
CSC373 Notes

Last updated March 7, 2024

Contents

1 Proofs	5
1.1 How do I Structure Proofs?	6
1.2 Bubble sort Inductive Proof	7
1.3 Bubble Sort Contradiction Proof	8
2 Divide and Conquer	9
2.1 Merge Sort	9
2.1.1 Proving the Mergesort Algorithm	9
2.2 Master Theorem	10
2.3 Counting Inversions	11
2.4 Closest Pair in R ²	12
2.5 Karatsuba's Algorithm	14
2.6 Strassen's Algorithm	15
2.7 Kth Smallest / Quick Select	16
2.7.1 Median of Medians	16
3 Greedy Algorithms	17
3.1 When should I use it?	18
3.2 Interval Scheduling	18
3.2.1 Interval Scheduling Proof of Optimality By Contradiction	20
3.2.2 Proof of Optimality by Induction	20
3.3 What are the Arguments Saying?	22
3.3.1 Another Flavor of Greedy Proofs	22
3.4 Interval Partitioning / Scheduling Lectures In Rooms	23
3.4.1 Runtime	24
3.4.2 Proof of Optimality	24
3.5 Minimizing Maximum Lateness	26
3.6 Lossless Compression	29
3.6.1 Proof of Huffman Encoding Optimality	30

4 Dynamic Programming	32
4.1 Weighted Interval Scheduling	33
4.2 Top-Down vs. Bottom-up Approach	35
4.3 Optimal Value vs. Optimal Solution	36
4.4 Optimal Substructure Property	36
4.5 Knapsack Problem	36
4.5.1 Running Time	37
4.6 Single Source Shortest Paths	37
4.7 Maximum Length Paths	39
4.8 Chain Matrix Product	40
4.8.1 A Small Example	40
4.8.2 Why use DP?	41
4.8.3 Formalizing It + Runtime	42
4.9 Edit Distance	42
4.10 The Traveling Salesman Problem	44
5 Network Flow	46
5.1 What's a Flow?	47
5.2 Why Doesn't Greedy Work?	47
5.3 Residual Graph	48
5.4 Augmenting Paths	49
5.5 Augmenting Always Satisfies Capacity Constraints	50
5.6 Augmenting Always Conserves Flow	51
5.7 Ford-Fulkerson Algorithm	51
5.7.1 Runtime – Number of Augmentations	52
5.7.2 Runtime – Time to perform an augmentation	52
5.8 Can we convert this to polynomial time?	53
5.9 Types of Polynomial Running Times	53
5.10 How to Achieve Polynomial Time	54
5.11 Ford-Fulkerson Correctness	54
5.11.1 Graph Cuts	54

5.12	Edmonds-Karp Algorithm	57
5.12.1	Lemma 1: Proof that Distance never Decreases	58
5.12.2	Lemma 2: Critical	59
5.12.3	Proof of Edmonds-Karp Running Time	60
5.12.4	Notes on Running Time	60
6	Network Flow Applications	60
6.1	Bipartite Matching	61
6.1.1	Correctness	61
6.2	Hall's Marriage Theorem	62
6.3	Edge-Disjoint Paths	65
6.3.1	Using the Max-Flow Min-Cut Theorem	66
6.4	Multiple Sources and Sinks	66
6.5	Circulation	67
6.5.1	Network Flow Formulation: See if there is a circulation	67
6.5.2	Circulation with Lower Bounds	67
6.6	Survey Design	69
6.7	Profit Maximization	70
7	Linear Programming	71
7.1	LP Standard Formulation	75
7.2	LP Transformation Example	75
7.3	Does an LP Always Have an Optimal Solution?	76
7.4	Simplex Algorithm	77
7.5	Resolving Negative Constraints	80
7.6	Special Cases	80
7.6.1	Absolute Value Constraint	80
7.6.2	Absolute value objective function	80
7.6.3	Max Objective Function	81
7.7	Certificate of Optimality	81
7.8	Dual	84
7.8.1	Weak Duality Theorem	84
7.8.2	Strong Duality Theorem	85

7.9	Network Flow via LP	85
7.10	Shortest Paths	86
7.11	Why use LP?	86
7.12	Integer Linear Programming	86
8	Complexity Theory	86
8.1	Church-Turing Thesis	88
8.2	Tape Encoding	88
8.3	Encoding	89
8.4	Efficient Computability	89
8.5	Extended Church-Turing Hypothesis	90
8.6	P	90
8.7	NP	90
8.8	CO-NP	91
8.9	Open Questions	91
8.10	Cook's Conjecture	92
8.11	Reduction	92

1 Proofs

Why do particular algorithms run quickly? Why is it that traveling salesman takes forever, but something like sorting is quick? This course gives you the tools to help you design algorithms that are fast for solving these problems, and to give you what the features are that leads to something being fast or not.

We're trying to figure out whether an algorithm is efficient. There are two different types of efficiency:

- Weakly polynomial (the number)
- Strongly polynomial (no. of bits)

When can we find an example that runs in polynomial time and the structures of algorithms that end up running in polynomial time?

We have:

- Greedy
- Divide and conquer
- Network flow
- Dynamic programming
- Linear programming

These approaches are some of the most common approaches used to find examples for these algorithms that run in polynomial time.

We're also going to show example of problems we strongly suspect that do not run in polynomial time (NP hard or NP complete) – often by reducing them to instances of other NP-hard problems. For example, Super Mario Brothers is NP complete (you can build levels, and the level is beatable IFF the corresponding decision problem is correct). We'll see more fun examples along the way.

One of the things we're going to see is that some variants of problems are extremely easy, but when we change it a bit, it becomes extremely hard. Why?

Linear programming can be done in polynomial time, but if you change it into integer linear programming, all of a sudden, it becomes almost impossible to solve.

Randomized algorithms give us strategies where in case we get an NP problem, we argue how bad a random guess at a solution would be relative to the best case. In some cases, we can prove that the random solution isn't that bad.

This course is theoretical in nature. We'll drill down into the simplest problems.

1.1 How do I Structure Proofs?

Reduce everything down to a well-established proof structure. Putting the logic in a well-structured way makes errors easier to catch and makes it easier to be confident that your ideas are right. Use these structures:

- Induction, all forms

- Contradictions
 - Assume your claim isn't true, come up with an impossible statement
- Not of the two above
 - Beware, this is very error prone

How do I set up an inductive proof? Of the two common proof techniques, inductive proofs are easier to set up.

- Break the problem into a number of steps, $s(i)$
- Show the induction hypothesis holds for the base case $s(0)$, which is extremely easy to show
 - Show an array that contains one item is sorted
- Show that $s(i) \Rightarrow s(i + 1)$

When you're proving something with induction, you should begin by drawing a picture. Figure out the structure of the problem before trying to prove it. When you see problems, you won't know how to solve them right away, but drawing some figures and going through small examples makes the structure of the algorithm clearer. It makes it more obvious to set up the induction hypothesis, but it's rare to see it right away.

Don't try to do the prof without understanding the structure of the problem.

1.2 Bubble sort Inductive Proof

Prove that one iteration of bubble sort puts the lowest element in the right.

How does this end up working? Induction on the array length

BASE CASE: $n = 1$. Our list is already sorted.

INDUCTION STEP:

Assume bubble sort will put the lowest number on the right for an array with $n - 1$ items.

Say we have a list A with n items and run our first iteration of bubblesort on $A[0 \dots n-1]$. We'll end up with $\begin{bmatrix} - & - & x \\ & & y \end{bmatrix}$

There are two cases:

Case 1: $x \leq y$: swap x and y , our last element would be x which is the smallest

Case 2: Do nothing, nothing happens, the right most element is the smallest

As we've covered all possible cases, we know that the induction step holds.

So by induction, our argument here works.

1.3 Bubble Sort Contradiction Proof

Assume that the opposite of the hypothesis was true.

Prove that one iteration of bubble sort puts the lowest element in the right.

Show that if the opposite were true then the assumptions of the problem would be violated.

The opposite is: (One iteration of bubble sort ran AND the lowest element isn't on the right)

Proof goes like this:

- Assume bubble sort fails and position i is first from right with $a[i] \leq a[i+1]$ (where $i = \text{len}(A) - 2$)
 - Here, we **assume a case of failure, that this was the result after the swap**
- If $a[i] \leq a[i+1]$, a swap should've happened but because we said that bubble sort failed, we don't swap. But we assumed that bubble sort must have run

I could really rephrase that as:

- Lowest element on the right \Rightarrow One iteration of bubble sort couldn't have run

2 Divide and Conquer

2.1 Merge Sort

You know how it works. At least. Some preliminaries:

- Merging two sorted lists is $\mathcal{O}(n)$

Task: sort a list of integers [1, 3, 4, 2]

Merge sort works like this:

- Split A into L and R
- Call Mergesort on L and R
- Merge the two back together

I'm taking a problem, breaking them up into two, then I combine them back together.

2.1.1 Proving the Mergesort Algorithm

With induction: $P(n)$ means that Mergesort works for a list L of that size.

Base case: $P(2)$, there is no recursion, and we can very easily deduce that the list would be sorted.

Inductive step: For k that is a power of 2, assume $\forall i \in \{2, 4, \dots, k\}, P(i)$. Prove $P(2k)$.

The list of size $2k$ will be split into two, so we will get two sorted lists coming in. Now, what we need to do is argue that merging two sublists works.

So really, here's our proof for the merge algorithm:

Merge ran \Rightarrow Merged list is sorted

Let's prove the contrapositive:

\neg Merged list is sorted $\Rightarrow \neg$ Merge ran

Now, let's look at the merging algorithm again, and for this one concrete example: suppose that `merge` does this.

$$\begin{aligned} [4, 5][6, 7] &\downarrow \\ [4, 6, 5, 7] = S \end{aligned}$$

`Merge` couldn't have done this. Let's look at the first location on the list that isn't correctly sorted: at index 1, 6.

Because the list isn't sorted, $\exists i, S[i] > S[j], j > i$. If we assume this, we end up going through the list element by element and compare the two results. When we're breaking up the two lists, we sort them and iterate over their positions.

This argument goes: if we actually run `merge`, we would get $[4, 5, 6, 7]$. Moreover, for $[4, 6, 5, 7]$ to be put as the output, $S[j] = 5 > S[i] = 6$, which is impossible according to the `merge` algorithm. This means that the `merge` algorithm couldn't have ran.

2.2 Master Theorem

I've seen it before. It's an amazingly powerful result. What it is, is that it gives you the asymptotic scaling for the solution of a recurrence relation. This is hugely important for divide and conquer algorithms, as they always have a recursive structure to it.

The cost of Mergesort is:

$$T(n) \leq 2 T\left(\frac{n}{2}\right) + \mathcal{O}(n)$$

It goes by this:

Let $a \geq 1, b > 1$ be constants, $T(n)$ be defined on $\mathbb{Z}^{\geq 0}$, and $\frac{n}{b}$ can be $\lceil \frac{n}{b} \rceil$, and let $T(n)$ be:

$$T(n) \leq aT\left(\frac{n}{b}\right) + f(n)$$

Let $d = \log_b(a)$.

1. If $f(n) \in \mathcal{O}(n^{d-\varepsilon})$ for some $\varepsilon > 0$, $T(n) = \Theta(n^d)$
2. If $f(n) \in \Theta(n^d \log^k(n))$ for some $k \geq 0$, $T(n) = \Theta(n^d \log^{k+1} n)$
3. If $f(n) \in \Omega(n^{d+\varepsilon})$ for some $\varepsilon > 0$, $T(n) = \Theta(f(n))$

For case 2: Ideally, you want k as close to 0; 0 gives us the least upper bound. Picking a higher k gives a looser upper bound which we probably don't want.

For Mergesort, we have $a = 2$, $b = 2 \Rightarrow d = \log_2(2) = 1$.

In this case, we know that $f(n) \in \mathcal{O}(n)$. Case 1 can't be true: $f(n) \notin \mathcal{O}(n^{1-\varepsilon})$. Yet, case 2 work.

Case 2: $f(n) \in \Theta(n^1 \log^0 n) = \Theta(n) \Rightarrow T(n) \in \Theta(n^d \log^{k+1} n)$



If you're using a specific master theorem, let the TAs know which one you're using.

2.3 Counting Inversions

Given an array a of length m , count the number of pairs (i, j) such that $i < j$ but $a[i] > a[j]$.

An inversion is a pair of elements opposite in the sorted order: e.g. [1, **3**, **2**, 6, **9**, **8**]. A list that is sorted in the reverse order would have around n^2 inversions.

The brute-force algorithm would take $\mathcal{O}(n^2)$ time.

Let's divide and conquer count inversions on the left, count inversions on the right, and then count the inversions that are between the left and the right.

Sorting kills all information about the inversions, so we want to store the output. That's why this algorithm has two outputs.

Here's how it works, SORT-AND-COUNT which returns (result, sorted list):

- Take a list L
- In the case of a base case, return $(0, L)$
- Break it in half. A, B
- Apply SORT-AND-COUNT, getting $\left(r_A, \text{NUM INVERSIONS}, A\right), (r_B, B) \text{ back}$
- Merge it back using the steps below

How do we count inversions (a, b) such that $a \in A, b \in B$? And how do we not merge sort over and over again?

We start with two halves, and we combine our inversion counting and sorting:

- Scan A and B LTR
- Compare a_i, b_j
- If $a_i \leq b_j$, do your merge step like usual
- Otherwise, if $a_i > b_j$, add 1 to the inversion count and do the merge step like usual

Meaning our runtime formula:

$$\begin{aligned} T(n) &= 2T\left(\frac{n}{2}\right) + \mathcal{O}(n) \\ &\Rightarrow \mathcal{O}(n \log(n)) \end{aligned}$$

If we just relied on Mergesort and done this iteratively, this cost would be $n \log(n)$ so the second bit of the master theorem, we couldn't do $k = 0$ and we would have a squared algorithm.

2.4 Closest Pair in R2

Given n points of the form (x_i, y_i) in the plane, find the closest pair of points. Assume that if I extract all x and y from all points I won't have duplicates.

Brute force: $\Theta(n^2)$. We'll have to look at every possible pair in the data. $\binom{n}{2} \in \mathcal{O}(n^2)$

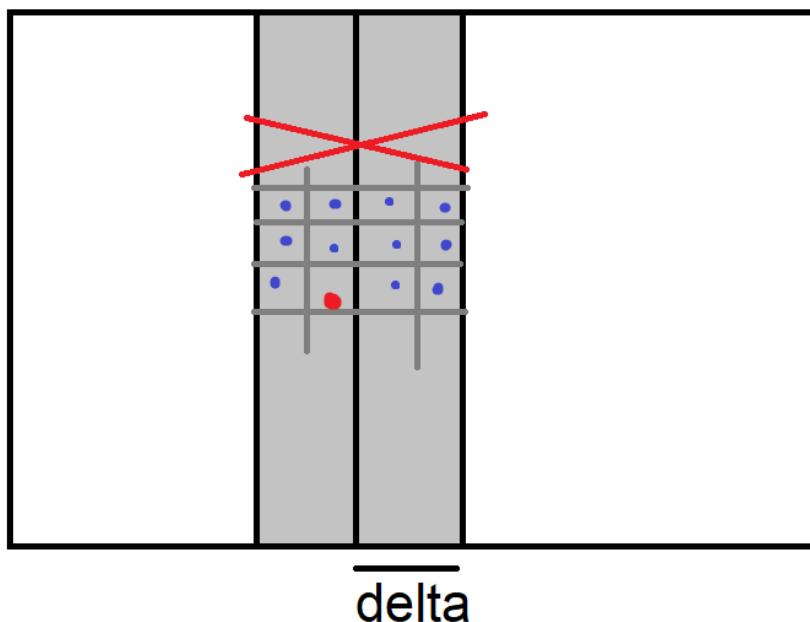
Divide and conquer what we do, is draw a line partway through the data such that half of the data is on the left, and half of the data is on the right. What I'll do, is I'll find the nearest pair on both sides then merge.

However, there's a big complication: merging is very hard.

What we need to do for the combine step, is to go through and check all points in the cross step. However:

Actually, we shouldn't think about doing our comparisons for everything. Reason why, is that:

δ is the lower of the shortest distance between two points on the left, on the right



We can save ourselves a bit of trouble and restrict our attention to our values in the δ region. Yet, we could have a situation where the large fraction of our points could lie inside this region.

