Math 182 - Algorithms for Winter 2025

The content is motivated by [1], lecture slides from Will Adkisson, which in turn was motivated by Patrick Lutz. [1]

Contents

1 Discussion 1

Big-Oh Notation/Asymptotic Order of Growth

Definition 1. ($\mathcal{O}$ Notation.) If $f : \mathbb{N} \to \mathbb{R}_{\geq 0}$, then $\mathcal{O}(f)$ is the set of functions $g : \mathbb{N} \to \mathbb{R}_{\geq 0}$ such that

$$\exists N, C > 0 \text{ such that } \forall n \geq N, \text{ we have } g(n) \leq Cf(n).$$

Intuitively, "eventually, f grows at least as fast as g, up to a constant factor."
In other words, "f serves as some sort of an upper bound for g".

Definition 2. (Ω Notation.) If $f : \mathbb{N} \to \mathbb{R}_{\geq 0}$, then $\Omega(f)$ is the set of functions $g : \mathbb{N} \to \mathbb{R}_{\geq 0}$ such that

$$\exists N, C > 0 \text{ such that } \forall n \geq N, \text{ we have } Cg(n) \geq f(n).$$

Intuitively, "eventually, f grows no slower than g, up to a constant factor."
In other words, "f serves as some sort of an lower bound for g".

Definition 3. (Θ Notation.) If $f : \mathbb{N} \to \mathbb{R}_{\geq 0}$, then $\Theta(f)$ is the set of functions $g : \mathbb{N} \to \mathbb{R}_{\geq 0}$ such that

$$\exists N, C > 0 \text{ such that } \forall n \geq N, \text{ we have } g(n) \leq Cf(n) \text{ and } Cg(n) \geq f(n).$$

Alternatively, $\Theta(f) := \mathcal{O}(f) \cap \Omega(f)$.

Intuitively, "eventually, f and g grow at the same rate, up to a constant factor."
In other words, "g is "grows approximately/asymptotically like" f".

Remarks:

- Rigorously speaking, we say $g \in \mathcal{O}(f)$ (or $\Omega(f)$ or $\Theta(f)$) since these are sets of functions, but most people just write it as $g = \mathcal{O}(f)$.
- In general, $f = \mathcal{O}(g)$ is often meant to be $f = \Theta(g)$. The idea is that we can always artificially inflate the upper bound as much as we want. Thus, the understanding is that the "growth rate" given is the "optimal" one, that is, $\Theta(g)$ (ie also serve as some sort of a lower bound.)

  Example: $n = \mathcal{O}(n)$, but also, $n = \mathcal{O}(n^2)$. However, $n = \Theta(n)$ but $n \neq \Theta(n^2)$.
- Note that it does not matter if we let $\mathbb{N}$ to start from 0 or 1 - the definitions convey the same meaning.

Next, we look at how to utilize the rigorous definition for an example below.

Example 4. Show that $5n + 20 = \Theta(n)$.

Suggested Solution: To do so, we would want to show that $5n + 20 = \mathcal{O}(n)$ and $5n + 20 = \Omega(n)$.

- $\underbrace{5n+20}_{g(n)} = \mathcal{O}(\underbrace{n}_{f(n)})$. Observe that for $n \geq 1$, we have

$$5n + 20 \leq 5n + 20n = 25n.$$

  Hence, pick $C = 25$ and $N = 1$. We thus have that for all $n \geq 1$, $\underbrace{5n+20}_{g} \leq 25 \underbrace{n}_{f}$ and we are done.
- Similarly, observe that

$$5n + 20 \geq 5n$$

  for any n (≥ 1). Hence, if we pick $C = 5$ and $N = 1$, we can show that $5n + 20 = \Omega(n)$, and conclude that $5n + 20 = \Theta(n)$. □

Note that multiple useful properties can be obtained with regard to $\mathcal{O}, \Omega$, and Θ. For a list of such properties, refer to Chapter 2.2 of the textbook or the lecture notes. Instead, we will look at asymptotic order of growths for some functions.

- (Polynomial = Θ(Term with the highest power).)

  If $f = a_k n^k + \cdots + a_1 n + a_0$ for some non-negative integer k, then $f = \Theta(n^k)$.

