Contents

0 Introduction and Preliminaries

Some Quite Useless Info

- **Who**: Hi, I am Qilin Ye, USC '24. Originally admitted to USC, in particular USC Thornton, as a piano major. Currently pursuing B.S. in MATH/CSCI and a PDP M.S. in applied math, along with piano still as minor. And I'm dying to graduate on time. :)
- **What**: This document presents comprehensive lecture notes for CS270. And you are reading this probably because you are taking CS270, so I hope you find my write-up helpful.
- **When**: I took CS270 in Fall 2022, taught by Professor David Kempe. These notes are based on the class material back then. I crammed these notes in basically one week in January 2024, so please excuse me from certainly inevitable typos. If you spot some, I can be reached via yql.skorpion@gmail.com (not leaving my USC email since I don't know when they'll recycle it) and I greatly appreciate it.
- **Why**: I love writing notes. I have probably compiled notes for 10+ upper-division and/or graduate math courses (including this one which was used as "textbook" for one semester of MATH425a), but only one CS course so far (CS567, machine learning). You can find all of them on my website. CS270 was — and has always been — one of my favorite CS courses (especially since I don't really like software engineering), and I feel bad not doing something about it. (*Plus I owe Professor Kempe many bars of chocolate.*)
- **How**: I type LaTeX in MacOS's Terminal using vim, with a few extremely powerful plug-ins: UltiSnips + LaTeX-live-preview, and Skim. Real-time. Probably even faster than writing, except when drawing diagrams. Check out this incredibly amazing post for a demo as well as how to get started. Also welcome to reach out to me regarding environment setups.

Learning Goals

- Fundamental algorithm design techniques.
- Specific algorithms (see below).
- Rigor and mathematical precision in describing and reasoning about algorithms[1].
- Limits of (efficient) computation.
- Experience in abstraction and mathematical modeling of real-world problems.

Topics Covered

(1) Greedy Algorithms:

- *"Pick whatever choice seems best at the moment and address future problems later."* More formally: make locally optimal irrevocable choices without taking long-term effects into consideration.
- Examples: Dijkstra, Prim, Kruskal.
- First thing almost everyone tries.
- Usually don't work (and are not optimal).

[1] You are not going to like this. You can't skip steps. You have to make the graders happy!

- When it does, the reason is usually interesting.

(2) Divide & Conquer:

- Break problem into two or more subproblems, solve one or more of them, and combine solutions.
- Examples: Merge Sort, Binary Search, Quick Sort.
- Most often improves runtime from $\mathcal{O}(n^2)$ to $\mathcal{O}(n \log n)$, from $\mathcal{O}(n^3)$ to $\mathcal{O}(n^2 \log n)$, or something similar.

(3) Dynamic Programming

- *"In order to do this, what do I need to do between the previous step and now?"*
- Key is to notice that many subproblems in an exhaustive search are repeated, so saving them saves time.
- Often allows polynomial-time solutions that are quite unexpected.
- Closely related to recursion, but not identical.

(4) Max-Flow / Min-Cut

- Two equivalent names for one algorithm; many useful applications.

(5) Theory of NP-completeness & Computability Theory

- Theory, a lot of theories.
- Limits of efficient (polynomial-time) computation, and of computation itself.
- We connect various problems via "reduction": if you can solve X, you can solve Y.

Some Additional Notes

- The primary textbook used throughout this course is **Algorithm Design** by Jon Kleinberg and Éva Tardos, referred to as 'KT.' Occasionally when I'm too lazy to draw diagrams, I directly take screenshots from the book. In this case you'll find 'KT' in the caption.
- I try my best to present proofs in a detailed manner, step-by-step, especially if a new algorithm/concept has just been introduced. However, as we get more familiar with certain proof techniques, I would start to make the related proofs more concise. This is especially the case in the chapter of DP — too many similar, templated "correctness/optimality" proofs and it starts to get boring, so I start to omit more and more.
- In these notes, I included a few optional sections that were not discussed in-depth in lectures but were nevertheless interesting ideas. Feel free to skip. They won't affect your understanding of CS270.
- Two cheatsheets, one for midterm, one for final, have been attached at the end of the file.

1 Stable Matching

1.1 Introducing the Stable Matching Problem

We begin the course with a problem that bears significant practical application: the **Stable Matching Problem**. In 1962, mathematical economists David Gale and Lloyd Shapley asked the following question:

> Could one design a college admissions process, or a job recruiting process, that was self-enforcing?

Internally, each student holds a ranking over universities, and each university has its ranking over all students. Our goal, simpler said than done, is to find a "good" assignment of students to universities that makes everyone — students and universities — happy. And of course, this theme easily generalizes to any matching problem that involves the matching of two types of entities (e.g. owners to pets).

But what do they mean by a self-enforcing process? Think of it as stability. Once the final admission decisions have been made, we want to ensure that no chaos may ensue. For the university, since it needs to admit a certain number of students each year, it wants to ensure that those admitted will actually commit to the school. Conversely, the worst possible nightmare for a student applicant is if they receive an offer, only to be retracted by the university later because now it wants someone else instead.

To describe the situation as a matching problem, we imagine that there is an invisible power that takes care of this admission process, assigning each student to a university. Now we restate what self-enforcing or stable means in a more rigorous manner:

Definition: Stable Matchings (College Admission ver.)

A matching is **stable** if, for each university-student pair (u, v) that are not assigned, at least one of the following is true:

(1) s prefers their assigned university over u;

(2) u prefers all of their assigned students over s.

It should be clear that if neither is satisfied, then "backroom deals" might happen: university u can secretly kick a student it ranks below student s and offer s admission, and now both u and s are happier than before. And these backroom deals may lead to more backroom deals. Chaos!

Simplifications & Justification

For the purpose of this course, we make the following assumptions: (i) we use "men" and "women" to represent the two types of entities, (ii) there are as many men as women, and (iii) each man is matched with precisely one women (and vice versa). Our goal can now be summarized as follows:

> Given a bipartite[2] graph of n men and n women, each holding a total order[3] of preference over the other gender, find a perfect[4] stable matching.

[2] Nodes are divided into two sets (in this case men and women), and edges only exist between the two sets, not inside either set.
[3] Each man ranks all women in his heart, and vice versa.
[4] Graph in which each node is incident on exactly one edge; a bijection between men and women represented by the matching.

A natural question, as you may now ask, arises: **did we make the problem too easy**?
Luckily, no. Our simplification includes (i) monogamy (one-to-one relationship) and (ii) equal number of entities (same number of men and women). We can easily convert our simplified stable matching problem into these more general cases:

(1) Universities/companies with multiple slots. For a university u with k slows, we generate k copies of the school: $u_1, ..., u_k$. While copying, we also copy the original university's ranking over all students. And for each student, update their ranking of universities by inserting all k copies to where the original u is in arbitrary order (key idea: what was above u is still above each u_i, and what was below u is now below each u_i).

(2) Unequal number of men and women. Create dummies. If there are n men and $n+k$ women, create k "dummy men" that every women ranks at the bottom. Their rankings don't matter. If a women gets matched to a dummy man, simply keep her unmatched.

Of course, there are also other implicit modeling simplifications we've made, a linear scale of preference represented by ranking, everyone prefers matched over unmatched, everyone really knows their ranking, and so on. But we shall ignore these and delve into our first algorithm of the course. One more thing though:

Remark: Stable matching is not unique[5]. Consider a perfect matching problem between two men A, B and two women $1, 2$. Their preference is given as follows:

Person	Preference
A (man)	prefers 1 over 2
B (man)	prefers 2 over 1
1 (woman)	prefers B over A
2 (woman)	prefers A over B

You can trust me, or you can convince yourself that both $\{(A,1),(B,2)\}$ and $\{(B,1),(A,2)\}$ are stable.

This space is intentionally left blank

[5]There certainly exist scenarios where there is only one stable matching, but as shown above, there are also scenarios where we can find more than one stable matching. Mathematicians are known for bad phrasing, and this could be one example — when I say "X is not unique," what I really mean is, "X is not *necessarily* unique."

1.2 The Gale-Shapley Proposal Algorithm

Having introduced the problem, we now present an algorithm that always provides an affirmative answer to the goal formulated above.

Theorem: Gale-Shapley

The following algorithm returns a perfect stable matching.

Algorithm 1: Gale-Shapley Proposal Algorithm

1 **Inputs**: men $\{m_i\}_{i=1}^n$ and women $\{w_i\}_{i=1}^n$
2 **Notation**: $\nu(f)$ denotes the value of flow f; $f(e)$ denotes flow passing of f through edge e

---

3 start with empty assignment $S = \{\}$
4 **while** there is still a single man **do**
5 pick any single man m.
6 let w be m's highest-ranked woman to whom he has never proposed
7 **if** w is single **or** prefers m over her current partner **then**
8 match m and w (temporarily)
9 **if** w was previously matched **then** her old partner becomes single
10 **else**
11 do nothing (and m will never propose to w again)
12 **Return** final temporary assignment

Before delving into the main proof, we first observe the following lemmas:

Lemma. Once a woman is matched for the first time, she will never become single again, and her matches can only improve (according to her ranking) over time.

Proof. Trivial[6]. This is exactly what the `if` statement on line 7 does. □

Lemma. The algorithm terminates. In particular it terminates in $\mathcal{O}(n^2)$ time.

Proof. If not, then there would be a single man forever. And since there are a equal number of men and women, there would be a single woman. But by the previous lemma, she was nevder proposed to, in particular not by this forever-single man. He can then just propose to her. Contradiction.
The $\mathcal{O}(n^2)$ comes from the fact that each iteration includes one proposal, and there are at most n^2 proposals in total (n man, each proposing to every woman). □

Lemma. We get a perfect matching at termination.

Proof. By the previous proof, there is no single (unmatched) man nor woman at termination. □

[6]Really bad word. Don't learn from me. I am beyond salvation, and I will continue throwing "trivial" into proofs.

Proof of correctness of Gale-Shapley. That the output is a perfect matching has been proven above, so it remains to prove that it is also stable. Suppose for contradiction that the output isn't. That is, there exists a man m and woman w, partnered with w' and m', respectively, that would prefer each other over their current partners. Since men propose down their preference list, m must have proposed to w before he ends up being matched with w'. Two possibilities: (i) he got rejected immediately, or (ii) he was dumped by w later.
Either way, when this event happened, w was with some other man m'' that she (strictly) preferred over m. By the first lemma, since things only improve for women, that she ends up with m' indicates that she prefers m' over m'' (or equally). Relating m, m', and m'', we see that she likes m' better than m, contradiction. □

1.3 Gale-Shapley, Men-Optimality & Women-Pessimality

A complication of the G-S algorithm comes from the phrasing of line 5: "pick any single man m." It is far from obvious whether different execution sequences of the G-S algorithm will lead to the same outcome. After all, recall also that perfect stable matchings need not be unique. But once again, the answer is yes. Even better, we can precisely characterize the matching produced by the algorithm. First, some definitions.

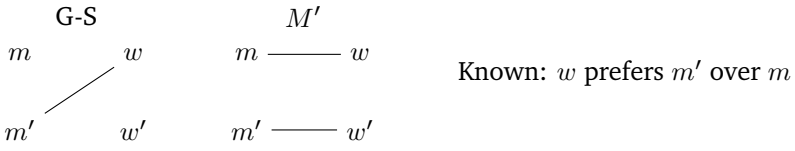
Definition: Possible Matchings, Best/Worst Possible Match

We say a woman w is a possible match of a man m if there exists a stable matching that contains the pair (m, w). Naturally, the **possible matchings** for m is then the set of all women w satisfying this property. We denote this set by $P(m)$.
We define the **best possible match** for m, denoted $b(m)$, to be the highest-ranked w in $P(m)$ according to m's ranking. We define the **worst possible match** for m analogously.

Theorem: Gale-Shapley is Men-Optimal

Gale-Shapley returns a matching in which each man m matches with $b(m)$. It is, in this sense, **men-optimal**.

Proof. We prove by contradiction, assuming that some man is not matched up with his best possible match in the G-S output. Hence, at some point he must have been rejected or dumped by her.
Now consider the first time where *some* man m got dumped or rejected by one of his possible matches $w \in P(m)$. Say w rejected/dumped m for another man m', whom she ranks strictly higher than m. [*Note it's always possible to find such m and w, as the previous paragraph offers one particular example.*]
From the definition of $P(m)$, there exists another stable matching M' in which (m, w) end up with each other. In particular, m' is with someone else in M'. Let us call this woman w'.



Known: w prefers m' over m

The important question to consider now is: who does m' prefer between w and w'?

(1) If m' prefers w over w', then M' is not stable — (m', w) are not matched yet they both prefer each other over their current partners.

(2) If m' prefers w' over w, then in G-S, where m' ends up paired with w, we deduce that he must have proposed to w' earlier and either got rejected or dumped later. Note that this event must have happened

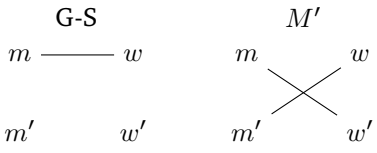
earlier than when w rejected/dumped m: to see this, observe that w rejected/dumped m because of m', but m' proposed to w only after his failed attempt to match with w'. Since (m', w') is indeed paired in a stable matching M', w' is a possible match of m', but this contradicts the highlighted assumption.

Therefore, G-S is man-optimal. □

The opposite result, unfortunately, can be easily derived for women, though this was not covered in lecture:

Gale-Shapley is women-pessimal.

Proof. Once again, for contradiction, suppose some woman w is matched with m but m is not her worst possible match in G-S. By definition of possible matchings, there exists another stable matching M' in which w is paired with another man m' whom she likes even less than m. Let w' be the partner of m in M'.
Using men-optimality we see that m prefers w over w', since $(m, w) \in$ G-S.



Known: m prefers w over w'

Known: w prefers m over m'

But then it becomes obvious that (m, w) leads to instability in M' — they both prefer each other over their current partners, contradicting the assumption that M' is stable.
Note that this proof is not specific to the output of G-S; it only uses men-optimality. That is, more generally, **if a paring is men optimal, it is also women pessimal (and vice versa)**. □

The National Residency Matching Program (NRMP) is a prime example of Gale-Shapley's applications. Historically, residency programs faced competitive and chaotic recruitment, especially when there were more residency slots than medical graduates. NRMP was established in the 1950s, initially adopting a non-stable paring system but switched to a propose-and-reject algorithm in 1952, which finally fixed the long-lasting chaos. Traditionally hospital-oriented, the system produced, as we now know, hospital-optimal pairings. Recently the roles reversed, making the system student-optimal, among other improvements.
It was not a typo — NRMP first used this propose-and-reject algorithm in 1952, a decade before Gale and Shapley formally analyzed the algorithm in 1962!

2 Greedy Algorithms

In this section, we investigate the effects short-sighted greed may have in the design of algorithms. Certainly, in many situations, the mindset of myopically optimizing the current situation, i.e., *"picking whatever choice that seems best at the moment and addressing future problems later,"* can fail miserably. Yet, there are situations where such greedy algorithms successfully achieve optimality, and usually this implies that there is something interesting about the structure of the problem itself.

2.1 Interval Selection

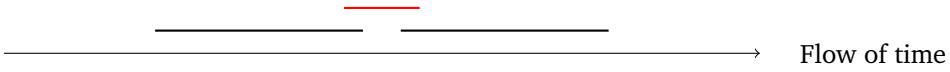
The formulation of the **interval selection** problem is extremely simple:

> Given n intervals with starting points $s(i)$ and finishing points $f(i)$, $i \in [n]$[7], select as many of them as possible but without overlaps.

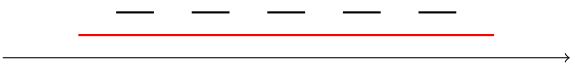
Motivation: there are n presentations, and you want to attend as many as possible, but you clearly cannot attend two presentations that overlap in their times.

A question naturally arises: **how do we define "greedy?"** We consider the following intuitive attempts:

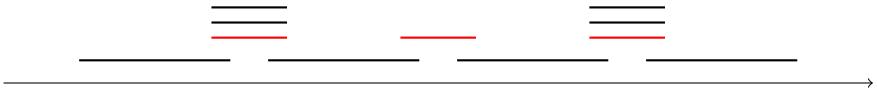
(Attempt 1) *Always pick the shortest remaining interval*. Not optimal, as in the following example we would have picked the one short interval as opposed to the two long intervals. (We can however prove that the result using this strategy will be within a factor of $1/2$ of optimal. We say this is a $1/2$-*approximation algorithm*[8].)



(Attempt 2) *Always pick the interval that starts first*. Terrible, just terrible.



(Attempt 3) *Always pick the interval that overlaps with the fewest number of other intervals.* Also not optimal, as shown in the following example: the middle red interval has fewest overlaps so we pick it, but then in doing so we can only pick a total of 3 intervals, while clearly there is another way to pick 4.



(Attempt 4) *Always pick the interval that finishes first*. We'll show this indeed leads to the optimal solution.

2.1.1 Proof Technique: *Greedy Stays Ahead*

Let us now formulate our aforementioned approach:

[7] An equivalent way of saying $i \in \{1, ..., n\}$, but more concise, so I will adopt this notation. (Very common in machine learning.)
[8] Check CS672 out. It's literally called "approximation algorithms."

Algorithm 2: Greedy Algorithm: Interval Selection

1 **Inputs**: n intervals, with starting points $s(i)$ and finishing points $f(i)$, $i \in [n]$
2 **Notation**: $R :=$ set of remaining intervals, and $A :=$ set of chosen intervals

---

3 sort and relabel the intervals by non-decreasing finishing time $f(i)$, so now $f(1) \leqslant f(2) \leqslant \cdots \leqslant f(n)$
4 initialize $R = [n]$ and $A = \varnothing$
5 **while** R is nonempty **do**
6 let i be a/the remaining interval with earliest finishing time
7 add i to A
8 remove from R all interval i as well as all intervals intersecting it
9 **Return**: A, the set of chosen intervals

Our first observation is that **this algorithm always outputs a valid solution.** No overlaps are possible, since every time we pick an interval, we also delete everything intersecting it.

For runtime:

(1) Sorting on line 3 takes $\mathcal{O}(n \log n)$.

(2) Now that we sorted the intervals by finishing time, it is possible to implement the entire while loop in $\mathcal{O}(n)$:

```
for idx in (1,2,..., n):
   if (sorted and reordered) interval idx starts after when the most recent one finishes:
      pick this one as well
   else: skip it
```

Therefore the total runtime is $\mathcal{O}(n \log n)$.

> ***Proof of greedy optimality***. Intuitively, this greedy algorithm is optimal because it picks intervals that end as early as possible, leaving us more room to pick more intervals.
> Put formally, let $\mathcal{I}(1) < \mathcal{I}(2) < \cdots < \mathcal{I}(k)$ be the first k intervals picked greedily, and $\mathcal{J}(1) < \mathcal{J}(2) < \cdots < \mathcal{J}(k)$ be the first k intervals picked by any other algorithm. We claim that $f(\mathcal{I}(k)) \leqslant f(\mathcal{J}(k))$. **Greedy stays ahead**.
>
> To prove that "greedy stays ahead" we induct on k. Let $\mathcal{I}(\cdot), \mathcal{J}(\cdot)$ be given as described earlier.
> The base case is trivial, as when we only pick one interval, greedy picks precisely the one that ends the earliest. Hence $f(\mathcal{I}(1)) \leqslant f(\mathcal{J}(1))$.
> For the inductive step we assume the claim holds for k, i.e., $f(\mathcal{I}(k)) \leqslant f(\mathcal{J}(k))$. Since $\mathcal{J}$ is a valid solution, the intervals it picks do not overlap. Hence, $\mathcal{J}(k+1)$ starts after $\mathcal{J}(k)$ ends, or $s(\mathcal{J}(k+1)) > f(\mathcal{J}(k))$. By induction hypothesis, $s(\mathcal{J}(k+1)) > f(\mathcal{J}(k)) \geqslant f(\mathcal{I}(k))$. This means that the interval $\mathcal{J}(k+1)$ is a candidate for the $(k+1)^{\text{th}}$ interval in $\mathcal{I}$, since there is no overlap. Therefore, $\mathcal{I}$ either picks $\mathcal{J}(k+1)$ as its next interval, or something even better. Either way, $f(\mathcal{I}(k+1)) \leqslant f(\mathcal{J}(k+1))$. END OF PROOF OF "GREEDY STAYS AHEAD"
>
> Going back to the main claim, we now show that $\mathcal{I}$ selects as many intervals as possible. Suppose not, that greedy selects t intervals in total, but some other solution $\mathcal{J}$ selects $t+1$[9]. "Greedy stays ahead" implies that $f(\mathcal{I}(t)) \leqslant f(\mathcal{J}(t))$, so $\mathcal{J}(t+1)$, so $\mathcal{I}$ could have also included $\mathcal{J}(t+1)$ but it didn't, contradiction. Therefore, $\mathcal{I}$ contains the largest number of intervals, and the proof is complete! □

Extensions and variants:

[9] Assuming $\mathcal{J}$ has $t+1$ suffices. You can replace $t+1$ with larger value, and repeat the remainder of the proof verbatim.

- Different weights assigned to different intervals. We'll solve this using **dynamic programming** later.
- Additional trade-offs: e.g. between time spent and number and value accrued.
- Prerequisites: e.g. determine an optimal course sequence given prerequisites for each course.
- Start/finishing times are not determined but (partially) under our control.
- Partial overlap: e.g. multiple machines.
- You only know interval info real-time, but not ahead of time: *online algorithms*.

2.2 Scheduling with Deadlines

Consider instead a scheduling problem, formulated as follows:

> Given: n jobs, each with a duration $t(i)$ and deadline $d(i)$, $i \in [n]$. Jobs can be performed in any order, but all jobs must be performed.
> Goal: minimize maximum lateness, $\max_i \mathcal{L}(i)$, where $\mathcal{L}(i)$ is the *lateness* of job i, defined by its finishing time minus its deadline (0 if done before deadline).

One key observation is that it never makes sense to split up a job: we simply push the splatted job to the end so it finishes at the same time, and all other jobs would benefit from an earlier finishing time! See the figure below. Therefore, we will assume that the **solution does not split up jobs**.

Another observation is that an optimal solution should never contain any idle times, since we can just delete the idle time and shift everything afterwards forward. Without idle time, the total duration is $\sum t(i)$, constant. Therefore, **the solution is completely characterized by the order in which jobs are done**.

Like before, we consider some natural candidates for what defines greediness:

(Attempt 1) *Longest job first.* Clearly bad if we have two tasks, $(t_1 = 10, d_1 = 10)$ and $(t_2 = 1, d_2 = 1)$.

(Attempt 2) *Shortest job first.* Equally bad: $(t_1 = 10, d_1 = 10)$ and $(t_2 = 1, d_2 = 100)$.

(Attempt 3) *Least "slack" job first, where slackness* $= d(i) - t(i)$. Not optimal: $(t_1 = 10, d_1 = 10), (t_2 = 1, d_2 = 2)$.

(Attempt 4) *Earliest deadline first, regardless of duration.* This works. Since we only need to sort the set of tasks, the runtime is trivially $\mathcal{O}(n \log n)$.