Actually, something cool ends up happening. When you look at the sorted list of points, the maximum no. of points you need to compare on the opposite side is a constant. We only need to look at **11** points (I could skip work if the points are on the same side – the POINT of this is that it's a constant no. of points).

Let's break everything up.

Claim: If two points are at least 12 positions apart in the sorted list, their distance is at least δ

Proof: No points lie in the same $\frac{\delta}{2} \times \frac{\delta}{2}$ box

- If two of them were in the same cell, the maximum distance between the two corners is $\frac{\delta}{2}\sqrt{2} = \frac{\delta}{\sqrt{2}} < \delta$
- If two of them happen to show up in the same cell, the distance must be less than δ , but δ is the smallest distance between the two

So, what are the total comparisons that we need to do?

Merging takes $\mathcal{O}(n)$.

So, running time analysis.

- Finding points on strip: $\mathcal{O}(n)$
- Sorting by y coord: $\mathcal{O}(n \log(n))$
- Testing against 11 points: $\mathcal{O}(n)$

Runtime:

$$T(n) \leq 2T\left(\frac{n}{2}\right) + \mathcal{O}(n \log(n))$$

Master theorem tells us that the cost is $\mathcal{O}(n \log^2(n))$

By using divide-and-conquer, we can speed up our algorithm by a lot.

We can improve this to $\mathcal{O}(n \log(n))$ by doing a simple sort by y-coordinates at the start.

2.5 Karatsuba's Algorithm

Divide each integer into two parts:

$$x = x_1 \cdot 10^{\frac{n}{2}} + x_2, y = y_1 \times 10^{\frac{n}{2}} + y_2$$

$$xy = (x_1y_1) 10^n + (x_1y_2 + x_2y_1) 10^{\frac{n}{2}} + (x_2y_2)$$

For $\frac{n}{2}$ -digit multiplications can be replaced by three:

$$x_1y_2 + x_2y_1 = (x_1 + x_2)(y_1 + y_2) - x_1y_1 - x_2y_2$$

We can reuse multiplications that we've already done.

Runtime:

$$T(n) \leq 3T\left(\frac{n}{2}\right) + O(n) \Rightarrow T(n) = \mathcal{O}(n^{\log_2(3)})$$

Most of these multiplication algorithms have great scaling but they have horrible constant factors. Karatsuba is an exception.

2.6 Strassen's Algorithm

Instead of multiplying numbers, let's look at multiplying matrices.

Imagine we have two “block” matrices:

$$C = A \cdot B = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix}$$

Naively, this requires two multiplications of size $\frac{n}{2}$.

However, some of these multiplications, we can memorize and replicate after doing a clever substitution. By doing that, you would replace the 8 multiplications you would normally do with 7:

$$T(n) \leq 7T\left(\frac{n}{2}\right) + \mathcal{O}(n^2) \Rightarrow T(n) = \mathcal{O}(n^{\log_2(7)})$$

The lower bound for matrix multiplication is $\Omega(n^2)$, but in practice it's $\mathcal{O}(n^{2.37})$. It's interesting because the gap between n^2 and what we have right now isn't known. The problem with finding an optimum is really theoretically interesting.

2.7 Kth Smallest / Quick Select

Sorting it, find k th element, or using a heap isn't that quick. It won't be better than $n \log(n)$. Good news – selection is easier than sorting.

`QuickSelect` goes like follows:

- Find a pivot p
- Divide A into two sub-arrays: $A_{\text{less}} = \text{elements } \leq p$, $A_{\text{more}} = \text{elements } > p$
- If $|A_{\text{less}}| \geq k$, return k th smallest in A_{less} , else find $(k - |A_{\text{less}}|)$ -th smallest element in A_{more}

Problem? If the pivot is close to the min or the max, we would get $T(n) \in \mathcal{O}(n^2)$. Best to find a pivot that evenly distributes the two elements.

So, are we going to select a pivot randomly? We want to do things deterministically. Select the average? That could be subject to outliers. Best solution?

2.7.1 Median of Medians

- Divide n elements into $\frac{n}{5}$ groups of 5 each
- Find the median of each group
 - Takes $\mathcal{O}(n)$ to here
- Find the median of $\frac{n}{5}$ medians = p^*
 - Takes $T\left(\frac{n}{5}\right)$, **this has to be done recursively**
- Create A_{less} and A_{more} according to p^*
 - Takes $\mathcal{O}(n)$

- $\frac{n}{10}$ of the $\frac{n}{5}$ medians are $\leq p^*$
- For each such median, there are *at least* 3 elements $\leq p^*$
- \Rightarrow There can be at most $\frac{7n}{10}$ elements that can be $\geq p^*$
- In the other direction, there can be at most $\frac{7n}{10}$ elements that can be $\leq p^*$
- **Our work here is done. But, reiterating the last point from Quick Select:** Run selection on one of A_{less} or A_{more} , however you did it before
 - Takes $T\left(\frac{7n}{10}\right)$

We get that

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

Note that $\frac{n}{5} + \frac{7n}{10} = \frac{9n}{10}$. Only a fraction of n , using a similar analysis to the one in the master theorem, $T(n) = \mathcal{O}(n)$

By the way, you don't need to memorize the specific algorithms – you just need to know how to use these basic ideas. It may be easier or harder than the memorization challenges.

You will be tested on the ability to use these tools.

3 Greedy Algorithms

Find a solution x maximizing or minimizing an objective function f . The challenge is when the space of possible solutions x is too large. Does it find the best solution possible, and how much time does it take to find the best solution possible?

Here's our approach:

- **Compute it one part at a time**
 - For each part, they will always pick the best solution right now **without regard for the future**

- Select the next part greedily to get the most immediate benefit (this needs to be defined carefully for each problem)
- Guarantees polynomial time
- Need to prove that this will always return an optimal solution despite having no foresight

Greedy algorithms don't always give the optimal solution! We'll play with some heuristics that we can prove its optimal, and then we can prove that the heuristic is in fact optimal.

Beware:

- Most of the time, greedy algorithms, will find one of many equivalently optimal solutions. Greedy algorithms will return one of the optimal solutions.

3.1 When should I use it?

When **all of the** following hold:

- Optimal substructure
 - Solving one problem contains optimal solutions to all other subproblems
 - An optimal solution at one level can be thought of adding onto an optimal solution to a smaller level.
 - Otherwise, greedy algorithms won't give you the best option.
- Greedy choice property
 - Among all choices, the greedy solution performs best

3.2 Interval Scheduling

Given some intervals (job j starts at s_j and finishes at f_j), two jobs are compatible in the same way how UofT timetables don't conflict. Find the maximum-size subset of mutually compatible jobs.

How do we do it?

- Always pick the task that finishes first.

Here comes the earliest finishing time heuristic.

For scheduling-type problems, there is a set of typical heuristics that could end up working well. Go through each heuristic and come up with a counterexample. If you come up with one, abandon strategy and try another.

Look for the one you can't prove is suboptimal, then prove that it is optimal.

What are the heuristics? Choose in the order of

- Earliest start time
 - Not – if the earliest task takes the entire day, blocking everything else out
- **Earliest finishing time (the best)**
- Shortest interval
 - Which happens to rule everything else out: two other tasks that one ends at noon, one starts at noon, yet the shortest job overlaps noon
- Fewest conflicts
 - Which happens to block everything else out; we can cram a bunch of other conflicts

You would struggle looking for a contradiction with earliest finishing time. You would switch over and take a look at the finishing time as the heuristic.

Before we improve its optimality, let's look at how long it will take for it to run.

To do earliest finishing time, we need to compute the earliest finishing time. The easiest way to do that is by sorting. To find the one that has the earliest initial starting time, just sort the list. Sorting takes $\mathcal{O}(n \log(n))$. Tiebreaks do not matter.

We then start picking tasks according to that list. Look at all tasks that end later than our current task in the same order and choose the earliest compatible one. You might think this will take $\mathcal{O}(n^2)$ time, but no, because we only need to compare all future-ending intervals with our previous interval, it's constant time in total.

3.2.1 Interval Scheduling Proof of Optimality By Contradiction

- Assume what we're proving is false
- Derive an inconsistency

For a contradiction, **assume that greedy is optimal**

Suppose that greedy selects jobs i_1, \dots, i_k sorted by finish time

Consider an optimal solution j_1, j_2, \dots, j_m which matches greedy for as many indices as possible from the start (we want $j_1 = i_1, \dots, j_r = i_r$ for the greatest possible r)

Both i_{r+1} and j_{r+1} must be compatible with the previous selection (the Greedy algorithm can never choose something invalid; so is the optimal solution)

What are we going to do with this?

Consider a new solution $i_1, i_2, \dots, i_r, i_{r+1}, j_{r+2}, \dots, j_m$

We have replaced j_{r+1} with i_{r+1} in our reference optimal solution.

- A conflict is impossible: $j_r = i_r$ and greedy cannot select anything that conflicts
- $i_{r+1}.\text{finish} \leq j_{r+1}.\text{finish}$. If that wasn't the case, greedy would've chosen j_{r+1} instead.
- $j_{r+1}.\text{finish} \leq j_{r+2}.\text{start}$, so, the right side of i_{r+1} is still compatible.

This results in a contradiction: the new solution matches greedy for $r + 1$ intervals. This means that greedy is optimal, **as any step of it can be extended into an optimal strategy**.

Key fact:

Can be extended into an optimal strategy in any step \Rightarrow Optimal

3.2.2 Proof of Optimality by Induction

Induction gives you a lot of scaffolding and gives you a structure to work off from. The structure of inductive proof gives you guides of what you should do.

Let S_j be the **subset of jobs picked by greedy after considering the first j jobs in the increasing order of finish time.**

- Define $S_0 = \emptyset$.

We call a partial solution *promising* if there is a way to extend it to an optimal solution by picking some subset of jobs $j + 1, \dots, n$ (indices of jobs sorted by finish time)

- $\exists T \subseteq \{j + 1, \dots, n\}$ such that $O_j = S_j \cup T$ is optimal

Our inductive claim: $\forall t \in \{0, 1, \dots, n\}$, S_t is promising.

If S_n is promising, then it must be optimal as there are no more jobs to consider.

Base case:

For $t = 0$, $S_0 = \emptyset$ is promising, because any optimal solution can extend this set.

Induction hypothesis: Suppose claim holds for $t = j - 1$ and optimal solution O_{j-1} extends S_{j-1} . WTP: O_j extends S_{j-1} with no issues

Induction step: At $t = j$, we have two possibilities.

On considering the j th job:

If greedy **did not** select job j :

- j must conflict with some job in S_{j-1}
- Since $S_{j-1} \subseteq O_{j-1}$, O_{j-1} cannot include job j either
- $O_j = O_{j-1}$ also extends $S_j = S_{j-1}$ (transitivity of the $=$ sign)

If greedy **did** select job j :

- $S_j = S_{j-1} \cup \{j\}$
- Consider the earliest job r in $O_{j-1} \setminus S_{j-1}$
- Consider O_j obtained by replacing r with j in O_{j-1}
- What happens if $r \neq j$? We still need to show that there are no conflicts.
 - We know greedy can't select anything with conflicts, so there are no conflicts

- Because greedy selects jobs in ascending order of finish time, job j 's finish time must be **before** job r 's finish time
- We know that O_j extends S_j with no issues (really obvious just read the above again).

This means that $\forall j \in \{0, \dots, n\}, S_j$ is promising.

3.3 What are the Arguments Saying?

Both proof methods make the same claim:

- The greedy solution after j iterations can be extended into an optimal solution \forall

The same key argument is used:

- If the greedy solution after j iterations can be extended to an optimal solution, then the greedy solution after $j + 1$ iterations can be extended to an optimal solution as well

The difference:

- Induction: this is the induction step
- Contradiction: we take the greatest j for which the greedy solution can be extended to an optimal solution (where anything ahead **we assumed for a contradiction cannot**), and we derive a contradiction by saying that it can (by extending the greedy solution after $j + 1$ iterations).

3.3.1 Another Flavor of Greedy Proofs

Greedy stays ahead

Let i_1, \dots, i_k be the greedy solution sorted by finish time

Let j_1, \dots, j_m be an optimal solution sorted by finish time

Claim: $f_{i_r} \leq f_{j_r} \forall r$

Proof:

Base case: $f_{i_1} \leq f_{j_1}$ because greedy sorts in finish time and will always initially choose the lecture that finishes the earliest.

Inductive step: Assume $f_{i_l} \leq f_{j_l}$ (inductive hypothesis). Show $f_{i_{l+1}} \leq f_{j_{l+1}}$.

By the greedy algorithm, i_{l+1} is the lecture that finishes the earliest that is compatible with f_{i_l} . Can $f_{i_{l+1}} > f_{j_{l+1}}$? No, as if it were, greedy would've selected $f_{j_{l+1}}$ (contradiction spotted). So, this means that our claim is proven.

Why does the claim imply Greedy is optimal?

Suppose greedy is not optimal ($k < m$).

By the claim, $f_{i_k} \leq f_{j_k}$.

However, $s_{j_{k+1}} \geq f_{j_k}$. But then, $s_{j_{k+1}} \geq f_{i_k}$, so $s_{j_{k+1}}$ must have been considered by the greedy algorithm. This is our contradiction.

3.4 Interval Partitioning / Scheduling Lectures In Rooms

Jobs j starts at time s_j and finishes at f_j . Two jobs are compatible if they don't overlap. Goal: group jobs into fewest partitions such that jobs in the same partition are compatible.

So, what heuristic should we use? Schedule, considering *this* ordering first:

- Finish time
- Shortest interval
- Fewest conflicts
- **Start time**
 - Start time is the one that works; everything else has counterexamples

So, the algorithm goes like this:

Input: a set of n lectures

- Sort lectures by start time

- $d = 0$ (number of allocated classrooms)
- For $j = 1$ to n :
 - If lecture j is compatible with some classroom (doesn't matter which one)
 - * Schedule it in that classroom
 - Else:
 - * Allocate a new classroom $d + 1$
 - * Schedule lecture j there
 - * $d = d + 1$
- Return the schedule

3.4.1 Runtime

Key step to check if a lecture is compatible with some classroom:

- Store classrooms in a priority queue, key = latest finish time of any lecture in the classroom

If lecture j compatible with some classroom?

- Same as, is $s_j \geq$ latest finish time of that classroom
 - Yes: add it, increase minimum key of that classroom to f_j
 - No: Create a new classroom, add lecture j , set key to f_j
- $\mathcal{O}(n)$ priority queue operations, $\mathcal{O}(n \log(n))$ time

3.4.2 Proof of Optimality

This is a different style of proof than the one before. We're going to prove a lower bound: that it is not possible to do interval partitioning with fewer than some number of classrooms. Then, we need to show that our algorithm matches that number.

First, prove that there is no way I can do better than k (the depth). Then, I need to show that greedy will always give me k (the depth) classrooms. If greedy achieves that and there is no way to do better than that, then greedy is optimal.

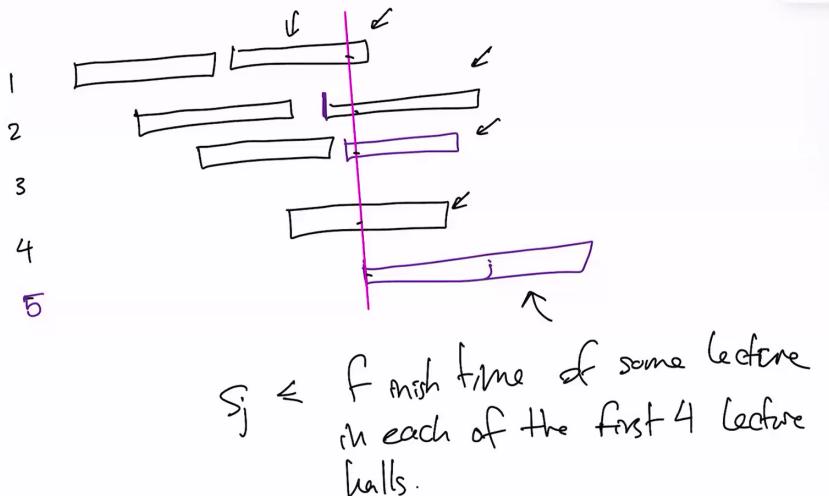
Proof of optimality (lower bound) – I CAN'T DO BETTER THAN DEPTH:

- WTS: $d = \text{Classrooms needed by greedy} \geq \text{depth}$
 - Where depth = maximum no. of lectures running at any time
 - Job i runs in $[s_i, f_i]$
 - The rationale, is that if you have 4 classes running at the same time, you at least will have to pack them into four different classrooms

Now, we want to claim our greedy algorithm uses only these many classrooms

Proof of optimality (upper bound) – I CAN'T DO WORSE THAN DEPTH:

- Let $d = \text{no. classrooms used by greedy}$. Show that $d \leq \text{depth}$.
- Classroom d was opened because there was a lecture j which was incompatible with some lecture already scheduled in each of $d - 1$ other classrooms.
 - All these d lectures end after s_j .
 - Since we sorted by the start time, they all start at or before s_j .
- \Rightarrow So, at time s_j , we have d mutually overlapping (conflicting) lectures
 - **We're showing that greedy can also detect the minimum depth (max no. of concurrently conflicting lectures).** (With our lower bound, we can get rid of the word “minimum”)
- Hence, $\text{depth} \geq d = \text{no. classrooms used by greedy}$. In other words, **the depth (highest no. of concurrently conflicting lectures) is at most the no. of classrooms used by greedy**.
 - When I opened the d th classroom, the depth at that point was d . The depth cannot be smaller than d .
 - From the lower bound proof: the depth cannot be larger than d , as if it were so the classroom would've already been made for that.



