_0) + \sum_i c_i.$$

Note that what happened here is that the terms *telescoped*. The $\Phi(S_i)$ terms all appeared once as a positive and once as a negative, so they all canceled out, except for the first and last terms which each only appeared once. Rearranging we get

$$\sum_i c_i = \left(\sum_i ac_i\right) + \Phi(S_0) - \Phi(S_m).$$

If $\Phi(S_0) \le \Phi(S_m)$ (as will frequently be the case) we get

$$\sum_i c_i \le \sum_i ac_i.$$

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 total amortized cost is indeed an upper bound on the total actual cost, or, the average actual cost is at most the average amortized cost. It is very common (but not required) to define our potential functions such that $\Phi(S_0) = 0$ and $\Phi(S_i) \ge 0$ for all i. Doing so guarantees that the bound above holds and removes the need for us to do an additional proof that $\Phi(S_m) \ge \Phi(S_0)$ to show that our amortized cost is actually valid.

Most of the art of doing an amortized analysis is in choosing the right potential function. This is typically the hardest part. 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 $\Phi(S_0) - \Phi(S_m)$ appropriately.

7.4 Lists Revisited

Let's do the analysis of the array-doubling list using the potential method. The hardest part is coming up with a good potential function. This often involves making educated guesses and then iterating with trial and error. There's no real way to guarantee that you'll pick the right one the first time.

Some key ideas to keep in mind are these:

- The goal is to make operations that have an expensive actual cost have a much cheaper amortized cost. So we want to rig the potential so that it *goes down* a lot when an expensive operation occurs, so that the change in potential is negative, making the amortized cost less than the actual cost.
- Operations that were previously cheap will have to get more expensive to compensate. Much like the bankers method, we want each cheap operation to result in the potential *increasing*, so that their amortized cost will be more than their actual cost.

In this case, the expensive operation that we are trying to cheapen is the resizing. If we can identify a property of the data structure that changes drastically when a grow occurs, then we want the potential function to go down according to this property. Since the actual values of the list do not affect the cost, the only properties of the list that matter are the capacity c and the current number of elements n. Here's an observation that we might exploit. As we append more elements, the value of n gets closer to the value of c. When a grow occurs, c gets larger again and this gap widens. So the gap between n and c looks to be just the quantity we want to base our potential on!

Let's start our trial-and-error loop.

- (**First guess**) Our first natural guess for the potential will be

  $$\Phi(n, c) = n - c,$$

  since this has the properties discussed above. Unfortunately it has the undesirable property of never being positive. We'd like to stick to our plan of having our potential function never be negative, so we need something different.
- **(Second guess)** We'd still like to have the property that Φ increases when we append a new element, and decreases when we grow, but we want it to be non-negative too. Lets use the fact that since the capacity doubles when $n = c$, we have $n \geq \frac{c}{2}$, and change our potential to

  $$\Phi(n, c) = n - \frac{c}{2}.$$

This looks promising. It has the properties we want and appears to always be non-negative[2]. Lets do the analysis and see if it works. Consider an append operation, and as usual, forgo analyzing the cost of a grow initially. The actual cost $ac = 1$ and we increase n by 1, so the potential increases by 1. The amortized cost so far is therefore 2.

Now we consider the cost of a grow. The actual cost $ac = n$, but what about the potential? Well, since we just triggered a grow, it must be true that $n = c$, so $\Phi(S_{i-1})$, i.e., the initial potential, must be $n - \frac{n}{2} = \frac{n}{2}$. After performing the grow, it will now be the case that $c = 2n$, so $\Phi(S_i) = n - \frac{2n}{2} = 0$. Therefore, the potential decreased by $\frac{n}{2}$. The amortized cost of a grow is therefore

$$n + \left(0 - \frac{n}{2}\right) = \frac{n}{2}.$$

Hmmm, so that didn't quite work. It **almost** worked, though. The potential went down by $\frac{n}{2}$, which is proportional to the actual cost of n that we were trying to cancel. Really we just needed the potential to change by twice as much. So, lets correct our potential by making it twice as large!

- (**Third time's a charm**) Using our latest observation, lets define our new potential function to be

$$\Phi(n, c) = 2\left(n - \frac{c}{2}\right).$$

Now we do the analysis again and see what happens. Consider an append operation separate from the cost of any grow. We pay an actual cost of $ac = 1$, and we increase n by 1 which increases the potential by 2. The amortized cost so far is 3.

Now we analyze the grow method. The actual cost is $ac = n$, and since $n = c$, the original potential is $\Phi(S_{i-1}) = 2\left(n - \frac{n}{2}\right) = n$, and the new potential (now that $c = 2n$) is $2\left(n - \frac{2n}{2}\right) = 0$. So the difference in potential is $-n$, and hence the amortized cost of a grow is

$$n + \left(0 - 2\left(\frac{n}{2} - 0\right)\right) = 0.$$

Hooray! Since the amortized cost of a grow is zero, the amortized cost of any append is 3.

Are we done? Not 100%. We still have to check our initial condition on the potential. Remember that we want $\Phi(S_m) \geq \Phi(S_0)$. It turns out that we were wrong about our Φ never being negative, because when we first initialize the data structure, we have $n = 0, c = 1$, which means that $\Phi(0, 1) = -\frac{1}{2}$. Oops. Luckily this doesn't matter at all because it is still true that $\Phi(S_m) \geq \Phi(S_0)$. So we can conclude the following.

Theorem: The amortized cost of array-doubling lists using the potential method

The amortized cost (using the potential method) of `append` using the array-doubling algorithm is 3.

Proof. Choose the potential function $\Phi(n, c) = 2\left(n - \frac{c}{2}\right)$, and observe that, as above, the amortized cost of append is 3, and that $\Phi(S_m) \geq \Phi(S_0)$. □

[2] Except for a little corner case that we'll fix momentarily

7.5 An Even-more-dynamic Array

A data structure that supports deletes can both grow and shrink in size. It would be nice if the size that it occupies is not *too much* bigger than necessary. This is where a list that shrinks in size is useful. Lets try to design and analyze a data structure that supports the following operations.

- `initialize()`: create an empty list
- `append`(x): insert x at the end of the list
- `get`(i): Read and return the ith element of the list
- `pop()`: remove the last element of the list

As before, we will denote the capacity of the current array by c and the number of actual list elements by n. To support shrinking the list, we will replace our grow operation from earlier with two operations: grow and shrink. `grow()` increases the capacity of the array from c to $2c$, and `shrink()` decreases the capacity of the array from c to $c/2$.

Since our original cost model didn't specify deletions, lets add that in. Our new cost model is

- Writing a value to any location in an array costs 1
- Moving a value from one array to another costs 1
- Erasing a value from an array costs 1
- All other operations are free

Using these primitives, here's how we implement the interface.

- `initialize()`: create an empty list with capacity 2. ($c = 2$ and $n = 0$)
- `append()`: if $c = n$ then `grow()`. Now insert the element into the table.
- `pop()`: if $n = c/4$ and $c \geq 4$ then `shrink()`. Now erase the element from the table.

There is a little bit of subtlety in this design. The situation immediately after a `grow()` or a `shrink()` is that $n = c/2$. The key thing is that right after one of these expensive operations, the system is very far from from having to do another expensive operation. This allows it time to build up its piggy bank to pay for the next expensive operation.

Lets now try to analyze our growing-and-shrinking array using a potential function. Since the algorithm is quite similar, we would expect a similar potential function to do the trick. As mentioned earlier, one nice observation is that $c/2$ is still the "midpoint" in some sense of the capacity, since we will always have $n = c/2$ immediately following a grow *or* a shrink. So, it would be nice if the potential function still had the property that it was equal to zero when $n = c/2$.

When we perform an append operation, we have the same desire as before. We would like to charge 2 to the potential so that after performing $c/2$ of them, we have saved up c potential which is enough to pay for the grow. This suggests that as a starting point, our potential should still be $2\left(n - \frac{c}{2}\right)$ whenever $n \geq c/2$, the same as before!

How should we handle pops? They are a little different. If we start at half capacity right after a grow or shrink ($n = c/2$), it only takes $c/4$ pops to trigger a shrink, not $c/2$ like was the case for append. However, note that the shrink operation only has to move $c/4$ elements to the newly shrunk array (because $n = c/4$ when a shrink is triggered). So unlike grow, which requires each of its appends to pay for 2 moves, shrink only needs a 1 to 1 charge for each pop. This suggests that we do not need a constant of 2 for pop/shrink! This suggests the following potential:

$$\Phi(n, c) = \begin{cases} 2\left(n - \frac{c}{2}\right), & \text{if } n \geq \frac{c}{2}, \\ \frac{c}{2} - n & \text{if } n < \frac{c}{2}. \end{cases}$$

The first case handles the situation where the array is in a "growing state", it is currently larger than it was after the most recent resize, and it is heading towards needing a grow operation. The second case handles the situation where it is in a "shrinking state", where it is smaller than it was after the last resize, and it is heading towards a shrink operation.

Theorem 7.1: The amortized cost of the growing-shrinking list

Using the doubling-halving array data structure, the amortized cost of `append` is at most 3, the amortized cost of `pop` is at most 2, and the amortized cost of `initialize` is at most 1.

Proof. As usual, lets first consider the cost of an append operation without accounting for a grow (yet). The actual cost is 1, but what about the change in potential? Well now our potential function has two cases so we have to consider two cases. If $n \geq c/2$, then the math is the same as before. n increases by 1, so the potential increases by 2. If $n < c/2$, then the potential actually decreases by 1. Therefore the worst-case potential increase is 2, and hence the amortized cost of append (not including grow) is at most 3.

Now we account for grow. This is again the same as the doubling array. Before growing, we have $n = c$ and hence the potential is n. After growing the potential is zero, so the potential has dropped by n, which is exactly the cost of moving the n elements to the new array, and hence the amortized cost of grow is zero.

Now lets look at pop and shrink. First, consider the cost of pop separately to the cost of shrink. Erasing the element from the array costs 1, and n decreases by 1. How does this affect the potential? If $n \geq c/2$, then the potential goes down by 2. If $n < c/2$, then the potential increases by 1. So, in the worst-case, the potential increase is 1, and hence the amortized cost of pop (not including shrink) is at most 2.

Now we account for shrink. Before a shrink occurs, $c = 4n$, and hence the initial potential is n. After a shrink occurs, the potential is zero, so it has decreased by n, which is exactly the cost of moving the n elements to the new array. Therefore, the amortized cost of shrink is zero.

Lastly, we should consider the final and initial potential. $\Phi(S_0) = 1$ and $\Phi(S_m) \geq 0$, so $\Phi(S_0) - \Phi(S_m) \leq 1$. We could modify our potential function to fix this, but I'm just going to charge that 1 to be the cost of initialize for simplicity. □

Exercises: Amortized Analysis

Problem 10. Describe the difference between amortized analysis using the aggregate method, and average-case analysis. How are they different? What about *expected* cost analysis? Where does this fit in and how is it different from the first two? Make sure you understand the differences!

Problem 11. Give a proof for Theorem 7.1 using the banker's method. Where would you put the banker's tokens in the data structure?

Problem 12. Suppose we change `pop()` to shrink when $s = n/2$, show a sequence of operations that incur large amortized cost.

Problem 13. Modify the potential function used in the proof of Theorem 7.1 so that we do not need to charge 1 to the cost of initialization (in other words, change the potential so that the initial potential is zero but the proof still works).

Lecture 8
Union-Find

In this lecture we describe the *disjoint-sets* problem and the family of *union-find* data structures. This is a problem that captures (among many others) a key task one needs to solve in order to efficiently implement Kruskal's minimum-spanning-tree algorithm. We describe several variants of the union-find data structure and prove that they achieve good amortized costs.

Objectives of this lecture

In this lecture, we will

- Design a very useful data structure called *Union-Find* for the *disjoint sets* problem
- Practice amortized analysis using potential functions

Recommended study resources

- CLRS, *Introduction to Algorithms*, Chapter 21 (3rd edition) or Chapter 19 (4th edition), Data Structures for Disjoint Sets

8.1 Motivation

To motivate the disjoint-sets/union-find problem, let's recall Kruskal's Algorithm for finding a minimum spanning tree (MST) in an undirected graph. Remember that an MST is a tree that includes all the vertices and has the least total cost of all possible such trees.

Kruskal's Algorithm:

> Sort the edges in the given graph G by weight and examine them from lightest to heaviest. For each edge (u, v) in sorted order, add it into the current forest if u and v are not already connected.

Today, our concern is how to implement this algorithm efficiently. 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 *disjoint-sets* or *union-find* problem.

8.2 The Disjoint-Sets / 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. Each set will have a *representative element* (or *canonical element*) that is used to identify it. The representative element is arbitrary, but it is important is that it is consistent so that we can identify whether two elements are in the same set.

Interface: Union-Find

The disjoint-sets/union-find API consists of:

MakeSet(x): Create a new set containing the single element x. Its representative element is x. (x must not be in another set.)

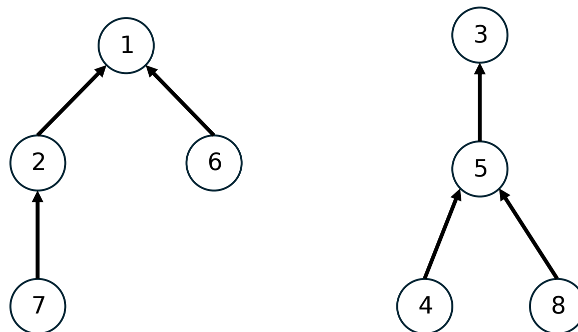
Find(x): Return the representative element of the set containing x.

Union(x, y): x and y are representative elements of two different sets. This operation forms a new set that is the union of these two sets, and removes the two old sets.

Implementing Kruskal's Algorithm 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 first compute v' =Find(v) and w' =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.

8.2.1 A Tree-Based Data Structure

We will represent the collection of disjoint sets by a forest of rooted trees. Each node in a tree corresponds to one of the elements of a set, and each set is represented by a separate tree in the forest. The root of the tree is the representative element of the corresponding set. We will refer to the total number of elements in all of the sets (i.e., the number of nodes in the forest) as n.



The trees are defined by parent pointers. So every node x has parent $p(x)$. A node x is a root of its tree if $p(x) = x$. To find the representative element of a set, it suffices to walk up the tree until we reach a root node, which must be the answer. To union two trees, we can simply make one tree a child of the other. To do this, we can first find the roots of the two trees by calling Find, and then setting one as the child of the other.

It will be convenient for our cost analysis to distinguish between the cost incurred by Union when it calls Find, and the actual cost of the step that joins the two trees. To do so, we define a subroutine **Link**, which is not part of the API but just used internally by Union. With all of this set up, here is a basic implementation.

Algorithm: Union-Find Forests

Union-Find forests (a.k.a. disjoint-set forests) implement the API as follows:

MakeSet(x): Create node x and set $p(x) \leftarrow x$.

Find(x): Starting from x, follow the parent pointers until you reach the root, r, then return r.

Union(x, y): **Link**(**Find**(x), **Find**(y))

Link(x, y): Set $p(y) \leftarrow x$.

We are going to analyze several variants of this data structure.

Analysis with no optimizations The vanilla version of the data structure unfortunately has terrible worst-case (even in an amortized analysis) performance. Too see this, suppose we create n sets and then Union them into a long chain of length n. Now every find operation on the bottom-most element of the chain costs $\Theta(n)$.

We will now explore two optimizations that substantially improve the performance of the data structure: *union-by-size* and *path compression*.

8.3 The union-by-size optimization

Our first optimization, and the simpler of two to analyze is *union-by-size*. The pathological cost of the no-optimization example was caused by the fact that we were able to build a long chain of nodes with the Union operation, allowing all subsequent Finds to be very expensive.

Key Idea: Union by size optimization

When performing the Union (Link) operation, always make the smaller (by number of nodes) tree the child of the larger tree.

To implement this, we augment each element x with an additional field $s(x)$. If x is a root of a tree, then $s(x)$ contains the size of the subtree rooted at x. If x is not the root of a tree, then

we do not care about the value of $s(x)$ since it would be too expensive to update $s(x)$ on every single node, and its value is never needed on nodes other than roots.

Algorithm: Link with union-by-size optimization

Augment each element x with a field $s(x)$. MakeSet(x) sets $s(x) \leftarrow 1$.

Link(x, y):
- **if** $s(x) < s(y)$ **then** swap(x, y),
- Set $p(y) \leftarrow x$,
- Set $s(x) \leftarrow s(x) + s(y)$.

With this minor change to the algorithm, the claim is that we now get great performance! Even better, we don't need amortized analysis for this one, we get a worst-case bound.

