EN.601.433 : Introduction to Algorithms

Problem Sets

James Guo

Spring 2025

Contents

Acknowledgements:

- The pratice problem sets are practices for EN.601.433 Introduction to Algorithms instructed by *Dr. Gagan Garg* at *Johns Hopkins University* in the Spring 2025 semester.
- The solutions might contain minor typos or errors. Please point out any notable error(s) through this link.

I Problem Set 1

Problem I.1. Matching with Indifferences in Ranking **[7 points]**.
The Stable Matching Problem, as discussed in the text, assumes that all men and women have a fully ordered list of preferences. In this problem we will consider a version of the problem in which men and women can be indifferent between certain options. As before, we have a set M of n men and a set W of n women. Assume each man and each woman ranks the members of the opposite gender, but now we allow ties in the ranking.

For example (with $n = 4$), a woman could say that m_1 is ranked in the first place; second place is a tie between m_2 and m_3 (she has no preference between them); and m_4 is in the last place.

We will say that w prefers m to m' if m is ranked higher than m' on her preference list (they are not tied). With indifferences in the rankings, there could be two natural notions for stability. And for each, we can ask about the existence of stable matchings, as follows.

(a) A **strong instability** in a perfect matching S consists of a man m and a woman w, such that both m and w prefers the other to their partner in S. Does there always exist a perfect matching with no strong instability?

Either give an example of a set of men and women with preference lists for which every perfect matching has a strong instability; or give an algorithm that is guaranteed to find a perfect matching with no strong instability and prove the correctness. [4 points]

Solution. There always exists a perfect matching with no strong instability.
First, we can develop an algorithm as follows:

---

while there exists some $m \in M$ such that his ranking has ties **do**
  Break the tie by assigning the ties with a random order.
end while
while there exists some $w \in W$ such that her ranking has ties **do**
  Break the tie by assigning the ties with a random order.
end while
Do the Gale-Shapley Algorithm, and **return** the matching from Gale-Shapley M as the result.

Proof. The Gale-Shapley algorithm returns a stable matching without any unstable pair. Suppose that M would contain a strong instability pair, say $(m, w) \notin M$ such that m prefers w to $M(m)$ and w prefers m to $M(w)$, then (m, w) must be a unstable pair for the Gale-Shapley algorithm, which is contradiction.
Therefore, by the validity of Gale-Shapley algorithm, the provided algorithm works. □

By showing that the algorithm work, there always exists a perfect matching with no strong instability. ⌟

(b) A **weak instability** in a perfect matching S consists of a man m and a woman w, such that their partners in S are w' and m', respectively, and one of the following holds:

- m prefers w to w', and w either prefers m to m' or is indifferent between these two choices; or
- w prefers m to m', and m either prefers w to w' or is indifferent between these two choices.

In other words, the pairing between m and w is either preferred by both, or preferred by one while the other is indifferent. Does there always exist a perfect matching with no weak instability?

Either give an example of a set of men and women with preference lists for which every perfect matching has a weak instability; or give an algorithm that is guaranteed to find a perfect matching with no weak instability and prove the correctness. [3 points]

Solution. There does not necessarily exist a perfect matching with no weak instability.
We can form the counter example. Let $M := \{m_1, m_2\}$ and $W := \{w_1, w_2\}$. We set the preference matrices as follows:

	1st	2nd
m_1	w_1	w_2
m_2	w_1	w_2

	1st	2nd
w_1	m_1 or m_2	
w_2	m_1 or m_2	

Table I.1. Preference matrices for M and W.

For $|M| = |W| = 2$, there could only be 2 possible matches, namely:

$$\{(m_1, w_1), (m_2, w_2)\} \text{ or } \{(m_1, w_2), (m_2, w_1)\}.$$

However, both matchings contain a weak instability pair:

- For $\{(m_1, w_1), (m_2, w_2)\}$, (m_2, w_1) is an instability pair, since m_2 prefers w_1 to w_2 and w_1 is indifference about m_1 or m_2.
- For $\{(m_1, w_2), (m_2, w_1)\}$, (m_1, w_1) is an instability pair, since m_1 prefers w_1 to w_2 and w_1 is indifference about m_1 or m_2.

Hence, for the above arrangement, there exists no perfect matching without weak instability. ⌟

Problem I.2. Stable Matching with Malicious Intent **[5 points]**

So far, we have assumed that when people give their preference list as an input to the Gale-Shapley algorithm, they are acting truthfully. In other words, their preference list truly reflects their desire. However, what will happen if someone lied about their list? Could they achieve better results by manipulating their preference list?

Consider a specific scenario where there are 3 universities U_1, U_2, U_3 and 3 students S_1, S_2, S_3. Assume that the universities are proposing to the students. Also assume that the preferences of all the universities and all the candidates are known to everyone. Without loss of generality, assume that student S_1 prefers universities U_1, U_2, U_3 in this order, that is, U_1 is their first preference and U_3 is their last preference. Also assume that when you run the Gale-Shapley algorithm, S_1 ends up matching with U_2, which is their second preference.

Is it possible that, by swapping the order of universities U_2 and U_3 in its preference list, S_1 ends up matching with U_1? That is, is it possible that by lying about their preference list, S_1 can gain some advantage? The real preference list for S_1 is U_1, U_2, U_3 but they submit U_1, U_3, U_2 as their preference list to the system.

Resolve this question by either proving that swapping the order does not lead to S_1 getting matched with U_1, or by providing an example where S_1 is matched with U_1 as a result of the switch.

Solution. Yes, swapping the order could make S_1 to pair with U_1.
Consider the following arrangement:

	1st	2nd	3rd
U_1	S_2	S_1	S_3
U_2	S_1	S_2	S_3
U_3	S_1	S_3	S_2

	1st	2nd	3rd
S_1	U_1	U_2	U_3
S_2	U_2	U_1	U_3
S_3	U_3	U_1	U_2

Table I.2. Preference matrices before S_1 does malicious move.

In this setup, the matching pair by Gale-Shapley algorithm is:

$$M = \{(U_1, S_2), (U_2, S_1), (U_3, S_3)\}.$$

After the malicious move, we have:

	1st	2nd	3rd
U_1	S_2	S_1	S_3
U_2	S_1	S_2	S_3
U_3	S_1	S_3	S_2

	1st	2nd	3rd
S_1	U_1	U_3	U_2
S_2	U_2	U_1	U_3
S_3	U_3	U_1	U_2

Table I.3. Preference matrices after S_1 does malicious move.

In this setup, the matching pair by Gale-Shapley algorithm is:

$$M = \{(U_1, S_1), (U_2, S_2), (U_3, S_3)\}.$$

Hence, swapping the order lead to S_1 getting matched with U_1. ⌟

Problem I.3. CA Applications **[6 points]**

Every semester, the CS department must handle a large number of Course Assistant (CA) applications. Hypothetically, this semester, Jamie is responsible for managing the process. She needs to assign 5 CAs Alice, Bob, Cathy, Drew, and Eve to 3 courses, each led by a different professor: X, Y, and Z. Each professor, p, has requested for c_p (a positive integer) number of CAs for their course.

Fortunately, the total number of CA applications exactly matches the total number of CAs requested by the professors (so every applicant gets a position, and every course gets the required number of CAs). Moreover, both the applicants and the professors have submitted their preference lists, and they all agree to use a modified version of the Gale-Shapley algorithm, with the students proposing, to determine the CA assignments.

(a) Describe how to modify the Gale-Shapley algorithm to handle this CA application problem. Prove the correctness of your algorithm. [4 points]

Solution. Here, we modify the algorithm as follows:

---

for all p being a professor such that $c_p > 1$ **do**
  Make a $c_p - 1$ distinct copies of p, denote them by $p_2, p_3, \cdots, p_{c_p}$.
  Copy the same preferences of p to $p_2, p_3, \cdots, p_{c_p}$.
end for
for all a being a CA **do**
  Extend their ranking into 5 slots.
  for all p on the preference list on a that is not a copy **do**
    if $c_p > 1$ **then**
      Keep the current ranking of p.
      Move all the professors ranked after p backwards for $c_p - 1$ positions.
      Make the next $c_p - 1$ rankings after p as $p_2, p_3, \cdots, p_{c_p}$.
    end if
  end for
end for
Do the Gale-Shapley Algorithm, and **return** M which is the result of the Gale-Shapley Algorithm.

Proof. Again, our preprocessing make the problem into a stabling matching problem such that there exists a perfect matching, the Gale-Shapley algorithm guarantees a stable match M. For the sake of contradiction, suppose the result contains a pair (p, a) such that p prefers a to at least one of their assignment and a prefers p to their current matched professor. Then, (p_{c_p}, a) would be an unstable pair in M, since by construction p_{c_p} would be matched to the least favored one on the matched list, and then p_{c_p} would prefer a to its current match, and by assumption a prefers p to its current match. Thus, a unstable pair exists in M, which is a contradiction. □

Therefore, this algorithm is correct. ⌟

Ranking	Alice	Bob	Cathy	Drew	Eve
1	X	X	Y	Y	X
2	Z	Y	Z	X	Y
3	Y	Z	X	Z	Z

Ranking	X	Y	Z
1	Alice	Alice	Alice
2	Cathy	Eve	Drew
3	Drew	Cathy	Bob
4	Eve	Bob	Eve
5	Bob	Drew	Cathy

(b) Use your modified algorithm to compute the matching for these preferences, given that $c_X = 2$, $c_Y = 2$, and $c_Z = 1$. [1 point]

Solution. Now, we use the algorithm, as we modify the matrices of preferences as:

Ranking	Alice	Bob	Cathy	Drew	Eve
1	X	X	Y	Y	X
2	X_2	X_2	Y_2	Y_2	X_2
3	Z	Y	Z	X	Y
4	Y	Y_2	X	X_2	Y_2
5	Y_2	Z	X_2	Z	Z

Ranking	X	X_2	Y	Y_2	Z
1	Alice	Alice	Alice	Alice	Alice
2	Cathy	Cathy	Eve	Eve	Drew
3	Drew	Drew	Cathy	Cathy	Bob
4	Eve	Eve	Bob	Bob	Eve
5	Bob	Bob	Drew	Drew	Cathy

Figure I.4. Modified matrices of preferences.

Now, we apply the Gale-Shapley algorithm through student proposing to obtain the match:

$$M = \{(\text{Alice}, X), (\text{Bob}, Z), (\text{Cathy}, Y_2), (\text{Drew}, X_2), (\text{Eve}, Y)\}.$$

Then, the actual pairing would be $\{(X, \{\text{Alice}, \text{Drew}\}), (Y, \{\text{Cathy}, \text{Eve}\}), (Z, \text{Bob})\}$. ⌟

(c) Use your modified algorithm to compute the matching for these preferences, given that $c_X = 3$, $c_Y = 1$, and $c_Z = 1$. [1 point]

Solution. Again, we use the algorithm, as we modify the matrices of preferences as:

Ranking	Alice	Bob	Cathy	Drew	Eve
1	X	X	Y	Y	X
2	X_2	X_2	Z	X	X_2
3	X_3	X_3	X	X_2	X_3
4	Z	Y	X_2	X_3	Y
5	Y	Z	X_3	Z	Z

Ranking	X	X_2	X_3	Y	Z
1	Alice	Alice	Alice	Alice	Alice
2	Cathy	Cathy	Cathy	Eve	Drew
3	Drew	Drew	Drew	Cathy	Bob
4	Eve	Eve	Eve	Bob	Eve
5	Bob	Bob	Bob	Drew	Cathy

Figure I.5. Modified matrices of preferences.

Now, we apply the Gale-Shapley algorithm through student proposing to obtain the match:

$$M = \{(\text{Alice}, X), (\text{Bob}, Z), (\text{Cathy}, Y), (\text{Drew}, X_2), (\text{Eve}, X_3)\}.$$

Then, the actual pairing would be $\{(X, \{\text{Alice}, \text{Drew}, \text{Eve}\}), (Y, \text{Cathy}), (Z, \text{Bob})\}$. ⌟

Problem I.4. Order the Big-$\mathcal{O}$ Complexities **[4 points]**

Sort the following list of functions in ascending order of growth rate and briefly explain why you put them in such order. For example, if $f(n)$ appears before $g(n)$, then $f(n) = O(g(n))$ and you need to explain why $f(n) = O(g(n))$.

Note: A graph depicting the growth of these functions does not count as an explanation.

$$
\begin{aligned}
g_1(n) &= e^{\frac{\log n}{3}} \\
g_2(n) &= n! \\
g_3(n) &= n(\log n)^3 \\
g_4(n) &= \log(\log n) \\
g_5(n) &= n^{100} + n^{99} + n^{98} + \cdots + n^2 + n \\
g_6(n) &= 2^{2^n} \\
g_7(n) &= 2^{n^2} \\
g_8(n) &= n^n
\end{aligned}
$$

Solution. Here, we first give the order:

$$g_4 \precsim g_1 \precsim g_3 \precsim g_5 \precsim g_2 \precsim g_8 \precsim g_7 \precsim g_6,$$

where $a \precsim b$ is defined to be $a \in \mathcal{O}(b)$ for any functions a and b.
Prior to showing the orders, we first rewrite some of the expressions, as follows:

$$
\begin{aligned}
g_1(n) &= \exp\left(\frac{\log n}{3}\right) = \exp\left(\log n \times \frac{1}{3}\right) = (\exp(\log n))^{1/3} = n^{1/3}, \\
g_6(n) &= \exp\left(\log(2^{2^n})\right) = \exp\left(2^n \log(2)\right), \\
g_7(n) &= \exp\left(\log(2^{n^2})\right) = \exp\left(n^2 \log(2)\right), \\
g_8(n) &= \exp\left(\log(n^n)\right) = \exp\left(n \log(n)\right).
\end{aligned}
$$

Then, we prove each of the order, respectively.

Proof.

- ($g_4 \precsim g_1$:) Note that by the definition of $\exp(-)$, we have:

$$\exp(x) = \sum_{n=0}^{\infty} \frac{x^n}{n!} = 1 + x + \frac{x^2}{n!} + \cdots,$$

so $e^{(-)}$ is monotonic, and $e^x > x$. Moreover, the inverse function must satisfy that $\log(x) < x$, so by logarithms and polynomials formula (c.f. pp.22, slide 2) have:

$$g_4(n) = \log(\log n) < \log(n) < Cn^{1/3} = Cg_1(n) \text{ for all } n > n_0 \text{ for some } C \text{ and } n_0.$$

- ($g_1 \precsim g_3$:) First, we note that $\log(x) > 1$ for $x > e$. Hence, we have for all $n > e$ such that:

$$g_1(n) = n^{1/3} < n < n(\log n)^3 = g_3(n).$$

- ($g_3 \precsim g_5$:) From the previous part, we have shown that $\log n < x$, and since n is strictly increasing, consider $n > 1$, we have $(\log n)^3 < n^3$, hence for all $n > 1$:

$$g_3(n) = n(\log n)^3 < n \times n^3 = n^4 < n^{100} < n^{100} + n^{99} + \cdots + n^2 + n = g_5(n).$$

- ($g_5 \precsim g_2$:) Here, we note that for $n > 200$:

$$\begin{aligned} g_5(n) &= n^{100} + n^{99} + \cdots + n^2 + n < 100n^{100} = 100\left(\frac{n}{2} \times 2\right)^{100} = 100\left[\left(\frac{n}{2}\right)^{100} \times 2^{100}\right] \\ &= 100\Bigg[\underbrace{\frac{n}{2} \times \cdots \times \frac{n}{2}}_{100 \text{ terms}} \times \underbrace{2 \times \cdots \times 2}_{100 \text{ terms}}\Bigg] \\ &< 100\Bigg[n \times \cdots \times \underbrace{\left\lceil \frac{n}{2} + 100 \right\rceil \times \left\lceil \frac{n}{2} + 99 \right\rceil \times \cdots \left\lceil \frac{n}{2} + 1 \right\rceil}_{100 \text{ terms greater than } n/2} \times \underbrace{\left\lceil \frac{n}{2} \right\rceil \times \cdots \times \left\lceil \frac{n}{2} - 99 \right\rceil}_{100 \text{ terms greater than2}} \times \cdots \times 1\Bigg] \\ &= 100n! = 100g_2(n). \end{aligned}$$

- ($g_2 \precsim g_8$:) Here, we can expand the multiplication with the limit as:

$$\lim_{n\to\infty} \frac{g_2}{g_8} = \lim_{n\to\infty} \frac{n!}{n^n} = \lim_{n\to\infty} \Bigg(\underbrace{\frac{n}{n}}_{1} \times \underbrace{\frac{n-1}{n} \times \frac{n-2}{n} \times \cdots \times \frac{1}{n}}_{\text{less than 1 and tending towards 0}} \Bigg) = 0.$$

  In particular, we shall note that the infinite has all terms no larger than 1, and the later terms eventually gets to 0 as $n \to \infty$, hence the limit is 0, thus we have $g_2 \in \mathcal{O}(g_8)$.
  Remark. More formally, we can argue that for each $n \in \mathbb{N}^+$, we have $g_2(n)/g_8(n) < 1/n$ and is positive, and thus as the sequence $\{1/n\}_{n=1}^{\infty}$ converges to 0, through the dominance convergence (or the *squeeze theorem*), we must have the limit of $g_2/g_8 \to 0$ as $n \to \infty$.

- ($g_8 \precsim g_7$:) Recall that $\exp(-)$ is monotonic, we just need to compare $n\log(n)$ and $n^2\log(2)$, recall that $n > \log(n)$ for $n > 0$, we have $n^2 > n\log(n)$ for all $n > 1$, hence we have for $n > 1$ that:

$$g_8(n) = \exp\big(n\log(n)\big) < \exp(n^2 \log 2) \in \mathcal{O}\big(g_7(n)\big).$$

- ($g_7 \precsim g_6$:) Again, $\exp(-)$ is monotonic, we just need to compare $n^2\log 2$ and $2^n \log 2$.
  Now, we wan to show that $n^2 < 2^n$ for $n > 4$, through induction:
  - (Base case:) For $n = 5$, we have $5^2 = 25 < 32 = 2^5$.
  - (Inductive step:) Suppose for $n \in \mathbb{Z}_{\geq 5}$, we have $n^2 < 2^n$, then we have:

$$(n+1)^2 = n^2 + 2n + 1 < n^2 + n^2 = 2n^2 < 2 \cdot 2^n = 2^{n+1}.$$

    Hence, we have $(n+1)^2 < 2^{n+1}$, which completes the inductive proof.

  Hence, we have $n^2 \log 2 < 2^n \log 2$, so we have:

$$g_7(n) = \exp\big(n^2\log(2)\big) < 2^n \log(2) \in \mathcal{O}\big(g_8(n)\big).$$

□

By showing all the $\precsim$ pre-orders, we have justified our ordering. ⌟

Problem I.5. Dropping Jars **[8 points]**

You're doing some stress-testing on various models of glass jars to determine the height from which they can be dropped and still not break. The setup for this experiment, on a particular type of jar, is as follows. You have a ladder with n rungs, and you want to find the highest rung from which you can drop a copy of the jar and not break it. We call this the *highest safe rung*.

It might be natural to try binary search: drop a jar from the middle rung, see if it breaks, and then recursively try from rung $\frac{n}{4}$ or $\frac{3n}{4}$ depending on the outcome. But this has the drawback that you could break a lot of jars in finding the answer.

If your primary goal was to conserve jars, on the other hand, you could try the following strategy. Start by dropping a jar from the first rung, then the second rung, and so forth, climbing one higher each time until the jar breaks. In this way, you only need a single jar at the moment it breaks, you have the correct answerbut you may have to drop it n times (rather than $\log n$ as in the binary search solution).

So here is the trade-off: it seems you can perform fewer drops if you're willing to break more jars. To understand better how this trade-off works at a quantitative level, let's consider how to run this experiment given a fixed "budget" of $k \geq 1$ jars. In other words, you have to determine the correct answer - the highest safe rung – and can use at most k jars.

(a) Suppose you are given a budget of $k = 2$ jars. Describe a strategy for finding the highest safe rung that requires you to drop a jar at most $f(n)$ times, for some function $f(n)$ that grows slower than linearly. In other words, it should be the case that $\lim_{n\to\infty} \frac{f(n)}{n} = 0$. You should explain your algorithm in detail and give a short proof of its run time. [5 points]

Solution. We give the following strategy, let $k = \lceil \sqrt{n} \rceil$,[1] then we may develop the algorithm as follows:

Let $\{m_i\}_{i=0}^{k} = \{0, k, 2k, 3k, \cdots, k^2\}$ (replace all terms that are greater than n by n).
for $i \leftarrow 1, 2, 3, \cdots, k$ **do**
  Drop the first jar at floor m_i.
  if the jar breaks **then**
    Record the value of i and break the for loop.
  end if
end for
for $j \leftarrow m_{i-1}, m_{i-1}+1, \cdots, m_i - 1$ **do**
  Drop the second jar at floor j.
  if the jar breaks **then**
    return $j - 1$ as the maximum non-breaking floor.
  end if
end for
return $m_i - 1$ as the maximum non-breaking floor.

This algorithm terminates since the for loops are finite, and it will return a floor if both jars are broken. Moreover, it returns the result that is one less than the level in which the jar is broken and itself not broken, hence it is working.

[1] Here, I chose to round up for the effectiveness of the algorithm below, it is asymptotically the same with the square root.

Given this algorithm, the first for loop runs at most $\lceil\sqrt{n}\rceil$ times, and the second loop runs at most $\lceil\sqrt{n}\rceil$ times, so the total run time at most $2\lceil\sqrt{n}\rceil$ times, while:

$$\lim_{n\to\infty}\frac{2\lceil\sqrt{n}\rceil}{n} < \lim_{n\to\infty}\frac{2\sqrt{n}+2}{n} = \lim_{n\to\infty}\frac{2\sqrt{n}}{n} + \lim_{n\to\infty}\frac{2}{n} = \lim_{n\to\infty}\frac{2}{\sqrt{n}} + \lim_{n\to\infty}\frac{2}{n} = 0.$$

while since both numerator and denominator is non-negative, the limit is 0 as required. ⌟

Remark. As a side note, we may provide another algorithm that is more efficient, but asymptotically perform the same. Here, we consider the strategy as following:

Let x be the smallest integer such that $1+2+3+\cdots+x \geq n$.
Let $m \leftarrow 0$.
for $i = x, x-1, x-2, \cdots, 1$ **do**
  Let $m \leftarrow m+i$.
  Drop the first jar on floor m, cap at floor n for the highest floor.
  if The jar breaks **then**
    Let $m \leftarrow m-i$.
    Record the value of i and break this for loop.
  end if
end for
while The second jar is not broken and $m < \min\{n, m+i\} - 1$ **do**
  Drop the second jar on floor $m+1$.
  if The jar breaks **then**
    Break this while loop.
  end if
  Let $m \leftarrow m+1$.
end while
return m as the highest floor in which the jar does not break.

For this strategy, it is valid since we are never attempting to drop a jar after both jars are broken. Time-wise, the second jar will be tested within the interval nominated from the previous i-th term, and as i decrements before breaking the loop in the first loop, they together runs within x times dropping the jar. Note that we have:

$$1+2+3+\cdots+x = \frac{(1+x)\times x}{2},$$

so we can solve x as:

$$x = \left\lceil\frac{-1+\sqrt{1+8n}}{2}\right\rceil \in \mathcal{O}\left(\frac{-1+\sqrt{1+8n}}{2}\right) \subset \mathcal{O}\left(-1+\sqrt{1+8n}\right) \subset \mathcal{O}(\sqrt{1+8n}) \subset \mathcal{O}(\sqrt{n}).$$

Therefore, we have $f(n) = x \in \mathcal{O}(\sqrt{n})$, so there exists some $C > 0$ and $N \in \mathbb{N}^+$ such that for all $n > N$ in which $f(n) < C\sqrt{n}$, now we have the limit:

$$\lim_{n\to\infty}\frac{f(n)}{n} \leq \lim_{n\to\infty}\frac{C\sqrt{n}}{n} = \lim_{n\to\infty}\frac{C}{\sqrt{n}} = 0,$$

while since both numerator and denominator is non-negative, the limit is 0 as required.

(b) Now consider a budget of $k = 3$ jars. Provide an algorithm and a quick proof of its run time. Note that your algorithm using 3 jars should run faster than your algorithm that uses only 2 jars. More precisely, let your algorithm using 3 jars requires you to drop a jar at most $g(n)$ times. Then, it should be the case that $\lim_{n\to\infty} \frac{g(n)}{f(n)} = 0$. [3 points]

Solution. Still, we first give the following algorithm based on the preceding part, let $k = \lceil \sqrt[3]{n^2} \rceil$, we have:

Let $\{m_i\}_{i=0}^{\lceil \sqrt[3]{n} \rceil} = \{0, k, 2k, 3k, \cdots, \lceil \sqrt[3]{n} \rceil k\}$ (replace all terms that are greater than n by n).
for $i \leftarrow 1, 2, 3, \cdots, \lceil \sqrt[3]{n} \rceil$ **do**
  Drop the first jar at floor m_i.
  if the jar breaks **then**
    Apply the algorithm in part (a) on interval $[m_{i-1}, m_i - 1]$, **return** its output.
  end if
end for
return n as the maximum non-breaking floor.

Again, this algorithm terminates since all the for loop is finite, while it either recognize the jar does not break at the highest floor or turn the problem into a problem of part (a) of a certain interval, hence it is working, since the highest floor of non-breaking is either the highest floor, or the highest floor such that the jar breaks at one floor higher.
To analyze the time, we know that the for loop runs at most $\lceil \sqrt[3]{n} \rceil$ times, and for the rest interval of length at most $\lceil \sqrt[3]{n^2} \rceil$, recall that the algorithm of the previous part makes sure to solve it in $\mathcal{O}\big((\sqrt[3]{n^2})^{1/2}\big) \sim \mathcal{O}(\sqrt[3]{n})$, therefore, the total run time is a constant multiple of $\sqrt[3]{n}$ (denote the constant by C)[2], also we recall that the algorithm involving two jars had the worst case as a multiple of $\sqrt{n}$ (denote the constant by $\tilde{C}$), thus, we have:

$$\lim_{n\to\infty} \frac{g(n)}{f(n)} \leq \lim_{n\to\infty} \frac{C\sqrt[3]{n}}{\tilde{C}\sqrt{n}} = \lim_{n\to\infty} \frac{C/\tilde{C}}{n^{1/6}} = 0,$$

while since both numerator and denominator is non-negative, the limit is 0 as required. ⌟

Remark. Also as a side note, we can also provide another algorithm that is more efficient, but asymptotically perform the same. For simplicity of notation, we denote $S_n := \sum_{i=1}^{n} i$. Here, we consider the strategy as following:

Let x be the smallest integer such that $S_1 + S_2 + S_3 + \cdots + S_x \geq n$.
Let $m \leftarrow 0$.
for $i = S_x, S_{x-1}, S_{x-2}, \cdots, S_1$ **do**
  Let $m \leftarrow m + i$.
  Drop the first jar on floor m, cap at floor n for the highest floor.
  if The jar breaks **then**

[2]It can be shown that we have some multiple of $\sqrt{n}$ times for the algorithm in the previous part: by having $\mathcal{O}(\sqrt{n})$, and it can also be interpreted $\Omega(\sqrt{n})$ since it need at least some multiple of $\sqrt{n}$ for the worst case floor, so we can write as a constant multiple of $\sqrt{n}$.