Overall, we showed that $\text{depth} \geq d$ and $d \geq \text{depth}$, then it can only be concluded that $d = \text{depth}$ and greedy only uses as many classrooms as the depth.

3.5 Minimizing Maximum Lateness

Problem:

- We have a single machine
- Each job j requires t_j units of time and is due by time d_j
- If it's scheduled to start at s_j , it will finish at $f_j = s_j + t_j$
- Lateness: $l_j = \max(0, f_j - d_j)$
- Goal: figure out the ordering of the tasks that minimize the **maximum** lateness of any of my tasks: $L = \max_j l_j$. *This is not minimizing the sum of all task lateness!*

Contrast with interval scheduling:

- We decide the start time
- There are soft deadlines

Let's look at the greedy template:

- Shortest processing time: t_j ascending

- Counter: $(t, d) : (1, 100), (10, 10)$ – opt: 0, this heuristic: 1
- Earliest deadline first: d_j ascending
 - It's this one
- Smallest slack first: $d_j - t_j$ ascending
 - Counter: $(t, d) : (1, 2, s = 1), (10, 10, s = 0)$ – opt: 1, this heuristic: 11

So, how do we prove the optimality of earliest deadline first? Let's make a couple observations:

1. All optimal solutions have no idle time. Solutions with idle time can be optimized to have no idle time.
2. Earliest deadline first has no idle time.
 - a. An inversion is (i, j) such that $d_i < d_j$ but j is scheduled before i (later deadline scheduled earlier)
3. By definition, earliest deadline first has no inversions.
 - a. Any schedule with no idle time and no inversions have the same maximum lateness. If two tasks have the same deadlines, it doesn't matter what order I pick them.
4. If a schedule with no idle time has at least one inversion, it has a pair of inverted jobs scheduled consecutively.
 - a. Whenever we have an inversion, we can swap two tasks such that the maximum lateness doesn't increase, and we will have no inversions.
5. Swapping adjacently scheduled inverted jobs do not increase lateness but reduces inversion count by one
 - a. Draw it out. This pushes overall due times back for both jobs.
 - b. Proof: $d_j \geq d_i$ yet j was scheduled first. Check that swapping an adjacent inverted pair reduces the total no. of inversions by 1. Tasks not i, j : swapping does not change lateness. Task i is shifted earlier, so task i can only be less late. Task j is shifted later, but it cannot be later than how late task i was.

- c. Lateness of j after swap is $f - d_j$
- d. Lateness of i before swap is $f - d_i$
- e. Since $d_j > d_i$, $f - d_j < f - d_i$

Here's how the proof of optimality by contradiction works **by showing that EDF is an optimal solution:**

Remember that optimal solutions can have inversions, for example if the only have due times as ∞ .

We use this fact: *If an optimal solution did not fully match greedy (has inversions), we can swap an adjacent inverted pair and reduce the no. of inversions by 1.*

- Suppose that the greedy earliest deadline first solution isn't optimal
- Consider optimal schedule S^* with the **fewest but at least one inversion** (otherwise it would be the same as EDF). Suppose it has no idle time.
- Because EDF is not optimal, S^* has at least one inversion \Rightarrow it has an adjacent inversion.
- Swapping the adjacent pair keeps the schedule optimal but reduces the no. of inversions by 1
- Contradiction

Here's the proof by reverse induction:

Claim: For each $r \in \{0, 1, \dots, \binom{n}{2}\}$, there is an optimal schedule with at most r inversions.

Base case $r = \binom{n}{2}$: trivial, any optimal schedule satisfies this as it covers every single combination possible

Induction hypothesis: Suppose the claim holds for $r = t + 1$.

Induction step: Take an optimal schedule with at most $t + 1$ inversions.

- If it has at most t inversions, we're done
- If it has exactly $t + 1 \geq 1$ inversions

- Assume no idle time without loss of generality
- Find and swap an adjacent inverted pair, which removes the inversion
- No. of inversion reduces by 1 to t (meaning there is an optimal schedule with at most t inversions)

Claim for $r = 0$ shows optimality of EDF

3.6 Lossless Compression

We have a document that is written using n distinct labels, represented in binary.

The naïve way is that if we have n distinct labels, we can encode them in $\lceil \log(n) \rceil$ bits. If the document has length m , this uses $m \log(n)$ bits. Thing is, in English, there are some characters that are way more frequent. The idea is, can we save space by assigning shorter codes to more frequent letters?

The problem:

- $a = 0$, $b = 1$, $c = 01$: what happens if we observe the encoding 01? Is it **ab** or is it **c**?

We need a prefix-free encoding. **No encoding may be the prefix to the other.**

The formal problem:

Given n symbols and their frequencies (w_1, \dots, w_n) find a prefix-free encoding with lengths (l_1, \dots, l_n) assigned to the symbols which minimizes $\mathbf{w}^T \mathbf{l}$.

A prefix-free encoding can be represented as a binary tree.

Here are some observations:

- A prefix-free encoding can be seen as a tree

The idea for Huffman is:

- Build a priority queue by adding (x, w_x) for each symbol x
- While $|\text{queue}| \geq 2$

- Take two symbols with the lowest weight (x, w_x) and (y, w_y)
- Merge them into one symbol with weight $w_x + w_y$

3.6.1 Proof of Huffman Encoding Optimality

Runtime is $\mathcal{O}(n \log(n))$.

Proof of optimality:

- Induction on the number of symbols n , that the Huffman tree built is optimal

Base case: For $n = 2$, both encoding which assigns 1 bit to each symbol are optimal

Hypothesis: Assume it returns an optimal encoding with $n - 1$ symbols

Idea: Run Huffman for 1 step. Let x, y be the characters selected by Huffman. x and y get put into a binary tree with priority $w_x + w_y$.

Treat this as a single symbol “ $(w_x + w_y)$ ” with weight $(w_x + w_y)$ and now we have $n - 1$ symbols.

By IH, Huffman will produce the optimal tree where $(w_x + w_y)$ is treated as one symbol.

Let T be optimal (for n symbols). Transform T such that x and y are siblings with the lowest weights, argue that the length of $H \leq$ length of T .

3.6.1.1 Lemma 1 Consider the case of n symbols. If $w_x < w_y$, then $l_x \geq l_y$ in any optimal tree. “If y occurs more often than x , then it shouldn’t have a longer encoding”.

3.6.1.2 Proof of that lemma Suppose for contradiction that $w_x < w_y$ and $l_x < l_y$. Swapping x and y strictly reduces the overall length as $w_x l_y + w_y l_x < w_x l_x + w_y l_y$. This is the rearrangement inequality:

If $a_1 \dots a_n$ are sorted and so are b_1, \dots, b_n , $\mathbf{a}^T \mathbf{b}$ maximizes the sum. Any rearrangement of the bs will only decrease the sum.

3.6.1.3 Lemma 2 Consider the two symbols x and y with lowest frequency which Huffman combines in the first step. \exists optimal tree T in which x and y are siblings. In other words, for some p , they are assigned encodings p_0 and p_1 .

3.6.1.4 Proof of that lemma:

1. Take any optimal tree
2. Let x be the label with the lowest frequency.
3. If x doesn't have the longest encoding, swap it with one that has (as swapping does not increase the score)
4. Due to optimality, x must have a sibling (if it didn't, then that prefix code would have to be used somewhere in the tree).
5. If x 's sibling is not y (the "second lowest frequency"), then swap it with y
6. Check that we didn't increase the average compression rate $\mathbf{w}^T \mathbf{l}$
7. Initial encoding was optimal so the modified one was also

Why this insight? Let's look at how the Huffman encoding tree was constructed. Right from the start, we take our two lowest frequency symbols and split them. Every time we split them, two children get created. The first children that are created are the entries with the lowest frequency. This process will always end up generating trees where the outcomes will have the lowest frequency notes as siblings.

Hence why there will always exist an optimal tree with the lowest frequency symbols as siblings.

3.6.1.5 Back to the Proof

Proof of Huffman encoding optimality (that minimizes $\mathbf{w}^T \mathbf{l}$)

WTP: All Huffman trees of size $\leq n - 1$ are optimal \Rightarrow all Huffman trees of size n are optimal

Let x and y be the two least frequent symbols that Huffman combines in the first step into xy

- Let H be the Huffman tree produced
- Let T be an optimal tree in which x and y are siblings (Lemma 2)
- Let H' and T' be obtained from H and T respectively by treating xy as one symbol with frequency $w_x + w_y$
 - This case, when encoding x and y would be treated as the same character
- Induction hypothesis: $\text{Length}(H') \leq \text{Length}(T')$ (expected length of an arbitrary symbol)
- Induction step:
 - $\text{Length}(H) = \text{Length}(H') + (w_x + w_y) \cdot 1$
 - * By disambiguating “ xy ” we need 1 more bit per xy to represent the whole string
 - $\text{Length}(T) = \text{Length}(T') + (w_x + w_y) \cdot 1$

4 Dynamic Programming

Where and when should we use greedy algorithms? In practice, greedy algorithms will almost never give you the optimal solution for anything. The cases covered in class before were just very specific examples where you can prove that greedy algorithms will give you the best answer. That doesn't usually happen.

Going forward, let's look at algorithms that give you advantages that aren't greedy algorithms.

Dynamic programming is just:

- Breaking the problem down into simpler subproblems, solve each subproblem once, and store their solutions. This is called “memoization.” It's just the art of keeping the best solutions in a type of data structure to store the optimal solutions to all the subproblems we end up getting.

4.1 Weighted Interval Scheduling

This is very similar to the interval scheduling problem. Find a set of mutually compatible jobs with the highest total weights/

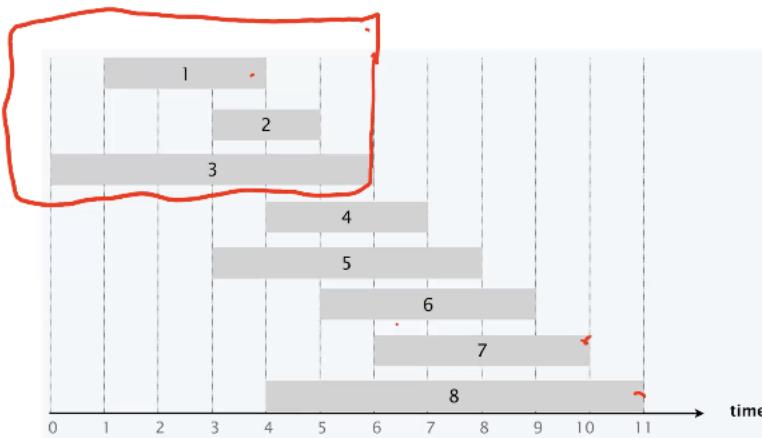
- Job j starts at s_j and finishes at f_j
- Each job has a weight w_j
- Two jobs are compatible if they don't overlap
- Goal: Find a set of S mutually compatible jobs with the highest total weight:
$$\sum_{j \in S} w_j$$

This problem will fail horribly if there is a massive imbalance of the weights if we tried to use our algorithms for interval scheduling. So is picking the largest weight. You can actually prove that any greedy algorithm can't guarantee an optimal solution.

So, how do we solve this?

- Jobs are sorted by finish time: $f_1 \leq f_2 \leq \dots \leq f_n$
- $p[j] = \text{largest } i < j \text{ such that job } i \text{ is compatible with job } j (f_i < s_j)$. This takes $\mathcal{O}(\log(n))$ time.

Why do we want to sort them by finish time? By sorting them like that, it's easier to see if a particular job is compatible. It's a convenient thing to do, as we'll be able to say, if we have job 1 that has a finishing time that is before job 2's start time, by sorting them according to the finishing time, we can binary search to figure out which jobs are compatible with the other.



- $p[8] = 1$
- $p[7] = 3$
- $p[2] = 0$
- $Opt[7] = \max(Opt[6], w_7 + Opt[3])$

If our optimal strategy has job 7, then our optimum has the weight of 7 plus the optimal value of $p[7] = 3$ which is $Opt[3]$

So really, (Opt : optimal choice given you start at that job)

$$Opt[N] = \max(Opt[N - 1], w_N + Opt[p[N]])$$

The bellman equation is:

$$Opt[j] = (0 \text{ if } j == 0 \text{ else } \max(Opt[j - 1] + w_j + Opt(P[j])))$$

We won't solve this algebraically. The bellman equation tells us how to write our code.

What's the cost of executing this?

No dynamic programming? It's going to take 2^n times. We end up creating a binary tree of depth n for our execution, so the leaf count is going to be $\mathcal{O}(2^n)$. At least we're getting an optimal solution – the optimum value is going to be the maximum of the optimal

value we are going to get if we take or don't take the action. Just by construction, the value you get at the next level up has to be optimal.

How do we do better? We store the values we were computing along the way. How many different values do we end up getting. We only have n items. So, there are only n distinct calls we end up making to each of these.

This gets us down to tie $\mathcal{O}(n \log(n))$.

The idea is to use the optimal substructure property.

4.2 Top-Down vs. Bottom-up Approach

Assume our problem looks like computing the n th Fibonacci number.

- Top down uses recursive calls, from n .
 - Translates right from the bellman equation.
 - Looks like: $\max(Opt(j - 1), w_j + Opt(p[j]))$
- Bottom up uses an iterative loop, building the look-up table. A loop would go from 1 to n , where a value i calculated will always depend on values less than i .
 - However, it can waste memory or time. You may not need to fill up the entire look-up table.
 - However, it does not waste stack space. Recursion is slow.
 - It does not require a global array, unlike top-down.
 - No waste in the worst-case scenario.
 - Looks like: For $j = 1$ to n : $M[j] = \max(M[j - 1], w_j + M[p[j]])$

Neither one of the two approaches is inherently better. Pay attention to the use cases.

In this course, do what you want. Figure out which way of thinking works the best for you.

4.3 Optimal Value vs. Optimal Solution

For the weighted scheduling problem, the optimal value is:

$$OPT(j) = \begin{cases} 0 & j = 0 \\ \max(OPT(j-1), w_j + OPT(p[j])) & j > 0 \end{cases}$$

Yet, the optimal solution is, as we need to know what `max` selects:

$$S(j) = \begin{cases} \emptyset & j = 0 \\ S(j-1) & (j > 0) \wedge OPT(j-1) \geq w_j + OPT(p[j]) \\ \{j\} \cup S(p[j]) & \text{else} \end{cases}$$

You're going to have to run both. This works for top-down and bottom-up.

4.4 Optimal Substructure Property

Where our optimal values depend on the optimal values for a sliced version of our input.

4.5 Knapsack Problem

Problem:

- n items: item i provides value $v_i > 0$ and weight $w_i > 0$. Items are not sorted.
- Knapsack has weight cap W
- Assume all values are integers
- Maximize $\sum v$ subject to $\sum w \leq W$

Solution:

$OPT(i, w)$ = max. value we can pack using only items 1 to i in backpack of capacity w . Compute $OPT(n, W)$.

