CPSC 320 Study Guide

Sasha Avreline

version of August 7, 2020

Current Status. Final version for now.

Notes. This guide was written by Sasha Avreline, a former BCS program teaching assistant, over the summer of 2020 and is meant to serve as a supplement and as a study tool to those taking CPSC 320, UBC's second algorithm's course. Examples used here are believed to be quite standard and widely available examples in the subject matter and are not meant to give away solutions to any assigned course work problems. Also see acknowledgments. Any solutions or lengthy code snippets contained in this document must be referenced appropriately when used elsewhere.

Contents

1 Stable Matching Problem

Suppose that we have two sets of entities that need to be matched up. Those could be applicants and jobs, men and women, users and servers, etc. Each entity ranks the opposing entities using a preference list. We do not necessarily care to come up with a matching that perfectly satisfies everyone's preference list in the best way possible, but what we do care is to come up with a *self-enforcing* or a *stable* matching. Essentially, we would like to see everyone get matched up in a way that discourages any further movement, so that the matching process actually terminates.

To begin, we will solve the stable matching problem (SMP) for the very simple case where both sets are of size n and each entity in each list wants to be matched with just a *single* entity from the opposing list. We will see that once we have a solution for this simple case, it will be easy to *reduce* more complicated problems to this simple case and to just reapply the solution we already have. Historically, much of literature and the textbook denotes the sets of entities as men and women when illustrating this simple problem, the notation used here is consistent with the one used in the textbook for now.

1.1 Terminology

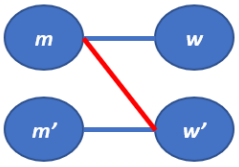
Let

- $M = \{m_1, \ldots, m_n\}$ be the set of n men.
- $W = \{w_1, \ldots, w_n\}$ be the set of n women.
- $M \times W = \{(m, w) : m \in M, w \in W\}$ be the set of all possible ordered pairs of men and women, a notation that will be used to denote various match-ups.

1. A **matching** S is a subset of $M \times W$ such that every member of M and every member of W appears in *at most* one pair of $M \times W$.

2. A **perfect matching** S' is a subset of $M \times W$ such that every member of M and every member of W appears in *exactly* one pair of $M \times W$.

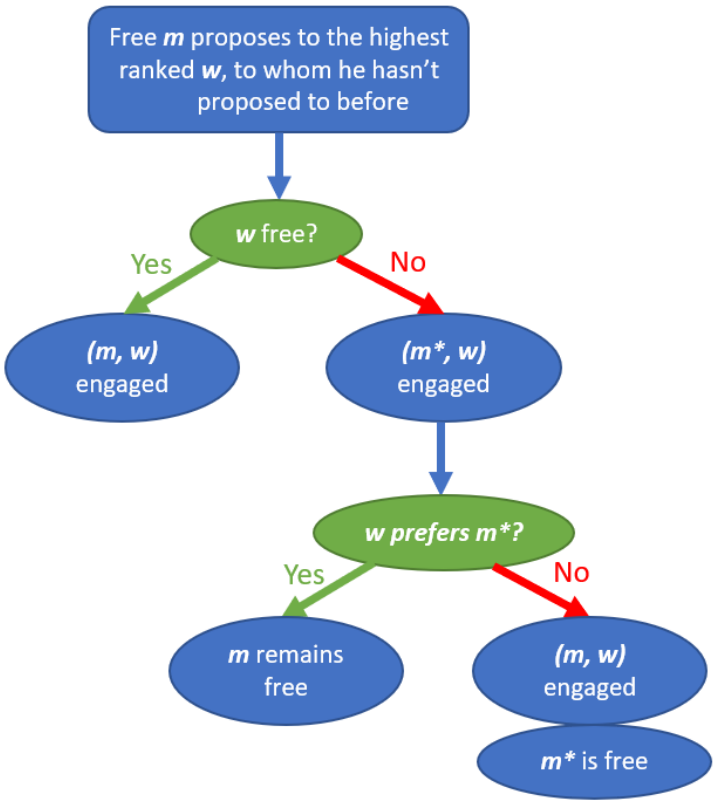
3. A pair (m, w') is an **instability** if
   - m prefers w' more than his current partner w and
   - w' prefers m more than her current partner m'



4. A **stable matching** S' is a matching that is both
   - perfect
   - has no instabilities

1.2 The Gale-Shapley Algorithm

```
Initially all m in M and w in W are free

while (there is a man m who is free && has not proposed to every woman) {
    Choose such a man m
    Let w be the highest-ranked woman in preference list of m to whom m has not
        proposed to before

    if (w is free)                      { (m,w) are engaged }
    else {

        w is currently engaged to m*
        if (w prefers m* to m)          { m remains free }
        else {

            w prefers m to m*
            (m,w) are engaged
            m* is free

        }
    }
}

return the set S of engaged pairs
```



1.3 Proof of Gale-Shapley Algorithm

LEMMA 1. Any $w \in W$ remains engaged from the point at which she receives her first proposal; and the sequence of partners to whom she is engaged to only gets better.

LEMMA 2. The sequence of women to whom $m \in M$ proposes only gets worse.

Proofs. Everything here is quite clear from the algorithm itself. Once a woman becomes engaged, if someone else proposes to her, she gets to pick the better of her current partner and the one who is proposing to her. When a man proposes to women, he starts from the highest ranked woman on his preference list; and, if rejected, must move down the list. □

LEMMA 3. The algorithm terminates after at most n^2 iterations.

Proof. Each iteration consists of a man proposing to a woman he hasn't proposed to before. Since there are n men and n women, we can have at most n^2 proposals. By the second condition on the while loop, once all proposals are done, the while loop must exit. □

LEMMA 4. If m is free, then there is a woman to whom he has yet to propose to.

Proof. Proceed by contradiction. Suppose that there exists a free m who has already proposed to every woman on his list. Since his list includes all women in some order, then every woman has received a proposal, and by lemma 1 must remain engaged. But there are n men, so some woman will be engaged to m and m is not free. ↯ □

LEMMA 5. The set S is a *perfect matching*.

Proof. It is clear that the algorithm returns a matching. In particular, for there to be a matching, every man and woman must appear in *at most* one pair and this is essentially guaranteed by the algorithm itself [whenever a man proposes to a woman who is already engaged, she must breakup to get a new partner].

Next, we show that the algorithm returns a perfect matching. A perfect matching means that every man and every woman appear in *exactly one* pair in the matching. So to go from "at most" to "exactly one" we just need to ensure everyone is matched when the algorithm terminates. Proceed by contradiction and suppose this was not the case and WLOG we had a free man remaining at termination. Lemma 4 showed this cannot happen. ↯ □

LEMMA 6. The set S is a *stable matching*.

Proof. We already seen that the set S is a perfect matching and it remains to show that the set is a stable matching. Proceed by contradiction and suppose that at termination there exists an instability. In particular, there are two pairs (m, w) and (m', w') such that

(1) m prefers w' to w and

(2) w' prefers m to m'.

Proceed by cases on the question: did m propose to w' at some point earlier in the process? If he didn't, then w appears on his preference list higher than w' which contradicts (1).

So suppose m did propose to w' at some point earlier in the process. Then, at that point, w' must have rejected him in favor of some other man m'' whom she prefers more than m [it could be that $m'' = m'$ or m'' is a man ranked lower than m']. This contradicts (2). ↯ □

1.4 Valid Partners and Uniqueness

While for some sets we end up with a unique stable matching, for other sets there are multiple valid stable matchings. The Gale-Shapley algorithm will return just one of them, the same one everytime. The fact the the algorithm always returns the same matching is an important result, especially when using the algorithm in an asynchronous programming environment. We will now discuss the specifics.

For example, suppose that

- m prefers w to w'
- m' prefers w' to w
- w prefers m' to m
- w' prefers m to m'

Then

- $\{(m, w), (m', w')\}$ is the stable matching returned by the Gale-Shapley algorithm
- $\{(m, w'), (m', w)\}$ is an another valid stable matching

Terminology

1. w **is a valid partner of** m if there exists some stable matching that contains (m, w).
2. w **is the best valid partner of** m if
   - w is a valid partner of m and
   - no woman whom m ranks higher than w is a valid partner of his
3. the set S^* of men and their best valid partners is: $S^* = \{(m, \text{best}(m)) : m \in M\}$

Theorem. Every execution of Gale-Shapley algorithm returns the set S^*.

Proof. Proceed by contradiction. Suppose that there is a set of men and women and some execution $\mathcal{E}$ of the algorithm that results in stable matching S in which there exists some man who is not paired with his best valid partner. We recall that in the algorithm men propose in order of decreasing preference, so in this case, some man must been rejected by some woman at some point. Consider the ***first*** time such an event took place, call the man m and his best valid partner $w = \text{best}(m)$. It must be that w is engaged and there is a pair (m', w) where w prefers m' over m.

Since rejection of m was the first rejection during the execution $\mathcal{E}$, then m' must not been rejected by any woman when he became engaged to w. So w must be at the ***top*** of m''s preference list as men propose in decreasing order of preference.

However, since m is a valid partner of w, there exists another stable matching S' that:

- by definition, contains the pair (m, w), and
- in which the man m' must also be paired with someone in S, call this woman $w' \neq w$

However, as we have seen,

- m' prefers w over w' (as w is at the top of m''s preference list) and
- w prefers m' over m (as she rejected m when he proposed)

so we get an instability and contradict the assumption that S' was stable. Since the specific situation considered here leads to a bogus matching S', the situation cannot exist. ↯ □

Corollary. In a stable matching S^*, each woman is paired with her worst valid partner.

Proof. Proceed by contradiction. Suppose there exists a pair (m, w) in S^* such that m is not the worst valid partner of w. Then there is a stable matching S' which

- contains pair (m', w) where m' is ranked lower than m by w (as m was not the worst)
- contains pair (m, w') where m prefers w to w' (as w was his best, by definition of S^*)

However, then (m, w) is an instability in S', contradicting the assumption that S' was stable. As before, the specific situation considered here leads to a bogus matching S', so the situation cannot exist. ↯ □

ASIDE: PROOF STRATEGIES

⋆ When we aim to show that a matching is stable, we proceed by contradiction and consider some instability (m, w'). We then consider the cases where (1) m and w' came into contact with each other at some point and where (2) they didn't. Out of each case we contradict the existence of instability.

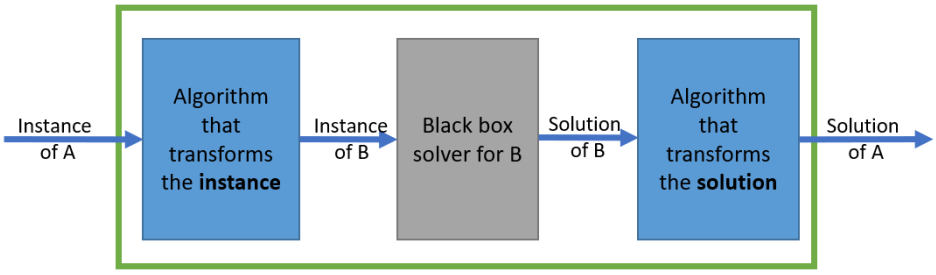
⋆ When we aim to show that an algorithm returns some very specific matching (i.e. all men are paired with their best valid partners), we proceed by contradiction and suppose that for at least one pair in the matching the property doesn't hold. Since partners were valid, there must be an another stable matching S' in which they are paired. We then proceed to show that if that was the case, there would be an *instability* in S'. In doing so, we must show that both parts of the instability definition hold.

1.5 Reductions

We came up with and proved the validity of an algorithm to solve the SMP in its simplest form. There are many variations and extensions of the SMP and for most of them we could come up with an algorithm that reduces their instances to an instance of SMP in its simplest form.

We could follow the process shown below for SMP or for any other problem. Once we came up with a reduction, we could either

- prove that if the solution of B is correct, then solution of A must also be correct, or
- prove that if the solution of B is incorrect, then solution of A must also be incorrect.



Important points to keep in mind are:

(1) the run times of transformations should not exceed the run time of obtaining a solution to B (we should always aim for transformations to be at most of polynomial run time)

(2) it is not always the case that a transformation is possible, is valid or useful

1.6 Implementation of the Gale-Shapley Algorithm

Suppose we have an instance of the SMP with n men and n women. We can then use the following data structures to represent this instance:

- *Preference lists of men and women:*
  `ManPref[m,i], WomanPref[w,i]` ($n \times n$ arrays)
  given a man or a woman, indices $[m, i]$ and $[w, i]$ to point to the woman or man that is in the i-th position of his or her preference list

- *Next woman to propose to:*
  `Next[m]` ($n \times 1$ array)
  for each man, we store the number of the next woman in his preference list to propose to

- *Current partner of a woman:*
  `Current[w]` ($n \times 1$ array)
  for each woman, we store the number of the man she is currently engaged to

- *List of free men:*
  `FreeMan` (linked list)
  in this case the linked list implementation is best as we can add, delete men off the list, know when the list is empty and use the `next` pointer to access the next free man

- *Relative Ranking of Men:*
  `Ranking[w,m]` ($n \times n$ array)
  index $[w, m]$ stores the position that m holds in w's preference list; this is a more useful version of `WomanPref[w,i]` that we can use when a woman checks if she should dump the man she is currently with in favor of the one who is proposing to her

`ManPref`

0	1	0	2	4	3
1	3	4	0	2	1
2	4	2	1	0	3
3	1	0	4	3	2
4	4	1	3	0	2

`WomanPref`

0	1	2	4	0	3
1	0	1	4	2	3
2	1	0	2	3	4
3	4	0	2	1	3
4	3	0	2	4	1

`Ranking`

	0	1	2	3	4
0	3	0	1	4	2
1	0	1	3	4	2
2	1	0	2	3	4
3	1	3	2	4	0
4	1	4	2	0	3

`Next`

0	1	2	3	4
0	0	0	0	0

`Current`

0	1	2	3	4

`FreeMan`

$\rightarrow \boxed{0} \rightarrow \boxed{1} \rightarrow \boxed{2} \rightarrow \boxed{3} \rightarrow \boxed{4}$

So in the rankings array, we see that entry $[0, 0]$ is 3. This means that for woman no. 0, man no. 0 is her 4th preferred man (taking 0-based indexing into account) as is the case in the woman's preference table.

1.7 Example 1: Incomplete Ranking Lists

The Problem. Suppose that we still have a stable matching problem with n men and n women; however, now each man and each woman is able to rank just a subset of the other group. In other words, the preference list for each person i consists of just some k_i entries where $0 \leq k_i \leq n$.

Reduction to Standard SMP. As before, let $M = \{m_1, \ldots, m_n\}$ and $W = \{w_1, \ldots, w_n\}$. Let $\mathcal{W}_i$ be the ordered list of women that a man i prefers and let $\mathcal{M}_i$ be the ordered list of men that a woman i prefers.

To reduce to the standard SMP, we create n "dummy" men denoted as $M_d = \{u_1, \ldots, u_n\}$ and also create n "dummy" women denoted as $W_d = \{v_1, \ldots, v_n\}$. We modify the existing preferences lists and create new preference lists as follows:

- for each man i: $\mathcal{W}_i$, followed by W_d, followed by $W - \mathcal{W}_i$.
- for each woman i: $\mathcal{M}_i$, followed by M_d, followed by $M - \mathcal{M}_i$.
- for each "dummy" man i: W, followed by W_d.
- for each "dummy" woman i: M, followed by M_d.

At this point each man and each woman has a preference list consisting of $2n$ entities and we can run the standard Gale-Shapley algorithm to solve this problem. Once we have a solution to the modified problem, we remove any "dummy" man or woman from that solution by saying that any man or woman married to a "dummy" man or woman is single.

Reduction to SMP required modification of $2n$ ranking lists as well as creation of $2n$ ranking lists, with $2n$ entries in each list. Therefore we need to perform up to $(2n + 2n)(2n) = 8n^2$ elementary operations and get a run time of $\mathcal{O}(n^2)$ for the reduction. In order to retrieve the solution, we make a single pass through $2n$ solution pairs, so the run time is $\mathcal{O}(n)$.

An Example.

Men	1	2	3	4
m_1	w_3	w_1	–	–
m_2	w_1	–	–	–
m_3	–	–	–	–
m_4	w_2	w_3	w_4	–

Women	1	2	3	4
w_1	m_2	m_1	–	–
w_2	m_3	m_4	m_1	m_2
w_3	m_2	m_3	–	–
w_4	m_4	m_2	–	–

Men	1	2	3	4	5	6	7	8
m_1	w_3	w_1	v_1	v_2	v_3	v_4	w_2	w_4
m_2	w_1	v_1	v_2	v_3	v_4	w_2	w_3	w_4
m_3	v_1	v_2	v_3	v_4	w_1	w_2	w_3	w_4
m_4	w_2	w_3	w_4	v_1	v_2	v_3	v_4	w_1
u_1	w_1	w_2	w_3	w_4	v_1	v_2	v_3	v_4
u_2	w_1	w_2	w_3	w_4	v_1	v_2	v_3	v_4
u_3	w_1	w_2	w_3	w_4	v_1	v_2	v_3	v_4
u_4	w_1	w_2	w_3	w_4	v_1	v_2	v_3	v_4

Women	1	2	3	4	5	6	7	8
w_1	m_2	m_1	u_1	u_2	u_3	u_4	m_3	m_4
w_2	m_3	m_4	m_1	m_2	u_1	u_2	u_3	u_4
w_3	m_2	m_3	u_1	u_2	u_3	u_4	m_1	m_4
w_4	m_4	m_2	u_1	u_2	u_3	u_4	m_1	m_3
v_1	m_1	m_2	m_3	m_4	u_1	u_2	u_3	u_4
v_2	m_1	m_2	m_3	m_4	u_1	u_2	u_3	u_4
v_3	m_1	m_2	m_3	m_4	u_1	u_2	u_3	u_4
v_4	m_1	m_2	m_3	m_4	u_1	u_2	u_3	u_4

Gale-Shapley algorithm returns the following matching

$$S^* = \{(m_1, v_1), (m_2, w_1), (m_3, v_2), (m_4, w_2), (u_1, w_3), (u_2, w_4), (u_3, v_3), (u_4, v_4)\}$$

Once we transform the solution back we get the matching $S = \{(m_2, w_1), (m_4, w_2)\}$ with m_1, m_3, w_3, w_4 being single.

PROOF. Proceed by contradiction and suppose that the solution we get to the original problem is not correct, in particular that it contains some instability. Then there are two possibilities:

(1) there are pairings (m, w) and (m', w') such that
   - m would rather be with w' than with w
   - w would rather be with m than with m'

(2) there is a pairing (m, w) such that WLOG m would rather be alone than with w

Let us examine the first case. The pairings (m, w) and (m', w') would also be present in the solution to the standard SMP problem we constructed (as our solution retrieval process would do nothing to those pairs). However, we know that the Gale-Shapley algorithm produces a stable solution, so that is a contradiction. ↯

Now consider the second case. Since m would rather be alone than with w, then, in the standard SMP problem we constructed, w must come after all "dummy" women in m's preference list. Therefore, m prefers any "dummy" woman to w. Also by construction, any "dummy" woman prefers m to all dummy men.

Finally, since (m, w) is pair in our solution and everyone must be matched, there must be a pair (u, v) of a dummy man and a dummy woman somewhere. The pairs (m, w) and (u, v) form an instability as noted in the previous paragraph. Once again, all solutions of the Gale-Shapley algorithm are stable, so once again we get a contraction. ↯ □

ASIDE: PROOF STRATEGIES

Anytime we want to prove that a reduction is correct, we rely on the fact that the Gale-Shapley algorithm returns a correct, stable result. We proceed by contradiction and assume there is an instability. There are usually two cases to consider:

- ⋆ The case of instability among normal $(m, w), (m', w')$ pairs. This is the easy case and here we can just cite the fact Gale-Shapley algorithm returns a stable result.
- ⋆ The case of an instability among the pairs that are constructed in a "special" way for this specific problem. In this case more work must be done; however, we still derive the contradiction from the fact that Gale-Shapley algorithm returns a stable result!

NOTES.

- Preference lists for dummy men and women are ordered in a the specific way so that the above proof works out.
- This was an assignment problem in 2018S2 but is otherwise a quite standard problem.
- The variant of SMP with unequal number of men and women (i.e. $|M| < |W|$) is handled in a rather similar way.

1.8 Example 2: A Bad Reduction

The Problem. Consider a set of m students that need to be matched into roommates so that they could move into $\frac{m}{2}$ dorms. Each of the students ranks the other $m-1$ students in a preference list.

Reduction to the Standard SMP. A natural reduction to the standard SMP is of course to split the list of m students into two lists of $\frac{m}{2}$ students each. Call one list the "men" and the other list the "women". For each man, remove any other men from the rankings list and do the same for women.

A Good Example.

Students	1	2	3
s_1	s_2	s_3	s_4
s_2	s_4	s_1	s_3
s_3	s_4	s_1	s_2
s_4	s_1	s_3	s_2

The reduced instance is:

Men	1	2
s_1	s_3	s_4
s_2	s_4	s_3

Women	1	2
s_3	s_1	s_2
s_4	s_1	s_2

The solution is $S = \{(s_1, s_3), (s_2, s_4)\}$.

A Bad Example. The following set of preference lists is an example of a case where one student s_4 is "universally disliked". Beyond that, we arrange the rest of preferences in a "diagonal" manner, as illustrated via colour coding.

Students	1	2	3
s_1	s_2	s_3	s_4
s_2	s_3	s_1	s_4
s_3	s_1	s_2	s_4
s_4	any	any	any

It turns out that *every* possible solution to this problem and *every* possible matching set is unstable:

- $S_1 = \{(s_1, s_2), (s_3, s_4)\}$: s_2 prefers $s_3 > s_1$ and s_3 prefers $s_2 > s_4$.
- $S_2 = \{(s_1, s_3), (s_2, s_4)\}$: s_1 prefers $s_2 > s_3$ and s_2 prefers $s_1 > s_4$.
- $S_3 = \{(s_1, s_4), (s_2, s_3)\}$: s_3 prefers $s_1 > s_2$ and s_1 prefers $s_3 > s_4$.

So we didn't even need to consider a reduction to show this. Therefore, ***any*** reduction will give a solution to this set that is unstable.

This was a tutorial problem in 2018S2.

2 Run Time Analysis

2.1 Review of Big-$\mathcal{O}, \Omega, \Theta$

Recall the following definitions:

- $g(n) \in \mathcal{O}(f(n))$ if $\exists c \in \mathbb{R}^+,\ \exists n_0 \in \mathbb{N},\ \forall n \in \mathbb{N},\ n \geq n_0 \Rightarrow g(n) \leq c \cdot f(n)$,
- $g(n) \in \Omega(f(n))$ if $\exists d \in \mathbb{R}^+,\ \exists n_0 \in \mathbb{N},\ \forall n \in \mathbb{N},\ n \geq n_0 \Rightarrow g(n) \geq d \cdot f(n)$,
- $g(n) \in \Theta(f(n))$ if $\exists c, d \in \mathbb{R}^+,\ \exists n_0 \in \mathbb{N},\ \forall n \in \mathbb{N},\ n \geq n_0 \Rightarrow d \cdot f(n) \leq g(n) \leq c \cdot f(n)$

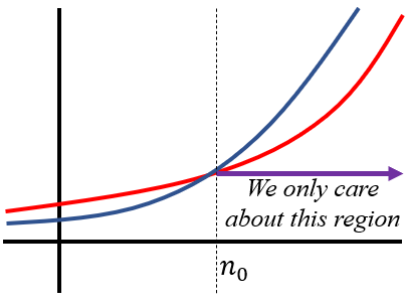
Equivalently:

- $g(n) \in \Omega(f(n))$ iff $f(n) \in \mathcal{O}(g(n))$
- $g(n) \in \Theta(f(n))$ iff $g(n) \in \mathcal{O}(f(n))$ and $g(n) \in \Omega(f(n))$
  iff $g(n) \in \mathcal{O}(f(n))$ and $f(n) \in \mathcal{O}(g(n))$

We have the following list of common run times, arranged from fastest to slowest:

Constant	$\mathcal{O}(1)$	
Log	$\mathcal{O}(\log n)$	
PolyLog	$\mathcal{O}((\log n)^k)$	$1 < k$
SubLinear	$\mathcal{O}(n^c)$	$0 < c < 1$
Linear	$\mathcal{O}(n)$	
LogLinear	$\mathcal{O}(n \log n)$	
SubQuadratic	$\mathcal{O}(n^d)$	$1 < d < 2$
Quadratic	$\mathcal{O}(n^2)$	
LogQuadratic	$\mathcal{O}(n^2 \log n)$	
Cubic	$\mathcal{O}(n^3)$	
Polynomial	$\mathcal{O}(n^a)$	
	$\mathcal{O}(n^b)$	$3 < a < b$
Exponential	$\mathcal{O}(\alpha^n)$	
	$\mathcal{O}(\beta^n)$	$1 < \alpha < \beta$
Factorial	$\mathcal{O}(n!)$	
Power	$\mathcal{O}(n^n)$	

Recall that for one function to be in $\mathcal{O}, \Omega$ or Θ of an another function, the respective inequalities need to hold only for all n that are greater than or equal to the fixed n_0, as is shown in the image to the right. We don't actually care if the desired inequality happens to be false when $n < n_0$.



2.2 Basic Examples

Which of the following are true?

1. $n^2 \in \Omega(n^3)$
2. $n^3 \log n \in \Omega(n^3)$
3. $n2^n \in \mathcal{O}(2^n)$
4. $7n^2 + 8n \in \Theta(n^2 + 5)$
5. $n + 100n^{0.1} \in \Theta(n + \log n)$
6. $1000n^{15} \in \mathcal{O}(2^n/100n^{15})$
7. $\log n^2 \in \mathcal{O}((\log n)^2)$
8. $8^{2\log_2 n} \in \mathcal{O}(3n^7 + 7n)$
9. $3^{\log_2 n} \in \Omega(3n)$
10. $2^{\sqrt{\log_2 n}} \in \mathcal{O}(\sqrt{n})$
11. $n^{18/17} \in \mathcal{O}(1.01^{1.01^n})$
12. If $k < n$, then $n \log k \in \mathcal{O}(k \log n)$.

Answers.

1. **False,** as $n^2 \not\geq n^3$. In fact this one is the other way around: $n^2 \leq n^3$, so $n^2 \in \mathcal{O}(n^3)$.
2. **True,** as $\log n \geq 1, \forall n \geq 3$ then $n^3 \log n \geq n^3, \forall n \geq 3$.
3. **False,** as $n \geq 1, \forall n \in \mathbb{N}$ so $n2^n \not\leq 2^n$. This one is also the other way around: $n2^n \in \Omega(2^n)$.
4. **True,** we just look at the dominant term which is n^2 in both expressions and drop all lower order terms.
5. **True,** once again the dominant term in both expression is n, as both, $\log n$ and $n^{0.1}$ are smaller than n.
6. **True,** this one could be viewed as $(1000n^{15})(100n^{15}) = 10^5 n^{30} \leq c \cdot 2^n$. Mind you, it is a bit of a silly example as n would have to be very large for 2^n to dominate.
7. **True,** as $\log n^2 = 2 \log n$ and $\log n \in \mathcal{O}((\log n)^2)$.
8. **True,** the first step is to make the left hand side look like the right hand side. To do this, we need to make use of the formula $a^{\log_a x} = x$: exponents cancel with the logarithm of the same base, e.g. $2^{\log_2 16} = 2^4 = 16$.

   To apply the formula, we need to express 8 as an exponent with base 2:

   $$8^{2\log_2 n} = (2^3)^{2\log_2 n} = 2^{6\log_2 n} = 2^{\log_2 n^6} = n^6 \in \mathcal{O}(3n^7 + 7n)$$

9. **True,** in this case we once again make the left hand side look like the right hand side. However, this time is it difficult to express 3 as an exponent with base 2. One approach is to change the base of the logarithm to 3 using the change of base formulas, $\log_a b = \frac{\log_c b}{\log_c a}$ and $\log_a b = \frac{1}{\log_b a}$:

$$3^{\log_2 n} = 3^{\log_3 n / \log_3 2} = 3^{\frac{1}{\log_3 2} \log_3 n} = 3^{\log_2 3 \cdot \log_3 n} = 3^{(\log_3 n)^{\log_2 3}} = n^{\log_2 3}$$

Now, as $3 > 2$, then $\log_2 3 > 1$ and $n^{\log_2 3} > n$ so $3^{\log_2 n} \in \Omega(3n)$. N.B. What we essentially done in this problem is we derived the identity $a^{\log_c b} = b^{\log_c a}$ for any $a > 0, b > 0$. This identity has quite a number of applications throughout run time analysis.

10. **True,** this example is done on pg. 65-66 of the Kleinberg textbook. Here, there is a square root in the exponent, so it is not possible to make the left hand side look like the right hand side. The best approach is to take logarithms of both sides:

$$\log_2 \left(2^{\sqrt{\log_2 n}}\right) = \sqrt{\log_2 n} = (\log_2 n)^{1/2}$$
$$\log_2(\sqrt{n}) = \log_2(n^{1/2}) = \tfrac{1}{2} \log_2 n$$

Then we observe that $(\log_2 n)^{1/2} < \log_2 n$ so $2^{\sqrt{\log_2 n}} \in \mathcal{O}(\sqrt{n})$.

11. **True,** this problem is from the 2016W2 midterm 1 exam. Here we can once again use the trick of taking the logarithm of both sides:

$$\log_{1.01}(n^{18/17}) = \tfrac{18}{17} \log_{1.01} n \approx \log n$$
$$\log_{1.01}(1.01^{1.01^n}) = 1.01^n \log_{1.01} 1.01 = 1.01^n$$

and then we just use the fact that $\log n \in \mathcal{O}(1.01^n)$.

12. **False,** as n is the dominant term, then comparing $n \log k$ and $k \log n$ essentially amounts to comparing n to $\log n$ and $n \notin \mathcal{O}(\log n)$.

Rigorously (this analysis is attributed to Prof. Steve Wolfman), we would like to show that $n \log k > k \log n$. Re-arrangement of this inequality results in $\frac{\log k}{k} > \frac{\log n}{n}$. We know that $k < n$ so we would like to see if this implies $\frac{\log k}{k} > \frac{\log n}{n}$. In other words we would like to know if $f(x) = \frac{\log x}{x}$ is a decreasing function of x. Indeed it is, as the derivative is negative once $\log x > 1$: $f'(x) = \frac{1 - \log x}{x^2}$.

REMARKS. Examples 2 and 6-11 would all be false if either the $\mathcal{O}$ or Ω were replaced with Θ as reverse inclusions do not hold. Also, if we were to consider *sets* of $\mathcal{O}$ functions, then, for example:

- $\mathcal{O}(7n^2 + 8n) = \mathcal{O}(n^2 + 5)$ as the sets of functions belonging to either side are identical,
- $\mathcal{O}(2^{\sqrt{\log_2 n}}) \neq \mathcal{O}(\sqrt{n})$ as there would be functions in $\mathcal{O}(\sqrt{n})$ but not in $\mathcal{O}(2^{\sqrt{\log_2 n}})$ [namely $f(n) = \sqrt{n}$ is such a function]

So another way to think about the $f(n) \in \Theta(g(n))$ definition is that it means that, as sets, $\mathcal{O}(f(n)) = \mathcal{O}(g(n))$.

2.3 Proof Examples

Prove or disprove the following.

Example 1. $\log_2 n \in \Theta(\log_3 n)$

To prove that $\log_2 n \in \Theta(\log_3 n)$ we must prove two things: $\log_2 n \in \mathcal{O}(\log_3 n)$ and $\log_3 n \in \mathcal{O}(\log_2 n)$. The change of base formulas $\log_a b = \frac{\log_c b}{\log_c a}$ and $\log_a b = \frac{1}{\log_b a}$ form the basis of both proofs.

To show that $\log_2 n \in \mathcal{O}(\log_3 n)$ let $c = \log_2 3$ and $n_0 = 1$. Then for any $n \in \mathbb{N}, n \geq n_0$:

$$\log_2 n = \frac{\log_3 n}{\log_3 2} = \frac{1}{\log_3 2} \log_3 n = \log_2 3 \cdot \log_3 n \leq c \cdot \log_3 n$$

as required. Likewise, to show that $\log_3 n \in \mathcal{O}(\log_2 n)$ let $c = \log_3 2$ and $n_0 = 1$. Then for any $n \in \mathbb{N}, n \geq n_0$:

$$\log_3 n = \frac{\log_2 n}{\log_2 3} = \frac{1}{\log_2 3} \log_2 n = \log_3 2 \cdot \log_2 n \leq c \cdot \log_2 n$$

as required. □

Example 2. $2^n \in \Theta(3^n)$

Once again, we would need to show that $2^n \in \mathcal{O}(3^n)$ and $3^n \in \mathcal{O}(2^n)$. The former is true from just properties of exponential functions, but could be shown rigorously by induction. The latter is false and this could be shown either directly or by contradiction.

Proof of $2^n \in \mathcal{O}(3^n)$ by induction. Let $n = 1$ and then the base case holds: $2^1 = 2 < 3 = 3^1$. Now fix some $n \geq 1$ and suppose that the result holds for this n, in particular, suppose that $2^n \leq 3^n$. We will show that the result holds for $n + 1$, in particular, we will show that $2^{n+1} \leq 3^{n+1}$. Indeed:

$$\begin{aligned}
2^{n+1} &= 2 \cdot 2^n \\
&\leq 2 \cdot 3^n && \text{[by the inductive hypothesis]} \\
&< 3 \cdot 3^n && \text{[as } 2 < 3\text{]} \\
&= 3^{n+1}
\end{aligned}$$

Proof of $3^n \notin \mathcal{O}(2^n)$ by contradiction. Suppose that it was the case that $3^n \in \mathcal{O}(2^n)$. Then $\exists c \in \mathbb{R}^+$ and $\exists n_0 \in \mathbb{N}$ such that $3^n \leq c \cdot 2^n$ for all $n \in \mathbb{N}, n \geq n_0$. We take logarithms of both sides of the inequality:

$$\begin{aligned}
3^n &\leq c \cdot 2^n \\
\tfrac{3^n}{2^n} &\leq c \\
\log\left(\left(\tfrac{3}{2}\right)^n\right) &\leq \log c \\
n \log\left(\tfrac{3}{2}\right) &\leq \log c
\end{aligned}$$

The above amounts to $n \leq \frac{\log c}{\log(3/2)}$ and this is of course a contradiction: an arbitrary natural number n cannot be bounded from above by a constant. So, all in all, $2^n \notin \Theta(3^n)$ as $3^n \notin \mathcal{O}(2^n)$. □

Here is a direct proof of the latter statement. Recall that the definition of $f(n) \notin \mathcal{O}(g(n))$ is $\forall c \in \mathbb{R}^+, \forall n_0 \in \mathbb{N}, \exists n \in \mathbb{N} : (n \geq n_0) \wedge (f(n) > cg(n))$.

So if given a $c \in \mathbb{R}^+$ and an $n_0 \in \mathbb{N}$, we would like to find an $n \in \mathbb{N}$ such that both of the above inequalities hold.

So if we are given some $c \in \mathbb{R}^+$ and $n_0 \in \mathbb{N}$, set $n = \max\{n_0 + 1, \lceil \frac{\log c}{\log(3/2)} \rceil\}$. Then clearly $n \geq n_0 + 1 > n_0$ and

$$\begin{aligned} n \geq \lceil \frac{\log c}{\log\left(\frac{3}{2}\right)} \rceil &\geq \frac{\log c}{\log\left(\frac{3}{2}\right)} \\ n \log\left(\tfrac{3}{2}\right) &\geq \log c \\ \log\left(\tfrac{3}{2}\right)^n &\geq \log c \\ \left(\tfrac{3}{2}\right)^n &\geq c \\ 3^n &\geq c \cdot 2^n \end{aligned}$$

So we have shown that both of the inequalities of the definition $f(n) \notin \mathcal{O}(g(n))$ hold in this case as required. □

Example 3. If $f(n) \in \Omega(h(n))$ and $g(n) \in \Omega(h(n))$ show that $f(n) + g(n) \in \Omega(h(n))$.

Since $f(n) \in \Omega(h(n))$ then $\exists d_1 \in \mathbb{R}^+, \exists n_1 \in \mathbb{N}$ such that for any $n \geq n_1 : f(n) \geq d_1 \cdot h(n)$. Also, since $g(n) \in \Omega(h(n))$ then $\exists d_2 \in \mathbb{R}^+, \exists n_2 \in \mathbb{N}$ such that for any $n \geq n_2 : g(n) \geq d_2 \cdot h(n)$.

If we set $n_0 = \max\{n_1, n_2\}$, then both of the above inequalities would hold for all $n \geq n_0$ and it is possible to just add the two inequalities:

$$f(n) + g(n) \geq d_1 \cdot h(n) + d_2 \cdot h(n) = (d_1 + d_2) \cdot h(n)$$

Taking $d = d_1 + d_2$ shows that $f(n) + g(n) \in \Omega(h(n))$ as required. □

Remarks. The above proof was really about unwrapping the definitions of the given facts $f(n) \in \Omega(h(n))$ and $g(n) \in \Omega(h(n))$ and then manipulating them to arrive at the definition of $f(n) + g(n) \in \Omega(h(n))$. We set $n_0 = \max\{n_1, n_2\}$ as we would like to go far enough on the n axis for both of the definition of the given facts to hold at the same time.

This problem appeared on the 2016W2 midterm 1 exam, but is otherwise a very standard problem.

2.4 Little o and ω

The definitions of little o and ω are as follows: we take the definitions of $\mathcal{O}$ and Ω and change the quantifier at front of the c from $\exists$ to $\forall$. So instead of requiring that there is just at least *one* c that works, we now require that the result holds for *any* c.

- $g(n) \in o\left(f(n)\right)$ if $\forall c \in \mathbb{R}^+,\ \exists n_0 \in \mathbb{N},\ \forall n \in \mathbb{N},\ n \geq n_0 \Rightarrow g(n) \leq c \cdot f(n)$,
- $g(n) \in \omega\left(f(n)\right)$ if $\forall d \in \mathbb{R}^+,\ \exists n_0 \in \mathbb{N},\ \forall n \in \mathbb{N},\ n \geq n_0 \Rightarrow g(n) \geq d \cdot f(n)$

As before, $g(n) \in \omega\left(f(n)\right)$ iff $f(n) \in o\left(g(n)\right)$.

So the definitions of little o and ω are *stronger* or are more restrictive. For example consider $f(n) = 2n^2$ and $g(n) = n^2$. Then clearly $g(n) \in \mathcal{O}(f(n))$, take $c = 1$ and $n_0 = 1$ then $g(n) \leq c \cdot f(n)$ for any $n \geq n_0$ since $n^2 \leq 2n^2$ for any $n \geq 1$.

However, if we take $c = \frac{1}{4}$ then we have a problem: $g(n) \not\leq c \cdot f(n)$ as $n^2 \not\leq \frac{1}{4} \cdot 2n^2 = \frac{1}{2}n^2$ for any choice of n_0. So while $n^2 \in \mathcal{O}(2n^2)$ it is not the case that $n^2 \in o(2n^2)$.

So any function $f(n)$ that is in $o(g(n))$ is also in $\mathcal{O}(g(n))$, but not the other way around. Likewise, any function $f(n)$ that is in $\omega(g(n))$ is also in $\Omega(g(n))$.

Generally, for some function $f(n)$ to be in little o of an another function $g(n)$, the run time of $f(n)$ must be an order of magnitude faster than the run time of $g(n)$. In other words, $f(n)$ and $g(n)$ cannot be in the same big-$\mathcal{O}$ family.

$f(n)$	$g(n)$	$f(n) \in \mathcal{O}\left(g(n)\right)$?	$f(n) \in o\left(g(n)\right)$?
n	n	Yes	No
n	n^2	Yes	Yes
n^2	n	No	No
$\log n$	n	Yes	Yes
$\log n$	$2 \log n$	Yes	No

Another way to think about it, the $\mathcal{O}$ notation is sort of the equivalent of the $\leq$ sign. For two functions to be in $\mathcal{O}$ of each other, they could be in the same family or one could be in a faster family than the other. The little o notation on the other hand is sort of the equivalent of the $<$ sign. For two functions to be in little o of each other, they cannot be in the same family, one must be *strictly* faster than the other. Likewise, the definitions of Ω and ω are similarly related to the notions of the $\geq$ and $>$ signs respectively.

Yet another way to think about little o and ω are the following relationships:

- $o(f(n)) \approx \mathcal{O}(f(n)) - \Theta(f(n))$ [i.e. to get $<$ take $\leq$ and remove the $=$]
- $\omega(f(n)) \approx \Omega(f(n)) - \Theta(f(n))$ [i.e. to get $>$ take $\geq$ and remove the $=$]

Those relationships are generally great analogies but are not valid all the time. It is possible to come up with a few rather bizarre examples that do **not** follow those analogies, see the following pages.

Something else to note is that a definition of little θ doesn't exists since it doesn't make sense. Recall that Θ means $\mathcal{O}$ and Ω, so we would expect for little θ to mean little o and ω. If we were to use our analogy, it makes sense the intersection of $\leq$ ($\mathcal{O}$) and $\geq$ (Ω) amounts to $=$ (Θ). However, the intersection of $<$ (o) and $>$ (ω) would be just an empty set. In other words, sets $o(f(n))$ and $\omega(f(n))$ are disjoint.

More formally, suppose we were to define little θ as
$f(n) \in \theta(g(n))$ if $\forall c, d \in \mathbb{R}^+, \exists n_0 \in \mathbb{N}, \forall n \in \mathbb{N}, n \geq n_0 \Rightarrow d \cdot f(n) \leq g(n) \leq c \cdot f(n)$

Now take $f(n) = g(n) = n$ and $c = \frac{1}{2}, d = 2$. Then $2n \not\leq n \not\leq \frac{1}{2}n$ for any choice of n_0. So the definition doesn't even make sense if $f(n) = g(n)$, let alone if those were different functions.

Example 1: Proof of $n \in o(n^2)$. Since the quantifier on c is universal, we start the proof by assuming that c is an arbitrary positive real number and we then look for an $n_0 \in \mathbb{N}$ that works. In particular, we would like to find n_0 such that for any $n \geq n_0, f(n) \leq c \cdot g(n)$ or $n \leq cn^2$. This reduces to $1 \leq cn$ or $\frac{1}{n} \leq c$. So as long as $\frac{1}{n}$ is smaller than c, we are fine.

How do we ensure that $\frac{1}{n}$ is smaller than c? Well we know that $n \geq n_0$ or $\frac{1}{n_0} \geq \frac{1}{n}$. So if we set $c = \frac{1}{n_0}$ or $n_0 = \frac{1}{c}$ then $c = \frac{1}{n_0} \geq \frac{1}{n}$ and we satisfy that $c \geq \frac{1}{n}$. That is it, we are done with the proof, since we found an n_0 that works.

Let's see how this works in practice. Suppose we would like to see if the result holds when $c = 10$. We would like to know when is it that $n \leq 10n^2$. Set $n_0 = \frac{1}{c} = \frac{1}{10} = 0.1$ which rounds up to 1 for it to be a natural number and then certainly $n \leq 10n^2$ for any $n \geq 1$.

Next, what if $c = 0.001$? When is it the case that $n \leq 0.001n^2$? Not a problem, we just set $n_0 = \frac{1}{c} = 1000$ and then $n \leq 0.001n^2$ whenever $n \geq 1000$. So having a general formula for n_0 in terms of any c makes the desired result hold for any c.

Example 2: $5n^2 + 3n + 1 \in o(n^3)$.

We would like to find n_0 such that for any $c \in \mathbb{R}^+$ and $n \geq n_0$, $f(n) \leq c \cdot g(n)$ or $5n^2 + 3n + 1 \leq cn^3$. We know that $5n^2 + 3n + 1 \leq 9n^2$ so we can rewrite the previous statement as $9n^2 \leq cn^3$, which reduces to $9 \leq cn$. So it seems like $n_0 = \frac{9}{c}$ is a good choice.

Proof. Let c be arbitrary and set $n_0 = \frac{9}{c}$. Then for any $n \geq n_0$ we have $n \geq \frac{9}{c}$ or $c \geq \frac{9}{n}$ and so:

$$\begin{aligned} f(n) &= 5n^2 + 3n + 1 \\ &\leq 9n^2 \\ &= \tfrac{9}{n} \cdot n^3 \\ &\leq c \cdot n^3 \\ &= c \cdot g(n) \end{aligned}$$

□

Example 3: A function $f(n)$ that is in $\mathcal{O}(g(n))$, not in $\Theta(g(n))$ and not in $o(g(n))$.

Take $f(n) = n \bmod 2$ and $g(n) = 1$. So the function $f(n)$ is 0 whenever n is even and is 1 whenever n is odd.

Proof of $f(n) \in \mathcal{O}(g(n))$. Let $c = 1$ and $n_0 = 1$. For any $n \in \mathbb{N}$, the value of $f(n)$ is at most 1, so of course $f(n) \leq c \cdot g(n)$. □

Proof of $f(n) \notin o(g(n))$. Proceed by contradiction. Suppose there is an $n_0 \in \mathbb{N}$ such that for any $c \in \mathbb{R}^+$ and for any $n \in \mathbb{N}, n \geq n_0, f(n) \leq c \cdot g(n)$. Take $c = \frac{1}{2}$ and if n_0 is odd, set $n = n_0$, otherwise set $n = n_0 + 1$. Then $n \geq n_0$ and $f(n) = 1 \not\leq \frac{1}{2} \cdot 1 = c \cdot g(n)$, so we arrive at a contradiction. ↯ □

In other words, with the choice of $c = \frac{1}{2}$ and any choice of n_0, we can pick an n that is larger than n_0 and is odd, so that the required inequality $f(n) \leq c \cdot g(n)$ doesn't hold.