Use the optimized algorithm from part (a) over the interval $[m-i+1, m-1]$, **return** the result as the output of the highest floor in which the jar does not break.

end if

end for

return n as the highest floor in which the jar does not break.

This algorithm terminates and all the cases are controlled within S_n steps, consider the sum:

$$\sum_{i=1}^{x} S_i = \sum_{i=1}^{x} \frac{(1+i)\times i}{2} = \frac{x(x+1)(x+2)}{6}.$$

Thus, when we are solving for the minimum integer x given the above sum, so the total number of drops is S_x. Note that this is polynomial of degree 3, there exists a closed-form solution, but it will be very complicated, and it can be interpreted using the degree three polynomial and whose root is of $\mathcal{O}(\sqrt[3]{n})$, but the verification will be hard. Although it would be more efficient than the provided solution, they perform asymptotically the same.

Problem I.6. There are many settings in which we can ask questions related to some type of *stability* principle. Heres one, involving competition between two enterprises.

Suppose we have two television networks, whom well call A and B. There are n prime-time programming slots, and each network has n TV shows. Each network wants to devise a schedule – an assignment of each show to a distinct slot – so as to attract as much market share as possible.

Here is the way we determine how well the two networks perform relative to each other, given their schedules. Each show has a fixed rating, which is based on the number of people who watched it last year; well assume that no two shows have exactly the same rating. A network wins a given time slot if the show that it schedules for the time slot has a larger rating than the show the other network schedules for that time slot. The goal of each network is to win as many time slots as possible.

Suppose in the opening week of the fall season, Network A reveals a schedule S and Network B reveals a schedule T. On the basis of this pair of schedules, each network wins certain time slots, according to the rule above. Well say that the pair of schedules (S, T) is stable if neither network can unilaterally change its own schedule and win more time slots. That is, there is no schedule S' such that Network A wins more slots with the pair (S', T) than it did with the pair (S, T); and symmetrically, there is no schedule T' such that Network B wins more slots with the pair (S, T') than it did with the pair (S, T).

The analogue of Gale and Shapleys question for this kind of stability is the following: For every set of TV shows and ratings, is there always a stable pair of schedules? Resolve this question by doing one of the following two things:

(a) give an algorithm that, for any set of TV shows and associated ratings, produces a stable pair of schedules; or

(b) give an example of a set of TV shows and associated ratings for which there is no stable pair of schedules.

Solution. There is no such algorithm. To prove this, we simply need to provide an example in which there are two schedules such that for one schedule the first networks wins, whereas for the other schedule the second networks wins.

Proof. Let $n = 3$, and A, B be two networks, with TV shows A_1, A_2, A_3 for A and B_1, B_2, B_3 for B, respectively. Let the rating of the TV shows be as follows:

$$A_1 > B_1 > A_2 > B_2 > A_3 > B_3.$$

Let the first arrangement be:

Time Slot	Show from A	Show from B	Result
Slot 1	A_1	B_1	A wins
Slot 2	A_2	B_2	A wins
Slot 3	A_3	B_3	A wins

Table I.6. Arrangement Schedule 1: A wins 3 times, B wins 0 times.

Then, let the second arrangement be:

Time Slot	Show from A	Show from B	Result
Slot 1	A_1	B_3	A wins
Slot 2	A_2	B_1	B wins
Slot 3	A_3	B_2	B wins

Table I.7. Arrangement Schedule 2: A wins 1 time, B wins 2 times.

Note that since we just have two networks, we can think of the adjustment of a single network as corresponding the TV show with what is listed from the other firm.
Now, given any arrangement, as long as A is winning for 3 times, B can adjust its setup corresponding to Table I.7 to win more, if A is not winning for 3 times, A can adjust its setup corresponding to Table I.6 to win more, so there is no arrangement that is a stable arrangement. □

Hence, there could not be an algorithm for this problem to work.

Remark. This question is similar to an interesting Chinese story about Tian Ji's Horse Racing Strategy, feel free to check the strategy out. ⌟

Problem I.7. Construct an instance of the stable matching problem with 4 companies $c_1, \cdots, c_4$ and 4 applicants $a_1, \cdots, a_4$ with two stable matchings M_1, M_2 such that

- c_1 prefers its partner in M_1 to its partner in M_2.
- c_2 prefers its partner in M_2 to its partner in M_1.

Note that c_1 (and similarly c_2) must have different partners in M_1, M_2.

Solution. Now, we form our preference matrices as:

Com.	1st	2nd	3rd	4th
c_1	a_1	a_3	(-)	(-)
c_2	a_2	a_4	(-)	(-)
c_3	a_3	a_1	(-)	(-)
c_4	a_4	a_2	(-)	(-)

Appl.	1st	2nd	3rd	4th
a_1	c_3	c_1	(-)	(-)
a_2	c_4	c_2	(-)	(-)
a_3	c_1	c_3	(-)	(-)
a_4	c_2	c_4	(-)	(-)

Table I.8. Preference matrices for companies and applicants, (-) indicates whatever is left.

We can form the stable matching pairs as:

$$M_1 = \{(c_1, a_1), (c_2, a_4), (c_3, a_3), (c_4, a_2)\} \text{ and } M_2 = \{(c_1, a_3), (c_2, a_2), (c_3, a_1), (c_4, a_4)\}.$$

Clearly, both above matching are stable (*we invite readers to check this on their own*), and:

- c_1 prefers $M_1(c_1) = a_1$ over $M_2(c_1) = a_3$, while
- c_2 prefers $M_2(c_2) = a_2$ over $M_1(c_2) = a_4$.

Hence, this completes the case for this problem. ⌟

Problem I.8. Let f and g be two functions that take nonnegative values, and suppose that $f \in \mathcal{O}(g)$. Show that $g \in \Omega(f)$.

Solution. We provide with the following proof.

Proof. By the assumption, there exists some $C \in \mathbb{R}^+$ and $N \in \mathbb{N}$ such that for all $n \geq N$, we have:

$$0 \leq f(n) \leq C \cdot g(n).$$

Hence, we have $\tilde{C} = 1/C \in \mathbb{R}^+$ such that for all $n \geq N$, we have:

$$g(n) \geq \frac{1}{C} f(n) \geq 0.$$

Hence, we have $g(n) \in \Omega(f)$. □

Therefore, we have completed the proof. ⌟

Problem I.9. Consider the following recurrence relation:

- $T(n) = 2$ if $n = 2$;
- $T(n) = 2T(n/2) + n$ if $n = 2k$ for $k > 1$

Assume that n is an exact power of 2. Use mathematical induction to show that the solution is $T(n) = n \log_2 n$.

Solution. Mathematical induction works for any countable sets, so we use induction on this problem for $\{2^k : k \in \mathbb{Z}^+\}$.

Proof.

- (Base case:) Consider $n = 2$, we have $T(n) = 2 = 2 \times 1 = 2 \times \log_2(2)$.
- (Inductive step:) By the induction hypothesis, let $k = 2^n$ such that $n \in \mathbb{Z}^+$ be arbitrary, we suppose:

$$T(k) = k \log_2(k) = k \cdot n = n2^n.$$

Then, we have:

$$T(2^{n+1}) = T(2k) = 2 \cdot T(k) + 2^{n+1} = 2n2^n + 2^{n+1} = n2^{n+1} + 2^{n+1} = (n+1)2^{n+1} = 2^{n+1} \log_2(2^{n+1}),$$

which completes the inductive step.

Hence, we have shown that $T(n) = n \log_2(n)$ for all n as an exact power of 2. □

This completes the proof of the problem. ⌟

Problem I.10. Describe a $\mathcal{O}(n \log_2 n)$ time algorithm that, given a set S of n integers and another integer x, determines whether or not there exist two elements in S whose sum is exactly x.

Solution. Here, we develop the algorithm as follows.

Let $L \leftarrow$ list of n integers.
Sort L (increasing order) using *merge sort* or *heap sort*.[3]
Let s points to the smallest number and let b points to the biggest number.
while s and b are not pointing to the same number **do**
  Let $v \leftarrow$ sum of the number that s and b is pointing to.
  if $v > x$ **then**
    Let s points to the number on the right of the current one that s is pointing to.
  else if $v < x$ **then**
    Let b points to the number on the left of the current one that b is pointing to.
  else
    return true. There exists two elements in S whose sum is exactly x.
  end if
end while
return false. There does not exist two elements in S whose sum is exactly x.

Note that the sort is $\mathcal{O}(n \log_2 n)$ and the while loop runs at most $n + 1 \in \mathcal{O}(n)$ times and the executions in the while loop is constant time, so the total runtime is $\mathcal{O}(n \log_2 n + n) = \mathcal{O}(n \log_2 n)$, as desired.
Consider why this algorithm works, we may prove its correctness.

Proof. Since if the pair does not exist, the algorithm would not be able to find a pair such that the sum is correct, so we reduce our justification to verifying that it would not miss a pair.
For the sake of contradiction, suppose that there exists a pair $a \leq b$ such that $a + b = x$, but the above algorithm fails to recognize that.
We quickly denounce the case in which the pointers meet on the left of a or the right of b since in those cases, the pointers cannot be moving in the direction when the sum is already larger or smaller, so we consider that the pointers meet between a and b.
Now, the left pointer must have reached a at some moment, and while if the right pointer is on the right of b, it shall move to b and terminate, so the right pointer must be already at the left of b. Likewise, when the right pointer is at b, the left pointer must have already passed to the right of a. However, this is a contradiction since the one pointer must pass a or b first before meeting at the center, thus they would have to get to the pair, not ignoring it. □

Remark. For the left-right pointer part, an easier approach is for each number n in the list, find the number $x - n$ from the list (that is not itself) using **binary search**. In this way, the runtime is still $\mathcal{O}(n \log n + n \log n)$ but the justification of the algorithm will be much easier. ⌟

[3] We use these two types of sort so that the sorting time is of $\mathcal{O}(n \log_2 n)$.

Problem I.11. Suppose $f \in \mathcal{O}(g)$:

(a) Is $f^2 \in \mathcal{O}(g^2)$?

Solution. Yes.

Proof. Suppose $f \in \mathcal{O}(g)$, then there exists $n_0 > 0$ and $C > 0$ such that for all $n > n_0$, we have:

$$0 \leq f(n) \leq Cg(n),$$

hence, by multiplying the inequality with itself, we have:

$$0 \leq \big(f(n)\big)^2 \leq C^2\big(g(n)\big)^2,$$

for all $n > n_0$ and $\tilde{C} = C^2$, so $f^2 \in \mathcal{O}(g^2)$. □

Therefore, we have completed the proof. ⌟

(b) Is $2^f \in \mathcal{O}(2^g)$?

Solution. Not necessarily. We provide the following example. Let:

$$f(n) = 2n \text{ and } g(n) = n,$$

clearly $f \in \mathcal{O}(g)$ (*verify this by yourself*) and we note that:

$$2^f = 2^{2n} = (2^n)^2 \text{ whereas } 2^g = 2^n.$$

When we take the limit:

$$\lim_{n\to\infty} \frac{(2^n)^2}{2^n} = \lim_{n\to\infty} 2^n = \infty,$$

hence $2^f \notin \mathcal{O}(2^g)$. ⌟

II Problem Set 2

Problem II.1. Wireless Network Connectivity **[6 points]**

Some friends of yours work on wireless networks, and they're currently studying the properties of a network of n mobile devices. As the devices move around (actually, as their human owners move around), they define a graph at any point in time as follows: there is a node representing each of the n devices, and there is an edge between device i and device j if the physical locations of i and j are no more than 500 meters apart. (If so, we say that i and j are "in range" of each other.)

They'd like it to be the case that the network of devices is connected at all times, and so they've constrained the motion of the devices to satisfy the following property: at all times, each device i is within 500 meters of at least $n/2$ of the other devices. We'll assume n is an even number. What they'd like to know is: Does this property by itself guarantee that the network will remain connected?

Here's a concrete way to formulate the question as a claim about graphs.

> *Claim: Let G be a graph on n nodes, where n is an even number. If every node of G has degree at least $n/2$, then G is always connected.*

Determine whether the claim is true or false, and either give a proof of the claim or a counter-example.

Solution. This is **true**.

Proof. Here, we define an equivalence relationship $\sim_G$ for the set of nodes V in G such that for any two nodes $v, v' \in V$ are $v \sim_G v'$ if v and v' are connected, in particular, we are say a node is connected to itself. We can trivially show that this is an equivalence relationship.

- **Reflexive**: By definition, a node is connected to itself, so for any $v \in V$, $v \sim_G v$.
- **Symmetry**: Suppose $v, v' \in V$, we have $v \sim_G v'$ implies that v and v' are connected, so $v' \sim_G v$.
- **Transitive**: Consider $v, v', v'' \in V$ in which $v \sim_G v'$ and $v' \sim_G v''$, then there exists a path from v to v' and a path from v' to v'', hence existing a path from v to v'', so $v \sim_G v''$.

Given an equivalence relationship, we may partition the set via the equivalence relationship, which is consisted of nonempty pairwise disjoint sets of nodes while the union of these sets is V.[4]
Now, let $v \in V$ be arbitrary, by assumption v is connected with at least $n/2$ nodes in G, so the partition of V by v, denoted $[v]_{\sim_G}$, contains at least $n/2 + 1$ nodes.
For the sake of contradiction, suppose that there exists another node $v' \in V \setminus [v]_{\sim_G}$, then $[v']_{\sim_G}$ must be a disjoint partition, *i.e.*:

$$[v]_{\sim_G} \cap [v']_{\sim_G} = \emptyset.$$

However, since v' is connected with at least $n/2$ other nodes, so $[v']_{\sim_G}$ must also contain at least $n/2 + 1$ nodes.
Given that there is a total of n nodes and $n/2 + 1 + n/2 + 1 = n + 2 > n$, there are more nodes in two disjoint subsets of a set with n nodes, which is a contradiction. □

[4] This is a conclusion from CS230 Mathematical Foundation for Computer Science.

Thus, we have proven that such strategy guarantee the network to be connected. ⌟

Problem II.2. Network Latency **[6 points]**

In large wireless networks, latency is often experienced in communication. For simplicity, assume that the network graph contains no cycles (that is, it forms a tree structure), so that every pair of devices will be communicating via a unique path. We define the network latency to be the distance between furthest pair of devices.

More formally, let the network be represented by an undirected tree $T = (V, E)$ where V represents the devices and E represents the edges. The network latency L is define to be the maximum distance $l(v, w)$ between two vertices $v, w \in V$, where the distance $l(v, w)$ is the number of edges on the path from v to w.

(a) Describe an algorithm to compute the network latency for T in $\mathcal{O}(|V|)$ steps. [4 points]

Solution. Here, we provide the following algorithm.

Let $v \leftarrow$ arbitrary node in V.
Conduct BFS on v, let $w \leftarrow$ the last node explored by the BFS starting from v.
Conduct BFS on w, **return** the highest level of the BFS starting from w.

(b) Give a brief justification for the correctness of your algorithm. [1 point]

Proof. For the sake of contradiction, suppose that there exists a pair $(a, b) \in V \times V$ that exhibits longer distance between the distance from our algorithm in part (a).
Since we have done a BFS on v and the network is connected, there must exist some $c, d \in V$ such that $l(a, c) \leq l(c, w)$ and $l(b, d) \leq l(d, w)$, namely illustrated as below:

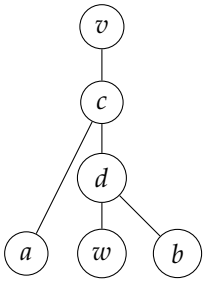


Figure II.1. Demonstration of the abstract case. Note: each edge in the above diagram might have varing length.

Note that here, v, c, d are not necessarily distinct (they could coincide) and the order of c, d may not be as illustrated in the figure. Without loss of generality, we suppose that $l(v, c) \leq l(v, d)$, since we can simply make a transposition between a and b in our case. Note that the path is unique, then the path from a to b must be a to c to d to b, but we have:

$$l(a, w) = l(a, c) + l(c, d) + l(d, w) \geq l(a, c) + l(c, d) + l(d, b) = l(a, b).$$

Hence, when we conduct BFS from w, we must find the path $l(a, w)$ that is no shorter than $l(a, b)$, which is a contradiction to assumption that $l(a, b)$ exhibits longer distance compared to the BFS result from w.
Thus, we have shown that the algorithm is valid. □

(c) Analyze the runtime and show that it is $\mathcal{O}(|V|)$. [1 point]
Hint: You may think of using BFS.

Proof. Recall that the runtime for BFS is $\mathcal{O}(|V| + |E|)$,[5] since our (undirected) graph is connected and contains no cycle, it has $|V| - 1$ edges, so the runtime of each BFS is $\mathcal{O}(2|V|) = \mathcal{O}(|V|)$.
Since we run the BFS twice, so the total runtime is $\mathcal{O}(2|V|) = \mathcal{O}(|V|)$, as desired. □

[5] This is a conclusion from CS226 Data Structures.

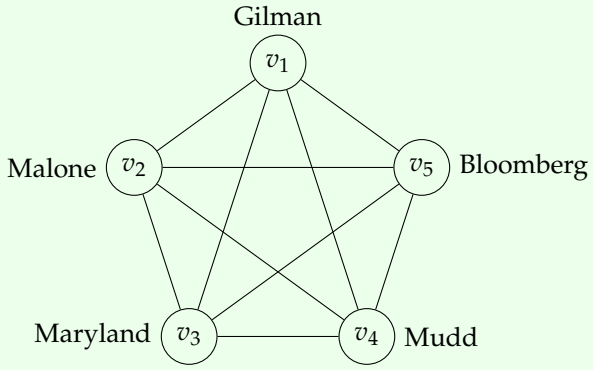
Problem II.3. JHU Campus Tour **[6 points]**

The year is 2092. Spring is approaching, and JHU is preparing to hold campus tours. This year, you are responsible for leading a tour, and you have received a campus map from the university. You realize two interesting observations: the good news is every pair of buildings are adjacent, i.e., they are directly connected by a skywalk, but the bad news is that all sidewalks are one-way paths due to ongoing construction (some things never change).

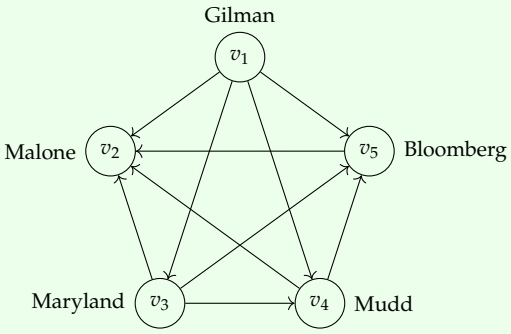
More formally, the campus can be represented by a directed graph $G = (V, E)$ where each vertex represents a building and each edge represents a one-way skywalk, and for every pair of $v, w \in V$, precisely one of the directed edges (v, w) or (w, v) is contained in E.

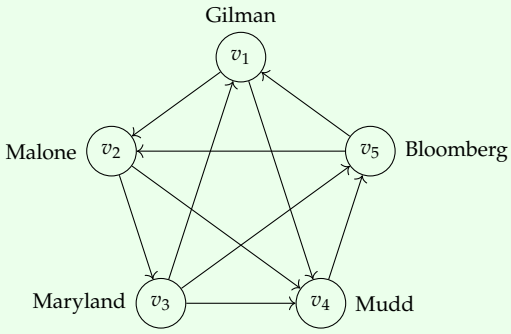
You would like to design a tour path so you can lead the tour to visit every building exactly once. You can start from any building, then visit every building exactly once, you don't visit any building twice, you don't come back to the building from where you started.

Consider the toy example below. Suppose we have 5 buildings only, then a campus map is a graph obtained by assigning a direction to each edge of the following prototype:



For the example on the left side below, one could start the tour from Gilman, then goto Maryland, then to Mudd, then to Bloomberg and finally to Malone.
For the example on the right side below, one could start the tour from Malone, then goto Maryland, then goto Mudd, then to Bloomberg and finally to Gilman. This graph has many more possibilities, especially since the 5 vertices form a cycle: you could start from any node.





You need to prove that this is always possible, for a graph of any size. More formally, let $G = (V, E)$ be an undirected **complete graph** on n vertices, that is, there is an edge between every pair of nodes. For each

edge, toss a coin and randomly assign a direction so that it becomes a directed graph. This graph is such that for each $v, w \in V$, exactly one of (v, w) and (w, v) is contained in E. Prove that G always has a simple path visiting each vertex exactly once.

Solution. For this question, we prove with mathematical induction with $n \in \mathbb{N}_{\geq 2}$ representing the number of nodes. We want to show that whatever arrangement, all the undirected complete graphs with n nodes assigned with any directions contains a simple path through all the nodes without repetition.

Proof.

- (Base case:) For $n = 2$, the edge is always from one direction to another, namely for nodes $v_1, v_2 \in V$, we can illustrate the cases as follows:

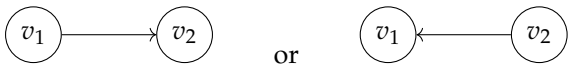


Figure II.2. 2 cases for the graph with two nodes.

hence the argument is trivially true for the base case.

- (Inductive step:) Now, suppose that the argument hold for all graphs with n nodes, where $n \in \mathbb{N}_{\geq 2}$, we want to show that all the undirected complete graphs with $n+1$ nodes assigned with any directions contains a simple path through all the nodes without repetition.
Here, we let an arbitrary node v be fixed, and the rest $n + 1 - 1 = n$ nodes has a directed graph, by the induction hypothesis, there exists a path from $\sigma(1), \cdots, \sigma(n)$ where $\sigma : \{1, \cdots, n\} \to \{1, \cdots, n\}$ is a permutation of the numbers (or bijection), now, we consider all the connections between v and $v_{\sigma(i)}$'s:

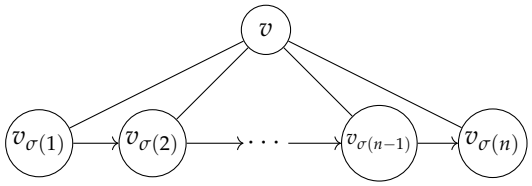


Figure II.3. Considering how to add the node v into the current path.

Here, if we have $v_{\sigma(1)} \leftarrow v$ or $v_{\sigma(n)} \to v$, we can simply add v to the start or end of the path, and we have constructed a working pathway.
Otherwise, if we have $v_{\sigma(1)} \to v$ and $v_{\sigma(n)} \leftarrow v$, there must exists some nodes in the middle where there is a difference in direction, namely:

$$v_{\sigma(i)} \to v \qquad \text{and} \qquad v_{\sigma(i+1)} \leftarrow v,$$

so we may construct the path:

$$v_{\sigma(1)} \to v_{\sigma(2)} \to \cdots \to v_{\sigma(i)} \to v \to v_{\sigma(i+1)} \to \cdots \to v_{\sigma(n-1)} \to v_{\sigma(n)},$$

which completes the inductive step. □

Hence, we have shown that G always has a simple path visiting each vertex exactly once. ⌟

Problem II.4. Some Short Paths in a Graph **[6 points]**

A number of art museums around the country have been featuring work by an artist named Mark Lombardi (1951–2000), consisting of a set of intricately rendered graphs. Building on a great deal of research, these graphs encode the relationships among people involved in major political scandals over the past several decades: the nodes correspond to participants, and each edge indicates some type of relationship between a pair of participants. And so, if you peer closely enough at the drawings, you can trace out ominous-looking paths from a high-ranking U.S. government official, to a former business partner, to a bank in Switzerland, to a shadowy arms dealer.

Such pictures form striking examples of social networks, which, as we discussed in book chapter 3.1, have nodes representing people and organizations, and edges representing relationships of various kinds. And the short paths that abound in these networks have attracted considerable attention recently, as people ponder what they mean. In the case of Mark Lombardi's graphs, they hint at the short set of steps that can carry you from the reputable to the disreputable.

Of course, a single, spurious short path between nodes v and w in such a network may be more coincidental than anything else; a large number of short paths between v and w can be much more convincing. So in addition to computing a single shortest v-w path in a graph G, social networks researchers have looked at the problem of determining the number of shortest v-w paths.

Suppose we are given an undirected graph $G = (V, E)$, and we identify two nodes v and w in G.

(a) Design an algorithm that computes the number of shortest v-w paths in G. The algorithm should not list all the paths; just the *number* of shortest paths. Briefly justify why your algorithm works. [4 points]

Solution. First, we briefly give the following algorithm.

Let $S \leftarrow \emptyset$ be the set of all explored vertices.[6]
Let $\varphi : V \to \mathbb{N}$ be a function mapping each vertex to the number of shortest way to reach it, and initialize all map to 0, *i.e.*, $\varphi(x) = 0$ for all $x \in V$.[7]
Let $\{\mathcal{C}_i\}_{i=0}^{n}$ be a ordered sequence of sets, we denote $\mathcal{C}_i$ for the i-th set, while the $\mathcal{C}_0$ is the first set.
Let $\ell \leftarrow 0$, let $\mathcal{C}_\ell \leftarrow \{v\}$, let $S \leftarrow S \cup \mathcal{C}_\ell$, and let $\varphi(v) = 1$.
while w is not explored and $\ell \leq n$ **do**
  for all $x \in \mathcal{C}_\ell$ **do**
    for all $y \leftarrow$ neighbor of x such that $y \notin S$ **do**
      Let $\mathcal{C}_{\ell+1} = \mathcal{C}_{\ell+1} \cup \{y\}$.
      Let $\varphi(y) \leftarrow \varphi(y) + \varphi(x)$.
    end for
  end for
  Let $S \leftarrow S \cup \mathcal{C}_\ell$, and let $\ell \leftarrow \ell + 1$.

[6] Given a finite collection of vertices, there are always ways to index the vertices properly to develop a `hash set` type of data structure so that the add and check are of constant time.

[7] Given a finite collection of vertices, there are always ways to index the vertices properly to develop a `hash map` type of data structure so that the modifications are of constant time.

end while
return $\varphi(w)$ as the number of shortest path.

