# CS 161 NOTES

MOOR XU
NOTES FROM A COURSE BY TIM ROUGHGARDEN

ABSTRACT. These notes were taken during CS161 (Design and Analysis of Algorithms) taught by Tim Roughgarden in Winter 2011 at Stanford University. They were live-TEX-ed during lectures in `vim` and compiled using `latexmk`. Each lecture gets its own section. The notes are not edited afterward, so there may be typos; please email corrections to `moorxu@stanford.edu`.

## 1. 1/4: INTRODUCTION

We're going to ask and answer lots of questions. Why are you here? For most people, because it is required. How come? It is fundamental, it is useful, and it is fun.

### 1.1. **Two example problems.**

**Example 1.1.** Internet routing.

**Remark.** We can think of the internet as a graph, with vertices and with edges. The vertices are end hosts and routers, while the directed edges are physical or wireless connections.

There are other internet related graphs as well, such as social networks, Web, etc.

Which Stanford to Cornell path to use? We prefer to use the shortest path. Let's assume that the fewest number of hops would satisfy this shortest condition. Then evidently we need an algorithm to find the shortest path.

The first such algorithm is Dijkstra, but running this requires knowing what all the nodes are, which requires knowing the whole graph; this is infeasible for the internet. We need an alternative shortest path algorithm that needs only local computation. It's not obvious that such an algorithm should exist, but it does exist. It is the Bellman-Ford algorithm, and it is an application of dynamical programming.

**Example 1.2.** Sequence alignment. This is a fundamental problem in computational genomics. The input is two strings over the alphabet {A, C, G, T}. For example, the input to the algorithm might be AGGGCT and AGGCA, and these would be portions of genomes.

**Problem 1.3.** Figure out how "similar" two strings are.

Think about how many steps it takes to go from one string to the other. This is one way to defines similarity. Here's another: Define similarity as the quality of the "best" alignment. Consider some intuition: AGGGCT and AGGCA can be "nicely" aligned; write them as AGGGCT and AGG_CA. We had to insert a gap, and there was one mismatch.

Assume that we have experimentally determined penalties for gaps and mismatches. So one gap would produce some penalty, while mismatching A and T would produce some other penalty. Penalties are additive. In our example, the total penalty would be for one gap and one A/T mismatch.

The best match is therefore defined as the alignment with smallest total penalty. This is a famous concept in computational genomics, called the Needleman-Wunsch score. They were biologists in the early 70's. A small penalty score means that the sequences are similar.

Why would people care about doing this?

- Extrapolation of genome substrings
- Similarity of genomes can reflect proximity in evolutionary tree.

**Note.** The definition of NW score is inherently algorithmic. We need a practical algorithm to find the best alignment.

That algorithm is called brute-force search: try all possible alignments, pick the best one. But there are a massive number of alignments. The number of alignments exceeds the number of atoms in the known universe even for string lengths in the hundreds. This is bad.

We instead need a faster nontrivial algorithm. Use dynamical programming.

1.2. **Simple problem.** Consider the multiplication of two $n$-digit numbers $x$ and $y$. For example, $x = 1234$ and $y = 5678$. There exists the grade school algorithm for doing this. This is $O(n^2)$. Can we do better?

Let's think about recursive algorithms for doing this.

**Algorithm 1.4.** Write $x = 10^{n/2}a + b$ and $y = 10^{n/2}c + d$. The point is that each of $a, b, c, d$ are $\frac{n}{2}$-digit numbers. We are being asked to multiply

$$xy = 10^n ac + 10^{n/2}(ad + bc) + bd. \quad (1)$$

Now, all of the multiplications that remain involve numbers with fewer digits. This is a recursive problem. We now recursively compute $ac$, $ad$, $bc$, $bd$, and then finish evaluating the obvious way. Clearly, this algorithm is correct and terminates.

**Algorithm 1.5** (Gauss)**.** Recursively compute $ac$, $bd$, and $(a+b)(c+d) = ac+bd+ad+bc$. The point is that by subtraction we can compute $ad + bc$, and this only required three multiplications. This should be better than the previously algorithm, but it's unclear how to compare it to the grade school algorithm.

## 2. 1/6: Divide and Conquer

2.1. **Merge sort.** Today we begin with divide and conquer algorithms. The canonical example is mergesort. It remains the pedagogically cleanest example.

Consider given $n$ numbers that are unsorted, we want to get an array of $n$ numbers sorted in increasing order.

**Algorithm 2.1** (Mergesort)**.**

(1) Recursively sort 1st half
(2) Recursively sort 2nd half
(3) Merge two subsets into one.

Pseudocode for merge step:

```
c = output array, length > n
A = first sorted array, length > n/2
B = second sorted array, length > n/2
i = 1; j = 1;
```

```
for k from i to n
   if A(i) < B(j)
      c(k) = A(i); i++
   else c(k) = B(j); j++
```

We are now interested in the running time of merge sort. This means the number of operations executed.

The initialization step is 2 operations. Inside the for loop, we need three operations, and we need to increment $k$, so each loop iteration requires 4 operations. The loop runs $n$ times. So the running time of merge on an array of $m$ numbers is $\leq 4m + 2 \leq 6m$.

**Claim.** Merge sort requires $\leq 6n \log_2 n + 6n$ operations to sort $n \geq 1$ numbers.

*Proof.* Assume that $n$ is a power of 2 for simplicity. We will use a "recursion tree." The point is to write down all of the recursive calls into a tree structure. There are $\log_2 n$ levels to the tree. The key idea is to anaylze the work done by merge sort, level by level.

Consider a level $j = 0, 1, \ldots, \log_2 n$. There are $2^j$ subproblems, each of size $n/2^j$. So the total number of operations that we need at a level $j$ is

$$\leq 2^j \cdot 6 \left(\frac{n}{2^j}\right) = 6n.$$

We add this up over all of the levels to get that the total is $\leq 6n(\log_2 n + 1)$. $\square$

**Remark.** We need to make some assumptions.

(1) We used "worst case analysis": our bound applies to every input of length $n$. It is difficult to define "practical inputs". Also, this is easier to analyze.
(2) We will ignore constants and lower order terms. This makes life much easier and does not lose predictive power. Constants depend on domain-dependent factors anyway.
(3) We will bound running time for large inputs of size $n$. For example, we will claim that $6n \log n$ is "better than" $\frac{1}{2}n^2$. By Moore's Law, the only interesting problems are big problems.

This course adopts these as guiding principles. Fast algorithms are those where the worst case running time grows slowly with input size.

Usually, we want as close to linear $O(n)$ as possible.

2.2. **Lightning fast review of asymptotic notation.** The most common is $O(\cdot)$, $\Omega(\cdot)$, $\Theta(n)$. Let $T(n)$ be a function defined on $n = 1, 2, 3, \ldots$. This is usually the worst case running time of an algorithm.

**Definition 2.2.** $T(n) = O(f(n))$ if there exist constants $c, n_0 > 0$ such that

$$T(n) \leq c \cdot f(n)$$

for every $n \geq n_0$.

**Example 2.3.** If $T(n) = a_k n^k + \cdots + a_1 n + a_0$, then $T(n) = O(n^k)$.

*Proof.* Choose $n_0 = 1$. Choose $c = |a_k| + |a_{k-1}| + \cdots + |a_1| + |a_0|$. We need to show that for all $n \geq 1$, $T(n) \leq c \cdot n^k$. We have, for every $n \geq 1$,

$$T(n) \leq |a_k|n^k + \cdots + |a_1|n + |a_0| \leq |a_k|n^k + \cdots + |a_1|n^k + |a_0|n^k = c \cdot n^k. \qquad \square$$

**Example 2.4.** For every $k \geq 1$, $n^k$ is *not* $O(n^{k-1})$.

*Proof.* We prove this by contradiction. Suppose for contradiction that $n^k = O(n^{k-1})$. There exist constants $c, n_0 > 0$ such that $n^k \leq c \cdot n^{k-1}$ for all $n \geq n_0$. But then just cancel $n^{k-1}$, so that $n \leq c$ for all $n \geq n_0$, which is clearly false. $\square$

**Definition 2.5.** $T(n) = \Omega(f(n))$ if and only if there exist constants $c, n_0$ such that $T(n) \geq c \cdot f(n)$ for all $n \geq n_0$.

**Definition 2.6.** $T(n) = \Theta(f(n))$ if and only if $T(n) = O(f(n))$ and $T(n) = \Omega(f(n))$.

**Example 2.7.** $\frac{1}{2}n^2 + n$ is $O(n^3)$, $\Omega(n)$, and $\Theta(n^2)$.

## 3. 1/11: Advanced Divide and Conquer

The motivation here is that cool algorithmic ideas often need nontrivial analysis to evaluate.

Recall that grade-school multiplication of $n$-digit numbers takes $\Theta(n^2)$ time. The recursive approach was to write $x = 10^{n/2}a + b$ and $y = 10^{n/2}c + d$ where $a, b, c, d$ have $n/2$ digits. Then

$$x \cdot y = 10^n ac + 10^{n/2}(ad + bc) + bd.$$

We had two algorithms. The first was to recursively compute $ac, ad, bc, bd$. To write down the running time, we need a recurrence.

Let $T(n)$ be the worst-case runtime of this algorithm for $n$-digit numbers. We want to express $T(n)$ in terms of running time of recursive calls. The base case is $T(1) \leq$ constant. If $n > 1$, then $T(n) \leq 4T(n/2) + O(n)$.

We have a second algorithm that uses only three recursive calls. This is Gauss's algorithm, and here, we just need to recursively compute $ac$, $bd$, and $(a + b)(c + d)$. There are only three recursive calls, and we want to find a new recurrence. Again, $T(1) \leq$ constant and $T(n) \leq 3T(n/2) + O(n)$.

**Remark.** Mergesort would have almost the same recurrence, with the 3 replaced by 2.

We can now say some qualitative things about this recurrence. The second algorithm is better than the first, but worse than mergesort, which is $O(n \log n)$.

### 3.1. Master method.

The master method is a "black box" for solving recurrences.

We will assume that all subproblems have equal size, just as in all of the examples that we've seen so far.

Formally, suppose that we have a recurrence

$$T(n) = aT(n/b) + O(n^d)$$

where $a \geq 1$ is the number of recursive calls and $b$ is the input size shrinkage factor. We require $b > 1$ for the algorithm to terminate. $d$ is the exponent in the running time of the "combine" step. Note that $a$, $b$, $d$ need to be constants independent of $n$. There is also a base case $T(n) = O(1)$ for all $n$ sufficiently small.

**Proposition 3.1** (Master Method). *Given the above recurrence, there are three cases for the master method.*

$$T(n) = \begin{cases} O(n^d \log n) & \textit{if } a = b^d \\ O(n^d) & \textit{if } a < b^d \\ O(n^{\log_b a}) & \textit{if } a > b^d \end{cases}$$

We'll prove this in a bit, but it's fairly obvious.

**Example 3.2.** Sanity check: Merge sort. Here, $a = 2$, $b = 2$, and $d = 1$. We are in the first case, so $T(n) = O(n \log n)$.

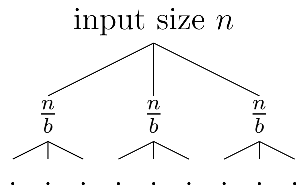
**Example 3.3.** First integer multiplication. $a = 4$ and $b = 2$ and $d = 1$. The runtime is then $T(n) = O(n^2)$.

**Example 3.4.** The second integer multiplication has $a = 3$, so $T(n) = O(n^{\log_2 3}) = O(1^{1.59})$. This is interesting; the first recursive algorithm is no better than the grade school algorithm, but Gauss's trick makes it much better.

**Example 3.5.** Suppose we have a recurrence $T(n) \le 2T(n/2) + O(n^2)$. Then we are in the second case, so $O(n^2)$.

*Proof of master method.* Let $T(1) = 1$. It doesn't matter that we assume this, but it makes life easier. Then we know that $T(n) \le aT(n/b) + cn^d$. Also, we will assume that $n$ is a power of $b$. The general case is similar but more tedious.

We make a recursion tree. The branching factor is $a$. Each operates on input of size $n/b$.



How many subproblems do we have at level $j$? There are $a^j$ problems, each with input size $m = n/b^j$, and there are $\log_b n$ levels. The total work performed at level $j$ (ignoring work in recursive calls) is

$$\le a^j c \left(\frac{n}{b^j}\right)^d = cn^d \left(\frac{a}{b^d}\right)^j$$

The total work is therefore

$$cn^d \sum_{j=0}^{\log_b n} \left(\frac{a}{b^d}\right)^j. \tag{2}$$

**Remark.** Here's how to intuitively interpret this formula. $a$ is the rate of subproblem proliferation, and $b^d$ is the rate of work shrinkage per subproblem.

First, suppose that $a = b^d$; the rate of proliferation equals the rate of work shrinkage. This is like merge sort; same amount of work per level, and now it's obvious how much work is done at each level.

If the rate of work shrinkage is larger, the root should dominate.

If subproblem proliferation is larger, then the work per level increases, so the bottom most level dominates.

We now evaluate (2). In the case $a = b^d$, this becomes

$$cn^d \sum_{j=0}^{\log_b n} \left(\frac{a}{b^d}\right)^j = cn^d \sum_{j=0}^{\log_b n} 1 = cn^d \log_b n = O(n^d \log n).$$

Note that

$$1 + r + r^2 + \cdots + r^k = \frac{r^{k+1} - 1}{r - 1} = \begin{cases} O(r^k) & r > 1 \\ O(1) & r < 1. \end{cases}$$

So taking $r = \frac{a}{b^d}$, if $a < b^d$, then

$$cn^d \sum_{j=0}^{\log_b n} \left(\frac{a}{b^d}\right)^j = cn^d O(1) = O(n^d).$$

For the final case, if $a > b^d$, then

$$cn^d \sum_{j=0}^{\log_b n} \left(\frac{a}{b^d}\right)^j = cn^d O\left(\left(\frac{a}{b^d}\right)^{\log_b n}\right) = cn^d O\left(a^{\log_b n} n^{-d}\right) = O(a^{\log_b n}) = O(n^{\log_b a}).$$

**Remark.** This last quantity is the number of leaves of the tree. The leaves dominate the running time. □

## 4. 1/13: Selection Algorithm

4.1. **Linear time median.** The input is an array $A$ with $n$ distinct integers and a given number $i$. The output is the $i$-th order statistic (i.e. $i$-th smallest number).

**Algorithm 4.1** ($n \log n$ algorithm)**.** Apply merge sort, return $i$th element of sorted array.

We'd like a sorting algorithm that's better, but sorting algorithm better than $n \log n$ does not exist. We can't do better if we insist on also solving the sorting problem. Without sorting, however, we can solve this problem in $O(n)$ time via divide and conquer.

There are two really nice algorithmic ideas. The first idea is to partition the input array around the pivot element. Pick an element of the array and call it the pivot element. Rearrange the array so that numbers less than the pivot are left of the pivot, and numbers greater than the pivot are to the right of it. This in particular means that the pivot is placed in its rightful position.

There are two cool facts about partitions. This is linear time (see the handout for details) and this reduces the problem size.

Consider now an underdefined version of the algorithm.

**Algorithm 4.2.**

```
(0) if n > 1, return A[1]
(1) p = ChoosePivot(A, n)
(2) B = Partition(A, n, p)
(3) Suppose $p$ is the $j$-th element of $B$.
      • if j = i, return p
      • if j > i, return Select(1st part, j-1)
      • else (j < i), return Select(2nd part, i-j)
```

This terminates because we only recurse on smaller problems. The proof of correctness is straightforward and uses induction (see the handout). Use the proof in the handout as a template for divide and conquer correctness proofs.

What would the running time of this algorithm be? This depends on ChoosePivot. If we make bad pivot choices, the algorithm will run slowly. A sequence of bad pivot choices will even make this run in quadratic time. What makes a good pivot? We want a balanced split; the median would be the perfect pivot. But that's what we were trying to compute originally, so this doesn't seem like a useful idea. However, what we can do is try to find a "pretty good" pivot very quickly.

The key idea is to do this recursively. We recursively compute a "median of medians." Here's the implementation of ChoosePivot.

**Algorithm 4.3.**

- break the array $A$ into group of 5 elements
- sort each group of 5 elements using merge sort
- get $n/5$ medians (middle element of each group)
- recursively find the medians of these, return as our pivot

Here's how to actually implement ChoosePivot:

1A. break $A$ into groups of 5 and sort each group
1B. Let $C$ be the middle group elements
1C. `return Select(C, n/5, n/10)`

Note that we are calling merge sort in a supposedly linear time algorithm. Isn't that bad? No, because merge sort is being applied to input sizes that are fixed, and it takes a finite and bounded number of operations to sort a list of 5 elements. By lecture 2, this takes $\leq 6(5\log_2 5 + 1) \leq 120$ per group, so it takes $\leq 120(n/5) = 24n = O(n)$ to sort the groups.