Consider item i :

- If $w_i > w$, can't choose it. Pointless; use $OPT(i - 1, w)$
- If $w_i \leq w$ (otherwise):
 - If we choose i , the best value is $v_i + OPT(i - 1, w - w_i)$
 - If we didn't, best value is $OPT(i - 1, w)$

This gives us the Bellman equation:

$$OPT(i, w) = \begin{cases} 0 & i = 0 \\ OPT(i - 1, w) & w_i > w \\ \max(OPT(i - 1, w), v_i + OPT(i - 1, w - w_i)) & w_i \leq w \end{cases}$$

So, our look-up table is 2D.

4.5.1 Running Time

Considering for $OPT(i, w)$, $i \in \{1, \dots, n\}$ and $w \in \{1 \dots W\}$, there are $\mathcal{O}(nW)$ possible evaluations of OPT . However, each is evaluated at most once using memorization. The total running time is $\mathcal{O}(nW)$. The algorithm scales with the maximum weight, not the no. of bits needed to represent this algorithm. If we assume $W = \text{poly}(n)$, then the algorithm would run in polynomial time.

Is this polynomial? No, it's pseudo-polynomial. The inputs, W , \mathbf{v} , \mathbf{w} – the time should be polynomial in $\log W + \sum_{i=1}^n (\log v_i + \log w_i)$

4.6 Single Source Shortest Paths

This is a really important in general for navigation. Let's imagine that we have a **connected** graph that has a series of nodes on it. We would like to get between 2 nodes, one labeled s and t . We'd like to do it using the shortest possible paths. Each path has a distance (weight). We have a directed graph $G = (V, E)$.



Negative path weights don't really make sense in this context. If there is a negative path cycle, shortest paths are not even well-defined – you can traverse the cycle arbitrarily many times to get arbitrarily short paths. We'll be removing these cases. So, assume all our weights are positive.

Claim: with no negative cycles, there is always a shortest path from any vertex to any other vertex that is **simple (does not loop)**.

- Consider the shortest $s \rightarrow t$ path with the fewest edges among all the shortest $s \rightarrow t$ paths
- If it has a cycle, removing the cycle creates a path with fewer edges that is no longer the original path.

We're trying to attack the optimal substructure property. I'm trying to argue that the optimum path can be always chosen to have a particular form – simple and no loop. Why are we doing this? When we're coming up with a dynamic programming problem, we want to take the optimum of the score as a function of the problem size. The problem size is now the no. of edges our path ends up including.

The idea is, let's take a look at the optimum solution that we can find to the problem that uses paths up to a fixed length.

Consider a simple shortest $s \rightarrow t$ path P : it could be just a simple edge. But if P has more than one edge, consider u with immediately precedes t in the path.

- If $s \rightarrow t$ is shortest, $s \rightarrow u$ must be shortest as well and it must use one fewer edge than the $s \rightarrow t$.

Let $OPT(t, i) =$ length of the shortest path from s to t using at most i edges.

Then:

- Either this path uses at most $i - 1$ edges $\Rightarrow OPT(t, i - 1)$
- It uses exactly i edges $\Rightarrow \min_u OPT(u, i - 1) + l_{u \rightarrow t}$ where u and t has an edge (specifically, $u \rightarrow t$).

So, the full bellman equation takes up the following form:

$$OPT(t, i) = \begin{cases} 0 & i = 0 \vee t = s \\ \infty & i = 0 \wedge t \neq s \\ \min \{OPT(t, i - 1), \min_u OPT(u, i - 1) + l_{u \rightarrow t}\} & \text{otherwise} \end{cases}$$

Running time: The node count is n . $O(n^2)$ calls, each takes $O(n)$ times, so the TOTAL runtime is $O(n^3)$ where n is the edge count. Pay attention to the bellman equation. Now, we have to take the minimum value over all possible stopovers. Because it's the minimum value, if we have a series of nodes we're interested in, is that we have to look at all nodes.

No, it's not $O(|E|)$ as there are no cycles. The longest path length is n .

This is a polynomial time algorithm. It is not a terribly dumb algorithm. Trying to do a brute-force algorithm can take $n!$ Time.

For your interest, \tilde{O} is big O but neglecting subdominant polylog functions. Some authors disagree.

4.7 Maximum Length Paths

Can we use a similar DP to compute maximum length paths from s to all other vertices? (If there are positive cycles, we want SIMPLE paths).

The previous algorithm doesn't work for this. Why? If we use a path $s \rightarrow t$? The path $s \rightarrow u$ might in turn go through t , making the path no longer simple causing a cycle at the end. The maximum-length simple path is NP-hard. This means:

- We have good reason to suspect that a small variant on these existing problems lead to situations where no polynomial time graph exists.

The problems that we can solve in polynomial time is fragile. A small change can make them difficult is a lot. So, how can we tell if a problem is hard, and when will the techniques like dynamic programming won't work.

4.8 Chain Matrix Product

Need to matrix multiply? You better do it.

- Input: Matrices $M_{1\dots n}$, where M_i is $d_{i-1} \times d_i$ (the matrix multiplication chain is valid)
- Goal: Compute $\prod_{i=1}^n M_i$

Matrix multiplication is associative. While $A(BC) = (AB)C$, computing either may take very different time. So, it's not just doing exactly $n - 1$ multiplications in any order. **We have to decide the order of the bracketing.**

How expensive is matrix multiplication? If I need to multiply a $P \times Q$ and $Q \times R$ matrix, the complexity is $\mathcal{O}(PQR)$. For the purposes of this discussion, we'll just consider the $\mathcal{O}(n^3)$ matrix multiplication for now.

In the case where we do a matrix-vector multiplication, we're multiplying an $N \times N$ by a $N \times 1$ is $\mathcal{O}(N^2)$. Yet, for a matrix-matrix multiplication, it's $\mathcal{O}(N^3)$. So, what's our insight:

- $(MM)\mathbf{v}$ has complexity $\mathcal{O}(N^3 + N^2)$
- $M(M\mathbf{v})$ has complexity $\mathcal{O}(2N^2)$

Bracketing our matrix multiplication so that our matrix multiplication always happens on the vector gives substantial savings. Can we generalize this? In this case, it was dead simple.

But if all the matrices have different shapes, it's going to be harder. What we would like to do, is take a specification of a bunch of matrices being fed to us, and we'd like to output an algorithm that finds the cheapest way to multiply all of them together. Hence, this acts as a multiplier compiler, which figures out the cost and the optimal way to multiply.

4.8.1 A Small Example

- M_1 is 5×10

- M_2 is 10×100
- M_3 is 100×50

You will notice that

$$(M_1 M_2) M_3 \rightarrow 5 \cdot 10 \cdot 100 + 5 \cdot 100 \cdot 50 = 30000$$

$$M_1 (M_2 M_3) \rightarrow 10 \cdot 100 \cdot 50 + 5 \cdot 10 \cdot 50 = 52500 \text{ operations}$$

4.8.2 Why use DP?

DP requires an optimal substructure problem. We want to argue that an optimal solution can be expressed as a series of smaller optimal sub-solutions. At the end of the day, you can collapse matrix multiplications, but what's the best way to collapse them?

$$[X_1 X_2] [X_3, X_4, X_5] B$$

The way that dynamic programming comes into play, is that if we figure out the optimal cost of a matrix, it will lower the cost of the entire algorithm.

So, how are we doing to do it? We're just going to brute force ourselves every way we can decompose a series of matrix multiplications into two. For instance: from $ABCDE$ to

- $A(BCDE)$
- $(AB)(CDE)$
- $(ABC)(DE)$
- $(ABCD)E$

That's it. Because those are all the options we have for doing the final multiplication, by definition we are choosing the cheapest, and then choosing the cheapest implementation inside the outer one that is the cheapest.

This is how you should approach the DP problem. How can we think about the optimal solution being a function of smaller optimal solutions.

Now, let's formalize it.

4.8.3 Formalizing It + Runtime

$OPT(i, j) = \min$ operations required to compute $M_i \cdot \dots \cdot M_j$, where $1 \leq i \leq j \leq n$.

All that we need to specify is the first and the last index.

Then, we'll try to write an optimal solution for $M_1 \cdot M_2 \cdot M_3 \cdot M_4 \cdot M_5$. Then, $OPT(1, n)$ would be the whole optimum expression.

$$OPT(i, j) = \begin{cases} 0 & i = j \\ \min_{k : k \in [i, j]} OPT(i, k) + OPT(k + 1, j) + d_{i-1} d_k d_j & i < j \end{cases}$$

To outline the expression better:

$$\begin{matrix} [X_i \dots X_k] & [X_{k+1} \dots X_j] \\ OPT(i, k) & OPT(k+1, j) \end{matrix}$$

The cost of then multiplying the two together is $d_{i-1} d_k d_j$ (where d_{i-1} is the height of X_i)

So, what's the runtime?

- Running over all decompositions, `min` takes $\mathcal{O}(n)$. Assuming recursive runtimes are constant, the non-recursive runtime is just $\mathcal{O}(n)$.
- We have up to $\mathcal{O}(n^2)$, which is the size of the memorization array we're making.
- Our total runtime is $\mathcal{O}(n^2) \times \mathcal{O}(n) = \mathcal{O}(n^3)$.

4.9 Edit Distance

List indices are inclusive here.

Early spellcheckers use this exclusively. The edit distance problem is: how similar are the two strings?

$$X = x_1, \dots, x_m \quad Y = y_1, \dots, y_m$$

Where **similarity** is how many symbols I need to **delete** or **replace** to have one match the other.

For example, two strings:

```
1  occurrance
2  occurrence
```

This would take 6 replacements and 1 deletion.

Formalizing the problem:

- Strings X of length m and Y of length n
- $d(a)$: cost of deleting symbol a
- $r(a, b)$: cost of replacing symbol with a with b
 - Assume $r(a, b) = r(b, a)$ and $r(a, a) = 0 \forall a, b$

Goal: compute the minimum total cost for matching the two strings. What's the optimal structure?

- **Want to delete/replace at one end and recurse**
- Figure out distance of $X[: m - 1]$ and $Y[: n - 1]$ and then determine cost of $X[m]$ and $Y[m]$ (last index)

How are we going to deal with this? So, the bellman equation is, where $E[i, j]$ is the edit distance between x_1, \dots, x_i and $y_1 \dots y_j$:

$$E[i, j] = \begin{cases} 0 & i = j = 0 \\ B & i = 0 \wedge j > 0 \\ A & i > 0 \wedge j = 0 \\ \min(A, B, C) & \text{otherwise} \end{cases}$$

Where:

$$\begin{aligned} A &= d(x_i) + E[i - 1, j] \\ B &= d(y_j) + E[i, j - 1] \\ C &= r(x_i, y_j) + E[i - 1, j - 1] \end{aligned}$$

What does each variable mean?

- A : delete from $X[: i]$
- B : delete last from $Y[: j]$
- C : replace last of $X[: i]$ with last of $Y[: j]$, or the other way around. It doesn't matter.

So, what's the complexity?

- $\mathcal{O}(nm)$ time
- $\mathcal{O}(nm)$ space

There is no idea of insertion as we can pad the string with “invisible” characters.

4.10 The Traveling Salesman Problem

Given a network of locations, figure out the way I can visit each of them with the shortest path.

The general problem:

- Given a graph in n vertices such that each edge has positive weight w_{ij} , find a path P such that:
 - First vertex is 1
 - Last vertex is 1 (this is a cycle)
 - Each vertex except 1 appears once in the path
 - $\sum_{i=1}^{n+1} w_{P_i P_{i+1}}$ is minimal

How efficient do we expect this problem to be? The brute force method would take $\mathcal{O}(n!)$. The question, is can we improve this with DP? If we're assuming that everything is a cycle, start with our start node, jump to the rest of them, and consider the fastest way we can travel among them.

How are we going to do it?

Input:

- **Complete** (all vertices are connected to each other by one edge) directed graph $G = (V, E)$
- $d_{i,j}$ = distance from node i to node j

Output:

- Min. distance which needs to be travelled to the start from some node v , visit every other node exactly once, and come back to v . That is, the minimum cost of a Hamiltonian cycle.

The correct starting point doesn't matter.

Approach:

- Let's start at node $v_1 = 1$.
- We want to visit the other nodes in some order, say v_2, \dots, v_n
- Total distance is $\sum_{i=1}^n v_i v_{(i+1)}$, which we want to minimize

With DP, we can do a little bit better than $n!$ Solutions needed. Consider v_n (the last node before returning to $v_1 = 1$).

If $v_n = c$, find the optimal order of visiting nodes $\{2, \dots, n\}$ that ends at c . We need to keep track of the subset of nodes to be visited and the end node.

$OPT[S, c] =$ minimum total distance when starting at 1, visiting each node in S exactly once, and ending at $c \in S$

Answer to the original problem, which has to go back to 1:

$$\min_{c \in S} OPT[S, c] + d_{c,1} \quad S = \{2, \dots, n\}$$

To compute $OPT[S, c]$, we can condition over the vertex visited right before c in the optimal trip. So, the bellman equation is:

$$OPT[S, c] = \min_{m \in S \setminus \{c\}} (OPT[S \setminus \{c\}, m] + d_{m,c})$$

And the final solution is just:

$$\min_{c \in \{2, \dots, n\}} (OPT[\{2, \dots, n\}, c] + d_{c,1})$$

We'll have to do a total of $\mathcal{O}(n \cdot 2^n)$ calls. It takes $\mathcal{O}(n)$ time per call, so the total runtime is $\mathcal{O}(n^2 \cdot 2^n)$. This is much better than the naïve solution which has $(\frac{n}{e})^n$ runtime.

This problem is NP-complete. Some problems have fundamental limits to how good algorithms can go.

5 Network Flow

Network flow deals with the following type of problem:

Input:

- A directed graph $G = (V, E)$
- Edge capacities $c : E \rightarrow \mathbb{R}_{\geq 0}$
- Source node s , target node t

Output:

- Maximum flow from s to t

Sorry, greedy does not work well for this. What's a flow?

5.1 What's a Flow?

An assignment of weights to individual edges of a graph, which denotes how much material can flow through the pipe – that is $f(e)$, on edge e.

Here are the flow axioms:

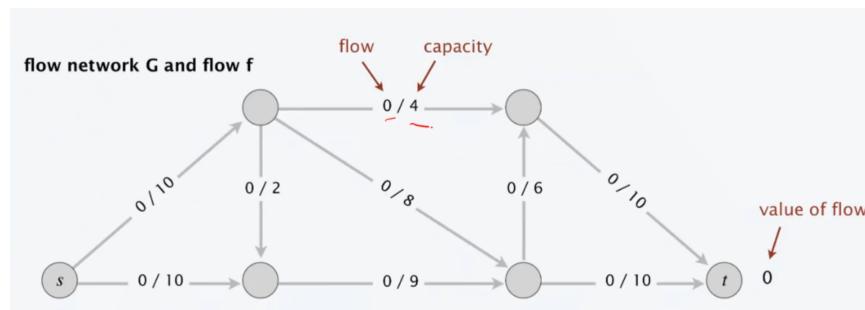
- Actual flow through a pipe can't exceed its capacity
- Flow is conserved: for each node: \sum what goes in = \sum what comes out

5.2 Why Doesn't Greedy Work?

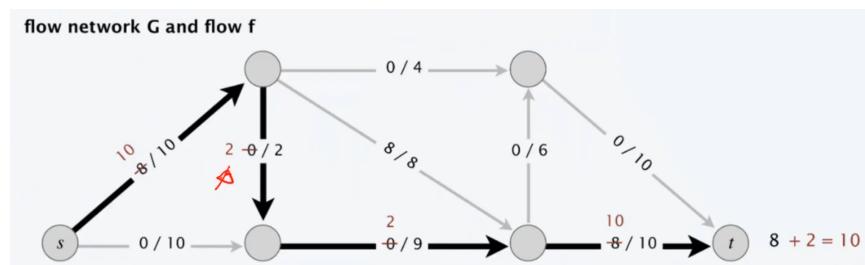
Greedy doesn't let us backtrack from bad decision. What else works? We'll need to be able to reverse bad decisions here.