Theorem 8.1: Union by size performance

Consider the union-find forest data structure where the union-by-size optimization is used. Then MakeSet and Link each cost $O(1)$ and Find has a worst-case cost of $O(\log n)$.

Proof. MakeSet and Link each perform a constant number of operations regardless of the input, so they they cost $O(1)$.

To analyze the cost of Find, we observe that every edge of the tree is created by the union of two sets, and using union by size, the smaller tree must have always been made into a child of the larger one. Therefore, whenever the Find algorithm moves from a node to its parent, the total size of the rooted at the current node *at least doubles*. Since no tree has size more than n, the number of edges on any path to the root is at most $\log n$, so any Find operation costs at most $O(\log n)$ in *the worst case*. □

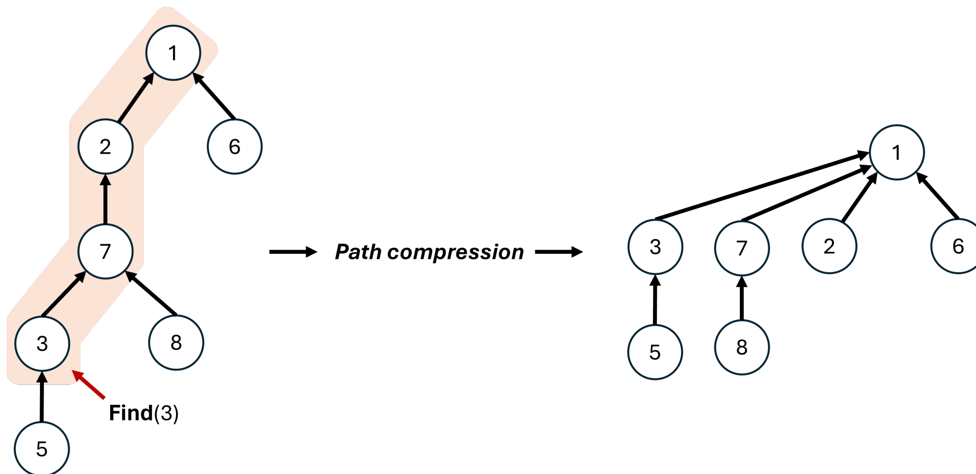
8.4 The path compression optimization

In the previous section, we improved the worst-case performance of Find by improving the Union operation, but leaving the Find operation unchanged. What if we instead want to improve the Find operation itself? Assuming worst-case Unions, is there a way to make the Find operation $O(\log n)$ cost on its own? With the power of amortized analysis, the answer is yes, and the technique is elegant and intuitive (though the analysis is hard!)

The worst case for a Find operation is encountering a long root-to-leaf path. When that happens, we can't avoid having to traverse it to the root, **but**, we can at least try to make the situation better for future Find operations that happen to come along one of the same nodes.

Key Idea: Path compression optimization

Each time the data structure performs a Find operation, it changes the parent of every node encountered on the path to point directly to the root so that future Finds do not have to travel the same long path.



Unlike union by size, this does not require augmenting the data structure with any additional fields, and the implementation is nice and elegant.

Algorithm: Path compression implementation

Find with path compression can be implemented elegantly with recursion.

Find(x): - **if** $p(x) \neq x$ **then** set $p(x) \leftarrow$ **Find**$(p(x))$,

- **return** $p(x)$

Note that we are not combining union by size with path compression (yet). We are just using path compression with worst-case unions. The claim is that this algorithm is also efficient, but we are going to need amortized analysis to prove it. The idea is that a single Find operation can be inefficient (the worst-case cost is still the same), but after performing an inefficient Find, the forest must become shallower as a result, making future Finds more efficient. To make the analysis clean, we will define a simple cost model that we will use.

Cost model We will analyze path compression under the cost scheme:

- **MakeSet**(x) costs 1,
- **Link**(x, y) costs 1,
- **Find**(x) costs the number of nodes on the path from x to the root.

Note that these costs are accurate up to constant factors in the word RAM, so our results will be asymptotically valid in the word RAM, but without having to deal with the arbitrary constants.

Theorem 8.2: Path Compression without Union by Size

Consider the forest-based union-find data structure where path compression is applied but union by size is not. Then the amortized cost of Link is $(1+\log n)$, the amortized cost of Find is $(2+\log n)$, and Makeset still costs 1.

The proof is not as simple as the proof for union by size, and uses a very non-trivial potential function. Before diving right into it, lets try to get a handle on how we might measure the potential. The main observation that we need is that *balance* defines the difference between an efficient Find and an inefficient Find (which must therefore be paid for by the stored potential). A very balanced tree can have low potential since operations will be cheap anyway, but an imbalanced tree will need a lot more potential to pay for the Finds.

Heavy and light nodes While balance appears to be the key thing that we care about, a particular Find operation doesn't care how balanced the tree is as a whole, it only cares about the specific nodes along the $x \to r$ path that it encounters. So what we need is a way of measuring how balanced a tree is on a per-node level. In other words, what would it mean for a single node to be good or bad for balance?

In a balanced tree, all of the children of a particular node x would have an even share of x's descendants as their descendants. Nodes that deviate significantly from this (such as a long chain of nodes) will break this. This motivates us to define the concept of **heavy** and **light** nodes. Let $\text{size}(x)$ be the number of nodes that are descendants of x (including x).

Definition: Heavy and light nodes

In a tree, a node u (other than the root) with parent $p(u)$ is called:

1. **heavy** if $\text{size}(u) > \frac{1}{2}\text{size}(p(u))$, i.e., u's subtree has a majority of $p(u)$'s descendants,
2. **light**, if $\text{size}(u) \leq \frac{1}{2}\text{size}(p(u))$, i.e., u's subtree has at most half of $p(u)$'s descendants.

A perfectly balanced tree would have no heavy nodes, only light nodes (except for the root). A maximally imbalanced tree (a chain) would consist entirely of heavy nodes (except for the root). We have the following very useful observation about heavy and light nodes.

Lemma 8.1: Heavy-light lemma

On any root-to-leaf path in a tree on n vertices, there are at most $\log n$ light nodes.

Proof. Consider a node x. If x is light, then by definition, $\text{size}(p(x)) \geq 2\text{size}(x)$. Since there are no more than n nodes in any tree, the number of times one can double the size of the current node is at most $\log n$. Therefore there are at most $\log n$ light nodes on any root-to-leaf path. □

This gives us an idea. We want the amortized cost of Find to be $O(\log n)$, and it costs us 1 for every node we touch on the root-to-leaf path. So, we can afford to traverse a path of light

nodes, since there are at most $\log n$ of them. We only get sad when we have to touch heavy nodes, because there could be a lot of them, so lets try to use a *potential function* to save up and pay for the heavy nodes!

So, how many heavy nodes can there be? Well, a lot, unfortunately, but what about how many heavy *children* can a particular node have? By definition, a node can only have one heavy child at a time since a heavy child must contain a majority of the descendants! When a Find operation path compresses a heavy child, another child *might* become a heavy child in its place. However, the node must therefore *lose over half of its descendants*, sine the heavy child was a majority of the subtree. This can not happen very many times. In particular, a node u with size(u) descendants can only halve its size log(size(u)) times before it has no children anymore!

Therefore, the number of heavy children a single node could ever have is bounded by log(size(u)), so the total number of heavy children we could *ever have in our tree* is

$$\Phi(F) = \sum_{u \in F} \log(\text{size}(u)),$$

which will be our potential function! As usual, log is base-2. It is important to note that this sum is over *every node in the forest*, not only the roots. This potential function is quite powerful and is also used in the analysis of other data structures, such as Splay Trees. It is, at some intuitive level, a very good potential function for measuring *how imbalanced a tree is*. It can be shown as an exercise that a perfectly balanced tree on n vertices has $\Theta(n)$ potential, while a long chain (the most imbalanced tree) has $\Theta(n \log n)$ potential.

This potential has a few nice important properties:

1. The potential is initially zero (all trees have size 1),
2. at any point in time the potential is non-negative,
3. the potential increases every time a Link is done,
4. and it decreases (or stays the same) every time a Find is done.

The first two properties mean that we can use the total amortized cost to bound the actual total cost. The last two properties tell us intuitively that the potential should be on the right track for the analysis, since union is the cheap operation (which we therefore want to make more expensive by paying into the potential) and find is the expensive operation, which we want to make cheaper by withdrawing from the potential.

Analysis of MakeSet Calling **MakeSet**(x) creates a new singleton set $\{x\}$ represented by a node with no parent or children and does not modify any of the existing sets. Therefore the potential contributed by all of the existing nodes does not change, and we just get one new term in the potential from the new node.

$$\Delta\Phi_{\text{MakeSet}} = \log(\text{size}(x)) = \log(1) = 0.$$

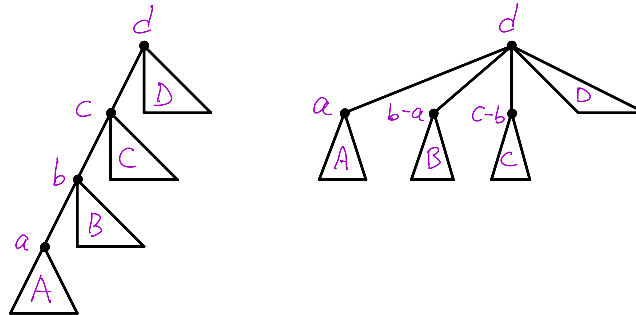
Since we just created it, the size of x's subtree is just one, and $\log 1 = 0$, so the potential does not change at all. Therefore the actual and amortized cost of **MakeSet** is 1.

Analysis of Link Consider a Link operation. Suppose the operation links a node y to a node x. The only node whose size changes is x. Let size(x) be the size of x before the Link and size$'(x)$ be the size after the Link. We know that size$(x) \geq 1$, because any tree has at least one node in it. So $\log(\text{size}(x)) \geq 0$. Similarly size$'(x) \leq n$ and $\log(\text{size}'(x)) \leq \log n$. So we have:

$$\begin{aligned}\Delta\Phi_{\text{Link}} &= \Phi_{\text{final}} - \Phi_{\text{initial}},\\ &= \log(\text{size}'(x)) - \log(\text{size}(x)),\\ &\leq \log n\end{aligned}$$

Since the actual cost of a Link is 1, the amortized cost of Link is at most $1 + \log n$.

Analysis of Find Let's consider the Find operation. The following figure shows a typical Find with path compression. On the left is the tree before and on the right is after. The triangles labeled with capital letters represent arbitrary trees, which are unchanged. A Find operation is applied to the node labeled a on the left. The cost of this operation is 4 (it touches 4 nodes).



The find path passes through vertices a, b, c, and d. These labels are the sizes of the nodes. Note that the node labeled b has size b before the Find, and has size $b - a$ after the Find. Similarly the node labeled c has size c before the Find and has size $c - b$ after the Find. Those are the only two nodes whose size changes as a consequence of the Find. In general, all nodes except for the first and last have their size decreased, while the first and last remain the same. Since sizes only decrease, the potential from every node can only decrease, never increase!

There are (1+# heavy nodes+# light nodes) on the Find path (the root is neither heavy nor light). We recall from Lemma 8.1 that there are at most $\log n$ light nodes on the path, so we are happy to pay for those in the final cost. Our goal is therefore to cancel out the cost of the heavy nodes using the potential function. Given any path from a vertex to its corresponding root node, we consider all of the *heavy nodes* on the path except the child of the root (because it does not move as a result of path compression). For every such vertex u, whose parent we will denote as p, the new size of p, which we will denote as size$'(p)$, is given by size$'(p) = \text{size}(p) - \text{size}(u)$.

By the definition of heavy, u and p satisfy size$(u) > \frac{1}{2}\text{size}(p)$, so from this equation, we have

$$\begin{aligned}\text{size}'(p) &= \text{size}(p) - \text{size}(u),\\ &< \text{size}(p) - \frac{1}{2}\text{size}(p),\\ &< \frac{1}{2}\text{size}(p),\end{aligned}$$

i.e., the size of p at least halves! This means that the change in potential at node p is

$$\begin{aligned}\Delta\Phi_{\text{at node } p} &= \log(\text{size}'(p)) - \log(\text{size}(p)),\\ &< \log\left(\frac{1}{2}\text{size}(p)\right) - \log(\text{size}(p)),\\ &= -1.\end{aligned}$$

Therefore, the potential of every node (other than the root) whose child on the Find path is heavy *decreases by at least one*. The overall decrease in the potential is at least (# heavy nodes−1) and hence the amortized cost of a find operation is

$$\begin{aligned}\text{Amortized cost of Find} &\le \underbrace{1 + \#\text{ heavy nodes} + \#\text{ light nodes}}_{\textbf{actual cost}} - \underbrace{(\#\text{ heavy nodes} - 1)}_{\Delta\Phi},\\ &\le 2 + \#\text{ light nodes},\\ &\le 2 + \log n.\end{aligned}$$

8.5 Both optimizations at once

This problem has been extensively analyzed. In the 1970s, a series of upper bounds were proven for the amortized cost of Union and Find, going from $\log n$ to $\log\log n$ to $\log^* n$, and finally to $\alpha(n)$. Bob Tarjan proved the final result and matched it with a corresponding lower bound, thus "closing" the problem and showing that $O(1)$ time is impossible.

Here $\log^* n$ denotes the function that counts the number of times you have to apply log to n until it doesn't exceed 1. The function $\alpha(n)$ is an inverse of Ackermann's function, and grows insanely slowly[1], even slower than $\log^* n$. We won't be studying the proofs of these results since that would take another entire lecture, but they are useful results to know so that you can analyze the runtime of algorithms that use Union-Find as an ingredient.

Theorem 8.3: Union-Find with union-by-size and path compression

Consider the forest-based union-find data structure with both path compression and the union-by-size optimizations. Then the amortized cost of MakeSet is $O(1)$, and the amortized costs of Link and Find are $O(\alpha(n))$.

The proof uses a very complicated potential function. If you want to read it, you can find it in CLRS (Chapter 19 in the fourth edition, or Chapter 21 in the third edition).

[1] For n up to the number of particles in the universe, $\alpha(n) \le 4$, so while it may not be a constant for theoretical purposes, in practice it is indistinguishable from a small constant

Exercises: Union Find

Problem 14. Consider the union-find forest data structure with path compression. Suppose we perform a sequence of n MakeSet operations, followed by m Unions, then f Finds *in that order*, i.e., the operations are not interleaved. Show that the total cost of this sequence of operations is $O(n+m+f)$, i.e., each operation takes constant amortized time. (Hint: define a potential function that is the degree of the root of the tree.)

Problem 15. Show that the potential function from Section 8.4 is $\Theta(n)$ for a perfectly balanced tree on n vertices and $\Theta(n \log n)$ for a chain of n vertices.

Problem 16. Suppose we implement the union-find forest data structure with both union-by-size and path compression. Show, using the same potential function as Section 8.4 that the cost of Union is $O(1)$.

Problem 17. Consider the union-find forest data structure with union by size. Show that the worst-case bound of $O(\log n)$ cost per Find operation is tight, i.e., describe an input for which the Find operation costs $\Omega(\log n)$.

Problem 18. Consider the union-find forest data structure with path compression. Show that the amortized bound of $O(\log n)$ cost per Union and Find operation is tight, i.e., describe an input with m Union operations and f Find operations for which the total cost is at least

$$\Omega(m \log n + f \log n).$$

Lecture 9

Dynamic Programming I

Dynamic Programming is a powerful technique that often allows you to solve problems that seem like they should take exponential time in polynomial time. Sometimes it allows you to solve exponential time problems in slightly better exponential time. It is most often used in combinatorial problems, like optimization (find the minimum or maximum weight way of doing something) or counting problems (count how many ways you can do something). We will review this technique and present a few key examples.

Objectives of this lecture

In this lecture, we will:

- Review and understand the fundamental ideas of Dynamic Programming.
- Study several example problems:
    - The Knapsack Problem
    - Independent Sets on Trees
    - The Traveling Salesperson Problem

Recommended study resources

- CLRS, *Introduction to Algorithms*, Chapter 15/14 (3rd/4th ed.), Dynamic Programming
- DPV, *Algorithms*, Chapter 6, Dynamic Programming
- Erikson, *Algorithms*, Chapter 3, Dynamic Programming

9.1 Introduction

Dynamic Programming is a powerful technique that can be used to solve many combinatorial problems in polynomial time for which a naive approach would take exponential time. Dynamic Programming is a general approach to solving problems, much like "divide-and-conquer", except that the subproblems will *overlap*.

You may have seen the idea of dynamic programming from your previous courses, but we will take a step back and review it in detail rather than diving straight into problems just in case you have not, or if you have and have completely forgotten!