We consider the running time. Let's build a rough recurrence and fix it later. There exists a constant $c > 0$ such that $T(1) = 1$ and $T(n) \leq cn + T(n/5) + T(?)$. The $cn$ is the nonrecursive piece, $T(n/5)$ comes from finding the median of medians. If we actually got the median, $T(?)$ would be $T(n/2)$, but we might not be so lucky. We need the following lemma:

**Lemma 4.4.** *The second recursive call is guaranteed to be on an array of size $\leq \frac{7}{10}n$.*

*Proof.* Let $k = n/5$ be the number of groups. Let $x_l$ be the $l$-th smallest of the $k$ middle elements. In this notation, the pivot that we choose is $x_{k/2}$.

Envision the unsorted array $A$ as laid out in a grid. Each column will be a group of 5 elements, with the columns sorted by their middle element. The winning pivot is the one at the center. Now, there is a geographic region of the grid that we can confidently say is smaller than the pivot. Using transitivity of $<$, we see that the bottom left quadrant is certainly smaller than the center pivot. This means that $x_{k/2}$ is bigger than 3 out of 5 (60%) of $\approx 50\%$ of the groups. This means that the pivot is in fact bigger than at least 30% of $A$. Similarly, it is guaranteed to be smaller than $\geq 30\%$ of $A$. □

This means that we are justified in claiming the recurrence

$$T(n) \leq cn + T\left(\frac{n}{5}\right) + T\left(\frac{7}{10}n\right). \tag{3}$$

**Remark.** This algorithm was found in 1973. The creators were Blum, Floyd, Pratt, Rivest, and Tarjun. There are four Turing award winners among these five people.

4.2. **Substitution method.** See all Section 5.1 and 5.2, and handout 6.

The idea is more or less "guess and check," or in our case, "hope and check." We are hoping for linear time.

First, we guess an upper bound on $T(n)$. Then we try to verify by induction. If Step 2 fails, we might just be off by a bit; we might want to tweak the bounds.

*Proof of* (3). We are now going to guess that there exists a constant $a > 0$ such that $T(n) \leq an$ for all $n \geq 1$. Note that if this were true, then $T(n) = O(n)$. When using this method, it is a good idea to write out all of the constants.

We are going to try to reverse engineer a value for $a$. In order to streamline the lecture, we're just going to magically claim a value of $a$. We choose $a = \max(1, 10c)$. We analyze our recurrence (3), i.e.

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

The check is by strong induction on $n$. The base case is trivial since $T(1) = 1 \leq a$ since $a \geq 1$. The inductive hypothesis for $T(k) \leq ak$ for all $k < n$. Then

$$T(n) \leq cn + T\left(\frac{n}{5}\right) + T\left(\frac{7}{10}n\right) \leq cn + a\frac{n}{5} + a\frac{7}{10}n = n\left(c + \frac{9}{10}a\right) \leq an.$$

This proves our recurrence. □

**Remark.** This isn't too practical as it is, because it cannot be done in place; step 1B in ChoosePivot requires an auxiliary array. It only takes a small tweak to make it useful. Just pick at random, which is constant time. Then this is really fast.

## 5. 1/18: Basic graph algorithms

We want to be able to solve problems super quickly. This class will produce a toolbox of subroutines that are good. One example of this is sorting. We've seen a couple of this already, and we'll discuss a few for graphs.

5.1. **Representing graphs.** First, a few points of review. How would someone represent a graph? There's a couple ways to do it.

**Definition 5.1.** Let $G = (V, E)$ be a graph with vertices and edges. The number of vertices will be $n$ and the number of edges will be $m$.

We will think about simple graphs: only one edge between two vertices.

**Remark.** Suppose we have an undirected graph. How large could $n$ be as a function of $m$? The densely possible graph has $m = \binom{n}{2}$. How about if $G$ is directed? We'd have twice as many: $m \leq n(n-1)$. In any case, $m = O(n^2)$. Taking logs, we have $\log m = O(\log n)$. Therefore, we will be sloppy distinguishing $\log m$ and $\log n$.

If $G$ is connected, then $m \geq n - 1$.

One way to represent a graph is using an adjacency matrix. Represent $G$ by $n \times n$ 01 matrix $A$, where $A_{ij} = 1$ if and only if $G$ has an $ij$ edge. What would be the space of the adjacency graph representation. This is $O(n^2)$ space regardless of how many edges there are.

There is also the adjacency list. There is an array of vertices and an array of edges. Each edge points to both endpoints and each vertex points to their incident edges. The array of vertices takes space $\Theta(n)$ and the array of edges takes $\Theta(m)$. The edge pointers and vertex points are each $\Theta(m)$. The total is $\Theta(n+m)$. So for a sparse graph, this is much better.

Both of these representation schemes are used. In this class, we will only use the second one. The main reason is that we will primarily consider problems in graph search. Also, the types of graphs we care about in the 21st century are often large and sparse (such as web graphs).

5.2. **Graph primitives.** We will quickly review breadth and depth first search (BFS and DFS).

Breadth-first search is where you explore in a cautious, tentative way.

**Algorithm 5.2** (BFS)**.**

- Start at a node $S$ (layer 0).
- Explore the neighbors of $S$ (layer 1).
- Then neighbors of layer 1 nodes (layer 2).
- etc.

(We will a boolean per node to avoid exploring twice.)

Many people learn this using stacks and queues. In this case, we manage the next nodes to explore in a queue.

Depth first search is aggressive search, sort of like exploring a maze.

**Algorithm 5.3** (DFS)**.**

- When you reach a node, immediately start exploring its (not yet visited) neighbors.
- Keep track of next nodes via a stack, or alternatively, write a recursive loop.

Recall that both of these algorithms run in linear ($O(n+m)$) time. The details are in the book. The point is that the algorithm only visits any given node once. The claim is that we will look at each edge at most twice: once from each endpoint.

We will use this numerous times. This is one of the most fundamental graph primitives.

5.3. **Dijkstra.** Dijkstra's algorithm solves the single source shortest path problem.

We will consider a directed graph. Each edge has a length $c_e$. There is a source vertex $s$. For each possible destination $v \in V$, compute the length $D(v)$ of the shortest $s-v$ path in $G$.

A special case is when every edge has length 1. Then the shortest path is the one with fewest hops, and $D(v)$ gives the number of hops. In fact, Dijkstra is simply BFS in this case; it is simply a generalization of BFS to different edge lengths.

We will assume that for all $v \in V$, there exists a path from $s$ to $v$. In addtion, edge lengths must be nonnegative. We will eventually have an algorithm that can handle negative edge lengths. For example, we could model financial transactions using a graph, and negative numbers would be selling things.

The goal is to start somewhere and grow as a mold. Eventually, we'll cover the entire graph.

**Algorithm 5.4** (Dijkstra)**.**

- Initialize $X = \{s\}$.
- $A[s] = 0$

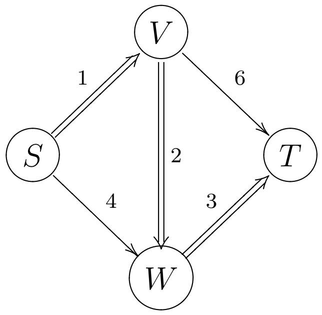
- $B[s]$ is the empty path.

The goal is that for all $v \in V$, $A[v] = D(v)$. We want the array $B[v]$ to contain the actual shortest paths from $s$ to $v$. $X$ will be the vertices whose $A$ values we've already computed.

- While $x \neq v$
- Among all edges $(v, w) \in E$ that "cross the frontier" ($v \in X$, $w \notin X$), pick the one minimizing
$$A(v) + c_{vw}.$$
Let this be the edge $(v^*, w^*)$.
- Add $w^*$ to $X$
- Set $A[v^*] := A[v^*] + c_{v^*w^*}$
- Set $B[w^*] := B[v^*] \cup (v^*, w^*)$.

Think of the frontier from $X$ to $V \setminus X$. At any iteration, we want to cross the frontier. The mold will advance by one step on each iteration.

**Example 5.5.** Consider vertices $S$, $V$, $W$, $T$, where $SW = 4$, $SV = 1$, $VT = 6$, $WT = 3$, and $VW = 2$.



Start at $S$. First, we include $V$, labeling it with 1. There are now three edges in the frontier. We advance using edge $VW$, and we label $W$ with 3. Finally, we want to get to $T$, and we choose the $WT$ edge, labeling it with 6, which is shortest path $S \to V \to W \to T$.

We should prove that this is correct.

*Proof.* This is by induction on the number of iterations. The base case should be clear.

Suppose that $A[v]$ and $B[v]$ have been computed correctly for all $v \in X$. In the inductive step, we pick $(v^*, w^*)$, and add $w^*$ to $X$. The length of the $s - w^*$ path $B[w^*]$ is the length of $B[v^*] \cup (v^*, w^*)$, which by the inductive hypothesis is $A[v^*] + c_{v^*w^*}$.

We want to say that this is the shortest path. Note that any such path must cross the frontier from $X$ to $V \setminus X$, so in particular, there is some initial edge $y - z$ that it uses to cross the frontier. Therefore, we can think of it as having three pieces: $s - y - z - w^*$. Let's lower bound each piece separately.

First, the $y - z$ path is only one edge, so it has length $c_{yz}$. We know the shortest path $s - y$ by the inductive hypothesis; this is at least $A[y]$. For the final path $z - w^*$, we don't know anything except that the length is $\geq 0$. Therefore, every path $s - w^*$ has length $\geq A[y] + c_{yz}$ for some $y, z$ edge across the frontier. Now, we use Dijkstra's greedy criterion to get that the length is $\geq A[v^*] + c_{v^*w^*}$. Therefore, our path is indeed the shortest one. $\square$

**Definition 5.6.** Let $G = (V, E)$ be undirected. The connected components of $G$ are the "pieces of G". Formally, these are the equivalence classes of the equivalence relation $u \sim v$ if $G$ has a $uv$ path.

**Proposition 5.7.** *We claim that BFS started at $s$ explores exactly the nodes in connected components of $S$. The run time is proportional to the size of the component.*

*Proof.* For all levels $i$, the level $i$ nodes of BFS are precisely the nodes that are $i$ hops away from $s$. (Check this; prove it by induction.) The point is that we can't miss anything. □

**Proposition 5.8.** *We can compute* all *connected components in an undirected graph in linear ($O(n+m)$) time.*

*Proof.*

```
For i = 1 to n
  if i not yet explored
    BFS from i
```

One BFS is called for each component. There is overhead $O(n)$ for each check in the loop, and we do one BFS per component, which have total runtime $O(\sum \text{component size}) = O(n+m)$. So overall, this is $O(m+n)$. □

## 6. 1/20: Connectivity in Directed Graphs

We do another lecture on linear time graph primitives. This time, we do directed graphs. They are important but are a bit trickier to work with.

It is even trickier to define things like connected components for directed graphs. A directed graph might be weakly connected in that there are edges between any two vertices. However, it might be impossible to go from certain origins to certain destinations.

**Definition 6.1.** The *strongly connected components* (SCCs) of a directed graph are equivalence classes of the equivalence relation $u \sim v$ if and only if there exists a path $u \to v$ and a path $v \to u$ in $G$.

We do need to check that this is an equivalence relation, but this is fairly easy.

The goal for this lecture is a linear time algorithm to compute the strongly connected components of a graph. Let's talk about the algorithm. It is very clever.

This will use depth first search.

**Algorithm 6.2.**

(1) Let $G^{rev} = G$ will have all arcs reversed.
(2) Run General-DFS on $G^{rev}$, let $f(v)$ be the finishing time of each $v \in V$.
(3) Run General-DFS on the original graph $G$, processing nodes in decreasing order of $f(v)$.

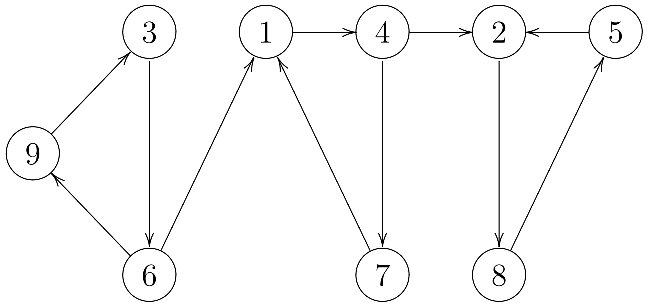
Here is General-DFS:

```
global var t = 0 [# nodes processed so far]
global var s = NULL [current source node]
Assume that nodes are labeled 1 to n
for i = 1 to n
  if i not yet explored
    set s := i
    DFS(G, i)
DFS(graph G, node i)
- mark i as explored
- set leader(i) := node s
- for each arc (i,j) in G
  - if j not yet explored
    - DFS(G, j)
- t++; set f(i) := t
```

At the end, the SCCs of $G$ are exactly the nodes with the same leader.

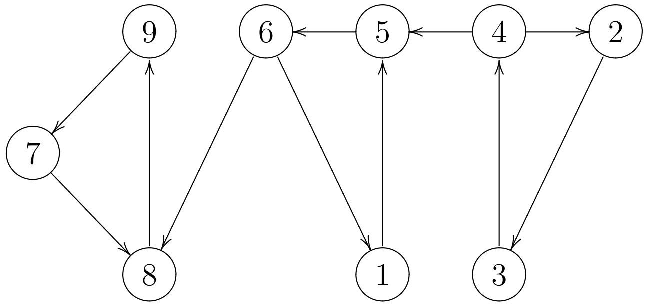
Let's do an example to show how this works.

**Example 6.3.** We first run DFS on $G^{rev}$.



After the first call to DFS, we have $f(7) = 1$, $f(5) = 2$, $f(8) = 3$, $f(2) = 4$, $f(4) = 5$, $f(1) = 6$. Finally, $f(9) = 7$, $f(6) = 8$, and $f(3) = 9$.

At the end, the SCCs of $G$ are the nodes with the same leader in the second call to DFS. In the second pass, we relabel the nodes with our $f(v)$ values.



Nodes 7, 8, 9 have leader = 9. Nodes 1, 5, 6 have leader = 6. Nodes 2, 3, 4 have leader = 4.

It seems that this works, but it isn't clear why it works.

Consider the running time of this algorithm. This is two depth first searches plus some overhead, which is certainly linear $O(m + n)$ time. What about correctness?

**Note.** The SCCs of a graph form an acyclic metagraph, where the meta nodes are the SCCs $C_1, \ldots, C_k$. There is an arc $C \to \hat{C}$ if and only if there exists an arc $(i, j)$ with $i \in C$ and $j \in \hat{C}$. In the example above, this is simply

$$\circ \to \circ \to \circ.$$

This is acyclic because if there were a cycle of SCCs then they would collapse into a single SCC. This is a cool way of looking at directed graphs.

Here's a key lemma:

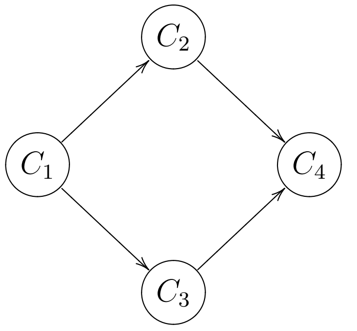
**Lemma 6.4.** *Consider two "adjacent" SCCs $C_1$ and $C_2$ in a graph $G$. This means that there are $i \in C_1$ and $j \in C_2$ such that there exists an arc $i \to j$. Let $f(v)$ be the finishing times of general-DFS in $G^{rev}$. Then,*

$$\max_{v \in C_1} f(v) < \max_{v \in C_2} f(v).$$

Let's sketch the proof of correctness using this key lemma.

**Corollary 6.5.** *The maximum $f$ value must lie in a "sink SCC", i.e. an SCC that is not connected to any other SCC.*

**Example 6.6.** Consider the graph of SCCs



$$\begin{tikzcd} & C_2 \arrow[dr] & \\ C_1 \arrow[ur] \arrow[dr] & & C_4 \\ & C_3 \arrow[ur] & \end{tikzcd}$$

Say $C_i$ has maximal $f$ value $f_i$. Then $f_1 < f_2$ and $f_3 < f_4$. Here, $C_4$ is a sink SCC.

*Correctness Intuition of Algorithm 6.2.* The second pass of General-DFS begins somewhere in a sink SCC $C^*$ of $G$. Since we cannot escape a sink SCC, we will find everything in $C^*$ but not be able to discover anything elsewhere. The first call to DFS discovers $C^*$ and *nothing else.*

Now, consider the next call of DFS. We effectively recurse on the graph after peeling off the already explored $C^*$. This starts in a sink SCC of the residual graph $G \setminus C^*$.