- (All logs are the same.) For any $h, k > 1$, we have

  $$\log_k(n) = \Theta(\log_h(n)).$$

  Remark: Hence, it is common to see $\Theta(\log(n))$ without specifying the base of the logarithm.

- (Logs make all polynomials look the same.) For any $k > 0$, we have

  $$\log_2(n^k) = k\log_2(n) = \Theta(\log_2(n)) \text{``} = \text{''} \Theta(\log(n)).$$

- (Logs are slower than any polynomial of positive degree.)

  For any $b > 1$, we have
  $$\log_b(n) = \mathcal{O}(n^k)$$
  for any $k > 0$.

Roughly speaking, we should expect ($k > 0$)

$$\log(n) \ll n < n\log(n) < n^2 < \cdots \ll 2^n < n^k 2^n.$$

Time Complexity and Examples of Running Times.

Standard assumptions:

- Array access takes $\mathcal{O}(1)$ time.

  (Example: The time it takes the i-th element of an array L does not depend the index i or the (size of the) array itself L.)

- Numerical Operations take $\mathcal{O}(1)$ time.

  (Example: The time it takes to do $a + b$ does not depend on a or b.)

Here, we also say that these operations run in constant time.
For the remaining parts of this supplement, we will go through examples on how to analyze the time complexity of algorithms. Here, we will only look at the worst-case time complexity - the maximum amount of time required for inputs of a given size.

Example 5. Consider the Slow_add algorithm for a positive integer $n > 0$ below.

```
Slow_add($n$):
1 $x \leftarrow 0$;
2 for $i = 1, 2, \cdots, n$ do
3 |   $x \leftarrow x + 1$;
4 end
5 return $x$
```

What is the (worst-case) time complexity for this algorithm?

Suggested Solution:

- Lines 1 and 5 only execute once each.
- Line 2 will be executed n times (ie assigning a value to i).
- Line 3 will be executed once per execution of line 2.

Hence, we have

$$1 + 1 + n + \sum_{i=1}^{n} 1 = 2n + 2 = \Theta(n).$$

Example 6. Consider the Slow_lowest2sum algorithm below. It takes in an array L of two or more real numbers, computes the distinct pairwise sum of these integers, and returns the lowest of all these pairwise sums.

Slow_lowest2sum(L):

```
1  current_lowest2sum ← L[1] + L[2];
2  n ← size(L);
3  for $i = 1, 2, \cdots, n$ do
4      for $j = 1, 2, \cdots, n$ and $j \neq i$ do
5          sum ← L[i] + L[j];
6          if sum < current_lowest2sum then
7              current_lowest2sum ← sum;
8          end
9      end
10 end
11 return current_lowest2sum
```

What is the (worst-case) time complexity for this algorithm?

Suggested Solution:

- Note that each of the operations (including accessing the element in the array, adding two integers, etc.) all run in constant time.
- Lines 3 and 4 suggest that lines 5 - 7 will run a total of $n(n-1)$ times.
- Note that lines 5 and 6 are compulsory, while line 7 depends on the truth value of line 6. In the worst-case scenario, even if we have to run line 7, these are all constant-time ($\mathcal{O}(1)$) operations. Hence, each instance of the loop runs in constant time.

All in all, we thus have that the time complexity is $n(n-1) = n^2 - n = \Theta(n^2)$, where n is the size of the array L.

Remark: Food for thought - Why can't we initialize current_lowest2sum to 0 in line 1?

Remark: We can speed this algorithm up by approximately a factor of two by observing that $L[i] + L[j] = L[j] + L[i]$ (the order in which we add the two terms is not important; ie $+$ is commutative). To do so, it suffices to change line 4 to $\boxed{\textbf{for } j > i}$.

Remark: In fact, by making use of sorting algorithms and the fact that they only require $\mathcal{O}(n \log(n))$ time to sort the list, it is possible to re-write the function such that it runs in $\mathcal{O}(n \log(n))$ time.

Example 7. Consider the Slow_lowestprod algorithm below. It takes in a non-empty array L of real numbers, picks a subsequence of any size of the array, computes the total product of the elements in the subsequence, and returns the lowest of all these products.

