15-451 / 15-651 Fall 2025

Algorithm Design and Analysis

Fall 2025 taught by

Daniel **Anderson**
Danny **Sleator**

Lecture notes by

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

Last updated November 16, 2025

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 . . . 20

3 Integer sorting **27**
- 3.1 Models of computation for integers . . . 28
- 3.2 Sorting small integers: Counting Sort . . . 30
- 3.3 A side quest: Tuple sorting . . . 32
- 3.4 Sorting bigger integers: Radix Sort . . . 34

Exercises: Integer Sorting **37**

Data structures and their analysis

4 Hashing: Universal and Perfect Hashing **39**
- 4.1 Dictionaries, hashing, and hashtables . . . 40
- 4.2 Universal Hashing . . . 42
- 4.3 More powerful hash families . . . 47
- 4.4 Perfect Hashing . . . 48

Exercises: Hashing **51**

5 Fingerprinting & String Matching **53**
- 5.1 How to Pick a Random Prime . . . 53
- 5.2 How Many Primes? . . . 54
- 5.3 The String Equality Problem . . . 55
- 5.4 The Karp-Rabin Algorithm (the “Fingerprint” method) . . . 58

Exercises: Fingerprinting **61**

6 Range query data structures **63**
6.1 Range queries . . . 63
6.2 Making range queries dynamic . . . 64
6.3 The data structure . . . 67
6.4 Speeding up algorithms with range queries . . . 69
6.5 Extensions of SegTrees . . . 70

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 **84**

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 Longest Increasing Subsequence . . . 102

Exercises: Dynamic Programming **104**

10 Dynamic Programming II **105**
10.1 All-pairs Shortest Paths (APSP) . . . 105
10.2 Traveling Salesperson Problem (TSP) . . . 108

Exercises: Dynamic Programming II **111**

11 Network Flows I **113**
11.1 The Maximum Network Flow Problem . . . 113
11.2 The Ford-Fulkerson algorithm . . . 117
11.3 Bipartite Matching . . . 122

Exercises: Flow Fundamentals **124**

12 Network Flows II **125**
12.1 Network flow recap . . . 125
12.2 Shortest Augmenting Paths Algorithm (Edmonds-Karp) . . . 126
12.3 Dinic's Algorithm . . . 127
12.4 Dinic's algorithm for unit-capacity graphs . . . 131

13 Minimum-cost Flows **135**
13.1 Minimum-Cost Flows . . . 135
13.2 An augmenting path algorithm for minimum-cost flows . . . 137
13.3 An optimality criteria for minimum-cost flows . . . 139
13.4 Cycle canceling: Another algorithm for min-cost flow . . . 143

14 Matrix Games **145**
14.1 Introduction to Game Theory . . . 145
14.2 Definitions and Examples . . . 146
14.3 Von Neumann's Minimax Theorem . . . 153
14.4 Techniques for Solving Games . . . 154
14.5 Lower Bounds for Randomized Algorithms . . . 156
14.6 General-Sum Two-Player Games . . . 159

15 Linear Programming I **161**
15.1 Introduction . . . 161
15.2 Definition of Linear Programming . . . 162
15.3 Modeling problems as Linear Programs . . . 163

Exercises: Linear Programming Fundamentals **168**

16 Linear Programming Duality **169**
16.1 Linear Programming Terminology and Notation . . . 169
16.2 The Dual Program . . . 171
16.3 A General Formulation of the Dual . . . 172
16.4 Example: Zero-Sum Games . . . 174
16.5 The Geometric Intuition for Strong Duality . . . 177

Exercises: Linear Programming Duality **179**

17 Linear Programming: Polytopes and Integrality **181**
17.1 Polytopes, Vertices, and the Simplex Algorithm . . . 181
17.2 Matchings . . . 183
17.3 An algebraic perspective . . . 185

Approximation and uncertainty

18 Approximation Algorithms **189**
18.1 Introduction . . . 189
18.2 Scheduling Jobs on Multiple Machines to Minimize Load . . . 190

18.3 Vertex Cover via LP Rounding . . . 194
18.4 Metric Traveling Salesperson . . . 196

19 Online Algorithms **199**
19.1 Framework and Definition . . . 199
19.2 Rent or buy? . . . 200
19.3 List Update . . . 201
19.4 Paging . . . 204

20 Streaming Algorithms **209**
20.1 Introduction . . . 209
20.2 Finding ε-Heavy Hitters . . . 211
20.3 Heavy Hitters with Deletions . . . 214

Geometry

21 Computational Geometry: Fundamentals **219**
21.1 Introduction . . . 219
21.2 Fundamental Operations . . . 220
21.3 The Convex Hull . . . 225

Exercises: Geometry **231**

22 Computational Geometry: Randomized Incremental Algorithms **233**
22.1 Model and assumptions . . . 233
22.2 The closest pair problem . . . 234
22.3 The smallest enclosing circle problem . . . 238

Exercises: Randomized Incremental Geometry **245**

Advanced algorithm design tools

23 Splay Trees **247**
23.1 Binary Search Trees . . . 247
23.2 Splay Trees (self-adjusting search trees) . . . 249
23.3 Standard BST Operations Using Splaying . . . 251
23.4 Analysis of Splaying . . . 251
23.5 The Amortized Analysis . . . 253
23.6 Balance Theorem . . . 257
23.7 Improvement and Applications of the Access Lemma . . . 257

24 Polynomials in Algorithm Design **261**
24.1 Introduction . . . 261
24.2 Operations on Polynomials . . . 262
24.3 How Many Roots? . . . 263

24.4 Another Representation for degree-d Polynomials 263
24.5 Application: Error Correcting Codes 266
24.6 Multivariate Polynomials and Matchings 270

Exercises: Polynomials in algorithm design **272**

25 The Fast Fourier Transform **273**
25.1 Preliminaries 273
25.2 Polynomial Multiplication: The High Level Idea 275
25.3 To Point-Value Form (The Fast Fourier Transform) 275
25.4 The Inverse of the DFT 278

Exercises: Fast Fourier Transform **280**

Acknowledgements

These lecture notes represent the collective creativity and effort of many generations of 15-451/651 instructors. Originally authored by Avrim Blum, they have since been thoughtfully expanded and refined by Anupam Gupta and Danny Sleator. These days they are edited and maintained by Daniel Anderson. We owe a debt of gratitude to the numerous teaching assistants who have contributed improvements and to the students who have diligently spotted and corrected typos along the way (many more of which certainly remain!).

We would also like to extend our heartfelt thanks to Optiver, our Spring 2025 sponsor, for generously providing the funding for the much-needed sustenance at our TA grading meetings. Without their support, our TAs would face the daunting task of grading on empty stomachs—a situation that may or may not result in a mysterious drop in student grades!

Optiver

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 practice writing and solving recurrence relations, and bounding the *expected* cost of a randomized algorithm, both of which are key tools for the analysis of algorithms that we will use throughout this class.

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 $\leftarrow \{a_i \text{ such that } a_i < p\}$
3. GREATER $\leftarrow \{a_i \text{ such that } a_i > p\}$
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.

Idea: Eliminate redundancy 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!

Implementation 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

[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 $O(n)$.

and partitioning we find that LESS has 200 elements, then we just need to find the 87th smallest element in LESS. On the other hand, if we find LESS has 40 elements, then we just need to find the $87-40-1=46$th smallest element in GREATER. (And if LESS has size exactly 86 then we can just return the pivot). One might at first think that allowing the algorithm to only recurse on one subset rather than both would just cut down time by a factor of 2. However, since this is occuring recursively, it compounds the savings and we end up with $\Theta(n)$ rather than $\Theta(n \log n)$ time. This algorithm is often called Randomized-Select, or QuickSelect.

Algorithm 1.3: QuickSelect

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

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 for any value of k. What we want is a recurrence relation that looks like

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

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.

A first attempt Before giving a formal proof, here's some intuition and a slightly incorrect first attempt. 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.

$$T(n) \leq n-1+T(3n/4). \qquad \textit{(This is wrong!)}$$

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! (As an exercise, argue that the two could differ by a lot.)[2] Let's now see this a bit more formally.

A correct proof 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. By the definition of expected value, we want to bound the following:

$$\mathbb{E}[T(X)] = \sum_{x=1}^{n-1} \Pr[X = x] \cdot T(x)$$

It *is* possible to bound the cost by attacking the above expression with induction and a lot of algebra, but its a bit messy and does not provide very much intuition. Instead, here is a simpler way to go about it. Since we are okay with an upper bound and don't need an exact solution to the recurrence, we can try to upper bound most of the terms in the sum by making them the same. There isn't a single correct way to do this, but the following intuition works. We already know from our slightly incorrect attempt that if the recurrence recurses on $T(3/4n)$ then we get an $O(n)$ upper bound, so we can try to make the correct recurrence also depend on $T(3/4n)$. To do so, we use the fact that $T(n)$ is an increasing function.[3]

So, we can upper bound the quantity in question as follows.

$$\mathbb{E}[T(X)] \leq \Pr\left[X \leq \frac{3n}{4}\right] \cdot T\left(\frac{3n}{4}\right) + \Pr\left[X > \frac{3n}{4}\right] \cdot T(n).$$

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. 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. Although this might sound rather loose, this is good enough to write down a good upper bound recurrence!

[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]If we wanted to be extremely thorough, we could prove this claim separately, but it should be very reasonable to believe that the algorithm does not get faster if we give it a larger input.

Proof Theorem 1.1. We can now use the bound above to show that $T(n) = O(n)$ as desired. We first write

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

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 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:

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

Algorithm 1.4: DeterministicSelect

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

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.

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 $\le p$, and at least 3/10 of the array is $\ge p$. Assuming for the moment that this claim is true, Step 4 takes time at most $T(7n/10)$, and we have the recurrence:

$$T(n) \le cn + T\left(\frac{n}{5}\right) + T\left(\frac{7n}{10}\right),$$

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:

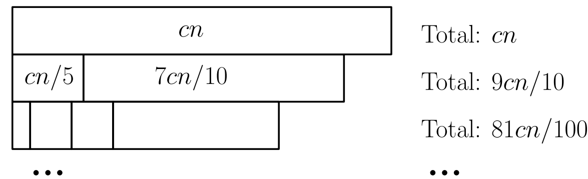
$$\{1,2,3,10,11\},\ \{4,5,6,12,13\},\ \{7,8,9,14,15\}.$$

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 $\le p$) at least three of the five elements are $\le p$. Therefore, the total number of elements $\le p$ is at least $3\lceil g/2 \rceil \ge 3n/10$. Similarly, the total number of elements $\ge p$ is also at least $3\lceil g/2 \rceil \ge 3n/10$.

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

$$c\,n(1+(9/10)+(9/10)^2+(9/10)^3+\ldots),$$

which is at most $10c\,n$. This proves the theorem. □

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

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

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 find the second-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 Finding the second-largest of n elements

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

> ***Claim: Lower bound on select-second-max in the comparison model***
>
> $n-1$ comparisons are needed in the worst-case to find the second-largest of n elements.

Proof. The same argument used in the lower bound for finding the maximum still holds. □

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

Claim: Upper bound #1 on select-second-max in the comparison model

$2n-3$ comparisons are sufficient to find the second-largest of n elements.

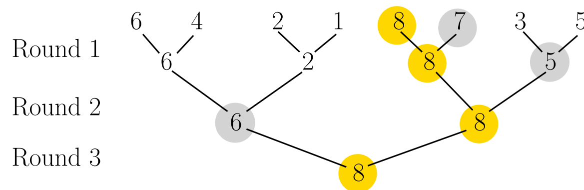
Proof. Just find the largest using $n-1$ comparisons, and then the largest of the remainder using $n-2$ comparisons, for a total of $2n-3$ comparisons. □

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

Claim: Upper bound #2 on select-second-max in the comparison model

$n+\lg n-2$ comparisons are sufficient to find the second-largest of n elements.

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



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

At this point, we have a lower bound of $n-1$ and an upper bound of $n+\lg(n)-2$, so they are nearly tight. It turns out that, in fact, the lower bound can be improved to exactly meet the upper bound, but the proof is rather complicated so we won't do it for now.

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

2.2.3 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*.

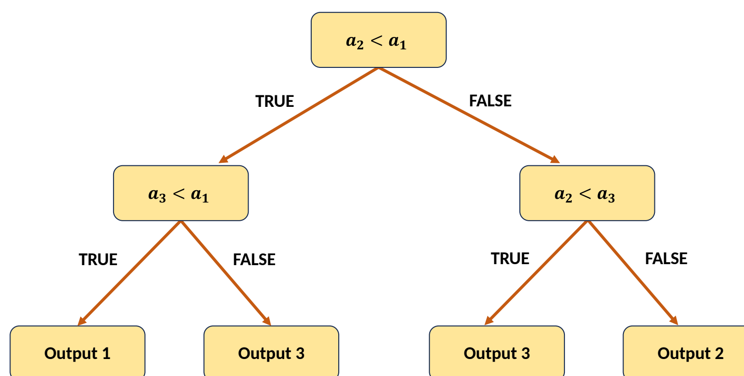
A decision tree is a binary tree that represents the behavior of a specific algorithm based on the outcomes of each comparison it makes. Specifically, each internal node corresponds to a comparison such that the left subtree corresponds to the outcome of the comparison being true and the right subtree 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

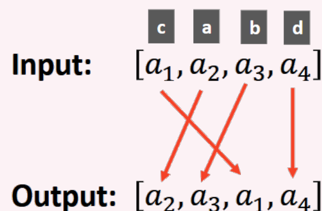
For the problem of *sorting* in the comparison model, the input is an array $a = [a_1, a_2, \dots, a_n]$, and the output is a permutation of the input $\pi(a) = [a_{\pi(1)}, a_{\pi(2)}, \dots, a_{\pi(n)}]$ in which the elements are in increasing order.

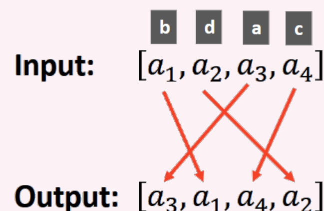
Remark: Correctly defining the "output" of an algorithm

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.*

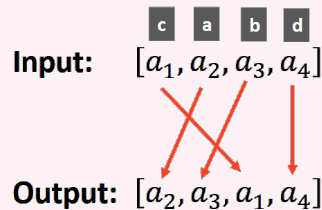
For example, suppose we ask an algorithm to sort $[c, a, b, d]$ and $[b, d, a, c]$. Both of these will become $[a, b, c, d]$ when sorted, so does this mean they were the same output? **No!** The former is sorted by outputting $[a_2, a_3, a_1, a_4]$, and the latter is sorted by outputting $[a_3, a_1,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

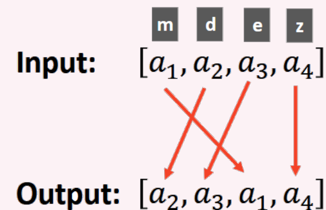




On the other hand, suppose we ask to sort both $[c, a, b, d]$ and $[m, d, e, z]$. These will sort to $[a, b, c, d]$ and $[d, e, m, z]$, which are *both* the permutation $[a_2, a_3, a_1, a_4]$. So these are in fact **the same output**, because their elements are in the same relative permuted order, and 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, since the elements are distinct, there is only a single correct sorted order, and therefore **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 Extra example: 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 $a_1, \ldots, a_n$ 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. There are also multiple valid outputs for a single input since duplicates can be interchanged without violating sorted order.

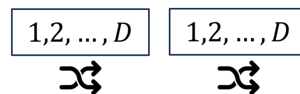
Finding a hard subset of inputs A powerful technique that we will use to overcome this issue is to focus on a *specific subset of inputs* to the problem. If we find some family I of inputs and prove a lower bound on the cost to solve any input from I, then of course that lower bound

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.

Stability 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

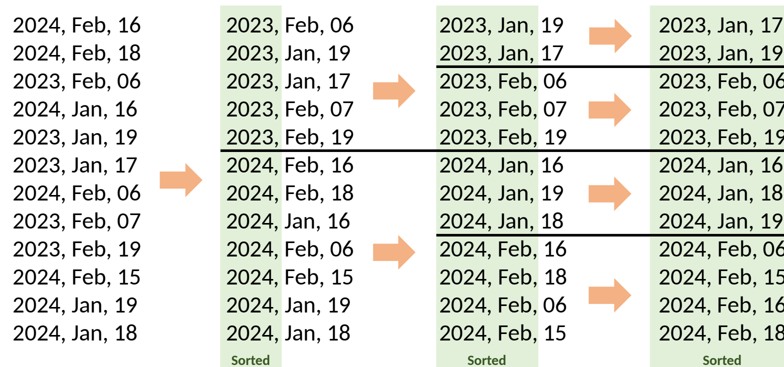
multiply the cost (number of comparisons) by the length of the tuples, i.e., d. This will give us an $O(d\,n\log n)$ algorithm. This works fine in the comparison model, but it doesn't generalize very well outside the comparison model, e.g., to tuples of integers and integer sorting.

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 sort them by year, then recursively sort the dates that have equal years by their month, and then recursively sort the dates with equal years and months.



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! Whether this is efficient depends on the value of u.

3.3.2 Bottom-up tuple sorting

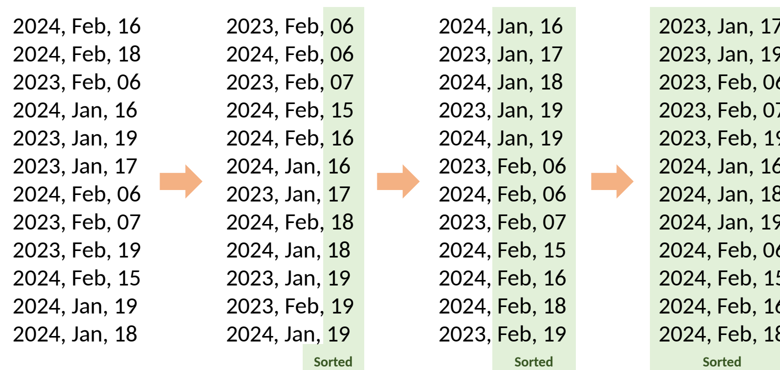
Like most recursive algorithms, we can also find a non-recursive alternative. What if instead of recursively sorting each subarray of equal tuple elements, we just sorted the entire array one tuple element at a time? Say we first sort the entire array using the first tuple element as the key, then the 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.



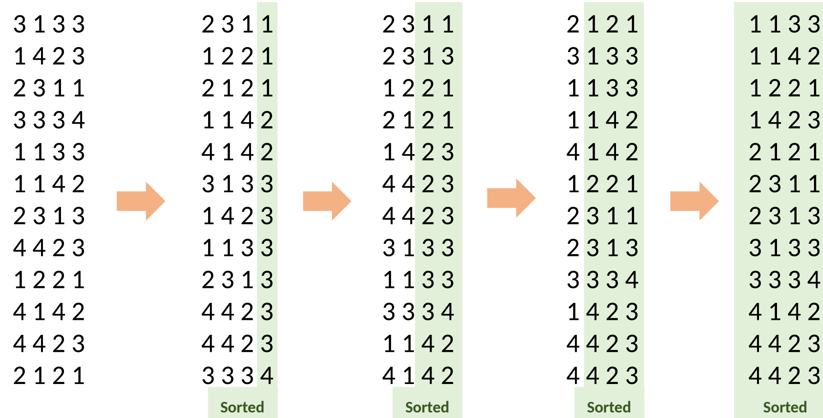
The key difference is that unlike the top-down approach, each round sorts the whole array, rather than recursively subdividing the array into smaller pieces that are sorted separately.

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.

Problem 6. What is the complexity of a recursive MSD Radix Sort? For what base/digit counts is it faster/slower/the same cost as LSD Radix Sort?

Problem 7. Given n integers $a_1, a_2, \ldots, a_n$ in $\{1, 2, 3, \ldots, n^2\}$, give an algorithm that finds indices $i \neq j$ for which $|a_i - a_j|$ is minimized, i.e., find the closest pair of integers. How would you solve this problem with or without what you learned in this lecture? How does the cost differ?

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. Both of these data structures work in the *comparison model*. *Hashtables* are an alternative approach that is often the fastest and most convenient way to solve these problems in practice. It is also, much like integer sorting, a way to beat the best possible algorithms in the comparison model by taking advantage of the assumption that the inputs can be represented as integers. 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

[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.

then we would never be able to find it again. So, we want h to be something "pseudorandom" in some formal sense.

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

Claim: 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 $\mathcal{H}$ where each $h \in \mathcal{H}$ maps $U \rightarrow \{0,\ldots,m-1\}$ is called **universal** (or is called a *universal family*) if for all $x \neq y$ in U, we have

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

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

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

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

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

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 $\mathscr{H}$ 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 $\mathscr{H}$, 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 $O(L)$. □

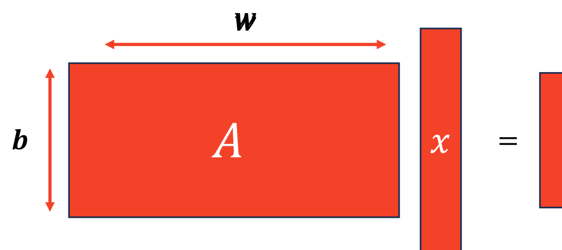
Can we construct a universal $\mathscr{H}$? If not, this this is all useless. Luckily, the answer is yes.

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 $U = 0, \dots, 2^w - 1$. 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 $(1/2)^b = 1/m$. $\square$

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 $\mathcal{H}$ 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

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

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

Claim

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

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

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,

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

□

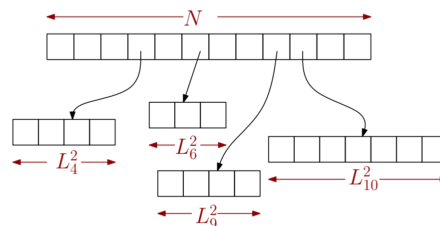
This is like the other side to the "birthday paradox". If the number of days is a lot *more* than the number of people squared, then there is a reasonable chance *no* pair has the same birthday.

So, we just try a random h from $\mathcal{H}$, 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 $\mathcal{H}$, 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 $\mathcal{H}$ 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 8. Show that any pairwise independent hash family is also a universal hash family and also a uniform hash family.

Problem 9. Prove that a hash family that is both uniform and universal is not necessarily pairwise independent.

Problem 10. 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 probabilistically 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[Bob says **equal**] $= 1$.
- If $x \neq y$, then Pr[Bob says **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 $\Pr[\text{ Bob says } \textbf{equal} \,] \leq \frac{1}{100}$.

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

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

□

[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 1%, 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 $\{1,\dots,M\}$ 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.

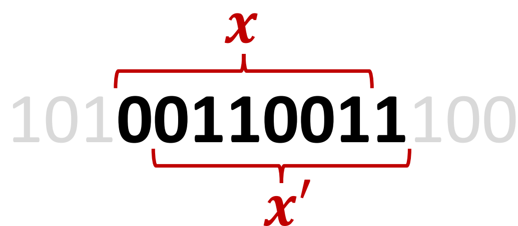
5.4.1 The Karp-Rabin Idea: "Rolling the hash"

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

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

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 $h_p(T_{0\ldots m-1})$, 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 $h_p(T_{i+1\ldots i+m})$ using $h_p(T_{i\ldots i+m-1})$
   (ii) If $h_p(T_{i+1\ldots i+m}) = h_p(P)$, 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

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

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 11. 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 12. 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.

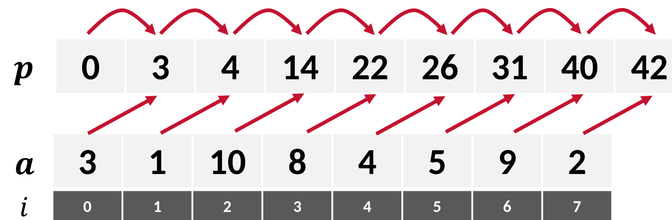
Today, we are going to design a data structure for this problem with the same bounds, $O(\log n)$ *worst-case* time per operation and $O(n)$ words of space, but that is much simpler to implement, and much faster in practice due to smaller constants hidden by the big-O. We will refer to this data structure 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

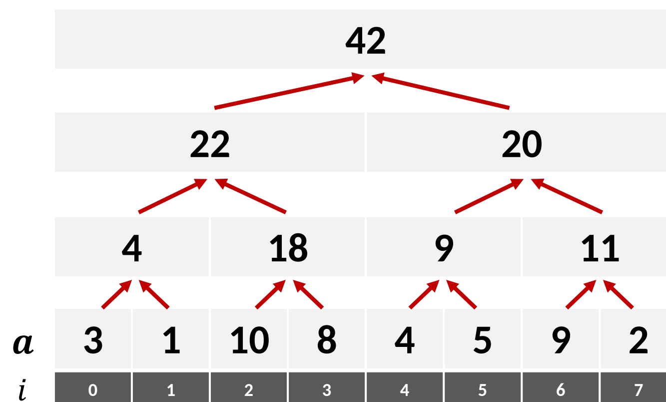
Let's take a step back and have a closer look at Approach #2 from earlier and see if we can find inspiration for a better algorithm. What the algorithm from Approach #2 is really doing is computing the sum from 1 to n in a sequential loop and just saving the results along the way.

[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.



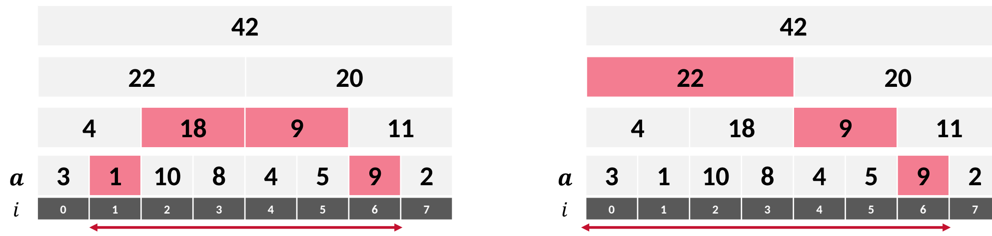
The *dependencies* here are what make the update algorithm slow. Is there instead an alternative way to break up the computation such that most of the intermediate calculations depends on fewer of the numbers? You may or may not have seen this idea before when seeking a *parallel algorithm* for computing 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.

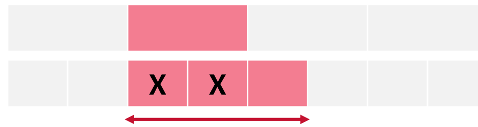


We need to somehow prove that we don't need too many intervals to make up any query interval. Because of course, we could always answer any query $[i, j)$ by summing up all of the intervals of size 1 from i to j, but that's just the first algorithm which takes $O(n)$ time.

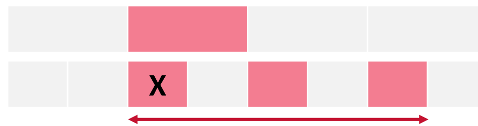
Lemma 6.1: Few blocks per level

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.

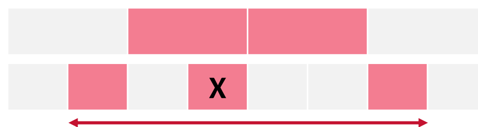
Proof. Suppose for the sake of contradiction that we use three *adjacent* blocks on the same level. Since each block is half the size of its parent, it must be the case that one of the two pairs of adjacent blocks could be replaced by its parent to cover the same interval, yielding a construction that 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 indexing 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 indexing 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 $2i+2$

<table>
<tr><td></td><td colspan="8">0</td></tr>
<tr><td></td><td colspan="4">1</td><td colspan="4">2</td></tr>
<tr><td></td><td colspan="2">3</td><td colspan="2">4</td><td colspan="2">5</td><td colspan="2">6</td></tr>
<tr><td>a</td><td>7</td><td>8</td><td>9</td><td>10</td><td>11</td><td>12</td><td>13</td><td>14</td></tr>
</table>

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:

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

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

Building the tree To build the tree, we start with the input $a[0\dots(n-1)]$ which we copy into the leaves of the tree. We can then iterate over all of the internal nodes and fill in their values based on the values of their children. Note that the order we do this is important. We go from node $n-2$ down to node 0 (these are the indices of the internal nodes) since this corresponds to looping over the nodes on the lower levels before nodes on higher levels. If we tried to start at the root, its children might not have values yet, so this would not work!

Algorithm: Building a SegTree

```
class SegTree {
 nodes : list⟨Node⟩
 n : int }

class Node {
 val : int
 leftIdx : int
 rightIdx : int }

fun SegTree::constructor(a : list⟨int⟩) {
 n = size(a)
 nodes = array⟨Node⟩(2*n-1)
 for i in 0 to (n-1) do
  nodes[n + i - 1] = Node(a[i], i, i+1)
 for u in n-2 to 0 do {
  lNode = nodes[LeftChild(u)]
  rNode = nodes[RightChild(u)]
  nodes[u] = Node(lNode.val + rNode.val,
                  lNode.leftIdx,
                  rNode.rightIdx)
 }
}
```

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

```
fun Assign(i : int, x : int) {
 u := i + n - 1
 nodes[u].val = x
 while u > 0 do {
  u = Parent(u)
  nodes[u] = nodes[LeftChild(u)].val + nodes[RightChild(u)].val
 }
}
```

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 $[i, j)$. Note, importantly, that we only recurse on both sides when necessary! Most of the time, the query will only recurse left or right but rarely both.

Algorithm: Querying a SegTree

```
fun RangeSum(i : int, j : int) {
  return sum(0, i, j)
}

fun sum(u : int, i : int, j : int) {
  node = nodes[u]
  if (i == node.leftIdx and node.rightIdx == j) return node.val
  else {
    mid = (node.leftIdx + node.rightIdx)/2
    if i >= mid then
      return sum(RightChild(u), i, j)
    else if j <= mid then
      return sum(LeftChild(u), i, j)
    else {
      return sum(LeftChild(u), i, mid) + sum(RightChild(u), mid, j)
    }
  }
}
```

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 $p[i] > p[j]$.

```
fun inversions(p : list⟨int⟩) {
  n = size(p)
  result = 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

```
fun inversions(p : list⟨int⟩) {
  n = size(p)
  counts = SegTree(list⟨int⟩(n, 0)) // SegTree containing n zeros
  result = 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 that 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 $X+Y$ 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

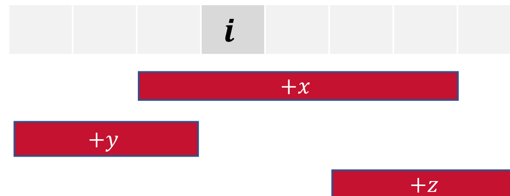
Our vanilla SegTree supports *point updates* and *range queries*, that is, we can edit the value of one element of the sequence and then query for properties (e.g., sum, min, max) of a range of values in $O(\log n)$ time. What if we want to do the opposite? Lets imagine that we want to support the following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

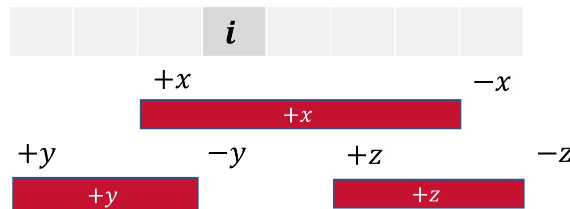
Rather than come up with a brand new data structure, lets try to use our original SegTree as a black box and reduce this new problem to the old one. This should always be your first choice

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.

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 doubling algorithm 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.

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!

A cost model 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 $c = [\lceil m \rceil]$ (this notation means the smallest power of two that is at least m, also called the "super ceiling of m"). Also, notice that $[\lceil m \rceil] \le 2m$ 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

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

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 $\mathbf{ac}_i$ of operation σ_i by the following formula:

$$\mathbf{ac}_i \;=\; c_i + \Phi(S_i) - \Phi(S_{i-1}),$$

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:

$$\sum_i \mathbf{ac}_i \;=\; \sum_i (c_i + \Phi(S_i) - \Phi(S_{i-1})) \;=\; \Phi(S_m) - \Phi(s_0) + \sum_i c_i.$$

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

$$\sum_i c_i \;=\; \left(\sum_i \mathbf{ac}_i\right) + \Phi(S_0) - \Phi(S_m).$$

This leads to an important fact:

Theorem

If $\Phi(S_0) \le \Phi(S_m)$, then

$$\sum_i c_i \le \sum_i \mathbf{ac}_i\,.$$

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 $\Phi(S_m) \geq \Phi(S_0)$ 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 $n \ge \frac{c}{2}$, 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 $c = 1$ and we increase n by 1, so the potential increases by 1. The amortized cost **ac** so far is therefore 2.

Now we consider the cost of a grow. The actual cost $c = n$, 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

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

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 $c = 1$, and we increase n by 1 which increases the potential by 2. The amortized cost **ac** so far is 3.

Now we analyze the grow method. The actual cost is $c = n$, 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

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

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 $\Phi(S_m) \ge \Phi(S_0)$. It turns out that we do not satisfy Φ never being negative, because when we first initialize the data structure, we have $n = 0, c = 1$, which means that $\Phi(0, 1) = -1$. Luckily this doesn't matter at all because it is still true that $\Phi(S_m) \ge \Phi(S_0)$. 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.

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

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 $\Phi(S_m) \geq \Phi(S_0)$. $\square$

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.

Operations Lets 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 $c/2$.

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

- 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

The doubling and halving algorithm 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.

7.5.1 Cost analysis

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 unfortunately we do not satisfy our usual desired criteria of $\Phi(S_m) \geq \Phi(S_0)$. However, we still have

$$\sum_i c_i = \left(\sum_i \mathbf{ac}_i\right) + \Phi(S_0) - \Phi(S_m) \leq \left(\sum_i \mathbf{ac}_i\right) + 1$$

In other words, we just have an extra 1 cost to account for somewhere. We could modify our potential function to fix this, but a simple solution is to just say that 1 is the amortized cost of initialization. □

Exercises: Amortized Analysis

Problem 13. 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 14. 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 15. Suppose we change `pop()` to shrink when $s = n/2$, show a sequence of operations that incur large amortized cost.

Problem 16. 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.)

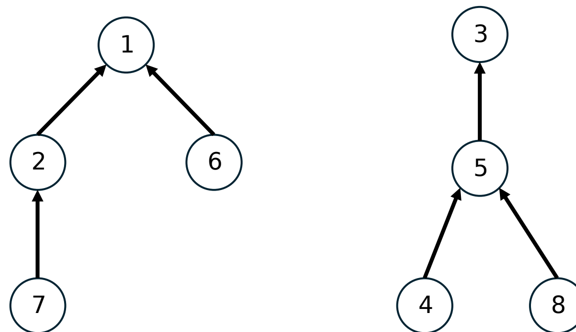
Find(x): Return the representative element of the set containing x.

Union(x, y): x and y are 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*. □

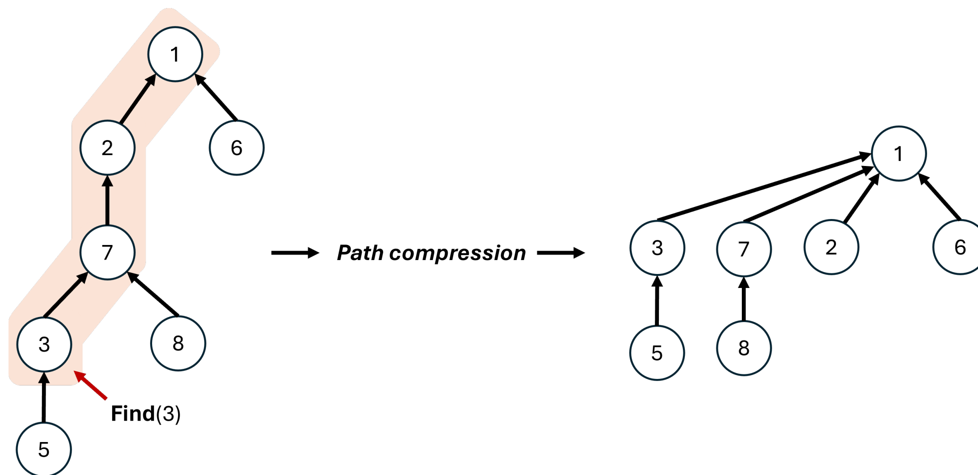
8.4 The path compression optimization

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 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.