2.2.1 Proof Technique: *Exchange Argument*

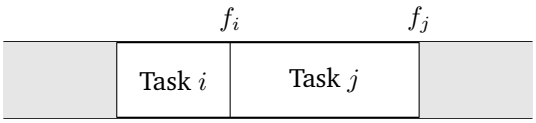
The main idea involved in the **exchange argument** is slightly more complicated than what we saw before: *If a solution were not greedy, we aim to show that we can swap the order of two adjacent jobs in this solution and make the outcome no worse; and the swap makes the solution more similar to greedy, so if we repeat this process, we eventually recover the greedy solution, meaning that greedy is no worse than the original solution.*

Specifically, the intuition behind its optimality for job scheduling is roughly as follows: *If we don't follow this sorting, then certainly some more urgent job was scheduled after some less urgent one. What if we swap them? Well, it's better off for the urgent job as it's completed earlier. Probably worse for the less urgent job, but its induced lateness is not as bad as the original one induced by the more urgent job.*

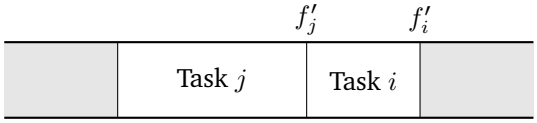
Proof of greedy optimality. WLOG[10]assume that the greedy order is $1 < 2 < \cdots < n$. Let OPT denote an optimal solution. If they are equal then we are done, else, we show that **with one swap, we can make OPT more similar to greedy while retaining its optimality.**
We need to treat swapping with care — ideally we want to swap an adjacent pair, since otherwise we may inadvertently affect the finishing time, and hence lateness, of all other jobs squeezed in-between.

Subclaim. Comparing OPT against greedy, there is an adjacent pair that is out of order.[11]
Certainly, since these two are different, there exist two jobs a and b such that a is less urgent, i.e., $d(a) > d(b)$, but is scheduled before b. We traverse through all jobs between a and b, inclusive, advancing in the scheduled order, and paying attention to the deadline of each task. Since $d(a) > d(b)$, traversing from a to b, there has to be an adjacent pair of tasks where $d(\cdot)$ decreases, and so they are out of order. END OF PROOF OF SUBCLAIM

Having proved its existence, let indices $(k, k+1)$ be one such pair, corresponding to tasks i and j in OPT (so task j finishes earlier than i, or $d_i > d_j$)[12]. Our goal is to show that upon switching tasks i and j in OPT, keeping all others unchanged, the resulting maximum lateness remains optimal.



Original OPT



Swapped OPT

Define f_i, f_j, f'_j, and f'_i as the finishing times of tasks as indicated above. Observe that $f_j = f'_i$. Since the lateness $\mathcal{L}(\cdot)$ of all other tasks remain unchanged, it suffices to show that

$$\max(\mathcal{L}(i)', \mathcal{L}(j)') \leqslant \max(\mathcal{L}(i), \mathcal{L}(j)).$$

This is indeed true:

$$\mathcal{L}(i)' = f'_i - d_i = f_j - d_i \leqslant f_j - d_j = \mathcal{L}(j) \qquad \text{since } f_j = f'_i \text{ and greedy} \Rightarrow d_i > d_j$$
$$\mathcal{L}(j)' = f'_j - d_j \leqslant f_j - d_j = \mathcal{L}(j) \qquad \text{since } f'_j \leqslant f_j.$$

Therefore, optimality is preserved! It remains to show that by repeating this process, we can recover the greedy solution. To see this, note that every time we fix one such pair, the *Kendall tau distance*[13]between OPT and greedy decreases by precisely 1. (This out-of-order adjacent pair is the only out-of-order pair fixed, and no new out-of-order pair, adjacent or not, is created.) We apparently begin with a finite Kendall tau distance, so after finitely many swaps, we are left with none, i.e., we recover greedy. This completes the proof. □

[10]Abbreviation for "without loss of generality."
[11]Alternatively, do an easy induction.
[12]Note that it is (in general) wrong to say $(k, k+1)$ correspond to tasks j and i, respectively!!! All we assumed was that tasks i and j are out of order. They need not retain the same indices as compared to the greedy assignment. For example, T_1 and T_2 are out of order when we compare sequence $\{T_1, T_2, T_3, T_4\}$ against $\{T_3, T_4, T_2, T_1\}$, and they fail to keep their original indices as well!
[13]The total number of pairs (adjacent or not) that are out of order between two orders.

2.3 Minimum Spanning Tree: Kruskal's & Prim's Algorithms

Now we look at a second example of the exchange argument: **constructing a minimum spanning tree (MST)**.

Imagine if there were n cities, and we want to construct roads between some of them. Our goal is simple: make it possible for anyone to drive from any city to another, while keeping the total cost as low as possible. (*Too unrealistically simple — we don't care if two geographically adjacent cities have a short path between them. We are happy with any path, even if it is insanely long. And we don't need backup plans, as all roads are assumed to be perfectly functional at all times. Nevertheless, this oversimplified question bears significance to more complex, realistic problems*)

This problem can be converted into a graph problem using knowledge from CS170:

> Given $V = \{\text{set of cities}\}$, weighted undirected edges $E = \{e = (u, v) : u, v \in V\}$ with distinct[14]weights $w(e) \geqslant 0$, find a subset $E' \subset E$ such that (V, E') is also connected, and the total cost $\sum_{e \in E'} w(e)$ is minimized.

Observe that the optimum solution cannot contain any cycles — for otherwise we can just remove an edge from the cycle and obtain another cheaper, connected graph. What does this mean? We want **trees**! And the optimal solution is called a **minimum spanning tree (MST)**.

The Cut Property

Our ultimate goal involves picking or un-picking edges. Therefore, it would be helpful if there is a notion that determines whether it is "safe" to include a certain edge in a MST. This is where the following comes in handy:

Proposition: "Cut Property"

Given a graph $G = (V, E)$, a **cut** is a partition of the nodes into two sets, S and $S' = V \backslash S$. We denote it by the pair (S, S'). We say an edge e "crosses" a cut (S, S') if one of its endpoints is in S and the other in S'.

The **cut property** states that, if an edge e is the cheapest edge crossing some cut (S, S') then e is in every minimum spanning tree of G.

Proof. We prove by contrapositive. Let $e = (u, v)$ be a cheapest crossing edge of a cut (S, S') with $u \in S, v \in S'$. We show that, for any spanning tree T not including e, it is not a MST.
The idea is that we can insert e into T, replacing a more expensive edge $e' \in T$. Since T is spanning, there certainly exists a path between u and v. We walk down that path, beginning from u. Since we start in S and end up in S', there has to be an edge e' that first "crosses the border," going from S into S'. By assumption, e is cheaper than e'. Therefore, $T \cup \{e\} \backslash \{e'\}$ is cheaper overall, and it is also a spanning tree — $T \cup \{e\}$ is a cycle, and removing an edge e' in the cycle does not break path connectivity. Hence the cost of T is not minimal, and our proof is complete. □

Kruskal's & Prim's Algorithms

Based on the cut property, we may have formed a general idea of how to construct an MST:

- Start with no edges selected.

[14]We will initially assume that all weights are different for convenience, and show later that this assumption is redundant.

- Unless we are done, add an edge that is known to be the cheapest edge across some cut. (But how?)

We first take a look at **Kruskal's Algorithm**:

Algorithm 3: Kruskal's Algorithm

1 **Inputs**: graph $G = (V, E)$ and distinct edge weights $w(e) \geqslant 0$

---

2 sort edges by increasing weight $w(e)$
3 **for** edge e in the order of increasing weights **do**
4 **if** e creates a cycle with already picked edges **then** discard it
5 **else** pick it
6 **Return**: the set of selected edges

Proof of Kruskal optimality. *Main idea: show that (i) output ⊂ MST, and (ii) output is a spanning tree.*

For the first claim, by cut property it suffices to show that each added edge is cheapest for some cut. The main observation is that Kruskal's algorithm builds or grows connected components. (*If the new edge is all by itself builds one; else it extends an existing one.*) In either case, we can view the addition of edge e as a connection between two components, C_1 and C_2. (*In the "build" case, C_1, C_2 are both singletons; in the "grow" case, C_1 is the existing component and C_2 a singleton.*) Since we are picking the cheapest possible remaining edge connecting two components (*e won't connect two components if it induces a cycle*), e is in fact the cheapest edge across cuts (C_1, C_1') [and (C_2, C_2')]. Therefore $e \in$ MST. Hence the output ⊂ MST.

For the second claim, since the MST has $n - 1$ edges[15], it suffices to show that the output, a subset of MST, also has $n - 1$ edges. This is true, because if the output had fewer edges, it would not be connected, so there is a partition (S, S') with no selected edge crossing. But the original graph is connected, so there are unselected edges that cross the partition. Kruskal would have picked one of these edges. Contradiction. □

Now we look at the (possibly) simpler, more explicit **Prim's Algorithm**:

Algorithm 4: Prim's Algorithm

1 **Inputs**: graph $G = (V, E)$ and distinct edge weights $w(e) \geqslant 0$.

---

2 start with $T = \{\}$, and $S := \{s\}$ where s is an arbitrary start index
3 **while** $S \neq V$ (not all nodes are covered) **do**
4 find the cheapest edge $e = (u, v)$ crossing (S, S')
5 $T \leftarrow T \cup \{e\}$, $S \leftarrow S \cup \{v\}$
6 **Return**: T, the set of selected edges

Proof of Prim optimality. Line 4 explicitly guarantees that each added edge is the cheapest edge across the cut (S, S') at that iteration, so output ⊂ MST.

Further, each iteration of the loop adds precisely one edge, and the loop terminates after $n - 1$ iterations, so output contains $n - 1$ edges. Done! □

2.3.1 (Optional) The Union-Find Data Structure

We first take a look at the complexity of Prim's algorithm. The loop is repeated for $\Theta(n)$ iterations. Inside the loop, we can naively implement brute force search, taking $\Theta(m)$ time each, resulting in a total naive complexity of

[15] Recall from CS170 that a tree with n nodes has $n - 1$ edges.

$\Theta(mn)$.

But apparently we can adopt better approaches. For example, we can use a min heap containing all nodes that are not in S of current iteration, and for each node, we store the minimum cost of any edges connecting it to S. Figuring out which node to add is simply a matter of finding the minimum in the min heap. Of course, then we might need to update the values of this newly added node's neighbors.

Observe that each edge leads to at most one update of a value in the heap. In a binary heap, updating takes $\mathcal{O}(\log n)$ time, and there are m total edges, so the total runtime is $\mathcal{O}(m \log n)$. Using Fibonacci heaps this improves to $\mathcal{O}(m + n \log n)$.

Kruskal's algorithm, on the other hand, is much harder to analyze. Sorting takes $\Theta(m \log m)$ as we all know, and the `for` loop is repeated $\Theta(m)$ times. But what about the code inside? How do we detect a cycle?

One intuitive way is to check whether the two endpoints of e are already connected before we attempt to add e. If yes, then adding e creates a cycle; else no. The naive way to test this is using BFS, which takes $\Theta(n)$, since Kruskal's algorithm picks at most $n - 1$ edges. This leads to a total runtime of $\Theta(m \log m + mn) = \Theta(mn)$.

(*The following content is out of scope and optional.*) Is it possible to further improve this runtime? The answer is yes. With some smart implementations we can essentially make the $\mathcal{O}(m \log m)$ from sorting the new bottleneck. For a more detailed walkthrough, refer to section 4.6 of our textbook, *Algorithm Design* by Kleinberg and Tardos.

Methods: to implement a `Union-Find` data structure, we need the following functionalities:

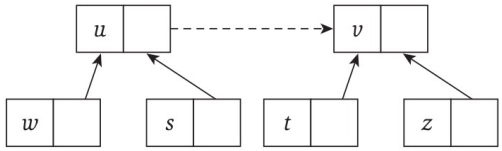
- `init`(S): initializes the data structure. We want to initialize when Kruskal starts, i.e. all nodes are in separate sets. We do it in $\mathcal{O}(n)$.
- `find`(u): returns the name of the component containing u. Specifically, in our implementation, we will use one particular element of each component as a representative. We show that this can be done easily in $\mathcal{O}(\log n)$ time, and with some smart optimization, in "almost" linear time.
- `union`(u, v): merges components represented by u and v into a larger component. We show that this can be done in $\mathcal{O}(1)$ while also keeping `find()` fast.

Basic idea: in `Union-Find`, each struct typically stores a node, as well as a pointer to a parent, which helps us identify the component the node lies in. Each component will assume a tree-like structure, with its root being the representative.

As we call `init()`, we create n single trees, where each tree consists of one node only, and that node serves as the representative of that component.

Core Operations: `find()` and `union()`

- `find`(u) is used to determine the root of a component to which node u belongs. Intuitively, u, we traverse through the tree upward, until we reach the root, the representative of the component, and claim that u belongs to the same component as the root. Apparently this can be achieved in $\mathcal{O}(\log n)$ time with heaps.
- `union`(u, v) merges two sets into one. Simply point u to v (or v to u). Easy $\mathcal{O}(1)$. We have two trees of size 3 in the following example, represented by u and v. We merge them into one, keeping v as the true representative.



Source: KT figure 4.12

Implementing Kruskal's Algorithm

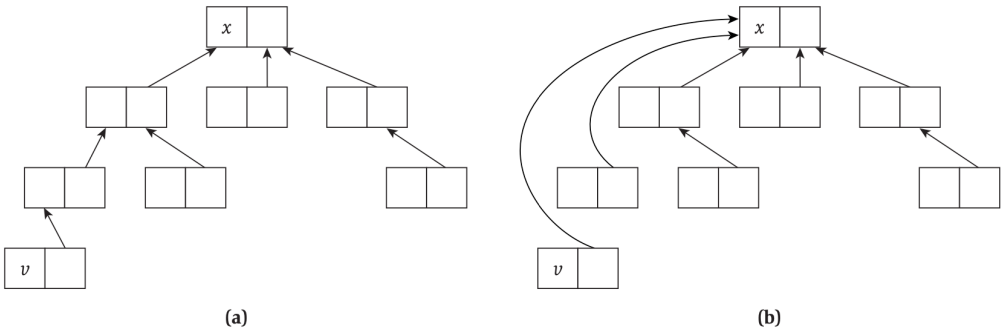
We first sort the edges using $\mathcal{O}(m \log m)$ time. In each of the following m iterations, each time we consider an edge $e = (u, v)$, we compute `find(`u`)` and `find(`v`)`. They belong to the same component if and only if the results agree, in which case we do nothing. Else we call `union()`, passing in the two roots, which takes $\mathcal{O}(1)$ time. We invoke `find()` at most $2m$ times since there are m iterations, but we add a total of $n - 1$ edges, so we only call `union()` $n - 1$ times. With `find()` being $\mathcal{O}(\log n)$ and `union()` $\mathcal{O}(1)$, the total runtime after sorting is $\mathcal{O}(m \log n)$.

It remains to notice that since $m \leqslant n^2$, $\log m \leqslant 2 \log n$, so the complexity of sorting can be rewritten as $\mathcal{O}(m \log m) = \mathcal{O}(m \log n)$. And we see that Kruskal's algorithm can indeed be implemented in $\mathcal{O}(m \log n)$ time.

Even Faster `find()`?

Now we show a clever trick to significantly improve the runtime of `find()`. Consider a very tall tree with root r, and we called `find()` on its leaf u. We traversed all the way from u to r and obtained the answer we want — u belongs to "component r." What if for some reason we need to call `find(`u`)` again? Or what if we need to soon call `find(`v`)`, where v is one of the nodes along our way from u to r? Both would have been $\mathcal{O}(1)$ had we somehow saved the root info somehow.

This is precisely where the optimization comes in: path compression. Each time we do a hard traverse, we compress the path by resetting all pointers along the path to point to the current root. Double the work for the initial function call, but $\mathcal{O}(1)$ for all future calls involving any node along this path. Refer to the following diagram (node names are different from what I used but you get the idea).



Source: KT figure 4.13

Tarjan and Leeuwen showed in 1984 [link] that a sequence of n `find()` operations has its runtime bounded from above by $\mathcal{O}(n\alpha(n))$, where $\alpha(n)$ is the *inverse Ackermann function*, an extremely slow-growing function that can be viewed almost as constant.[16] And thus, `find()` can be implemented in almost linear time!

Unfortunately, this optimization does not help with the overall runtime of Kruskal, since sorting takes $\mathcal{O}(m \log m) = \mathcal{O}(m \log n)$ regardless. But still, quite cool idea to present.

[16] $\alpha(\cdot)$ is piecewise monotonic integer-valued constant, with $\alpha(2) = 1, \alpha(2^2) = \alpha(4) = 2, \alpha(2^4) = \alpha(16) = 3, \alpha(2^{16}) = \alpha(65536) = 4$, and $\alpha(2^{65536}) = 5$. Basically in real life $\alpha(\cdot) \leqslant 4$.

3 Divide and Conquer Algorithms

In this section we look at an algorithmic technique called **Divide and Conquer**. The high-level idea is simple:

(1) Take a problem instance I of size n.

(2) Divide the problem into smaller sub-instances $I(1), I(2), \cdots, I(k)$. Often times $k = 2$ (but not always), and each subproblem has the same size n/k (again, not always).

(3) Recursively solve for smaller problem instances.

(4) If a problem is simple/small enough, solve it directly without further breaking it down.

(5) Do some post-processing, combining the solutions of smaller problems into a solution of the whole problem.

3.1 MergeSort: a First Example

As a first example, we reexamine the `MergeSort` algorithm, which hopefully sounds familiar:

Algorithm 5: MergeSort

```
1  Function MergeSort(array a[], int L, int R):
2      if R = L then
3          do nothing
4      else
5          let m ← (R + L)/2, rounded down
6          call MergeSort(a, L, m)
7          call MergeSort(a, m+1, R)
8          call Merge(a, L, m, R)
9  Function Merge(array a[], int L, int m, int R):
10     let i ← L, j ← m + 1, k ← 0
11     let b ← a new array of size R − L + 1, zero-indexed
12     while i ⩽ m or j ⩽ R do
13         if j > R or (i ⩽ m and a[i] ⩽ a[j]) then
14             set b[k] ← a[i], i ← i + 1, k ← k + 1
15         else
16             set b[k] ← a[j], j ← j + 1, k ← k + 1
17     for i = L, L + 1, ⋯, R do
18         set a[i] ← b[i − k]
```

Before proving that `MergeSort` is correct (i.e. (i) the algorithm terminates, and (ii) the array is correctly sorted upon termination, and (iii) the elements contained in the array remain the same), we assume the correctness of the helper function `Merge()`, the proof of which involves an easy induction on k or $i + j$.

> ***Proof of `MergeSort` correctness.*** We use strong induction on $n = R - L + 1$, the size of the array.
> In the base case where $n = 1$ or equivalently $R = L$, the algorithm does nothing, so it is trivially correct.
> Now we assume the algorithm functions correctly given any array of size $\leqslant n - 1$. On lines 6 and 7, we called `MergeSort()` twice, each time feeding a even smaller subarray, so by induction hypothesis, both calls terminate, and both subarrays are sorted and contain the same elements as before. Assuming `Merge()` does its job, the final

output $a[L:R]$ is also sorted and contains the same elements. □

Now comes the interesting part — **what is the runtime of this recursive algorithm**?
Let us use $T(n)$ to denote the complexity when the size of the input is n. In this example, it refers to the size of the array, or $L - R + 1$. We observe the following:

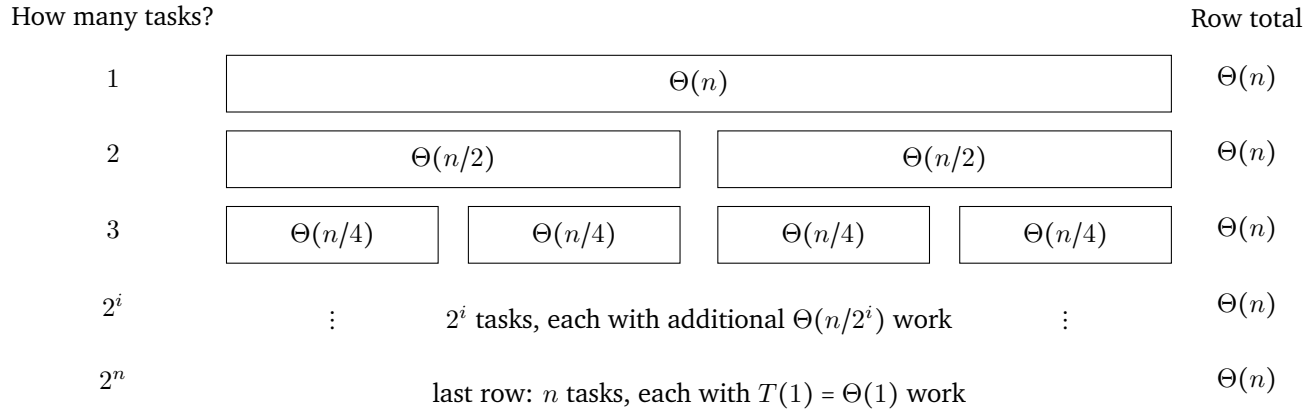
Algorithm 5: MergeSort main function

```
1 Function MergeSort(array a[], int L, int R):
2     if R = L then
3         do nothing                                   T(1) = Θ(1)
4     else
5         let m ← (R + L)/2, rounded down              Θ(1)
6         call MergeSort(a, L, m)                      T(n/2)
7         call MergeSort(a, m + 1, R)                  T(n/2)
8         call Merge(a, L, m, R)                       Θ(R − L) = Θ(n)
```

From this we conclude

$$T(1) = \Theta(1) \qquad T(n) = 2T(n/2) + \Theta(n).$$

Now let us visualize how a larger task gets broken down into smaller subtasks in a binary fashion:



There are $\log n$ rows in total, and in each row we have $\Theta(n)$ work to do. Hence `MergeSort()`'s recursive complexity satisfies $T(n) = \Theta(n \log n)$, agreeing with our prior knowledge.

3.2 The Master Theorem

In the above example, we noticed that by breaking the recurrent task into a "tower" of smaller tasks, we can compute the total work $T(n)$ by adding up the work accumulated within each row, or each "level" in this tower. There are three certain scenarios that particularly pique our interest:

(1) If the original big task pales in comparison with the countless small tasks, namely, clones of $T(1)$ dominate;

(2) If the original big task is really where all (well, most) the heavylifting is at, namely, $T(n)$ dominates; or

(3) The subdivision into subtasks just happens to be at the equilibrium, like the example above, that keeps the workload at each level uniform.

I feel guilty to spring the Master theorem on you right here, but I don't think more context would make a difference...

Theorem: Master Theorem

Let $a \geqslant 1$ and $b > 1$. Assume some recursion relation complexity satisfies

$$T(n) = aT(n/b) + f(n) \qquad T(1) = \Theta(1).$$

For notational simplicity define the critical exponent $c_0 := \log_b a$.

(MT1) (*Subproblems dominate, leaf-heavy*) If $f(n) = \mathcal{O}(n^c)$ for some $c < c_0$[17] then $T(n) = \Theta(n^{c_0})$.

(MT2) (*Balanced*) If $f(n) = \Theta(n^{c_0})$ then $T(n) = \Theta(n^{c_0} \log n)$.

(MT3) (*Main problem dominates, root-heavy*) If $f(n) = \Omega(n^c)$ for some $c > c_0$, and the regularity condition

$$af(n/b) \leqslant kf(n) \qquad \text{for some } k < 1^{18} \text{ and all sufficiently large } n,$$

is satisfied, then $T(n) = \Theta(f(n))$.