Anyway, here's the greedy approach:

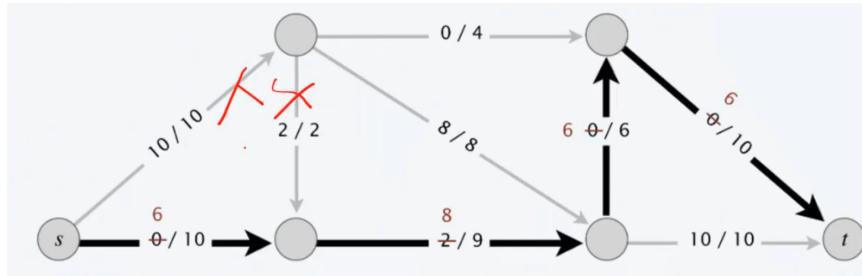
Let's imagine we label our graph with flow and capacity.



A natural greedy way is to pick up any path and figure out the maximum matter we can shove through there. In this case, the **bottleneck** is the maximum matter we can shove through. We can no longer shove through the bottleneck.



And finally:



This is unfortunately not the optimal solution.

Greedy just keeps making paths.

To recap, this is our greedy algorithm:

- Start from zero flow ($f(e) = 0$ for each e)
- While there exists an $s - t$ path P in G such that $f(e) < c(e)$ for each $e \in P$
 - Find any path P
 - Compute $\Delta = \min_{e \in P} (c(e) - f(e))$ (the bottleneck)
 - Increase the flow on each edge $e \in P$ by Δ

We made a sub-optimal allocation mid-way. This is a problem with greedy strategies. We don't get the ability to move back or correct our problems.

5.3 Residual Graph

Suppose the current flow is f . Define the residual graph G_f of flow f . G_f has the same vertices as G , but:

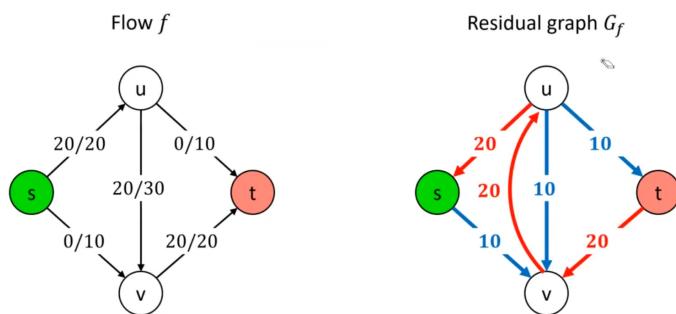
- For each edge $e = (u, v)$ in G , G_f has at most two edges
 - Forward edge: $e = (u, v)$ with capacity $c(e) - f(e)$
 - * We can send this much additional flow on e
 - * $c(e)$ is the original capacity of that edge
 - * $c(e) - f(e)$ is the residual capacity of the pipe

- Reverse edge $e^{rev} = (v, u)$ with capacity $f(e)$
 - * The maximum reverse flow we can send is the maximum amount by which we can reduce flow on e , which is $f(e)$
- We only really add edges of capacity > 0

All the information about the flow and the capacity are stored. Now, we have the flow left and the flow in total we can push through.

For any edge in the following diagrams that have 0 capacity, we don't draw that edge.

Here's an example:



The forward edge in the residual graph is the residual capacity. It is $c(s, v) - f(s, v) = \text{leftover}$ that can flow through.

The backwards edge represents how much flow is going through it.

5.4 Augmenting Paths



An augmentation involves finding a path from s to t and adding the bottleneck to each forward edge. Then, recompute the backward edges. I don't care which path I find as of right now, just find **a** path.

Let P be an $s - t$ path in the residual graph G_f . In a residual path, you wouldn't care whether you're walking the forward and reverse direction. The new graph just generates itself.

This lets us undo our mess-ups. Can you figure out why? When you flow stuff through a backwards edge, what you're doing is cancelling what flows into the tip of the backwards edge.

How does augmenting work? We augment flow f by sending $\text{bottleneck}(P, f)$ units of flow along path P . To send x units of flow along P , would mean:

- For each forward edge $e \in P$, increase flow on e by x
- For each reverse edge $e^{rev} \in P$, decrease the flow on e by x

For each edge, the sum of the forwards and backwards edge will always equal to the same value.

Why is this so neat? When we cannot augment anymore, we can prove that what we have is optimal. That is, you cannot have anything more flow out of s , the start node.

5.5 Augmenting Always Satisfies Capacity Constraints

Argue that the new flow after augmenting is a valid flow. Well:

- Increasing flow on e , we can do by at most the capacity of the forward edge e in G_f , which is $c(e) - f(e)$, so the new flow can be at most $f(e) + (c(e) - f(e)) = c(e)$
- Decreasing flow on e , we can do most by the capacity of the reverse edge e^{rev} in G_f , which is $f(e)$, so new flow is at least $f(e) - f(e) = 0$
- Meaning flow will always be $\in [0, c(e)]$

An alternative argument is by saying that for each edge we pass through (directions have to make sense), we note that:

$$\text{bottleneck} \leq \text{minimum residual edge capacity}$$

So no edge can be subtracted to be below 0.

For backward edges, our argument changes to:

$$\text{bottleneck} \leq \text{min flow down a reverse edge}$$

5.6 Augmenting Always Conserves Flow

For each node, what goes in, must come out. How do we prove this?

Say we have a path that has a series of forward edges and backwards edges. The idea is that, let's assume that x is the bottleneck. If x is the bottleneck, what we want to do for all edges is add x units of flow each time we pass a forward edge and subtract x units of flow each time we pass a backward edge.

For instance:

- This assumes that all edges are forwards edges; backwards edges would look like $\overset{\leftarrow}{A}$
- $\vec{A} \circ$ means net flow is $+A$
- $\circ \vec{A}$ means that net flow is $-A$

Each node on the path, except for s and t , has exactly **two** incident edges on the path.

- If both edges go into the node or leave the node, then the net flow for that node is increased on both or decreased on both, to that node.
 - $\vec{x} \circ \overset{\leftarrow}{x}$ $x + (-x) = 0$
 - $\overset{\leftarrow}{-x} \circ \vec{x}$ $-(-x) - x = 0$
- If there is one forward and one reverse node, then flow is increased on the incoming node but that same flow has to be decreased on the other side
 - $\vec{x} \circ \vec{x}$ $x - x = 0$
 - $\overset{\leftarrow}{-x} \circ \overset{\leftarrow}{-x}$ $-(-x) - (-x) = 0$

Since net flow remains 0, the new flow is a valid flow.

5.7 Ford-Fulkerson Algorithm

Here it is:

```

1 def MaxFlow(G) -> Flow:
2     set f(e) = 0 for all e in G
3     # while there is an s-t path in G_f
4     while P = FindPath(s, t, Residual(G, F)) != None:
5         f = Augment(f, P)
6         UpdateResidual(G, f)
7     return f

```

Will this algorithm halt?

5.7.1 Runtime – Number of Augmentations

- At every step, flow and capacities remain integers
- For path P in G_f , $\text{bottleneck}(P, f) > 0 \Rightarrow \text{bottleneck}(P, f) \geq 1$
- Each augmentation increases flow by at least 1
- Max flow, hence max no. of augmentations (they are equal) is at most $C = \sum_{e \text{ leaving } s} c(e)$
 - Sum of the capacities of all the edges leaving the sources

5.7.2 Runtime – Time to perform an augmentation

- G_f has n vertices and at most $2m$ edges (where m is the existing no. of edges in the non-residual non-augmented graph)
- Finding P and computing $\text{bottleneck}(P, f)$, and updating G_f has $\mathcal{O}(m + n)$ time
 - It takes $\mathcal{O}(m)$ time to check and update all edges
 - * By BFS/DFS
 - Updating all nodes, taking $\mathcal{O}(n)$

So, the total time is $\mathcal{O}((m + n)C)$

This is pseudo-polynomial time. The value of C can be exponentially large in the input length (the no. of bits required to write down the edge capacities). So, this can take an

extremely long time under some circumstances. With irrational capacities, you can show that this algorithm could loop forever (with rationals, we can use LCMs). Can we convert this to polynomial time?

5.8 Can we convert this to polynomial time?

Not if we choose an arbitrary path in G_f at each step. In the path below, we might end up repeatedly sending 1 unit of flow across $a \rightarrow b$ then reversing it. It takes X steps, which can be exponential in the input length.

5.9 Types of Polynomial Running Times

We have two quantities: number of integers provided as input and total number of bits provided as input

- Strongly polynomial
 - Running time polynomial in number of bits
 - Operation count polynomial in integers but not dependent on bits
- Weakly polynomial
 - Number of operations polynomial in number of bits
 - Example: scales logarithmically with an input integer
- Pseudo-polynomial
 - Number of operations is polynomial in no. of bits if input was written in unary (in other words, polynomial in the values of the input integers)
 - For example, polynomial to the value of type `int` you pass in

5.10 How to Achieve Polynomial Time

To achieve a weakly polynomial time, find the maximum bottleneck capacity augmenting path. This results in $\mathcal{O}(m^2 \log(C))$ operations. This is weakly polynomial time.

We can use BFS, which is how the Edmonds-Karp algorithm goes. It runs in $\mathcal{O}(nm^2)$ operations.

Would you prefer a pseudo-polynomial with better scaling of m and n , or one with worse scaling but with no scaling on C ?

5.11 Ford-Fulkerson Correctness

Notation:

For a node u , $f^{out}(u)$ and $f^{in}(u)$ is the total flow out of and into u respectively. This can apply for a set of nodes as well: $f^{out}(X)$ and $f^{in}(X)$.

$$v(f) = f^{out}(s) = f^{in}(t) = \text{value of the flow for the graph}$$

So, we can rewrite our constraints:

- Capacity: $0 \leq f(e) \leq c(e)$
- Flow conservation: $f^{out}(u) = f^{in}(u), \forall u \neq s, t$

5.11.1 Graph Cuts

(A, B) is an $s - t$ cut if a partition of vertex set V (such as $A \cup B = V$, $A \cap B = \emptyset$ with $s \in A$ and $t \in B$)

Its capacity, denoted $cap(A, B)$, is the **sum of capacities of edges leaving A**

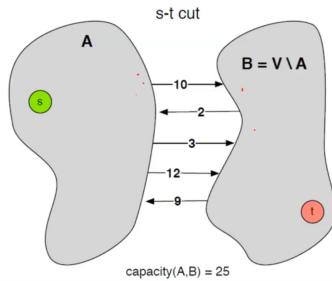


Image above: $10 + 3 + 12 = 25$

Cuts aren't unique.

Why do we care about this notion?

Theorem: for any flow f and any $s - t$ cut (A, B):

$$v(f) = f^{out}(A) - f^{in}(A)$$

Things drained to t is stuff coming in minus stuff coming out.

Theorem: For any flow f and any $s - t$ cut (A, B), $v(f) \leq cap(A, B)$.

This seems quite obvious as t can't drain more than what I could possibly have entering the cut that has t , but I'll lay it out:

$$\begin{aligned} v(f) &= f^{out}(A) - f^{in}(A) \\ &\leq f^{out}(A) \\ &= \sum_{e \text{ leaving } A} f(e) \\ &\leq \sum_{e \text{ leaving } A} c(e) \\ &= cap(A, B) \end{aligned}$$

Why do we care? Now we know that $v(f) \leq cap(A, B)$, then

$$\max_f v(f) \leq \min_{A, B} cap(A, B)$$

Which in other words, means **max value of any flow \leq min capacity of any $s - t$ cut (which follows from previous)**

We will now prove that the value of the flow generated by Ford-Fulkerson = capacity of some cut.

Implications:

- Max flow = min. cut
- Ford-Fulkerson generates max flows

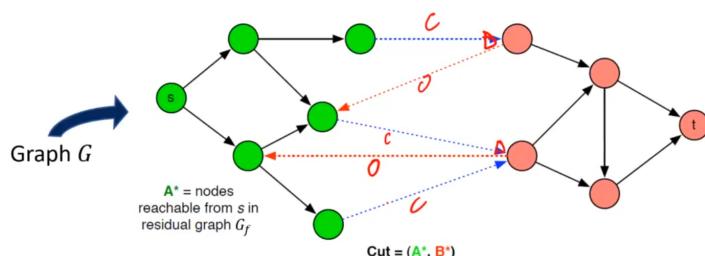
Theorem: Ford-Fulkerson finds maximum flow.

Proof:

- f : flow returned by FF
- A^* = nodes reachable from s in G_f
- B^* = $V \setminus A^*$

Claim: (A^*, B^*) is a valid cut. $s \in A^*$, by definition. $t \in B^*$, because when FF terminates, there are no $s - t$ paths in G_f as nothing can come out of s , so $t \notin A^*$

- Each blue edge (u, v) must be saturated
 - Otherwise G_f would have its forward edge (u, v) and then $v \in A^*$
- Each red edge (v, u) must have zero flow
 - Otherwise G_f would have its reverse edge (u, v) and then $v \in A^*$



Saturated means edge at its cap

$$\text{So } v(f) = f^{out}(A^*) - f^{in}(A^*) = \text{cap}(A^*, B^*)$$

Max flow-min cut theorem: in any graph, the value of the max. flow is equal to the capacity of the minimum cut.

Our proof already gives an algorithm to find a min cut:

- Run FF to find a max flow f
- Construct this residual graph G_f
- Let A^* = set of all nodes reachable from s in G_f
 - Easy to compute with BFS
- Then, $(A^*, V \setminus A^*)$ is a min cut (cut that minimizes $\text{cap}(A, B)$).

5.12 Edmonds-Karp Algorithm

At every step, find the shortest path from s to t in G_f , and augment.

```

1 def MaxFlow(G) -> Flow:
2     set f(e) = 0 for all e in G
3     # find the shortest s-t path in G_f
4     while P = BFS(s, t, Residual(G, F)) != None:
5         f = Augment(f, P)
6         UpdateResidual(G, f)
7     return f

```

What do we know about the bottleneck edge? It disappears, and it gets replaced by a backwards edge.

All edges but the bottleneck one will end up having both have a forward edge and a backward edge.



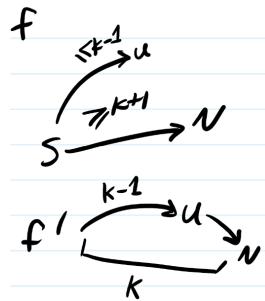
The new shortest path from $s \rightarrow t$ will definitely *not* be our previous path since the path was eliminated. At least, the new shortest path will never be shorter than the previous path. This means:

Distance between $s \rightarrow t$ will always increase.

5.12.1 Lemma 1: Proof that Distance never Decreases

$d(v)$ = shortest distance of v from s in residual graph G_f .

During the execution of the algorithm, $d(v)$ does not decrease for any v .



Proof:

- Suppose augmentation $f \rightarrow f'$ decreases $d(v)$ for some v
- Choose the v with the smallest $d(v)$ in $G_{f'}$ ($v \in V_d$, where $V_d = \{v \in V : d(v) \text{ decreased due to } f \rightarrow f'\}$)
 - Say $d(v) = k$ in $G_{f'}$, so $d(v) \geq k+1$ in G_f .
- Look at node u just before v on a shortest path $s \rightarrow v$ in $G_{f'}$, the new graph.
 - $d(u) = k-1$ in $G_{f'}$
 - $d(u)$ didn't decrease, so $d(u) \leq k-1$ in G_f
 - * $d(u)$ didn't decrease because $d(u) < d(v)$ in $G_{f'}$, meaning $d(u) \notin V_d$ (since v had the lowest value of $d(v)$ in $G_{f'}$ across all elements in V_d)

	$d(u)$	$d(v)$
G_f	$\leq k-1$	$\geq k+1$
\downarrow		\downarrow
$G_{f'}$	$k-1$	k

In G_f , (u, v) must be missing. We must have added (u, v) by selecting (v, u) in augmenting path P . But since P is the shortest path, it cannot have edge (v, u) with $d(v) > d(u)$ otherwise it would not be the shortest path.