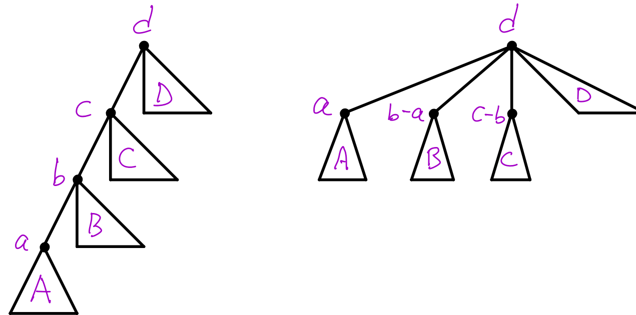
Analysis of Link Consider a Link operation. Suppose the operation links a node y to a node x. The only node whose size changes is x. Let size(x) be the size of x before the Link and size′(x) be the size after the Link. We know that size(x) ≥ 1, because any tree has at least one

node in it. So $\log(\text{size}(x)) \geq 0$. Similarly $\text{size}'(x) \leq n$ 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 $\text{size}'(p)$, is given by $\text{size}'(p) = \text{size}(p) - \text{size}(u)$.

By the definition of heavy, u and p satisfy $\text{size}(u) > \frac{1}{2}\text{size}(p)$, 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

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

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 17. 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 18. 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 19. 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 20. 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 21. 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 (Tree DP)
  - The Longest Increasing Subsequence Problem (SegTree DP)

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 $0 \dots n$, 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 $1 \ldots i$ for $0 \le i \le n$. Combining these two ideas, both the capacity reduction and the subset of items, we define our subproblems as:

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

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 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?

9.4.1 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 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
  results := SegTree(array⟨int⟩(size(a), 0))
  for val, index in sortedByVal do {
    answer := results.RangeMax(0, index) + 1 // Solve the subproblem
    results.Assign(index, answer)            // Store the subproblem
  }
  return results.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

Problem 22. 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 23. 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 24. 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 25. Can you find a greedy algorithm that matches the $O(n \log n)$ performance of the LIS algorithm above?

Problem 26. 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 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.

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 *subset DP* and use it to solve the *traveling salesperson problem*

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

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

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

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

Deriving a recurrence We need to consider two cases. For the pair u, v, either the shortest path using the intermediate vertices $\{1, 2, \ldots, k\}$ 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

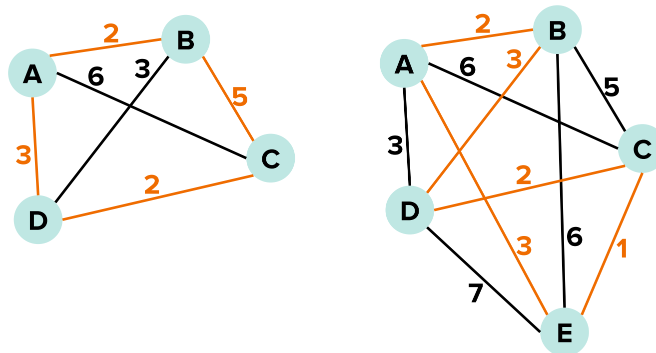
Sometimes our subproblems might not all be necessary to the solve the problem, so if we can eliminate many of them, we will either speed up the algorithm or reduce the amount of space it requires.

10.2 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.[1] 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:

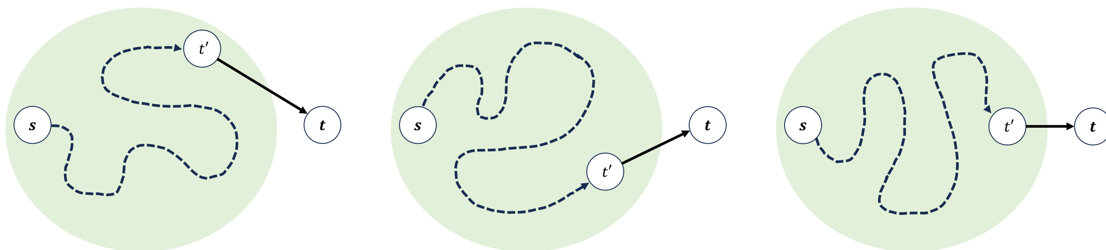


[1]Note that under this definition, it doesn't matter which vertex we select as the start. The *Traveling Salesperson Path Problem* is the same thing but does not require returning to the start. Both problems are NP-hard.

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 (wait, is that reasonable?).

Remark: Efficiently representing the set

How would we actually represent the sets used in the dynamic programming subproblems? Since each set has up to n elements, does each set require $\Theta(n)$ space to store and lookup? That would make the runtime worse by a factor of $\Theta(n)$.

Luckily, there's a trick to work around this! Since our algorithm has to store exponentially many such subsets, specifically 2^{n-1} of them, we have to assume that we have enough space to actually store them. In the word RAM model, we assume that we can index memory up to 2^w words (i.e., we have to be able to write the indices in w bits), which means that to even store this many subproblems, we are forced to assume that $2^{n-1} \le 2^w$, or $n - 1 \le w$. This means that n can not be very large, and that we can actually write all of the integers from 0 to 2^{n-1} in a **single word each**. This enables us to represent the subsets as **bitmasks**. Specifically, we represent each subset as single word-size integer such that the ith bit of the integer is 1 if and only if the ith vertex is in the subset. This lets us represent each subset in $\Theta(1)$ space!

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 II

Problem 27. 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.

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:

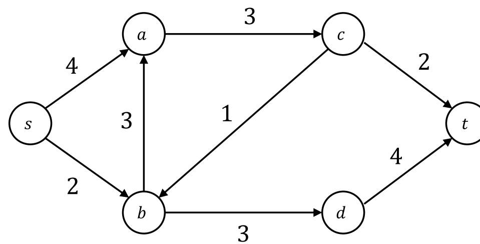
- Define the maximum network flow problem
- Derive and analyze the Ford-Fulkerson algorithm for maximum network flow
- Prove the famous *max-flow min-cut theorem*
- See how to solve the bipartite matching problem by reducing it to a maximum flow problem

Recommended study resources

- CLRS, *Introduction to Algorithms*, Chapter 26, Maximum Flow
- DPV, *Algorithms*, Chapter 7.2, Flows in Networks
- Erikson, *Algorithms*, Chapter 10, Maximum Flows & Minimum Cuts

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 $0 \leq f(e) \leq c(e)$.

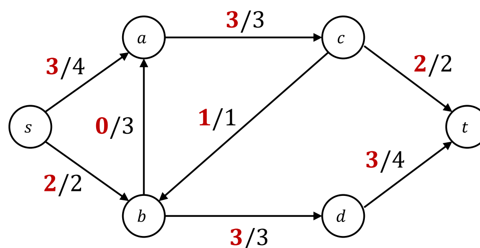
Flow conservation: For any vertex $v \notin \{s, t\}$, flow in equals flow out:

$$\sum_u f(u, v) = \sum_u f(v, u).$$

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

$$|f| = \sum_{u \in V} f(s, u) - \sum_{u \in V} f(u, s)$$

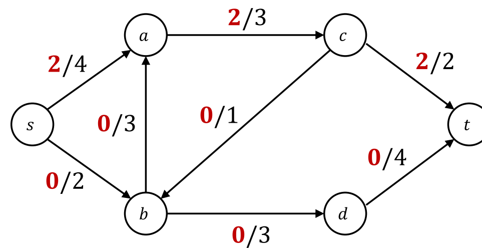
Note that what we are measuring here is the net flow coming out of s. It can be proved, due to conservation, that this is equal to the net flow coming into t. So, in the above graph, what is the maximum flow from s to t? Answer: 5. Using the notation"**flow**/capacity", a maximum flow looks like this. 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.

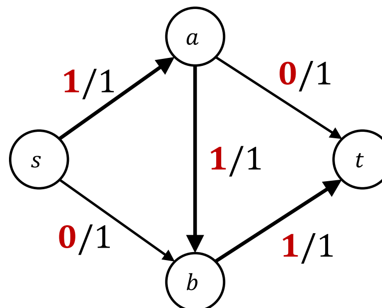
Now to formalize our observation from before, what we noticed is that the value of any s-t flow is less than the value of any s-t cut. Since every flow has a value that is at most the maximum flow, we can argue that in general, the maximum s-t flow ≤ the capacity of the minimum s-t cut, or

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.

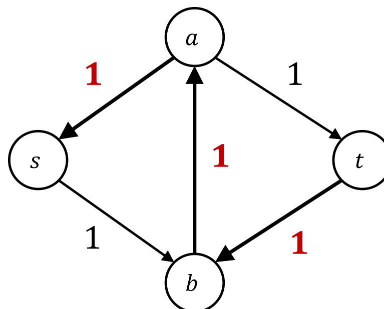
Definition: Residual capacity

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.

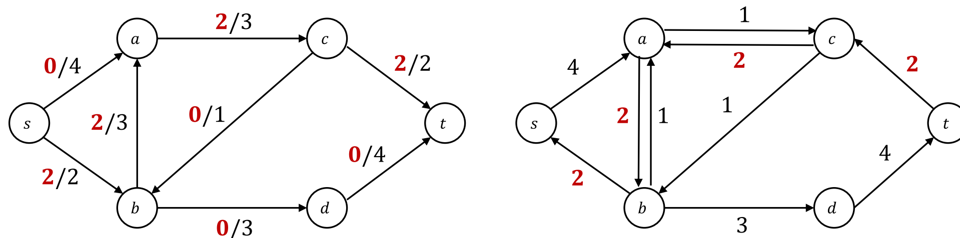
Algorithm: Ford-Fulkerson Algorithm for Maximum Flow

```
while (there exists an s→t path P of positive residual capacity)
push the maximum possible flow along P
```

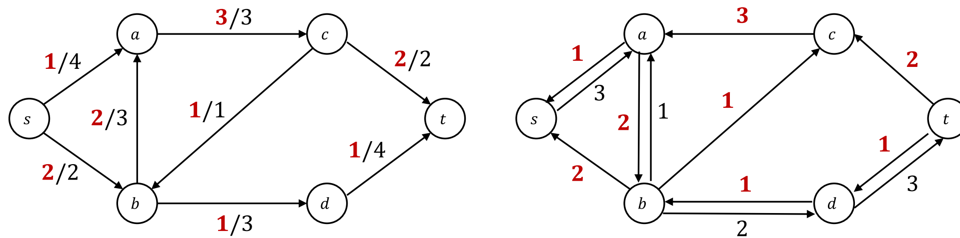
Remark: Handing anti-parallel edges

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.

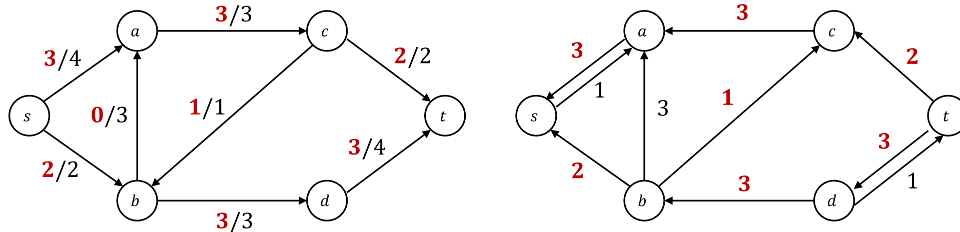
Let's see another example. Consider the graph we started with and suppose we push two units of flow on the path $s \to b \to a \to c \to t$. We then end up with the following flow (left) and residual graph (right).



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

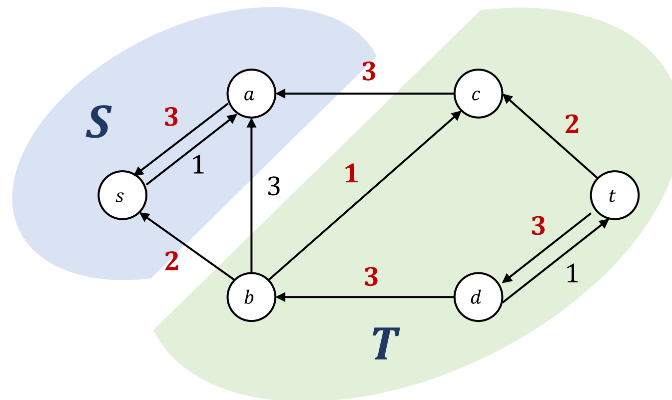
If the given graph G has integer capacities, Ford-Fulkerson terminates in time $O(mF)$ where F is the value of the maximum s-t flow.

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 $c = \text{cap}(S, T)$, the capacity of the cut in the *original* graph. From our earlier discussion, we know that $|f| = f(S, T) \le c$. 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.

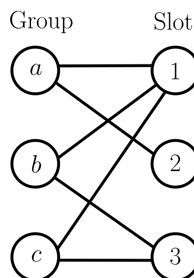
What if the capacities are irrational? In this case Ford-Fulkerson may not terminate. And the solution it converges to (in the limit) may not even be the max-flow! But the maxflow-mincut theorem still holds, even with irrational capacities. There are several ways to prove this; here's one. Suppose not, 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:

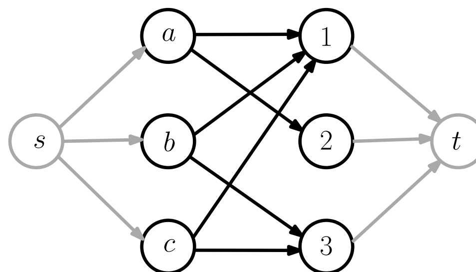


This is an example of a **bipartite graph**: a graph with two sides L and R such that all edges go between L and R. A **matching** is a set of edges with no endpoints in common. What we want here in assigning groups to time slots is a **perfect matching**: a matching that connects every point in L with 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 $F \leq n$, and hence the runtime is $O(nm)$.

Exercises: Flow Fundamentals

Problem 28. 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 29. 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 30. 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.

We will then 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 properties, and is also highly practical since it generalizes the max flow problem.

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

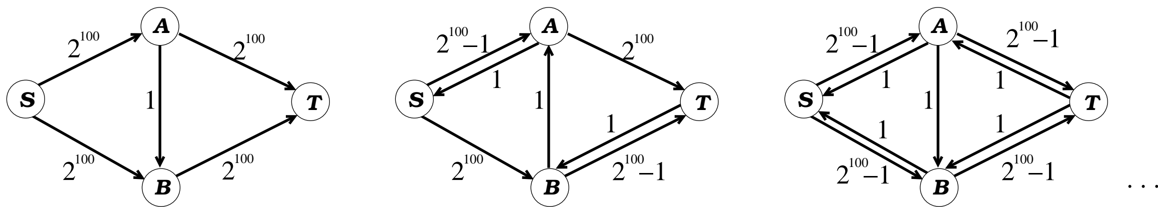
- CLRS, *Introduction to Algorithms*, Chapter 26, Maximum Flow
- DPV, *Algorithms*, Chapter 7.2, Flows in Networks
- Erikson, *Algorithms*, Chapter 10, Maximum Flows & Minimum Cuts

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.

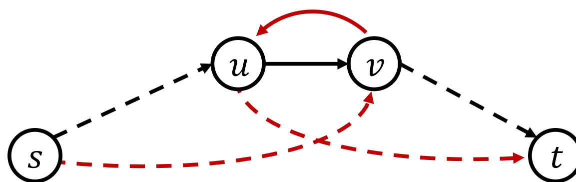
Unsurprisingly, the Shortest Augmenting Paths (Edmonds-Karp) algorithm works by always picking the *shortest* path in the residual graph (the one with the fewest number of edges). By example, we can see that this would in fact find the max flow in the graph above in just two iterations, but what can 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 $v_1, v_2, \dots v_k$. 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 $e = (u, v)$ goes from zero to nonzero flow, $e' = (v, u)$ 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

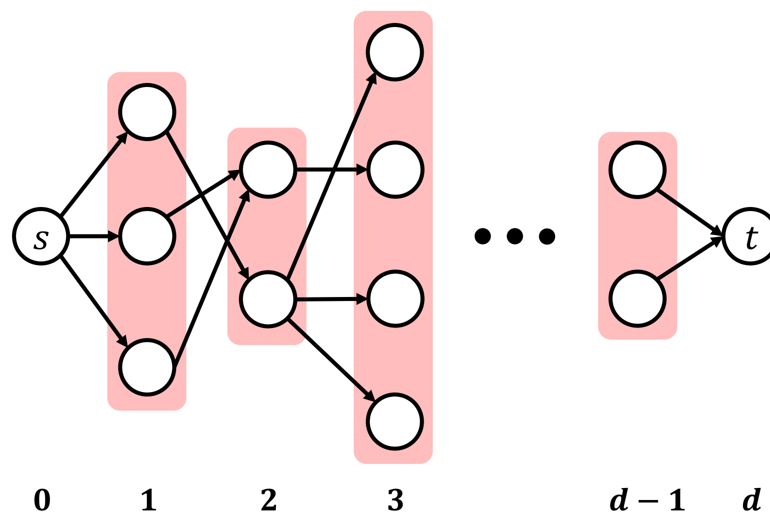
The Edmonds-Karp algorithm already gives us polynomial runtime, but now we will try to do better. The key idea is going to be to eliminate some redundancy in the Edmonds-Karp algorithm. Here are two observations that can guide us towards a better algorithm. Let d be the distance from s to t in the current 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.

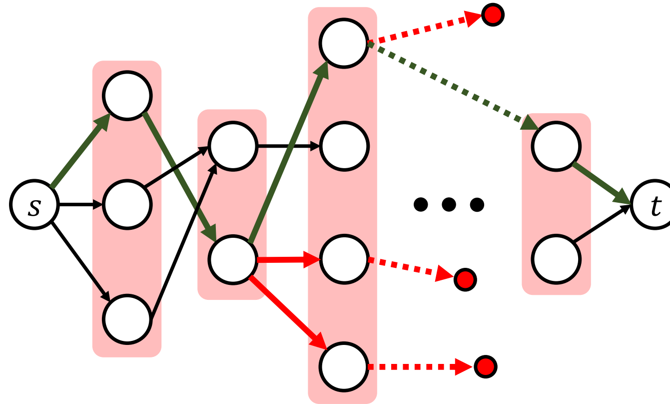
Key Idea: Implicit layered graph

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

$$n + \text{number of new edges marked dead}$$

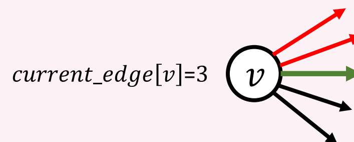
Summing this up over m DFS's, the total running time is

$$\sum_{i=0}^{m}\left(n + \text{number of new edges marked dead in iteration } i\right),$$
$$= nm + \sum_{i=0}^{m}\left(\text{number of new edges marked dead in iteration } i\right),$$
$$= nm + \text{total number of new edges ever marked dead},$$
$$= nm + m,$$
$$= O(nm).$$

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!

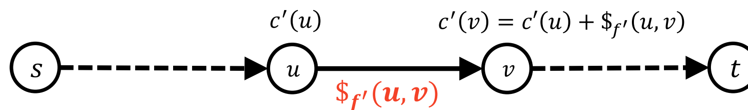
Lemma 13.1: Cheapest path does not decrease

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 $c'(v) = c'(u) + \$_{f'}(u,v)$.

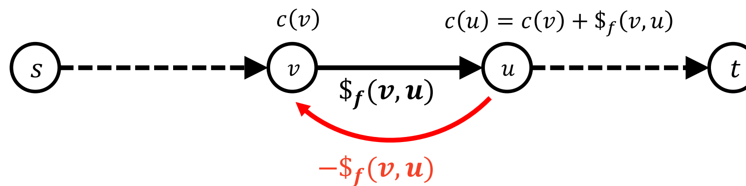


Since v is the earliest such vertex on the path, we know that $c'(u) \geq c(u)$ (it might be greater, not equal, because an edge that was previously on the cheapest path got saturated), so

$$c'(v) \geq c(u) + \$_{f'}(u,v).$$

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

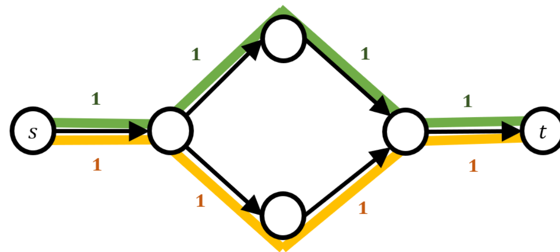
We just argued that the presence of a negative-cost cycle implies that we can find a flow of lesser cost by *adding* two flows together. Adding flows together is defined in the natural way you would think of. We add the values of the flow on each edge to get a new flow. One can show from the definition 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).

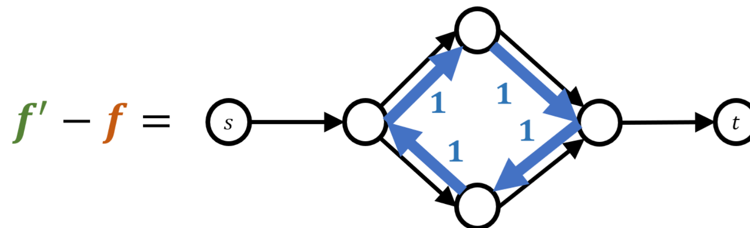
To complete the analysis of minimum cost flows, we will also need the slightly less intuitive notion of the *difference* between two flows. Given a flow f' and a flow f, what would we want $f' - f$ to mean? We could just take the edgewise difference between each of the flows, but this could sometimes 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,

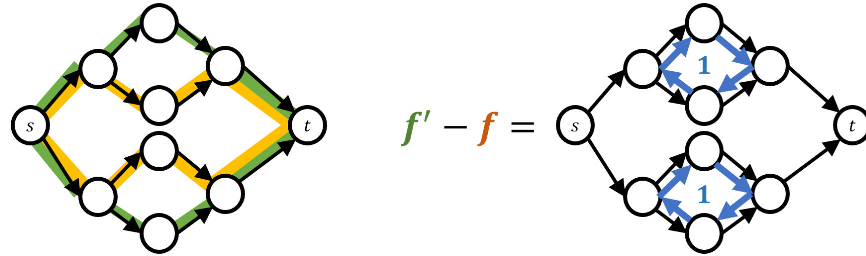
- if $f'(u,v) - f(u,v) < 0$, then we will say that (v,u) has a flow of $f(u,v) - f'(u,v)$.

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 $\vec{e}$ has flow $f(u,v) - f'(u,v) \leq f(u,v)$, which is the definition of the residual capacity of $c_f(\vec{e})$.

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

Lecture 14

Matrix Games

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

Objectives of this lecture

In this lecture, we will:

- Define two-player zero-sum games, and the concept of minimax-optimal strategies.
- Some techniques for solving two-player zero-sum games.
- See the connection of two-player zero-sum games to randomized algorithms.

14.1 Introduction to Game Theory

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

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

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

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

14.2 Definitions and Examples

In a game, we have

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

In this lecture we'll consider a class of games called *matrix games*. These games are powerful enough to model interesting problems, but also amenable to theoretical analysis. These involve just two players, and a *payoff* matrix.

14.2.1 The Shooter-Goalie Game

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

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

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

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

		goalie	
		L	R
shooter	L	$(-1, 1)$	$(1, -1)$
	R	$(1, -1)$	$(-1, 1)$

The rows of the game matrix are labeled with actions for one of the players (in this case the shooter), and the columns with the actions for the other player (in this case the goalie). The

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

entries are pairs of numbers, indicating who wins how much: e.g., the L, L entry contains $(-1, 1)$, the first entry is the payoff to the row player (shooter), the second entry the payoff to the column player (goalie). In general, the payoff is (r, c) where r is the payoff to the row player, and c the payoff to the column player.

In this case, note that for each entry (r, c) in this matrix, the sum $r + c = 0$. Such a game is called a **zero-sum game**. The zero-sum-ness captures the fact that the game is the amount that one player wins is the amount that the other player loses. (Matrix games that are not zero-sum are discussed in section 14.6 of these notes.)

Note that being zero-sum does not mean that the game is "fair" in any sense—a game where the payoff matrix has $(1, -1)$ in all entries is also zero-sum, but is clearly unfair to the column player.

Lastly, for 2-player games, we will define the *row-payoff matrix* R which consists of the payoffs to the row player, and the *column-payoff matrix* C consisting of the payoffs to the column player. The tuples in the payoff matrix M contain the same information, i.e.,

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

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

		goalie L	**goalie** R
shooter	L	−1	1
	R	1	−1

Note that the row payoff matrix R has all of the information about the game, since we can deduce the column player's payoff by taking the negative of the row player's payoff. We will generally omit the C values when writing a matrix for a zero-sum game.

14.2.2 Pure and Mixed Strategies

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

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

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

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

Definition: Pure strategy

A pure strategy for a player is one in which the player deterministically selects a single action to always play, e.g., always shoot left.

But as will become obvious vary soon, pure strategies in these games are not what we play. We are trying to compete with the worst adversary, and we may like to "hedge our bets". Hence we may use a randomized algorithm: e.g., the players dive/shoot left or right with some probability, or when playing the classic game of Rock-Paper-Scissors the player choose one of the options with some probability. This is called a *mixed strategy*

Definition: Mixed strategy

A mixed strategy for a player consists of a non-negative real probability p_i for each action such that $\sum_i p_i = 1$, i.e., a probability distribution over the actions for the player.

When talking about a pair of mixed strategies (one for the row player and one for the column player), we will usually write p_i for each row, such that $\sum_i p_i = 1$ for the row player, and similarly, a $q_i \geq 0$ for each column, such that $\sum_i q_i = 1$ for the column player. The probability distributions $\mathbf{p}, \mathbf{q}$ are called the *mixed strategies* for the row and column player respectively. And then we look at the *expected payoff*

Claim: Expected payoff

The expected payoff to the *row player* is

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

and the expected payoff to the *column player* is

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

This being a two-player zero-sum game, we know that $V_R(\mathbf{p}, \mathbf{q}) = -V_C(\mathbf{p}, \mathbf{q})$, so we will just mention the payoff to one of the players (say the row player). For instance, if $\mathbf{p} = (0.5, 0.5)$ and $\mathbf{q} = (0.5, 0.5)$ in the shooter-goalie game, then $V_R = 0$, whereas $\mathbf{p} = (0.75, 0.25)$ and $\mathbf{q} = (0.6, 0.4)$ gives $V_R = 0.45 - 0.55 = -0.1$.

14.2.3 Minimax-Optimal Strategies

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

Definition: Lower bound for the row player

Consider a strategy $\mathbf{p}$ for the row player. After picking $\mathbf{p}$ we then allow the column player to pick $\mathbf{q}$, the best strategy for the column player who knows the row player is playing $\mathbf{p}$. So we write:

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

This is the minimum amount that the row player is guaranteed to achieve using strategy $\mathbf{p}$. We can then define:

$$\mathsf{lb}^* := \max_{\mathbf{p}} \mathsf{lb}(\mathbf{p})$$

which is the highest possible lower bound over all row strategies.

Make sure you parse this correctly:

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

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

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

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

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

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

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

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

So the payoff in this case to the row player is

Definition: Upper bound for the row player

Consider a strategy **q** for the column player. After picking **q** we then allow the row player to pick **p**, the best strategy for the row player who knows the column player is playing **q**. So we write:

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

This is the maximum amount that the row player can achieve, given that the column player is using strategy **q**. We can then define:

$$\mathsf{ub}^* := \min_{\mathbf{q}} \mathsf{ub}(\mathbf{q})$$

which is the lowest possible upper bound over all column strategies.

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

Claim 14.1: Lower bounds are below upper bounds

For any **p** and any **q** we have:

$$\mathsf{lb}(\mathbf{p}) \quad \leq \quad \mathsf{ub}(\mathbf{q})$$

Proof. The left hand side is an amount that the row player can achieve. The right hand side is an amount that the row player cannot exceed. The result follows. □

This is a very useful approach, as we'll see later. In order to actually evaluate a game we'll find two strategies **p** and **q** such that $\mathsf{lb}(\mathbf{p}) = \mathsf{ub}(\mathbf{q})$, which will be the value of the game.

Now we can make a powerful observation which will make it much easier to actually evaluate these lower and upper bounds.

Claim 14.2: A pure response is optimal

To evaluate the $\mathsf{lb}(\mathbf{p})$ we can assume that the column player plays a pure strategy. That is:

$$\mathsf{lb}(\mathbf{p}) \;=\; \min_{\mathbf{q}} V_R(\mathbf{p},\mathbf{q}) \;=\; \min_{j} \sum_{i} p_i R_{i,j}$$

Proof. Once the row player fixes a mixed strategy **p**, the column player then has no reason to randomize: her payoffs will be some average of the payoffs from playing the individual columns, so she can just pick the best column for her.

More fomally we have:

$$\begin{aligned} \mathsf{lb}(\mathbf{p}) &= \min_{\mathbf{q}} V_R(\mathbf{p},\mathbf{q}) = \min_{\mathbf{q}} \sum_{i,j} p_i q_j R_{i,j} \\ &= \min_{\mathbf{q}} \sum_j q_j \left(\sum_i p_i R_{i,j} \right) = \min_j \sum_i p_i R_{i,j} \end{aligned}$$

The last equation follows because the term in parentheses is just a function of j. It is the value of column j to the row player. So the optimum choice for $\mathbf{q}$ is to have $q_j = 1$ for the index j where formula in parentheses is minimum, and zero for other indices. □

Corollary: Alternate definition for upper and lower bounds

The quantity lb^* can be equivalently defined as

$$\mathsf{lb}^* = \max_{\mathbf{p}} \min_j \sum_i p_i R_{ij}.$$

Similarly ub^* can be written as

$$\mathsf{ub}^* = \min_{\mathbf{q}} \max_i \sum_j q_j R_{ij}.$$

The Shooter-Goalie Game Example

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

Example: Lower and upper bounds for the shooter-goalie game

Row Player: For the row player (shooter), suppose $\mathbf{p} = (p_1, p_2)$ is the mixed strategy. Note that $p_1 \geq 0, p_2 \geq 0$ and $p_1 + p_2 = 1$. So it is easier to write the strategy as $\mathbf{p} = (p, 1-p)$ with $p \in [0,1]$.

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

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

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

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

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

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

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

Column Player: The calculation for the column player (goalie) is similar in this case. We let q be the probability of the goalie going left, and $1-q$ the probability of going right.

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

We take the minimum of this for $q \in [0,1]$. This is achieved at $q = 1/2$, and the value is 0, so $\mathsf{ub}^* = 0$.

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

An Asymmetric Goalie Example

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

		goalie L	**goalie** R
shooter	L	$-\frac{1}{2}$	1
shooter	R	1	-1

Example: Asymmetric shooter-goalie game

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

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

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

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

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

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

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

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

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

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

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

which will again give $(4/7, 3/7)$ and guarantees that the expected loss is never more than $\mathsf{ub}^* = 1/7$.

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

Problem 31. What if both players have somewhat different weaknesses? What if the payoffs are:

$$\begin{bmatrix} -\frac{1}{2} & \frac{3}{4} \\ 1 & -\frac{3}{2} \end{bmatrix}$$

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

Problem 32. For the game with payoffs:

$$\begin{bmatrix} -\frac{1}{2} & \frac{3}{4} \\ 1 & -\frac{2}{3} \end{bmatrix}$$

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

Problem 33. For the game with payoffs:

$$\begin{bmatrix} -\frac{1}{2} & -1 \\ 1 & \frac{2}{3} \end{bmatrix}$$

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

14.3 Von Neumann's Minimax Theorem

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

Theorem 14.1: Minimax Theorem (von Neumann, 1928)

Given a finite 2-player zero-sum game with payoff matrices $R = -C$,

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

This common value is called the value of the game.

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

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

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

14.4 Techniques for Solving Games

If you're confronted with a zero-sum matrix game how do you solve it? Sometimes it's possible by trial and error, and intuition, or luck, to come up with strategies $(\mathbf{p}, \mathbf{q})$ such that $\mathsf{lb}(\mathbf{p}) = \mathsf{ub}(\mathbf{q})$.

In the upcoming lectures on linear programming, we show how to solve games using an LP solver. In the remainder of this section, we give some methods that are often effective at solving small games by hand.

14.4.1 Removing Dominated Rows or Columns

Suppose there's a row that is dominated by another row, that is, every element of row j is at least the corresponding element of row k. Then we can just delete row k from the game, and its value stays the same. For example:

$$\begin{bmatrix} 2 & 3 & 1 \\ 3 & 3 & 5 \\ 7 & 1 & 8 \end{bmatrix}$$

We can immediately delete the first row, because the 2nd one dominates it.

The same idea applies to columns. So, after deleting the first row of the above matrix, look at the resulting matrix. Now it's clear that the first column. is always better for the column player than the 3rd column. So the 3rd column can be deleted.

14.4.2 Convex Combinations of Rows or Columns

Consider the following game:

$$\begin{bmatrix} 10 & 0 \\ 0 & 10 \\ 3 & 3 \end{bmatrix}$$

In this case, no one row dominates another. However adding $\frac{1}{2}$ of the first row to $\frac{1}{2}$ of the second row is equivalent to having $(5\ \ 5)$ as a row. This dominates the last row $(3\ \ 3)$, so we can just delete it. Any solution that puts any weight on the $(3\ \ 3)$ row would be better off putting half that weight on the $(10\ \ 0)$ and the other half on the $(0\ \ 10)$ row.

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

In general, a row can be removed it is dominated by a convex combination of the other rows.

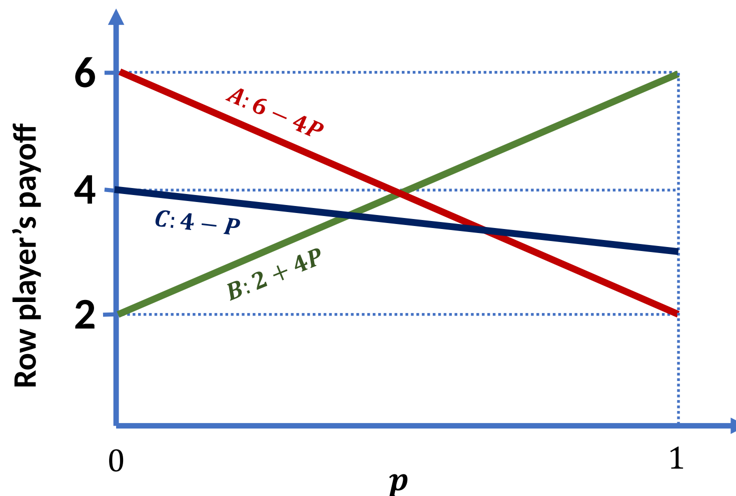
14.4.3 A General Method for Solving 2-Row Games

In this section we give a general method for solving games with two rows (or, by symmetry, two columns). We'll develop the method with the following example:

		column player		
		A	**B**	**C**
row	**1**	2	6	3
player	**2**	6	2	4

The game we'll analyze is shown in the 2×3 matrix above with values for the row player. We'll call the three options for the column player A, B, and C, and the two options for the row player 1 and 2.

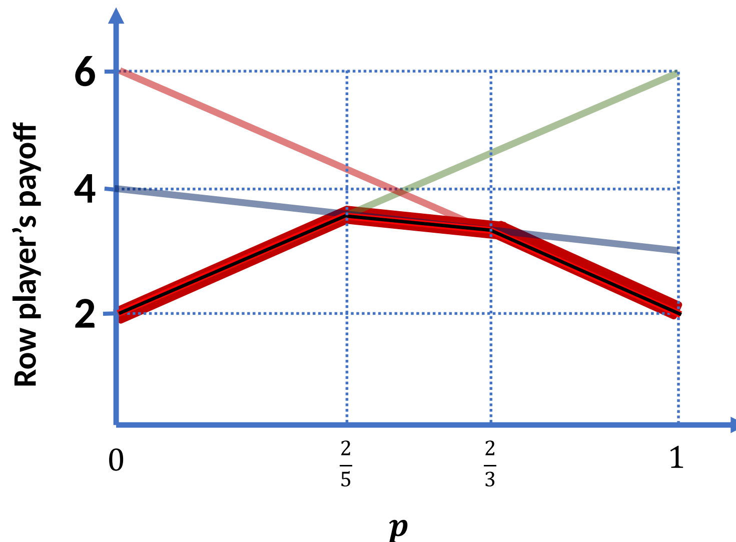
The general mixed strategy for the row player is to choose option 1 with probability p and option 2 with probability $1-p$. For any choice of p, the column player can respond with A, B, or C. We can plot the payoff to the row player with respect to every possible choice of p and use this to analyze the game.