Successive calls of DFS peels off the SCCs one by one. □

That's why the algorithm works. Fundamentally, the first DFS call is just a preprocessing step. The point is that we will be able to start at sink SCCs, providing an order to find SCCs. This guarantees that the second DFS call works as intended.

*Proof of key lemma 6.4.* Consider $G^{rev}$. Suppose we have $j \in C_2$ and $i \in C_1$, and there is an arc $j \to i$ in this reversed graph. Note that SCCs are preserved under graph reversal.

Let $v$ be the first node in $C_1 \cup C_2$ reached by the DFS call on $G^{rev}$. There are two cases.

Suppose that $v \in C_1$. DFS is going to find everything in $C_1$ but nothing in $C_2$ because the metagraph is acyclic and hence contains no arc $C_1 \to C_2$. Therefore, all of $C_1$ is explored before any of $C_2$. Hence, $f(x) < f(y)$ for every $x \in C_1$ and $y \in C_2$.

In the second case, if $v \in C_2$, DFS will find everything in $C_2$ and it will find everything in $C_1$ as well because of the arc $C_2 \to C_1$. DFS from $v$ will not finish until all of $C_1 \cup C_2$ is completely explored, in particular, $f(v) > f(w)$ for all $w \in C_1$. This proves the key lemma. □

This is a great algorithm if everything fits in main memory. It is very fast. What about enormous graphs that don't fit in memory?

6.1. **Application: structure of the web.** Consider an application to the structure of the web. Broder et al published a paper in 2000 considering the SCCs of the web graph. This was a nontrivial task. Search engines now have systems to do this, but 2000 was a long time ago. This paper proposed something called the bowtie structure.

They found a giant strongly connected component in the web graph, but there are other SCCs as well. There are SCCs from which you can get to the core but not conversely (e.g. new pages), and there are SCCs from which you get to from the core but not conversely (e.g. corporate websites). There were also tubes and tendrils of other SCCs.

The main findings were:

(1) All four parts have roughly the same size.
(2) Within the core, the graph is very well connected. This is the small world property.
(3) Outside the core, the graph is surprisingly poorly connected.

## 7. 1/25: Greedy algorithms

The exasperating thing about algorithm design is that there is no silver bullet. We will discuss the major design paradigms to form a toolbox.

7.1. **Overview of greedy algorithms.** What is a greedy algorithm? You know it when you see it. The point is to iteratively make "myopic" decisions and hope that it works out in the end.

**Example 7.1.** Dijkstra is a greedy algorithm. This was a one-pass algorithm, and we made a choice at each node.

This has a different flavor than divide and conquer. In divide and conquer, it might take a bit of thought to see the correct algorithm to recursively solve the problem. For greedy algorithms, on the other hand, they are really easy to propose. In addition, the running time is very easy: sort things and do one pass. For divide and conquer, correctness was usually mechanical, but for greedy algorithms, proving correctness can be tricky. There are often greedy algorithms that seem correct but do not work. It take practice to prove correctness: There isn't a simple template.

Here are some recurrent principles of correctness proofs. There are two methods. The first method is the "greedy stays ahead" proof. This uses proof by induction, and Dijkstra is the canonical example of this. The second method is the "exchange argument". This is what we will mainly discuss today. Ultimately, just do whatever works.

7.2. **Applications.** There are some applications of greedy algorithms.

**Example 7.2.** Optimal caching. We have different types of memory, in a memory hierarchy. When programs request memory, we want it to be in the small fast memory. To get something not in fast memory, we have to evict something from fast memory to make space. This is an algorithmic problem.

**Theorem 7.3.** *We kick out whatever we will need furthest in the future. This minimizes the number of cache misses.*

Not all greedy algorithms are correct, but this is correct. However, it requires full knowledge of the future, which in practice doesn't happen. It does provide a good benchmark for comparing actual performance to what is optimal.

7.3. **Scheduling.** Today, we will discuss the application of scheduling.

Suppose we have a shared resource, and a lot of people or processes want to use this resource. We want to sequence the jobs on the processor. Of course, this means that we need an objective function. There are numerous ways to define an objective function; we'll consider one example.

The input is $n$ jobs. Each has a weight $w_j$ (importance) and length $l_j$ (needed time). Our goal is to sequence these jobs.

**Definition 7.4.** The completion time $c_j$ of a job $j$ is the sum of the job lengths up to and including $j$.

**Example 7.5.** Suppose we have three jobs, $l_1 = 1$, $l_2 = 2$, and $l_3 = 3$, and they all have equal weight $w_1 = w_2 = w_3 = 1$. If we schedule in order $1, 2, 3$, our completion time would be $c_1 = 1$, $c_2 = 3$, and $c_3 = 6$.

**Problem 7.6.** Our goal is to minimize the weighted sum of the completion times $c_j$:

$$\min \sum_{j=1}^{n} w_j c_j.$$

In our example 7.5, this minimum total completion time is $1 + 3 + 6 = 10$.

Why would we schedule a job before another. Suppose two jobs have the same length. Then we should prefer jobs with larger weight. What if we considered two jobs of equal weight. Then smaller length is better. This is because a job of smaller length has less impact on future lengths.

What if $l_i < l_j$ but $w_i < w_j$? It's not clear what to do, but a greedy algorithm might work. The idea is to assign a "score" for each job, and sequence jobs by score. Here, scores should be decreasing in weight and increasing in length. What are some ways to combine $l_i$ and $w_i$ satisfying this property?

Guess 1. Order jobs in increasing value of $l_j - w_j$.

Guess 2. Order by $l_j/w_j$.

Here, we break ties arbitrarily. We immediately know that at least one of these are wrong because they are different algorithms. Are either of them right?

To distinguish, we want to find a minimal example where guesses 1 and 2 give different solutions.

**Example 7.7.** Suppose $l_1 = 5$, $w_1 = 3$ and $l_2 = 2$, $w_2 = 1$.

Guess 1 prefers smaller differences and puts the second job first. The sum of weighted completion times of this schedule is $1 \cdot 2 + 3 \cdot 7 = 23$.

Guess 2 will do the jobs in the other order, putting the first job first. The sum of weighted completion times is $3 \cdot 5 + 1 \cdot 7 = 22$.

Therefore, the first algorithm is not correct for this example.

We have strong ambition to have an algorithm that is always correct. We can therefore run out the first algorithm.

**Proposition 7.8.** *In fact, we claim that the second algorithm is always correct.*

The proof is a completely canonical exchange argument.

*Proof by exchange argument.* Consider some input of $n$ jobs. Without loss of generality, rename the jobs by ratio. After this, our greedy algorithm picks the schedule $\sigma = 1, 2, 3, \dots, n$. Thus,

$$\frac{l_1}{w_1} \le \frac{l_2}{w_2} \le \frac{l_3}{w_3} \le \cdots \le \frac{l_n}{w_n}.$$

Consider any other schedule $\sigma^*$. Our goal is to prove that $\sigma$ is at least as good as $\sigma^*$. This would imply that $\sigma$ is the optimal schedule, which is what we want to show.

**Lemma 7.9.** *Note that if $\sigma^* \ne \sigma$, then at some point in $\sigma^*$, there is a job $j$ immediately after a job $i$ with higher index $j < i$.*

*Proof.* Consider the contrapositive. Consider a schedule $\sigma^*$ where job indices only go up. Then $\sigma^*$ must be the greedy order. □

In $\sigma^*$, we have the inverted jobs $j < i$ such that $j$ is after $i$. Consider swapping $j$ and $i$ in $\sigma^*$ and leaving everything else the same. This is modifying $\sigma^*$ to look more like the claimed optimal schedule $\sigma$. We do a cost benefit analysis of this swap of jobs $i$ and $j$.

For a job $k$ other than $i$ and $j$, $c_k$ is unaffected. For job $i$, the completion time goes up by $l_j$. This means that $c_i$ goes up by $l_j$, so its objective goes up by $w_i l_j$. Similarly, for job $j$, the completion time $c_j$ goes down by $l_i$, so the objective decreases by $w_j l_i$.

We haven't yet used the greedy rule yet, and we should do so here. Our claim is that the benefit exceeds the cost. We think of this inversion as correcting a flaw in $\sigma^*$. Using the fact that $j < i$, this means that

$$\frac{l_j}{w_j} \le \frac{l_i}{w_i},$$

so clearing denominators, this means that $w_i l_j \le w_j l_i$. This precisely means that the benefit does indeed exceed the cost because of our greedy assumption.

The point is that by exchanging an "adjacent inversion" (such as $i$ and $j$ above), we only make $\sigma^*$ better, and it decreases the number of inverted pairs. After $\le \binom{n}{2}$ such exchanges, we can transform $\sigma^*$ into $\sigma$. This is like bubble sort. Therefore, $\sigma$ is at least as good as $\sigma^*$, so the greedy algorithm is optimal. □

In some cases, we can view correctness as a resource that we can trade with running time. We might give up some correctness in exchange for a large speed up in running time. We won't consider this type of problem now.

## 8. 1/27: Minimum Spanning Trees

For this class, the purpose is mostly pedagogical. This does have real applications, however.

The input is an undirected graph $G = (V, E)$ and cost $c_e$ for each edge.

$$\begin{tikzcd}
A \arrow[r, "1"] \arrow[d, "4"'] \arrow[dr, "3"] & B \arrow[d, "2"] \\
C \arrow[r, "5"'] & D
\end{tikzcd}$$

The output should be the min-cost tree $T \subset E$ that spans all vertices.

We will assume that $G$ is connected and all of the edges have distinct costs. It doesn't matter if there are ties; we just won't consider that case here.

This is like Dijkstra, in that we have a mold growing with a frontier. The intuition and the picture are more or less the same.

**Algorithm 8.1** (Prim's MST Algorithm)**.**

```
• Initialize $X = \{s\}$. This is chosen arbitrarily.
• $T = 0$. We have the invariant that $X$ will be the vertices spanned so far by $T$.
• while $X \neq V$
    – Let $e = (u, v)$ be the cheapest edge with $u \in X$, $v \notin X$.
    – Add $e$ to $T$
    – Add $v$ to $X$
```

**Example 8.2.** In the graph above, starting at node $A$, Prim would select edges $AB$, $BD$, and $AC$, in that order.

8.1. **Correctness.** We claim that Prim always computes a minimum spanning tree.

**Definition 8.3.** A *cut* of a graph is a nonempty partition of the vertices of a graph into two pieces. An edge connecting vertices of the two partitions is called a *crossing edge.*

**Lemma 8.4** (Empty cut lemma)**.** *A graph is not connected if and only if there exists a cut* $(A, B)$ *with no crossing edges.*

*Proof.* This is easy. Check this at home and just talk through it. □

**Lemma 8.5** (Double crossing lemma)**.** *Suppose we have a cycle* $C \subset E$ *of a graph, and this cycle has an edge that crosses a cut* $(A, B)$*. Then some other edge of* $C$ *also crosses* $(A, B)$*. In general, a cycle crosses a cut an even number of times.*

**Corollary 8.6** (Lonely cut corollary)**.** *If* $e$ *is the only edge crossing some cut* $(A, B)$*, it is not in any cycle.*

Before we prove that Prim is a minimum spanning tree, we should first make sure that it is actually a spanning tree.

**Lemma 8.7.** *Prim outputs a spanning tree.*

*Proof.* By induction, this maintains the invariant that $T$ spans $X$. Do this in the privacy of your own home.

Why can't we get stuck with $X \neq V$? This would mean that we failed to find an edge crossing the frontier, which by the Empty Cut Lemma 8.4 would imply that $G$ is not connected.

Why are there no cycles? When $e$ is added to $T$, it is the first edge of $T$ to cross this cut $(X, V \setminus X)$. This means that its addition cannot create a cycle in $T$ by the Lonely Cut Corollary 8.6. □

There are a lot of spanning trees, but Prim cleverly finds the best one.

Even though Prim looks really like Dijkstra, the proofs of correctness are very different. For Dijkstra, we used a greedy stays ahead proof; for Prim, we use an exchange argument. This is used to proved the following Cut Property.

**Proposition 8.8** (Cut Property)**.** *Suppose that $e$ is the cheapest edge of the graph $G$ crossing some cut $(A, B)$. Then $e$ belongs to every minimal spanning tree.*

A greedy algorithm is something that makes myopic decisions. The cut property is precisely the type of property that we want to use to justify greedy algorithms.

**Proposition 8.9.** *The cut property implies the correctness of Prim.*

*Proof.* By our lemma 8.7, Prim outputs a spanning tree $T^*$.

Prim considers a sequence of $n - 1$ cuts, and it by definition the cheapest edge crossing that cut. Each edge of $T^*$ is therefore justified by the Cut Property 8.8. Therefore, $T^*$ is the (unique) minimal spanning tree. □

*Proof of the Cut Property 8.8.* We will prove this by contradiction, using an exchange argument.

Suppose that $e$ is the cheapest edge crossing some cut $(A, B)$ but *not* is an minimal spanning tree $T$. $T$ is connected, so the Empty Cut Lemma 8.4 implies that $T$ contains an edge $f \neq e$ that crosses $(A, B)$.

We know that $c_e < c_f$. Then $T \cup \{e\} \setminus \{f\}$ has lower cost than the original $T$. However, this might no longer be a spanning tree. We just need to be a little bit more careful.

$$\begin{tikzcd}
\circ \arrow[r, "e'"] \arrow[d] & \circ \arrow[d] \\
\circ \arrow[r, dash, dotted, "e"] \arrow[d] & \circ \\
\circ \arrow[r, "f"] & \circ
\end{tikzcd}$$

Is there some other swap that would work? Let $C$ be the cycle created by adding $e$ to $T$. Now we apply the Double Crossing Lemma 8.5 to see that there is some other edge $e'$ of $C$ (also in $T$) that crosses $(A, B)$.

Check that $T' = T \cup \{e\} \setminus \{e'\}$ is a spanning tree. Since $c_e < c_{e'}$, this is cheaper than $T$, which is a contradiction. □

That was the mathematical part of the lecture. Let's talk about implementation and running time.

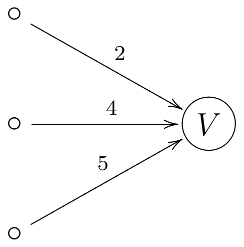
8.2. **Running time.** The naive running time is to have $O(n)$ iterations and $O(m)$ time per iteration, so $O(mn)$ overall.

Data structure play two roles. We will see data structures so that we know what they do. They will also be used to speed up algorithms like Prim's algorithm.

8.2.1. *Speed up via heaps.* Recall that heaps support the operations of `Insert`, `Extract-Min`, and `Delete`. All of these operations take time linear in the height of the tree (to restore the heap property after a change), so these are all $O(\log n)$.

There will be two invariants. The elements in the heap will be $V \setminus X$. For $v \in V \setminus X$, the key value of $v$ is the cheapest edge $(u, v)$ with $u \in X$, or $+\infty$ if no such edges exist.

**Example 8.10.** For example,



would have that the key of $v$ is 2.

**Note.** `Extract-Min` yields the next vertex $v$ and edge $(u, v)$ to add to $X$ and $T$.

The point is that this is like a two-round knockout tournament. First, we obtain minimum edges locally at each vertex, and then we choose the minimum of the minimums, which is a global minimum. This isn't how it is implemented, but it's a nice way to think about things.

The above note implies that we only have to do an `Extract-Min` at each iteration. We should check that our invariants are preserved. The first invariant causes no problems. The second invariant, though, might get broken. After one iteration, $v$ is no longer in the heap. The frontier advances, which means that the edges crossing the frontier changes; edges that used to not cross the frontier will now cross the frontier, and key values can drop. In each iteration, we also need to update our key values to preserve the invariants.

When $v$ is added to $X$, only edges incident to $v$ need to be updated.

- for each edge $(v, w) \in E$
  - if $w \in V \setminus X$
    - delete $w$ from heap
    - recompute key$[w] = \min(\text{key}[w], c_{vw})$
    - reinsert $w$ into the heap

8.3. **Running time with heaps.** In this implementation, all we do is use the heap, so the running time is dominated by heap operations. We do one `Extract-Min` for each iteration, for $(n - 1)$ `Extract-Min`s. We will do the key restoration once for each edge: each time when the first endpoint of the edge becomes incorporated into $X$. Each edge triggers one `Delete` and `Insert`. So we have in total $O(m)$ heap operations. (Recall that $m \geq n - 1$ for connected graphs, so $O(m + n) = O(m)$.)

Each heap operation takes time $\log n$, so this takes time $O(m \log n)$.

For Dijkstra, the same argument holds. The only difference is that we use the Dijkstra greedy rule instead of the greedy edge crossing, so we let the key of $v$ be the minimum of $A[u] + c_{uv}$ over all edges $(u, v)$ crossing the frontier.