5.12.2 Lemma 2: Critical

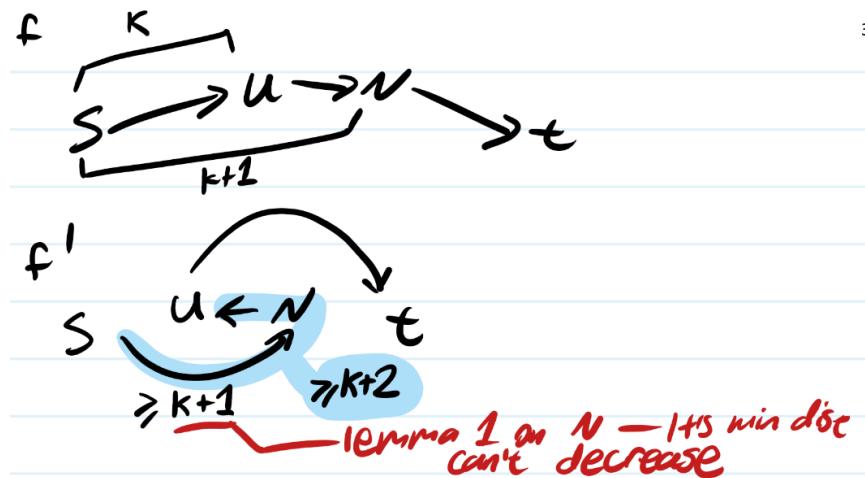
Call edge (u, v) critical in an augmentation step if

- It's part of the augmenting path P and its capacity is equal to $\text{bottleneck}(P, f)$.
- Augmentation step removes e and adds e^{rev} , if missing.

Lemma 2: Between any two steps in which (u, v) is critical, $d(u)$ increases by at least 2.

Proof: Suppose (u, v) was critical in G_f , so the augmentation step must have removed it.

- Let $k = d(u)$ in G_f
 - Because (u, v) is part of a shortest path, $d(v) = k + 1$ in G_f
- For (u, v) to be a critical again, it must be added back at some point
- Suppose $f' \rightarrow f''$ adds it back
- Augmenting path in f' must have selected (v, u)
- In $G_{f'} : d(u) = d(v) + 1 \geq (k + 1) + 1 = k + 2$



5.12.3 Proof of Edmonds-Karp Running Time

n = vertex count

- Each $d(u)$ can go from 0 to n (lemma 1)
- Each edge can be critical at most $\frac{n}{2}$ times (lemma 2)
- No. of needed augmentation steps is at most $m \cdot \frac{n}{2}$.
 - m edges, $\frac{n}{2}$ times it can be critical
- Each augmentation takes $\mathcal{O}(m)$ time to perform
- Hence, $\mathcal{O}(m^2n)$ operations in total

All that we ended up needing to do is we need lemmas 1 and 2 to show how many times we needed to perform these updates to the shortest path to get to the conclusion of the algorithm.

5.12.4 Notes on Running Time

FF is $\mathcal{O}((m + n)C)$, EK is $\mathcal{O}(m^2n)$. FF runs faster if C is small. This is a case where you don't want to run the efficient algorithm.

6 Network Flow Applications

Integrality theorem: forces our allocation to always be an integer flow count. This means that everything is forced to be a discrete decision.

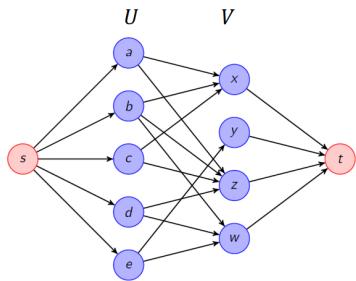
The following are always integral as the result of this theorem (are an integer):

- **Max flow** computed by FF and any of its variants
- **The flow on each edge**

6.1 Bipartite Matching

Given a bipartite graph $G = (U \cup V, E)$, find a maximum cardinality matching. For example, if edges between U and V represent mutual relationships, we want to max out the no. of people who are assigned dates, under the constraints each person can date at most one person.

Well, we can reduce this to a network flow problem:



We have a directed flow graph where we add a source node s and a sink node t . In this graph:

- All edges have capacity 1.
- $s \rightarrow$ all of U
- All of $U \rightarrow$ all of V
 - For a total of $|U| \times |V|$ edges
- All of $V \rightarrow t$

6.1.1 Correctness

Observation: There is a 1-1 correspondence between matchings of size k in the original graph and flows with value k in the corresponding flow network. **Value of flow = value of matching.**

Proof: Start with our original graph, only with the matchings, and translate this into a flow. Just take our flow diagram, and everywhere there is an edge, we would draw a link.

MATCHING \Rightarrow INTEGRAL FLOW

Given a matching, construct a flow graph. Argue that the matching size is the same as the flow size k .

Take a matching $M = \{(u_1, v_2), \dots, (u_k, v_k)\}$ of size k , and **construct a new flow f_m where:**

- For all $i \in [1, k]$, construct these edges with flow 1: $s \rightarrow u_i, u_i \rightarrow v_i, v_i \rightarrow t$. All other edges have flow 0.

Which, we can see, the flow is k , which is the same as our matching.

INTEGRAL FLOW \Rightarrow MATCHING

Given a flow, reconstruct the graph's matching M . Argue that the flow size is same as the matching size.

All edges have a flow of 1.

Take any flow f of value k . Construct the corresponding unique matching:

$$M_f = \text{set of edges from } U \text{ to } V \text{ with flow of 1}$$

- Since flow of k comes out from s , unit flow must go to k distinct vertices in U .
- From such vertex in U , one unit of flow goes to a distinct vertex in V .
- Each edge has unit flow, and we have k edges that connect U to V , so $|M_f| = k$.

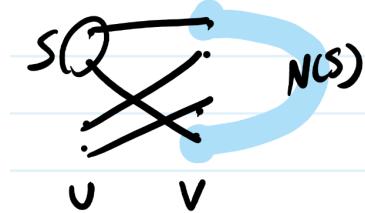
6.2 Hall's Marriage Theorem

When does a bipartite graph have a perfect matching?

- When the corresponding flow network has value n , where $n = |U| = |V|$, U and V being the two parts of a bipartite graph.
- For $S \subseteq U$, let $N(S) \subseteq V$ be the set of all nodes in V adjacent to some node in S

Observation:

If G has a perfect matching, $|N(S)| \geq |S|$ for each $S \subseteq U$ because each node in S must be matched to a distinct node in $N(S)$.



Hall's Theorem:

For all graphs G :

G has a perfect matching $\Leftrightarrow |N(S)| \geq |S|$ for each $S \subseteq U$

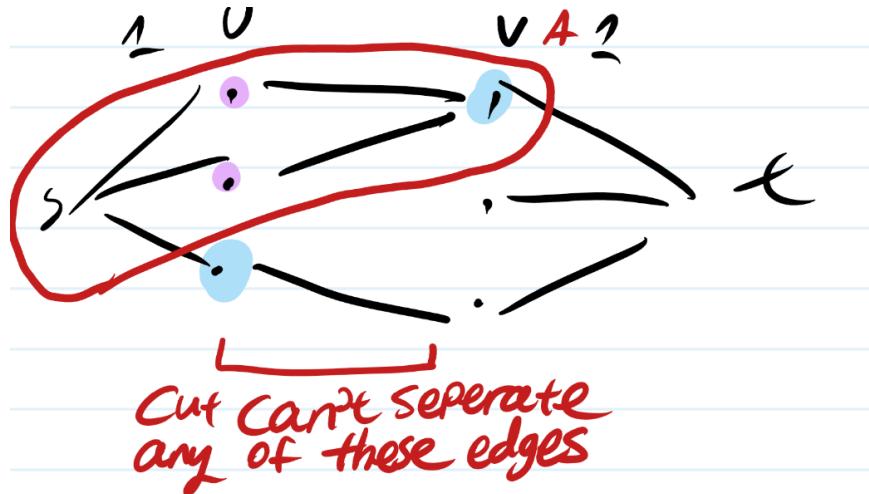
The proof for the \Rightarrow direction is easy to reason about. But

Proof (\Leftarrow): Using the contrapositive

G does not have a perfect matching $\Rightarrow \exists S \subseteq U$ such that $|N(S)| < |S|$

Set the capacities of the edges between U and V to ∞ .

- Suppose G doesn't have a perfect matching
- Hence, $\text{max-flow} = \text{cap(min cut)} < n = \text{no. of vertices in } U$
- Let (A, B) be the min-cut.
 - The cut can't slice up $U \rightarrow V$ edges as they have ∞ capacity
 - Has unit capacity edges otherwise: $s \rightarrow U \cap A$ and $V \cap B \rightarrow t$
- Let's form a cut. The cut can't be cutting up any of the ∞ edges. If you picture it out, here's an example:



Capacity of the cut = $|A \cap V| + |B \cap U| < |U|$
from not perfect matching assumption

$$\Rightarrow |A \cap V| < |U| - |B \cap U|$$

$$\Rightarrow |A \cap V| < |A \cap U|$$

vertices not in B are in A

Recall what we have to prove: $\exists S \subseteq U$ such that $|N(S)| < |S|$

- We can let $A \cap U$ be our analogue of S , so pick $S = |A \cap U|$.
- $N(S) \subseteq A$, otherwise, it will cross the cut.
- In hindsight, $N(S) \subseteq (A \cap V)$, because V is on the right side and S focuses on U , the left side.
- This follows:

$$|N(S)| \leq |A \cap V|$$

We then use what we've derived previously (which arrived at $|A \cap V| < |A \cap U|$):
vertices not in B are in A

$$|N(S)| \leq |A \cap V| < |A \cap U| = S$$

$$\Rightarrow |N(S)| < S$$

6.3 Edge-Disjoint Paths

Given a directed graph $G = (V, E)$, two nodes s, t , find the max. no of edge-disjoint $s \rightarrow t$ paths.

Two $s \rightarrow t$ paths P and P' are edge disjoint if they don't share an edge.

Max-flow formulation: Assign capacity of 1 to all edges.

How the algorithm works: Find a path, find another one, find another one, find another one, using a greedy algorithm. Keep in mind that with reverse edges, we can undo bad decisions and still maintain correctness.

Theorem: There is a 1-1 correspondence between **sets of k edge-disjoint $s \rightarrow t$ paths** and **integral flows of value k** .

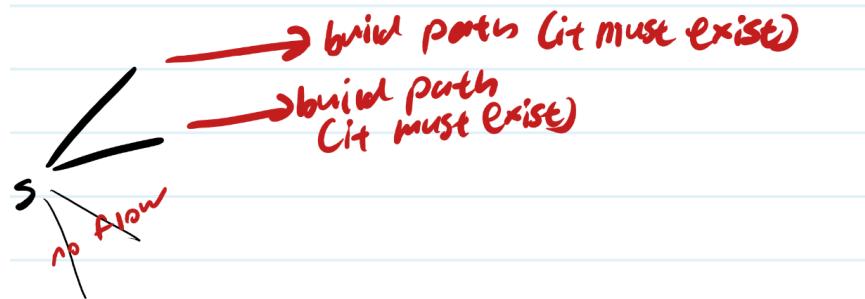
Proof: Paths \Rightarrow Flow

GIVEN paths, what's the flow? Just construct the flow given each path, knowing that constraints will be satisfied

- Let $\{P_1, \dots, P_k\}$ be a set of edge-disjoint $s \rightarrow t$ paths
- Define flow f where $f(e) = 1$ whenever $e \in P_i$ for some i , and 0 otherwise
- Since paths are edge-disjoint, flow conservation and capacity constraints are satisfied
- Unique integral flow of value k

Proof: Flow \Rightarrow Paths

- Let f be an integral flow of value k
- k outgoing edges from s have unit flow
- Pick one such edge (s, u_1)
 - By flow conservation, u_1 must have 1 outgoing flow.
 - Pick such an edge and continue building a path until you hit t
- Repeat for the other $k - 1$ edges



6.3.1 Using the Max-Flow Min-Cut Theorem

- Exercise!

- Show that to compute the maximum number of edge-disjoint $s-t$ paths in an undirected graph, you can create a directed flow network by adding each undirected edge in both directions and setting all capacities to 1

\hookrightarrow classic trick

- Menger's Theorem

- In any directed/undirected graph, the maximum number of edge-disjoint (resp. vertex-disjoint) $s \rightarrow t$ paths equals the minimum number of edges (resp. vertices) whose removal disconnects s and t

6.4 Multiple Sources and Sinks

Network flow is quite general and can model many problems.

Problem: Given a directed graph $G = (V, E)$ with edge caps $c : E \rightarrow \mathbb{N}$, sources $s_1 \dots s_k$ and sinks $t_1 \dots t_l$, find the maximum total flow from sources to sinks.

Solution: Create a true source and sink. Connect infinite edges. Simple.

- Add a new source s , edges from $s \rightarrow s_i \forall i \in 1 \dots k$ with ∞ cap
- Add a new sink t , edges from $t_j \rightarrow t \forall j \in 1 \dots l$ with ∞ cap
- Find max-flow from $s \rightarrow t$

Claim: 1-1 correspondence between flows in two networks

6.5 Circulation

Directed graph $G = (V, E)$

Each node has a demand: $d : V \rightarrow \mathbb{Z}$.

Output: want some circulation $f : E \rightarrow \mathbb{N}$ satisfying

- $\forall e \in E, 0 \leq f(e) \leq c(e)$
- $\forall v \in V : \sum_{e \text{ entering } v} f(v) - \sum_{e \text{ leaving } v} f(v) = d(v)$

Note that you need $\sum_{v:d(v)>0} d(v) = -\sum_{v:d(v)<0} d(v)$. Negative demands indicate supply. Positive demands indicate drain that the node will use up.

- $d(v) > 0$: take some flow out, more incoming than outgoing
- $d(v) < 0$: take some in
- $d(v) = 0$: they just transport. This is the transshipment node.

No more source and target. We want to see if a circulation is possible.

6.5.1 Network Flow Formulation: See if there is a circulation

Construct a graph G' : **Append to G** : Add source s and sink t

- For each supply node v with $d(v) < 0$, add edge (s, v) with capacity $|d(v)| = -d(v)$
- For each demand node with $d(v) > 0$, add edge (v, t) with capacity $d(v)$

Claim: G has a circulation $\Leftrightarrow G'$ has a max flow of value $\sum_{v:d(v)>0} d(v) = -\sum_{v:d(v)<0} d(v)$

i.e. all edges leaving s are saturated, and all edges entering t are saturated

6.5.2 Circulation with Lower Bounds

HAVE THE NODES THEMSELVES:

- **ABSORB ALL REQUIREMENTS (LEFT SIDE)**

- TO COMPENSATE FOR ABSORPTION, HAVE THE RIGHT SIDE DEMAND LESS

Input:

- Direct graph $G = (V, E)$
- Edge caps $c : E \rightarrow \mathbb{N}$ and lower bounds $l : E \rightarrow \mathbb{N}$
- Node demands: $d : V \rightarrow \mathbb{Z}$

Output:

- Some circulation $f : E \rightarrow \mathbb{N}$ satisfying
 - $\forall e \in E : l(e) \leq f(e) \leq c(e)$
 - $\forall v \in V : \sum_{e \text{ entering } v} f(v) - \sum_{e \text{ leaving } v} f(v) = d(v)$

Just transform it to a circulation problem without lower bounds:

$$\begin{array}{c} v \\ d(v) \end{array} - [2, 9] \rightarrow \begin{array}{c} w \\ d(w) \end{array}$$

↓

$$\begin{array}{c} v \\ d(v)+2 \end{array} - [0, 7] \rightarrow \begin{array}{c} w \\ d(w)-2 \end{array}$$

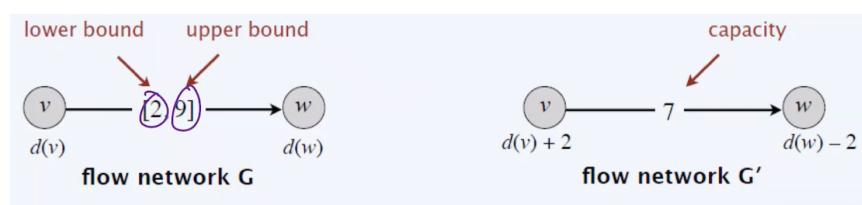
More generally, with $a \leq b$:

$$\begin{array}{c} v \\ d(v) \end{array} - [a, b] \rightarrow \begin{array}{c} w \\ d(w) \end{array}$$

↓

$$\begin{array}{c} v \\ d(v)+a \end{array} - [0, b-a] \rightarrow \begin{array}{c} w \\ d(w)-a \end{array}$$

Have the left output a *bit* less, and have the right demand a *bit* less. No change in satisfaction.1



Claim: circulation in $G \Leftrightarrow$ circulation in G'

Proof sketch: $f(e)$ gives a valid circulation in $G \Leftrightarrow f(e) - l(e)$ gives a valid circulation in G' .