In the diagram above, the horizontal axis represents p, and the vertical axis represents the payoff to the row player. The three lines in the diagram correspond to the three options for the column player. For example, the line labeled B, which is the graph of the function $2+4p$, is the expected value of the game for the row player (as a function of p) if the column player chooses column B.

So what does the actual payoff consist of for the row player? Whatever value of p they pick, their opponent, if playing optimally, will play whichever of A, B and C gives the lowest payoff, so the row player's payoff at any point looks like the *lowest of the three lines*. We can represent

this on the diagram as follows. Its called the *lower envelope* of the three options for the column player.



This concave function represents the expected payoff for the row player if he chooses option 1 with probability p, assuming that the column player plays optimally knowing the value of p. Thus, it represents a lower bound on the value of the game for the row player for each value of p. By inspection of this graph, the concave function achieves its maximum value at the point of intersection between B and C. This point is $p = \frac{2}{5}$, and with that choice of p the value of the game is $3 + \frac{3}{5}$. That's the lower bound on the game's value.

What's a good strategy for the column player? Consider the convex combination of B and C chosen such that the result is a horizontal line. This is $\frac{4}{5}C + \frac{1}{5}B$, and corresponds to a mixed strategy of picking B with probability $1/5$ and C with probability $4/5$. With this mixed strategy, the value of the game for the row player is $3 + \frac{3}{5}$, no matter what the row player does. So this is an upper bound on the value of the game for the row player. Since the lower and upper bound are equal, we know that this is the value of the game.

The same technique can be applied to any game with two rows. The value of the game is obtained by constructing the concave function and finding where it achieves its maximum.

14.5 Lower Bounds for Randomized Algorithms

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

- The columns are all *deterministic* algorithms for sorting n elements.

- The rows are all the possible inputs (all $n!$ of them).

- The entry R_{ij} is the cost of the algorithm j on the input i (say the number of comparisons).

This may be a huge matrix, but we'll never actually write it down; it's just a conceptual guide which we will use to reason about lower bounds. It may be huge, but for it to be a valid matrix it does have to be finite. Is that the case here?

Remark: There are a finite number of (sensible) algorithms

Remember that in the comparison model, every deterministic algorithm can be encoded as a *decision tree* which encodes the comparisons made by the algorithm. Although there are technically an infinite number of possible algorithms and hence decision trees for the problem, there are only a finite number of them that never perform any redundant comparisons. This is because after performing $\binom{n}{2}$ comparisons, one has compared every pair of elements and definitely knows the answer, so the decision trees can have depth at most $\binom{n}{2}$.

Since this is finite, the number of possible decision tree configurations is finite. Lastly, since every algorithm that performs redundant comparisons only has a strictly worse cost than one that does not, we can ignore these algorithms when thinking about lower bounds and efficient algorithms.

So, we know that the matrix is finite and hence valid to interpret as a zero-sum game. What does it tell us? A lot, as it turns out.

- A deterministic algorithm with good worst-case guarantee is a column that does well against all rows: all entries in this column are small.
- A randomized algorithm with good expected guarantee is a probability distribution $\mathbf{q}$ over columns, such that the expected cost for each row i is small. This is a mixed strategy for the column player. It gives an upper bound.

This second claim is actually subtle and not at all obvious. Why can we say that a randomized algorithm is just a distribution over deterministic algorithms?

Remark: Randomized algorithms ≡ randomly choosing a deterministic algorithm

A randomized algorithm is just a deterministic algorithm that may also read from a source of randomness and base some of its decisions on that. If we imagine pre-generating the results of all randomness in the algorithm and "hardcode" it in, the algorithm just becomes a deterministic algorithm! So, given the distribution of the source of randomness, we can infer a distribution on resulting (deterministic) algorithms.

So, a randomized algorithm is essentially a *mixed strategy* $\mathbf{q}$ for the column player in this zero-sum game! How would we find the *best* randomized algorithm? That would correspond to an upper bound strategy $\mathbf{q}^*$ (which would be the minimax-optimal strategy)!

What is a lower bound for randomized algorithms? It is a mixed-strategy over rows (a probability distribution $\mathbf{p}$ over the inputs) such that for every column (i.e., deterministic algorithm j), the expected cost of j (under distribution $\mathbf{p}$) is high.

Key Idea: Lower bounds

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

14.5.1 A Lower Bound for Sorting Algorithms

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

Theorem

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

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

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

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

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

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

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

14.6 General-Sum Two-Player Games

Optional content — Not required knowledge for the exams

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

		Bob	
		L	R
Alice	L	$(1,1)$	$(-1,-1)$
	R	$(-1,-1)$	$(1,1)$

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

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

		Bob	
		C	D
Alice	C	$(8,2)$	$(0,0)$
	D	$(0,0)$	$(2,8)$

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

		Bob	
		C	D
Alice	C	$(2,2)$	$(-1,3)$
	D	$(3,-1)$	$(0,0)$

14.6.1 Nash Equilibria

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

$$\text{row player's new payoff } = \sum_{ij} p_i q_j^* R_{ij} \le \sum_{ij} p_i^* q_j^* R_{ij} = \text{ row player's old payoff}$$

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

$$\text{column player's new payoff } = \sum_{ij} p_i^* q_j C_{ij} \le \sum_{ij} p_i^* q_j^* C_{ij} = \text{ column player's old payoff.}$$

Here are some examples of Nash equilibria:

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

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

Theorem 14.2: Existence of Stable Strategies

Every finite player game (with each player having a finite number of strategies) has at least one (mixed-strategy) Nash equilibrium.

This theorem implies the Minimax Theorem (Theorem 14.1) as a corollary.

Proof. Take any two-player zero-sum game. Let $(\mathbf{p}^*, \mathbf{q}^*)$ be a Nash equiblibrum. We have:

$$\mathsf{lb}(\mathbf{p}^*) = \min_{\mathbf{q}} V_R(\mathbf{p}^*, \mathbf{q}) = V_R(\mathbf{p}^*, \mathbf{q}^*)$$

The latter equality follows from the fact that there is no better option for the column player than $\mathbf{q}^*$.

Similarly we have:

$$\mathsf{ub}(\mathbf{q}^*) = \max_{\mathbf{p}} V_R(\mathbf{p}, \mathbf{q}^*) = V_R(\mathbf{p}^*, \mathbf{q}^*)$$

It immediately follows that $\mathsf{lb}(\mathbf{p}^*) = \mathsf{ub}(\mathbf{q}^*)$, which proves the minimax theorem. □

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

Lecture 15

Linear Programming I

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

Objectives of this lecture

In this lecture, we will cover

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

15.1 Introduction

In recent lectures we have looked at the following problems:

- Maximum bipartite matching
- Maximum flow (more general than bipartite matching).
- Min-Cost Max-flow (even more general than plain max flow).

Today, we'll look at something even more general that we can solve algorithmically: **linear programming**. Linear Programming is important because it is so expressive: many, *many* problems can be coded up as linear programs (LPs). This especially includes problems of allocating resources and business supply-chain applications. In business schools and Operations Research departments there are entire courses devoted to linear programming. There are also commercial software packages charging tens of thousands of dollars per license for solving linear programs (okay they also solve more general problems too, but linear programming is the basis for most of them)! Today we will mostly say what they are and give examples of encoding problems as LPs. We will only say a tiny bit about algorithms for solving them.

15.2 Definition of Linear Programming

Formally, a linear programming problem is specified as follows.

Definition: Linear program

Given:

- n **real-valued** *variables* $x_1, \ldots, x_n$.
- A linear *objective function.* e.g., $2x_1 + 3x_2 + x_3$.
- m *linear inequalities* in these variables (equalities are OK too).

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

Goal:

- Find values for the x_i's that satisfy the constraints and maximize or minimize the objective function.

Remark: No strict inequalities

Linear programs **can not** contain strict inequalities, e.g., $x_1 < 3$ or $x_2 > 5$. Why? Suppose we wrote maximize x_1 such that $x_1 < 3$, then there does not exist an optimal solution.

Remark: Variables are real-valued

This is super important to remember!! Linear program variables take on real values, i.e., the variables can not be guaranteed to take integer values!

An LP may also come without an objective function, in which case we simply wish to find any value for the x_i's that satisfy all of the constraints. Such an LP is called a "feasibility problem". You can think of this as a special case of a linear program where the objective value is just a constant (e.g., zero). We can write either minimization problems or maximization problems.

A set of x_i's that satisfies all of the constraints is called a *feasible solution.* Not all linear programs have a solution; it may be impossible to find any x_i's that satisfy the constraints. Alternatively, it may be the case that there is no optimal objective value because there exists feasible solutions of arbitrarily high value. We can classify all LPs this way into three categories.

Definition: Classification of LPs

Every LP falls into one of three categories:

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

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

15.3 Modeling problems as Linear Programs

15.3.1 An Operations Research Problem

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

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

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

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

Variables: x_1, x_2, x_3, x_4, where x_i denotes the number of cars at plant i.
Objective: maximize $x_1 + x_2 + x_3 + x_4$.
Constraints:

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

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

15.3.2 Modeling Maximum Flow

We can model the max flow problem as a linear program too. In fact, when we wrote down the definitions of the max flow problem, like the capacity and conservation constraints, and defined the net s-t flow, we pretty much did write down a linear program already!

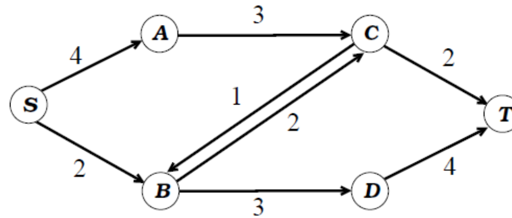
Variables: Set up a variable f_{uv} for each edge (u, v), representing $f(u, v)$

Objective: maximize $\sum_{u \in V} f_{su} - \sum_{u \in V} f_{us}$. (the net s-t flow)

Constraints:

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

For instance, consider this example:



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

Capacity	**Conservation**
$0 \leq f_{sa} \leq 4$	$f_{sa} = f_{ac}$
$0 \leq f_{ac} \leq 3$	$f_{sb} + f_{cb} = f_{bc} + f_{bd}$
$0 \leq f_{ct} \leq 2$	$f_{ac} + f_{bc} = f_{cb} + f_{ct}$
$0 \leq f_{sb} \leq 2$	$f_{bd} = f_{dt}$
$0 \leq f_{bd} \leq 3$	
$0 \leq f_{cb} \leq 1$	
$0 \leq f_{bc} \leq 2$	
$0 \leq f_{dt} \leq 4$	

15.3.3 Modeling Minimum-cost Max Flow

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

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

Approach #1: Maximize flow first One simple way to do this is to first solve for the maximum flow f, ignoring costs. Then, set up a new linear program and add the constraint that flow must equal f, i.e.

$$\sum_{u \in V} f_{su} - \sum_{u \in V} f_{us} = f$$

(plus the original capacity and flow conservation constraints), then set the objective to minimize the cost

$$\text{minimize} \sum_{(u,v)\in E} \$(u,v) f_{uv}$$

Note that we have used a minimization objective function rather than a maximization this time. This is allowed. If you want to stick strictly to maximization problems, then you can maximize the negative of the cost instead, as this will give you the same solution.

Approach #2: Combine the two objectives The reason that min-cost max-flow is annoying to write as an LP is that it kind of has two objectives, to maximize flow then to minimize cost, and we can't directly encode two objectives into an LP. However, in *some* cases, it is possible to combine two objectives into one (but not always).

To simplify the presentation, lets add two new variables, one representing the net flow and another representing the total cost:

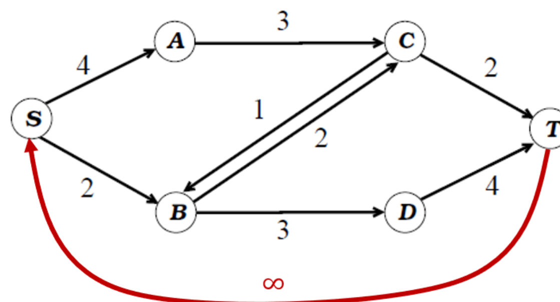
$$f = \sum_{u\in V} f_{su} - \sum_{u\in V} f_{us},$$
$$C = \sum_{(u,v)\in E} \$(u,v) f_{uv}$$

How can we represent the idea of maximizing the flow while tiebreaking by costs using these variables? One way is to multiply the cost by a very small constant ε, small enough that it is definitely less than the difference between the values of any two feasible flows. Then, we can use it to tiebreak since it will only ever matter if two feasible flows have the same value! So our objective function becomes:

$$\text{maximize } f - \varepsilon C$$

Approach #3: Reduce to "Minimum-cost circulation" As our last idea, instead of trying to encode the problem with two objective functions, we can instead reduce the problem to a similar one that only has one objective, and model that problem as a linear program.

To do so, lets forget about "producing" flow at s and "consuming" it at t and imagine that instead we take all of the flow into t and "circulate" it back to s, forming a big cycle of flow instead.



A *circulation* is the same thing as a flow, except that we now require that s and t also satisfy the conservation constraint. In this problem, there is no such concept as "net-flow" anymore, because the net flow should always be zero, so we can't directly ask about maximum flow. However, if we have costs on the edges, then we can write an LP for a *minimum-cost circulation*, which finds a circulation of minimum possible cost.

Variables: Set up a variable f_{uv} for each edge (u, v), representing $f(u, v)$

Objective: minimize $\sum_{(u,v)\in E} \$(u, v) f_{uv}$

Constraints:

- For all edges (u, v), $0 \le f_{uv} \le c(u, v)$. (capacity constraints)
- For **all** v, $\sum_{u\in V} f_{uv} = \sum_{u\in V} f_{vu}$. (flow conservation, **includes s and t!**)

To reduce minimum-cost max flow to a minimum-cost circulation problem, we draw the graph as above where we connect an infinite-capacity edge from t to s, and then give that edge a very negative cost. By giving (t, s) a very negative cost, the solution is encouraged to send as much flow as possible along (t, s), which we can observe is equivalent to maximizing the s-t flow! There are other variants of the minimum-cost circulation problem but we won't cover those.

15.3.4 2-Player Zero-Sum Games

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

	column player		
row player	20	-10	5
	5	10	-10
	-5	0	10

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

Our goal is to maximize the worst case (minimum), over all columns our opponent can play, of our expected gain. This is a little confusing because we are maximizing a minimum. However, we can use a trick: we will add one new variable v (representing the minimum), put in *constraints* that our expected gain has to be at least v for every column, and then define our objective to be to maximize v. Assume our input is given as an array m where m_{ij} represents the payoff to the row player when the row player plays i and the column player plays j. Putting this all together we have:

Variables: $p_1, \ldots, p_n$ and v.
Objective: Maximize v.
Constraints:

- $p_i \geq 0$ for all $1 \leq i \leq n$,
- $\sum_{i=1}^{n} p_i = 1.$ (the p_i form a probability distribution)
- $\sum_{i=1}^{n} p_i m_{ij} \geq v$ for all columns $1 \leq j \leq m$

Exercises: Linear Programming Fundamentals

Problem 34. Our second approach to modeling minimum-cost maximum flow as an LP was to introduce a small ε constant and use it to make the maximum flows be tiebroken by costs. What would be a suitable value for ε that would guarantee that this method works?

Problem 35. Suppose you have an algorithm that can tell you whether an LP is feasible, but does not give you the optimal objective value. Describe a simple method for determining the optimal objective value using the feasibility algorithm as a black box.

Lecture 16

Linear Programming Duality

In this lecture we discuss the general notion of Linear Programming *Duality*, a powerful tool that can allow us to solve some linear programs easier, gain theoretical insights into the properties of a linear program, and has many more applications that we might see later in the course. We will show how duality connects to some topics we have already seen, like minimax optimal strategies in zero-sum games.

Objectives of this lecture

In this lecture, we will

- Motivate and define the idea of the dual of a linear program
- See how to convert a linear program into its dual program
- Learn some powerful theorems that tell us about the behavior of a linear program and its dual
- See how duality can teach us about minimax optimal strategies for zero-sum games

16.1 Linear Programming Terminology and Notation

It's going to be useful as we discuss more properties of LPs, as well as for the concept of duality, to introduce some terminology and notation for linear programs.

Definition: Standard Form

A linear program (LP) over n variables $x_1, \ldots, x_n$, with m constraints written in the following form is said to be in *Standard Form*:

$$\begin{aligned} \text{maximize } & c_1 x_1 + \ldots + c_n x_n, \\ \text{subject to } & a_{11} x_1 + \ldots + a_{1n} x_n \le b_1, \\ & a_{21} x_1 + \ldots + a_{2n} x_n \le b_2, \\ & \vdots \\ & a_{m1} x_1 + \ldots + a_{mn} x_n \le b_m, \\ & x_i \ge 0 \text{ for all } i. \end{aligned}$$

In other words, the objective is maximization, the constraints are all $f(x) \leq b$ for a constant b and the variables are required to be non-negative.

Remark: Standard form in matrix notation

Another way to write this in a much more compact form is to package the c and b values into vectors and the a values into a matrix so we can write:

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

Here there are n non-negative variables $x_1, x_1, \ldots, x_n$, and m linear constraints encapsulated in the $m \times n$ matrix A and the $m \times 1$ matrix (vector) b. Note that in standard form, the non-negativity constraints $x_i \geq 0$ **are not** counted towards the number of constraints m. The objective function to be maximized is represented by the $n \times 1$ matrix (vector) c.

Standard form turns out to be convenient and useful, especially when we will discuss duality. For it to be useful, we should convince ourselves that we can actually always use it!

Claim: Any LP can be written in standard form

Given an LP not in standard form, we can write an equivalent LP in standard form.

- **Converting min to max** Suppose we have a minimization LP. Then we can negate the objective function and make it a maximization problem instead.
- **Handling equality constraints** Suppose we have linear equalities in our LP.
  - We can replace LHS = RHS with two inequalities: LHS ≤ RHS and LHS ≥ RHS
  - We negate the second inequality to get -LSH ≤ -RHS. Now we have only ≤ inequalities
- **Handling unbounded variables** If we have a variable x_i which can take on any real value, we can replace it by two variables, x_i^+ and x_i^-, and substitute x_i with $x_i^+ - x_i^-$ everywhere. The LP is equivalent, but we can constrain $x_i^+ \geq 0$ and $x_i^- \geq 0$.

16.2 The Dual Program

16.2.1 A motivating example – the carpenter

Suppose you are a humble carpenter; you spend your days making tables, chairs, and shelves, all out of wood, nails, and paint. Each item you make requires a specific amount of each of the three materials and can be sold at the market for a specific price. Given the amount of material you have, you would like to determine the best way to use them to make the most money.

Item	Wood	Nails	Paint	Sale Price
Table	8	20	5	$50
Chair	4	15	3	$30
Shelf	3	5	3	$20
Stock	100	300	80	

Ignoring rounding issues, since we don't want to deal with integrality, we can write this problem as a linear program. This leads us to the following. Lets define the variables x, y, z to be the number of tables, chairs, and shelves, respectively, that we should make.

$$\begin{aligned} \text{maximize } & 50x + 30y + 20z \\ \text{subject to } & 8x + 4y + 3z \leq 100 \\ & 20x + 15y + 5z \leq 300 \\ & 5x + 3y + 3z \leq 80 \\ & x, y, z \geq 0 \end{aligned}$$

If we use a computer program to solve this, we will find that an optimal solution is $x \approx 1.82$, $y \approx 14.55$, $z \approx 9.09$, with an objective value of $709.09.

Along comes a merchant A merchant approaches you, the humble carpenter, with the prospect of buying your materials, i.e., all of your wood, nails, and paint. You consider selling them at a fair price, but what should that be? You'd like to make sure that you sell them for at least as much as you could make if you turned them into furniture, but you know that the merchant will try to get the lowest price from you, so you'll settle for that.

If we denote by w, s, p, the price of wood, nails, and paint, respectively, we can model this problem as another LP as follows:

$$\begin{aligned} \text{minimize } & 100w + 300s + 80p \\ \text{subject to } & 8w + 20s + 5p \geq 50 \\ & 4w + 15s + 3p \geq 30 \\ & 3w + 5s + 3p \geq 20 \\ & w, s, p \geq 0 \end{aligned}$$

This LP has an optimal solution of $w \approx 2.73$, $s \approx 0.73$, $p \approx 2.73$, and an objective value of $709.09. What a coincidence! The objective value is the same as the previous LP. Perhaps that should not

be so surprising... we were not willing to sell the materials for less than we could turn them into items, so we would expect it to be *at least as much*, but the fact that they are *exactly equal* is not quite so obvious in advance...

The structure of this pair of LPs is very special, and if we look closely at them we will see that they are made up of the same exact ingredients, just laid out a little differently. Since the first LP is in standard form, we can write it in matrix form with

$$A = \begin{bmatrix} 8 & 4 & 3 \\ 20 & 15 & 5 \\ 5 & 3 & 3 \end{bmatrix}, \qquad \mathbf{b} = \begin{bmatrix} 100 \\ 300 \\ 80 \end{bmatrix}, \qquad \mathbf{c} = \begin{bmatrix} 50 \\ 30 \\ 20 \end{bmatrix}, \qquad \mathbf{x} = \begin{bmatrix} x \\ y \\ z \end{bmatrix}$$

This way, the LP can be written as

$$\begin{aligned} \text{maximize } & \mathbf{c}^T \mathbf{x} \\ \text{subject to } & A\mathbf{x} \leq \mathbf{b} \\ & \mathbf{x} \geq \mathbf{0}. \end{aligned} \tag{16.1}$$

Looking closely at the pricing LP, we see that if we define the variable vector $\mathbf{y} = [w \;\; s \;\; p]^T$, then it can be written as

$$\begin{aligned} \text{minimize } & \mathbf{b}^T \mathbf{y} \\ \text{subject to } & A^T \mathbf{y} \geq \mathbf{c} \\ & \mathbf{y} \geq \mathbf{0}. \end{aligned} \tag{16.2}$$

We happened to derive this particular LP by intuition, but in fact, given *any LP in standard form*, one could apply this transformation to obtain this second program. It turns out to be a wildly useful and powerful concept, so it has a name – its called the dual program!

16.3 A General Formulation of the Dual

Definition 16.1: The dual of a linear program

The dual of the standard form LP (16.1) is

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

The original standard form LP (16.1) is referred to as the *primal problem*.

And if you take the dual of (16.2) , what do you think you will get back? You'll get (16.1). *The dual of the dual is the primal.* Because of this, which program we refer to as the primal and which we refer to as the dual is just a matter of convention, it is completely symmetric. Think about how you would actually take the dual of the dual as an exercise. Since the dual as written is not in standard form, it would need to first be converted to standard form.

16.3.1 The Theorems

Our intuitive derivation of the dual program as a pricing problem for the carpenter implied that the value of the dual solutions should be at least as large as the profit the carpenter could make from turning the materials into furniture, i.e., it should always give at least as large of a value as the primal problem.

We can formally prove that it indeed always does just that. This fact is called *weak duality*.

Theorem 16.1: Weak Duality

If $\mathbf{x}$ is a feasible solution to the primal (16.1) and $\mathbf{y}$ is a feasible solution to the dual (16.2) then

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

Proof. This follows by applying the constraints of the primal and dual LPs in (16.1) and (16.2) and the fact that $\mathbf{x} \ge 0$ and $\mathbf{y} \ge 0$. Since $A^T\mathbf{y} \ge \mathbf{c}$, we can plug this into the objective $\mathbf{c}^T\mathbf{x}$ and get

$$\mathbf{c}^T\mathbf{x} \le (A^T y)^T\mathbf{x} = (\mathbf{y}^T A)\mathbf{x}$$

Now we can move the brackets (associativity), and use the fact that $A\mathbf{x} \le \mathbf{b}$, to get

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

□

The amazing (and deep) result here is to show that the dual actually gives not just an upper bound on the primal, but, assuming some mild conditions, it perfectly equals the primal!

Theorem 16.2: Strong Duality Theorem

Suppose the primal LP (16.1) is feasible (i.e., it has at least one solution) and bounded (i.e., the optimal value is not ∞). Then the dual LP (16.2) is also feasible and bounded. Moreover, if $\mathbf{x}^*$ is the optimal primal solution, and $\mathbf{y}^*$ is the optimal dual solution, then

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

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

We will not prove Theorem 16.2 in this course, since the proof is a bit long, though it isn't too difficult (feel free to look it up if interested). Why is this useful? If I wanted to prove to you that $\mathbf{x}^*$ was an optimal solution to the primal, I could give you the solution $\mathbf{y}^*$, and you could check that $\mathbf{x}^*$ was feasible for the primal, $\mathbf{y}^*$ feasible for the dual, and they have equal objective function values.

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

16.3.2 Using duality to determine feasibility and boundedness

In addition to helping us bound feasible solutions to our LPs, duality can also be used as a tool to determine when certain programs are feasible or infeasible, or perhaps show that they are bounded or unbounded.

- If the primal is feasible and bounded, strong duality says the dual is also feasible and bounded.
- Suppose the primal (maximization) problem is unbounded. What can duality tell us? Weak duality says $\mathbf{c}^T x \le \mathbf{b}^T y$... If there existed any feasible $\mathbf{y}$ for the dual, this would imply that the primal is bounded, and hence by the contrapositive, if the primal is unbounded, then the dual *must be infeasible*.
- By the exact same logic (reversed), if the dual is unbounded, since the primal is a lower bound on the dual, the primal must be infeasible.
- Can both the primal and dual be unbounded? No, because as the two previous points show, if one of them is unbounded, then the other is infeasible, and if a program is infeasible, it certainly can not be unbounded.

We can use these facts to represent all of the possible situations in a table like so:

		Dual		
		Inf	**F&B**	**Unb**
Primal	**Inf**	✓	X	✓
	F&B	X	✓	X
	Unb	✓	X	X

Here, **Inf** means infeasible, **F&B** means feasible and bounded, and **Unb** means unbounded. The only scenario that duality does not cover for us is the top-left cell. You can figure that out as an exercise.

Remark: Usefulness

This table has some very useful implications. If we have an LP for some problem, we might want to prove conditions on when it is feasible or infeasible. Directly proving that the LP is infeasible might be too difficult. Instead, if we can write the dual program and give a proof that the dual is unbounded, then we have indirectly proven that the primal is infeasible! A useful trick.

16.4 Example: Zero-Sum Games

Consider a 2-player zero-sum game defined by an n-by-m payoff matrix R for the row player. To simplify things a bit, let's assume that all entries in R are positive (this is without loss of generality since as pre-processing we can always translate values by a constant and this will

just change the game's value to the row player by that constant). The *lower bound* for the row player was defined to be

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

We can solve for the lower bound by writing it as an LP.

Variables: $p_1, \ldots, p_n$ and v.
Objective: Maximize v.
Constraints:

- $p_i \geq 0$ for all $1 \leq i \leq n$,
- $\sum_{i=1}^{n} p_i = 1.$ (the p_i form a probability distribution)
- $\sum_{i=1}^{n} p_i R_{ij} \geq v$ for all columns $1 \leq j \leq m$

To apply our techniques for duality, we need to put it in standard form. To do so, we can do the following:

- we can replace $\sum_i p_i = 1$ with $\sum_i p_i \leq 1$ since we said that all entries in R are positive, so the maximum will occur with $\sum_i p_i = 1$,
- since all entries in R are positive, we can also safely add in the constraint $v \geq 0$,
- we can also rewrite the third set of constraints as $v - \sum_i p_i R_{ij} \leq 0$.

This then gives us an LP in the form of (16.1) with

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

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

We can now write the dual, following (16.2). Let $\mathbf{y}^T = (q_1, q_2, \ldots, q_m, v')$. We now are asking to minimize $\mathbf{b}^T\mathbf{y}$ subject to $A^T\mathbf{y} \geq \mathbf{c}$ and $\mathbf{y} \geq \mathbf{0}$. Writing out the objective function, we get $\mathbf{b} \cdot \mathbf{y} = [0\ 0\ \cdots\ 0\ 1] \cdot [q_1\ q_2\ \cdots\ q_m\ v'] = v'$, so the objective is just minimize v'. If we transpose A, we get

$$A^T = \left[\begin{array}{ccc|c} 1 & \cdots & 1 & 0 \\ \hline & & & 1 \\ & -R & & \vdots \\ & & & 1 \end{array}\right]$$

Now we can write out the constraints, we have

1. $q_1 + \ldots + q_m \geq 1$,

2. $-q_1 R_{i1} - q_2 R_{i2} - \ldots - q_m R_{im} + q_{m+1} \geq 0$ for all rows i,

Since the objective is to minimize, and all R entries are positive, we can make a similar argument to the primal case that $q_1 + \ldots + q_m = 1$, i.e., this constraint must be tight because increasing any of the q values would only further increase the objective. Some algebra turns the second constraint into $q_1 R_{i1} + q_2 R_{i2} + \ldots + q_m R_{im} \leq v'$ for all rows i, so we obtain the LP:

Variables: $q_1, \ldots, q_m$ and v'.
Objective: Minimize v'.
Constraints:
- $q_i \geq 0$ for all $1 \leq i \leq m$,
- $\sum_{i=1}^{m} q_i = 1$.
- $\sum_{j=1}^{m} q_j R_{ij} \leq v'$ for all rows $1 \leq i \leq n$

This LP look an awful lot similar to the primal LP, which was computing lb^* for the row player. What is this LP saying? We can interpret v' as being the value of the game to the row player once again, and $q_1, \ldots, q_m$ as the randomized strategy of the *column player* this time, and we want to find a randomized strategy for the column player that minimizes v' subject to the constraint that the row player gets *at most* v' no matter what row he plays. In other words, we've just found an LP for the *upper bound* ub^* to the row player!

Notice that the fact that the maximum value of v in the primal is equal to the minimum value of v' in the dual follows from strong duality. Therefore, the minimax theorem is a corollary to the strong duality theorem!

Corollary 16.1: Minimax Theorem

Given a finite 2-player zero-sum game with payoff matrices $R = -C$,

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

This common value is called the value of the game.

Proof. Follows from strong duality and the argument above, where we showed that the dual problem to computing lb^* is a linear program that computes ub^*. □

16.5 The Geometric Intuition for Strong Duality

Optional content — Not required knowledge for the exams

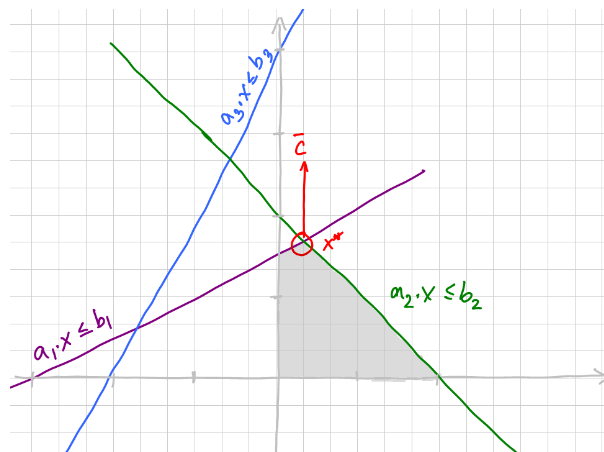
To give a geometric view of the strong duality theorem, consider an LP of the following form:

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

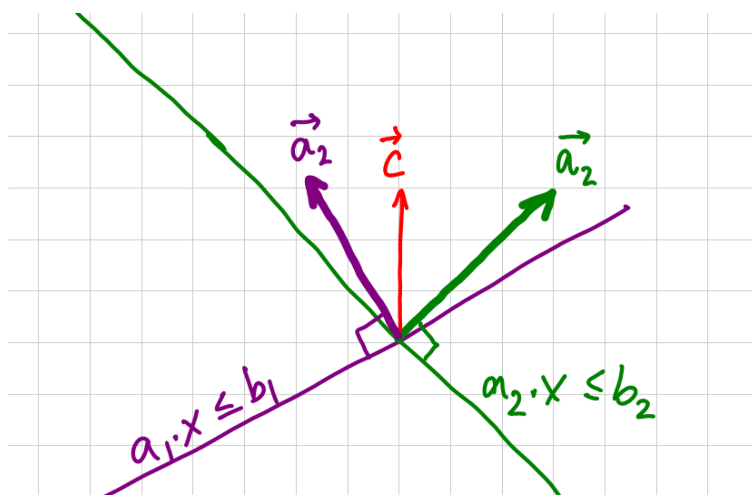
For concreteness, let's take the following 2-dimensional LP:

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

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



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



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

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

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

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

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

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

The last inequality follows because

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

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

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

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

Exercises: Linear Programming Duality

Problem 36. Using the definitions we gave, show that the dual of the dual is the primal problem.

Problem 37. Find an LP that is infeasible such that its dual is also infeasible.

Problem 38. (Hard - optional) Prove the min-cut max-flow theorem using strong duality.

Lecture 17

Linear Programming: Polytopes and Integrality

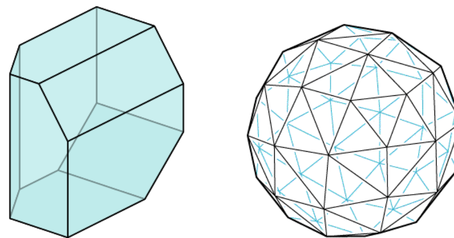
Objectives of this lecture

In this lecture, we will

- Discuss properties of convex polytopes as they relate to linear programming
- See how to prove that particular polytopes have integer vertices, which implies that their corresponding linear programs have optimal integer solutions.

17.1 Polytopes, Vertices, and the Simplex Algorithm

It's sometimes helpful to visualize linear programs geometrically. The set of points that satisfy a linear inequality in n variables is all the points on, or on one-side of a "plane" in $\mathbb{R}^n$. This set of points is called a *half-space*. The points satisfying all the inequalities of an LP is therefore the intersection of a finite number of half-spaces. Such a set of points is called a *convex polytope*. (Figures from Wiktionary, and Geometry Junkyard.)



Given two points a and b in $\mathbb{R}^n$, we can define the *convex combination* of a and b as follows:

$$\text{Conv}(a,b) = \{\alpha a + (1-\alpha)b \mid 0 \leq \alpha \leq 1\}$$

$\text{Conv}(a,b)$ is simply the all the points on the straight line in $\mathbb{R}^n$ between points a and b.

A set of points S in $\mathbb{R}^d$ is *convex* iff for all $a, b \in S$ we have that $\text{Conv}(a,b) \subseteq S$. Note that the intersection of two convex sets is convex, because if a and b are in both of the convex sets, then the convex combination is also in both of the sets. It follows that the polytope of an LP is a convex set.

Polytopes have many other interesting properties. They can be decomposed as a collection of interrelated *facets* of dimensions varying from 0 to n. (In three dimensions these are vertices, edges, faces, and volumes.) We're not going to discuss that rich theory here except to talk about the *vertices* of a polytope, that is, the facets of dimension 0.

In a linear program, we have a linear objective function maximize $c^T x$ over points x inside such a convex polytope. Our first important definition is that of a **vertex** of a polytope.

Definition 17.1: Vertex of a convex polytope

A **vertex** of a convex polytope is a point in the polytope that is the unique maximum for some objective function.

Intuitively, a vertex is the furthest point in the polytope for some direction. Visually, vertices are the *corner points* of the polytope. This is the optimization way of looking at the problem. We can also think about the problem from the point of view of convex geometry with the related notion of **extreme points**.

Definition 17.2: Extreme points of a convex polytope

A point q is an **extreme point** of a convex polytope P if

- $q \in P$
- For any $v \in \mathbb{R}^d$ with $v \neq 0$ then at least one of $q + v$ or $q - v$ is not in P.