Proof of $f(n) \notin \Theta(g(n))$. We already know that $f(n) \in \mathcal{O}(g(n))$, so we must show that $f(n) \notin \Omega(g(n))$ or that $g(n) \notin \mathcal{O}(f(n))$. Proceed by contradiction. Suppose that there is a $c \in \mathbb{R}^+$ and an $n_0 \in \mathbb{N}$ such that for any $n \in \mathbb{N}, n \geq n_0, g(n) \leq c \cdot f(n)$. In a way that is similar to the previous case, if n_0 is even, set $n = n_0$, otherwise set $n = n_0 + 1$. Then $n \geq n_0$ and $g(n) = 1 \not\leq c \cdot 0 = c \cdot f(n)$ for any choice of c. So once again we get a contradiction. ↯ □

The difference between the last proofs is that in the latter case we had to arrive at a contradiction for any choice of c and in the former we were free to choose a c that was convenient, due to the differences in the quantifiers on c in the definitions of little o and Θ.

Essentially the relationship $o(f(n)) \approx \mathcal{O}(f(n)) - \Theta(f(n))$ fails when there are oscillations in $f(n)$ and/or $g(n)$. This example itself is from `https://cs.stackexchange.com/questions/57469/big-o-and-not-little-o-implies-theta`.

2.5 Relationship with Limits

Working with the definitions of $\mathcal{O}, \Omega, \Theta, o$ and ω could be rather tedious. If you happen to remember how to work with limits from a calculus course, this could make things a bit simpler when working with $\mathcal{O}, \Omega, \Theta, o$ and ω.

The following table summarizes the relationships between the limit at infinity of the ratio $\frac{f(n)}{g(n)}$ and $\mathcal{O}, \Omega, \Theta, o$ and ω.

Value of Limit	Run time
$\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$	$f(n) \in o(g(n))$
$\lim_{n\to\infty} \frac{f(n)}{g(n)} \neq \infty$	$f(n) \in \mathcal{O}(g(n))$
$\lim_{n\to\infty} \frac{f(n)}{g(n)} \neq 0, \infty$	$f(n) \in \Theta(g(n))$
$\lim_{n\to\infty} \frac{f(n)}{g(n)} \neq 0$	$f(n) \in \Omega(g(n))$
$\lim_{n\to\infty} \frac{f(n)}{g(n)} = \infty$	$f(n) \in \omega(g(n))$

Table 2.1

As an aside, if you seen the rigorous $(\epsilon - \delta)$ definition of a limit in calculus, that definition is essentially the same as the definition of little o.

The $(\epsilon - \delta)$ definition of $L = \lim_{n\to\infty} a(n)$ where $a(n)$ is some function with domain $\mathbb{N}$ is: $\forall \epsilon > 0, \exists n_0 \in \mathbb{N}, \forall n > n_0, |a(n) - L| < \epsilon$.

In the little o case, take $L = 0$, $\epsilon = c$ and $a(n) = \frac{f(n)}{g(n)}$ and we get the exact same statement.

Example 1. Use the limit definition to show that $f(n) = 5n^2 + 3n + 1$ is in $o(n^3)$.

$$\begin{aligned}\lim_{n\to\infty} \frac{5n^2 + 3n + 1}{n^3} &= \lim_{n\to\infty} \left(\frac{5}{n} + \frac{3}{n^2} + \frac{1}{n^3}\right) \\ &= \lim_{n\to\infty} \left(\frac{5}{n}\right) + \lim_{n\to\infty} \left(\frac{3}{n^2}\right) + \lim_{n\to\infty} \left(\frac{1}{n^3}\right) \\ &= 0 + 0 + 0 = 0\end{aligned}$$

Example 2. Use the limit definition to show that $f(n) = 5n^2 + 3n + 1$ is in $\Theta(n^2)$.

$$\begin{aligned}\lim_{n\to\infty} \frac{5n^2 + 3n + 1}{n^2} &= \lim_{n\to\infty} \left(5 + \frac{3}{n} + \frac{1}{n^2}\right) \\ &= \lim_{n\to\infty} (5) + \lim_{n\to\infty} \left(\frac{3}{n}\right) + \lim_{n\to\infty} \left(\frac{1}{n^2}\right) \\ &= 5 + 0 + 0 = 5 \neq 0 \text{ or } \infty\end{aligned}$$

Example 3. Use the limit definition to show that $f(n) = 5n^2 + 3n + 1$ is in $\omega(n)$.

$$\begin{aligned}\lim_{n\to\infty} \frac{5n^2 + 3n + 1}{n} &= \lim_{n\to\infty} \left(5n + 3 + \frac{1}{n}\right) \\ &= \lim_{n\to\infty} (5n) + \lim_{n\to\infty} (3) + \lim_{n\to\infty} \left(\frac{1}{n}\right) \\ &\to \infty + 3 + 0 \to \infty\end{aligned}$$

Example 4. Use the limit definition to show that $f(n) = 10n^2 + 2n$ is in $\mathcal{O}(5n^2 - n)$.

$$\begin{aligned}\lim_{n\to\infty} \frac{10n^2 + 2n}{5n^2 - n} &= \lim_{n\to\infty} \left(\frac{\frac{1}{n^2}\left(10 + \frac{2}{n}\right)}{\frac{1}{n^2}\left(5 - \frac{1}{n}\right)}\right) \\ &= \frac{\lim_{n\to\infty}\left(10 + \frac{2}{n}\right)}{\lim_{n\to\infty}\left(5 - \frac{1}{n}\right)} \\ &= \frac{10 + 0}{5 - 0} = 2 \neq \infty\end{aligned}$$

Example 5. Use the limit definition to show that $f(n) = n!$ is in $o(n^n)$.

First we will analyze the quotient $f(n)/g(n) = n!/n^n$:

$$\frac{n!}{n^n} = \underbrace{\frac{n}{n}}_{1} \cdot \underbrace{\frac{n-1}{n}}_{<1} \cdots \underbrace{\frac{2}{n}}_{<1} \cdot \frac{1}{n} < \frac{1}{n}$$

Then we find the limit via a "squeezing argument":

$$0 < \frac{n!}{n^n} < \frac{1}{n}$$

$$\lim_{n\to\infty} 0 \leq \lim_{n\to\infty} \frac{n!}{n^n} \leq \lim_{n\to\infty} \frac{1}{n}$$

$$0 \leq \lim_{n\to\infty} \frac{n!}{n^n} \leq 0$$

Therefore it must be that $\lim_{n\to\infty} \frac{n!}{n^n} = 0$ as required [it is the application of the limit squeeze theorem that converts $<$ into $\leq$ above].

Most of the above examples would been very tedious, if not impossible to rigorously prove using the definitions of $\mathcal{O}, \Omega, \Theta, o$ and ω. Using the limit approach, most of the solutions are quite straight forward. In fact, example 1 of this section is the same as example 2 of the previous section.

However, the limit definitions do run into issues as well. For example, if we wish to approach example 3 of the previous section where $f(n) = n \bmod 2$ and $g(n) = 1$ using the limit technique, then $\lim_{n\to\infty} f(n)/g(n)$ doesn't exist. The limit doesn't exist since the ratio $f(n)/g(n)$ oscillates between 0 and 1 as n switches between even and odd numbers, so we don't approach any particular number as n tends to infinity. Therefore, we can certainly conclude that $f(n) \notin o(g(n))$ but cannot conclude anything about $\mathcal{O}(g(n))$.

So, if the limit exists and falls within some row of the table 2.1, then the corresponding run time statement is true. However, the converse may not hold for $\mathcal{O}, \Omega$ or Θ. In the above example the limit doesn't exists yet we know that $f(n) \in \mathcal{O}(g(n))$. The converse issue is addressed using the concepts of *limit superior* and *limit inferior* which are beyond the scope of this discussion. The converse does hold in the cases of little o and ω as the definitions of those are essentially the mathematical definitions of a limit.

Also take a look at `http://web.mit.edu/broder/Public/asymptotics-cheatsheet.pdf`

Example 6. Show that $\log\log(n)$ is in $o(\log n)$ and therefore is in $\mathcal{O}(\log n)$.

We know that $\log m \in o(m)$ and it is the little o here; therefore, we can use the converse of table 2.1 and state that $\lim_{m\to\infty} \frac{\log m}{m} = 0$. Then we can make a substitution $m = \log n$ and arrive at:

$$\lim_{n\to\infty} \frac{\log\log(n)}{\log n} = \lim_{m\to\infty} \frac{\log m}{m} = 0$$

2.6 Amortized Analysis: Aggregate Method

Some algorithms consist of two kinds of tasks:

- tasks that happen quite often and take minimal time, and
- tasks that happen not that often and take up a lot of time

A classical example is the process of adding elements to an array-based stack. Here we have the following two types of tasks:

- pushing an element into a stack that ***is not*** full: we do not resize the stack and the run time is $\mathcal{O}(1)$, and
- pushing an element into a stack that ***is*** full: we must resize the stack (we usually double the size) which requires copying all elements over and the run time is $\mathcal{O}(n)$.

Of course the latter doesn't happen very often, as if we just doubled the size of the stack, we would not need to run the resize operation for quite sometime. We could analyze the run time of this algorithm in a quick way and claim that we, at worst, resize on every push, so the run time is $\mathcal{O}(n^2)$. While this is technically true, we could come up with a more realistic and tighter estimate if we take into account how often each operation actually occurs. In the end we will report the *average* run time of all operations involved.

To carry out the analysis we apply the following formula:

$$\text{Aggregate Run Time} = \frac{\sum_{\text{all operations}} \text{No of Times An Operation Occurs} \times \text{Its Run Time}}{\text{Total Number Of Operations}}$$

Example 1. Amortized calculation for the push operation of an array-based stack.

Solution. Suppose we start with an array that is of size n. For every $n+1$ pushes:

- ✓ we have n regular pushes that take $\mathcal{O}(1)$ each, and
- ✓ we also have the one push that requires the stack to be resized and takes $\mathcal{O}(n)$.

The total number of operations is $n+1$, so

$$\text{Aggregate Run Time} = \frac{n \cdot \mathcal{O}(1) + 1 \cdot \mathcal{O}(n)}{n+1} \approx \frac{2 \cdot \mathcal{O}(n)}{n+1} \approx \mathcal{O}(1)$$

Therefore, on average, a push operation is $\mathcal{O}(1)$.

Example 2. Suppose we have a k-bit binary counter that uses a binary number to keep a count of something. We would like to determine the amortized cost of the `increment` operation.

If the least significant digit is 0, then we just need to change that one digit when we increment: `00100 -> 00101`

However, if that is not the case, then we may need to change many digits on a single increment: `01111 -> 10000`

While the latter case happens rather infrequently, in all cases we would need to change at least one digit. We also have some cases that are "in-between" those: `01011 -> 01100`

```
Increment(A) {
    i := 0                                  // Start from the least significant digit

    while (i < length[A] && A[i] = 1) { // If we come across a digit that is already 1,
        A[i] := 0                           // we set it to 0 and continue to do so
        i := i+1                            // until we come to a digit that isn't 1
    }

    if (i < length[A])    { A[i] := 1 } // We set our first digit that isn't 1 to 1
}
```

Solution. We observe that

- ✓ the least significant digit is flipped on every increment,
- ✓ the next digit is flipped on every other increment,
- ✓ the next digit is flipped on every 4th increment, etc., and
- ✓ the most significant digit is flipped only on every k-th increment.

This patterns leads to the following general formula for the total number of operations per every n increments. We approximate the sum using geometric series:

$$\text{No. of Operations} = \sum_{i=0}^{k-1} \left\lfloor \frac{n}{2^i} \right\rfloor < \sum_{i=0}^{k-1} \frac{n}{2^i} = n \sum_{i=0}^{k-1} \frac{1}{2^i} = n \cdot 2 = 2n$$

Above value is substituted into the formula for aggregate run time and the result is $\mathcal{O}(1)$:

$$\text{Aggregate Run Time} = \tfrac{2n}{n} = 2 \in \mathcal{O}(1)$$

Example 3. Suppose there is a data structure such that every i-th insertion runs in $\mathcal{O}(n)$, if i is an exact power of 2 [i.e. $i = 2^k, k \in \mathbb{Z}$], and runs in $\mathcal{O}(1)$ otherwise. This example is essentially a slightly different variation of example 1: we start with a very small stack that we double whenever i is a power of 2, which requires copying the $i = 2^k$ elements over.

Solution. Once again, we determine the total number of operations per every n insertions, which we then sum using geometric series (the first sum is when we copy, the second sum is when we don't):

$$\begin{aligned}
\text{No. of Operations} &= \sum_{k=0}^{\lfloor \log_2 n \rfloor} 2^k + \sum_{\substack{\text{all } i \text{ such that} \\ i \neq 2^k, k \in \mathbb{Z}}} 1 \\
&= \frac{1 - 2^{\lfloor \log_2 n \rfloor + 1}}{1 - 2} + \underbrace{(n - \lfloor \log_2 n \rfloor)}_{\substack{\text{no. of elements that} \\ \text{are not powers of 2}}} \cdot 1 \\
&= 2^{\lfloor \log_2 n \rfloor + 1} - 1 + n - \lfloor \log_2 n \rfloor \\
&= 2 \cdot \underbrace{2^{\lfloor \log_2 n \rfloor}}_{\leq n} + n \underbrace{- (\lfloor \log_2 n \rfloor + 1)}_{\leq 0} \\
&\leq 2n + n = 3n
\end{aligned}$$

Finally, we divide by the total number of operations and get $\frac{3n}{n} = 3 \in \mathcal{O}(1)$.

Example 4. Continuing from the previous example, this time suppose there is a data structure such that every i-th insertion runs in $\mathcal{O}(n^2)$, if i is an exact power of 2, and runs in $\mathcal{O}(1)$ otherwise. This example would illustrate a situation where we have an array or an array-based stack, and in addition to doubling the data structure whenever we run out of space, we also sort the data it contains using insertion sort.

Solution. Similar to example 3. In this case, the following pattern develops:

- ✓ operation no. 2 costs $2^2 = 4 = 4^1$,
- ✓ operation no. 4 costs $4^2 = 16 = 4^2$,
- ✓ operation no. 8 costs $8^2 = 64 = 4^3$, etc.

So, we get the following general formula for the total number of operations per every n insertions, which we once again sum using geometric series:

$$\begin{aligned}
\text{No. of Operations} &= \sum_{k=0}^{\lfloor \log_2 n \rfloor} 4^k + \sum_{\substack{\text{all } i \text{ such that} \\ i \neq 2^k, k \in \mathbb{Z}}} 1 \\
&= \frac{1 - 4^{\lfloor \log_2 n \rfloor + 1}}{1 - 4} + \underbrace{\left(n - \lfloor \log_2 n \rfloor\right)}_{\substack{\text{no. of elements that} \\ \text{are not powers of 2}}} \cdot 1 \\
&= \tfrac{1}{3} \cdot \left(4^{\lfloor \log_2 n \rfloor + 1} - 1\right) + n - \lfloor \log_2 n \rfloor \\
&= \tfrac{1}{3} \cdot 4 \cdot \underbrace{4^{\lfloor \log_2 n \rfloor}}_{\leq n^2} + n \underbrace{- \left(\lfloor \log_2 n \rfloor + \tfrac{1}{3}\right)}_{\leq 0} \\
&\leq \tfrac{4}{3} n^2 + n
\end{aligned}$$

Finally, we divide by the total number of operations and get $\frac{4}{3}n + 1 \in \mathcal{O}(n)$.

Example 5. Suppose that we use a linked list to implement a stack. This time we do not need to worry about any resizing operations. However, this time we introduce the "multi-pop" operation: we can choose to either pop one element off at a time, or to pop all elements off the stack at once. Of course a single push or pop would have a run time of $\mathcal{O}(1)$, while the multi-pop operation would have a run time of $\mathcal{O}(n)$. The goal is to find aggregate run time for the pushing and popping all elements onto and off the stack.

Solution. The analysis is similar to that in example 1. Suppose we push and pop n off and onto the stack.

For every such n elements, we have n regular pushes that take $\mathcal{O}(1)$ each. We also have the one multi-pop that takes $\mathcal{O}(n)$. The total number of operations is $n + 1$, so

$$\text{Aggregate Run Time} = \frac{n \cdot \mathcal{O}(1) + 1 \cdot \mathcal{O}(n)}{n + 1} \approx \frac{2 \cdot \mathcal{O}(n)}{n + 1} \approx \mathcal{O}(1)$$

Therefore, we can push and pop all n elements onto and off the stack in $\mathcal{O}(1)$.

2.7 Amortized Analysis: Accounting Method

In the accounting method, whenever we estimate the cost of an elementary operation, we do so that the cost would be large enough to build up *enough credit* for any of the infrequent, more complex operations to take place.

Example 1. Example 5 from the previous section (push, pop, multi-pop operations in a linked list based stack).

Solution. Suppose that we have n elements in the stack and we wish to do the multi-pop operation that costs $\mathcal{O}(n)$. How much credit should we have saved up at this point from all elementary push operations? Well, we need a total of n credits saved up for the multi-pop; therefore, each push operation should be assigned a cost of 2 out of which:

- ✓ 1 was used to pay for the push itself, and
- ✓ 1 is accumulated and is used to build up the credit for the multi-pop that costs n

As before, we get exactly $\frac{2n}{n} = 2 \in \mathcal{O}(1)$.

Example 2. Example 2 from the previous section (k-bit binary counter).

Solution. All operations of setting a bit would be $\mathcal{O}(1)$ and would typically cost 1 elementary operation. However, once again, we increase the cost of setting a bit to 2. Out of the credit build up by each bit:

- ✓ 1 is used to pay to set the bit to 1, and
- ✓ 1 is accumulated and is used to reset the bit back to 0 when appropriate (once all bits are reset back to 0, the counter has ran through a full cycle)

As before, we get exactly $\frac{2n}{n} = 2 \in \mathcal{O}(1)$.

Example 3. Example 3 from the previous section (whenever i is a power of 2, the operation runs in $\mathcal{O}(n)$, otherwise all operations are $\mathcal{O}(1)$).

Solution. Once we come across an i-th operation, where i is a power of 2, how much credit must we have saved up to carry out that operation? We would need i credit; however, we have only $\frac{i}{2}$ elementary operations to build up that credit. We can only use the last $\frac{i}{2}$ elements of the array to pay for the operation, because the credit of the first $\frac{i}{2}$ was already used up to pay for the previous such operation. Therefore, we increase the cost of each elementary operation to 3, out of which:

- ✓ 1 is used to pay for the initial elementary operation itself
- ✓ 2 is accumulated over the $\frac{i}{2}$ elements for a total of $2 \cdot \frac{i}{2} = i$ and is used to pay for the i-th operation whenever i is a power of 2, to copy the i elements over

As before, we get exactly $\frac{3n}{n} = 3 \in \mathcal{O}(1)$.

3 Graphs

3.1 Graph Basics

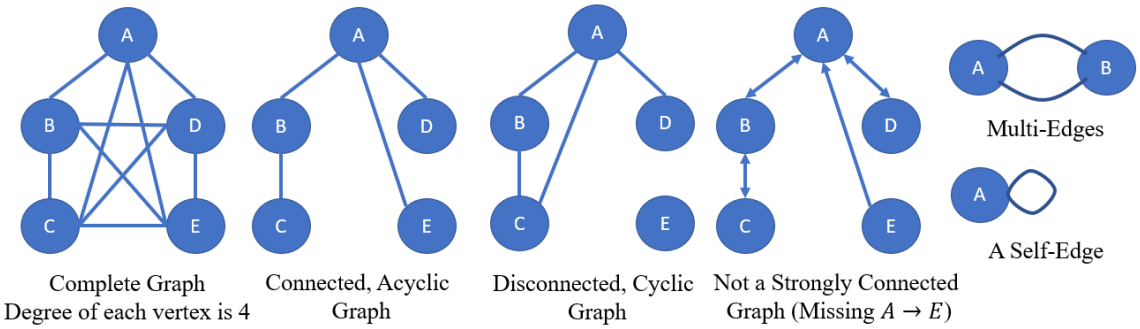
Formal Definitions.

- **Graph:** is a pair of sets (V, E) where V is the collection of nodes or vertices and E is the collection of edges.
- **Edge:** is an *unordered* pair of vertices $(u, v) \subset V \times V$ where vertices u, v are called endpoints of the edge. An edge is said to be *incident* to vertices u and v in this case.
- **Directed Edge:** is an *ordered* pair of vertices $(u, v) \subset V \times V$ where vertex u is called the head or the source and vertex v is called the tail or the sink. A directed edge can only be traversed from head to tail.
- **Self-Edge:** an edge with identical endpoints [i.e. the edge (u, u)].
- **Multi-Edges:** two or more edges that are incident on the same vertices.
- **Degree of a Vertex:** the number of edges that are incident on the vertex.
- **Path:** is a sequence of vertices $\{v_1, v_2, \cdots, v_k\}$ such that there exists an edge between consecutive vertices.
- **Simple Path:** is a path doesn't pass through any vertex more than once.
  [i.e. is a path $\{v_1, v_2, \cdots, v_k\}$ where $i \neq j \Rightarrow v_i \neq v_j$]
- **Cycle:** is a path with a common beginning and an end.
  [i.e. is a path $\{v_1, v_2, \cdots, v_k\}$ where $v_1 = v_k$ and $k - 1$ is the cycle's length]
- **(Minimum) Distance:** in an unweighted graph is the (minimum) number of edges along a path between two vertices; in a weighted graph is the (minimum) sum of edge weights along a path between two vertices. If there isn't a path between two vertices, typical convention is to set distance to ∞.
- **Connected Component of Vertex v:** is the set of all vertices to which there exists a path from v [i.e the set of all vertices reachable from v].

Types of Graphs.

- **Simple Graph:** has no self-edges or multi-edges.
- **Directed Graph:** is a graph in which each edge is a directed edge.
- **Weighted Graph:** is a graph in which each edge is assigned a weight.
- **Cyclic Graph:** is a graph that contains at least one cycle.
- **Connected Graph:** every pair of distinct vertices has a *path* between them.
- **Complete Graph:** every pair of distinct vertices has an *edge* between them.
- **Strongly Connected Graph:** is a directed graph such that for any two vertices u and v, there is a path both, from u to v and from v to u. Such vertices are called *mutually reachable* and a strongly connected graph is the one in which every pair of vertices is mutually reachable.
- **Tree:** is an undirected connected acyclic graph.

Some Examples.



Complete Graph
Degree of each vertex is 4

Connected, Acyclic Graph

Disconnected, Cyclic Graph

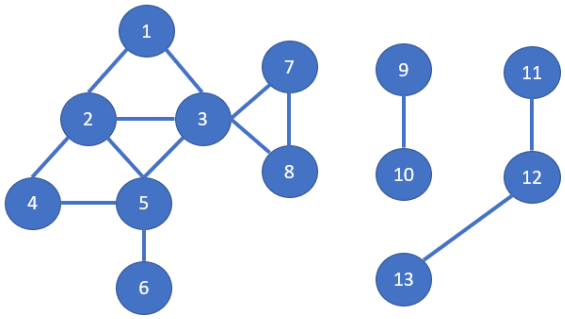
Not a Strongly Connected Graph (Missing $A \to E$)

A connected graph would be made up of just a single connected component.

This is the same graph as in Figure 3.2, pg. 79 of Kleinberg

This graph is undirected, disconnected and is cyclic.

Number of connected components: 3
Degree of vertex no. 3 is 5.



Let n be the number of vertices and m be the number of edges. For simple graphs:

- The degree of each vertex is: $0 \leq \deg(v) \leq n-1$.
- Sum of degrees in a graph is: $\sum_{v \in V} \deg(v) = 2m$.
- In a connected graph it must be $m \geq n-1$.
- In a complete graph $m = \frac{1}{2}n(n-1)$.

Therefore for connected, simple graphs we have $\mathcal{O}(n) \subset \mathcal{O}(m) \subset \mathcal{O}(n^2)$.

- A graph is said to be *sparse* if $m \in \mathcal{O}(n)$.
- A graph is said to be *dense* if $m \in \mathcal{O}(n^2)$.

Proof Of: In a complete graph $m = \frac{1}{2}n(n-1)$.

The first vertex needs to be connected to $n-1$ other vertices, the next one needs to be connected to $n-2$ vertices [as it is already connected to the first one], the next one needs to be connected to $n-3$ vertices, etc. So the total number of edges is just the sum of integers from 1 to $n-1$:

$$(n-1) + (n-2) + \cdots + 1 = \sum_{i=1}^{n-1} i = \frac{1}{2}n(n-1)$$

□

Proof Of: $\sum_{v \in V} \deg(v) = 2m$.

Each edge contributes twice to degrees of vertices: once to degree the of it starting point and once to the degree of its end point. There are m edges in total; therefore, the sum is $2m$. □

The other approach to the proof of $m = \frac{1}{2}n(n-1)$ for a complete graph is via combinatorics: there are n vertices and 2 of them are involved in any given edge. All possible combinations (i.e. all possible ways to choose 2 vertices from a set of n vertices) of such choices are:

$$\binom{n}{2} = \frac{n!}{2!(n-2)!} = \frac{n(n-1)(n-2)\cdots 3\cdot 2\cdot 1}{2(n-2)\cdots 3\cdot 2\cdot 1} = \frac{1}{2}n(n-1)$$

The following five propositions are meant to not only state certain useful and important properties about graphs, but perhaps are also just some examples on how to approach proofs that involve graphs.

Proposition 1. For any pair of vertices in a tree, there is a *unique* path that connects them.

Proof. Proceed by contradiction. Take two arbitrary vertices u and v of a tree and suppose that there isn't a unique path that connects them. This could mean one of two things: either there isn't a path at all, or there is more than one path.

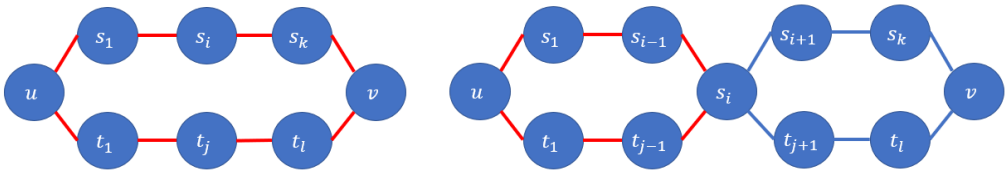
In the former case it would mean that two vertices are not connected, which means that the tree is not connected and this contradicts the definition of a tree. ↯

In the latter case, suppose there are two paths between u and v, the label the paths as $\mathcal{P}_1 = \{u, s_1, s_2, \ldots, s_i, \ldots, s_k, v\}$ and $\mathcal{P}_2 = \{u, t_1, t_2, \ldots, t_j, \ldots, t_l, v\}$. There are three cases to consider from here. Cases 1 and 2 are illustrated in the diagram below where cycles are highlighted in red.

Case 1. Suppose that all s_i and t_j are distinct vertices. Then we can form the path $\{u, s_1, s_2, \ldots, s_k, v, t_l, \ldots, t_2, t_1, u\}$ which is actually a cycle, so once again we contradict the definition of a tree. ↯

Case 2. Suppose that exist one or more i and j such that $s_i = t_j$. Then, WLOG take the smallest such i and j and there would be a cycle $\{u, s_1, \ldots, s_{i-1}, s_i, t_{j-1}, \ldots, t_1, u\}$. Once again we contradict the definition of a tree. ↯

Case 3. Suppose that this time $s_i = t_j$ for all i and all j. Then paths $\mathcal{P}_1$ and $\mathcal{P}_2$ are identical, so there is just a unique path between u and v and this contradicts the assumption that paths were distinct. ↯ □



Proposition 2. Any graph with n vertices and n edges contains a cycle.

Proof. This is perhaps a bit intuitive, but here is how to show it rigorously. Proceed by contradiction and suppose a graph with n vertices and n edges doesn't contain a cycle. Now suppose we hypothetically start with a graph that has n vertices and no edges. So this graph starts off by having n connected components (i.e. at the start each vertex is not connected to any other vertex, so each vertex is just a connected component on its own).

As we start to add edges, if we add an edge that runs between two connected components, then the number of connected components decreases by one per every edge we add. If we add an edge within a connected component, then we introduce a cycle. So, avoiding adding edges within a connected component, after adding $n-1$ edges we will be down to just a single connected component. Since the addition of an edge to a connected component introduces a cycle, upon the addition of the n-th edge, we will have to introduce a cycle. $\lightning$ $\square$

Proposition 3. In any graph with n vertices and $n-1$ edges there exists a vertex v with $\deg(v) \leq 1$. In other words, some vertex is a "leaf" in such a graph.

Proof. Proceed by contradiction. Suppose that in a graph with n vertices and $n-1$ we have $\deg(v) > 1$ or equivalently $\deg(v) \geq 2$ for all vertices. Then:

- $\sum_{v \in V} \deg(v) \geq \sum_{v \in V} 2 = 2n$
- $2m = 2(n-1) = 2n-2$

so $\sum_{v \in V} \deg(v) \geq 2n > 2n-2 = 2m$ and we contradict the formula we shown earlier. $\lightning$ $\square$

Proposition 4. Let G be an undirected graph with n vertices. If any two of the following statements are true, then all three of the statements are true.

1. G is connected.
2. G is acyclic.
3. G has $n-1$ edges.

Proof that 1 & 2 imply 3. The holistic argument here is that 1 and 2 mean that G is a tree, and each vertex in a tree (expect for the root) has just a single parent (if there were two different ways to arrive at the same vertex, there would be a cycle). So therefore, there needs to be precisely $n-1$ edges to connect all the vertices.

A perhaps more rigorous argument is by induction on n, the number of vertices.

Base Case: Let $n = 1$ and then the graph is just a single vertex which of course has $n - 1 = 1 - 1 = 0$ edges.

Inductive Hypothesis: Let $n \geq 1$ and suppose that a connected and acyclic graph with n vertices has $n-1$ edges.

Inductive Step: We need to show that a connected and acyclic graph with $n+1$ vertices has n edges. So let graph G be a connected and acyclic graph with $n+1$ vertices, how can we make use of the inductive hypothesis? Well, we must turn G into a connected acylic graph with n vertices. To do so we must remove a vertex from G, so which vertex do we remove? As per proposition 3, some vertex v in G is a leaf with $\deg(v) \leq 1$. In fact, since G is connected, we know that $\deg(v) = 1$ for v in this case. So lets remove this vertex and call the resulting graph G'.

Since we removed a leaf vertex, then G' remains connected. Moreover, since G didn't contain any cycles, then of course G' must also be acylic. Now we can apply the inductive hypothesis to G' and deduce that G' has $n-1$ edges. Since we removed just a single edge, then G must have $(n-1)+1 = n$ edges as required. $\square$

Proof that 2 & 3 imply 1. Proceed by contradiction. Suppose that there is a disconnected, acyclic n vertex graph with $n-1$ edges. Since the graph is disconnected, it must be made up of at least two acyclic connected components. Suppose we were to add an edge to connect the two components, then we would have an acyclic graph with n vertices and n edges. However such existence of such a graph contradicts proposition 2. ↯ □

Proof that 1 & 3 imply 2. Proceed by contradiction. Suppose that there is a connected, cyclic n vertex graph with $n-1$ edges. Since the graph is cyclic, there must be a cycle of length $k < n$ that involves k vertices and k edges. That leaves $n-1-k$ edges to connect the remaining $n-k$ vertices, which is just enough to connect the $n-k$ vertices between themselves. However, there are now no edges left to connect the component of $n-k$ vertices to the cycle of the other k vertices, so the graph would be disconnected. ↯ □

Proposition 5. For any two vertices u and v in a graph, their connected components are either identical or disjoint.

Proof. Once again this statement is sort of intuitive, but here is the rigorous proof. Let u and v be arbitrary vertices in a graph. There are two cases to consider.

Case 1. Suppose there is a path that connects u and v. Let w be an arbitrary vertex in the connected component of u. Then w is also in the connected component of v as to reach w from v we could follow the path $\{v, u, w\}$. Since we shown that any *arbitrary* vertex that is in the connected component of u is also in the connected component of v, the connected components are identical.

Case 2. Suppose there isn't a path that connects u and v. In this case we claim the connected components are disjoint. Proceed by contradiction and suppose the connected components were not disjoint. Then then there is some vertex w in both, the connected component of u and in the connected component of v. So there would be a path between u and v, contradictory to the assumption of this case. ↯ □

Aside: Proof Strategies

1. Sketching out diagrams really helps!
2. Induction on the number of vertices in many cases is a good approach.
3. Contradiction, especially on facts such as "this graph is connected" or "this graph is acylic" or "this is graph is a tree", etc. are good approaches.

Proofs of larger propositions, such as proposition 4 are often made up of smaller building blocks, such as propositions 2 and 3 in this case. So when proving a large proposition and running into the need of some relatively small fact (e.g. a graph with n vertices and n edges must contain a cycle), it is not a bad idea to explore if this is indeed something that is true and/or to try to prove it to complete the larger proof.

The handout at this URL is quite comprehensive, some of the ideas for the proof of proposition 4 are from here: `https://www2.cs.duke.edu/courses/spring19/compsci230/Notes/lecture16.pdf`, page 4.

Representing Graphs. We can use either an adjacency list or an adjacency matrix to represent a graph. In an adjacency matrix, for each combination of vertices we store either a boolean or a weight value. In an adjacency list, for a given vertex, we store a linked list of vertices that could be reached from that vertex. The nodes of the linked list store the vertex ids and weights in a weighted graph.

Adjacency list takes up $\mathcal{O}(n+m)$ space, while the adjacency matrix takes up $\mathcal{O}(n^2)$ space. Run times of typical operations for each of those implementations are summarized below.

	Adjacency Matrix	Adjacency List
Insert Vertex	$\mathcal{O}(n)$	$\mathcal{O}(\deg(v))$
Remove Vertex	$\mathcal{O}(n)$	$\mathcal{O}(\deg(v))$
Insert Edge	$\mathcal{O}(1)$	$\mathcal{O}(1)$
Remove Edge	$\mathcal{O}(1)$	$\mathcal{O}(\deg(v))$
Vertices Adjacent?	$\mathcal{O}(1)$	$\mathcal{O}(\deg(v))$
Incident Edges	$\mathcal{O}(n)$	$\mathcal{O}(\deg(v))$

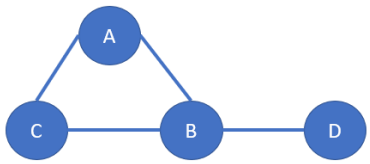
The adjacency matrix implementation is very efficient at inserting, removing edges or at telling if there is an edge between two vertices as all of those require just doing a constant time look up or a constant time edit of some entry in the matrix. In the adjacency list implementation we may need to traverse an entire adjacency list to determine if two vertices are adjacent. To remove an edge, we may also need to traverse an entire list to locate the edge. So both of those run times are $\mathcal{O}(\deg(v))$.

The adjacency matrix implementation is less efficient at inserting or removing new vertices as those require modifying entire rows of the matrix. In some cases the matrix may need to be resized which pushes the run time to $\mathcal{O}(n^2)$ for the particular insertion. In the adjacency list implementation, we modify the relevant linked lists which contain $\deg(v) \leq n$ elements. Overall, the adjacency list implementation works better with sparse graphs.

The "incident edges" operation specified in the last row would return the list of all edges incident to a given vertex. This operation is a also bit more efficient when using the adjacency list representation with sparse graphs.

Example.

- There are just zeros along the diagonal of the adjacency matrix, which means that there are no self-edges in this graph.
- The adjacency matrix is symmetric (i.e. it is equal to its transpose), which means that this is an undirected graph.



Adjacency Matrix

	A	**B**	**C**	**D**
A	0	1	1	0
B	1	0	1	1
C	1	1	0	0
D	0	1	0	0

Adjacency Lists

A → B → C

B → A → C → D

C → A → B

D → B

3.2 BFS and DFS

The following pseudo code is a rough outline of an algorithm that could be used to traverse a graph. There s represents the starting point, the vertex at which we would start the traversal. Something that is of a significant concern when traversing a graph (as opposed to say traversing a tree) is that the algorithm actually terminates at some point and we don't keep going in cycles. This concern is certainly addressed in the following algorithm as we do have keep a set of "visited" vertices R and run the while loop only for as long as there are edges that reach beyond R. So this algorithm does indeed terminate.

However, something that the following algorithm fails to specify is how do we look for such edges. In the worst case, we might find ourselves looking through all m edges in the graph on each iteration to see if they fit the criteria we are looking for. Then, as each iteration of the loop would add one vertex to R, we are pushing a run time of $\mathcal{O}(mn)$ with this algorithm. For example, suppose we start at some vertex a the neighbors of which are vertices b, c and d. Then we find the edge (a, b), add b to R and ignore all other edges stemming from a. Having lost that information, on the following iteration we would be back to square one, once again checking all edges of the graph to find the other edges that stem from a.

```
1  traverse(G, s) { // [Kleinberg, pg. 82]
2      define R to be the set of vertices to which s has a path to, init. R = {s}
3
4      while (there is an edge (u,v) where u in R && v not in R) {
5          add v to R
6      }
7  }
```

To reduce the run time involved in the traversal, the algorithm needs a bit more structure. In particular, we need some sort of a data structure to use as a list to track the vertices that we discovered so we don't have to keep going back to the starting point all the time. The decision left to be made is should the list of vertices be stored in FIFO or LIFO order? In other words, should the data structure be a queue or a stack? The former leads to breadth first search (BFS) while the latter leads to depth first search (DFS).

	BFS	**DFS**
Data Structure	Queue	Stack or Recursion
Methodology	Stay close and explore the graph in layers	Go out as far as possible and then backtrack
Mark v as Visited	Before adding to the queue	After popping from the stack

Pseudo-codes for BFS and DFS are given below. By traversing the graph using BFS or DFS we examine each edge just twice: once when we are passing its starting point and once when we are passing its end point.

Suppose the graph was implemented using adjacency lists, then it is quite efficient to pinpoint the neighbors of a given vertex. Given that we come across each edge twice, the for loop in lines 8-12 of the following code would run for a *total* of $2m$ times between *all* iterations of the while loop (this relates back to the sum of degrees of vertices formula). If there are disconnected vertices in the graph, the while loop could also run for up to n times

without triggering the for loop at all. So the total run time for BFS or DFS in this case is $\mathcal{O}(n+m)$.

If the graph is sparse then $m \in \mathcal{O}(n)$ and the run time becomes $\mathcal{O}(2n) = \mathcal{O}(n)$. If the graph is dense then $m \in \mathcal{O}(n^2)$ and the run time is $\mathcal{O}(n+n^2) = \mathcal{O}(n^2)$. So the actual run time depends on if it is the n or the m that is dominant in the sum.

Suppose the graph was implemented using an adjacency matrix. Then the for loop in lines 8-12 would need to examine the full row of the matrix each time (even if the row is empty) which pushes the run time to $\mathcal{O}(n^2)$ regardless if the graph is sparse or dense.

```
1  BFS(G, s) {
2      queue.enqueue(s)
3      mark s as visited
4
5      while (!queue.empty) {
6          u = q.dequeue()
7
8          for each edge (u,v) {
9              if (v is not visited) {
10                 mark v as visited
11                 q.enqueue(v)
12             }
13         }
14     }
15 }
```

The pseudo-code for DFS is almost identical, except that:

1. we of course use a stack instead of a queue and
2. we delay the point at which we label vertices as visited and therefore the order of the for loop and of the if statement switches relative to BFS

As far the second difference is concerned, in BFS all vertices are marked as visited *before* they are placed on the queue. So in BFS a visited vertex just means it shouldn't be added to the queue again; however, we will check the edges stemming from this vertex later on. In DFS, vertices are marked as visited *after* they are popped from the stack. So in DFS visited vertices are then completely done with, we don't check any of their edges.

So why do we do the latter? Well it has to do with treating cycles properly when it comes to DFS. Consider the following graph and suppose we start DFS at A. Had the code been written in the same order as in BFS, then DFS would proceed `A-B-C-E-D` and would hit D only after backtracking. However, we expect DFS to proceed `A-B-C-D-E` and hit D prior to backtracking. Essentially, with DFS we want to go to the vertex that is *furthest* away from A first, before starting to backtrack. The algorithm would still work if the order of the for loop and the if statement was kept the same as in BFS, just the resulting vertex traversal order would be different from what is expected.

This example is from `https://stackoverflow.com/questions/25990706/breadth-first-search-the-timing-of-checking-visitation-status`.

```
DFS(G, s) {
    stack.push(s)

    while (!stack.empty) {
        u = stack.pop()

        if (u is not visited) {
            mark u as visited
            for each edge (u,v) {
                stack.push(v)
            }
        }
    }
}
```

DFS could be implemented recursively (this is sort of how graphs are traversed in CPSC 110). Here, instead of using an explicit stack, we use the operating system's memory stack which stores the frames of all of the recursive function calls along with their local variables.

```
DFSRecur(G, u) {
    mark u as visited

    for each edge (u,v) {
        if (v is not visited) {
            DFSRecur(G, v)
        }
    }
}
```

Both BFS and DFS visit the same set of vertices but in a different order and have identical run times. So which algorithm should be used? Well if we are looking for some specific vertex, it depends if we expect this vertex to be close by or far away. In the first case, BFS would be more efficient, in the second case, DFS would be the choice. Also, since BFS traverses the graph layer-by-layer, BFS could be used to determine shortest distances to vertices in an unweighted graph.

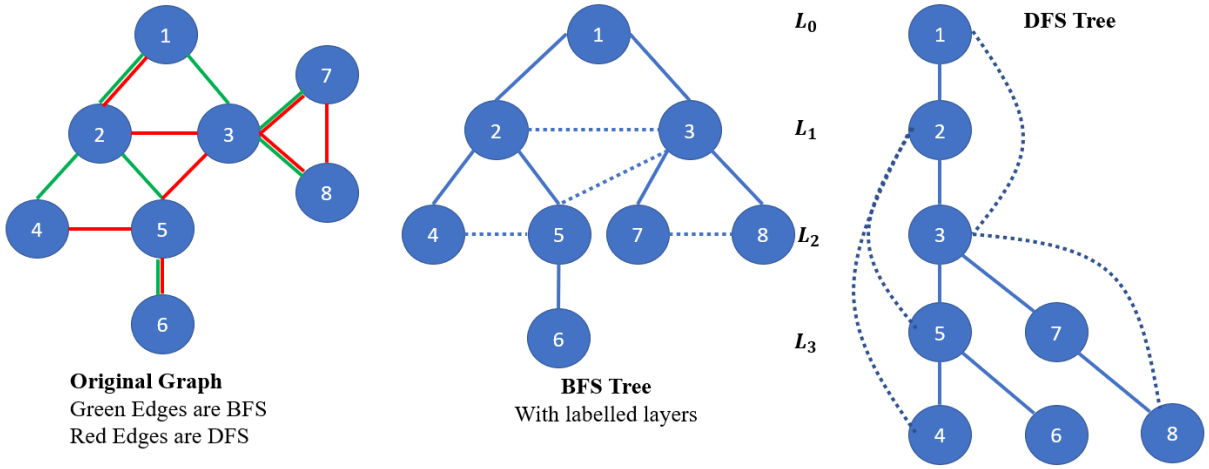
The set of visited vertices at termination of either algorithm is the connected component of the starting vertex s. Here is the algorithm to produce the set of ***all connected components of a graph***, just need to run BFS or DFS once for each component:

```
allConnectedComp(G) {
    define set R to be the set of all connected components, init R = {}

    while (there is an unvisted vertex v) {
        run BFS or DFS starting from v
        add the resulting connected component to R
    }

    return R
}
```

Structure of BFS and DFS Trees. If we were to follow the edges of the graph that contribute to the discovery of new vertices during BFS or DFS, those edges result in a tree. We will call the remaining edges present in the graph as *non-tree edges* and those are shown using dotted lines in the following diagrams. The final goal of this section would be to examine the *structure* of those trees and of the non-tree edges in particular. Understanding the structure of BFS and DFS trees comes in useful when proving correctness of algorithms that are build upon BFS and DFS (e.g. the algorithm for determining if a graph is bipartite, see section 3.3). The following graph that is used throughout examples is the same as in the previous section and is the same one as in figure 3.2 on page 79 of the Kleinberg textbook.

Summary Table.

	BFS Trees	**DFS Trees**
Visually	Look like typical balanced trees	Tend to be long and deep
Non-tree edges	Connect same or adjacent layers (see proposition 2)	Connect ancestors-descendants (see proposition 3)



Define layers $L_0, L_1, L_2, \ldots, L_k$ constructed by the BFS algorithm as follows

1. $L_0 = \{s\}$
2. $L_1 = \{\text{all vertices that are neighbors of } s\}$
3. Layer L_i consists of vertices that
   - have an edge to a vertex in layer L_{i-1} and
   - do not belong to an earlier layer

Proposition 1. For any $i \geq 1$, the layer L_i consists of all vertices that are at a (minimum) distance i from s.

Proof. This is sort of just a formalized restatement of the definition. Proceed by induction.

Base Case: Let $i = 1$ then L_1 consists of all neighbors of s. This means that all elements of L_1 have an edge to s, so all elements of L_i are precisely a distance of 1 from s.

Inductive Hypothesis: Fix $i > 1$ and assume that layer L_{i-1} consists of all vertices that are at a distance of $i - 1$ from s.

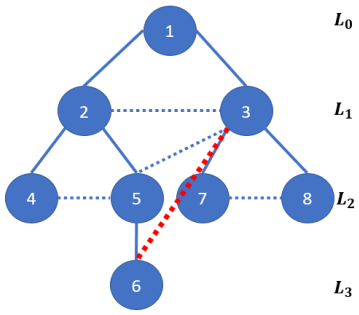
Inductive Step: Consider the layer L_i. By definition it consists precisely of all vertices that have an edge to a vertex in layer L_{i-1} and do not belong to an earlier layer. So all of those vertices are an extra edge away from s than the ones in L_{i-1}. Since, by the inductive hypothesis, all elements of L_{i-1} were at a distance of $i-1$ from s, then all elements of L_i are at a distance of i from s due to the extra edge, as required. □

Proposition 2. Let T be a BFS tree, let u and v be vertices in T belonging to layers L_i and L_j respectively, and let there be an edge (u, v). Then i and j differ by at most 1.

Essentially what the proposition is saying is that we can't have edges that jump over layers, like the edge highlighted in red in the diagram off to the side. This property is sort of intuitive given how BFS is set and layers L_i are defined, but the proof could be made rigorous by contradiction.

Proof. Proceed by contradiction and suppose that i and j differ by more than 1. In particular, WLOG, suppose that $j - i > 1$ or $j > i + 1$.