Then, we want to show the validity of the algorithm.
The algorithm adds more features than the BFS while not changing any structure of BFS, so it will be capable of finding the shortest path of v-w if it exists, otherwise, it would return 0 as the initialized value for the map φ.
The algorithm uses a recursive structure, *i.e.*, adding the number of shortest path as the sum of shortest paths to all the parents on the previous level, since the total number of shortest ways to approach must approach the parents with one less step at the same time, so it is a sum.
Now, we shall give a formal proof as well.

Proof. First, it is clear that the algorithm will terminate since the while loop has the cap condition. It will return 0 if w is never explore, *i.e.*, not connected with v.
Then, we can prove that the count by adding the previous terms via induction on the number of levels $k \in \mathbb{N} \cap [1, n]$:

- (Base case:) Consider the first level, the only (shortest) way to proceed their is directly through v and it gets $f(v) = 1$, so we are good.
- (Inductive step:) Assume that we use the algorithm to obtain correct number of ways to get to a node on level $m \in \mathbb{N} \cap [1, n-1]$, we want to show that this works for all nodes in level $m+1$, since we are considering the minimum, we shall only consider nodes in $\mathcal{C}_m$ since there would have been faster ways to get to all previous sets or would not have been explored, the number of ways to approach would be exactly the sum of ways of distinct parents in $\mathcal{C}_m$ to get to the node in $m+1$, so the algorithm is working in summing the values.

Thus, the algorithm would give the right number of shortest paths. □

Now, the algorithm will be returning the correct number of shortest paths. ⌟

(b) Show that your algorithm runs in $O(m+n)$ for a graph with n nodes and m edges. [1 point]

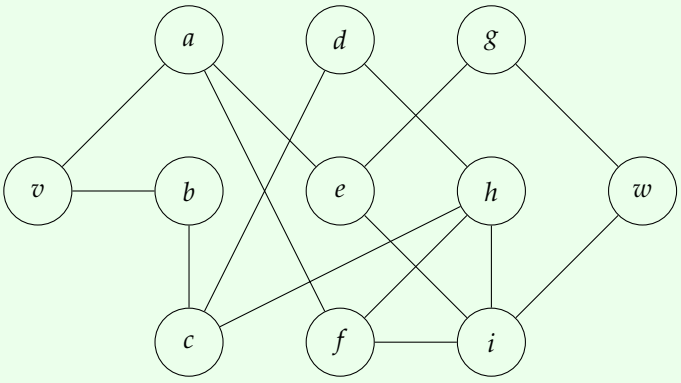
Proof. By assumption, we assume some set and map operation in which add, retrieve and modification is of $\mathcal{O}(1)$. The initial set of are constant time and initializing the map f would be $\mathcal{O}(n)$ for n nodes.
Then, the while loop would run until w is explored or at most n times, and within the while loop, the for loops are exploring all the nodes and all their neighbors, and we have the sum of all nodes multiplying their neighbors of $\mathcal{O}(2m) = \mathcal{O}(m)$, *i.e.*, the number of edges.[8] As all operations within the for loops are assumed to be constant time, so the total run time is:

$$\mathcal{O}(m) + \mathcal{O}(n) = \mathcal{O}(m+n),$$

as desired. □

[8] This is a conclusion from CS226 Data Structures.

(c) Demonstrate your algorithm on the graph given below by computing the number of shortest paths between v and w. [1 point]



Solution. To demonstrate the step of the algorithm, we will keep track of S, $\mathcal{C}_i$, and f for each ℓ.

- $\ell = 0$: $\varphi(v) = 1$, $S = \{v\}$, $\mathcal{C}_0 = \{v\}$.
- $\ell = 1$: $\varphi(b) = \varphi(a) = 1$, $S = \{v, a, b\}$, $\mathcal{C}_1 = \{a, b\}$.
- $\ell = 2$: $\varphi(c) = \varphi(e) = \varphi(f) = 1$, $S = \{v, a, b, c, e, f\}$, $\mathcal{C}_2 = \{c, e, f\}$.
- $\ell = 3$: $\varphi(d) = \varphi(g) = 1, \varphi(h) = \varphi(i) = 2$, $S = \{v, a, b, c, e, f, d, g, h, i\}$, $\mathcal{C}_3 = \{d, g, h, i\}$.
- $\ell = 4$: $\varphi(w) = 3$, $S = V$, $\mathcal{C}_4 = \{w\}$.

Hence, the algorithm observes $\boxed{3}$ shortest paths from v to w. ⌟

Problem II.5. All paths go through r_{ome} **[6 points]**

There's a natural intuition that two nodes that are far apart in a communication network – separated by many hops – have a more fragile connection than two nodes that are close together. There are a number of algorithmic results that are based to some extent on different ways of making this notion precise. Here's one that involves the vulnerability of paths to the deletion of nodes.

Suppose that an n-node undirected graph $G = (V, E)$ contains two nodes s and t such that distance $l(s, t) > \frac{n}{2}$. Either prove or disprove that there exists a node r, such that deleting r from G will destroy all paths between s and t. Note that r is different from nodes s and t.

Solution. Here, we will **prove** this argument, that is, there exists a node r such that deleting r from G will destroy all paths between s and t, where r is not s or t.

Proof. For the sake of contradiction, suppose that removing any point other than s and t would *not* cause s and t to be disconnected, in order for s and t to be having shortest path being $> n/2$, we need at least $k := \lfloor n/2 \rfloor$ number of nodes between s and t to construct this skeleton, say:

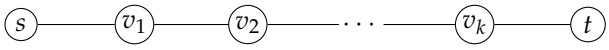


Figure II.4. Skeleton of a path from s to t.

Now, suppose that we do a BFS from s, we know that s is level 0, and there must be at least $\lfloor n/2 \rfloor$ levels between s and t and t must be of level at least $\lfloor n/2 \rfloor + 1$, namely, as follows:

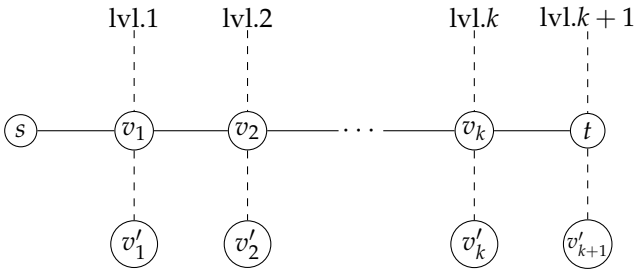


Figure II.5. Present the graph as a BFS tree, with only edges on the same level or on adjacent levels.

Note that on the BFS tree, there will be only edges on the same level or between adjacent level. Here, we claim that there must be at least two nodes on the same level. This can be proven by contradiction. Suppose there is only one node on any level from level 1 to level k, as long as we remove the only node on that level, then s and t will be disconnected since there are no edges that connects nodes that are separated by at least a level in the middle, which is a contradiction. Thus, there must be at least $2k$ nodes in the middle, adding the two nodes s and t, we need at least:

$$2k + 2 = 2\lfloor n/2 \rfloor + 2.$$

Note that if n is even, we have a total of $n + 2$ nodes, and if n is odd, we have $2\big((n-1)/2\big) + 2 = n + 1$, so there will be more than n nodes needed for constructing a graph of n nodes, hence it is a contradictions. □

Hence, there always exists a node r such that deleting r from G will destroy all paths between s and t. ⌟

Problem II.6. Give an algorithm to detect whether a given undirected graph contains a cycle. If the graph contains a cycle, then your algorithm should output a cycle. It should not output all the cycles in the graph, just one cycle. The running time of your algorithm should be $O(m+n)$ for a graph with n nodes and m edges.

Solution. For this problem, we would want to conduct a BFS that reports cycle when an explored node is explored again. The algorithm will be as follows:

Let $v \leftarrow$ arbitrary node in the graph.
Let $E \leftarrow \emptyset$ be the set of all explored nodes.[9]
Let $f : V \to V$ be a function that maps each node to its parent in the algorithm.[10]
Let Q be an empty priority queue.
`Push` v into Q, and let $E \leftarrow E \cup \{v\}$.
while Q is not empty **do**
  Let $n \leftarrow$ `pop` of Q.
  for all $w \in \text{Neighbor}(n)$ **do**
    if $w \in E$ **then**
      Break while loop, the graph contains a cycle, and remember the node n and w.
    end if
    Let $E \leftarrow E \cup \{w\}$ and `push w` to Q.
  end for
end while
if the graph contains a cycle **then**
  Retrieve the parent ancestors from w and n with f, say with lists L_w and L_n, that gets the parents of nodes recursively until reaching v.
  From the end of L_w and L_n, truncate them until the next from the last is different from L_w and L_n, then **return** L_w, L_n (in inverse), and then n to w as the cycle.
else
  return There is no cycle(s).
end if

Then, we prove that this algorithm works.

Proof. Suppose there is no cycle, then there is a unique path from v to each point, so each node cannot be explored twice.
Suppose there is a cycle, when we apply the BFS, there must exists a highest level. If that level contains at least two nodes in this cycle, then one of the node will be explored twice, once by its parent and once by the other node in the cycle that is on the same level. If that level contains only one node in this cycle, then it must have two parents on the lower level, so it will be explored twice.
Hence, having a cycle is equivalently that there exists a point that will be explored twice.
To show the processing cycle part, not that L_w and L_n must both have v, possible same points to some v_i,

[9] Typically, we use a `hash set` here, so the average runtime to adding, checking, and removing is $\mathcal{O}(1)$.
[10] Typically, we use a `hash map` here, so the average runtime to adding and modifying is $\mathcal{O}(1)$.

then completely different nodes after that, or else it would have terminated earlier. Then, our algorithm truncate up to v_i (keeping it), and then what it returns is the cycle. □

In terms of the algorithm, we now analyze the runtime:

Proof. Note that the first while loop is basically a BFS, so its runtime is $\mathcal{O}(|V| + |E|) = \mathcal{O}(m + n)$, and for the retrieving the cycle part, the two lists will be no longer $2|V| \in \mathcal{O}(m)$, and the truncation is with in $\mathcal{O}(m)$, and reporting it will be within $3\mathcal{O}(m) = \mathcal{O}(m)$.
Thus, the total runtime is of $\mathcal{O}(m + n) + \mathcal{O}(m) = \mathcal{O}(m + n)$, as desired. □

Now, we have the algorithm ready and justified.

Remark. This problem can also be done using DFS, in which the finding cycle algorithm could be more efficient, but the main idea is also to find the parents. ⌟

Problem II.7. There are two types of professional wrestlers: good guys and bad guys. Between any pair of professional wrestlers, there may or may not be a rivalry. Suppose we have n professional wrestlers and we have a list of r pairs of wrestlers for which there are rivalries. Give an $O(n+r)$ algorithm that determines whether it is possible to designate some of the wrestlers as good guys and the remainder as bad guys such that each rivalry is between a good guy and a bad guy. If it is possible to perform such a designation, your algorithm should produce it.

Solution. This question is basically another modification on BFS. Let's state the algorithm again.

Let $E \leftarrow \emptyset$ be the set of all explored nodes.[11]
Let $f : V \to \mathbb{Z}/2\mathbb{Z}$ be a function that maps node to number, initialize the function to all as undefined.[12]
while $E \neq V$ **do**
  Let $v \leftarrow$ arbitrary node in $V \setminus E$.
  Let Q be an empty priority queue.
  Let $E \leftarrow E \cap \{v\}$, and `push` v into Q, and let $f(v) = 0$.[13]
  while Q is not empty **do**
    Let $n \leftarrow$ `pop` of Q.
    for all $w \in \text{Neighbor}(n)$ **do**
      if $f(w)$ is undefined **then**
        Let $f(w) \leftarrow f(n) + 1$.
      else if $f(w) \neq f(n) + 1$ **then**
        return there is no solution.
      end if
      if $w \notin E$ **then**
        Let $E \leftarrow E \cup \{w\}$, and `push` `w` to Q.
      end if
    end for
    end while
  end while
  return $f^{-1}(0)$ as good guys and $f^{-1}(1)$ as bad guys.

Then, we verify the validity of the algorithm.

Proof. Basically, the BFS explores the connected proportion of the tree and can terminate, and we enforce all points to be explored, whereas the check of $f(w) \neq f(n) + 1$ makes sure that for any edge in this tree, it will check that all marked neighbors are compliant. □

Then, we analyze the runtime.

Proof. Note that each node analyze all its neighbors once through the while loop, so it is $\sum_{n \in N} \deg(n) = r$ and since we do this until all nodes are explored, it is $\mathcal{O}(n+r)$, and the initialization and retrieve requires getting around all nodes, which is $\mathcal{O}(n)$, so the total runtime is $\mathcal{O}(n+r)$, as desired. □

[11] Typically, we use a `hash set` here, so the average runtime to adding, checking, and removing is $\mathcal{O}(1)$.
[12] Typically, we use a `hash map` here, so the average runtime to adding and modifying is $\mathcal{O}(1)$.
[13] For simplicity, we denote $0 := [0]$ and $1 := [1]$, hence $0 + 1 = 1$ and $1 + 1 = 0$.

Now, we have the algorithm ready and justified.

Remark. We may consider the BFS algorithm with layers $L_0, L_1, \cdots, L_k$, as we denote the neighbors of L_i level as neighbor(L_i), if neighbor$(L_i) \cap L_i \neq \emptyset$, there is an error, whereas this can be checked using `hash sets` in $\mathcal{O}(|L_i|)$ which sums to $\mathcal{O}(n)$, then we can color the layers by parity. ⌟

Problem II.8. Let G be a tree. Use induction to prove that we can color vertices of G with 2 colors such that the endpoints of every edge have different colors.

Solution. We prove by mathematical induction of the number of nodes of the tree as $n \in \mathbb{Z}^+$.

Proof.
- (Base case:) For tree of 1 node, we may color it as a color.
- (Inductive step:) Suppose all trees with $k \in \mathbb{Z}^+$ nodes can be colored with 2 colors such that the endpoints of every edge have different colors, then for any tree with $k+1$ nodes, it is a tree with k nodes and a node connected to a single node (this can be found using BFS), as long as we color it with a different color, we are fine, so all trees with $k+1$ nodes can be colored in two colors so that the endpoints of all edges have different colors.

Thus, we finish the induction proof. □

Remark. An alternative proof can be made using **strong induction**, in which the selection of a node in the inductive step is arbitrary, since you can think of each connected branch as color-able, and invert colors if needed. ⌟

Problem II.9. Given a connected undirected graph $G = (V, E)$ with $n = |V|$ vertices and $m = |E|$ edges, design an $O(m + n)$ time algorithm to find a vertex in G whose removal does not disconnect G. Note that as a consequence, this algorithm shows that every connected graph contains such a vertex.

Solution. Again, this question shall utilize a BFS.

Let $x \leftarrow$ an arbitrary node in V.
Conduct BFS on x, let $v \leftarrow$ the last node explored.
return v as the node to be removed.

We first show the validity of the algorithm.

Proof. By BFS, v is a node of the highest level, it would not be connected to any node of a deeper level. All other nodes of the same level of v must be connected to some node in the previous level, and all the nodes of previous levels are connected to all other points through earlier levels. □

In terms of runtime.

Proof. We just conduct a BFS, which is $\mathcal{O}(|V| + |E|) = \mathcal{O}(m + n)$. □

Now, we have the algorithm ready and justified. ⌟

Problem II.10. Let $G = (V, E)$ be a directed graph, let H be the graph of the strongly connected components (SCC) of G, prove that H is a DAG, *i.e.*, there is a topological order.

Solution. Here, we can prove the statement using a contradiction.

Proof. For the sake of contradiction, suppose that H is not a DAG, *i.e.*, there exists a cycle of various strong connected components, say $H_1, H_2, \cdots, H_k$, for $k \geq 2$. Since there exists a cycle, for any $h, h' \in \bigsqcup_{i=1}^{k} H_i$, there exists a path from h to h' and from h' to h (otherwise, if there is no path, without loss of generality, from $h \in H_\alpha$ to $h' \in H_\beta$, then H_α and H_β is not strongly connected in H), hence, we know that $\bigsqcup_{i=1}^{k} H_i$ contains all H_i for $i = 1, \cdots, k$, which violates the condition that the strongly connected components have to be maximal, *i.e.*, there should not be any superset of the strongly connected component that is still strong connected. This is a contradiction. □

Hence, we have shown that H is a DAG, and equivalently, it exhibits a topological order. ⌟

III Problem Set 3

Problem III.1. Scheduling Computation **[6 points]**

The wildly popular Spanish-language search engine El Goog needs to do a serious amount of computation every time it recompiles its index. Fortunately, the company has at its disposal a single large supercomputer, together with an essentially unlimited supply of high-end PCs.

They've broken the overall computation into n distinct jobs, labeled $J_1, J_2, \ldots, J_n$, which can be performed completely independently of one another. Each job consists of two stages: first it needs to be pre-processed on the supercomputer, and then it needs to be finished on one of the PCs. Lets say that job J_i needs p_i seconds of time on the supercomputer, followed by f_i seconds of time on a PC.

Since there are at least n PCs available on the premises, the finishing of the jobs can be performed fully in parallel – all the jobs can be processed at the same time. However, the supercomputer can only work on a single job at a time, so the system managers need to work out an order in which to feed the jobs to the supercomputer. As soon as the first job in order is done on the supercomputer, it can be handed off to a PC for finishing; at that point in time a second job can be fed to the supercomputer; when the second job is done on the supercomputer, it can proceed to a PC regardless of whether or not the first job is done (since the PCs work in parallel); and so on.

Let's say that a schedule is an ordering of the jobs for the supercomputer, and the completion time of the schedule is the earliest time at which all jobs will have finished processing on the PCs. This is an important quantity to minimize, since it determines how rapidly El Goog can generate a new index.

(a) Give a polynomial-time algorithm that finds a schedule with as small a completion time as possible. [3 points]

Solution. Here, we first give an algorithm for this problem.

Let $\mathcal{J} \leftarrow \{J_i\}_{i=1}^n$.
Sort $\mathcal{J}$ with respect to f_i in descending order, *i.e.*, $J_i \prec J_j$ if and only if $f_i \leq f_j$.[14] [15]
return $\mathcal{J}$ (head-to-tail, start J_{i+1} on supercomputer immediately after the supercomputer finishes J_i, and start J_i on PC immediately after the supercomputer finishes J_i) as the desired schedule.

In terms of its runtime, initializing an ordered list is $\mathcal{O}(n)$ and the sort is $\mathcal{O}(n \log n)$, so the total runtime is $\mathcal{O}(n \log n) \subset \mathcal{O}(n^2)$, which is in polynomial-time. ⌟

(b) Prove that your algorithm results in an optimal solution (it needs to be a formal proof using one of the 3 methods studied to prove optimality of Greedy Algorithms). [3 points]

Solution. Then, we prove the validity of the above given algorithm, step by step.
First, we note that the optimal schedule should have no *idle time* on the supercomputer, and the PC should

[14] The sort can be done using `Heap sort` or `Merge sort`, which runs $\mathcal{O}(n \log n)$.
[15] For the case $f_i = f_j$, the order does not matter and can be arbitrary.

work on the job right after the supercomputer finishes. This is exactly the same as our construction. Then, we similarly define the *inversion* in this context.

Definition. (Inversion). Given any schedule $\mathcal{J}$, an inversion is a pair of jobs J_i and J_j such that $f_i > f_j$ but J_j is scheduled before J_i on the supercomputer.

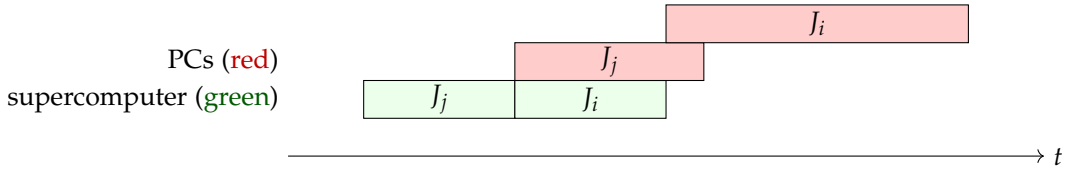


Figure III.1. An example of inversion, $f_i > f_j$ but J_j is scheduled before J_i.

Then, we want to extend the existence of inversion to the existence of adjacent inversion.

Lemma. (Inversion $\Longrightarrow$ Adjacent Inversion). If an idle-free schedule has an inversion, it has an adjacent inversion.

Proof. Now, suppose that J_i-J_j is the closest non-adjacent inversion. Let J_k be the job on the right after J_j.

- If $f_k > f_j$, then J_k-J_j is an adjacent inversion.
- If $f_k \leq f_j$, then J_i-J_k is a closer inversion, since $f_k \leq f_j < f_i$, which is contradiction.

Hence, if there is an inversion, then there is an adjacent inversion. □

Lemma. Exchanging two adjacent inversion jobs does not make the latest finish time later.

Proof. Consider the J_i-J_j inversion, *i.e.* $f_i > f_j$, then the schedules of them would be as:

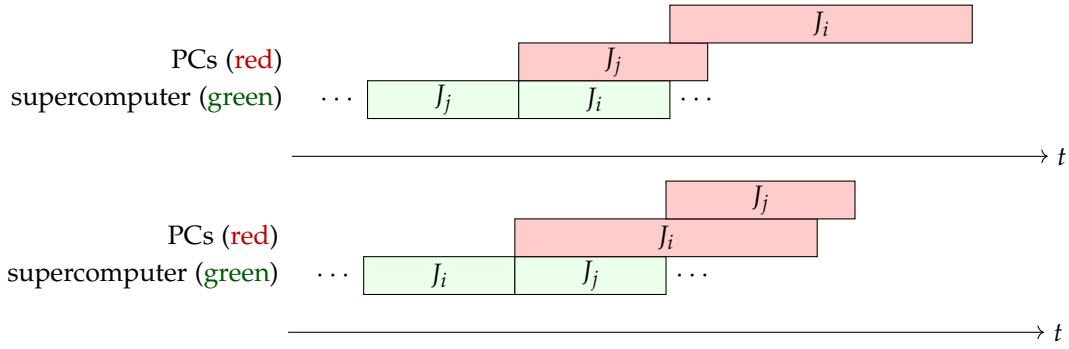


Figure III.2. Schedule with adjacent inversion (above), and after exchange (below).

It is clear that with the adjacent inversion, the finishing time of those two jobs is the start time adding $p_j + p_i + f_i$ and the finishing time of the exchanged version is the start time adding $p_i + p_j + f_j$. Since $f_i > f_j$, then:

$$p_j + p_i + f_i > p_i + p_j + f_j,$$

hence the exchange of the inversion would not make the latest finish time later. □

Then, we shall proceed to the proof of our main argument. It can be noted that the provided algorithm has no inversions.

Proposition. The provided algorithm is optimal.

Proof. Suppose that $\mathcal{J}^*$ is the optimal schedule with the fewest inversion. Without loss of generality, suppose $\mathcal{J}^*$ has no idle time.

- If $\mathcal{J}^*$ has no inversion, then $\mathcal{J}$ and $\mathcal{J}^*$ are equivalent up to events with the same runtime on PCs.
- If $\mathcal{J}^*$ has an inversion, then it has an adjacent inversion, and exchanging the inversion would decrease the inversion without increasing the total finish time, so it contradicts that $\mathcal{J}^*$ has the fewest inversion.

□

Therefore, we have shown the optimality of our algorithm. ⌟

Problem III.2. A Greedy Time Series **[6 points]**

Some of your friends have gotten into the burgeoning field of time-series data mining, in which one looks for patterns in sequences of events that occur over time. Purchases at stock exchanges – whats being bought are one source of data with a natural ordering in time. Given a long sequence S of such events, your friends want an efficient way to detect certain patterns in themfor example, they may want to know if the four events

buy Yahoo, buy eBay, buy Yahoo, buy Oracle

occur in this sequence S in order, but not necessarily consecutively.
They begin with a collection of possible events (e.g., the possible transactions) and a sequence S of n of these events. A given event may occur multiple times in S (e.g., Yahoo stock may be bought many times in a single sequence S). We will say that a sequence S' is a subsequence of S if there is a way to delete certain events from S so that the remaining events, in order, are equal to the sequence S'. So, for example, the sequence of four events above is a subsequence of the sequence

buy Amazon, buy Yahoo, buy eBay, buy Yahoo, buy Yahoo, buy Oracle

Their goal is to be able to dream up short sequences and quickly detect whether they are subsequences of S. So this is the problem they pose to you:

Give a greedy algorithm that takes two sequences of events – S' of length m and S of length n, each possibly containing an event more than once – and decides in time $O(m+n)$ whether S' is a subsequence of S.

Provide a proof of runtime for your greedy algorithm, proof of correctness is not required.

Solution. For the sake of consistency, we use that each list/array starts with index 1. We provide the following algorithm.

Let $v \leftarrow 1$ and $w \leftarrow 1$.
while $v \leq m$ and $w \leq n$ **do**
  if $S'_v = S_w$ **then**
    Let $v \leftarrow v+1$ and $w \leftarrow w+1$.
  else
    Let $w \leftarrow w+1$.
  end if
end while
if $v > m$ **then return** S' is a subsequence of S.
elsereturn S' is not a subsequence of S.
end if

The contents inside the while loop is constant time, and for the while loop, at worst, the index (or the pointer to the element) will just traverse S and S', so the total runtime is $\mathcal{O}(m+n)$, as desired. ⌟

Problem III.3. CluNet has no clue **[6 points]**

A group of network designers at the communications company CluNet find themselves facing the following problem. They have a connected graph $G = (V, E)$, in which the nodes represent sites that want to communicate. Each edge e is a communication link, with given available bandwidth b_e.

For each pair of nodes $u, v \in V$, they want to select a single $u - v$ path P on which this pair will communicate. The bottleneck rate $b_{(P)}$ of this path P is the minimum bandwidth of any edge it contains; that is, $b_{(P)} = \min_{e \in P} b_e$. The best achievable bottleneck rate for the pair (u, v) in G is simply the maximum, over all $u - v$ paths P in G, of the value $b_{(P)}$.

It's getting to be very complicated to keep track of a path for each pair of nodes, and so one of the network designers makes a bold suggestion: Maybe one can modify graph G and then find its spanning tree T so that for every pair of nodes (u, v), the unique $u - v$ path in the tree actually attains the best achievable bottleneck rate for (u, v) in G. (In other words, even if you could choose any $u - v$ path in the whole graph, you couldn't do better than the $u - v$ path in T.)

This idea is roundly heckled in the offices of CluNet for a few days, and there's a natural reason for the skepticism: each pair of nodes might want a very different-looking path to maximize its bottleneck rate; why should there be a single tree that simultaneously makes everybody happy? But after some failed attempts to rule out the idea, people begin to suspect it could be possible.