Put another way, if you're at a point for which there exist two opposite directions such that you can move in these directions and still stay inside the polytope, then you're *not* at a vertex. Another equivalent definition is that an extreme point is a point that can not be written as the average of any two other points in the polytope (or even more generally, can not be written as a convex combination of any other points in the polytope).

You may have guessed this intuitively already, but **vertices** and **extreme points** are actually the same thing! They are just two different points of view, the optimization point of view (the maximizer for some objective), and the convex geometry point of view (a point that is not a convex combination of any others). This is very useful for analyzing polytopes because we can use whichever is most convenient for a particular problem and switch between them at will.

Vertices/extreme points are important because any LP that has an optimal solution and has at least one vertex **has an optimum solution that is on a vertex**. The intuition for this is that if you're at a non-vertex and you can move in directions v and $-v$, then moving in at least one of these directions will not cause the objective function to not increase. So you move in that direction as far as you can go. When you stop you must be entering a facet of a lower dimension. This process is then repeated until you reach a vertex.

Furthermore, the simplex algorithm always moves from vertex to vertex, and therefore the solution it finds is a vertex of the convex polytope. This is important because if we can prove that all the vertices of a polytope have a particular property, then we know that the solution output by the simplex algorithm necessarily has that property!

17.2 Matchings

Given an undirected graph $G = (V, E)$, a *matching* is a set M of edges of G such that no two edges of M share a vertex. Typically we're interested in finding the matching of largest cardinality. We've already seen how to use maximum flow to solve the matching problem in bipartite graphs. It's instructive to see how LP can be used for the matching problem.

For each edge $\{u, v\} \in E$ we create a variable $x_{\{u,v\}}$. And we assert the following inequalities:

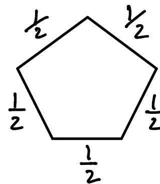
$$\begin{aligned} 0 \le x_{\{u,v\}} \le 1 \qquad & \forall \{u, v\} \in E \\ \sum_{v:\{u,v\}\in E} x_{\{u,v\}} \le 1 \qquad & \forall u \in V \end{aligned} \tag{17.1}$$

The objective function is:

$$\max \sum_{\{u,v\}\in E} x_{\{u,v\}}$$

It's easy to see that any matching in G can be used to give a solution to this LP (set $x_{\{u,v\}} = 1$ for edges in the matching, and zero otherwise). In other words, if Z_G is the maximum objective function value of this LP on a graph G, and M_G is the size of the maximum matching in G then $Z_G \ge M_G$. Another way to phrase this is that the integral solutions are a subset of the space being considered by the LP. We call such a linear program a *relaxation* of the problem.

But the converse is not necessarily true, because the solution to the LP might not be integral. It turns out that when G is bipartite $Z_G = M_G$, but equality does not hold in general graphs. For example, consider the following graph of five vertices and five edges:



The maximum objective function value $Z_G = 2.5$ (as shown here) but the maximum matching is of size $M_G = 2$. However, we can prove that in special cases, we **will** be able to guarantee that we can find integer solutions to certain LPs. We do so by **analyzing the vertices of the constraint polytope**.

17.2.1 Vertices of the Matching Polytope

Given a graph $G = (V, E)$, denote by $\mathrm{MP}_G \subseteq \mathbb{R}^{|E|}$ the polytope of the matching LP given by inequalities (Equation (17.2)).

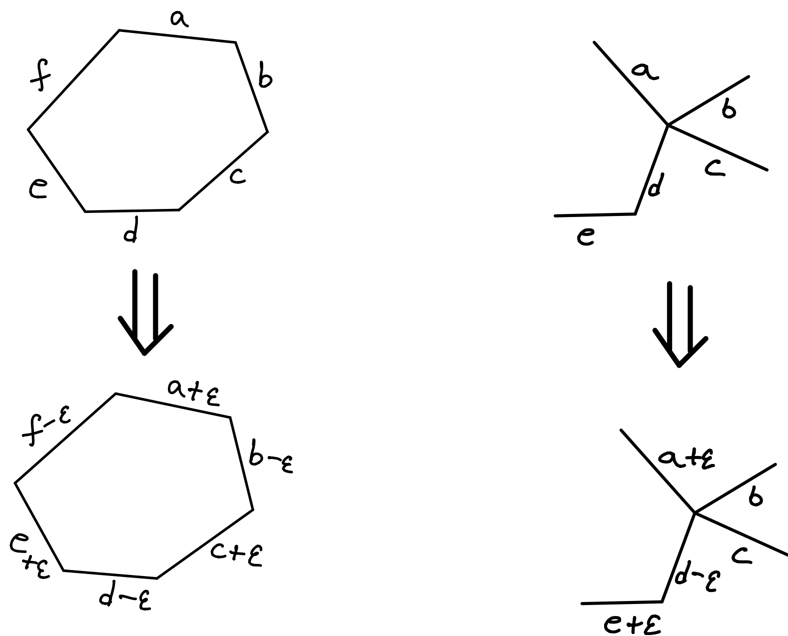
Theorem 17.1

Let G be a bipartite graph, and MP_G be its matching polytope. If q is an extreme point of MP_G, then q has integer coordinates.

Proof. We will prove the contrapositive. Suppose q is a point in MP_G that is non-integral. Then we will show that it is not an extreme point MP_G.

So we have a point q in MP_G some of whose variables are strictly between zero and one. Consider the subgraph of G (called G') consisting only of those edges whose values are fractional (strictly between 0 and 1).

There are two cases. Suppose G' has a cycle. Because G' is bipartite, the cycle is of even length. Each edge of the cycle has a variable whose value is strictly between 0 and 1. Let ϵ be a number which is so small that when added to or subtracted from any one of the variables on the cycle, its value remains in the closed interval $[0, 1]$. The left side of the figure below illustrates this.



We can now alternately add and subtract ϵ from the values around the cycle. This preserves the sum of the variables at every node of G'. Also notice that we could have swapped + with − in every case with the same result. This gives us a vector v such that the point $q + v$ and the point $q - v$ are both in the polytope.

The second and final case is illustrated in the figure above on the right. Suppose there is no cycle in G'. Then G' is a tree. We find a path from a leaf to another leaf. We can apply the same technique of alternately adding and subtracting ϵ along this path. One difference is that the sum total of the variables impinging on the starting and ending nodes do change. But since the leaf nodes of G' have only one variable strictly between zero and one impinging on them, we can increase it or decrease it without violating the constraint for that node.

The result is again a direction v such that we can move in direction v and direction $-v$ and stay inside the polytope. By definition q is therefore not an extreme point. □

Notice how the proof crucially depends on there not being an *odd* cycle in G. However, the following theorem holds in the non-bipartite case.

Theorem 17.2

Let G be a graph, and MP_G be its matching polytope. If q is an extreme point of MP_G, then every component of the vector q is in the set $\{0, \frac{1}{2}, 1\}$.

A similar but somewhat more complicated argument can be used to prove this, but we won't.

17.3 An algebraic perspective

When we solve an LP written in standard form (i.e., constraints of the form $Ax \leq b$ and $x \geq 0$), some of the constraints become **tight**, that is, we get, for some of the rows i,

$$(a_i)^T x = b,$$

and for some variables $x_i = 0$. Yet another way to characterize solutions to linear programs is by considering **which subset** of constraints is tight. This leads us to the notion of a **basic solution**.

Definition 17.3: Basic solution

A **basic solution** to a linear program $\{Ax \leq b;\ x \geq 0\}$ is the intersection of n linearly independent tight constraints, i.e., the unique solution to a set of n linearly independent rows of $Ax = b$ or $x = 0$.

Basic solutions are not necessarily feasible because they might violate $x \geq 0$ or violate one of the other constraints in the system. We therefore also have the concept of a basic feasible solution, whose definition is really just its name.

Definition 17.4: Basic feasible solution

A **basic feasible solution** is a basic solution that is feasible.

A really cool and fundamental fact in polyhedral analysis is that this is **yet another equivalent characterization of the vertices of a polytope**! So now we have three angles from which to view vertices, the optimization point of view (vertices), the convex geometry point of view (extreme points), and the linear algebra perspective (basic feasible solutions).

17.3.1 Integrality of basic feasible solutions

Basic feasible solutions are the intersections of hyperplanes, which we can solve for algebraically by solving a set of linearly independent rows of $Ax = b$. That is, we can select a subset R of linearly independent rows and solve $A_R x = b_R$.

To infer a sufficient condition for this to give rise to integral solutions, we can look at **Cramer's Rule**, which says that the solution can be written as

$$x_i = \frac{\det(A_R^i)}{\det(A_R)}$$

where the notation A^i means to replace the ith column of A with the right-hand side vector b_R. If we want this to be an integer for all subsets of rows R, we probably need to enforce that at least all the entries of A and b are integers. Additionally, we need the resulting fraction to be integral, so the easiest way to enforce this would be to require $\det(A_R) = \pm 1$. This property is very critical and as such has a special name.

Definition 17.5: Unimodular matrix

A square integer-valued matrix is called **unimodular** if its determinant is 1 or -1.

Now lastly, to ensure that *every* basic feasible solution is integral, we need this to hold for any choice of rows R, which will be true if we can guarantee that every submatrix that yields a feasible solution is unimodular. This leads us to the final definition.

Definition 17.6: Totally unimodular matrix

A matrix is called **totally unimodular** if every square non-singular submatrix is unimodular. Equivalently, every square submatrix has determinant 1, 0, or -1.

Theorem 17.3: Integrality of totally unimodular linear programs

Every basic feasible solution of a linear program with polytope $\{Ax \leq b;\ x \geq 0\}$ where A is totally unimodular and b is integral is an integral basic feasible solution.

Proof. Cramer's rule shows that for any non-singular basic solution, each x_i will be an integer divided by 1 or -1, which is an integer. □

17.3.2 Integrality of bipartite matching via total unimodularity

Earlier we proved that the bipartite matching polytope is integral by arguing that its extreme points are all integral. We can prove the same fact using total unimodularity as well. Recall that the constraints defining the polytope are

$$\begin{aligned} 0 \leq x_{\{u,v\}} &\leq 1 && \forall \{u,v\} \in E \\ \sum_{v:\{u,v\}\in E} x_{\{u,v\}} &\leq 1 && \forall u \in V \end{aligned} \tag{17.2}$$

Actually we can simply it a little bit. We don't need the first constraint since we will naturally never have $x_{\{u,v\}} > 1$ since this would consequently violate the second constraint. So we can ignore that one since it is redundant.

In the resulting constraints, the matrix A consists of m columns (one for each variable, i.e., one for each edge) and n rows (one for each constraint, i.e., one for each vertex.) The A and b matrices both consist only of integers, so all that remains to show is that A is totally unimodular.

Theorem

The bipartite matching constraint matrix is totally unimodular.

Proof. First observe that every column of the matrix contains exactly two 1s, one for each each vertex for which the edge corresponding to the column is incident on. Furthermore, the rows of the matrix can be divided into two groups, those that correspond to vertices on the left side of the bipartite graph, and those on the right[1]. Now, consider any square submatrix of the matrix and consider the following three cases:

1. **There exists a column containing all 0s**: In this case, the determinant is zero.
2. **There exists a column containing a single 1**: In this case, using *Laplace expansion*[2] (a common formula for computing determinants), the determinant is (± 1) times the determinant of the square submatrix resulting from removing the corresponding row and column. By induction, if all smaller square submatrices are unimodular, the determinant is guaranteed to be $1, -1$, or 0. As a base case, since every 1×1 submatrix has entry 0 or 1, all such matrices are unimodular.
3. **Every column contains two 1s**: We divide the rows into the two groups, those corresponding to vertices on the left of the bipartite graph and those on the right. Since every column has exactly one 1 in the left and one in the right, subtracting every row in the right group from every row in the left group yields a row full of zeros, which indicates that the rows of the matrix are not linearly independent, and therefore their determinant is zero.

□

[1] This property is what makes this proof **not work** for non-bipartite graphs

[2] `https://en.wikipedia.org/wiki/Laplace_expansion` if its been a while since you took linear algebra

Lecture 18
Approximation Algorithms

While we have good algorithms for many optimization problems, the unfortunate reality elucidated by theoretical computer science is that so very many important real-world optimization problems are **NP**-hard. What do we do? Suppose we are given an **NP**-hard problem to solve. Assuming $\mathbf{P} \neq \mathbf{NP}$, we can't hope for a polynomial-time algorithm for these problems. But can we get polynomial-time algorithms that always produce a "pretty good" solution? (a.k.a. *approximation algorithms*)

Objectives of this lecture

In this lecture, we will

- define and motivate **approximation algorithms**
- derive two **greedy algorithms** for scheduling jobs on multiple machines to minimize their makespan (a.k.a. stacking blocks to minimize the height of the tallest stack)
- see how **rounding linear programs** can give us an approximate vertex cover
- show how **scaling** can be used to turn pseudopolynomial-time algorithms into efficient approximation algorithms.

18.1 Introduction

Given an **NP**-hard problem, we don't hope for a fast algorithm that always gets the optimal solution — if we had such a polynomial algorithm, we would be able to use it to solve everything in NP, and that would imply that $\mathbf{P} = \mathbf{NP}$, something we expect is false. But when faced with an **NP**-hard problem, giving up is not the only reasonable solution! There are several ways that we might try to move forward

- First approach: find a polynomial-time algorithm that guarantees to get at least a "pretty good" solution? E.g., can we guarantee to find a solution that's within 10% of optimal? If not that, then how about within a factor of 2 of optimal? Or, anything non-trivial?
- Second approach: Find heuristics that speed up the algorithm for some cases, but still exponential time in the worst case.

Today's lecture focuses on the first idea, to derive polynomial-time *approximation algorithms*.

18.1.1 Formal definition

Definition: Approximation Algorithm

Given some optimization problem with optimal solution value OPT, and an algorithm which produces a feasible solution with value ALG, we say that the algorithm is a c-*approximation algorithm* if ALG is *always* within a factor of c of OPT. The convention differs depending on whether the optimization problem is a minimization or maximization problem.

Minimization An algorithm is a c-approximation ($c > 1$) if for all inputs, $\text{ALG} \leq c \cdot \text{OPT}$.

Maximization An algorithm is a c-approximation ($0 < c < 1$) if for all inputs, $\text{ALG} \geq c \cdot \text{OPT}$.

18.2 Scheduling Jobs on Multiple Machines to Minimize Load

Problem: Scheduling jobs on multiple machines to minimize the makespan

You have m identical machines on which you want to schedule some n jobs. Each job $i \in \{1, 2, \ldots, n\}$ has a processing time $p_i > 0$. You want to partition the jobs among the machines to minimize the load of the most-loaded machine. In other words, if S_j is the set of jobs assigned to machine j, define the *makespan* of the solution to be

$$\max_{1 \leq j \leq m} \left(\sum_{i \in S_j} p_i \right)$$

You want to minimize the makespan of the solution you output.

This is the formal definition of the problem that is usually used in textbooks, but here's a nicer and (subjectively) more intuitive way to describe the problem.

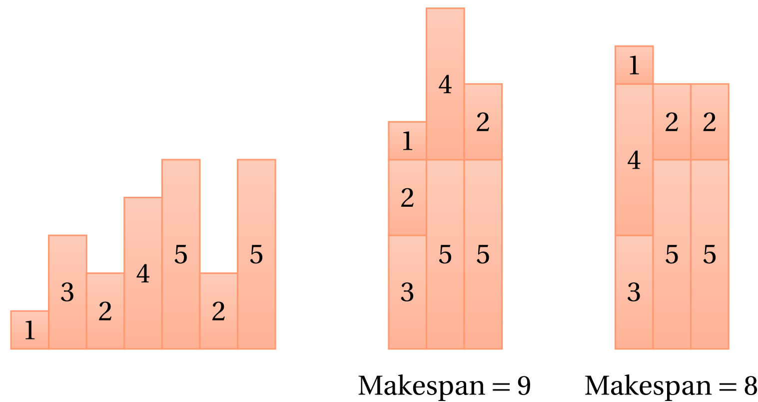
Problem: Stacking blocks to minimize the height

You have n blocks, the ith of which has height p_i. You want to arrange the blocks into m stacks such that the height of the tallest stack is as short as possible.

Observe that these two problems are exactly the same, just with a different story, but the latter (I think) is easier to visualize and think about.

Example Here's an example input to the job scheduling / block stacking problem. Say we have $p = \{1, 3, 2, 4, 5, 2, 5\}$ and $m = 3$. The blocks are shown on the left, and two possible ways

to stack them are shown on the right. The makespan is the height of the tallest stack, which is 9 for the first example, and 8 for the second example.



Makespan = 9 Makespan = 8

The second example turns out to be optimal. Can you think of a proof of why?

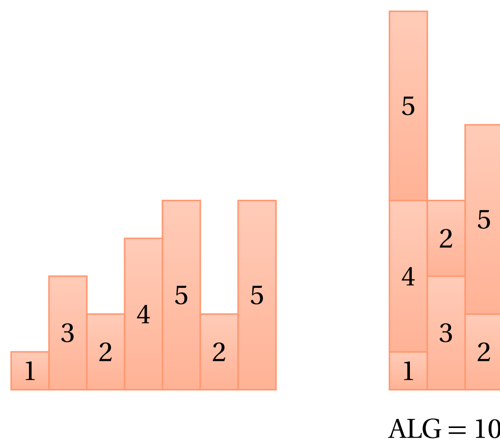
Problem 39. Give a concise argument that a makespan of 8 is optimal for the block stacking example above.

18.2.1 Algorithms for job scheduling / block stacking

Our first approach to the solve the block stacking problem is a *greedy algorithm*. Recall that greedy algorithms are those that just look for some locally best choice and make that at each step, rather than planning ahead in any way. Greedy algorithms are often very good for producing approximations.

> ***Algorithm: Greedy job scheduling / block stacking***
>
> Start with m empty stacks, then, for each block, place it on the current shortest stack.

Applying this to the example above, we would get the following configuration, which has a makespan of 10, which is only 25% more than the optimal, so not too bad.



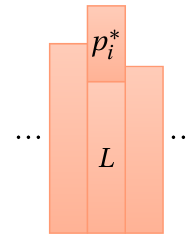
ALG = 10

But that was just one example, how bad can it get in general? We need to prove that this algorithm always gives us something that is pretty good, or near optimal.

Theorem 18.1: Quality of greedy job scheduling

The greedy approach outputs a solution with makespan at most 2 times the optimum, i.e., it is a 2-approximation algorithm.

Proof. Let's start by looking at the height of the tallest stack in our solution, since this is what defines the makespan (the answer). Call the last block added to the tallest stack i^*, so its height is p_{i^*}. Now call the remaining height of the tallest stack L. So we have by definition $\text{ALG} = L + p_{i^*}$. The hardest part of these greedy algorithm proofs is relating the value of ALG to the value of OPT.



$\text{ALG} = p_i^* + L$

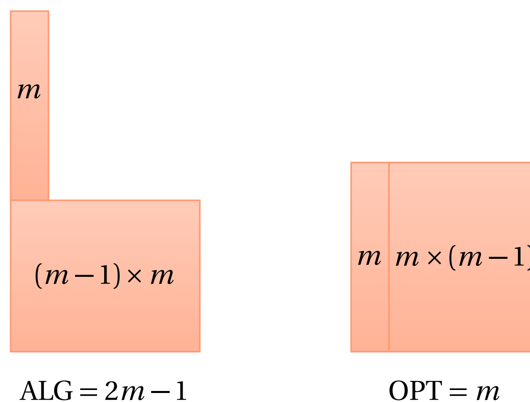
First, note that since every block must be placed somewhere, $\text{OPT} \geq p_i$ for all i and specifically, $\text{OPT} \geq p_{i^*}$. What can we say about L? Remember that the algorithm always chooses the *shortest stack* to place the next block, so when it decided to place i^* on the stack, it was because L was the height of the shortest stack at the time. This means that every stack has height at least L, which means that $\text{OPT} \geq L$.

So, combining these two inequalities, we get

$$\text{ALG} = L + p_{i^*} \leq \text{OPT} + \text{OPT} = 2\text{OPT}$$

18.2.2 A worst-case example

Is this analysis tight? Sadly, yes. Suppose we have $m(m-1)$ jobs of size 1, and 1 job of size m, and we schedule the small jobs before the large jobs. The greedy algorithm will spread the small jobs equally over all the machines, and then the large job will stick out, giving a makespan of $2m-1$, whereas the right thing to do is to spread the small jobs over $m-1$ machines and get a makespan of m.



$\text{ALG} = 2m-1$ $\text{OPT} = m$

The approximation ratio in this case is $\frac{2m-1}{m} = 2(1-\frac{1}{m}) \approx 2$, so this looks almost tight. It can be made tight with a bit more analysis, but 2 is the tightest constant approximation ratio.

Problem 40. Improve the analysis slightly to show that the approximation ratio of the greedy algorithm is actually $2(1-\frac{1}{m})$, which makes the above example tight.

Can we get a better algorithm? The worst-case example helps us see the problem: when small jobs come before big jobs they can cause big problems! So lets prevent this...

18.2.3 An improved greedy algorithm

Algorithm: Sorted greedy job scheduling / block stacking

Sort the blocks from biggest to smallest, then do the greedy algorithm.

This algorithm prevents the worst-case example that we showed before, but what does it do in general? Let's prove that it is in fact an improvement.

Theorem 18.2: Quality of sorted greedy job scheduling

The sorted greedy approach outputs a solution with makespan at most 1.5 times the optimum, i.e., it is a 1.5-approximation algorithm.

Proof. Let's use the same setup as before and say that i^* is the last block added to the tallest stack, and L is the height of the rest of the stack underneath i^*, so the makespan (tallest stack) is $L + p_{i^*}$. We still have the facts that OPT $\geq L$ and OPT $\geq p_{i^*}$. We need to make some new observation in order to get the approximation ratio lower.

First, suppose $L > 0$, which means there are some blocks underneath i^*. Since the blocks were processed in sorted order (**important**), all of the blocks underneath are at least as large. Furthermore, since L was the height of the shortest stack at the time that i^* was added, every other stack is also non-empty and contains blocks that are at least as large as i^*. From this, we can deduce that there exists at least $m+1$ blocks of size at least p_{i^*} because there was at least one in each of the m stacks before we processed i^*. So, by the pigeonhole principle, since there are only m stacks, for any possible configuration, there must always be a stack that contains two blocks of size at least p_{i^*}. Therefore OPT $\geq 2p_{i^*}$, or equivalently $p_{i^*} \leq \frac{1}{2}$OPT.

Combining this with our original inequality, we get

$$\text{ALG} = L + p_{i^*} \leq \text{OPT} + \tfrac{1}{2}\text{OPT} = 1.5\text{OPT}.$$

Now we have an edge case to deal with. What if $L = 0$? Then we can't say that there are any blocks underneath i^* or make any argument about the number of large blocks. In this case, we just have ALG $= p_{i^*}$, but we know that OPT $\geq p_{i^*}$, so actually ALG = OPT, so we get the exact answer in this case. □

So again we ask if this analysis is tight. In fact, this time it isn't. We can further reason about the properties of the sorted algorithm to get a better approximation ratio.

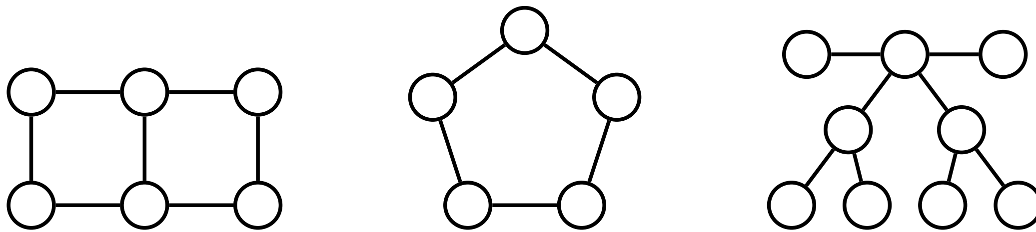
Problem 41. It is possible to show that the makespan of Sorted Greedy is at most $\frac{4}{3}$OPT, and that this approximation ratio is indeed tight. Try it.

18.3 Vertex Cover via LP Rounding

Recall that a *vertex cover* in a graph is a set of vertices such that every edge is incident to (covers) at least one of them. The minimum vertex cover problem is to find the smallest such set of vertices.

> ***Definition: Minimum Vertex Cover***
>
> Given a graph G, find a smallest set of vertices such that every edge is incident to at least one of them.

For instance, this problem is like asking: what is the fewest number of guards we need to place in a museum in order to cover all the corridors.



Problem 42. Find a vertex cover in the graphs above of size 3. Argue that there is no vertex cover of size 2.

18.3.1 A linear-programming-based algorithm

We don't expect to find the optimal solution in polynomial-time, but we will show in this section that we can use *linear programming* to obtain a 2-approximate solution. That is, if the graph G has a vertex cover of size k^*, we can return a vertex cover of size at most $2k^*$. Let's remind ourselves of the LP *relaxation* of the vertex cover problem:

$$\begin{aligned} \text{minimize} \quad & \sum_{v \in V} w_v \\ \text{s.t.} \quad & w_u + w_v \geq 1 && \forall \{u, v\} \in E \\ & w_v \geq 0 && \forall v \in V \end{aligned}$$

Remember that in the integral version of the problem, the variables denote that a vertex v is in the cover if $x_v = 1$ and not in the cover if $x_v = 0$. Solving the integral version is NP-hard, so we settle for a relaxation, where we allow fractional values of x_v, so a vertex can be "half" in the cover. This is called an "LP relaxation" because any true vertex cover is a feasible solution,

but we've made the problem easier by allowing fractional solutions too. So, the value of the optimal solution now will be at least as good as the smallest vertex cover, maybe even better (i.e., smaller), but it just might not be legal any more.

Since, in an actual vertex cover we can not take half of a vertex, our goal is to convert this fractional solution into an actual vertex cover. Here's a natural idea.

Algorithm: Relax-and-round for vertex cover

Solve the LP relaxation for x_v for each $v \in V$, then pick each vertex for which $x_v \geq \frac{1}{2}$

This is called *rounding* the linear program (which literally is what we are doing here by rounding the fraction to the nearest integer — for other problems, the "rounding" step might not be so simple).

Theorem: Relax-and-round is a 2-approximation

The above algorithm is a 2-approximation to VERTEX-COVER.

Proof. We need to prove two things. First, that we actually obtain a valid vertex cover (every edge must be incident to a vertex that we pick) and that the size of the resulting cover is not too large.

Feasibility Suppose for the sake of contradiction that there exists an edge (u, v) that is not covered. Then that means we did not pick either u or v, which means that $x_u < \frac{1}{2}$ and $x_v < \frac{1}{2}$. Therefore, $x_u + x_v < 1$, but this contradicts the LP constraint that $x_u + x_v \geq 1$. So the algorithm always outputs a vertex cover.

Approximation ratio Let LP denote the objective value of the relaxation. Since it is a relaxation, its solution can not be worse than the optimal integral solution (the actual minimum vertex cover), so LP $\leq$ OPT. Furthermore, when we round the solution, in the worst case, we increase variables from $\frac{1}{2}$ to 1, so we double their value. Since the objective is $\sum x_v$, this at most doubles the objective value, so ALG $\leq$ 2LP. Combining these inequalities, we get ALG $\leq$ 2LP $\leq$ 2OPT. □

Again, we ask if this is analysis is tight. It in fact is, and a simple example to show that is the graph with two vertices u and v connected by an edge. If the LP relaxation assigns $x_u = x_v = \frac{1}{2}$, we will pick both vertices for the cover, giving ALG = 2 when OPT = 1.

18.3.2 Hardness of Approximation

Interesting fact: nobody knows any approximation algorithm for vertex cover with approximation ratio 1.99. Best known is $2 - O(1/\sqrt{\log n})$, which is $2 - o(1)$.

There are results showing that a good-enough approximation algorithm will end up showing that **P**=**NP**. Clearly, a 1-approximation would find the exact vertex cover, and show this.

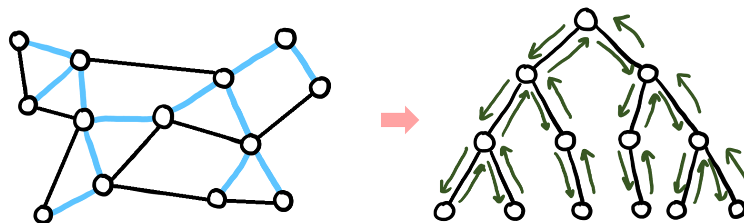
H**r**astad showed that if you get a 7/6-approximation, you would prove **P**=**NP**. This 7/6 was improved to 1.361 by Dinur and Safra. Beating $2-\varepsilon$ has been related to some other problems (it is "unique games hard"), but is not known to be **NP**-hard.

18.4 Metric Traveling Salesperson

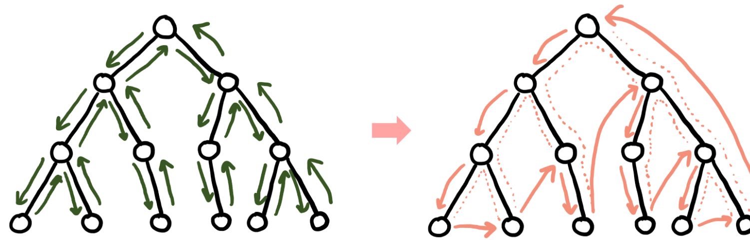
The metric traveling salesperson (METRIC TSP) problem asks you to find the shortest path to visit n cities, each exactly once, returning to where you started. You are given distances d_{ij} between any pair of cities i and j (assume the graph is complete, you have all the distances). The "metric" part of the problem is that these distances are symmetric and obey the triangle inequality. That is, for any pair of vertices u, v and any intermediate vertex k, we always have $d(u,v) \le d(u,k)+d(k,v)$. In other words, its never worse to go somewhere directly rather than via some other vertex.

To solve the problem, we'd like to re-use some previous problems that we already know how to solve and relate their solution to the solution of the TSP problem. To gain some intuition here, remember that the TSP problem is essentially asking for a cycle that visits every vertex of the graph and has minimum possible weight. That sounds quite similar to another problem we know... A problem where we want to pick a bunch of edges that connects the whole graph but minimizes the total weight. It sounds a lot like the minimum spanning tree (MST) problem! The difference is that we now have to build a cycle rather than a tree so we are a bit more constrained. Lets think about how we might pull a cycle out of an MST.

Here's an idea. Root the MST arbitrarily, now just do a depth-first traversal, and record each time we go down or up each edge. This uses each edge in the MST twice, but gives us a clean cycle that starts at the root, visits each vertex (at least) once and returns to the root. Since it uses the edges of the MST, it is hopefully not too expensive. Here's a diagram to illustrate. Say the MST of the graph is in blue, then we lay it out with an arbitrary root on the right and perform the depth-first traversal.



This sounds pretty promising, but there's a technicality we have to solve. This traversal visits every vertex *at least* once, but it visits many vertices multiple times, which is not allowed in the final TSP. What can we do about that? Well, the simplest solution is just to *not* do that. Any time we encounter a duplicate vertex, lets just skip it and go to the next non-duplicate that the traversal would visit. This looks like the following.



We call the process of cutting out duplicates "shortcutting". We are taking a shortcut from the traversal and going straight to the next non-duplicate! Now here's the last cool observation we need. What did we just end up with? When we shortcutted the traversal, we ended up with a sequence of vertices in a depth-first traversal order... Hmmm, this is nothing but the *pre-order* of the vertices in the tree!

Algorithm: MST Approximation for Metric TSP

- Compute an MST M of the graph G
- Root M at an arbitrary vertex r
- Output a pre-order of M followed by r (return to the root)

Theorem: MST-TSP is a 2-approximation

This MST approach gives a 2-approximation to the METRIC TSP problem.

Proof. First, we want to compare the cost of the optimal TSP tour with the weight of the MST. Consider any TSP tour (i.e., a cycle that visits every vertex). If we remove any one edge of the tour, what are we left with? We will have $n-1$ edges that connect all n vertices... Oh, that's just a spanning tree! Since its a spanning tree, by definition its weight can not be less than that of the MST, so

$$\text{weight}(MST) \leq \text{weight}(TSP - \{e\}) \leq \text{weight}(TSP).$$

Now how costly is our solution? We started by traversing the MST depth-first, using each edge twice, one down then once back up. Therefore the cost of this walk was $2\text{weight}(MST)$. Then we had to shortcut out the duplicates to obtain a valid TSP tour. However, remember that the triangle inequality says that going directly to a vertex is never worse than taking a detour, so therefore this shortcutting process can only make the cost lower, so

$$\text{ALG} \leq 2 \cdot \text{weight}(MST) \leq 2 \cdot \text{weight}(TSP) = 2\text{OPT}.$$

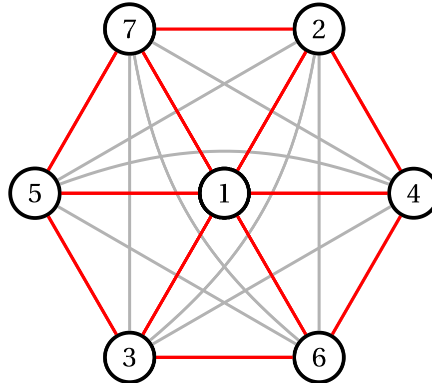
□

The second step is where we needed the assumption that the distances were a metric. Without it, we can not make the argument that the shortcuts do not increase the total cost.

18.4.1 Worst Case Graphs

To prove tightness for this approximation ratio we will look at a certain type of graphs.

Consider for any odd n larger than 5, the wheel graph on n vertices W_n. On our complete graph, we can make any edge on the wheel graph distance 1, and all other edges distance 2. Also, we will label the vertices in a particular zig-zagging fashion. The reasoning for this will become clear later. Here is such a graph on 7 vertices, where the red edges have distance 1 and the rest have distance 2.



It's easy to see how these distances satisfy our distance metric. We can also see OPT is 7, following the tour $\{1,2,4,6,3,5,7,1\}$.

Now, consider the MST on this graph that is the star graph centered on vertex 1. A preorder tour of this MST will be $\{1,2,3,4,5,6,7,1\}$. Here, our labeling comes into effect, because after shortcutting, every edge this tour traverses except for the first and last is distance 2. If we had labeled the vertices the obvious way, these shortcutted edges would still be distance 1. Therefore, ALG is 12.

If we generalize this graph, we can see that OPT will always remain n, by using the wheel graph edges. As mentioned, every edge the MST TSP traverses except for the first and last will be distance 2, meaning ALG will be $2(n-2)+2=2(n-1)$.

The approximation ratio in this case is $\frac{2(n-1)}{n}=2(1-\frac{1}{n})$. So as n grows larger this will tend to 2, and we can see that 2 is indeed the tightest constant approximation ratio.

Lecture 19

Online Algorithms

Today we'll be looking at finding approximately-optimal solutions for problems where the difficulty is not that the problem is necessarily *computationally* hard but rather that the algorithm doesn't have all the *information* it needs to solve the problem up front.

Specifically, we will be looking at *online algorithms*, which are algorithms for settings where inputs or data is arriving over time, and we need to make decisions on the fly, without knowing what will happen in the future. This is as opposed to standard problems like sorting where you have all inputs at the start. Data structures are one example of online algorithms (they need to handle sequences of requests, and to do well without knowing in advance which requests will be arriving in the future).

Objectives of this lecture

In this lecture, we will

- define and motivate **online algorithms**
- solve the rent-or-buy problem with an online algorithm and analyze its performance
- analyze various strategies for the *list update* problem. In particular, we will see how *potential functions* are key ingredients in the analysis of online algorithms.
- analyzing online paging algorithms and see how *randomization* allows us to achieve provably better performance than any deterministic algorithm.

19.1 Framework and Definition

We are given a problem in which the input arrives over time rather than being known entirely up front. At each point in time, we have to make some decision, and each such decision is irrevocable, i.e., we can not change our mind later. Depending on the choices we make, we incur some cost, depending on the cost model of the problem. The goal is to perform well relative to an optimal *omniscient* algorithm, i.e., one that can predict the future and see the entire input in advance.