3.2.1 (Optional) Understanding the Master Theorem

Let us revisit the recurrence relation we encountered in `MergeSort`:

$$T(n) = 2T(n/2) + \Theta(n) \implies \begin{cases} a = 2 \\ b = 2 \\ c_0 = \log_b a = 1 \\ f(n) = \Theta(n^1). \end{cases}$$

It follows that (MT2) is satisfied, and indeed, from the diagram we saw earlier, each row is assigned the same amount of work, namely $\Theta(n)$.

Now, suppose someone wants to break (MT2) and make (MT1) happen. **How can an adversary modify the parameters** to achieve this goal? On a high-level idea, they want to make the subproblems overwhelm the original one. Well, a few ways.

(1) Allow each problem to "clone" themselves even more when subdivided. Certainly, this will make the lower layers of this tower/tree much more populated, and consequently more task to be done. *How?* Simple. Increase a, the "clone factor." *Why?* With a increased, now critical exponent c_0 increases, breaking (MT2) and leading to (MT1).

(2) Another easy way is to "divide less and keep more" when subdivided. Decrease b so n/b is not much smaller than n. As an extreme example consider $b = 1.01$, so $n/b \approx 0.99$. With a still $= 2$, we essentially doubled the number of problem instances without reducing either of them to a significantly smaller size. It would be no surprise that the further down we traverse through the tower/tree, the more work to be done. Same reason as above: $c_0 = \log_b a$ increased.

[17] Equivalent to the definition involving ϵ: if $f(n) = \mathcal{O}(n^{c_0 - \epsilon})$ for some $\epsilon > 0$. I personally don't feel like using ϵ-δ language in an introductory CS course, despite its rigor. Leave it to more abstract/rigorous/advanced CS/math classes, I guess.

[18] Equivalently: there exist $\delta > 0$ (probably very small) and $n_0 > 0$ (probably very large) such that $af(n/b) \leqslant (1 - \delta)f(n)$ for all $n > n_0$.

(3) Not as intuitive as the previous two, but we can also play around with $f(n)$, making it sublinear to achieve the same result. Suppose $f(n) = \Theta(n^{1-\epsilon})$ for some $\epsilon > 0$. Then, on row j, the total induced work comes from 2^j tasks, each with $\Theta((n/2^j)^{1-\epsilon})$ work. Time for some math:

$$2^j \cdot \Theta((n/2^j)^{1-\epsilon}) = 2^j 2^{-j(1-\epsilon)} \Theta(n^{1-\epsilon}) = 2^{j\epsilon} \Theta(n^{1-\epsilon}).$$

Viewing this as a function of j, row depth, we essentially see that the workload grows exponentially as we walk down the tower/tree.

It follows immediately that we can do the opposite things in order to allow $T(n)$ dominate over copies of $T(1)$: we can reduce a, increase b, or making $f(n)$ superlinear.

What about (MT3)'s regularity condition???

Tough question. I myself haven't been able to find a satisfying explanation online – most of them simply give examples where we cannot directly apply (MT3) because of a violation of the regularity condition, but fail to provide any intuition nor an explicit inequality between $T(n)$ and $\Theta(f(n))$.

Regardless, let us first prove (MT3) and see which steps might fail without the regularity condition.

Proof of (MT3). To show $T(n) = \Theta(f(n))$ we need to show $T(n) = \Omega(f(n))$ and $T(n) = \mathcal{O}(f(n))$ simultaneously. Observe, from the tower diagram, that we can get a closed-form solution for $T(n)$:

$$T(n) = \sum_{\text{rows}} \text{work done by each row} = \sum_{i=0}^{\log_b n} a^i f(n/b^i) + \underbrace{\mathcal{O}(n^{\log_b a})\Theta(1)}_{\text{last layer (leaves)}}.$$

Setting $i = 0$ we see that $f(n)$ is part of the sum, so $T(n) = \Omega(f(n))$ trivially holds. Conversely, observe that by assumption, for some $k < 1$ and sufficiently large n, as well as each j,

$$\begin{aligned} af(n/b) \leqslant kf(n) &\Rightarrow f(n/b) \leqslant (k/a)f(n) \\ &\Rightarrow af(n/b^2) \leqslant k(k/a)f(n) \Rightarrow f(n/b^2) \leqslant (k/a)^2 f(n) \\ &\Rightarrow \cdots \Rightarrow f(n/b^j) \leqslant (k/a)^j f(n) \Rightarrow a^j f(n/b^j) \leqslant k^j f(n). \end{aligned} \tag{Δ}$$

Therefore,

$$T(n) = \sum_{i=0}^{\log_b n} a^i f(n/b^i) + \mathcal{O}(n^{\log_b a}) \leqslant \sum_{i=0}^{\log_b n} k^i f(n) + \mathcal{O}(n^{\log_b a}) \leqslant \frac{f(n)}{1-k} + \mathcal{O}(n^{\log_b a}) = \mathcal{O}(f(n)).$$

Without regularity condition, we cannot assume the chain inequalities in (Δ), and if $k \geqslant 1$ the series $\sum k^i$ diverges, so the red term no longer serves as a valid upper bound. □

In fact, the conditions in (MT3) are slightly redundant: regularity condition implies that $f(n) = \Omega(n^{c_0+\epsilon})$. To see this, we can rewrite (Δ) as $f(b^j n) \geqslant (a/k)^j f(n)$. Letting $n = 1$ we have

$$f(b^j) \geqslant (a/k)^j f(1),$$

and letting $j = \log_b n$ so that $b^j = n$, we obtain

$$f(n) \geqslant (a/k)^{\log_b n} f(1) = n^{\log_b \overbrace{(a/k)}^{>a}} f(1) = n^{\log_b a+\epsilon} \underbrace{f(1)}_{\text{const}} = \Omega(n^{c_0+\epsilon}).$$

An Explicit Example Where (MT3) Fails Without Regularity Condition

I tried to directly come up with a counterexample violating either (Δ) or the convergence of $\sum k^i$ but to no avail, but the example found here gets the job done.

Let $a = 1$ and $b = 2$, so the recurrence relation is given by

$$T(n) = T(n/2) + f(n) \qquad \text{or equivalently} \qquad T(2^n) = \sum_{k=0}^{n} f(2^k).$$

Here $c_0 = \log_2 1 = 0$, so to invoke (MT3) we require $f(n) = \Omega(n^\epsilon)$ for some $\epsilon > 0$. Consider

$$f(2^n) := 2^{2^{\lfloor \log_2 n \rfloor}} > 2^{2^{(\log_2 n)-1}} = 2^{n/2} = (2^n)^{1/2}$$

so in this case letting $\epsilon = 1/2$ suffices. Now given integer m we consider $n = 2^{m+1} - 1$. Then

$$f(2^n) = 2^{2^{\lfloor \log_2 (2^{m+1}-1) \rfloor}} = 2^{2^m},$$

but

$$T(2^n) = \sum_{k=0}^{2^{m+1}-1} f(2^k) = \sum_{j=0}^{m} \sum_{t=2^j}^{2^{j+1}-1} f(2^{2^j}) = \sum_{j=0}^{m} \sum_{t=2^j}^{2^{j+1}-1} 2^{2^j} = \sum_{j=0}^{m} 2^j 2^{2^j} = \sum_{j=0}^{m} 2^{j+2^j} = \Theta(2^{m+2^m}).$$

3.3 Binary Integer Multiplications

Consider two n-bit binary numbers x, y. How do we efficiently compute their product $x \cdot y$? Using brute force, multiplying x with each digit of y to obtain partial products, shifting and adding the final output together, we have a naive runtime of $\Theta(n^2)$. Can it be faster?

Divide and conquer, shall we? We split x and y into four $n/2$-bit binary numbers x^+, x^-, y^+, and y^-, where x^+, y^+ consist of the top $n/2$ MSBs of x and y, and x^-, y^- the LSBs[19]. Then $x = 2^{n/2} \cdot x^+ + x^-$ and $y = 2^{n/2} \cdot y^+ + y^-$. Then

$$x \cdot y = (x^+ \cdot 2^{n/2} + x^-)(y^+ \cdot 2^{n/2} + y^-) = 2^n x^+ y^+ + 2^{n/2} x^- y^+ + 2^{n/2} x^+ y^- + x^- y^-.$$

From this we spot 4 recursive calls, each with size $n/2$. For post-processing: multiplying by powers of 2 is equivalent to shifting, which takes $\mathcal{O}(n)$ time, and adding takes $\mathcal{O}(n)$ time as well. Therefore the recurrence is given by

$$T(n) = 4T(n/2) + \mathcal{O}(n).$$

Unfortunately by (MT1) we are not seeing any asymptotic improvement here: $T(n) = \Theta(n^{\log_2 4}) = \Theta(n^2)$. The problem is that our four-way branching is recursively creating too much work.

[19] We don't worry about whether n is odd here. Take ceiling or floor of $n/2$ if necessary, but the general idea remains the same.

3.3.1 Karatsuba's Algorithm

In 1960, Andrey Kolmogorov[20] conjectured that any integer multiplciation task would require $\Omega(n^2)$ elementary operations, and he brought this up in a seminar. It took Anatoly Karatsuba, then a 23-year-student, only one week to disprove this conjecture, reducing the complexity of integer multiplication to $\mathcal{O}(n^{\log_2 3}) \approx \mathcal{O}(n^{1.59})$. As a comic relief, Kolmogorov was excited, and proceeded to publish a paper under the name of Karatsuba but never told him. Karatsuba would later find out about this himself.

The difference between Karatsuba's algorithm and the previous divide and conquer can be summed up in a few words: **the coefficients for $2^{n/2}$ can be computed using one additional recursive call, not two**:

$$x^+y^- + x^-y^+ = (x^+ + x^-)(y^+ + y^-) - x^+y^+ - x^-y^-,$$

where we already have access to the latter two terms. Therefore we only need to compute the $n/2$-bit multiplication of $(x^+ + x^-)$ and $(y^+ + y^-)$, along with some pre- and post-processing. To put formally:

Algorithm 6: Karatsuba's Algorithm for Integer Multiplication

1 **Inputs**: n-bit binary integers x and y

---

2 **if** $n = 1$ (base case) **then**

3 multiply directly and **return**

4 let $n/2$-bit x^+, x^- be such that $x = 2^{n/2} \cdot x^+ + x^-$, similarly for y^+, y^-

5 let $a \leftarrow x^+ \cdot y^+, b \leftarrow x^- \cdot y^-, c \leftarrow (x^+ + x^-)(y^+ + y^-)$ `// three recursive calls`

6 **Return**: $2^n \cdot a + 2^{n/2} \cdot (c - a - b) + b$

Our updated recurrence relation is given by $T(n) = 3T(n/2) + \Theta(n)$, which by (MT1) again gives $T(n) = \Theta(n^{\log_2 3})$.

3.4 The 2D "Closest Pair" Problem

So far, both `MergeSort` and integer multiplication involve relatively simple techniques to merge solutions of the subproblems. We will now look at a problem that is more challenging in this regard.

> Given n points, $\{p_i\}_{i=1}^n = \{(x_i, y_i)\}_{i=1}^n$, find the pair of closest points w.r.t.[21]Euclidean norm[22].

Apparently, if we just brute force try all pairs of points, we obtain a solution in $\mathcal{O}(n^2)$ time. But we aim for a faster algorithm using divide and conquer:

(1) (Base case) If we just have one or two points, easy.

(2) (Divide) Divide the region into left and right halves. Recursively compute the closest pair on the left, and the closest pair on the right. Maybe the closer pair between these two is in fact the closest pair overall?

(3) (Conquer) **Do something about pairs with one point on the left and one on the right**. Hard part.

[20] 20th century Soviet mathematical giant who made major contributions to literally every branch of mathematics. And consequently you'll see his name literally everywhere in graduate textbooks.

[21] "With respect to."

[22] Defined by $d((x_1, y_1), (x_2, y_2)) = ((x_1 - x_2)^2 + (y_1 - y_2)^2)^{1/2}$.

We can't just brute force the "conquer" part either by trying all left-right pairs: doing so involves at least $(n/2)^2$ steps. No improvement compared to overall brute force.

What can we deduce, based on our goal for the "conquer" step?

- Anything that's too (horizontally) far from the (vertical) dividing left-right dividing line need not be considered. **Therefore, we can assume that both endpoints of our left-right pair have to be within a narrow vertical strip of a certain width δ.**
- We need a systematic way to investigate the points within this strip. Since we are dealing with 2-dimensional points, ideally we'd like to define some kind of order. Simple examples: look at one component only.
- And we need to be "smart" as we inspect these points, making use of all the constraints we have, or else a brute force inspection might take quadratic time. For example, in the first bullet point we noted that a candidate pair cannot be too distant horizontally. Here we note that **within the strip, they cannot be too distant vertically either**.

We now propose the following recursive algorithm:

Algorithm 7: Finding the Closest Pair of Points

1 **Inputs**: a set of n points, $\{p_i\}_{i=1}^n = \{(x_i, y_i)\}_{i=1}^n$
2 **Additional pre-condition**: twp copies of $\{p_i\}$, one sorted in x-component, the other in y.

---

3 `// handle base cases: if one point, undefined; if two points, return distance`
4 let m be the middle (e.g. median) x-coordinate
5 partition points into L and R based on m, and invoke two recursive calls on them
6 let $\delta \leftarrow$ the smaller returned distance from the two sub-calls
7 let $S \leftarrow \{(x_i, y_i) : |x_i - m| \leqslant \delta\}$ `// middle vertical strip`
8 `// check close pairs with one endpoint in` L `and the other in` R
9 index and sort points in S by y-coordinates, in ascending order
10 $\delta_{\min} \leftarrow \delta$
11 **for** each point $i \in S$ **do**
12 **for** each j such that $j \geqslant i+1$ and $y(j) < {}^{23}y(i) + \delta$ **do**
13 compute $d(p(i), p(j))$
14 **if** $d(p(i), p(j)) < \delta$ **then** $\delta_{\min} \leftarrow d(p(i), p(j))$
15 **Return**: $\delta_{\min}$ and/or the pair inside the δ-strip achieving this distance

Correctness proof. *We will give a brief proof showing that the above algorithm is correct, but it is not our main focus here. We use induction on array size n.*

The base cases are... trivial based on the comment given on line 3. For the inductive step, we assume by IH that the recursive calls on the left and right halves return the closest pairs in those sets, respectively, so δ on line 6 is correct. It remains to prove that the `for` loop finds the minimum pairwise distance inside the δ-strip. But this is clear based on our straightforward loop structure.

Therefore the algorithm is correct, as it either directly returns a result from one of its recursive sub-calls or a

[23] Both $<$ and $\leqslant$ are fine. One overrides results from recursive calls if it finds an equally good (but not strictly better) pair inside the strip; the other doesn't.

new pair found within the δ-strip proven to be even better. □

Runtime Analysis

For now, we will assume that we automatically obtain sorted copies of $\{p_i\}_{i=1}^n$ by both coordinates, for free. (Not S - we still have to sort that later.)

(line 4) Finding the median x-coordinate takes at most $\mathcal{O}(n)$.

(line 6) Computing δ takes $\mathcal{O}(1)$ time once both values have been returned.

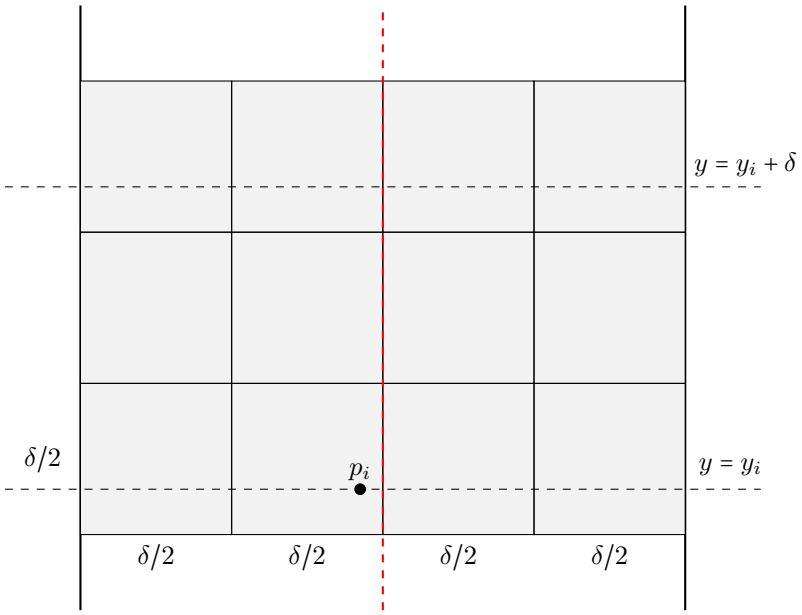
(line 7) Computing $\mathcal{S}$ takes at most $\mathcal{O}(n)$: one scan through the x-sorted $\{p_i\}$ suffices.

(inside the `for` loop) This is the most challenging part, but **we claim that the entire loop can be completed in $\Theta(n)$ time**. In particular, line 11 is looped at most n times, and we prove that for each point $i \in S$, their are at most[24] 12 distinct j's satisfying the condition of the inner `for` loop on line 12.

Proof. This is still a crude upper bound. Try improving it yourself if interested. At least I can think of an easy way to make this 8.

Let us begin with a point $p_i = (x_i, y_i) \in S$. It is clear that we only need to consider an area containing p_i such that its boundaries are not too far from p_i. Consider a rectangular area with width 2δ (same width as the δ-strip) and height $3\delta/2$. Let the bottom be somewhere below p_i, say by the line $x = y_i - \delta/3$. Our goal is to prove that **there can be at most 12 points stuffed inside this area**.

We further subdivide our rectangular region into 12 small squares with side lengths $\delta/2$, as shown below.



We want to put as many points into the shaded area as possible. What are the constraints? Well, from the definition of δ, the distance between any two points in the left half must be $\geqslant \delta$, and the same applies to the right half. And that's precisely why we have these little squares of side length $\delta/2$. If we put two dots inside the same square, their distance cannot be larger than the diagonal length, $(\sqrt{2}/2)\delta < \delta$. Therefore, to abide by the constraint, **each square can contain at most one point**. Convince yourself that given any point (x, y), the target region $[m-\delta, m+\delta] \times [y, y+\delta]$ (the intersection of area between the two dashed lines

[24] The choice 12 is somewhat arbitrary. The textbook chose 15. The key takeaway is that the second loop can be bounded by an absolute constant. Its exact value is of little interest to us, at least.

> and the δ-strip) can always be covered by 12 adjacent squares (hint: check width and height). Therefore in each iteration of line 11, the second `for` loop on line 12 is satisfied at most 12 times, and each time when this happens we perform $\Theta(1)$ work. Since line 11 is looped for at most n times, the total runtime of the outer `for` loop is $\Theta(n)$. □

Combining all of these, we see that the recurrence is given by

$$T(n) = 2T(n/2) + \mathcal{O}(n)$$

which (is identical to that of `MergeSort` and) has complexity $\mathcal{O}(n \log n)$.

Finally, to address the sorting issue mentioned at the top of the runtime analysis, we can pre-sort the entire array twice, once for x-coordinates, and once for y. Once we know the sets of points to deal with, we simply copy the corresponding entries. $\mathcal{O}(n \log n)$ for sorting and $\mathcal{O}(n)$ for copying. They clearly don't affect the overall complexity $\mathcal{O}(n \log n)$.

4 Dynamic Programming

Dynamic Programming is not dynamic, nor is it about programming.

— Professor David Kempe

Dynamic Programming (DP) is a useful computational technique that involves breaking complex problems into simpler sub-problems (*does this remind you of something we just learned?*), with the additional ability of storing and reusing past solutions, which turns out to be an extremely powerful optimization in certain problems. When using DP, you secretly ask yourself the following question:

> In order to do *this*, what do I need to do between the previous step and now?

4.1 Memoization

DP is closely related to recursion, as well as exhaustive search and backtracking. The main difference is that **it avoids unnecessary re-computation.**

Consider, for example, the famous recursion problem of computing Fibonacci numbers. We define `Fib(n)` to be such that it returns 1 if $n = 0$ or 1, and returns `Fib(n-1) + Fib(n-2)` otherwise. Runtime-wise, we have

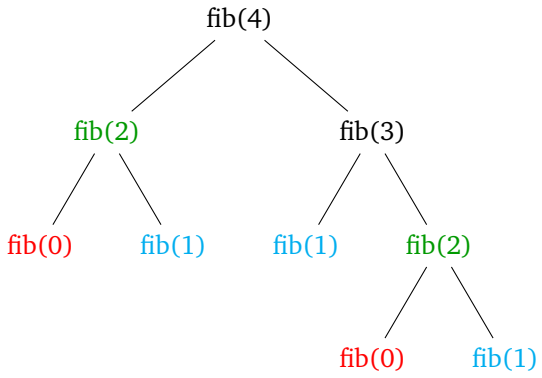
$$T(n) = T(n-1) + T(n-2) + \mathcal{O}(1) \qquad T(0) = T(1) = \mathcal{O}(1).$$

Dropping the $\mathcal{O}(1)$ work, we see $T(n) \geqslant T(n-1) + T(n-2)$, which is the Fibonacci recurrent itself. You can use an easy induction to prove that

$$\texttt{Fib(n+1)} = \frac{1}{\sqrt{5}}\left(\frac{1+\sqrt{5}}{2}\right)^n - \frac{1}{\sqrt{5}}\left(\frac{1-\sqrt{5}}{2}\right)^n$$

which, as n gets larger, approximates $(1/\sqrt{5})(((1+\sqrt{5})/2))^n \approx 1.618^n/\sqrt{5}$, since the second term vanishes because $|(1-\sqrt{5})/2| \approx 0.618 < 1$. From this, we see that **computing Fibonacci using recursion takes exponential time $\mathcal{O}(1.618^n)$**, which is... certainly really bad.

What's the problem? Too many re-computations. Suppose you call `Fib(4)`. What happens? Well, we see a lot of repetitions, even for an argument as small as 4 — the lower levels in the computation tree consist of many repetitions, and they increase exponentially as the depth increases.



How do we fix this? We maintain an array of size $n+1$ (0-indexed, from 0 to n) when computing `Fib(n)`, and we replace recursive calls with direct access to corresponding indices in the array. This technique is called **memoization**.

Obviously, the new runtime is $\mathcal{O}(n)$. So much better.

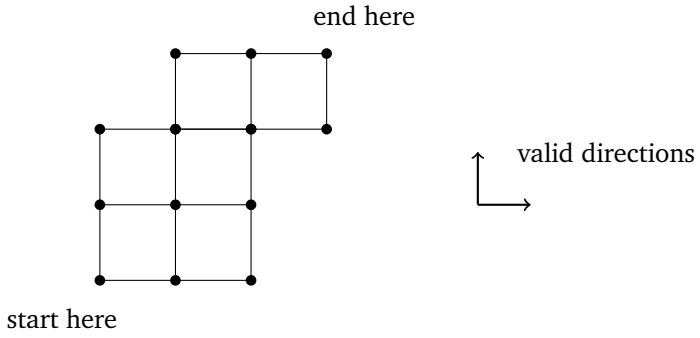
Algorithm 8: Fibonacci with DP

```
Function Fibonacci(n):
    allocate integer array F[n + 1], zero-indexed
    F[0] = F[1] = 1
    for i = 2, 3, ···, n do
        F[i] = F[i − 1] + F[i − 2] // only array lookups, no recursive calls
    return F[n]
```

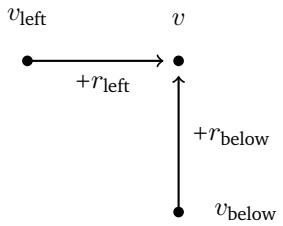
4.2 (Optional) Intuition Behind DP

"In order to do *this*, what do I need to do between the previous step and now?"