## 9. 2/1: Kruskal's MST algorithm

Recall that we are working with minimum spanning trees. As before, the input is a undirected graph $G = (V, E)$ and we output a minimum spanning tree that is connected and has no cycles. We assume that $G$ is connected and has distinct edge costs. Last time, we gave a number of properties about cuts of graphs, and we will use these properties again today.

We consider another greedy algorithm to find minimum spanning trees.

**Algorithm 9.1** (Kruskal's MST Algorithm)**.**

```
• Sort edges in order of increasing cost
    [rename edges $\{1, 2, \ldots, m\}$ so that $c_1 < c_2 < \cdots < c_m$]
• $T = \emptyset$
• for edges $i = 1$ to $m$
    – if $T \cup \{i\}$ has no cycles
        ∗ Add $i$ to $T$.
• Return $T$
```

Just like in the case of Prim's algorithm 8.1, it's not at all obvious that this outputs a minimum spanning tree.

## 9.1. **Correctness.**

**Theorem 9.2.** *Kruskal is correct.*

*Proof.* Let $T^*$ be the output of Kruskal. There are clearly no cycles by the definition of the algorithm.

We claim that $T^*$ is connected. It is sufficient to show that $T^*$ crosses every cut (by the Empty Cut Lemma 8.4). So fix a cut $(A, B)$.

The Lonely Cut Corollary 8.6, Kruskal will include the first (i.e. cheapest) edge it sees crossing $(A, B)$ because this edge cannot yet participate in a cycle, and the only reason Kruskal might skip the edge is because of cycles. This shows that Kruskal outputs a spanning tree.

We now claim that that every edge of $T^*$ is justified by the Cut Property 8.8. As we discussed last time, this would imply that $T^*$ is the unique minimum spanning tree.

To see this final claim, suppose that $(u, v)$ is an edge that Kruskal added to $T$. At this time, $T$ had no $(u, v)$ path since otherwise there would be a cycle. Therefore, the edges $T$ that we have chosen so far has $u$ and $v$ in different connected components. We can therefore cut the graph with some cut $(A, B)$ such that $u \in A$ and $v \in B$ and such that there are no edges of $T$ crossing it. Now, the greedy property of Kruskal will take the cheapest edge that creates no cycles. Since there are no edges crossing $(A, B)$, the first edge crossing $(A, B)$ creates no cycles, and therefore the edge $(u, v)$ is added by Kruskal and justified by the Cut Property 8.8. □

In Prim's Algorithm, it is obvious where the cuts are, while in Kruskal it is not obvious, so this proof was a bit more subtle. Fundamentally, the Cut Property was what we needed.

9.2. **Running time.** First, we consider the naive running time of Kruskal's algorithm. The time to sort the $m$ edges is $O(m \log m)$. We do $O(m)$ loop iterations, and we need $O(n)$ time to check for cycles, using DFS or BFS in the graph $(V, T)$, yielding $O(mn)$ overall. We've been getting spoiled by near linear time algorithms, and we'd like this to be like an algorithm as well. This takes a bit of work. For Prim and Dijkstra, this was done using heaps. Here, we need a new data structure.

## 9.3. **Speed-up via Union-Find.**

9.3.1. *Idea 1: Union-Find.* We will maintain one auxiliary linked structure per connected component of $(V, T)$. Suppose we have

$$\begin{tikzcd}
 & & 2 & \\
 & 1 \arrow[ur] & 0 \arrow[u] & 1 \arrow[ul] \\
0 \arrow[ur] & 0 \arrow[u] & & 0 \arrow[u]
\end{tikzcd}$$

Define the rank of a node $v$ in the linked structure to be the number of hops in the linked structure in the longest path to $v$ (from a leaf). Hence, $\text{rank}(v) = \text{maximum child rank} + 1$.

Here is the `find` operation in this data structure:

- Given $v$, return $v$'s leader by chasing up parent pointers.

The running time of this operation is $O(\text{max rank})$.

Therefore, we can check if $T \cup \{(u, v)\}$ has a cycle with two `find` operations. There is a cycle if and only if $u$ and $v$ have the same leader. Therefore, we use $O(\text{max rank})$ cycle checks. If we can keep this around $O(\log n)$, then Kruskal would be near linear time.

We also have the `union` operation. If we add a new edge $(u, v)$ to Kruskal's tree $T$, we merge two components of $(V, T)$. This requires combining the new linked structures $c_1$ and $c_2$. To do this, have the leader of $c_2$ point to the leader of $c_1$.

The running time of `union` operation requires two `find` operations (to find the leaders of each structure) and an extra $O(1)$ to make the new pointer.

9.3.2. *Idea 2: Union by Rank.* We want to keep the maximum rank as small as possible. What would be bad is long chains, since that gives slow cycle checks. The idea is to preferentially pick which of roots of the two linked structures becomes the new root. This is called *union of rank* to keep the trees balanced.

- Store the rank at each node.
- Make roots with larger rank the new root.

Therefore, all ranks are unchanged except possibly the new root (which might have a new child). In addition, if two roots have equal root $r$ before the `union` operation, the new root has rank $r + 1$. Otherwise, the new root rank is the same as before.

The hope is that this simple heuristic will help us avoid long chains. In fact, these trees aren't balanced, but they are at least as dense as a perfectly balanced binary tree.

**Claim.** At all times, if node $v$ has rank $r$, then the subtree rooted at $v$ has at least $2^r$ nodes.

Since each component has at most $n$ nodes, the claim implies that $\max_v \text{rank}(v) \le \log_2 n$. Therefore, cycle checking is $O(\log n)$, implying that Kruskal's algorithm is $O(m \log n)$. This means that running time of Kruskal is dominated by the initial step of sorting the edges.

*Proof of Claim.* We proceed by induction on the number of `union` operations.

This clearly holds initially when $\text{rank}(v) = 0$ for all $v$. In the inductive step, consider a union.

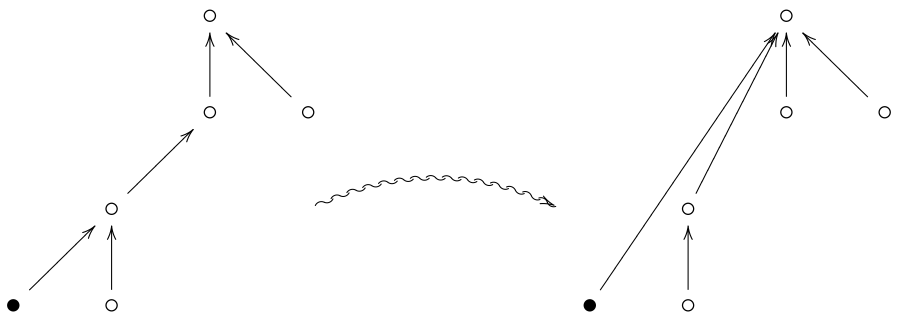
By the inductive hypothesis, we only need to check the case where the new root $u$ has rank one more than before. This is because if no ranks change then any lower bound on the number of nodes that was true before is still true now.

Suppose that the rank of $u$ increases from $r$ to $r+1$. All other ranks stay the same size and subtree sizes only increase. As we noted before, this happens only when the two old roots $u$ and $v$ both have rank $r$. By the inductive hypothesis, the old subtrees previously rooted at $u$ and $v$ each had $\geq 2^r$ nodes. Together, the two subtrees have $\geq 2^r + 2^r = 2^{r+1}$ nodes. □

9.3.3. *Open research questions.* Is there a linear minimum spanning tree algorithm? We know that we can actually beat $O(m \log n)$. In the mid 90's, there was a randomized linear time algorithm. Without randomization, it is still unknown.

9.3.4. *Idea 3: Path compression.* This further optimizes Union-Find, though it won't help Kruskal's running time. This is a natural optimization to think about.

After or during `find` from a vertex, update all pointers on the $v \to$ root path to point directly to the root.



This adds only constant overhead to find; the external behavior is unchanged. Suppose that we did two find calls. The first call should take $O(\log n)$ time, while the second call should be $O(1)$.

It is sort of remarkable that this can be quantified.

**Theorem 9.3.** *Every sequence of $m$ union-find operations with path-compression takes $O(m \log^* n)$ time.*

**Definition 9.4.** $\log^* n$ is defined as the number of times you can apply the logarithm function to $n$ until you get to 1. For all practical purposes, this is constant.

This is good, but can we do better? It turns out that this analysis of path compression was a bit sloppy. There is something known as the Ackermann function. This grows so quickly that is not primitive recursive.

**Definition 9.5.** Define $A_0(r) = r + 1$. For $k \geq 0$, $r \geq 1$, define $A_{k+1}(r)$ is applying $A_k$ $r$ times to $r$.

**Example 9.6.** $A_1(r) = 2r$. For example, $A_1(2) = 4$.

$A_2(r) = r2^r$. For example, $A_2(2) = 8$.

$A_3(r)$ is bigger than a tower of $r$ twos: $2^{2^{2^{\cdot^{\cdot^{\cdot}}}}}$, so $A_3(2) = 2^{11}$.

$A_4(r)$ is much bigger than the number of atoms in the universe.

**Theorem 9.7.** *Actually, every sequence of $m$ union-find operations with path-compression takes $O(m\alpha(n))$ time, where $\alpha(n)$ is the smallest $k$ such that $A_k(2) \geq n$. (Here, $\alpha(n)$ is* not *$O(1)$.) Moreover, this is the best we can do; there is a matching lower bound.*

## 10. 2/3: Greedy algorithms

This will the final lecture in our sequence on greedy algorithms.

10.1. **More on Union-Find.** Recall that we have been working with Kruskal's algorithm, which suggested that we use the union-find data structure. There is one rooted tree per connected component. The only two supported operations are `union` and `find`.

We had the idea to union by rank. Here, rank($v$) is the length of the longest path to go to if from a leaf. When two roots with the same rank are linked, we increment the rank of the new root. Note that $v$ is not a root, it will never become a root, and hence its rank will never change again. These ranks can be maintained in constant time, and the ranks always increase.

Last time, we showed:

**Proposition 10.1.** rank($v$) $= r$ *implies that the subtree rooted at* $v$ *has at least* $2^r$ *nodes.*

Last time, we showed that this means that the maximal rank is $\log_2 n$, so Kruskal runs in $O(m+n)$ time. We want to refine this today.

**Corollary 10.2.** *At all times, at most* $n/2^r$ *nodes have rank* $r$*.*

We also had the idea of path compression. After `find`, we redirect all pointers on the path straight to the root. We do *not* update ranks. After path compression, note that rank($v$) is no longer the maximum number of hops to $v$ from a leaf. The corollary 10.2 above still holds.

**Theorem 10.3** (Hopcraft-Ullman)**.** *$m$ operations of union-find take $O((m+n)\log^* n)$ time. This is not constant, but for all practical purposes, it is a tiny constant.*

This is a subtle difference between what we show now and what we showed before. Last time, we showed that any operation takes $\log n$ time. This is no longer true here. Some operations take more time, while some operations take less. In total, we get the above bound averaged over all operations. This is sometimes called amoritized analysis.

*Proof.* We define sets of numbers called *rank blocks.* The smallest examples are

$$\{0\}, \{1\}, \{2\}, \{3, 4\}, \{5, \dots, 16\}, \{17, \dots, 2^{16}\}, \{2^{16}+1, \dots, 2^{2^{16}}\}, \cdots .$$

The pattern should now be obvious. Note that there are $O(\log^* n)$ blocks. Our notion of rapid progress will be to say that we usually go from one rank block to a higher ranked rank block.

**Definition 10.4.** A node $v$ is *good* if

(1) $v$ or $v$'s parent is the root of the tree.
(2) $v$'s parent is in a higher rank block than $v$.

$v$ is *bad* otherwise.

If there were no bad nodes in the world, we'd be done because all find operations would be taking $\log^* n$ time already. More generally, each call to `find` visits only $O(\log^* n)$ good nodes. Additionally, if `find` visits a bad node $w$, then $w$'s parent pointer will be updated.

To finish the proof, we will show that for each rank block, there will be at most $n$ pointer updates to bad nodes in the block.

To see this, fix a rank block $\{k+1, \ldots, 2^k\}$. Corollary 10.2 tells us that the number of nodes in this block is

$$\leq \frac{n}{2^{k+1}} + \frac{n}{2^{k+2}} + \frac{n}{2^{k+3}} + \cdots \leq \frac{n}{2^k}$$

after summing the geometric series.

Consider a node in this rank block. Every time the pointer of $v$ is updated, its new parent is an ancestor of the previous parent, so the parent ranks increase. Once $v$'s pointer is updated $\geq 2^k$ times, this points to something outside of the current rank block. Therefore, it must become a good node. Furthermore, once it becomes a good node in this way, its parent will always be in a higher rank block and hence $v$ will be forevermore be a good node.

Summing over all nodes in the rank block, this means that there will be

$$\leq \frac{n}{2^k} \cdot 2^k = n$$

bad node pointer updates in this block. □

10.2. **Clustering.** Given a set of "points," we want to classify these points into "coherent groups." This is called *unsupervised learning* in machine learning.

Assume that we are given some measure of similarity. This is a distance $d(p, q)$ between each point pair, and like most distances, it is symmetric, so that $d(p, q) = d(q, p)$. For example, this could be physical distance or the NW score from the first lecture. Points are said to be in the same cluster if they are nearby. Assume that $k$ is the number of clusters desired.

**Definition 10.5.** We say that two points $p$ and $q$ are *separated* if they are in different clusters. The *spacing* of a clustering is defined as

$$\min_{\substack{(p,q)\\ \text{separated}}} d(p, q).$$

We want to maximize the spacing to increase the separation between clusters.

Let's use a greedy algorithm.

**Algorithm 10.6.**

- initially, put each point in a separate cluster
- repeat until there are $k$ clusters
  - let $p$ and $q$ be the closest pair of separated points
  - merge clusters of $p$ and $q$ together.

This is exactly like Kruskal's algorithm, except we stopped earlier instead of running the algorithm to $k = 1$; we stopped before the last $k-1$ edges were added. The vertices are the points, the edges are a complete graph, and the costs are distances between points. This flavor of algorithm is called *single-link clustering.*

**Theorem 10.7.** *Greedy is correct for algorithm 10.6.*

*Proof.* Let $c_1, \ldots, c_k$ be greedy clusters, and let $\widehat{c_1}, \ldots, \widehat{c_k}$ be some other clusters. Let $p$ and $q$ be points such that

(1) they are in the same greedy cluster $c$
(2) $p \in \widehat{c_j}$ but $q \notin \widehat{c_j}$ for some $j$.

Let $D^*$ be the spacing of the greedy clusters $c_1, \dots, c_k$. Suppose we have a number of clusters; two of these are $D^*$ apart. Within each cluster, there are edges, where an edge between $x$ and $y$ denotes that the greedy algorithm merged $x$ with $y$. Since the algorithm was greedy, all merged pairs of points have distance $\leq D^*$.

Now, since $p$ and $q$ are in the same cluster $C_i$, there exists a sequence of points inside $C_i$ that starts at $p$ and terminates at $q$. Each edge in this sequence has distance $\leq D^*$.

Since $p \in \widehat{c_j}$ and $q \notin \widehat{c_j}$. At some point, this path leaves $\widehat{c_j}$. There exists a consecutive pair $p_i, p_{i+1}$ that are in the path but are separated in $\widehat{c_1}, \dots, \widehat{c_k}$. The spacing of the $\widehat{c}$'s is $\leq D^*$, since we already showed that $d(p_i, p_{i+1}) \leq D^*$.

Therefore, this other cluster is no better than greedy, and hence the greedy algorithm is optimal as promised. □

## 11. 2/8: Randomized Algorithms

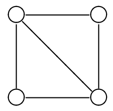
Today we start to discuss randomized algorithms. The basic idea is that we allow our algorithms to "flip coins" during their operation. We've already seen an example of this in the linear-time median algorithm, with Partitioning around a random pivot. For randomized algorithms, we need not prove an asymptotic run time. Such algorithms are also often simper, more elegant, and practical.

### 11.1. Min cut.

Given an undirected graph $G = (V, E)$, our goal is to find a cut $(A, B)$ with a minimum number of crossing edges.

This problem has practical uses. For example, graph partition, networks, and image segmentation.

**Example 11.1.**

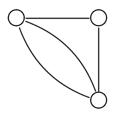


has min cut with the bottom left vertex separated from everything else.

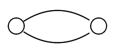
**Algorithm 11.2** (Contraction Algorithm (Karger))**.**

- while $> 2$ nodes remain
  - pick a remaining edge $(u, v)$ uniformly at random
  - merge $u$ and $v$ into a single node
  - remove any self-loops
- return cut represented by final 2 nodes.

**Example 11.3.** In the graph in the example above, suppose we first picked the downward edge on the left. Then we get the graph



Picking the bottom most edge now yields the graph



corresponding to the cut separating the top right node from the rest of the graph.