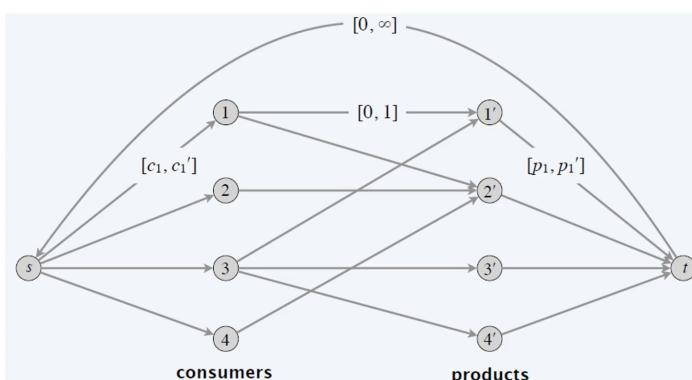
6.6 Survey Design

We want to design a survey about m products. We have one question in mind for each product. We need to ask product j 's question to between p_j and p'_j consumers.

There are a total of n consumers. Consumer i owns a subset of products O_i . We can ask consumer i questions about only these products. We want to ask consumer i between c_i and c'_i products.

Is there a survey meeting all these requirements?

- Create a network with special nodes s, t
 - Edge from s to each consumer i with flow $\in [c_i, c'_i]$
 - Edge from each consumer i to each product $j \in O_i$ with flow $\in [0, 1]$
 - Edge from each product $j \rightarrow t$ with flow $\in [p_j, p'_j]$
 - All demands and supplies are 0
- **Max-flow formulation:**
- Feasible survey iff feasible circulation in this network



6.7 Profit Maximization

There are n tasks. Performing task i generates a profit of p_i . We allow $p_i < 0$, i.e. performing a task might be costly.

There is a set E of precedence relations: $(i, j) \in E$ indicates that if we perform i , we must also perform j .

Goal: Find a subset of tasks S , subject to the precedence constraints, that maximizes $\text{profit}(S) = \sum_{i \in S} p_i$

How to solve: Nodes are tasks, node weights are profit, edges are precedence constraints. Goal: Find a subset of nodes S with highest total weight such that if $i \in S$ and $(i, j) \in E$, then $j \in S$

Way to formulate as a min cut:

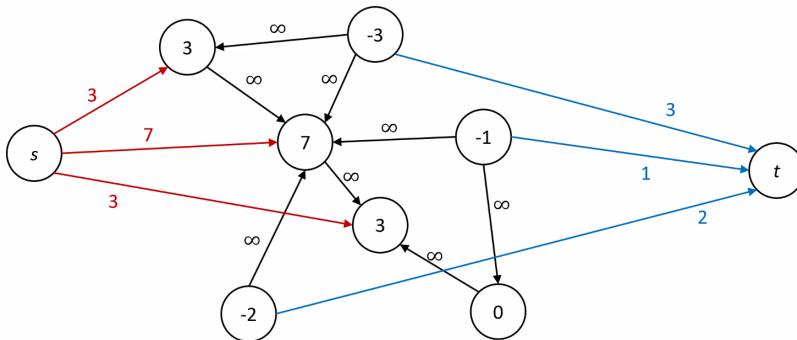
- Add source s and target t . Min-cut(A, B) \Rightarrow want desired solution to be $S = A \setminus \{s\}$

Construction:

Add each $(i, j) \in E$ with infinite capacity

For each i :

- If $p_i > 0$, add (s, i) with capacity p_i
- If $p_i < 0$, add (i, t) with capacity $-p_i$



To find the max. profit, we need to find a cut. IF we take a cut that breaks up two regions, what is the capacity of the cut?

For a graph:

1. A finite capacity cut exists
 - a. A cut will never have an infinite capacity as that can't be the min-cut
2. If $\text{cap}(A, B)$ is finite, then $A \setminus \{s\}$ is a valid solution
3. Minimizing $\text{cap}(A, B)$ maximizes $\text{profit}(A \setminus \{s\})$
4. Show that $\text{cap}(A, B) = \text{constant} - \text{profit}(A \setminus \{s\})$, where the constant is independent of the choice of (A, B)

7 Linear Programming

Linear programming lets us figure out how to deal with a more robust set of constraints. But, we need to look at a linear objective function.

Beverage	Corn (pounds)	Hops (ounces)	Malt (pounds)	Profit (\$)
Ale (barrel)	5	4	35	13
Beer (barrel)	15	4	20	23
constraint	480	160	1190	

Given all these constraints, how do we maximize profits? This is an example of a linear program, which is a lot easier to think than a profit maximization example using network flow. The way we're going to solve this is called a technique called linear program.

objective function

$$\begin{aligned}
 & \text{max } 13A + 23B \\
 \text{s. t. } & 5A + 15B \leq 480 \\
 & 4A + 4B \leq 160 \\
 & 35A + 20B \leq 1190 \\
 & A, B \geq 0
 \end{aligned}$$

constraint

decision variable

Profit
Corn
Hops
Malt

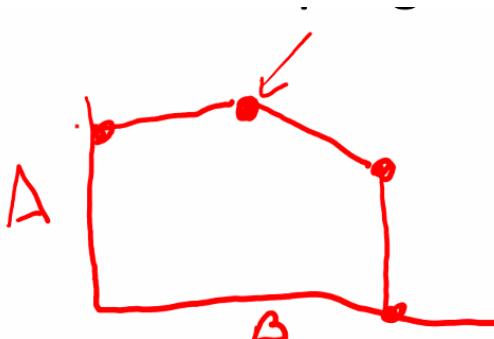
We want to express this as an optimization problem. What is the maximum value of $13A + 23B$? If this was the only thing, then the maximum value would be ∞ . Except, we have constraints.

We're going to look at different techniques we can use to solve this.

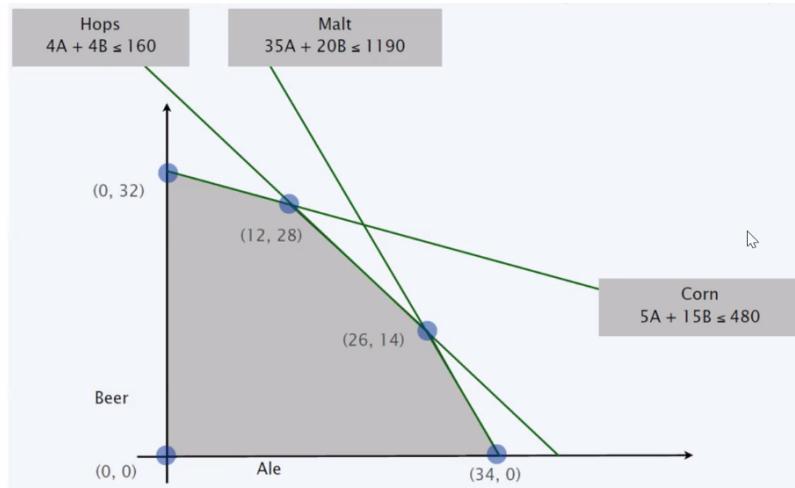
The core intuition we get for linear programming, that is finding the optimum value, is essentially just from the fact that linear functions are convenient to work with. The reason why they're so easy because we know where the min and the max of a linear function is. They're always on the boundary of the region you're interested in.

This is why LP programs are efficient.

Here's how we'll set up our problem: in A, B space, draw out the constraints. Their optimum must always be on their boundary. We need to show, moreover, that the optimum values are on the vertices of this diagram. That's how the simplex algorithm works.



From EVT, max/min must be at the boundary or any interior point where the derivative is 0.

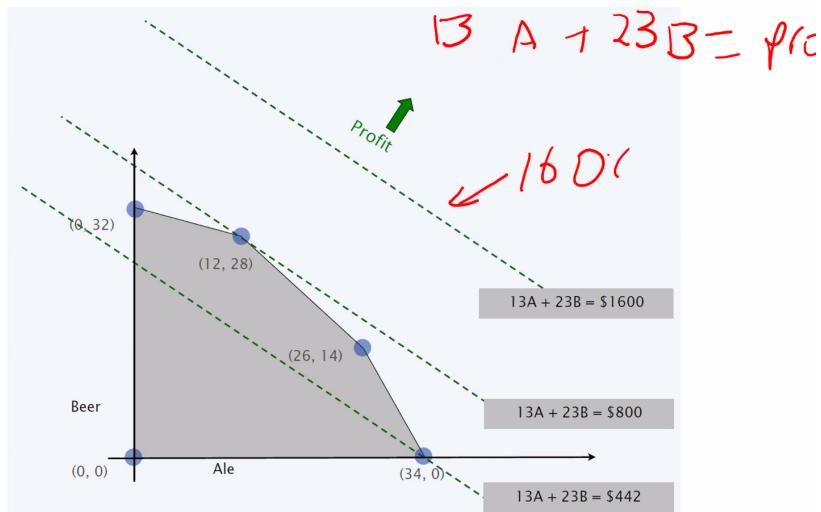


Inside the grey region, that is a feasible region. What to do? Find an initial point in the feasible region.

What's the first feasible region to look for? The one at the trivial point (0, 0).

Then, we're going to take each of the lines that correspond to the constraints. Everything above a line will violate constraints.

Now, the lines you see here?



On the same dashed line, you'll get the same profit.

Along the direction of increasing profit, things end up going up. You'll get a preferred

direction. Imagine height was profit – if you wanted to max out profit, how would you do it?

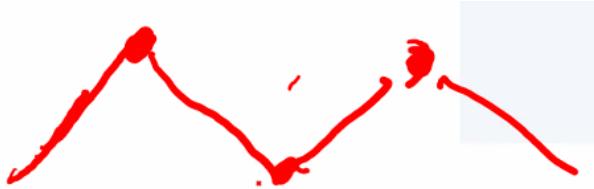
You want the profit line to go as high as you can until no other point ends up going outside. So, the point (12, 28) must be the global optimum. It is the furthest point feasible in our set that has a profit increase.

Now, it exists, but it is unique? For a linear program, it is almost always unique. What we're going to need, is to set up our constraints in such a way such that the feasible region always ends up forming a convex set.

What is a convex set? My set looks like this, such that if I have points x, y in S , then the line connecting x, y is also in that set.

Assume \exists only 2 feasible points x, y such that $P(x) = P(y)$ and $x \neq y$.

If that's true, what we'd have to have is NOT a convex set. With precisely two equal values, I'm forced to do something like this



If our feasible region is always convex, then there will exist a maximum, and there will always be a unique maximum.

Under very rare circumstances, if one of your constraints happened to be perfectly aligned with your feasible region, that is a degenerate situation. Tune a constraint by a small value and that breaks it. **Most of the time, you don't have to worry about it.**

How, exactly, are we going to end up guarantee that we'll have a convex set? By guaranteeing that our constraints are in the same form:

- Always \leq constraints (with constants on the right side), meaning $A_{11}x_1 + B_{12}x_2 \leq c_1$ such that the left side is always ≥ 0 .

Regardless of the objective function, there must be a vertex that is an optimal solution (except for the degenerate case).

7.1 LP Standard Formulation

Executing the simplex algorithm is easy if you eyeball it. Problem, is that when solving an optimization problem, there could be hundreds of thousands of dimensions. Can we have computers do this effectively? How can we rephrase the problem to have computers figure this out?

Input: $\mathbf{c}, \mathbf{a}_1, \mathbf{a}_2, \dots, \mathbf{a}_m \in \mathbb{R}^n, \mathbf{b} \in \mathbb{R}^m$

Goal:

- Figure out \mathbf{x} such that $\mathbf{c}^T \mathbf{x}$ is maxed out.
- Subject to $A\mathbf{x} \leq \mathbf{b}$ (vector inequality: element wise, ALL)
- Expanded:
 - $a_1^T x \leq b_1$
 - $a_2^T x \leq b_2$
 - \vdots
 - And $\mathbf{x} \geq 0$, element wise

What if the LP is not in standard form?

- Constraints that use \geq : $a^T x \geq b \Leftrightarrow -a^T x \leq -b$
- Constraints that use equality: $a^T x = b \Leftrightarrow a^T x \leq b \wedge a^T x \geq b$
- Minimization in our objective function: $\min c^T x \Leftrightarrow \max -c^T x$
- Variable x is unconstrained: replace x with two variables x' and x'' , replace all occurrence of x with $x' - x''$, and add constraints $x' \geq 0, x'' \geq 0$

7.2 LP Transformation Example

Min $-2x_1 + 3x_2$

Subject to

$$x_1 + x_2 = 7$$

$$x_1 - 2x_2 \leq 4$$

$$x_1 \geq 0$$

We change this to

Max $2x_1 - 3x_2$

Subject to

$$x_1 + x_2 = 7$$

$$x_1 - 2x_2 \leq 4$$

$$x_1 \geq 0$$

We change this to

Max $2x_1 - 3x_2' + 3x_2''$

Subject to

$$x_1 + x_2' - x_2'' = 7$$

$$x_1 - 2x_2' + 2x_2'' \leq 4$$

$$x_1, x_2', x_2'' \geq 0$$

7.3 Does an LP Always Have an Optimal Solution?

No! The LP can fail for two reasons:

1. It is infeasible, i.e. feasible region is empty i.e. $\{x \mid Ax \leq b\} = \emptyset$ and you can't meet any constraint.
 - a. The algorithm may not halt
 - b. The simplest algorithm must start in a point in the feasible region

- c. Can be prevented by perturbing b by a small random amount in each coordinate
- 2. It is unbounded (objective function can be made arbitrarily large or small). E.g., maximize x_1 subject to $x_1 \geq 0$

If LP has an optimal solution, we know that there must be a vertex which is optimal

7.4 Simplex Algorithm

- Let v be any vertex of the feasible region
- While there is a neighbor v' of v with a better objective value:
 - Set $v = v'$

Simple algorithm, easy to specify geometrically

Worst case running time is exponential

Excellent performance in practice

How do we implement this? We'll work with the slack form of LP. This is convenient for implementing simplex operations. We want to maximize z in the slack form.

Standard form:

$\text{Max } c^T x$ subject to

$$\begin{aligned} Ax &\leq b \\ x &\geq 0 \end{aligned}$$

In slack form:

$$\begin{aligned} z &= c^T x \\ s &= b - Ax \\ s, x &\geq 0 \end{aligned}$$

Where z is our objective function. We'll be working in the slack form, as it is easier to think about the manipulation as it makes it easier to move the vertices around. We want to push things until we end up in a bottleneck in the slack form.

For example:

Max $2x_1 - 3x_2 + 3x_3$ subject to

$$\begin{aligned} x_1 + x_2 - x_3 &\leq 7 \\ -x_1 - x_2 + x_3 &\leq -7 \\ x_1 - 2x_2 + 2x_3 &\leq 4 \\ x_1, x_2, x_3 &\geq 0 \end{aligned}$$

We will introduce two types of variables:

$$\begin{array}{ll} \text{Nonbasic Variables} & \\ & \overbrace{\quad\quad\quad}^{\text{Nonbasic Variables}} \\ \text{maximize} & 2x_1 - 3x_2 + 3x_3 \\ \text{subject to} & \\ \text{Basic Variables } \left\{ \begin{array}{l} x_4 = 7 - x_1 - x_2 + x_3 \\ x_5 = -7 + x_1 + x_2 - x_3 \\ x_6 = 4 - x_1 + 2x_2 - 2x_3 \\ x_1, x_2, x_3, x_4, x_5, x_6 \geq 0 . \end{array} \right. & \end{array}$$

The non-basic variables are the actual variables you're thinking for your program. The basic variables correspond to how much slack you have in the problem right there.

x_4 corresponds to the slack on the top equation, x_5 corresponds to the slack on the middle of the equation, and so on.

The way we're going to do this, is take z to just be the variable we're optimizing. We've written x_4, x_5 , and x_6 . What we want to do, is to be able to use this to move from a particular vertex, an initial value of x_1, x_2, x_3 to a new one that also is a point of minimum slack.

The way we're going to end up doing this:

1. Start at a feasible vertex. Which is **0**. Finding a feasible vertex is not so bad for these kinds of problems.
2. We set $x = 0$, which is all the non-basic variables.

To increase the value of z :

- Find a non-basic variable with a positive coefficient. This is called an entering variable. For the example above, that's x_1 .
- See how much you can increase the value without violating the constraints.

Try to increase!

Obstacles!