Let us consider a very simple problem. You have a grid, where each edge has a certain amount of reward associated with it. You start from the bottom-left vertex, and your goal is to move to the top-right vertex. You can only move up or right, not down or left. How do you maximize your reward, among all valid paths?



Well, how do we arrive at a vertex? We either walked upward or rightward to reach it. That is, we were at either the vertex to its left, or the vertex below it, at the previous step.



Let $f(\cdot)$ denote the optimal reward function. If we somehow already know the optimal reward obtainable at the two predecessor vertices, i.e. $f(v_{\text{left}})$ and $f(v_{\text{below}})$, then it becomes clear that one of them eventually plays a role in $f(v)$. What is the maximum possible reward at v, given that the penultimate vertex we visit is v_{left}? Well, the last edge has reward r_{left}, so the answer is $f(v_{\text{left}}) + r_{\text{left}}$. Here, we implicitly used an extremely important insight of DP: **any optimal solution to the current problem must contain optimal solutions to the sub-problems involved**. Similarly, if we arrive at v from v_{below}, the maximum possible reward is $f(v_{\text{below}}) + r_{\text{below}}$. Since we want to maximize the reward, it naturally follows that

$$f(v) = \max\{f(v_{\text{left}}) + r_{\text{left}}, f(v_{\text{below}}) + r_{\text{below}}\}.$$

Of course, if v only has one valid processor (for example, if it's on the bottom row so it can only be reached from left, not below) then one of the $\max$ terms vanish. The above equation is called the DP formula of the question. With it, starting from the bottom-left vertex, we populate a two-dimensional lookup table/array storing the values of f at each vertex one at a time, all the way until we reach the top-right vertex. (One viable order is by completing the first column, then the second column, and so on.)

4.3 Weighted Interval Selection

Now we revisit the (unweighted) interval selection problem and assign weights to each interval:

> Given n intervals with starting points $s(i)$, finishing points $f(i)$, and weights $w(i)$, select a set of them without overlap that achieves maximum total weight.

Motivation: instead of visiting as many events as possible, prioritize those that we consider more important.

First note that our previous "earliest finish time" greedy algorithm fails: if we have two tasks, one from $t = 1$ to $t = 100$ with weight 100, the other from $t = 2$ to $t = 3$ with weight 1, "earliest finish time" picks the latter.
In fact, we claim that no myopic greedy algorithm works, since in this example, short-term decisions in fact depend on long-term knowledge, something greedy algorithms do not have access to. Consider the following example:

weights: blue = 2

If the red interval has weight 1, then the optimal solution would be selecting the bottom 4 intervals, but if the red interval has instead weight 3, then it is clearly more optimal to select the top 4. There is no way for any greedy algorithm to figure out which interval to pick first!

A DP Approach to Weighted Interval Selection

For DP, **we often analyze relatively easy** (sometimes even trivial, as we see in this example) **properties of the optimum solution, and exploit them to derive an efficient algorithm**.
In this example, what kind of trivial/apparent property can we abuse? Given a set S of intervals, define $\mathrm{OPT}(S)$ to be the total weight in the optimal solution. The key observation is that **for any interval $i \in S$, either i is selected by the optimal solution or not**. No third possibility. We can derive the following based on this trivial claim:

- If i is not picked by the optimal solution, then the optimal solution is identical to that of $S \setminus \{i\}$, since we wouldn't pick i anyways. In other words, $\mathrm{OPT}(S) = \mathrm{OPT}(S \setminus \{i\})$.
- On the other hand, if i is picked by the optimal solution, then the optimal solution for S consists of i, plus the optimal solution of $S \setminus \{\text{every interval intersecting } i\}$ — apparently, if i were to be chosen, anything else intersecting it cannot. In other words, $\mathrm{OPT}(S) = w(i) + \mathrm{OPT}(S \setminus \{\text{everything intersecting } i\})$.

Implementing Solution in Polynomial Time

We can implement this naively using exhaustive search with backtracking, obtaining a runtime of $\mathcal{O}(2^n)$ since a set of size n has 2^n subsets. Even with a memoization table, we still need to store info related to each subset, so still 2^n. How do we improve?

Now it's a good time to compare our weighted interval selection against the previous lattice graph example:

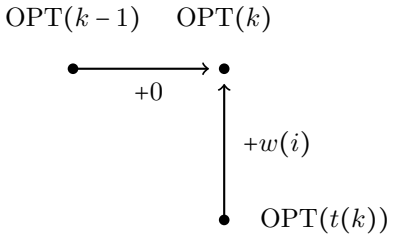
- In both examples, we find two potential "previous steps" that lead to the optimal solution of the current step.
- In the lattice graph example, the previous steps are: (i) taking a step up from the vertex below, and (ii) taking a step to the right from the vertex on the left.

- In the weighted interval example, we do have two previous steps, but it is not immediately clear how to find them, in particular computing $S\backslash\{\text{everything intersecting } i\}$, unlike in the lattice graph example, where we know immediately what the two predecessor vertices are. **The notion of a clear "direction" is missing here**.

Fortunately, we do have the power to pick the interval number i that we branch over and hence obtain a sense of direction. A good choice is again, the finish time, and we show by doing so we reduce runtime to $\mathcal{O}(n \log n)$ (in fact, the DP itself takes only linear time, making the sorting of finishing time the bottleneck).

We now assume the following:

- All intervals are sorted by non-decreasing finish times, so $f(i) \leqslant f(j)$ if $i < j$.
- We have an additional function $t(\cdot)$, such that $t(k)$ is the last interval j that finishes before before k starts. In other words, this is the last interval, index-wise, that is before k and does not overlap with k. Define $t(k) = 0$ if all previous intervals overlap with k.



With these extra assumptions, we can vastly simplify the recurrence relation. **Instead of computing the optimal solution w.r.t. a set, we compute it w.r.t. this sorted index**. Now, given a set S of sorted intervals $\{1, 2, \cdots, k\}$, the last interval k is either selected by $\mathrm{OPT}(k)$ or not. And there are precisely two paths to reach it:

- k is not in $\mathrm{OPT}(k)$, In this case $\mathrm{OPT}(k) = \mathrm{OPT}(k-1)$ since they correspond to the same set of intervals.
- k is in $\mathrm{OPT}(k)$. This means interval k is selected, incurring an additional weight $w(i)$. What is the previous step if we chose this path? Well, all other intervals must have indices $\leqslant t(k)$, since they cannot overlap with k. And of course, we want to pick as much weight as possible from the first $t(k)$ intervals, and we know this quantity is $\mathrm{OPT}(t(k))$. Hence in this path $\mathrm{OPT}(k) = w(i) + \mathrm{OPT}(t(k))$.

Now we have all the ingredients to cook up the polynomial-time solution.

Algorithm 9: Weighted Interval Selection via DP

Inputs: n intervals, with starting points $s(i)$, finishing points $f(i)$, and weights $w(i)$

---

sort the intervals by non-decreasing finishing time
pre-compute all $t(\cdot)$ values
allocate array $\mathrm{A}[n+1]$, zero-indexed
$\mathrm{A}[0] \leftarrow 0$
for $i = 1, 2, \cdots, n$ **do**
    $\mathrm{A}[i] \leftarrow \max(\mathrm{A}[i-1], w(i) + \mathrm{A}[t(i)])$
Return: $\mathrm{A}[n]$

Analyzing its runtime is straightforward — $\mathcal{O}(n \log n)$ for sorting, $\mathcal{O}(n \log n)$ for computing $t(\cdot)$ using n binary searches ($\mathcal{O}(n)$ also possible), and $\mathcal{O}(n)$ for the DP loop. Total runtime: $\mathcal{O}(n \log n)$.

Correctness proof. To prove $\mathrm{A}[n] = \mathrm{OPT}(n)$ we use strong induction and prove this holds for all $0 \leqslant i \leqslant n$. The base case is clear, since $\mathrm{OPT}(0) = 0$ as there is nothing to select, and we initialized $\mathrm{A}[0]$ to 0.

Using strong induction, we see[25]

$$\begin{aligned} \mathrm{A}[i] &= \max(\mathrm{A}[i-1], w(i) + \mathrm{A}[t(i)]) && \text{by line 7} \\ &= \max(\mathrm{OPT}(i-1), w(i) + \mathrm{OPT}(t(i))) && \text{by strong IH and } t(i) < i \\ &= \mathrm{OPT}(i) && \text{by recurrence relation.} \end{aligned}$$

□

4.4 Edit Distance / String Alignment

In this section we consider the following problem:

> Given two strings $x[1:n]$ and $y[1:m]$ (one-indexed, lengths n, m respectively), how similar are them?

Here we choose edit distance as our metric for similarity. It cares only about single-character edits (insertions, deletions, and substitutions). Clearly, this is closely related to applications like spell checkers.
An alternate way to view these three operations is string alignment: for example, to align words *"occurrence"* and *"ocurrannce"*, we align the two words vertically, adding hyphens to both words whenever we want to make them the same length.

o	-	c	u	r	r	a	n	n	c	e
o	c	c	u	r	r	e	-	n	c	e

It follows that to transform *"ocurrannce"* into *"occurrence"*, one way is to insert a c, replace an "a" with "e", and remove an redundant "n." Insertion and deletion correspond to having a blank in one of the words, and replacement corresponds to having different characters at the same index.

In our course, **we will assume that insertion and deletion both cost A, and overwriting/replacement costs B.** And we will use the alignment version when constructing our DP solution. We can easily generalize this to more complicated scenarios, e.g. more specific costs, or non-constant costs even for the same operation.

A DP Approach to String Alignment

In order to construct the DP formula for this question, we'd like to know the last position of the optimal alignment of $x[1:n]$ and $y[1:m]$. Since we only had three operations, there are three (four, to be precise) cases:

(1) *An alignment (correct or incorrect, hence two sub-cases) of $x[n]$ and $y[m]$.* In this case, $x[1:n-1]$ and $y[1:m-1]$ will be aligned with each other. To obtain optimal solution of aligning $x[1:n]$ and $y[1:m]$, the subproblem must have also been optimized, so $x[1:n-1]$ and $y[1:m-1]$ must have been aligned optimally.

- If $x[n]$ and $y[m]$ are aligned correctly (i.e. same character), no additional cost compared to the optimal solution of aligning $x[1:n-1]$ and $y[1:m-1]$.
- Otherwise, an additional cost B of replacement will be incurred compared to the optimal subsolution.

(2) *An alignment of $x[n]$ with a blank.* In this case, $x[1:n-1]$ and $y[1:m]$ must have been aligned optimally. Compared to this optimal subsolution, we have an additional cost of A.

[25] A lot of DP correctness proofs look like this, feeling as if we were just stating the obvious. Usually a three-liner: (i) state what the algorithm does, (ii) using IH to replace the table entry into OPT, and (iii) use recurrence relation.

(3) *An alignment of a blank with $y[m]$.* Symmetric to the previous case. $x[1:n]$ and $y[1:m-1]$ must have been aligned correctly. Additional cost A again.

We now translate above into formulas, denoting the optimal solution of aligning $x[1:i]$ and $y[1:j]$ by $\mathrm{OPT}(i,j)$:

$$\mathrm{OPT}(0,0) = 0 \qquad \mathrm{OPT}(i,0) = iA \qquad \mathrm{OPT}(0,j) = jA \qquad \text{(base cases)}$$

$$\begin{aligned}\mathrm{OPT}(i,j) = \min(&\mathrm{OPT}(i-1,j-1) + B \cdot \mathbf{1}[x[i] \neq y[j]]^{\ddagger},\\ &\mathrm{OPT}(i-1,j) + A,\\ &\mathrm{OPT}(i,j-1) + A).\end{aligned} \qquad \text{(recurrence relation)}$$

Note that the base cases also include the situations where we align a non-empty string with an empty one (length 0), in which case the only operation we should perform is deletion. This recurrence relation might look daunting overall, but implementing it isn't. All we need is one two-dimensional array.

Algorithm 10: Edit Distance / String Alignment via DP

Inputs: one-indexed strings $x[1:n], y[1:m]$, last-index inclusive

allocate table $\mathrm{A}[0:n][0:m]$, zero-indexed, last-index inclusive
for $i = 0, 1, \cdots, n$ **do** $\mathrm{A}[i][0] \leftarrow iA$
for $j = 0, 1, \cdots, m$ **do** $\mathrm{A}[0][j] \leftarrow jA$
for $i = 1, 2, \cdots, n$ **do**
  for $j = 1, 2, \cdots, m$ **do**
    if $x[i] = y[j]$ **then**
      $\mathrm{A}[i][j] \leftarrow \min\{\mathrm{A}[i-1][j-1], \mathrm{A}[i-1][j] + A, \mathrm{A}[i][j-1] + A\}$
    else
      $\mathrm{A}[i][j] \leftarrow \min\{\mathrm{A}[i-1][j-1] + B, \mathrm{A}[i-1][j] + A, \mathrm{A}[i][j-1] + A\}$
Return: $\mathrm{A}[n][m]$

It is also clear[27] that this algorithm runs in $\Theta(nm)$ time: $\Theta(m+n)$ for base cases, and $\Theta(nm)$ iterations of the double loop, each time doing $\mathcal{O}(1)$.

Correctness proof. Our goal is to prove that $\mathrm{A}[n][m] = \mathrm{OPT}(n,m)$, so we will use strong induction on $i+j$ to prove $\mathrm{A}[i][j] = \mathrm{OPT}(i,j)$ for all $0 \leqslant i \leqslant n$ and $0 \leqslant j \leqslant m$.
Base case: when $i + j = 0$ we must have $i = j = 0$, and indeed, $\mathrm{A}[0][0] = 0$, and aligning two empty strings is free of any cost.
Using strong induction on $i + j$, we now prove the inductive step:

(1) If $i = 0$, then $a[0][j] = jA$ according to the "base case" stated in the DP formula. And indeed, aligning an empty string with a string of length j involves j insertions whose total cost is jA.

(2) If $j = 0$, the proof is analogous to above.

[‡]Indicator function: the second term is B if $x[i] \neq y[j]$ and 0 otherwise.
[27]One thing I love about DP is that their runtime analyses are usually extremely simple, since the recurrence relationship is basically all we need to analyze.

(3) If $x[i] = y[j]$, then

$$\begin{aligned} \mathrm{A}[i][j] &= \min\{\mathrm{A}[i-1][j-1], \mathrm{A}[i-1][j] + A, \mathrm{A}[i][j-1] + A\} && \text{(line 8)} \\ &= \min\{\mathrm{OPT}(i-1, j-1), \mathrm{OPT}(i-1, j) + A, \mathrm{OPT}(i, j-1) + A\} && \text{(strong IH on index sum)} \\ &= \mathrm{OPT}(i, j). && \text{(recurrence relation)} \end{aligned}$$

(4) Finally, if $x[i] \neq y[j]$, then

$$\begin{aligned} \mathrm{A}[i][j] &= \min\{\mathrm{A}[i-1][j-1] + B, \mathrm{A}[i-1][j] + A, \mathrm{A}[i][j-1] + A\} && \text{(line 10)} \\ &= \min\{\mathrm{OPT}(i-1, j-1) + B, \mathrm{OPT}(i-1, j) + A, \mathrm{OPT}(i, j-1) + A\} && \text{(strong IH on index sum)} \\ &= \mathrm{OPT}(i, j). && \text{(recurrence relation)} \end{aligned}$$

□

4.5 The Knapsack Problem

> Suppose there are n items, each having integer weight $w(i)$ and arbitrary value $v(i)$. You can carry a total weight of at most W (assuming for convenience that $W \in \mathbb{Z}$). How to maximize the total value of the selected items?

Motivation: imagine if you were a thief and you broke into a clock shop. Each clock has its own values and weight. You could only carry a certain amount of weight. How would you make the most out of this burglary?

We can come up with easy counterexamples to show that greedy approaches fail[28]. So we resort to DP.

A DP Approach to the Knapsack Problem

Like before, our trivial insight is that **each item i is either included or not included in the optimal solution**.

As a first approach, we consider the subproblems of form $\mathrm{OPT}(i)$, where we are presented with the same Knapsack problem but with items $1, 2, \cdots, i$ only. Using our observation:

- If $\mathrm{OPT}(i)$ does not include item i, then it is as if item i didn't exist: $\mathrm{OPT}(i) = \mathrm{OPT}(i-1)$.
- If $\mathrm{OPT}(i)$ includes item i, then its total value consists of $v(i)$ due to item i, as well as everything else from $\mathrm{OPT}(i-1)$.

Therefore, we propose the following recurrence relation:

$$\mathrm{OPT}(i) = \max(\mathrm{OPT}(i-1), v(i) + \mathrm{OPT}(i-1))$$

... which just includes every item and is clearly wrong. The problem is **we never used the assumption on total weight W**. But if we just stated our subproblem as "optimal solution for items $1, 2, \cdots, i$, of total weight $\leqslant W$," it would instead violate the optimality of sub-solutions: if we included item i, the solution for items $1, 2, \cdots, i-1$ now has a lower total weight.

[28] It fails for the integral Knapsack problem, i.e., either take an entire item or leave it, but being greedy on $v(i)/w(i)$ solves fractional Knapsack.

To address this, we introduce an additional variable: the total allowed weight, and redefine $\mathrm{OPT}(\cdot)$ as follows:

$$\mathrm{OPT}(i, w) \coloneqq \text{maximum total value obtainable from items} \{1, 2, \cdots, i\} \text{ with weight limit } w.$$

This time:

- If $\mathrm{OPT}(i, w)$ does not include item i, then like before, $\mathrm{OPT}(i, w) = \mathrm{OPT}(i-1, w)$.
- However, if $\mathrm{OPT}(i, w)$ indeed selects item i, then the subproblem now has a reduced weight budget $w - v(i)$. If this quantity is negative, then it means $\mathrm{OPT}(i, w)$ could not have chosen item i. If now, the equation is now $\mathrm{OPT}(i, w) = v(i) + \mathrm{OPT}(i-1, w - w(i))$.

Combining these observations and adding base cases, we now arrive at the correct recurrence relation:

$$\mathrm{OPT}(i, w) = 0 \qquad \mathrm{OPT}(i, w) = \begin{cases} \mathrm{OPT}(i-1, w) & w < w(i) \\ \mathrm{OPT}(i, w) = \max(\mathrm{OPT}(i-1, w), v(i) + \mathrm{OPT}(i-1, w - w(i))) & w \geqslant w(i). \end{cases}$$

Implementing this recurrence relation is also straightforward:

Algorithm 11: Knapsack via DP

Inputs: n items, with weights $w(i)$ (integer) and values $v(i)$; total budget W (integer)

```
2  allocate n × W array A[][]
3  for w = 0, 1, ···, W do
4      A[0][w] ← 0 // base cases
5  for i = 1, 2, ···, n do
6      for w = 0, 1, ···, W do
7          if w(i) > w then
8              A[i][w] ← A[i − 1][w] // DP case 1
9          else
10             A[i][w] ← max(A[i − 1][w], v(i) + A[i − 1][w − w(i)]) // DP case 2
11 Return: A[n][W]
```

Correctness proof. Omitted. Same trick as always. Weak induction on i suffices. □

Further, we can also use backtracking to retrieve the optimal subset of items in $\mathcal{O}(n)$ time by allocating another $n \times W$ table, setting the (i, w) entry to 0 if $\mathrm{A}[i][w] = \mathrm{A}[i-1][w]$, and 1 otherwise. Then, we backtrack from (n, W), decrementing the values by $i \leftarrow i - 1$ and $w \leftarrow w - w(i)$ every time we see a 1 in this table along the path.

4.5.1 Pseudo-Polynomial Runtime

Lines 4, 8, and 10 all take $\Theta(1)$ time, so the entire algorithm takes $\Theta(W)$ for base cases and $\Theta(nW)$ for the main DP loop, resulting in a total of $\Theta(nW)$ for the Knapsack problem. This looks polynomial, but is it? Consider $W = 2^{100}$. We can easily store it as a 100-bit binary number, but running the algorithm takes forever. The problem here is that $\mathcal{O}(nW)$, while polynomial w.r.t. W, is **not polynomial w.r.t. its input size, $\log_2 W$.**

On the other hand, we also want to distinguish this type of exponential runtime from 2^n resulted from exhaustive search, for intuitively the latter is a lot worse. The distinction is that, **if the weights $\{w_i\}, W$ is reasonably small, then DP Knapsack is good, and it only starts to grow bad if we feed it with astronomical values of W.**

This leads to the following definition(s):

Definition: Polynomial & Pseud-Polynomial Algorithms

We define an algorithm to be **pseudo-polynomial** if it runs in polynomial time when the input is given in unary encoding (e.g. $8 = 11111111$ in unary). We define an algorithm to be **polynomial**[29]if runs in polynomial time when the input is given in binary encoding (e.g. $8 = 1000$ in binary).
(*Note that every pseudo-polynomial algorithm is polynomial.*)

With this definition, it is clear that if we view $W = 2^{100}$ as truly monstrous number using unary encoding (2^{100} 1's in a row), then yes, DP Knapsack is polynomial. But if we view 2^{100} as binary, then the runtime becomes polynomial w.r.t. W. Therefore, **DP Knapsack is pseudo-polynomial**. (*Of course, no one is ever insane enough to write numbers in unary. This curated definition serves the sole purpose of addressing the issue we discussed above.*)

But why didn't we bring this up earlier? The answer is that **it wasn't a problem until now — running times of arithmetic operations are polynomial w.r.t. binary encoding input size**. Consider a simple function that loops through an array $a[1:n]$, computing the sum of all elements. Suppose the largest number in the array is K, so we need $\log K$ bits[30] to represent it.

- The total input size is bounded by $[n + \log K, n \log K]$ (if all other numbers are single bit v.s. if all numbers are K).
- The largest possible partial sum involved is bounded by nK.
- Addition of two numbers a, b takes $\mathcal{O}(\max(\log a, \log b)) = \mathcal{O}(\log a + \log b)$, so each addition takes at most $\mathcal{O}(\log(nK)) = \mathcal{O}(\log n + \log K)$.
- At most n additions, so the total running time is at most $\mathcal{O}(n \log n + n \log K)$.

This space is intentionally left blank

[29]To further break it down, this is actually called *weakly polynomial*. There is yet a stronger category, *strongly polynomial*, that requires the algorithm to run in polynomial time on the number of input items, regardless of input values.
[30]I drop the base 2 here since they are equivalent in big-$\mathcal{O}$ notation.

4.6 Shortest Paths, Revisited

Back in CS104/170 we we introduced to the shortest path problem and Dijkstra's algorithm, but we always assumed that edge lengths are nonnegative/positive. Here we relax this assumption and consider the following, more general question:

> Given a directed graph $G = (V, E)$ with edge costs $c(e) \in \mathbb{R}$ (possibly negative), a start node s, and a finish node t, find a shortest path (minimum sum of edge costs) from s to t.