> ***Key Idea: Dynamic programming***
>
> *Dynamic programming* involves formulating a problem as a set of *subproblems*, expressing the solution to the problem *recursively* in terms of those subproblems and solving the recursion *without repeating* the same subproblem twice.

The two key sub-ideas that make DP work are ***memoization*** (don't repeat yourself) and ***optimal substructure***. Memoization means that we should never try to compute the solution to the same subproblem twice. Instead, we should store the solutions to previously computed subproblems, and look them up if we need them again.

9.1.1 Warmup: Climbing Steps

Lets start with a nice problem to break down these key ideas. Suppose you can jump up the stairs in 1-step or 2-step increments. How many ways are there to jump up n stairs?

Where is the *substructure* in this problem? Well, with n stairs, we have two *choices*, either we jump up 1 or 2 steps. After jumping up 1 step, we will have $n-1$ steps remaining, or after jumping up 2 steps we will have $n-2$ steps remaining. So it sounds like some sensible smaller problems to consider would be the number of ways to jump up n steps for any smaller value of n. Lets define a function *stairs* which counts exactly this. We can evaluate stairs *recursively*

```
function stairs(n : int) -> int = {
  if (n <= 1) return 1;
  else {
    waysToTake1Step = stairs(n-1);
    waysToTake2Steps = stairs(n-2);
    return waysToTake1Step + waysToTake2Steps;
  }
}
```

We found the substructure in the problem, but we're not done yet. Implemented as such, the above code would perform exponentially many recursive calls because it would end up **repeatedly evaluating the same problem**. Notice that stairs(n) calls stairs($n-1$) and stairs($n-2$). But stairs($n-1$) *also* calls stairs($n-2$), so we will call that twice. Going deeper into the recursion, we will see that we compute the same values exponentially many times!

To rectify this, we apply the other key idea of dynamic programming, which is don't repeat yourself, aka. **memoization**. Lets store a lookup table of previously computed values, and instead of recomputing from scratch every time, we will just reuse values that already exist in the table! By convention we will refer to the lookup/memoization table as "memo". The most generic way to implement the memo table is to use a *dictionary* that maps subproblems to their values.

```
dictionary<int, int> memo;

function stairs(n : int) -> int = {
  if (n <= 1) return 1;
  if (memo[n] == None) {
    waysToTake1Step = stairs(n-1);
    waysToTake2Steps = stairs(n-2);
    memo[n] = waysToTake1Step + waysToTake2Steps;
  }
  return memo[n];
}
```

Note that for very many problems, the memo table does not need to be implemented as a hashtable dictionary. In the majority of problems, including this one, the subproblems are just identified by integers from $0 \ldots n$, so the dictionary can actually just be implemented as *an array*! A hashtable dictionary would only be required if the subproblem identifiers can not easily be mapped to a set of small integers.

9.1.2 The "recipe"

With these key ideas in mind, lets give a high-level recipe for dynamic programming (DP). A high-level solution to a dynamic programming problem usually consists of the following steps:

1. **Identify the set of subproblems** You should **clearly and unambiguously** define the set of subproblems that will make up your DP algorithm. These subproblems must exhibit some kind of *optimal substructure* property. The smaller ones should help to solve the larger ones. This is often the **hardest part** of a DP problem, since locating the optimal substructure can be tricky.

2. **Identify the relationship between subproblems** This usually takes the form of a *recurrence relation*. Given a subproblem, you need to be able to solve it by combining the solutions to some set of smaller subproblems, or solve it directly if it is a *base case*. You should also make sure you are able to solve the original problem in terms of the subproblems (it may just be one of them)!

3. **Analyze the required runtime** The runtime is **usually** the number of subproblems multiplied by the time required to process each subproblem. In uncommon cases, it can be less if you can prove that some subproblems can be solved faster than others, or sometimes it may be more if you can't look up subproblems in constant time.

This is just a *high-level* approach to using dynamic programming. There are more details that we need to account for if we actually want to implement the algorithm. Sometimes we are satisfied with just the high-level solution and won't go further. Sometimes we will want to go down to the details. These include:

4. **Selecting a data structure to store subproblems** The vast majority of the time, our subproblems can be identified by an integer, or a tuple of integers, in which case we can store our subproblem solutions in an array or multidimensional array. If things are more complicated, we may wish to store our subproblem solutions in a hashtable or balanced binary search tree.

5. **Choose between a *bottom-up* or *top-down* implementation** A bottom-up implementation needs to figure out an appropriate *dependency order* in which to evaluate the subproblems. That is, whenever we are solving a particular subproblem, whatever it depends on must have already been computed and stored. For a top-down algorithm, this isn't necessary, and recursion takes care of the ordering for us.

6. **Write the algorithm** For a bottom-up implementation, this usually consists of (possibly nested) for loops that evaluate the recurrence in the appropriate dependency order. For a top-down implementation, this involves writing a recursive algorithm with *memoization*.

9.2 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?[1]

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

Definition: 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 $O(nS)$. Lets go through our recipe book for dynamic programming and see how we can solve this.

[1] 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!

Step 1: Identify some optimal substructure Lets imagine we have some instance of the knapsack problem, such as our example $\{A, B, C, D, E, F, G\}$ above with total size capacity $S = 15$. Here's a seemingly useless but actually very useful observation: The optimal solution either does contain G or it does not contain G. How does this help us? Well, suppose it does contain G, then what does the rest of the optimal solution look like? It can't contain G since we've already used it, and it has capacity $S' = S - s_G$. What does the optimal solution to this remainder look like? Well, by similar logic to before, it must be the *optimal solution* to a knapsack problem of total capacity S' on the set of items not including G! (Formally we could prove this by contradiction again—if there was a more optimal knapsack solution for capacity S' without G, we could use it to improve our solution.) This is another case of *optimal substructure* appearing!

Step 2: Defining our subproblems Now that we've observed some optimal substructure, lets try to define some subproblems. Our observation seems to suggest that the subproblems should involve considering a *smaller capacity*, and considering one fewer item. How should we keep track of which items we are allowed to use? We could define a subproblem for every subset of the input items, but then we would have $\Omega(2^n)$ subproblems, and that's no better than brute force! But here's another observation: it doesn't really matter what order we consider inserting the items if for every single item we either use it or don't use it, so we can instead just consider subproblems where we are using items $1 \dots i$ for $0 \le i \le n$. Combining these two ideas, both the capacity reduction and the subset of items, we define our subproblems as:

$$\text{V}(k, B) = \text{The value of the best subset of items from } \{1, 2, \dots, k\} \text{ that uses at most } B \text{ space}$$

The solution to the original problem is the subproblem $V(n, S)$.

Step 3: Deriving a recurrence Now that we have our subproblems, we can use our substructure observation to make a recurrence. If we choose to include item k, then our knapsack has $B - s_k$ space remaining, and we can no longer use item k, so this gives us

$$v(k, B) = v_k + V(k-1, B - s_k) \qquad \text{if we take item } k$$

Otherwise, if we don't take item k, then we get

$$v(k, B) = V(k-1, B) \qquad \text{if we don't take item } k$$

Finally, we need some base case(s). Well, if we have no items left to use $k = 0$, that seems like a good base case because we know the answer is zero! So, putting this together, we can write the recurrence:

Algorithm: Dynamic programming recurrence for Knapsack

$$V(k, B) = \begin{cases} 0 & \text{if } k = 0 \\ V(k-1, B) & \text{if } s_k > B \\ \max\{v_k + V(k-1, B - s_k), V(k-1, B)\} & \text{otherwise} \end{cases}$$

Note here that we had to check whether $s_k > B$. In this case, we can't choose item k even if we wanted to because it doesn't fit in the knapsack, so we are forced to skip it. Otherwise, if it fits, we try both options of taking item k or not taking item k, then use the best of the two choices.

Step 4: Analysis We have $O(nS)$ subproblems and each of them requires a constant amount of work to evaluate for the first time. So, using dynamic programming, we can implement this solution in $O(nS)$ time.

A top-down implementation This can be turned into a recursive algorithm. Naively this again would take exponential time. But, since there are only $O(nS)$ *different* pairs of values the arguments can possibly take on, so this is perfect for memoizing. Let us initialize a 2D memoization table `memo[k][b]` to "unknown" for all $0 \le k \le n$ and $0 \le b \le S$.

```
function V(k : int, B : int) -> int = {
  if (k == 0) return 0;
  if (memo[k][B] != unknown) return memo[k][B];     // <- added this
  if (s_k > B) result := V(k-1,B);
  else result := max{v_k + V(k-1, B-s_k), V(k-1, B)};
  memo[k][B] = result;                              // <- and this
  return result;
}
```

Since any given pair of arguments to V can pass through the memo 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 `memo[k][B]` = `memo[k-1][B]` then we *didn't* use the kth item so we just recursively work backwards from `memo[k-1][B]`. Otherwise, we *did* use that item, so we just output the kth item and recursively work backwards from `memo[k-1][B-s_k]`. One can also do bottom-up Dynamic Programming.

9.3 Max-Weight Indep. Sets on Trees (Tree DP)

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 vertices are adjacent (i.e., 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 vertex r, which defines a notion of parents/children (and ancestors/descendents) for the tree. Lets go through our usual motions.

Step 1: Identify some optimal substructure Suppose we choose to include r (the root) in the independent set. What does this say about the rest of the solution? By definition, it means that the children of the root are not allowed to be in the set. Anything else though is fair game. In particular, for every *granchild* of the root, we would like to build a max-weight independent set rooted at that vertex. A proof by contradiction would as usual verify that this has optimal substructure.

On the other hand, if we choose to not include r in our independent set, then all of the children are valid candidates to include. Specifically, we would like to construct a max-weight independent set in all of the subtrees rooted at the children (which may or may not contain those children themselves).

Step 2: Define our subproblems The optimal substructure suggests that our subproblems should be based on *particular subtrees*. This general technique is often referred to as "tree DP" for this reason. Our set of subproblems might therefore be

$$W(v) = \text{the max weight independent set of the subtree rooted at } v$$

The solution to the original problem is $W(r)$.

Step 3: Deriving a recurrence Like many of our previous algorithms, we build the recurrence by casing on possible decisions we can make. Keeping with the spirit of that, it seems like the decision we can make at any given problem $W(v)$ is whether or not to include the root vertex v in the independent set. If we choose to not include it, then we should just recursively find a max-weight set in the children's subtrees. We let $C(v)$ be the set of children of vertex v. Then we have

$$W(v) = \sum_{u \in C(v)} W(u) \qquad \text{if we don't choose } v.$$

Suppose we do choose v, then what can we do? As discussed, we can no longer include any of v's children without violating the rules, but we can consider any of v's grandchildren and their subtrees. Let $GC(v)$ denote the set of v's grandchildren. Then we have

$$W(v) = w_v + \sum_{u \in GC(v)} W(u) \qquad \text{if we choose } v.$$

Finally, what about base cases? If v is a leaf then the max-weight independent set just contains v for sure. To write the full recurrence, we just take the best of the two choices, choose v or don't choose v.

Algorithm: Dynamic programming recurrence for max-weight independent set on a tree

$$W(v) = \max \left\{ \sum_{u \in C(v)} W(u), \quad w_v + \sum_{u \in GC(v)} W(u) \right\}$$

Wait, stop! Where's the base case? We must have forgotten it. Or did we? Suppose v is a leaf. Then both of the sums in the recurrence are empty because $C(v)$ and $GC(v)$ will both be empty. Therefore $W(v) = w_v$ for a leaf from the second case. This means that this particular recurrence doesn't need an explicit base case, because it sort of comes built in to the sum over the children.

Step 4: Analysis This is our first example of a DP where the runtime needs a more sophisticated analysis than just multiplying the number of subproblems by the work per subproblem. If we were to do that naive analysis, we would get $O(n^2)$ since there are $O(n)$ subproblems and

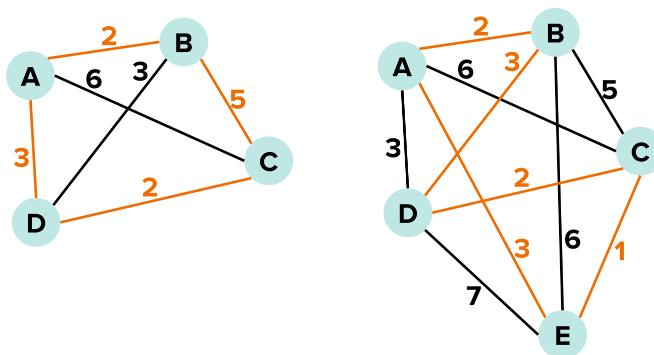
each might have to loop over up to $O(n)$ children/grandchildren. However, we can do better. Note that each vertex is only the child or grandchild of exactly one other vertex (its parent or its grandparent respectively). Therefore, each subproblem is only ever referred to by at most two other vertices. So we can do the analysis "in reverse" or "upside down" in some sense, and argue that each subproblem is only used at most twice, and hence the total work done is just two times the number of subproblems, or $O(n)$.

9.4 Traveling Salesperson Problem (TSP)

The NP-hard *Traveling Salesperson Problem* (TSP) asks to find the shortest route that visits *all* vertices in a graph exactly once and returns to the start.[2] We assume that the graph is complete (there is a directed edge between every pair of vertices in both directions) and that the weight of the edge (u, v) is denoted by $w(u, v)$. This is convenient since it means a solution is really just a *permutation of the vertices*.

Since the problem is NP-hard, we don't expect to get a polynomial-time algorithm, but perhaps dynamic programming can still help get something better than brute force. 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're going to use Dynamic Programming to reduce this to $O(n^2 2^n)$. This is still exponential time, but it's not as brutish as the naive algorithm. As usual, let's first just worry about computing the *cost* of the optimal solution, and then we'll later be able to recover the path.

Step 1: Find some optimal substructure This one harder than the previous examples, so we might have to try a couple of times to get it right. Suppose we want to make a tour of some subset of nodes S. Can we relate the cost of an optimal tour to a smaller version of the problem? Its not clear that we can. In particular, suppose we call out a particular vertex t, and then ask whether it is possible to relate the cost of the optimal tour of $S-\{t\}$ and S. It doesn't seem so, because its not clear how we would splice t into the tour formed by $S-\{t\}$. In fact, we can even show an example which demonstrates that the optimal tour of $S-\{t\}$ may not actually have all that much in common with the optimal tour of S:

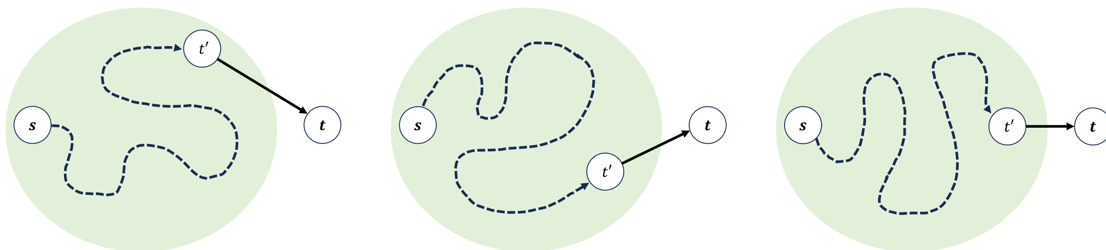


[2]Note that under this definition, it doesn't matter which vertex we select as the start. The *Traveling Salesperson Path Problem* is the same thing but does not require returning to the start. Both problems are NP-hard.

On the left is an optimal tour of a graph with four vertices. After adding a fifth vertex, we can see that the new optimal tour does not actually contain the old one as a subset, so there does not appear to be any *optimal substructure* here and we can not use these as our subproblems!

When faced with such an issue, it can quite often be resolved by adding more information to the subproblems. Adding a vertex into a cycle seems difficult to do because we don't know where to put it and how it might affect everything after it, but adding a vertex *onto the end of a path* sounds simpler, so lets try that. We will fix an arbitrary starting vertex x, and now consider the cheapest path that starts at x, visits all of the vertices in S and **ends at a specific vertex t**. The last part is essentially the "additional information" that we added to make the subproblems more specific. If we just considered any paths consisting of the vertices in S but without fixing the final vertex, we would run into the same problem as before.

Can we find any substructure in this much more specific object? The path now has a final vertex, which means it also has a second-last vertex! Specifically, the optimal path from x to t must have some second-last vertex t', and the path from x to t' must be an optimal such path using the vertices $S - \{t\}$ over all possible choices of t'. Lets use this for our subproblems.