Prove that such a tree exists, and give an efficient algorithm to find one. That is:

- Show how to modify graph G. [1 point]
- Give an algorithm constructing a spanning tree T in which, for each $u, v \in V$, the bottleneck rate of the $u - v$ path in T is equal to the best achievable bottleneck rate for the pair (u, v) in G. [2 points]

Solution. Here, we would want to modify the *blue rule* and the *red rule* in the MST algorithm.

Modified Blue Rule.

- Let D be a cutset with no blue edge.
- Select an uncolored edge of maximum bandwidth and color it blue.

Modified Red Rule.

- Let C be a cycle with no red edge.
- Select an uncolored edge of minimum bandwidth and color it red.

Then, we can provide the greedy algorithm.

---

while there are less than $|V| - 1$ blue edges or there are less than $|E| - |V| + 1$ red edges **do**
  Apply the modified blue rule or the modified red rule.

end while

Remark. Technically, we can just apply one of the *modified blue rule* or *modified red rule*, but both are provided here anyways.
Also, in particular, we can equivalently develop a modification on G as:

- For any $e \in E$, we define its weight as $w_e = \max_{\epsilon \in E}(b_\epsilon) - b_e$.
- Then let $\tilde{E}$ be the set of edges with the above weight, so $\tilde{G} = (V, \tilde{E})$.

Then, we can just use any MST algorithm on $\tilde{G}$, and the output MST is the tree as requested. ⌟

- Provide a formal proof to show the optimality (correctness) of your algorithm. You may assume that all the edge weights are distinct. [3 points]

Solution. For simplicity, we call the graph that contains the path of the maximum bottleneck rate in E for any pair (u, v), where $u, v \in V$, as *MBG*.[16] Also, we denote a spanning tree by ST.
Here, we develop the justification step-by-step, *i.e.*, we need a few separated claims.

Lemma. (Color Invariant). There exists a MBG and a ST containing every blue edge and no red edge.

Proof. We prove this inductively on the number of iterations $n \in \mathbb{N} := \mathbb{Z}_{\geq 0}$.

- Base case, $n = 0$: When none of the edges are colored, simply every MBG and ST satisfies the invariant.

Then, we turn to the inductive step, we suppose that color invariant is true, we shall discuss each case.

- Modified blue rule and existence of MBG: Consider a cutset with no blue edge, the uncolored edge of maximum bandwidth, as for that edge, the two endpoints has that edge as the maximum possible bandwidth connecting those two endpoints. Hence, coloring this edge blue make the graph still color invariant.
- Modified blue rule and existence of ST: Consider a cutset with non blue edge, adding coloring a blue edge would not form a cycle, as we need minimum two blue edges in a cutset to connect the two components into a cycle.
- Modified red rule and existence of MBG: Consider a cycle with no red edge, then we can get a path with larger minimum bandwidth from any endpoints in the cycle to another without using this minimum bandwidth edge in the cycle (just get around from the other edges in the cycle), for the edge with the minimum bandwidth being colored red, the MBG would still be kept as it not going through that newly added red edge would not delete the longer path.

[16] Note that this may not necessarily be a tree, but it is a spanning graph.

- Modified red rule and existence of ST: When we have a cycle and just mark one edge as red, *i.e.*, removing one edge, if the graph was connected, it will still be connected, as any path with that edge can simply get around the other kept edges in the cycle.

With these four cases shown, we have proven that the use of *modified blue rule* and *modified red rule* is color invariant. □

For the sake of the prove, we may temporarily ignore the early terminating condition for the while loop, as it is just ensure that what we have left is exactly a tree.

Lemma. (Termination of Algorithm). The greedy algorithm terminates.

Proof. First, we show that either the modified blue rule or the modified red rule applies.
Suppose an edge e is uncolored, the blue edges form a forest.

- If both endpoints of e belong to the same blue tree, we apply the modified red rule to the cycle inside that blue tree.
- If endpoints of e belong to different blue tree, we apply the modified blue rule to the cutset induced by either of the blue trees.

Hence, this the use of the modified blue rule and the modified red rule will run on until all the edges are colored. □

Then, we show a strong result of the above conclusion.

Lemma. (No Blue Cycle). After the algorithm terminates, the blue edges do not form a cycle.

Proof. This could also been shown using induction on the number of iterations $n \in \mathbb{N}$.

- Base case, $n = 0$: When none of the edges are colored, the blue edges form no cycles.
- Inductive step: Suppose after $k \in \mathbb{N}$ iterations, the blue edges form no cycle. Then, for the next step, since the modified blue rule enforces to color blue for a cutset with no blue edges in prior. Suppose that after the coloring, there forms a cycle with the blue edges, then the endpoints of the newly added blue edge were connected before applying the rule, so those two endpoints were in the same tree in the forest already and it contains no cutset in which the endpoints were separated and a all edges were not blue, which is contradiction.

Hence, the blue edges will never form a cycle. □

Proposition. After the algorithm terminates, the blue edges form a spanning tree that contains the bottle neck rate of any $u - v$ path equaling to the best achievable bottleneck rate for (u, v) in G.

Proof. By the previous 3 lemmas, we have shown that the algorithm terminates, the blue edges forms a tree by its termination, and that tree must contain a ST and a MBG, then that tree must be the one as desired. □

Hence, we have shown the validity of the algorithm and the existence of such tree. ⌟

Problem III.4. Timing VLSI **[6 points]**

Timing circuits are a crucial component of VLSI chips. Heres a simple model of such a timing circuit. Consider a complete balanced binary tree with n leaves, where n is a power of two. Note that *balanced* here means that each leaf will have exactly $\log_2(n)$ edges between the leaf and the root of the tree. However, the paths between the root and the leaf nodes can have different *weights*.

Each edge e of the tree has an associated weight l_e , which is a positive number. The *distance* from the root to a given leaf is the sum of the weights of all the edges on the path from the root to the leaf.

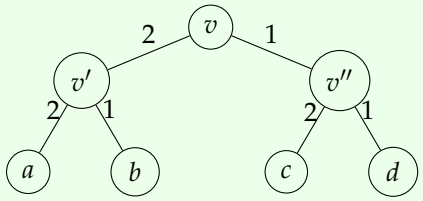


Figure III.3. An instance of the zero-skew problem, described in this problem. The unique optimal solution for this instance would be to take the three weight-1 edges and increase each of their weights to 2. The resulting tree has zero skew, and the total edge weight is 12, the smallest possible.

The root generates a clock signal which is propagated along the edges to the leaves. Well assume that the time it takes for the signal to reach a given leaf is proportional to the *distance* from the root to the leaf. Now, if all leaves do not have the same distance from the root, then the signal will not reach the leaves at the same time, and this is a big problem. We want the leaves to be completely synchronized, and all to receive the signal at the same time. To make this happen, we will have to increase the weights of certain edges, so that all the root-to-leaf paths have the same weight (were not able to decrease edge weights). If we achieve this, then the tree (with its new edge weights) will be said to have zero skew. Our goal is to achieve zero skew in a way that keeps the sum of all the edge weights as small as possible.

(a) Give an algorithm that increases the weights of certain edges so that the resulting tree has zero skew and the ***total edge weight is as small as possible***. You do not need to prove that your algorithm is correct or prove your algorithm's runtime. [4 points]

Solution. Here, we give the algorithm recursively.

---

function UPDATE_WEIGHT(v)
  if v is a leaf **then**
    return 0.
  end if
  Let $l \leftarrow$ UPDATE_WEIGHT(left child), $r \leftarrow$ UPDATE_WEIGHT(right child).
  if $l > r$ **then**
    add the minimum possible weight to one edge between v and the left or right child so that the edge connecting between v and the left children is $l - r$ less than the edge connecting between v and the right children.
  else
    add the minimum possible weight to one edge between v and the left or right child so that the

edge connecting between v and the left children is $r - l$ more than the edge connecting between v and the right children.

end if

return $l+$ weight of the edge connecting between v and the left children.

end function

Call UPDATE_WEIGHT on the root of the tree. ⌟

(b) Use your algorithm to compute the new edge weights on figure b. [2 points]

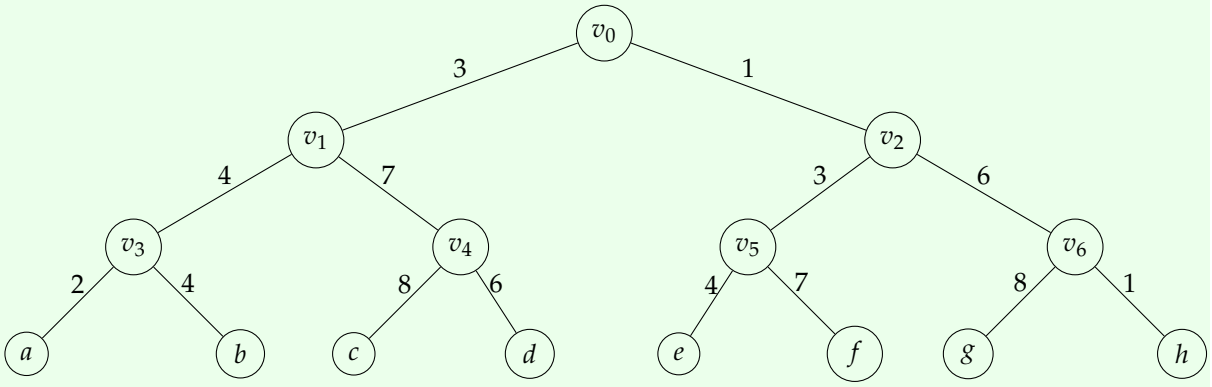
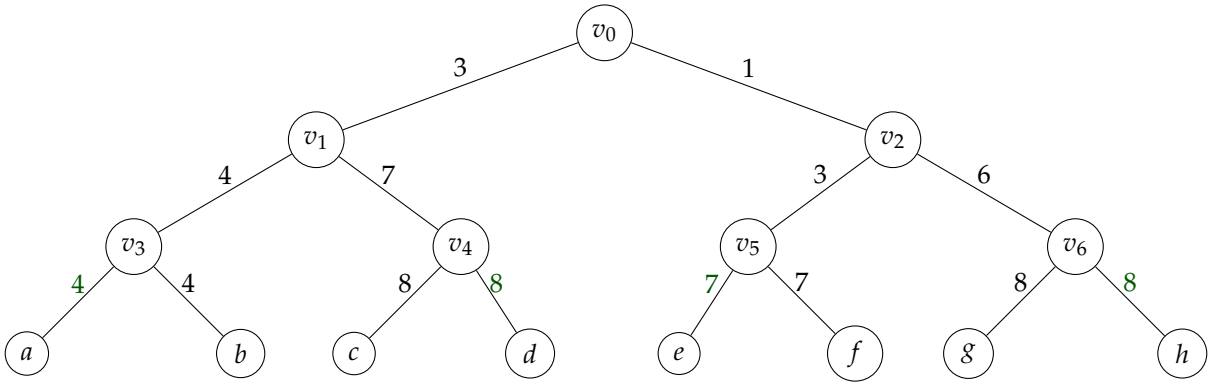


Figure III.4. Show your the new weights calculated by your algorithm on the above tree.

Solution. Here, we apply the algorithm recursively, and we will represent the result when each level is zero skews from the deepest.



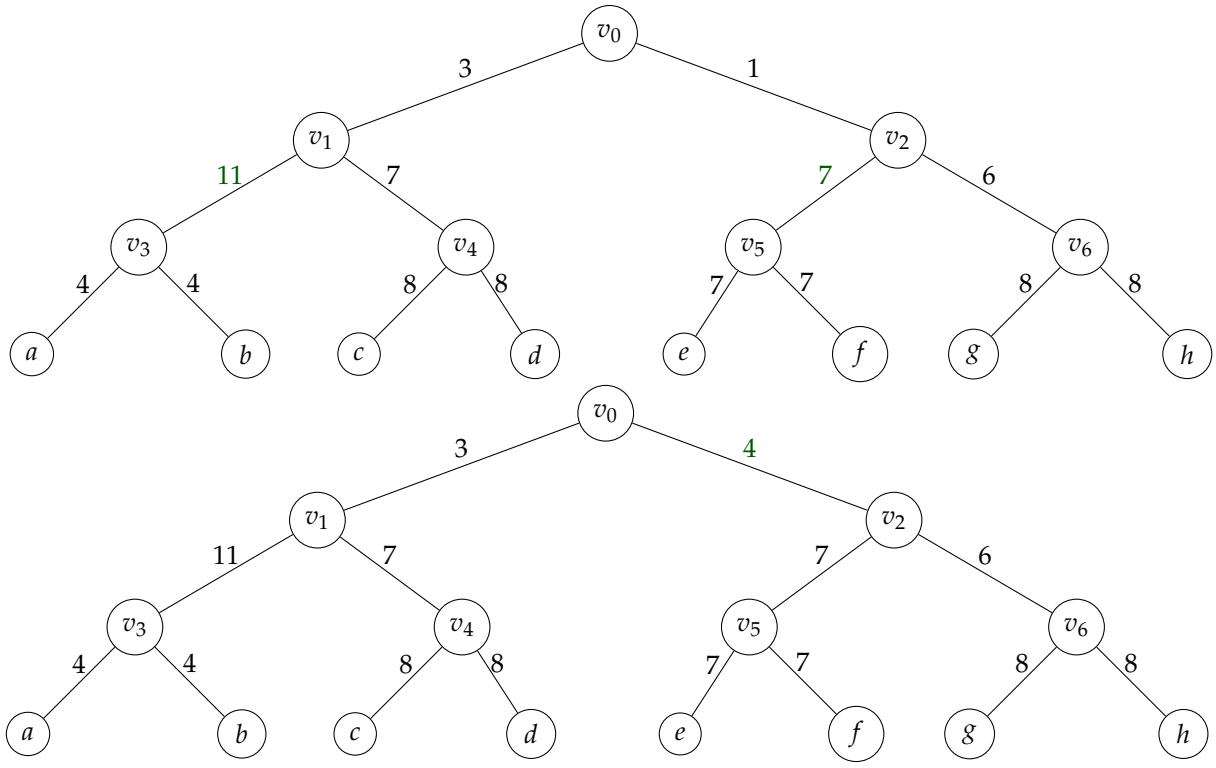


Figure III.5. Modifying the tree step by step, differences are marked in green.

The final result is (all changes were marked green):

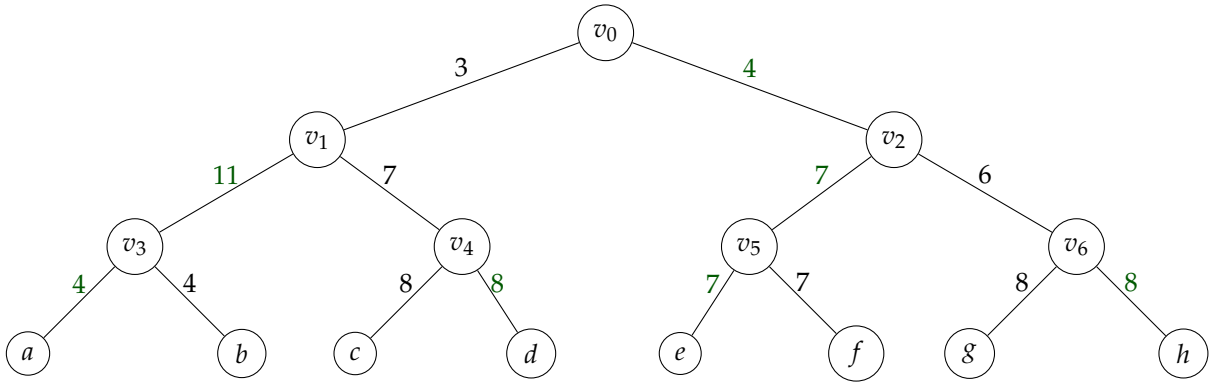


Figure III.6. Modified result, differences are marked in green.

⌟

Problem III.5. Better QuickSort Pivots? **[6 points]**

Recall that when using the QuickSort algorithm to sort an array A of length n, we picked an element $x \in A$ which we called the *pivot* and split the array A into two arrays A_S, A_L such that $\forall y \in A_S, y \leq x$ and $\forall y \in A_L, y > x$.

We will say that a pivot from an array A provides $t|n-t$ separation if t elements in A are smaller than or equal to the pivot, and $n-t$ elements are strictly larger than the pivot.

Suppose Bob knows a secret way to find a good pivot with $\frac{n}{3}|\frac{2n}{3}$ separation in constant time. At the same time, Alice knows her own secret technique, which provides separation $\frac{n}{4}|\frac{3n}{4}$, her technique also works in constant time.

Alice and Bob apply their secret techniques as subroutines in the QuickSort algorithm to pick pivots.

(a) Whose algorithm works **asymptotically** faster? Or are the runtimes **asymptotically** the same? [2 points]

Solution. The two algorithms are asymptotically the same. ⌟

(b) Prove your statement. [4 points]
Hint: You may use a recursion tree to approach this question.

Solution. Here, we give a proof step-by-step, again.

> **Lemma.** $2n/3 > n/3$, $3n/4 > n/4$, $(2/3)^k n > (1/3)^k n$, and $(3/4)^k n > (1/4)^k n$ for any $k \in \mathbb{Z}^+$.

Proof. This is a direct consequence that $n > 0$, and $(-)^k$ function is strictly monotonic when the input is positive. □

Then, we want to consider the two separations as recursion trees.

> **Lemma.** The height of the recursion tree for Bob is $\lceil \log_{3/2}(n) \rceil$, the lowest (shallowest) depth of the leaf is $\lceil \log_3(n) \rceil$.

Proof. Here, we suppose the tree is k-levels tall, and from the previous lemma, $2n/3$ part will have the most remaining, so the subtrees of $n/3$ part will not have a higher depth compared to the $2n/3$ part. Hence when there is a total of n nodes, we have:

$$\left(\frac{2}{3}\right)^k \cdot n = 1,$$
$$n = \left(\frac{3}{2}\right)^k,$$
$$k = \log_{3/2}(n).$$

For the lowest depth possible leaf, recall that the $n/3$ part vanishes to 1 first, so say if it is m, we have:

$$\begin{aligned}\left(\frac{1}{3}\right)^m \cdot n &= 1,\\ n &= 3^m,\\ m &= \log_3(n).\end{aligned}$$

□

Lemma. The height of the recursion tree for Alice is $\lceil \log_{4/3}(n) \rceil$, the lowest (shallowest) depth of the leaf is $\lceil \log_4(n) \rceil$.

Proof. Here, we suppose the tree is k-levels tall, and from the first lemma, $3n/4$ part will have the most remaining, so the subtrees of $n/4$ part will not have a higher depth compared to the $3n/4$ part. Hence when there is a total of n nodes, we have:

$$\begin{aligned}\left(\frac{3}{4}\right)^k \cdot n &= 1,\\ n &= \left(\frac{4}{3}\right)^k,\\ k &= \log_{4/3}(n).\end{aligned}$$

For the lowest depth possible leaf, recall that the $n/4$ part vanishes to 1 first, so say if it is m, we have:

$$\begin{aligned}\left(\frac{1}{4}\right)^m \cdot n &= 1,\\ n &= 4^m,\\ m &= \log_4(n).\end{aligned}$$

□

Proposition. The runtime for both algorithm is $\mathcal{O}(n \log n)$.

Proof. First, we note that:

$$\log_{4/3}(n) = \frac{\log n}{\log 4 - \log 3} \text{ and } \log_{3/2}(n) = \frac{\log n}{\log 3 - \log 2}, \tag{1}$$

Then, note that for each level, the `QuickSort` at most traverse all numbers on the S_l and S_r with respect to each partition, so it will at most traverse all the elements once, so that operation is $\mathcal{O}(n)$.
Hence, for both algorithm, the total runtime is $\mathcal{O}(n \cdot \log n)$. □

Proposition. The runtime for both algorithm is $\Omega(n \log n)$.

Proof. Similar to equation (1), the we have:

$$\log_4(n) = \frac{\log n}{\log 4} \text{ and } \log_3(n) = \frac{\log n}{\log 3}.$$

so the depth of the recursion tree with full leafs on the level is asymptotically the same, namely, $\Theta(\log n)$. Then, note that for those level, the `QuickSort` must traverse all numbers on the S_l and S_r with respect to each partition, so it will at most traverse all the elements once, so that operation is n, so the lower bound of the algorithm is $\Omega(n \log n)$. □

Hence, the runtime of the function is $\Theta(n \log n)$ for both Bob and Alice.
As both algorithms have the same tight asymptotic bound, they are asymptotically the same. ⌟

Problem III.6. Suppose you are given a connected graph G, with edge costs that are all distinct. Prove that G has a unique minimum spanning tree.

Proof. For the sake of contradiction, suppose that G has distinct edge costs, and suppose G has two different minimum spanning trees. For simplicity, we denote n as the number of nodes.
Now, let $\mathcal{G}$ be the graph with all nodes, and edges being the union of the edges from the minimum spanning trees by our assumption. Since the minimum spanning trees are distinct, $\mathcal{G}$ has at least n edges, so there exists a cycle in $\mathcal{G}$. Since G has distinct edge costs, $\mathcal{G}$ also have distinct edge costs, and the cycle must have distinct edge weights.
According to the red rule, we can remove the largest weight in the cycle. We can repetitively repeat the red rule until there are $n-1$ edges so there are no cycles. However, this implies that at least one of the minimum spanning tree is not the minimum spanning tree, and this is a contradiction. □

Problem III.7. Let us say that a graph $G = (V, E)$ is a *near*-tree if it is connected and has at most $n + 8$ edges, where $n = |V|$. Give an algorithm with running time $O(n)$ that takes a near-tree G with costs on its edges, and returns a minimum spanning tree of G. You may assume that all the edge costs are distinct.

Solution. Here, we basically apply the red rule for 9 times.

---

while G contains a cycle **do**

Traverse the cycle and remove the edge of the largest weight in the cycle.

end while

Here, note that with $n + 8$ edges, we at most need to remove edges for 9 times. Also, note that finding and returning a cycle if $\mathcal{O}(2n + 8) = \mathcal{O}(n)$, traversing the cycle and removing one is $\mathcal{O}(n + 8) = \mathcal{O}(n)$, and hence, we have the total runtime as:

$$\mathcal{O}(9n) = \mathcal{O}(n),$$

as desired. ⌟

Problem III.8. Consider a sorted sequence of n integers $[x_1, x_2, \ldots, x_n]$. Additionally, assume that the integers are distinct, that is, $x_i \neq x_j$ whenever $i \neq j$. You may also assume that the integers are sorted in increasing order. You need to design an $O(\log n)$ algorithm to determine if there exists an index i such that the value at that index is equal to the index. The indices start from 1 (not from 0).

For example, in the sorted sequence $[-5, -1, 3, 14, 23]$, $x_3 = 3$, so the algorithm should output 3. In $[2, 4, 5, 7, 8, 19]$, there is no such i. Prove that the runtime complexity of your algorithm is $O(\log n)$.

Solution. Here, we basically use a similar idea with binary search, to search the middle element and decide if to search the left or the right part.

Let $l \leftarrow 1$ and $r \leftarrow n$.
while $l \neq r$ **do**
  $m \leftarrow \lfloor (l + r)/2 \rfloor$.
  if $x_m = m$ **then return** m.
  else if $x_m < m$ **then**
    Let $l \leftarrow m + 1$.
  else
    Let $r \leftarrow m - 1$.
  end if
end while
return There is no such output.

Here, we can think of this problem forming a decision tree, each iteration, the problem either ends, or it turns into a problem of (almost) half size.
The tree takes at most $\log(n) + 1$ level, and the arguments in the while loop is constant time, so the total runtime is $\mathcal{O}(\log n) \cdot \mathcal{O}(1) = \mathcal{O}(\log n)$, as desired. ⌟

Remark. Note that there is an alternative method to tackle on this problem. Here, we utilize the following property:

> **Proposition.** Say $\{S_i\}_{i=1}^n$ is a strict monotonically increasing sequence of integers, then $\{S_i - i\}_{i=1}^n$ is monotonically increasing.

This property can be verified quite easily, so it is left as an exercise to the readers.

Then, as long as we modify the sequence into $\{S_i - i\}_{i=1}^n$, we just need to apply a **binary search** with a sorted array for 0, but possibly with repetition, whose runtime is $\mathcal{O}(\log n)$.

Problem III.9. Alice and Bob are solving a problem. They are given a matrix A of size $n \times n$, where elements are sorted along every column and every row.
They need to provide an algorithm which takes integer x as an input, and checks if x is an element of matrix A or not. Both Alice and Bob decided to apply divide and conquer method.
Alice's algorithm is quite simple, she uses binary search routine to check each row.
Bob's algorithm is more advanced. Bob found out that if matrix A is represented in block-view:

$$A = \left[\begin{array}{c|c} A_1 & A_2 \\ \hline A_3 & A_4 \end{array}\right],$$

where all matrices A_1, A_2, A_3, A_4 are of the size $\frac{n}{2} \times \frac{n}{2}$, then he can compare element x with $A_4[1,1]$ and consider only three options:

1. $x = A_4[1,1]$ then his algorithm outputs "YES".
2. $x < A_4[1,1]$ then he knows that he only needs to recursively check if $x \in A_1$, if $x \in A_2$ and if $x \in A_3$.
3. $x > A_4[1,1]$ then he knows that he only needs to recursively check if $x \in A_2$, if $x \in A_3$ and if $x \in A_4$.

Whose algorithm is **asymptotically** faster on the worst case input? Prove your statement.

Solution. It is trivial to analyze Alice's algorithm first. In the worst case, there will be n columns and each having a binary search of $\Theta(\log n)$, so the total runtime is $\Theta(n \log n)$.
Then, we take a look on Bob's algorithm. This makes it a divide and conquer problem. By assuming it is a square matrix, without loss of generality, we assume that n is dyadic, since in the divide and conquer that divide the matrix into four sub-matrices, the size of the largest matrix will be equivalent as if n is the larger dyadic number. Hence, for n being the dimension of the matrix, *i.e.*, $A_s \in \mathbb{R}^{n \times n}$, we can form the following recursive (worst case) runtime:

$$T(n) = 3T\left(\frac{n}{2}\right) + \Theta(1),$$

where we have $T(0) = 0$ and $T(1) = \Theta(1)$, so we can apply **master's theorem** of the case $0 < \log_2 3$ to obtain that:

$$T(n) = \Theta\big(n^{\log_2(3)}\big).$$

Here, we use a limit test to obtain that:

$$\begin{aligned} \lim_{n\to\infty} \frac{n^{\log_2 3}}{n \log n} &= \lim_{n\to\infty} \frac{n^{\log_2(3/2)}}{\log n} \Big(= \frac{\infty}{\infty} \Big) \\ &\overset{\text{L'Hop}}{=\!=} \lim_{n\to\infty} \frac{\log_2(3/2) n^{\log_2(3/2)}/n}{1/n} = \lim_{n\to\infty} \log_2(3/2) n^{\log_2(3/2)} = \infty, \end{aligned}$$

since $\log_2(3/2) > 0$, hence, we have:

$$n^{\log_2 3} \in \Omega(n \log n) \text{ but } n^{\log_2 3} \notin \mathcal{O}(n \log n).$$

Hence, the algorithm of Alice will be asymptotically faster. ⌟

IV Problem Set 4

Problem IV.1. Maximum Profit During an Interval **[6 points]**

Tian decides to start doing stock trading (albeit only in his imagination!)
He recorded his weekly net profits for the initial 12 weeks:

$$[-17, 5, 3, -10, 6, 1, 4, -3, 8, 1, -13, 4]$$

Tian wants to identify the period of his most significant gains, so it gives him something to brag about. For example, in the above data, Tian lost money overall. However, when he sums up his profits from week 5 to week 10, he gets a profit of $17 = (6 + 1 + 4 + (-3) + 8 + 1)$ that he can show off to impress his friends.