First, we note that with these weaker assumptions, the original Dijkstra immediately fails: consider a graph with 3 nodes, s, u, t. There is one direct edge from $s \to t$ with cost 1. Another path, $s \to u \to t$, has edge costs $c(s, u) = 2$ and $c(u, t) = -2$. Dijkstra would have picked $s \to t$ directly, but in reality the cost of $s \to u \to t$ is 0.

For simplicity, we also assume that G contains no negative cycles (cycles with negative weight sums that are reachable from s and reach t): otherwise we can loop through this cycle multiple times and obtain even "cheaper" paths, making our definition of shortest path ill-defined.

A DP Approach to the Generalized Shortest Path Problem

We first try to define $\mathrm{OPT}(v)$ as the minimum cost of any path from v to t. This leads to the trivial base case $\mathrm{OPT}(t) = 0$. For other nodes, the subproblem is decided by the next hop of the path, observing that **the optimal path from v to t has to take one of the edges (v, u) first, then follow the optimal path from u to t**. Hence, the general recurrence relation is given by

$$\mathrm{OPT}(v) = \min_{\text{outgoing edges from } v} (c(v, u) + \mathrm{OPT}(u)).$$

This is called the **Bellman equation**. It is correct, but it is not clear how to convert it to a tabular/recursive algorithm. Once again, we are lacking the notion of order here. Hence, like Knapsack, we introduce an auxiliary variable k limiting the total number of hops

$$\mathrm{OPT}(v, k) := \text{minimal cost of } v \to t \text{ paths, with } \leqslant k \text{ hops.}$$

It follows that

$$\begin{cases} \mathrm{OPT}(t, k) = 0 & \text{(base cases) for all } k \\ \mathrm{OPT}(v, 0) = \infty & \text{(base cases) for all } v \neq t \\ \mathrm{OPT}(v, k+1) = \min\limits_{u:(v,u) \text{ is an edge}} c(v, u) + \mathrm{OPT}(u, k). & \text{(recurrence)} \end{cases}$$

This is called the **Bellman-Ford algorithm**. Before we implement the algorithm, observe that **the shortest path has $\leqslant n$ nodes** (so $\leqslant n - 1$ hops). Otherwise, by pigeonhole principle at least one node would repeat, thus creating a cycle, and by the no-negative-cycle assumption, this cycle has non-negative weight. So we can just remove it and make the path cheaper (or the same, in which case why not?).

With this in mind, we know precisely the size of the tabular dimension DP requires: $n \times n$. Implementation should be a breeze now:

Algorithm 12: Generalized Shortest Path via DP; Bellman-Ford

1 **Inputs**: directed graph $G = (V, E)$ with edge costs $c(e)$; start/end nodes $s, t \in V$

---

2 allocate table of dimension $n \times n$; first argument = node[31], second = path length upper bound

3 $\mathrm{A}[t][0] \leftarrow 0$

4 **for** all $v \in V$ **do** $\mathrm{A}[v][0] \leftarrow \infty$

5 **for** $k = 1, 2, \cdots, n-1$ **do**

6 **for** all nodes $v \neq t$ **do**

7 set $\mathrm{A}[v][k] \leftarrow \min_{u:(v,u) \text{ is an edge}} c(v,u) + \mathrm{A}[u][k-1]$

8 **Return**: $\mathrm{A}[s][n-1]$[32]

Correctness proof. Omitted. Use induction on k and prove that for all k, $\mathrm{A}[v][k] = \mathrm{OPT}(v, k)$ for all nodes v. Standard three-step proof as always. □

Viewing lines 6 and 7 as a whole, we basically inspected each edge once, and so the runtime is $\Theta(m)$, where $m = |E|$. Hence the outer loop has runtime $\Theta(mn)$. This dominates all other work (initialization, base cases), so this implementation of Bellman-Ford runs in $\Theta(mn)$. Slightly worse than Dijkstra's $\Theta(m \log n)$.

A final question: What if there were negative cycles? How do we detect and find them? The solution is simple: we run one more iteration (now totaling $n + 1$). We showed previously that, if there were no negative cycles, then all results would be final. Therefore, if we obtain different values after running one additional iteration, we know it must be caused by a negative cycle. Put more formally: if $\mathrm{OPT}(u, n-1) \neq \mathrm{OPT}(u, n)$ for some u, then there exists a negative cycle along the longer path consisting of n hops and $n+1$ nodes, by our previous pigeonhole argument.

4.6.1 (Optional) Adding Asynchronicity: Distance Vector Protocols

In our previous implementation of Bellman-Ford, we adopted a "pull-based" approach, in the sense that each node *actively* contacts its neighbor, asking for information, even when the neighbor often times has no new information to provide.

This pull-based approach is somewhat analogous to an obnoxious person waiting in line at a restaurant, shouting, "Is it my turn yet? Is it my turn yet?" This is both annoying and inefficient, so instead we can consider a "push-based" approach, such that we patiently wait until the reception brings news to us.

Converting this idea back to the shortest path problem, we have an alternate approach to constructing the table: (i) if a node v figures out a better path from itself to t, notify all incoming neighbors (nodes u with an $u \to v$ edge), and (ii) otherwise stay put, and patiently wait for updates from its outgoing neighbors to do what is described in (i), so that the only way it figures out a better path exists is if one of its outgoing neighbors told it so.

Think of it as ripple effect, or even BFS. At first, t is the only node knowing how to get to t (trivial case also). Every other node thought t is unreachable, represented by distance ∞. Node t then tells all of its incoming neighbors that "hey, you can actually reach me," and then each incoming neighbor tells its own incoming neighbors that "hey, you can actually reach t," and so on and so forth.

There is one more thing we'd like to add: **asynchronicity**. The motivation is simple and practical: routers and

[31] You of course need some additional implementation to convert nodes into node IDs. Not included here.

[32] Professor Kempe's notes said we return $\mathrm{A}[s][n]$, but I believe $n-1$ is correct: we want a path of at most n nodes, and hence $\leqslant n-1$ hops.

network. When we have multiple machines, how do we efficiently send a message from one machine to another? Further, we have no control over the performance of each individual machine — some routers may be slower at reporting updates, some may need more time to process the data before doing computations, and so on. Therefore, the BFS-alike, push-based algorithm needs some optimization. Instead of propagating updates in rounds/iterations, we say a node becomes *active* if it needs to spread some update, and once it is done notifying all its incoming neighbors, the state reverts to *inactive* again. The following algorithm implements this idea. Compared to our previous versions, it has the advantage of having no (or little) constraint on the ordering of updates.

Algorithm 13: Asynchronous Shortest Path (source: KT §6.9)

Inputs: directed graph $G = (V, E)$ with edge costs $c(e)$; start/end nodes $s, t \in V$

---

allocate node-indexed array M of size $n = |V|$
initialize $M(t) \leftarrow 0$ and $M(v) \leftarrow \infty$ for all $v \neq t$
declare t to be *active* (and all other nodes *inactive*)
while there exists an active node **do**
    $w \leftarrow$ any active node
    y **for** all edges $v \to w$ **do**
        $M(v) \leftarrow \min(M(v), c(v, w) + M(w))$
        if the value of $M(v)$ changed **then**
            set v's next hop to be w `// for backtracing, if needed`
            declare v to be *active*
    declare w to be *inactive*
Return: backtracing results, and/or $M(s)$

5 Network Flow: Max-Flow / Min-Cut

In this section we are back dealing with graphs, but instead of assigning a weight/cost to each edge, we introduce the notion of *edge capacities*. We will be using graphs to model *transportation networks*. For example, think of a highway system, where the edges, represented by highways, can each tolerate a specific amount of traffic (edge capacities), and they intersect at interchanges, represented by nodes, that serve the purpose of merging or diverting traffic.

There are several main components of such network models: (i) a graph representing the entire system, (ii) **edge capacities** representing the width of each road, (iii) a **source** node that only generates traffic (think of an arrival-only airport), (iv) a **sink** node that only absorbs traffic (think of an departure-only airport), and (v) the actual **flow** (traffic) traversing through the graph via the edges.

Formally, we define the flow as follows:

Definition: Flow (on graphs)

Let a graph $G = (V, E)$ with edge capacities $c(e)$ be given, as well as a **source node** $s \in V$ and a **sink node** $t \in V$. An $\boldsymbol{s-t}$ **flow** is a function $f : E \to \mathbb{R}$ (assigning amount of flows to each edge) satisfying

- (conservation) for all **internal nodes** $v \notin \{s, t\}$, $\sum_{e \text{ out of } v} f(e) = \sum_{e \text{ into } v} f(e)$, and
- (constraints) $0 \leqslant f(e) \leqslant c(e)$ for all edges e.

Further, we define the **value** of the flow, $\nu(f)$, to be $\sum_{e \text{ out of } s} f(e)$, the total flow out of the source (which can be proven to equal to the total flow into the sink).

There is a possibly more intuitive perspective of viewing an $s-t$ flow: we can "break down" an $s-t$ flow into multiple $s-t$ paths, $P(1), P(2), \cdots, P(k)$, where there is a nonnegative flow $a(i) \geqslant 0$ along each path. The constraint it needs to satisfy, certainly, is that the edge capacities are obeyed:

$$\sum_{i : e \in P(i)} a(i) \leqslant c(e) \qquad \text{for all edges } e.$$

These two definitions are equivalent. In particular, given a flow specified in one way, we can convert it to the other in polynomial time. We refer to this result as the **path decomposition lemma**. For example, given a flow f, we can find $k \leqslant m$ paths, associated with $a(i)$, such that

$$f(e) = \sum_{i : e \in P(i)} a(i) \qquad \text{for all edges } e.$$

We will not prove the lemma here — it just involves BFS and induction. But see here for a complete proof.

Introducing the Max-Flow / Min-Cut Duality

A natural optimization that arises from this definition is, **how do we maximize $\boldsymbol{\nu(f)}$ among all $\boldsymbol{s-t}$ flows?**

Before going into any theorems or proofs, let us connect these new concepts with one that we are well familiar with — cuts. Recall that a cut (S, S^c) is simply a partition of the nodes. We further define an $\boldsymbol{s-t}$ **cut** to be one such that

$s \in S$ and $t \in S^c$. Given the context of network flows, the capacity of a cut (S, S^c) can be defined as total capacity of edges crossing the cut: $\sum_{u \in S, v \in S^c} c(u, v)$.

An interesting observation is that the capacity of cuts serve as a bottleneck for flows: **the maximum flow must not exceed the capacity of any cut.** This is a direct result from the constraint $f(e) \leqslant c(e)$ for all edges e. **In particular, the maximum flow cannot exceed the capacity of the minimum cut**.

In fact, things get even nicer — we will prove later that these two objectives are completely equivalent, in the sense that if we find Max-Flow, we also find Min-Cut, and vice versa.

I will now state the theorem to blow your mind, but then we'll revisit the bipartite matching problem and see how we can conveniently reduce it to Max-Flow.

Theorem: Max-Flow/Min-Cut (Ford-Fulkerson)

Given a directed graph $G = (V, E)$ with edge capacities $c(e) \geqslant 0$, as well as source node s and sink node t:

(1) There exists a polynomial time algorithm for finding a maximum $s - t$ flow f and a minimum $s - t$ cut (S, S^c) with respect to cut capacity;

(2) If all edge capacities $c(e)$ are integers, then the algorithm outputs a maximum $s - t$ flow in which all flows $f(e)$ are integers; and

(3) Max-Flow value equals Min-Cut capacity.

5.1 A First Example: Maximum Bipartite Matching

The question can be summarized into a one-liner:

> Given a bipartite graph $G = (V, E) = (X \cup Y, E)$, find a matching of maximal cardinality.

(Recall a bipartite graph $G = (X \cup Y, E)$ is one where the nodes are partitioned into two sets, and edges only go between the two sets, not within either. And a matching M is a set of edges where each node is incident on at most one edge in M.) This was one of the oldest problems in combinatorial algorithms, and historically it was first solved in polynomial time by reducing it into a network flow problem, as we will see now.

5.1.1 Reduction to Max-Flow

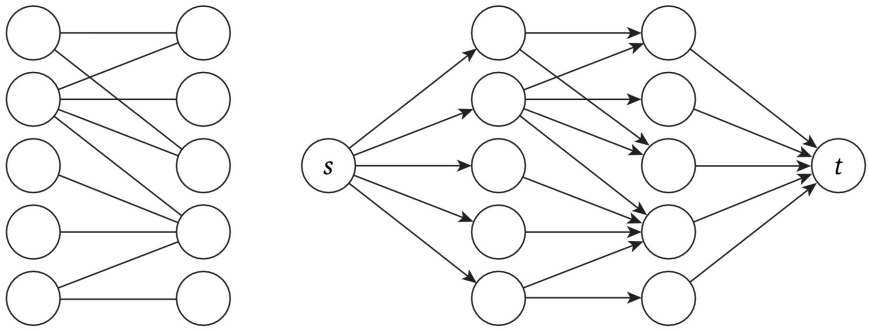
We apply the following to the bipartite graph:

(1) Add a new source s on the left and a sink t on the right.

(2) Connect s to all nodes in X, setting edge capacity to 1.

(3) Connect all nodes in Y to t, setting edge capacity to 1 as well.

(4) Make each original edge from X to Y directed, with capacity ∞.



Source: KT 7.9

Correctness proof. Now suppose we obtained a maximum $s-t$ flow f of the extended graph. We define our matching M to be the collection of $X \to Y$ edges whose flow is nonzero. We need to show two things: (i) M is a matching, and (ii) M is maximal.

To prove (i), we note that each $x \in X$ as at most 1 unit of flow entering, so it has at most 1 unit of flow leaving. By the second assumption of the theorem, since we have integer edge capacities, we also have an integral Max-Flow. Hence if an $X \to Y$ edge has nonzero flow then the flow must be precisely 1. And when this happens, there is no remaining flow, so there can be at most one edge incident on x in M. Same reasoning applies to each $y \in Y$.

To prove (ii), we notice that $|M| = \nu(f)$. Suppose there exists another matching M' with $|M'| > |M|$. If so, for each $e = (u, v) \in M$, where $u \in X, v \in Y$, we send one unit of flow along the path $s \to u \to v \to t$. Since M is a matching, there are no duplicate u's or v's, and so the capacity constraints are not violated and the result a valid $s-t$ flow. Calling this new flow f', we have $\nu(f') = |M'| > |M| = \nu(f)$, contradicting the assumption that f is a Max-Flow. This completes the proof. □

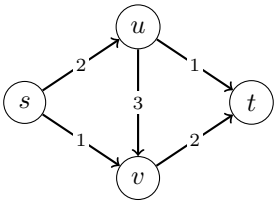
5.2 Proving the Max-Flow/Min-Cut Theorem

5.2.1 Explaining the Algorithm

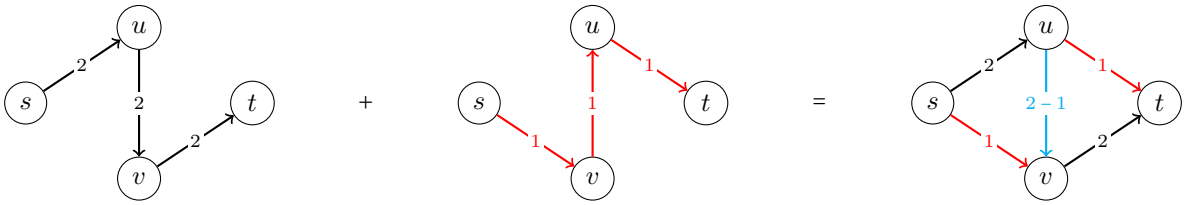
Naturally, we attempt a greedy-alike algorithm, using the path decomposition perspective of flows: while there is a $s-t$ path that we can "stuff" more flow into it, we do so. Put more formally:

Algorithm 13: Max-Flow: DP approach

```
// initializations, etc.
while there is a s – t path on which all edges have capacity remaining do
    pick one such path P, and put as much flow as possible on it (i.e. min remain capacity)
// return
```



It is clear that this greedy approach returns a valid flow, because nowhere in the algorithm did we break the edge capacity constraints. However, as the example on the left demonstrates, it does not necessarily return the maximum flow. There is only one valid $s \to t$ path: $s \to u \to v \to t$, so we send a 2 units of flow along it. But then there is no path left, even though the following diagrams show that it is possible to construct a flow with $\nu(f) = 3$.



Greedy: $\nu(f) = 2$ Augment Max-Flow: $\nu(f) = 3$

The problem in this specific example is that the edges $e \to v$ and $u \to t$ are never used, because greedy considers $u \to v$ as a unidirectional edge. **We can solve this problem by "undoing" flows.** Specifically, we push 1 unit of flow along $s \to v$ because we want to increase $\nu(f)$. But now, v is receiving more flow than its output capacity, so we "undo" 1 unit of flow along $u \to v$. But then u receives 2 units of flow from s and is only currently outputting 1 to v, so we send the other surplus unit of flow to t. In essence, we created an additional, hidden flow through $s \to v \to u \to t$, even though the directed edge $v \to u$ does not exist in the graph.

So far, I haven't been able to find an intuitive explanation of the idea of "undoing flows" using real-life examples. Using the highway example, it simply doesn't make sense if we ask some cars to drive backwards. And I don't think making a one-way road two-way is a proper explanation either, as that essentially makes the directed graph undirected. Any additional feedback would be appreciated. However, one thing for sure is that the resulting flow still satisfies both flow properties.

To formalize these, we introduce **residual graphs** and two types of auxiliary edges: **forward** and **backward edges**.

Definition: Residual Graphs

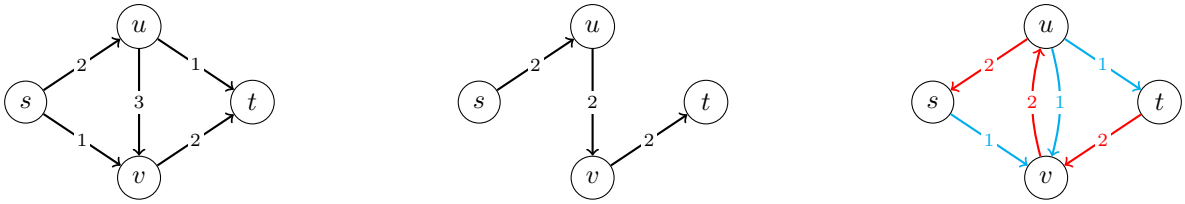
Given a capacity-embedded graph $G = V(E)$ and an $s - t$ flow f on it, the corresponding **residual graph** $G(f)$ is defined as follows:

(1) The node set of $G(f)$ is V, identical to that of G.

(2) For each edge $e = (u, v)$ of G:

- If $f(e) < c(e)$, there are $c(e) - f(e)$ "leftover" capacity, so we add a **forward edge** (u, v) with capacity $c'(u, v) = c(e) - f(e)$.
- If $f(e) > 0$, there are $f(e)$ amount of flow that we can "undo", so we add a **backward edge** (v, u) [*note the direction*] with capacity $c'(v, u) = f(e)$.

With these definitions, we now look at our simple example again. The residual graph corresponding to the greedy flow f is drawn below, with blue edges representing forward edges, and red edges representing backward edges.



Original graph G Greedy flow f Residual graph $G(f)$

Here is an alternate way to explain how we improved our greedy f to Max-Flow. Notice that there is precisely one $s - t$ path remaining in this residual graph $s \to v \to u \to t$, and we see that this path can transmit up to 1 unit of

flow. This is precisely how we updated the original graph G: we **augmented** 1 unit of flow along this path found in the residual graph $G(f)$. Specifically, we pay attention to the unidirectional $u \to v$ path in the original graph G: the augmentation process essentially un-sends 1 unit of flow from $u \to v$, since the $v \to u$ path in the residual graph $G(f)$ is an backward edge. Therefore, **when augmenting G using $G(f)$, we need to pay attention to whether each edge is a forward or backward edge in $G(f)$.**

We now have all the ingredients to cook up the **Ford-Fulkerson Algorithm**:

Algorithm 14: Ford-Fulkerson Algorithm

Inputs: directed graph $G = (V, E)$ with edge capacities $c(e)$; source s, sink t

start with zero flow, i.e., $f(e) = 0$ for all edge
while residual graph $G(f)$ contains an $s - t$ path **do**
    let P be one such $s - t$ path
    `augment`(f, P), update flow
    `compute_residual_graph`$(G(f), f)$, update residual graph
Return: flow f

Function `compute_residual_graph`(*Graph* $G = (V, E)$, *flow* f):
    start with empty edge set $G(f) = (V, \varnothing)$
    for *each edge* $e = (u, v) \in E$ **do**
        if $f(e) < c(e)$ **then** add (u, v) to $G(f)$ with capacity $c(e) - f(e)$
        if $f(e) > 0$ **then** add (v, u) [*reversed order*] to $G(f)$ with capacity $f(e)$
    Return: residual graph $G(f)$

Function `augment`(*flow* f, *path* P):
    let ϵ be the smallest residual capacity along path P in $G(f)$
    for *each edge* $e = (u, v) \in P$ **do**
        if e *is a forward edge* **then**
            $f'(e) \leftarrow f(e) + \epsilon$
        else
            $f'(e) \leftarrow f(e) - \epsilon$ `// backward edge`
    Return: flow f'

5.2.2 The Main Proof

The correctness proof of Ford-Fulkerson is rather long, so we'll extract the key intermediate steps into Lemmas.

Lemma. The helper function `augment()` works correctly. In particular:

- If f is a flow, then the output f' of `augment()` is also a flow.
- If f is integral (integer-valued), and all capacities are integers, then f' will be integral too.
- $\nu(f') > \nu(f)$.

Proof. We first prove that f' is a flow. That is, we need to prove conservation, non-negativity, and capacity constraints.

(1) (Conservation) Clearly if a node v is not on the path P along which we augment f, then the conservation of flow remains unchanged. Let v be a node on the path, with incoming edge (u, v) and outgoing edge (v, w). Note that these are edges within the *residual graph*; we did not make any assumptions on the direction of the edges in the original G. This is particularly crucial in the second subcase stated below. Four cases:

- (u, v) and (v, w) are both forward. In this case, in G, u points to v and v points to w. Since $f'(u, v) = f(u, v)+\epsilon$ and $f'(v, w) = f(v, w)+\epsilon$ and all other edges incident on v remain unchanged, we experience a net gain of ϵ for both flows into and out of v. Conservation is preserved.
- (u, v) is forward and (v, w) is backward. This means that in the original graph G, both directed edges point to v. Then $f'(u, v) = f(u, v) + \epsilon$ and $f'(w, v) = f(w, v) - \epsilon$, and with all other nodes unchanged, the total flow into v is unchanged. The total flow out of v is also unchanged since we didn't use any outgoing edges.
- The other two cases are analogous to the two above.

(2) (Non-negativity) Forward edges were positive before, so the addition of $\epsilon > 0$ simply makes it larger. For backward edges (u, v): we are subtracting $\epsilon = \min c'(e) \leqslant c'(u, v) = f(v, u)$, so the result is still non-negative. (Here $c'(\cdot)$ is the capacity function of the residual graph.)

(3) (Capacity constraints) Trivial for backward edges, since we are reducing the flow in the original graph. For forward edges (u, v): we are adding $\epsilon = \min c'(e) \leqslant c'(u, v) = c(u, v) - f(u, v)$, so the sum is still bounded from above by $c(u, v)$, i.e., the capacity.