Step 2: Define our subproblems Based on the above, lets make our subproblems

$C(S, t) =$ The minimum-cost path starting at x and ending at t going through all vertices in S

What is the solution to our original problem? Is it one of the subproblems? Actually the answer is no this time. But we can figure it out by combining a handful of the subproblems. Specifically, we want to form a tour (a cycle) using a path that starts at x. So we can just try every other vertex in the graph t, and make a path from x to t then back to x again to complete the cycle. So the answer, once we solve the DP will be

$$\text{answer} = \min_{t \in (V - \{x\})} (C(V, t) + w(t, x))$$

Step 3: Deriving a recurrence Using the substructure we described above, the idea that will power the recurrence is that to get a path that goes from x to t, we want a path that goes from x to t' plus an edge from t' to t. Which t' will be the best? We can't know for sure, so we should just try all of them and take the best one. We also need a base case. We can't use an empty path as our base case since we assume a starting vertex x and an ending vertex t for every subproblem, so lets use a path of two vertices as our base case. This gives us the recurrence

Algorithm: Dynamic programming recurrence for TSP

$$C(S,t) = \begin{cases} w(x,t) & \text{if } S = \{x,t\}, \\ \min\limits_{\substack{t' \in S \\ t' \notin \{x,t\}}} C(S - \{t\}, t') + w(t',t) & \text{otherwise.} \end{cases}$$

Step 4: Analysis The parameter space for $C(S,t)$ is 2^{n-1} (the number of subsets S considered) times n (the number of choices for t). For each recursive call we do $O(n)$ work inside the call to try all previous vertices t', for a total of $O(n^2 2^n)$ time. This is assuming we can lookup a set (e.g, $S - \{t\}$) in constant time.

Remark: Efficiently representing the set

Since this algorithm is too slow for large values of n, we usually assume that n is small, and so a highly efficient way to store the set S is as a single integer, where you use the individual bits of the integer to indicate which vertices are in or not in the set. This makes it easy to store and lookup the subproblem solutions because an integer is a much better key for an array or hashtable than a set!

This technique is sometimes called "Subset DP". These ideas apply in many cases to reduce a factorial running to time to a regular exponential running time.

Exercises: Dynamic Programming

Problem 19. We showed how to find the weight of the max-weight independent set. Show how to find the actual independent set as well, in $O(n)$ time.

Problem 20. Give an example where using the greedy strategy for the 0-1 knapsack problem will get you less than 1% of the optimal value.

Problem 21. 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.

Problem 22. Given the results of $C(S, t)$ for a TSP problem, explain how to find the actual sequence of vertices that make up the tour.

Lecture 10

Dynamic Programming II

In the previous lecture, we reviewed Dynamic Programming and saw how it can be applied to problems about sequences and trees. Today, we will extend our understanding of DP by applying it to other classes of problems, like shortest paths, and explore how to speed up dynamic programming implementations with clever tricks like fancier data structures!

Objectives of this lecture

In this lecture, we will:

- Learn about optimizing DPs by **eliminating redundancies** via the all-pairs shortest path problem
- Learn about optimizing DPs **with data structures** via the longest increasing subsequence problem
- Learn about optimizing tree DPs **by adding child information one at a time**

10.1 All-pairs Shortest Paths (APSP)

Say we want to compute the length of the shortest path between *every* pair of vertices in a weighted graph. This is called the **all-pairs** shortest path problem (APSP). If we use the Bellman-Ford algorithm (recall 15-210), which takes $O(nm)$ time, for all n possible destinations t, this would take time $O(mn^2)$. We will now see a Dynamic-Programming algorithm that runs in time $O(n^3)$, but let's warm up with a simpler but worse algorithm.

10.1.1 A first attempt in $O(n^4)$

We dealt with the Traveling Salesperson Problem (TSP) just last lecture, which we also solved in terms of substructure over paths. It therefore seems natural to try the same thing for APSP. In our DP solution for TSP, we created paths by extending them by one edge at a time. We also had to remember the current subset of vertices since the goal was to visit every vertex once and only once. In this problem, we no longer have that restriction, so it won't be necessary to store all of the subsets.

Instead, to build a path out of k vertices, all we need is a path of $k-1$ vertices! We don't actually care *which* vertices they are, so we can just parameterize the subproblems by an integer (the number of vertices in the path) rather than a subset of vertices.

Defining the subproblems Based on the above observation that we can build a path from a shorter path plus a new edge, we can define our subproblems as

$$D[u][v][k] = \text{the length of the shortest path from } u \text{ to } v \text{ using } k \text{ vertices}$$

Deriving a recurrence To build a path from u to v with k vertices, we just need to build a path from u to some other vertex, then add v. We can try every possible intermediate vertex to ensure that we definitely get the right one, which gives us a recurrence that looks like

$$D[u][v][k] = \min_{v' \in V} \big(D[u][v'][k-1] + w(v', v)\big)$$

For simplicity, assume that if $(v', v) \notin E$, then $w(v', v) = \infty$. Our base case will be when there is just a single vertex v in the path, in which case the distance from v to itself is zero.

$$D[v][v][1] = 0.$$

Otherwise for $u \neq v$, we want $D[u][v][1] = \infty$ (it is impossible to have a path with one vertex that goes from u to v when $u \neq v$). After evaluating D for all u, v, k where $1 \leq k \leq n$, the length of the shortest path from u to v is given by the minimum of $D[u][v][k]$ for all $1 \leq k \leq n$.

Analysis We have $O(n^3)$ subproblems and each of them takes $O(n)$ time to evaluate, so this takes $O(n^4)$ time. Actually, we can be a little bit more clever in a similar way to which we analyze tree DP algorithms. Note that we only have to check $v' \in V$ for v' that have an outgoing edge pointing to v, i.e., such that $(v', v) \in E$, so the total amount of work spent evaluating the minimum over all v for any fixed value of u and k is $O(m)$, so the total work is at most $O(n^2 m)$, which matches the strategy of repeatedly running Bellman-Ford.

If we think about it carefully, perhaps this isn't so surprising since Bellman-Ford also builds paths by repeatedly adding one edge at a time starting from a single source node, so this algorithm is kind of doing the same thing as running Bellman-Ford simultaneously from every possible start vertex! (We just reinvented Bellman-Ford, oops).

10.1.2 An improved version in $O(n^3)$

The nice thing about paths is that there are lots of ways of breaking them up into pieces. The previous strategy was to just add a single edge at a time, but that seems inefficient because every time we want to add an edge, we have to look at every possible second-last vertex.

Another way to break paths up is to chop them into two paths! A path from u to v can be broken at any point along the path k into the two paths u to k and k to v. How exactly does this let us decompose the problem into *smaller problems* though? We can not just define our subproblems as the shortest path between u and v and then solve them by trying all intermediate vertices k, because this would assume that we already had all the shortest paths between all u and all possible k to begin with, i.e., the problems aren't guaranteed to be smaller problems.

To make the problems smaller, we need one crucial observation: In a valid shortest path, there is no reason to use the same vertex twice! So, when we decide to break a shortest path u to v into two paths U to k and k to v, we don't need k to occur inside either of those paths! Let's therefore try adding new intermediate vertices one at a time.

Define our subproblems The idea is that we want to build paths out of increasingly larger sets of intermediate vertices. When vertex k is introduced, we can stitch together a path from u to k and k to v to build a path from u to v. Those smaller paths do not need to contain k since there is no point in using k more than once. So, we will use the subproblems

$$D[u][v][k] = \text{length of the shortest path from } u \text{ to } v \text{ using intermediate vertices } \{1,2,\dots,k\}$$

Deriving a recurrence We need to consider two cases. For the pair u, v, either the shortest path using the intermediate vertices $\{1,2,\dots,k\}$ goes through k or it does not. If it does not, then the answer is the same as it was before k became an option. If k now gets used, we can *break the path at* k and use the optimal substructure to glue together the two shortest paths divided at k to get a new shortest path. Writing the recurrence using this idea looks like this.

$$D[u][v][k] = \min\{D[u][v][k-1], D[u][k][k-1] + D[k][v][k-1]\}.$$

Our base case will just be

$$D[u][v][0] = \begin{cases} 0 & \text{if } u = v, \\ w(u,v) & \text{if } (u,v) \in E, \\ \infty & \text{otherwise.} \end{cases}$$

Analysis We have $O(n^3)$ subproblems and each of them takes $O(1)$ time to evaluate, so this takes $O(n^3)$ time! Notice that the key improvement here over the previous algorithm was that for each subproblem, we only needed to make a single binary decision: use k or don't use k, which is much more efficient than our earlier algorithm which had to try *every vertex*.

10.1.3 Optimizing the space: eliminating redundancies

One downside of the algorithm is that it uses a lot of space, $O(n^3)$, which is a factor n larger than the input graph. This is bad if the graph is large. Can we reduce this? There are two ways. First, notice that the subproblems for parameter k only depend on the subproblems with parameter $k-1$. So, we don't actually need to store all $O(n^3)$ subproblems, we can just keep the last set of subproblems and compute bottom-up in increasing order of k.

Here's an even simpler but more subtle way to optimize the algorithm. We can just write:

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

So what happened here, it looks like we just forgot the k parameter of the DP, right? It turns out that this algorithm is still correct, but now it only uses $O(n^2)$ space because it just keeps a single 2D array of distance estimates. Why does this work? Well, compared to the by-the-book implementation, all this does is allow the possibility that $D[u][k]$ or $D[k][v]$ accounts for vertex k already, but a shortest path won't use vertex k twice, so this doesn't affect the answer! This algorithm is known as the *Floyd-Warshall* algorithm.

Key Idea: Optimize DP by eliminating redundant subproblems

Sometimes our subproblems might not all be necessary to the solve the problem, so if we can eliminate many of them, we will either speed up the algorithm or reduce the amount of space it requires.

10.2 Advanced Tree DP

In the previous lecture we saw the bread and butter of tree DP, which involves solving the problem for every possible rooted subtree of a tree and then combining the answers to the child problems to form the answer for the root problem. Sometimes this is not quite enough and is not efficient, so let's see a trick to make it even better. The problem arises when we want to solve problems on trees with high degree, but the problem involves making decisions about how much of a quantity to split between the children. Here's an example problem:

Problem: Counting subtrees

Given a rooted tree with n nodes and root r, count the number of distinct subtrees rooted at r with exactly K nodes.

To get the main idea we will first solve the problem with the additional assumption that the tree is a *binary tree*, then we will see how to remove this assumption.

10.2.1 Solution for binary trees

Under the assumption that the tree is binary, counting the number of possible subtrees rooted at a particular vertex v with exactly k nodes just consists in deciding how to divide the $k-1$ nodes other than v between the two children.

Defining the subproblems For simplicity we will assume that the tree is a full binary tree, i.e., every node is either a leaf or has two children. We will use the subproblems:

$$S(v,k) = \text{the number of subtrees rooted at } v \text{ containing } k \text{ nodes}$$

The solution to the problem is $S(r,K)$.

Deriving a recurrence To solve the subproblems, we just sum over all of the possible ways to divide the $k-1$ remaining nodes between the two children:

$$S(v,k) = \begin{cases} 1 & \text{if } k=0 \text{ or } k=1, \\ 0 & \text{if } k>1 \text{ and if } v \text{ is a leaf} \\ \displaystyle\sum_{k'=0}^{k-1} S(L(v),k') \cdot S(R(v),(k-1)-k') & \text{otherwise} \end{cases}$$

The first base cases are when $k \in \{0,1\}$ since we can make exactly one subtree out of one node (just itself) and one subtree using zero nodes (the empty subtree). Otherwise if $k > 1$ and v is a leaf then we can not possible make a subtree with k nodes since all we have is a single leaf!

Analysis We have $O(nK)$ subproblems and each takes $O(K)$ time, so the cost is $O(nK^2)$.

10.2.2 Generalization to high-degree trees

How generalizable is this solution to high-degree trees? Well, it works, but its going to be slow. For a node with two children, there were $O(K)$ ways to divide the subtree nodes between them. For a node with three children, there would be $O(K^2)$ ways to do it, for four children, $O(K^3)$ and so on... This is exponential in K, so trying all possibilities is off the table.

However, there is a lot of redundancy in these possibilities. If I decide to put k' out of k nodes on the first child, then there are $k-1-k'$ remaining nodes to distribute between the remaining children. The fact that I put k' nodes on the first child does not affect the others at all. So we can eliminate the redundancy by further parameterizing the subproblems by which children we are considering, and just adding in one child at a time!

Defining the subproblems For simplicity we will assume that the tree is a full binary tree, i.e., every node is either a leaf or has two children. We will use the subproblems:

$$S(v,k,i) = \begin{cases} \text{the number of subtrees rooted at } v \text{ containing } k \\ \text{nodes using only the } i^{\text{th}} \text{ child and after of } v \end{cases}$$

The solution to the problem is $S(r,K,0)$.

Deriving a recurrence To solve the subproblems, we sum over all of the possible ways to divide the k nodes between the current child and the root with the rest of the children:

$$S(v,k,i) = \begin{cases} 1 & \text{if } k=0 \text{ or } k=1, \\ 0 & \text{if } k>1 \text{ and if } i=\deg(v) \\ \displaystyle\sum_{k'=0}^{k-1} S(C(v,i),k',0)\cdot S(v,k-k',i+1) & \text{otherwise} \end{cases}$$

For notation, we have used $C(v,i)$ to mean the ith child of v, and $\deg(v)$ to be the degree (number of children) of v. The base cases are similar to the binary case, we can make only one tree with a single node or no nodes. Note that $S(v,k,\deg(v))$ refers to a subproblem in which we have ran out of children of v to consider, i.e., we are considering v alone. If v is a leaf then $\deg(v)=0$ and all the subproblems for v will fall into this or the first case.

Note the subtle difference between this and the binary version where the recursion divides the k nodes into k' and $k-k'$ instead of $(k-1)-k'$. This is because in the binary case, we used that one on v and then split the remaining $k-1$ nodes between the children, but in this recurrence, the second subproblem $S(v,\ldots)$ *still includes the root*, so we do not lose one node anymore.

Analysis How many subproblems do we have now? There are n choices for v, and n choices for i, and K choices for K, so our new cost should be $O(n^2K^2)$, right? No! This is tree DP of course, and there aren't very many possible values of i for every single v! In particular, note that each (v, i) **pair** maps one-to-one with a specific child vertex, so there at most n of them in total. This means that the number of subproblems is actually still at most nK and hence the cost is still $O(nK^2)$, so we just got to generalize to non-binary trees *for free*! It does not actually cost any more than the binary case.

10.3 Longest Increasing Subsequence

Our next problem is the "longest increasing subsequence" (LIS) problem, which has an $O(n^2)$ solution, but can then be improved with some clever optimizations!

Problem: Longest Increasing Subsequence

Given a sequence of comparable elements $a_1, a_2, \ldots, a_n$, an increasing subsequence is a subsequence $a_{i_1}, a_{i_2}, ..., a_{i_{k-1}}, a_{i_k}$ $(i_1 < i_2 < \ldots i_k)$ such that

$$a_{i_1} < a_{i_2} < \ldots < a_{i_{k-1}} < a_{i_k}.$$

A longest increasing subsequence is an increasing subsequence such that no other increasing subsequence is longer.

Find some optimal substructure Given a sequence $a_1, \ldots, a_n$ and its LIS $a_{i_1}, \ldots, a_{i_{k-1}}, a_{i_k}$, what can we say about $a_{i_1}, \ldots, a_{i_k-1}$? Since $a_{i_1}, \ldots, a_{i_k}$ is an LIS, it must be the case that $a_{i_1}, \ldots, a_{i_{k-1}}$ is an LIS of $a_1, \ldots a_{i_k}$ such that $a_{i_{k-1}} < a_{i_k}$. Alternatively, it is also an LIS that ends at (and contains) $a_{i_{k-1}}$. This suggests a set of subproblems.

Define our subproblems Lets define our subproblems to be

$$\text{LIS}[i] = \text{the length of a longest increasing subsequence of } a_1, \ldots a_i \text{ that contains } a_i$$

Note that the answer to the original problem **is not** necessarily LIS$[n]$ since the answer might not contain a_n, so the actual answer is

$$\text{answer} = \max_{1 \le i \le n} \text{LIS}[i]$$

Deriving a recurrence Since LIS$[i]$ ends a subsequence with element i, the previous element must be anything a_j before i such that $a_j < a_i$, so we can try all possibilities and take the best one

$$\text{LIS}[i] = \begin{cases} 0 & \text{if } i = 0, \\ 1 + \max\limits_{\substack{0 \le j < i \\ a_j < a_i}} LIS[j] & \text{otherwise.} \end{cases}$$

Analysis We have $O(n)$ subproblems and each one takes $O(n)$ time to evaluate, so we can evaluate this DP in $O(n^2)$ time. Is this a good solution or can we do better?