Since $u \in L_i$, the only vertices discovered by BFS from u would be those in the layer L_{i+1} or in earlier layers. In particular, all vertices discovered from u would belong to at most layers $L_0 \cup L_1 \cup \cdots \cup L_{i+1}$. Since v is connected to u by an edge, it should be discovered too, but then $v \in L_j$ and $j > i + 1$. ↯ □

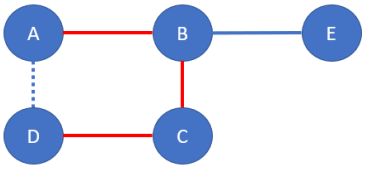


Proposition 3. Let T be a DFS tree, let u and v be vertices in T, and let (u, v) be and an edge that is *not* an edge of T. Then one of u or v is an ancestor of the other.

Proof. Suppose (u, v) is an edge in G and not an edge of T. WLOG, suppose that u is reached and is marked as visited by DFS first. Then:

- in any given `DFSRecur(G, u)` call, all vertices marked as visited between the start and end of this recursive call are naturally the descendants of u
- vertex v was not marked as visited initially when `DFSRecur(G, u)` was called
- however, since edge (u, v) wasn't eventually added to T, then v must been marked as visited sometime throughout execution of `DFSRecur(G, u)`
- therefore as per the first bullet point, v is a descendant of u □

Here is a practical example: we invoke DFS on vertex A and mark it as visited. All other vertices, including vertex D are not marked as visited. We first follow edges `A-B-C-D` as highlighted in red and mark D as visited. Eventually we will come back to A to investigate the edge `A-D`; however, by this time, D is already visited, so the edge isn't included in the tree. Of course D is a descendant of A.



Layers L_i in BFS and the ancestor-descendant relationship in DFS could easily be tracked while the respective algorithms run by adding a few additional lines to the existing code.

3.3 Bipartiteness

A graph $G = (V, E)$ is said to be ***bipartite*** if the set of vertices V can be split into two *disjoint* sets X and Y such that every edge $e \in E$ has one endpoint in X and the other endpoint in Y.

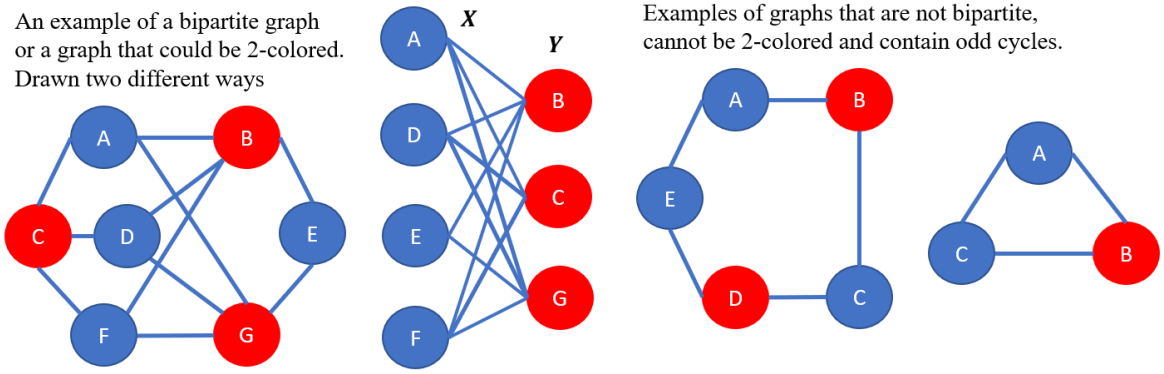
There are at least three other equivalent ways to characterize bipartite graphs and they are summarized below. The last characterization in terms of BFS trees naturally leads to an algorithm that could be used to determine if a graph is bipartite.

A graph is bipartite:

- iff it could be 2-colored (see below)
- iff it does not contain an odd cycle (see proposition 1)
- iff there does not exist an edge joining two vertices of the same layer in a BFS tree (see proposition 2)

Moreover all of the above are equivalent among themselves.

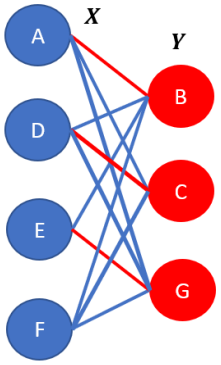
What does it mean to 2-color a graph? More broadly, the graph coloring problem asks if the vertices of a graph could be colored using k colors such that no two adjacent vertices are of the same color. This is actually quite an important problem with many applications in computer science, including compiler design, exam scheduling and will be studied in more detail towards the end of CPSC 320. At the moment, we will limit ourselves to the notion that a bipartite graph is precisely the one that could be 2-colored. In particular, as per the following illustrations, it is impossible to 2-color an odd cycle.



An aside: bipartite graphs arise in certain matching problems. Sets X and Y could represent sets of entities that need to be matched. While the focus in SMP was on rankings and there every entity in the opposing list was ranked, here the focus is more on compatibility. Absence of an edge between a vertex in X and a vertex in Y means that those two specific entities are not compatible and not meant to be matched. For example, if sets X and Y represent users and servers, we build a bipartite graph by drawing an edge between a user and every server that is compatible to service the user's request.

Such matching problems are called bipartite matching problems. As another cool example, suppose there is a set of empty boxes that we would like to store away in a way that they take up the least amount of space. Let sets X and Y both be identical copies of the entire set of boxes. We draw an edge from a box in set X to a box in set Y if one box fits inside the

other. The edges contained in the resulting matching would represent which boxes should be placed inside each other when being stored away.



Once a bipartite graph been constructed, the goal is to find a maximal matching: the largest set of non-adjacent edges. A perfect matching, if one exists, would be a matching that includes all vertices in the graph. Edges that make up the maximal matching are highlighted in red in the illustration on the side. Ford-Fulkerson algorithm is the algorithm to solve those problems and is described in chapter 7 of the textbook. It is rarely covered in CPSC 320, but is covered in CPSC 420.

Proposition 1. A graph G is bipartite iff it does not contain an odd cycle.

Proposition 2. A graph G is bipartite iff there does not exist an edge joining two vertices of the same layer in a BFS tree.

To avoid repetition, the two propositions will be proved simultaneously in four parts that build one upon the other. Part 1 of the proof corresponds to statement 3.14 in the textbook and parts 2 & 3 of the proof correspond to theorem 3.15 in the textbook.

Proof Part 1: $\Rightarrow$ direction of proposition 1. This direction states:

- if a graph G is bipartite then it does not contain an odd cycle.

The holistic argument is that it is just not possible to 2-color an odd cycle, so of course it is not possible to 2-color a graph that contains an odd cycle, so such a graph is not bipartite. Here is how to make the argument a bit more rigorous.

Proceed by contradiction and suppose that G is bipartite and contains an odd cycle $\{v_1, v_2, \ldots, v_n, v_1\}$ [here n has to be odd]. Since the graph is bipartite, then we should be able to split V into disjoint sets X and Y. Assign v_1 to X, v_2 to Y, v_3 to X, etc. In other words, assign v_i to X whenever i is odd and assign v_i to Y whenever i is even. Eventually we will assign v_n to X, but then v_1 is also in X, so there is an edge (v_n, v_1) with both endpoints in X and this contradicts the definition of a bipartite graph. $\lightning$ $\square$

Proof Part 2: $\Leftarrow$ direction of proposition 2. This direction states:

- if there does not exist an edge joining two vertices of the same layer in a BFS tree, then the graph is bipartite.

From proposition 2 in the previous section, we know that if there is an edge that runs between BFS tree layers L_i and L_j, then i and j differ by at most 1. In this case, since we assume that there does not exist an edge joining two vertices of the same layer, we eliminate $i = j$ and then i and j differ by *exactly* 1.

Therefore *every* edge joins vertices in *adjacent* layers. Having established that, it becomes quite easy to 2-color the graph: just color all L_i's for i odd one color and all L_j's for j odd the other color. Since it is possible to 2-color the graph, then the graph is bipartite. $\square$

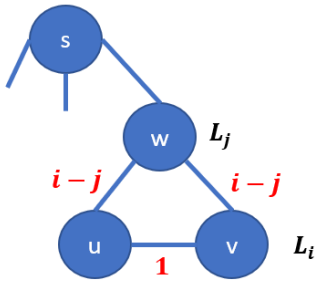
Proof Part 3: $\Rightarrow$ direction of proposition 2. This direction states:

- if a graph G is bipartite then there does not exist an edge joining two vertices of the same layer in a BFS tree.

We will proceed by contrapositive and the contrapositive of the above statement is:

- if exists an edge joining two vertices of the same layer in a BFS tree then G is not bipartite.

So suppose there is an edge $e = (u, v) \in E$ that joins two vertices in the same layer of a BFS tree T rooted at s. Let this layer be L_i and then $u, v \in L_i$. Let $w \in L_j$ be the vertex that is the lowest common ancestor of vertices u and v. In other words, w is an ancestor to both u and v and w is in L_j such that $j < i$ but j is the largest such j: see illustration.



Consider the cycle defined by $C = \{w, u, v\}$. Since all vertices in layers L_i and L_j are distances i and j away from the root of the tree s respectively, then distances `w-u` and `w-v` are both $i - j$. Therefore the length of the cycle C is $2(i-j)+1$ where the additional 1 comes from the edge `u-v`. But $2(i-j)+1$ is an odd number and therefore C has odd length. By part 1 of the proof, G is not bipartite as required. $\square$

Proof Part 4: $\Leftarrow$ direction of proposition 1. This direction states:

- if a graph G does not contain any odd cycles then it is bipartite.

Suppose that G does not contain any odd cycles, we need to show that G is bipartite. To show some graph is bipartite, we would need to construct sets X and Y that satisfy the definition. The approach is to split vertices of G into X and Y according to their minimal distance from some fixed vertex s. Typically, this part is quite technical; however, in our case, we can take advantage of the BFS tree layer framework we already established and its properties that we proved. Let T be a BFS tree rooted at s.

So split the vertices of G into sets X and Y based on the layers of T to which those vertices belong to. If $v \in L_i$ and i is even, then place v into X. If $v \in L_i$ and i odd, then place v into Y. Now, since layers L_i are by definition disjoint and every vertex in G belongs to some layer L_i, then sets X and Y will partition all vertices of G as required.

Ok, so we established how to partition vertices of G into sets X and Y. Is there anything else left to check to establish that G is bipartite? Yes, we need to satisfy the part of the definition that states that every edge $e \in E$ has one endpoint in X and the other endpoint in Y. Proceed by contradiction and suppose that exists an edge e whose both endpoints WLOG are in X. Then there are two possible cases to consider.

1. Edge e is an edge that joins two vertices in the same layer of a BFS tree. Then, by part 3 of the proof, G contains an odd cycle, which contradicts our assumption. $\lightning$
2. Edge e is an edge that joints two vertices in different layers of a BFS tree. Then, given that X was constructed to accept all vertices with an even distance away from s, the distance between two layers is at least 2. This is a contradiction to proposition 2 of the previous section. $\lightning$ $\square$

The following corollary establishes that results of propositions 1 and 2 hold for disconnected graphs as well.

Corollary. A graph G is bipartite iff all of its connected components are bipartite.

Proof. A graph G doesn't contain an odd cycle iff all of its connected components do not contain odd cycles. □

Algorithms to Check if a Graph is Bipartite. This is LeetCode problem no. 785. Well since once of the equivalent definitions of a bipartite graph involved the notion of BFS tree layers, then it might be natural to build the algorithm that checks if a graph is bipartite upon BFS.

One approach is to explicitly keep track of layers L_i during BFS, then color the adjacent levels in different colors and check all edges if there exists any edge for which both endpoints have the same color.

Another approach, the pseudo-code for which is given below is to define two colors as "true" and "false" and then just keep an array of booleans that stores the color of each vertex. If, when examining some vertex, we see that at least one of its neighbors is of the same color, we immediately know that the graph is not bipartite. Otherwise, we keep going and color the vertex the opposite of its parent's color.

A perhaps even more efficient approach is to use an array that would store "0" if the corresponding vertex hasn't been visited, "-1" if it been colored one color and "1" if it been colored the other color. Then we don't need a separate array that keeps tack of visited vertices and it still just as easy to check if connected vertices are of opposite colors.

It is also possible to use DFS as well, but perhaps is bit more convoluted.

```
isBipartite(G, s) {
    define color array
    queue.enqueue(s)
    mark s as visited
    color s as true

    while (!queue.empty) {
        u = q.dequeue()

        for each edge (u,v) {
            if (v is not visited && color[u] != color[v]) {
                mark v as visited
                color v as !(color of u)
                q.enqueue(v)
            } else if (color[u] == color[v]) {
                return false
            }
        }
    }

    return true
}
```

3.4 DAG and Topological Ordering

DAG stands for a *directed acyclic graph*. While undirected acyclic graphs are either connected or disconnected trees, directed ayclic graphs look like typical graphs, it is just that edges are directed in such a way that there aren't any cycles.

DAGs have many applications in situations where a graph is the right choice of data structure to represent data, yet cycles would lead to a "chicken-and-egg" problem. Typical situations involve precedence relations or dependencies and examples are course prerequisites, sequencing of tasks in the compilation process or management of resources in multi-threaded programming. It would suck to have a cycle in a course prerequisite chain as then we won't know which course to enroll in first, but a graph is certainly the right data structure to represent course prerequisites as several courses may be required in order to take a certain course. Likewise, if there happens to be a cyclic dependency in the C/C++ header file import statements, the compiler would fail to compile them.

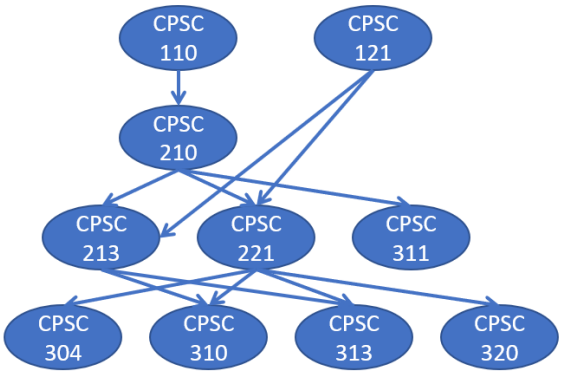
Some Terminology.

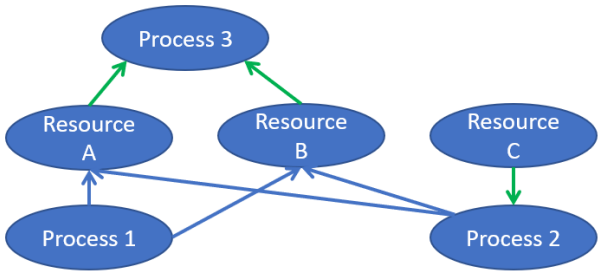
- **Topological Ordering:** is an ordering of vertices $\{v_1, v_2, \ldots, v_n\}$ such that for any edge $e = (v_i, v_j)$ we have $i < j$. In other words, all vertices are lined up in such a way that all edges point only forward or to the right.
- **In-Degree of a Vertex:** is the number of edge heads adjacent to the vertex [i.e. the number of edges that point to the vertex]. Out-degree of a vertex is defined in a similar way.

When working with DAGs it is natural to work with their topological orderings as those give a clearer picture of what is going on. For example, given a list of C/C++ header files import statements, the compiler would want to determine which header file should be compiled first. So the most common algorithm to run on a DAG is the algorithm that determines a topological ordering. A topological ordering need *not* be unique, there could be many valid topological orderings for some graphs.

Moreover, the existence of a topological ordering is an equivalent way to define a DAG. Just like in the previous section, the proof of this equivalency will naturally lead to the algorithm that produces the topological ordering.

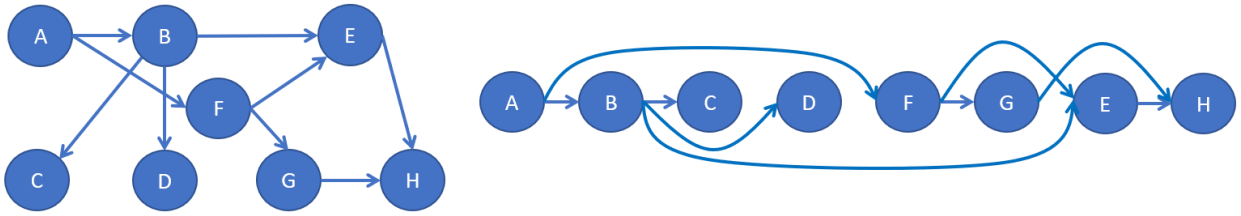
- A graph G is a DAG iff G has a topological ordering.





Blue arrows mean "process x waits for resource y"
Green arrows mean "resource y held by process x"
There are no cycle here, so there is no **deadlock**.

An example of a DAG and its topological ordering: note that all edges in the topological ordering point forward or to the right. Ordering $\{A, F, G, B, E, H, C, D\}$ is also a valid topological ordering while orderings $\{A, B, C, D, E, F, G, H\}$ or $\{A, F, B, C, D, G, H, E\}$ are not. For example, in the latter, vertices H and E clearly appear in the wrong order.



Lemma. In every DAG, there is a vertex v with an in-degree of zero.

Proof. Proceed by contradiction and suppose that there is a DAG where the in-degree of *every* vertex is at least 1. In other words, suppose there is DAG where *every* vertex has at least one edge that points to it.

Pick any such vertex v, suppose the incoming edge is (u, v) and follow this incoming edge backwards to u. Since u also has an in-degree of at least 1, we can also follow some incoming edge (w, u) backwards from u to w. Since *all* vertices have an in-degree of at least 1 we can continue this process indefinitely until we reach v again. But then that is a cycle, so we contradict the definition of DAG. $\lightning$ □

Proposition. A graph G is a DAG iff G has a topological ordering.

Proof Part 1: $\Leftarrow$ direction. This direction is:

- if a graph G has a topological ordering, then G is a DAG.

Proceed by contradiction and suppose that G has a topological ordering $\{v_1, v_2, \ldots, v_n\}$ but is not a DAG. In other words, suppose that G has the specified topological ordering and also that G contains a cycle C.

Let v_i be the lowest-indexed vertex in C and let v_j be the vertex that comes before v_i in the cycle C [there has to be such a vertex since C is a cycle]. Then (v_i, v_j) is an edge but i was the smallest index; therefore, $j > i$ which contradicts the definition of a topological ordering. $\lightning$ □

Proof Part 2: $\Rightarrow$ direction. This direction is:

- if graph G is a DAG, then G has a topological ordering.

Proceed by induction on the number of vertices in the graph n.

Base Case: Let $n = 1$. The topological ordering for a single-vertex graph is just that vertex itself.

Inductive Hypothesis: Fix $n \geq 1$, and suppose that an n vertex DAG has a topological ordering $\mathcal{O}$.

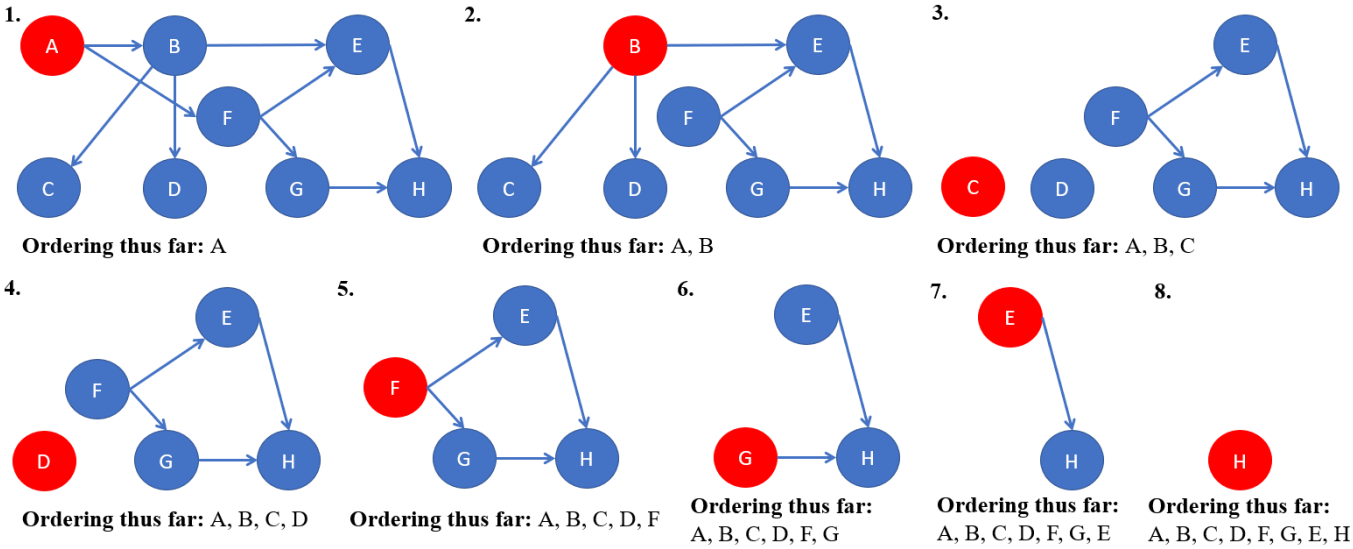
Inductive Step: Let G be an $n+1$ vertex DAG. To make use of the inductive hypothesis, we need to turn G into an n vertex graph (sounds familiar, we used this technique in the proof of proposition 4 in section 3.1). So which vertex should we remove from G?

According to the lemma, there exists a vertex with an in-degree of zero, or a vertex with no incoming edges. Let v be this vertex. Remove v from G, its removal will clearly not introduce any cycles to G. Therefore, $G - \{v\}$ is also a DAG and by the inductive hypothesis has a topological ordering $\mathcal{O}$. We can append v to the front of this topological ordering and since v has no incoming edges, then $\{v\} \cup \mathcal{O}$ will be a valid topological ordering as required. □

NOTE. In the inductive step, we choose to remove a vertex with no incoming edges as in foresight that is precisely the vertex we would want to add to the topological ordering $\mathcal{O}$ to not mess up the ordering.

Algorithms to Determine a Topological Ordering of a Graph. The first possible algorithm falls naturally out of the proof by induction we just did. If the graph is represented using a pair of adjacency lists for each vertex (one list for incoming edges and one list for outgoing edges), the run time here is $\mathcal{O}(n^2)$. We would need n iterations to order n vertices and each iteration would take $\mathcal{O}(n)$ time to identify and delete a vertex with no incoming edges. To do the latter, we would need to scan through all vertices to see which one has an empty incoming adjacency list. Pseudo code and an illustrated example are shown below.

```
topOrderingRecur(G) { // [Kleinberg, pg. 102]
    find a vertex v with an in-degree of zero and add it to the ordering
    delete v from G

    return append(ordering, topOrderingRecur(G)) // recursive call
}
```

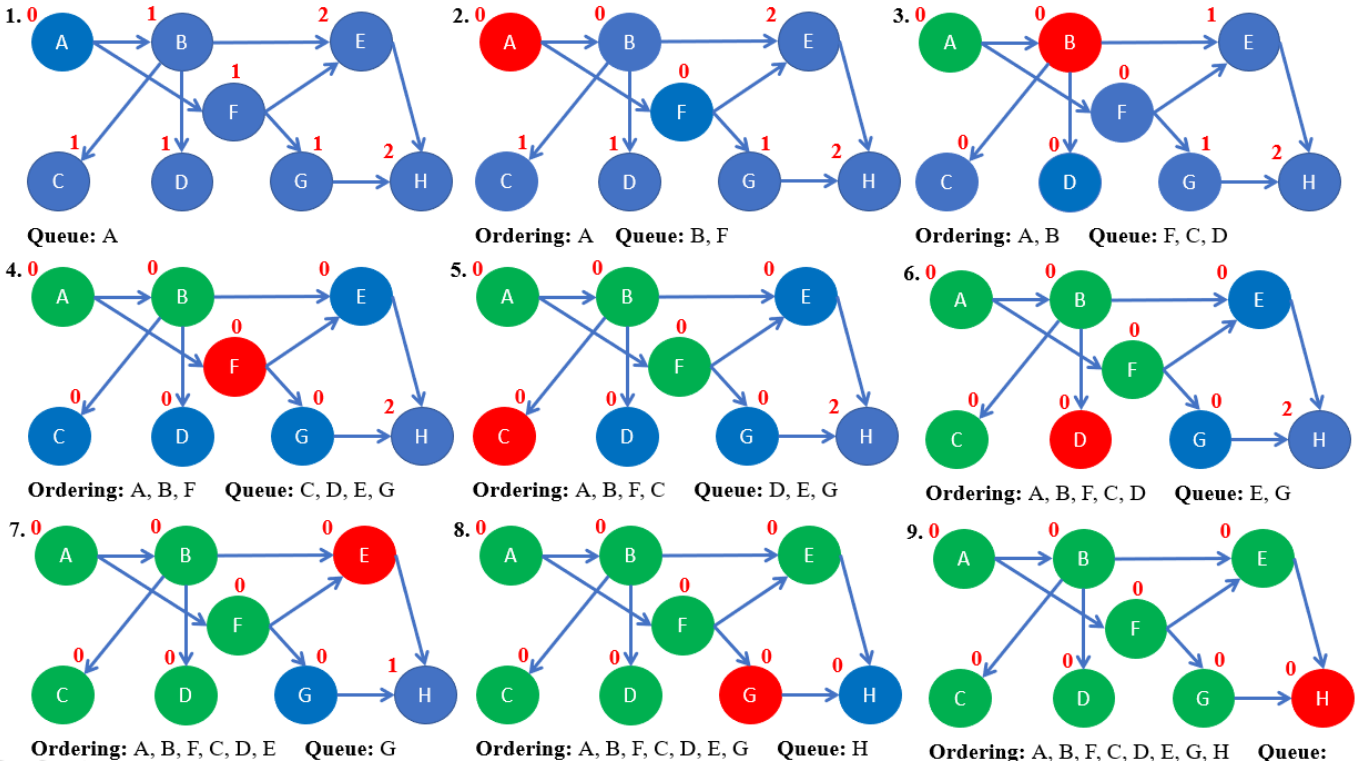


An algorithm that is more efficient for sparse graphs runs in $\mathcal{O}(n+m)$. The pseudo-code and an illustrated example are shown below. Once again, if the graph is implemented using adjacency lists, then step $*$ takes $\mathcal{O}(n+m)$ to traverse all of the adjacency lists and step $**$ requires $\mathcal{O}(\deg(v))$ per iteration of the while loop for a *total* of $\mathcal{O}(n+m)$ over all iterations of the while loop.

```
topOrdering(G) {
  initialize the result
  compute the in-degree of each vertex // *
  initialize a queue to contain all vertices with an in-degree of zero

  while (!queue.isEmpty()) {
    dequeue vertex v from the queue and add it to the result
    reduce the in-degree of all vertices adjacent to v by one // **
    enqueue any vertices that now have an in-degree of 0
  }

  return result
}
```

In this example:

- in-degrees are labeled in red
- vertex being currently processed is shown in red
- vertices that been processed are shown in green

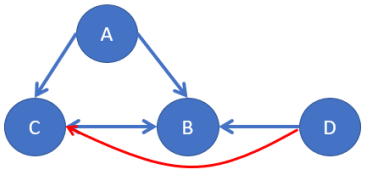


Finally, a third possible approach is to use DFS. The motivation to use DFS here is that the topological ordering looks like one of those elongated DFS trees. The specific algorithm is to run DFS and keep track of the "finish time" for each of the vertices. Then in the topological ordering, we output the vertices in the *reverse* order of their finish time (i.e. the vertex that finished first is the one that didn't have any incoming edges).

3.5 Additional Run Time Examples

Example 1. This is problem 22.1-5 from the "CLRS" algorithms textbook. Let G be a directed graph. We construct a *square* of G, denoted as G^2 by adding an edge to G whenever there is path of length 2 between some two vertices in G and such an edge doesn't exist in the original graph, see illustration. The task is to come up with pseudo-code and the run time for an algorithm that produces the square of a graph.

For example, to square the following graph, we would need to add the edge highlighted in red and keep all other edges as is. We add the edge highlighted in red since there is a path of length 2 given by `D-B-C`. There are no other paths of length 2 in this graph for which an edge doesn't already exist (for example path the `A-C-B` is already represented by the edge `A-B`).



Pseudo-code is given by below, it is essentially a set of nested for loops that determine which vertices are one or two hops away from any given vertex. If the graph is represented using adjacency lists, then we can run through all edges in the adjacency lists in $\mathcal{O}(m)$ time [this corresponds to the two outer most for loops]. For every edge, we then check at most n neighbors of the *tail* end of the edge in $\mathcal{O}(n)$ [this corresponds to the inner most for loop]. If we were to represent G^2 using an adjacency matrix, then insertion of each edge is $\mathcal{O}(1)$ and the total run time is $\mathcal{O}(nm)$. If we were to represent G^2 using adjacency lists, then additional time would be required to remove any duplicates from the resulting lists once all edges been inserted.

Now, had the original graph been represented using an adjacency matrix, then there would be just three nested for loops without any "optimization" and the run time would be $\Theta(n^3)$. Of course, even with an adjacency lists representation, in a dense graph with $m \in \mathcal{O}(n^2)$ we also get $\mathcal{O}(n^3)$.

As an aside: what is beautiful about this problem is that if the original graph is represented using an adjacency matrix A, then the adjacency matrix representation of G^2 is $A + A^2$. So essentially squaring the adjacency matrix produces the square of the graph! If we were to carefully write out the equation of matrix multiplication, then we would observe that the product of any given row and column sort of "checks" all possible combinations of traveling from the row vertex to the column vertex in two hops. Coincidentally, the run time of matrix multiplication is also $\mathcal{O}(n^3)$.

```
1 for each vertex v in G {                    } O(m) : the nested loops cover all edges (v,u)
2     for each neighbor u of v {              }
3         add edge (u,v) to G^2 // copy exisiting edges
4
5         for each neighbor w of u            }
6             add edge (u,w) to G^2           } O(n) // add new edges
7     }
8 }
```

Example 2. Suppose that a graph G is represented using adjacency lists. What is the run time to determine if G contains a triangle (that is, three vertices u, v, w such that u and v are adjacent, u and w are adjacent, and v and w are adjacent)?

The run time is $\mathcal{O}(n^2m)$. For each edge (u, v) in the graph and for each vertex w in the graph, we need to check if w is in u's adjacency list and whether w is in v's adjacency list. So given n vertices and m edges, there are nm checks in total. Each such check takes $\mathcal{O}(n)$ as discussed in the previous example. So the total time is $\mathcal{O}(n^2m)$ and if the graph is dense we are pushing $\mathcal{O}(n^4)$.

We could cut the overall run time down a bit by converting implementation of the graph into an adjacency matrix. The conversion takes $\mathcal{O}(n^2)$ and then each check is just $\mathcal{O}(1)$. In that case the run time is $\mathcal{O}(n^2 + nm)$. If the graph is sparse we get $\mathcal{O}(n^2)$ and if the graph is dense we get $\mathcal{O}(n^3)$.

This was an assignment problem in 2018W2 but is otherwise a quite standard problem with much research devoted to getting a run time that is better than ones discussed here. `https://en.wikipedia.org/wiki/Triangle-free_graph`

NOTE. In the previous example, the adjacency lists representation worked best as we wanted to just discover all possible edges to discover all possible paths of length 2. In this example, the adjacency matrix representation is better as we are checking to see if certain edges exist in specific locations. x

```
1  for each vertex w in G {                                  O(n)
2
3      for each vertex v in G {                              }
4          for each neighbor u of v {                        } O(m) : the nested loops cover all edges (v, u)
5
6              check: is w is a neighbor of v                }
7              check: is w is a neighbor of u                } O(n) or O(1)
8          }
9      }
10 }
```

4 Greedy Algorithms

An algorithm is *greedy* if it builds up the solution in small steps, making a short-sighted, or myopic, decision at each step with the goal of optimizing some underlying criterion. It is easy to come up with potential greedy algorithms, yet it is challenging to come up with the algorithm that produces an optimal solution and to then prove that is the case. It may be helpful to look at simple, special cases of the problem when coming up with the algorithm.

There are two techniques that are commonly used to prove that a greedy algorithm produces an optimal solution.

1. ***Greedy Stays Ahead.*** We use induction to show that the greedy algorithm stays ahead of any other solution at *each* step of the process (see section 4.1).
2. ***The Exchange Argument.*** We start with *any* solution to the problem and carry out a series of exchanges, to arrive at the solution that was produced by the greedy algorithm, while showing that each exchange doesn't undermine the quality of the solution. Therefore, the solution produced by the greedy algorithm must be at least as good as any other solution (see section 4.2).

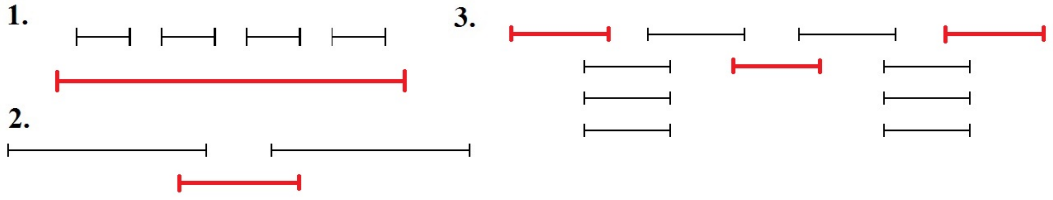
4.1 Interval Scheduling Problem

The Problem. Suppose we have a set E of n events. Each event i has a start time of $s(i)$ and an end time of $f(i)$. Two events are *compatible* if they do not overlap in time. We would like to find a subset $A \subseteq E$ of compatible events that is the largest in size.

The Algorithm. To solve this problem, our greedy algorithm, at each step, would pick some event i from set E, add it to set A, and then remove all events from E that are not compatible with i. It remains to determine which criterion we should use to pick the event i at each step. There are several possible choices here:

(1) pick the event that starts first
(2) pick the event with the shortest duration
(3) pick the event that is the most compatible
(4) pick the event that ends first

It turns out that the first three choices do not lead to an optimal solution. The image below illustrates counter-examples for each of the three cases. The solution produced by each of the potential greedy algorithms is highlighted in red, while an optimal solution takes up the top row in each image (in image 3, all events highlighted in red are the most compatible ones as they conflict with ≤ 3 events while all black ones conflict with 4 events each).



Essentially, criterion no. 4 schedules as many events as possible by "rushing" the scheduled events to finish up as fast as possible and to free up space for the other compatible events.

```
initially let E be the set of all events && let A be empty // [Kleinberg, pg. 118]

```

```
while (E is not empty) {
   choose an event i in E that finishes first
   add i to A
   delete all events from E that are not compatible with i
}
return A
```

The Proof. We will now prove that the greedy algorithm that, at each step, picks the event that finishes first, does indeed produce an optimal solution. Suppose that:

- our greedy algorithm returns the set $A = \{i_1, \ldots, i_m\}$ and the size of A is m, and
- an optimal solution is the set $O = \{j_1, \ldots, j_k\}$ and the size of O is k

The goal will be to show just that $m = k$, as the exact make up of sets might differ.

We begin by showing, by induction, that at each step of the process, the greedy algorithm *always stays ahead*. We will use $r = \min\{m, k\}$ to represent the current step and this will be the variable we will do induction on. Mathematically, we would like to show that the end times for all events in set A are earlier than the end times of the corresponding events in O, or that $f(i_r) \leq f(j_r), \forall r$.

Base Case. Let $r = 1$ and it is clearly true that $f(i_1) \leq f(j_1)$, as the greedy algorithm picks the event with the earliest end time before considering any other event.

Inductive Hypothesis. Fix $r > 1$ and suppose that $f(i_{r-1}) \leq f(j_{r-1})$ for this r.

Inductive Step. We need to show that $f(i_r) \leq f(j_r)$ $\boxed{\star}$.

The goal is to show that event j_r is available for our greedy algorithm to select at step r. If that is the case, then, at step r, the greedy algorithm will pick some event i_r that finishes first, which could be the event j_r or even an event that finishes earlier than j_r. So then $\boxed{\star}$ will certainly hold.

For event j_r to be available we must show that this event is compatible with all events chosen thus far. Mathematically we must show that $f(i_{r-1}) \leq s(j_r)$ $\boxed{\star\star}$. In other words, for j_r to be compatible, we must show that all events chosen thus far finish before j_r starts. We know that:

- all events in the solution set O are compatible so $f(j_{r-1}) \leq s(j_r)$
- the inductive hypothesis is $f(i_{r-1}) \leq f(j_{r-1})$

Putting those together we get that $f(i_{r-1}) \leq f(j_{r-1}) \leq s(j_r)$ and therefore $\boxed{\star\star}$ holds. As discussed above, the inductive step follows.

Having shown that the greedy algorithm always stays ahead, we now argue by contradiction that its solution is optimal. So, contrary to $k = m$, suppose that $k > m$ and then set O contains at least one additional event j_{m+1}. As shown above, $f(i_m) \leq f(j_m) \leq s(j_{m+1})$ and this extra event j_{m+1} will be available when the greedy algorithm terminates. In other words, $j_{m+1} \in E$ when the greedy algorithm terminates. However, the greedy algorithm terminates if and only if E is empty. $\lightning$ □

A possible approach to prove optimality via exchange argument is as follows: start with any solution to the problem, substitute an event one by one from the greedy algorithm's

solution. Then show that each event substituted in finishes at least as fast, if not faster, than the corresponding event in the solution we started off with. So then the greedy solution will schedule at least the same amount of events.

LeetCode problem no. 435 is based on this algorithm.

RUN TIME. The run time time for this algorithm is $\mathcal{O}(n \log n)$. This comes from the need to first sort the events by end times, which we can do in $\mathcal{O}(n \log n)$ at best. After the events have been sorted, we just pass through the list of those events once, which is done in $\mathcal{O}(n)$.

4.2 Weighted Interval Scheduling Problem

THE PROBLEM. Lets modify the previous problem in a couple of ways. We still have a set E of n events, which we must schedule on a single, shared resource in a non-overlapping way. However, this time, the events do not have a predetermined start and end time. We are free to schedule them whenever we wish. Also, we must schedule *all* of the events. What we do have for each event is:

- the length of time it needs to run l_i, and
- the importance (or weight, or priority) that it carries w_i.

The goal is to schedule events such that events of greater importance finish as soon as possible. We need some criteria to measure this. We introduce a measure of *completion time* c_i which is the cumulative sum of events' lengths up to and including the current event, in the order they are to be scheduled. For example, if the the first three scheduled events were $1, 2, 3$ units long respectively, their completion times would be $1, 3, 6$ units respectively. Finally, we define the weighted sum of completion times, and so the goal of the algorithm is to minimize this sum:

$$\min \sum_{i \in E} w_i c_i$$

i.e the c includes completion time of previous tasks

notice this is w*c (so i's weight times c which is completion time of everything before)

THE ALGORITHM. In this problem there are essentially two subproblems to analyze, two criteria to consider: length and importance. So we begin by considering those two subproblems separately which will allow us to get a better understanding of the original problem. So we consider the subproblems where:

(1) all events have equal lengths, but different weights
(2) all events have equal weights, but different lengths

It is not too difficult to see that, to minimize the weighted sum, in the case of (1) we would want to schedule the event with the largest weight first. Then, in the case of (2) we would want to schedule the shortest event first.

Now, once we established how to tackle each of the special cases, we would like to come up with a way to combine those observations. In the combined problem, there is a *trade-off* between length and weight at play, so we would need to come up with some expression that *relatively* compares l_i and w_i for each event.

Two possibilities on how to combine w_i and l_i that come to mind are:

i. schedule the problem with the largest $w_i - l_i$ first
ii. schedule the problem with the largest w_i / l_i first

Length is the term that we subtract and place in the denominator, because we want our criteria to be *inversely* proportional to the length variable. In other words, as length decreases, we would want $w_i - l_i$ or w_i/l_i to increase, so that events of shorter length would be scheduled before others. Weight, on the other hand, should of course be positive and be in the numerator, so that events with the greater weight are to be scheduled before other.

Generally, there could only be one correct approach for a greedy algorithm and the following example shows that approach (i) is not correct. Suppose we have the following events: $l_1 = 5, w_1 = 3; l_2 = 2, w_2 = 1$.

- Approach (i): we get $w_1 - l_1 = -2$ and $w_2 - l_2 = -1$ so we schedule event 2 first. The completion times are $c_1 = 7, c_2 = 2$ and the weighted sum is $3 \cdot 7 + 1 \cdot 2 = 23$.
- Approach (ii): we get $w_1/l_1 = \frac{3}{5}$ and $w_2/l_2 = \frac{1}{2}$ so we schedule event 1 first. The completion times are $c_1 = 5, c_2 = 7$ and the weighted sum is $3 \cdot 5 + 1 \cdot 7 = 22$.

To summarize, our proposed greedy algorithm is the one that schedules the problem with the largest w_i/l_i first. The next step is to rigorously prove its correctness.

The Proof. We will proceed via an exchange argument. Suppose that we start from a set of events $E = \{e_1, \ldots, e_n\}$. For simplicity and WLOG, suppose that the lengths and weights of events e_i are ordered as follows (if they are not, we can simply rename the events to arrive at this order):

$$\frac{w_1}{l_1} \geq \frac{w_2}{l_2} \geq \cdots \geq \frac{w_{n-1}}{l_{n-1}} \geq \frac{w_n}{l_n}$$

Therefore, based on the criteria we picked, our greedy algorithm will return the *order* $A = \{e_1, \cdots, e_n\}$ as the solution. Now, suppose there exists another arbitrary solution B. If solution $B = A$, then we are done. If $B \neq A$, then there exists at least one pair of *consecutive* events e_i, and e_j in B that are not in the same order as they are in A. In particular, given that events in A are ordered such that their indices are consecutive and are increasing, it means that in B there are indices i, j such that $i < j$ but e_j comes before e_i.

To start turning B into A, we must switch the order of events e_i and e_j. In other words, we must perform an exchange. As $i < j$, then we know the following mathematical fact holds which we will save for now to make use of later on:

$$\frac{w_i}{l_i} \geq \frac{w_j}{l_j} \quad \rightarrow \quad w_i l_j - w_j l_i \geq 0 \quad \boxed{\star}$$

Let us consider the weighted sum of completion times for A and B. We will focus only on the parts that are concerned with events e_i and e_j as those are the only parts that would change during the exchange. So those events are isolated from the weighted sums:

$$\text{WSum}_A = \sum_{k \in E} w_k c_k = \sum_{k \in E; k \neq i,j} w_k c_k + w_i c_i + w_j c_j$$

$$\text{WSum}_B = \sum_{k \in E} w_k c_k = \sum_{k \in E; k \neq i,j} w_k c_k + w_i c_i + w_j c_j$$

Let L denote the sum of lengths of events that come before e_i and e_j. In A, as e_i comes before e_j then $c_i = L + l_i$ and $c_j = L + l_i + l_j$. In B, as e_j comes before e_i, those expressions switch so $c_i = L + l_i + l_j$ and $c_j = L + l_j$. Substituting those results into expressions for $\text{WSum}_A, \text{WSum}_B$:

$$\begin{aligned}
\text{WSum}_A &= \sum_{k \in E; k \neq i,j} w_k c_k + w_i(L + l_i) + w_j(L + l_i + l_j) \\
&= \sum_{k \in E; k \neq i,j} w_k c_k + w_i L + w_i l_i + w_j L + w_j l_j + w_j l_i \\
\text{WSum}_B &= \sum_{k \in E; k \neq i,j} w_k c_k + w_i(L + l_i + l_j) + w_j(L + l_j) \\
&= \sum_{k \in E; k \neq i,j} w_k c_k + w_i L + w_i l_i + w_i l_j + w_j L + w_j l_j
\end{aligned}$$

The only terms that differ between the two sums are highlighted in red. We subtract the two sums and get expression $\boxed{\star}$ which we already know is ≥ 0:

$$\text{WSum}_B - \text{WSum}_A = w_i l_j - w_j l_i \geq 0 \quad \Longrightarrow \quad \text{WSum}_B \geq \text{WSum}_A$$

So the weighted sum for A is smaller than the weighted sum for B, which means that after carrying out the exchange, we are better off. Moreover, there will be at most m events out of place in B. We can carry out such an exchange for each of those m events and therefore A will always be better off. □

This example is from Stanford's Coursera Algorithms course and a simpler variant of it appeared on the 2018S2 midterm.

4.3 Dijkstra's Algorithm

Dijkstra's algorithm is a greedy algorithm that finds the shortest path from some fixed source vertex s to any vertex other vertex in a weighted graph.

- If we want to determine the shortest distance from the fixed source vertex s to some fixed destination vertex d in the graph, then we can just stop the algorithm early as soon as it reaches d.
- The graph could be directed or undirected, in the latter case we just assume that all edges are bidirectional.
- The algorithm fails if there are edges with *negative* weights (see example a bit further down). Bellman-Ford algorithm is a dynamic programming algorithm that addresses the negative weight issue (see section 6.3).
- Dijkstra's algorithm is a *single-source shortest path* (SSSP) algorithm. So we have to specify some starting point where the path starts and then re-run the algorithm every time the starting point changes. Floyd-Warshall algorithm is the algorithm that determines the shortest paths between *all* pairs of vertices in a single run.