That f' remains integral is trivial: since f is integral and so are all capacities, the entire residual graph is integral, so ϵ is an integer. Finally, the first edge of P is out of s, and no edge goes into s. With $\epsilon > 0$, the flow out of s increases. □

It follows from this lemma that **Ford-Fulkerson outputs a valid $s - t$ flow**. It remains to prove that this is the maximum $s - t$ flow.

Lemma. For any $s - t$ flow f and $s - t$ cut (S, S^c), $\nu(f) = \sum_{e \text{ leaves } S} f(e) - \sum_{e \text{ enters } S} f(e)$.

Proof. This proof is nothing but manipulation of summations. We know that internal nodes (nodes other than s and t) satisfy flow conservation (flow in equals flow out). Therefore,

$$\nu(f) = \sum_{(s,v)\in E} f(s, v) = \sum_{(s,v)\in E} f(s, v) + \sum_{v\in S\setminus\{s\}} \Big(\underbrace{\sum_{(v,u)\in E} f(u, v) - \sum_{(u,v)\in E} f(v, u)}_{=0 \text{ for each internal node}} \Big).$$

It remains to notice that if (u, v) is an edge with both end nodes in S, then the corresponding term $f(u, v)$ will appear twice, with one "+" and one "–", so they cancel each other out. If (u, v) is such that $u \in S$ but $v \notin S$, then only the negative term appears. If (u, v) is such that $u \notin S$ but $v \in S$, then only one positive term appears.

Therefore we can drop the double sum (and apply the same reasoning to the very source node s):

$$\nu(f) = \sum_{v \in S} \Big(\sum_{e \text{ out of } v} f(e) - \sum_{e \text{ into } v} f(e) \Big) = \sum_{e \text{ leaves } S} f(e) - \sum_{e \text{ enters } S} f(e). \qquad \square$$

Lemma. Every $s-t$ flow f and $s-t$ cut (S, S^c) satisfies $\nu(f) \leqslant c(S, S^c)$ [cut capacity, $\sum_{e \text{ leaves } S} c(e)$].

Proof. One-liner, where the first equality is due to the previous lemma.

$$\nu(f) = \sum_{e \text{ leaves } S} f(e) - \sum_{e \text{ enters } S} f(e) \leqslant \sum_{e \text{ leaves } S} f(e) \leqslant \sum_{e \text{ leaves } S} c(e) = c(S, S^c). \qquad \square$$

We now prove that **Fold-Fulkerson outputs a maximum $s-t$ flow**. The key idea is *duality*: by the previous lemma, the value of any flow is bounded from above by the capacity of any cut. In particular,

$$\max_{\text{flow } f} \nu(f) \leqslant \min_{\text{cut } (S,S^c)} c(S, S^c).$$

There is no guarantee that the equality can be ever obtained, **but if we are able to find a flow f and a cut (S, S^c) attaining it, then it follows that we've found the Max-Flow and the Min-Cut at the same time**. And we shall.

Proof of Ford-Fulkerson. We know that when Ford-Fulkerson ends, the residual graph $G(f)$ contains no more $s-t$ path. Based on this observation, we define

$$S := \{v \in V : \text{there still exists a valid } s - v \text{ path}\}.$$

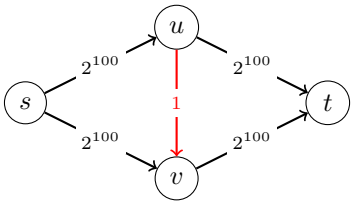
Notice that for every edge $e = (u, v)$ crossing the cut (with $u \in S$), we must not have a forward edge $u \to v$ in $G(f)$, for otherwise we can append $u \to v$ to the valid $s \to u$ path and prove that $v \in S$. By the definition of forward edges, this means $\underline{f(e) = c(e)}$.
On the other hand, for every edge $e = (u, v)$ with $u \in S^c$ and $v \in S$, we must have $f(e) = 0$. Similar reasoning, as we must not have the backward edge $u \to v$ in $G(f)$.
Using the summation-manipulation lemma, we see

$$\nu(f) = \sum_{e \text{ leaves } S} f(e) - \sum_{e \text{ enters } S} f(e) = \sum_{e \text{ leaves } S} c(e) - \sum_{e \text{ enters } S} 0 = c(S, S^c). \qquad \square$$

Runtime Analysis of Ford-Fulkerson

With the assumption on integral flow and edge capacities, we see that the value of the flow, $\nu(f)$, increases by at least 1 in each iteration. The final value of the flow is $F := \sum_{e \text{ out of } s} f(e)$. This leaves us a total runtime of $\mathcal{O}(F(m+n))$, **yet another pseudo-polynomial** algorithm.

An easy example to visualize this is shown on the right. If Ford-Fulkerson unfortunately chooses to start with $s \to u \to v \to t$, then it will iterate between augmenting along $s \to u \to v \to t$ and $s \to v \to u \to t$ forever. Each time the flow can only increase by 1, since the residual edge (u, v) or (v, u) has capacity at most 1. But the Max-Flow has value 2^{101}.



Solutions? We mention two heuristics to fix this problem:

(1) **Always use the widest path**, i.e., path P with the largest minimum residual capacity among all $s-t$ residual paths. This reduces the number of iterations to $\mathcal{O}(m \log F)$, so the algorithm is now (weakly) polynomial.

(2) **Always pick the shortest path** (known as **Edmonds-Karp Algorithm**), i.e., one with the minimum number of edges. This leads to a total of $\mathcal{O}(mn)$ iterations, free of F, making the algorithm *strongly polynomial*. (See this footnote if you need a recap of its definition.)

5.3 Image / Network Segmentation

We consider a type of interesting binary classification problem:

- Given an image, separate the foreground from the background. Additional constraint: adjacent pixels are more likely to be in the same category. Also, the image will look funny if the foreground-background boundary is too far from being smooth.
- Given a group consisting of a mixture of USC and UCLA students, classify each of them as either a USC or a UCLA student. Additional constraint: friendships are more likely to happen between similar people, so a USC student is more likely to befriend another USC student, compared to someone from UCLA. Trojans may become sad seeing their Trojan friends misclassified as Bruins.
- More generally, consider a network where edges link individuals that are more similar. Goal: given two labels A and B, label each individual with A or B, subject to a penalty if different labels are given to two individuals connected via an edge.

We now formulate our problem and objective:

> Given $G = (V, E)$, each node v has a *foreground score* $a(v) \geqslant 0$ if we label it with "A", a *background score* $b(v) \geqslant 0$ if we label it with "B". Further, for each edge $e = (u, v) \in E$, a penalty $p(e) \geqslant 0$ is incurred if u, v are labeled with different labels. Find a partition (S, S^c) of V maximizing
>
> $$Q(S, S^c) = \sum_{v \in S} a(v) + \sum_{v \notin S} b(v) - \overbrace{\sum_{\substack{e=(u,v) \\ u \in S, v \notin S}} p(e)}^{\text{penalty term}} .$$

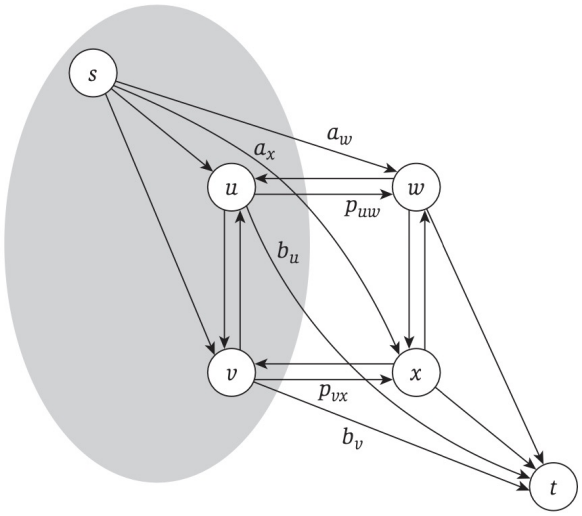
Solution[33]. Once all the data are given, $\sum_{v \in V} a(v) + \sum_{v \in V} b(v)$ is a fixed value. We note that the objective can be alternatively written as

$$Q(S, S^c) = \sum_{v \in V} (a(v) + b(v)) - \Big(\sum_{v \in S^c} a(v) + \sum_{v \in S} b(v) + \sum_{\substack{e=(u,v) \\ u \in S, v \notin S}} p(e) \Big).$$

It follows immediately that maximizing $Q(S, S^c)$ is equivalent to minimizing the sum of the red terms, and we will adopt this alternate objective. The conditions of last red term, i.e., all edges of form (u, v) where $u \in S, v \notin S$, scream for edges crossing the cut (S, S^c). But we need some careful rework to make sense of the two additional terms involving $a(v)$ and $b(v)$. To do so:

[33] Not my favorite type of solution. Quite challenging to provide helpful insights into the algebraic manipulations, as well as the reduction into Min-Cut.

- We define an external source s, and connect it with each node v with capacity $a(v)$.
- We also define an external sink t, and connect each node v to t with capacity $b(v)$.



Source: KT 7.19

This way, when we consider a cut of form $(\{s\} \cup S, S^c \cup \{t\})$, all edges of form (s, v) where $v \notin S$ serve as edges leaving $\{s\} \cup S$, and similarly, all edges of form (v, t) where $v \in S$ also serve as edges leaving $\{s\} \cup S$. Refer to the figure. And there are no additional edges coming into S other than the ones already included in the last term.

Since the capacity of the cut $(\{s\} \cup S, S^c \cup \{t\})$ is precisely the sum of the three red terms, running Edmonds-Karp (or the pseudo-polynomial Ford-Fulkerson) on the modified graph yields a Min-Cut in $\mathcal{O}(mn(m+n))$ time. Post-processing is easy: construct the cut corresponding to the Max-Flow / Min-Cut by selecting the component connected to s in the residual graph. Then remove s and recover the foreground section of the original image. Voilà!

6 Hardness & Impossibility

If you still say NP *stands for "not polynomial," don't tell anyone you studied with me.*

— Professor David Kempe

So far, we have been mostly concerned with developing *efficient* algorithms and optimizations over a variety of problems, and we focused on analyzing individual algorithms. But what about the problems themselves? Here, we take one step further and analyze the problems themselves — *Can we solve a particular problem efficiently, like in polynomial time? How "hard" can a problem get? Does there exist problems that are simply impossible to be solved?*

For the purpose of this course, **we will look at decision problems**, i.e., problems that can be simply answered by YES or NO. Also, problems are characterized by inputs and outputs (a decision problem is essentially a mapping of inputs to $\{\text{YES}, \text{NO}\}$). For convenience, **we assume that inputs are binary strings** (since such encoding is always possible).

It certainly feels as if decision problems are far less expressive than the class of problems we have been working on previously (e.g. optimization). However, this is not really the case, as for many problems, being able to solve the optimization problem implies the ability to solve the decision problem, and vice versa, though with a moderate amount of extra work. Consider, for example, the following problems related to MST:

- (DECISION) Given a cost-embedded graph G and target cost C, *does there exist a spanning tree with cost* $\leqslant C$*?*
- (OPTIMIZATION) Given a cost-embedded graph G, *what is the minimum cost* $C_{\min}$ *among all spanning trees?*

It is clear that if we have a solver for OPTIMIZATION, then DECISION becomes trivial: given any C, compare its value to the $C_{\min}$ outputted by the OPTIMIZATION solver.
The converse is slightly more complicated, but we can still adopt a binary search-alike method to slowly pinpoint the exact minimal cost of all spanning trees. Many iterations possibly, but we will get the right answer eventually.

Upshot: one loses a little, but not too much, by considering only decision problems. The benefits are also apparent: some things become significantly easier to state.

We haven't really defined what it means for an algorithm A to "solve" a problem X, so we'll fill the gap right now: we say A **solves** X if:

(1) A terminates on every input x; and

(2) $A(x)$ returns YES if and only if the correct answer for x is YES.

Efficient versus *Polynomial-time*?

So far, we have always implicitly associated the two words with each other. **Certainly, these two words are not efficient, but in practical situations, more or less the same.**

- Consider two algorithms, one runs in $10^{10} \cdot n^{100}$ and the other $10^{-10} \cdot 2^n$. The first is basically useless for any $n \geqslant 2$, whereas the second, albeit exponential, may be useful for some two-digit valued n's. It's clear that polynomial-time isn't always superior over exponential.

- However, the above is too much of an extreme case, since **typically, the polynomial-time algorithms we design usually have reasonable constants and exponents**. Hence, we almost always assume that polynomial is better than non-polynomial ones, e.g. exponential.
- In addition, improving the coefficient is usually a lot easier than improving along the runtime hierarchy (e.g. exponential to polynomial).

6.1 Defining P and NP

We informally introduce two classes of problems, **P** and **NP**:

- **P** (**p**olynomial time): problems that can be *solved* efficiently.
- **NP** (**n**ondeterministic **p**olynomial time): problems that can be *verified* efficiently.

So far in this course, we provided a polynomial algorithm for every problem we encountered, so they (problems, not algorithms) all belong to **P**.

NP does not stand for "not polynomial!!!" Traditionally it was defined via *nondeterministic Turing Machines*, but here we will use adopt a "modern" definition in terms of *efficient certifiers*:

Definition: Efficient Certifiers

An **efficient certifier** for a problem X is a polynomial-time algorithm $B(x, y)$ with inputs x (regular input for X) and y (**certificate**) such that

(1) If x is a YES-input for X, then for at least one y, with $|y| \leqslant r(|x|)$, such that $B(x, y)$ answers YES, and r is polynomial;

(2) If x is a NO-input for X, then for every y, $B(x, y)$ answers NO.

Some intuitions behind this definition:

- Suppose a professor assigns a proof-based problem, but didn't explicitly guarantee that the claim is provable. The problem itself may be a trick question so that it cannot be possibly proven.
- The certifier in this case are the TA's who check two things: (i) the input from professor: whether or not the problem is flawed, and (ii) the write-up from a student.
  - If the problem is well-defined (corresponding to the first case in definition), then the TA will quickly look over the student's response, in polynomial time w.r.t. the length of the write-up. The TA will accept the solution if and only if the student actually manages to write a correct, convincing proof.
  - If the problem is inherently flawed (second case in definition), then regardless of what the student attempts to prove, the TA rejects the solution, since it simply isn't right.
- Why do we need r to be polynomial w.r.t. $|x|$? Consider the scenario where a terrible student wrote a thousand-page long gibberish response (e.g. the certificate y is exponentially long w.r.t. $|x|$), and since the TA got so bored reading it, they derived the proof themselves. Then B's runtime is indeed polynomial in $|x|$ and $|y|$, but such action clearly violated academic integrity. We want to rule this out.

With efficient certifiers defined, **we define NP to be the class of problems with efficient certifiers**.

Since this may be confusing to understand at first, we give two concrete examples of efficient certifiers:

(1) (*Computing MST*) Given a graph with edge costs and a target cost, does there exist a spanning tree satisfying the cost constraint?

- Input: a graph G with edge costs $c(e)$ and target cost C.
- Certificate: a proposed spanning tree T of G.
- The certifier will check (i) whether T is indeed a spanning tree, and (ii) if it costs $\leqslant C$. Return YES if and only if both are satisfied.
- Connecting this to the definition of efficient certifiers:
  - If G does not have a spanning tree with cost $\leqslant C$, then (i) and (ii) cannot be both satisfied, so the certifier will always answer NO, as expected.
  - If G indeed has a spanning tree with cost $\leqslant C$, then (i) and (ii) will be both satisfied only when our input is convincing, i.e., we actually give the certifier some $G, \{c(e)\}$, and C that works.
- Clearly doable in polynomial time.

(2) (SUDOKU) Given a partially completed Sudoku puzzle, is it possible to complete it legally?

- Input: a partially completed Sudoku puzzle.
- Certificate: a proposed, completed solution.
- The certifier will check (i) whether the solution is valid (i.e. rows, columns, and square subgrids obey the rules) and (ii) whether it is a valid solution built on the partially completed original Sudoku puzzle (i.e. numbers must remain unchanged within cells initially occupied by the problem). Output YES if and only if both are satisfied.
- A few loops. Clearly doable in polynomial time.

Why classify problems as such? As humans, we like to interact with problems for which we can at least write down and verify solutions. The easier, more simplistic, more efficient, the more we like. That is, we love problems in **P**. Yet, most abstract problems one runs into as a computer scientist tend to be **NP** (or even worse).

Proposition: P ⊆ NP

P ⊆ **NP**. Informally, if a problem can be solved efficiently, it can be verified efficiently.

Proof. The proof really is just abusing the definition. Let $X \in \mathbf{P}$, and let a be an algorithm that runs in polynomial time and solves X. We define $B(x, y) := A(x)$, so in particular it completely ignores the certificate y. Apparently, $B(\cdot, \cdot)$ is polynomial w.r.t. both arguments, since $A(x)$ is polynomial w.r.t. $|x|$.
If x is a YES-answer for X, then $B(x, y) = A(x) =$ YES regardless of y (so it holds for at least one y). And since $A(x)$ takes polynomial time w.r.t. $|x|$, this process also takes polynomial time w.r.t. $|x|$ (so the requirement imposed by $r(\cdot)$ in the definition is satisfied).
If x is a NO-answer for X, then $B(x, y) = A(x) =$ NO for all y. □

Proposition

While we ensure solving **NP** problems in polynomial time, they can, indeed, be solved in exponential time.

Proof. We can derive a brute-force algorithm that tries all certificates y for a given x, and run $B(x, y)$ on all of them. If the answer is YES at least once, then we return YES. Otherwise return NO. Once again, abusing the definition.
The total number of candidates is $2^{r(|x|)}$, hence exponential. □

Bonus question. **Is P = NP? If a problem can be efficiently verified, can it be efficiently solved?** *No points awarded in-class, but you get everlasting fame and a million dollars*[34][35].

6.2 Karp Reduction & NP-Complete Problems

If **P** ≠ **NP**, then we would like to find problems that are in **NP** but not **P**. How so? Our best candidates would be the "hardest" problems in **NP**. Comparing runtime does not work, not to mention that the lower bound of many problems are constant changing due to researchers' works.

So far, we haven't systematically defined a way to compare the hardness of problems, but we have seen something related: *reduction*.

Recall our previous encounter with the maximum bipartite matching problem. We never bothered to directly solve it, but instead we took a detour and derived Ford-Fulkerson (or any Max-Flow / Min-Cut algorithm) and overkilled the matching problem. We were able to solve the matching problem as soon as we know how to solve the network flow problem. In this sense, the network flow problem is "harder" than the matching problem.

We now put this idea of reduction in the context of complexity theory: a reduction from X to Y is a polynomial-time algorithm that assumes a solution to Y as a subroutine and uses it to solve problem X. More formally:

Definition: Polynomial-Time Reduction (Karp)

We say X is **polynomial-time reducible** to Y, written $X \leqslant_p Y$, if we can solve X by calling a "black box" solver of Y a polynomial number of times.

Prof. Kempe's equivalent definition [which I find to be slightly more elusive]: a Karp reduction from X to Y is a polynomial f with the following properties:

- *The input of f is an input to X; the output of X is an input to Y. (Essentially, f transforms inputs of X into inputs of Y, since we are invoking Y to solve X.)*
- *x is a YES input for X, if and only if $f(x)$ is a YES input for Y.*

With reduction properly defined, we are able to define some "harder" problems: **we say a problem X is NP-hard if every problem $Y \in$ NP can be reduced to it** (i.e. $Y \leqslant_p X$ for all $Y \in$ **NP**), and **we say a problem X is NP-complete if it is in NP while also being NP-hard** (i.e. it is the "hardest" **NP** problem).

Even with this definition, is it no at all clear that **NP**-complete problems should exist. What if we have an infinite sequence of problems, $X_1, X_2, \cdots$, such that each problem is "strictly harder" than its predecessor, $X_1 \leqslant_p X_2 \leqslant_p \cdots$? Or what's worse, is it possible that $\leqslant_p$ is not a total order, so there exist non-comparable problems? Luckily, the answer to the first question is affirmative, and this result was first proven independently by Cook and Levin in 1971. It was a huge deal.

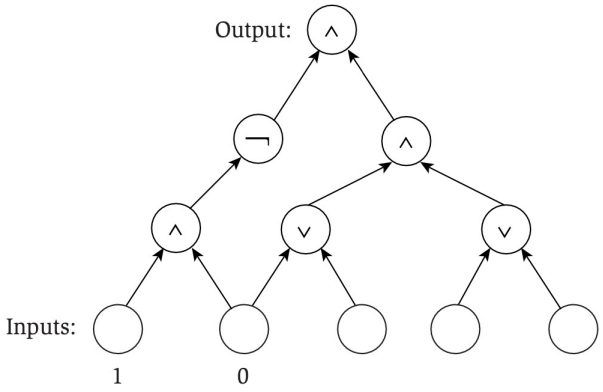
[34] `https://www.claymath.org/millennium/p-vs-np/`
[35] One of my favorite professors once cued the Riemann Hypothesis as the very last question of a final exam, and I quoted him here. (I also provided an ingenious solution.)

6.2.1 Cook-Levin Theorem: a First NP-Complete Example

Theorem: Cook-Levin Theorem (1971)

SAT is **NP**-complete.
(SAT / Satisfiability: given a boolean formula consisting of AND $\wedge$, OR $\vee$, and NOT $\neg$, is there an assignment of true/false to all variables that makes the formula true?)

Proof sketch (optional). First note that SAT is clearly **NP**: given a proposed variable assignment, we simply plug in the variables into the formula and evaluate whether the formula is true. Return YES if and only if true.
To prove that SAT is **NP**-complete, we need to prove that $X \leqslant_p$ SAT for every $X \in$ **NP**. We also won't give the original proof that involves nondeterministic Turing Machines[36], but rather one that uses efficient certifiers (and of course, technical details omitted).
Let $X \in$ **NP** be given, and let $B(\cdot, \cdot)$ be its efficient certifier. Given an input x, we'd like to ask, "is x a YES-answer to X?" Normally, we'd rely on an additional certificate y of length $\leqslant r(|x|)$, feeding both arguments into the certifier, and wait for the return value of $B(x, y)$.
How do we translate this into a SAT problem? We transform the problem into a circuit satisfiability problem. A sample circuit is shown below, where the bottom row contains the *source* nodes, each of which can be assigned 0 or 1, the top node contains the *output* (also 0 or 1), and all in-between nodes consist of the three Boolean operators (AND, OR, & NOT).



Source: KT 8.4

Specifically, we build a polynomial-size circuit with $|x|+|y|$ sources, where the first $|x|$ are hard-coded to represent the input x, and the last $|y|$ are variables, leaving freedom to the candidate certificates! We'll also carefully arrange the Boolean operators so that the circuit logic matches that of the algorithm. We will not prove this in detail here, but it should intuitively make sense — after all, any program is just a complicated composition of these simple, bitwise operations. (In the above example, the first 2 source nodes are hard-coded, leaving the other 3 to-be-assigned.)
It follows that x is a YES-input to X if and only if there is a way to assign values to the source nodes of the circuit so that it produces a 1, i.e., if and only if the circuit is satisfiable. Finally, I'll also skip the part where we prove that SAT and "circuit satisfiability" are equivalent, but you can hopefully spot some resemblance between the two. □

[36] Because I didn't have time to grind the proof. Winter break is too short!