10.3.1 Optimizing the runtime: better data structures

The by-the-definition implementation of the recurrence for LIS gives an $O(n^2)$ algorithm, but sometimes we can speed up DP algorithms by solving the recurrence more cleverly. Specifically in this case, the recurrence is computing a **minimum over a range**, which sounds like something we know how to do faster than $O(n)$...

How about we try to apply a range query data structure (a SegTree) to this problem! Initially, its not clear why this would work, because although we are doing a range query over $1 \le j < i$, we have to account for the the constraint that $a_j < a_i$, so we can not simply do a range query over the values of $\text{LIS}[1\ldots(i-1)]$ or this might include larger elements.

So here's an idea... Let's solve the subproblems *in order* by the value of the final element, instead of just left-to-right by order of i. That way, when we solve a particular subproblem corresponding to a particular final element, we will have only processed the subproblems corresponding to all smaller elements, which are all legal to append the next larger element to!

```
function LIS(a : list⟨int⟩) -> int = {
  sortedByVal := sorted list of (value, index) pairs
  // SegTree is endowed with the RangeMax operation
  LIS := SegTree(array⟨int⟩(size(a), 0))
  for val, index in sortedByVal do {
    answer := LIS.RangeMax(0, index) + 1   // Solve the subproblem
    LIS.Assign(index, answer)              // Store the subproblem
  }
  return LIS.RangeMax(0, size(a))
}
```

This optimized algorithm performs two SegTree operations per iteration, and it has to sort the input, so in total it takes $O(n \log n)$ time.

Key Idea: Speed up DP with data structures

If your DP recurrence involves computing a minimum, or a sum, or searching for something specific, you can sometimes speed it up by storing the results in a data structure other than a plain array (e.g., a SegTree or BST).

Exercises: Dynamic Programming II

Problem 23. Provide a formal proof by induction that the Floyd-Warshall algorithm gives a correct answer. The comment in the pseudocode is the inductive hypothesis on which you should use induction on k.

Problem 24. Can you find a greedy algorithm that matches the $O(n \log n)$ performance of the LIS algorithm above?

Problem 25. Write a different implementation of the LIS algorithm that still uses a SegTree but loops over i and uses range queries on j instead (the opposite of the solution above).

Lecture 11

Network Flows I

In these next three 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 model a wide variety of different kinds of problems. So, by developing good algorithms for solving flows, we get algorithms for solving many other problems as well. In Operations Research there are entire courses devoted to network flow and its variants.

Objectives of this lecture

In this lecture, we will:

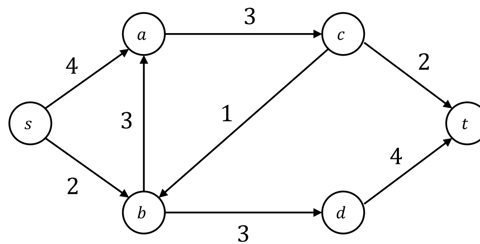
- Define the maximum network flow problem
- Derive and analyze the Ford-Fulkerson algorithm for maximum network flow
- Prove the famous *max-flow min-cut theorem*
- See how to solve the bipartite matching problem by reducing it to a maximum flow problem

Recommended study resources

- CLRS, *Introduction to Algorithms*, Chapter 26, Maximum Flow
- DPV, *Algorithms*, Chapter 7.2, Flows in Networks

11.1 The Maximum Network Flow Problem

We begin with a definition of the problem. We are given a directed graph G, a source 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 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. For example, consider the graph below.



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,

Definition: Flow constraints

Capacity constraint: On any edge e we have $0 \leq f(e) \leq c(e)$.

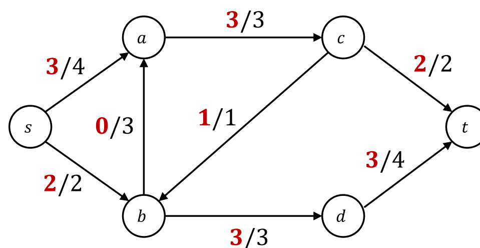
Flow conservation: For any vertex $v \notin \{s, t\}$, flow in equals flow out:

$$\sum_u f(u, v) = \sum_u f(v, u).$$

A flow that satisfies these constraints is called a *feasible flow*. Subject to these constraints, we want to maximize the net flow from s to t. We can measure this formally by the quantity

$$|f| = \sum_{u \in V} f(s, u) - \sum_{u \in V} f(u, s)$$

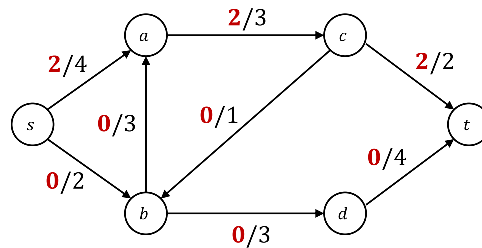
Note that what we are measuring here is the net flow coming out of s. It can be proved, due to conservation, that this is equal to the net flow coming into t. So, in the above graph, what is the maximum flow from s to t? Answer: 5. Using the notation"**flow**/capacity", a maximum flow looks like this. Note that the flow can split and rejoin itself.



11.1.1 Improving a flow: s-t paths

Suppose we start with the simplest possible flow, the all-zero flow. This flow is feasible since it meets the capacity constraints, and all vertices have flow in equal to flow out (zero!). But for the network above, the all-zero flow is clearly not optimal. Can we formally reason about *why* it isn't optimal? One way is to observe that there exists a path from s to t in the graph consisting of edges with available capacity (actually there are many such paths). For example, the path $s \to a \to c \to t$ has at least 2 capacity available, so we could add 2 flow to each of those edges

without violating their capacity constraints, and improve the value of the flow. This gives us the flow below



An important observation we have just made is that if we add a constant amount of flow to all of the edges on a path, we never violate the flow conservation condition since we add the same amount of flow in and flow out to each vertex, except for s and t. As long as we do not violate the capacity of any edge, the resulting flow is therefore still a feasible flow.

This observation leads us to the following idea: A flow is not optimal if there exists an s-t path with available capacity. In the above graph, we can observe that the path $s \to b \to d \to t$ has 2 more units of available capacity, and hence add 2 units of flow to the edges. Lastly, we could find the $s \to a \to c \to b \to d \to t$ path has 1 more unit of available capacity, and we would arrive at the maximum flow from before.

11.1.2 Certifying optimality of a flow: s-t cuts

We just saw how to convince ourselves that a flow was **not** maximum, but how can you see that the above flow was indeed maximum? We can't be certain yet that we didn't make a bad decision and send flow down a sub-optimal path. Here's an idea. Notice, this flow saturates the $a \to c$ and $s \to b$ edges, and, if you consider the partition of vertices into two parts, $S = \{s, a\}$ and $T = \{b, c, d, t\}$, all flow that goes from S to T must go through one of those two edges! We call this an "s-t cut" of capacity 5. 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 that no flow can have a value greater than 5. Let's formalize this idea of a "cut" and use it to formalize these observations.

Definition: s-t cut

An **s-t cut** is a partition of the vertex set into two pieces S and T where $s \in S$ and $t \in T$. (The edges of the cut are then all edges going from S to T).

The **capacity** of a cut (S, T) is the sum of the capacities of the edges crossing the cut, i.e., the sum of the capacities of the edges going from S to T:

$$\text{cap}(S, T) = \sum_{u \in S} \sum_{v \in T} c(u, v)$$

Note importantly that the definition of the capacity of a cut **does not** include any edges that go from T to S, for example, the edge (c, b) above would not be included in the capacity of the cut $(\{s, a, b\}, \{c, d, t\})$.

Definition: Net flow

The **net flow** across a cut (S, T) is the sum of flows going from S to T minus any flows coming back from T to S, or

$$f(S,T) = \sum_{u \in S} \sum_{v \in T} f(u,v) - \sum_{u \in S} \sum_{v \in T} f(v,u)$$

From this definition, we can see that the value of a flow $|f|$ is equivalent to the net flow across the cut $(\{s\}, V \setminus \{s\})$. In fact, using the flow conservation property, we can prove that the net flow across *any* cut is equal to the flow value $|f|$. This should make intuitive sense, because no matter where we cut the graph, there is still the same amount of flow making it from s to t.

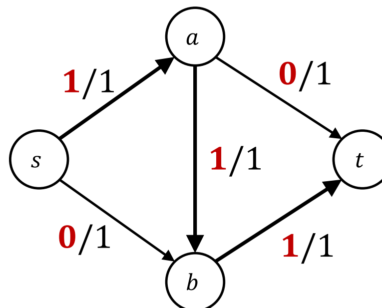
Now to formalize our observation from before, what we noticed is that the value of any s-t flow is less than the value of any s-t cut. Since every flow has a value that is at most the maximum flow, we can argue that in general, the maximum s-t flow ≤ the capacity of the minimum s-t cut, or

the value of *any* s-t flow ≤ maximum s-t flow ≤ minimum s-t cut ≤ capacity of *any* s-t cut

This means that if we can find an s-t flow whose value is equal to the value of *any cut*, then we have proof that is must be a maximum flow!

11.1.3 Finding a maximum flow

How can we find a maximum flow and prove it is correct? We should try to tie together the two ideas from above. 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. Is this the case for a naive algorithm that simply searches for s-t paths with available capacity and adds flow to them? Unfortunately it is not. Consider the following graph, where the algorithm has decided to add 1 unit of flow to the path $s \to a \to b \to t$



Are there any s-t paths with available capacity left? No, there are not. But is this flow a maximum flow? Also no, because we could have sent 1 unit of flow from $s \to a \to t$ and 1 unit of flow from $s \to b \to t$ for a value of 2. The problem with this attempted solution was that

sending flow from a to b was a "mistake" that lowered the amount of available capacity left. To arrive at an optimal algorithm for maximum flow, we therefore need a way of "undoing" such mistakes. We achieve this by defining the notion of *residual capacity*, which accounts for both the remaining capacity on an edge, but also adds the ability to undo bad decisions and redirect a suboptimal flow to a more optimal one. This leads us to the Ford-Fulkerson algorithm.

11.2 The Ford-Fulkerson algorithm

The magic idea behind the Ford-Fulkerson algorithm is going to be "undoing" bad previous decisions by redirecting flow along a different path. To enable this, we define the concept of the *residual graph*. Residual capacity is just the capacity left over given the existing flow and also accounts for the ability to push existing flow back and along a different path, in order to undo previous bad decisions. Paths in the residual graph are called *augmenting paths*, because you use them to augment the existing flow.

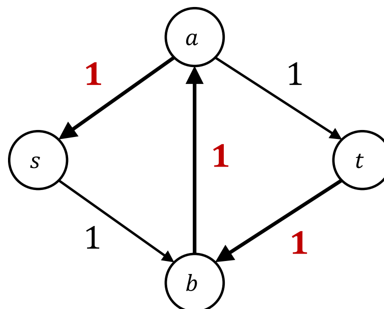
Definition: Residual capacity

Given a flow f in graph G, the **residual capacity** $c_f(u,v)$ is defined as

$$c_f(u,v) = \begin{cases} c(u,v) - f(u,v) & \text{if } (u,v) \in E, \\ f(v,u) & \text{if } (v,u) \in E. \end{cases}$$

Definition: Residual graph

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 reverse-edges of the original graph G.



When $(u,v) \in E$, the residual capacity on the edge corresponds to our existing intuitive notion of "remaining/leftover" capacity, it is the amount of additional flow that we could put through an edge before it is full. The second case is the key insight that makes the Ford-Fulkerson algorithm work. If $f(v,u)$ flow has been sent down some edge (v,u), then we are able to *undo* that by sending flow back in the reverse direction (u,v)! For example, if we consider the suboptimal flow from earlier, the residual graph looks like the following.

Unlike before, there is now an s-t path with capacity 1! The path $s \to b \to a \to t$ "undoes" the flow that was originally pushed through $a \to b$ and redirects it to $a \to t$, thus improving the flow and making it optimal.

The Ford-Fulkerson algorithm is then just the following procedure.

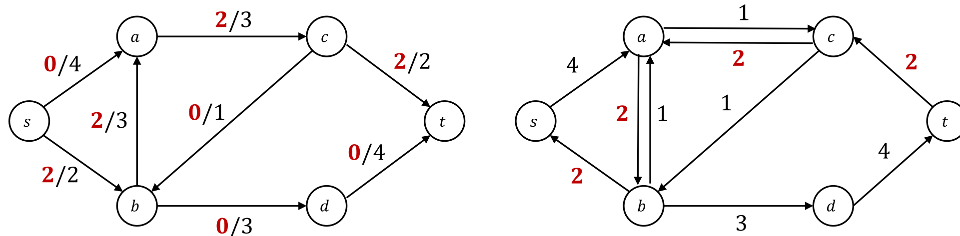
Algorithm: Ford-Fulkerson Algorithm for Maximum Flow

```
while (there exists an s→t path P of positive residual capacity)
push the maximum possible flow along P
```

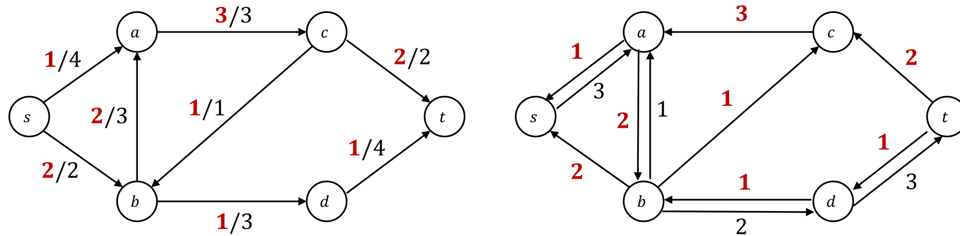
Remark: Handing anti-parallel edges

The definition of residual capacity becomes a little funny if we suppose that the graph contains anti-parallel edges. Recall that parallel edges are those with the same endpoints u and v that point in the same direction, while anti-parallel edges are those with the same endpoints but point in opposite directions. In this case, it might be the case that both $(u,v) \in E$ and $(v,u) \in E$, so how do we handle this? One option is to simply disallow it and not admit these kinds of graphs (this is what the textbook does!) Another valid answer is that we should use *both*. We have the choice of either combining both into a single edge with the sum of their capacities, or including a pair of parallel edges in the residual graph. Both of these options are fine and will work in theory and practice.

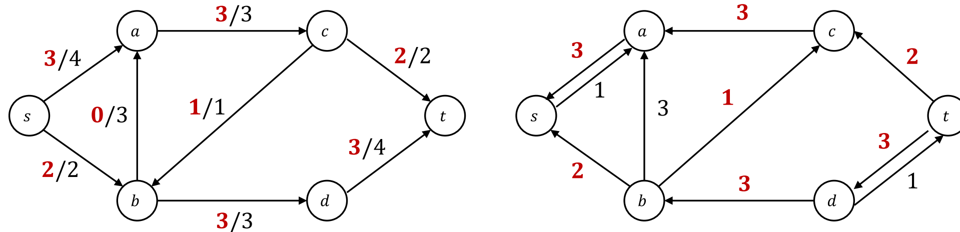
Let's see another example. Consider the graph we started with and suppose we push two units of flow on the path $s \to b \to a \to c \to t$. We then end up with the following flow (left) and residual graph (right).



If we continue running Ford-Fulkerson, we might find the *augmenting path* $s \to a \to c \to b \to d \to t$ which has capacity 1, bringing the flow value to 3, and yielding the following flow (left) and residual graph (right).



Finally, there is one more augmenting path, $s \to a \to b \to d \to t$ of capacity 2. This gives us the maximum flow that we saw earlier (left). At this point there is no longer a path from s to t in the residual graph (right) so we know 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. 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 or BFS).

11.2.1 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. If we assume the graph is simple, which is reasonable, $m \geq n-1$, so this is $O(m)$, so we get the following result.

Theorem: Runtime of the Ford-Fulkerson Algorithm

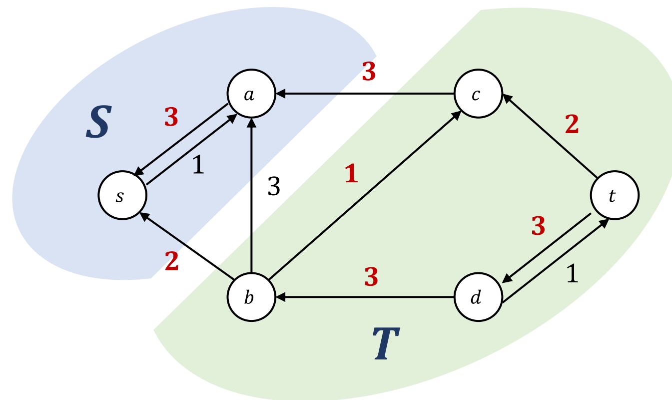
If the given graph G has integer capacities, Ford-Fulkerson terminates in time $O(mF)$ where F is the value of the maximum s-t flow.