Note that since this is a randomized algorithm, the answer doesn't have to come out to the same thing. If in the second step we had chosen to use the downward edge, we would have split the graph down the middle, which is incorrect.

**Question.** We want to know the probability of success of this algorithm.

11.2. **Review of probability.**

**Definition 11.4.** The *sample space* $\Omega$ is the set of all possible algorithm. In Karger, $\Omega$ contains all possible random choices for edge contractions.

**Definition 11.5.** Also, each outcome $i \in \Omega$ has *probability mass* $p(i) \geq 0$. We have the constraint that

$$\sum_{i \in \Omega} p(i) = 1.$$

**Definition 11.6.** An *event* is a subset $S \subseteq \Omega$. Define

$$\Pr[S] = \sum_{i \in S} p(i).$$

**Example 11.7.** Suppose that $G = (V, E)$ has $n$ vertices, $m$ edges, and min cut $(A, B)$ has $k$ crossing edges. We know that if the min cut algorithm picks one of these crossing edges, then we would be guaranteed not to get the min cut $(A, B)$. Conversely, if we never contract any of these edges, we do get precisely the desired cut $(A, B)$.

Let $S_i$ be the event that one of these $k$ crossing edges gets contracted in the $i$th iteration. Note that $\Pr[S_1] = k/m$.

We claim that in $G$, every node has degree $\deg(v) \geq k$; otherwise, we could simply cut that node out of the graph. Therefore, the number of edges is at least $m \geq kn/2$ because each one of $n$ nodes has at least $k$ edges and each edge can be seen from exactly two nodes. Therefore, $\Pr[S_1] \leq \frac{2}{n}$.

**Definition 11.8.** Let $S$ and $T$ be two events. Then the conditional probability is

$$\Pr[S \mid T] = \frac{\Pr[S \cap T]}{\Pr[T]}.$$

**Example 11.9.** We now have at most $n - 1$ edges, but possibly less (due to discarding self-loops). Then

$$\Pr[S_2 \mid \text{not } S_1] = \frac{k}{\text{number of remaining edges}}.$$

Note that all nodes, including "supernodes" from contracted edges, induce cuts in $G$. So, in every iteration of the algorithm, we maintain the invariant that every node of the contracted graph has degree $\deg(v) \geq k$. Therefore, in the second iteration, the number of remaining edges is $\geq k(n-1)/2$. Hence

$$\Pr[S_2 \mid \neg S_1] \leq \frac{2}{n-1}.$$

Therefore,

$$\Pr[\neg S_1 \cap \neg S_2] = \Pr[\neg S_1] \Pr[\neg S_2 \mid \neg S_1] \geq \left(1 - \frac{2}{n}\right)\left(1 - \frac{2}{n-1}\right).$$

We can of course iterate this process to get the probability of success, or the probability of always missing the $k$ crossing edges. This is

$$\begin{aligned}\Pr[\neg S_1 \cap \neg S_1 \cap \cdots \cap \neg S_{n-2}] &\geq \left(1 - \frac{2}{n}\right)\left(1 - \frac{2}{n-1}\right)\left(1 - \frac{2}{n-2}\right)\cdots\left(1 - \frac{2}{4}\right)\left(1 - \frac{2}{3}\right) \\ &= \frac{n-2}{n}\cdot\frac{n-3}{n-1}\cdot\frac{n-4}{n-2}\cdots\frac{2}{4}\cdot\frac{1}{3} = \frac{2}{n(n-1)} \geq \frac{1}{n^2}.\end{aligned}$$

We have just lower bounded the success probability of this algorithm. In this case, we end up with the min cut $(A, B)$! So we're happy, right? There's a minor problem: $\frac{1}{n^2}$ is rather small. Happily, this isn't too hard to fix.

## 11.3. **Improving Karger's algorithm.**

**Definition 11.10.** Events $S, T \subseteq \Omega$ are *independent* if and only if

$$\Pr[S \cap T] = \Pr[S]\Pr[T].$$

Equivalently, this means that $\Pr[S \mid T] = \Pr[S]$ or $\Pr[T \mid S] = \Pr[T]$.

Be very careful about assuming that things are independent. Here, we do something that is by definition independent. We run Karger's algorithm repeatedly on the same graph.

Suppose we run Karger's algorithm $N$ different times. Let $T_i$ be the event that $(A, B)$ is found on the $i$th try. By definition, the different $T_i$'s are independent. So the probability that we are hoping won't happen is

$$\Pr[\text{all } N \text{ trials fail}] = \prod_{i=1}^{N}(1 - \Pr[T_i]) \leq \left(1 - \frac{1}{n^2}\right)^N.$$

We want to say something about the rate at which this goes to zero.

**Proposition 11.11.** *For all real numbers $x$, we have $1 + x \leq e^x$.*

*Proof.* To be rigorous, use the Taylor series expansion of $e^x$. Otherwise, just draw a picture. Alternatively, note that $1 + x$ is a tangent line to $e^x$ at $x = 0$ and $e^x$ is a convex function. □

So, if we take $N = n^2$, then the probability

$$\Pr[\text{all } N \text{ trials fail}] \leq \left(e^{-1/n^2}\right)^{n^2} = \frac{1}{e}.$$

If we take $N = 2n^2 \ln n$, then

$$\Pr[\text{all } N \text{ trials fail}] \leq \left(\frac{1}{e}\right)^{2\ln n} = \frac{1}{n^2},$$

which is really tiny. By running many trials, we have improved a small success probability to a small failure probability.

Now, let's consider the running time of this algorithm. This is polynomial in $n$ and $m$. With a lot of clever tricks, this can be implemented to run in near-linear time.

We do not give a proof of correctness because such a proof does not exist. We can only estimate the probability of correctness, as we did above. Even if we missed the minimum, we're still going to get the next-best cut. How to tune the failure probability is a domain-dependent question. There exists an always-correct algorithm that runs in polynomial time.

## 12. 2/10: QuickSort

12.1. **The QuickSort algorithm.** Recall that we had a linear time selection algorithm 4.2. Therefore, we used a ChoosePivot function to partition the array and recurse on each half. QuickSort builds on these idea of selecting pivots and recursing:

**Algorithm 12.1.**

- if $n > 1$, return
- $p = \text{ChoosePivot}(A, n)$
- Partition around $p$.
- Recurse on the first part.
- Recurse on the second part.

Note that assuming that all of the subroutines run in place, the algorithm does too.

**Proposition 12.2.** *The QuickSort algorithm 12.1 runs correctly.*

*Proof.* Induction on $n$. □

The running time of this algorithm depends on the quality of the pivot. In the worst case, the algorithm would have time $\Omega(n^2)$ because it chose bad pivots. With ideal pivots, such as finding the running time by computing the linear-time median with Algorithm 4.2, the running time would satisfy the recurrence $T(n) \leq 2T(n/2) + O(n)$, which yields $O(n \log n)$.

This is pretty good, but not what's actually done in practice due to large constant factors. We can do better. The hope is that a random pivot is "pretty good" "often enough." It's really not obvious that this should work. The benefit is that choosing a pivot is constant time, but our pivots clearly won't be optimal.

**Theorem 12.3.** *For every input array (of size n), the average running time of QuickSort is $O(n \log n)$.*

Notice that the theorem holds for arbitrary input, so we are still doing a worst-case analysis. We average over the internal random choices of the algorithm and not over the possible inputs. We are *not* doing "average-case analysis," in which we would assume that the data itself is random.

12.2. **Probability review.**

**Definition 12.4.** A *random variable* $X$ is a real-valued function $X : \Omega \to \mathbb{R}$.

**Example 12.5.** Consider an input array $A$. Let $z_i$ be the $i$th smallest element of $A$. Let $\Omega$ be the sample space of all possible random choices QuickSort could make (i.e., pivot sequences). Let $X_{ij}(\sigma)$ be the number of times $z_i$, $z_j$ get compared in QuickSort with the pivot sequence $\sigma \in \Omega$.

Note that for all $\sigma$, $X_{ij}(\sigma)$ is either 0 or 1. This is because $z_i$ and $z_j$ are compared only when one of them is the pivot, which is then excluded from all recursive calls. This type of random variable is called an *indicator random variable.*

**Definition 12.6.** Let $X : \Omega \to \mathbb{R}$ be a random variable. Then the *expectation* $E[X]$ of $X$ is the average value of $X$, i.e.

$$E[X] = \sum_{\sigma \in \Omega} p(\sigma) X(\sigma).$$

**Example 12.7.** For $X_{ij}$ above, we have

$$E[X_{ij}] = 0 \cdot \Pr[X_{ij} = 0] + 1 \cdot \Pr[X_{ij} = 1] = \Pr[x_i \text{ and } x_j \text{ get compared}].$$

**Proposition 12.8** (Linearly of Expectation)**.** *Let $X_1$, $X_2$, …, $X_n$ be random variables on $\Omega$,* not *necessarily independent. Then*

$$E\left[\sum_{i=1}^{n} X_i\right] = \sum_{i=1}^{n} E[X_i].$$

*Proof.* Expand both sides and reverse the order of summation. □

We return to QuickSort and apply the above ideas. Let $C(\sigma)$ be the number of comparisons make by QuickSort given random choices $\sigma$. By definition, we have

$$C(\sigma) = \sum_{i=1}^{n} \sum_{j=i+1}^{n} X_{ij}(\sigma).$$

By linearity of expectation,

$$E[C(\sigma)] = \sum_{i=1}^{n} \sum_{j=i+1}^{n} E[X_{ij}(\sigma)] = \sum_{i=1}^{n} \sum_{j=i+1}^{n} \Pr[z_i \text{ and } z_j \text{ get compared}]. \tag{4}$$

**Lemma 12.9.** *The running time of QuickSort is dominated by the number of comparisons, i.e. by $C$.*

*Proof.* Just think about it. For full rigor, see the handout. □

*Proof of Theorem 12.3.* To prove the running time of the QuickSort algorithm, we need to show that (4) is $O(n \log n)$. Consider $z_i$ and $z_j$ for some fixed $i < j$. Take the set of elements $z_i, z_{i+1}, \dots, z_{j-1}, z_j$, and consider the first of them to be chosen as a pivot.

If $z_i$ or $z_j$ are the first to be chosen as a pivot, then $z_i$ and $z_j$ get compared. If one of $z_{i+1}, \dots, z_{j-1}$ are first chosen as a pivot, then $z_i$ and $z_j$ are split into different recursive calls and are *never compared.* Since pivots are chosen uniformly at random, this means that

$$\Pr[z_i \text{ and } z_j \text{ are compared}] = \frac{2}{j-i+1}.$$

There's still a calculation to be done. Plugging into equation (4), we have

$$E[C] = 2\sum_{i=1}^{n} \sum_{j=i+1}^{n} \frac{1}{j-i+1}.$$

For each fixed $i$, the inner sum is $\leq \frac{1}{2} + \frac{1}{3} + \frac{1}{4} + \cdots$, then

$$E[C] \leq 2 \cdot n \cdot \sum_{k=2}^{n} \frac{1}{k}.$$

We need a calculus fact:

**Lemma 12.10.**

$$\sum_{k=2}^{n} \frac{1}{k} \leq \ln n.$$

*Proof of Lemma.* Consider a lower Riemann sum of the function $f(x) = \frac{1}{x}$, and draw a picture! This gives

$$\sum_{k=2}^{n} \frac{1}{k} \leq \int_1^n \frac{dx}{x} = \ln x|_1^n = \ln n. \qquad \square$$

Therefore,

$$E[C] \leq 2 \cdot n \cdot \ln n,$$

completing the proof of the running time of QuickSort. $\square$

12.3. **Sorting lower bounds.** Why can't we have a linear $O(n)$ time algorithm for sorting?

**Theorem 12.11.** *Every "comparison-based" sorting algorithm has worst case running time $\Omega(n \log n)$.*

**Example 12.12.** Examples of comparison-based sorting algorithms are MergeSort, HeapSort, and QuickSort.

**Example 12.13.** Non-comparison based sorting algorithms include RadixSort, BucketSort, and CountingSort. These are very useful, but you have to make assumptions about the input data. When these assumptions are made, it is possible to bypass the $\Omega(n \log n)$ lower bound.

*Proof of Theorem 12.11.* Suppose that our algorithm always uses $\leq k$ comparisons to sort arrays of length $n$. The only thing that the algorithm knows about relative order of input elements is from comparison outcomes. Using $\leq k$ comparisons implies that there are $\leq 2^k$ possible distinct results. This means that it can only differentiate between $\leq 2^k$ distinct relative orderings. Since we need to distinguish $n!$ possible relative orderings, this means that we need to have $2^k \geq n! \geq (\frac{n}{2})^{n/2}$. Therefore,

$$k \geq \frac{n}{2} \log_2 \frac{n}{2} = \Omega(n \log n). \qquad \square$$

## 13. 2/15: Hashing

Today we will talk about hash tables, which are an important use of randomization in data structures.

**Definition 13.1.** A *hash table* supports fast (in expectation) operations of

- Insert
- Delete
- Lookup.

They are sometimes called a "dictionary."

A classic application of hash tables is the symbol table of a compiler.

For a hash table, we have a universe $U$ of everything that might possibly be represented. (For example, $U$ might be all $2^{32}$ IP addresses.) We want to maintain an evolving set $S \subseteq U$. (For example, $S$ might contain IP addresses of around 200 clients.)

13.0.1. *Naive solutions.* Here are some naive solutions to this problem. We might have bit vectors with $A[x] = 1$ if and only if $x \in S$. The problem is that these arrays are massive with large universes. We need $O(1)$ operations but $O(|U|)$ space.

Another method is to use a linked list. Then this requires $O(|S|)$ space but $O(|S|)$ lookup.

13.1. **Hash tables.** Our hash tables will require $O(|S|)$ space and $O(1)$ time for all operations (in expectation). Here's the basic idea.

Pick $n$ buckets, where $n \approx |S|$. This might change over time, so in practice, we'd keep track of this and rehash if necessary.

**Definition 13.2.** A *hash function* is a function $h : U \to$ bucket, $\{0, 1, \ldots, n-1\}$. For each element of $U$, this function assigns it to a bucket.

We use an array $A$ of length $n$ and store $x \in S$ in $A[h(x)]$. Therefore, the function $h$ maps the large $U$ onto a much smaller array $A$. This is too good to be true, and indeed it is: We need to worry about collisions, where two elements of $S$ are sent to the same bucket. This means that $h(x) = h(y)$ for $x \neq y$.

To resolve collisions we can use the method called *chaining*, where we keep a linked list in each bucket. Given $x$, we simply Insert, Delete, or Lookup for $x$ in the list $A[h(x)]$; if there are multiple elements in each bucket, we simply resort to the naive solution. There are other ways of solving this problem, such as the method known as "open addressing."

13.2. **Running time.** Since all operations are operations on one of the linked lists, the running time is determined by the length of the linked list in the relevant bucket. Therefore, our running times is $O(\text{list length})$ for all operations. How well our algorithm does is therefore dependent on how well $h$ spreads things out between the different buckets. If we had a bad hash function that sent everything to the first bucket, our hash table would be really slow.

**Remark.** It is not trivial to come up with good hash functions. Naive hash functions work poorly in both theory and practice.

**Example 13.3.** In the IP address problem, a bad idea would to take the most significant or least significant 8 bits. We would have a lot of clumps, and we would end up with very unbalanced buckets.

**Example 13.4.** In 2003, there was a paper by Crosby and Wallach. They used an "algorithmic complexity" attack on a naive hash function in a network intrusion detection system.

What would be a good hash function? We would love to use some clever hash function that would be guaranteed to spread every data set out evenly. Unfortunately, this is not possible. For any hash function, there is a pathological data set for it, in which everything devolves to the single linked list solution. Why is this true? Fix (arbitrarily clever) any hash function $h : U \to \{0, 1, \ldots, n-1\}$, assuming $|U| \ll n$. Imagine if we hashed every object of the universe and stuck them into the buckets. By the pigeonhole principle, there is some bucket $i$ such that at least $|U|/n$ elements of $U$ hash to $i$. Our pathological data set is therefore drawn from these $|U|/n$ elements given by $h^{-1}(i)$.

The solution is to design a family $\mathcal{H}$ of hash functions such that, for all data sets $S$, "almost all" $h \in \mathcal{H}$ spreads $S$ out evenly.

**Definition 13.5.** Let $\mathcal{H}$ be a set of hash functions from $U$ to $\{0, 1, 2, \ldots, n-1\}$. $\mathcal{H}$ is *universal* if for all distinct $x, y \in U$, if $h \in \mathcal{H}$ is chosen uniformly at random, the probability of a collision is $\Pr_{h \in \mathcal{H}}[h(x) = h(y)] \leq \frac{1}{n}$.