The "hardness relation" $\leqslant_p$ also has an additional nice property: transitivity.

Lemma. If $X \leqslant_p Y$ and $Y \leqslant_p Z$, then $X \leqslant_p Z$.

Proof. To prove this, we use the "polynomial-time black-box" definition of Karp reduction. If $X \leqslant_p Y$ then given an input x for X, we can call a black-box solver of Y for a polynomial number of times, along with some other work, to obtain the result. In each of these black-box calls of Y, we also implicitly invoke a black-box solver of Z a polynomial number of times. Since the composition of two polynomials is still polynomial, when solving for X, we essentially invoked Z's black-box a polynomial number of times. This shows $X \leqslant_p Z$. □

Lemma. If X is **NP**-hard and $X \leqslant_p Y$, then Y is also **NP**-hard.

Proof. By definition, every $Z \in$ **NP** satisfies $Z \leqslant_p X$. Then by transitivity, $Z \leqslant_p X \leqslant_p Y \Rightarrow Z \leqslant_p Y$. Done. □

This is an extremely useful result! From now on, **if we want to show a new problem Y is NP-hard**, instead of reducing from all **NP** problems, **we only need to reduce from one known NP-hard problem X to Y, i.e., show $X \leqslant_p Y$**. [*Note the direction! It is an extremely common mistake that one picks the wrong direction for reduction. Always remember, $X \leqslant_p Y$ is a reduction from X to Y, and it shows Y is "harder."*]

6.3 Finding More NP-Complete Problems

In general, the workflow of proving Y is **NP**-complete is as follows:

(1) Prove $Y \in$ **NP** by giving an efficient certifier.

(2) Find a problem X that is known to be **NP**-hard. Goal: show $X \leqslant_p Y$. This is done by defining a function f such that (refer to the definition of Karp reduction):

- The input of f is an input to X; the output of f, $f(x)$, is an input to Y;
- the answer to x is YES if and only if the answer to $y = f(x)$ is YES; and
- f runs in polynomial time.

Below we will adopt this approach, and consider (and prove) a few **NP**-complete problems.
In particular, we will introduce four additional problems: 3-SAT, INDEPENDENT SET, VERTEX COVER, and SET COVER, and we prove their **NP**-completeness via a chain of Karp reductions:

$$\text{SAT} \leqslant_p \text{3-SAT} \leqslant_p \text{INDEPENDENT SET} \leqslant_p \text{VERTEX COVER} \leqslant_p \text{SET COVER}$$

This also serves as a roadmap for the remainder of this section. However, it is recommended that once you know the statement of each question, go over the proofs in reverse order: SET COVER first (where the proof is almost trivial — just renaming stuff), then VERTEX COVER (where the proof requires some insight, but not much actual work), and finally, INDEPENDENT SET (where the proof really requires some smart tricks). We won't formally prove the first reduction.

6.3.1 3-SAT / 3-Conjunctive Satisfiability / 3-CNF-SAT

> Given a formula in 3-conjunctive normal form, is it satisfiable?
> (A 3-conjunctive normal form is a formula with a big AND and many small OR's, where each clause is an OR of exactly 3 literals.)
> Example: $(x_1 \vee \overline{x}_2 \vee \overline{x}_4) \wedge (x_2 \vee \overline{x}_3 \vee x_4) \wedge (\overline{x}_1 \vee x_2 \vee \overline{x}_3)$.

Proof sketch of NP-completeness. It's clear that 3-SAT is **NP**. Just plug in the value of a proposed assignment and check whether the formula is now true.
To show 3-SAT is **NP**-complete, we attempt to establish SAT $\leqslant_p$ 3-SAT (say this out loud so you don't get confused: **we reduce from SAT to 3-SAT**). The basic idea is that we start with a general SAT formula, and we replace complicated expressions E with a new variable z, and add an AND with (E is equivalent to z). This reduces the nesting more and more, until we are left with a 3-SAT formula.
(This result holds for any k – SAT where $k \geqslant 2$.) □

A general rule of thumb: ***when attempting a reduction $X \leqslant_p Y$, if you find all parameters getting in your way, maybe you are reducing the wrong way.*** *Also:*

- *The easier X is, the easier the reduction: less cases we need to consider and invoke black-box of Y.*
- *The harder Y is, the easier the reduction: more parameters, so more likely we can exploit those parameters when invoking a black-box.*

6.3.2 Independent Set

> Given an undirected graph G and an integer k, is there an independent set $G \subset V$ of size at least k?
> (No two nodes in an independent set are connected by an edge.)

Proof of NP-completeness. To build an efficient certifier for Independent Set is **NP**, given a proposed set of nodes, we check (i) whether the set contains at least k nodes, and (ii) whether the set is independent, i.e., no two nodes are connected by an edge. If both turns out to be true, we return Yes. Else we return No. This clearly runs in polynomial time.

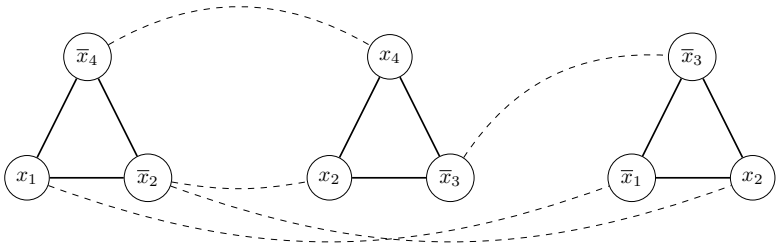
To show **NP**-completeness, we attempt the reduction 3-SAT $\leqslant_p$ Independent Set (say this out loud: **we reduce from 3-SAT to Independent Set**). Suppose we are given an instance of 3-SAT with n variables, $\{x_1, x_2, \cdots, x_n\}$, and k clauses, $\{C_1, C_2, \cdots, C_k\}$. We construct a graph consisting of $3k$ nodes as follows:

- Initially, the graph is broken into k smaller components, each having 3 nodes. *Each component corresponds to a clause: 3 nodes per component, 3 literals per clause.*
- Within each component, inter-connect all nodes, i.e., draw triangles!
- For each pair of nodes representing conflict literals (e.g. x_1 and $\overline{x}_1$), draw an edge connecting them.

The following diagram illustrates how we construct the graph corresponding to the formula given in the 3-SAT

example:

$$(x_1 \vee \overline{x}_2 \vee \overline{x}_4) \wedge (x_2 \vee \overline{x}_3 \vee x_4) \wedge (\overline{x}_1 \vee x_2 \vee \overline{x}_3)$$



Intuition: think of the possibility of finding an independent set of size k from this graph. Apparently, for each triangle, at most one node can be chosen, so any independent set of this graph cannot contain $> k$ nodes. This means that we have to pick one node from each triangle, subject to the additional constraint that we also don't pick both endpoints of any dashed edge.

Clearly, constructing this graph takes polynomial time, so it remains to prove that **[the 3-SAT formula is satisfiable] if and only if [the constructed graph contains an independent set S of size (at least) k]**.

To prove $\Rightarrow$, we suppose the original formula is satisfiable. In particular, each clause has at least one term evaluating to 1. We construct S by picking one such node from each triangle, resulting in $|S| = k$. Since the endpoints of a dashed line must have different values, and we only picked 1's, none of the selected nodes can possibly be connected by an intra-component dashed edge. And they clearly cannot be connected by inter-component solid edges either. Therefore S is independent, and it has size $\geqslant k$.

Conversely, suppose S is an independent set of size $\geqslant k$. By the italicized comment above, we must have $|S| = k$, nothing more. We assign values to the variables as follows. For each x_i:

- If neither x_i nor $\overline{x}_i$ appears as a label of any node in S, we don't care about its value — assign arbitrarily.
- Otherwise, exactly one between $\{x_i, \overline{x}_i\}$ appear as a label of a node in S. Set the variable to 1 if x_i appears in S; otherwise, set it to 0. This ensures that each triangle clause evaluates to 1, and hence the overall formula to 1 as well.

We've now shown that the original 3-SAT instance is satisfiable if and only if the constructed graph admits an independent size of at least k. And the transformation takes polynomial time. Done. □

6.3.3 VERTEX COVER

> Given G and $k \in \mathbb{N}$, does there exist at most k nodes such that each edge has at least one endpoint among these nodes?

Proof of NP-completeness. VERTEX COVER is clearly **NP**: given a certificate, a set S of nodes, we simply (i) check if $|S| \leqslant k$, and (ii) iterate through all edges and see if every one of them is incident on at least one node in S. Return YES if both are satisfied. Clearly polynomial.

To show **NP**-completeness, we attempt to show that INDEPENDENT SET $\leqslant_p$ VERTEX COVER (say this out loud: **we reduce from INDEPENDENT SET to VERTEX COVER**).

The entire transformation can be summed up via the following observation:

$$(S \text{ is an independent set}) \Leftrightarrow (\text{no edge has both endpoints in } S)$$
$$\Leftrightarrow (\text{each edge has} \geqslant 1 \text{ endpoint(s) in } S^c) \Leftrightarrow (S^c \text{ is a vertex cover}).$$

Therefore, for a given instance of INDEPENDENT SET, say graph G with n nodes, and threshold k,

$$(G \text{ has an independent set of size } \geqslant k) \Leftrightarrow (G \text{ has a vertex cover of size } \leqslant n - k).$$

Our transformation? Feed the same graph G and threshold $k' := n - k$ into VERTEX COVER. Done! □

6.3.4 SET COVER

> Given a set U and subsets $S_1, S_2, \cdots, S_n \subset U$, is there a collection of at most k of these subsets whose union cover U entirely?

Proof of NP-completeness. SET COVER is **NP**: given a certificate, a collection of sets, we check (i) whether there are at most k elements in it, and (ii) whether the union of these sets cover the universal set. Return YES if both are satisfied. Clearly polynomial.

To show **NP**-completeness, we attempt to show that VERTEX COVER $\leqslant_p$ SET COVER (say this out loud one last time: **we reduce from VERTEX COVER to SET COVER**).
Given an instance of the VERTEX COVER problem of graph $G = (V, E)$, we apply the following transformation:

- Set $U = E$. *Our goal: cover all of the edges.*
- For each $v \in V$: define S_v to be the collection of edges incident on v. (May need some kind of labeling.)
- Keep k unchanged.

And we ask, *does there exist (at most) k sets of form S_v whose union cover E?* Since all we did was basically renaming: E to U, edges to set elements, adjacency edge set to subsets, and so on, the question is completely equivalent to *does there exist (at most) k nodes whose union of adjacency edge set covers E*, or *does there exist (at most) k nodes that cover all the vertices?* Apparently this transformation is done within polynomial time, so we are done. □

6.4 Undecidable & Non-Computable Problems

In this section, we take our level of abstraction one step further and investigate the limits of computations (not about how much additional computational power is needed for astronomically large arithmetic operations, but rather, problems that are inherently "not computable"). Due to the nature of the topic, many solutions/proofs may even feel like begging the question — that is completely normal.

We begin by defining computability: we say a function f is **computable** if there exists a program P (written in any language, e.g. Python, C++) which correctly computes it: $P(x) = f(x)$ for all inputs x. Our assumptions here are very loose:

- We allow programs to not terminate on "bad" inputs, and we do not assume that $f(\cdot)$ is defined for all inputs.
- We write $P(x) \uparrow$ if $P(x)$ does not terminate; we write $P(x) \downarrow$ if it does.

Like before, we'd like to focus exclusively on decision problems. In the context of computability, this translates to sets. We say a set S is **decidable** if there exists a program P that can tell whether each element x is in S or not. In terms of computable functions, this means the *indicator function* of S, $\mathbf{1}[S]$, is computable.

Intuitively, one might tend to believe that every function is computable. If it appears not, simply give the program more time or memory and everything will work out. This turns out to be false! How do we realize this?

(1) First observation: **all programs are finite pieces of code**. Yes, there are infinitely many strings with finite length, but the cardinality is countably many. (We know that $\mathbb{N}$ is countable, and $\mathbb{N}$ can be viewed as the collection of all finite strings created out of an alphabet of 10 letters. Program codes are basically the same, except with a larger dictionary: alphanumeric characters and special characters.) **Countably many programs!**

(2) On the other hand, **there are uncountably many functions**. For each $x \in [0,1]$, we define $f_x : \mathbb{N} \to \mathbb{N}$ by $f_x(n) :=$ the n^{th} decimal digit of x. It is clear that different x's correspond to different functions, so in particular the carnality[37] of the collection of functions is at least as large as that of $[0,1]$, which, using *Cantor's diagonalization*, is known to be uncountable.

... and, if you recall results from CS170 or elsewhere, this implies that **most (in fact, "almost all"[38]) functions are uncountable**.

Most of these non-computable functions are boring, so we don't care anyways. However, **are there "interesting functions" which turn out to be non-computable / undecidable?**

6.4.1 The Halting Problem: a First Undecidable Problem

It would be really useful if we can have a program that detects infinite loops, or more specifically, bad inputs to another code snippet that leads to infinite loops. You surely have encountered this in your prior programming assignments. We hereby present the **General Halting Problem**:

> Given a program P and an input x, will $P(x)$ terminate?

The section title already spoiled everything for you — indeed, this function is undecidable. Specifically, we will look at a special case of this problem, which turns out to be also undecidable. Introducing the **Diagonal Halting Problem**, or just HALT:

> Given a program P, does $P(P)$ terminate?

(*It's not unusual to feed an entire program into another program as inputs. Think of your compilers, or* `valgrind` *from CS104. Historically, it has also been an important paradigm shift when we started to realize that source or even executable code can just be treated like data.*)

Theorem

HALT is undecidable. *So your dream is shattered — the perfect infinite loop detector does not exist.*

[37] I deliberately avoided the word "number:" when dealing with hierarchies of infinity, the word "number" becomes meaningless.

[38] No need to get overly technical here, but the closest analogy is that "almost all" real numbers are irrational. Informally, if you "randomly" pick a number $x \in [0,1]$ then "with probability 1," this number is irrational.

Proof. (*This proof feels somewhat like begging the question, like I mentioned earlier. So is basically every other proof from now on.*)
High-level idea: if a HALT-SOLVER *ever existed, then a new program could call it, find out what the program itself would do, and intentionally do the opposite. In retrospect, the* HALT-SOLVER *would have given the wrong answer.*
Assume for contradiction that a HALT-SOLVER exists, which returns 1 if $P(P)$ terminates and 0 if $P(P)$ doesn't. Now consider the following funny program:

Algorithm 15: Glitching the HALT-SOLVER

```
1 Function Destroyer(program/string P):
2     flag ← HALT-SOLVER(P)
3     if flag = 1 then
4         while true do nothing // infinite loop
5     else return 0
```

What if we call `Destroyer` on itself? In other words, letting $P :=$ the source code of `Destroyer`, what happens to $P(P)$ now?

(Case 1) *If* `Destroyer(Destroyer)` *terminates*: then HALT-SOLVER(`Destroyer`) returns 1, so we are stuck in the infinite loop... which means `Destroyer(Destroyer)` does not terminate. Contradiction!

(Case 2) *If* `Destroyer(Destroyer)` *does not terminate*: then HALT-SOLVER(`Destroyer`) returns 0, so we skip lines 3 and 4. As we reach line 5, `Destroyer(Destroyer)` terminates as it returns 0. Contradiction again.

Either way we have a contradiction. Therefore, HALT-SOLVER cannot exist in the first place. □

6.4.2 Many-to-One Reduction & More Undecidable Problems

Now that we have shown HALT is undecidable, we want to exploit this knowledge while trying to prove another problem undecidable. Hence, we define a reduction highly analogous to Karp reduction for complexity of problems:

Definition: Many-to-One Reduction / Mapping Reduction

A **many-to-one reduction** from X to Y is a function f satisfying the following:

* The input of f is an input to X; the output of X is an input to Y. (*Identical to that in Karp reduction.*)
* x is a YES input for X, if and only if $f(x)$ is a YES input for Y. (*Again, identical.*)
* f is computable and always terminates.

Notation-wise, we write $X \leqslant_m Y$ if such a reduction exists, and we say we reduce from X to Y.

We will now consider two undecidable problems that we can reduce from HALT: TOTAL and ODDEVEN.

Let TOTAL be the collection of programs P such that $P(x)$ terminates for all inputs x. **Claim: Total is undecidable.**

Proof. We will give a reduction f from HALT to TOTAL, so HALT $\leqslant_m$ TOTAL, proving the latter undecidable.

Algorithm 16: Proving TOTAL Undecidable

```
1 Function Total_destroyer(string x):
2     run P(P) // doesnt even look at your input
3     Return: 0
```

The process of rewriting this reduction always terminates — it simply copy pastes some code. It remains to show that P YES-input to HALT if and only if the transformed program is a YES-input to TOTAL:

- If $P(P)$ terminates, meaning it is a YES instance for HALT, then `Total_destroyer` will not get stuck on line 2, and in particular it returns 0. This holds for all input x, so the transformed program is TOTAL.
- Conversely, if $P(P)$ fails to terminate, then the transformed program gets stuck on line 2, regardless of the input string. Therefore it is not TOTAL (since it fails to terminate for at least one input).

Therefore the reduction is correct, and we are done. □

A second example:

> Let ODDEVEN be the collection of programs P such that $P(x)$ terminates if and only if x is even. **Claim: ODDEVEN is undecidable.**

Proof. Once again, we reduce from HALT. What properties do we want our transformation to have?

- It is computable and always terminates. Call the input program P and transformed program Q.
- If $P(P)$ terminates, then Q terminates for all even inputs but not for any odd input.
- If $P(P)$ doesn't terminate, then *either $Q(x)$ does not terminate for at least one even x, or $Q(x)$ terminates for at least one odd x.*

Now consider the following function:

Algorithm 17: Proving ODDEVEN Undecidable

```
1 Function OddEven_destroyer(int x):
2     run P(P)
3     if x is odd then
4         while true do nothing // infinite loop
```

It is clear that the transformation always terminates and is computable. In addition:

- If $P(P)$ terminates, then the program Q always reaches the `if` statement, after which it terminates if and only if the input is even, as desired.
- If $P(P)$ fails to terminate, then Q never terminates, so it is not an YES-instance for ODDEVEN.

Therefore the reduction is correct, and we are done. □

CSCI 270 Midterm Cheatsheet

December 8, 2022, updated Jan. 7, 2024

1. Stable Matching

- A **matching** is a graph in which each node is incident on $\leqslant 1$ edge.
- A **perfect matching** is ... on exactly one edge.
- A **bipartite** graph is one where nodes are divided into two sets, and edges only exist between the two sets, not inside either.
- Given a bipartite instance with rankings, a matching is **stable** if for each pair (u, s) *not* assigned (no edge), either u prefers its assigned node over s or s their assigned node over u.
- Stable matching is not unique: men A, B, women $1, 2$, A prefers 1, B prefers 2, 1 prefers B, and 2 prefers A, then $\{(A,1),(B,2)\}$ and $\{(B,1),(A,2)\}$ are both stable.

Gale-Shapley proposal algorithm: WLOG assume there are n men and n women, each with a ranking of everyone of the opposite gender (if numbers don't equal, we can create dummies with bottom rankings). The goal is to create a stable matching between men and women.

```
Start with empty assignment
While there is still single man:
  - Pick single man m
  - Let w be m's highest ranked woman not yet
      proposed
  - If w is single OR prefers m over her
      current partner:
    - Match m with w
    - If w had partner, he becomes single
  - Else: do nothing
```

Key facts of G-S:

- Once a woman becomes matched, she never becomes single again and her matches can only improve.
- The algorithm terminates. There cannot be a single man forever: if so, since #men = #women, there will be a single woman, so she was never proposed to. The man can just proposed!

Stability proof

Proof.
- Assume not. Let (m, w') and (m', w) be pairs with m, w preferring each other.
- m have once proposed to w, and it took place earlier than when m proposed to w'.
- m either got rejected or later dumped because of some other m'' that w preferred over m.
- But then w must have chosen m'' at some point. Since she ended up with m', $m' \geqslant m''$ for w, and so $m' \geqslant m'' > m$ for w, contradiction. □

Man-optimality of G-S: we first define $P(m)$:= the set of all women w that m can end up with in some stable matching, and we define the best valid choice, $b(m)$, to be the best choice in $P(m)$ according to m's ranking.

Theorem

Gale-Shapley returns a matching in which each m is matched with $b(m)$.

Proof.
- Suppose not, that some man is not with their best valid choice $b(\cdot)$ in G-S. At some point he must have been rejected/dumped by best valid choice.
- Look at the first time that some man m got dumped/rejected by some woman in $P(m)$. Call this woman w.
- The man m' that w rejects/dumps m for must have $m' > m$ on w's ranking.
- Since $w \in P(m)$, there exists stable matching M' with (m, w) matched. In this matching, let w' be the partner of m'.
- *If m' preferred w over w', then in M', m' and w would have resulted in the matching being unstable.* Therefore, m' prefers w' over w.
- In GS, m' ended up being with w, so he must have been rejected/dumped by w', and this happened even before w rejected/dumped m.
- We assumed (m', w') could be matched in a stable matching, so we have a contradiction that m being rejected by w is *not* the first time *some* man gets rejected by *some* woman in his $P(\cdot)$.
- Therefore G-S is man-optimal. □

2. Greedy Algorithms

Greedy algorithm in a nutshell: pick whatever choice seems best at the moment and address future problems later.

Example – interval selection: given intervals $[a_i, b_i]$, pick as many without overlap as possible.

Algorithm $\mathcal{O}(n \log n)$:

```
Sort intervals by non-decreasing finish times
    f(i)
R = all intervals, A = {}
while R not empty
  - let i be the index of earliest finishing
      intervals in R
  - add i to A
  - remove interval i and all intervals
      intersecting it from R
```

Greedy stays ahead: if $i(1) < i(2) < \ldots < i(k)$ and $j(1) < j(2) < \ldots < j(k)$ are the first k intervals picked by greedy and by any optimal solution, then the end time of $i(k) \leqslant$ the end time of $j(k)$. *In each stage, greedy performs no worse than the optimal solution.*

Example – job selection: Given, n jobs with deadlines d_i, $1 \leqslant i \leqslant n$. do all jobs and minimize max lateness. NOTE: optimal solution contains no rest between jobs.

Algorithm: do jobs in the order of their deadlines, from earliest to latest.

Optimality – adjacent inversion: if the greedy order is $G(i) = i$ for $1 \leqslant i \leqslant n$ and the optimal solution $\mathcal{O}$ is not identical, then there exist jobs i, j forming an *adjacent inversion*:

- job j is scheduled immediately after job i by optimal $\mathcal{O}$
- deadline of j is earlier than deadline of i ($d_j < d_i$).

Optimality – proof:

- Let G be greedy output and $\mathcal{O}$ any optimal solution.
- Pick adjacent, inverted jobs i, j in $\mathcal{O}$.
- Show switching jobs i, j in $\mathcal{O}$ preserves optimality.
- Recover G from $\mathcal{O}$, proven.

Example: MST construction: given a graph with distinct edge weights $G = (V, E)$, construct a MST (min cost connected subgraph of G). NOTE: MST cannot contain cycles.

Proposition 1: Cut Property

A **cut** is a partition (S, S') of V.
If an edge e is the cheapest among all edges crossing *some* cut (S, S') then $e \in$ *every* MST.

(Proof sketch: add e to form a circle and remove the original cut-crossing edge; this results in a lower total cost subgraph.)