Consider Tian's weekly net profits, denoted as p_k for each week k ($1 \leq k \leq n$).

(a) Design a Divide and Conquer algorithm to find the index pair i, j representing the weeks such that Tian gains the maximum net profit between week i and week j, both inclusive? In other words, find the index pair i, j that maximizes $\sum_{k=i}^{j} p_k$. [4 points]

Note: You are required to use Divide and Conquer approach to solve this problem. The intention is that you become proficient in using the technique. In the chapter on Dynamic Programming, we will study a faster method of solving this problem.

Solution. Here, we can make a very easy edit to turn this problem into the problem on **March 11** problem. Without loss of generality, assume the stock price start at 0 (this does not matter and can be anything), and we can have a list of the stock price on each week $\{S_i\}_{i=0}^{n}$ so that:

$$S_i = \begin{cases} 0, & \text{when } i = 0; \\ \sum_{\ell=1}^{i} p_i, & \text{when } 1 \leq i \leq n. \end{cases}$$

Then, with the list S_i, we can easily apply the **March 11** problem algorithm, and the problem will be solved.
To make this idea easier to understand, if we are have consecutive profit sums, this is the equivalent of having a stock price, buy on some day and sell on another day, since the sum of the daily profits is basically the total profit since the day you buy in and until the day you sell out. The reason this adaption is done is because through this approach, the find minimum price on the left and maximum price on the right would significantly reduce the number of computations. Although it is asymptotically the same, this would be *relatively* less computations.
However, we will still give a formal algorithm.

Let $\{S_i\}_{i=0}^{n}$ be a (ordered) sequence to hold the stock price deviation from week 0, and let $S_0 \leftarrow 0$.
for $i \leftarrow 1, 2, \cdots, n$ **do**
  Let $S_i \leftarrow S_{i-1} + p_i$.
end for
function FIND OPTIMAL(start week, end week)
  if start week $+1$ = end week **then**

return (start week, end week, $S_{\text{end week}} - S_{\text{end week}}$).
end if
if start week = end week **then**
return (start week, end week, $-\infty$).
end if
Let middle week $\leftarrow$ (end week + start week)/2.
Let $(l_1, l_2, l_v) \leftarrow$FIND OPTIMAL(start week, middle week).
Let $(r_1, r_2, r_v) \leftarrow$FIND OPTIMAL(middle week+1, end week).
Let $l_m \leftarrow \arg\min_{\text{start week} \le j \le \text{middle week}}$ and $r_m \leftarrow \arg\max_{\text{middle week}+1 \le j \le \text{end week}}$
Let $f \leftarrow S_{r_m} - S_{l_m}$.
if $l_v = \max\{l_v, r_v, f\}$ **then return** (l_1, l_2, l_v)
else if $r_v \ge f$ **then return** (r_1, r_2, r_v)
else return (l_m, r_m, f).
end if
end function
Let $(i, j, v) \leftarrow$ FIND OPTIMAL$(0, n)$, **return** $(i + 1, j)$. ⌟

(b) Analyze the computational complexity of your algorithm. [1 point]

Solution. For the algorithm:

- The preprocessing (which is finding the first i partial sum) is $\Theta(n)$, as we do the addition for n times.
- For the FIND OPTIMAL function on n elements (end week $-$ start week $= n$), say the runtime is T, the left and right subintervals would take $T\left(\frac{n}{2}\right)$, and the finding minimum of the left partition and the maximum of the right partition is $\mathcal{O}(n)$ whereas other comparisons in the function is $\mathcal{O}(1)$, so:

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

Note that for the base case, $T(1) = \mathcal{O}(1)$. Also, without loss of generality, we may assume n is dyadic (power of 2), since we can just round it to the next dyadic number so that double the size of the input does not impact $\mathcal{O}(\cdot)$ notation.
This falls into the second case for **Master's theorem**, and so $T(n) = \Theta(n \log n)$, and for n weeks, the complexity of the FIND OPTIMAL function is $\Theta(n \log n)$.

- The later function call is just $\Theta(1)$.

Hence, the total runtime is:

$$\Theta(n) + \Theta(n \log n) + \Theta(1) = \boxed{\Theta(n \log n)}.$$ ⌟

(c) Demonstrate the execution of your Divide and Conquer algorithm: determine the maximum profit Tian achieved over any given interval for the provided weekly profit data:

$$[-8, 4, 3, -12, 7, 12, -4, 5, -15, 11, 2, -9, 6, -1, 5, -6]$$

For Part (c), you may show your work by hand and then paste a photo/screenshot in the PDF file. [1 point]

Solution. First, we compute the partial sum list, namely as:

$$\{S_i\}_{i=0}^{16} = \{0, -8, -4, -1, -13, -6, 6, 2, 7, -8, 3, 5, -4, 2, 1, 6, 0\}.$$

Then, we simulate the function call and compute all the FIND OPTIMAL functions.

- FIND OPTIMAL(0,16):
  - Minimum from 0 to 8 is week 4 for -13, maximum from 9 to 16 is week 15 for 6, so here, the maximum profit is 19 (week 5 to 15).
  - FIND OPTIMAL(0,8):
    - Minimum from 0 to 4 is week 4 for -13, maximum from 5 to 8 is week 8 for 7, so here, the maximum profit is 20 (week 5 to 8).
    - FIND OPTIMAL(0,4):
      - Minimum from 0 to 2 is week 1 for -8, maximum from 5 to 8 is week 3 for -1, so here, the maximum profit is 7 (week 2 to 3).
      - FIND OPTIMAL(0,2):
        - Minimum from 0 to 1 is week 1 for -8, maximum from 2 to 2 is week 2 for -4, so here, the maximum profit is 4 (only 2).
        - FIND OPTIMAL(0,1): $S_0 - S_1 = -8$, the maximum profit is -8 (only week 1)
        - FIND OPTIMAL(2,2): $-\infty$ (invalid input).
        - Return 4 for week 2 only.
      - FIND OPTIMAL(3,4): $S_4 - S_3 = -8$, the maximum profit is -8 (only week 4).
      - Return 7 for week 2 to 3.
    - FIND OPTIMAL(5,8):
      - Minimum from 5 to 6 is week 5 for -6, maximum from 7 to 8 is week 8 for 7, so here, the maximum profit is 13 (week 6 to 8).
      - FIND OPTIMAL(5,6): $S_6 - S_5 = 12$, the maximum profit is 12 (only week 6).
      - FIND OPTIMAL(7,8): $S_8 - S_7 = 5$, the maximum profit is 5 (only week 8).
      - Return 13 for week 6 to 8.
    - Return 20 for week 5 to 8.
  - FIND OPTIMAL(9,16):
    - Minimum from 9 to 12 is week 9 for -8, maximum from 13 to 16 is week 15 for 6, so here, the maximum profit is 14 (week 10 to 15).
    - FIND OPTIMAL(9,12):
      - Minimum from 9 to 10 is week 9 for -8, maximum from 11 to 12 is week 11 for 5, so here, the maximum profit is 13 (week 10 to 11).
      - FIND OPTIMAL(9,10): $S_{10} - S_9 = 11$, the maximum profit is 11 (only week 10)

- FIND OPTIMAL(11,12): $S_{12} - S_{11} = -9$, the maximum profit is -9 (only week 12)
- Return 13 for week 10 to 11.

- FIND OPTIMAL(13,16):
  - Minimum from 13 to 14 is week 14 for 1, maximum from 15 to 16 is week 15 for 6, so here, the maximum profit is 5 (week 15 to 15).
  - FIND OPTIMAL(13,14): $S_{14} - S_{13} = -1$, the maximum profit is -1 (only week 14)
  - FIND OPTIMAL(15,16): $S_{16} - S_{15} = -6$, the maximum profit is -6 (only week 16)
  - Return 5 for week 15 only.
- Return 14 for week 10 to 15.

- Return 20 for week 5 to 8.

Hence, the maximum profit is $\boxed{\text{from week 5 to 8 for 20}}$. ⌟

Problem IV.2. Smart Stock Surge **[6 points]**

Imagine you're looking at the price of a given stock over n consecutive days, numbered $i = 1, 2, \ldots, n$. For each day i, we have a price $p(i)$ per share for the stock on that day. We'll assume for simplicity that the price was fixed during each day. How should we choose a day i on which to buy the stock and a later day $j > i$ on which to sell it, if we want to maximize the profit per share, $p(j) - p(i)$? If there is no way to make money during the n days, we should conclude this instead.

(a) Give an algorithm to find the optimal pair of days i and j in time $O(n)$ using Dynamic Programming by completing the following: [5 points]

1. Define optimizing function(s) OPT.
2. Define the Bellman equation(s), provide a justification.
3. Describe your algorithm to solve this problem based on your Bellman equation: your algorithm should return the optimal pair (i, j) that maximizes the profit.
4. Analyze the runtime of your algorithm.

Note: We discussed this problem in the lecture on March 11 (before the Spring break). We saw that there is an algorithm that solves this problem in $\mathcal{O}(n^2)$ time. We then solved this problem using Divide and Conquer in $\mathcal{O}(n \log n)$ time. We would now like to solve this problem using Dynamic Programming in $\mathcal{O}(n)$ time.

Solution. Here, we define/justify all components as required:

1. First, for optimization function OPT : $\{1, 2, \cdots, n\} \to \mathbb{R}$: We define the optimization function *evaluated* at i as $\boxed{\text{the maximum revenue assuming the stock being sold on day } i}$.
2. Then, Bellman equation:

$$\text{OPT}(i) = \boxed{\begin{cases} 0, & \text{when } i = 1; \\ \max\{0, p(i) - p(i-1) + \text{OPT}(i-1)\}, & \text{when } 2 \le i \le n. \end{cases}}$$

   Note that for having a stock sold on day i, it is either bought on the same day and sold on the same day (making no revenue), or bought on some earlier days and hold until the previous day for maximum possible revenue, and on day 1, if selling on day 1, it is not possible to make any revenue.
3. Then, we think about the algorithm.

Let $\{R_i\}_{i=1}^n$ be the (ordered) sequence to record $\text{OPT}(i)$, let $\text{OPT}(1) = 0$.
for $i \leftarrow 2, 3, \cdots, n$ **do**
  Compute $R_i \leftarrow \max\{0, p(i) - p(i-1) + R_{i-1}\}$
end for
Let $D_{\max} \leftarrow \arg\max_{1 \le i \le n} R_i$ (in repetitive cases, select the smallest).
Let $SD \leftarrow \max\{d \le D_{\max} : R_d = 0\}$.
return (SD, D).

4. For the runtime, we note that the computation of R_i is constant time, the looping it for n times gives runtime of $\mathcal{O}(n)$.
   For finding the maximum data of sold and maximum revenue is $\mathcal{O}(n)$.
   For backtracking and finding the start day (buy in date), the function will verify n desired indices for worst case, so it is also $\mathcal{O}(n)$.
   Hence, the runtime of the algorithm is $\boxed{\mathcal{O}(n)}$, as desired. ⌟

(b) Demonstrate your algorithm on the following example: [1 point]

$$p = [9, 3, 6, 7, 2, 5, 8, 1, 4]$$

Solution. Here, we first compute the OPT function in terms of the sequence:

$$\{R_i\}_{i=1}^{n} = \{0, 0, 3, 4, 0, 3, 6, 0, 3\}.$$

We note that the maximum is $R_7 = 6$, so the stock is sold on day 7.
Then, we backtrack to find the buy-in date, note $R_5 = 0$ is the maximum index less than 7 that has zero possible revenue.
So we buy in on day 5 for price of 2 and sell out on day 7 for price of 8 to get revenue of 6. ⌟

Problem IV.3. Spy-namic Programming **[6 points]**

As a loyal spy of your home country, you are given a task where you have to infiltrate enemy's capital city. The capital contains streets defined by a $m \times n$ grid where m refers to the street number and n refers to the avenue number. Your helicopter drops you on the upper-left corner of the city on the intersection of 1^{st} Street and 1^{st} Avenue. Your task is to navigate from your location to the Enemy Headquarters situated at the lower-right corner of the grid on the intersection of m-th Street and n-th Avenue.

To efficiently infiltrate, you can only make rightward and downward moves. You are not allowed to move up or left; you can only move down or right. In other words, you are not allowed to retreat. To assist in your stealthy intrusion, Supervisor GG has provided you a $m \times n$ matrix called `outpost`, where outpost$[i, j]$ = **yes** if the intersection of i^{th} Street and j^{th} Avenue is a location filled with enemy soldiers and needs to be avoided.

Consider the following example in which outpost$[i, j]$ = **yes** for $(1, 4), (2, 6), (3, 2), (4, 3)$ and $(4, 5)$.

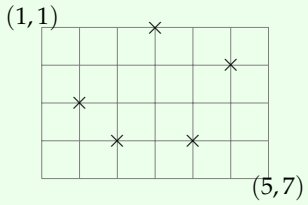


Figure IV.1. Example for Spy-namic programming.

In the figure, $\times$ represents an outpost.

For instance, there are three paths from $(1, 1)$ to $(3, 3)$, given by

- $(1,1) - (1,2) - (1,3) - (2,3) - (3,3)$,
- $(1,1) - (1,2) - (2,2) - (2,3) - (3,3)$, and
- $(1,1) - (2,1) - (2,2) - (2,3) - (3,3)$.

(a) Give an $\mathcal{O}(mn)$ algorithm that takes the matrix `outpost` as input and returns the number of safe paths from $(1, 1)$ to (m, n); it returns 0 if no such path exists.
Complete the following: [5 points]

1. Define optimizing function(s) OPT.
2. Define the Bellman equation(s), provide a justification.
3. Describe your algorithm to solve this problem based on your Bellman equation, your algorithm should output the number of paths from $(1, 1)$ to (m, n).
4. Analyze the runtime of your algorithm.

Solution. Here, we define/justify all components as required:

1. First, for optimization function OPT : $\{1, 2, \cdots, m\} \times \{1, 2, \cdots, n\} \to \mathbb{R}$: We define the optimization function *evaluated* at i, j as $\boxed{\text{the number of } \textit{non-retreat} \text{ path from } (1,1) \text{ to } (i,j)}$.

2. Then, Bellman equation:

$$\mathrm{OPT}(i,j) = \boxed{\begin{cases} 0, & \text{when outpost}[i,j] = \textbf{yes}; \\ 1, & \text{when } i = j = 1 \text{ and outpost}[i,j] = \textbf{no}; \\ \mathrm{OPT}(i,j-1), & i = 1, j > 1 \text{ and outpost}[i,j] = \textbf{no}; \\ \mathrm{OPT}(i-1,j), & i > 1, j = 1 \text{ and outpost}[i,j] = \textbf{no}; \\ \mathrm{OPT}(i-1,j) + \mathrm{OPT}(i,j-1), & i,j > 1 \text{ and outpost}[i,j] = \textbf{no}. \end{cases}}$$

Note that for any points that has an outpost, there is no way to get there. Otherwise, if $i = 1$ and $j = 1$, there is one way to get there, and all other cases, will be the number of paths from its left or above if that point is accessible.

3. Then, we think about the algorithm.

Let $\{R_{i,j}\}_{i=1,j=1}^{i=m,j=n}$ be the (ordered) (2-dimensional) sequence to record $\mathrm{OPT}(i,j)$, initialize everything as `NULL`.

function COMPUTE OPT(i,j)
  if $R_{i,j} \neq$ `NULL` **then return** $R_{i,j}$
  else if outpost[i,j] = **yes then return** 0
  else if $i = 1$ and $j = 1$ **then return** 1
  else if $i = 1$ **then return** COMPUTE OPT($i, j-1$)
  else if $j = 1$ **then return** COMPUTE OPT($i-1, j$)
  else return COMPUTE OPT($i, j-1$) + COMPUTE OPT($i-1, j$)
  end if
end function
return COMPUTE OPT(m,n).
for $i \leftarrow 1, 2, \cdots, m$ **do**
  for $j \leftarrow 1, 2, \cdots, n$ **do**
    Let $R_{i,j} \leftarrow$ COMPUTE OPT(i,j).
  end for
end for
return $R_{m,n}$.

Note that we just start from $(1,1)$ and computer all the way through the *dictionary order* (this is exactly $(1,1)$, $(1,2)$, $\cdots$, $(1,n)$, $(2,1)$, $\cdots$, $(2,n)$, $(3,1)$, $\cdots$, (m,n)), so we do not need a big call stack.

4. For the runtime, we note that the initialization of the $R_{i,j}$ (2-dimensional) sequence is $\Theta(mn)$, or $\Theta(1)$ if the programming language initialize `NULL` by default.
Note that the each call of the COMPUTE OPT function, if not accounting for the other call stacks recursively, will be constant time. Note that since our dynamic programming remembers each value at each computation, so the worst case is to compute all the number of ways on the m-by-n matrix, so it will be for worst $\mathcal{O}(mn)$ Hence, the runtime of the algorithm is $\boxed{\mathcal{O}(mn)}$, as desired. ⌟

(b) Demonstrate the execution of your algorithm on *Figure IV.1* to compute the number of safe paths from $(1,1)$ to $(5,7)$. [1 point]
PS: It should be noted that the instructor had proposed a non-violent setting of driving an ambulance from JHU to JHH while avoiding certain intersections that are full of potholes. However, some members of the teaching team are fans of spy movies.

Solution. We will start from $(1,1)$ using a dictionary order all the way to $(5,7)$.
For simplicity of computation, we will use a tabular with row being row and column as column, where the green ones are zero due to an outpost.

1	1	1	0	0	0	0
1	2	3	3	3	0	0
1	0	3	6	9	9	9
1	1	0	6	0	9	18
1	2	2	8	8	17	35

Table IV.2. Compuation of the 2-dimensional sequence (or matrix) of $\{R_{i,j}\}_{i,j=1}^{m,n}$.

Hence, the total number of ways getting to $(5,7)$ is $\boxed{35}$ ways. ⌟

Problem IV.4. Longest Path in a Graph **[6 points]**

Let $G = (V, E)$ be a directed graph with nodes $v_1, \ldots, v_n$. We say that G is an ordered graph if it has the following properties.

(i) Each edge goes from a node with a lower index to a node with a higher index. That is, every directed edge has the form (v_i, v_j) with $i < j$.

(ii) Each node except v_n has at least one edge leaving it. That is, for every node v_i, where $i = 1, 2, \ldots, n-1$, there is at least one edge of the form (v_i, v_j).

Goal: *Given an ordered graph G, find the length of the longest path that begins at v_1 and ends at v_n.* The length of a path is the number of edges in it.

For instance,

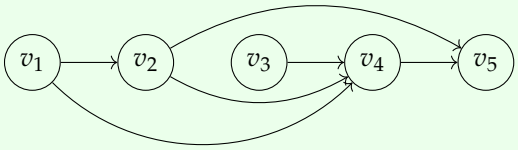


Figure IV.3. The longest path from v_1 to v_n uses 3 edges (v_1, v_2), (v_2, v_4), (v_4, v_5).

(a) Show that the following algorithm does not correctly solve this problem, by giving an example of an ordered graph on which it does not return the correct answer [1 point]

"Longest Path" Greedy Algorithm

```
Set w = v_1
Set L = 0
while there is an edge out of the node w do
    Chose the edge (w, v_j) for which j is as small as possible
    Set w = v_j
    Increment L by 1
end while
return L as the length of the longest path
```

Solution. Here, we form a counter example:

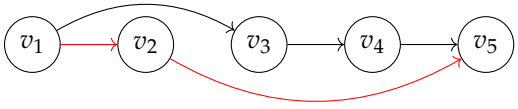


Figure IV.4. Counter Example to Greedy algorithm, red arrow refer to the path found by greedy algorithm.

Note that the greedy algorithm would find $v_1 \to v_2 \to v_5$ of length 2, but $v_1 \to v_3 \to v_4 \to v_5$ is longer of length 3 (and this happens to be the longest, since there are only two paths from v_1 to v_5). ⌟

(b) Give an efficient algorithm that takes an ordered graph G and returns the length of the longest path that begins at v_1 and ends at v_n, by completing the following: [4 points]

1. Define optimizing functions OPT.
2. Define the Bellman equation(s), provide a justification.
3. Describe your algorithm to solve this problem based on your Bellman equation, your algorithm should return the length of the longest path that begins at v_1 and ends at v_n.
4. Analyze the runtime of your algorithm.

Solution. Here, we define/justify all components as required:

1. First, for optimization function OPT : $\{1, 2, \cdots, n\} \to \mathbb{R}$: We define the optimization function *evaluated* at i as $\boxed{\text{the longest path from } v_1 \text{ to } v_i}$, it will be $-\infty$ if there is not a path.

2. Then, Bellman equation:

$$\text{OPT}(i) = \boxed{\begin{cases} 0, & \text{when } i = 1; \\ -\infty, & \text{when } i \neq 1 \text{ and } v_i \text{ has no parent node;} \\ 1 + \max_{j:v_j \text{ is a parent of } v_i} \text{OPT}(j) & \text{when } v_i \text{ has at least one parent node.} \end{cases}}$$

As a side note, v_j being a parent of v_i means that $j < i$ and there is path of length 1 from v_j to v_i, which is an extension of parent concept in graph theory.
Here, the longest path is one longer than the longest path of all its parents (and $-\infty + 1 = -\infty$).

3. Then, we write the algorithm here:

for $m \leftarrow 2, 3, \cdots, n$ **do** ▷ (*This initialization makes sure accessing list of parent is constant time.*)
  for all $t \leftarrow 1, \cdots, m$ **do**
    Verify if t is a parent of m, record if necessary.
  end for
end for
Let $\{L_i\}_{i=1}^n$ be the (ordered) sequence to record $\text{OPT}(i)$, let $\text{OPT}(1) = 0$.
for $i \leftarrow 2, 3, \cdots, n$ **do**
  if v_i has no parent node **then** Let $L_i = -\infty$.
  else Let $L_i \leftarrow 1 + \max_{j:v_j \text{ is a parent of } v_i} L_j$.
  end if
end for
return L_n.

4. For the runtime. For the initialization, the runtime is $\mathcal{O}(n^2)$, as for each node we check for at most m times for the m from 2 to n, so it is an arithmetic series, whose sum is at most n^2. Also, we note that the computation of each L_i is $\mathcal{O}\big(\deg^+(v_i)\big)$ (where $\deg^+$ denotes the number of incoming edges),

and since all prior L_i's are computed, we have the total runtime as:

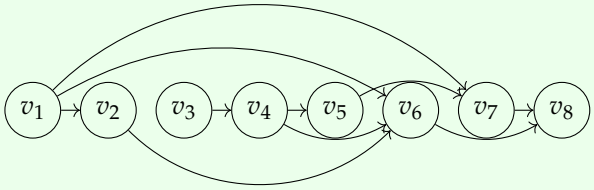
$$\sum_{i=1}^{n} \mathcal{O}\big(\deg^+(v_i)\big) = \mathcal{O}(|E|).$$

Eventually, we note that the node has at most n^2 edges, so we have the runtime as:

$$\mathcal{O}(|E|) + \mathcal{O}(n^2) = \boxed{\mathcal{O}(n^2)}.$$

⌟

(c) Demonstrate the execution of your algorithm to compute the length of the longest path on the following graph [1 point]



Solution. Here, we just compute the maximum length from v_1 to each node v_i:

- To v_1: 0.
- To v_2: $1 + 0 = 1$ (parent: v_1).
- To v_3: $-\infty$ (no parent).
- To v_4: $1 - \infty = -\infty$ (parent: v_3).
- To v_5: $1 - \infty = -\infty$ (parent: v_4).
- To v_6: $1 + \max\{0, 1, -\infty\} = 2$ (parent: v_1, v_2, v_4).
- To v_7: $1 + \max\{0, -\infty\} = 1$ (parent: v_1, v_4).
- To v_8: $1 + \max\{2, 1\} = 3$ (parent: v_6, v_7).

Hence, the longest path from $v1$ to v_8 is of length $\boxed{3}$. ⌟

Problem IV.5. This is Zion and we are not afraid! **[6 points]**

The residents of the underground city of Zion[1] defend themselves through a combination of kung fu, heavy artillery, and efficient algorithms. Recently, they have become interested in automated methods that can help fend off attacks by swarms of robots. Here's what one of these robot attacks looks like.

- A swarm of robots arrives over the course of n seconds; in the i-th second, x_i robots arrive. Based on remote sensing data, you know this sequence $x_1, x_2, ..., x_n$ in advance.
- You have at your disposal an *electromagnetic pulse* (EMP), which can destroy some of the robots as they arrive; the EMP's power depends on how long it's been allowed to charge up. To make this precise, there is a function $f(\cdot)$ representing the power of EMP activation: if j seconds have passed since the EMP was last used, then it is capable of destroying up to $f(j)$ robots.
- So specifically, if it is used in the k-th second, and it has been j seconds since it was previously used, then it will destroy $\min(x_k, f(j))$ robots. After this use, it will be completely drained.
- We will also assume that the EMP starts off completely drained, so if it is used for the first time in the j-th second, then it is capable of destroying up to $f(j)$ robots.

Problem Given the data on robot arrivals $x_1, x_2, ..., x_n$, and given the recharging function $f(\cdot)$, choose the points in time at which you're going to activate the EMP so as to destroy as many robots as possible.

Example: Suppose $n = 4$ and the values of x_i and $f(i)$ are given by the table below.

i	1	2	3	4
x_i	1	10	10	1

i	1	2	3	4
$f(i)$	1	2	4	8

The best solution would be to activate the EMP in the 3^{rd} and the 4^{th} seconds. In the 3^{rd} second, the EMP has gotten to charge for 3 seconds, and so it destroys $\min(10, 4) = 4$ robots. In the 4^{th} second, the EMP has only gotten to charge for 1 second since its last use, and it destroys $\min(1, 1) = 1$ robot. This is a total of 5.