**Theorem 13.6.** *If $\mathcal{H}$ is universal, then all operations will run in $O(1)$ time (in expectation). Here, we assume that $|S| = O(n)$.*

The proof is another example of the type of argument we saw for QuickSort, using the decomposition principle and linearity of expectation.

*Proof.* Let $S$ be the data set. We want to determine if $x \in S$; this means that we want to look up $x$. The running time is $O(\text{list length in } A[h(x)])$. Let $L = \text{list length in } A[h(x)]$. This is the crucial random variable for the running time; we need to show that the expected length of this list is constant.

For $y \in S$, $y \neq x$, define the indicator random variables

$$Z_y = \begin{cases} 1 & h(x) = h(y) \\ 0 & \text{otherwise.} \end{cases}$$

Observe that

$$l \leq 1 + \sum_{\substack{y \in S \\ y \neq x}} Z_y.$$

So

$$E[l] \leq 1 + \sum_{\substack{y \in S \\ y \neq x}} E[Z_y] = 1 + \sum_{\substack{y \in S \\ y \neq x}} \Pr[h(x) = h(y)] \leq 1 + \sum_{\substack{y \in S \\ y \neq x}} \frac{1}{n} \leq 1 + \frac{|S|}{n} = O(1)$$

under the assumption $|S| = O(n)$. □

Do universal hash functions exist? It turns out that there are many families of universal hash functions. Let's consider an example and a proof of concept that such universal hash functions actually exist.

**Example 13.7.** Suppose we just want a hash function for IP addresses. View each IP address as a 4-tuple $(x_1, x_2, x_3, x_4)$, where $x_i \in \{0, 1, 2, \ldots, 255\}$. Let $n = 257$ be a prime number. Suppose we choose $a = (a_1, a_2, a_3, a_4)$ with each $a_i \in \{0, 1, \ldots, n-1\}$. Define a hash function $h_a : \text{IP addresses} \to \text{buckets}$ by

$$h_a(x) = (a_1 x_1 + a_2 x_2 + a_3 x_3 + a_4 x_4) \bmod n.$$

Let

$$\mathcal{H} = \{h_a \mid a_1, a_2, a_3, a_4 \in \{0, 1, 2, \ldots, n-1\}\}.$$

This is easy to evaluate and it is easy to choose $a$ uniformly at random.

**Theorem 13.8.** *The family $\mathcal{H}$ of Example 13.7 is universal.*

*Proof.* Let $x, y$ be distinct. Assume that $x_4 \neq y_4$. Condition on choices of $a_1, a_2, a_3$; here $a_4$ is still random. Then suppose we get the undesirable case $h_a(x) = h_a(y)$; this is true if and only if

$$a_4(x_4 - y_4) \bmod n = [a_1(y_1 - x_1) + a_2(y_2 - x_2) + a_3(y_3 - x_3)] \bmod n.$$

Here, the right hand side is some fixed (nonrandom) number in $\{0, 1, \ldots, n-1\}$. Since $x_4 - y_4 \neq 0$, $n$ is prime, and $a_4$ is chosen uniformly at random. This means that $a_4(x_4 - y_4)$ is also a uniform random variable in $\{0, 1, \ldots, n-1\}$, so the probability of a collision is

$$\Pr[h_a(x) = h_a(y)] = \frac{1}{n}.$$ □

13.3. **Bloom filters.** There have been a lot of nice developments in hashing over the past ten years. Many ideas involve multiple hash functions.

The idea of Bloom filters is that it is more space efficient, allowing us to save significant constant factors. There are no deletions, and there are also sometimes false positives, where the data structure will think that an element exists when it doesn't. In some applications, this is tolerable; in other applications, this is not.

One example where this is acceptable and where the space savings is desirable is the list of forbidden passwords in a network router, for example.

Here is the data structure. We have an array of $m$ bits, for $m \geq |S|$, all initially zero. We have $k$ hash functions. (In practice, we have 8 bits per object, and $1 - 2\%$ false positives.)

Here is Insert($x$):

- for $i = 1, 2, \ldots, k$
  - set $A[h_i(x)] = 1$.

Then Lookup($x$) returns true if and only if $A[h_i(x)] = 1$ for all $i = 1, \ldots, k$. Observe that there can be no false negatives, but there can be false positives coming from the interaction between multiple insertions.

## 14. 2/17: More randomization

Today is the final lecture on randomization and data structures.

14.1. **Binary search tree.** Each node has a key $x$. Each node has two subtrees; the left subtree contains values $< x$ and the right subtree contains values $> x$. Note that there are many possible trees of a given set of keys. In addition, the height of a binary search tree could be anywhere from $\approx \log_2 n$ to $\approx n$.

Binary search trees support a number of operations:

- Insert / Delete / Search
- Min / Max / Predecessor / Successor

The running time of these operations is $\Theta$(height).

**Example 14.1.** For the Search operation, start at the root, and walk down the tree, going left or right as appropriate.

**Example 14.2.** For the Delete operation, we first want to search for $x$. If $x$ is a leaf, delete it. If $x$ has 1 child, splice it out by replacing it by its child. The general case is when $x$ has 2 children. Here, swap $x$ with its successor (which is the minimum of the right subtree); this does not violate the search tree property. Now, delete or splice out $x$ from its new position.

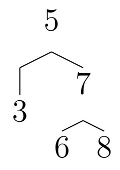
14.2. **Balanced Search Trees.** In binary search trees, the height could be anything between $\log_2 n$ and $n$, and the running time is determined by the height. The idea is to ensure that height is always $O(\log n)$.

14.2.1. *Red-black trees.* We maintain 4 additional invariants.

(1) Each node is labeled either red or black.
(2) The root is always black.
(3) No two reds in a row. This means that red nodes can only have black children.
(4) Every root-null path (unsuccessful traversal of the tree) has exactly the same number of black nodes.

**Example 14.3.** A chain of length 3 cannot be a red-black tree.

**Example 14.4.**



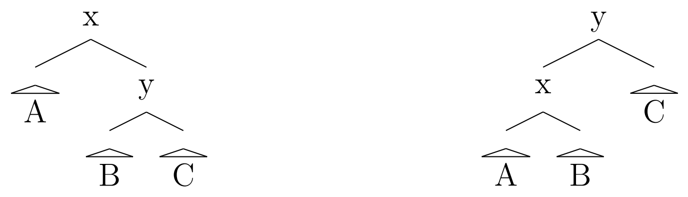
**Proposition 14.5.** *A red-black tree with $n$ nodes has height $\leq 2\log_2(n+1)$.*

*Proof.* If every root-null path has $\geq k$ nodes. Then the tree includes a full binary search tree of depth $k$. Therefore, $n \geq 2^k - 1$. So the minimum number of nodes on a root-null path is $\leq \log_2(n+1)$. Every, every root-null path has $\leq \log_2(n+1)$ black nodes. Hence, every root-null path has $\leq 2\log_2(n+1)$ total nodes. □

**Theorem 14.6.** *We can implement Insert/Delete such that the invariants are maintained, and they take $O(\log n)$ time each.*

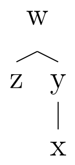
*Proof.* The proof is complicated and we won't cover this in lecture. See CLRS chapter 13.

The key primitive is the *rotation*. This is how we always keep search trees balanced. This preserves the search tree property, which takes $O(1)$ work. The left rotation is the reverse of the right rotation.



The idea for the Insert/Delete operation is to recolor and rotate until invariants are restored. For example, here's what we do in the case of Insert.

- insert $x$ as usual; this makes a leaf.
- try coloring it red.
- if parent $y$ is black, we are done
- otherwise, $y$ is red, so the grandparent $w$ is black.



We have some cases. In Case 1, the grandparent's other child $z$ is absorbed. So we recolor $y$ and $z$ to be black, and $w$ to be red. This either fixes double red or propagates it upward. This can only be done $O(\log n)$ times. If we reach the root, recolor it black.

In case 2, $z$ is black or null. We can fix this in $O(1)$ via 2 or 3 rotations and suitable recoloring. See references for details. □

14.3. **Randomized alternative: Skip lists.** Suppose we are storing $S$. Then level 0 is a sorted linked list of all of $S$. In level 1, we have a sorted list of random 50% of level 0. Level 2 is a random 50% of level 1, and so on.

$$\begin{tikzcd}
\boxed{-\infty} \arrow[rr] \arrow[d] & & \boxed{2} \arrow[d] & & & & \\
\boxed{-\infty} \arrow[rr] \arrow[d] & & \boxed{2} \arrow[rrr] \arrow[d] & & & \boxed{5} \arrow[d] & \\
\boxed{-\infty} \arrow[rr] \arrow[d] & & \boxed{2} \arrow[r] \arrow[d] & \boxed{3} \arrow[rr] \arrow[d] & & \boxed{5} \arrow[d] & \\
\boxed{-\infty} \arrow[r] & \boxed{1} \arrow[r] & \boxed{2} \arrow[r] & \boxed{3} \arrow[r] & \boxed{4} \arrow[r] & \boxed{5} \arrow[r] & \boxed{6}
\end{tikzcd}$$

The insert operation is

- Set $i = 0$
- while (true)
  - insert $x$ into level $i$ list
  - halt with 50% probability
  - else $i++$.

The search operation is

- search for $x$ in top list
- if unsuccessful, drop down at the predecessor of $x$
- repeat until $x$ is found or unsuccessful at level 0.

The delete operation is

- search for $x$ and delete from all lists.

Here is the intuition for the analysis. This can be made rigorous, but we won't do so here. We expect $\frac{n}{2^i}$ elements at level $i$, and we expect $\approx \log_2 n$ levels. The expected space is $O(n)$. The expected search time is $O(1)$ work per level. The number of records visited corresponds to the number of consecutive predecessors missing from the level above. This is $O(1)$ time on average and $O(\log n)$ time overall. The same is true for the Insert and Delete operations above.

## 15. 2/22: Introduction to Dynamic Programming

Today we introduce the final of our major algorithmic design paradigms: Dynamic Programming. We'll do lots of examples.

### 15.1. **Maximum weight independent set.**

**Example 15.1.** We are given a line graph $G = (V, E)$ with weights $m_v \geq 0$ on the vertices. For example, we might have

$$\begin{tikzcd}
1 \arrow[r] & 4 \arrow[r] & 5 \arrow[r] & 4
\end{tikzcd}$$

Our goal is find a subject of nonadjacent vertices (an *independent set*) of maximum total weight.

Consider some possible approaches.

- Brute-force search. This is exponential time.

- Greedy (iteratively choose max-weight vertex not adjacent to vertices already chosen). This is not correct in general.
- Divide and conquer. If the recursive sub-solutions conflict, it is not clear how to quickly and optimally fix it.

We now present a new approach. We first reason about what the structure of an optimal solution, in terms of optimal sub-solutions. Let $S \subseteq V$ be a max weight independent set of of $G$. Let $v_n$ be the last vertex of the line. This vertex is either in or not in the independent set. We have two cases.

Case 1: $v_n \notin S$. Let $G' = G$ with $v_n$ removed. Note that $S$ is an independent set of $G'$. Additionally, $S$ is maximum weight in $G'$ – if $S^*$ were better, then $S^*$ is also better than $S$ in $G$, which would be a contradiction.

Case 2: $v_n \in S$ Since $s$ is an independent set, $v_{n-1} \notin S$. Let $G'' = G$ with $v_{n-1}, v_n$ deleted. Note that $S - \{v_n\}$ is an independent set of $G''$, and note additionally that this must have maximum weight in $G''$ – if $S^*$ were better than $S - \{v_n\}$ in $G''$, then $S^* \cup \{v_n\}$ would be better than $S$ in $G$, which would be a contradiction.

The point is that if we knew whether or not $v_n$ is in a maximum weight independent set, we could recursively compute the maximum weight independent set of $G'$ or $G''$, and we would be done.

We cannot know a priori which of the two cases we are in, however. The idea is to try both possibilities and return the better solutions. Here's the proposed algorithm:

**Algorithm 15.2.**

- recursively compute $S_1$ and $S_2$ as the maximum weight independent sets of $G'$ and $G''$ respectively.
- return either $S_1$ or $S_2 \cup \{v_n\}$, whichever is better.

Happily, this algorithm is clearly correct, based on what we have said earlier. To prove this formally would involve a divide-and-conquer style inductive argument. Unfortunately, this is exponential time: We have a branching factor of two and a linear recursion depth.

**Question.** How many *distinct* subproblems ever get solved?

Only $n$ distinct subproblems are ever solved: all graph $\{v_1, \dots, v_k\}$ for $1 \le k \le n$. Each subproblem is being solved many times. There is an obvious fix for this: memoization. The first time you solve a subproblem, cache the solution for subsequent $O(1)$-time lookup.

There is an even better iterative way to do this problem. Let $G_i$ be the first $i$-th vertices of $G$. Populate an array $A$ from left to right, where $A[i]$ is the value of the maximum weight independent set of $G_i$.

**Algorithm 15.3.** Base Case: $A[0] = 0$, $A[1] = w_1$. For $i = 2, 3, \dots, n$, set

$$A[i] = \max \begin{cases} A[i-1] \\ A[i-2] + v_i. \end{cases}$$

The running time of this iterative version of the algorithm is obvious $O(n)$, and the correctness proof is verbatim the same as the recursive version.

We now want to present an algorithm to reconstruct the vertices of the maximum weight independent set, given a complete array $A$. We claim that vertex $v_i$ belongs to a maximum weight independent set of $G_i$ if and only if

$w_i$+maximum weight independent set of $G_{i-2} \geq$ maximum weight independent set of $G_{i-1}$.

This is similar to our previous case analysis – check it.

Our reconstruction algorithm is then as follows:

**Algorithm 15.4.**

```
• set $S = \emptyset$ and $i = n$
• while $i \geq 1$
    – if $w_i + A[i-2] \geq A[i-1]$, add $v_i$ to $S$ and decrease $i$ by 2.
    – otherwise, decrease $i$ by 1.
```

What we just did is the quintessential example of dynamic programming. Here are the key ingredients:

- identify a small number of subproblems, e.g. find the maximum independent set in $G_i$ for all $i$.
- after solving all subproblems, we can quickly compute the final solution to the main problem
- can quickly and correctly solve "larger" subproblems given solutions to "smaller subproblems". This is usually via a recursive formula.

15.2. **Knapsack problem.** Imagine a burglar who breaks into a museum and wants to escape with as much value as possible.

We have $n$ items, with values $v_1, \ldots, v_n \geq 0$ and integral sizes $w_1, \ldots, w_n \geq 0$. Our knapsack has integral capacity $W \geq 0$.

The goal is to have a subset $S \subseteq \{1, 2, \ldots, n\}$ that maximizes $\sum_{i \in S} v_i$ subject to the constraint $\sum_{i \in S} w_i \leq W$.

*Step 1.* We formulate a recurrence based on the structure of the optimal solution. Let $S$ be the optimal solution. Note that either $n \in S$ or $n \notin S$. We have two cases:

Case 1: If $n \notin S$, then $S$ is optimal for the first $n-1$ items with the same capacity $W$.

Case 2: Now suppose that $n \in S$. Then $S - \{n\}$ is optimal for the first $n-1$ items, but we also need to work with a reduced capacity $W - w_n$. To see why this is true, note that if $S^*$ is a better solution, then $S^* \cup \{n\}$ is better than $S$ in the original problem, which is a contradiction.

*Step 2.* We now want to identify subproblems.

Subproblems can be given by all prefixes $\{1, 2, \ldots, i\}$ of items, and all possible integral remaining capacities $\{0, 1, 2, \ldots, W\}$.

*Step 3.* We relate the subproblems via a recursive formula. In this case, let $V_{i,x}$ be the value of the best solution using only a subset of the first $i$ items and total weight $\leq x_i$.

We claim that for all $i \in \{1, 2, \ldots, n\}$ and for all $x \in \{0, 1, \ldots, W\}$. Then

$$V_{i,x} = \max \begin{cases} V_{i-1,x} \\ V_{i-1,x-w_i} + v_i. \end{cases}$$

This is just mathematical notation for what we proved in Step 1.

*Step 4.* Use the formula to systematically solve all subproblems.

**Algorithm 15.5.** Let $A$ be a two-dimensional array.

```
For $i = 1, 2, \ldots, n$:
        For $y = 0, 1, 2, \ldots, w$:
```

$$A[i, x] = \max \begin{cases} A[i-1, x] \\ v_i + A[i-1, x - w_i]. \end{cases}$$

15.2.1. *Run time.* The running time of this algorithm is $O(nW)$, or $O(1)$ per array entry. This has a dependence on $W$, which is bad if $W$ is in the billions. There are heuristic approaches to make this more efficient if $W$ is too big.

This linear dependence on $W$ is needed (unless P = NP). This is "pseudopolynomial". In fact, the Knapsack problem is NP-hard.