Slow_lowestprod(L):

```
1  current_lowestprod ← L[1];
2  for each non-empty subsequence $S$ of $L$ do
3      prod ← 1;
4      for $i = 1, \ldots,$ length(S) do
5          prod ← prod × S[i];
6      end
7      if prod < current_lowestprod then
8          current_lowestprod ← prod;
9      end
10 end
11 return current_lowestprod
```

What is the (worst-case) time complexity for this algorithm?

Suggested Solution:

- Let n be the size of the array L. Then, note that the total number of subsequences of L would be 2^n. Excluding the empty subsequence, the total number of subsequences would be $2^n - 1$.
  (For each element, we can either include it in the subsequence or not, thus giving us 2^n different subsequences to consider.) However, note that the time it takes to compute the product depends on the length of the subsequence S!
- In other words, note that the runtime for lines 4 - 5 depends on the length of the subsequence, while lines 7 - 8 run in constant time. Thus, the "bottleneck" comes from computing lines 4 - 5.

Hence, the total runtime is of the order

$$\sum_{\text{non-empty subsequences } S} \mathit{length}(S)$$

Using the $\mathcal{O}$ notation, this is equivalent to

$$\sum_{k=1}^{n} k\binom{n}{k}$$

where k represents the size of the subsequence, and $\binom{n}{k}$ represents the number of subsequences with size k. Observe that

$$\sum_{k=1}^{n} k\binom{n}{k} \le n\sum_{k=1}^{n}\binom{n}{k} = n(2^n - 1)$$

and hence

$$\sum_{k=1}^{n} k\binom{n}{k} = \boxed{\mathcal{O}(n2^n)}.$$

Here, we have used the fact that $\sum_{k=1}^{n}\binom{n}{k}$ corresponds to the number of non-empty subsequences (since we are summing over the number of all possible sizes of the subsequences) and hence is given by $2^n - 1$.

Remark: Intuitively, if we just want an upper bound for the time complexity, we can instead argue that the maximum runtime for lines 4 - 5 is at most the length of the array, n. Coupled with the fact that the total number of subsequences would be $2^n - 1$, we get $\mathcal{O}(n(2^n - 1)) = \mathcal{O}(n2^n)$.

Remark: In fact, one can rigorously prove that

$$\sum_{k=1}^{n} k\binom{n}{k} = n2^{n-1} = \Theta(n2^n).$$

In other words, instead of the time complexity being of $\mathcal{O}$, it is actually Θ!

Remark: In the previous version of the supplement, I've used the term "subarray" (ie contiguous subarrays - a subsequence of consecutive elements in an array) but was actually thinking about subsequences. Thanks to a student who pointed this out!

2 Discussion 2

Example 8. Let $f, g : \mathbb{N} \to \mathbb{N}$. Show that $\max\{f(n), g(n)\} \in \Theta(f(n) + g(n))$ for any such functions f and g.

Suggested Solutions: To show the required claim, we have to show that it is both $\mathcal{O}$ and Ω.

- $\underline{\max\{f(n), g(n)\} \in \Omega(f(n) + g(n)).}$
  This is equivalent to showing that there exists an N and C such that for all $n \geq N$, we have $C \max\{f(n), g(n)\} \geq f(n) + g(n)$.

  To do so, observe that for all $n \geq 1$, by the definition of max,
  $$\max\{f(n), g(n)\} \geq f(n)$$
  and
  $$\max\{f(n), g(n)\} \geq g(n).$$
  Adding the two equations together, we have
  $$2 \max\{f(n), g(n)\} \geq f(n) + g(n).$$
  Hence, by setting $C = 2$ and $N = 1$, we are done.

- $\underline{\max\{f(n), g(n)\} \in \mathcal{O}(f(n) + g(n)).}$
  This is equivalent to showing that there exists an N and C such that for all $n \geq N$, we have $\max\{f(n), g(n)\} \leq C(f(n) + g(n))$.