First, lets introduce some notation that we will use as we traverse the graph:

- let V be the set of all vertices for the graph [as before]
- let S be the set of all visited vertices for the graph
- let u represent a visited vertex
- let v represent an unvisited vertex
- let w represent the closest unvisited vertex
- let $e = (u, v)$ be an edge between vertices u and v with length l_e
- let $d(x)$ be the shortest distance from source s to some vertex x
- let $d'(x)$ be the shortest distance from source s to some vertex x at the current step

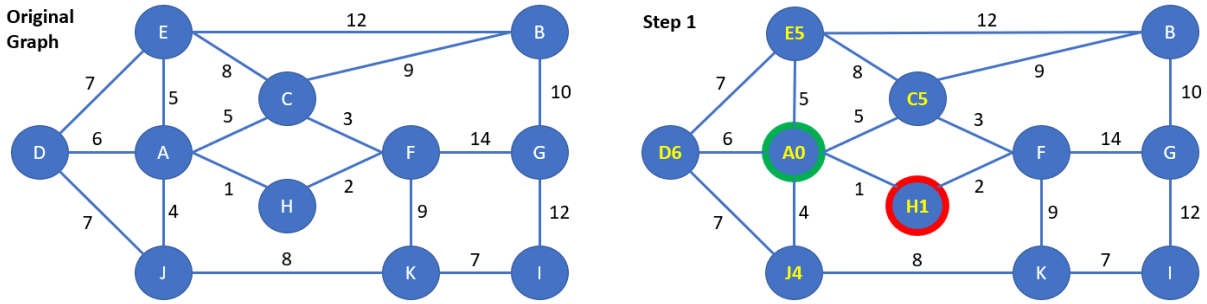
We initialize the set of visited vertices to be just the source vertex: $S = \{s\}$, and declare $d(s) = 0$. Whenever we take the next step to add a vertex to the visited set, we add the *closest unvisited* vertex w and set its distance. We do so by scanning over *all* unvisited vertices $v \notin S$ that are adjacent to *all* visited vertices $u \in S$:

$$d(w) = \min_{v \notin S} \Big[d'(v) \Big] = \min_{v \notin S} \left[\min_{e=(u,v):\ u \in S} [d(u) + l_e] \right]$$

So the first (inner) minimum in the formula determines the minimum distance to *some fixed reachable vertex* [if there are multiple ways to get to that vertex from S] while the second minimum takes the minimum of all of those distances over all reachable vertices.

```
Dijkstras Algorithm (G,s) { // [Kleinberg, pg. 138]
    let S be the set of visited vertices
    with each u in S, we store a distance d(u)
    initially S = {s} and d(s) = 0

    while (S != V) {
        look through all vertices v not in S which are one edge away from S so that
            d'(v) = min (over all edges starting from any u in S) of {d(u) + le}
        is as small as possible, call the vertex that gives minimal distance w

        set S := S U {w}, and d(w) := d'(w) // could also keep track of predecessor
    }                                      // to then print out the actual minimal path
}
```

Example. Consider the following graph and start from vertex A.



STEP 1: $S = \{A\}$ with distance of $\{0\}$.

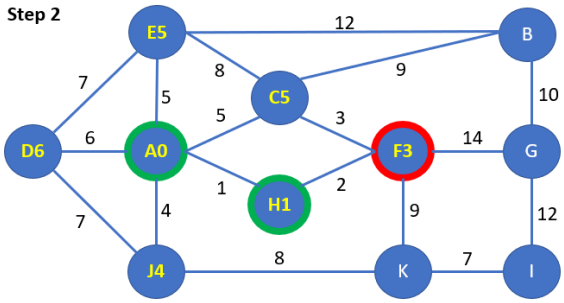
- from A we can reach C, D, E, H, J with distances of $5, 6, 5, 1, 4$
- the minimum of those is 1, so we select vertex $w = H$, add it to S and set $d(H) = 1$

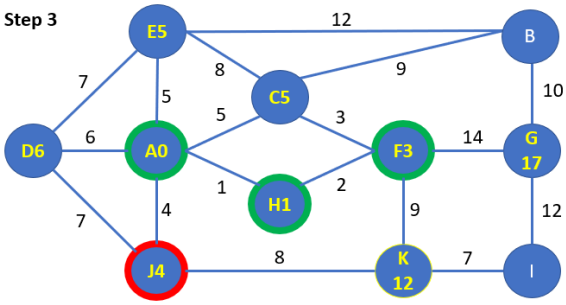
STEP 2: $S = \{A, H\}$ with distances of $\{0, 1\}$ respectively.

- from A we can reach C, D, E, J with distances of $5, 6, 5, 4$
- from H we can reach F with a distance of 3 [here $d(u) = 1$ and $l_e = 2$]
- the minimum of those is 3, so we select vertex $w = F$, add it to S and set $d(F) = 3$

STEP 3: $S = \{A, H, F\}$ with distances of $\{0, 1, 3\}$ respectively.

- from A we can reach C, D, E, J with distances of $5, 6, 5, 4$ [as before]
- from F we can reach C, G, K with distances of $6, 17, 12$
- there are no vertices not already in S that could be reached from H
- therefore, the minimum distances to C, D, E, J, G, K at this step are $5, 6, 5, 4, 17, 12$ respectively [here the first minimum in the formula determines the minimum distance to C, which is the minimum of 5 via `A-C` and of 6 via `A-H-F-C`, this is really the first time in the algorithm were the we had to non-trivially apply the first minimum]
- the minimum of all of those is 4 [second minimum], so we add J to S with $d(J) = 4$





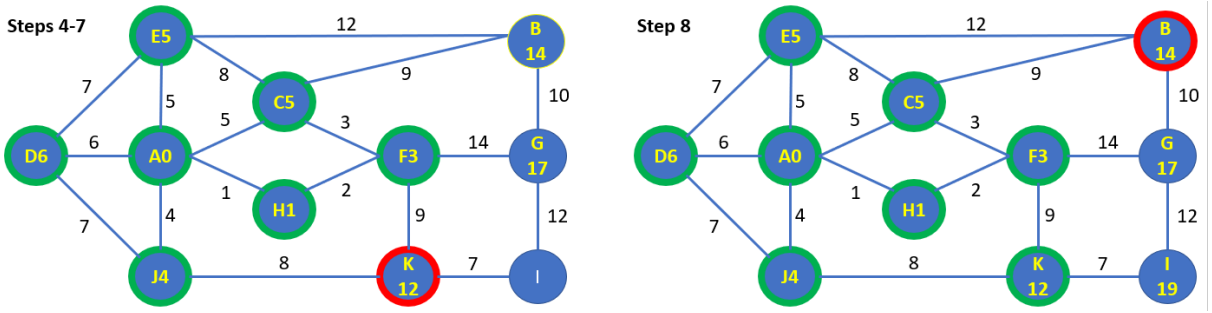
In steps 4-6 we will add vertices C, E, D to S as it will be the case that those vertices are the ones with shortest distances from A in those steps.

STEP 7: $S = \{A, H, F, J, C, E, D\}$ with distances of $\{0, 1, 3, 4, 5, 5, 6\}$

- from F we can reach G, K with distances of $17, 12$
- from J we can reach K with a distance of 12
- from C we can reach B with a distance of 14
- from E we can reach B with a distance of 17
- there are no vertices already not in S that could be reached from A, H or D
- therefore, the minimum distances to B, G, K at this step are $14, 17, 12$ [here we once again apply the first minimum to determine the distance to K and B; in the case of K the minimum was between `A-H-F-K` and `A-J-K`, but either of those was just 12]
- the minimum of all of those is 12 [second minimum], so we add K to S with $d(K) = 4$

STEP 8: $S = \{A, H, F, J, C, E, D, K\}$ with distances of $\{0, 1, 3, 4, 5, 5, 6, 12\}$

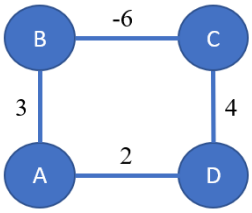
- from F we can reach G with a distance of 17
- from C we can reach B with a distance of 14
- from E we can reach B with a distance of 17
- from K we can reach I with a distance of 19
- there are no vertices that are not in S that could be reached from A, H, J or D
- therefore, the minimum distances to B, G, I at this step are $14, 17, 19$ [application of the first minimum on B, comparison of 14 via `A-C-B` and of 17 via `A-E-B`]
- the minimum of all of those is 14 [second minimum], so we add B to S with $d(B) = 4$



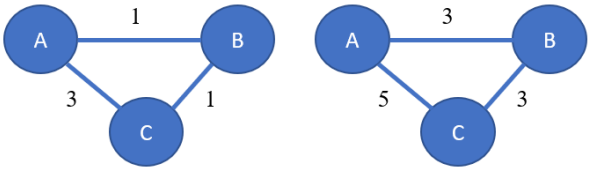
We finish off by adding G and I to S with distances of 17 and 19 in the last two steps. Then the set S is equal to the entire set of vertices V so we exit the while loop.

REMARKS.

1. Here is an example of how Dijkstra's algorithm fails if some of the edge weights are negative. The shortest path from A to D is `A-B-C-D` with a distance of 1 while Dijkstra's algorithm greedily picks the path `A-D` with a distance of 2. Greediness fails!



2. Shortest paths are *not* necessarily preserved if all edge weights are increased or decreased by the same constant weight. In the first case, the shortest path from A to C is `A-B-C` with a total distance of 2. In the second case [here we add 2 to each edge], the shortest path from A to C is `A-C` with a total distance of 5.



3. Shortest paths *are* preserved if all edge weights are scaled by a constant factor [e.g. all edge weights are multiplied or are divided by two].

PROOF. The strategy will be to show, by induction, that our greedy algorithm always stays ahead of the optimal solution. Proceed by induction on the size of set S.

Base Case. Suppose that $|S| = 1$, then $d(s) = 0$ which is clearly minimal.

Inductive Hypothesis. Fix $k \geq 1$ and suppose that our greedy algorithm provides an optimal solution when $|S| = k$.

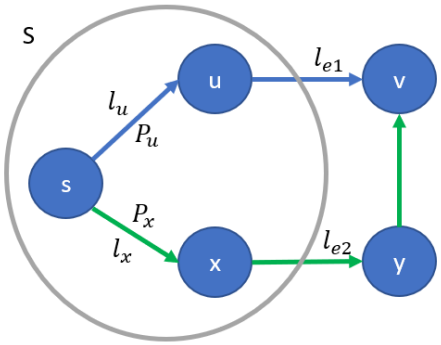
Inductive Step. We need to show that our greedy algorithm still provides an optimal solution when $|S| = k + 1$.

Suppose that we expand the set from $|S| = k$ to $|S| = k + 1$ by adding the vertex v. Let P_v be the path picked by our greedy algorithm from s to v. Let edge $e_1 = (u, v)$ be the last edge that was added to the path P_v.

Suppose that there exists another path from s to v, call it P. To show that the greedy algorithm stays ahead at this step, we must show that P_v is just as short or shorter that P.

In order to reach v, the path P must leave the set S at some point, as $v \notin S$. Let y be the first vertex on P that is not in S and let $x \in S$ be the vertex just before y (vertices x and y could be vertices u and v themselves, in that case $P = P_v$ and we are done).

Let P_u and P_x be the paths from s to u and s to x with lengths l_u and l_x, respectively. By the inductive hypothesis, since $u \in S$ and $x \in S$, we know that those are the *shortest paths* to u and x respectively.



When Dijkstra's algorithm makes the next choice, it considers all edges leaving S, including edges $e_1 = (u, v)$ and $e_2 = (x, y)$. Since Dijkstra's picks the edge that gives the smallest total length then:

$$l_{P_v} = l_u + l_{e_1} \leq l_x + l_{e_2} \leq l_x + l_{e_2} + l_{yv} = l_P$$

and we have satisfied the claim that the length of P_v [blue] is less than or equal to the length of P [green]. We didn't even need to consider the additional length of the edge between vertices y and v in P; however, we do assume its length is *greater than or equal to zero.* □

REMARK. Once again, this proof could fail if the graph contains negative weights and the problem is with the edge between y and v. Suppose that $l_u = 1, l_x = 2, l_{e_1} = 1, l_{e_2} = 2$ and the length of the edge between vertices y and v is -3. Then, the first bit of the above inequality holds:

$$l_u + l_{e_1} = 1 + 1 = 2 \leq 4 = 2 + 2 = l_x + l_{e_2}$$

However, the distance from s to v via u is 2, while the distance from s to v via x is just 1.

RUN TIME. Naive estimate of run time is $\mathcal{O}(mn)$, coming from the need to traverse n vertices and consider m edge lengths on each iteration.

However, we can use a priority queue and avoid recomputation of all distances on each iteration, cutting the run time down to $\mathcal{O}(m \log n)$. Vertices are stored in the queue with $d'(v)$ as their key [as we did with the numbers assigned to vertices in the diagrams that went along with the example]. It takes $\mathcal{O}(\log n)$ to extract the vertex with the minimal distance from a binary heap, for a total of $\mathcal{O}(n \log n)$ over n vertices.

The values of $d'(v)$ in the priority queue need to be updated at most once per edge [when the edge's tail is added to S] and each update is $\mathcal{O}(\log n)$ in a binary heap, for a total of $\mathcal{O}(m \log n)$. Therefore, overall run time for Dijkstra's is $\mathcal{O}(m \log n + n \log n) = \mathcal{O}(m \log n)$.

4.4 Prim's Algorithm

Recall that a minimal spanning tree (MST) of a graph G is a subgraph $M \subseteq G$ such that:

1. M spans G
2. M doesn't contain any cycles
3. the total edge length $\sum_{e \in E} l_e$ of M is as small as possible

so take in a connected graph and produces a spanning tryy (connected and acyclic graph) with all vertices of G with least total cost

For the all forms of MST algorithms presented here, we will assume that graphs are connected. For all *proofs*, we will assume that edge lengths are *distinct*.

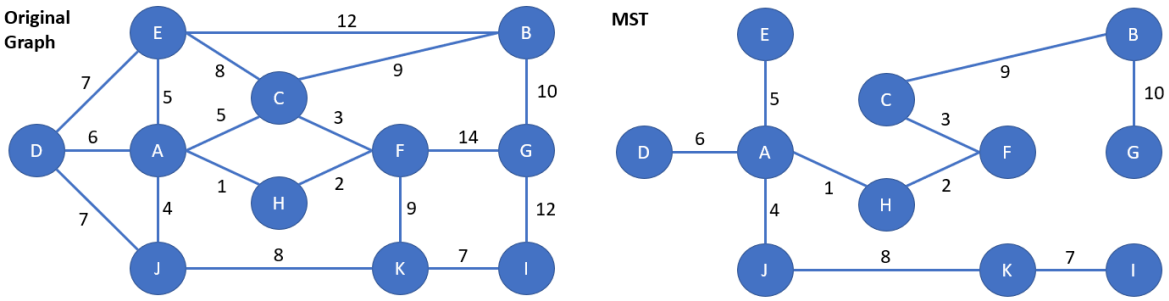
Prim's algorithm builds an MST in a way that is very similar to how Dijkstra's algorithm finds the shortest path [note the very similar structure of the respective pseudo-codes]. We start from a source vertex s and initialize our MST to an empty set. As before, let u represent vertices already visited and added to set S. At each iteration of the algorithm we look for the next edge $e = (u, v)$ to add to the MST and pick the edge with the smallest length l_e. We do so by scanning over all edges originating from any $u \in S$. The while loop exits once $S = V$, so in other words once our MST includes all vertices of G, as required.

Prim's algorithm is just a bit simpler than Dijkstra's, as we do not need to keep track of any cumulative distances. Via a priority queue, run time is also $\mathcal{O}(m \log n)$.

```
Prim(G) {
    choose a vertex s
    set S := {s}, T := empty set

    while (S != V) {
        look through all vertices v not in S which are one edge away from S so that
            min (over all edges starting from u in S) of {le} // different from Dijkstra
        is as small as possible, call the vertex that gives minimal edge length w

        set S := S U {w} and T := T U {e}
    }
}
```

Example. Edges are added to the MST in the following order:

1. AH
2. HF
3. FC
4. AJ
5. AE
6. AD
7. JK
8. KI
9. CB
10. BG



This page is concerned with preliminary facts about Prim's algorithm. The main part of the proof that Prim's algorithm greediness leads to an MST is on the following page.

Proof Part 1. Preliminary Facts About Cuts.

A ***graph cut*** is a partition of a graph $G = (V, E)$ into two *non-empty* disjoint sets. We use notation (A, B) to denote the cut, where A and B are the two non-empty disjoint sets.

For any graph there are $2^n - 1$ possible cuts. Graph cuts are a *natural way* to characterize MST algorithms as at any point in the algorithm we have the cut $(S, V - S)$.

Empty Cut Lemma. A graph is disconnected iff exists a cut with no crossing edges.

Proof. We will do the $\Leftarrow$ direction first. Let $G = (V, E)$ be a graph and suppose exists a cut (A, B) with no crossing edges. Pick a vertex $u \in A$ and $v \in B$. As no edge crosses the cut, then there is no edge connecting u and v, so the graph is disconnected.

Now we will do the $\Rightarrow$ direction. So, suppose the graph is disconnected and we will proceed to construct the desired cut. Pick any vertex $u \in V$ and define A to be the set of all vertices reachable from u. Define $B = V - A$. Then (A, B) is a cut with no crossing edges. Suppose, by contradiction, there was an edge $e = (v, w)$ crossing from A to B, with $v \in A$ and $w \in B$. However, by definition of A, since w is reachable, we have $w \in A$, a contradiction. $\lightning$ $\square$

Double Crossing Lemma. Suppose there exists a cycle C in G that has an edge crossing the cut (A, B), then some other edge of C must also cross the cut (A, B).

Lonely Cut Corollary. If edge e is the only edge crossing some cut (A, B), then it is not contained in any cycle.

Proofs. The fact that the double crossing lemma is true is quite clear from the definition of a cycle [i.e. if the cycle starts in A and leaves A, we must come back to A somehow for it to be a cycle]. The corollary is essentially the contrapositive of the lemma. $\square$

Proof Part 2. Prim's Algorithm Produces a Spanning Tree.

Let $G = (V, E)$ be a connected graph and $S \subseteq G$ be the output of Prim's algorithm. To show that Prim's algorithm produces a spanning tree [but not necessarily a minimal such tree for now], we need to show that (1) S spans G and (2) that S doesn't contain any cycles.

There are only two possibilities as to when the Prim's algorithm could halt. The first possibility is that we satisfied the condition $S = V$ and the while loop exits. If that is the case, then S spans G, as required. The other possibility is that the algorithm runs out of edges to scan and vertices to add to S yet $S \neq V$. Then we are at some cut $(S, V - S)$ with no crossing edges. By empty cut lemma this graph is disconnected, a contradiction. $\lightning$

Next, suppose we are at a point in the algorithm where we have a fixed cut $(S, V - S)$ of visited and unvisited vertices. The algorithm is looking to add the first edge e that would cross this cut. By the lonely cut corollary, this edge e would not be contained in any cycle. This is repeated at every step of the algorithm, each time with a *new* cut $(S, V - S)$. So at every point in the algorithm we consider a new cut and S won't contain any cycles.

So, we have shown that S spans G and doesn't contain any cycles, as required. $\square$

This page now presents the main part of the proof that Prim's algorithm produces a spanning tree that is indeed minimal/optimal. Optimality will be shown via an exchange argument. Once we prove the cut property, the result we are looking for is quite immediate.

Proof Part 3. Proof of Optimality.

Cut Property Theorem. Suppose we have a connected graph $G = (V, E)$ and a cut (A, B). Suppose that $e \in G$ is an edge that crosses the cut (A, B) and that e is the edge with the minimum length to do so. Then e must be included in the MST for G.

Proof. We will proceed by contradiction and will use an exchange argument to prove the optimality of greediness. Let e be the edge that is described in the theorem, that is the edge with the minimum length across the cut (A, B) and suppose that it runs between vertices $v \in A$ and $w \in B$.

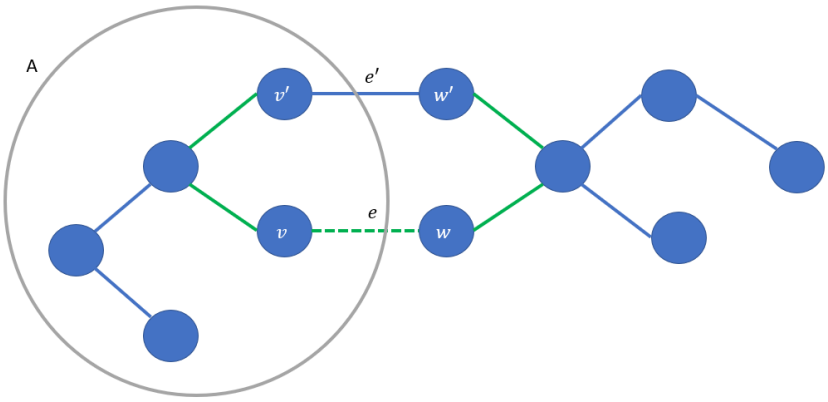
Now suppose that M is a MST of G and that M does *not* include e. Since M is a MST of a connected graph, there must be some path from v to w and there must be some other edge e' that crosses the cut (A, B), as otherwise we would contradict the empty cut lemma.

Suppose that e' runs between vertices $v' \in A$ and $w' \in B$ and by assumption we know that $l_e < l_{e'}$. We will proceed to substitute or *exchange* e for e' in M. If we do so, we get a new set of edges, the set $M' = M \cup \{e\} - \{e'\}$.

First, we observe that M' spans G. Indeed, any path in M that used the edge e' to span G could be re-routed via edge e and its adjoining edges (shown in green).

Second, we observe that M' does not contain any cycles. We recall that M was a MST, thus M did not contain any cycles. Only possibility to introduce a cycle is by adding an extra edge. We do add an extra edge e, and we do have a cycle in the set $M \cup \{e\}$; however, we then remove the edge e' and break precisely the cycle that was introduced.

Therefore, M' is a spanning tree, and one that has total edge length smaller than that of M, due to $l_e < l_{e'}$. Therefore, M cannot be a *minimal* spanning tree, a contradiction. ↯ □



Prim's Algorithm Produces the MST of a Graph.

Proof. We already know that Prim's algorithm produces a spanning tree. At every step of the Prim's algorithm we have a cut of the graph $(S, V - S)$. By the cut property, we know that the edge e crossing this cut that is of the minimal length must be included in the MST. This is precisely the edge Prim's algorithm adds at every step. □

4.5 Kruskal's Algorithm

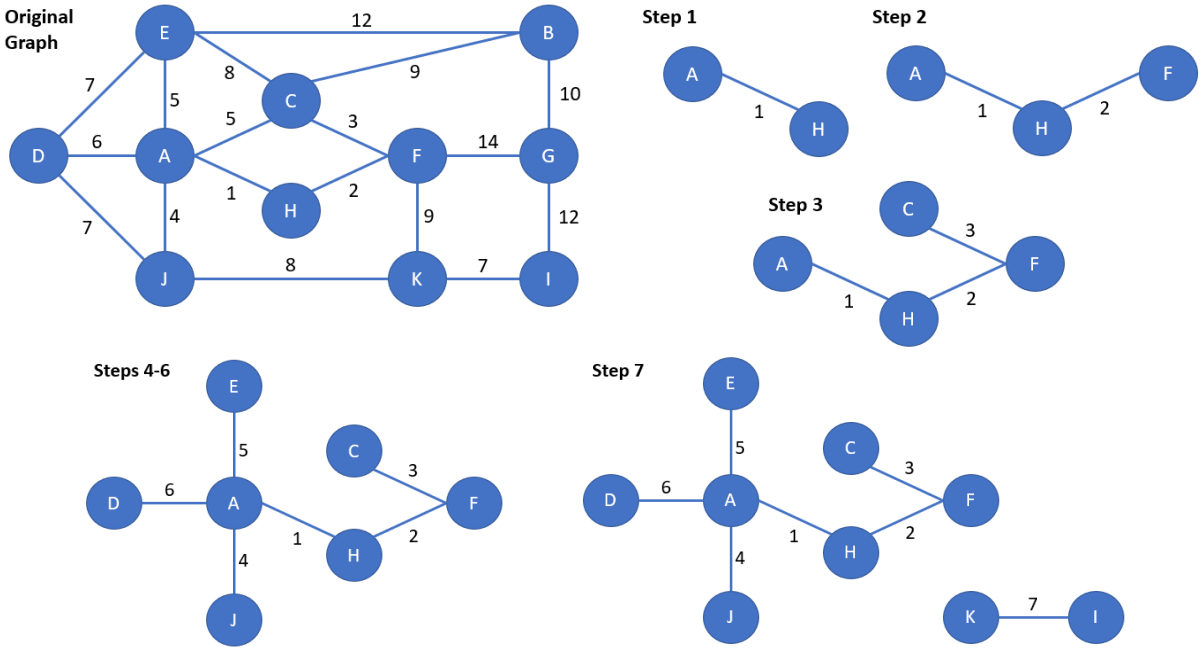
Kruskal's algorithm is also a greedy algorithm that finds an MST of a graph. However, Kruskal's approach is rather different from Prim's (and therefore is rather different from Dijkstra's shortest path approach as well).

Instead of starting from some source vertex s, we start from any empty set and look for *any* edge, *anywhere* that has the minimal weight. We add the edge to MST as long as doing so does not introduce any cycles. Via a disjoint set implementation, run time is $\mathcal{O}(m \log n)$.

```
Kruskal(G) {
    set S := empty set

    while (|S| != |V| - 1) {
        pick an edge e with minimum weight such that
            it is not in S and S U {e} does not contain cycles

        set S := S U {e}
    }
}
```

Example. This is the same example as the one in the previous section. Edges are added to the MST in the following order. The order is almost identical, just the order of edges KI and JK is reserved. Once we we have added the edge AD in step 6, the remaining edge with the smallest length is KI and not JK. So we add the edge IK in step 7.

1. AH	3. FC	5. AE	7. **KI**	9. CB
2. HF	4. AJ	6. AD	8. **JK**	10. BG



Proof That Kruskal's Algorithm Produces a MST.

Kruskal's algorithm produces a spanning tree.

Proof. This part is fairly straightforward and essentially follows from the design of the algorithm itself. Suppose that we have a graph $G = (V, E)$ and the output of Kruskal's algorithm is the set S.

First, we will clarify that Kruskal's algorithm does not introduce any cycles. This is precisely stated in the algorithm itself.

Second, we show that S spans G. The argument here is similar to that in the case of Prim's algorithm. There are two possibilities for the algorithm to halt: either $|S| = |V| - 1$ and the while loop exits, or we run out of edges to add. In the former case, as $|S| = |V| - 1$ so we have an acylic graph with n vertices and $n - 1$ edges, so all vertices in S will be connected by proposition 4 of section 3.1. So S spans G as required.

In the latter case there are two cases. Either there are no edges to add, or an addition of any new edge introduces a cycle. If there are no new edges to add, yet $|S| \neq |V| - 1$, the original graph G is disconnected ($m \geq n - 1$ in a simple connected graph). If an addition of *any* new edge introduces a cycle, yet $|S| \neq |V| - 1$, then once again we are not reaching some disconnected vertices. So in either case G is disconnected. $\lightning$ □

Kruskal's algorithm produces a MST.

Proof. The proof here is also not too difficult and just like with Prim's is an application of the cut property. Consider any edge $e = (v, w)$ that is to be added by Kruskal's algorithm and let S be the set of vertices connected to v. Then $w \notin S$, as if it was, w would been already connected to v and the addition of e would introduce a cycle, so the addition of e would not be considered by the algorithm. Therefore, just like in the empty cut lemma, we have have constructed a cut $(S, V - S)$.

Moreover, edge e is the first edge we are considering across the cut $(S, V - S)$; otherwise, once again e would been contained in some cycle. Since it is the first edge across this cut and as Kruskal's algorithm adds edges of minimal length first, no edge of smaller length across this cut been considered by this point. Therefore e must the the edge of the smallest length across the cut $(S, V - S)$ and by the cut property must be in the MST. □

Implementation of Kruskal's Algorithm.

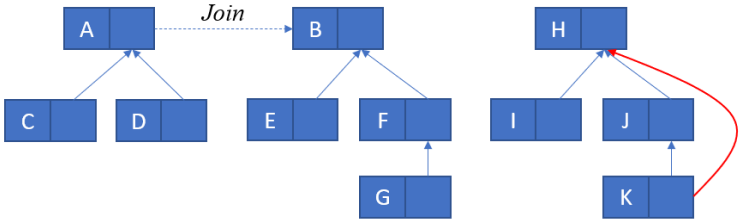
Throughout its run, Kruskal's algorithm is much preoccupied with checking if the potential edge to be added next would introduce any cycles. This is equivalent to checking if the edge's endpoints belong to different *connected components* of the graph. Determining sets of connected components on each iteration via BFS or DFS is of course very expensive, so we need to have some way to keep track of the connected components as we go.

The disjoint sets data structure is the data structure that works quite well. The disjoint sets will be the connected components. We initialize each vertex to be its own connected component or to be its own disjoint set and name that disjoint set after the vertex. The operation to do this will be called `MakeUnionFind` and runs in $\mathcal{O}(n)$ time.

Whenever an edge is added by Kruskal's, two connected components of the graph are joined together and their respective disjoint sets are also of course joined together. The operation here is called `Join` and runs in $\mathcal{O}(1)$. We just re-point the tip of the smaller connected component to point to and take the name of the *larger* connected component.

Whenever Kruskal's wants to check if an edge's endpoints belong to two different connected components, it checks if the names of the respective disjoint sets are different. This is done using the `Find` operation that runs in $\mathcal{O}(\log n)$ time. We just take the vertex and follow the pointers until we reach the name of the disjoint set. Since disjoint sets always took the name of the *larger* set upon join, then sets always *doubled* in size, so that is why the find operation runs in logarithmic time: we would need to at most travel through $\log n$ pointers to reach the tip.

Example: the name of the first component is A and the name of the second component is B. Once they are joined, A's name is changed to B as B was the larger component. If we wish to look up the name of the component that contains D, we follow exactly two pointers: `D-A` and `A-B` and determine that the name is B. A potential optimization is just to link all lower vertices to the tip directly, as is shown by the red arrow, this is done during the first call to `Find`. So, the initial `Find` operation would take a bit longer, but the subsequent calls to the same `Find` would be faster.



Over the run of Kruskal's there would be up to a total of $2m$ `Find` operations, $n-1$ `Union` operations. The total from `Find`s is $\mathcal{O}(m \log n)$. We need to sort edges initially by length which is done in $\mathcal{O}(m \log m)$ time. As $\mathcal{O}(\log m) = \mathcal{O}(\log n)$ the run time is $\mathcal{O}(m \log n)$.

ASIDE: PROOF STRATEGIES Recall that we prove greediness of algorithm via one of the following two methods:

1) Greedy Stays Ahead & Induction
2) Exchange Argument

Recall the proof strategies illustrated in each of the subsections of this chapter.

- ⋆ Section 4.1 (Interval Scheduling Problem): Greedy Stays Ahead & Induction
- ⋆ Section 4.2 (Weight Interval Scheduling Problem): Exchange Argument
- ⋆ Section 4.3 (Dijkstra's Algorithm): Greedy Stays Ahead & Induction
- ⋆ Section 4.4 (Prim's Algorithm): Exchange Argument (Cut Prop.)
- ⋆ Section 4.5 (Kruskal's Algorithm): Exchange Argument (Cut Prop.)

It should be noted that the approaches to the exchange argument slightly differ between sections 4.2 and sections 4.4, 4.5. In section 4.2 we swap *all* possible differences between a generic solution and the greedy solution. In sections 4.4, 4.5 we focus on just a single difference and then derive a *contradiction*. Either approach could be used in either section (in fact perhaps the approach of section 4.2 is a bit stronger argument).

5 Divide and Conquer

5.1 Master Theorem

THEOREM. Let $a \geq 1$ and $b > 1$ be constants, let $f(n) : \mathbb{N} \to \mathbb{R}^+$, and let

$$T(n) = a\, T\left(\tfrac{n}{b}\right) + f(n)$$

1. If $f(n) \in \mathcal{O}(n^{\log_b a - \epsilon})$ for some $\epsilon > 0$, then $T(n) \in \Theta(n^{\log_b a})$
2. If $f(n) \in \Theta(n^{\log_b a})$, then $T(n) \in \Theta(n^{\log_b a} \cdot \log n)$
3. If $f(n) \in \Omega(n^{\log_b a + \epsilon})$, for some $\epsilon > 0$, and if $af\left(\tfrac{n}{b}\right) \leq cf(n)$ for some $0 < c < 1$ and for all large enough n, then $T(n) \in \Theta(f(n))$

Essentially master theorem compares the run time of $f(n)$ with the run time of $n^{\log_b a}$ and determines which one is larger. If $n^{\log_b a} > f(n)$, then the work done by the leaves in the recurrence tree dominates and we get case 1. If $n^{\log_b a} < f(n)$, then the work done by the root of the recurrence tree dominates and we get case 3. However, in those cases, $f(n)$ must be *polynomially* larger or smaller than $n^{\log_b a}$ as dictated by the additional n^{ϵ} or $n^{-\epsilon}$ factor.

Case 3 comes with an extra condition called *the regularity condition*. It ensures that the total work done by children $[af\left(\tfrac{n}{b}\right)]$ is *always* less than the work done by the parent $[cf(n)]$.

Example 1. $T(n) = 27T\left(\tfrac{n}{3}\right) + n$.

- Here $a = 27, b = 3, f(n) = n$.
- We compute $n^{\log_b a} = n^{\log_3 27} = n^3$.
- We observe that $f(n) = n \in \mathcal{O}(n^{3-\epsilon})$ for $0 < \epsilon \leq 2$.
- By case 1 of master theorem $T(n) \in \Theta(n^3)$.

Example 2. $T(n) = T\left(\tfrac{2n}{3}\right) + 1$.

- Here $a = 1, b = \tfrac{3}{2}, f(n) = 1$.
- We compute $n^{\log_b a} = n^{\log_{1.5} 1} = n^0 = 1$.
- We observe that $f(n) = 1 \in \Theta(1)$.
- By case 2 of master theorem $T(n) \in \Theta(\log n)$.

Example 3. $T(n) = 3T\left(\tfrac{n}{4}\right) + n \log n$.

- Here $a = 3, b = 4, f(n) = n \log n$.
- We compute $n^{\log_b a} = n^{\log_4 3} = n^{0.793}$.
- We observe that $f(n) = n \log n \in \Omega(n^{\log_4 3 + \epsilon})$ as $n \in \Omega(n^{\log_4 3 + \epsilon})$ for $0 < \epsilon < 0.207$.
- Regularity condition holds as $3\left(\tfrac{n}{4} \log\left(\tfrac{n}{4}\right)\right) \leq c(n \log n)$ for $c = \tfrac{3}{4} < 1$ and for all n.
- By case 3 of master theorem $T(n) \in \Theta(n \log n)$.

Example 4. Master theorem doesn't apply in the following cases:

- $T(n) = 2^n T\left(\frac{n}{2}\right) + 1$ [$a = 2^n$ is not a constant]
- $T(n) = \frac{1}{2} T\left(\frac{n}{2}\right) + 1$ [$a = \frac{1}{2}$ is not greater or equal to 1]
- $T(n) = 2T(n) + 1$ [$b = 1$ is not greater than 1]
- $T(n) = T\left(\frac{n}{2}\right) - n \log n$ [$f(n) = -n \log n$ is not positive]
- $T(n) = T(n-1) + 1$ [recurrence is not in the right form]
- $T(n) = T\left(\frac{n}{2}\right) + T\left(\frac{n}{4}\right) + 1$ [recurrence is not in the right form]

Example 5. $T(n) = 2T\left(\frac{n}{2}\right) + n \log n$.

- Here $a = 2, b = 2, f(n) = n \log n$.
- We compute $n^{\log_b a} = n^{\log_2 2} = n$.
- Cases 1 and 2 do not apply as $f(n) = n \log n \notin \mathcal{O}(n^{1-\epsilon})$ and $f(n) = n \log n \notin \Theta(n)$.
- Case 3 doesn't apply either as $n \log n \notin \Omega(n^{1+\epsilon})$ for any $\epsilon > 0$ [if $\epsilon = 0$ was allowed then $n \log n \in \Omega(n)$, but we need $\epsilon > 0$].
- None of the cases of the theorem apply, so the theorem doesn't apply to this recurrence.

Example 6. $T(n) = T\left(\frac{n}{2}\right) + n(2 - \cos n)$.

Here $a = 1, b = 2, f(n) = n(2 - \cos n)$. We compute $n^{\log_b a} = n^{\log_2 1} = n^0 = 1$. As for $f(n) = n(2 - \cos n)$ we observe that $3 \geq 2 - \cos n \geq 1, \forall n$ so $f(n)$ behaves roughly as n. Cases 1 and 2 do not apply as $f(n) \notin \mathcal{O}(1)$ and $f(n) \notin \Theta(1)$.

We are left to consider case 3. It is the case that $f(n) = n(2 - \cos n) \in \Omega(n^{0+\epsilon})$ for any $0 < \epsilon \leq 1$. However, consider $n = 2\pi k$ for k odd, then:

- $f\left(\frac{n}{2}\right) = f\left(\frac{2\pi k}{2}\right) = f(\pi k) = \pi k(2 - \cos \pi k) = \pi k(2 + 1) = 3\pi k$
- $f(n) = f(2\pi k) = f(2\pi k) = 2\pi k(2 - \cos 2\pi k) = 2\pi k(2 - 1) = 2\pi k$

In particular, we see that $af\left(\frac{n}{b}\right) \not\leq cf(n)$ for any $c < 1$. Essentially $f(n)$ is not monotone and oscillates much, so we cannot *always* guarantee that the total work done by children is smaller than the work done by the parent. So the theorem doesn't apply to this recurrence.

Example 7. Suppose that an algorithm X has a recurrence relation $T(n) = 16T\left(\frac{n}{4}\right) + n^3$ and an algorithm Y has a recurrence relation $T'(n) = aT'(\frac{n}{8}) + n^3$. What must be the value of a for Y to be slower than X?

In X we have $a = 16, b = 4, f(n) = n^3$. We determine that $n^{\log_4 16} = n^2$ and observe that $f(n) = n^3 \in \Omega(n^{2+\epsilon})$ for $0 < \epsilon \leq 1$. So by case 3 of master theorem $T(n) \in \Theta(n^3)$.

We would like Y to be slower than X. So we would like to have $T'(n) > cn^3$. We are in case 3, where the outcome of the algorithm is dominated by $f(n)$. Since we cannot change $f(n)$, we must force $n^{\log_b a}$ to be large enough to dominate $f(n)$ and lead us to case 1 or 2. This is equivalent to having $n^{\log_8 a} \geq n^3$ or $\log_8 a \geq 3$ which means that $a \geq 8^3$ or $a \geq 512$.

In particular, for $a = 512$ we end up in case 2 and get the run time of $\Theta(n^3 \log n)$, for $a > 512$ we end up in case 1 and get the run time of $\Theta(n^{3+c})$, $c > 0$.

Here is a great resource with more examples on master theorem, in fact, example 6 is from this site: `https://www.csd.uwo.ca/~mmorenom/CS424/Ressources/master.pdf`. A problem quite similar to example 7 appeared on both, 2018S2 midterm and final exams.

5.2 Recurrences: Substitution Method

Much of this is perhaps review from CPSC 221 but is summarized here for reference. Examples 2, 4 and 5 below could also be solved much faster via master theorem. In fact, master theorem is a great way to check ones work in those cases. Master theorem doesn't apply to examples 1, 3 and 6. Assume that in all of the following examples $T(1) = b$ unless specified otherwise.

> The Steps
>
> 1. Recursively substitute $T(\dots)$ expressions into each other to get a general form of the $T(n)$ expression for the k-th step.
> 2. Set the value of k so that the resulting general form of $T(n)$ is expressed in terms of $T(1)$ and thus eliminate all k.

Example 1. $T(n) = T(n-1) + c$.
This is a simple linear recurrence example, the run time is $\mathcal{O}(n)$:

$$\begin{aligned}
T(n) &= T(n-1) + c \\
&= [T(n-2) + c] + c \\
&= [[T(n-3) + c] + c] + c \\
&\dots \\
&= T(n-k) + kc && [k\text{-th step}] \\
&= T(1) + (n-1)c && [n - k = 1 \text{ or } k = n-1] \\
&= b + (n-1)c
\end{aligned}$$

Example 2. $T(n) = 2T\left(\frac{n}{2}\right) + cn$.
This is a simple logarithmic recurrence, the run time is $\mathcal{O}(n \log n)$:

$$\begin{aligned}
T(n) &= 2T\left(\frac{n}{2}\right) + cn \\
&= 2\left[2T\left(\frac{n}{4}\right) + \frac{cn}{2}\right] + cn \\
&= 2\left[2\left[2T\left(\frac{n}{8}\right) + \frac{cn}{4}\right] + \frac{cn}{2}\right] + cn \\
&= 2^3 \cdot T\left(\frac{n}{8}\right) + \frac{2^2 \cdot cn}{4} + \frac{2 \cdot cn}{2} + cn \\
&\dots \\
&= 2^k \cdot T\left(\frac{n}{2^k}\right) + kcn && [k\text{-th step}] \\
&= 2^{\log_2 n} \cdot T(1) + \log_2 n \cdot cn && \left[\tfrac{n}{2^k} = 1 \text{ or } k = \log_2 n\right] \\
&= bn + cn \log_2 n
\end{aligned}$$

Example 3. $T(n) = T(n-1) + cn$.
This linear recurrence requires finding the sum of first $n-2$ integers, the run time is $\mathcal{O}(n^2)$:

$$\begin{aligned}
T(n) &= T(n-1) + cn \\
&= [T(n-1) + c(n-1)] + cn \\
&= [[T(n-3) + c(n-2)] + c(n-1)] + cn \\
&= [[[T(n-4) + c(n-3)] + c(n-2)] + c(n-1)] + cn \\
&= T(n-4) + cn + cn + cn + cn - (3+2+1)c \\
&\cdots \\
&= T(n-k) + kcn - c\sum_{i=0}^{k-1} i && [k\text{-th step}] \\
&= T(1) + (n-1)cn - c\sum_{i=0}^{n-2} i && [n-k=1 \text{ or } k=n-1] \\
&= T(1) + (n-1)cn - c\frac{(n-2)(n-1)}{2} && [\text{sum of first } n-2 \text{ integers}] \\
&= b - c + \frac{c}{2}n + \frac{c}{2}n^2
\end{aligned}$$

Example 4. $T(n) = T\left(\frac{n}{2}\right) + cn$.
This logarithmic recurrence requires summing up geometric series, the run time is $\mathcal{O}(n)$:

$$\begin{aligned}
T(n) &= T\left(\frac{n}{2}\right) + cn \\
&= \left[T\left(\frac{n}{4}\right) + \frac{cn}{2}\right] + cn \\
&= \left[\left[T\left(\frac{n}{8}\right) + \frac{cn}{4}\right] + \frac{cn}{2}\right] + cn \\
&\cdots \\
&= T\left(\frac{n}{2^k}\right) + \sum_{i=0}^{k-1} \frac{cn}{2^i} && [k\text{-th step}] \\
&= T(1) + \sum_{i=0}^{\log_2 n - 1} \frac{cn}{2^i} && \left[\tfrac{n}{2^k} = 1 \text{ or } k = \log_2 n\right] \\
&= b + cn\left(\frac{1-(1/2)^{\log_2 n}}{1-(1/2)}\right) && [\text{sum of geometric series}] \\
&= b + 2cn\left(1-(1/2)^{\log_2 n}\right) \\
&= b + 2cn\left(1-2^{-\log_2 n}\right) \\
&= b + 2cn\left(1-n^{-1}\right) \\
&= b - 2c + 2cn
\end{aligned}$$

Example 5. $T(n) = 8T\left(\frac{n}{2}\right) + cn^2$. This is the previous example with a factor of 8 at front of the $T\left(\frac{n}{2}\right)$ term and a quadratic time of cn^2 involved in processing of each step. The resulting run time is $\mathcal{O}(n^3)$:

$$\begin{aligned}
T(n) &= 8\left[T\left(\frac{n}{2}\right)\right] + cn^2 \\
&= 8\left[8\left[T\left(\frac{n}{4}\right)\right] + c\frac{n^2}{4}\right] + cn^2 \\
&= 8\left[8\left[8T\left(\frac{n}{8}\right) + c\frac{n^2}{16}\right] + c\frac{n^2}{4}\right] + cn^2 \\
&= 8 \cdot 8 \cdot 8 \cdot T\left(\frac{n}{8}\right) + c\frac{8 \cdot 8 \cdot n^2}{16} + c\frac{8 \cdot n^2}{4} + cn^2 \\
&= 8^3 \cdot T\left(\frac{n}{2^3}\right) + cn^2\left(4 + 2 + 1\right) \\
&\cdots \\
&= 8^k \cdot T\left(\frac{n}{2^k}\right) + cn^2 \sum_{i=0}^{k-1} 2^i && [k\text{-th step}] \\
&= 8^{\log_2 n} \cdot T(1) + cn^2 \sum_{i=0}^{\log_2 n - 1} 2^i && \left[\tfrac{n}{2^k} = 1 \text{ or } k = \log_2 n\right] \\
&= 8^{\log_2 n} \cdot b + cn^2\left(\frac{1 - (1/2)^{\log_2 n}}{1 - (1/2)}\right) && [\text{sum of geometric series}] \\
&= bn^3 + 2cn^2(1 - n^{-1}) = -2cn + 2cn^2 + bn^3 && [\text{as } 8^{\log_2 n} = 2^{3\log_2 n} = n^3]
\end{aligned}$$

Example 6. $T(n) = T\left(\sqrt{n}\right) + 1$, assuming $\exists k \in \mathbb{Z}$ such that $2^{2^k} = n$ and $T(2) = b$. The run time is $\mathcal{O}(\log\log(n))$:

$$\begin{aligned}
T(\sqrt{n}) &= \left[T\left(\sqrt{\sqrt{n}}\right) + 1\right] + 1 \\
&= \left[T\left(\left(n^{1/2}\right)^{1/2}\right) + 1\right] + 1 \\
&= \left[T(n^{1/4}) + 1\right] + 1 \\
&= \left[\left[T\left(\sqrt{n^{1/4}}\right) + 1\right] + 1\right] + 1 \\
&= \left[\left[T\left(\left(n^{1/4}\right)^{1/2}\right) + 1\right] + 1\right] + 1 \\
&= T(n^{1/8}) + 1 + 1 + 1 \\
&\cdots \\
&= T\left(n^{1/2^k}\right) + k && [k\text{-th step}] \\
&= T(2) + \log_2(\log_2 n) && [n^{1/2^k} = 2 \text{ or } 2^{2^k} = n \text{ or } k = \log_2(\log_2 n)] \\
&= b + \log_2(\log_2 n)
\end{aligned}$$

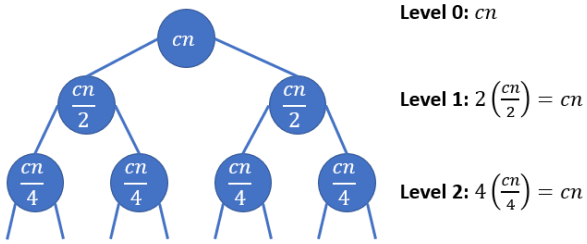
5.3 Recurrences: Tree Method

THE STEPS

1. Draw a tree that shows the amount of work done at each recursive call of the algorithm. Root starts off with $f(n)$ amount of work and, at each level, that amount divides up between a children as per the b value of the recurrence relation.
2. Determine the total amount of work done per level, as a function of that level's no.
3. Determine the number of levels needed to reach the base case.
4. Sum the expression obtained in 2 over the number of levels determined in 3.