$\dots - \curvearrowleft$

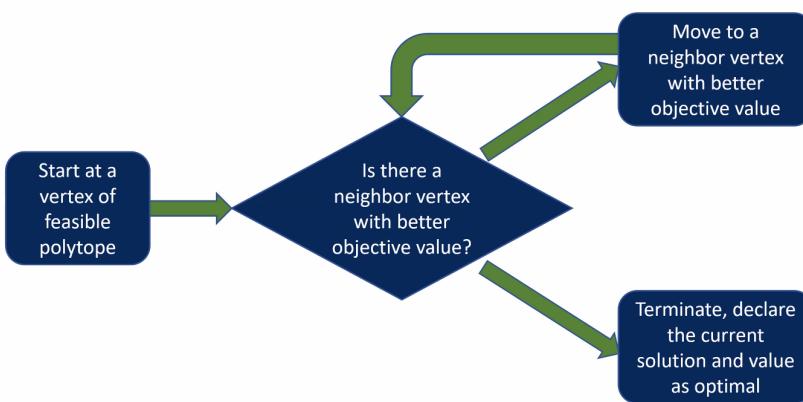
$$\begin{aligned}
 z &= 3x_1 + x_2 + 2x_3 \\
 x_4 &= 30 - x_1 - x_2 - 3x_3 \rightarrow x_1 \leq 30 \\
 x_5 &= 24 - 2x_1 - 2x_2 - 5x_3 \rightarrow x_1 \leq 24/2 = 12 \quad \cancel{x_1 \leq 12} \\
 x_6 &= 36 - 4x_1 - x_2 - 2x_3 \rightarrow x_1 \leq 36/4 = 9 \quad \cancel{x_1 \leq 9} \\
 x_1, x_2, x_3, x_4, x_5, x_6 &\geq 0
 \end{aligned}$$

Tightest obstacle!

$x_1 \leq 9$ is the tightest bottleneck, so set $x_1 = 9$. Once we end up doing that, we solve the tightest obstacle for the non-basic variable. we will end up with:

$$x_1 = 9 - \frac{x_2}{4} - \frac{x_3}{2} - \frac{x_6}{4}$$

Now, we repeat this process.



7.5 Resolving Negative Constraints

We started with $b \geq 0$, but then started with the vertex $x = 0$.

In the event $b < 0$ for any b , we will multiply every constraint with negative b_i by -1 so RHS is positive.

Convert to slack form:

Max $c^T x$ such that

$$\begin{aligned} a_1^T x &\leq b_1 \\ &\vdots \end{aligned}$$

To slack:

$$a_1^T x + s_1 = b_1$$

And multiply the constraint with negative b_i by -1 :

$$-a_1^T x - s_1 = -b_1$$

And that's your problem resolved.

7.6 Special Cases

7.6.1 Absolute Value Constraint

Constraint: $|x| \leq 3$

Replace with constraints $x \leq 3$ and $-x \leq 3$

7.6.2 Absolute value objective function

Minimize $3|x| + y$

- Add variable t
- Add constraints $t \geq x$ and $t \geq -x$ (same as $t \geq |x|$)
- Change objective to minimize $3t + y$

7.6.3 Max Objective Function

Minimize $\max(3x + y, x + 2y)$

Well: min t subject to

$$t \geq 3x + y$$

$$t \geq x + 2y$$

7.7 Certificate of Optimality

If the objective function $F = a^T x$, then x^* is good enough if $|a^T x_{opt} - a^T x^*| \leq \delta$. I care that the difference is less than δ for some $\delta > 0$.

The simplest algorithm runs in exponential time. It might take an extremely long time to find the absolute best solution. However, it turns out that we can *check* how close our solution is to our best possible one.

Even if we haven't found the best possible solution, we can use the dual formulation of an LP to test if our result is good enough. If so, then we can stop iterating.

If we were given the optimal value (or optimal input) to a linear program, how do we check that the optimal value is truly optimal? We want to get an upper bound on how high the optimal value can be if it was a maximization problem.

Now, look at our constraints. We can modify our constraints as much as we wish:

$$x_1 \leq 200$$

$$6x_2 \leq 1800$$

If I have the following objective function $\max x_1 + 6x_2$ and constraints $x_1 \leq 200$ and $x_2 \leq 300$, then I could just combine these two constraints by adding a linear combination of them:

$$\begin{aligned} x_1 &\leq 200 \\ 6x_2 &\leq 1800 \\ x_1 + 6x_2 &\leq 2000 \end{aligned}$$

Well, that's an upper bound, but I only used two constraints. Now, say I have this other constraint $(x_1 + x_2) \leq 400$. Then, I can write:

$$\begin{aligned} (x_1 + x_2) + 5x_2 \\ x_1 + 6x_2 \leq 400 + 300 \cdot 5 = 1900 \end{aligned}$$

So, the best value I can get is 1900. If I learn from the primal that the optimum is 1900, then I am done and solving the primal version of the problem worked.

Can we get a computer to go forward and actually be able to solve the problem, and tell us the optimum value? Can we do this at the same time as we are doing the primal algorithm? What this does, is it will end up coming up with a dual. The dual score is always going to be an upper bound on the optimum score. The primal score is always going to be a lower bound on the optimum score.

If you can minimize the value of the dual program through this linear combination of constraints, then the smallest value we can possibly find will correspond to the best score we can get from the primal problem.

We're approaching the same point from two different directions. If you build an algorithm that is chasing the optimum from both sides, you can close the gap to any tolerance δ you can set.

Is there a general way to verify that our solution for an LP is correct?

If we have 3 constraints, let's introduce variables $\mathbf{y} = \begin{bmatrix} y_1 \\ y_2 \\ y_3 \end{bmatrix}$. If we do that, then any

constraint that we end up getting, or at least the value of the score will be found by multiplying the inequalities by the multipliers:

Multiplier	Inequality
y_1	$x_1 \leq 200$
y_2	$x_2 \leq 300$
y_3	$x_1 + x_2 \leq 400$

After adding the constraints, we get (note that we factored the LHS to isolate for each \mathbf{x} , but you could do it anyway regardless):

$$(y_1 + y_3)x_1 + (y_2 + y_3)x_2 \leq 200y_1 + 300y_2 + 400y_3$$

What do we want for the y values?

$y_1, y_2, y_3 \geq 0$ because otherwise, the direction of the inequality flips.

We want the LHS to look like our objective function. It is sufficient for LHS to be an upper bound on the objective. If our objective function is $x_1 + 6x_2$, then we want $y_1 + y_3 \geq 1$ and $y_2 + y_3 \geq 6$

What do we want?

$$\begin{aligned} & \mathbf{y} \geq 0 \\ & y_1 + y_3 \geq 1, y_2 + y_3 \geq 6 \\ & \min 200y_1 + 300y_2 + 400y_3 \end{aligned}$$

If we claim that $x_1, x_2 = 100, 300$ is optimal with objective value 1900:

Then **pick** $y_2 = 5, y_3 = 1$

And from this:

$$5x_2 + (x_1 + x_2) \leq 5 \cdot 300 + 400 = 1900$$

This is an informal statement of the LP in dual form. There is a standard form:

7.8 Dual

Primal:

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

Dual:

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

Any solution for the dual, even if not optimal, will always be an upper bound for the primal solution.

No, the dual form is not necessarily easier.

You can use dual to verify optimality of a solution for a primal solution (given \mathbf{x}).

For the dual, the \mathbf{y} have nothing to do with the \mathbf{x} values. It is what we need to multiply the constraints to get the best upper bound for the solution. It is a different idea entirely. The objective function for the primal problem ends up making the constraints on the dual. The constraints on the primal form the objective on the dual. Most of the information you have from the objective function gets seen in the constraints and vice versa.

Don't think of the dual and the primal as giving you the same information. They are telling you subtly different things. The only thing that is similar, is if you find the global optimum for both, then they'll end up at the same point.

7.8.1 Weak Duality Theorem

For any primal feasible x and dual feasible y , $c^T x \leq y^T b$

Proof:

$$c^T x \leq (y^T A)x = y^T(Ax) \leq y^T b$$

7.8.2 Strong Duality Theorem

Just that $c^T x = y^T b$ for the optimum value.

7.9 Network Flow via LP

Problem: input directed graph $G = (V, E)$, edge caps $c : E \rightarrow \mathbb{R}_{\geq 0}$

Output: value $v(f^*)$ of a maximum flow f^*

Flow f is valid if capacity constraints are met and flow conservation is met. These are linear constraints.

We want to maximize $v(f) = \sum_{(s,v) \in E} f(s, v)$, which is a linear objective.

So what you are trying to do is maximize a linear function subject to linear constraints, which is an LP. So, here's our problem:

$$\begin{aligned} & \text{maximize} \quad \sum_{(s,v) \in E} f_{sv} \text{ subject to} \\ & 0 \leq f_{uv} \leq c(u, v) \quad \forall (u, v) \in E \\ & \sum_{(u,v) \in E} f_{u,v} = \sum_{(v,w) \in E} f_{v,w} \quad \forall v \in V \setminus \{s, t\} \end{aligned}$$

What is the dual? What is the dual trying to find? Since the max flow = the min cut, once you find the dual, you will notice that it's trying to find the cut with the smallest capacity.

7.10 Shortest Paths

Input: directed graph $G = (V, E)$, edge weights $w : E \rightarrow \mathbb{R}_{\geq 0}$, source s , target t

Output: weight of shortest weight path from $s \rightarrow t$

Variables: for each vertex v , we have d_v , distance from s .

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

7.11 Why use LP?

For these problems, we have different combinatorial algorithms that are much faster and run in strongly polynomial time. Why would we use LP? For some problems, we don't have faster algorithms than solving them with LP.

7.12 Integer Linear Programming

Variable values are restricted to be integers. For example:

$$\begin{aligned} & \max c^T x \text{ subject to} \\ & Ax \leq b \\ & x \in \{0, 1\}^n \end{aligned}$$

This makes the problem harder. We'll prove that this is NP-complete.

8 Complexity Theory

A Turing machine is a theoretical model of a computer that consists of two elements. What it has:

- A tape, infinitely long. Like an infinite array.

In this tape, there are a set of elements we can write. These elements can form some sort of alphabet $S = \{a, b, c, \dots\}$. These symbols are your data or your instructions.

You also have a head that reads the elements of the long tape.

When reading a particular value, there is a transition rule that will move the head to a different location. There can be different states for the Turing machine.

Under most circumstances, most of these models are computationally equivalent.

A Turing machine is not a real object. It is not a real computer. It is intended to be a model that carries the essence of what a computer ought to be. It is a theoretical model that describes a general-purpose computational device.

Back in the age of mechanical computers, computers would look close to the Turing machine. That's how the tapes end up storing the information. Later, they moved to magnetic storage. You would basically end up reading in these particular values and storing them in memory, making changes to the tape as you end up going through. Turing was inspired with this. Again, this is just a theoretical model.

A **computer** is nothing more than a set of nested abstractions that we are able to describe the parts of a device that we feel are important. As we take a look at the modern computers around us, a computer. Stores information as bits. It stores 0s and 1s.

But 0s and 1s don't exist. It's just voltage, which we quantize. These voltages are abstractions we end up using to model the collective behavior of many electrons moving around a system.

Nowhere in your computational model should there be bit flips. These are levels of details we don't care about, as computers are simply a model we can use to give us an understanding of how a device ends up working. There will be things that are very accurate with this model, and things that are not accurate with this model. Even if the Turing machine does not end up resembling a modern computer. Computers, it is still an essential theoretical cornerstone, as this model carries the essence of what we mean by computation. This boils down to what we are going to talk about *P*-reducibility, the

ability for turing machines to simulate other models of computation in polynomial time.

8.1 Church-Turing Thesis

The Church Turing thesis says that everything that is computable can be computed by a Turing machine.

- Computable \Leftrightarrow computable on a Turing machine.

What *isn't* computable? The halting problem. It is a generalization of the following:

If we have a while loop, is my code going to run forever, or is it going to halt?

The halting problem cannot be solved on a turing machine. So, not everything is computable on a turing machine. But every sensible function is **believed** to be computable on a turing machine (not formally statable or has been proven).

What problems are computable? What can we solve, and how do we define what that means? In complexity theory, we refer to this as the set of all problems that can be solved with a polynomial number of operations.

8.2 Tape Encoding

How does the encoding work on the tape? If we want to talk about encoding a particular set of numbers, we have to be able to figure out a way of representing that through a finite set of symbols.

S = Finite set of symbols

$$S^* = \bigcup_{n \geq 0} S^n = \text{set of all finite strings using symbols from } S$$

(The definition carries over from CSC236).

Input: $w \in S^*$, length of input is $|w|$, or the length of w on the tape.

Output: $f(w) \in S^*$. Length of output = $|f(w)|$.

Decision problems: output = “YES” or “NO”. A computation of a Boolean function. That is, $f : S^* \rightarrow \{0, 1\}$

8.3 Encoding

If I had two symbols $\{A, B\}$. We can map them to $\{0, 1\}$. Say I wanted to represent the number 2 using this. A natural way of representing that would be using binary: two would correspond to 10. We can use a binary representation to efficiently represent number through encoding.

Since turing machines don’t have numbers built into them, let’s think about how to formally encode inputs. Some type of binary represent is usually what we want to do, but we are not limited to that. We can use any base we want that we feel reasonable.

Our choice of any of these finite-base sizes will only change everything by a multiplicative logarithmic factor. For any of these types of encoding, we don’t care about how we do it.

The length of our encoding won’t matter for this case, but for anything above binary encoding.

8.4 Efficient Computability

A TM solves a problem in poly time if there is a polynomial p such that on every instance of n -bit input and m -bit output, the TM halts in at most $p(n, m)$ steps.

A computation is efficient if its time is polynomial.

The word “efficient” doesn’t seem intuitive to use.

In practice, most algorithms we design will end up having n or n^2 complexities. It is very rare that we get optimized algorithm that has scaling will goes to n^{100} .

8.5 Extended Church-Turing Hypothesis

Everything that is **efficiently** computable is computable by a TM in **polynomial** time.
This isn't proven.

8.6 P

The set of all decision problems that can be decided in polynomial time in a deterministic TM

Example:

- +
- -
- \sqrt{n}
- Network flow
- FFT
- Prime checking (yes, someone found it)

The set of things that are reasonable that we can end up doing.

8.7 NP

Nondeterministic polynomial time. Can be verified TRUE in polynomial time.

Bogo sort is one of them. Oh wait, this isn't a decision problem. Here's a better problem: the subset sum problem – given an array, is there a zero-sum subset?

Enumerating all subsets are exponential, but you can easily verify if a decision is correct.

A nondeterministic TM could guess the solution then test it, and if it is really lucky verify it.

NP is the set of all decision problems such that when you're given a piece of advice, you can check if the instance is a yes or not in polynomial time.

Formally speaking:

- There is a poly-time verifier TM V and another poly p such that:
- For all YES inputs x , there exists advice y with $|y| = p(|x|)$ on which $V(x, y)$ returns YES
- For all NO inputs x , $V(x, y)$ returns NO for every possible y

8.8 CO-NP

Same as NP, but except whenever the answer is NO, there is a proof of it. For example, in the subset sum problem, you can conclude “inconclusive” or “yes”. But CO-NP can quickly verify “NO” instances. For problems in CO-NP, it is difficult to check if a solution exists, but if a solution doesn't exist, you can check it.

8.9 Open Questions

We **do not know these**.

$$NP \stackrel{?}{=} coNP$$

$$P \stackrel{?}{=} NP \cap coNP$$

$$P \stackrel{?}{=} NP$$

For what we know:

NP and $coNP$ are not disjoint, and $P \subseteq NP \cap coNP$

Factoring is a weird example of a problem that is in $NP \cap coNP$.

8.10 Cook's Conjecture

Cook's conjecture

Like every sane person's belief. P is likely not equal to NP

8.11 Reduction

An NP-hard problem is a problem that is as hard or harder than every problem inside this.

x is NP-hard if $\forall y \in NP$, you can solve y given a black box that solves x in polynomial time.

If that is the case, then $y \leq_p x$.

A problem is NP-hard if that is the case.

A problem is NP-complete if:

$x \in NP$ and

x is NP hard

If we have a single problem that is NP-complete and someone finds a poly algorithm for, then we know that $P = NP$. But for over a century, we have found not a single instance.