Now the important result that proves the correctness of the algorithm and comes with a powerful corollary.

Theorem: Optimality of the Ford-Fulkerson Algorithm

When it terminates, the Ford-Fulkerson algorithm outputs a flow whose value is equal to the minimum cut of the graph.

Proof. Let's look at the final residual graph. Since Ford-Fulkerson loops until there is no longer a path from s to t, this graph must have s and t disconnected, otherwise the algorithm would keep looping. Let S be the set of vertices that are still reachable from s in the residual graph, and let T be the rest of the vertices.



It must be that $t \in T$ since otherwise s would be connected to t, so this is a valid s-t cut. Let $c = \operatorname{cap}(S, T)$, the capacity of the cut in the *original* graph. From our earlier discussion, we know that $|f| = f(S, T) \leq c$. The claim is that we in fact *did* find a flow of value c (which therefore implies it is maximum). Consider all of the edges of G (the original graph) that cross the cut in either direction. We consider both cases:

1. Consider edges in G that cross the cut in the $S \to T$ direction. We claim that all of these edges are at maximum capacity (the technical term is *saturated*). Suppose for the sake of contradiction that there was an edge crossing from S to T that was not saturated. Then, by definition of the residual capacity, the residual graph would contain an edge with positive residual capacity from S to T, but this contradicts the fact that T contains the vertices that we can not reach in the residual graph, since we would be able to use this edge to reach a vertex in T. Therefore, we can conclude that every edge crossing the cut from S to T is saturated.

2. Consider edges in G that go from T to S. We claim that all such edges are *empty* (their flow is zero). Again, suppose for the sake of contradiction that this was not true and there is a non-zero flow on an edge going from T to S. Then by the definition of the residual capacity, there would be a reverse edge with positive residual capacity going from S to T. Once again, this is a contradiction because this would imply that there is a way to reach a vertex in T in the residual graph. Therefore, every edge crossing from T to S is empty.

Therefore, we have for every edge crossing the cut from S to T that $f(u, v) = c(u, v)$, and for every edge crossing from T to S that $f(u, v) = 0$, so the *net flow* across the cut is equal to the capacity of the cut c. Since every flow has a value that is at most the capacity of the minimum cut, this cut must in fact be the minimum cut, and the value of the flow is equal to it. $\square$

As a corollary, this also proves the famous maximum-flow minimum-cut theorem.

Theorem: Max-flow min-cut theorem

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

Proof. For any integer-capacity network, Ford-Fulkerson finds a flow whose value is equal to the minimum cut. Since the value of any flow is at most the capacity of any cut, this must be a maximum flow. □

We can also deduce *integral-flow theorem.* This turns out to have some nice and non-obvious implications.

Theorem: Integral flow theorem

Given a flow network where all capacities are integer valued, there exists a maximum flow in which the flow on every edge is an integer.

Proof. Given a network with integer capacities, Ford-Fulkerson will only find integer-capacity augmenting paths and will never create a non-integer residual capacity. Therefore it finds an integer-valued flow, and this is maximum flow, so there always exists an integer-valued maximum flow. □

It is not necessarily the case that every maximum flow is integer valued, only that there exists one that is. Lastly, the proof also sneakily teaches us how to actually find a minimum cut, too!

Remark: Min-cut max-flow for non-integer capacities

Technically, we only proved the min-cut max-flow theorem 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). We 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 min-cut max-flow for rational capacities as well.

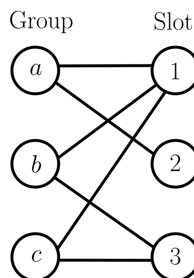
What if the capacities are irrational? In this case Ford-Fulkerson may not terminate. And the solution it converges to (in the limit) may not even be the max-flow! But the maxflow-mincut theorem still holds, even with irrational capacities. There are several ways to prove this; here's one. Suppose not, 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.

Alternatively, in the next lecture we'll see that if we modify Ford-Fulkerson to always choose an augmenting path with the fewest edges on it, it's guaranteed to terminate. This proves the max-flow min-cut theorem for arbitrary capacities.

In the next lecture we will look at methods for reducing the number of iterations. 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.

11.3 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:

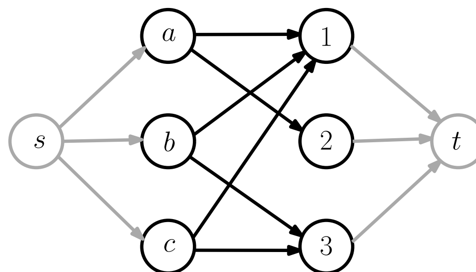


This is an example of a **bipartite graph**: a graph with two sides L and R such that all edges go between L and R. A **matching** is a set of edges with no endpoints in common. What we want here in assigning groups to time slots is a **perfect matching**: a matching that connects every point in L with a point in R. More generally (say there is no perfect matching) we want a **maximum matching**: a matching with the maximum possible number of edges. We claim that we can solve this as follows, with a **reduction** to maximum flow!

Algorithm: Bipartite Matching

1. Set up a source node s connected to all vertices in L. Connect all vertices in R to a 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.

Let's run the algorithm on the above example. We build this flow network as described and then run the Ford-Fulkerson algorithm from earlier to find the maximum flow:



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! Amazing. Matchings come up in many different applications, and are also a building block of other algorithmic problems.

Proof of correctness To prove that a reduction like this is correct, we usually approach it in two parts. We need to show **both** that matchings correspond to feasible flows in our network and also that feasible flows correspond to matchings, and that their size/values are equal.

$\exists$ matching M in original graph $\implies$ $\exists$ integral feasible flow of value $|M|$ in our flow network:

Let M be a matching in the original graph, and create a flow f as follows. For each edge (u,v) in the matching assign a flow value of 1 to $f_{u,v}$, as well as a flow value of 1 to each of $f_{s,u}$ and $f_{v,t}$. We claim that this flow is feasible. The capacity constraints are satisfied because we assign a flow of at most 1 per edge. What about conservation? For each original edge (u,v), if (u,v) is matched, then we have $f_{s,u} = f_{u,v} = f_{v,t} = 1$. Since M is a matching, there are no other edges adjacent to u or v that have nonzero flow, and hence the conservation condition is satisfied at u and v. For any vertex that $u \in L$ that is not matched, we $f_{s,u} = f_{u,v} = 0$ for all $v \in R$, and symmetrically the same holds for any $v \in R$ that is not matched. So, conservation is satisfied on these vertices as well. Lastly, note that we create exactly one s-t path with flow value 1 for each matched edge, so the value of the flow $|f|$ exactly equals the size of the matching $|M|$.

$\exists$ integral feasible flow f in our flow network $\implies$ $\exists$ matching of size $|f|$ in the original graph:

Now consider any integral feasible flow, and construct a matching by taking all edges (u,v) such that $f_{u,v} = 1$ where $u \neq s$ and $v \neq t$ (i.e., the matching consists of all of the middle edges with non-zero flow). We claim this is a valid matching. Why? Because of the capacity constraint, each matched vertex has at most 1 flow coming in. Then because of the conservation constraint, it has at most 1 flow going out, so no vertex is matched multiple times. Also note that the value $|f|$ of the flow is exactly equal to the number of edges we put in the matching (one way to see this is to construct a cut $(\{s\} \cup L, R \cup \{t\})$ and note that the net flow across this cut is exactly the values of the middle edges), and therefore $|M| = |f|$.

Since our network as constructed only has integer capacities, by the integrality theorem, there always exists an integral maximum flow. Therefore, together, these two pieces show that the value of the maximum flow is at most the size of the maximum matching, and that the size of the maximum matching is at most the value of the maximum flow, and hence the two must be equal, so our reduction is correct!

Runtime What about the number of iterations of Ford-Fulkerson? This is at most the number of edges in the matching since each augmenting path gives us one new edge. There can be at most n edges in the matching, so we have $F \le n$, and hence the runtime is $O(nm)$.

Exercises: Flow Fundamentals

Problem 26. Prove that the net flow across any (S,T) cut is equal to the value of the flow $|f|$. You should use the flow conservation constraint to do so. This also proves our earlier statement that the flow value $|f|$ is equal to the net flow into t.

Problem 27. Give an example of a flow network with all integer capacities and a maximum flow on that network which has a non-integer value on at least one edge.

Problem 28. Finding a minimum cut Explain how to construct a minimum s-t cut given a maximum s-t flow of a network. Hint: The answer is hidden in the proof of the min-cut max-flow theorem.

Lecture 12
Network Flows II

The Ford-Fulkerson algorithm discussed in the last class takes time $O(mF)$, 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 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 regardless of the value of the max flow or capacities.

Objectives of this lecture

In this lecture, we will:

- See an algorithm for max flow with polynomial running time (Edmonds-Karp)
- Analyze an even better algorithm for max flow that runs in polynomial time (Dinic's)

Recommended study resources

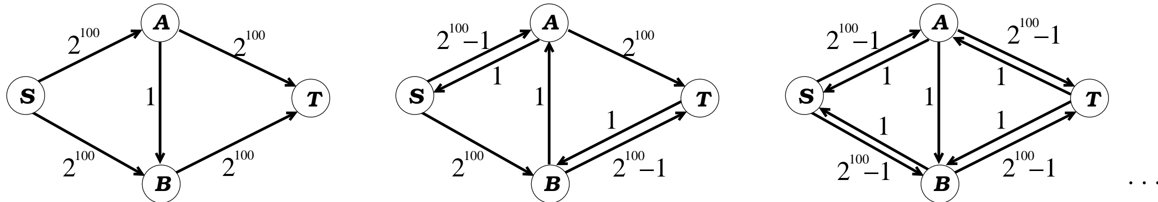
- CLRS, *Introduction to Algorithms*, Chapter 26, Maximum Flow
- DPV, *Algorithms*, Chapter 7.2, Flows in Networks

12.1 Network flow recap

Recall that in the maximum flow proble, 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 min-cut max-flow theorem, as well as the integrality theorem for flows. 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", which accounts for remaining capacity as well as the ability to redirect existing flow (and hence "undo" bad previous decisions) and repeat the process, continuing until there are no more paths of positive residual capacity left between s and t. 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. (We assume that every vertex in the graph

is reachable from s, so $m \geq n-1$.) 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 large and F could be very large. Here's an example that could make the algorithm take a very long time. If the algorithm selects the augmenting paths $s \to A \to B \to t$, then $s \to B \to A \to t$, repeating..., then each iteration only adds one unit of flow, but the max flow is 2^{101}, so the algorithm will take 2^{101} iterations. If the algorithm selected the augmenting paths $s \to A \to t$ then $s \to B \to t$, it would be complete in just two iterations! So the question on our minds today is can we find an algorithm that provably requires only polynomially many iterations?



12.2 Shortest Augmenting Paths Algorithm (Edmonds-Karp)

There are several strategies for selecting better augmenting paths than arbitrary ones. Here's one that is quite simple and has a provable polynomial runtime. Its called the Shortest Augmenting Paths algorithm, or the Edmonds-Karp algorithm. The name of the algorithm might give away a slight hint of what it does.

Algorithm: Shortest Augmenting Paths (Edmonds-Karp)

Edmonds-Karp implements Ford-Fulkerson by selecting the *shortest* augmenting path each iteration.

Unsurprisingly, the Shortest Augmenting Paths (Edmonds-Karp) algorithm works by always picking the *shortest* path in the residual graph (the one with the fewest number of edges). By example, we can see that this would in fact find the max flow in the graph above in just two iterations, but what can we say in general? In fact, the claim is that by picking the shortest paths, 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.

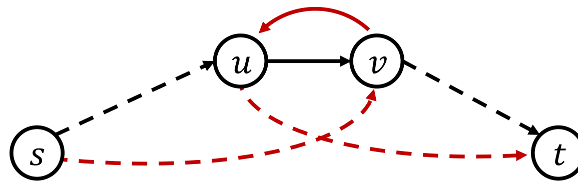
Theorem: Runtime of Edmonds-Karp

The Shortest Augmenting Paths algorithm (Edmonds-Karp) 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:

Claim (a): d never decreases Consider one iteration of the algorithm. Before adding flow to the augmenting path, every vertex v in G has some distance d_v from the source vertex s *in the*

residual graph. Suppose the augmenting path consists of the vertices $v_1, v_2, \ldots v_k$. What can we say about the distances of the vertices? Since the path is by definition a *shortest path*, it must be true that $d_{v_i} = d_{v_{i-1}} + 1$, that is, every vertex is one further from s. Now perform the augmentation and consider what changes in the residual graph. Some of the edges (at least one) become *saturated*, which means that the flow on the edge reaches its capacity. When this happens, that edge will be removed from the residual graph. But another edge might appear in the residual graph! Specifically, when $e = (u, v)$ is saturated, $e' = (v, u)$ may appear in the residual graph as a back edge (if it doesn't exist already). Can this lower the distance of any vertex? No, $d_v = d_u + 1$, so adding an edge from v to u can't make a shorter path from s to t.



Therefore, since the distance to any vertex can not decrease, d can not decrease.

Claim (b): every m iterations, d has to increase by at least 1 Each iteration saturates (fills to capacity) at least one edge. Once an edge is saturated it can not be used because it will not appear in the residual graph. For the edge to become usable again, it must be the case that its back edge in the residual graph is used, which means that the back edge needs to appear on the shortest path. However, if $d_v = d_u + 1$, then it is not possible for the back edge (v, u) to be on a shortest path, so this can *only* occur if d increases. Since there are m edges, d must increase by at least one every m iterations.

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

This shows that the running time of this algorithm is $O(nm^2)$. Note that this is true for **any** capacities, including large ones and non-integer ones. So we really have a polynomial-time algorithm for maximum flow!

12.3 Dinic's Algorithm

The Edmonds-Karp algorithm already gives us polynomial runtime, but now we will try to do better. The key idea is going to be to eliminate some redundancy in the Edmonds-Karp algorithm. Here are two observations that can guide us towards a better algorithm. Let d be the distance from s to t in the current residual graph:

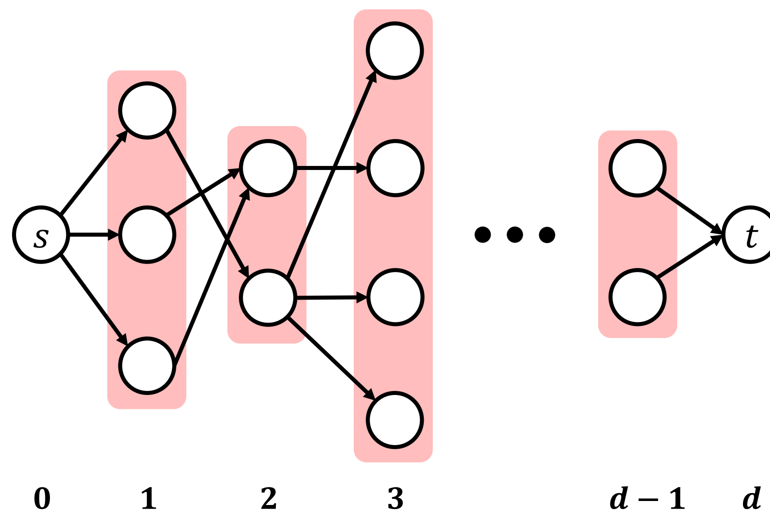
- Our analysis above implies that there are up to mn iterations, but d can only increase n times, so there could be up to m iterations for each value of d

- The breadth-first search algorithm, which we use to find a shortest path, doesn't only return a single shortest path. It finds *all of the distances* in the graph from s, which means it encodes *every possible shortest path* of length d.

So maybe we don't actually need to do a BFS for every augmenting path. Could we do a BFS and use the resulting information to find *multiple shortest augmenting paths* such that any additional paths would contain at least one saturated edge and hence not be usable? That is exactly the idea of *Dinic's algorithm*.

12.3.1 The layered graph

Our tool for finding many shortest paths at once is going to be a way of visualizing all of the shortest paths in the graph, called the *layered graph*. Suppose we run a BFS from s to compute the shortest path distance d_v for every vertex $v \in V$. By definition, $d = d_t$. Now we visualize the graph in the following way: lay out all of the vertices of distance one in a "layer", followed by all vertices of distance two, and so on. The following diagram illustrates the idea.



The layered graph L only includes the edges that go from one distance layer to the next higher layer. Note that there can not exist any edges that go from a layer to a layer more than one higher (because then the distance to that vertex would actually be lower than our layered graph implies). There can be edges in the original graph that go from a vertex to a vertex in a lower layer, but we ignore those because they can not be part of a shortest path.