This is similar to the way in which we analyze approximation algorithms, by comparing the performance of our algorithm to that of the best possible algorithm (except that in this case, our definition of "best" is that it can cheat and see the future). We define the *competitive ratio* of an online algorithm very similarly to the approximation ratio.

Definition: Competitive Ratio

An online algorithm is called c *-competitive* if for all possible inputs σ

$$\mathrm{ALG}(\sigma) \leq c \cdot \mathrm{OPT}(\sigma),$$

where $\mathrm{ALG}(\sigma)$ is the cost incurred by the online algorithm on the input σ (which it does not know in advance) and $\mathrm{OPT}(\sigma)$ is the cost of an optimal omnipotent algorithm that can see σ in advance. The factor c is called the *competitive ratio* of the algorithm.

19.2 Rent or buy?

Here is a simple online problem that captures a common issue in online decision-making, called the rent-or-buy problem.

Problem: Rent or buy

Its the middle of the snow season and you are planning on going skiing. You can rent a pair of skis at $\$r$ per day, or buy a pair for $\$b$ and keep them forever. You would like to ski for as many days as possible, however, you do not know how many more days of the season will be viable weather for skiing. Each day you find out whether the weather is still good. At some point, you discover that the ski season is over. Your choice is to decide whether to rent or buy skis on each day, with the goal of minimizing the total amount of money that you spend.

Let's walk through a concrete example. You can either rent skis for $50 or buy them for $500. If we know the future in advance, the solution is to buy immediately if we know that there are at least 10 days of viable skiing weather, and if not, just always rent. The tricky part is designing an *online* algorithm that doesn't know the future. It has no idea how many days of viable whether there will be. Lets start with some simple but sub-optimal strategies to illustrate:

- **Always immediately buy:** One valid online strategy is to immediate buy on the first day if the weather is good. The worst case input for the this strategy is when we only get to go skiing once, so we could have just paid $50, so the competitive ratio is $500/50 = 10$, we paid $10\times$ more than we could have.

  Rent forever: Another strategy is to never buy skis and just always rent. In this strategy, the worst case input is that the ski season goes on arbitrarily long, and we end up paying an arbitrarily high amount of money, when the optimal choice would have been to buy immediately, so the competitive ratio is actually ∞ (or unbounded).

In general, since after buying the skis the algorithm has no more decisions to make, we can characterize any online algorithm for the rent-or-buy problem by the day on which it decides to buy. Now observe that in general, the worst-case input for such an algorithm is that the weather is bad on the day after it buys the skis. With this in mind, here is one more bad strategy before we hone in on the optimal one.

- **Rent five times then buy:** How about we rent five times, then decide that it is time to buy. The worst-case input is that the weather is good for six days, then bad. In this case, our algorithm pays $5 \times 50 + 500 = 750$, but the optimal algorithm would just always rent, which costs $6 \times 50 = 300$, so this is 2.5-competitive.

Well that's certainly a lot better than 10. It seems like if we hedge our bets by renting longer, we get a better competitive ratio. At some point, this will stop being true, though. In particular, it never makes sense to plan to rent for more than 10 days, because then we should have just bought the skis for sure. So the most hedging we can do is to rent for 10 days then buy. This is called the *better-late-than-never* algorithm.

Algorithm: Better-late-than-never

We rent for $b/r - 1$ days[a], then we buy. In other words, we buy on day b/r.

[a] If r does not divide b, then we should rent for $\lceil b/r \rceil - 1$ days, but we will just assume that r divides b for simplicity

Theorem: Better-late-than-never is 2-competitive

Better-late-than-never is a 2-competitive algorithm for the rent-or-buy problem.

Proof. Suppose the weather is good for n days. We have to consider two cases:

1. If $nr < b$ (i.e., $n < b/r$), then the optimal solution is to always rent, but in this case, our algorithm doesn't buy either, so it is optimal.
2. If $nr \geq b$, the optimal solution buys immediately, but our algorithm first rents for $b/r - 1$ days before buying, so the ratio is

$$\frac{\left(\frac{b}{r} - 1\right) r + b}{b} = \frac{b - r + b}{b} = 2 - \frac{r}{b} \leq 2. \qquad \square$$

Now as we will naturally want to ask: can we do better?

Problem 43. Show that *better-late-than-never* has the best possible competitive ratio for the rent-or-buy problem for deterministic algorithms when b is a multiple of r.

19.3 List Update

This is a nice problem that illustrates some of the ideas one can use to analyze online algorithms. Here are the ground rules for the problem:

Problem: List Update

- We begin with a list of n items $1, 2, \ldots, n$. Imagine a linked list starting with 1 and ending with n.

- An item x can be *accessed*. The operation is called `Access`(x). The cost is the position of x.
- The algorithm can rearrange the list by swapping adjacent elements. The cost of a swap is 1.

So an on-line algorithm is specified by describing which swaps are done and when. The goal is to devise and analyze an on-line algorithm for doing all the accesses `Access`(σ_1), `Access`(σ_2), `Access`(σ_3), ... with a small competitive factor. Here are several algorithms to consider.

Problem 44. Do no swaps Consider an online algorithm that does no swaps. What is a worst-case input for this algorithm? What is the resulting competitive ratio?

Problem 45. Single exchange Consider an online algorithm that, for each accessed element, swaps it one position closer to the front of the list. What is a worst-case input for this algorithm? What is the resulting competitive ratio?

Problem 46. Frequency count Consider an online algorithm that remembers the frequency of each element being accessed, and keeps the list sorted by frequency (largest to smallest). What is a worst-case input for this algorithm? What is the resulting competitive ratio?

Okay, those were warm-ups, now its time for the real algorithm.

Algorithm: Move-to-front

After an access to an element x, do a series of swaps to move x to the front of the list.

Theorem 19.1: Competitive ratio of MTF

MTF is a 4-competitive algorithm for the list-update problem.

Proof. We'll use the potential function method. There will be a potential function that depends on the state of the MTF algorithm and the state of the "opponent" algorithm B, which can be any algorithm, even one which can see the future. Using this potential, we'll show that the amortized cost to MTF of an access is at most 4 times the cost of that access to B.

What is the potential function Φ? Define

$$\Phi_t = 2 \cdot (\text{The number of inversions between } B\text{'s list and MTF's list at time } t)$$

Recall that an inversion is a pair of distinct elements (x, y) that appear in one order in B's list and in a different order in MTF's list. It's a measure of the similarity of the lists.

We can first analyze the amortized cost to MTF of `Access`(x) (where it pays for the list traversal and its swaps, but B only does its access). Then we separately analyze the amortized cost to MTF that is incurred when B does any swaps. (Note that in the latter case MTF incurs zero cost, but it will have a non-zero amortized cost, since the potential function may change. To be

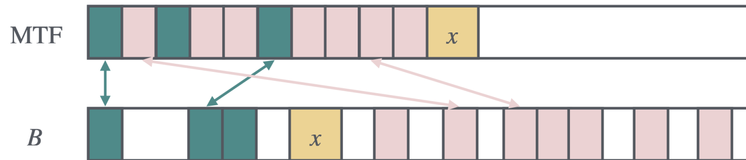
complete the analysis must take this into account.). In each case we'll show that the amortized cost to MTF (which is the actual cost, plus the increase in the potential) is at most 4 times the cost to B.

For any particular step, let C_{MTF} and C_B be the actual costs of MTF and B on this step, and $AC_{MTF} = C_{MTF} + \Delta\Phi$ be the amortized cost. Here $\Delta\Phi = \Phi_{new} - \Phi_{old}$ is the increase in Φ. Hence observe that $\Delta\Phi$ may be negative, and the amortized cost may be less than the actual cost. We want to show that

$$AC_{MTF} \leq 4 \cdot C_B$$

We can then sum the amortized costs, which would equal the actual cost of the entire sequence of accesses to MTF plus the final potential (non-negative) minus the initial potential (zero). This would be the four times total cost of B, which would give the result.

Analysis of `Access`(x). First look at what happens when MTF accesses x and brings it to the front of its list. Say the picture looks like this:



Look at the elements that lie before x in MTF's list, and partition them as follows:

$$S = \{y \mid y \text{ is before } x \text{ in MTF and } y \text{ is before } x \text{ in } B\}$$
$$T = \{y \mid y \text{ is before } x \text{ in MTF and } y \text{ is after } x \text{ in } B\}$$

In the above picture, S is dark/teal, and T is light/pink.

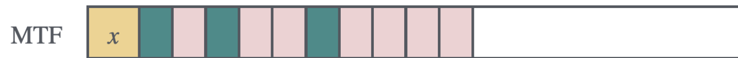
What is the cost of the access to MTF in terms of these sets?

$$C_{MTF} = 1 + \underbrace{|S| + |T|}_{\text{find cost}} + \underbrace{|S| + |T|}_{\text{swap cost}} = 1 + 2(|S| + |T|).$$

On the other hand, since all of S lies before x in B, the cost of the algorithm B is at least

$$C_B \geq |S| + 1.$$

What happens to the potential as a result of this operation? Well, here's MTF after the operation:



The only changes in the inversions involve element x, because all other pairs stay in the same relative order. Hence, for every element of S the the number of inversions increases by 1, and for every element of T the number of inversions decreases by 1. Hence the increase in Φ is precisely:

$$\Delta(\Phi) = 2 \times (|S| - |T|)$$

Now the amortized cost is

$$
\begin{aligned}
AC_{\text{MTF}} = C_{MTF} + \Delta(\Phi) &= 2(|S| - |T|) + 1 + 2(|S| + |T|) \\
&= 1 + 4|S| \leq 4(1 + |S|) \leq 4\, C_B
\end{aligned}
$$

This completes the amortized analysis of `Access`(x).

Analysis of B swapping. As explained above, we view B doing a swap as an event in its own right, not associated with any of the access events. For each such swap, observe that $C_{\text{MTF}} = 0$ and $C_B = 1$. Moreover, $\Delta(\Phi) \leq 2$, since the swap may introduce at most one new inversion. Hence,

$$AC_{MTF} \leq 2\, C_B \leq 4\, C_B$$

Putting the parts together. Summing the amortized costs we get:

$$\text{Total Cost to MTF} + \leq 4(\text{Total Cost to } B) + \Phi_{\text{init}} - \Phi_{\text{final}}$$

But $\Phi_{\text{init}} = 0$, since we start off with the same list as B. And $\Phi_{\text{final}} \geq 0$. Hence $\Phi_{\text{init}} - \Phi_{\text{final}} \leq 0$. Hence,

$$\text{Total Cost to MTF} \leq 4 \times (\text{Total Cost to } B).$$

Hence the MTF algorithm is 4-competitive. □

Key Idea: Using a potential function to analyze online algorithms

We saw potential functions earlier in the course for coming up with *amortized* cost bounds. Another powerful use of them is to analyze online algorithms. They help us relate the cost of our algorithm to the cost of the optimal omniscient algorithm, which is often very tricky to figure out without a potential function.

19.4 Paging

Our last problem shows up frequently in systems but also has relevance to efficient algorithms. Its called the *paging* problem.

Problem: Paging

In the paging problem, we say we have:

- a disk with N pages of memory,
- a *cache* with space for $k < N$ pages. Initially, the cache stores pages $1, 2, \ldots, k$.

We have to process a sequence of *requests*. When a request is made, if the page is in the cache, the request is free. If the page is not in the cache, we have a *page fault* (this is also called a *cache miss*). We then need to bring the page into the cache and throw something else out if the cache is full. The decision here is *which page* to throw out. Our goal / cost

model is to minimize the number of page faults.

The paging problem has several important applications in the area of computer systems. For example, operating systems have to manage virtual memory and decide how to map it onto physical memory (which is much smaller). CPU caches need to decide which cache lines to keep and which ones to throw out when they incur a cache miss, and many database systems store caches of recently/frequently accessed items to improve their performance. Essentially, any time you have a large amount of stuff but you want to make repeatedly accessing the same small amount of stuff much faster, you have the paging problem!

An online algorithm: LRU A standard online algorithm is the *least recently used* heuristic (LRU): "throw out the least recently used page". This is used commonly in practice because it is simple and performs reasonably well empirically. What's a bad case for LRU? What if the request sequence looks like 1,2,3,4,1,2,3,4,1,2,3,4... Notice that in this case, the algorithm makes a page fault every time and yet if we knew the future we could have thrown out a page whose next request was 3 time steps ahead. More generally, this type of example shows that the competitive ratio of LRU is at least k.

In fact, it's not hard to show that you can't do better than a competitive ratio of k with *any deterministic* algorithm.

Theorem 19.2

Any deterministic algorithm cannot have a competitive ratio better than k.

Proof. Set $N = k+1$ and consider a request sequence that always requests whichever page the algorithm *doesn't* have in its cache. By design, this will cause the algorithm to have a page fault every time. However, if we knew the future, every time we had a page fault we could always throw out the item whose next request is farthest in the future (instread of the least recently used page). Since there are k pages in our cache, for one of them, this next request has to be at least k time steps in the future, and since $N = k+1$, this means we won't have a page fault for at least $k-1$ more steps (until that one is requested). So, the algorithm that knows the future has a page fault at most once every k steps, and the ratio is k. □

We can also show that LRU can indeed achieve a competitive ratio of k. In other words, it is (one of) the optimal deterministic algorithm(s).

Theorem 19.3

LRU achieves a competitive ratio of k.

Proof. Define a phase to contain k distinct requests, and moreover, the next request is distinct from all requests in the phase. First, it is not hard to see that LRU incurs at most k page faults in each phase, and thus the total number of page faults is at most $k \cdot m$ where m is the number

of phases. Now, we show that any algorithm must pay a cost of at least $m-1$ for m phases. This shows a competitive ratio of k as m grows large. □

Claim 19.1

Any algorithm must incur a cost of $m-1$ for m phases.

Proof. For any arbitrary phase i, we can define an associated *offset phase* from the second request of phase i to the first request of phase $i+1$. We claim that any algorithm must incur at least one page fault in each offset phase. In particular, either the first request of phase $i+1$ incurs a page fault, or if it doesn't, then the first request of phase $i+1$ must reside in the cache throughout phase i. Further, when the first request of phase i is served, that page is fetched into the cache. Now, there still remain $k-1$ distinct pages in the offset chunk, and as there are only $k-1$ spots on the cache left, one of them must incur a page fault. There are $m-1$ such offset phases (from the second request of some phase i to the first request of the next phase $i+1$). Thus, the total number of page faults of any algorithm is at least $m-1$. □

Does this mean that we cannot get any algorithms that are better than k-competitive? No! While deterministic algorithms have this limitation, we can use randomization to help us. Here is a neat *randomized* algorithm with a competitive ratio of $O(\log k)$.

Algorithm: Randomized marking algorithm

- Initiall. pages $1, 2, \ldots, k$ are in the cache. Start with these pages all marked.
- When a page is requested,
  - if it's in cache already, mark it and return.
  - Otherwise,
    - if every page is marked, unmark every page.
    - Throw out a random unmarked page, bring in the requested page, and mark it.

We can think of this as a 1-bit randomized LRU, where marks represent "recently used" vs "not recently used". For analysis purposes, we call the step in which every page is unmarked the beginning of a *phase*.

The figure below illustrates a phase of the algorithm for $N = 5, k = 4$. Each page (shown as a box) contains the probability that that page is in cache. Orange shading means that that page is marked. The column on the left shows the requested page, and the column on the right shows the expected cost of that request. The phase is comprised of accesses to k distinct pages. (Access to pages that are in the cache with probability 1, i.e., that have probability 0 of incurring a page fault, are not shown.) At the end of the phase, all marks are erased, and the next phase begins.

request	pages 1	2	3	4	5	expected cost
	1	1	1	1	0	
5						1
	$\frac{3}{4}$	$\frac{3}{4}$	$\frac{3}{4}$	$\frac{3}{4}$	1	
2						$\frac{1}{4}$
	$\frac{2}{3}$	1	$\frac{2}{3}$	$\frac{2}{3}$	1	
1						$\frac{1}{3}$
	1	1	$\frac{1}{2}$	$\frac{1}{2}$	1	
4						$\frac{1}{2}$
	1	1	0	1	1	

Theorem 19.4

When the marking algorithm is run on a sequence σ of accesses, we have:

$$\frac{\mathbb{E}[\text{MARKING}(\sigma)]}{\text{OPT}(\sigma)} \leq \begin{cases} H_k & \text{if } N = k+1 \\ 2H_k & \text{if } N > k+1 \end{cases}$$

Note that $H_k := 1 + 1/2 + \ldots + 1/k$ is the kth harmonic number. Recall that $H_k \leq 1 + \ln k$.

Proof. We will only show the proof for the special case of $N = k+1$. For general N, the proof follows similar lines but just is a bit more complicated.

Each phase contains k distinct page requests. The first time each of these pages are requested in the page they will be unmarked, since the first request of a phase will not be in the cache, and at that point the entire cache will be marked from the previous phase. So, we will unmark everything. Thus, the probabilities of these first accesses being to pages not in the cache is nonzero.

So, we can find the expected cost of each of these. The first request, as mentioned, always results in a page fault. Thus, it has a cost of 1.

For any other request $i \in \{2, 3, \ldots, k\}$, at that point there are $i-1$ marked pages in the cache (one for all previous requests). This leaves $N-i-1$ unmarked pages. One of these pages is out of the cache, and it is equally likely for the requested page to be any of them, so the expected cost of this request is

$$\frac{1}{N-i+1}.$$

Summing over all requests in a phase, we have that the total expected cost of a phase is

$$
\begin{aligned}
1+\sum_{i=2}^{k}\frac{1}{N-i+1} &= 1+\sum_{i=0}^{k-2}\frac{1}{N-1-i} \\
&= 1+\sum_{i=0}^{k-2}\frac{1}{k-i} \\
&= 1+\sum_{2}^{k}\frac{1}{i} \\
&= H_k.
\end{aligned}
$$

So, over m phases, the marking algorithm will incur mH_k expected cost in total. It suffices to show that any algorithm must incur at least m cost for m phases. This was proven in Claim 1. □

Lecture 20
Streaming Algorithms

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

Objectives of this lecture

In this lecture, we will:

- Introduce the data streaming model, and its concerns.
- Analyze an algorithm for heavy hitters in the arrivals-only model.
- Analyze an algorithm for heavy hitters with both arrivals and departures.

20.1 Introduction

Today's lecture will be about a slightly different computational model called the *data streaming* model. In this model you see elements going past in a "stream", and you have very little space to store things. For example, you might be running a program on an Internet router, the elements might be IP Addresses, and you have limited space. You certainly don't have space to store all the elements in the stream. Once you have read an element of the stream, you can not look back at prior elements of the stream. The question is: which functions of the input stream can you compute with what amount of time and space?

We will denote the stream elements by

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

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

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

$$3, 1, 17, 4, -9, 32, 101, 3, -722, 3, 900, 4, 32, \dots \tag{$\diamond$}$$

Computing the sum Here, $F(a_{[1:t]}) = \sum_{i=1}^{t} a_i$. We want the outputs

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

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

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

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

We just need to store b bits.

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

$$3, 1, 3, 3, 3, 3, 4, 3, \dots$$

Doing this with small space is a lot more tricky.

The number of distinct elements Again, this is quite tricky to do exactly in low space.

Heavy hitters These are elements that have appeared most frequently in the stream.

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

- **Approximate solutions**: in several cases, it will be impossible to compute the function exactly using small space. Hence we'll explore the trade-offs between approximation and space. For example, we will develop algorithms that admit *false positives*, and algorithms that approximate the answers with some amount of error.
- **Hashing**: this will be a very powerful technique.

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

Remark: Measuring space in terms of bits

In this lecture, we will be focusing on space bounds rather than runtime. Additionally, in this model, we will be measuring the space usage of an algorithm in terms of the number of *bits* needed, rather than the number of words. In many other models, we would think of the space required to store n integers as $O(n)$ words. Outside the word RAM model, storing a sequence of n integers in the range 0 to n uses $O(n \log n)$ *bits* of space, because each integer requires $O(\log n)$ bits. In general, in this model, storing n integers of b bits each takes $O(nb)$ space. It is important to keep this in mind.

20.2 Finding ε-Heavy Hitters

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

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

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

Definition 20.1: ε-heavy-hitters

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

The goal is simple — given an threshold $\varepsilon \in [0, 1]$, maintain a data structure that is capable of outputting ε-heavy-hitters. At any point in time, we can query the data structure, and it outputs a set of at most $1/\varepsilon$ elements that contains all the ε-heavy hitters. At any moment in time, there are at most $1/\varepsilon$ elements that are ε-heavy-hitters, so this request is not unreasonable.

Remark

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

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

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

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

- at time 5, the element D is the only $\frac{1}{3}$-heavy-hitter,
- at time 11 both B and D are $\frac{1}{3}$-heavy-hitters, and
- at time 15, there is no $\frac{1}{3}$-heavy-hitter, and

- at time 16, only E is a $\frac{1}{3}$-heavy-hitter.

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

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

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

20.2.1 Finding a Majority Element

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

Algorithm 20.1: Boyer-Moore Majority

We keep one element from Σ, which is our candidate majority element, in a variable called `candidate`, and a counter

```
candidate = ⊥
counter = 0

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

At the end, return `candidate`.

Intuitively, this algorithm works because a majority element occurs strictly more than half the time, and the counter variable is counting at least how many *more times* this element occurs than any other element, which must be a positive amount. It may, however, output a false positive when there does not exist a majority element.

- Suppose there is no majority element, then we'll output a false positive. As we said earlier, that's OK: we want to output a set of size 1 that contains the majority element, *if any*.
- If there is a majority element, we *will* indeed output it. Why? Observe: when we discard an element a_t, we also throw away another (different) element as well. (Decrementing the counter is like throwing away one copy of the element in `candidate`.) So every time we

throw away a copy of the majority element, we throw away another element too. Since there are fewer than half non-majority elements, we cannot throw away all the majority elements.

20.2.2 Finding an ε-heavy-hitter

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

Algorithm 20.2: ε-heavy-hitters

We keep an array Candidates$[1 \dots k]$, where each location can hold one element from Σ; and an array Counts$[1 \dots k]$, where each location can hold a non-negative integer.

```
Candidates[i]=⊥ for all 1≤i≤k
Counts[i]=0 for all 1≤i≤k

When element a_t arrives.
  if a_t ==Candidates[j] for some j, then Counts[j]++.
  else if Counts[j]==0 for some j then Candidates[j]←a_t and Counts[j]←1.
  else decrement all the counters by 1.
```

An an exercise, check that this algorithm is identical to the one above for $\varepsilon = 1/2$.

To analyze the algorithm and prove that it is correct, we define the *estimated count* which corresponds to the current count that the algorithm is maintaining for each of the candidates (or zero for those elements which are not currently a candidate):

$$\text{est}_t(e) = \begin{cases} \text{Counts}[j] & \text{if } e == \text{Candidates}[j] \\ 0 & \text{otherwise} \end{cases}$$

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

Lemma 20.1

The estimated counts satisfy:

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

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

To prove the other inequality, think of the arrays Counts and Candidates saying that we have Counts[1] copies of Candidates[1], Counts[2] copies of Candidates[2], etc., in our hand. Look at some time when the difference between $\text{count}_t(e)$ and $\text{est}_t(e)$ increases by 1. Each time

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

this happens, we decrement k different counters by 1 and discard an element (which is not currently present in the candidates array). This is like dropping $k+1$ distinct elements (one of which is e). We can drop at most t elements until time t. So the gap can be at most $t/(k+1)$. And since $k = \lceil 1/\varepsilon \rceil - 1 \geq 1/\varepsilon - 1$, we get that $t/(k+1) \leq \varepsilon t$. □

Corollary 20.1

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

Proof. After we've seen n elements, the ε-heavy-hitters have occurred at least $\text{count}_t(e) > \varepsilon t$ times. If e is a ε-heavy-hitter, by Lemma 20.1, the estimate

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

So element e must be in the array T. □

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

20.3 Heavy Hitters with Deletions

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

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

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

then the sets at various times are

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

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

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

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

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

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

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

20.3.1 A Hashing-Based Solution: First Cut

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

Algorithm 20.3: Approximate count

Maintain an array counts$[1\ldots k]$ capable of storing non-negative integers.

```
when update a_t arrives
 if (a_t == (add, e)) then
  counts[h(e)]++;
 else // a_t == (del, e)
  counts[h(e)]--;
```

One way to think about this is that we are just maintaining a hashtable *without collision resolution*. This was the update procedure. And what is our estimate for the number of copies of element e in our active set S_t? It is

$$\text{est}_t(e) := \text{counts}[h(e)].$$

In words, we look at the location $h(e)$ where e gets mapped using the hash function h, and look at counts$[h(x)]$ stored at that location. What does it contain? It contains the current count for element e for sure. But added to it is the current count for any other element that also gets mapped to that same location. In math:

$$\text{counts}[h(e)] = \sum_{e'\in\Sigma} \text{count}_t(e')\cdot \mathbf{1}(h(e')=h(e)),$$

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

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

or using the definition of the estimate, as

$$\text{est}_t(e) - \text{count}_t(e) = \sum_{e'\neq e} \text{count}_t(e')\cdot \mathbf{1}(h(e')=h(e)). \tag{20.1}$$

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

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

Definition: Universal hashing

A family $\mathcal{H}$ of hash functions from $\Sigma \to [k]$ is universal if for any pair of keys $x_1, x_2 \in \Sigma$ with $x_1 \neq x_2$,

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

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

Good. So we drew a hash function h from this universal hash family $\mathcal{H}$, and we used it to map elements to locations $\{0, 1, \ldots, k-1\}$. What is its expected error? From (20.1), it is

$$E\left[\sum_{e' \neq e} \text{count}_t(e') \cdot \mathbf{1}(h(e') = h(e))\right] = \sum_{e' \neq e} \text{count}_t(e') \cdot E\left[\mathbf{1}(h(e') = h(e))\right] \tag{20.2}$$

$$= \sum_{e' \neq e} \text{count}_t(e') \cdot \Pr[h(e') = h(e)]$$

$$\leq \sum_{e' \neq e} \text{count}_t(e') \cdot (1/k) \tag{20.3}$$

$$= \frac{|S_t| - \text{count}_t(e)}{k} \leq \frac{|S_t|}{k}.$$

We used linearity of expectations in equality (20.2). To get (20.3) from the previous line, we used the definition of a universal hash family. Let's summarize:

Claim 20.1

The estimator $\text{est}_t(e) = \text{counts}[h(e)]$ ensures that

(a) $\text{est}_t(e) \geq \text{count}_t(e)$, and

(b) $E[\text{est}_t(e)] - \text{count}_t(e) \leq |S_t|/k$.

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

That's pretty awesome. (But perhaps not so surprising, once you think about it.) Note that if we were only doing arrivals and no deletions, the size of S_t would be exactly t. So the expected error would be at most t/k, which is about the same as we were getting in Section 20.2 using an array of size $(k-1)$. But now we can also handle deletions!

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

20.3.2 Amplification of the Success Probability

Any ideas how to *amplify* the probability that we are close to the expectation? The idea is simple: *independent repetitions*. Let us pick m hash functions $h_1, h_2, \ldots, h_m$. Each $h_i : \Sigma \to \{0, 1, \ldots, k-1\}$. How do we choose these hash functions? *Independently* from the universal hash family H.[3]

Algorithm: CountMin sketch

We have m arrays $\text{counts}_1, \text{counts}_2, \ldots, \text{counts}_m$, one for each hash function. The algorithm now just uses the ith hash function to choose a location in the ith array, and increments or decrements the same as before.

```
when update a_t arrives
  for each i from 1..m
    if (a_t == (add, e)) then
      counts_i[h_i(e)]++
    else // a_t == (delete, e)
      counts_i[h_i(e)]--
```

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

$$\text{best}_t(e) := \min_{i=1}^{m} \text{counts}_i[h_i(e)].$$

In other words, each (h_i, counts_i) pair gives us an estimate, and we take the least of these. It makes perfect sense — the estimates here are all *<u>over</u>*estimates, so taking the least of these is the right thing to do. But how much better is this estimator? Let's do the math.

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

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

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

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

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

[3] If we use the random binary matrix hash family construction given in the hashing lecture, this means the $(\lg k) \cdot (\lg |\Sigma|)$-bit matrices for each hash function must be filled with independent random bits.

Final Bookkeeping

Let's set the parameters now. Set $k = 2/\varepsilon$, so that the error bound $2|S_t|/k = \varepsilon|S_t|$. And suppose we set $m = \lg 1/\delta$, then the failure probability is $(1/2)^m = \delta$, and our query will succeed with probability at least $1-\delta$. [4]

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

$$\Pr\left[\left|\text{best}_t(e) - \text{count}_t(e)\right| \le \varepsilon|S_t|\right] \ge 1-\delta.$$

Just as we wanted. And the total space usage is

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

Each counter has to store at most $\lg T$-bit numbers after T time steps. [5]

Space for Hash Functions: We need to store the m hash functions as well. How much space does that use? The random binary matrix method that we learned earlier in the course used $s := (\lg k) \cdot (\lg \Sigma)$ bits per hash function. Since $k = 1/\varepsilon$, the total space used for the functions is

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

And in Summary...

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

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

[5] So a 32-counter can handle a data stream of length 4 billion. If that is not enough, there are further techniques to reduce this space usage as well.

Lecture 21

Computational Geometry: Fundamentals

Computational geometry is the design and analysis of algorithms for geometric problems that arise in low dimensions, typically two or three dimensions. Many elegant algorithmic design and analysis techniques have been devised to attack geometric problems, and these problems have huge applications in many other fields.

Objectives of this lecture

In this lecture, we will

- see some motivating applications of computational geometry
- define some fundamental objects (points, lines, polygons) and common operations on them
- derive an efficient algorithm for the *convex hull* problem, a classic and very fundamental problem in computation geometry

21.1 Introduction

Computational geometry has applications in, and in some cases is the entire basis of, many fields. E.g.,

- Computer Graphics (rendering, hidden surface removal, illumination)
- Robotics (motion planning)
- Geographic Information Systems (Height of mountains, vegetation, population, cities, roads, electric lines)
- Computer-aided design (CAD) /Computer-aided manufacturing (CAM)
- Computer chip design and simulations
- Scientific Computation (Blood flow simulations, Molecular modeling and simulations)

21.1.1 Representations and Assumptions

The basic approach used by computers to handle complex geometric objects is to decompose the object into a large number of very simple objects. Examples:

- An image might be a 2D array of dots.
- An integrated circuit is a planar triangulation.
- Mickey Mouse is a surface of triangles

It is traditional to discuss geometric algorithms assuming that computing can be done on ideal objects, such as real valued points in the plane. The following chart gives some typical examples of representations.

Abstract Object	**Representation**
Real Number	Floating-point Number
Point	Pair of Reals
Line	Pair of Points, An Equation
Line Segment	Pair of Endpoints
Triangle	Triple of points

For the purpose of this course, we will assume a rather idealized model of computation, where we assume we can compute on the objects we care about (which might be integers, or sometimes might be arbitrary real numbers) in constant time without any loss of precision. For example, we will ignore things like rounding errors which would be a real-life consideration if you were actually implementing many of these algorithms.

Remark: Rounding errors...

In real life, computational geometry algorithms are often extremely susceptible to rounding errors. Algorithms which might be simple to describe on paper or in pseudocode can end up being very subtle and difficult to implement correctly because of it. In this course, we will mostly sweep this under the rug and focus on the elegance of the algorithms without this pain point in mind.

21.2 Fundamental Operations

21.2.1 Operations on points and vectors

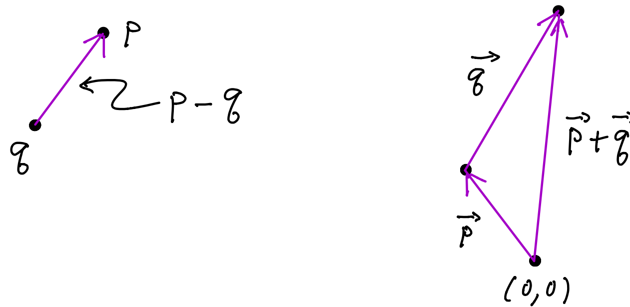
In computational geometry, the most primitive object is a *point*, which we can represent as a tuple of real numbers. We will mostly be focusing on 2D computational geometry, so points will be represented as pairs (x, y) of real numbers. In 2D or 3D space, a vector is a pair of 3-tuple of real numbers, respectively. Wait, that's just the same thing as a point! Indeed, the difference

between a point and a vector is often just context, points are typically used to denote locations, and vectors are typically used to denote directions/lengths or differences, but mathematically they are interchangeable.

Addition and subtraction Let's start by defining pointwise addition/subtraction of vectors:

$$(x_1, y_1) + (x_2, y_2) := (x_1 + x_2, y_1 + y_2)$$
$$(x_1, y_1) - (x_2, y_2) := (x_1 - x_2, y_1 - y_2)$$

The effect of vector addition and subtraction can be visualized as in the following pictures. $p - q$ results in a vector that points from the point q to the point p. Adding $p + q$ results in a vector that points to the location you would get to if you start at $(0,0)$ then follow p then q (equivalently, q then p).



Scalar multiplication We can define the multiplication of a vector by a scalar. Geometrically, multiplying a vector by a scalar α makes it α times longer.

$$\alpha(x, y) := (\alpha x, \alpha y)$$

Magnitude/length The magnitude or length of a vector is given by the formula

$$\|(x, y)\| := \sqrt{x^2 + y^2}$$

A particularly useful use case is to compute the distance between two points p and q, given by $\|p - q\|$.

The dot product The dot product operation takes two vectors and returns a real number. It is the first of two kinds of "multiplication" we will define for vectors. It is defined as

$$(x_1, y_1) \cdot (x_2, y_2) = x_1 x_2 + y_1 y_2$$

Dot products are useful for computing angles.

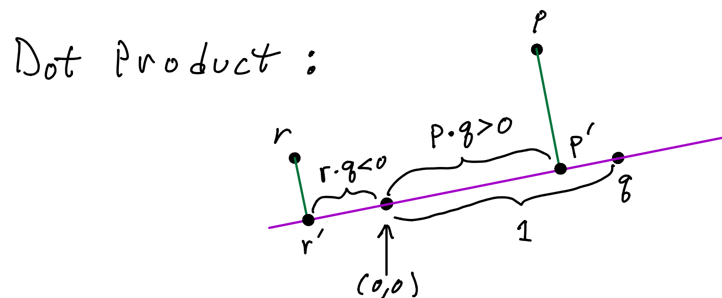
Theorem: Dot-product-angle formula

For any vectors u, v, we have

$$u \cdot v = \|u\|\|v\| \cos(\theta)$$

where θ is the angle between the vectors u and v.

Geometrically, the dot product can be seen as a projection operator. Consider the line L through $(0,0)$ and q. Project p to the line L with a perpendicular (shown in green in the following figure.) Call this point p'. Then the value of $p \cdot q$ is the length of p', denoted $|p'|$, times the length of q. That is, $|p'| * |q|$. Like the dot product this value is signed. If $(0,0)$ is not between p' and q then it's positive, otherwise it's negative. The figure below shows the case when $|q| = 1$, which is commonly used to compute projections.



Algorithm: Projection onto a line

Given a point p and a line L, such that L goes through the origin and a point q such that $|q| = 1$, we can compute the projection[a] of p onto the line by

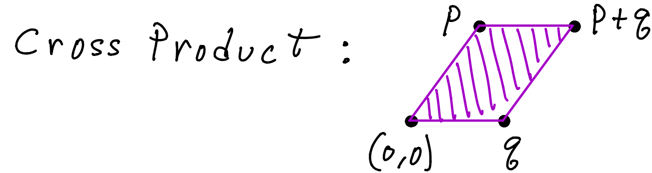
$$p' = (p \cdot q)q$$

[a]The *projection* p' of the point p onto a line is the closest point on the line to p

The cross product The cross product is the other kind of "multiplication" between two points. In 2D, the cross product is defined as

$$(x_1, y_1) \times (x_2, y_2) := \det\left(\begin{bmatrix} x_1 & y_1 \\ x_2 & y_2 \end{bmatrix}\right) = x_1 y_2 - y_1 x_2$$

where $\det(M)$ is the determinant of M. The cross product of p and q is the signed area of the parallogram with the four vertices $(0,0), p, q, p+q$. Equivalently, it is twice the signed area of the triangle with vertices $(0,0), p, q$. The sign is determined by the "right hand" rule. This means that if the angle at $(0,0)$ starting at p, going clockwise around $(0,0)$ and ending at q is less than 180 degrees, the cross product is negative, otherwise it's positive. It's zero of the angle between p and q is 0 or 180 degrees.



There is also a formula analogous to the dot product angle formula for the cross product.

Theorem: Cross-product-angle formula

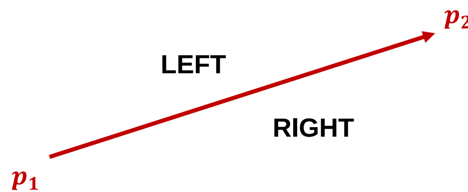
For any 2D vectors u, v, we have

$$u \times v = \|u\| \|v\| \sin(\theta)$$

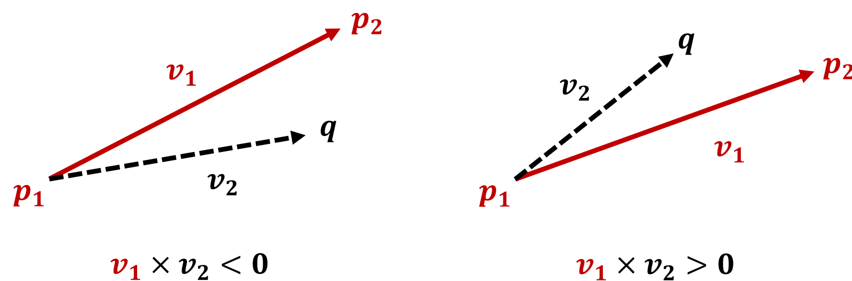
where θ is the angle between the vectors u and v.

21.2.2 Line-side test

The line-side test is the next most fundamental primitive we will introduce, and turns out to be a critical ingredient in a lot of geometry algorithms. Given three points p_1, p_2, q, the output of the *line side test* is "LEFT" if the point q is to the left of ray $p_1 \to p_2$, "RIGHT" if the point q is to the right of ray $p_1 \to p_2$, and "ON" if it is on that ray.



The algorithm is to construct vectors $v_1 = p_2 - p_1$ and $v_2 = q - p_1$, then take the cross product of v_1 and v_2 and look at its value compared to 0.



Notice that if a point q is on the right, then the cross product $v_1 \times v_2$ will be negative, or if q is on the left, it will be positive. If q happens to be on the ray, then the cross product is zero.

Algorithm: Line-side test