## 16. 2/24: Sequence alignment and shortest paths

16.1. **Sequence alignment.** Recall from the first lecture 1 that we considered the sequence alignment problem. This was quantified by the NW score to measure the similarity between strings. The problem is purely algorithmic: We need to compute the NW score efficiently.

**Example 16.1.** For example, the strings `AGGGCT` and `AGG_CA` have total penalty $\alpha_{\text{gap}} + \alpha_{\text{AT}}$.

We are given two strings $X = x_1 \ldots x_m$ and $Y = y_1 \ldots y_n$ over an alphabet $\Sigma$. There are penalties

- $\alpha_{\text{gap}} \geq 0$ for inserting a gap, and
- $\alpha_{ab} \geq 0$ for (mis)matching $a$ with $b$, with presumably $a = b$ implying $\alpha_{ab} = 0$.

Feasible solutions are alignments with inserted gaps and equalize the lengths of the strings. The goal is to find the alignment with minimum total penalty. We will do this with dynamic programming. There are many ways to identify subproblems for dynamic programming; we will do this by thinking recursively.

16.1.1. *Optimal substructure.* Consider the optimal alignment for $X$ and $Y$ and the final positions of each string. We have three cases:

(1) $x_m$ and $y_n$ are matched
(2) $x_m$ is matched with a gap
(3) $y_n$ is matched with a gap.

If someone told us which case we were in, we would be done by recursion.

**Proposition 16.2.** *Let $X' = X - x_m$ and $Y' = Y - y_n$. If case 1 holds, then the induced alignment of $X'$ and $Y'$ is optimal. If case 2 holds, then the induced alignment of $X'$ and $Y$ is optimal. If case 3 holds, then the induced alignment of $X$ and $Y'$ is optimal.*

*Proof.* We will prove case 1 here and leave the rest as exercises to the reader.

Suppose that the induced alignment of $X'$ and $Y'$ has penalty $p$, yet some other alignment has penalty $p^* < p$. Appending $x_m$ and $y_n$ to the purportedly better alignment gives an alignment of $X$, $Y$ with penalty $p^* + \alpha_{x_m y_n} < p + \alpha_{x_m y_n}$. This is a contradiction. □

The goal here is to think about all subproblems that we would ever need to solve. This is very similar to the line graph problem, except we have two strings. The relevant subproblems have the form $(X_i, Y_j)$, where $X_i$ are the first $i$ letters of $X$ and $Y_j$ are the first $j$ letters of $Y$.

Let $P_{ij}$ be the penalty of the best alignment of $X_i$ and $Y_j$. Then for all $i, j \geq 1$,

$$P_{ij} = \min \begin{cases} P_{i-1,j-1} + \alpha_{x_i y_j} \\ P_{i-1,j} + \alpha_{\text{gap}} \\ P_{i,j-1} + \alpha_{\text{gap}}. \end{cases}$$

The correctness of this recurrence is immediate from the optimal substructure. There are only three candidates, and we brute-force search over them. The algorithm will simply again be populating a suitable array with this recurrence.

**Algorithm 16.3.** We have a two-dimensional array $A$.

```
In the base case, we have $A[i,0] = A[0,i] = i \cdot \alpha_{\text{gap}}$.
For $i = 1, \dots, m$
        For $j = 1, \dots, n$
```

$$A[i,j] = \min \begin{cases} A[i-1, j-1] + \alpha_{x_i y_j} \\ A[i-1, j] + \alpha_{\text{gap}} \\ A[i, j-1] + \alpha_{\text{gap}}. \end{cases}$$

The correctness of this algorithm is true by induction and the correctness of our recurrence. The running time is $O(mn)$.

To reconstruct the solution:

- Trace back through $A$, starting at $A[m,n]$.
  - If $A[i,j]$ was produced from case 1, match $x_i$ and $y_j$, and go to $A[i-1, j-1]$.
  - If $A[i,j]$ was produced from case 2, match $x_i$ with a gap, and go to $A[i-1, j]$.
  - If $A[i,j]$ was produced from case 3, match $x_i$ with a gap, and go to $A[i, j-1]$.

**Remark.** Why do we use the name dynamic programming? Bellman first created this concept. He worked at RAND, and his superiors hated the word "research". Therefore, he hid the fact that he was doing research. At the time, "programming" was thought of as tabulation and not coding, and also, "dynamic" cannot be used in a pejorative sense. That's why he named it "dynamic programming".

16.2. **Shortest paths.** Recall the single source shortest path problem: Given a directed graph $G = (V, E)$ with edge lengths, compute the lengths of the shortest $s - v$ path for all $v \in V$. Recall that we already solved this problem using Dijkstra's algorithm 5.4. However, Dijkstra works only if all edge costs are nonnegative. Here are some drawbacks of Dijkstra:

(1) Some applications have negative edge lengths. For example, edges might correspond to financial transactions.
(2) Dijkstra is also not very distributed. It needs the entire graph at each iteration, which is not feasible for large graphs in applications such as internet routing.

The solution to these two problems is the Bellman-Ford algorithm. This is in fact how internet routing is done, though with lots of tweaks.

**Question.** How should we define shortest paths when $G$ has a negative cycle?

**Example 16.4.** The problem is that we can have a negative cycle:

$$\begin{tikzcd}
\textcircled{s} \arrow[r] & \circ \arrow[r, "4"] \arrow[d, "-6"'] & \circ \arrow[d, "-5"] \\
& \circ \arrow[r, "3"'] & \circ.
\end{tikzcd}$$

If we just want the shortest path $s - v$ path, with cycles allowed, we will fail because we would keep traversing the negative cycle forever.

Instead, we could look for the shortest $s - v$ path with no cycles allowed. Unfortunately, this problem is NP-hard, which means that there are no polynomial time algorithms unless P=NP.

Here, we will assume that no negative cycles exists. This means that there will be no negative cycles in shortest paths. In fact, it is possible to modify the Bellman-Ford algorithm (without changing the running time) to detect if a negative cycle exists.

16.2.1. *Optimal substructure.*

**Lemma 16.5.** *Let $G = (V, E)$ be a directed graph, with a source $s$ and edges $c_e$. We will assume that there are no negative cycles. For some $v \in V$, let $i \in \{1, 2, \ldots, n-1\}$. Let $P$ be the shortest $s - v$ path with at most $i$ edges (and no cycles). Then we have two cases.*
Case 1: *If $P$ has $\leq i - 1$ edges, it is a shortest $s - v$ path with $\leq i - 1$ edges and no cycles.*
Case 2: *If $P$ has $i$ edges, with the last hop $(w, v)$. Let $P' = P - (w, v)$ be the path $P$ with the final edge $(w, v)$ removed. Then $P'$ is a shortest $s - w$ path with $\leq i - 1$ edges (and no cycles).*

*Proof.* We have to do something nontrivial in the proof because we need to use the assumption that there are no negative cycles.

Case 1 follows by trivial contradiction, analogously to the problem of independent set.

Now, consider case 2. Assume for a contradiction that $Q$ is shorter than $P'$, where $Q$ is a $s - w$ path, has $\leq (i-1)$ edges, and has no cycles. Then $Q + (w, v)$ is an $s - v$ path, has $\leq i$ edges, and is shorter than $P$. We would be done, but what if $Q$ passed through $v$ already? Then $Q + (w, v)$ could have a cycle.

However, we can still get a contradiction. If there is a negative cycle, remove it from $Q + (w, v)$ to get an even shorter $s - v$ path with $\leq i$ edges. So there are no negative cycles! $\square$

## 17. 3/1: Shortest Paths – Bellman-Ford and Floyd-Warshall

17.1. **Bellman-Ford algorithm.** Recall that we are thinking about the single source shortest path algorithm. The Bellman-Ford algorithm is good because it is distributed. We allow negative edge lengths but disallow negative cycles.

We discussed the optimal substructure lemma 16.5 last time. What are the subproblems that we might want to consider? For all $i \in \{0, 1, \ldots, n-1\}$ and for all $v$, we want to compute the length $L_{i,v}$ of shortest $s - v$ path with $\leq i$ edges.

The recurrence for this problem is

$$L_{i,v} = \min \begin{cases} L_{i-1,v} \\ \min_{(w,v) \in E}(L_{i-1,w} + c_{wv}). \end{cases}$$

As usual, this can be proved using the optimal substructure lemma and induction. All that remains is to populate a table.

$A$ is a two dimensional array because there are two parameters: the length and the destination. The base case is that $A[0, s] = 0$ and $A[0, v] = +\infty$ for all $v \neq s$.

**Algorithm 17.1** (Bellman-Ford)**.**

```
for $i = 1, 2, \ldots, n - 1$
        for each $v \in V$:
```

$$A[i, v] = \min \begin{cases} A[i-1, v] \\ \min_{(w,v) \in E} A[i-1, v] + c_{wv}. \end{cases} \tag{5}$$

Now, we should consider the running time of this algorithm. We still have constant lookup time, but it does not take constant time to compute an element of the array because it requires searching over all edges incoming to a vertex $v$. It therefore takes $O(\text{in-degree}(v))$ time to compute each $A[i, v]$. The total running time is therefore

$$O\left(n \cdot \sum_{v \in V} \text{in-degree}(v)\right) = O(nm).$$

## 17.2. **Optimizations.**

### 17.2.1. *Stopping early.*

Suppose that for $i < n - 1$, we have $A[i, v] = A[i - 1, v]$ for all $v$. In this case, the $A[i, v]$ never change again, so we can safely halt.

### 17.2.2. *Negative cycle checking.*

**Proposition 17.2.** *Consider any graph $G$ with arbitrary edge lengths. Suppose there exists $i$ such that $A[i, v] = A[i - 1, v]$ for all $v$. Then $G$ has no negative cycles.*

A consequence is that to check for a negative cycle, we can just run Bellman-Ford for one extra iteration (with $i = n$). This works because $G$ has no negative cycle if and only if $A[n, v] = A[n - 1, v]$ for all $v$.

*Proof.* Define $d(v) = A[i, v] = A[i - 1, v]$. Then our recurrence (5) becomes

$$d(v) = \min \begin{cases} d(v) \\ \min_{(w,v) \in E} d(w) + c_{wv}. \end{cases}$$

Thus, for $(w, v) \in E$, we have that $d(v) \leq d(w) + c_{wv}$. For a cycle $c$, for each edge $e = (w, v)$, we would therefore have $d(v) - d(w) \leq c_{wv}$.

$$\begin{tikzcd} W \arrow[r] & V \arrow[d] \\ X \arrow[u] & Y \arrow[l] \end{tikzcd}$$

Then we have the inequalities

$$\begin{aligned} d(v) - d(w) &\le c_{wv} \\ d(y) - d(v) &\le c_{vy} \\ d(x) - d(y) &\le c_{yx} \\ d(w) - d(x) &\le c_{xw}. \end{aligned}$$

Summing the inequalities then yields that

$$0 \le \sum_{e \in C} c_e,$$

so there all cycles are positive. □

17.2.3. *Space optimization.* Note that we only need the $A[i-1, v]$'s to compute the $A[i, v]$'s. Therefore, we only need $O(n)$ space to remember the last round of subproblems. Happily, this is $O(1)$ space per node! The worry is that throwing away the history of computation will prevent reconstruction of the shortest path. There is a simple fix for this, however.

For each node $v$, we maintain a *predecessor pointer* (initially `NULL`). Whenever $A[i, v]$ gets reset to $A[i-1, w] + c_{wv}$, we reset $v$'s predecessor to $w$.

**Proposition 17.3.** *At termination, tracing pointers back from $v$ yields a shortest $s - v$ path.*

*Proof.* The proof uses induction, as in previous examples. The details are in the book. □

17.3. **All-pairs shortest path problem.** Now, we consider the all-pairs shortest path problem. We are given $G$ as before, allowing negative edges but disallowing negative cycles. The goal is to compute the length $d(u, v)$ of the shortest $u - v$ path for all $u, v \in V$. In particular, there is no fixed source.

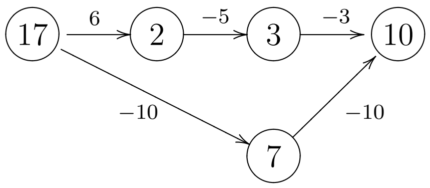
**Remark.** We already know how to do this in $O(n^2 m)$ time by running the Bellman-Ford algorithm once for each choice of an origin and source.

We will solve this using the Floyd-Warshall algorithm. In previous problems, we always use sequentiality. It was unclear how to have an ordering on a graph, so we used the ordering of the output. In the Floyd-Warshall algorithm, we will actually directly impose an ordering on the edges.

Assume that $V = \{1, 2, \ldots, n\}$. Let $V^{(k)} = \{1, 2, \ldots, k\}$.

17.4. **Optimal substructure.** Fix some source $i \in V$ and destination $j \in V$, and $k \in \{1, 2, \ldots, n\}$. Suppose that $P$ is the shortest $i - j$ path with all internal nodes in $V^{(k)}$.

**Example 17.4.** $i = 17$, $j = 10$, and $k = 5$.



Here, the bottom path is ineligible for selection at the step of the algorithm because it requires using the node 7; therefore, the top path would be the path that is selected.

We now have two cases:
Case 1: If $k$ is not internal to $P$, then $P$ is the shortest path with all internal nodes in $V^{(k-1)}$.
Case 2: If $k$ is internal to $P$, then our path $P$ breaks up into two paths: the $i-k$ path $P_1$ and the $k-j$ path $P_2$. Therefore, the internal nodes of $P_1$ and the internal nodes of $P_2$ are in $V^{(k-1)}$. Therefore, $P_1$ is the shortest $i-k$ path with all internal nodes in $V^{(k-1)}$ and $P_2$ is the shortest $k-j$ path with all internal nodes in $V^{(k-1)}$.

*Proof.* The proof of this statement is very similar to Bellman-Ford. □

**Algorithm 17.5** (Floyd-Warshall)**.** Let $A$ be a three-dimensional array. The base case is

$$A[i,j,0] = \begin{cases} 0 & \text{if } i=j \\ c_{ij} & \text{if there exists an edge } (i,j)\in G \\ +\infty & \text{otherwise} \end{cases}$$

We can then fill the array $A$:

```
for $k = 1, \dots, n$
        for $i = 1, \dots, n$
                for $j = 1, \dots, n$
```

$$A[i,j,k] = \min \begin{cases} A[i,j,k-1] \\ A[i,k,k-1] + A[k,j,k-1]. \end{cases}$$

We analyze the running time of this algorithm. This is constant time per subproblem, and there are $n^3$ table entries. The running time is therefore $O(n^3)$.

Compared to $O(mn^2)$ for Bellman-Ford, it is not clear that Floyd-Warshall is much better. For very sparse graphs, the performances of these two algorithms is comparable. For denser graphs, Floyd-Warshall is much faster.

## 18. 3/3: NP-complete problems and what to do about them

The focus of this course has been to discuss practical algorithms and supporting theory for fundamental problems. The sad fact is that many important problems seem impossible to solve efficiently.

### 18.1. NP-completeness.

First, we need to formalize the meaning of "seem impossible to solve efficiently."

**Definition 18.1.** A problem is *polynomial-time solvable* if there exists a correct algorithm for it running in $O(n^k)$ time for some constant $k$. Here, $n$ is a measure of the input size of the problem, e.g. as input length in keystrokes or in bits.

**Definition 18.2.** Let $P$ be all polynomial-time solvable problems.

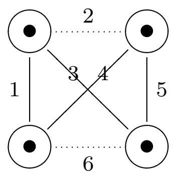
**Example 18.3.** Everything in this course so far (except the knapsack problem 15.2) has been in $P$.

This is a rough litmus test for being "computational tractable."

**Example 18.4** (Traveling Salesman Problem)**.** Our input is a complete graph $G=(V,E)$ with nonnegative edges costs $c_e$ for all $e\in E$. The idea is that there is a traveling salesman who wants to visit every city and get home as soon as possible.

We want to output the minimum cost tour (cycle visiting each vertex once).

**Example 18.5.**



has a minimum cost tour of 13.

**Conjecture 18.6** (Edmonds, 1965)**.** *There is no polynomial-time algorithm for the traveling salesman problem.*

This conjecture predates and is actually equivalent to P≠NP.

A good idea is to amass evidence of intractability via relative difficulty.

**Definition 18.7.** Problem $\Pi_1$ reduces to $\Pi_2$ if given a polynomial-time subroutine for $\Pi_2$, we can solve $\Pi_1$ in polynomial time.

**Example 18.8.** We saw in lecture 4 that the median algorithm reduces to sorting.

Here, we will use the contrapositive as reduction. If $\Pi_1$ is not in $P$, then neither is $\Pi_2$. This says that $\Pi_2$ is as hard as $\Pi_1$.

**Definition 18.9.** Let $\mathcal{C}$ be a set of problems. If $\Pi \in \mathcal{C}$ and everything in $\mathcal{C}$ reduces to $\Pi_1$, then $\Pi$ is $\mathcal{C}$-complete. Think of this as the "hardest problem in $\mathcal{C}$".