Once we have build the layered graph, our goal is to find a set of paths in it such that we can not find any more capcitated paths, i.e., we want to find a maximal set of shortest augmenting paths. This general concept turns out to be extremely useful in the theory of network flow, so it even has a name. Its called a *blocking flow*.

12.3.2 Blocking flows

Definition: Blocking Flow

A *blocking flow* in a network H is a flow such that every path in H has at least one edge that gets saturated (filled to capacity).

Another way to think of it is that a blocking flow is a collection of paths that saturate at least one edge of every possible path in the network. Note that this is not the same thing as a maximum flow! Every maximum flow is also a blocking flow, since otherwise there would exist an augmenting path, but not every blocking flow is a maximum flow. Blocking flows can be thought of informally as "maximal flows", i.e., they can not be made larger with any capacitated $s-t$ path. Yet another way to think of them is that a blocking flow is what you would get if you ran Ford-Fulkerson but you forgot to include the back edges in the residual graph!

In the layered graph specifically, this means that we are looking for a collection of shortest paths of length d such that after augmenting all of them, there are no more paths in the residual graph of length d, which is exactly what we want. It completely "uses up" distance d. If we find a blocking flow and augment along it, then d must increase, and hence we can only find n blocking flows before we are done. Our goal is therefore reduced to designing a fast algorithm for finding blocking flows!

12.3.3 An algorithm for blocking flows

We can compute the layered graph implicitly with a BFS from s in $O(n+m)$ time. Although it doesn't matter in theory, in practice, it is a nice optimization to realize that you don't actually have to make an explicit copy of the layered graph, rather, you can just read it out of the original graph by ignoring all edges (u,v) that **don't** satisfy $d_v = d_u+1$. In other words, the layered graph is just the original graph but only counting edges that are actually on a shortest path from s.

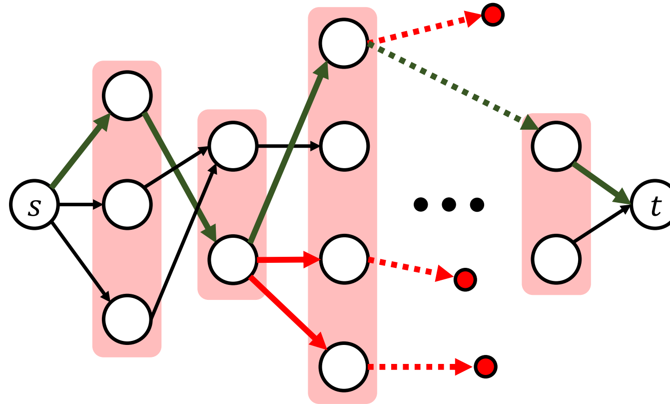
Key Idea: Implicit layered graph

The layered graph is the subgraph of G_f containing only (u,v) that satisfy $d_v = d_u + 1$.

Once we have the layered graph, at a high level, we are just going to find augmenting paths in it using DFS. How fast would this be with a naive implementation? Well, each DFS could cost $O(m)$ and there could be up to m augmenting paths in a blocking flow since each of them saturates one edge, so this could take $O(m^2)$ time. This so far isn't an improvement over Edmonds-Karp since the whole algorithm would take $O(nm^2)$ time to find up to n blocking flows. Somehow we need to speed this up and make finding the augmenting paths in the blocking flow more efficient.

Lets consider what actually happens during a DFS that is searching for an augmenting path in the layered graph. The search has to follow two rules: It can only use edges in the layered graph (i.e., $d_v = d_u + 1$) and the edge must not already be saturated, so $f(u,v) < c(u,v)$. Here's

a key idea. A successful augmenting path will always contain at most $n-1$ edges, otherwise there would be repeat vertices which is pointless. But the naive DFS could take $O(n+m)$ time, why? Because it might visit a bunch of edges that end up not actually being part of the final augmenting path, because the search reached a dead end.



Here's a diagram illustrating the idea. Suppose the green path is the successful augmenting path found. The red edges/paths represent unsuccessful branches of the DFS that did not manage to find an unsaturated path to t. If the DFS could magically only take the correct branches every time, it would take $O(n)$ and we would be very happy! So here's the idea. Any time our DFS finds an unsuccessful branch, like the ones marked in red above, we know that future DFS iterations shouldn't bother to try those paths, because they didn't work last time and they won't work this time either! So, once an edge has been searched and found to be unsuccessful, we can mark it as "dead" and never search it in future iterations.

12.3.4 Runtime analysis

Given the strategy above, we claim that we can find a blocking flow faster than the naive strategy.

Theorem: Runtime of blocking flows

A blocking flow can be found in $O(nm)$ time.

Proof. First, we do a BFS to find all of the distances to each vertex and establish the layered graph. This takes $O(n+m)$ time. Now, we do up to m DFS's to find augmenting paths in the layered graph. Naively, these take $O(m)$ time each and we are in trouble. Instead, we use the strategy above of marking edges on unsuccessful search paths as "dead" and never search them in a future DFS. Each edge in the graph can only be searched unsuccessfully once across all DFS iterations, and the running time of each DFS is therefore

$$n + \text{number of new edges marked dead}$$

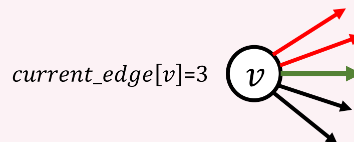
Summing this up over m DFS's, the total running time is

$$\begin{aligned}
&\sum_{i=0}^{m}\big(n + \text{number of new edges marked dead in iteration } i\big),\\
&= nm + \sum_{i=0}^{m}\big(\text{number of new edges marked dead in iteration } i\big),\\
&= nm + \text{total number of new edges ever marked dead},\\
&= nm + m,\\
&= O(nm).
\end{aligned}$$

What we have basically done here is an aggregate amortized analysis to show that the amortized cost of each DFS in the blocking flow is at most $O(n)$. Therefore, we can find a blocking flow in $O(nm)$ time. □

Remark: Implementing the blocking flow algorithm

How should we actually implement the blocking flow algorithm in practice? Its not obvious how we actually mark edges as "dead" during a search. We could copy the entire graph and then delete the edges that are marked dead from the adjacency list, but this is quite inefficient since it requires (a) copying the entire graph and (b) deleting edges from the adjacency list is not efficient if it is stored as an array-based list, so we'd have to use a linked list, BST, or dynamic hash table. Here's a common alternative that can be used to get a fast and practical implementation. For each vertex, store a counter that marks the index of the current edge that we are searching from.



When the search arrives at vertex v, search from *current_edge*$[v]$ and if successful, do not modify *current_edge*$[v]$ (it may still be useful to use the same edge again in the next augmenting path if it is not yet saturated). If the edge is saturated, or the search is unsuccessful and backtracks, then increment *current_edge*$[v]$, which marks the edge as dead, and continue searching from the next edge.

By using this algorithm for blocking flows, we get an algorithm for max flow in $O(n^2 m)$ time, which is an improvement over Edmonds-Karp's $O(nm^2)$.

12.4 Dinic's algorithm for unit-capacity graphs

A remarkable thing about Dinic's algorithm is that it seems to perform really well in practice – much better than the $O(n^2 m)$ bound would suggest. In fact, we can prove that for many classes of common real-world graphs, it is actually asymptotically faster!

Lemma 12.1: Finding a blocking flow in a unit-capacity graph

If every edge in the graph has capacity one, then a blocking flow can be found in $O(m)$ time.

Proof. First note that the unit capacity property is preserved in all residual graphs. We now just make a slight modification to the analysis above. Previously, we said that there can be up to $O(m)$ augmenting paths, and each of them is at most $n-1$ edges long, so the total runtime of finding a blocking flow was $O(nm)$ (the total length of all the augmenting paths), plus the cost of visiting the dead-end edges at most once which was $O(m)$. We can be tighter when the edges are all unit capacity.

Since the edges have capacity one, no edge can be used in multiple augmenting paths (they will all be edge disjoint now), so the total length of all augmenting paths is at most m. The cost of finding the blocking flow is the sum of the lengths of the augmenting paths, which is m, plus the cost of visiting the dead-end edges, which is an additional $O(m)$, so the whole blocking flow takes $O(m)$. □

Lemma 12.1 shows that in unit-capacity graphs, we can therefore find the max flow in at most $O(mn)$ time since we need at most n blocking flows. It turns out that we're not done and we can still improve more!

Lemma 12.2: Number of blocking flows in a unit-capacity graph

If every edge in the graph has capacity one, then we need at most $2\sqrt{m}$ blocking flows.

Proof. Let us consider the state of the residual network after k blocking flows have been found, for any arbitrary value of k. At this point, d must be at least k since each blocking flow increases the distance. Now we ask, what is the max flow of this residual graph, i.e., how much more flow can we augment in the graph before we hit the max flow? Well, since the distance is currently at least k, every augmenting path has length at least k, and there are m total edges in the graph. Since the edges are unit capacity, each can only be used in one path, so we can form at most m/k augmenting paths. Each augmenting path has capacity one, and hence the max flow in the residual network is at most m/k.

Since each blocking flow adds at least one unit of flow, this means that we need at most m/k additional blocking flows. Since we have already done k blocking flows, the total number of blocking flows across the whole algorithm is at most

$$k+\frac{m}{k}.$$

Now what was k again? Well it could be any arbitrary constant we like, so if we can pick k to minimize this expression, we get a good bound on the number of blocking flows needed. We could do calculus to find the minimum, but that's too hard, so here's an easier trick that is quite a nice one to know. We want to minimize the sum of a term that is increasing with k and a term

that is decreasing with k. Asymptotically (within a factor of two), we can minimize such an expression by finding when they are equal[1]. So when $k = m/k$, we have $k = \sqrt{m}$, and therefore number of blocking flows is at most $2\sqrt{m}$. □

This shows that on a unit-capacity network, Dinic's algorithm runs in time $O(m\sqrt{m})$. For sparse graphs ($m = \Theta(n)$), this is an improvement over $O(nm)$. For dense graphs ($m = \Theta(n^2)$), it is about the same.

[1]This works because $f(k) + g(k) \leq 2\max(f(k), g(k))$, and the maximum is minimized when the two are equal.

Lecture 13

Minimum-cost Flows

We have discussed max flows, min cuts, their applications, and several algorithms for it. Today, we consider a generalization of max flow called *minimum-cost* flows, where edges have costs as well as capacities, and the goal is to find flows with low cost. This problem has a lot of nice theoretical properties, and is also highly practical since it generalizes the max flow problem. We will give two algorithms for the problem that illustrate a nice duality in algorithm design.

Objectives of this lecture

In this lecture, we will:

- Define and motivate the minimum-cost flow problem
- Analyze the properties of minimum-cost flows such as how to determine when they are optimal
- Derive and analyze some algorithms for minimum-cost flows

13.1 Minimum-Cost Flows

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

The generalization of this problem to flows is called the *minimum-cost flow* problem.

Problem: Minimum-cost flow

We are given a directed graph G where each edge has a *cost*, $\$(e)$, and a capacity, $c(e)$. As before, an s-t *flow* is an assignment of values to edges that satisfy the capacity and conservation constraints, and the *value* of a flow is the net outflow of the source s. The **cost** of a flow is then defined as the sum over all edges, of the cost per unit of flow:

$$\text{cost}(f) = \sum_{e \in E} \$(e) f(e).$$

Remark: Cost is per unit of flow

Note **importantly** that the cost is charged *per unit* of flow on each edge. That is, we are not paying a cost to "activate" an edge and then send as much flow through it as we like. This similar problem turns out to be NP-Hard.

The goal of the minimum-cost flow problem is to find feasible flows of minimum possible cost. There are a few different variants of the problem.

- We will consider the **minimum-cost maximum flow** problem, where we seek to find the minimum-cost flow out of all possible maximum flows.
- An alternative formulation is to try to find the minimum-cost flow of value k for some given parameter k, rather than a maximum flow.

These problems can be reduced to each other so they are all equivalent. There are also many other variants of the problem, but we won't consider those for now. Formally, the min-cost max flow problem is defined as follows. Our goal is to find, out of all possible maximum flows, the one with the least total cost. What should we assume about our costs?

- In this problem, we are going to allow *negative costs*. You can think of these as rewards or benefits on edges, so that instead of paying to send flow across an edge, you get paid for it.
- What about negative-cost cycles? It turns out that minimum-cost flows are still perfectly well defined in the presence of negative cycles, so we can allow them, too! Some algorithms for minimum-cost flows however don't work when there are negative cycles, but some do. We will be explicit about which ones do and do not.

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.

13.1.1 The residual graph for minimum-cost flows

Let's try to re-use the tools that we used to solve maximum flows for minimum-cost flows. Remember that our key most important tool there was the *residual graph*, which we recall has edges with capacities

$$c_f(u,v) = \begin{cases} c(u,v) - f(u,v) & \text{if } (u,v) \in E, \\ f(v,u) & \text{if } (v,u) \in E. \end{cases}$$

We will re-use the residual graph to attack minimum-cost flows. We need to generalize it, though, because our current definition of the residual graph has no ideas about the costs of the edges. For forward edges e in the residual graph (i.e., those corresponding to $e = (u,v) \in E$), the intuitive value to use for their cost is $\$(e)$, the cost of sending more flow along that edge. What about the reverse edges, though? That's less obvious. Let's think about it, when we send flow along a reverse in the residual graph, we are removing or *redirecting* the flow somewhere else. Since it cost us $\$(e)$ per unit to send flow down that edge initially, when we remove flow on an edge, we can think of getting a *refund* of $\$(e)$ per unit of flow – we get back the money

that we originally paid to put flow on that edge. Therefore, the right choice of cost for a reverse edge in the residual graph is $-\$(e)$, the negative of the cost of the corresponding forward edge!

Definition: The residual graph for minimum-cost flows

The residual graph G_f for a minimum-cost flow problem has the same capacities as the original maximum flow problem, but now has costs of $\$(e)$ on a forward edge, and $-\$(e)$ on a back edge. That is, suppose we denote by $\overleftarrow{e}$, the corresponding reverse edge of e in the residual graph. We have

$$c_f(e) = c(e) - f(e), \qquad \$_f(e) = \$(e),$$
$$c_f(\overleftarrow{e}) = f(e) \qquad \$_f(\overleftarrow{e}) = -\$(e)$$

The definitions on the left are the same as regular max flow.

13.2 An augmenting path algorithm for minimum-cost flows

Now that we have defined a suitable residual graph for minimum-cost flows, we can build an algorithm based on augmenting paths in the same spirit as Ford-Fulkerson. If we just try to pick arbitrary augmenting paths, then it is unlikely that we will find the one of minimum cost, so how should we select our paths in such a way that the costs are accounted for? The most intuitive idea would be to select the *cheapest augmenting path*, i.e., the shortest augmenting path with respect to the costs. This is not to be confused with the shortest augmenting path algorithm that selects the path with the fewest edges (i.e., the Edmonds-Karp algorithm).

Since the edges are weighted by cost, a breadth-first search won't work anymore. How about our favorite shortest paths algorithm, Dijkstra's? Well, that won't quite work since the graph is going to have negative edge costs (note that even if the input graph does not have any negative costs, the residual graph will, so Dijkstra's will not work here). So, let's use our next-favorite shortest paths algorithm that is capable of handling negative edges: Bellman-Ford! It is important to note here that since the algorithm makes use of shortest path computations, if there is a negative-cost cycle, this algorithm will not work.

Algorithm: Cheapest augmenting paths

Implement Ford-Fulkerson by selecting the *cheapest* augmenting path with respect to residual costs.

While there exists any augmenting path in the residual graph G_f, find the augmenting path with the cheapest cost and augment as much flow as possible along that path. Since this is just a special case of Ford-Fulkerson, the fact this algorithm finds a maximum flow is immediate, right? Well, not quite. Remember that since shortest paths only exist if there is no negative-cost

cycle, we need to prove that our algorithm never creates one after performing an augmentation, or the next iteration's shortest path computation will fail.

This result will be a direct corollary of a cool lemma. This lemma should seem familiar and intuitive since it is really just the weighted version of the lemma used to prove the correctness of the Edmonds-Karp algorithm last time!

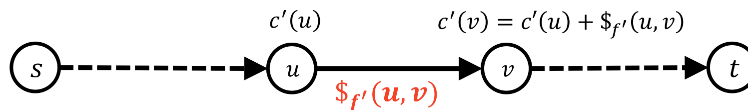
Lemma 13.1: Cheapest path does not decrease

Consider a flow network with costs G and a flow f such that G_f contains no negative-cost cycles. Let f' denote a flow obtained by augmenting f with a cheapest augmenting path from G_f. Then, the cost of the cheapest s-t path in $G_{f'}$ is at least as large as the cheapest s-t path in G_f. (In other words, the cheapest path never gets cheaper!)