Example 1. $T(n) = 2T\left(\frac{n}{2}\right) + cn$.

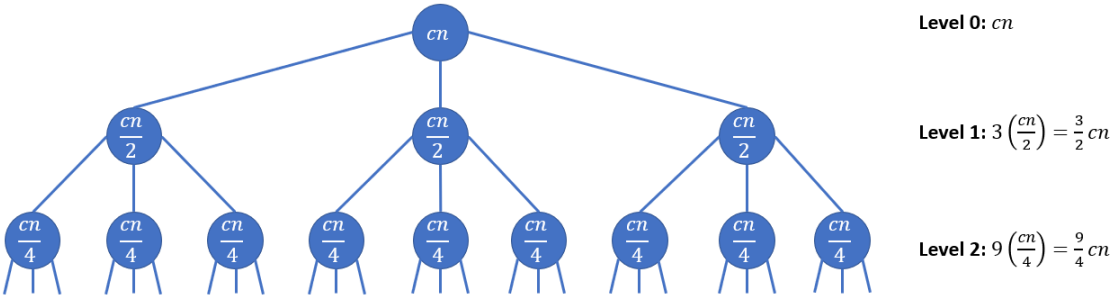
- As per diagram, there is cn amount of work involved per *any* level.
- Each child is half the size of its parent, so we need $\log_2 n$ levels to reach the base case.
- Total amount of work: $\sum_{i=0}^{\log_2 n - 1} cn = \log_2 n \cdot cn \in \mathcal{O}(n \log n)$.



Example 2. $T(n) = 3T\left(\frac{n}{2}\right) + cn$.

- As per diagram, there is $\left(\frac{3}{2}\right)^i cn$ amount of work involved at some level i.
- Each child is half the size of its parent, so we need $\log_2 n$ levels to reach the base case.
- Total amount of work [on line 3 we use $a^{\log_c b} = b^{\log_c a}$ for any $a > 0, b > 0$]:

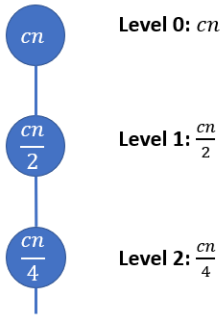
$$\begin{aligned}
\sum_{i=0}^{\log_2 n - 1} \left(\frac{3}{2}\right)^i cn &= cn \sum_{i=0}^{\log_2 n - 1} \left(\frac{3}{2}\right)^i = cn \left(\frac{1 - (3/2)^{\log_2 n}}{1 - (3/2)}\right) \\
&= 2cn \left((3/2)^{\log_2 n} - 1\right) \\
&= 2cn \left(n^{\log_2 (3/2)} - 1\right) \\
&= 2cn \left(n^{0.59} - 1\right) \\
&= 2cn^{1.59} - 2cn \ \in \ \mathcal{O}(n^{1.59})
\end{aligned}$$



Example 3. $T(n) = T\left(\frac{n}{2}\right) + cn$.

- As per diagram, there is $\left(\frac{1}{2}\right)^i cn$ amount of work involved at some level i.
- Each child is half the size of its parent, so we need $\log_2 n$ levels to reach the base case.
- Total amount of work [using the log property from ex. 2]:

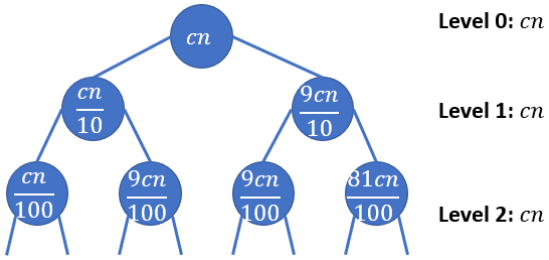
$$\begin{aligned}\sum_{i=0}^{\log_2 n-1}\left(\frac{1}{2}\right)^i cn &= cn\sum_{i=0}^{\log_2 n-1}\left(\frac{1}{2}\right)^i = cn\left(\frac{1-(1/2)^{\log_2 n}}{1-(1/2)}\right)\\ &= 2cn\left(1-(1/2)^{\log_2 n}\right)\\ &= 2cn\left(1-n^{\log_2(1/2)}\right)\\ &= 2cn\left(1-n^{-1}\right)\\ &= 2cn-2c \;\in\; \mathcal{O}(n)\end{aligned}$$



Example 4. $T(n) = T\left(\frac{9n}{10}\right) + T\left(\frac{n}{10}\right) + cn$.

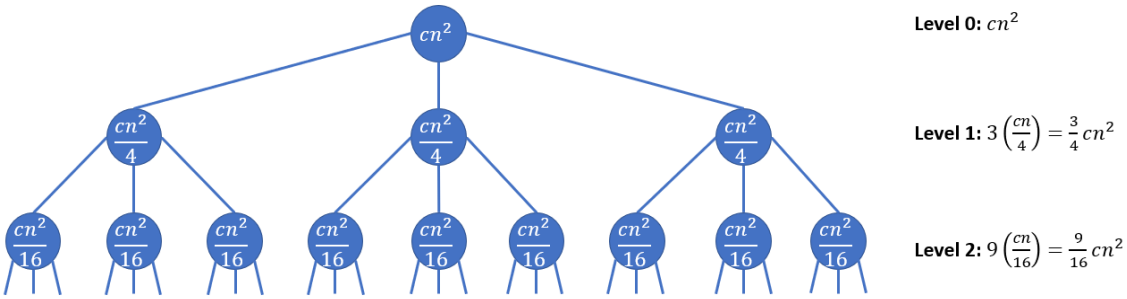
This recurrence relation is an example of the phenomena of uneven splits in quick sort. In particular, this recurrence relation occurs if the quick sort pivots occur in such a way that the list being sorted gets split consistently in a 90-10% fashion. We will see that despite such an uneven, bad split, we still end up with a $\mathcal{O}(n \log n)$ run time.

- As per diagram, there is cn amount of work involved per *any* level.
- The tree is uneven, so to determine the total number of levels required to reach the base case, we must consider the longest path to get there. In this case that is the path on the furthest right, each child does $\frac{9}{10}$ of the parent's work, so we need $\log_{10} n$ levels.
- Total amount of work: $\sum_{i=0}^{\log_{10} n-1} cn = \log_{10} n \cdot cn \in \mathcal{O}(n \log n)$.



Example 5. $T(n) = 3T\left(\frac{n}{2}\right) + cn^2$: a slight modification of example 2 as now $f(n) = cn^2$.

$$\sum_{i=0}^{\log_2 n-1}\left(\frac{3}{4}\right)^i cn^2 = cn^2\left(\frac{1-(3/4)^{\log_2 n}}{1-(3/4)}\right) = 4cn^2\left(1-n^{\log_2(3/4)}\right) = 4cn^2 - 4cn^{1.59} \in \mathcal{O}(n^2)$$



5.4 Divide And Conquer Overview

The Key Stages

1. ***Divide.*** We recursively divide the main problem into two or more smaller pieces that usually are of equal size. Division is done based on some criteria and that criteria could depend on step 3.
2. ***Base Cases.*** Once we have divided the main problem into small enough pieces, we carry out the computations to solve each of those subproblems. Those computations are usually just brute force.
3. ***Combine.*** We recursively combine the results of the two adjacent subproblems. In doing so, we compare their results and use some criteria to build up the overall result. Generally, there are three cases to consider [with iii being the most typical one]:
   - **i.** the result from the first subproblem is the one we want
   - **ii.** the result from the second subproblem is the one we want
   - **iii.** the result that could be formed using elements from *both* subproblems is the one we want

   The work in this step should usually be done in at most *linear* time and often this step is the most challenging one to design.

Divide and conquer algorithms reduce the number of computations that need to be done, relative to the brute force way of solving the same problem. By dividing up the work, we perform computations on small groups of items and then use the result to represent that small group when putting things together. Divide and conquer algorithms usually work on problems for which the brute force approaches already run in polynomial time and the divide and conquer approach is applied to slightly improve that polynomial run time.

5.5 Review of Merge Sort

```
MergeLists(A, B) { // STAGE 3 [Kleinberg pg. 49]
    maintain a CURRENT pointer into each list, initialized to point to the front elements

    while (both lists are not empty) {
        let ai and bj be the element pointed to by the CURRENT pointer
        append the smaller of ai and bj to the output list
        advance the CURRENT pointer in the list from which the element was selected from
    }

    append the remainder of the non-empty list to the output
    return the merged list                                              }
```

```
MergeSort(L) {
    if (the list is a single-element list) { it is already sorted }  // STAGE 2

    divide the list into two halves                                  // STAGE 1
        A containing the first ceil(n/2) elements
        B containing the remaining floor(n/2) elements
    A = MergeSort(A)
    B = MergeSort(B)
    L = MergeLists(A, B)                                             // STAGE 3

    return L                                                         }
```

In the above algorithm we saw the illustration of the three key stages:

1. ***Divide.*** In this case we have two very simple recursive calls that just split the given list into two halves.
2. ***Base Cases.*** A list that is one element long is already sorted.
3. ***Combine.*** This is the job of the `MergeLists` function. The main goal here is to merge lists efficiently: we take advantage of the fact that lists coming in are sorted and we need to just interleave the lists. Each element in the n long list is compared to an another element at most once at each merge time, so we must perform at most n comparisons per merge and we get a *linear* run time for each merge.

RUN TIME. The recurrence relation is $T(n) = 2T\left(\frac{n}{2}\right) + cn$: we split the original problem into two halves $[a = 2]$, each problem is half the size of the original $[b = 2]$ and linear time is involved in the merge step $[f(n) = cn]$. By master theorem the run time is $\Theta(n \log n)$.

5.6 Counting Inversions

THE PROBLEM. Suppose we have a list of n distinct numbers $a_1, \ldots, a_n$. We say that we have an *inversion* if there are two indices such that $i < j$ but $a_i > a_j$. Essentially an inversion occurs when any two elements in the list are out of order. We are interested in counting the total number of inversions in the given list.

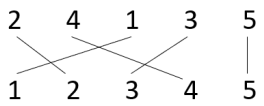
For example, consider the list $\{2, 4, 1, 3, 5\}$. We count the number of inversions by considering which elements need to come *before* each element in this list:

- 1 comes before $3, 5$ as it should
- 2 comes before $4, 3, 5$ as it should; but it should NOT come before 1 as it does
- 3 comes before 5 as it should
- 4 comes before 5 as it should; but it should NOT come before $1, 3$ as it does
- 5 doesn't come before any other element as it should

So we have a total of three inversions in this case:

- 2 and 1
- 4 and 1
- 4 and 3

Another way to notice inversions is by drawing the following diagram of (1) the given list and of (2) the fully sorted list and connect all identical elements [Kleinberg, pg. 222]. Anytime we have two lines that intersect we have an inversion.



In particular, the first intersection corresponds to the inversion of 1 and 2, since the lines connecting those elements to each other are the ones that intersect. The next intersection corresponds to the inversion of 4 and 1, and the final intersection corresponds to the inversion of 3 and 4.

If we were to unscramble those three inversions, we would obtain a perfectly sorted list. By counting the number of inversions we essentially introduce a way of measuring how scrambled the two lists are relative to each other. The two lists of numbers could represent, for example, rankings assigned to a set of movies by two individuals. By counting the total number of inversions between the two lists, we could get a measure of by how much do the movie tastes differ between those two individuals.

THE ALGORITHM. The algorithm for counting inversions is just the merge sort algorithm with a few modifications and a few extra lines of code. Those new lines are marked by *** in the following pseudo-code.

As the merge sort algorithm is already designed to sort a list, all we need to do is to also keep track of the *number of inversions we needed to unscramble* to complete the sort. We observe that whenever we are merging two lists and need to move an element b_j from the second list past k elements in the first list, we unscramble k inversions. Therefore, we include an if statement that makes use of this observation to do the count.

```
MergeAndCount(A, B) { // STAGE 3 [Kleinberg, pg. 224]
    maintain a CURRENT pointer into each list, initialized to point to the front elements
    maintain a variable COUNT of the number of inversions, initialized to 0     // ***

    while (both lists are not empty) {
        let ai and bj be the element pointed to by the CURRENT pointer
        append the smaller of ai and bj to the output list

        if (bj is the smaller element) {                                        // ***
            increment COUNT by the number of elements remaining in A            // ***
        }                                                                       // ***

        advance the CURRENT pointer in the list from which the element was selected from
    }

    append the remainder of the non-empty list to the output
    return COUNT and the merged list
}
```

```
MergeSort(L) {
    if (the list is a single-element list) { there are no inversions } // STAGE 2

    divide the list into two halves                                    // STAGE 1
        A containing the first ceil(n/2) elements
        B containing the remaining floor(n/2) elements
    (cA, A) = MergeSort(A)
    (cB, B) = MergeSort(B)          so like c is the sorted list and we're comparing it with the initial list
    (c, L) = MergeAndCount(A, B)                                       // STAGE 3

    return c = cA + cB + c, and sorted list L
}
```

RUN TIME. As before, the recurrence relation is $T(n) = 2T\left(\frac{n}{2}\right) + cn$ and by master theorem the run time is $\Theta(n \log n)$.

5.7 Closest Pair

THE PROBLEM. Given n points in the plane, find the pair that is closest together.

Brute force approach finds the solution in $\mathcal{O}(n^2)$: we just compute the distance between each possible pair of points and then find the minimum distance.

```
double closestPairBF(Point P, int n) {
    float min = Double.MAX_VALUE;
    for (int i = 0; i < n; i++) {
        for (int j = i + 1; j < n; j++) {
            if (distance(P[i], P[j]) < min)
                min = distance(P[i], P[j]);
        }
    }
    return min;
}

double distance(Point p1, Point p2) {
    return sqrt((p1.x - p2.x)*(p1.x - p2.x) + (p1.y - p2.y)*(p1.y - p2.y));
}
```

However, it seems, that we do not need to necessarily compute the distance between ***all*** of the points. For example, we would for sure know that the point located in the very lower-right corner of the plane and the point located in the very upper-left corner of the plane would not be the ones closest together (unless n is very small). Thus, just like the other divide and conquer algorithms, this one would look at small groups of points to avoid carrying out all possible comparisons. For simplicity, we will assume that all x and all y coordinates of all points are distinct.

Algorithm's Stages

(1) ***Divide.*** Recursively divide points into left and right groups by the proximity of their x coordinates. Do so until each group has ≤ 3 points.

(2) ***Base Cases.*** Determine the minimum distance between a pair of points for each group using brute force [this would involve at most 3 calculations].

(3) ***Combine.*** Combine the results for each pair of adjacent groups. To do so, we must determine which of the following is the *smallest*:

- **i.** computed minimum distance of the left group
- **ii.** computed minimum distance of the right group
- **iii.** some distance where one point is in the left group, and the other point in the right group

By recursively dividing points into groups based on the proximity of their x coordinates, we ensure that we compute distances only between points that are at least somewhat close together. The hardest part of the algorithm is the combination step, where we are once again vulnerable to having to compute all possible distances between all possible points from the two adjacent groups. Just like with merge sort, we must rely on some property of the adjacent groups to do this efficiently. Here comes a key point of the algorithm: in lemma 2 below we will show that we must check at most 15 distances at this stage.

Divide and Base Case Stages. In order to divide the points into left and right groups based on the proximity of their x coordinates, we must sort the given list of points by the increasing x coordinate. Call the resulting list P_x. We will also produce a list P_y of the same points sorted by the increasing y coordinate. This list will turn out to be useful later on, in the combination step. All sorting can be done in $\mathcal{O}(n \log n)$.

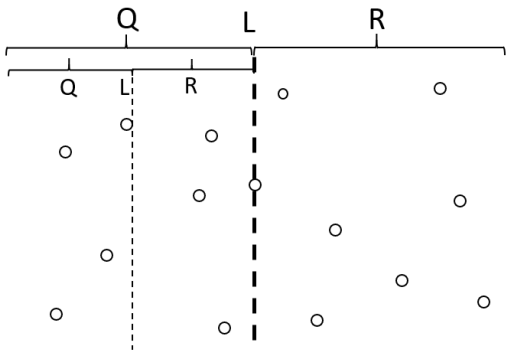
The recursive calls will take the sorted lists P_x, P_y as arguments. Each recursive call will split each list P into two halves, the lists Q and R, which will be the new arguments to the following recursive calls. Let x_m be the median of all x coordinates of P. Array Q will contain all points whose x coordinate is $\leq x_m$ and R will contain all points whose x coordinate is $> x_m$. Also let L be the line $x = x^*$ that marks the boundary between R and Q, where x^* is the rightmost point of Q [which may or may not be the same as x_m]. Refer to the diagram at the bottom of this page.

Then we make the recursive call on each of the halves with Q_x, Q_y and R_x, R_y as arguments. In order to do so, we must of course produce lists Q_x, Q_y, R_x, R_y. This can be done in $\mathcal{O}(n)$ run time: the original lists are already sorted, so it is easy to find the median x coordinate x_m and then split the original lists depending on if the x coordinate of a particular point is less than or equal to or greater than the median.

The recursive calls will return the closest pairs of points in each of the halves. Let q_0^*, q_1^* denote the closest pair of points for the particular Q half, and let r_0^*, r_1^* denote the closest pair of points to the particular R half.

Finally we have the base case which returns minimum distance using brute force if the number of points remaining is ≤ 3.

```
ClosestPair(P) { // A wrapper to initialize the recursive calls [Kleinberg, pg. 230]
    Construct Px and Py
    (p0*, p1*) = ClosestPairRec(Px, Py)
}

ClosestPairRec(Px, Py) {
    if (|P| <= 3) { find closest pair by brute force }    // STAGE 2

    Consruct Qx, Qy, Rx, Ry                              // STAGE 1
    (q0*, q1*) = ClosestPairRec(Qx, Qy)
    (r0*, r1*) = ClosestPairRec(Rx, Ry)
    ...
```



COMBINATION STAGE. Recall that in the combination stage we must pick the *smallest* of the following:

i. computed minimum distance of the left group

ii. computed minimum distance of the right group

iii. some distance where one point is in the left group, and another point in the right group

Moreover, we must ensure that (iii) is fast and does not involve computation of distances between all possible pairs of points from the left and right halves.

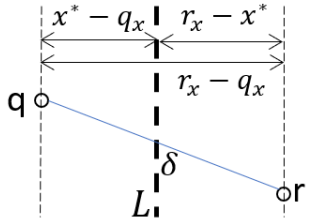
First lets rephrase this question. Define $\delta = \min\{d(q_0^*, q_1^*), d(r_0^*, r_1^*)\}$ and recall that q_0^*, q_1^* are the closest points in the left half and that r_0^*, r_1^* are the closest points in the right half. We would like to know if exist $q \in Q, r \in R$ such that $d(q, r) < \delta$. In other words, we would like to know if there are points q and r such that (iii) beats (i) and (ii). The following two lemmas will show that we do not need to look very far to answer this question.

Lemma 1. If there exists $q \in Q$ and $r \in R$ for which $d(q, r) < \delta$, then each of q and r lies within a distance δ of line L.

Proof. Let $q = (q_x, q_y)$ and $r = (r_x, r_y)$ and recall that x^* is the point with the largest x coordinate in Q that also defines the line L. Being within δ of L means that $x^* - q_x < \delta$ and $r_x - x^* < \delta$.

The argument really follows from the fact that hypotenuse is the longest side in a triangle. From the statement of the lemma we know that $d(q, r) < \delta$ and then looking at the following diagram we can come up with the required inequalities:

$$x^* - q_x \leq r_x - q_x \leq d(q, r) < \delta$$
$$r_x - x^* \leq r_x - q_x \leq d(q, r) < \delta$$

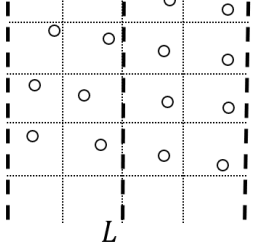


□

We will now let S denote the set of points within δ of L. This will be the set to which will restrict our search to from here on. Following prior notation, let S_y be the list of points in S sorted by the increasing y coordinate. We construct the set S_y by filtering sets Q and R for points that lie within δ of L and then by getting the positions of those points relative to each other via a single pass through P_y. This way we don't have to do any additional sorting here (see "remarks" below as to why this is important). All this is done in $\mathcal{O}(n)$ time.

Lemma 2. If $d(s, s') < \delta$ for $s, s' \in S$, then s and s' are within 15 positions of each other in the sorted list S_y.

Proof. Suppose that set S_y is split into boxes where the length of each box's side is $\delta/2$. We first observe that each box contains *at most one point*. To see this, proceed by contradiction and suppose that some box contained two points. Observe that points in any given box are contained entirely within either Q or R and the *minimum* distance between any pair of points in those lists has to be δ.



Could the distance between two points in the box beat this minimum? Well, the *maximum* distance between two points in a box is its hypotenuse which is:

$$\sqrt{\left(\frac{\delta}{2}\right)^2 + \left(\frac{\delta}{2}\right)^2} = \sqrt{2\left(\frac{\delta}{2}\right)^2} = \frac{\sqrt{2}\delta}{2}$$

Then $\frac{\sqrt{2}}{2}\delta < \delta$, so the maximum distance between two points is smaller than the minimum distance and therefore a box cannot contain two points. $\lightning$

Continue proceeding by contradiction. Now suppose that there are at least 16 positions between s and s'. WLOG suppose that s has the smaller y coordinate. As there is at most one point per box and points are sorted by the increasing y coordinate in S_y, as per the above diagram, we would need at least three rows of boxes to separate s and s' by as much as 16 points. However, this would mean that the distance between s and s' is $3(\text{side length}) = 3\,(\delta/2) > \delta$, a contraction to lemma 1. $\lightning$ $\square$

Therefore, we see that we need to check at most 15 distances during the combination stage. All other work involved in the combination stage is $\mathcal{O}(n)$. Pseudo-code is continued below.

```
    ...
    delta = min(d(q0*, q1*), d(r0*, r1*))              // STAGE 3
    x* = maximum x-coordinate of a point in set Q
    L = {(x,y): x = x*}
    S = points in P within distance delta of L

    construct Sy
    for (each point s in Sy) { // O(15*n) ~ O(n)
        compute distance from s to each of the next 15 points in Sy
        let s, s* pair achieving minimum of those distances
    }

    if (d(s,s*) < delta)                        { return (s,s*) }
    else if (d(q0*,q1*) < d(r0*, r1*))          { return (q0*, q1*) }
    else                                        { return (r0*, r1*) }
}
```

Remarks.

1. Was it necessary to sort the points by their y coordinates as well? Would the proof of lemma 2 still be valid if we worked with points sorted by their x coordinates [i.e. would the proof of lemma 2 be valid if we worked with a set like S_x]? The answer is no. The argument of lemma 2 relies on S being four boxes wide across x, yet S is unbounded in the y direction, so the same argument can't be made if we were to just use the ordering of points by their x coordinates.
2. Why were the points sorted by their y coordinates at the very start and then lists S_y were constructed from lists Q, R and P_y in a rather awkward way? This was done to avoid sorting during the combination stage. Had we sorted anything during the combination stage, the run time of that stage would been $\mathcal{O}(n \log n)$ instead of $\mathcal{O}(n)$ and the overall run time of the algorithm would been $\mathcal{O}(n^2 \log n)$ by master theorem, which is worse than brute force.

3. By doing a more thorough analysis, the number of points to check during the combination stage could actually be reduced to just 6. So the run time of the combination stage is $6n \sim \mathcal{O}(n)$ to be precise.
4. In this case, the criteria that was used to combine adjacent groups in stage 3 driven the decision making on how to divide up the initial data in stage 1.

RUN TIME. As before, the recurrence relation is $T(n) = 2T\left(\frac{n}{2}\right) + cn$ and the run time is $\Theta(n \log n)$.

5.8 Integer Multiplication

Consider n digit numbers. Recall the grid method of multiplication, where to multiply two n digit numbers we must compute n partial products. In the first example the partial products are:

- $453 \times 2 = 906$
- $453 \times 1 = 453$
- $453 \times 4 = 1812$

$$\begin{array}{r} 4\,5\,3 \\ \times\ 4\,1\,2 \\ \hline 9\,0\,6 \\ 4\,5\,3 \\ 1\,8\,1\,2 \\ \hline 1\,8\,6\,6\,3\,6 \end{array} \qquad \begin{array}{r} 1\,1\,0\,0 \\ \times\ 1\,0\,0\,1 \\ \hline 1\,1\,0\,0 \\ 0\,0\,0\,0 \\ 0\,0\,0\,0 \\ 1\,1\,0\,0 \\ \hline 1\,1\,0\,1\,1\,0\,0 \end{array}$$

As is shown in the second example on the left, grid multiplication also works for binary numbers and also requires computation of n partial products. If we consider multiplication of two single digit numbers to be an elementary operation, then the run time for this algorithm is $\mathcal{O}(n^2)$ as we would need to do n multiplications for each of the n partial products.

Suppose that instead we tried a divide and conquer approach where we would multiply numbers that have half $(\frac{n}{2})$ the amount of digits. In binary we would write $x = x_1 \cdot 2^{n/2} + x_0$ where x_1 are the higher order bits and x_0 are the lower order bits. For example, if $x = 1001$ then we have $10 \cdot 2^2 + 01$ or `x1 = 10; x0 = 01; x = x1 << 2 + x0`. The product is:

$$\begin{aligned} xy &= \left(x_1 \cdot 2^{n/2} + x_0\right)\left(y_1 \cdot 2^{n/2} + y_0\right) \\ &= x_1y_1 \cdot 2^n + (x_1y_0 + x_0y_1) \cdot 2^{n/2} + x_0y_0 \quad \boxed{\star} \end{aligned}$$

Here we need to compute four products of numbers which are $\frac{n}{2}$ long and combine them in linear time, so the recurrence relation is $T(n) = 4T\left(\frac{n}{2}\right) + cn$. By master theorem the run time is $\Theta(n^2)$ which gives no improvement over the original problem. We must reduce the number of products in the subproblem to be less than four!

Consider the following algebraic manipulation that allows us to express the two products of the middle term of $\boxed{\star}$ in terms of the product $(x_1 + x_0)(y_1 + y_0)$ and two other known products:

$$(x_1 + x_0)(y_1 + y_0) = x_1y_1 + x_1y_0 + x_0y_1 + x_0y_0$$

$$x_1y_0 + x_0y_1 = (x_1 + x_0)(y_1 + y_0) - x_1y_1 - x_0y_0$$

Therefore $\boxed{\star}$ is:

$$xy = x_1y_1 \cdot 2^n + ((x_1 + x_0)(y_1 + y_0) - x_1y_1 - x_0y_0) \cdot 2^{n/2} + x_0y_0$$

So now we have just three products and three subproblems. The recurrence relation for this algorithm is $T(n) = 3T\left(\frac{n}{2}\right) + cn$. By master theorem the run time is $\Theta(n^{1.59})$.

```
RecursiveMultiply(x,y) { // [Kleinberg, pg. 233]
   write  x = x1 * 2^(n/2) + x0
          y = y1 * 2^(n/2) + y0

   p = RecursiveMultiply(x1 + x0, y1 + y0)
   x1y1 = RecursiveMultiply(x1, y1)
   x0y0 = RecursiveMultiply(x0, y0)

   return x1y1 * 2^n + (p - x1y1 - x0y0) * 2^(n/2) + x0y0 }
```

5.9 Additional Examples

Example 1: Calculating Median. Suppose that we have two lists A and B and each list consist of n sorted integers. We would like to determine the median of the merged list $A \cup B$. Recall that the median is the number located in the exact middle position of the list (if the list's length is odd) or is the average of the two numbers located in the middle of the list (if the list's length is even).

We of course could determine the median of $A \cup B$ in linear time by just merging the two lists and then accessing the middle elements of the resulting list. However, we could do better, we could apply a divide and conquer approach by *predicting* where the median of the merged list will be relative to the medians of the original lists. Making use of this information will turn our algorithm into a form of a binary search with a run time of $\mathcal{O}(\log n)$.

Algorithm's Stages

(1) ***Divide.*** We divide each list into two halves, based on the results of stage 3.

(2) ***Base Case***. If the length of each list is ≤ 2 then we calculate the median using brute force. The way stage 3 is set up, lists will always be of equal length throughout. If the lists are one element long, we just take the average of the two elements. If the lists are two elements long, the following formula works:

$$m_M = \tfrac{1}{2}\big[\max\{A(0), B(0)\} + \min\{A(1), B(1)\}\big]$$

(3) ***Combine.*** We compare the medians of the two lists. Denote the medians by m_A, m_B and let m_M denote the median of the merged list (we need to determine m_M). Then:

- **i.** if $m_A = m_B$ we return either of the medians as our median
- **ii.** if $m_A > m_B$ and as the lists are of equal length, it means that we *overshot* m_M in list A and *undershot* m_M in list B. Therefore, we should now recurse with *lower* half of A and *upper* half of B as inputs:
  - if n is even `A[0 .. floor(n/2)], B[floor(n/2)-1 .. n-1]`
  - if n is odd `A[0 .. floor(n/2)], B[floor(n/2) .. n-1]`
- **iii.** if $m_A < m_B$ and as the lists are of equal length, it means that we *undershot* m_M in list A and *overshot* m_M in list B. Therefore, we should now recurse with *upper* half of A and *lower* half of B as inputs:
  - if n is even `A[floor(n/2)-1 .. n-1], B[0 .. floor(n/2)]`
  - if n is odd `A[floor(n/2) .. n-1], B[0 .. floor(n/2)]`

Example. Let $A = \{2, 3, 5, 8, 10\}$ and $B = \{1, 4, 10, 12, 13\}$. The median of the merged list will turn out to be $m_M = 6.5$, it is used for illustrative purposes in the analysis.

Iteration 1:

- We start off with $m_A = 5$ and $m_B = 10$ and get that $m_A < m_B$.
- We are below 6.5 in A and are above 6.5 in B.
- We will look in the upper A and in the lower B.
- Our new lists are $A' = \{5, 8, 10\}$ and $B' = \{1, 4, 10\}$.

Iteration 2:

- Now we have $m_{A'} = 8$ and $m_{B'} = 4$ and get that $m_{A'} > m_{B'}$.
- This time we are above 6.5 in A' and are below 6.5 in B'.
- We will now look in the lower A' and in the upper B'.
- Our new lists are $A'' = \{5, 8\}$ and $B'' = \{4, 10\}$.

At this point we are down to just two elements in each list, so we perform the calculation:

$$\begin{aligned} m_M &= \frac{\max\{A(0), B(0)\} + \min\{A(1), B(1)\}}{2} \\ &= \frac{\max\{5, 4\} + \min\{8, 10\}}{2} \\ &= \frac{5 + 8}{2} = 6.5 \end{aligned}$$

Run time. The recurrence relation is $T(n) = T\left(\frac{n}{2}\right) + 1$, as each time we recurse, we half the size of the problem. All other work (e.g. finding and comparing medians, determining new indices) takes constant time.

By master theorem $T(n) \in \Theta(\log n)$. Of course we didn't really need to do this formal analysis to realize that we have logarithmic run time in this case.

Remarks.

1. This was a tutorial problem in 2018S2. Then one of the midterm problems in 2018S2 asked to modify this algorithm so that instead of returning the median, it would return the k-th smallest element of the combined list.
2. This is a simpler version of LeetCode problem no. 4. The main difference is that in the LeetCode problem lists A and B are of different sizes n and m respectively. That does change things quite a bit: the overshoot/undershoot approach of predicting the location of median used here is only valid when lists are of equal size.
3. This is an example of a divide and conquer algorithm where results of stage 3 directly influence how division is done in stage 1.
4. Many solutions to this problem on the web, including the one on www.geeksforgeeks.org, do not return correct results for lists that are of even length, as recursive splits are not done correctly for such lists in those solutions.

A rough Java implementation of the above algorithm is something like:

```
public double medianTwoLists(int[] A, int[] B, int startA, int endA, int startB, int endB) {

    /* Lists should always be the same size coming into the method */
    int n = endA - startA + 1;
    if (n == 2) {
        return (double) (Math.max(A[startA], B[startB]) + Math.min(A[endA], B[endB]))/2;
    } else if (n == 1) {
        return (double) (A[0] + B[0])/2;
    }

    double mA = computeMedian(A, startA, endA);
    double mB = computeMedian(B, startB, endB);

    if (mA == mB)
        return mA;
    else if (mA > mB) {
        if (n % 2 == 0)
            return medianTwoLists(A, B, startA, startA + n/2, n/2-1, endB);
        else
            return medianTwoLists(A, B, startA, startA + n/2, n/2, endB);
    } else {
        if (n % 2 == 0)
            return medianTwoLists(A, B, n/2-1, endA, startB, startB + n/2);
        else
            return medianTwoLists(A, B, n/2, endA, startB, startB + n/2);
    }
}

public double computeMedian(int[] arr, int start, int end){
    int n = end - start + 1;
    if (n % 2 == 0)
        return (double) (arr[start + (n/2)] + arr[start + (n - 1)/2])/2;
    else
        return arr[start + (n - 1)/2];
}

public static void main(String[] args) {
    MedianTwoLists m = new MedianTwoLists();
    int[] a = {2,3,5,8,10};
    int[] b = {1,4,10,12,13};
    int[] c = {1,3,5,7};
    int[] d = {2,4,6,8};
    double x = m.medianTwoLists(a, b, 0, a.length - 1, 0, b.length - 1);
    double y = m.medianTwoLists(c, d, 0, c.length - 1, 0, d.length - 1);
    System.out.println("Median of combined sorted lists a and b is: " + x); // 6.5
    System.out.println("Median of combined sorted lists c and d is: " + y); // 4.5
}
```

Example 2: Majority Element. Given an n element list, we would like to find the majority element. The majority element is the one that appears in the list more than $\lfloor \frac{n}{2} \rfloor$ times. For example, if the list is $\{2, 2, 3, 4, 5, 2, 2\}$, then the majority element is 2. There won't always be a majority element: the list $\{2, 2, 3, 4, 5, 6, 2\}$ does not have one.

This is an extended version of LeetCode problem no. 169 [the LeetCode problem assumes that the majority element always exists] and it was also a tutorial problem in 2018S2. There is a quite simple $\mathcal{O}(n)$ solution: we can just use a hash map to store counts of the number of times each number appears in the list. At the end we would check to see if there is a number in the hash map the count for which is greater than $\lfloor \frac{n}{2} \rfloor$. The divide and conquer algorithm for this problems runs only in $\mathcal{O}(n \log n)$, nevertheless it is a good exercise to work through.

Algorithm's Stages

(1) ***Divide.*** Just divide the list down the middle into two halves each time.

(2) ***Base Case***. If the list is one element long, its sole element is the majority element.

(3) ***Combine.*** There is an additional hurdle to handle in the combination step: some of the halves, the results of which we are aiming to combine, may not necessarily have a majority element. We need to consider the following cases:

- **i.** Neither half has a majority element, then the combined list will not have a majority element.
- **ii.** Both halves have the same majority element. In this case we just return either of the elements.
- **iii.** The left half has a majority element. Regardless if the right half has a majority element, count the number of elements in the combined list equal to the majority element of the left half. If the count is large enough for that element to remain a majority, return it. Otherwise, the combined list will not have a majority element.
- **iv.** The right half has a majority element. This case is symmetric to the above case.

The case where both halves will have a majority element is handled as a special case of either cases iii or iv above. A rough Java implementation is shown below, it will throw an exception if no element is the majority element, this makes the code a bit convoluted.

Run time. All of the counting involved in the combination step is done in linear time. We split the problem into two subproblems, each of which operates on half of the original list. Therefore, as before, the recurrence relation is $T(n) = 2T\left(\frac{n}{2}\right) + cn$ and the run time is $\Theta(n \log n)$.

```
// parts of this code are adapted from the solution presented on LeetCode
// helper that counts the number of elements in the list equal to the given element
private int count(int[] list, int num, int start, int end) {
    int count = 0;
    for (int i = start; i <= end; i++) {
        if (list[i] == num)
            count++;
    }
    return count;
}
```

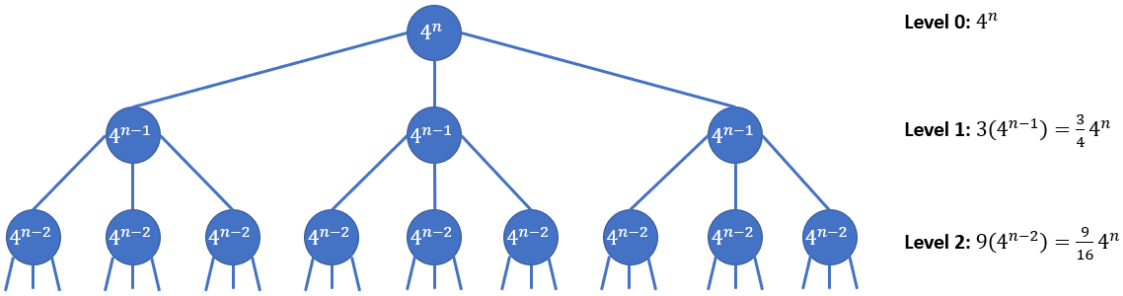
```java
private int majorityElement(int[] list, int start, int end) throws MajorityElementException {
    // base case
    if (start == end) return list[start];

    // set up variables
    int left = 0, right = 0, leftCount = 0, rightCount = 0;
    boolean leftExp = false, rightExp = false;
    int length = end - start + 1;
    int half = length/2;
    int mid = (start + end)/2;

    // recursive calls: try to catch the exceptions and throw a new
    // exception iff both of the calls threw one
    try { left = majorityElement(list, start, mid); }
    catch(MajorityElementException e) { leftExp = true; }
    try { right = majorityElement(list, mid+1, end); }
    catch(MajorityElementException e) { rightExp = true; }
    if (leftExp && rightExp) throw new MajorityElementException();

    // if the two halves agree on the majority element, return it
    if (!leftExp && !rightExp) {
        if (left == right)
            return left;
    }

    // otherwise, count the number of elements equal to the left and the right
    // majority elements in the combined list
    if (!leftExp) leftCount = count(list, left, start, end);
    if (!rightExp) rightCount = count(list, right, start, end);

    // make the final decision
    if (leftCount > half) return left;
    else if (rightCount > half) return right;
    else throw new MajorityElementException();
}

public static void main(String[] args) throws MajorityElementException {
    MajorityElement MajorityElement = new MajorityElement();

    int[] a = {2,2,1,1,1,2,2};
    int[] b = {3,3,3,4,2,1,3};
    int[] c = {2,2,3,3,4,4,5};
    int[] d = {1,2,3,4,5,6,7};
    System.out.println(MajorityElement.majorityElement(a, 0, a.length - 1)); // 2
    System.out.println(MajorityElement.majorityElement(b, 0, b.length - 1)); // 3
    System.out.println(MajorityElement.majorityElement(c, 0, c.length - 1)); // exception
    System.out.println(MajorityElement.majorityElement(d, 0, d.length - 1)); // exception
}

private static class MajorityElementException extends Throwable { }
```

Example 3: A Recurrence. $T(n) = 3T(n-1) + 4^n$.

This was a problem on the 2015W2 sample midterm 2. Master theorem doesn't apply as the recurrence isn't in the right form. The tree method (or the substitution method) works:

- As per diagram, there is $\left(\frac{3}{4}\right)^i 4^n$ amount of work involved at some level i.
- Each child is just *one less* the size of its parent, so we need $n-1$ levels to reach the base case.
- Total amount of work, via summation of geometric series is:

$$\begin{aligned}\sum_{i=0}^{n-1}\left(\frac{3}{4}\right)^i 4^n &= 4^n \sum_{i=0}^{n-1}\left(\frac{3}{4}\right)^i = 4^n\left(\frac{1-(3/4)^n}{1-(3/4)}\right)\\ &= 4^n \cdot 4\left(1-(3/4)^n\right)\\ &= 4^{n+1} - 4^{n+1}(3/4)^n\\ &= 4^{n+1} - 4\cdot 3^n \;\in\; \mathcal{O}(4^n)\end{aligned}$$



Example 4: Master Theorem. This problem is from 2015W2 sample final exam. Suppose that we have a recurrence given by $T(n) = aT(n/4) + n^x$ for $a \geq 1$ and $n \geq 4$. For which values of a and x will the algorithm run in $\Theta(n^2)$ time?

We have to consider cases 1 and 3 of the master theorem [we don't consider case 2 as it results in a run time of the form $\Theta(n^{\log_b a} \cdot \log n)$ and we are not looking for a run time that has a factor of $\log n$].

Case 1: The result of this case would be $\Theta(n^{\log_4 a})$. We want that to be $\Theta(n^2)$, which means that we must have $\log_4 a = 2$ or $a = 16$.

Now, to be in case 1 of master theorem, we must have $n^x \in \mathcal{O}(n^{\log_4 16 - \epsilon})$ for some $\epsilon > 0$. Looking at the exponents, we want $x \leq \log_4 16 - \epsilon$ or $x < \log_4 16$ or $x < 2$.

Case 2: The result of this case is $\Theta(n^x)$, so to get $\Theta(n^2)$ we must have $x = 2$.

To be in case 3 of master theorem, we must have $n^2 \in \Omega(n^{\log_4 a + \epsilon})$ for some $\epsilon > 0$. Looking at the exponents, we want $2 \geq \log_4 a + \epsilon$ or $2 > \log_4 a$. Raising both sides of this inequality to the exponent of 4 results in $2^4 > a$ or $16 > a$.

To be in case 3, we must also satisfy the regularity condition which is $a\left(\frac{n}{4}\right)^2 \leq cn^2$ for some $0 < c < 1$ in this case. We can divide both sides by n^2 and this inequality reduces to $\frac{a}{16} \leq c$. Since $a < 16$, then $\frac{a}{16} < 1$ so taking $c < 1$ works in this case.

In summary, we want either $a = 16$ and $x < 2$ or $x = 2$ and $1 \leq a < 16$.

6 Dynamic Programming

In a way, dynamic programming is a fancy term for recursion. In dynamic programming problems we solve a complex problem by finding a way to *relate* it to smaller subproblems. Then those smaller subproblems are related in the same way to base cases for which the solution should be obvious.

In an another way dynamic programming could be seen as a more elaborate divide and conquer approach. Instead of dividing the problem into a *small* number of *disjoint* subproblems, we divide the problem into *many* potentially *overlapping* subproblems in a way that run time is still manageable (more on this at the end of example 5).

As a preview, consider a game where we can traverse an $n \times m$ grid by starting on any tile in the top row and finishing on any tile in the bottom row. We collect $p(i, j)$ amount of points for each square that we pass through. The goal is collect the maximum amount of points possible.

So what are the base cases? Well we can start on any tile in the top row, and the amount of points we earn to start is just the value of that tile. This is stated in the top row of the recurrence given below.

Then, as we consider the tiles in the subsequent rows, there are up to three possible ways to arrive at those tiles (see red arrows), so we should take the maximum value of points from all of those incoming paths. This is stated in rows 2 and 3 of the following recurrence for the cases on the left and right edges of the grid (green and purple squares in the diagram) and then in the last row for the general case (all the blue squares).

All in all, we get a recurrence that relates the points collected at some square i, j to the points collected at squares $i-1, j$ and $i, j-1$ and $i-1, j-1$ (i.e. we get a recurrence that relates the problem its subproblems). Then, to determine the maximum amount of points that could be collected in this game, we need to substitute $i = n, j = m$ into the recurrence.

Once we have a recurrence, there is still the issue of implementing it in a sensible way. While the following recurrence could be implemented naively, the naive implementation of recursion is very resource inefficient and the program would either take a very long time to finish or would crash when n and m are moderately large. So the first issue, that we will address in section 6.1, is how to efficiently implement recurrences. Then, the following sections will then look at various standard dynamical programming problems.

$$T(i,j) = \begin{cases} p_{(i,j)} & \text{if } i = 1 \\ p_{(i,j)} + \max\{T(i-1,j), T(i-1,j+1)\} & \text{if } i > 1, j = 1 \\ p_{(i,j)} + \max\{T(i-1,j-1), T(i-1,j)\} & \text{if } i > 1, j = m \\ p_{(i,j)} + \max\{T(i-1,j-1), T(i-1,j), T(i-1,j+1)\} & \text{otherwise} \end{cases}$$

5	2	3	10	8	9	11
1	12	4	20	7	11	22
7	1	8	30	5	4	33
10	15	16	40	1	3	55

line 2 and 3 are edge cases (purple and green) - only 2 ways to reach the from the top

6.1 Four Ways to Implement Fibonacci

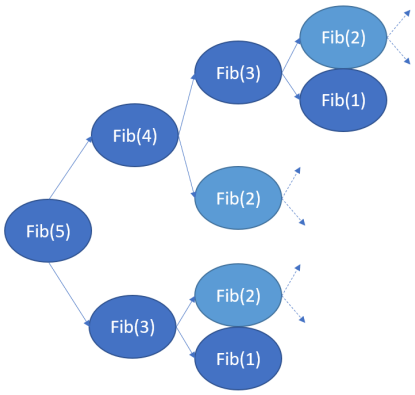
Recall the recurrence for the Fibonacci numbers:

$$F(n) = \left\{ \begin{array}{ll} 1 & \text{if } n \leq 2 \\ F(n-1) + F(n-2) & \text{otherwise} \end{array} \right\}$$

We will use this simple recurrence to illustrate all of the possible ways a recurrence could be implemented in code. All code in this chapter is written in Java.

1: Naive Recursion. The simplest way to implement a recurrence is to literally translate the recurrence into code as is done below. However this implementation is very inefficient and is not practical for large n.

The inefficiency is a result of the method being called several times with the exact same input. In the partial recursion tree for the `Fib(5)` call shown on the side, there are three repeated calls to `Fib(2)`. This problem becomes exponentially worse as n increases.




```java
int FibRecur(int n) {
    if (n <= 2)
        return 1;

    return FibRecur(n - 1) + FibRecur (n - 2);
}
```

2: Memoization (Top-Down). One solution to the above issue is to cache the redundant calls or save the results of those calls into memory. We use an array to do this and pass the array to each recursive call.