Kruskal's algorithm:

```
Sort edges by increasing cost
For each edge e in this order
  Add e if it does not create a cycle
```

Proof.
- Any edge added is a min cost edge across some cut, so output ⊂ MST.
- Any two disconnected components have edges connecting them, and Kruskal adds $\geqslant 1$ such edge, so output is a spanning tree. □

Kruskal's algorithm runs in $\mathcal{O}(m \log m) + \mathcal{O}(m \log^* n) = \mathcal{O}(m \log m)$ using **Union-Find** for $\mathcal{O}(\log^* n)$ amortized lookup and merge.

Prim's algorithm $\Theta(m \log n)$:

```
Start with any S = {s}
While S ≠ V
  Find the cheapest edge e = (u, v) between S
      and S'
  Add e to tree, add v to S
```

Proof. Output ⊂ MST by cut property. Output obviously is a spanning tree. □

3. Divide & Conquer

Divide & Conquer in a nutshell:

- Take a problem instance I of size n
- Divide into smaller $I(1), \cdots, I(k)$.
- Solve small instances separately and return solutions $\text{Sol}(j)$.
- Post-process solutions to produce a big solution.
- If input small enough, directly solve.
- **Example**: merge sort. *Proof sketch*: prove correctness of `merge` by induction, then prove correctness of `MergeSort` by induction again.

Theorem: Master Theorem

Let $a \geqslant 1, b > 1$. Assume some recursion relation's complexity satisfies

$$T(n) = aT(n/b) + f(n) \qquad T(1) = \Theta(1).$$

(MT1) If $f(n) = \mathcal{O}(n^{\log_b a - \epsilon})$ for some $\epsilon > 0$, then $\boxed{T(n) = \Theta(n^{\log_b a})}$.

(MT2) If $f(n) = \Theta(n^{\log_b a})$, then $\boxed{T(n) = \Theta(n^{\log_b a} \log n)}$.

(MT3) If $f(n) = \Omega(n^{\log_b a + \epsilon})$ for some $\epsilon > 0$ and $af(n/b) \leqslant kf(n)$ for large n and some $k < 1$, then $\boxed{T(n) = \Theta(f(n))}$.

Intuition: (MT1) says $f(n)$ is overwhelmed by $n^{\log_b a}$ small tasks like $T(1)$; (MT2) says $f(n)$ and small tasks have similar workload; (MT3) says most work is done by $f(n)$.

Int mult – Karatsuba algorithm: cut n-bit x into two $n/2$-bit x^+ and x^- with $x = 2^{n/2}x^+ + x^-$. Same for y. Then

$$x \cdot y = 2^n x^+ y^+ + 2^{n/2} \underbrace{(x^+ y^- + x^- y^+)}_{= x \cdot y - x^+ y^+ - x^- y^-} + x^- y^-$$

has complexity

$$T(n) = 3T(n/2) + \Theta(n)$$

which by (MT1) has $T(n) = \Theta(n^{\log_3 2})$.

Example: closest pair of points: given $\{p_i\}_{i=1}^n = \{(x_i, y_i)\}_{i=1}^n$, find the pair of closes points w.r.t. Euclidean norm.

Algorithm (high-level sketch):

```
1  // Works to be done before starting recursion:
2  // Sort all points by x-coordinates and store
       in an array
3  // Sort all points by y-coordinates and store
       in another array
4
5  Partition points into L and R based on median
       of x-coord
6  Two recursive calls on L and R subsets
7  // 2T(n/2) work
8
9  Let δ be the smaller return value from the two
       cursive calls
10 S = set of points within δ from the middle line
11 // O(n) or O(log n) passing indices
12
13 Check close pairs between L and R
14 - Index and sort points p(i) in S by y-coord
       y(i)
15     - For each point index i:
16     - start with j = i + 1, stop when
           y(j) > y(i) + δ
17     - for each j, compute d(p(i), p(j))
18     - keep track of min d(p(i), p(j)), update δ
           when necessary
```

Remark. Given δ and i, the loop over j repeats at most 12 times due to the fact that points in L must be $\geqslant \delta$ apart and points in R must also be $\geqslant \delta$ apart. Then lines 14 to 18 takes $\mathcal{O}(n)$ time, giving

$$T(n) = 2T(n/2) + \mathcal{O}(n),$$

giving $\boxed{T(n) = \mathcal{O}(n \log n)}$ runtime to search for closest pair.

4. Dynamic Programming

DP in a nutshell: in order to do *this*, what do I need to do the previous step? DP is closely related to brute-force / backtracking but avoids unnecessary recomputation.

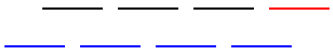
Example – Fibonacci: recursion blows up, but DP is $\mathcal{O}(n)$ using an additional array:

```
Fib[0] = Fib[1] = 1
for i in 2 : n+1
   Fib[i] = Fib[i-1] + Fib[i-2]
```

Example – interval selection w/ weights: given n intervals with start times $s(i)$, finish times $f(i)$, and additionally weights $w(i)$, find the set of non-overlapping intervals with maximal total weight.

Note: no greedy algorithm works. Consider the following example where every interval but the red one has weight 2.

- If red weight 3: optimal solution = pick first row
- If red weight 1: optimal solution = pick second row
- no way for greedy to decide which interval to pick first!



Key insight:

- If optimal solution includes interval i, then it does not include anything else intersecting i.
- If optimal solution does not include i, it is the same as the optimal solution for intervals $1, \ldots, i-1, i+1, \ldots, n$.
- Formula:

$$\mathrm{Opt}(j) = \max\{v_j + \mathrm{Opt}(p(j)), \mathrm{Opt}(j-1)\}$$

where $p(j)$ is the largest $i < j$ such that intervals i, j are disjoint.

Memoization in one sentence: storing values for future recursive calls, like the array used in Fibonacci previously.

Algorithm using memoization ($\mathcal{O}(n)$):

```
M[0] = 0
For j = 1, 2, ···, n
   M[j] = max{v_j + M[p(j)], M[j − 1]}
```

When to use DP?

- Only a polynomial number of subproblems.
- Solution to original problem can be easily computed from that to subproblems.
- Subproblems can be ordered from "smallest" to "largest" with easy recurrence.

Example – Subset Sums: given n jobs, each with w_i work time, and a cap W on total work time, maximize work time $\sum w_i$. Let $\mathcal{O}$ be optimal. Let $\mathrm{Opt}(n, W)$ be the optimal value on first n jobs and w total time. For job i:

- If $n \notin \mathcal{O}$, $\mathrm{Opt}(n, W) = \mathrm{Opt}(n-1, W)$. (Makes no difference to $\mathcal{O}$ if we exclude n.)
- If $n \in \mathcal{O}$, $\mathrm{Opt}(n, W) = w_n + \mathrm{Opt}(n-1, W - w_n)$. (Do the job, set total time allowance to $W - w_n$, combined with the optimal solution on first $n-1$ jobs given $W - w_n$ time.)
- To sum up:

$$\mathrm{Opt}(i, w) = \max(\mathrm{Opt}(i-1, w), w_i + \mathrm{Opt}(i-1, w-w_i)). \quad (*)$$

Algorithm (tabular, $\mathcal{O}(nW)$):

```
// First initialize n × W matrix M

For i = 1, 2, ···, n
   For w = 0, 1, ···, W
      Compute M[i][w] := Opt(i, w) using (*)
```

Example – knapsacks: given n items, each with value v_i and weight w_i, and a cap W on total weight, maximize $\sum w_i$, the value of items taken. Let $\mathcal{O}$ be an optimal solution. Let $\mathrm{Opt}(n, W)$ be the optimal value with first n items and total allowance W. For job i:

- If $n \notin \mathcal{O}$, $\mathrm{Opt}(n, W) = \mathrm{Opt}(n-1, W)$.
- If $n \in \mathcal{O}$, $\mathrm{Opt}(n, W) = v_n + \mathrm{Opt}(n-1, W - w_n)$.
- Combining:

$$\mathrm{Opt}(i, w) = \max(\mathrm{Opt}(i-1, w), v_i + \mathrm{Opt}(i-1, w-w_i)).$$

- A similar algorithm with an $n \times W$ array gives $\mathcal{O}(nW)$ runtime.

CSCI 270 Final Exam Cheatsheet

March 22, 2023, updated Jan. 7, 2024

Pre-MT Materials

1. Stable Matching

- A matching is **stable** if for each pair (u, s) not assigned, either u prefers its assigned node or s or s prefers its assigned node over u.
- Stable matching is *not* unique: if A prefers 1, B prefers 2, 1 prefers B, and 2 prefers A, then $\{(A,1),(B,2)\}$ and $(A,2),(B,1)$ are both stable.
- **Gale-Shapley**: (1) pick any single man m, (2) proposed to highest ranked women w not yet proposed, (3) if w is single OR prefers m over her current partner, match (m, w), (4) repeat until done.
- Properties of G-S: terminates, stable, & men-optimal.

2. Greedy Algorithms

- "Pick whatever seems best at the moment."
- "Greedy stays ahead" (proof technique): in each stage greedy performs no worse than optimal solution.
- **Interval selection**: given $[a_i, b_i]$, pick as many as possible without overlap. Greedy solution: sort by finish times from earliest to latest, and pick the interval as long as it does not conflict with earlier ones.
- **MST construction**: construct MST of $G = (V, E)$.
  - **Kruskal**: (i) sort edges by increasing cost, and (ii) iteratively pick the cheapest as long as it does not create a cycle.
  - **Prim**: start with any $S = \{s\}$; iteratively find and add the cheapest edge in cut (S, S').

3. Divide & Conquer

- "Break questions into smaller ones until they are small enough to be solved directly."
- Most familiar example: merge sort.

Theorem: Master Theorem

Let $a \geqslant 1, b > 1$. Assume some recursion relation complexity satisfies

$$T(n) = aT(n/b) + f(n), T(1) = \Theta(1).$$

(MT1) If $f(n) = \mathcal{O}(n^{\log_b(a-\epsilon)})$ for some $\epsilon > 0$ then $\boxed{T(n) = \Theta(n^{\log_b a})}$.

(MT2) If $f(n) = \Theta(n^{\log_b a})$ then $\boxed{T(n) = \Theta(n^{\log_b a} \log n)}$.

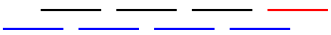
(MT3) If $f(n) = \Omega(n^{\log_b(a+\epsilon)})$ for some $\epsilon > 0$ with $af(n/b) \leqslant kf(n)$ for some $k < 1$ and sufficiently large n, then $\boxed{T(n) = \Theta(f(n))}$.

- Intuition: (MT1) says $f(n)$ is overwhelmed by $n^{\log_b a}$ small tasks like $T(1)$; (MT2) says $f(n)$ and small tasks have similar workload; (MT3) says most work is done by $f(n)$.
- **Closest pair of points**: given n points on $\mathbb{R}^2$, find a pair with closest Euclidean distance?

  Solution in a nutshell: divide points to left and right, and do some math in the middle.

4. Dynamic Programming

- "In order to achieve optimal solution at this step, what must be done by the previous step?"
- Memoization: storing values for future recursive calls.
- **Interval election with weights w_j**:
  - Greedy doesnt work — if all non-red intervals have weight 2, the optimal solution depends on the weight of the red one.



  - Observation: (i) if OPT includes interval i, then it does not include any intersecting i; if OPT does not, pretend as if interval i never existed.
  - $\text{OPT}(j) = \max\{w_j + \text{OPT}(p(j)), \text{OPT}(j-1)\}$ where $p(j)$ is the largest $i < j$ where intervals i, j are disjoint. Using memoization this can be done with $\mathcal{O}(n)$ time and space.
- **Knapsack**: given items with weights w_j and values v_i, maximize the total items taken subject to a weight constraint W. For each integer $0 \leqslant w \leqslant W$ and $1 \leqslant j \leqslant n$ consider the subproblem $\text{OPT}(i, w)$:

  $\text{OPT}(i, w) = \max\{\text{OPT}(i-1, w), v_i + \text{OPT}(i-1, w-w_i)\}$.

  Since we have two variables, we implement this using a **tabular** method, keeping track of a $n \times W$ array.
- **String alignment** (sketch): three possibilities for $\text{OPT}(i, j)$, the optimal cost of aligning first i letters of word x with first j letters of word j:
  - $x[i]$ aligns with $y[j]$: no extra cost compared to $\text{OPT}(i-1, j-1)$.
  - $x[i]$ aligns with a blank: extra cost of removing a letter from x compared to $\text{OPT}(i-1, j)$.
  - $y[j]$ aligns with a blank: extra cost of adding a letter to x compared to $\text{OPT}(i, j-1)$.
  - Take minimum, and set up base cases...

5. Max-Flow, Min-Cut, & Duality

Definition: Flow (on graph)

Given a graph $G = (V, E)$, an $s-t$ **flow** is a function $f : E \to \mathbb{R}$ satisfying

- (conservation) for all nodes $v \neq s, t$,
$$\sum_{e \text{ out of } v} f(e) = \sum_{e \text{ into } v} f(e), \text{ and}$$
- (constraints) $0 \leqslant f(e) \leqslant c(e)$ for all edges e, where $c(e)$ is the capacity of e.

We define the **value** of a flow, $\nu(f)$, to be the total flow out of the source (or into the sink).

Definition: Cut (on graph)

An $s-t$ **cut** (S, S^c) is a partition of V with $s \in S$ and $t \in S^c$. An edge $e = (u, v)$ **crosses** the cut if $u \in S, v \in S^c$. The total **capacity** of the cut is defined by $c(S, S^c) := \sum_{e \text{ crosses } (S,S^c)} c(e)$.

Theorem: Max-Flow / MinCut (Ford-Fulkerson)

Given a graph G with nonnegative edge capacities and nodes s, t:

- max $s-t$ flow and min $s-t$ cut can be found in polynomial times,
- if $c(e)$ are integers, then the algorithm outputs an integer max-flow, and
- max-flow value equals min-cut capacity.

Algorithm 18: Ford-Fulkerson Algorithm

```
Start with zero flow, i.e. f(e) = 0 for all edge.
while residual graph G(f) contains an s – t path do
    Let P be one such path in G(f) [BFS/DFS]
    Augment f along P
Function residual_graph(G = (V, E)):
    Start with empty G(f)
    for each edge e = (u, v) do
        if f(e) > 0: add (v, u) to G(f) with
          capacity f(e), backward edge.
        if f(e) < c(e): add (u, v) to G(f) with
          capacity c(e) − f(e), forward edge.
Function augment(f, P):
    Find ϵ, smallest residual capacity along P
    for each edge e in path P do
        if e is forward: increase f(e) by ϵ.
        if e is backward: decrease f(e) by ϵ.
```

- **Duality**: since any flow and any cut on G satisfies $\nu(f) \leqslant c(S, S^c)$, we have $\max \nu(f) \leqslant \min c(S, S^c)$ so if equality can be attained, we must have simultaneously found max-flow and min-cut.
- (Applcation) **maximum bipartite matching**: to find a maximum bipartite matching between sets X and Y, we create a source s and sink t. We connect s to each node in X with capacity 1, v to each node in Y with capacity 1, and fully connect nodes between X and Y with infinite capacity. Now find a max-flow.
- [See below] Problem with Ford-Fulkerson: it is pseudopolynomial — potentially bad if huge edges!
- Workaround: always choose the widest path (weakly polynomial) or always choose the shortest path (**Edmonds-Karp** algorithm, strongly polynomial, $\mathcal{O}(m^2 n)$).
- **Pseudopolynomial** runtime example: DP Knapsack with $W = 2^{100}$, where W can be stored in 100 bits, but running it takes ... (recall $\mathcal{O}(nW)$). Still, this *is* polynomial in n, W. [Weakly/strongly polynomial not included, as they were barely mentioned in lectures.]

Application: Image Segmentation

- Question: given an image (many pixels), is each particular pixel foreground or background?

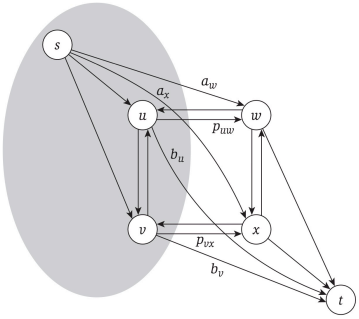
- Formulation: given $G = (V, E)$, each node v has a foreground score $a(v) \geqslant 0$ that we get from labelling v with "A", a background score $b(v) \geqslant 0$ for labelling it with "B". For each edge $e = (u, v)$, penalty $p(e) \geqslant 0$ for labelling u, v with different labels. Find a partition (S, S^c) maximizing

$$Q(S, S^c) = \sum_{v \in S} a(v) + \sum_{v \notin S} b(v) - \sum_{\substack{e=(u,v) \\ u \in S, v \notin S}} p(e).$$

- Solution: since

$$Q(S, S^c) = \sum_v a(v) + \sum_v b(v) - \left(\sum_{v \notin S} a(v) + \sum_{v \in S} b(v) + \sum_{\substack{e=(u,v) \\ v \in S, u \notin S}} p(e) \right)$$

it suffices to minimize the red terms, which we call $R(S, S^c)$. But this is just a min-cut: (i) add source s, sink t, (ii) connect s with each v with capacity $a(v)$, (iii) connect all v to t with capacity $b(v)$, (iv) find a min $s-t$ cut $(\{s\} \cup S, \{t\} \cup S^c)$, and (v) return S. See the image from KT's book.



6. NP-Completeness

- We focus on **decision problems**, i.e., those outputting YES or NO.
- Decision problems are polynomial-time equivalent to optimization ones (compute the optimum): do binary search for the right value.
- **P**: problems solvable in polynomial time. **NP** stands for **nondeterministic polynomial time** (Kempe said don't tell anyone you studied with him if you still say NP is "not polynomial").
- Another definition for NP: problems with **efficient certifiers**, i.e., "a proposed solution (**certificate**) can be verified in polynomial time." Formal definition:

Definition: Efficient Certifier

An efficient certifier for a problem X is a polynomial-time algorithm $B(x, y)$ with inputs x (regular input for X) and y (certificate) such that

(1) if the correct answer for x is YES, then there exists one y with $|y| \leqslant r(|x|)$ such that $B(x, y)$ answers YES;

(2) if the correct answer for x is NO, then for every y, $B(x, y)$ answers NO; and

(3) r is polynomial.

- Efficient certifier example:

  X: "given graph G and target cost C, is there a spanning tree with cost $\leqslant C$?"

  Certificate: y, a proposed spanning tree.

  Verification process: check (i) y is a spanning true, and (ii) cost of $y \leqslant C$. If both are satisfied, return YES, else return NO.

  Efficient? Show this verification process runs in polynomial time.

Proposition: P ⊂ NP

Proof. Let $X \in P$ and let A be an algorithm that solves X in polynomial time. Let B be the trivial certifier such that $B(x, y)$ copies $A(x)$'s answer.

- *Clearly efficient.*
- *If x is a YES answer for X, then $B(x, y)$ answers YES (or any, so at least one y).*
- *If x is a NO answer for X, then $B(x, y)$ answers NO for all y.* □

Definition 2: Polynomial-Time Reduction (Karp)

We say X is **polynomial-time reducible** to Y, written $X \leqslant_p Y$, if we can solve X by calling a "black box" solver of Y a polynomial number of times.

Prof. Kempe's equivalent definition [which I find to be slightly more elusive]: a Karp reduction from X to Y is a polynomial f with the following properties:

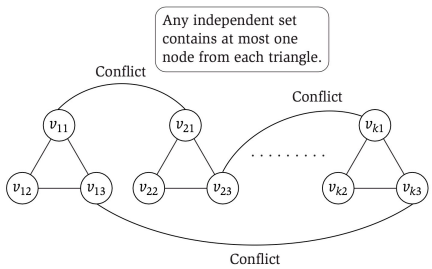
- *The input of f is an input to X; the output of f is an input to Y.*
- *x is a YES input for X, if and only if $f x$ is a YES input for Y.*

- $\leqslant_p$ is transitive.
- Think of $\leqslant_p$ as "level of hardness": bigger means "harder."
- A problem is **NP-complete** if (i) it is NP and (ii) for all $X \in$ NP, $X \leqslant_p$ this problem. A problem is **NP**-hard if it satisfies (ii) [but not necessarily (i)].

NP-Complete Problems

- To show Y is NP-complete:
  - Show Y is NP by giving an efficient certifier.
  - Unless you are crazy, pick a known NP-complete problem X, and reduce from X to Y, i.e., showing $X \leqslant_p Y$.
- Cook-Levin: **SAT** and **3SAT** are NP-complete.
- 3SAT example: is $(x_1 \vee \overline{x}_2 \vee \overline{x}_4) \wedge (x_2 \vee x_3 \vee x_4)$ satisfiable? [$\vee$ as "or", $\wedge$ as "and"]
- The following are NP-complete:
  - **INDEPENDENT SET**: "given G and $k \in \mathbb{N}$, does there exist k nodes such that no two nodes are connected by an edge?"

    3SAT $\leqslant_p$ INDEPENDENT SET: [image from book] conflict refers to negations of literals, e.g. x_4 and $\overline{x}_4$ from the 3SAT example above.



  - **VERTEX COVER**: "given G and $k \in \mathbb{N}$, does there exist k nodes such that each edge has at least one endpoint among these nodes?"

    INDEPENDENT SET $\leqslant_p$ VERTEX COVER: (S is an independent set) iff (each edge has $\geqslant 1$ endpoint in S^c) iff (S^c is a vertex cover), so given INDEPENDENT SET with G, k, transform to $G', \{S_v\}, n - k$, with S_i the set of nodes adjacent to v.
  - **SET COVER**: given universal set U, subsets $S_1, \ldots, S_n \subset U$, and $k \in \mathbb{N}$, does there exist k sets whose union cover U?

    VERTEX COVER $\leqslant_p$ SET COVER: given inputs G, k to VERTEX COVER, transform into $V, \{S_v\}, k$ like above.

7. Undecidability

- A (yes/no) function is **computable** if there exists a program that tells if each input leads to YES or NO.
- Uncountable functions, countable programs (finite strings) ⇒ "most" functions are not computable.
- **HALT** (diagonal halting): given a program P, does $P(P)$ terminate?

  HALT is undecidable. Suppose for contradiction that a program SOLVER solves HALT. Define

```
1  int Destroyer (string P) // program P {
2        if (Solver says P terminates):
3           while (true); // infinite loop
4        else return 0;
5  }
```

  If DESTROYER(DESTROYER) terminates then the program is stuck at line 3 and never terminates; if DESTROYER(DESTROYER) doesn't, the program terminates at line 4. Contradiction!
- **Many-to-one reduction**: $X \leqslant_m Y$ if there exists some f mapping inputs to X to inputs to Y such that (i) YES instances map to YES instances, (ii) NO to NO, and (iii) f always terminates. (Similar to Karp!)
- **TOTAL**: "does a program P terminates for all inputs?"
  - TOTAL is undecidable: HALT $\leqslant_m$ TOTAL.
  - Reduction:

```
1  int Q (string x) {
2        Run P(P) // doesn't even look at x
3        return 0;
4  }
```

  - If $P(P)$ terminates, then $Q(x)$ terminates and returns 0 for all x, i.e., Q is TOTAL.
  - If $P(P)$ does not, then $Q(x)$ does not terminate for any (in particular some) string, so Q is not TOTAL.

GOOD LUCK!