```
let v1 = p2 − p1 and v2 = q − p1
if v1 × v2 < 0 then return RIGHT
else if v1 × v2 > 0 then return LEFT
else return ON
```

21.2.3 Convex combinations

Since we now know how to add and scale points/vectors, lets see what kinds of other objects we can make just by combining points together in various ways. Our focus will be in 2D, but you could also think about what some of these constructions look like in 3D if you want. We will consider *convex combinations* of points.

Definition: Convex combination

A *convex combination* of the points $p_1, p_2, \ldots p_k \in \mathbb{R}^d$ is a point

$$p' = \sum_{i=1}^{k} \alpha_i p_i,$$

such that $\sum a_i = 1$ and $\alpha_i \geq 0$.

Given this definition, an interesting question to ask is given a set of points, what does the set of *all possible convex combinations* of those points look like? We can start with just two points.

Claim: A convex combination of two points

Given two points p and q, any convex combination of p and q is a point on the line connecting p and q. More specifically, the set of all convex combinations of p and q is the line between p and q.

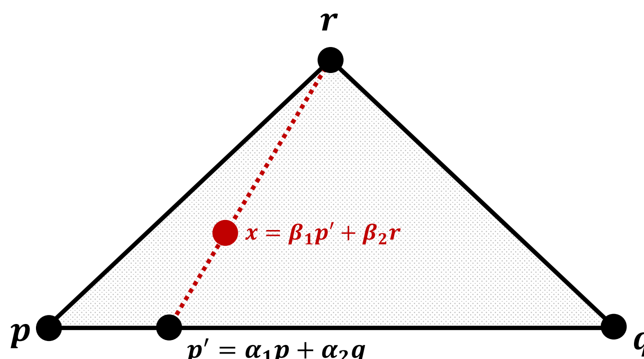
For example, if we take two points p and q and set $\alpha_1 = \alpha_2 = \frac{1}{2}$, we get $\frac{1}{2}p + \frac{1}{2}q$, which is the midpoint of p and q (the middle of the line connecting them). What if we take three points p, q, r?

Claim: A convex combination of three points

The convex combinations of three points p, q, r form the triangle with vertices p, q, r.

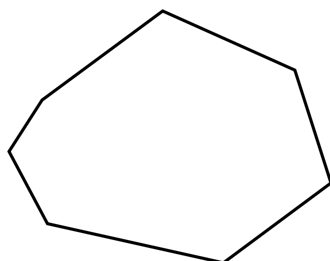
How can we justify this? Lets use the fact from above that the convex combinations of two points are the line between them. So, for any point on the line between p and q, we can form a convex combination that achieves that point. Now, for any point inside the triangle pqr, say x, that we want to reach, we can first take a convex combination $p' = \alpha_1 p + \alpha_2 q$ such that p' is the intersection of the lines pq and rx. Then take a second convex combination with that

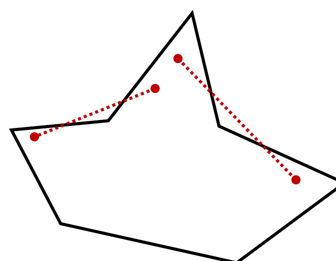
point and r, to get $x = \beta_1 p' + \beta_2 r$.



21.3 The Convex Hull

This is the "sorting problem" of computational geometry. There are many algorithms for the problem, and there are often analogous to well-known sorting algorithms. There are also well studied lower bounds! A point set $P \subseteq \mathbb{R}^d$ is *convex* if it is closed under convex combinations. That is, if we take any convex combination of any two points in A, the result is a point in P. In other words, when we walk along the straight line between any pair of points in P we find that the entire path is also inside of P.


Convex


Non-convex

We saw convex sets before when we talked about linear programming. Based on this definition, we can define the *convex closure* of a set of points P to be the smallest convex set containing P. This is well-defined and unique for any point set P. This brings us to the definition of the object we really care about:

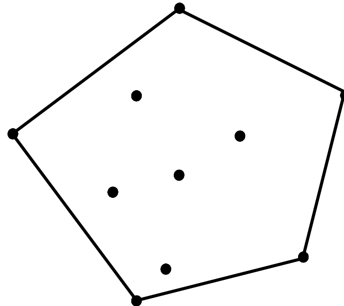
Definition: Convex hull

The *convex hull* of a set of points P is the boundary of the convex closure of P. That is, it is the smallest convex polygon that contains all of the points in P, either on its boundary or interior.

These definitions are general and apply to any closed set of points, including infinite sets, but for our purposes we're only interested in the ConvexClosure(P) and ConvexHull(P) when P is

a finite set of points.

A computer representation of a convex hull must include the combinatorial structure. In two dimensions, this just means a simple polygon in, say counter-clockwise order. (In three dimensions it's a planar graph of vertices edges and faces) The vertices are a subset of the input points. So in this context, a 2D convex hull algorithm takes as input a finite set of n points $P \in \mathbb{R}^2$, and produces a list H of points from P which are the vertices of the ConvexHull(A) in counter-clockwise order. This figure shows the convex hull of 10 points.



Today we're going to focus on algorithms for convex hulls in 2-dimensions. We first present an $O(n^2)$ algorithm, than refine it to run in $O(n \log n)$. To slightly simplify the exposition we're going to assume that no three points of the input are colinear.

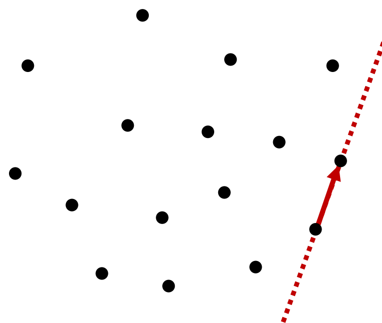
21.3.1 Warmup: Slow algorithms for 2D Convex Hulls

First we give a trivial $O(n^3)$ algorithm for convex hull. It might be slow, but it starts us off in the right direction by thinking about some useful facts about them.

Claim

A directed segment between a pair of points (p_i, p_j) is on the convex hull if and only if all other points p_k are to the left of the ray through p_i and p_j.

We illustrate this fact with the following "proof by picture".

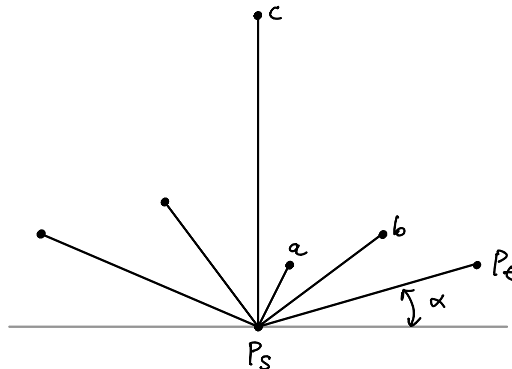


Note that no point to the right of the ray can be in the convex hull because that entire half-plane is devoid of points from the input set. And the points on the segment (p_i, p_j) are in the ConvexClosure of the input points. Therefore the segment is on the boundary of the ConvexClosure. Therefore it is on the convex hull.

An $O(n^3)$-time algorithm Using this observation, we get an $O(n^3)$ algorithm by brute-force. Just try every pair of points p_i, p_j and then line side test every other point p_k. If every p_k is on the left, then add the edge $p_i \to p_j$ to the hull. After adding all of the edges, sort them into counterclockwise order.

An $O(n^2)$-time algorithm To get this to run in $O(n^2)$ time we just have to be a bit more organized. In the first algorithm we just found all of the edges of the hull in an arbitrary order. Lets try to find them in order this time. This will cut down the amount of work we have to do.

The first observation is that if we take the point with the lowest y-coordinate, this point must be on the convex hull. Call it p_s. Suppose we now measure the angle from p_s to all the other points. These angles range from 0 to π. If we take the point p_t with the smallest such angle, then we know that (p_s, p_t) is on the convex hull. The following figure illustrates this.



All the other points must be to the left of segment (p_s, p_t). We can continue this process to find the point p_u which is the one with the smallest angle with respect to (p_s, p_t). This process is continued until all the points are exhausted. The running time is $O(n)$ to find each segment. There are $O(n)$ segments, so the algorithm is $O(n^2)$.

Actually we don't need to compute angles. The line-side-test can be used for this instead. For example look at what happens after we've found p_s and p_t. We process possibilities for the next point in any order. Say we start from a in the figure. Then we try b, and note that b is on the right side of segment (p_t, a) so we jettison a and continue with (p_t, b). But then we then throw out b in favor of c. It turns out that the remaining points are all to the left of segment (p_t, c). Thus $c = p_u$ is the next point on the convex hull.

21.3.2 Graham Scan, an $O(n \log n)$ Algorithm for 2D Convex Hulls

The $O(n^2)$ algorithm above is actually pretty close. We can convert it into an $O(n \log n)$ algorithm with a slight improvement. We went from $O(n^3)$ to $O(n^2)$ by finding the sides of the hull

in order, but we are still searching through the *points* in an arbitrary order to do so. So, instead of processing the points in an arbitrary order, lets process them in order of increasing angle with respect to the point p_s.

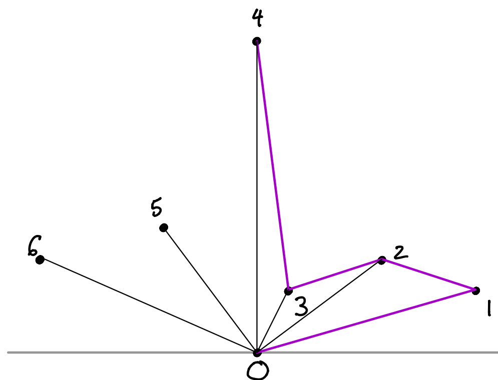
Let's relabel the points so that $p_0 = p_s$ is the starting point, and $p_1, p_2 \ldots$ are the remaining points in order of increasing angle with respect to p_0. From the discussion above we know that (p_0, p_1) is an edge of the convex hull. The Graham Scan works as follows.

Algorithm: Graham Scan

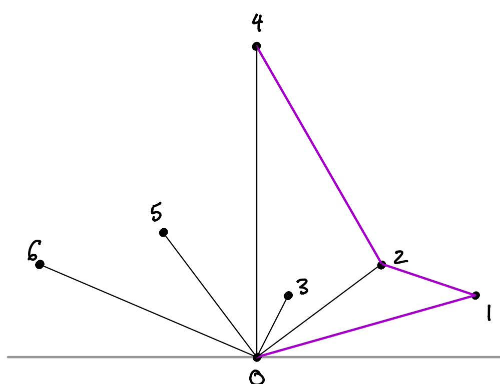
We maintain a "chain" of points that starts with $p_0, p_1, \ldots$. This chain has the property that each step is always a left turn with respect to the previous element of the chain. We try to extend the chain by taking the next point in the sorted order. If this has a left turn with respect to the current chain, we keep it. Otherwise we remove the last element of the chain (repeatedly) until the chain is again restored to be all left turns.

```
find lowest point p0
sort points p1,p2,... counterclockwise by their angle with p0
H = [p0, p1]
for each point i = 2,3,...
   while LineSideTest(H[-2],H[-1],pi) is RIGHT
      H.pop()
   H.append(pi)
return H
```

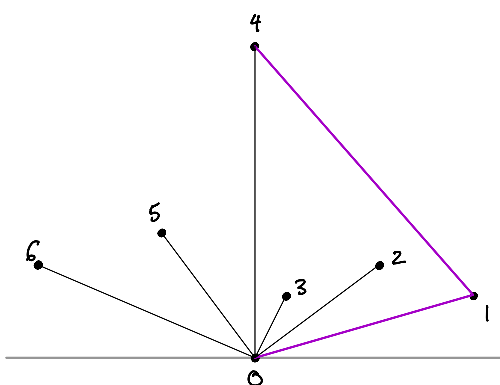
Here's an example of the algorithm.



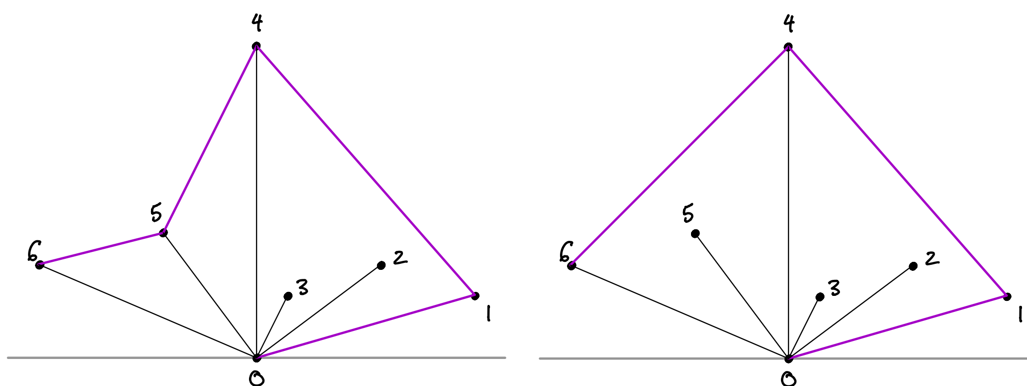
At this point we've formed the chain p_0, p_1, p_2, p_3, p_4. But the last step (from p_3 to p_4) is a right turn. So we delete p_3 from the chain. Now we have:



Now at p_2 we have a right turn, so we remove it, giving:



Now the process continues with points p_5 and p_6. When p_6 is added, p_5 becomes a right turn, so it's removed.



After all the points are processed in this way, we can just add the last segment from p_{n-1} to p_0, to close the polygon, which will be the convex hull.

Each point can be added at most once to the sequence of vertices, and each point can be removed at most once. Thus the running time of the scan is $O(n)$. But remember we already paid $O(n \log n)$ for sorting the points at the beginning of the algorithm, which makes the overall running time of the algorithm $O(n \log n)$.

The reason this algorithm works is because whenever we delete a point we have implicitly shown that it is a convex combination of other points. For example when we deleted p_3 we know that it is inside of the triangle formed by p_0, p_2 and p_4. Because of the presorting p_3 is to the left of (p_0, p_2), and to the right of (p_0, p_4). And because (p_2, p_3, p_4) is a right turn, p_3 is to the left of (p_2, p_4).

At the end the chain is all left turns, with nothing outside of it. Therefore it must be the convex hull.

21.3.3 Lower bound for computing the convex hull

We said earlier that convex hull is the "sorting" of computational geometry because there are a huge range of algorithms for it that use a wide range of different algorithmic techniques. Also like sorting, it admits a similar lower bound! We won't prove it here because we would need to address too many subtle details of the model, but the result is quite nice!

Theorem: Sidedness lower bound for convex hull

For any algorithm that computes a convex hull using line-side tests, at least $\Omega(n \log n)$ line-side tests are necessary.

The cool thing here is how we measure complexity. Just like how, for sorting, we counted the number of comparisons and showed a lower bound based on that, for convex hull, we can provide a lower bound in terms of the number of *line-side tests* that we need to perform. Just like we can also do sorting outside the comparison model and obtain different bounds, it is also possible to write convex hull algorithms that do not use line-side tests, so this result does not apply to those algorithms. It turns out, however, that the vast majority of known convex hull algorithm are based on line-side tests, so it applies quite widely!

Exercises: Geometry

Problem 47. Explain how to compute the projection of a point p in 2D onto any line L defined by two points a and b by using the algorithm from earlier with some extra steps.

Problem 48. Given a point p in 2D and a line L defined by two points a and b, describe how to compute the distance from p to the line L using the cross product. (One solution is to use the projection algorithm from earlier, but here we want a different solution.)

Problem 49. Suppose we consider $k > 3$ points $p_1, \ldots, p_k$. Describe the set of points formed by all convex combinations of them. Justify your answer by using the fact above that the convex combinations of three points span a triangle.

Problem 50. Given two line segments defined by the points a and b and the points c and d respectively, determine whether the two of them intersect or not. Hint: Use the line-side test primitive.

Lecture 22

Computational Geometry: Randomized Incremental Algorithms

In this lecture, we will see how the technique of *randomized incremental algorithms* can be applied to solve problems in computational geometry. Randomized incremental algorithms are a powerful technique for improving algorithms that otherwise have a slow worst-case bound against an adversarial input by introducing randomness in the order of the the input to defeat the adversary. We will then use probability to prove bounds on the expected running time of these algorithms.

Objectives of this lecture

In this lecture, we will

- see how randomized incremental algorithms can be used for computation geometry
- give an expected linear-time algorithm for the *closest pair* problem
- give an expected linear-time algorithm for the *smallest enclosing circle* problem

22.1 Model and assumptions

In this lecture we make the following assumptions. Like last time, we want to operate on real numbers, but we are now assuming a little bit more than last time.

- We assume the points are presented as real number pairs (x, y).
- We assume arithmetic on reals is accurate and runs in $O(1)$ time.
- We will assume that we can take the floor function of a real in $O(1)$ time.
- We also assume that hashing is $O(1)$ time in expectation.

These assumptions (in this context) are reasonable, because the algorithms will not abuse this power.

22.2 The closest pair problem

Problem: Closest pair

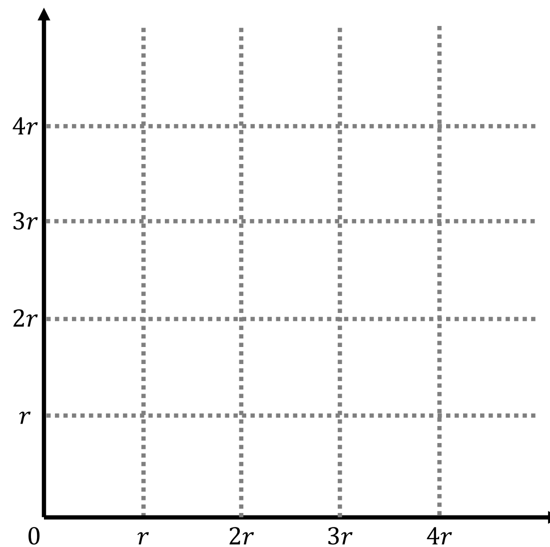
For any set of points P, let $\mathrm{CP}(P)$ be the closest pair distance in P, i.e.,

$$\mathrm{CP}(P) = \min_{\substack{p,q \in P \\ p \neq q}} \|p - q\|$$

Given a point set P, our goal is to compute $\mathrm{CP}(P)$.

The naive algorithm of trying all pairs of points will cost $O(n^2)$ time. Our goal today is to come up with a faster algorithm. We will see that with some nifty randomization, we can actually achieve linear time!

A "grid" data structure We're going to define a "grid" data structure, and an API for it. The grid (denoted G) stores a set of points (that we'll call P) inside square cells of size $r \times r$. The number r is called "the grid size of G". The point (x, y) therefore goes in the grid cell $(\lfloor x/r \rfloor, \lfloor y/r \rfloor)$.



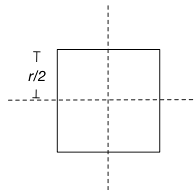
Now how does this thing help us find the closest pair of points? Well, intuitively, if we choose the grid size large enough, then the closest pair of points should end up in either the same grid cell, or in neighboring grid cells. Of course, if the grid size is *too large*, then there will be too many points in some cells, and finding the closest pair of points in a cell will take $O(n^2)$ time again...

So, our goal is to somehow figure out just the right grid size such that it is small enough for there to be very few points per cell, but large enough such that the closest pair of points are in nearby cells. If we can do that, we'll find an efficient algorithm.

Choosing the right grid size Lets first focus on the criteria that we want the closest pair of points to live in the same or in neighboring grid cells. If the grid size is much smaller than the distance between the closest pair of points, then they might end up being very far away, so intuitively we want to choose the the grid size $r \approx \text{CP}(P)$. What if we just choose exactly $r = \text{CP}(P)$? I claim that this has all of the nice properties we want.

> ***Claim 22.1: The right grid size***
>
> Given a grid G with points P and grid size $r = \text{CP}(P)$, no cell contains more than four points.

Proof. Imagine splitting a given cell into four $r/2$-sized sub-boxes.



The diameter of each sub-box is $\sqrt{2}r/2 < r$, so none of these sub-boxes can have more than one point in it, or that would imply that there exists a pair of points whose distance is less than r, which would contradict that $r = \text{CP}(P)$. Therefore there are at most four points in each cell. □

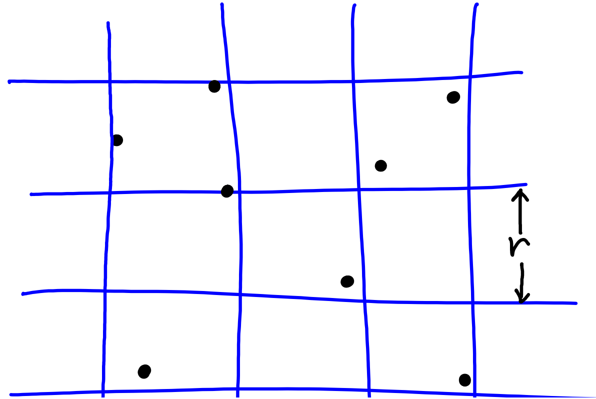
Okay great, so the perfect grid size is just to pick $r = \text{CP}(P)$, and then we're good! Oh wait... $\text{CP}(P)$ is exactly what we were trying to compute in the first place... What can we do? Lets try to solve the problem *incrementally*.

22.2.1 An incremental approach

We will start with just two points p_1 and p_2 and put them in a grid with $r = \|p - q\|$. Then we will continuously add the rest of the points into the grid, and if we violate the invariant that r is equal to the closest pair distance, we will just throw away the old grid and build a new one from scratch. Lets go into a bit more detail for each of these operations. Our grid will support the following API:

- MakeGrid(p, q): Make and return a new grid containing points p and q using $r = \|p-q\|$ (the distance between p and q) as the initial grid size.
- Lookup(G, p): p is a point, G is a grid. This returns two types of answers. Let r' be the closest distance from p to a point in P. If $r' < r$ then return r'. If $r' \geq r$ return "Not Closest". Note that Lookup() does not need to compute r' if $r' \geq r$. Essentially, the lookup function detects whether we have found a new closest pair.
- Insert(G, p): G is a grid. p is a new point not in the grid. This inserts p into the grid. It returns the new grid size.

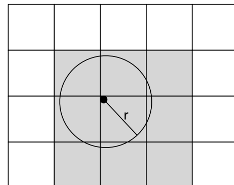
Here's an example Grid data structure satisfying the invariant.



We now discuss how to implement this API. Note that the grid could have arbitrarily many cells depending on the values of the point coordinates, so we can not afford to store them all explicitly. However, the vast majority of them will be empty, so we can store the grid using a dictionary data structure, powered by a hashtable to get $O(1)$-time operations. Specifically, we will maintain a hashtable whose keys are integer pairs (i, j), representing grid cells in the grid. Given a point (x, y), it belongs to the cell with key $(\lfloor x/r \rfloor, \lfloor y/r \rfloor)$.

MakeGrid(p, q) is simple. Just create a blank table, set $r = \|p - q\|$, and insert p and q .

Lookup(G, p) first computes the cell (i, j) where p belongs. It then looks in that cell, and the 8 surrounding ones, and computes the distance between p and every point in those neighboring cells. Call the smallest found distance r'. If $r' < r$, then we have found a new closest pair, so we return r'. Otherwise, if $r' > r$ then return "Not Closest". This works because we know that if there is a point closer to p than r, it must be in one of the 9 cells that are searched by this function:



Also note that the running time of this is $O(1)$ because it does 9 lookups in the hash table, and the total number of points it has to consider is at most 36. This is because a cell contains at most 4 points by Claim 22.1.

Insert(G, p) works as follows. It first does a Lookup(G, p). If the result is "Not Closest" it just inserts p into the data structure into the correct cell (i, j). This is correct, since it means that p does not create a new closest pair, so the grid size should be unchanged. This is $O(1)$ time. On the other hand if the Lookup() returns $r' < r$, then the algorithm has to throw away the current grid and start from scratch to build a new grid with size r'. This takes $O(i)$ time if there are i points now being stored in the data structure.

These algorithms are correct because they maintain the invariant that the grid size r is always

equal to the closest pair distance.

Runtime analysis In the worst case, every newly inserted point might cause the closest pair to change, and hence require a regrid, so the runtime is

$$\sum_{i=1}^{n} i = \Theta(n^2).$$

Unfortunately this is no better than the brute-force approach! But how likely is it that we get so unlucky to have the closest pair change every iteration? What can we do to make this unlikely for any possible input?

22.2.2 Randomized $O(n)$ algorithm for closest pair

We can use the same approach as Seidel's 2D LP algorithm! Lets *randomly shuffle* the input points, then run the above algorithm. The claim is that by randomly shuffling, the probability that an insertion causes the closest pair to change is low.

Algorithm: Randomized incremental closest pair

```
RandomizedClosestPair(P) {
   Randomly permute the points. Call the new ordering p_1, p_2, ..., p_n.
   G = MakeGrid(p_1, p_2)
   for i = 3 to n do {
      r = Insert(G, p_i)
   }
   return r
}
```

Claim: Randomized incremental closest pair is fast

The algorithm runs in expected $O(n)$ time.

Proof. Consider a random permutation $\pi_1, ..., \pi_n$ of the points, and denote the prefix of the first i points in the corresponding shuffled order as

$$P_i = \langle p_{\pi_1}, \ldots, p_{\pi_i} \rangle$$

Recall the time to do Insert() is $O(1)$ if the grid size does not change, and $O(i)$ (i = the number of points in the grid) if the grid size does change. Let us define an indicator random variable

$$X_i = \begin{cases} 1 & \text{if } \mathrm{CP}(P_i) \neq \mathrm{CP}(P_{i-1}), \\ 0 & \text{otherwise.} \end{cases}$$

The running time of the algorithm is

$$T = O\left(\sum_{i=2}^{n} (1 + X_i \cdot i) \right),$$

so by linearity of expectation, the expected running time is

$$\mathbb{E}[T] = O\left(n + \sum_{i=2}^{n} i \cdot \mathbb{E}[X_i]\right) = O\left(n + \sum_{i=2}^{n} i \cdot \Pr[X_i = 1]\right).$$

All we have to do is bound $\Pr[X_i = 1]$ and we can complete the proof. Consider the points P_i, and call a point *critical* if $\mathrm{CP}(P_i \setminus \{q\}) > \mathrm{CP}(P_i)$. How many critical points can there be?

- If there are no critical points, then $\Pr[X_i = 1] = 0$.
- If there is one critical point, then $\Pr[X_i = 1/i]$ since the critical point would have to be the final element of P_i, which is randomly shuffled, and hence that happens with probability $1/i$.
- If there are two critical points, then $\Pr[X_i = 2/i]$ since either one of the two critical points would have to be the final element of P_i.

The final claim is that there can not be more than two critical points. If p and q are both critical, then $\|p - q\| = \mathrm{CP}(P_i)$, so if there were a third critical point r, its removal could not lower the closest pair distance. Therefore

$$\mathbb{E}[T] = O\left(n + \sum_{i=2}^{n} i \cdot \frac{2}{i}\right) = O(n + 2n) = O(n).$$

Thus the expected cost of the entire algorithm is $O(n)$. □

22.3 The smallest enclosing circle problem

Problem: Smallest enclosing circle

Given $n \geq 2$ points in two dimensions, find the smallest circle (by radius) that contains all of the points.

The smallest enclosing circle is kind of like the convex hull. We are looking for a body that encloses all of the given points, except that this time, instead of searching for a polygon, we are looking for a circle. For notation purposes, lets define $\mathrm{SEC}(p_1, \ldots, p_n)$ to be the smallest enclosing circle of the points $p_1, \ldots, p_n$.

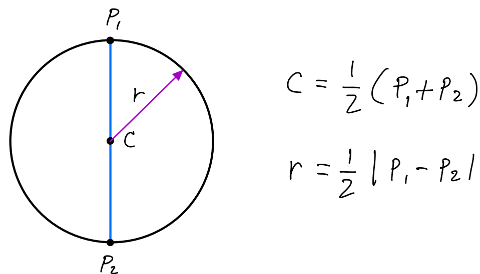
As usual, we will sweep edge cases under the rug by assuming that there are no sets of three colinear points in our input. Before we dive right into the algorithm, lets look at some useful properties of circles.

22.3.1 Base cases and useful properties of circles

Two points Suppose we start with exactly two points p_1 and p_2. There are infinitely many possible circles that enclose p_1 and p_2, and our algorithm can not try an infinite number of things, but we can restrict ourselves to a more reasonable set of possibilities.

Here's an idea that might seem obvious when we're just considering two points, but turns out to be one of the most powerful and useful observations when deriving computational geometry algorithms: pick a circle that touches the points. If I give you a circle that encloses the two points but doesn't touch them, then it can't be the SEC because I could shrink it and it will still contain the points.

How many circles are there that touch the points p_1 and p_2? Unfortunately there's still infinitely many of them, but there is a unique *smallest* circle that contains them, where p_1 and p_2 are on a diameter:



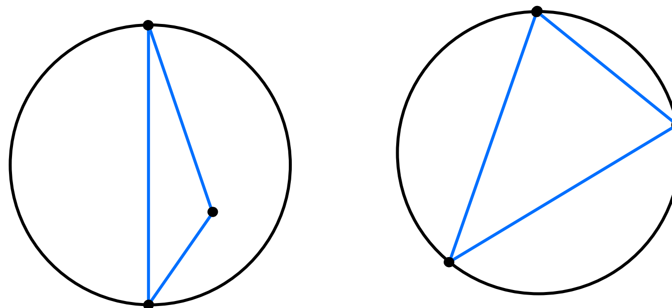
Three points Three points is where things get interesting. In the case of two, there were infinitely many possible circles that touch them, but it turns out that for three non colinear points, there exists a single unique circle that touches them.

Claim 22.2: Three points make a circle

Given three non-colinear points, there exists a unique circle that touches them.

You can try to prove this as an exercise.

Is this point always the smallest enclosing circle of the three points though? It might not be. There are two cases, either the triangle is acute, and the smallest enclosing circle will touch all three points, or it is obtuse, and the smallest enclosing circle will just touch two of them.



We can just try both cases, which is a constant number of choices, and we will have the smallest enclosing circle for three points.

22.3.2 The general case

Now we come to the general case, given $n > 3$ points, how do we find the smallest enclosing circle? Like before, we are faced with infinity many possible circles to choose from, so even if we wanted to implement a brute-force approach, it is not clear how to. We need to somehow reduce our infinite set of choices to a finite one... To do so, we will prove the following claim:

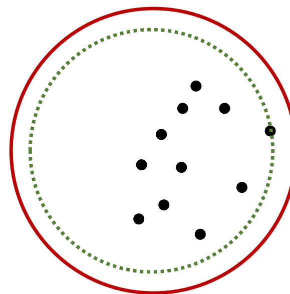
Claim 22.3: Three points defines the smallest enclosing circle

For any set of points, $\text{SEC}(p_1, \ldots, p_n)$ either touches two points p_i and p_j at opposite ends of a diameter, or touches three points p_i, p_j, p_k that form an acute triangle.

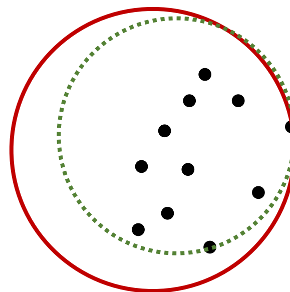
In other words, there exists i, j, k such that $\text{SEC}(p_1, \ldots, p_n) = \text{SEC}(p_i, p_j, p_k)$.

Proof. We'll consider a few cases:

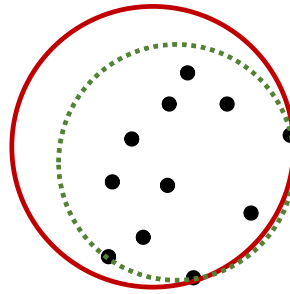
1. **No points:** Suppose I give you an enclosing circle (a circle that contains all of the points) but it doesn't touch any of them. Then I can shrink the circle by some ε until it does touch something and it still contains all the points, so it can't be the SEC.



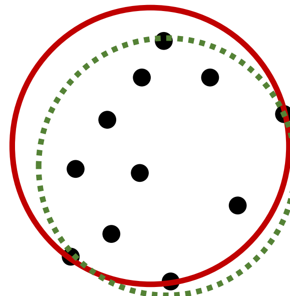
2. **One point:** Suppose I give you an enclosing circle that touches just one point. Then I can shrink the circle by some ϵ and translate it so that it still touches that one point, so it can't be the SEC.



3. **Two points:** Suppose an enclosing circle touches two points and these points are on opposite ends of a diameter. Then this is a lower bound on the size of any enclosing circle since it must contain these points, hence the circle is optimal. Suppose instead that the two points are not on opposite ends of a diameter, i.e., there is a greater than 180 degree gap between them. Then once again, we can shrink the circle by some ε and translate it.



4. **Three or more points:** If the circle touches at least three points but none of them form an acute triangle, we can again shrink the circle and translate it.



Therefore an SEC must either touch two points at a diameter, or touch three points of an acute triangle. □

What this claim is saying is that the smallest enclosing circle is determined by just some subset of three of the points in the input! So we do not have to try infinitely many possible circles to find the SEC, we have reduced to a finite number of possibilities. This gives us an $O(n^3)$-time brute-force algorithm: just try every triple of points and find the smallest enclosing circle, then return the largest one.

Note that we return the **largest one**, not the smallest one, because the smallest one might not contain all of the other points, but the largest one is guaranteed to, by Claim 22.3.

22.3.3 Beating brute force: an incremental algorithm

Continuing with the theme of this lecture, lets try to develop an incremental algorithm for the problem. That means we will start with just two points and find their SEC. Then, one by one we will add in another point and check whether it is contained within the current circle. If it is, we are good to continue. If it is not, we must find the new smallest enclosing circle of all the points. If we are lucky (or clever), the algorithm won't need to do the slow case very often.

Building an incremental algorithm Suppose we've computed the SEC of $p_1, \ldots, p_{i-1}$. How do we add the next point p_i? We have to consider two cases:

1. **Case 1:** p_i is inside SEC($p_1, \ldots, p_{i-1}$): In this case, we don't have to do anything because the circle still contains all of the points, and adding a new point can not make the SEC smaller.

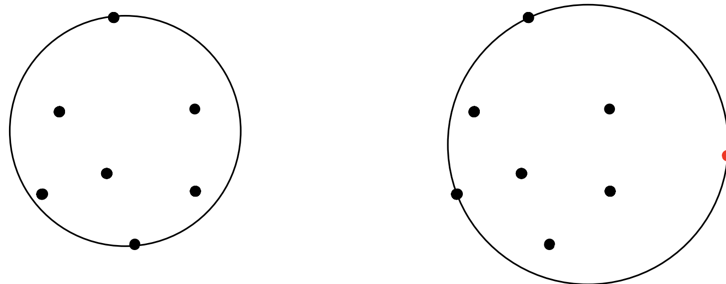
2. **Case 2:** p_i is not inside SEC($p_1, \ldots, p_{i-1}$): This is the hard case. We could just decide to throw away the current circle and find the new SEC from scratch with brute force, but this would end up being really slow, even with the incremental approach, so we're going to need something more sophisticated.

Here's an observation that will help:

Claim: A new point locks in the new SEC

For a set of points $p_1, \ldots, p_i$, if $\text{SEC}(p_1, \ldots, p_{i-1}) \neq \text{SEC}(p_1, \ldots, p_i)$, then p_i is on the boundary of the new SEC.

Proof. If p_i were not on the boundary, then by Claim 22.3, the SEC is determined by either two points along a diameter, or three points, but these points were all in the set before p_i was added, so the SEC shouldn't have changed. □



This allows us to improve on using brute force. If we insert a new point p_i and discover that it is outside the current SEC, we don't need to completely go from scratch and spend $O(i^3)$ time, we could lock p_i in and just do $O(i^2)$ work to find the other two points. In fact, we will see momentarily that we can do even better. To make this concrete, lets define a subroutine SEC1, which takes two parameters, a list of points $p_1, \ldots p_{i-1}$, and a point p_i, and finds the smallest enclosing circle of $p_1, \ldots, p_i$, but using the knowledge that p_i is definitely on the boundary. So, our algorithm for SEC currently looks like

```
SEC([p1, p2, ..., pn]) = {
   let C be the SEC of p1 and p2 (formed by a diameter between p1 and p2)

   for i = 3 to n do {
      if pi is not inside C then C ← SEC1([p1,...,p(i−1)], pi)
   }
   return C
}
```

Implementing SEC1 We could implement SEC1 by trying all $O(i^2)$ pairs of other points and taking the best SEC that results, but this would take $O(i^2)$ time. We can still do much better than this! How? Same exact idea as the outer SEC algorithm! Lets solve SEC1 incrementally as well. Suppose we are given SEC1($[p_1, \ldots, p_n], q$) to solve.

- We can start with a circle around p_1 and q, then incrementally add the remaining points, one by one.
- If a point p_i is outside the current SEC, then we have to rebuild a new SEC.
- However, from the same argument as before, if this happens, we know for sure that p_i is on the boundary of the new one. So, we now need to solve the problem of finding the SEC of a set of points with *two fixed points* p_i and q. Hmm... time for another subroutine!