The subsequent recursive calls check if the value they need to compute is already available in the array and then just return it if that is the case. Otherwise, those calls would compute the required value and save it to the array. This approach is known as memoization or the "top-down" approach.

```java
/* A wrapper to initialize the recursive calls with an empty array. */
int FibMem(int n) {
    return FibMemHelper(n, new int[n + 1]);
}                                  makes an array of size n+1
                                   (everything initialized to be 0)

/* The actual recursive method. */
int FibMemHelper(int n, int[] arr) {
    if (n <= 2)
        return 1;
    if (arr[n] != 0)
        return arr[n];     this case means if arr[n] has already be found

    return arr[n] = FibMemHelper(n - 1, arr) + FibMemHelper(n - 2, arr);
}
```

3: Dynamic Programming (Bottom-Up). The general idea here is the same as with memoization: we use an array to save the values that we already know. However, in this approach, the array is filled in the "bottom-up" fashion. We start with the known base cases and then just run a for loop to fill in the rest of the array. The statement inside the for loop is just a direct translation of the recurrence statement. This is the approach that is typically used to implement all dynamic programming problems, once the recurrence relationship has been established.

```
1  int FibDp(int n) {
2      int[] arr = new int[n + 1];
3      arr[1] = 1; arr[2] = 1;
4
5      for (int i = 3; i <= n; i++)
6          arr[i] = arr[i - 1] + arr[i - 2];
7                          this is the recurrence relation
8      return arr[n];
9  }
```

4: Two Pointers. The two pointer approach is special to the Fibonacci recurrence as well as a few others. Something special about the Fibonacci recurrence is that when determining $F(i)$, it relies only upon the two values in the sequence that come *immediately* before the current one. So we don't necessarily need to keep an entire array of all historical Fibonacci numbers as we work our way through the for loop.

So what we do is use two integers: `prev` and `sum`. Coming into each iteration:

- `prev` stores $F(i-2)$
- `sum` stores $F(i-1)$

In the iteration:

- in line 6, `prev` is added to `sum`, so then `sum` becomes $F(i)$, which once again becomes $F(i-1)$ once the loop advances
- in line 7, `prev` is updated to store $F(i-1)$, which once again becomes $F(i-2)$ once the loop advances
- in line 5, we must store the previous value of `sum` into `temp` to use for updating `prev` after `sum` been updated

```
1  int FibDpOpt(int n) {
2      int prev = 1, sum = 1, temp = 0;
3
4      for (int i = 3; i <= n; i++) {
5          temp = sum;
6          sum += prev;
7          prev = temp;
8      }
9
10     return sum;
11 }
```

6.2 1D Dynamic Programming

The following steps apply to both 1D and 2D dynamic programming problems.

The Steps

0. In some cases, transform the problem into a form such that dynamic programming could be applied (see example 5).
1. Identify the subproblems and determine how those relate to the main problem. The latter can be done by asking the question: *knowing a solution to the subproblem, how can we apply that solution to solve the larger problem, or vice versa?* Some typical scenarios are:
   - if the problem involves arrays or matrices, subproblems would be about arrays or matrices of smaller sizes – e.g. think about how a problem for the array `A[0..i-1]` relates to the problem for the array `A[0..i]`;
   - if the problem involves some discrete quantity [e.g. number of items, amount of money rounded to the dollar, weight rounded to the kilogram, etc.], then the subproblems would be about all smaller amounts of those quantities – e.g. if the main problem asks for a solution for a 5 item set, the subproblems would look at solutions for sets of 4, 3, 2, 1 or 0 items.
2. Identity the base cases.
3. Derive the recurrence based on the answers to 1 and 2.
4. Implement the recurrence in code.

Example 1: Climbing Stairs. This is LeetCode problem no. 70. There is an n step staircase, and to climb it, one must go up by taking either *one* or *two* steps at a time. In how many different ways it is possible to reach the top?

So what are the subproblems here? The number of steps in a staircase is certainly a discrete quantity, so lets relate an n step staircase to staircases with smaller numbers of steps. Well, if we know the number of different ways to go up to the top of $n-1$ and $n-2$ step staircases, then to get to the top of an n step staircase, we could either take an additional single step from the top of an $n-1$ stair staircase or two additional steps from the top of an $n-2$ stair staircase.

Therefore, to determine the number of ways to the top of the n step staircase, just *add* the number of ways to get to the top of the $n-1$ and $n-2$ stair staircases. In other words, we put together the only two possible options of either taking one additional step or two additional steps from the shorter staircases. The base cases here are that there is just one way to climb a single-step staircase and two ways to climb a two-step staircase.

$$\begin{aligned}\text{Stairs}[1] &= 1\\ \text{Stairs}[2] &= 2\\ \text{Stairs}[n] &= \text{Stairs}[n-2] + \text{Stairs}[n-1]\end{aligned}$$

So, this is just Fibonacci numbers. Finally an another real world application of Fibonacci numbers! Implementation of this recurrence was covered in section 6.1.

Example 2: Coin Change. This is LeetCode problem no. 322. We are given a fixed set of n coins of different denominations and an amount of money m. We need to determine the fewest number of coins that could make change for the given amount. Any coin could be used as many times as need be. We assume that denominations of all coins are integers.

For example, consider the set of coins with denominations $\{1, 3, 5\}$ and an amount of \$12 for which we need to make change. Then the smallest number of coins that we need is 4 as $12 = 5 + 5 + 1 + 1$ or $12 = 3 + 3 + 3 + 3$.

So what are the subproblems here? Well the denominations of all coins are integers, so the given amount of money has to be an integer and thus the amount of money is a discrete quantity. Therefore, if we know how to make change for some smaller amount of money, we perhaps could use that solution to determine how to make change for a larger amount.

So how do we go from the smaller problem to the larger problem and vice versa? First, if we know how to make change for an amount of x dollars, do we immediately know how to make change for some larger amount? The answer is yes, if the larger amount is larger than the smaller amount by exactly the denomination of some coin. For example, if we know how to make change for \$12, then with the coin set that we have, we immediately know how to make change for \$13, \$15 and \$17 by taking just *one* additional coin. Conversely, if we are given some arbitrary amount of money x, we could try subtracting the denominations of all coins c_i from x and see if we know how to make change for at least one of the amounts $x - c_i$, as those amounts are just a single coin away from x.

Second, once we know all the ways we can make change for some given amount, how do we know which one of those results in the *smallest* number of coins being used? Well we just take the minimum of all possibilities. So here is how the recurrence and the algorithm work: given an amount x, fetch the results for all possible amounts $x - c_i$. Take the minimum of all of those results and add one to it, since we need to use one additional coin to get to x.

Of course it makes sense to examine only cases where $x - c_i$ is positive. This is what the if statement is checking for in line 8 of the following implementation. The process of finding the minimum is done on a "rolling" basis in line 9, so that there is no need to keep track of results for all amounts $x - c_i$ and take their minimum at the end of the inner for loop.

What are the base cases? Well actually the only base case in this problem is that it requires zero coins to make change for \$0. For all other amounts, if there is no coin with denomination of 1, we don't know if we can make change until we start examining the set of given coins and this is best handled via the recurrence. So we just initialize spots in the array corresponding to all other amounts to some large value (`amount+1` in this case) so that it works nicely when determining the minimum in line 9. Then, if the recurrence determines that change could be made for some amount, the recurrence will update the corresponding value there. Otherwise, if `amount+1` remains in any spot in the array, it means it wasn't possible to make change for that amount and the method returns `-1` (see line 15).

$$\text{NoOfCoins}[0] = 0$$
$$\text{NoOfCoins}[x] = \min_{\substack{i \in \{0,\ldots,n-1\}, \\ x - c_i \geq 0}} \{\text{NoOfCoins}[x - c_i]\} + 1$$

So why use `amount+1` to initialize the array instead of say the typical `Integer.MAX_VALUE`? The issue with the latter is that line 9 would add 1 to the already maximum value for integer and cause an overflow, resulting in bogus values being compared to determine the minimum. Technically, any integer that is larger than `amount` or is one smaller than `Integer.MAX_VALUE` could be used to initialize the array. We don't want to initialize the array to `amount` or to a smaller value, as that would cause an issue if the coin set was just the set $\{1\}$.

```
1  public int coinChange(int[] coins, int amount) {
2      int[] noCoins = new int[amount + 1];
3      Arrays.fill(noCoins, amount + 1);
4      noCoins[0] = 0;        didn't use inf bc didn't want to cause overflow (on line 9)
5
6      for (int curAmount = 1; curAmount <= amount; curAmount++) {
7          for (int coin : coins) {    search through set of bills/coins available (ex. nickle, penny, dime)
8              if (coin <= curAmount) {
9                  if (noCoins[curAmount - coin] + 1 < noCoins[curAmount])
10                     noCoins[curAmount] = noCoins[curAmount - coin] + 1;
11             } // or replace the two previous lines with Math.min()
12         }
13     }
14
15     return noCoins[amount] == amount + 1 ? -1 : noCoins[amount];
16 }                                  if amount+1 was in any spot in the array, return -1. IDK this syntax
```

The run time of the solution is $\mathcal{O}(nm)$ where n is the number of coins and m is the amount. Below is a concrete example of the array `noCoins` for a coin set $\{2, 5, 10\}$ and an amount of \$12. By looking up 12 in the table we see that we would need two coins to make change for 12. It is not possible to make change for either \$1 or \$3 with this set of coins, as is confirmed by the corresponding 13s in the second row.

Amount	0	1	2	3	4	5	6	7	8	9	10	11	12
`noCoins`	0	13	1	13	2	1	3	2	4	3	1	4	2

N.B. There are two key differences between this problem and the previous on climbing stairs:

1. In this problem, there is the extra step of finding the minimum number of coins that do the job, as opposed to just returning the sum of all possible combinations of coins that could be used.
2. In the climbing stairs problem, when determining the result for an n stair staircase, the recurrence used the results of just the *two* previous subproblems. In this problem, the recurrence relies on results of potentially *any* of the previous subproblems. So the two pointer implementation approach (approach no. 4 of section 6.1) wouldn't be a valid option in this case.

Example 3: Longest Increasing Subsequence. This is LeetCode problem no. 300 and it also appeared as a tutorial problem in 2018S2. In this problem we are given an unsorted sequence of integers, say $\{1, 9, 17, 5, 8, 6, 4, 7, 12, 3\}$ and we are asked to determine the length of the longest increasing subsequence. A subsequence is a sequence that could be derived from the original sequence by deleting some or no elements, without changing the order of the original elements (Wikipedia). In this particular example, the longest increasing subsequence is $\{1, 5, 6, 7, 12\}$ and its length is 5.

Since this problem is about an array of numbers, then it is natural to expect that the subproblems will be concerned with smaller arrays of numbers. First thing to note in this problem is that any element of the sequence could be a starting element of some subsequence or even of the longest increasing subsequence. The length of that single-element subsequence is of course just one. How can we make that subsequence longer or, in other words, how do we go from the smaller problem to the larger problem and vice versa?

Well we need to go further down in the sequence until we reach an element that is larger than the current element. Then we can add that element to our subsequence and its length would increase by one. Conversely, starting from an element somewhere in the sequence, we could scan through all preceding elements of the sequence, looking for elements that are smaller than that element. Then we would increase by one the lengths of all of the longest increasing subsequences that end on those elements. Finally, just like in the previous problem when we were looking for the smallest amount of coins and took the minimum of the fetched results, here we must take the maximum of all of those lengths.

So we will have the dynamic programming array L that, at each index, would store the length of the longest increasing subsequence that ends on the corresponding element in the original array. We already discussed the base cases: the entire array would need to be initialized to 1 as each element could be a subsequence of length one on its own. Here is the recurrence relation:

$$L[i] = \begin{cases} \max\limits_{\substack{1 \le j < i, \\ A[j] < A[i]}} \{L[j]\} + 1 & \\ 1 & \text{otherwise} \end{cases}$$

When done, we must return the maximum value of the entire array L, as opposed to just returning the last entry in the array. The last entry in the array is just the maximum subsequence length of the subsequence that ends in that element and may not necessarily be the overall maximum subsequence length.

To illustrate this, the array L for the original array that was given at the start of this problem is shown below. In particular, at element 1 we have a length of 1, then at 9 we make that length 2 and at 17 we get up to 3. However, then, at the next element, the best length we can have is 2 which corresponds to subsequence $\{1, 5\}$ as $9 \not\leq 5$ and $17 \not\leq 5$. Likewise, for the last element, the best we can do is also a length of 2 which corresponds to subsequence $\{1, 3\}$. So we don't return the last element of L, but return the maximum value of L which is 5.

Original Array	1	9	17	5	9	6	4	7	12	3
Array L	1	2	3	2	3	3	2	4	5	2

In the implementation, the process of determining the maximum of the array L is also done on a "rolling basis" in line 11 below. Line 10 determines the other maximum, the one that is specified in the recurrence relation. The run time is $\Theta(n^2)$.

```
1  public int lengthOfLIS(int[] nums) {
2      if (nums.length == 0) return 0;
3      int[] lengths = new int[nums.length];
4      Arrays.fill(lengths, 1);
5      int maxLength = 1;
6
7      for (int i = 1; i < nums.length; i++) {
8          for (int j = 0; j < i; j++) {
9              if (nums[j] < nums[i]) {
10                 lengths[i] = Math.max(lengths[i], lengths[j] + 1);
11                 maxLength = Math.max(lengths[i], maxLength);
12             }
13         }
14     }
15
16     return maxLength;
17 }
```

here we're keeping track of the actual max length

contiguous = consecutive

Example 4: Maximum Subarray. This is LeetCode problem no. 53. Given an array of integers, we need to find the *contiguous* subarray which has the largest sum and return its sum. The subarray must be at least one integer long. For example, for the array $\{-2, 1, 1, 2, -1, 3, -4, -3, 3, 2\}$, the contiguous array with the largest sum is $\{1, 1, 2, -1, 3\}$ and its sum is 6.

This problem is quite similar to, yet somewhat different from example 3. Juts like in example 3, we are looking for a subarray of the original given array that satisfies a certain goal. However, this time we are looking for a *contiguous* subarray, while in example 3 we were looking for just a subsequence.

Just like in the previous example, any element of the original array could serve as the starting point of the maximum subarray that we are looking for. The starting sum would be just the value of that element. How could we increase the sum? Well, we could add an another element of the array to the sum. However, in this example, as the subarray must be contiguous, we are limited to just adding the *next* element to the sum, as opposed to the previous example where we considered all larger subsequent elements. Moreover, we only add the next element if it makes sense, as in, its addition actually improves the sum. Conversely, at some point in the array, we can either take just the value of the current element and start a new sum or we can try adding it to the sum of the subarray *immediately before*, if that helps. Here are two equivalent ways to write the recurrence (the first way is a direct translation of the proceeding discussion, the second way perhaps just simplifies things using the definition of maximum):

$$S[i] = \begin{cases} S[i-1] + A[i] & \text{if } i > 1 \text{ and } S[i-1] + A[i] > A[i] \\ A[i] & \text{otherwise} \end{cases} = \begin{cases} A[0] & \text{if } i = 0 \\ \max\{S[i-1] + A[i], A[i]\} & \text{otherwise} \end{cases}$$

In example 3, the base cases were that the length of a single-element subsequence is one. In this example, the base cases are that the maximum sum of a single-element subarray is just the value of that element.

Just like in example 3, at each index, the array S would store just the maximum sums for the subarrays that end on the respective element in the original array. So at the end, we would return the maximum value of S, as opposed to just returning the last element of S.

Two possible implementations are given below. The first implementation just translates the first version of the recurrence relation as it is written and is quite similar to the implementation of example 3. In this case, there just isn't a need for the inner for loop and an if statement to look for and fish out all preceding elements that are smaller than the current element. So, the run time in this case is just $\Theta(n)$.

The second implementation doesn't use an additional array to store sums. In this example, as was the case in example 1, the recurrence always reaches just one element back when determining the maximum subarray length for the next element of the array. So in this case, it is appropriate to use a "two-pointer" type approach (i.e. approach no. 4 of section 6.1). This implementation approach resembles the second version of the recurrence relation a bit more.

```java
/* Array-based DP implementation */
public int maxSubArray(int[] nums) {
    int n = nums.length;
    int maxSum = nums[0];
    int[] sums = nums; // initialize sums to just the values of respective elements
                       // to begin with, like we initialized L to all 1s in example 3

    for (int i = 1; i < n; i++) {
        sums[i] = Math.max(sums[i], sums[i - 1] + nums[i]);
        maxSum = Math.max(sums[i], maxSum);
    }

    return maxSum;
}
```

```java
/* Implementation that doesn't use an additional array */
public int maxSubArray(int[] nums) {
    int n = nums.length;
    int maxSum = nums[0];

    for (int i = 1; i < n; i++) {
        nums[i] = Math.max(nums[i], nums[i - 1] + nums[i]);
        maxSum = Math.max(nums[i], maxSum);
    }

    return maxSum;
}
```

Below is the array S for the original array given at the start of the problem. The maximum subarray is highlighted in blue. Observe that at the second last element of the array, the previous sum of -1 is completely ignored, as adding it to that element wouldn't help. Essentially, all negative sums are the ones that get ignored in favor of just starting a new subarray.

Original Array	-2	**1**	**1**	**2**	**-1**	**3**	-4	-3	3	2
Array S	-2	1	2	4	3	6	2	-1	3	5

Summary of Examples 1-4.

	Example 1: Climbing Stairs	**Example 2:** Coin Change	**Example 3:** Longest Increasing Subsequence	**Example 4:** Maximum Subarray
Subproblems are based on	Length of the Staircase	Amounts of Money	Sequence Length	Sequence Length
How many steps does the recurrence reach back for?	Two	Any	Any	One
Array-based or two-pointer implementation approach?	Two-Pointer	Array-Based	Array-Based	Two-Pointer
Return last element of the array or the maximum?	Last Element	Last Element	Maximum	Maximum

Overall, there is even a lot more symmetry in how recurrences of examples 2-4 translate to code. For example, the conditions on minimum or maximum statements in the recurrences naturally translate into if statements in the implementations. Dynamic programming is not the only possible way to solve example 4, other approaches include a "sliding window" type approach and a divide and conquer approach.

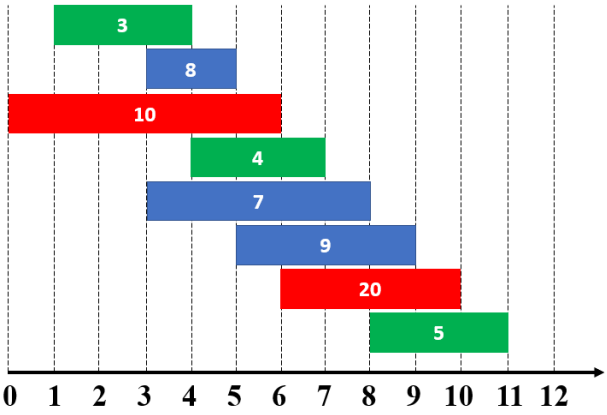
Example 5: Weighted Interval Scheduling. This is LeetCode problem no. 1235 and is also the first dynamic programming example discussed in the textbook (pg. 252-258).

We already seen two problems concerned with interval scheduling on a shared resource in the chapter on greedy algorithms. In section 4.1 all intervals were unweighted, had a start time s_i and an end time f_i. The goal was to schedule as many intervals as possible. In section 4.2 intervals were weighted, but only had their lengths specified, as opposed to precise start and end times. There the goal was to schedule all intervals in a way that the ones with greater weight would finish as soon as possible.

This problem is more so associated with the one in section 4.1. As before, let E be the set of n intervals with a start time of s_i and an end time of f_i for each interval. We will add the weight parameter w_i to each interval and there will be a parameter triplet (f_i, s_i, w_i) associated with each interval. The goal will be to schedule as many intervals as possible in a way that they don't overlap and that the sum of their weights is maximized.

Consider the intervals illustrated in the following diagram, they are already sorted by their end times in the ascending order. Recall that the greedy algorithm from section 4.1 schedules the greatest number of intervals by consistently picking the interval that finishes first and then discarding all non-compatible intervals from further consideration. So in this particular case, the greedy algorithm will pick the intervals highlighted in green, for a total of three intervals and a total weight of 12 (three is the greatest number of compatible intervals in this case). However, the solution we are looking for is the one highlighted in red which has fewer intervals yet gets us to the maximum total weight of 30.

It almost seems like we need to run the greedy algorithm several times here. It is almost like we want to do a run of the greedy algorithm starting from every one of the intervals that we have and then just pick the result with the greatest total weight. In particular, had we started the greedy algorithm from the interval that has the weight of 10, we would have arrived at the correct answer. This is sort of what the dynamic programming approach will end up doing.



As always, in dynamic programming we want to start from smaller subproblems and work towards the larger problem. In this case, the smaller subproblems will of course work with fewer intervals than the larger problem. We will move back and forth between the smaller subproblems and the larger problem by adding or removing some interval from the set of intervals that are up for consideration at any given time.

The base case will be either that the maximum weight achieved with no intervals is zero or that the maximum weight achieved with just one interval up for consideration is just the weight of that interval. Then, when we want to expand the subproblem by adding some specific interval, there will be two things to consider:

1. is the addition of the this interval valid, in particular, is the interval compatible with the pre-existing intervals, and
2. will the addition of the this interval actually improve the total weight.

An interval would only be added if the answer to both questions is affirmative. For comparison, recall the coin change problem. There we referred back only to the subproblems which were exactly a coin denomination away from the bigger problem. Then, to achieve the best result, we picked the subproblem with the minimal number of coins. Likewise here we will only add intervals that are compatible and only if they improve the weight. In the coin change problem it was quite easy to pinpoint which of subproblems we needed to refer back to (we simply subtracted the denominations of the different coins that we had). In this case there is more work involved in determining which of the remaining intervals will be compatible. An array labeled `p` will be constructed to facilitate the work that needs to be done. There are two possible approaches to constructing `p` and solving the problem. The first approach is the one most commonly used, while the second approach is presented for illustrative purposes and as a lead up to example 6.

Approach 1: Backward Recursion. Refer to the diagram on the left. This is the most common approach and is the one used by the textbook.

- Sort intervals by their *end* times in the ascending order [i.e. so that $f_1 \leq f_2 \leq \cdots \leq f_n$]
- Define `p[i]` to store the index of the *last* compatible interval. In other words, for every interval i, store the index of the interval j that ends right before i starts. Or store the index of the leftmost interval that ends right before i starts.
- In this example the array `p` is as follows (refer to the red lines in the diagram). In particular, there are no intervals that end before intervals 1-3 or 5 start. Intervals 1,2,3,5 are the leftmost intervals that end before intervals 4,6,7,8 start respectively.

0	0	0	1	0	2	3	5

- We have the following recurrence: at each interval i, if we want to add the interval, we must refer back to the result for the last interval that is compatible with this interval. If that result plus the weight of the current interval $[w_i + W(p(i))]$ is better than just ignoring the interval $[W(i-1)]$, we add it. The optimal value is $W[n]$.

$$W[i] = \left\{ \begin{array}{cc} w_1 & \text{if } i = 1 \\ \max\{w_i + W(p(i)),\ W(i-1)\} & \text{otherwise} \end{array} \right\}$$

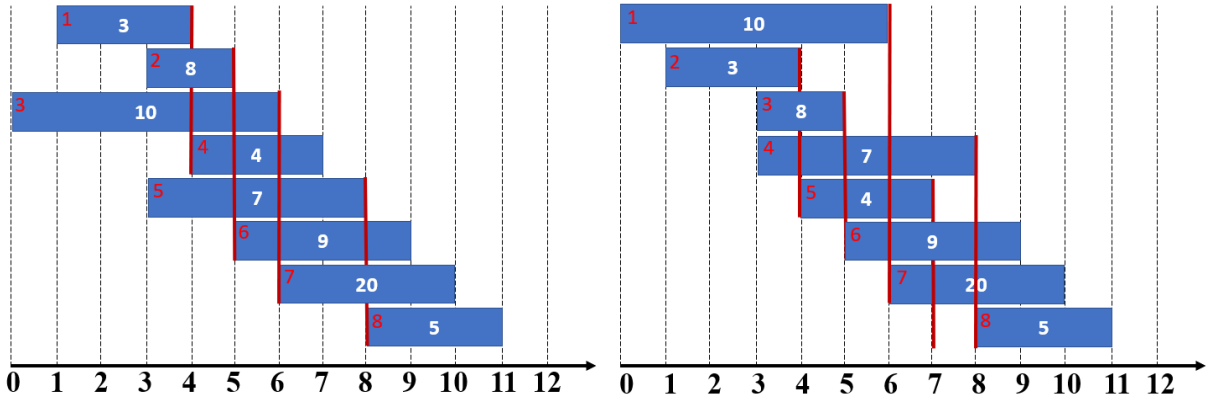
Approach 2: Forward Recursion. Refer to the diagram on the right. This approach is from here: `http://pages.cs.wisc.edu/~shuchi/courses/787-F09/scribe-notes/lec3.pdf`.

- Sort intervals by their *start* times in the ascending order [i.e. so that $s_1 \leq s_2 \leq \cdots \leq s_n$]
- Define `p[i]` to store the index of the *next* compatible interval. In other words, for every interval i, store the index of the interval j that starts right after i ends.
- In this example the array `p` is as follows (refer to the red lines in the diagram). In particular, there are no intervals that start after intervals 6-8 end. Intervals 7,5,6,8,8 are the ones that start right after intervals 1,2,3,4,5 end respectively.

7	5	6	8	8	0	0	0

- The recurrence has the same form as before, except that we now refer forward via the expression $W(i+1)$ instead of $W(i-1)$. The base case is now $i = n$ instead of $i = 1$. The optimal value is $W[1]$.

$$W[i] = \left\{ \begin{array}{cc} w_n & \text{if } i = n \\ \max\{w_i + W(p(i)),\ W(i+1)\} & \text{otherwise} \end{array} \right\}$$



The second approach is perhaps a bit more intuitive in the algorithmic sense: start from any interval i, take it, go forward, find the *next* compatible interval $p(i)$ and then decide if we want to include that interval $[w_i + W(p(i))]$ or skip it and proceed straight to $W(i+1)$.

On the other hand, the first approach better reflects how recursion is typically done, as naturally, at each step, we refer back to a smaller i and as is done via $W(i-1)$ or via $p(i)$ that refers to the *last* compatible interval. While both approaches worked fine in this problem, there are problems were only one of the approaches would work (see example 6).

Some take away points:

- In either approach, at each interval, we decide if we **take it or leave it**. The decision is made by taking the maximum of the two respective terms.
- In the greedy algorithm for the unweighted problem, we had to sort the intervals by their end times. That was the defining property of why the greedy algorithm worked. In this case, as we are doing dynamic programming, we have some flexibility to either sort by start times and go forward or sort by end times and go backward.

To implement the first approach we would run a for loop from 0 up to n and to implement the second approach we would run a for loop from n down to 0. Java implementations of both approaches are on the following pages.

Worked out arrays W for both approaches are presented below. In the first approach, values $W(i)$ represent the maximum weight achieved with the subset of $1, \ldots, i$ intervals. In the second approach, values $W(i)$ represent the maximum weight achieved with the subset of $n, \ldots, i$ intervals.

Green arrows illustrate the jumps eventually taken by the optimal solution. In the table on the right, if we start with the first interval of weight 10, the next compatible interval is the one with weight 20 (follow the green arrow) and we get the optimal value of 30. If we start with the first interval of weight 3, the best we can do is a total weight of 28.

Another example: in the table on the left, in the last column, the choice is between sticking to the 30 from before or trying to take the current interval with the weight of 5 plus the last compatible result with the weight of 10 (follow the arrow). The former wins as $15 < 30$.

Weights	3	8	10	4	7	9	20	5
$p(i)$	0	0	0	1	0	2	3	5
$W(i)$	3	8	10	10	10	17	30	30
Decision	N/A	Take	Take	Skip	Skip	Take	Take	Skip

Weights	10	3	8	7	4	9	20	5
$p(i)$	7	5	6	8	8	0	0	0
$W(i)$	30	28	28	20	20	20	20	5
Decision	Take	Skip	Take	Skip	Skip	Skip	Take	N/A

Run Time.

- We first need to sort the intervals, this is done in $\mathcal{O}(n \log n)$.
- We then loop over n intervals and at each interval we need to determine the value of $p(i)$. The latter could be done via binary search in $\mathcal{O}(\log n)$. So the total for the loop will be $\mathcal{O}(n \log n)$.
- Overall total is $\mathcal{O}(n \log n)$.

Brute force approach is $\mathcal{O}(2^n)$: for each interval, we would make an arbitrary binary decision if we should take it or leave it, resulting in 2^n possibilities. Sorting the intervals organizes the structure of the problem. Taking the dynamic programming approach which *immediately throws away results of hopeless subproblems* significantly decreases the run time.

Java implementation of the first approach: Java's built in binary search function is used here, which makes fetching indexes a bit convoluted. An alternate would be to implement a binary search helper custom to this problem from scratch. All arrays use 0-based indexing.

```
public static class Interval implements Comparable<Interval> {
    public int startTime, endTime, weight;

    public Interval(int startTime, int endTime, int weight) {
        this.startTime = startTime;
        this.endTime = endTime;
        this.weight = weight;
    }

    // needed for sorting and binary search (sort by end time)
    public int compareTo(Interval i) {
        return this.endTime - i.endTime;
    }
}

public int jobScheduling(int[] startTime, int[] endTime, int[] profit) {
    int n = startTime.length;

    // Convert given data into an array of interval objects and sort by end time
    Interval[] intervals = new Interval[n];
    for (int i = 0; i < n; i++)
        intervals[i] = new Interval(startTime[i], endTime[i], profit[i]);
    Arrays.sort(intervals);

    // Build the p array: for each interval we look for one that ends before this one
    // starts, adjust index when Arrays.binarySearch doesn't find an exact match [see
    // Java documentation]. If there isn't a last compatible event, set p[i] = -1.
    int[] p = new int[n], int idx;
    for (int i = 0; i < n; i++) {
        idx = Arrays.binarySearch(intervals, new Interval(0, intervals[i].startTime, 0));
        if (idx < -1)
            idx = - idx - 2;
        p[i] = idx;
    }

    // Do dp, initialize W[0] = Weight of first interval
    int[] W = new int[n]; W[0] = intervals[0].weight;
    for (int i = 1; i < n; i++) {
        int prevMaxProfit = p[i] < 0 ? 0 : W[p[i]];
        W[i] = Math.max(prevMaxProfit + intervals[i].weight, W[i - 1]);
    }

    return W[n - 1];
}
```

Java implementation of the second approach.

```java
public static class Interval implements Comparable<Interval> {
    public int startTime, endTime, weight;

    public Interval(int startTime, int endTime, int weight) {
        this.startTime = startTime;
        this.endTime = endTime;
        this.weight = weight;
    }

    // needed for sorting and binary search (sort by start time)
    public int compareTo(Interval i) {
        return this.startTime - i.startTime;
    }
}

public int jobScheduling(int[] startTime, int[] endTime, int[] profit) {
    int n = startTime.length;

    // Convert given data into an array of interval objects and sort by start time
    Interval[] intervals = new Interval[n];
    for (int i = 0; i < n; i++)
        intervals[i] = new Interval(startTime[i], endTime[i], profit[i]);
    Arrays.sort(intervals);

    // Build the p array: for each interval we look for one that starts right after this
    // one ends, adjust index when Arrays.binarySearch method doesn't find an exact match
    // [see Java documentation]. If there isn't a next compatible event, set p[i] = -1.
    int[] p = new int[n], int idx;
    for (int i = 0; i < n; i++) {
        idx = Arrays.binarySearch(intervals, new Interval(intervals[i].endTime, 0, 0));
        if (idx < -n)
            idx = -1;
        if (idx < -1)
            idx = - idx - 1;
        p[i] = idx;
    }

    // Do dp, initialize W[n - 1] = Weight of the last interval
    int[] W = new int[n]; W[n - 1] = intervals[n - 1].weight;
    for (int i = n - 2; i >= 0; i--) {
        int nextMaxProfit = p[i] < 0 ? 0 : W[p[i]];
        W[i] = Math.max(nextMaxProfit + intervals[i].weight, W[i + 1]);
    }

    return W[0];
}
```

Extracting the Result. In order to extract the actual intervals that make up the optimal solution, we need to examine the resulting array W and trace out the jumps that lead to the maximum weight (i.e. we need to locate the jumps that were shown using green arrows in the previous diagrams).

So, we loop through the generated W array and detect jumps by checking when it is that $W(i) \neq W(i-1)$ in the first approach or $W(i) \neq W(i+1)$ in the second approach. If such conditions are satisfied, then a jump must have happened, we add the corresponding interval to the resulting list and force the looping index to jump to the next compatible interval, which is stored in $p(i)$. Otherwise, we just increase or decrease the looping index by one.

We loop through W in the opposite order to order of the loops that were used to generate W. The run time of this step is at most $\mathcal{O}(n)$.

```
// Corresponds to the first approach: loop from n down to 0
ArrayList<Interval> result = new ArrayList<>();
int i = n - 1;
while (i >= 0) {
    if (i == 0 || W[i] != W[i - 1]) {
        result.add(intervals[i]);
        i = p[i];
    } else
        i--;
}
```

```
// Corresponds to the second approach: loop from 0 up to n
ArrayList<Interval> result = new ArrayList<>();
int i = 0;
while (i <= n - 1) {
    if (i < 0) break; // need this if we reach p[i] = -1
    if (i == n - 1 || W[i] != W[i + 1]) {
        result.add(intervals[i]);
        i = p[i];
    } else
        i++;
}
```

Example 6: A Frustrating Exam. This problem appeared on the 2017S2 final exam and is also available here: `https://www.cs.yale.edu/homes/aspnes/pinewiki/attachments/CS365(2f)Assignments(2f)FinalExam/final.solutions.pdf`.

Suppose there is an exam with questions numbered $1, 2, 3, \ldots, n$. The questions must be answered in order, but questions could be skipped. Each question i is worth p_i points; however, answering it causes enough frustration that the following f_i questions will be skipped. Given the values (p_i, f_i) for each question upfront, we want to determine the maximum score that could be obtained on this exam.

As before, we want to identify the subproblems and determine how to move between those subproblems and the larger problem. Naturally, the subproblems will be concerned with a smaller subset of questions. How do we go from a smaller subproblem to the larger problem

and vice versa? Well if we want to add a question to a smaller set of questions, we can either answer it, earn the points and skip ahead f_i questions or we can skip it and move onto question that immediately follows. Just like in the the previous problem, we are faced with a **take it or leave it** decision. However, in this case, only *forward* recursion is natural: if we skip question i, we know that we need to go f_i questions *forward* in the test. It would be considerably more work to try to do backward recursion in this case: while examining a question, we would need to know of and go back to all of the questions from which we skipped ahead to that question.

The base case is of course that the maximum score for a subproblem that consists of just a single question is just the score for that question. Here is the recurrence: it is essentially just a direct translation of the preceding discussion. We decide if we take the points and skip ahead $f_i + 1$ questions $[p_i + S(i + f_i + 1)]$ or we just go onto the next question $[S(i+1)]$.

$$S[i] = \left\{ \begin{array}{cc} p_n & \text{if } i = n \\ \max\left\{p_i + S(i + f_i + 1), S(i+1)\right\} & \text{otherwise} \end{array} \right\}$$

Here is the Java implementation: we must use a for loop that goes from n down to 0. The run time is of course just $\mathcal{O}(n)$.

```
public int frustratingExam(int[] points, int[] skips) {
    int n = points.length;
    int[] S = new int[n + 1];
    S[n - 1] = points[n - 1];

    for (int i = n - 2; i >=0; i--) {
        S[i] = Math.max(points[i] + S[i + skips[i] + 1], S[i + 1]);
    }

    return S[0];
}
```

Summary of Examples 5-6.

- In both of the examples we used the **take it or leave it** approach when coming up with a recurrence.
- In example 5 we were able to approach the problem equivalently using either backward or forward recursion, while in example 6 forward recursion was the more practical choice.

  The notions of backward and forward recursion should not be confused with top down and bottom up implementation approaches. The former addresses how we write the recurrence itself (i.e do we use $i - 1$ or $i + 1$ in the recurrence) while the latter is concerned with the implementation details. All of the examples in this chapter are implemented using the top down approach. Recurrences in all of examples 1-4 were written using the natural backward recursion approach; however, some of them could also be implemented using forward recursion, but there isn't a need to do so.
- In example 5, we had to preprocess the data (i.e. sort the intervals by either start or end times) prior to proceeding with the actual dynamic programming work.

6.3 2D Dynamic Programming

Example 1: Knapsack Problem. Suppose that we have a knapsack that can hold a maximum weight W. We also have a set of items, each of which has a weight w_i and a value v_i. We would like to fit those items into the knapsack in a way such that the total value of items in the knapsack is maximized, yet their total weight stays within the knapsack's weight constraint W.

A sample listing of items is shown in the table on the right. In this particular example we will see that placing items 3 and 4 into the knapsack will give the maximum total value of 40, yet respecting the knapsack's weight constraint of $W = 11$.

Example. $W = 11$.

Item	Weight	Value
1	1	1
2	2	6
3	5	18
4	6	22
5	7	28

Just like in all other dynamic programming problems, we must come up with a way to relate the smaller subproblems to the larger problem. Can we draw hints from some of the other problems that we already looked at?

In the previous section, we already seen problems that involved sets of items. For example, in example 5 from the previous section, we solved the weighted interval scheduling problem by considering subproblems that focused on subsets with fewer intervals.

In the previous section, we also seen problems that involved discrete quantities. For example, in example 2 from the previous section, we solved the coin change problem by considering subproblems that focused on making change for smaller amounts of money (assuming all amounts were rounded to the dollar). In this example, total weight capacity of the knapsack is also a discrete quantity (assuming all weights are rounded to the kilogram).

So which approach would we use to solve this problem? Well, here we will actually need to use both approaches. Here the subproblems will focus on both, considering a smaller set of *items* and considering knapsacks with smaller total *weight* capacity. Using just one of the approaches will not simplify the problem enough for a dynamic programming approach to be useful. So, there will be two variables involved and hence this is 2D dynamic programming. Each subproblem will look at the optimal solution when considering only the first i items for a knapsack with weight capacity of w where $0 \leq w \leq W$.

How do we go from the smaller subproblems to the larger problem and vice versa? Well, we will just *combine* the strategies we used in the corresponding examples from the previous section.

When adding a new item to the knapsack we will use the **take it or leave it** approach as we have done with intervals in the weighted interval scheduling problem. In other words, we will take an item only if doing so will actually improve the total value of items in the knapsack. The tradeoff here is that by adding an item, we might need to make room first by removing some other item.

If we do choose to remove some other item to make room, we need to recurse back to the subproblem that is exactly the item's weight away from the current problem. This is similar to what we have done in the coin change problem. An item could be added only if the current knapsack's weight limit allows it.

The Recurrence Relation. As before, let i represent the current number of items and w represent the current weight of the knapsack.

- If there are no items, then the value has to be zero, this is the base case [first line].
- If we can't take the item due to weight limitations, value stays the same [second line].
- Otherwise, we check if we would be better off taking the new item and adding its value v_i or just ignoring the item and keeping the current value [last line].

$$\text{Value}(i, w) = \begin{cases} 0 & \text{if } i = 0 \\ \text{Value}(i-1, w) & \text{if } w_i > w \\ \max \left\{ \begin{array}{c} \text{Value}(i-1, w), \\ v_i + \text{Value}(i-1, w-w_i) \end{array} \right\} & \text{otherwise} \end{cases}$$

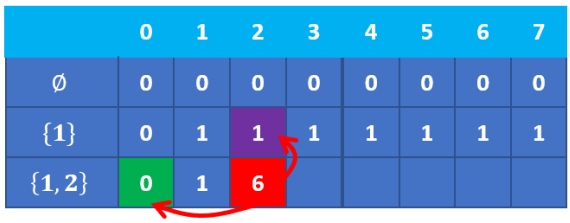
Sample DP Array. Below is a sample DP array for the numbers that were given at the start of this example. The array is filled in row-by-row, top to bottom, left to right.

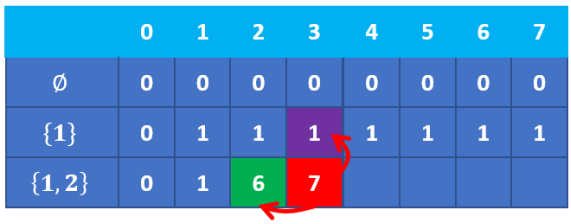
We start out by filling out the first row to be all zeros, which is the base case. We then fill out the subsequent rows using the recurrence relationship. The second row is all 1s whenever the weight limit is above zero, as then we can take the first item with the weight of 1. When the weight limit is zero, we of course aren't able to take any of the items.

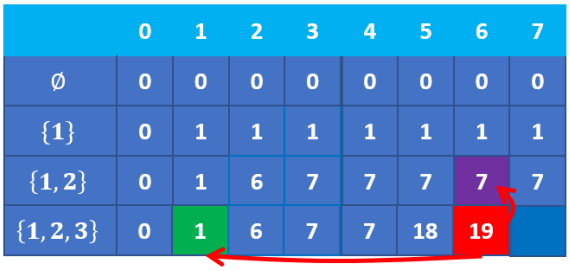
In the third row, as soon as we reach weight limit of 2, we have the choice between sticking with the just first item, or dropping it and taking the second item. We choose the latter as it results in a higher value. So there we pick $6 + \text{Value}(2, 2-2)$ [green] as opposed to $\text{Value}(1, 2)$ [purple]. Once we reach the weight limit of 3, we can take both of the items. There we pick $1 + \text{Value}(2, 3-1)$ [green] as opposed to $\text{Value}(1, 3)$ [purple], see diagrams.

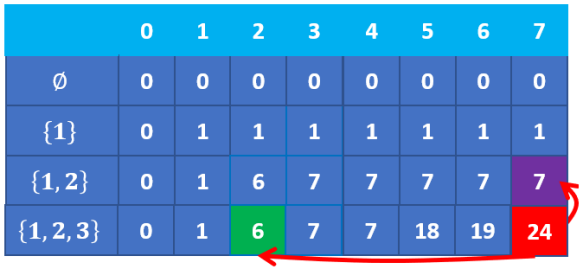
	0	1	2	3	4	5	6	7	8	9	10	11
$\emptyset$	0	0	0	0	0	0	0	0	0	0	0	0
$\{1\}$	0	1	1	1	1	1	1	1	1	1	1	1
$\{1,2\}$	0	1	6	7	7	7	7	7	7	7	7	7
$\{1,2,3\}$	0	1	6	7	7	18	19	24	25	25	25	25
$\{1,2,3,4\}$	0	1	6	7	7	18	22	24	28	29	29	40
$\{1,2,3,4,5\}$	0	1	6	7	7	18	22	28	29	34	35	40

Item	Weight	Value
1	1	1
2	2	6
3	5	18
4	6	22
5	7	28









In the fourth row, while $w < 5$ we cannot consider the 3rd item as its weight is $w = 5$, so we must just copy the above row (as per the second row of the recurrence). Once we reach $w = 5$, we can take the third item and nothing else. Once we reach $w = 6$, then we have a choice between $18 + \text{Value}(3, 6 - 5)$ [green] and $\text{Value}(2, 6)$ [purple]. Finally, when we reach $w = 7$, there the choice is between $18 + \text{Value}(3, 7 - 5)$ [green] and $\text{Value}(2, 7)$ [purple].

Run Time. The run time is $\Theta(nW)$ which corresponds to the size of the DP array and is also the consequence of the two nested for loops over n and W in the following implementation.

Implementation. Java implementation is shown below. Just like in the weighted interval scheduling problem, to determine which items amount to the optimal value of 40, we need to trace the algorithm back, starting from the cell (n, W) that holds the maximum value. We iterate over all items, starting from the last one. If we notice that the specific item has been added, then we need to determine relative to which weight limit and to which maximum value the addition took place. We do so by subtracting the weight and value of that item.

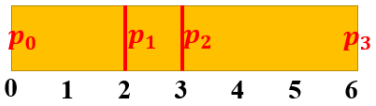
Greedy approach (on the value-weight ratio) produces an optimal solution only if we are able to take "fractions" of an item.

```
int knapSack(int W, int wt[], int val[], int n) {
    int i, w, result;
    int value[][] = new int[n + 1][W + 1];

    // determine optimal value
    for (i = 0; i <= n; i++) {
        for (w = 0; w <= W; w++) {
            if (i == 0 || w == 0)
                value[i][w] = 0;
            else if (wt[i - 1] <= w)
                value[i][w] = max(val[i - 1] + value[i - 1][w - wt[i - 1]],
                                  value[i - 1][w]);
            else
                value[i][w] = value[i - 1][w];
        }
    }

    w = W;
    result = value[n][W];

    // backtrack
    for (i = n; i > 0 && result > 0; i--) {
        if (result == value[i - 1][w])
            continue;
        else {
            // detected a jump, add the item to some list of results or print the values
            System.out.println(wt[i - 1] + " : " + val[i - 1] + " : " + i);
            result -= val[i - 1];
            w -= wt[i - 1];
        }
    }
}
```

Example 2: Plank Cutting. This problem appeared as a tutorial problem in 2018S2 and is also available here: `https://courses.csail.mit.edu/6.006/fall11/rec/rec24.pdf`.

Suppose that we have a plank of wood that is L meters long and needs to be cut at locations labeled $p_1, p_2, \ldots, p_n$. Those locations correspond to distances of $d_1, d_2, \ldots, d_n$ meters from the start of the plank. Let p_0 with $d_0 = 0$ represent the start of the plank and p_{n+1} with $d_{n+1} = L$ represent the end of the plank. To cut the plank, the lumber yard will charge price equal to the length of the plank being cut. The goal is to determine the minimum cost of cutting the plank at all of the specified points.

For example, consider the 6 meter long plank shown below. This plank needs to be cut at p_1 corresponding to a 2 meter point of the plank and at point p_2 corresponding to a 3 meter point of the plank.