(a) Show that the greedy algorithm below does not correctly solve this problem, by giving an instance on which it does not return the correct answer. In your example, say what the correct answer is and also the output of the algorithm. [1 point]

EMP-scheduling algorithm

```
1: function SCHEDULE-EMP(x_1, ..., x_n)
2:     Let j be the smallest number such that f(j) ≥ x_n    ▷ (If no such j exists, set j = n)
3:     Activate EMP in n^th second
4:     if n − j ≥ 1 then
5:         SCHEDULE-EMP(x_1, ..., x_{n−j})
6:     end if
7: end function
```

[1] This is a reference to the classic Hollywood movie **The Matrix**.

Solution. Here, we give an example of $n = 2$, where the values of x_i and $f(i)$ are given as follows:

i	1	2
x_i	1	1

i	1	2
$f(i)$	1	2

Table IV.5. Values of x_i and $f(i)$.

By the greedy algorithm, the EMP will be activated only at the 2nd second, destroying a total of 1 robots. However, the correct answer is the only other approach, to activate at the 1st and 2nd seconds which would destroy 2 robots. ⌟

(b) Give an efficient algorithm that takes the data on robot arrivals $x_1, x_2, \cdots, x_n$ and the recharging function $f(\cdot)$ and returns the maximum number of robots that can be destroyed by a sequence of EMP activations. Complete the following: [4 points]

1. Define optimizing functions OPT.
2. Define the Bellman equation(s), provide a justification.
3. Describe your algorithm to solve this problem based on your Bellman equation, your algorithm should return the maximum number of robots that can be destroyed.
4. Analyze the runtime of your algorithm.

Solution. Here, we define/justify all components as required:

1. First, for optimization function $\text{OPT} : \{0, 1, 2, \cdots, n\} \to \mathbb{R}$: We define the optimization function *evaluated* at i, j as $\boxed{\text{the maximum possible robots destroyed when an attack was executed at } n^{\text{th}} \text{ second}}$.

2. Then, Bellman equation:

$$\text{OPT}(i) = \boxed{\begin{cases} 0, & \text{when } i = 0; \\ \max_{1 \le j \le i} \{\min\{x_i, f(j)\} + \text{OPT}(i-j)\} & \text{when } 1 \le i \le n \end{cases}}$$

   Here, the higher is the maximum of all number of destroyed after all possible resting time and the optimal of the most number destroyed that many time ago.

3. Then, we write the algorithm here:

   Let $\{M_i\}_{i=0}^{n}$ be the (ordered) sequence to record $\text{OPT}(i)$, let $\text{OPT}(0) = 0$.
   for $i \leftarrow 1, 2, 3, \cdots, n$ **do**
      Let $\{X_k\}_{k=1}^{i}$ be a temporary sequence to hold maximum destroy value at the current stage.
      for $j \leftarrow 1, 2, \cdots, i$ **do**
         Compute $X_k \leftarrow \min\{x_i, f(k)\} + \text{OPT}(i-j)$.
      end for
      Let $M_i = \max_k X_k$.
   end for
   return M_n.

4. For the runtime, we note that the outer for loop will run n times and the inner for loop will be running $\Theta(i)$ time, so we have:

$$\sum_{i=1}^{n} \mathcal{O}(i) = \frac{\mathcal{O}(1) + \mathcal{O}(n)}{2} \cdot \mathcal{O}(n) = \mathcal{O}(n^2).$$

The other initialization and return will be of constant time, so the total runtime is $\boxed{\mathcal{O}(n^2)}$. ⌟

(c) Demonstrate your algorithm by computing the maximum number of robots that can be destroyed given the following example: [1 point]

i	1	2	3	4	5	6
x_i	2	4	10	4	5	11

i	1	2	3	4	5	6
$f(i)$	1	3	6	10	13	16

Solution. Here, we give a sample run here.

- $\text{OPT}(1) = 1$.
- $\text{OPT}(2) = \max\{3, 1+1\} = 3$.
- $\text{OPT}(3) = \max\{6, 3+1, 1+3\} = 6$.
- $\text{OPT}(4) = \max\{4, 4+1, 3+3, 1+6\} = 7$.
- $\text{OPT}(5) = \max\{5, 5+1, 5+3, 3+6, 1+7\} = 9$.
- $\text{OPT}(6) = \max\{11, 11+1, 10+3, 6+6, 3+7, 1+9\} = 13$.

Hence, the maximum number of robots that can be destroyed is $\boxed{13}$. ⌟

Problem IV.6. Consider a set with n elements, where $n = 2^k$. Design a divide-and-conquer algorithm to find the minimum and maximum element in the set. The algorithm should use at most $3n/2$ comparisons. Show your work, that is, give the algorithm, show the recurrence relation and show that the solution to the recurrence relation takes at most $3n/2$ steps.

Note: You should **not** use Master's theorem for this question because we've **not** asked for $\mathcal{O}(\cdot)$ number of steps.

Solution. Here, we define a divide-and-conquer algorithm, as follows:

For simplicity, we have the sequence of n elements $\{a_i\}_{i=1}^n$.
function FIND-EXTREME(l, r)
  if $l = r$ **then return** a_l, a_l.
  else if $l = r - 1$ **then**
    if $a_l < a_r$ **then return** a_l, a_r
    else return a_r, a_l
    end if
  end if
  Let $m \leftarrow \lfloor (l + r)/2 \rfloor$.
  $l_l, m_l \leftarrow$ FIND-EXTREME(l, m).
  $l_r, m_r \leftarrow$ FIND-EXTREME($m + 1, r$).
  return $\min(l_l, l_r), \max(m_l, m_r)$.
end function
Here, we call FIND-EXTREME($1, n$).

Basically, we can consider this as dividing the problem into two subproblems, finding the maximum and minimum on the left half and right half.
In terms of the number of comparisons, we can analyze its recurrence relation in terms of $n := r - l + 1$:

$$T(n) = 2 + T\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + T\left(\left\lceil \frac{n}{2} \right\rceil\right),$$

where we have $T(0) = 0$, $T(1) = 0$, and $T(2) = 1$.
Note that since we want to show the **exact bound**, we would produce a prove by strong induction that $T(n) \leq 3(n-1)/2$ on dyadic entries of n.
In this case, we can simplify the relation to:

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

Hence, we can have a proof with induction for all positive dyadic n, which is induction on $n = 2^k$, where $k \in \mathbb{N}^+$ that:

$$T(n) = \frac{3n}{2} - 2.$$

Proof. (Base case:) $T(2) = 1 = 3 - 1$, as desired.

(Inductive step:) Assume $T(2^k) = 3 \cdot 2^k/2 - 2$ for $k \in \mathbb{N}^+$, we have:

$$T(2^{k+1}) = 2 + 2T(2^k) = 2 + 2\left(\frac{3(2)^k}{2} - 2\right) = \frac{3 \cdot 2^{k+1}}{2} - 2,$$

which completes the induction proof. □

Therefore, we have verified the upper bound of the number of comparisons for all dyadic n that is no less than 1. ⌟

Problem IV.7. Suppose youre consulting for a company that manufactures PC equipment and ships it to distributors all over the country. For each of the next n weeks, they have a projected supply s_i of equipment (measured in pounds), which has to be shipped by an air freight carrier.

Each weeks supply can be carried by one of two air freight companies, A or B.

- Company A charges a fixed rate r per pound (so it costs $r \cdot s_i$ to ship a weeks supply s_i).
- Company B makes contracts for a fixed amount c per week, independent of the weight. However, contracts with company B must be made in blocks of four consecutive weeks at a time.

A *schedule* for the PC company, is a choice of air freight company (A or B) for each of the n weeks, with the restriction that company B, whenever it is chosen, must be chosen for blocks of four contiguous weeks at a time. The cost of the schedule is the total amount paid to company A and B, according to the description above.

Example: Suppose $r = 1$, $c = 10$, and the sequence of values is

$$11, 9, 9, 12, 12, 12, 12, 9, 9, 11.$$

Then the optimal schedule would be to choose company A for the first three weeks, then company B for a block of four consecutive weeks, and then company A for the final three weeks.

Give a polynomial-time algorithm that returns a schedule of minimum cost. The inputs to the algorithm are: a sequence of weekly supply values $s_1, s_2, \cdots, s_n$; fixed rate r per pound charged by company A; and fixed amount c per week charged by company B.
Complete the following steps:

1. Define optimizing functions OPT.
2. Define the Bellman equation(s), provide a justification.
3. Describe your algorithm to solve this problem based on your Bellman equation.
4. Analyze the runtime of your algorithm.

Solution. Here, we provide each requested parts, as necessary.

1. Here, we define our optimizing function as:

   $$\text{OPT} : \{1, 2, \cdots, n\} \to \mathbb{R},$$

   where OPT evaluated at i is the optimal pricing with week i assuming week i is the last week of the four week contract with company B.

2. Then, we define the Bellman equation as:

$$\mathrm{OPT}(i) = \begin{cases} r \cdot s_1, & \text{for } i = 1, \\ r \cdot s_i + \mathrm{OPT}(i-1), & \text{for } i = 2, 3, \\ \min\{r \cdot s_i + \mathrm{OPT}(i-1), 4c\}, & \text{for } i = 4, \\ \min\{r \cdot s_i + \mathrm{OPT}(i-1), 4c + \mathrm{OPT}(i-4)\}, & \text{for } i \geq 5. \end{cases}$$

Here, we can think of the Bellman equation to consider the optimal of a week of either would have signed a contract with B 4 weeks ago or would have signed a contract with A in the previous week.

3. Here, we can briefly give the algorithm:

Let $\{v_i\}_{i=1}^n$ be the sequence to hold the optimal values.
Let $v_1 \leftarrow r \cdot s_1$.
for $i \leftarrow 2, 3$ **do**
    Let $v_i \leftarrow r \cdot s_i + v_{i-1}$.
end for
Let $v_4 \leftarrow \min\{r \cdot s_4 + v_3, 4c\}$.
for $i \leftarrow 5, \cdots, n$ **do**
    Let $v_i \leftarrow \min\{r \cdot s_i + v_{i-1}, 4c + v_{i-4}\}$.
end for
The optimal cost is $\min\{v_{n-4} + s_n + s_{n-1} + s_{n-2} + s_{n-3}, v_n\}$.
Let $\{c_i\}_{i=1}^n$ be the choices of the company to sign. For the sake of simplicity, let them all initialize to A, and let $p \leftarrow n$ as the progress.
while $p \geq 4$ **do**
    if $v_{p-4} + s_p + s_{p-1} + s_{p-2} + s_{p-3} > v_p$ **then**
        Let $c_p, c_{p-1}, c_{p-2}, c_{p-3} \leftarrow B, B, B, B$, and let $p \leftarrow p - 4$.
    else
        Let $p \leftarrow p - 1$.
    end if
end while

4. Eventually, we analyze the runtime as follows:

- For computing all the OPT values, the computations are all in constant time assuming the previous data are computed, so there need to be a total of $\mathcal{O}(n)$ for computing the OPT values.
- For retrieving the optimal choice, the retrieval loop runs for at most $\mathcal{O}(n-4)$ times, and each retrieval is of constant time comparison, so the total runtime is also $\mathcal{O}(n)$.

Hence, the total runtime is $\mathcal{O}(n)$. ⌟

Problem IV.8. Youre helping to run a high-performance computing system capable of processing several terabytes of data per day. For each of n days, you're presented with a quantity of data; on day i, you're presented with x_i terabytes. For each terabyte you process, you achieve a fixed revenue, but any unprocessed data becomes unavailable at the end of the day (*i.e.*, you can't work on it in the future).

You can't always process everything each day because you're constrained by the capabilities of your computing system, which can only process a fixed number of terabytes in a given day. In fact, it's running some one-of-a-kind software that, while very sophisticated, is not totally reliable, and so the amount of data you can process goes down with each day that passes since the most recent reboot of the system. On the first day after a reboot, you can process s_1 terabytes, on the second day after a reboot, you can process s_2 terabytes, and so on, up to s_n; we assume $s_1 > s_2 > \cdots > s_n > 0$. (Of course, on day i you can only process up to x_i terabytes, regardless of how fast your system is.) To get the system back to peak performance, you can choose to reboot it; but on any day you choose to reboot the system, you can't process any data at all.

Problem: Given the amounts of available data $x_1, \cdots, x_n$ for the next n days, and given the profile of your system as expressed by $s_1, \cdots, s_n$ (and starting from a freshly rebooted system on day 1), choose the days on which you're going to reboot so as to maximize the total amount of data you process. Complete the following steps:

1. Define optimizing functions OPT.
2. Define the Bellman equation(s), provide a justification.
3. Describe your algorithm to solve this problem based on your Bellman equation.
4. Analyze the runtime of your algorithm.

Example: Suppose $n = 4$, and the values of x_i and s_i are given by the following table.

Day	1	2	3	4
x_i	10	1	7	7

Day	1	2	3	4
s_i	8	4	2	1

The best solution would be to reboot on day 2 only; this way, you process 8 terabytes on day 1, then 0 on day 2, then 7 on day 3, then 4 on day 4, for a total of 19. (Note that if you didn't reboot at all, you'd process $8 + 1 + 2 + 1 = 12$; and other rebooting strategies give you less than 19 as well.

Solution. Again, we provide all necessary parts, as required.

1. First, we can define our optimizing function as follow:

$$\{-1, 0, 1, 2, \cdots, n\} \times \{0, 1, 2, \cdots, n\} \to \mathbb{R},$$

where OPT evaluated at (i, j) is the optimal total work done by day i assuming the most recent reboot on day j.

2. Then, we define the Bellman equation as:

$$\mathrm{OPT}(i,j) = \begin{cases} 0, & \text{when } i = -1, 0, \\ \max_{0 \le m \le j-2} \mathrm{OPT}(j-1, m) + \sum_{l=j+1}^{i} \min\{x_l, s_{l-j}\}, & \text{when } i \ge 1. \end{cases}$$

Here, we can think of the maximum work achievable on a day is the maximum of a previous reboot adding the maximum work done before and done after.

3. Here, we can think of the implementation as follows:

Let $\{O_{i,j}\}_{i=-1,j=0}^{n,n}$ be the 2 dimensional sequence to hold the optimal values.
Let $\{P_i\}_{i=-1}^{n}$ be the sequence to hold the optimal values up to day i.
Let $O_{i,j} \leftarrow 0$ for all $i = -1, 0$, and let $P_{-1}, P_0 \leftarrow 0$.
for $i \leftarrow 1, 2, \cdots, n$ **do**
  for $j \leftarrow 0, 1, \cdots, i-1$ **do**
    Let $O_{i,j} \leftarrow P_{j-1} + \sum_{l=j+1}^{i} \min\{x_l, s_{l-j}\}$.
  end for
  Let $P_i \leftarrow \max_{0 \le m \le i-1} O_{i,m}$.
end for
The optimal production is P_n.
Let L be the list of rest days, and let $d \leftarrow n$ for backtrack.
while $d > 0$ **do**
  Let $p \leftarrow \arg\max_{0 \le j \le d-1} O_{d,j}$.
  Let $L \leftarrow L \cup \{j\}$, and let $d \leftarrow j$.
end while
Return L as the list of all rest days.

4. Then, we briefly analyze the runtime.

   - For the forward track of computing all the OPT values, we have a doubly for loop that sums the values up to n times within the loop so the runtime is $\mathcal{O}(n^2 \times n) = \mathcal{O}(n^3)$.
   - For the backtrack, it at most track for n times of the while loop and the execution within the while loop is $\mathcal{O}(n)$, so the total runtime is $\mathcal{O}(n^2)$.

   Hence, the runtime for the algorithm is $\mathcal{O}(n^3)$. ⌟

Remark. As a side note, you can also use one variable to do this Dynamic programming problem:

1. First, we can define our optimizing function as follow:

$$\{-1, 0, 1, 2, \cdots, n\} \to \mathbb{R},$$

where OPT evaluated at i is the optimal total work done by day i.

2. Then, we define the Bellman equation as:

$$\mathrm{OPT}(i) = \begin{cases} 0, & \text{when } i = -1, 0, \\ \max_{0 \le m \le j-2} \left\{ \mathrm{OPT}(j-1) + \sum_{l=j+1}^{i} \min\{x_l, s_{l-j}\} \right\}, & \text{when } i \ge 1. \end{cases}$$

Here, we can think of the maximum work achievable on a day is the maximum of a previous reboot adding the maximum work done before and done after.

Problem IV.9. On most clear days, a group of your friends in the Astronomy Department gets together to plan out the astronomical events they're going to try observing that night. We'll make the following assumptions about the events.

- There are n events, which for simplicity well assume occur in sequence separated by exactly one minute each. Thus event j occurs at minute j; if they dont observe this event at exactly minute j, then they miss out on it.

- The sky is mapped according to a one-dimensional coordinate system (measured in degrees from some central baseline); event j will be taking place at coordinate d_j, for some integer value d_j. The telescope starts at coordinate 0 at minute 0.

- The last event, n, is much more important than the others; so it is required that they observe event n.

The Astronomy Department operates a large telescope that can be used for viewing these events. Because it is such a complex instrument, it can only move at a rate of one degree per minute. Thus they do not expect to be able to observe all n events; they just want to observe as many as possible, limited by the operation of the telescope and the requirement that event n **must be observed**.

We say that a subset S of the events is viewable if it is possible to observe each event $j \in S$ at its appointed time j, and the telescope has adequate time (moving at its maximum of one degree per minute) to move between consecutive events in S.

Problem: Given the coordinates of each of the n events, find a viewable subset of maximum size, subject to the requirement that it should contain event n.

Example: Suppose the one-dimensional coordinates of the events are as shown:

Event	1	2	3	4	5	6	7	8	9
Coordinate	1	-4	-1	4	5	-4	6	7	-2

Then the optimal solution is to observe events 1, 3, 6, 9. Note that the telescope has time to move from one event in this set to the next, moving at one degree per minute.

(a) Show that the following greedy algorithm does not correctly solve this problem, by giving an instance on which it does not return the correct answer.

Algorithm. Starry night greedy algorithm.

1: Mark all events j with $|d_n - d_j| > n - j$ as illegal
2: Mark all other events as legal
3: Initialize current coordinate 0 at time 0
4: **while** not at the end of event sequence **do**
5: Find the earliest legal event j that can be reached without exceeding the maximum movement rate of the telescope
6: Add j to the set S
7: Update current position to be coordinate d_j at minute j
8: **end while**
9: **return** the set S.

In your example, say what the correct answer is and also what the algorithm above finds.

Solution. Here, we construct the following counter example:

Event	1	2	3	4
Coordinate	-1	0	2	1

The greedy algorithm would pick the events 4 and 3, but in fact, we can pick 4, 2, and 1 to observe more events. ⌟

(b) Give an efficient algorithm that takes values for the coordinates $d_1, \cdots, d_n$ of the events and returns the size of an optimal solution, by completing the following:

1. Define optimizing functions OPT.
2. Define the Bellman equation(s), provide a justification.
3. Describe your algorithm to solve this problem based on your Bellman equation.
4. Analyze the runtime of your algorithm.

Solution. Again, we provide all necessary parts, as required.

1. First, we can define our optimizing function as follow:

$$\{0, 1, 2, \cdots, n\} \to \mathbb{N},$$

where OPT evaluated at i is the maximum possible events to observe so that event i can be observed.

2. Then, we define the Bellman equation as:

$$\text{OPT}(i) = \begin{cases} 0, & \text{when } i = 0, \\ \max_{0 \le j \le i-1} \text{OPT}(j) \cdot \mathbb{1}(|d_i - d_j| < |i - j|) + 1, & \text{when } i \ge 1, \end{cases}$$

where $\mathbb{1}$ is the characteristic equation such that:

$$\mathbb{1}(|d_i - d_j| < |i - j|) = \begin{cases} 0, & \text{when } |d_i - d_j| \ge |i - j|, \\ 1, & \text{when } |d_i - d_j| < |i - j|. \end{cases}$$

Here, we can think of the maximum observable number of events is based off the previous observed day plus 1 if the previous day is observable.

3. Here, we can think of the implementation as follows:

Let $\{O_i\}_{i=0}^n$ be the sequence to hold the optimal values, let $O_0 \leftarrow 0$, initialize the rest as $-\infty$.
Let $\{P_{i,j}\}_{i=0}^n$ be the sequence to hold the optimal values, let $O_0 \leftarrow 0$, initialize the rest as $-\infty$.
for $i \leftarrow 1, 2, \cdots, n$ **do**
  for all $j \in \{1, \cdots, i-1 : |d_i - d_j| < |i - j|\}$ **do**
    Let $P_{i,j} \leftarrow O_j + 1$.
    Let $O_i \leftarrow \max\{O_i, O_j + 1\}$.
  end for
end for
The optimal number of observation is O_n.
Let $L = \{n\}$ be the list of observable days, and let $d \leftarrow n$ for backtrack.
while $d > 0$ **do**
  Let $p \leftarrow \arg\max_{0 \le j \le d-1} P_{d,j}$.
  Let $L \leftarrow L \cup \{p\}$, and let $d \leftarrow p$.
end while
Return L as the list of all observable moments.

4. Then, we briefly analyze the runtime.

   - For the forward track of computing all the OPT values, we have a doubly for loop that is constant time within the loop, so the runtime is $\mathcal{O}(n^2 \times n) = \mathcal{O}(n^2)$.
   - For the backtrack, it at most track for n times of the while loop and the execution within the while loop is $\mathcal{O}(n)$, so the total runtime is $\mathcal{O}(n^2)$.

   Hence, the runtime for the algorithm is $\mathcal{O}(n^2)$. ⌟

V Problem Set 5

Problem V.1. "You can win!" **[6 points]**

Its nearing the end of the Spring 2025 semester. Youre taking n courses, each with a final project that still needs to be done. Each project will be graded on the following scale: it will be assigned an integer number on a scale of 1 to g, where $g > 1$. Higher number means better grades. Your goal, of course, is to maximize your **average grade** on the n projects.

You have a total of $H > n$ hours in which to work on the n projects cumulatively, and you want to decide how to divide up this time. For simplicity, assume that H is a positive integer, and youll spend an integer number of hours on each project (no fractions).

To figure out how best to divide up your time, youve come up with a set of functions $\{f_i : i = 1, 2, \ldots, n\}$ (rough estimates, of course) for each of your n courses. If you spend $h \leq H$ hours on the project for course i, youll get a grade of $f_i(h)$ in course i. You may assume that the functions f_i are nondecreasing: if $h < h'$, then $f_i(h) \leq f_i(h')$. This simply means that more the amount of time you spend on a course, the higher the grade that you will get in that course. However, this might imply that your grade in another course goes down as the total amount of time is limited. So, you need to figure out how to divide your total time so that your **average grade** on the n projects is as high as possible.

The Problem: Given these functions $\{f_i\}$, decide how many hours to spend on each project (in integer values only) so that your **average grade**, as computed according to the f_i, is as large as possible. In order to be efficient, the running time of your algorithm should be a polynomial in n, g, and H; none of these quantities should appear as an exponent in your running time.

(a) Give an efficient algorithm that takes functions $\{f_i\}$, and returns the optimal schedule. That is, the algorithm returns the hours allocated to each course that maximize the average grade. Complete the following: [5 points]

(i) Define optimizing function(s) OPT.

(ii) Define the Bellman equation(s), provide a justification.

(iii) Describe your algorithm to solve this problem based on your Bellman equation, your algorithm should output the hours allocated to each course that maximize the average grade.

(iv) Analyze the runtime of your algorithm.

Solution. Here, we first note a few observations to make the problem easier:

- Finding the highest **average score** for a fixed number of class is the equivalent of finding the higher **total score**, so we can just make our goal to maximize the **total score** instead.
- The order of working **does not matter**, so we can complete the course projects in "sequential order," that is, we do all the work for course i prior to doing any work for course j for all $i < j$.

Then, we define all necessary components, as requested:

(i) We define the optimizing function as:

$$\text{OPT} : \{0, 1, 2, \cdots, H\} \times \{1, 2, \cdots, n\} \to \mathbb{R},$$

where OPT evaluated at (i, j) is the maximum possible total score earned for the first i hours, where works are done in "sequential order," and the work done up to hour j contains no classes after j.

(ii) Then, we give an explicit Bellman equation.

$$\text{OPT}(i, j) = \begin{cases} f_1(i), & \text{when } j = 1, \\ \max_{0 \le k \le i} \left\{ \text{OPT}(k, j-1) + f_j(i-k) \right\}, & \text{when } j > 1. \end{cases}$$

Here, the Bellman equation at each recursive definition is the maximum of the time additional to the previous our doing jobs earlier than (and not including) j.

(iii) Therefore, we give the algorithm for the problem:

Let $\{O_{i,j}\}_{i=0,j=0}^{H,n}$ be a doubly-indexed sequence (or 2D array) holding the results of the OPT function. Let $O_{i,1} \leftarrow f_1(i)$ for all $i \leftarrow 0, 1, \cdots, H$.
for $j \leftarrow 2, 3, \cdots, n$ **do**
  for $i \leftarrow 0, 1, 2, \cdots, H$ **do**
    Compute $O_{i,j} \leftarrow \max_{0 \le k \le i} \{O_{k,j-1} + f_j(i-k)\}$.
  end for
end for
Let $\{h_j\}_{j=1}^{n}$ be a sequence (or 1D array) holding the number of work hour for each class j.
Let $h_1 \leftarrow H$ represent the remaining number of hours left for the first subject.
for $j \leftarrow n, n-1, \cdots, 2$ **do**
  Let $h_j \leftarrow \arg\max_{0 \le k \le h} \{O_{h-k,j-1} + f_j(k)\}$, and let $h_1 \leftarrow h_1 - h_j$.
end for
return $\{h_j\}_{j=1}^{n}$ as the number of hours allocated to each class, and **return** $O_{H,n}/n$ as the maximized average grade.

(iv) Then, we analyze the runtime of the algorithm.

- The initialization of data structures is no more than $\mathcal{O}(nH)$.
- For the forward-track (computing all OPT values), for each column, there are H rows, and each computation involves constant time of all H rows in the previous column, so the total runtime for each column is $\mathcal{O}(H^2)$, and accounting for the n rows, the runtime is $\mathcal{O}(nH^2)$.
- For the backward-track (finding the allocation of time), for each j, it backtracks the comparison of all previous column rows, so the runtime is $\mathcal{O}(H)$, and for a total of n column, the total runtime is $\mathcal{O}(nH)$.