  To do so, we argue that all $n \geq 1$,
  $$\max\{f(n), g(n)\} \leq f(n) \leq f(n) + g(n)$$
  or
  $$\max\{f(n), g(n)\} \leq g(n) \leq f(n) + g(n).$$
  In both cases, we can add either $f(n)$ or $g(n)$ since the co-domain of the functions is the set of natural numbers, and hence $f(n), g(n) \geq 0$ for all $n \geq 1$. Henceforth, in both cases, we have
  $$\max\{f(n), g(n)\} \leq f(n) + g(n).$$
  Setting $C = 1$ and $N = 1$, we are done.

□

Example 9. We consider the task of sorting an array of integers, A, in increasing order. In this example, we consider the **selection sort** as described below.

```
Selection_Sort($A$):
1  $n \leftarrow$ length($A$);
2  for $i = 1, \cdots, n-1$ do
3      $min\_idx \leftarrow i$;
4      for $j = i+1, \cdots, n$ do
5          if $A[j] < A[min\_idx]$ then
6              $min\_idx \leftarrow j$;
7          end
8      end
9      Swap $A[j]$ and $A[i]$;
10 end
```

(i) What is the time complexity of Selection_Sort(A) as a function of n, the length of the array A?

(ii) Prove that this algorithm is indeed correct. That is, show that the array gets sorted in increasing order upon running the Selection_Sort() once on the array.

Suggested Solutions:

(i) Most of the operations are constant-time operations (ie lines 3 and 6 - 8). Hence, the bottleneck comes from the two nested for loops. This implies that the total number of this inner loop (lines 6 - 8) is being called would be given by

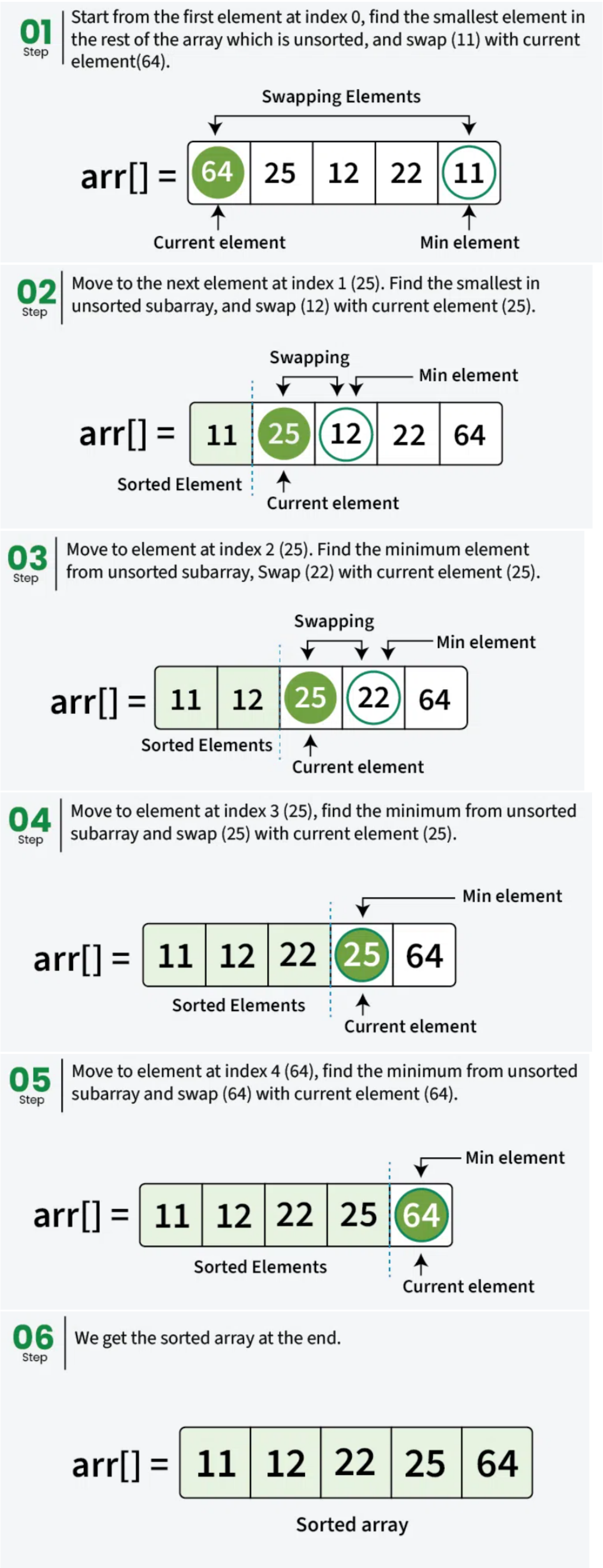
$$\sum_{i=1}^{n-1}\sum_{j=i+1}^{n} 1 = \sum_{i=1}^{n-1}(n-i) = \sum_{i=1}^{n-1} n - \sum_{i=1}^{n-1} i = n(n-1) - \frac{n(n-1)}{2} = \frac{n(n-1)}{2} = \Theta(n^2).$$