Let's introduce some notation here to make further discussion a bit simpler. Let $c(i, j)$ be the cost of cutting the plank that stretches from point p_i to point p_j. Since the cost is equal to the length of the plank, then $c(i, j) = d_j - d_j$. For example, in the above diagram:

- $c(0, 2) = d_2 - d_0 = 3 - 0 = 3$ and
- $c(1, 3) = d_3 - d_1 = 6 - 2 = 4$

What are the subproblems here? Of course the subproblems would be concerned with cutting smaller planks of wood. So, how do we combine those subproblems into a larger problem? Well to pay for the cost of cutting the larger plank of wood, we need to pay for:

- the initial cost of cutting the larger plank itself into two smaller pieces, and
- the "recursive" costs of cutting all of the resulting smaller pieces until they no longer need to be cut.

So, starting from the entire plank, we can scan through all the possible locations where the plank could be cut, recursively calculate the associated costs and finally take the minimum of those costs. This is exactly what is written in the last line of the following recurrence. The sum there corresponds to the costs outlined in the bullet points above.

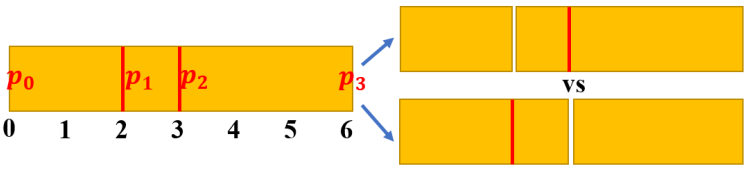
What are the base cases? Well it would cost \$0 to cut a plank of zero length, this is the case where $i = j$. Moreover, it would also cost \$0 to cut a non-zero length plank that doesn't need to be cut (i.e. when there are no points p_i internal to the plank), this is the case where we have $i = j - 1$. Those correspond to the first line in the following recurrence.

Finally, the way notation is set up in this problem, cases where $j > i$ do not make sense. We can either set the cost of those to ∞ or just ignore them, as the recurrence relation will never get to those. In fact, the way the last line is set up with the condition $i < k < j$ on the minimum, the recurrence relation will never really get to the $i = j$ base case either.

$$c(i,j) = \begin{cases} 0 & \text{if } i = j \\ & \text{or } i = j - 1 \\ \infty & \text{if } i > j \\ \min_{i<k<j} \{c(i,k) + c(k,j) + (d_j - d_i)\} & \text{otherwise} \end{cases}$$

Sample DP Array. Below is a sample array that corresponds to the plank draw on the previous page [i is down, j is across].

	0	1	2	3
0	0	0	3	9
1	∞	0	0	4
2	∞	∞	0	0
3	∞	∞	∞	0



The specific calculations are:

- $c(0,2) = c(0,1) + c(1,2) + (d_2 - d_0) = 0 + 0 + (3-0) = 3$
- $c(1,3) = c(1,2) + c(2,3) + (d_3 - d_1) = 0 + 0 + (6-2) = 4$
- $c(0,3) = \min \left\{ \begin{array}{l} c(0,1) + c(1,3) + (d_3 - d_0), \\ c(0,2) + c(2,3) + (d_3 - d_0) \end{array} \right\} = \min \left\{ \begin{array}{l} 0+4+6, \\ 3+0+6 \end{array} \right\} = 9$

Essentially, we start off by having to pay \$6 to cut the original plank, as it is 6m long. There is no way around that. However, we do have a choice as to where to make that first cut. If we first cut the plank at point p_1, we then deal with cutting a plank that is 4m long. Otherwise, if we first cut the plank at point p_2, we then deal with cutting a plank a that is just 3m long. The latter gives the minimal cost.

Run Time. Here we have to fill out an $n \times n$ table using two nested for loops. Moreover, filling out most of the cells involves taking a minimum of up to n values. Overall, there are three nested for loops in the implementation below. Therefore, the overall run time is $\mathcal{O}(n^3)$.

Implementation. The implementation is a bit tricky since the dynamic programming array must be filled in *diagonally*. Entry $(0, n)$ stores the optimal value.

```
int logCutting(int[] d) {
    int n = d.length;
    int[][] costs = new int[n][n];
    for (int[] row: costs)
        Arrays.fill(row, Integer.MAX_VALUE); // initialize everything to infinity

    for (int i = 0; i < n - 1; i++) { // fill-in the two zero diagonals
        costs[i][i] = 0;
        costs[i][i + 1] = 0;
    }
    costs[n - 1][n - 1] = 0;

    int noOfDiagonals = n - 2;
    for (int c = 0; c < noOfDiagonals; c++) { // all other diagonals
        for (int i = 0, j = c + 2; i < noOfDiagonals - c; i++, j++) {
            for (int k = i + 1; k < j; k++)
                costs[i][j] = Math.min(costs[i][j], // rolling minimum
                        costs[i][k] + costs[k][j] + (d[j] - d[i]));
        }
    }
    return costs[0][n - 1];
}
```

Example 3: Sequence Alignment. Suppose that we would like to have a concrete measure of by how much two strings differ from each other. This is something that would be useful when comparing words during spell check or measuring similarity of organisms by comparing the respective genomes.

For example, consider the three words on the right, the middle one is misspelled. Which one is it closer to, the top one or the bottom one? It is off from the top one by just a single letter (`e` vs `i`) and is just missing the letters `ve` when compared against the bottom one.

```
alternately
alternatily
alternatively
```

We say there is a *mismatch* when two letters are swapped, as is the case between the top and middle words. Alternatively (no pun intended), we can line up the middle and bottom words by introducing *gaps*, see the alignment shown on the right.

```
alternati--ly
alternatively
```

Such an alignment of two words could be formally seen as a matching of positions of the letters that make up the words. Suppose the first word is n letters long and denote the positions of the letters in the first word by $\{1, 2, \ldots, i, \ldots, n\}$. Likewise, suppose the second word is m letters long and denote the positions of the letters in the second word by $\{1, 2, \ldots, j, \ldots, m\}$. Recall that a *matching* M is the set of ordered pairs such that each item occurs in at most one pair. An *alignment* is a matching with no crossing pairs: for any two pairs $(i, j) \in M$ and $(i', j') \in M$ whenever $i < i'$ then $j < j'$.

In the above example, the alignment of "alternatily" and "alternatively" is given by the matching $\{(1, 1), (2, 2), \ldots, (9, 9), (10, 12), (11, 13)\}$. In other words, the first 9 letters line up exactly and then we have two gaps denoted by $(10, 12)$ and $(11, 13)$.

We measure the quality of alignment by defining mismatch penalties α_{x_i, y_j} for a mismatch between letters x_i and y_j and by defining a gap penalty given by δ. Of course $\alpha_{x_i, x_i} = 0$. We seek an alignment with the minimal total penalty.

The subproblems in this problem of course work with shorter strings. To expand a smaller subproblem, we add one letter at a time. In doing so, we take the minimum of three possibilities:

- align the words as is and take a potential mismatch penalty (the penalty could still be zero if the letters are identical)
- introduce a gap in the first word and take the gap penalty
- introduce a gap in the second word and take the gap penalty

The dynamic programming recurrence that computes the minimal mismatch cost is then as follows.

$$\mathrm{OPT}(i, w) = \begin{cases} i \cdot \delta & \text{if } j = 0 \\ j \cdot \delta & \text{if } i = 0 \\ \min \left\{ \begin{array}{c} \mathrm{OPT}(i-1, j-1) + \alpha_{x_i y_j}, \\ \mathrm{OPT}(i-1, j) + \delta, \\ \mathrm{OPT}(i, j-1) + \delta \end{array} \right\} & \text{otherwise} \end{cases}$$

The base cases are $i \cdot \delta$ if $j = 0$ and $j \cdot \delta$ if $i = 0$. In other words, if one of the words is empty, the only way to align the other word would be to use a sufficient number of gaps.

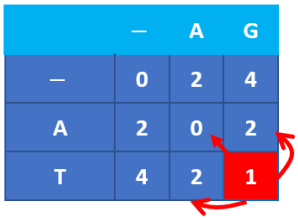
EXAMPLE. What is the best alignment between DNA strings given by `AGTAGT` and `ATCACT`? Here the penalties are: $\alpha_{A,C} = 1,\ \alpha_{A,G} = 3,\ \alpha_{A,T} = 2,\ \alpha_{C,G} = 2,\ \alpha_{C,T} = 3,\ \alpha_{G,T} = 1,\ \delta = 2$.

	-	A	G	T	A	G	T
-	0	2	4	6	8	10	12
A	2	0	2	4	6	8	10
T	4	2	1	2	4	6	8
C	6	4	3	4	3	5	7
A	8	6	5	5	4	6	7
C	10	8	7	7	6	6	8
T	12	10	9	7	8	7	6

For example, when we are filling in the cell (2,2) in zero-based indexing, we choose between the following:

- $0 + \alpha_{G,T} = 0 + 1 = 1$ (mismatch)
- $2 + \delta = 2 + 2 = 4$ (gap in the first word)
- $2 + \delta = 2 + 2 = 4$ (gap in the second word)

The minimum is those is of course 1.



In this problem, since we always go back just one step in the recurrence, it is quite easy to trace out the optimal alignment. Starting from the bottom-right corner, we determine which of the three possible options to arrive at that cell resulted in the actual minimum value stored in the cell. We then move back that way.

There could be ties in the different options of arriving at a given cell, so the optimal alignment need not be unique.

In this particular case, a possible trace out of the optimal alignment is highlighted in the above table in blue, the actual alignment is given on the right. Diagonal movements are mismatches, horizontal movements are gaps in the first word, vertical movements are gaps in the second word.

```
A-TCACT
AGT-AGT
```

RUN TIME. Since we are filling out an $n \times m$ table, the run time will be $\Theta(nm)$.

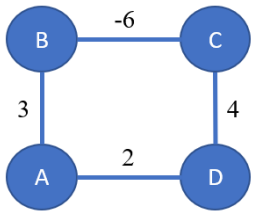
IMPLEMENTATION. Java implementation is provided on the following page. When filling out the table, we refer back to just the row immediately above (this was also the case in the knapsack problem). Therefore, we don't really need to store the entire table in memory, we can get away by just storing a $2 \times m$ table. This is sort of like the 2D version of the two-pointer approach of implementing the Fibonacci sequence (approach no. 4 of section 6.1).

LeetCode problem no. 1143 is very much like this problem.

```java
public int sequenceAlignment(String s1, String s2) {
    int n = s1.length();
    int m = s2.length();
    int delta = 2;
    int[][] opt = new int[2][m + 1];
    int[][] mismatches = {{0, 1, 3, 2},
                          {1, 0, 2, 3},
                          {3, 2, 0, 1},
                          {2, 3, 1, 0}};
    HashMap<Character, Integer> charMaps = new HashMap<Character, Integer>() {
        { put('A', 0); put('C', 1); put('G', 2); put('T', 3); }
    };

    // Here tricks 0 ^ 1 = 1 and 1 ^ 1 = 0 are used to keep flipping which of the
    // rows of the array we are referring to
    for (int i = 0, k = 0; i < n + 1; i++, k ^= 1) {
        for (int j = 0; j < m + 1; j++) {
            if (j == 0)
                opt[k][j] = delta * i;
            else if (i == 0)
                opt[k][j] = delta * j;
            else {
                int x = charMaps.get(s1.charAt(i - 1));
                int y = charMaps.get(s2.charAt(j - 1));
                opt[k][j] = Math.min(Math.min(
                        opt[k ^ 1][j - 1] + mismatches[x][y],
                        opt[k ^ 1][j] + delta),
                        opt[k][j - 1] + delta);
            }
        }
    }

    return opt[n % 2][m];
}
```

Example 4: Bellman-Ford Algorithm. Recall that Dijkstra's algorithm could fail to determine the shortest path if there are edges with negative weights.

In particular, in the example on the right, the shortest path between A and D given by Dijkstra's algorithm is A-D with a weight of 2; however, the actual shortest path is A-B-C-D with a weight of 1.



The problem with Dijkstra's algorithm is that it is greedy and looks at the *fastest* way to get to certain vertices. For example, it sees that it can get from A to D in just one hop with weight of 2 and accepts that result. However, had Dijkstra's algorithm considered *all* possible ways to get from A to D, including the one that is three hops long via B and C, it would have found a path with a weight of 1.

The Bellman-Ford algorithm addresses this issue by using dynamic programming with subproblems considering paths that use *any* number of hops to get to the destination vertex.

Therefore, the current maximum number of hops is the first variable in the table. Since we are looking for paths from any starting vertex to the destination vertex, vertex ids is the second variable in the table. *This algorithm will fail if the graph contains negative cycles.*

When deciding if we should increase the number of hops in the given path, we consider the tradeoff between doing nothing or adding an edge whose weight makes the path shorter.

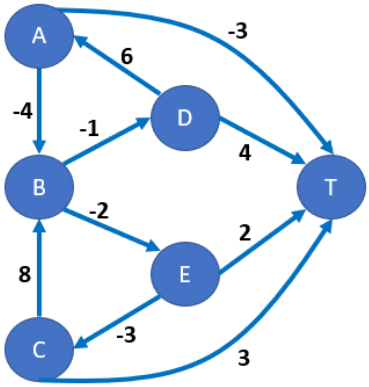
The base cases are:

- it is impossible to get to the destination vertex in one hop, unless we are at that vertex already (hence we assign infinity for weight in those cases – first line of the recurrence)
- it takes zero weight to get to the destination vertex if we are already there (second line of the recurrence)

$$\text{OPT}(i,v) = \begin{cases} \infty & \text{if } i = 0 \\ 0 & \text{if } v = T \\ \min \left\{ \begin{array}{c} \text{OPT}(i-1,v), \\ \min_{w \in V} \{\text{OPT}(i-1,w) + l_{vw}\} \end{array} \right\} & \text{otherwise} \end{cases}$$

The following example is from figure 6.23 on pg. 294 of the textbook. The last entry in each row is the weight of the shortest path from the given vertex to vertex T. For example:

- we can get from D to T with a weight of 4 if we consider just single-hop paths
- we can get from D to T with a weight of 0 if we consider paths that are five-hops long (the particular path in this case is D-A-B-E-C-T).



	0	1	2	3	4	5
T	0	0	0	0	0	0
A	∞	-3	-3	-4	-6	-6
B	∞	∞	0	-2	-2	-2
C	∞	3	3	3	3	3
D	∞	4	3	3	2	0
E	∞	2	0	0	0	0

Entry (4,A): the minimum of all those below is -6

- do nothing: -4 [taken from entry (3,A)]
- try to add edge A-T: $0 + (-3) = -3$ [the edge weight is -3]
- try to add edge A-B: $-2 + (-4) = -6$ [the edge weight is -4]
- try to add edge A-C: $3 + \infty = \infty$ [the edge doesn't exist]
- try to add edge A-D: $3 + \infty = \infty$ [the edge doesn't exist]
- try to add edge A-E: $0 + \infty = \infty$ [the edge doesn't exist]

Run Time. Naive analysis of run time is $\mathcal{O}(n^3)$ given that we need to fill out an $n \times n$ table and take minimum over n elements at each step. If we account for the number of edges actually incident to each vertex and use the sum of degrees formula from section 3.1, then the run time is $\mathcal{O}(nm)$.

7 Linear Time Sorting

All sorting algorithms studied in CPSC 221 are *comparison based* sorting algorithms. It could be rigorously shown that any comparison based sorting algorithm cannot achieve a run time that is better than $\mathcal{O}(n \log n)$. Furthermore, a sorting algorithm cannot achieve a run time that is better than $\mathcal{O}(n \log n)$ unless we can make some assumptions about the given dataset. The following three algorithms all have a run time of $\mathcal{O}(n)$; however, they all require some assumption to be made about the given dataset.

7.1 Counting Sort

Counting sort could be used to sort an array of n natural numbers, where each natural number is in the range from 0 to $k, k \in \mathbb{N}$.

Counting sort implementation uses three arrays, pseudo-code is shown on the next page:

- an input array `A` of size n (1-based indexing)
- an output array `B` of size n (1-based indexing)
- an intermediate array of "counts" `C` of size k (0-based indexing)

Step 1. All entries of `C`, the array of counts, are initialized to zero.

Step 2. We count how many occurrences of each integer from 0 to k we have in the input array `A` and store those counts into the indices corresponding to those integers in array `C`.

Step 3. The counts stored in array `C` are transformed into cumulative counts. This way we know how many elements, in the sorted array, can, at most, precede the given element.

Step 4. The cumulative counts are now applied to place the elements into the output array `B`. A given element is placed into array `B` at the index of its cumulative count, as we know that is *how far down* it must be located in the sorted array, to leave enough room for the elements that must precede it. The cumulative count of that element is then decreased by one, so that this property remains valid throughout.

As the algorithm will need to traverse an array `C` that is of size k, so the assumption we must make is that k is rather small in comparison to the size of the input array n. If that is the case, then $k \in \Theta(n)$ and counting sort runs in linear time. However, if that is not the case, counting sort could be very inefficient. For example, counting sort would be very inefficient if the input array is $\{500, 300, 100\}$; as here $k = 500 \gg 3 = n$ and $k \in \mathcal{O}(n^6)$.

Example. Suppose the input array is

3	2	2	3	2	0

- the array of "counts" is

0	**1**	**2**	**3**
1	0	3	2

we counted a single occurrence of 0, no 1s, three occurrences of 2, etc.

- the array of "cumulative counts" is

0	**1**	**2**	**3**
1	0	4	6

the only element that precedes 0 is it itself, 6 elements can precede a 3

- once we start building the array `B`, we look up in `C` where we should place the last element of `A`; the last element happens to be 0 and we place it in position 1

  the next element is 2 and we place it in position 4, decrementing that count to 3; so that when we encounter the next 2, we will place it into position 3, etc.

- the output array is

0	2	2	2	3	3

```
CountingSort(A,B,k) {
    for (i = 0 to k)      { set C[i]     := 0                }   // Step 1
    for (j = 1 to n)      { set C[A[j]]  := C[A[j]]++        }   // Step 2
    for (i = 1 to k)      { set C[i]     := C[i] + C[i-1]    }   // Step 3

    for (j = n down to 1) {                                      // Step 4
        set B[C[A[j]]]  := A[j]
        set C[A[j]]     := C[A[j]]--
    }
}
```

Recall that a sorting algorithm is called *stable* if it preserves the order of identical elements, as they are supplied in the input array. In particular, if $a_i = a_j$ and $i < j$, then a_i will be placed before a_j in the output array.

Proof that Counting Sort is Stable. Suppose that `A[i]` is identical to `A[j]` and $i < j$. In step 4, working from the back of array `A`, `A[j]` is first placed into the output array `B` at the index `C[A[j]]`.

Then, prior to placing `A[i]` into the output array `B`, the cumulative count `C[A[j]]` is decremented, so we will be guaranteed to place `A[i]` into `B` at a lower index, as required. □

7.2 Radix Sort

Radix sort sorts d-digit natural numbers, using some *stable* sorting algorithm to sort the numbers by one digit at a time, starting from the lowest significant digit first. Stability is critical, as once we have sorted the numbers by the lowest significant digit, their respective order with respect to *that* digit is preserved as we move onto the next significant digit. Therefore, sorting by each digit just once is sufficient.

Of course, radix sort need not be limited to just d-digit natural numbers. The simplest extension is to consider d-digit decimals as those could be easily converted to whole numbers via multiplication and transformed back via division. We could also use radix sort to sort arrays of objects by several properties, one property at a time. Once we sort an array of objects by the most important property, we could proceed to break ties using the other properties, without having to worry if the order w.r.t. the first property is still preserved.

Example.

329		720		720		329
457		355		329		355
657		436		436		436
839	$\Longrightarrow$	457	$\Longrightarrow$	839	$\Longrightarrow$	457
436		657		355		657
720		329		457		720
355		839		657		839

```
RadixSort(A,d) {
    for (i = 0 to d) {
        use a stable sorting algorithm to sort array A on digit i
    }
}
```

The run time of radix sort is $\mathcal{O}(dk)$ where d is the number of digits and k is the magnitude of the largest digit. Usually, d is quite small and $k \leq 9$ so we can claim that radix sort run time is linear (d small is the assumption we must make to ensure we get linear run time). If d becomes very large, then radix sort becomes rather inefficient. For example, radix sort would be inefficient if the input array was $\{129452343, 142434034, 123789456\}$.

Typically counting sort is used as the underlying stable sorting algorithm to sort on each digit in radix sort. In practice, many believe that radix sort is among the most efficient sorting algorithms.

Proof That Radix Sort Returns The Correct Result. There is not too much to show. Proceed by induction on d, the number of digits in each of the numbers being sorted.

Base Case. If $d = 1$, then radix sort will sort everything in a single pass, using some other stable sorting algorithm (e.g. counting sort) which is known to be correct.

Inductive Hypothesis. Fix $d > 1$ and suppose that radix sort returns the correct result for $d - 1$ digit numbers.

Inductive Step. We must show that radix sort returns the correct result for d digit numbers. By the inductive hypothesis, at this point in time, the numbers will be correctly sorted by their $d - 1$ least significant digits.

Suppose that we come across two numbers x and y with digits x_d and y_d in the d-th digit position. There are three possibilities to consider:

1) $x_d < y_d$: the algorithm will place x before y, while preserving stability, which is what we want.
2) $x_d > y_d$: the algorithm will place y before x, while preserving stability, which is what we want.
3) $x_d = y_d$: the algorithm will leave x and y in place, preserving the relative order of the elements based on their $d - 1$ digits, which is what we want.

Therefore, the algorithm sorts d digit numbers correctly. □

7.3 Bucket Sort

In bucket sort we sort via the following steps:

1) Divide the input array `A` into n equal buckets.
2) Sort each bucket using insertion sort.
3) Concatenate the results, into an array `B`, while preserving the order of buckets.

Bucket sort is efficient if the input could be distributed uniformly among all buckets (as is shown in the example on the following page). If that is the case, then there will be few entries per bucket and insertion sort will run in constant time, allowing for the entire algorithm to finish in linear time:

$$\underbrace{\Theta(n)}_{\text{split into buckets}} + \left(\underbrace{n}_{\text{no. of buckets}} \times \underbrace{\mathcal{O}(1)}_{\text{sort each bucket}} \right) \in \mathcal{O}(n)$$

However, if the input could not be uniformly distributed among all buckets, then bucket sort will be pushing a run time of $\mathcal{O}(n^3)$. Essentially, all entities would be clustered into one or two buckets and we will be running a $\mathcal{O}(n^2)$ insertion sort on those buckets with an additional $\Theta(n)$ useless overhead for bucketing.

Rigorous proofs of bucket sort run time use properties of distributions and expectations from probability and are quite complex.

Example. Suppose the input array is $\{78, 17, 39, 26, 72, 94, 21, 12, 23, 68\}$. The buckets will be:

Bucket [0-9]	Bucket [10-19]	Bucket [20-29]	Bucket [30-39]	Bucket [40-49]
Empty	12, 17	21, 23, 26	39	Empty

Bucket [50-59]	Bucket [60-69]	Bucket [70-79]	Bucket [80-89]	Bucket [90-99]
Empty	68	72, 78	Empty	94

So in this case the numbers are uniformly distributed, we have at most 3 elements per bucket and bucket sort is a good sorting algorithm to use on this dataset.

```
BucketSort(A) {
    set n := length(A)
    initialize an array of k lists B

    for (i = 1 to n) {
        insert A[i] into list B[A[i]]
    }

    for (i = 0 to k-1) {
        sort list B[i] using insertion sort
    }

    concatenate the lists B[0], B[1], ..., B[k-1] together in order
}
```

Proof That Bucket Sort Returns The Correct Result. Once again there is actually not too much to prove. The proof is really about formalizing the argument.

Suppose that we have two elements `A[i]` and `A[j]` in the input array `A` and WLOG suppose `A[i]` is the smaller one.

Once we apply bucket sort to `A`, one of the following two things will happen:

1) Elements `A[i]` and `A[j]` will be placed into the *same* bucket. Then they will be ordered correctly when insertion sort runs on that bucket.
2) Elements `A[i]` and `A[j]` will be placed into the *different* buckets. Then they will be ordered correctly once we concatenate buckets at the end of the algorithm.

So in either case we get the correct result and therefore bucket sort overall is correct. □

8 NP Completeness

8.1 Terminology and Theory

A *decision problem* is a problem that could be posed as a yes/no question on a set of input values. For example, consider the *subset sum problem*: given a an integer k and a set of n integers $V = \{v_1, v_2, \ldots, v_n\}$, is there a subset $U \subseteq V$ such that $\sum_{u_i \in U} u_i = k$?

An input to a decision problem will be represented as a finite binary string encoding s with length of $|s|$. The problem itself will be represented by X where X is the set of all binary input strings s on which the answer is "yes".

A *deterministic* algorithm is the one which, given a particular input, will always produce the same output and will follow the same sequence of steps to arrive at the solution. A *non-deterministic* algorithm on the other hand, given a particular input, can exhibit different behaviors and lead to different outputs.

There is always a brute-force way to find a solution to any decision problem. However, many of those brute-force approaches are computationally prohibitive. Ideally, we would like to work with algorithms that run in polynomial time as anything beyond polynomial time becomes computationally prohibitive. So let **P** denote the set of decision problems for which a *known deterministic* polynomial time algorithm exists.

A broader set of problems is **NP**, which stands for *non-deterministic polynomial time.* NP is the set of problems for which we do not necessarily know if a deterministic polynomial time algorithm exists. If a decision problem is in **NP** all that we do know is that if we are given a solution, we can check in polynomial time if that solution is correct. For example, for the subset sum problem mentioned in the opening paragraph, we do not know if there is a non-brute force way to determine which integers sum to the fixed integer k, there might be. What we do know is that given a candidate list of integers, it is easy to check if they do sum to k by simply computing their sum.

From the respective definitions, it is sort of intuitive that $\mathbf{P} \subseteq \mathbf{NP}$ and this is rigorously shown further down. Also, see the Venn diagram of all inclusions and summary of all definitions further down. Of course the famous million dollar (literally) question is: does $\mathbf{NP} = \mathbf{P}$? In other words, if we know how to verify in polynomial time that a given solution to a certain problem is correct, do we then immediately know there must be a polynomial time algorithm that produces the solution itself. Many believe that the answer is no; however, it hasn't been shown.

Efficient Certifiers. Before we can show $\mathbf{P} \subseteq \mathbf{NP}$ and establish other definitions, we have to look at the technical definition for the process of checking the correctness of a given solution. An algorithm B is said to be an *efficient certifier* for a problem X if:

- $B(s, t)$ is a polynomial time algorithm that takes an input string s and a proposed output string t (called a certificate) as inputs, and
- there is a polynomial function p such that for any input string s:

$$s \in X \iff \exists t \text{ such that } |t| \leq p(|s|) \text{ and } B(s, t) = \text{yes}$$

Essentially what the above definition is saying is that we can say that $s \in X$ [recall that X

is the set of inputs for which the answer is yes] if and only if we can produce a corresponding solution t and use the polynomial time algorithm B to check s against t. Moreover, the length of t must be of polynomial order of the length of s.

Going back to the subset sum example, let $k = 0$ and $V = \{-5, 1, 2, 3, 7\}$. Then the input string s is the set V encoded as a binary string, while the set $U = \{-5, 2, 3\}$ is encoded as the certificate t. It should be the case that $|t| \leq |s|$ so the polynomial order length condition is satisfied. Then B would be the algorithm that checks if all elements of U are in V and if U sums up to k. Having verified that, we can conclude that $s \in X$ in this case.

The process used by the efficient certifier to check a certificate gives rise to the brute-force algorithm for the problem: just try all possible certificates until one works. To solve the subset sum problem using brute-force, we would just check the sums of all possible subsets until we find one that sums to k. Checking sums is exactly what the efficient certifier does with the certificate it is given. However, the efficient certifier in no way hints at any non-brute force ways to solve the problem and provides no hints as to if $P = NP$ is true.

Proposition 1. $\mathbf{P} \subseteq \mathbf{NP}$.

Proof. Let $X \in \mathbf{P}$ be an arbitrary problem for which exists a polynomial time algorithm A. Recall that X is a decision problem; therefore, given any input, A returns either "yes" or "no" as output. We want to show that $X \in \mathbf{NP}$. In order to do so, we must come up with an efficient certifier B for X.

Let $s \in X$ be an input string and let t be a proposed certificate for s such that $|t| \leq p(|s|)$. Let B be the algorithm that, given the input string s, just returns the output of A for s. In other words, just set $B(s, t) = A(s)$, in particular B just throws away t. Then B:

- clearly runs in polynomial time, since A runs in polynomial time
- if $s \in X$, then $A(s)$ is yes, and therefore $B(s, t)$ is also yes as required
- conversely, by contrapositive, if $s \notin X$, then $A(s)$ is not yes, and $B(s, t)$ is also not yes as required

Therefore, B is an efficient certifier and hence $\mathbf{P} \subseteq \mathbf{NP}$ as required. □

Reductions. While it hasn't been shown that NP ≠ P there are certainly many problems in NP that are, at least at this point, not in P. The subset sum problem is such a problem (we don't know if there is a polynomial time algorithm that solves it). We would like to group those problems somehow and just say that all of those problems are all "hard". We will group those problems via polynomial time reductions and we will call the group **NP-Complete** problems. We know they're in NP because they can be reduced from a problem that's already in NP

We say that Y is *polynomial time reducible* to X and write $Y \leq X$ if arbitrary instances of Y could be solved by a polynomial number of standard computational steps and a polynomial number of calls to an algorithm that solves X. Essentially $Y \leq X$ if we can come up with a reduction from Y to X that runs in polynomial time.

We then say that a decision problem X is **NP-Complete** if

1. $X \in$ NP
2. $Y \leq X, \forall Y \in$ NP

In other words, for a decision problem X to be NP-Complete, it must be in NP and it must be that every other NP problem is polynomial time reducible to X (i.e. X is powerful enough to solve all of those problems). As polynomial time reductions are *transitive* (see proposition 2), in practice we need to show that just ***one*** problem in NP is reducible to X. In the following sections we will establish several classical NP-Complete problems that will be used as "go to" problems when it comes to showing that a certain problem is NP-Complete.

So what are the benefits of establishing the notion of NP-Completeness? Well, we can now talk about the class of problems that are *at least as hard* as each other in polynomial time sense. In particular, if we have established that a certain problem is NP-Complete, then we can show some other problem is also NP-Complete by finding a polynomial time reduction from the former to the latter. This is because the additional polynomial time spent on the reduction won't make the overall time any worse. Grouping all NP-Complete problems this way, if it happens that NP = P and a polynomial time algorithm is discovered that solves any one of the problems, there would a polynomial time algorithm for all such problems (see proposition 3). Moreover, if we suspect and show that a certain problem is NP-Complete, then it is a strong indication that the problem can't be solved in polynomial time.

Cook-Levin Theorem (8.13 in the textbook) established that any arbitrary NP problem could be reduced to the simple problem of satisfying a logic circuit; therefore, the complexity class NP-Complete is far from empty and indeed its definition makes sense.

Proposition 2. If $Z \leq Y$ and $Y \leq X$, then $Z \leq X$.

Proof. Roughly speaking, just compose the two polynomial time reductions, composition of two polynomials is a polynomial, so the overall reduction is of polynomial time.

More formally, since we know that $Z \leq Y$, then, by definition, arbitrary instances of Z could be solved by a polynomial number of standard computational steps and a polynomial number of calls to an algorithm that solves Y. Likewise for $Y \leq X$. So let R_i be the run time function for problem i and let a, b, c, d be polynomials. We can then write $R_Z = a + b(R_Y)$ and $R_Y = c + d(R_X)$.

Then $R_Z = a + b(R_Y) = a + b\,(c + d(R_X))$. This is a composition of polynomials, which is a polynomial, so $Z \leq X$ as required. □

Proposition 3. Suppose X is a NP-Complete problem. Then X is solvable in polynomial time iff P = NP.

Proof. We will address the $\Leftarrow$ direction first. Suppose that P = NP. Since $X \in$ NP, then $X \in$ P and X is solvable in polynomial time, as required.

Now consider the converse $\Rightarrow$ direction. Suppose that X is solvable in polynomial time. We already established that P $\subseteq$ NP, so it remains to show that the reverse inclusion NP $\subseteq$ P holds. Let $Y \in$ NP be an arbitrary problem, we want to show that $Y \in$ P. Indeed, since both $Y \in$ NP and $X \in$ NP, then $Y \leq X$ by definition of NP. Moreover, $X \in$ P by assumption. Therefore, Y can be solved in polynomial time as well, as required. □

Other Complexity Classes. There are many, many other complexity classes beyond P, NP and NP-Complete. Complexity classes are studied in greater depth in CPSC 421. Here we will briefly discuss the NP-Hard and complement classes.

NP-Hard class is obtained from the NP-Complete class by dropping the requirement of the existence of an efficient certifier. In other words, a problem is NP-Hard if not only there isn't necessarily a polynomial time algorithm to solve it, but there isn't necessarily even a polynomial time algorithm to verify if the solution is correct (but there might be one).

Some examples of NP-Hard problems are complex optimization problems. Not only is it difficult to find the optimum solution, it is just as difficult to decide if the solution is indeed the optimum one. For example, suppose production cost is a complex function of raw material, labor and transportation costs. A mathematical algorithm may output *some* minimum cost; however, it may be hard to prove there isn't an even smaller minimum.

A classical example of a NP-Hard problem is the halting problem: "given a program and its input, will it halt (yes/no)?" The halting problem is not computable; therefore, it is not possible to computationally verify a given solution.

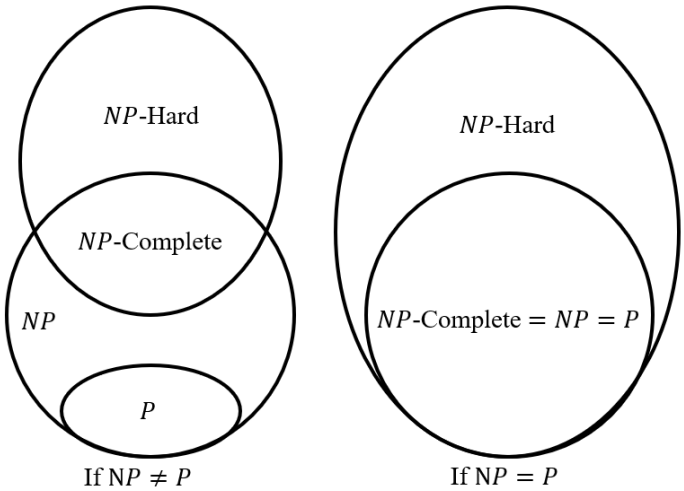
For every problem, there is also the complement problem which focuses on the "no" side of the problem. For example, in the subset sum problem we would want to determine if there isn't a subset that sums to the target k. Then, the definitions of complexity classes **co-NP** and **co-NP-complete** are similar to the definitions of NP and NP-complete classes.

SUMMARY. The following Venn diagrams are from Wikipedia.

- P = {all problems for which a *known* polynomial algorithm exists}
- NP = {all problems for which an *efficient certifier* exists}
- NP-Hard = $\left\{ \begin{array}{c} \text{class of decision problems to which all NP problems} \\ \text{could be reduced to in polynomial time} \end{array} \right\}$
- NP-Complete = NP $\bigcap$ NP-Hard

Another way to write the definition of NP-Hard is $\{X : Y \leq X,\ \forall Y \in \text{NP}\}$.

Another way to write the definition of NP-Complete is $\{X : X \in \text{NP}$ **and** $Y \leq X,\ \forall Y \in \text{NP}\}$.



	P	NP	NP-Complete	NP-Hard
Solvable in Polynomial Time	✓			
Solution Verifiable in Polynomial Time	✓	✓	✓	
Reduces any NP Problem in Polynomial Time			✓	✓

NOTES. The above table is from `https://cs.stackexchange.com/questions/9556/what-is-the-definition-of-p-np-np-complete-and-np-hard`.

Even if it was shown that P = NP, the answer might still not produce the magic algorithm that efficiently solves all NP-Complete problems. For instance, even if the run time of the discovered algorithm is polynomial, the algorithm could still be prohibitively slow. Recall example 6 from section 2.2, there surely $10^5 n^{30} \leq 2^n$ whenever $n \geq 257$, but for such an n the run time is 2^{257} and is on the order of the number of atoms in the universe. Factoring in all of the additional polynomial time reductions slows down the run time even further.

8.2 First Examples

SHOWING A PROBLEM X IS NP-COMPLETE. Based on the definition of a NP-Complete:

1. Show that there exists an efficient certifier for X [the easy part].
2. Pick a known NP-Complete problem Y and specify an algorithm that could be used to reduce Y to X in *polynomial time* [the hardest part]. The reduction is from the known hard problem to the new one. Do not do this step backwards!
3. Prove that the above reduction is correct [the harder part].

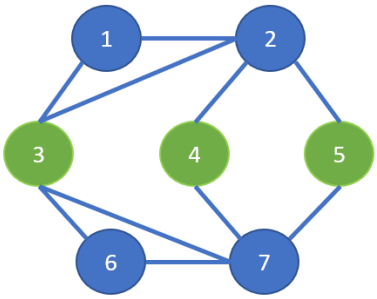
Problem Definitions.

1. ***Independent Set.*** For a graph $G = (V, E)$, a subset of vertices $S \subseteq V$ is *independent* if no two vertices in S that are joined by an edge. Given a graph G and $k \in \mathbb{N}$, does G contain an independent set of size k or larger?
2. ***Vertex Cover.*** For a graph $G = (V, E)$, a subset of vertices $S \subseteq V$ is a *vertex cover* if every edge $e \in E$ has at least one of its endpoints in S. Given a graph G and $k \in \mathbb{N}$, does G contain a vertex cover of size k or smaller?
3. ***Set Packing.*** Given an n-element set U, a collection of subsets $\{S_1, S_2, \ldots, S_m\} \subset U$ and a number $k \in \mathbb{N}$, does there exists a collection of at least k of those sets with the property that no two of them intersect?
4. ***Set Cover.*** Given an n-element set U, a collection of subsets $\{S_1, S_2, \ldots, S_m\} \subset U$ and a number $k \in \mathbb{N}$, does there exists a collection of at most k of those sets whose union is equal to U?
5. ***SAT (Satisfiability).***
   - Let $X = \{x_1, \ldots, x_n\}$ be a set of n Boolean variables [i.e. each x_i is 0 or 1].
   - A term t_i over X is either one of the variables x_i or its negation $\overline{x}_i$.
   - A clause C_j of length l is a disjunction of distinct terms: $C_j = t_1 \vee t_2 \vee \cdots \vee t_l$.
   - A collection of clauses is the conjuction $C_1 \wedge C_2 \wedge \cdots \wedge C_k$.

   A truth assignment is some assignment of values of 0 or 1 to each $x_i \in X$. In other words, a truth assignment is a function $f : X \to [0, 1]$. The collection of clauses $C_1 \wedge C_2 \wedge \cdots \wedge C_k$ is *satisfiable* if there exists a truth assignment that evaluates the collection to 1.

   A problem is called l-SAT if the length of all of its clauses C_j is exactly l.

For example, for the graph on the right, we have an independent set of size of at most 3: the set $\{3, 4, 5\}$. In particular, there are no edges between the green vertices. Of course, we could have smaller independent sets, in particular, each vertex is an independent set on its own of size 1.



We also need a vertex cover of size of at least 4: the cover $\{1, 2, 6, 7\}$. In particular, every edge touches at least one of the blue vertices. Of course, we could have larger vertex covers, in particular, the entire set of vertices V would always be a vertex cover.

The following is an instance of 2-SAT. To satisfy it, we must have at least one of the variables set to 1 in each of the clauses. The assignment $x_1 = 0, x_2 = 0$ and $x_3 = 0$ satisfies this problem, but the assignment $x_1 = 1, x_2 = 1, x_3 = 1$ does not.

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

There are polynomial time algorithms to solve a 2-SAT instance, so 2-SAT is in **P**. However, there isn't a known polynomial time algorithm to solve a 3-SAT instance. We will show that it is possible to reduce 3-SAT to any of the above other problems in polynomial time.

Independent Set $\leq$ Vertex Cover.

Step 1. Vertex Cover $\in$ NP. To verify that a given set S of size k is a vertex cover, take the set of graph's edges E and run through all vertices in S. For each vertex, delete all adjacent edges from E. When done, if E is empty, then S is a vertex cover. If we use adjacency lists, run time is $\mathcal{O}(m) \subseteq \mathcal{O}(n^2)$, which is polynomial, so vertex cover is in NP.

Step 2. The Reduction. The reduction will be based on the following proposition that states independent set and vertex cover problems are "complements" of each other. In particular, for a graph with n vertices, an independent set S of size k and a vertex cover $V - S$ of size $n - k$ we have $V \cup (V - S) = V$. We also seen this in the above diagram.

Therefore, if we already have an algorithm that determines if a graph has a vertex cover of at most of a certain size, then to determine if a graph G contains an independent set of at least size k, we can just run the algorithm that checks if G contains a vertex cover of at most size $n - k$ and take its result.

Step 3. Proposition. Set S is an independent set iff its complement $V - S$ is a vertex cover.

Proof. First, we will consider the $\Rightarrow$ direction. Proceed by contradiction and suppose that S is an independent set, yet $V - S$ is not a vertex cover. So, there must be an edge $e = (u, v)$ such that neither of the endpoints are in $V - S$: $u \notin V - S$ and $v \notin V - S$. Therefore, it must be that both $u \in S$ and $v \in S$. But then S is not an independent set. $\lightning$

Next, consider the $\Leftarrow$ direction. Once again, proceed by contradiction. Suppose $V - S$ is a vertex cover, yet S is not an independent set. So, there must be an edge $e = (u, v)$ such that both $u \in S$ and $v \in S$. But then $u \notin V - S$ and $v \notin V - S$, so e is an edge with no endpoint in $V - S$ and therefore $V - S$ is not a vertex cover. $\lightning$ $\square$

Set and vertex cover problems are examples of *covering* problems (with the set cover being a more general version of vertex cover), while set packing and independent set problems are examples of *packing* problems.

Vertex Cover $\leq$ Set Cover.

Step 1. Set Cover $\in$ NP. Given a candidate collection $\mathcal{C}$ of subsets of U, we need to check that the union of those subsets is the set U. We can do so by running through all elements of all of the subsets and then checking off in a Boolean array the elements of U that we have encountered. If we have encountered all elements of U, then union of $\mathcal{C}$ is U.

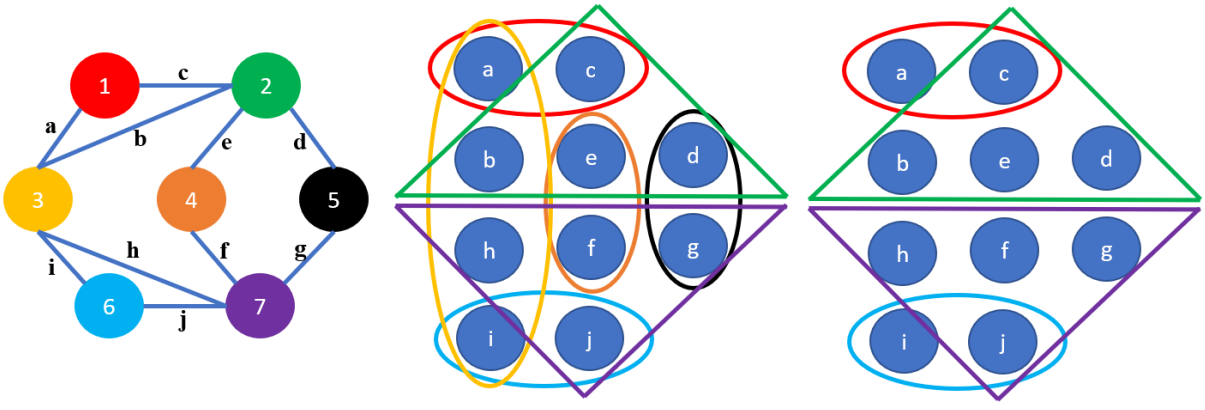
Recall that n is the size of U. Then there would be at most n subsets in $\mathcal{C}$, as otherwise, some of the sets would be either empty or redundant. Each of the subsets in $\mathcal{C}$ would also contain at most n elements. Therefore, the run time for the check is $\mathcal{O}(n^2)$ which is polynomial.

Step 2. The Reduction. Suppose that we have an algorithm that determines if a set has a cover of at most of a certain size. How can we use this algorithm to determine if a graph has a vertex cover of at most of a given size k? Well both of the algorithms have the word "cover" in them. The set cover algorithm determines if all of the set's *elements* could be covered by the certain number of *subsets*, while the vertex cover algorithm essentially determines if all of the graph's *edges* could be covered by the certain number of *vertices*.

So, it is elements = edges and subsets = vertices. Therefore, given a graph $G = (V, E)$, for each edge create a corresponding element in the set U and group edges that share a common vertex into subsets. To do so, traverse the graph's adjacency lists once in polynomial time of $\mathcal{O}(n + m)$ and create the corresponding elements and subsets. Then, run the set cover algorithm to determine if there is a set cover of at most size k and take its result.

More formally, define $U = E$. Then, for each vertex i, define $S_i \subset U$ to consist of all edges in G that are incident to i.

The following diagram color codes the relationships between subsets and vertices. This is the same example as the one on the previous page and as the one on pg. 457 of the textbook. The rightmost figure is the solution: here the number of subsets we need is reduced just to the bare minimum of 4, yet all elements are still covered. Those subsets correspond to the vertices that made up the vertex cover in the diagram on the previous page.



Step 3. Proposition. Let U and S_i be defined as above. Then U can be covered with at most k sets from $\{S_1, S_2, \ldots, S_m\}$ iff $G = (V, E)$ has a vertex cover of size of at most k.

PROOF. Consider the $\Rightarrow$ direction first. Suppose that there is a set cover of at most size k for the set U. Let $\{S_{i_1}, S_{i_2}, \ldots, S_{i_l}\}$ for some $l \leq k$ be the solution to the set cover problem. In other words, every element of $u \in U$ is in one of S_{i_j}.

Therefore, by construction of S_{i_j}, every edge $e \in E$ of G is incident to some vertex i_j. So every edge has at least one endpoint in the subset of vertices $I = \{i_1, i_2, \ldots, i_l\} \subseteq V$. Since the size of I is $l \leq k$, then G has a vertex cover of size of at most k, as required.