Hence, the total runtime is $\boxed{\mathcal{O}(nH^2)}$. ⌟

(b) You are taking 3 courses C++, Java and Python $(n = 3)$ and you have a total of 5 hours $(H = 5)$ to work on the projects. The functions f_i for each course is as follows:

Hours spent	0	1	2	3	4	5
C++ grades	0	1	3	5	5	5
Java grades	0	3	3	4	5	6
Python grades	0	1	4	4	4	7

Use your algorithm to find the optimal distribution of time to maximize the average grade. Your algorithm should output the grades achieved in each course. Also calculate the number of hours spent on each course.

Solution. Here, we give the results of the OPT function in a table form, with each column as course and each row as the number of hours.

	`C++` (1)	`Java` (2)	`Python` (3)
0	0	0	0
1	1	3	3
2	3	4	4
3	5	6	7
4	5	8	8
5	5	8	10

Figure V.1. OPT function results, with the backtrack trace marked by green.

This sample run gives the best average score as $\boxed{10/3}$ and the distribution of time is:

$\boxed{\texttt{C++} \text{ for 2 hours, } \texttt{Java} \text{ for 1 hour, and } \texttt{Python} \text{ for 2 hours}}$.

⌟

Problem V.2. Maryland State Fair Champion **[6 points]**

Maryland State Fair will be held in Baltimore County (Timonium) from August 21 to September 7 2025. They are hosting a new tournament this year, and you and Jamie have made it to the finals!

In the final competition, you'll be competing for prizes from a prize pool denoted by $P = [p_1, \cdots, p_n]$, where $p_i > 0$ denotes the value of the prize at position i. On each turn, you have the option to take either the first or the last prize (i.e., p_1 or p_n) from the pool, and Jamie will do the same with the smaller prize pool. This continues until no prizes remain. Keep in mind that Jamie will use the best possible strategy to secure her victory.

Assume that you are making the first move.

You initially decide to use a greedy strategy, but you soon realize that it is not optimal. The greedy algorithm that you tried is to pick p_i or p_j, whichever is of higher value. For instance, let $P = [5, 9, 3, 2]$. Your first choice is then to pick 5, as 5 is more than 2. After you have chosen 5, the new $P = [9, 3, 2]$. Now, Jamie picks 9. Then new $P = [3, 2]$. Based on your greedy algorithm, you pick 3 and finally Jamie picks 2. This gives you a total of $5 + 3 = 8$ and Jamie a total of $9 + 2 = 11$, and you end up losing the competition. The optimal strategy for you is to choose 2 at the first step. Then the new $P = [5, 9, 3]$. Now whether Jamie picks 5 or 3, you can choose 9 in the next step and win the competition with a total of 11.

Consequently, you decide to employ a dynamic programming approach.

(a) You need to design an efficient algorithm that takes the prize pool $P = [p_1, \cdots, p_n]$, and determines the maximum possible prize value that you can win. Assume that Jamie will use the best possible strategy. [5 points]

(i) Define optimizing function(s) OPT.

(ii) Define the Bellman equation(s), provide a justification.

(iii) Describe your algorithm to solve this problem based on your Bellman equation.

Note: We are not asking you for the runtime analysis for this problem. You may, however, do the runtime computation for your own practice.

Solution. Again, we first note some small remarks to reduce the problem easier:

- Note that with fixed number of total points, you and your opponent would split the points, so you can interpret as losing the points from the points left when the opponent is gaining points.

Then, we give each desired components:

(i) We define the optimization function as:

$$\text{OPT} : \{1, 2, \cdots, n\}^2 \to \mathbb{R},$$

where OPT evaluated at (i, j) is the maximum possible taken when one is picking on the sub-pool $\mathcal{P} = [p_i, \cdots, p_j]$. Note that if $i > j$, the result is trivially 1.

(ii) Now, why don't we give an explicit Bellman equation, as follows:

$$\mathrm{OPT}(i,j) = \begin{cases} 0, & \text{when } i > j, \\ p_i, & \text{when } i = j, \\ \max\{p_i, p_j\}, & \text{when } i+1 = j, \\ \sum_{\ell=i}^{j} p_\ell - \min\{\mathrm{OPT}(i+1,j), \mathrm{OPT}(i,j-1)\}, & \text{when } i+1 < j. \end{cases}$$

Here, when there is zero, one, or two prizes left, the pick is trivially selecting the only option or just the larger one. When there are more than two choices, our gain is at most the total prizes remaining subtracting the small of what your opponent can gain.

(iii) Then, we give the algorithm here.

Let $\{O_{i,j}\}_{i=0,j=0}^{n,n}$ be a doubly-indexed sequence (or 2D array) holding the results of the OPT function. Let all values initialized to 0.
for $j \leftarrow 1, 2, \cdots, n$ **do**
  Let $R \leftarrow 0$ be the slack variable for the remaining prizes left.
  for $i \leftarrow j, j-1, \cdots, 1$ **do**
    Let $R \leftarrow R + p_i$.
    if $i = j$ **then** Let $O_{i,j} \leftarrow p_i$.
    else if $i + 1 = j$ **then** Let $O_{i,j} \leftarrow \max\{p_i, p_j\}$.
    else Let $O_{i,j} \leftarrow R - \min\{O_{i+1,j}, O_{i,j-1}\}$
    end if
  end for
end for
return $O_{1,n}$ as the maximum you can earn. If $O_{1,n} \geq \min\{O_{2,n}, O_{1,n-1}\}$, you will *not lose*.

Remark. Note that the questions has not indicated how to classify the case of a draw, we would give the conclusion that you would not lose when you have more prizes in total. If you want to classify a draw as a win, then not lose is equivalent to win. ⌟

(b) Now is the competition time! [1 point]

You discover the prize pool is $P = [5, 2, 4, 1, 8, 7]$. Using the Bellman Equation you have provided above, compute the maximum prize value that you can win. Is it possible to win the competition against Jamie, in this particular instance? For this example, please include the sequence of moves for you and Jamie if both of you use the optimal strategy?

Note: Your algorithm is not required to output the optimal schedule (i.e., sequence of moves).

Solution. Here, we give the results of the OPT function in a table form, with each column being the value of i and each row as the value of j.

	1	2	3	4	5	6
1	5	0	0	0	0	0
2	5	2	0	0	0	0
3	7	4	4	0	0	0
4	9	3	4	1	0	0
5	11	12	11	8	8	0
6	17	10	12	8	8	7

Figure V.2. OPT function results, with initialized zeros in gray and our optimal result in green.

This sample run gives that the maximum prize value that we can win is $\boxed{17}$, and we can win Jamie, who gets 10 points.
Note that the strategy does **not** give a unique choice at some step, so we will give all the possible choices.

- For the first step, we pick 5.
- For the second step, Jamie can pick 2 or 7:
  - Suppose Jamie picks 2 for the second step.
    - For the third step, we pick 4.
    - For the fourth step, Jamie can pick 1 or 7.
    - Whatever Jamie picks on the fourth step, we pick 8 for the fifth step.
    - For the final step, Jamie pick 7 or 1, which is the one that was not selected in step 4.
  - Jamie picks 7 for the second step.
    - For the third step, we pick 8.
    - For the fourth step, Jamie can pick 1 or 2.
    - Whatever Jamie picks on the fourth step, we pick 4 for the fifth step.
    - For the final step, Jamie pick 2 or 1, which is the one that was not selected in step 4.

Eventually, we obtain $[5, 8, 4]$ and Jamie obtains $[2, 1, 7]$, assuming the order of picking the smallest of left and right when picking the left and right gives the same overall result. ⌟

Problem V.3. "Donate blood, save a life!" **[6 points]**

Statistically, the arrival of Winters typically results in increased accidents and increased need for emergency medical treatment, which often requires blood transfusions. Consider the problem faced by Johns Hopkins Hospital that is trying to evaluate whether its blood supply is sufficient.

The basic rule for blood donation is the following. A person's own blood supply has certain antigens present (we can think of antigens as a kind of molecular signature); and a person cannot receive blood with a particular antigen if their own blood does not have this antigen present. Concretely, this principle underpins the division of blood into four types: A, B, AB, and O. Blood of type A has the A antigen, blood of type B has the B antigen, blood of type AB has both, and blood of type O has neither. Thus, patients with type A can receive only blood types A or O in a transfusion, patients with type B can receive only B or O, patients with type O can receive only O, and patients with type AB can receive any of the four types.

(a) Let s_O, s_A, s_B, and s_{AB} denote the supply in whole units of the different blood types on hand. Assume that JHH knows the projected demand for each blood type d_O, d_A, d_B, and d_{AB} for the coming week. Give a polynomial-time algorithm to evaluate if the blood on hand would suffice for the projected need. [2 points]

Solution. Here, we can very easily turn this into a network flow problem.

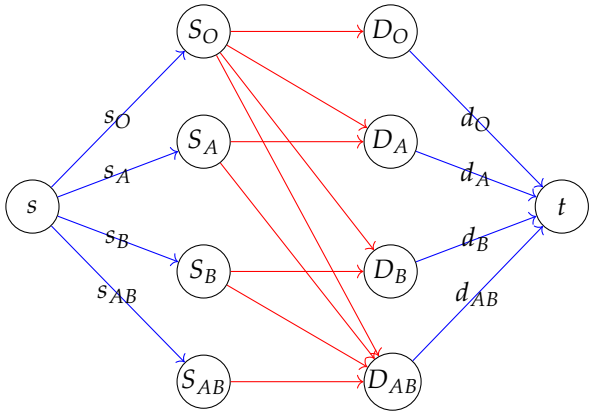


Figure V.3. Network flow graph $\mathcal{G}$ from s to t, with red edges having capacity ∞.

Then, we can give the algorithm:

Fill in variables $s_O, s_A, s_B, s_{AB}, d_O, d_A, d_B, d_{AB}$ on to the network graph $\mathcal{G}$ (Figure V.3).

Conduct the Ford-Fulkerson `algorithm` on $\mathcal{G}$, let $M \leftarrow$ the capacity of st-cut.

If $M \geq d_O + d_A + d_B + d_{AB}$, then the blood is sufficient, otherwise, the blood is not sufficient.

This runtime of the Ford-Fulkerson `algorithm` is polynomial time, *i.e.*, $\mathcal{O}\big((s_O + s_A + s_B + s_C) \cdot 9\big)$, and the initializations and comparisons are of $\mathcal{O}(1)$, so the total time is still polynomial time with respect to total supply. ⌟

Consider the following example. Over the next week, they expect to need at most 100 units of blood. The typical distribution of blood types in U.S. patients is roughly 45 percent type O, 42 percent type A, 10 percent type B, and 3 percent type AB. JHH wants to know if the blood supply it has on hand would be enough if 100 patients arrive with the expected type distribution. There is a total of 105 units of blood on hand. The table below gives these demands, and the supply on hand.

Blood Type	Supply	Demand
O	50	45
A	36	42
B	11	10
AB	8	3

Is the 105 units of blood on hand enough to satisfy the 100 units of demand?

(b) Find an allocation that satisfies the maximum possible number of patients. Use an argument based on a minimum-capacity cut to show why not all patients can receive blood. [2 points]

Solution. Here, we conduct the Ford-Fulkerson `algorithm` on the initialized graph for the st-cut problem.

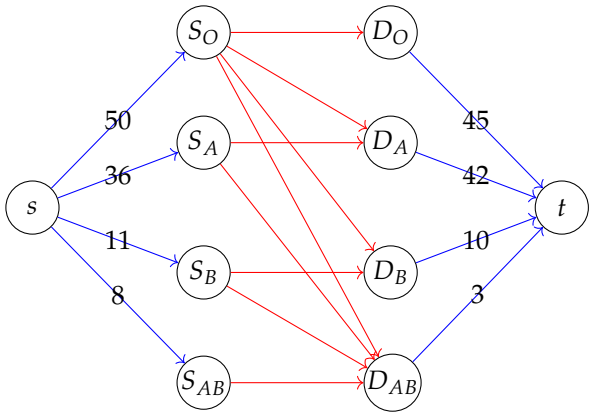


Figure V.4. Network flow problem from s to t with concrete example, and the red edges have capacity ∞.

- Find path $s \to S_O \to D_O \to t$:

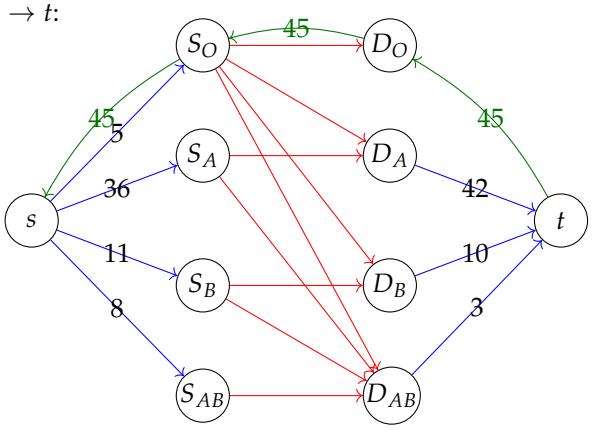


Figure V.5. Residual graph after the first conduct of the algorithm.

- Find path $s \to S_A \to D_A \to t$:

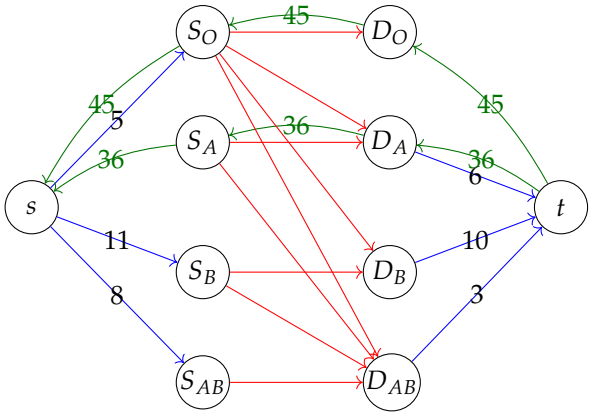


Figure V.6. Residual graph after the second conduct of the algorithm.

- Find path $s \to S_B \to D_B \to t$:

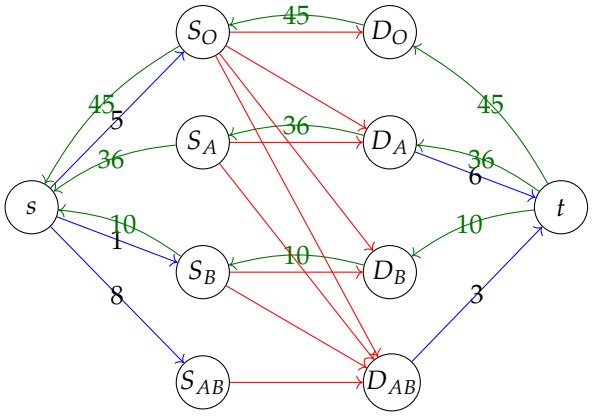


Figure V.7. Residual graph after the third conduct of the algorithm.

- Find path $s \to S_{AB} \to D_{AB} \to t$:

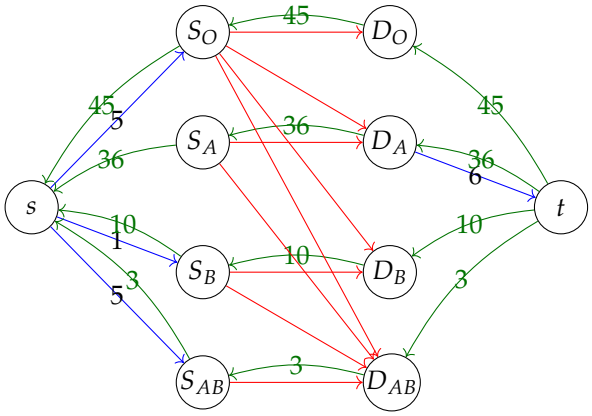


Figure V.8. Residual graph after the fourth conduct of the algorithm.

- Find path $s \to S_O \to D_A \to t$:

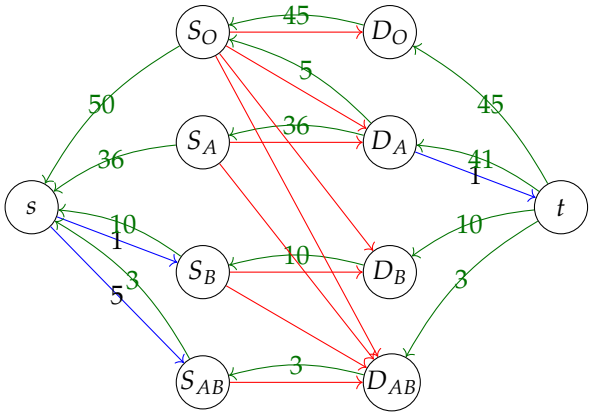


Figure V.9. Residual graph after the fifth conduct of the algorithm.

- Note that there are no paths from s to t, so the algorithm terminates, giving the network as:

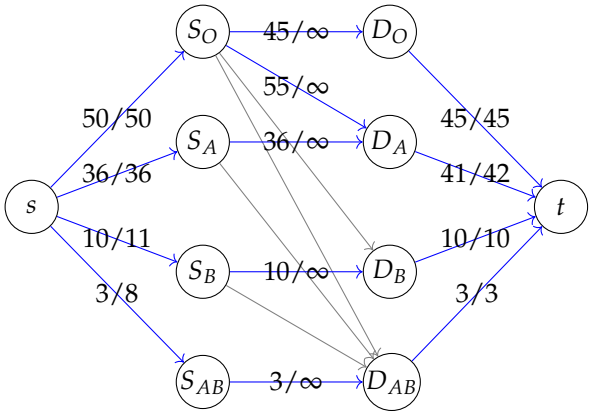


Figure V.10. Network flow result from s to t with concrete example, and the gray edges have capacity $0/\infty$.

Here, to build the min-cut argument, we classify the graph into two colors, red and blue.

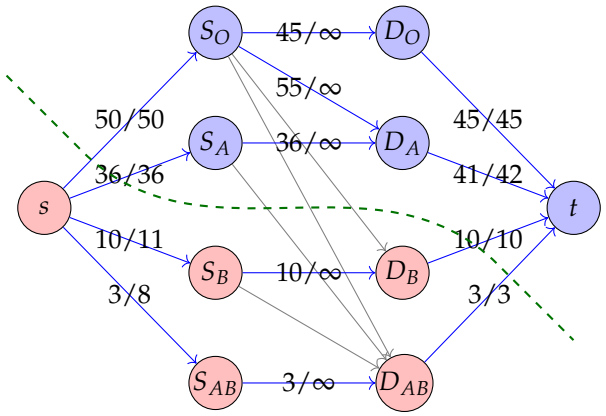


Figure V.11. Network flow result from s to t with concrete example with red and blue separation, and the gray edges have capacity $0/\infty$, and the green dashed line is the cut.

Note that we have the cut as $A = \{s, S_B, S_{AB}, D_B, D_{AB}\}$ and $B = \{S_O, S_A, D_O, D_A, t\}$. The capacity of the cut is:

$$50 + 36 + 10 + 3 = 99 < 100 = 45 + 42 + 10 + 3.$$

Since the capacity of the minimum-cut is smaller than the needed capacity of the blood, not all patients can receive blood. ⌟

(c) Also, provide an explanation for this fact that would be understandable to the clinic administrators at JHH, who have not taken a course on Intro. Algorithms. (So, for example, this explanation should not involve the words *flow, cut*, or *graph* in the sense we use them in this course.) [2 points]

Solution. To explain this to a doctor, we note that the demand for O and A blood type is in total:

$$45 + 42 = 87.$$

It is not hard to observe that people demanding for O and A blood type can only take blood from O and A, but in terms of supply for O and A blood type is in total:

$$50 + 36 = 86.$$

Hence, the total amount of supply for O and A blood is less than the amount of demand of O and A blood, hence, there will just be not enough of blood for O and A type however we arrange. ⌟

Problem V.4. We Are Cooking! **[6 points]**

Suppose that you and your friend Tian live, together with $n-2$ other people, at a popular off-campus cooperative apartment, the Academy on Charles. Over the next n nights, each of you is supposed to cook dinner for the co-op exactly once, so that someone cooks on each of the nights.

Of course, everyone has scheduling conflicts with some of the nights (e.g., exams, concerts, games, hackathons etc.), so deciding who should cook on which night becomes a tricky task. For concreteness, lets label the people $\{p_1, ..., p_n\}$,

the nights $\{d_1, ..., d_n\}$;

and for each person p_i, theres a set of nights $S_i \subset \{d_1, ..., d_n\}$ when they are **not** able to cook.

A feasible dinner schedule is an assignment of each person to a different night, such that

- each person cooks on exactly one night,
- there is someone cooking on each night, and
- if p_i cooks on night d_j, then $d_j \notin S_i$.

(a) Describe a bipartite graph G so that G has a perfect matching if and only if there is a feasible dinner schedule for the co-op. [2 points]

Solution. Here, we define the graph $G := (\{d_1, \cdots, d_n\} \cup \{p_1, \cdots, p_n\}, E)$, where:

- The nodes are the days and the people.
- $E \subset \{d_1, \cdots, d_n\} \times \{p_1, \cdots, p_n\}$ be the set of edges. There does not exist any edge within the set $\{d_1, \cdots, d_n\}$ or $\{p_1, \cdots, p_n\}$, and for any $d_i \in \{d_1, \cdots, d_n\}$ and $p_j \in \{p_1, \cdots, p_n\}$, there exists an edge in between if and only if $d_i \notin S_j$.

Here, G would have a perfect matching if and only if there is a feasible dinner schedule for the co-op.
To make this clearly, we will give a graph for demonstration:

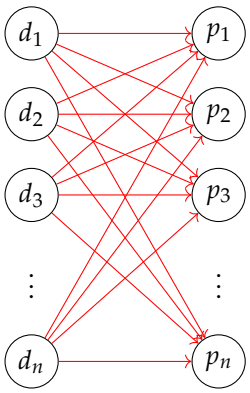


Figure V.12. Bipartite graph for days and people in the graph, with red edges having weigh 1 (or existing) only if the person is avaiable on that day, and 0 (or not existing) otherwise.

⌟

(b) Your friend Tian takes on the task of trying to construct a feasible dinner schedule. After great effort, he constructs what he claims is a feasible schedule and then heads off to classes for the day.

Unfortunately, when you look at the schedule that he created, you notice a big problem. $n-2$ of the people at the co-op are assigned to different nights on which they are available: no problem there. But for the other two people, p_i and p_j, and the other two days, d_k and d_l, you discover that he has accidentally assigned both p_i and p_j to cook on night d_k, and assigned no one to cook on night d_l.

You want to fix Tians mistake but without having to recompute everything from scratch. Show that its possible, using his almost correct schedule, to decide in only $O(n^2)$ time whether there exists a feasible dinner schedule for the co-op. If one exists, you should also output it. [4 points]

Solution. Here, we will give a modified graph of Figure V.12 for demonstration:

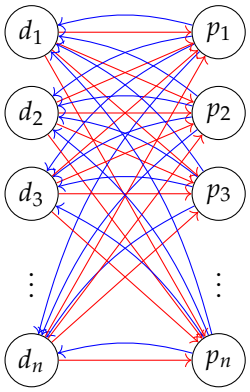


Figure V.13. Modified bipartite graph for days and people in the graph, with red edges existing only if the person is avaiable on that day, the blue edges existing only if Tian assigns the person on that day.

Then, by the condition of the question, there is some d_l that has no blue edge pointing towards it, and there exists some d_k having two blue edges pointing to it, say p_i and p_j.
Here, we can simply conduct a BFS starting from d_l, if the BFS gets to p_i or p_j, we terminate the search, and get the path from d_l to p_i or p_j, then, we can just delete all the blue edges in the path from Tian's assignment and add all the inverse of red edges in the path to Tian's assignment to make the modification. Formally, this algorithm can be written as a recursive algorithm to seek for a solution.

Let $\{a_m\}_{m=1}^n \leftarrow$ Tian's assignment.
function FIND-SUB($l, \{a_m\}_{m=1}^n, \mathcal{B}$)
  Let $\{b_m\}_{m=1}^n \leftarrow \{a_m\}_{m=1}^n$.
  if $d_l \notin S_i$ **then** Let $b_l \leftarrow p_i$, and **return** $\{b_m\}_{m=1}^n$.
  else if $d_l \notin S_j$ **then** Let $b_l \leftarrow p_j$, and **return** $\{b_m\}_{m=1}^n$.
  else
    for all $1 \le \ell \le n$ such that $d_l \notin S_\ell$ and $\ell \notin \mathcal{B}$ **do**
      Let $d_p \leftarrow$ the night in which p_ℓ cook in $\{b_m\}_{m=1}^n$.
      con, $\{c_m\}_{m=1}^n \leftarrow$ FIND-SUB$(p, \{b_m\}_{m=1}^n, \mathcal{B} \cup \{\ell\})$.
      if con is True **then return** True, $\{c_m\}_{m=1}^n$.
      end if

end for
return False
end if
end function
Let con, $\{c_m\}_{m=1}^n \leftarrow$ FIND-SUB$(l, \{a_m\}_{m=1}^n, \emptyset)$, if con is False, no solution, otherwise $\{c_m\}_{m=1}^n$ is the new assignment.

Remark. It should be noted that this algorithm is modified from a network flow problem. Assume we have a source and a target, finding a path from s to t when deleting an edge is the equivalent of starting from d_l (since there is no path from s to d_l yet), and note that since p_j and p_i are having two paths towards it, removing an edge is effectively releasing one of the person, so ending at p_i to p_j is the same.

Also as a note, the expected algorithm (equivalent to above) would be on a revised graph as follow steps:

1. Construct the network problem from s to d_i's to p_j's to t, where s is the source, t is the target, and we have intermediate days and people.

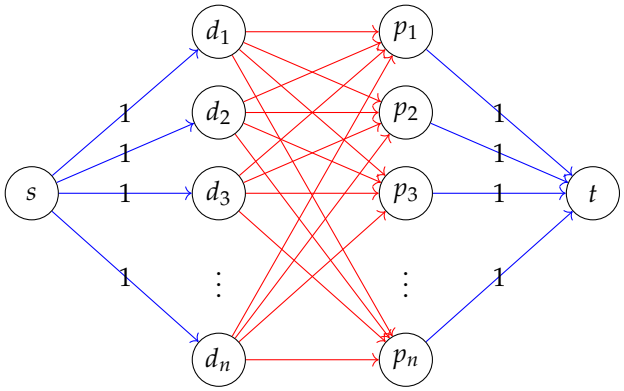


Figure V.14. Initialized network problem, the red edges are of weight ∞ from d_i to p_j if $d_i \notin S_j$, and 0 otherwise.