Proof. Let G be a flow network with costs and f be a flow such that G_f contains no negative-cost cycles. Let us denote by $c(v)$, the cost of the cheapest path in G_f from s to v for any v. Suppose we augment f with a cheapest augmenting path from G_f to obtain a new flow f'.

Suppose for the sake of contradiction that there exists a walk[1] from s to t whose cost is cheaper than $c(t)$. Let v be the earliest vertex such that the cost of walking from s to v along this walk is cheaper than $c(v)$, and denote this cost by $c'(v) < c(v)$. Then, let u be the vertex directly preceding v on the walk, and denote the cost of walking to that occurrence of u by $c'(u)$.

Since u precedes v, the cost of v is $c'(v) = c'(u) + \$_{f'}(u,v)$.

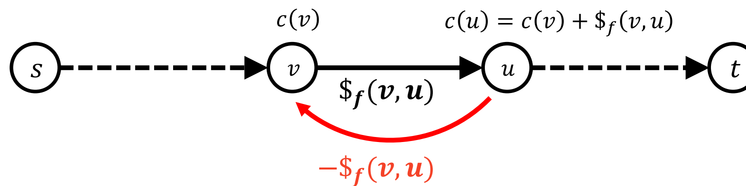


Since v is the earliest such vertex on the path, we know that $c'(u) \geq c(u)$ (it might be greater, not equal, because an edge that was previously on the cheapest path got saturated), so

$$c'(v) \geq c(u) + \$_{f'}(u,v).$$

Now, if (u,v) had the same residual cost in G_f and $G_{f'}$ (i.e., $\$_{f'}(u,v) = \$_f(u,v)$), then $c'(v) \geq c(u) + \$_f(u,v) \geq c(v)$, which would contradict $c'(v) < c(v)$, so it must be that the residual cost of (u,v) changed. This means that the augmenting path must have used (v,u) and activated the corresponding back edge (u,v). Therefore $\$_{f'}(u,v) = -\$_f(v,u)$.

[1] A walk is a path that might contain repeated vertices. Technically we are required to consider walks instead of paths because hypothetically if a negative-cost cycle were to appear in $G_{f'}$, the resulting cheaper "paths" would contain repeated vertices.



So, we have $c'(v) \geq c(u) - \$_f(v,u)$, however, in G_f note that since $c(u) = c(v) + \$_f(v,u)$, we also have $c(v) = c(u) - \$_f(v,u)$, which would imply that $c'(v) \geq c(v)$, a contradiction. □

Corollary 13.1: Maintenance of minimum-cost flows

If G_f contains no negative-cost cycles, then after augmenting with a cheapest augmenting path, it still contains no negative-cost cycles.

Proof. If there were a negative-cost cycle, then we could use that cycle to create paths of arbitrarily low cost, which would immediately imply that the cheapest path to t would be cheaper than before (or undefined). □

So, we have successfully proven that the algorithm will terminate, and since it is a special case of Ford-Fulkerson, we already know that it terminates with a maximum flow! We can also argue about the runtime using the same logic that we used for Ford-Fulkerson. At every iteration of the algorithm, at least one unit of flow is augmented, and every iteration has to run the Bellman-Ford algorithm which takes $O(nm)$ time. Therefore, the worst-case running time of cheapest augmenting paths is $O(nmF)$, where F is the value of the maximum flow, assuming that the capacities are integers.

What we still have to prove, however, is that this algorithm truly finds a *minimum-cost* flow.

13.3 An optimality criteria for minimum-cost flows

When studying maximum flows, our ingredients for proving that a flow was maximum were augmenting paths and the minimum cut. We'd like to find a similar tool that can be used to prove that a minimum-cost flow is optimal. First, let's be specific about what we mean by optimality.

Definition: Cost optimality of flows

We will say that a flow f is *cost optimal* if it is the cheapest of all possible flows *of the same value.*

That is to say we don't compare the costs of different flows if they have different values. Our goal is to find a tool to help us analyze the cost optimality of a flow. To do so, we're going to think about what kinds of operations we can do to modify a flow and change its cost *without* changing its value.

When finding maximum flows, our key tool was the *augmenting path*. If there existed an augmenting path in G_f, then by adding flow to it, we could transform a given flow into a more optimal one that was still feasible because adding flow to an s-t path preserved the flow conservation condition, and didn't violate the capacity constraint as long as we added an appropriate amount of flow. Therefore, the existence of an augmenting path proved that a flow was not maximum, and we were able to later prove that the lack of existence of an augmenting path was sufficient to conclude that a flow was maximum. We want to discover something similar for cost optimality of a flow. Augmenting paths were just one way to modify a flow while keeping it feasible. I claim that there is one other way that we can modify a flow while still preserving feasibility, but that also doesn't change its value. Instead of adding flow to an s-t path, what if we instead add flow to a *cycle* in the graph? Let's call this an *augmenting cycle*.

Since a cycle has an edge in and an edge out of every vertex, adding flow to a cycle in the residual graph preserves flow conservation, and hence feasibility. Furthermore, since the same amount of flow goes in and out of each vertex, the flow value is unaffected! However, adding flow to a cycle may in fact change the cost of the flow, which is what we wanted! Suppose the costs of the edges $e_1, e_2, ..., e_k$ on the cycle are $\$_1, \$_2, ..., \$_k$ for some cycle of length k. Then, adding Δ units of flow to the cycle will change the cost by

$$\sum_{i=1}^{k} \Delta \cdot \$_i = \Delta \cdot \sum_{i=1}^{k} \$_i.$$

Now, note that $\sum \$_i$ is just the weight of the cycle! So, we can say that if we add Δ units of flow to a cycle of weight W, then the cost of the flow changes by $\Delta \cdot W$. If $\Delta \cdot W$ is negative, then we have just shown that the cost of the flow *can be decreased*, so it wasn't optimal!

It turns out that this is the key ingredient for analyzing minimum-cost flows. Even better, we are getting a two-for-one deal because a lack of negative-cost cycles was what we already argued was required for our cheapest augmenting path algorithm to produce a maximum flow. It turns out that this same condition allows us to prove that the flow is cost optimal. We just need a few more concepts first, then we are done, I promise!

13.3.1 Differences of flows and circulations

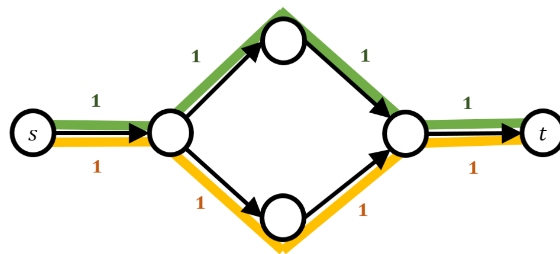
We just argued that the presence of a negative-cost cycle implies that we can find a flow of lesser cost by *adding* two flows together. Adding flows together is defined in the natural way you would think of. We add the values of the flow on each edge to get a new flow. One can show from the definition that if a flow f is feasible in G and another flow f' is feasible in G_f, then $f + f'$ is feasible in G (where we define adding flow to the reverse edge as removing flow from the corresponding forward edge, just like in Ford-Fulkerson).

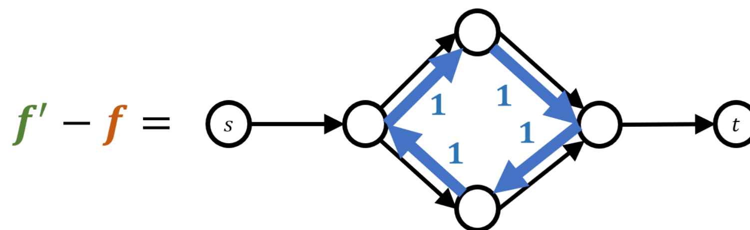
To complete the analysis of minimum cost flows, we will also need the slightly less intuitive notion of the *difference* between two flows. Given a flow f' and a flow f, what would we want $f' - f$ to mean? We could just take the edgewise difference between each of the flows, but this could sometimes result in a negative amount of flow which does not quite make sense. However, the following idea will make it make sense!

- if $f'(u, v) - f(u, v) \geq 0$, then we will say that (u, v) has $f'(u, v) - f(u, v)$ flow,

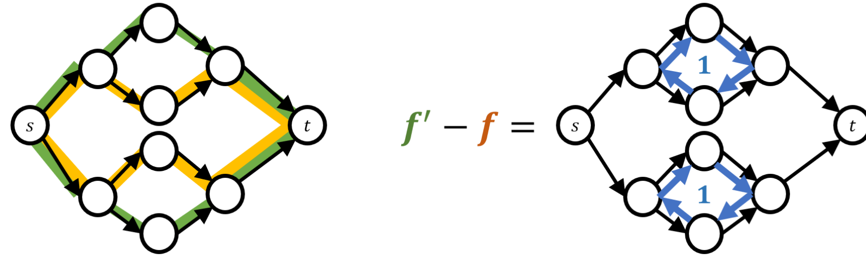
- if $f'(u,v) - f(u,v) < 0$, then we will say that (v,u) has a flow of $f(u,v) - f'(u,v)$.

In other words, a negative amount of flow will just be treated as a positive amount of flow going in the other direction! This is analogous to the way that reverse edges are treated by the Ford-Fulkerson algorithm; they remove flow in the other direction. Given two flows, what does this actually look like? Suppose we have two flows f' and f *of the same value*. In this example, the difference between the green flow and the yellow flow is that the flow on the edges adjacent to s and t cancel, and the flow around the bottom two edges *reverses*. This results in a *cycle* around the middle of the graph!





Now the claim is that this is what always happens: the difference between two flows of the same value is a collection of (possibly multiple) cycles. But how would we prove this, and how do we know that it is even a valid flow? Well, since both f' and f are feasible flows, they satisfy the flow conservation property, and hence their difference $f' - f$ also satisfies the flow conservation property (for example, if some vertex has 5 flow in and out in f' and 3 flow in and out in f, then their difference has 2 flow in and out). What is the value of this flow? By assumption, the values of f' and f are the same, hence their net flows out of s are the same. By taking the difference between the two, we can see that the net flow out of s in $f' - f$ is actually **zero**. So, we have a flow of value zero, but it isn't the all-zero flow, so what does this look like? Well, every vertex *including the source and sink* have flow in equal to flow out, which means that the flow is indeed a *collection of cycles*. For this reason, such a flow is often called a *circulation*.



13.3.2 Proving the optimality criteria

Theorem 13.1: Cost optimality and negative cycles

A flow f is cost optimal if and only if there is no negative-cost cycle in G_f.

Proof. Suppose that there exists a negative cost cycle in the residual graph. Then by adding flow to this cycle, the value of the flow doesn't change, but the cost changes by $\Delta \cdot W$, where Δ is the amount of flow we can add, and W is the cost/weight of the cycle. Since W is negative, the cost decreases, and hence the flow f was not cost optimal.

Now suppose that a flow f is not cost optimal. We need to prove that there exists a negative-cost cycle in its residual graph G_f. This case is much trickier. Since f is not cost optimal, there exists some other flow f' of the same value but which has a cheaper cost. Let's now consider the difference flow $f' - f$. From the previous section, this is a *circulation*, i.e., a collection of cycles of flow. What is the cost of this circulation? Since costs are just linear (we sum the cost per flow along each edge), we get that

$$\text{cost}(f' - f) = \text{cost}(f') - \text{cost}(f),$$

and since we assumed that $\text{cost}(f') < \text{cost}(f)$, this must be a *negative cost*. Now we wish to show that $f' - f$ is *feasible in* G_f. We consider the two cases in the definition of $f' - f$:

- If $f'(e) - f(e) \geq 0$, then the forward edge e has flow $f'(e) - f(e) \leq c(e) - f(e)$ which is the definition of the residual capacity $c_f(e)$,
- if $f'(e) - f(e) < 0$, then the reverse edge $\overleftarrow{e}$ has flow $f(u, v) - f'(u, v) \leq f(u, v)$, which is the definition of the residual capacity of $c_f(\overleftarrow{e})$.

Therefore, $f' - f$ is a feasible flow in the residual graph! So, to conclude, we have a collection of cycles of flow whose total cost is negative, therefore at least one of those cycles must have a negative cost. Since $f' - f$ is feasible in the residual graph, this negative-cost cycle exists in the residual graph, which is what we wanted to prove. □

Corollary: Optimality of cheapest augmenting paths

Since we already argued in Corollary 13.1 that the cheapest augmenting path algorithm never creates a negative-cost cycle in the residual graph, this proves that the resulting flow is also cost optimal. Therefore we can conclude that the cheapest augmenting path algorithm successfully finds the minimum-cost maximum flow!

13.4 Cycle canceling: Another algorithm for min-cost flow

Finally, lets talk about another algorithm for minimum-cost flows that demonstrates a really nice duality in algorithm design. We're going to derive and analyze an algorithm that solves the min-cost flow problem in rather the "opposite" way to the cheapest augmenting paths problem.

At a high level, the cheapest augmenting paths algorithm works by beginning with a flow that is cost optimal (the all zero flow, assuming no negative-cost cycles) but not maximum, and then iteratively making the flow more maximum via augmenting paths while maintaining the invariant that it is always cost optimal. What if we did the opposite and started with a flow that is maximum but not cost optimal, and then iteratively made it cheaper while maintaining the invariant that it is maximum? That is the idea of *cycle canceling*.

Recall that a flow is maximum if and only if there are no augmenting paths (we proved this implicitly while analyzing the Ford-Fulkerson algorithm). Theorem 13.1 gives an analogous tool for cost optimality. A flow is cost optimal if and only if it has no negative cycles. So, just like the Ford-Fulkerson algorithm works by identifying augmenting paths and adding flow to them until none exist, we can write an algorithm that does the same but for negative-cost cycles. We just need an algorithm that finds a negative-cost cycle or reports that none exist. Luckily, the Bellman-Ford algorithm does exactly this. If the algorithm finds a negative cycle, it can use it as an *augmenting cycle* to to decrease the cost of the flow. Remember that adding flow to a cycle maintains feasibility, doesn't affect the flow value, and if the cost of the cycle is negative, it results in a decrease to the cost of the flow.

Algorithm: Cycle-canceling for minimum cost flows

```
find a maximum flow f with any max flow algorithm
while there exists a negative-cost cycle in G_f
  augment the maximum possible amount of flow along the cycle
```

A great thing about this is that we don't really need to do any more analysis either. Theorem 13.1 immediately proves that this algorithm, if it terminates, is correct. Another huge plus of this algorithm is that it works even when there are negative-cost cycles in the original graph!

Remark: Minimum-cost flow with negative-cost cycles

Unlike the cheapest augmenting path algorithm, the cycle-canceling algorithm works when the input graph has negative-cost cycles!

How can we prove that it terminates? Well, lets proceed similarly to Ford-Fulkerson where we simplify a bit and assume that the costs are all integers. If so, every augmenting cycle lowers the cost by at least one, so as long as the problem is well defined (the minimum cost is finite), the algorithm will eventually terminate!

Theorem: Running time of cycle canceling

Assume that the graph has integer capacities and integer costs, Then the cycle-canceling algorithm runs in

$$O(nm^2 \max_e c(e) \max_e |\$(e)|)$$

Proof. The maximum possible cost of any flow is

$$\sum_{e \in E} c(e) \max(0, \$(e)) \leq \max_e c(e) \max_e |\$(e)| m,$$

Each iteration of cycle canceling improves the cost by at least one. Since the minimum cost could be negative, the cycle-canceling algorithm could take up to 2 times this many iterations (to go from the highest possible positive cost to the lowest possible negative cost). Each iteration takes $O(nm)$ time with Bellman-Ford, so the total runtime is at most

$$O(nm^2 \max_e c(e) \max_e |\$(e)|),$$

as desired. □

Just like we improved the Ford-Fulkerson algorithm from non-polynomial time to polynomial time by picking better augmenting paths rather than arbitrary ones, the cycle canceling algorithm can also be improved to polynomial time by picking better cycles rather than arbitrary ones. Unfortunately its more complicated, because although the intuitive thing to do would be to pick the most-negative cycle in order to make the most progress, the problem of finding the minimum-cost cycle in a graph is NP-Hard! Choices that work do exist, but we will not cover them. There are also other techniques for solving minimum-cost flow problems in polynomial time and for non-integer capacities and costs, so just know that it can be done even though we won't cover how.

Remark: Minimum-cost flow can be solved in polynomial time

There exists strongly polynomial time algorithms for the minimum-cost flow problem