We therefore want to show that the traveling salesman problem is $\mathcal{C}$-complete for some really big set $\mathcal{C}$.

What if we were really ambitious, and asked for $\mathcal{C}$ being the set of all problems? This doesn't make any sense. The halting problem is not solvable, while the traveling salesman problem is solvable by exponential-time brute force search. There are problems strictly harder than the traveling salesman problem.

The refined idea is that the traveling salesman problem is as hard as all brute-force-solvable problems.

**Definition 18.10.** A problem is in NP if

(1) correct solutions has polynomial length
(2) purported solutions can be verified in polynomial time.

**Example 18.11.** Is there a tour with length $\leq 1000$?

**Remark.** Every NP problem is solvable by brute-force search.

This definition is so abstract and general that almost any problem you would ever see is in NP. The vast majority of natural computational problems are in NP. All that we require is that we can recognize a solution if it is given to us.

By definition, a polynomial-time algorithm for a single NP-complete problem actually solves *every* NP problem. This would imply that P=NP. Thus, NP-completeness is strong evidence of intractability.

Are there NP-complete problems at all?

**Fact** (Cook, Karp, Levin, early 1970s)**.** NP-complete problems exist. In fact, there are thousands of them, including the traveling salesman problem.

There is an easy recipe to prove that $\Pi$ is NP-complete.

(1) Find a known NP-complete problem $\Pi'$.
(2) Reduce $\Pi'$ to $\Pi$.

**Question.** Does P$\neq$NP?

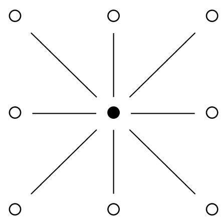
This is one of the Clay Institute's Millennium Problems.

18.2. **Coping with NP-completeness.** There are still productive things that can be done with NP-complete problems. Here are some possible approaches to NP-complete problems.

(1) Focus on a tractable special case (see Section 10.2 of the text, or Homework 7). See also the maximum weight independent set on line graphs 15.1.
(2) Use heuristics to find fast algorithms that are not always correct.
(3) Find an exponential time algorithm that is better than brute-force search. For example, for the Knapsack Problem 15.2, brute force search gives $\Omega(2^n)$ while dynamic programming had running time $O(nW)$.

18.2.1. *Minimum vertex cover.* We are given an undirected graph $G = (V, E)$. The goal is to produce a minimum size vertex cover $S \subseteq V$ with at least one endpoint of each edge.

**Example 18.12.**



has a minimum vertex cover size of 1. For a complete graph on $n$ vertices, the minimum vertex cover size is $n - 1$.

**Fact.** The minimum vertex cover problem is NP-complete.

Suppose that we instead want to determine if there exists a vertex cover with size $\leq k$, with $k$ small. We could try all possibilities, and this would take running time $\Omega(n^k)$. We would like something better.

Here's a smarter method. Note that for any edge $e = (u, v)$, a vertex cover contains either $u$ or $v$ (or both). Define $G'$ to be what remains if we delete vertex $u$ and all of its incident edges. Analogously, define $G''$ to be what remains if we delete vertex $v$ and all of its incident edges.

**Lemma 18.13.** *$G$ has a vertex cover of size $k$ if and only if $G'$ or $G''$ has a vertex cover of size $k - 1$.*

*Proof.* Exercise. □

**Algorithm 18.14.**

```
(1) Pick any edge $e = (u, v)$.
(2) Recursively search for a vertex cover $S$ of size $k - 1$ in $G'$. If found, return $S \cup \{u\}$.
(3) Recursively search for a vertex cover $S$ of size $k - 1$ in $G''$. If found, return $S \cup \{v\}$.
(4) If we haven't found a vertex cover, such a vertex cover does not exist.
```

Consider the running time of this algorithm. The branching factor is two, and the recursive depth in $k$, so there are $2^k$ function calls. There is also some overhead, which we will sloppily call $O(m)$. Then the running time is $O(2^k m)$.

## 19. 3/8: Approximation algorithms

Last time, we talked about what to do about NP-complete problems. Using the example of vertex cover, we looked at fully correct faster exponential algorithms. We also considered solvable special cases. Today, we will discuss using efficient heuristics. We will relax the requirement of perfect correctness. We will need to have some measure of how correct our algorithms are. Ideally, our approximation algorithms should provide a performance guarantee.

**Knapsack problem.** Recall the Knapsack Problem 15.2. We have $n$ items, with weights $w_1, \dots, w_n \leq W$ and total capacity $W$. The items have integral values $v_1, \dots, v_n$[1]. Our goal is to compute $S \subseteq \{1, 2, \dots, n\}$ maximizing $\sum_{i \in S} v_i$ such that $\sum_{i \in S} w_i \leq W$.

19.1. **Greedy algorithm.** The motivation is that ideal items have big value and small weight.

**Algorithm 19.1.**
*Step 1:* Sort and reindex the items by the ratio of value to weight, so that

$$\frac{v_1}{w_1} \geq \frac{v_2}{w_2} \geq \cdots \geq \frac{v_n}{w_n}.$$

*Step 2:* Then pack items in this order until one doesn't fit, and then halt.

**Example 19.2.** Suppose $W = 5$ and

$$\begin{aligned} v_1 &= 5 \quad & w_1 &= 1 \\ v_2 &= 4 \quad & w_2 &= 2 \\ v_3 &= 3 \quad & w_3 &= 3 \end{aligned}$$

The greedy algorithm picks $\{1, 2\}$, which happily is optimal.

**Example 19.3.** Now, consider the example of $W = 1000$, with values and weights

$$\begin{aligned} v_1 &= 2 \quad & w_1 &= 1 \\ v_2 &= 1000 \quad & w_2 &= 1000. \end{aligned}$$

The algorithm picks the first item, with value 2. The optimal solution is to choose the second item, with value 1000. So this algorithm might be arbitrarily bad.

There is a simple fix though.
*Step 3:* In the algorithm above, return either the previous solution or the maximum value item, whichever is better. This guarantees that we achieve at least 50% of the optimal solution.

**Theorem 19.4.** *The value of the greedy solution with the preceding three steps will always be $\geq 50\%$ of the value of the optimal solution. This runs in $O(n \log n)$ time, and we call this a $\frac{1}{2}$-approximation algorithm.*

[1]This is slightly different than our previous description of this problem

*Proof.* Suppose that in step 2, the greedy algorithm picks the first $k$ problems. What if we were allowed to fully fill the knapsack using a suitable "fraction" of item $k+1$?

**Exercise 19.5.** This would be as good as *every* knapsack solution. This is a typical greedy proof, as in 7.8.

Our greedy solution has

$$\begin{aligned}\text{value} &\geq \text{first } k \text{ items}\\ \text{value} &\geq \text{item } k+1,\end{aligned}$$

so therefore

$$2 \times \text{value} \geq \text{greedy solution with taking fractional items} \geq \text{best knapsack solution.} \quad \square$$

19.2. **Dynamic programming.** The goal is that given any user-specified $\varepsilon > 0$ (e.g. $\varepsilon = 0.01$), we want to get a $\geq (1-\varepsilon)$-approximation algorithm. The running time will increase as $\varepsilon \to 0$.

The idea is to exactly solve a slightly incorrect version of the problem.

**Lemma 19.6.** *We presented a dynamic programming algorithm for Knapsack (with integral weights) in 15.5 earlier. Here, we have another Knapsack dynamic programing algorithm (with integral values) with running time $O(n^2 v_{\max})$, where $v_{\max} = \max_i v_i$.*

*Proof.* Let $A[i,x]$ be the minimum weight needed to achieve value $\geq x$ using the first $i$ items. This is now a standard dynamic programming solution. $\square$

We now present a rounding algorithm to get integer values from a general Knapsack problem.

**Algorithm 19.7.**

(1) Round each $v_i$ down to the nearest multiple of $m$. Divide by $m$ to get $\hat{v}_i$. Here, we are defining
$$\hat{v}_i = \left\lfloor \frac{v_i}{m} \right\rfloor.$$
These $\hat{v}_i$ are integers.

(2) Compute the optimal solution for the $\hat{v}_i$'s using the dynamic programming algorithm from Lemma 19.6.

Note that after rounding,
$$m\hat{v}_i \leq v_i \leq m(\hat{v}_i + 1).$$
We now do accuracy analysis on this algorithm.

19.2.1. *Accuracy analysis.* Let $S^*$ be the optimal solution to the original problem. Let $S$ be our solution. Then
$$\sum_{i \in S} \hat{v}_i \geq \sum_{i \in S^*} \hat{v}_i.$$
Therefore,
$$\sum_{i \in S^*} v_i \leq m \sum_{i \in S^*} (\hat{v}_i + 1) = m|S^*| + m \sum_{i \in S^*} \hat{v}_i \leq m|S^*| + m \sum_{i \in S} \hat{v}_i \leq m|S^*| + \sum_{i \in S} v_i.$$

Therefore,

$$\sum_{i\in S} v_i \geq \sum_{i\in S^*} v_i - m|S^*|.$$

Our goal is to have the above quantity be $\geq (1-\varepsilon)\sum_{i\in S^*} v_i$. So choose $m$ to guarantee that

$$m|S^*| \leq \varepsilon \sum_{i\in S^*} v_i.$$

Being conservative, we have $m|S^*| \leq mn$, and $\sum_{i\in S^*} v_i \geq v_{\max}$. Then setting

$$m = \frac{\varepsilon v_{\max}}{n}$$

yields accuracy $1-\varepsilon$.

19.2.2. *Run time analysis.* For all $i$, we have

$$\hat{v}_i \leq \frac{v_i}{m} \leq \frac{v_{\max}}{m} = \frac{n}{\varepsilon}.$$

On the $\hat{v}_i$'s, the dynamic programming algorithm takes time $O(n^2 \cdot n/\varepsilon) = O(n^3/\varepsilon)$. This is actually very impressive, as we are taking an NP-hard problem and approximating its solution arbitrarily well with a polynomial time algorithm. Most NP-hard problems cannot be approximated in this way.

## 20. 3/10: The wilder world of algorithms

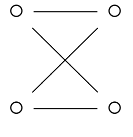
We have covered a lot of algorithmic design techniques in this course, but we have limited time. There are many more techniques that we have missed so far. We will discuss a few of them today. The goal is to give some basic word recognition for those techniques, and to advertise other CS courses at Stanford. We will also see that algorithms still remain an exciting discipline, by seeing a number of open problems.

20.1. **Bipartite matching.** Suppose that we have a graph with sets of nodes $U$ and $V$. Assume that $U$ and $V$ have the same number of nodes[2]. Each edge has one vertex in $U$ and one vertex in $V$.

Our goal is to find a *matching* $M \subseteq E$ such that each node is paired with at most one other.

A matching is *perfect* is every edge is paired, i.e. $|M| = n$.

**Example 20.1.** This is a perfect matching:

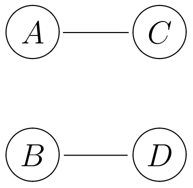


[2]This is almost never important; just add fictitious nodes to the smaller set.

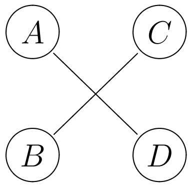
20.1.1. *Stable matching.* Each node has a ranked of nodes on the other side. For example, one side might represent students and the other side might represent schools, where the students are applying to schools.

The goal is to find a perfect matching such that if $u$ and $v$ are not matched, then either $u$ likes $v'$ more than $v$, or $v$ likes $u'$ more than $u$.

**Example 20.2.** Suppose that each student has school preferences $C, D$, and each school has school preferences $A, B$. This is the unique stable matching.



This is an unstable matching because both $A$ and $C$ would be happier if they were matched:



It is not clear even if a stable matching should exist. There is an algorithm that does this.

**Algorithm 20.3** (Gale-Shapley Proposal Algorithm)**.**

```
• while there exists an unpaired man $u$
    – $u$ proposes to top woman $v$ who hasn't rejected him yet
    – each woman entertains only the best proposal received so far
```

**Example 20.4.** In the example 20.2. $B$ proposes to $C$, and there is a tentative acceptance. $A$ proposes to $C$, and $C$ accepts $A$ and rejects $B$. Then $B$ is now open, so $B$ proposes to $D$, who accepts. The algorithm then terminates.

**Theorem 20.5.** *The algorithm always terminates with $O(n^2)$ proposals. It always terminates with a stable matching.*

*Idea of proof.* There are a total of $n^2$ possible proposals, which shows that the first part of the statement.

We always maintain the invariant that each node has only one incident edge, so we always terminate with a matching. Suppose that the matching is not perfect. Then some man was rejected by every woman, which means that all $n$ woman are engaged at the end of the algorithm. Since there are as many men as women, all men are also paired up, formed a contradiction.

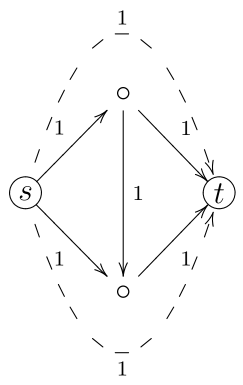
We should also prove that this gives a stable matching. This follows from the same type of argument, and it is left as an exercise. □

20.1.2. *Maximum matching.* We want to maximize then number of edges in a matching. We want to reduce this to a different problem for which we have good algorithms. This problem reduces to a problem called *maximum flow*[3].

[3]See CS 261.

20.2. **Max flow.** We have a directed graph $G = (V, E)$ with a source $s$ and a sink $t$. There are edge capacities $u_e \geq 0$. The goal is to set "flow" that sends the maximum possible amount.

**Example 20.6.**

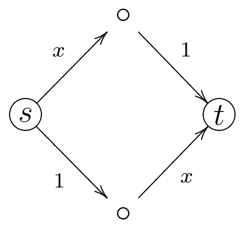


The maximum flow would be to send one unit in the top path and the bottom path.

We now present an algorithm to do this. Pretend that we can go backward across an edge. In the above example, take two zigzag paths that overlap in the middle path. Then the paths cancel over the center edge, which yields the above max flow solution[4].

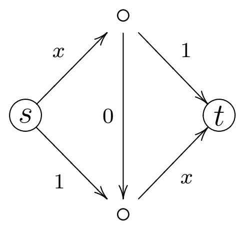
20.2.1. *Selfish flow.* We have a flow network, with a fixed and known number of commuters traveling between two cities. For each edge, there will be a cost function. We want to send one-unit of flow from $s$ and $t$. [5]

**Example 20.7.** Here, the cost $x$ is the fraction of drivers on that route.



Over enough time, we will stable with half of the drivers taking the top route and half of the drivers taking the bottom route. The stable solution is that everyone takes 1.5 hours, with 0.5 hours on the $x$ piece and one hour on the fixed cost piece.

Suppose that a Stanford student builds a teleportation device of cost 0, to get this graph:



[4]See CS 369 (advanced graph algorithms)

[5]See CS 364A (computational game theory).

Now, everyone wants to use the teleportation device, so everyone takes the top $x$ path, the bottom $x$ path, and the teleportation device, for a total time of two hours. The commute time increased with the addition of teleportation! This is *Braess's Paradox*.

**Remark.** There is a physical interpretation of this paradox. Given some complicated physical device with strings holding up a heavy weight, cutting a string might cause the weight to move upward instead of downward. This is because the same equations govern the physical weight and the commuters.

20.3. **Linear programming.** We want to optimize a linear function over the intersection of half spaces. In two dimensions, each half-space is a half-plane. The set of feasible solutions will form a polygon. The general idea is that we want to go as far as possible in a particular direction. This generalizes max flow and tons of other stuff, and is one of the most commonly used algorithms. [6]

Algorithms, especially the simplex method, were invented by Dantzig. Linear programming can be solved efficiently.

20.4. **Computational Geometry.** This deals with low-dimensional geometry, such finding convex hulls. In high dimensional geometry, we are interested in finding the nearest neighbor. There are cool data structure ideas with solutions to these problems. [7]

20.5. **Approximation and randomized algorithms.** A selection of applications include:[8]

- Heuristics for the Traveling Salesman Problem 18.4.
- Semidefinite programming
- Chernoff bounds
- Markov chains

20.6. **Complexity.** An important question is: What can't algorithms do? [9]

- Limits on algorithms
- Completeness for other complexity classes
- Inapproximable
- Unique games conjecture.

20.7. **Conclusion.** It's a joy to teach this class. We see many beautiful ideas. The computer scientists who predated us were really smart, and we had a lot of learn. We should feel smarter after this class.

*E-mail address*: `moorxu@stanford.edu`

[6]See CS 261 and CS 361.

[7]See CS 268 (geometric algorithms).

[8]See CS 261 and CS 365.

[9]See CS 254, CS 359.