```
SEC1([p₁,p2,...,pₖ],q) = {
   let C be the SEC of p₁ and q (formed by a diameter between p₁ and q)

   for i = 2 to k do {
      if pᵢ is not inside C then C ← SEC2([p₁,...,pᵢ₋₁], pᵢ, q)
   }
   return C
}
```

Here, SEC2($[p_1,\ldots p_k], q_1, q_2$) computes a smallest enclosing circle of $\{p_1,\ldots p_k, q_1, q_2\}$ with the knowledge that q_1 and q_2 must be on the boundary.

Implementing SEC2 Continuing the pattern, the last level subroutine is the simplest. We have two given points q_1, q_2 that are forced to be on the boundary, so we just need to loop through the k given points and pick the best third point. No more sophisticated tricks needed at this point!

```
SEC2([p₁,p2,...,pₖ],q₁,q₂) = {
   let C be the SEC of q₁ and q₂ (formed by a diameter between q₁ and q₂)

   for i = 1 to k do {
      if pᵢ is not inside C then C ← Circle that touches (pᵢ,q₁,q₂)
   }
   return C
}
```

Runtime analysis SEC2 always takes exactly $O(k)$ time no matter what. For SEC1, in the worst case, every new point triggers the need to find a new SEC, so SEC1 takes

$$\sum_{i=1}^{k} i = \Theta(k^2)$$

time. Similarly, in the worst case, SEC needs to find a new circle every iteration, so it takes

$$\sum_{i=1}^{n} i^2 = \Theta(n^3)$$

time. This is again, no better than brute force. Now its time to fix the problem with our favorite technique of the day, randomization!

22.3.4 Saving the day with randomization (again)

Lets add one additional line of code to SEC and SEC1 that randomly shuffles the points p, and then proceeds exactly as written. The almost-too-good-to-believe claim is that this brings us right down to linear time!

Claim: Randomized incremental smallest enclosing circle is linear time

SEC1 runs in $O(k)$ expected time, and SEC runs in $O(n)$ expected time.

Proof. Since SEC2 uses no randomization, it still always runs in $O(k)$ time deterministically. To analyze SEC1 and SEC, we use the exact same kind of technique as before, so we will omit the tedium of writing out all the indicator variables and get straight to the point.

Recall from Claim 22.3 that two or three points determine the SEC of the whole point set. So if we delete a point randomly among i points, the probability that the SEC changes is at most $\frac{3}{i}$. Therefore, the expected cost of one step of SEC1 is

$$\frac{3}{i} \times O(i) + \frac{i-3}{i} \times O(1) = O(1).$$

By linearity of expectation, the total expected cost of SEC1 is $O(k)$. By the same reasoning, the expected cost of one step of SEC is $O(1)$, and by linearity of expectation, the total expected cost is $O(n)$ □

Remark: Fun fact: less randomness suffices

In the algorithm described, we randomly permuted the points p in SEC and SEC1. It turns out that a single random permutation at the beginning of SEC is enough, and we don't actually need to permute inside SEC1. The proof becomes more difficult though because we don't have independence between different calls of SEC1 anymore, so we won't prove it.

Exercises: Randomized Incremental Geometry

Problem 51. Prove the claim that there is a unique circle that inscribes three non-colinear points.

Lecture 23

Splay Trees

Today's lecture will focus on a very interesting case study of amortized analysis, a powerful binary search tree (BST) data structure called the *Splay Tree*. Splay trees have a lot of nice practical properties, they make many problems that are tricky to solve using ordinary BSTs much easier. They also have many nice theoretical properties which show that they are often provably more efficient than other BSTs for several common use cases. We will focus mainly on the analysis, which makes heavy use of the potential method from the previous lecture.

Objectives of this lecture

In this lecture, we want to:

- Review binary search trees (BSTs)
- Understand the functionality of the Splay tree data structure
- Analyze the cost of the Splay tree data structure using the potential method

Recommended study resources

- `http://www.link.cs.cmu.edu/splay/`

23.1 Binary Search Trees

These lecture notes assume that you have seen binary search trees (BSTs) before. They do not contain much expository or backtround material on the basics of BSTs. Binary search trees are a class of data structures where:

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

Binary search trees are a way to store and update a set of items, where there is an ordering on the items. In general, there are two classes of applications. Those where each item has a key value from a totally ordered universe, and those where the tree is used as an efficient way to represent an ordered list of items.

Some applications of binary search trees:

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

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

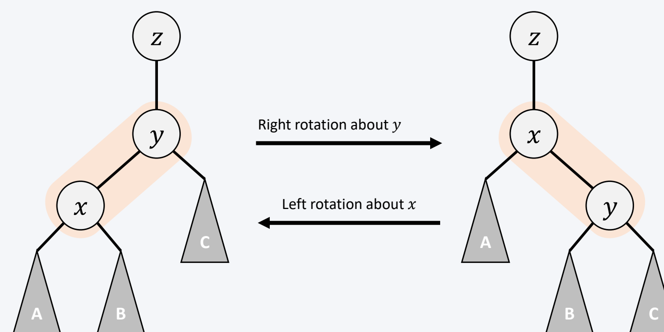
23.1.1 Rotations

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

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

Key Idea: Rotations

A *left rotation* and a *right rotation* are defined by the following schematic:



In a right rotation, a left child takes the place of its parent, and vice versa.

In these notes, we are defining a rotation with respect to the *parent* node of the rotation, e.g., a right rotation about y means to move the left child of y up, and a left rotation about x means

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

to move the right child of x up. This is just a convention, and others exist. Some sources will define rotations with respect to the lower node, for example, they might call the two rotations above a rotation of x and a rotation of y respectively.

23.2 Splay Trees (self-adjusting search trees)

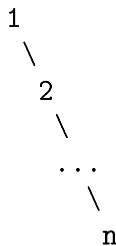
These notes just describe the *bottom-up splaying* algorithm, the proof of the access lemma, and a few applications. The key of idea of splay trees is *locality*. Locality is a common property of many real-world problems, which essentially says that data that was accessed recently is likely to be accessed again soon. How can we translate this idea into a binary search tree algorithm? Here's one way:

Key Idea: Move recently accessed nodes to the root

Whenever we access a node in the tree, move it to the root.

At a high level, every time a node is accessed in a splay tree, it is moved to the root of the tree. The exact way in which it is moved to the root of the tree turns out to be important, as doing so naively can be inefficient. The simplest way to move an element to the root would be to rotate with respect to its parent over and over again until it became the root. The following exercise should help you see why this is not very efficient. What would a worst-case sequence of accesses look like under this scheme?

Problem 52. Inefficient move-to-root algorithm Starting with a tree of height n consisting of the elements 1 through n:

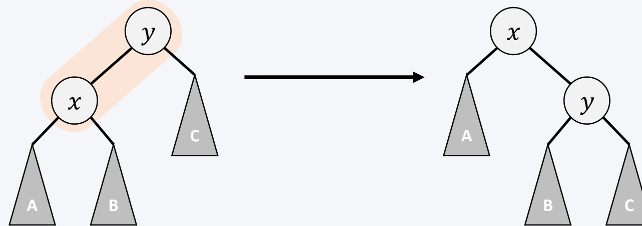


show what happens if you move n to the root by performing a sequence of left rotations. If we start with this tree and sequentially rotate n, then $n-1$, then $n-2$, etc. to the root, the tree that results is the same as the starting tree, but the total work is $\Omega(n^2)$, for an amortized lower bound of $\Omega(n)$ per operation.

We need to move the node to the root using a more sophisticated pattern to avoid this cost. This pattern is called *splaying*. We will show that the amortized cost of the splay operation is $O(\log n)$. We'll describe the algorithm by giving three rewrite rules in the form of pictures. In these pictures, `x` is the node that was accessed (that will eventually be at the root of the tree). By looking at the structure of the 3-node tree defined by `x`, `x`'s parent (`y`), and `x`'s grandparent (`z`) we decide which of the following three rules to follow. We continue to apply the rules until `x` is at the root of the tree.

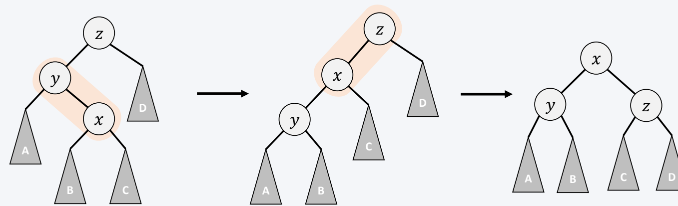
Definition: The zig rule

If we are splaying a node x, and its parent y is the root, we apply the zig rule. The zig rule is just a left or right rotation about y.



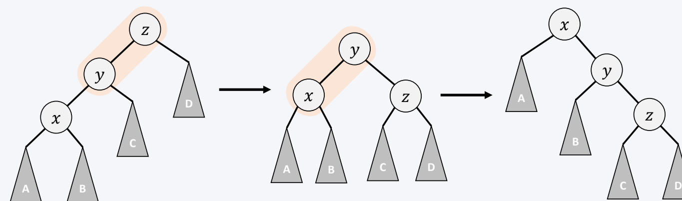
Definition: The zig-zag rule

If we are splaying a node x whose parent is y and grandparent is z, and x and y are not both left children or both right children, we apply the zig-zag rule. The zig-zag rule is two consecutive rotations with respect to x's parent (i.e., a rotation with respect to y followed by a rotation with respect to z).



Definition: The zig-zig rule

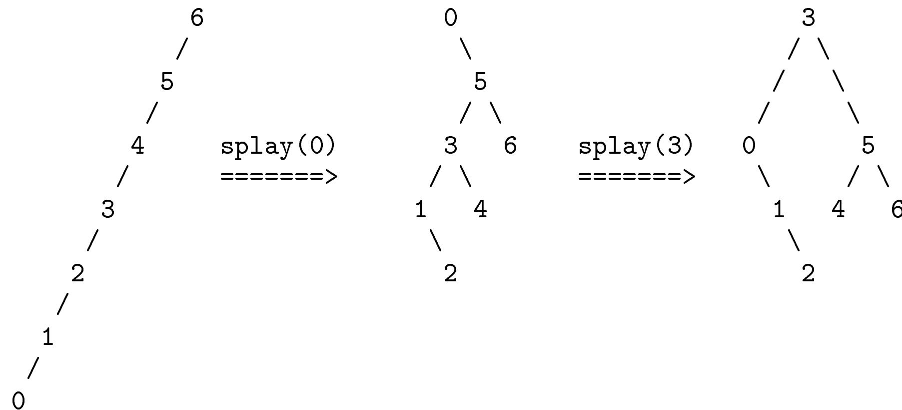
If we are splaying a node x whose parent is y and grandparent is z, and x and y are both left or both right children, we apply the zig-zig rule. The zig-zig rule is a rotation with respect to z followed by a rotation with respect to y.



Each of these rules has an exact mirror image variant which we do not depict. Note that the zig rule is just an ordinary rotation that we saw earlier, while the zig-zag and zig-zig rules are compositions of two separate rotations. The zig-zag case just rotates about x's parent twice, which is exactly the same thing that we argued doesn't always work. The magic idea that makes splay trees work is the zig-zag step. Instead of rotating about x's parent twice, it first rotates

about x's grandparent, and then about x's parent. This tiny subtle change turns out to make all the difference in making it into an efficient algorithm.

A *splay step* refers to applying one of these rewrite rules. Later we will be proving bounds on number of splay steps done. Each such step can be executed in constant time. There are a number of alternative versions of the algorithm, such as top-down splaying, which may be more efficient than the bottom-up version described here. Here are some examples:



Observe that unlike in the naive move-to-root algorithm, splaying 0 here actually decreased the height of the tree, making it more balanced.

23.3 Standard BST Operations Using Splaying

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

Split Say we are given a key x which exists in the tree, and we want to split the tree into two trees, such that the first contains all keys in the tree that are at most x, and the second contains all keys that are larger than x. To do so, we splay the node containing x to the root, then remove its right subtree.

Join Here we have two trees, say, A, and B. We want to put them together with all the items in A to the left of all the items in B. This is called the *Join* operation. This is done as follows. Splay the rightmost node of A. Now make B the right child of this node.

23.4 Analysis of Splaying

To analyze the performance of splaying, we start by assuming that each node x has a weight $w(x) > 0$. These weights can be chosen arbitrarily. For each assignment of weights we will be

able to derive a bound on the cost of a sequence of accesses. We can choose the weight assignment that gives the best bound. By giving the frequently accessed elements a high weight, we will be able to get tighter bounds on the running time. Note that the weights are only used in the analysis, and do not change the algorithm at all. (A commonly used case is to assign all the weights to be 1.)

23.4.1 Sizes and Ranks

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

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

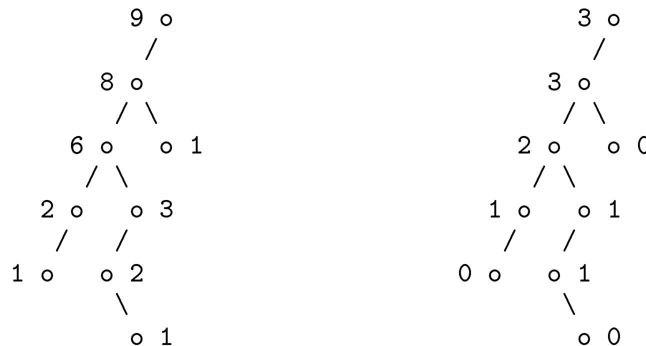
Now we define the *rank of a node* x as

$$r(x) = \lfloor \log_2(s(x)) \rfloor$$

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

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

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



Notes about this potential Φ:

1. Doing a rotation between a pair of nodes x and y only effects the ranks of the nodes x and y, and no other nodes in the tree. Furthermore, if y was x's parent before the rotation, then the rank of y before the rotation equals the rank of x after the rotation.
2. Assuming all the weights are 1, the potential of a balanced tree is $O(n)$, and the potential of a long chain (most unbalanced tree) is $O(n \log n)$.

3. In the banker's view of amortized analysis, we can think of having $r(x)$ tokens on node x. (Note however that the ranks could be negative if the weights are less than 1.)

23.5 The Amortized Analysis

Lemma 23.1: Access Lemma

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

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

Proof. Throughout the proof, keep in mind that we are counting splay steps as our cost. To do the proof, we are going to analyze each splay step individually. We will show the following lemma:

Lemma: Cost of a splay step

The amortized cost of a zig-zag or zig-zig splay step is $3(r(z) - r(x))$. The cost of the zig splay step is $3(r(y) - r(x)) + 1$.

We can express this a little bit differently. Take the zig-zag amortized cost $3(r(z) - r(x))$ mentioned above. We can write this as $3(r'(x) - r(x))$, where $r'(x)$ is the rank of x after the splay step, and $r(x)$ represents the rank of x before the splay step. This identity holds because the rank of x after the step is the same as the rank of z before the step.

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

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

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