Now consider the converse, the $\Leftarrow$ direction. Suppose that graph $G = (E, V)$ has a vertex cover of size of at most k and suppose $I = \{i_1, i_2, \ldots, i_l\} \subseteq V$ for some $l \leq k$ is this vertex cover. We can proceed via an argument that is essentially the same as before: every edge $e \in E$ is incident to at least one vertex $i_j \in I$, so by construction, every element $u \in U$ is in at least one of the subsets S_{i_j}. Therefore subsets $\{S_{i_1}, S_{i_2}, \ldots, S_{i_l}\}$ form a set cover of size $l \leq k$ as required. □

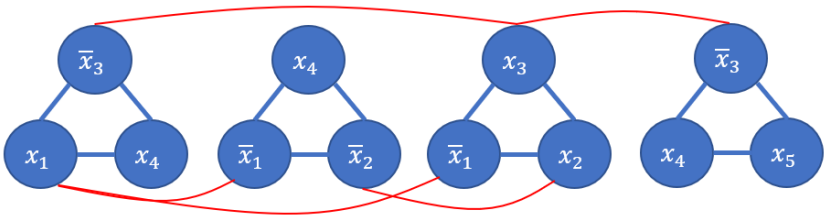
3-SAT $\leq$ Independent Set.

Graph Representation of 3-SAT.

- For each clause draw a triangle, the vertices of which represent the Boolean variables involved in that clause.
- Add additional edges to connect vertices x_i to vertices $\overline{x}_i$ for the same i.

Then, to find a satisfying assignment, from each triangle we have to pick one vertex to set to 1 in a way that avoids conflicts represented by those additional red edges (i.e. a conflict would be both x_i and $\overline{x}_i$ being true at the same time for the same i).

Example: $(x_1 \vee \overline{x}_3 \vee x_4) \wedge (\overline{x}_1 \vee \overline{x}_2 \vee x_4) \wedge (\overline{x}_1 \vee x_2 \vee x_3) \wedge (\overline{x}_3 \vee x_4 \vee x_5)$



Step 1. Independent Set $\in$ NP. To verify a set is independent, run through all vertices in that set and first check that all of the vertices in the are indeed vertices in the graph. Then check that there are no edges between any two vertices in the given set. Using adjacency lists, the run time is $\mathcal{O}(n + m)$, which is polynomial, so independent set is in NP.

Step 2. The Reduction. Suppose that we have an algorithm that determines if a graph G has an independent set of at least of a certain size. How can we use this algorithm to determine if a given 3-SAT instance with k clauses is satisfiable?

Well, we can follow the steps outlined above to construct a graph G that represents the instance. Then, we can run the algorithm to determine if G contains an independent set of size of at least k and take its result.

The resulting graph has $3k$ vertices, so it would take at most $\mathcal{O}(k^2)$ to construct it.

Step 3. Proposition. A 3-SAT instance with k clauses is satisfiable iff the corresponding graph G has an independent set of at least of size k.

PROOF. Consider the $\Rightarrow$ direction first. Suppose we have a 3-SAT instance with k clauses that is satisfiable. Let G be the corresponding graph and define G_i to be the subgraphs that correspond to each clause (i.e. let G_i's be the triangles). Since the 3-SAT instance is satisfiable, then there is at least one vertex in each G_i that is marked as 1. Let S be the set that formed by taking exactly one vertex from each of the subgraphs. Since there are k clauses, then there are k subgraphs and the size of S is k. Remains to show that S is indeed an independent set.

Claim. Set S is independent.

Proof. Proceed by contradiction and suppose that S is not independent. Then there must be two vertices u and v in S such that there is an edge e between them. Since u and v, by construction, must belong to different subgraphs G_i, then edge e must run between vertices that represent variables that are negations of each other in the 3-SAT instance (i.e. e must be one of those "conflict" edges). In other words, for some variable x_i we would have both $x_i = 1$ and $\overline{x}_i = 1$, a contradiction. $\lightning$

Now consider the converse, the $\Leftarrow$ direction. Suppose that G has an independent set S of at least of size k. Let $C = C_1 \vee C_2 \vee \cdots \vee C_k$ be the corresponding 3-SAT instance. As before, let G_i be the subgraphs that correspond to each clause C_i (i.e. let G_i's be the triangles). There are k such subgraphs.

First, the size of S cannot be larger then k, as then S would need to include more than one vertex from some G_i and would no longer be an independent set (as G_is are triangles and any two vertices in a triangle are connected by an edge forfeiting any hope of independence). So S contains exactly one vertex from each of the G_is.

Assign values to each x_i as follows:

- if the vertex corresponding to x_i appears in S, set $x_i = 1$
- if the vertex corresponding to x_i *doesn't* appears in S, set $x_i = 0$

Claim. The above assignment satisfies the 3-SAT instance C.

Proof. Consider a clause C_i that corresponds to some G_i, we want to show it evaluates to 1, there are three cases to consider. Recall that vertices of G_i correspond to terms of C_i.

CASE 1. A vertex of the form x_i from G_i is the one that appears in S. Then $x_i = 1$ by the above definition and C_i evaluates to 1.

CASE 2. A vertex of the form $\overline{x}_i$ from G_i is the one that appears in S and x_i never appears in S. Then $x_i = 0$ or $\overline{x}_i = 1$ by the above definition and C_i evaluates to 1.

CASE 3. A vertex of the form $\overline{x}_i$ from G_i is the one that appears in S and x_i also appears in S. This is impossible as then there is an edge from $\overline{x}_i$ to x_i and S is not independent. $\square$

Overall, we have shown that

$$\text{3-SAT} \leq \text{Independent Set} \leq \text{Vertex Cover} \leq \text{Set Cover}$$

8.3 Graph Coloring

Problem Definition. A graph $G = (V, E)$ is said to be k-colorable if the endpoints of any edge (u, v) could be colored using different colors when there is total of k available colors.

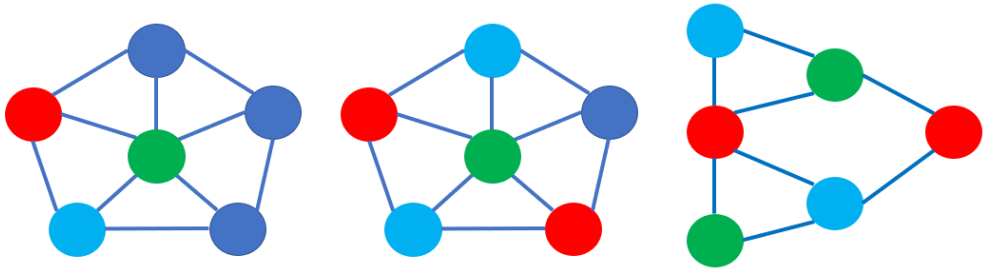
More formally a k-coloring of a graph is a function $f : V \to \{1, 2, \ldots, k\}$ such that for any edge $(u, v) \in E$ we have $f(u) \neq f(v)$.

Given a graph G and $k \in \mathbb{N}$, does G have a k-coloring?

Examples. The following graph cannot be 3-colored. To begin, we must color the three vertices of any triangle using the three different colors (leftmost diagram). This fixes the color of the two out of the three vertices in the adjacent triangles, so we must assign the remaining color to the remaining vertex in those triangles (diagram in the center). However, then we are unable to color the last vertex in this graph.

The graph on the right can be 3-colored but cannot be 2-colored. We seen in section 3.3 that a graph is 2-colorable if and only if it is bipartite. There we also saw that the BFS graph traversal could be applied to check if a graph is bipartite and hence we could use BFS to determine if a graph is 2-colorable. Therefore, the 2-coloring problem is in P.

However, we will see that 3-SAT is reducible to 3-coloring in polynomial time and therefore 3-coloring is an NP-Complete problem and is a significantly harder problem than 2-coloring. There is a common theme that the "2-versions" of many problems are solvable in polynomial time (another example is 2-SAT), yet as soon as we consider the "3-version" or a greater version of that same problem, the problem is NP-Complete.



A graph with more edges requires more colors. In general, a complete or a fully connected graph with n vertices requires n colors to be colored.

The graph coloring problem emerged from the need to color neighboring countries different colors when making maps. There are many other applications of graph coloring beyond maps, one of them is the register allocation algorithm that is used by compilers to assign variables to registers. Each CPU has a fixed number of registers (typically 16) to store the most immediate variables the CPU is working with at any given point in time. The machine code of the program is represented using a graph where vertices are the variables and edges represent conflicts by connecting variables that are active at same points in time. The compiler then determines the best way to 16-color the resulting graph, in other words, the best way to assign variables to 16 registers so that there are no conflicts. If there are more than 16 active variables at any point in time, remaining ones must be "spilled over" into memory that is beyond registers. While 16-coloring is of course an NP-Complete problem, compilers use best-effort approximations that produce solutions in reasonable amounts of time (see CPSC 411).

3-SAT $\leq$ 3-Coloring.

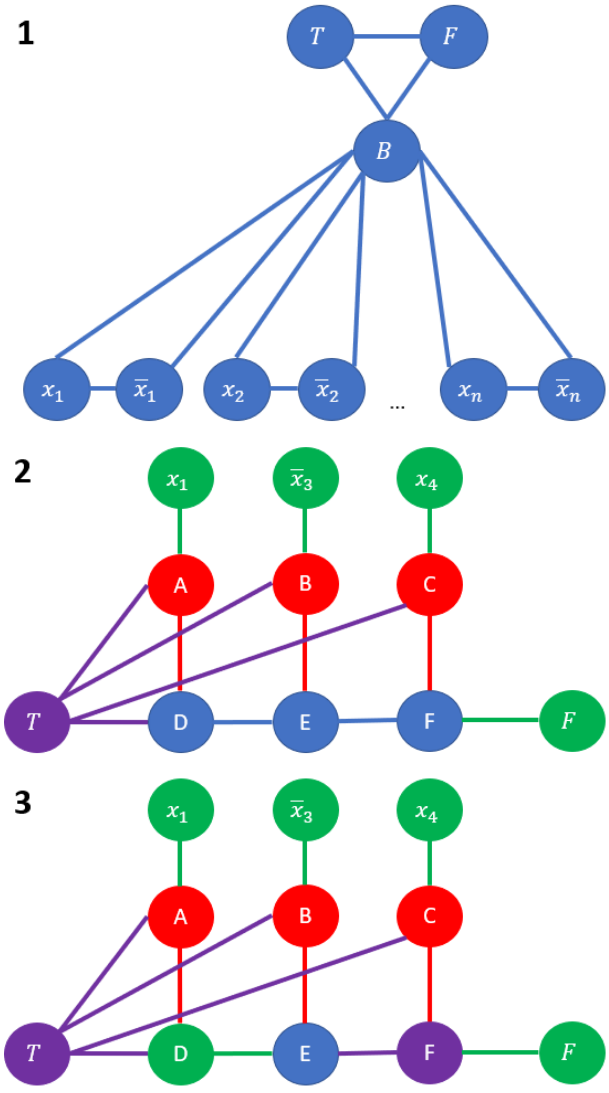
Step 1. 3-Coloring $\in$ NP. Suppose that we are given a graph G and a hash map that stores the color assignment of all vertices. We need to check that endpoints of every edge of G are assigned different colors. This could be done in $\mathcal{O}(m) \subseteq \mathcal{O}(n^2)$ time given either the adjacency lists or in the adjacency matrix implementations by traversing all of the edges and looking up colors of the endpoints. Therefore 3-Coloring is in NP.

Step 2. The Reduction. Suppose that we have an algorithm that determines if a given graph is 3-colorable? How can we use this algorithm to determine if a certain instance of the 3-SAT problem is satisfiable?

Well, the goal is to construct a graph that encodes the given 3-SAT instance in a way that the constructed graph is 3-colorable iff the 3-SAT instance is satisfiable.

The graph will have vertices that correspond to the Boolean variables x_i or their negations $\overline{x}_i$. One of the colors in the resulting coloring will signify *true*, meaning that if the vertex corresponding to some variable is colored that color, then the variable should be set to 1. Likewise, we will have another color signify *false*, meaning that the corresponding variable should be set to 0. The remaining color will be called *base*.

We will do much of the work by employing *triangles* to force certain color assignments to take place as all vertices of a triangle must have different colors. Here are the key parts:

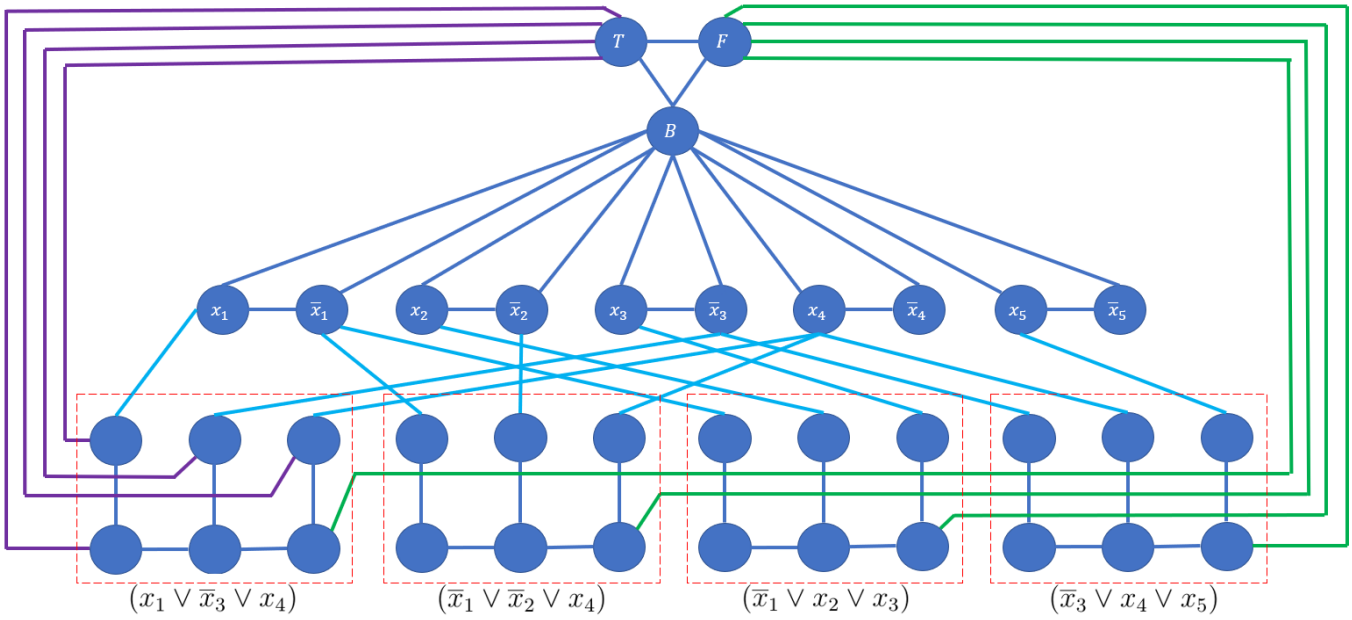


1. The top most triangle in the diagram no. 1 on the right will consist of just three vertices to fix the true, false and base colors to be different colors.
2. The next set of triangles will connect Boolean variables x_i to their negations $\overline{x}_i$ and then connect both to the base color. This way we force the Boolean variables to take on either the true or false colors (i.e. they cannot take on the base color).
3. There will be a 6-vertex subgraph for each of the clauses of the 3-SAT instance. The vertices representing the Boolean variables that are involved in that instance will be connected to the subgraph as is shown in diagram no. 2. The vertices that represent

true and false colors from diagram no. 1 will also be connected to this subgraph as is shown in same diagram. See the first part of the following proof to see why the connections must be made in this specific way.

The full graph is shown below for the following example (to reducing cluttering, here the vertex representing the true color is connected just to the first clause).

The example: $(x_1 \vee \overline{x}_3 \vee x_4) \wedge (\overline{x}_1 \vee \overline{x}_2 \vee x_4) \wedge (\overline{x}_1 \vee x_2 \vee x_3) \wedge (\overline{x}_3 \vee x_4 \vee x_5)$.



Step 3. Proposition. A 3-SAT instance $\mathcal{C}$ is satisfiable iff the corresponding graph G is 3-colorable.

Proof. Consider the $\Leftarrow$ direction first and proceed by contrapositive (i.e. we want to show that if the 3-SAT instance is not satisfiable then the corresponding graph G is not 3-colorable).

If the 3-SAT instance that is not satisfiable then there must be some clause C_i that doesn't evaluate to 1. Consider the subgraph G_i corresponding to this clause. Refer to diagrams labeled 2 and 3 above and let *false* be green, *true* be purple and *base* be red.

If C_i doesn't evaluate to 1, then all of its terms must be *false*, so color those green. Since *true* is connected to all of the following vertices (vertices A,B,C), then those must be colored *base*. Since *true* is also connected to D, and A is colored *base*, then D must be colored *false*. Since *false* is connected to F, and C is colored *base*, then F must be colored *true*. However, then E cannot be colored. So G cannot be colored as required.

As for the $\Rightarrow$ direction, suppose that the 3-SAT instance $\mathcal{C}$ is satisfiable. Then color vertices representing x_i the *true* color if $x_i = 1$ and the *false* color if $x_i = 0$. Color vertices representing $\overline{x}_i$ the remaining avaliable color (it would be the color opposite of the color assigned to x_i).

There are many cases to consider, but after due-diligent checking it can be shown that G can be 3-colored. In particular, if one of the Boolean variables is colored *true*, then one of A,B or C can be colored *base*, which gives the flexibility to color E this time. □

8.4 Additional Examples

Problem Definitions.

1. ***Subset Sum.*** Given a set of n integers $V = \{v_1, v_2, \ldots, v_n\}$, is there a subset $U \subseteq V$ such that $\sum_{u_i \in U} u_i = k$?
2. ***Set Partition.*** Given a set of n integers $V = \{v_1, v_2, \ldots, v_n\}$, can the elements of V be partitioned into two sets U and $V - U$ such that $\sum_{u_i \in U} u_i = \sum_{u_i \in V-U} u_i$?
3. ***Clique.*** For a graph $G = (V, E)$, a subset of vertices $S \subseteq V$ is a *clique* if every pair of vertices in V is joined by an edge. Given a graph G and $k \in \mathbb{N}$, does G contain a clique of size k or larger?
4. ***4-SAT.*** Same definition as 3-SAT, just the length of every clause is now 4.

Subset Sum $\leq$ Set Partition.

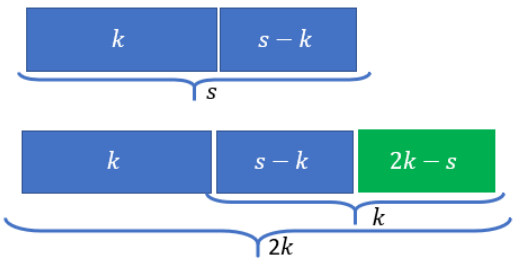
Step 1. Set Partition $\in$ NP. To verify that two given sets A and B are indeed the partition of U that we are looking for, we need to check the following:

- $A \cup B = U$
- $A \cap B = \emptyset$
- $\sum_{a_i \in A} a_i = \sum_{b_i \in B} b_i$

To check the first bullet point, we could add elements of A and B to a single set and then check that this set is equal to U. This would be done in $\mathcal{O}(n \log n)$ since the sets would need to be sorted to check equality. To check the second bullet point, we could place elements of A into a hash set and then iterate over B checking if any element of B is already in the hash set. Finally, we can compute the respective sums of A and B as we go. All of this takes $\mathcal{O}(n)$. Overall, the run time to do the checks is polynomial and set partition is in NP.

Step 2. The Reduction. Suppose that we have an algorithm that determines if set V can be partitioned into sets U and $V - U$ such that $\sum_{u_i \in U} u_i = \sum_{u_i \in V-U} u_i$. Given a $k \in \mathbb{Z}$, how can we use this algorithm to determine if there is a subset $W \subseteq V$ such that elements of W sum to k: $\sum_{w_i \in W} w_i = k$?

Both problems involve sums, we just need to find a way to translate one sum into the other! Suppose that the sum of all elements of V is some $s \in \mathbb{Z}$.



Then, since U and $V - U$ form a set partition, we get $\sum_{u_i \in U} u_i = \sum_{u_i \in V-U} u_i = s/2$. So the set partition problem gives us subsets that sum to $s/2$. However, we are not looking for subsets that sum to $s/2$. We are looking for a subset the sum of which is k.

How can we force the set partition problem to give us subsets that sum to k? Well, we would just need to get the sum of all elements of V to become $2k$. We can do so by adding a single number to V, the value of which is $2k - s$, as is illustrated in the above diagram.

So, given a set V, form a set $V' = V \cup \{2k - s\}$ and feed V' into the subset partition problem. Then examine the returned subsets, the sum of either one would be k. Take the one that doesn't contain the element $2k - s$, that is the subset we been looking for.

The run time to form the set V' is polynomial as we add just a single element to V.

Step 3. Proposition. Let $V = \{v_1, v_2, \ldots, v_n\}$ and $k \in \mathbb{Z}$. Define $\sum_{v_i \in V} v_i = s$ where $s \in \mathbb{Z}$ and let $V' = V \cup \{2k - s\}$.

Then there is a subset $W \subseteq V$ such that $\sum_{w_i \in W} w_i = k$ iff there is a partition of V': U and $V' - U$ such that $\sum_{u_i \in U} u_i = \sum_{u_i \in V' - U} u_i$.

Proof. Consider the $\Rightarrow$ direction first. Suppose that there is a subset $W \subseteq V$ such that $\sum_{w_i \in W} w_i = k$. Consider sets W and $V' - W$ [note that $W \subseteq V \subset V'$]. We need to show those sets are the partition of V' that we are looking for.

First observe that $W \cup (V' - W) = V'$ and $W \cap (V' - W) = \emptyset$ by definition of set complement. Then, since the sum of elements of V' is $s + (2k - s) = 2k$ and the sum of elements of W is k, it must be that the sum of elements of $V' - W$ is $2k - k = k$. Therefore $\sum_{w_i \in W} w_i = \sum_{w_i \in V' - W} w_i$ as required.

Consider the $\Leftarrow$ direction next. Suppose there is a partition of V': U and $V' - U$ such that $\sum_{u_i \in U} u_i = \sum_{u_i \in V' - U} u_i$. Let $\sum_{u_i \in U} u_i = x$ for some $x \in \mathbb{Z}$. Since the sum of elements of V' is $s + (2k - s) = 2k$ and U and $V' - U$ are disjoint, then

$$\sum_{u_i \in U} u_i + \sum_{u_i \in V' - U} u_i = \sum_{v_i \in V'} v_i \quad \longrightarrow \quad x + x = 2k \quad \longrightarrow \quad x = k$$

Therefore $\sum_{u_i \in U} u_i = \sum_{u_i \in V' - U} u_i = x = k$. Moreover, since U and $V' - U$ is a partition of V' and sets U and $V' - U$ are disjoint, then element $2k - s$ belongs just one of those sets. WLOG, suppose this set is $V' - U$. Then U is the subset of V we are looking for. $\square$

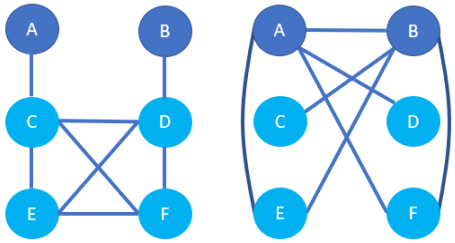
Independent Set $\leq$ Clique.

Step 1. Clique $\in$ NP. To verify that a set S is indeed a clique of size of at least k, we must first check that $|S| \geq k$. We then must verify that there is an edge between each pair of vertices in S. There are $\mathcal{O}(k^2)$ pairs of vertices in a clique of size k as a clique is a complete or a fully connected graph. Checking for an edge between two vertices takes $\mathcal{O}(n)$ at most. Since $k \leq n$ then overall run time is $\mathcal{O}(n^3)$ which is polynomial.

Step 2. The Reduction. Suppose that we have an algorithm that determines if a graph $G = (V, E)$ has a clique of size of at least k. How can we use this algorithm to determine if G has an independent set of size of at least k?

Well, both problems deal with edges of a graph, just in an opposite way. In the clique problem we do want to have all of the possible edges to connect all of the vertices. In the independent set problem, we want to have those same edges disappear to make the set of vertices independent. So, essentially what we want to work with is the *complement* of the graph G. A complement graph $\overline{G}$ of a graph G is the one in which two distinct vertices of $\overline{G}$ are connected if and only if they are *not* connected in G.

The diagram on the right shows a graph and its complement. In the first graph, the vertices in cyan form a clique of size 4 as there is an edge between any pair of those vertices. Then, in the second graph, it is precisely those same vertices that make up an independent set of size 4.



So if we are given a graph G, we would construct its complement $\overline{G}$, feed $\overline{G}$ to the clique problem and take its result.

Easiest way to construct $\overline{G}$ from G is to subtract its adjacency matrix from a matrix of all 1s and a diagonal of 0s (so that there are no self-edges). This can be done in $\mathcal{O}(n^2)$ time. For example:

$$\begin{bmatrix} 0 & 1 & 1 \\ 1 & 0 & 1 \\ 1 & 1 & 0 \end{bmatrix} - \begin{bmatrix} 0 & 1 & 1 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix} = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 1 \\ 1 & 0 & 0 \end{bmatrix}$$

Step 3. Proposition. A graph G has an independent set of size of at least k iff its complement $\overline{G}$ has a clique of size of at least k.

Proof. Consider the $\Rightarrow$ direction first. Suppose that G has an independent set of size of at least k. Let S be this independent set. By definition of an independent set, there isn't an edge between any two vertices in S. By definition of $\overline{G}$, there will be an edge between those two vertices in $\overline{G}$. Therefore S is a clique of size of at least k in G.

Consider the $\Leftarrow$ direction next. Suppose that $\overline{G}$ has a clique of size of at least k. Let S be this clique. By definition of a clique, there is an edge between any two vertices in S and therefore there will not be an edge between those two vertices in $\overline{\overline{G}} = G$. Therefore S is an independent set of size of at least k in G. □

3-SAT $\leq$ 4-SAT.

Step 1. 4-SAT $\in$ NP. Let $X = \{x_1, \ldots, x_n\}$. Suppose that we have an instance of 4-SAT that is written as $\mathcal{C} = C_1 \wedge C_2 \wedge \cdots \wedge C_i \wedge \cdots \wedge C_k$ where each clause is 4 terms long. To check that some given truth assignment of Boolean variables x_i satisfies $\mathcal{C}$ we need to check that each of the C_i's evaluates to 1. Checking each C_i involves taking 3 disjunctions. We then take k conjuctions and overall number of operations is $3k + k$. So the run time is $\mathcal{O}(k)$, which is polynomial.

Step 2. The Reduction. Suppose that we have an algorithm for solving an arbitrary instance of a 4-SAT problem. How could we use this algorithm to solve an instance of a 3-SAT problem?

Well to turn 3-SAT into 4-SAT we need to add an extra variable to each clause in a way that won't affect the truth assignment on the first 3 variables (i.e. so that once we have solved the 4-SAT instance we can ignore the additional variable and can still take the truth assignment of the other variables as our solution).

So, we need to add the extra variable in a way such that the algorithm that solves the 4-SAT instance won't get any additional help from that extra variable. To do so, we duplicate the clauses and add the additional variable *as is* to one clause and then its *negation* to the duplicate clause. This way, since all clauses must be satisfied, the 4-SAT algorithm won't be able to rely on the additional variable to satisfy the entire instance.

More formally, given the 3-SAT instance $\mathcal{C}$ on set X, define $X' = X \cup \{y\}$ and for every clause C_i define two clauses $C_{i1} = C_i \vee y$ and $C_{i2} = C_i \vee \overline{y}$. Then feed

$$\mathcal{C}' = C_{11} \wedge C_{12} \wedge C_{21} \wedge C_{22} \wedge \cdots \wedge C_{i1} \wedge C_{i2} \wedge \cdots \wedge C_{k1} \wedge C_{k2}$$

into the algorithm that solves a 4-SAT instance and take its result.

Since the reduction requires creation of $2k$ clauses, and creation of each clause takes constant time, the overall run time is $\mathcal{O}(k)$, which is polynomial.

Step 3. Proposition. 3-SAT instance $\mathcal{C}$ is satisfiable iff the constructed 4-SAT instance $\mathcal{C}'$ is satisfiable.

Proof. Consider the $\Rightarrow$ direction first. Suppose that the 3-SAT instance $\mathcal{C}$ is satisfiable. Then each of the clauses C_i evaluates to 1. Therefore, for each i, both $C_{i1} = C_i \vee y$ and $C_{i2} = C_i \vee \overline{y}$ evaluate to 1, regardless of the value of y. So all of clauses C_{i1} and C_{i2} evaluate to 1. Therefore the constructed 4-SAT instance $\mathcal{C}'$ is also satisfiable.

Consider the $\Leftarrow$ direction next. Suppose that the 4-SAT instance $\mathcal{C}'$ is satisfiable. Since the 4-SAT instance $\mathcal{C}'$ is satisfiable, then there is a truth assignment $f : X \to [0, 1]$ such that all of the clauses C_{i1} and C_{i2} evaluate to 1.

Claim. The truth assignment f satisfies the corresponding 3-SAT instance $\mathcal{C}$.

Proof. Proceed by contradiction and suppose that the truth assignment f doesn't satisfy the corresponding 3-SAT instance $\mathcal{C}$. Then there is at least one clause C_j that doesn't evaluate to 1 and evaluates to 0.

Consider the corresponding clauses $C_{j1} = C_j \vee y$ and $C_{j2} = C_j \vee \overline{y}$. There are two cases to consider.

Case 1. Suppose that $y = 0$, then since $C_j = 0$ we have $C_{j1} = 0 \vee 0 = 0$ and so C_{j1} doesn't evaluate to 1. Therefore truth assignment f doesn't satisfy the corresponding 4-SAT instance $\mathcal{C}'$, a contradiction. $\lightning$

Case 2. Suppose that $y = 1$, then since $C_j = 0$ we have $C_{j2} = 0 \vee \overline{1} = 0$ and so C_{j2} doesn't evaluate to 1. Therefore truth assignment f doesn't satisfy the corresponding 4-SAT instance $\mathcal{C}'$, a contradiction. $\lightning$

In both cases we derived a contradiction, therefore the original claim is true. $\square$

A Formula Sheet

Definitions of Big-$\mathcal{O}, \Omega, \Theta$ and little-o, ω.

- $g(n) \in \mathcal{O}(f(n))$ if $\exists c \in \mathbb{R}^+,\ \exists n_0 \in \mathbb{N},\ \forall n \in \mathbb{N},\ n \geq n_0 \Rightarrow g(n) \leq c \cdot f(n)$
- $g(n) \in \Omega(f(n))$ if $\exists d \in \mathbb{R}^+,\ \exists n_0 \in \mathbb{N},\ \forall n \in \mathbb{N},\ n \geq n_0 \Rightarrow g(n) \geq d \cdot f(n)$
- $g(n) \in \Theta(f(n))$ if $\exists c, d \in \mathbb{R}^+,\ \exists n_0 \in \mathbb{N},\ \forall n \in \mathbb{N},\ n \geq n_0 \Rightarrow d \cdot f(n) \leq g(n) \leq c \cdot f(n)$
- $g(n) \in o(f(n))$ if $\forall c \in \mathbb{R}^+,\ \exists n_0 \in \mathbb{N},\ \forall n \in \mathbb{N},\ n \geq n_0 \Rightarrow g(n) \leq c \cdot f(n)$
- $g(n) \in \omega(f(n))$ if $\forall d \in \mathbb{R}^+,\ \exists n_0 \in \mathbb{N},\ \forall n \in \mathbb{N},\ n \geq n_0 \Rightarrow g(n) \geq d \cdot f(n)$

Equivalently:

- $g(n) \in \Omega(f(n))$ iff $f(n) \in \mathcal{O}(g(n))$
- $g(n) \in \omega(f(n))$ iff $f(n) \in o(g(n))$
- $g(n) \in \Theta(f(n))$ iff $g(n) \in \mathcal{O}(f(n))$ and $g(n) \in \Omega(f(n))$
  iff $g(n) \in \mathcal{O}(f(n))$ and $f(n) \in \mathcal{O}(g(n))$

Relationships [oscillating functions are exceptions for the first two]:

- $o(f(n)) \approx \mathcal{O}(f(n)) - \Theta(f(n))$
- $\omega(f(n)) \approx \Omega(f(n)) - \Theta(f(n))$
- $f(n) \in o(g(n)) \Rightarrow f(n) \in \mathcal{O}(g(n))$
- $f(n) \in \omega(g(n)) \Rightarrow f(n) \in \Omega(g(n))$

From Fastest to Slowest.

Constant	$\mathcal{O}(1)$	
Log	$\mathcal{O}(\log n)$	
PolyLog	$\mathcal{O}((\log n)^k)$	$1 < k$
SubLinear	$\mathcal{O}(n^c)$	$0 < c < 1$
Linear	$\mathcal{O}(n)$	
LogLinear	$\mathcal{O}(n \log n)$	
SubQuadratic	$\mathcal{O}(n^d)$	$1 < d < 2$
Quadratic	$\mathcal{O}(n^2)$	
LogQuadratic	$\mathcal{O}(n^2 \log n)$	
Cubic	$\mathcal{O}(n^3)$	
Polynomial	$\mathcal{O}(n^a)$	
	$\mathcal{O}(n^b)$	$3 < a < b$
Exponential	$\mathcal{O}(\alpha^n)$	
	$\mathcal{O}(\beta^n)$	$1 < \alpha < \beta$
Factorial	$\mathcal{O}(n!)$	
Power	$\mathcal{O}(n^n)$	

Relationship with Limits and Inequality Analogies.

Value of Limit	Run time	Inequality Analogy
$\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$	$f(n) \in o(g(n))$	$<$
$\lim_{n\to\infty} \frac{f(n)}{g(n)} \neq \infty$	$f(n) \in \mathcal{O}(g(n))$	$\leq$
$\lim_{n\to\infty} \frac{f(n)}{g(n)} \neq 0, \infty$	$f(n) \in \Theta(g(n))$	$=$
$\lim_{n\to\infty} \frac{f(n)}{g(n)} \neq 0$	$f(n) \in \Omega(g(n))$	$\geq$
$\lim_{n\to\infty} \frac{f(n)}{g(n)} = \infty$	$f(n) \in \omega(g(n))$	$>$

Master Theorem.

THEOREM. Let $a \geq 1$ and $b > 1$ be constants, let $f(n) : \mathbb{N} \to \mathbb{R}^+$, and let

$$T(n) = a\, T\left(\tfrac{n}{b}\right) + f(n)$$

1. If $f(n) \in \mathcal{O}(n^{\log_b a - \epsilon})$ for some $\epsilon > 0$, then $T(n) \in \Theta(n^{\log_b a})$
2. If $f(n) \in \Theta(n^{\log_b a})$, then $T(n) \in \Theta(n^{\log_b a} \cdot \log n)$
3. If $f(n) \in \Omega(n^{\log_b a + \epsilon})$, for some $\epsilon > 0$, and if $af\left(\tfrac{n}{b}\right) \leq cf(n)$ for some $0 < c < 1$ and for all large enough n, then $T(n) \in \Theta(f(n))$

Exponents and Logarithms.

1. $a^x a^y = a^{x+y}$
2. $a^x / a^y = a^{x-y}$
3. $a^{x^y} = a^{xy}$
4. $a^0 = 1$
5. $a^{-n} = \frac{1}{a^n}$
6. $\log_a x + \log_a y = \log_a xy$
7. $\log_a x - \log_a y = \log_a x/y$
8. $\log_a x^y = y \log_a x$
9. $\log_a 1 = 0$
10. $a^{\log_a x} = x$
11. $\log_a b = \frac{\log_c b}{\log_c a}$
12. $\log_a b = \frac{1}{\log_b a}$
13. $a^{\log_c b} = b^{\log_c a}$ for all $a > 0, b > 0$
14. $a^{p/q} = (\sqrt[q]{a})^p$

Factorials.

1. $n! = n \cdot (n-1) \cdot (n-2) \cdots 3 \cdot 2 \cdot 1$
2. $n! = n \cdot (n-1)!$
3. $P(n,r) = \frac{n!}{(n-r)!}$
4. $C(n,r) = \binom{n}{r} = \frac{n!}{r!(n-r)!}$

Summations and Geometric Series.

$$\sum_{i=0}^{n-1} ar^i = a\left(\frac{1-r^n}{1-r}\right) \qquad \sum_{i=0}^{\infty} ar^i = \frac{a}{1-r} \text{ if } |r| < 1$$

$$\sum_{i=1}^{n} i = \frac{n(n+1)}{2} \qquad \sum_{i=1}^{n} i^2 = \frac{n(n+1)(n+2)}{6}$$

Run Times of Typical Graph Operations.

	Adjacency Matrix	Adjacency List
Insert Vertex	$\mathcal{O}(n)$	$\mathcal{O}(\deg(v))$
Remove Vertex	$\mathcal{O}(n)$	$\mathcal{O}(\deg(v))$
Insert Edge	$\mathcal{O}(1)$	$\mathcal{O}(1)$
Remove Edge	$\mathcal{O}(1)$	$\mathcal{O}(\deg(v))$
Vertices Adjacent?	$\mathcal{O}(1)$	$\mathcal{O}(\deg(v))$
Incident Edges	$\mathcal{O}(n)$	$\mathcal{O}(\deg(v))$

Run Times of Typical Graph Algorithms.

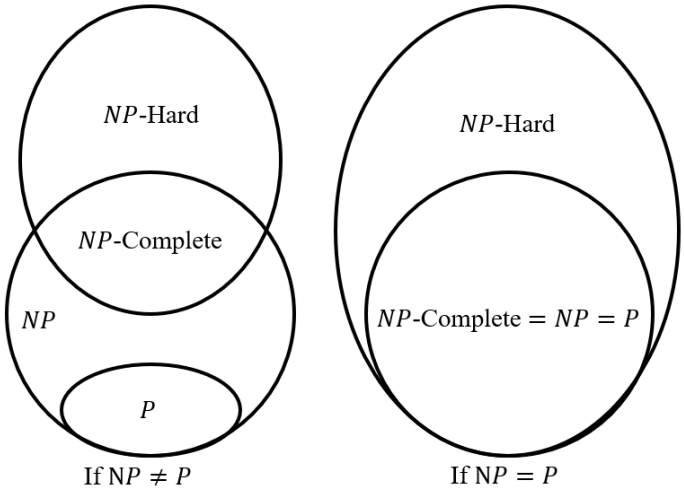
	Run Time	Data Structure
BFS	$\mathcal{O}(n+m)$	Queue
DFS	$\mathcal{O}(n+m)$	Stack or Recursion
Dijkstra's	$\mathcal{O}(m \log n)$	Priority Queue
Prim's	$\mathcal{O}(m \log n)$	Priority Queue
Kruskal's	$\mathcal{O}(m \log n)$	Disjoint Sets

NP Theory Summary.

- P = {all problems for which a *known* polynomial algorithm exists}
- NP = {all problems for which an *efficient certifier* exists}
- NP-Hard = $\left\{ \begin{array}{c} \text{class of decision problems to which all NP problems} \\ \text{could be reduced to in polynomial time} \end{array} \right\}$
- NP-Complete = NP $\bigcap$ NP-Hard

Another way to write the definition of NP-Hard is $\{X : Y \leq X,\ \forall Y \in \text{NP}\}$.

Another way to write the definition of NP-Complete is $\{X : X \in \text{NP}$ **and** $Y \leq X,\ \forall Y \in \text{NP}\}$.



	P	**NP**	**NP-Complete**	**NP-Hard**
Solvable in Polynomial Time	✓			
Solution Verifiable in Polynomial Time	✓	✓	✓	
Reduces any NP Problem in Polynomial Time			✓	✓

DP Recurrences.

Climbing Steps.

$$\begin{aligned}\text{Stairs}[1] &= 1\\ \text{Stairs}[2] &= 2\\ \text{Stairs}[n] &= \text{Stairs}[n-2] + \text{Stairs}[n-1]\end{aligned}$$

Coin Change.

$$\begin{aligned}\text{NoOfCoins}[0] &= 0\\ \text{NoOfCoins}[x] &= \min_{\substack{i \in \{0,\ldots,n-1\},\\ x - c_i \geq 0}} \{\text{NoOfCoins}[x - c_i]\} + 1\end{aligned}$$

Longest Increasing Subsequence.

$$L[i] = \begin{cases} \max_{\substack{1 \leq j < i,\\ A[j] < A[i]}} \{L[j]\} + 1 & \\ 1 & \text{otherwise} \end{cases}$$

Maximum Subarray.

$$S[i] = \begin{cases} A[0] & \text{if } i = 0 \\ \max\{S[i-1] + A[i], A[i]\} & \text{otherwise} \end{cases}$$

Weighted Interval Scheduling: Backward.

$$W[i] = \left\{ \begin{array}{cc} w_1 & \text{if } i = 1 \\ \max\{w_i + W(p(i)),\ W(i-1)\} & \text{otherwise} \end{array} \right\}$$

Weighted Interval Scheduling: Forward.

$$W[i] = \left\{ \begin{array}{cc} w_n & \text{if } i = n \\ \max\{w_i + W(p(i)),\ W(i+1)\} & \text{otherwise} \end{array} \right\}$$

Knapsack.

$$\text{Value}(i, w) = \begin{cases} 0 & \text{if } i = 0 \\ \text{Value}(i-1, w) & \text{if } w_i > w \\ \max\left\{ \begin{array}{c} \text{Value}(i-1, w), \\ v_i + \text{Value}(i-1, w - w_i) \end{array} \right\} & \text{otherwise} \end{cases}$$

Sequence Alignment.

$$\text{OPT}(i, w) = \begin{cases} i \cdot \delta & \text{if } j = 0 \\ j \cdot \delta & \text{if } i = 0 \\ \min\left\{ \begin{array}{c} \text{OPT}(i-1, j-1) + \alpha_{x_i y_j}, \\ \text{OPT}(i-1, j) + \delta, \\ \text{OPT}(i, j-1) + \delta \end{array} \right\} & \text{otherwise} \end{cases}$$

Bellman-Ford.

$$\text{OPT}(i, v) = \begin{cases} \infty & \text{if } i = 0 \\ 0 & \text{if } v = T \\ \min\left\{ \begin{array}{c} \text{OPT}(i-1, v), \\ \min_{w \in V} \{\text{OPT}(i-1, w) + l_{vw}\} \end{array} \right\} & \text{otherwise} \end{cases}$$

NP Complete Problems.

1. ***Independent Set.*** For a graph $G = (V, E)$, a subset of vertices $S \subseteq V$ is *independent* if no two vertices in S that are joined by an edge. Given a graph G and $k \in \mathbb{N}$, does G contain an independent set of size k or larger?

2. ***Vertex Cover.*** For a graph $G = (V, E)$, a subset of vertices $S \subseteq V$ is a *vertex cover* if every edge $e \in E$ has at least one of its endpoints in S. Given a graph G and $k \in \mathbb{N}$, does G contain a vertex cover of size k or smaller?

3. ***Set Packing.*** Given an n-element set U, a collection of subsets $\{S_1, S_2, \ldots, S_m\} \subset U$ and a number $k \in \mathbb{N}$, does there exists a collection of at least k of those sets with the property that no two of them intersect?

4. ***Set Cover.*** Given an n-element set U, a collection of subsets $\{S_1, S_2, \ldots, S_m\} \subset U$ and a number $k \in \mathbb{N}$, does there exists a collection of at most k of those sets whose union is equal to U?

5. ***Clique.*** For a graph $G = (V, E)$, a subset of vertices $S \subseteq V$ is a *clique* if every pair of vertices in V is joined by an edge. Given a graph G and $k \in \mathbb{N}$, does G contain a clique of size k or larger?

6. ***Subset Sum.*** Given a set of n integers $V = \{v_1, v_2, \ldots, v_n\}$, is there a subset $U \subseteq V$ such that $\sum_{u_i \in U} u_i = k$?

7. ***Set Partition.*** Given a set of n integers $V = \{v_1, v_2, \ldots, v_n\}$, can the elements of V be partitioned into two sets U and $V - U$ such that $\sum_{u_i \in U} u_i = \sum_{u_i \in V-U} u_i$?

8. ***Graph Coloring.*** A graph $G = (V, E)$ is said to be k-colorable if the endpoints of any edge (u, v) could be colored using different colors when there is total of k available colors. Given a graph G and $k \in \mathbb{N}$, does G have a k-coloring?

9. ***3D Matching.*** Given disjoint sets X, Y, Z each of size n, and given a set $T \subseteq X \times Y \times Z$ of ordered triples, does there exist a subset of n triples in T such that each element of $X \cup Y \cup Z$ is contained in *exactly one* of those triples?

10. ***SAT (Satisfiability).***
    - Let $X = \{x_1, \ldots, x_n\}$ be a set of n Boolean variables [i.e. each x_i is 0 or 1].
    - A term t_i over X is either one of the variables x_i or its negation $\overline{x}_i$.
    - A clause C_j of length l is a disjunction of distinct terms: $C_j = t_1 \vee t_2 \vee \cdots \vee t_l$.
    - A collection of clauses is the conjuction $C_1 \wedge C_2 \wedge \cdots \wedge C_k$.

    A truth assignment is some assignment of values of 0 or 1 to each $x_i \in X$. In other words, a truth assignment is a function $f : X \to [0, 1]$. The collection of clauses $C_1 \wedge C_2 \wedge \cdots \wedge C_k$ is *satisfiable* if there exists a truth assignment that evaluates the collection to 1.

    A problem is called l-SAT if the length of all of its clauses C_j is exactly l.

B Acknowledgments

This work is primarily based on the following resources and I am very grateful for the opportunity to learn from those.

1. "Algorithm Design" by Jon Kleinberg and Eva Tardos.
2. LeetCode problems, solutions and discussion boards.
3. Slides and tutorial problems from the 2018S2 offering of CPSC 320 by Dr. Geoffrey Tien. Some of those are still available at: `https://www.students.cs.ubc.ca/~cs-320/2018S/schedule.html`.
4. UBC Computer Science Student Society Exam Bank available online at `https://exams.ubccsss.org/cs320/`.