2. Then, we apply what Tian has arranged on his planning, except for the path $s \to d_k \to p_j \to t$, while applying his planning, we keep the residue graphs, and by the end of his planning, it would be:

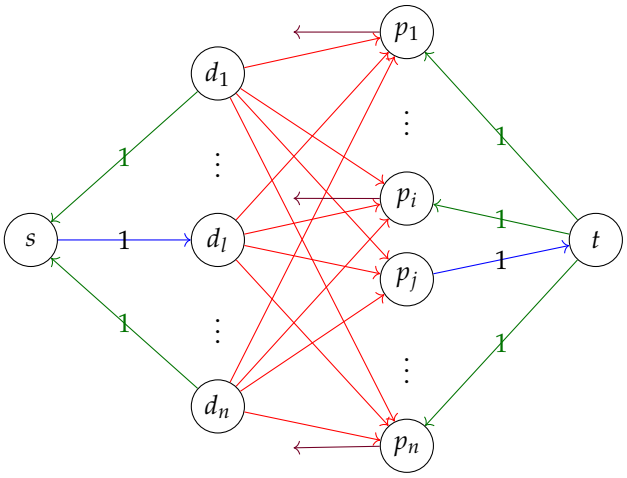


Figure V.15. After applying Tian's plan except a path on network problem, the red edges are of weight ∞ from d_i to p_j if $d_i \notin S_j$, and 0 otherwise; the green and purple arrows are the inverse edges created by the residual graph of weight 1.

3. Then, we effectively have a network flow problem, and we want to see if we can find a path from s to t, that is some s going towards d_l, then with some getting between the d and p layers, and eventually get to p_j to t. The Ford-Fulkerson `algorithm` will still give a network of capacity n if it was successful, which is our desired result, but it could also give nothing when there is no solution.

Note that this would be closer to the algorithm presented in the solutions, but it is equivalently finding a path from d_l to p_i or p_j with a single BFS, that is a little bit faster.
Note that doing a BFS or finding a 1 additional capacity with at most $n^2 + n$ edges is $\mathcal{O}(n^2)$, so the runtime is as desired. ⌟

Problem V.5. Gone With The Wind **[6 points]**

Your friends are involved in a large-scale atmospheric science experiment. They need to get good measurements on a set S of n different conditions in the atmosphere (such as temperature, moisture, wind speeds, ozone level etc), and they have a set of m balloons that they plan to send up to make these measurements.**Each balloon can make at most two measurements.**

Unfortunately, not all balloons are capable of measuring all conditions, so for each balloon $i = 1, \ldots, m$, they have a set S_i of conditions that balloon i can measure. Finally, to make the results more reliable, they plan to take each measurement from k different balloons. Note that a single balloon should not measure the same condition twice. They are having trouble figuring out which conditions to measure on which balloon.

Example: Suppose that $k = 2$, there are $n = 4$ conditions labeled c_1, c_2, c_3, c_4, and there are $m = 4$ balloons that can measure conditions, subject to the limitation that balloons 1 and 2 can measure the first three conditions, that is, $S_1 = S_2 = \{c_1, c_2, c_3\}$, and balloons 3 and 4 can measure the first, the third and the fourth condition, that is, $S_3 = S_4 = \{c_1, c_3, c_4\}$. Then one possible way to make sure that each condition is measured $k = 2$ times is to have

- balloon 1 measure conditions c_1, c_2,
- balloon 2 measure conditions c_2, c_3,
- balloon 3 measure conditions c_3, c_4, and
- balloon 4 measure conditions c_1, c_4.

(a) Give a polynomial-time algorithm that takes the input to an instance of this problem (the n conditions, the sets S_i for each of the m balloons, and the parameter k) and decides whether there is a way to measure each condition by k different balloons, while each balloon only measures at most two conditions. You don't need to give a proof for runtime or a proof of correctness. However, you do need to explain your algorithm. [3 points]

Solution. Here, we can trivially transform the problem into a network flow problem. Consider the following graph:

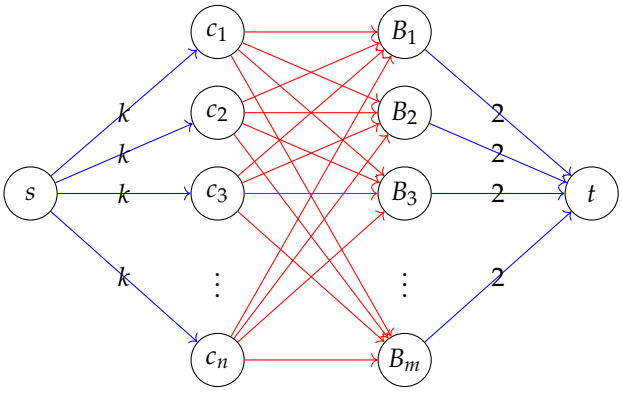


Figure V.16. Network flow graph $\mathcal{G}$ from s to t, with red edges from c_i to B_j having capacity 1 if $c_i \in S_j$, and 0 otherwise.

Here, we then give an algorithm:

Fill in the information about k and all edges between c_i to B_j for all $i \in \{1, \cdots, n\}$ and $j \in \{1, \cdots, m\}$ on the above network (Figure V.16).
Conduct the Ford-Fulkerson `algorithm` on $\mathcal{G}$, let $M \leftarrow$ the capacity of st-cut, and record the flow assignment from the algorithm.
If $M = nk$, then there exists an assignment, which follows all the active flows from c_i to B_j. Otherwise, there exists no assignments.

The graph should be self-explanatory, but to elaborate, it limits every balloon to be used at most 2 times, each experiment setup done at most k times, and the middle red edges existing once only if the balloon is compatible with the experiment setup. Hence, the maximum flow would be the same as nk if and only if we can conduct nk experiments with the desired balloon assignment. ⌟

(b) You show your friends the solution computed by your algorithm from (a), and to your surprise they reply, "This won't do at allone of the conditions is only being measured by balloons from a single subcontractor?' You hadn't heard anything about subcontractors before; it turns out there's an extra wrinkle they forgot to mention....

Each of the balloons is produced by one of three different subcontractors involved in the experiment. A requirement of the experiment is that there be no condition for which all the k measurements come from balloons produced by the same subcontractor.

For example, suppose balloon 1 comes from the first subcontractor, balloons 2 and 3 come from the second subcontractor, and balloon 4 comes from the third subcontractor. Then our previous solution no longer works, as both of the measurements for condition c_3 were done by balloons from the second subcontractor. However, we could use balloons 1 and 2 to each measure conditions c_1, c_2, and use balloons 3 and 4 to each measure conditions c_3, c_4, to get an acceptable solution.

Explain how to modify your polynomial-time algorithm for part (a) into a new algorithm that decides whether there exists a solution satisfying all the conditions from (a), plus the new requirement about the subcontractors. You don't need to give a proof for runtime or a proof of correctness. However, you do need to explain your algorithm. [3 points]

Solution. Here, we need to modify our construction of graph $\mathcal{G}$ on Figure V.16 with an additional layer of the subcontractor condition for each condition. Suppose there are $\ell = 3$ total subcontractors:

- There will be the original layers consisted of s as source, $c_1, \cdots, c_n$ as the experiment conditions, $s_{1,1}, \cdots, s_{n,\ell}$ as subcontractor information, $B_1, \cdots, B_n$ as the balloons, and t as the terminus.

Specifically, we define the weights from $s_{i,l}$ to B_j as:

$$\text{weight}(s_{i,l} \to B_j) = \begin{cases} 1, & \text{when } c_i \in S_j \text{ and } B_j \text{ is manufactured by subcontractor } l, \\ 0, & \text{otherwise,} \end{cases} \tag{1}$$

and the weights between c layer and s layers are all of weights $k - 1$.
Here, the graph becomes:

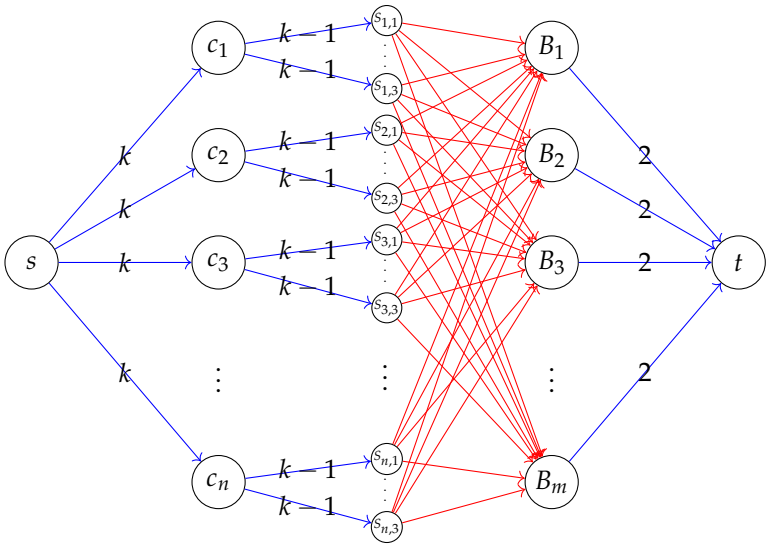


Figure V.17. Network flow graph $\mathcal{G}'$ from s to t, with red edges weights defined concretely in (1).

Likewise, the algorithm is almost the same:

Fill in the information about k and all edges between $s_{i,\ell}$ to B_j for all $i \in \{1, \cdots, n\}$, $j \in \{1, \cdots, m\}$, and $\ell \in \{1, \cdots, k\}$ on the above network (Figure V.17).
Conduct the Ford-Fulkerson `algorithm` on $\mathcal{G}'$, let $M \leftarrow$ the capacity of st-cut, and record the flow assignment from the algorithm.
If $M = nk$, then there exists an assignment, which follows all the active flows from c_i to B_j (through $s_{i,\ell}$ for a unique $\ell \in \{1, \cdots, k\}$). Otherwise, there exists no assignments.

Again, the graph should be self-explanatory, but to elaborate:

- Likewise to part (a), it limits every balloon to be used at most 2 times.
- Additionally, with the new connection layer between experiment and the subcontractor with weight $k - 1$, it limits each setup done with at least two subcontractor.
- For the red edges with concrete definition of weights, there will be an edge only if the balloon is compatible with the experiment setup and it is has the designated subcontractor, which, in this case, guarantees that there is an edge if and only if it can be used for the setup and subcontractor exactly once.

Hence, the revised model from Figure V.16 into Figure V.17 will resolve the additional condition. ⌟

Problem V.6. Suppose you're a consultant for the Ergonomic Architecture Commission, and they come to you with the following problem. They're really concerned about designing houses that are "user-friendly," and they've been having a lot of trouble with the setup of light fixtures and switches in newly designed houses.

Consider, for example, a one-floor house with n light fixtures and n locations for light switches mounted in the wall. You'd like to be able to wire up one switch to control each light fixture, in such a way that a person at the switch can see the light fixture being controlled. Sometimes this is possible and sometimes it isn't. Consider the two simple floor plans for houses in Figure Figure V.18. There are three light fixtures ($\otimes$, labeled a, b, c) and three switches ($\trianglelefteq$, labeled 1, 2, 3). It is possible to wire switches to fixtures in Figure Figure V.18(a) so that every switch has a line of sight to the fixture, but this is not possible in Figure V.18(b).

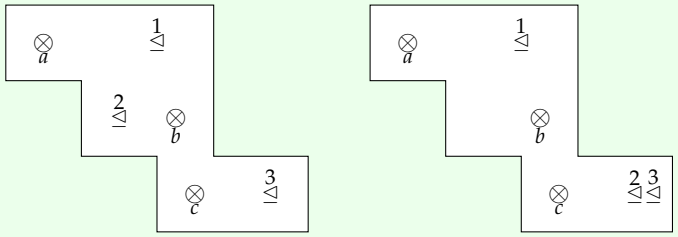


Figure V.18. The floor plan in (a) is ergonomic, because we can wire switches to fixtures in such a way that each fixture is visible from the switch that controls it. (This can be done by wiring switch 1 to a, switch 2 to b, and switch 3 to c.) The floor plan in (b) is not ergonomic, because no such wiring is possible..

Let's call a floor plan, together with n light fixture locations and n switch locations, ergonomic if it's possible to wire one switch to each fixture so that every fixture is visible from the switch that controls it. A floor plan will be represented by a set of m horizontal or vertical line segments in the plane (the walls), where the i^{th} wall has endpoints (x_i, y_i), (x_i', y_i'). Each of the n switches and each of the n fixtures is given by its coordinates in the plane. A fixture is visible from a switch if the line segment joining them does not cross any of the walls.

Give an algorithm to decide if a given floor plan is ergonomic. The running time should be polynomial in m and n. Provide the total runtime. You may assume that you have a subroutine with $\mathcal{O}(1)$ running time that takes two line segments as input and decides whether or not they cross in the plane.

Solution. This is a very natural *network flow* problem, and we can easily turn the model into a bipartite graph, as follows.

- Let $\mathcal{G} := (L \sqcup S, E)$, where L denotes the set of enumerated lights and S denotes the set of enumerated switches.
- $E \subset L \times S$ is the composed of directed edge so that there exists an edge from some $l \in L$ to $s \in S$ if and only if the light and the switch has their connection line not crossing the walls. (*It should be noted that the direction of the line does not matter, and the graph is naturally bipartite*.)

Set up the graph $\mathcal{G}$ as instructed above.

Attempt to find the perfect matching using the Ford-Fulkerson `algorithm`.
If there is a perfect matching, the perfect matching is the correct assignment. Otherwise, the room is not ergonomic.

Then, we *quickly* analyze the runtime of the algorithm.

- For the construction of the graph, since all validation is assumed to be $\mathcal{O}(1)$, we need to build the nodes in $\mathcal{O}(2n) = \mathcal{O}(n)$, and out of the n^2 potential edges, each validation takes $\mathcal{O}(n^2 \cdot m)$ so the total runtime is $\mathcal{O}(mn^2)$ for constructing the graph.
- For the network flow part, since the maximum capacity is at most n, and the cost of doing a BFS is $\mathcal{O}(2 + 2n + 2n + n^2) = \mathcal{O}(n^2)$, so the total runtime for attempting to find the perfect match is at most $\mathcal{O}(n^3)$.

Hence, the total runtime of the algorithm is $\mathcal{O}(n^3 + mn^2)$, which is polynomial with respect to m and n, as desired. ⌟

Problem V.7. Consider a set of mobile computing clients in a certain town who each need to be connected to one of several possible base stations. We'll suppose there are n clients, with the position of each client specified by its (x, y) coordinates in the plane. There are also k base stations; the position of each of these is specified by (x, y) coordinates as well. For each client, we wish to connect it to exactly one of the base stations. Our choice of connections is constrained in the following ways.

There is a range parameter r: a client can only be connected to a base station that is within distance r. There is also a load parameter L: no more than L clients can be connected to any single base station.

Your goal is to design a polynomial-time algorithm for the following problem, the algorithm needs to be polynomial in n and k. Given the positions of a set of clients and a set of base stations, as well as the range and load parameters, decide whether every client can be connected simultaneously to a base station, subject to the range and load conditions.

Solution. Again, this is a very straightforward application of network flow.
Let's construct a graph as follows:

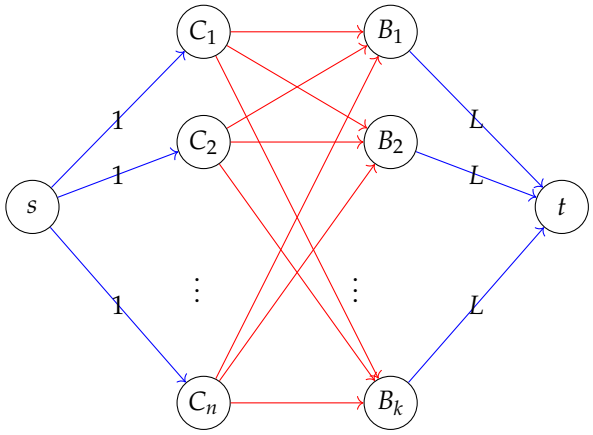


Figure V.19. Network flow problem from s to t with concrete example, and the red edges have capacity 1 if and only if the client and base stations are within distance r, while 0 otherwise.

Here, the algorithm would be as follows:

Construct the network flow problem as the diagram above.
Conduct Ford-Fulkerson `algorithm` on the network-flow problem.
If the total network flow is n, then there exists a suitable assignment, represented by the solutions to the network-flow problem. If the total network flow is less than n, there is no suitable solution.

Here, we can also briefly consider the runtime of our solution:

- For the initialization, there are $\mathcal{O}(n+k)$ nodes, and constant time for constructing at most $\mathcal{O}(n+k+nk)$ of them, so the total initialization time is $\mathcal{O}(nk)$.
- For the network flow process, we have at most n of capacity and the BFS is $\mathcal{O}(nk)$ so we have the total runtime is $\mathcal{O}(n^2k)$.

Therefore, the total runtime is $\mathcal{O}(n^2k)$. ⌟

Problem V.8. You are given a directed graph $G = (V, E)$ (picture a network of roads). A certain collection of nodes $X \subset V$ are designated as populated nodes, and a certain other collection $S \subset V$ are designated as safe nodes. (Assume that X and S are disjoint.) In case of an emergency, we want evacuation routes from the populated nodes to the safe nodes. A set of evacuation routes is defined as a set of paths in G so that (i) each node in X is the tail of one path, (ii) the last node on each path lies in S, and (iii) the paths do not share any edges. Such a set of paths gives a way for the occupants of the populated nodes to "escape" to S, without overly congesting any edge in G.

Given G, X, and S, show how to decide in polynomial time whether such a set of evacuation routes exists.

Solution. Again, we think about the question as a network flow problem.
Let's construct a graph as follows:

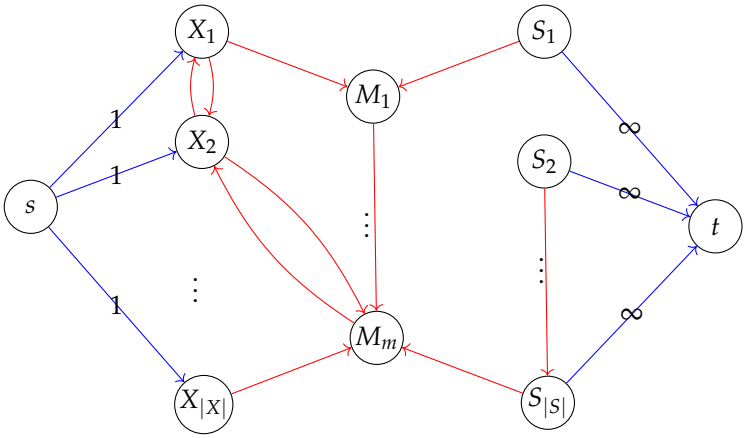


Figure V.20. Network flow problem from s to t with concrete example, and the red edges are the directed path in G with weight 1.

Here, the algorithm would be as follows:

Construct the network flow problem as the diagram above, we namely make all the edged in directed graph G into 1, put a weight of 1 from s to all X_i's and put a weight of ∞ from S_j's to t.
Conduct Ford-Fulkerson `algorithm` on the network-flow problem.
If the total network flow is $|X|$, then there exists a suitable assignment, represented by the solutions to the network-flow problem. If the total network flow is less than $|X|$, there is no suitable solution.

Here, we can also briefly consider the runtime of our solution:

- For the initialization, there are $\mathcal{O}(|V|)$ nodes, and constant time for constructing at most $\mathcal{O}(|E|)$ of them, so the total initialization time is $\mathcal{O}(|V| + |E|)$.
- For the network flow process, we have at most $\mathcal{O}(|X|) \subset \mathcal{O}(|V|)$ of capacity and the BFS is $\mathcal{O}(|V| + |E|)$ so we have the total runtime is $\mathcal{O}(|V|^2 + |V| \cdot |E|)$.

Therefore, the total runtime is $\mathcal{O}(|V|^2 + |V| \cdot |E|)$. ⌟

Problem V.9. Figure 1 shows a flow network on which an s-t flow has been computed. The capacity of each edge appears as a label next to the edge, and the numbers in boxes give the amount of flow sent on each edge. (Edges without boxed numbersspecifically, the four edges of capacity 3 have no flow being sent on them.)

(a) What is the value of this flow? Is this a maximum (s, t) flow in this graph?

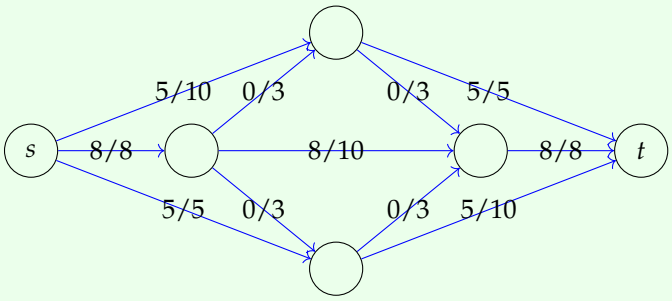


Figure V.21. Network flow graph.

Solution. The value of the flow is $5 + 8 + 5 = 18$. This is not the maximum flow since we can construct any path from s to t using not full forward and nonzero backward paths, that is:

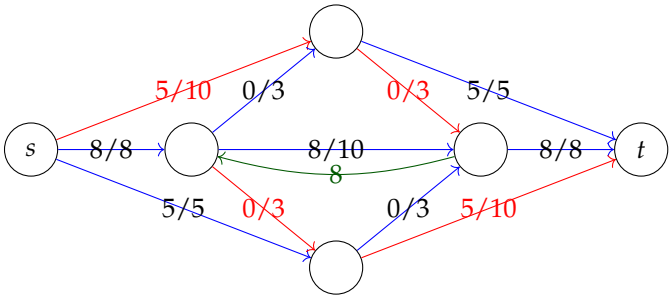


Figure V.22. An existing path on the Network flow graph.

So after using this path, we obtain that:

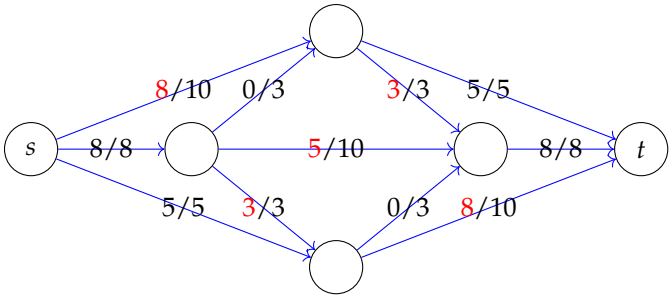


Figure V.23. Maximum Flow of Network flow graph.

⌟

(b) Find a minimum $s - t$ cut in the flow network pictured in the figure. What is the capacity of it?

Solution. We can easily mark the sets that are still reachable from s into *red* and others as blue.

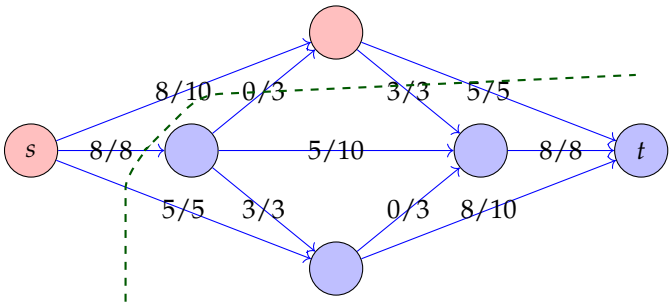


Figure V.24. Mininum $s - t$ cut in Network flow graph in green.

Hence, the capacity is:

$$5 + 8 + 3 + 5 = 21.$$

⌟

Problem V.10. In a standard s-t Maximum-Flow Problem, we assume edges have capacities, and there is no limit on how much flow is allowed to pass through a node. In this problem, we consider the variant of the Maximum-Flow and Minimum-Cut problems with node capacities.

Let $G = (V, E)$ be a directed graph, with source $s \in V$, sink $t \in V$, and nonnegative node capacities $\{c_v \geq 0\}$ for each $v \in V$. Given a flow f in this graph, the flow through a node v is defined as $f^{\text{in}}(v)$. We say that a flow is feasible if it satisfies the usual flow-conservation constraints and the node-capacity constraints: $f^{\text{in}}(v) \leq c_v$ for all nodes.

Give a polynomial-time algorithm to find an s-t maximum flow in such a node-capacitated network. Define an s-t cut for node-capacitated networks, and show that the analogue of the Max-Flow Min-Cut Theorem holds true.

Solution. For this question, we can trivially turn this into a standard s-t flow problem.
Here, we break each node $v \in V \setminus \{s, t\}$ into two nodes v^{in} and v^{out}, and we modify the edge in the following manner:

- each edge $(v_1, v_2) \in V \setminus \{s, t\} \times V \setminus \{s, t\} \cap E$ become the edge $(v_1^{\text{out}}, v_2^{\text{in}})$ keeping the same weight,
- each edge $(s, v) \in \{s\} \times V \setminus \{s, t\} \cap E$ become the edge (s, v^{in}) keeping the same weight,
- each edge $(v, t) \in V \setminus \{s, t\} \times \{t\} \cap E$ become the edge (v^{out}, t) keeping the same weight,
- for each $v \in V \setminus \{s, t\}$, we construct the edge $(v^{\text{in}}, v^{\text{out}})$ with weight c_v.

Here, for this new graph $\mathcal{G}$, we can just conduct the Ford-Fulkerson `algorithm` on the network-flow problem from s to t.
In terms of algorithm runtime, assume the maximum capacity is C, the runtime is $\mathcal{O}\big(C(2|V| + |V| + |E|)\big) = \mathcal{O}\big(C(|V| + |E|)\big)$, and is still polynomial.
Now, we correspondingly define a s-t cut as a partition of the nodes in $\mathcal{G}$, that is a partition of the incoming and outgoing nodes, in which the cut would be the sum of all the outgoing edges on the partition containing s.
Particularly, when projected to the original graph G, the cut through an $v_i^{\text{out}} \to v_j^{\text{in}}$ is just a cut between v_i and v_j, but a cut through an $v_i^{\text{in}} \to v_i^{\text{out}}$ is, instead, a cut through the node v_i.
When we compute the minimum cut it is now:

$$\text{min cut} := \sum_{e\text{: outgoing edges}} w_e + \sum_{v\text{: cut vertices}} c_v.$$

Here, we want to show that following adaption:

Theorem. The maximum of the flow is the same as the minimum of the cut.

Proof. Here, we just want to show the following equivalence:

1. There exists a partition (A, B) such that the value of the cut is the capacity of the flow f.
2. f is maximum flow.

3. There exists no augmenting path from f.

($1 \implies 2$:) Since the capacity of any flow is upper bounded by the partition, the existence follows through.
($2 \implies 3$:) Assume there exists an augmenting path added to f, then f adding that path is a larger flow.
($3 \implies 1$:) We can construct A as all reachable points from s, and $B = V \setminus A$. □

Hence, we have finished the process of this problem. ⌟