```
                2 o                     2 x
                 /                      / \
              2 o                    0 o   o 2
               /                           /
            2 o        splay(x)         2 o
             /         ==========>       / \
          2 x                         1 o   o 0
           / \                         /
        0 o   o 1                   0 o
             /
            o 0
    -----------------            ----------------
    Total:  9                           7
```

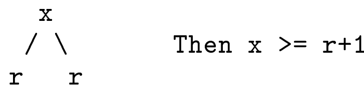
We allocated: 3(2-2)+1 = 1 tokens that are supposed to pay for the splay of x. There are two more tokens in the tree before than are needed at the end. (The potential decreases by two.) So we have a total of 3 tokens to spend on the splay steps of this splay. But there are 2 splay steps. So $3 > 2$,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

We will need one more lemma, called the Rank Rule.

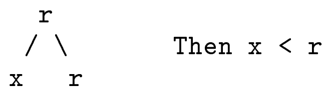
Lemma: The Rank Rule

Suppose that two siblings have the same rank, r. Then the parent has rank at least $r+1$.

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

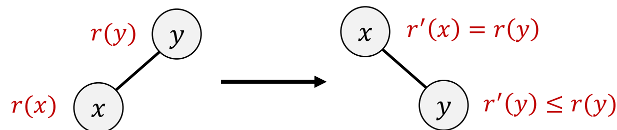


Conversly, suppose we find a situation where a node and its parent have the same rank, r. Then the other sibling of the node must have rank $< r$. So if we have three nodes configured as follows, with these ranks:



Now we can go back to proving the lemma. It remains to show these bounds for the individual steps. There are three cases, one for each of the types of splay steps.

The Zig Case: The following diagram shows the ranks of the two nodes that change as a result of the zig step. We need to show an amortized cost of at most $3(r(y)-r(x))+1$ to pay for the step.



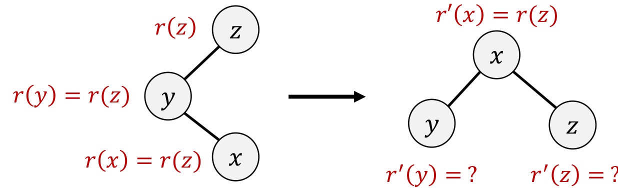
The actual cost is 1. Now, since x is the new root, $r'(x) = r(y)$. Furthermore, since y used to be the root, its rank can not have increased, so $r'(y) \le r(y)$. So the sum of the new ranks of x and y is at most $2r(y)$, and the sum of their old ranks is $r(y)+r(x)$, hence the difference in potential is at most

$$2r(y)-(r(y)+r(x)) = r(y)-r(x).$$

Hence the amortized cost is at most $(r(y)-r(x))+1 \le 3(r(y)-r(x))+1$.

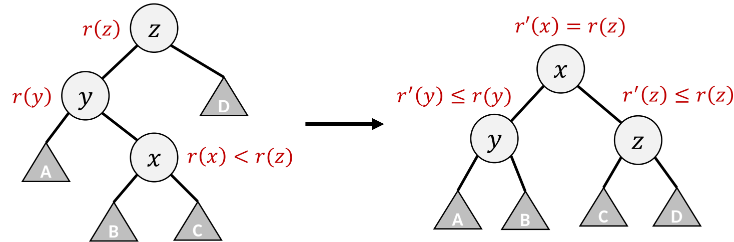
The Zig-zag Case: We'll split it further into 2 cases:

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



We know that $r'(x) = r(z)$, but by the rank rule, either $r'(y) < r(z)$ or $r'(z) < r(z)$. Therefore the sum of ranks is at least one lower than it was before the splay, so the potential has decreased by at least one. This pays for the actual cost of 1 of the splay, and hence the amortized cost of this step is zero, which is at most $3(r(z) - r(x)) = 0$.

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



First, observe that after the splay step, y and z both have a strict subset of their previous descendants, so their ranks can't have increased, i.e., $r'(y) \leq r(y)$ and $r'(z) \leq r(z)$. Since $r(x) < r(z)$, and since the ranks are integers, it must be true that $r(z) \geq r(x) + 1$.

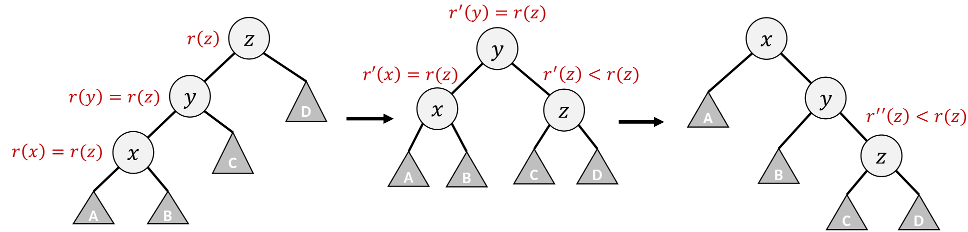
Now we write the difference in potentials as

$$
\begin{aligned}
\Delta\Phi &= r'(z) + r'(y) + r'(x) - r(z) - r(y) - r(x) \\
&\leq r(z) + r(y) + r(z) - r(z) - r(y) - r(x) && \text{(use } r'(y) \leq r(y),\ r'(z) \leq r(z),\ r'(x) = r(z)\text{)} \\
&\leq 2r(z) - r(z) - r(x) && \text{(cancel and simplify)} \\
&\leq 2r(z) - (r(x) + 1) - r(x) && \text{(use } r(z) \geq r(x) + 1\text{)} \\
&= 2(r(z) - r(x)) - 1 && \text{(simplify)}
\end{aligned}
$$

Adding the actual cost of 1, we get an amortized cost of at most $2(r(z) - r(x)) \leq 3(r(z) - r(x))$.

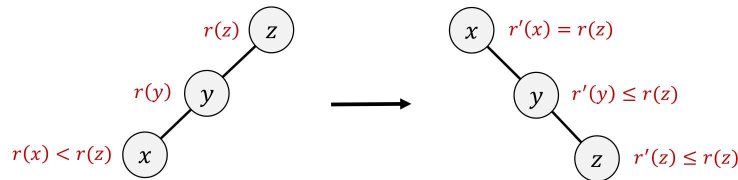
The Zig-zig Case: Again, we split into two cases.

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



Unlike the previous cases where we looked at only the starting and ending configurations, we will get some useful information from looking at the intermediate rotation. Since y is the root in the middle configuration, $r'(y) = r(z)$, and since x still has the same descendants it had before, $r'(x) = r(x) = r(z)$. By the rank rule, it must be true that $r'(z) < r(z)$. Then we look at the final configuration and notice that z still has the exact same descendants so $r''(z) = r'(z) < r(z)$. Therefore the total potential must decrease by at least 1, which pays for the actual cost of the splay, so we have $0 \le 3(r(z) - r(x)) = 0$ as required.

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



Once again since we assume $r(x) < r(z)$ and the ranks are integers, we have $r(z) \ge r(x) + 1$. We always have $r'(z) \le r(z)$ and $r'(y) \le r(z)$ and $r'(x) = r(z)$. Since y started as an ancestor of x, we also have $r(y) \ge r(x)$ so we can write

$$
\begin{aligned}
\Delta\Phi &= r'(z) + r'(y) + r'(x) - r(z) - r(y) - r(x) \\
&\le r(z) + r(z) + r(z) - r(z) - r(y) - r(x) && (\text{use } r'(z) \le r(z),\ r'(y) \le r(z),\ r'(x) = r(z)) \\
&\le 3r(z) - (r(x) + 1) - r(x) - r(x) && (\text{use } r(z) \ge r(x) + 1\ \&\ r(y) \ge r(x)) \\
&= 3(r(z) - r(x)) - 1 && (\text{simplify})
\end{aligned}
$$

Adding the actual cost of 1, the amortized cost for this step is at most $3(r(z) - r(x))$.

So in every splay step, we have shown that the amortized cost is at most $3(r(z) - r(x))$ (plus the extra $+1$ needed for the zig case at the very end). Thus these amortized costs telescope giving an amortized number of splay steps of $3(r(t) - r(x)) + 1$. □

23.6 Balance Theorem

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

Theorem 23.1: Balance Theorem

A sequence of m splays in a tree of n nodes takes time

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

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

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

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

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

repeatedly using this inequality m times shows:

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

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

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

This proves the Balance Theorem. □

23.7 Improvement and Applications of the Access Lemma

Optional content — Not required knowledge for the exams

The access lemma can be improved by using a different potential function. Instead of $r(x) = \lfloor \log_2(s(x)) \rfloor$ we use the *real rank*, defined as $rr(x) = \log_2(s(x))$. The potential function in this case will be the sum of the real ranks of all the nodes.

Lemma: Strong Access Lemma

Take any tree T with root t, and any weights $w(\cdot)$ on the nodes. Suppose you splay node x. Then using the real rank based potential function we have:

$$\text{amortized number of ROTATIONS } \leq 3(rr(t) - rr(x)) + 1.$$

Note that this lemma bounds the number of rotations, not splay steps. And the bound is almost the same as the access lemma. So this is roughly a factor of two stronger than the access lemma. We will not prove this lemma here.

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

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

Theorem: Static Optimality Theorem

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

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

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

Theorem: Static Finger Theorem

Consider a sequence of m accesses in an n-node splay tree. Let $a[j]$ denote the jth element accessed. Let f be a specific element called the finger. Finally let $|e-f|$ denote the distance (in the in-order list of the tree) between elements e and f. Then the following is a bound on the cost of splaying the sequence:

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

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

Theorem: Working Set Theorem

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

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

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

Theorem: Sequential Access Theorem

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

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

Theorem: Dynamic Finger Theorem

Incorporation the notation from the Static Finger Theorem, the cost of an access sequence is bounded by

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

Lecture 24

Polynomials in Algorithm Design

In this lecture, we will see some of the power of polynomials in algorithm design. In particular, we'll see the fundamental beautiful ideas behind the error-correction used in QR codes (like this one):

Objectives of this lecture

In this lecture, we will

- review some properties of polynomials and the operations that can be performed on them
- see the *unique reconstruction theorem* and learn how to *interpolate* a set of points to obtain that unique polynomial
- see how polynomials can be used to implement *error-correcting codes*
- see some *algebraic algorithms* for matchings in graphs that utilize polynomials

24.1 Introduction

You've probably all seen polynomials before: e.g., $3x^2 - 5x + 17$, or $2x - 3$, or $-x^5 + 38x^3 - 1$, or x, or even a constant 3. These are all polynomials over a single variable (here, x). The degree of a polynomial is the highest exponent of x that appears: hence the degrees of the above polynomials are $2, 1, 5, 1, 0$ respectively.

In general, a polynomial over the variable x of degree at most d looks like:

$$P(x) = c_d x^d + c_{d-1} x^{d-1} + .. + c_1 x + c_0$$

Remark: The coefficients completely describe P

Note that the sequence of $d+1$ coefficients $\langle c_d, c_{d-1}, \ldots, c_0\rangle$ completely describes $P(x)$.

In general the coefficients (and variables) of a polynomial can be drawn from any field you choose. In this lecture we will be making frequent use of the finite fields, specifically the field of integers modulo a prime p, which is denoted $\mathbb{Z}_p$.

If the coefficients were all drawn from the set $\mathbb{Z}_p$, then we have exactly p^{d+1} possible different polynomials of degree at most d (note that it is *at most* d and not exactly d because c_d could be zero). This includes the zero polynomial $0 = 0x^d + 0x^{d-1} + .. + 0x + 0$.

In this lecture, we will use properties of polynomials to construct error correcting codes, and do other cool things with them.

24.2 Operations on Polynomials

Before we study properties of polynomials, recall the following simple operations on polynomials:

- **Addition:** Given two polynomials $P(x)$ and $Q(x)$, we can add them to get another polynomial $R(x) = P(x)+Q(x)$. Note that the degree of $R(x)$ is at most the maximum of the degrees of P and Q. (Q: Why is it not equal to the maximum?)

$$(x^2+2x-1)+(3x^3+7x) = 3x^3+x^2+9x-1$$

  The same holds for the difference of two polynomials $P(x)-Q(x)$, which is the same as $P(x)+(-Q(x))$.

- **Multiplication:** Given two polynomials $P(x)$ and $Q(x)$, we can multiply them to get another polynomial $S(x) = P(x)\times Q(x)$.

$$(x^2+2x-1)\times(3x^3+7x) = 3x^5+4x^3+6x^4+14x^2-7x$$

  The degree of $S(x)$ is equal to the sum of the degrees of P and Q.

- **Division:** Polynomials can be divided to yield an algebraic expression, but do note that the result $P(x)/Q(x)$ may not itself be a polynomial. In some situations it will be, and this can turn out to be useful (we'll see an example soon).

- **Evaluation:** We can also *evaluate* polynomials. Given a polynomial $P(x)$ and a value a, $P(a) := c_d \cdot a^d + c_{d-1}\cdot a^{d-1} + .. + c_1 \cdot a + c_0$. For example, if $P(x) = 3x^5+4x^3+6x^4+14x^2-7x$, then

$$P(2) = 3\cdot 2^5 + 4\cdot 2^3 + 6\cdot 2^4 + 14\cdot 2^2 - 7\cdot 2 = 266$$

  Naive evaluation of a polynomial would take $O(d^2)$ time, but this can be improved to $O(d)$:

Algorithm: Horner's Rule

Evaluate the following recurrence for p_0.

$$p_i = \begin{cases} c_d & \text{if } i = d, \\ p_{i+1} \cdot a + c_i & \text{otherwise} \end{cases}$$

The result is $p_0 = P(a)$. Each step is one addition and one multiplication, hence this takes $O(d)$ time.

- **Roots:** A *root* of a polynomial $P(x)$ is a value r such that $P(r) = 0$. For example, $P(x)$ above has three real roots $0, -1+\sqrt{2}, -1-\sqrt{2}$, and two complex roots.

Here, we were implictly working over $\mathbb{R}$, the field of real numbers.

24.3 How Many Roots?

Let's start with the following super-important theorem.

Theorem 24.1: Few-Roots Theorem

Any non-zero polynomial of degree at most d has at most d roots.

This holds true, regardless of what field we are working over. When we are working over the reals (i.e., the coeffcients are reals, and we are allowed to plug in arbitrary reals for x), this theorem is a corollary of the fundamental theorem of Algebra. But it holds even if we are working over some other field (say $\mathbb{Z}_p$ for prime p).

Let's relate this to what we know. Consider polynomials of degree 1, also known as linear polynomials. Say they have real coefficients, this gives a straight line when we plot it. Such a polynomial has at most one root: it crosses the x-axis at most once. And in fact, any degree-1 polynomial looks like $c_1 x + c_0$, and hence setting $x = -c_0/c_1$ gives us a root. So, in fact, a polynomial of degree exactly 1 has exactly one root.

What about degree 2, the quadratics? Things get a little more tricky now, as you probably remember from high school. E.g., the polynomial $x^2 + 1$ has no real roots, but it has two complex roots. However, you might remember that if it has one real root, then both roots are real. But anyways, a quadratic crosses the x-axis at most twice. At most two roots.

And in general, Theorem 24.1 says, any polynomial of degree at most d has at most d roots.

24.4 Another Representation for degree-d Polynomials

Let's prove a simple corollary of Theorem 24.1, which says that if we plot two polynomials of degree at most d, then they can intersect in at most d points—*unless they are the same polyno-*

mial (and hence intersect everywhere)! Remember, two distinct lines intersect at most once, two distinct quadratics intersect at most twice, etc. Same principle.

Corollary 24.1

Given $d+1$ pairs $(a_0, b_0), (a_1, b_1), \dots, (a_d, b_d)$, there is *at most* one polynomial $P(x)$ of degree at most d, such that $P(a_i) = b_i$ for all $i = 0, 1, \dots, d$.

Proof. For a contradiction, suppose there are two distinct polynomials $P(x)$ and $Q(x)$ of degree at most d such that for all i,

$$P(a_i) = Q(a_i) = b_i.$$

Then consider the polynomial $R(x) = P(x) - Q(x)$. It has degree at most d, since it is the difference of two polynomials of degree at most d. Moreover,

$$R(a_i) = P(a_i) - Q(a_i) = 0$$

for all the $d+1$ settings of $i = 0, 1, \dots, d$. Once again, R is a polynomial of degree at most d, with $d+1$ roots. By the contrapositive of Theorem 24.1, $R(x)$ must be the zero polynomial. And hence $P(x) = Q(x)$, which gives us the contradiction. □

To paraphrase the theorem differently, given two (i.e., $1+1$) points there is at most one linear (i.e., degree-1) polynomial that passes through them, given three (i.e., $2+1$) points there is at most one quadratic (i.e., degree-2) polynomial that passes through them, etc.

Can it be the case that for some $d+1$ pairs $(a_0, b_0), (a_1, b_1), \dots, (a_d, b_d)$, there is *no* polynomial of degree at most d that passes through them? Well, clearly if $a_i = a_j$ but $b_i \neq b_j$. But what if all the a_i's are distinct?

Theorem 24.2: Unique Reconstruction Theorem

Given $d+1$ pairs $(a_0, b_0), (a_1, b_1), \dots, (a_d, b_d)$ where the a_i's are distinct, there always exists a polynomial $P(x)$ of degree at most d, such that $P(a_i) = b_i$ for all $i = 0, 1, \dots, d$.

We will prove this theorem soon, but before that note some implications. Combining Corollary 24.1 with Theorem 24.2, we get that given $d+1$ pairs $(a_0, b_0), (a_1, b_1), \dots, (a_d, b_d)$ with distinct a's, this means there is a *unique* polynomial of degree at most d that passes through them. Exactly one.

In fact, given $d+1$ numbers $b_0, b_1, \dots, b_d$, there is a unique polynomial $P(x)$ of degree at most d such that $P(i) = b_i$. (We're just using the theorem with $a_i = i$.) Earlier we saw how to represent any polynomial of degree at most d by $d+1$ numbers, the coefficients. Now we are saying that we can represent the polynomial of degree at most d by a different sequence of $d+1$ numbers: its values at $0, 1, \dots d$.

Two different representations for the same thing, cool! Surely there must be a use for this new representation. We will give at least two uses for this, but first let's see the proof of Theorem 24.2.

(If you are impatient, you can skip over the proof, but do come back and read it—it is very elegant.)

Proof. OK, now the proof of Theorem 24.2. We are given $d+1$ pairs (a_i, b_i), and the a's are all distinct. The proof is by consruction – it will give an algorithm to find this polynomial $P(x)$ with degree at most d, and where $P(a_i) = b_i$.

Let's start easy: suppose all the $d+1$ values b_i's were zero. Then $P(x)$ has $d+1$ roots, and now Theorem 24.1 tells us that $P(x) = 0$, the zero polynomial!

OK, next step. Suppose $b_0 = 1$, but all the d other b_i's are zero. Do we know a degree-d polynomial which has roots at d places $a_1, a_2, \ldots, a_d$. Sure, we do—it is just

$$Q_0(x) = (x-a_1)(x-a_2)\cdots(x-a_d).$$

So are we done? Not necessarily: $Q_0(a_0)$ might not equal $b_0 = 1$. But that is easy to fix! Just scale the polynomial by $1/Q_0(a_0)$. I.e., what we wanted was

$$\begin{aligned} R_0(x) &= (x-a_1)(x-a_2)\cdots(x-a_d)\cdot\frac{1}{Q_0(a_0)} \\ &= \frac{(x-a_1)(x-a_2)\cdots(x-a_d)}{(a_0-a_1)(a_0-a_2)\cdots(a_0-a_d)} \ . \end{aligned}$$

Again, $R_0(x)$ has degree d by construction, and satisfies what we wanted! (We'll call $R_0(x)$ the 0th "switch" polynomial.)

Next, what if b_0 was not 1 but some other value. Easy again: just take $b_0 \times R_0(x)$. This has value $b_0 \times 1$ at a_0, and $b_0 \times 0 = 0$ at all other a_i's.

Similarly, one can define switch polynomials $R_i(x)$ of degree d that have $R_i(a_i) = 1$ and $R_i(a_j) = 0$ for all $i \neq j$. Indeed, this is

$$R_i(x) = \frac{(x-a_0)\cdots(x-a_{i-1})\cdot(x-a_{i+1})\cdots(x-a_d)}{(a_i-a_0)\cdots(a_i-a_{i-1})\cdot(a_i-a_{i+1})\cdots(a_i-a_d)} \ .$$

So the polynomial we wanted after all is just a linear combination of these switch polynomials:

$$P(x) = b_0R_0(x) + b_1R_1(x) + \ldots + b_dR_d(x)$$

Since it is a sum of degree-d polynomials, $P(x)$ has degree at most d. And what is $P(a_i)$? Since $R_j(a_i) = 0$ for all $j \neq i$, we get $P(a_i) = b_iR_i(a_i)$. Now $R_i(a_i) = 1$, so this is b_i. □

The polynomial that goes through a given set of points is called the *interpolating* polynomial for those points. The technique described in the proof is called *Lagrange interpolation.*

Example

Consider the tuples $(5,1),(6,2),(7,9)$: we want the unique degree-2 polynomial that passes through these points. So first we find $R_0(x)$, which evaluates to 1 at $x=5$, and has roots at 6 and 7. This is

$$R_0(x) = \frac{(x-6)(x-7)}{(5-6)(5-7)} = \frac{1}{2}(x-6)(x-7)$$

Similarly

$$R_1(x) = \frac{(x-5)(x-7)}{(6-5)(6-7)} = -(x-5)(x-7)$$

and

$$R_2(x) = \frac{(x-5)(x-6)}{(7-5)(7-6)} = \frac{1}{2}(x-5)(x-6)$$

Hence, the polynomial we want is

$$P(x) = 1\cdot R_0(x) + 2\cdot R_1(x) + 9\cdot R_2(x) = 3x^2 - 32x + 86$$

Let's check our answer:

$$P(5) = 1, P(6) = 2, P(7) = 9.$$

Running Time: Note that constructing the polynomial $P(x)$ takes $O(d^2)$ time. (Can you find the simplified version in this time as well?)

24.5 Application: Error Correcting Codes

Consider the situation: I want to send you a sequence of $d+1$ numbers $\langle c_d, c_{d-1}, \ldots, c_1, c_0\rangle$ over a noisy channel. I can't just send you these numbers in a message, because I know that whatever message I send you, the channel will corrupt up to k of the numbers in that message. For the current example, assume that the corruption is very simple: whenever a number is corrupted, it is replaced by a $\star$. Hence, if I send the sequence

$$\langle 5, 19, 2, 3, 2\rangle$$

and the channel decides to corrupt the third and fourth numbers, you would get

$$\langle 5, 19, \star, \star, 2\rangle.$$

On the other hand, if I decided to delete the fourth and fifth elements, you would get

$$\langle 5, 19, 2, \star, \star\rangle.$$

Since the channel is "erasing" some of the entries and replacing them with $\star$'s, the codes we will develop will be called *erasure* codes. The question then is: how can we send $d+1$ numbers so that the receiver can get back these $d+1$ numbers even if up to k numbers in the message are erased (replaced by $\star$s)? (Assume that both you and the receiver know d and k.)

A simple case: if $d = 0$, then one number is sent. Since the channel can erase k numbers, the best we can do is to repeat this single number $k+1$ times, and send these $k+1$ copies across. At least one of these copies will survive, and the receiver will know the number.

This suggests a strategy: no matter how many numbers you want to send, repeat each number $k+1$ times. So to send the message $\langle 5,19,2,3,2\rangle$ with $k=2$, you would send

$$\langle 5,5,5,19,19,19,2,2,2,3,3,3,2,2,2\rangle$$

This takes $(d+1)(k+1)$ numbers, approximately dk. Can we do better?

Indeed we can! We view our sequence $\langle c_d, c_{d-1},\ldots,c_1,c_0\rangle$ as the $d+1$ coefficients of a polynomial of degree at most d, namely $P(x) = c_d x^d + c_{d-1}x^{d-1} + .. + c_1 x + c_0$. Now we evaluate P at some $d+k+1$ points, say $0,1,2,\ldots,d+k$, and send these $d+k+1$ numbers

$$P(0), P(1), \ldots, P(d+k)$$

across. The receiver will get back at least $d+1$ of these numbers, which by Theorem 24.2 uniquely specifies $P(x)$. Moreover, the receiver can also reconstruct $P(x)$ using Langrange interpolation.

Theorem: Erasure codes using polynomials

We can send a sequence of $d+1$ numbers across an erasure channel that erases up to k of them by sending $d+k+1$ points of the polynomial P encoded by the sequence.

Example

Here is an example: Suppose we want to send $\langle 5,19,2,3,2\rangle$ with $k=2$. Hence $P(x) = 5x^4+19x^3+2x^2+3x+2$. Now we'll evaluate $P(x)$ at $0,1,2,\ldots d+k=6$. This gives

$$P(0)=2, P(1)=31, P(2)=248, P(3)=947, P(4)=2542, P(5)=5567, P(6)=10676$$

So we send across the "encoded message":

$$\langle 2,31,248,947,2542,5567,10676\rangle$$

Now suppose the third and fifth entries get erased. the receiver gets:

$$\langle 2,31,\star,947,\star,5567,10676\rangle$$

So she wants to reconstruct a polynomial $R(x)$ of degree at most 4 such that $R(0) = 2, R(1) = 31, R(3) = 947, R(5) = 5567, R(6) = 10676$. (That is, she wants to "decode" the message.) By Langrange interpolation, we get that

$$R(x) = \frac{1}{45}(x-1)(x-3)(x-5)(x-6) - \frac{31}{40}x(x-3)(x-5)(x-6) + \frac{947}{36}x(x-1)(x-5)(x-6) - \frac{5567}{40}x(x-1)(x-3)(x-6) + \frac{5338}{45}x(x-1)(x-3)(x-5)$$

which simplifies to $P(x) = 5x^4+19x^3+2x^2+3x+2$!

Note on Efficiency The numbers can get large, so you may want work in the field $\mathbb{Z}_p$, as long as the size of the field is large enough to encode the numbers you want to send across. (Of course, if you are working modulo a prime p, both the sender and the receiver must know p.) This will save both time and reduce the number of bits in the encoded sequences.

Example

Let's do the previous example using a finite field. Since we want to send numbers as large as 19, let's work in $\mathbb{Z}_{23}$. Then you'd send the numbers modulo 23, which would be

$$\langle 2, 8, 18, 4, 12, 1, 4\rangle$$

Now suppose you get

$$\langle 2, 8, \star, 4, \star, 1, 4\rangle$$

Interpolate to get

$$R(x) = 45^{-1}(x-1)(x-3)(x-5)(x-6) - 5^{-1}x(x-3)(x-5)(x-6) + 9^{-1}x(x-1)(x-5)(x-6) \\ -40^{-1}x(x-1)(x-3)(x-6) + 2\cdot 45^{-1}x(x-1)(x-3)(x-5)$$

where the multiplicative inverses are modulo 23, of course. Simplifying, we get $P(x) = 5x^4 + 19x^3 + 2x^2 + 3x + 2$ again.

One final note. A beautiful application of these kinds of erasure codes is in the design of RAID systems. RAID stands for Redundant Array of Independent Disks. Suppose you wanted to design a storage system comprised of five identical 1TB hard drives. And you wanted it such that if any two of the drives died, all the data would still exist on the other three drives. And you wanted to be able to use these five drives to store 3TB of such space. Then what you can do is use the codes described here, where three words of data are encoded as five words, in such a way that any subset of three of them can be used to reconstruct the original three words. This is achieved by an erasure code.

24.5.1 Error Correction

One can imagine that the channel is more malicious: it decides to *replace* some k of the numbers not by stars but by other numbers, so the same encoding/decoding strategy cannot be used! Indeed, the receiver now has no clue which numbers were altered, and which ones were part of the original message! In fact, even for the $d = 0$ case of a single number, we need to send $2k+1$ numbers across, so that the receiver knows that the majority number must be the correct one. And indeed, if you evaluate $P(x)$ at $n = d+2k+1$ locations and send those values across, even if the channel alters k of those numbers, there is a unique degree-d polynomial that agrees with $d+k+1$ of these numbers (and this must be $P(x)$)

Theorem: Error correction code using polynomials

If we want to send $d+1$ numbers across a channel in which k of them could be replaced, we send $d+2k+1$ points of the polynomial P encoded by the sequence, then for any

subset of $d+k+1$ points on the receiving end (which may contain corruptions), if there exists a polynomial that interpolates these points, it must be P.

Proof. First, there exists some subset of points that interpolates to P, the uncorrupted $d+k+1$ of them. Now consider some other subset of points and suppose there exists a degree-d polynomial Q that interpolates them. Since there are k corrupted points, P and Q must agree on at least $d+1$ of the points and hence by the unique reconstruction theorem, P and Q are the same polynomial since there is a unique degree-d polynomial that interpolates $d+1$ points. □

So we can show that $d+2k+1$ points is good enough that it uniquely determines P even in the presence of adversarial replacements, but its not clear how to actually reconstruct P. The above theorem only tells us that it is possible, but by brute force we would have to try exponentially many subsets to find one that works. What is amazing is that there is an efficient algorithm that the receiver can use to reconstruct $P(x)$: this is known as the *Berlekamp-Welch* algorithm.

The Berlekamp-Welch Error-Correction Algorithm

Let $[n] := \{0, 1, \ldots, n-1\}$, where $n = d+2k+1$. Suppose we send over the n numbers

$$s_0, s_1, \ldots, s_{n-1},$$

where $s_i = P(i)$. We receive numbers

$$r_0, r_1, \ldots, r_{n-1},$$

where at most k of these r_is are not the same as the s_is. Define a set Z of size at most k such that $\{i \mid s_i \neq r_i\} \subseteq Z$: i.e., Z contains all the error locations.

Now define a degree-j "error" polynomial $E(x)$ such that

$$E(x) = \prod_{a \in Z} (x-a).$$

Observe that

$$P(x) \cdot E(x) = r_x \cdot E(x) \qquad \forall x \in [n]. \tag{24.1}$$

Indeed, $E(x) = 0$ for all $x \in Z$ (by construction of $E()$) and $P(x) = r_x$ for all $x \in [n] \setminus Z$ (by the definition of Z). Of course, we just received the r_is, so we don't know $P(x)$. Nor do we know $E(x)$, since we don't know Z. But we know that $E(x)$ looks like:

$$E(x) = x^k + e_{k-1} x^{k-1} + \ldots + e_1 x + e_0.$$

for some values $e_{k-1}, e_{k-2}, \ldots, e_0$. (So there are k unknown coefficients, since the coefficient of x^k in $E(x)$ is 1.) Moreover, we know that $P(x) \cdot E(x)$ has degree $d+k$, so looks like

$$P(x) \cdot E(x) = f_{d+k} x^{d+k} + f_{d+k-1} x^{d+k-1} + \ldots + f_1 x + f_0.$$

So the $n = d + 2k + 1$ equalities from (24.5.1) look like

$$f_{d+k}x^{d+k} + f_{d+k-1}x^{d+k-1} + \ldots + f_1 x + f_0 = r_x (x^k + e_{k-1}x^{k-1} + \ldots + e_1 x + e_0),$$

one for each $x \in [n]$. The unknown are e_i and f_i values—there are $k + (d + k + 1) = d + 2k + 1$ unknowns. So we can solve for these unknowns (say using Gaussian elimination), and get $E(x)$ and $P(x) \cdot E(x)$. Dividing the latter by the former gives back $P(x)$. It's like magic.

24.6 Multivariate Polynomials and Matchings

Optional content — Not required knowledge for the exams

Here's a very different application of polynomials in algorithm design. Now we'll consider multivariate polynomials, and use the fact that they also have "few" roots to get an unusual algorithm for finding matchings in graphs. We need to think carefully about what we mean by "few", since for multivariate polynomials, even linear polynomials can have many roots, more than any function of the degree. For example $P(x, y) = x - y$ has infinitely many roots, one for every point (x, y) such that $x = y$. This is unlike single variable polynomials where we could just bound the number of roots by d. So instead of trying to bound the number of roots, we'll instead try to show that the fraction of points that can be roots is small.

Definition: Multivariate polynomial

A multivariate polynomial is a sum of monomials, where a monomial is a product of powers of the variables (and possibly a constant), e.g.,

$$P(x_1, x_2, x_3, x_4) = x_1 x_2^2 x_4 + x_3 x_4^2 + x_1 x_2^2 x_3^2 x_4$$

The degree of the monomial $x_1^{i_1} x_2^{i_2} x_3^{i_3} x_4^{i_4}$ is $i_1 + i_2 + i_3 + i_4$. The degree of P is the maximum degree of any of its monomials.

Here's an alternate view of Theorem 24.1 that is possible to generalize to multivariate polynomials: suppose we fix a set S of values in the field we are working over (e.g., $\mathbb{R}$, or $\mathbb{F}_p$), and pick a random $x \in S$. Given a degree-d polynomial, what is the probability that we picked a root? There are at most d distinct root, so the probability that $P(x) = 0$ is at most $d/|S|$. One can extend this to the following theorem for multivariate polynomials $P(\mathbf{x}) = P(x_1, x_2, \ldots, x_m)$.

Theorem 24.3: Schwartz (1980), Zippel (1979)

For any non-zero degree-d polynomial $P(\mathbf{x})$ and any subset S of values from the underlying field, if each X_i is chosen independently and uniformly at random from S, then

$$\Pr[P(X_1,\ldots,X_m)=0] \leq \frac{d}{|S|}.$$

This theorem is useful in many contexts. E.g., we get an algorithm for perfect matchings.

24.6.1 Application: Perfect matchings

Definition: The Tutte matrix

For any graph $G=(V,E)$ with vertices $v_1, v_2, \ldots, v_n$, the *Tutte matrix*[a] is a $|V|\times|V|$ matrix $M(G)$:

$$M(G)_{i,j} = \begin{cases} x_{i,j} & \text{if } \{v_i, v_j\} \in E \text{ and } i<j \\ -x_{j,i} & \text{if } \{v_i, v_j\} \in E \text{ and } i>j \\ 0 & \text{if } (v_i, v_j) \notin E \end{cases}$$

[a] Named after William T. (Bill) Tutte, pioneering graph theorist and algorithm designer. Recently it was discovered that he was one of the influential code-breakers in WWII, making crucial insights in breaking the Lorenz cipher.

This is a square matrix of variables $x_{i,j}$. And like any matrix, we can take its determinant, which is a (multivariate) polynomial $P_G(\mathbf{x})$ in the variables $\{x_{i,j}\}_{\{i,j\}\in E}$. The degree of this polynomial is at most $n=|V|$, the dimension of the matrix. Here is a surprising and super useful fact:

Theorem 24.4: Tutte (1947)

A graph G has a perfect matching if and only if $P_G(\mathbf{x})$, the determinant of the Tutte matrix, is not the zero polynomial.

How do we check if $P_G(\mathbf{x})$ is zero or not? That's the problem: since we're taking a determinant of a matrix of variables, the usual way of computing determinants may lead to $n!$ terms, which eventually may all cancel out!

However, we can combine Theorems 24.3 and 24.4 together: take G, construct $M(\mathbf{x})$, and replace each variable by an independently uniform random value in some set S, and then compute the determinant of the resulting matrix of random numbers. This is exactly like plugging in the random numbers into $P_G(\mathbf{x})$. So if $P_G(\mathbf{x})$ was zero, the answer is zero for sure. And else, the answer is zero with probability at most $n/|S|$, which we can make as small as we want by choosing S large enough, or by repeating the process sufficiently many times.

Exercises: Polynomials in algorithm design

Problem 53. Take your favorite degree-3 polynomial $P(x)$ over the reals, and evaluate it at any 4 points. Use Lagrange interpolation to fit a degree-3 polynomial through those points, and verify that it equals $P(x)$.

Problem 54. Think about how you would use the algorithm from Section 24.6.1 (which tests for the existence of a perfect matching), to actually find a perfect matching (with high probability) in a graph, if one exists. Your PM-finder should perform at most $O(m)$ calls to the PM-existence-checker.

Problem 55. Given an algorithm to find perfect matchings in a graph (if one exists), use it to find maximum cardinality matchings in graphs.

Lecture 25
The Fast Fourier Transform

In this final lecture, we will see an algorithm that can multiply two polynomials of degree d in $O(d \log d)$ time. It is based on the famous *Fast Fourier Transform* algorithm, and combines classic ideas from algorithm design (e.g., divide-and-conquer) with algebraic techniques (polynomials) and math (complex numbers) in an extraordinarily cool way.

Objectives of this lecture

In this lecture, we will

- review some math that we will need, including polynomials and complex numbers
- derive the *Fast Fourier Transform* algorithm and use it for multiplying polynomials

25.1 Preliminaries

25.1.1 Polynomials

Recall that a polynomial of degree d is a function p that looks like

$$p(x) := \sum_{i=0}^{d} c_i x^i = c_d x^d + c_{d-1} x^{d-1} + \ldots + c_1 x + c_0$$

A polynomial of degree d can be described by a vector of its coefficients $\langle c_d, c_{d-1}, \ldots, c_1, c_0 \rangle$. According the *unique reconstruction theorem*, a polynomial can also be uniquely described by its values at $d+1$ distinct points $x_0, \ldots, x_d$.

Polynomials can be multiplied to yield another polynomial. The product of a degree n and degree m polynomial is a polynomial of degree $n+m$. Let $A(x)$ and $B(x)$ be two polynomials of degree d

$$\begin{aligned} A(x) &= a_0 + a_1 x + a_2 x^2 + \cdots + a_d x^d, \\ B(x) &= b_0 + b_1 x + b_2 x^2 + \cdots + b_d x^d, \end{aligned}$$

then their product is the polynomial $C(x)$:

$$C(x) = c_0 + c_1 x + c_2 x^2 + \cdots + c_{2d} x^{2d},$$

where

$$c_k = \sum_{\substack{0 \le i,j \le k \\ i+j=k}} a_i \cdot b_j = \sum_{0 \le i \le k} a_i \cdot b_{k-i}.$$

Computing the product C directly via the definition would take $O(d^2)$ time assuming we can perform all of the necessary arithmetic operations on the coefficients in constant time. Karatsuba's algorithm can improve this to $O(d^{1.58})$. Today, we will see how to do it even faster.

25.1.2 Complex numbers and roots of unity

The field of *complex numbers* consists of numbers of the form

$$a + bi$$

where $a, b \in \mathbb{R}$ and i is the special *imaginary unit*, which is defined to be the solution to the equation $i^2 = -1$. Complex numbers are very useful for analyzing the solutions to polynomials, since every polynomial equation has a solution over the complex numbers, even though it might not have any real-valued solution.

Roots of unity A *root of unity* is a fancy way of saying an nth root of 1 for some value of n. that is, a complex number ω is an nth root of unity if it satisfies

$$\omega^n = 1.$$

There are exactly n complex n^{th} roots of unity, which can be written as

$$e^{\frac{2\pi i k}{n}}, \qquad k = 0, 1, \ldots, n-1.$$

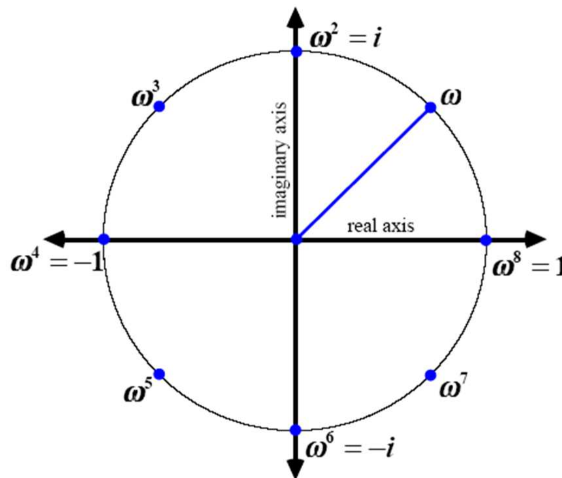
Observe the useful fact that

$$e^{\frac{2\pi i k}{n}} = \left(e^{\frac{2\pi i}{n}}\right)^k,$$

i.e., the roots of unity can all be defined as powers of the $k = 1^{\text{st}}$ one. We call this one a *primitive n^{th} root of unity*. More specifically, ω is a primitive n^{th} root of unity if

$$\begin{aligned} \omega^n &= 1 \\ \omega^j &\neq 1 \quad \text{for } 0 < j < n \end{aligned}$$

The following figure shows the eighth roots of unity. As the figure suggests, in general, the nth roots of unity are always equally spaced around the unit circle.



This graphical depiction also shows us a very useful property of the roots of unity.

Lemma: Halving lemma

For any even $n \geq 0$, the squares of the complex n^{th} roots of unit are precisely the $(n/2)^{\text{th}}$ roots of unity.

25.2 Polynomial Multiplication: The High Level Idea

By default, we usually represent polynomials in the coefficient representation, but we recall that if we know the value of a degree d polynomial at $d+1$ distinct points, that uniquely determines the polynomial. Not only that, but if we had A and B in this point-value representation, we could, in $O(d)$ time compute the polynomial C in that same representation: simply multiply the values of A and B at the specified points together. This is much faster than multiplying polynomials directly via the coefficient representation which takes $O(d^2)$ time.

This leads to the following outline of an algorithm for this problem:

> Let $N = 2d+1$ so the degree of C is less than N.
> (1) Pick N points $x_0, \ldots, x_{N-1}$ according to a secret formula.
> (2) Evaluate $A(x_0), \ldots, A(x_{N-1})$ and $B(x_0), \ldots, B(x_{N-1})$.
> (3) Now compute $C(x_0), \ldots, C(x_{N-1})$ where $C(x) = A(x)B(x)$.
> (4) Interpolate to get the coefficients of C.

The reason we like this is that multiplying is easy in the "value on N points" representation. So step 3 is only $O(N)$ time.

If we ignore steps (2) and (4), this algorithm just takes $O(N) = O(d)$ time. However, the best algorithm that we know so far to perform those steps each take $O(d^2)$ time: To compute the point representation, we could use Horner's rule $2d+1$ times, and to interpolate, we can use Lagrangian interpolation in $O(d^2)$ time.

We therefore shift the goalposts to finding a fast algorithm for performing steps (2) (4). Note that we have some flexibility in this framework. In order to compute the product, it is sufficient to evaluate the polynomial at *any* N points $x_0, \ldots x_{N-1}$, so perhaps there is some clever choice of points that will make the algorithm faster than just using any arbitrary set of points (indeed there is and this is the magic of the algorithm!)

Let's focus on forward direction first. In that case, we've reduced our problem to the following:

> **GOAL:** Given a polynomial A of degree $< N$, evaluate A at N points of our choosing in total time $O(N \log N)$. Assume N is a power of 2.

25.3 To Point-Value Form (The Fast Fourier Transform)

First here's a little intuition for how this is going to work. Consider the case where the degree of A is 7, and we want to evaluate it at two points: 1 and -1.

$$A(x) = a_0 + a_1 x + a_2 x^2 + a_3 x^3 + a_4 x^4 + a_5 x^5 + a_6 x^6 + a_7 x^7.$$

So:

$$\begin{aligned} A(1) &= a_0 + a_1 + a_2 + a_3 + a_4 + a_5 + a_6 + a_7 \\ A(-1) &= a_0 - a_1 + a_2 - a_3 + a_4 - a_5 + a_6 - a_7 \end{aligned}$$

These two computations take a total of 14 additions.

Suppose instead we were to compute $Z = a_0 + a_2 + a_4 + a_6$ and $W = a_1 + a_3 + a_5 + a_7$. Now $A(1) = Z + W$ and $A(-1) = Z - W$. This can be done in a total of only 8 additions. This optimization may not seem like much, but if we can figure out how to apply it recursively, it solves the problem – it gets us from $O(N^2)$ down to $O(N \log N)$.

Making it recursive To make the above idea recursive, the first step is to keep everything in terms of polynomials rather than evaluate immediately and just end up with numbers. What's the polynomial equivalent of our even and odd expressions (Z and W)? Lets say we just take those even and odd coefficients and use them to write smaller (half as big) polynomials!

$$\begin{aligned} A_{\text{even}}(x) &= a_0 + a_2 x + a_4 x^2 + a^6 x^3, \\ A_{\text{odd}}(x) &= a_1 + a_3 x + a_5 x^2 + a_7 x^3 \end{aligned}$$

Now the important question is how to recombine the smaller polynomials to get the larger one? Note that we basically halved all of the powers when we took the smaller polynomial, so to get back the original powers, we need to *square* them. The odd powers are then one higher, so we can multiply by x to recover those. This gives us the following important formula:

$$A(x) = A_{\text{even}}(x^2) + x A_{\text{odd}}(x^2).$$

So, to make our algorithm recursive, we want to split the polynomial into even and odd polynomials of half as many terms, divide-and-conquer on those and then combine the two halves back together using the above formula! The major question that remains is how do we choose the x values to use? 1 and -1 seemed like good choices above because they work when the polynomial is cut into odd and even, but its not obvious how to generalize that to N points.

25.3.1 Selecting the best points

The key insight into selecting the right points is to to keep the x^2 part of the formula in mind. When we are recombining the subproblems, the points are no longer the same set of points (but rather the squares of the previous points), so that presents a challenge.

In general, if we start with a set of N points, then take all of their squares, we get back a new set of N points. This is not very helpful since our divide-and-conquer will perform $O(N)$ work at every subproblem and therefore still take $O(N^2)$ time. Somehow, we need a magic set of points such that if we start with N of them, then square them all, we get $N/2$ points, since then the divide-and-conquer will correctly reduce the problem size at each level.

Do we know any special set of numbers that have this property??? Yes, we talked about them in the beginning of the lecture! We can use ***roots of unity*** as our set of points. When we take the square of the N N^{th} roots of unity, we get the $N/2$ $(N/2)^{\text{th}}$ roots of unity, which is exactly what we need to make our algorithm efficient!

These are the special magic numbers at which we will evaluate our polynomial. We will write A as a polynomial of degree $N-1$:

$$A(x) = a_0 + a_1 x + a_2 x_2 + \cdots + a_{N-1} x^{N-1},$$

and then define the DFT (Discrete Fourier Transform) of the coefficient vector $(a_0, \ldots, a_{N-1})$ to be another vector of N numbers as follows:

$$F_N(a_0, a_1, \ldots, a_{N-1}) = (A(\omega^0), A(\omega^1), \ldots, A(\omega^{N-1}))$$

That is, we are just evaluating A at the Nth roots of unity.

25.3.2 The Fast Fourier Transform algorithm

Now we can derive the fast algorithm for computing the DFT. (This is called the FFT algorithm.) Let A be the vector $(a_0, a_1, \ldots, a_{N-1})$. Let $F_N(A)_j$ denote the jth component of the DFT of the vector A.

$$\begin{aligned}
F_N(A)_j &= A(\omega^j) \\
&= \sum_{i=0}^{N-1} a_i \omega^{ij} \\
&= \sum_{\substack{i\,=\,0 \\ i \text{ even}}}^{N-1} a_i \omega^{ij} + \sum_{\substack{i\,=\,1 \\ i \text{ odd}}}^{N-1} a_i \omega^{ij} \\
&= \sum_{i=0}^{\frac{N}{2}-1} a_{2i} \omega^{2ij} + \sum_{i=0}^{\frac{N}{2}-1} a_{2i+1} \omega^{(2i+1)j} \\
&= \sum_{i=0}^{\frac{N}{2}-1} a_{2i} (\omega^2)^{ij} + \omega^j \sum_{i=0}^{\frac{N}{2}-1} a_{2i+1} (\omega^2)^{ij} \\
&= \sum_{i=0}^{\frac{N}{2}-1} a_{2i} (\omega_{N/2})^{i(j \bmod N/2)} + \omega_N^j \sum_{i=0}^{\frac{N}{2}-1} a_{2i+1} (\omega_{N/2})^{i(j \bmod N/2)}
\end{aligned}$$

In the last step we've simply used the fact that $\omega_N^2 = \omega_{N/2}$, and the observation that $\omega_{N/2}^{ij} = (\omega_{N/2}^j)^i = (\omega_{N/2}^{j \bmod N/2})^i$ because $\omega_{N/2}^j$ is a periodic funciton of j with period $N/2$.

Now the key point is that these two summations are in fact just DFTs half the size. Let's let A_{even} denote the vector of as with even subsubscripts (of length $N/2$) and A_{odd} be the odd ones. We

can write the final equation above using these vectors as follows:

$$F_N(A)_j = F_{N/2}(A_{\text{even}})_{j \bmod N/2} + \omega_N^j F_{N/2}(A_{\text{odd}})_{j \bmod N/2}$$

Notice that in this derivation we did use the fact that ω is a root of unity, but we *never used* the fact that it is a primitive root of unity. That will be needed when we compute the inverse.

So we can rewrite the above recurrence as pseudocode as follows:

Algorithm: Fast Fourier Transform

FFT($[a_0, \ldots, a_{N-1}], \omega, N$)
  if $N = 1$ **then return** $[a_0]$
  $F_{\text{even}} \leftarrow$ **FFT**($[a_0, a_2, \ldots, a_{N-2}], \omega^2, N/2$)
  $F_{\text{odd}} \leftarrow$ **FFT**($[a_1, a_3, \ldots, a_{N-1}], \omega^2, N/2$)
  $F \leftarrow$ a new vector of length N
  $x \leftarrow 1$
  for $j = 0$ to $N - 1$ **do**
    $F[j] \leftarrow F_{\text{even}}[j \bmod (N/2)] + x * F_{\text{odd}}[j \bmod (N/2)]$
    $x \leftarrow x * w$
  return F

Since at each recursive call we have a polynomial of half the size and evaluate on half of many points, the runtime of this algorithm follows the recurrence:

$$T(N) = 2T(N/2) + O(N),$$

which solves to $O(N \log N)$.

25.4 The Inverse of the DFT

Remember, we started all this by saying that we were going to multiply two polynomials A and B by evaluating each at a special set of N points (which we can now do in time $O(N \log N)$), then multiply the values point-wise to get C evaluated at all these points (in $O(N)$ time) but then we need to interpolate back to get the coefficients. In other words, we're doing $F_N^{-1}(F_N(A) \cdot F_N(B))$.

So, we need to compute F_N^{-1}. We'll develop this now.

First, we can view the forward computation of the FFT (evaluating A at $1, \omega, \omega^2, \ldots, \omega^{N-1}$) as a matrix-vector product:

$$\begin{pmatrix} \omega^{0\cdot 0} & \omega^{0\cdot 1} & \cdots & \omega^{0\cdot(N-1)} \\ \omega^{1\cdot 0} & \omega^{1\cdot 1} & \cdots & \omega^{1\cdot(N-1)} \\ \vdots & \vdots & \ddots & \vdots \\ \omega^{(N-1)\cdot 0} & \omega^{(N-1)\cdot 1} & \cdots & \omega^{(N-1)\cdot(N-1)} \end{pmatrix} \begin{pmatrix} a_0 \\ a_1 \\ \vdots \\ a_{N-1} \end{pmatrix} = \begin{pmatrix} A(\omega^0) \\ A(\omega^1) \\ \vdots \\ A(\omega^{N-1}) \end{pmatrix}$$

Note that the matrix on the left is the one where the contents of the kth row and the jth column is ω^{kj}. Let's call this matrix $\mathbf{DFT}(\omega, N)$. What we need is $\mathbf{DFT}(\omega, N)^{-1}$.

Let's see what happens if we multiply $\mathbf{DFT}(\omega^{-1}, N)$ by $\mathbf{DFT}(\omega, N)$.

$$\text{The } (k, j) \text{ entry of } \mathbf{DFT}(\omega^{-1}, N)\, \mathbf{DFT}(\omega, N) = \sum_{s=0}^{N-1} \omega^{-ks} \omega^{sj}$$

So let's try to evaluate the summation on the right in the two cases of $k = j$ and $k \neq j$.

If $k = j$ then we get:

$$\sum_{s=0}^{N-1} \omega^{-js} \omega^{sj} \;=\; \sum_{s=0}^{N-1} \omega^{0} \;=\; N$$

If $k \neq j$ then we get:

$$\sum_{s=0}^{N-1} \omega^{-ks} \omega^{sj} \;=\; \sum_{s=0}^{N-1} \omega^{(j-k)\cdot s} \;=\; \sum_{s=0}^{N-1} (\omega^{j-k})^{s}$$

Now let's make use of the fact that ω is a primitive root of unity. This means that $\omega^{j-k} \neq 1$. The sum is now a geometric series. So we can use the standard formula for summing a geometric series:

$$1 + r + r^2 + \cdots + r^{N-1} = \frac{1 - r^N}{1 - r}$$

So

$$\sum_{s=0}^{N-1} (\omega^{j-k})^{s} = \frac{1 - (\omega^{j-k})^N}{1 - \omega^{j-k}} = \frac{1 - 1}{1 - \omega^{j-k}} = 0$$

So, summarizing what we just learned:

$$\text{The } (k, j) \text{ entry of } \mathbf{DFT}(\omega^{-1}, N)\, \mathbf{DFT}(\omega, N) = \begin{cases} N & \text{if } k = j \\ 0 & \text{if } k \neq j \end{cases}$$

This is just the identity matrix times N. So what we've just proven is that:

$$\mathbf{DFT}(\omega, N)^{-1} = \frac{1}{N} \mathbf{DFT}(\omega^{-1}, N)$$

This means that we can compute the inverse of the DFT by using the FFT algorithm described above except running it with ω^{-1} instead of ω, and then multiplying the result by $1/N$. It still runs in $O(N \log N)$ time. Putting this all together we have an algorithm to multiply polynomials (or equivalently, compute a convolution) in $O(N \log N)$ time.

Exercises: Fast Fourier Transform

Problem 56. Prove the halving lemma algebraically using the definition of the nth roots of unity.