Remark: Rigorously speaking, the inner loop (lines 6 - 8) would take a longer time to run compared to calling line 3. Suppose that the inner loop takes C times as long to run as compared to running line 3. Hence, a slightly more rigorous argument would involve

$$\sum_{i=1}^{n-1}\left(1 + \sum_{j=i+1}^{n} C\right) = \sum_{i=1}^{n-1}(1 + C(n-i)) = \sum_{i=1}^{n-1}(1 + Cn) - C\sum_{i=1}^{n-1} i$$
$$= (1+Cn)(n-1) - C \cdot \frac{n(n-1)}{2} = \Theta(n^2).$$

Intuitively, since the constant C only affects the coefficients of the polynomial associated with the runtime as compared to the degree of the polynomial, we should expect the same time complexity. In other words, you don't have to worry about being "too correct/rigorous".

(ii) In general, we will attempt to do so by induction. Before we do that, let us start by running the algorithm above on an array of choice, say $[64, 25, 12, 22, 11]$. (The following diagrams are shamelessly lifted from https://www.geeksforgeeks.org/selection-sort-algorithm-2/!)



01 Step: Start from the first element at index 0, find the smallest element in the rest of the array which is unsorted, and swap (11) with current element(64).

02 Step: Move to the next element at index 1 (25). Find the smallest in unsorted subarray, and swap (12) with current element (25).

03 Step: Move to element at index 2 (25). Find the minimum element from unsorted subarray, Swap (22) with current element (25).

04 Step: Move to element at index 3 (25), find the minimum from unsorted subarray and swap (25) with current element (25).

05 Step: Move to element at index 4 (64), find the minimum from unsorted subarray and swap (64) with current element (64).

06 Step: We get the sorted array at the end.

Remark: In Python, arrays (lists) start with the index 0. In our analysis below, we will start the array with index 1.

As observed from the diagrams illustrating how the selection sort is at work, we see that at a given index i,

- Prior to activating the inner loops, the array $A[1 : i - 1]$ was sorted in increasing order. In fact, the array represents the first $(i - 1)$ integers in increasing order from the original array A (as compared to just sorting the first $i - 1$ terms in the array).
- Upon completing the inner loops, the array $A[1 : i]$ represents the first i integers from the original array in increasing order.

To prove the correctness of selection sort, we proceed by considering a fixed n and "inducting" on $i = 0, 1, \cdots, n$. Consider the following proposition:

$$P(i) := \text{“}A[1 : i] \text{ contains the first } i \text{ integers in increasing order from the original array } A \\ \text{upon completing the outer loop (lines 3 - 9) for index } i.\text{ ”}$$

Base Case: $P(0)$ is true. This is true since the array $A[1 : 0]$ is empty at the start of the algorithm (before $i = 1$), and is thus vacuously true.

Induction Case: Suppose that $P(i)$ is true for some $i = 0, \cdots, n - 2$. We would like to show that $P(i + 1)$ is true.

Since $P(i)$ is true, we have that $A[1 : i]$ contains the first i integers in increasing order from the original array. This also implies that all the integers in $A[i + 1 : n]$ are greater than or equals to any of the integers in $A[1 : i]$. We then run lines 3 - 9 for this value of i. Lines 4 to 8 searches for the smallest element to the right of the ith element (ie from $A[i + 1 : n]$), and swap that to the $(i + 1)$-th position. This implies that:

- By the fact that all the integers in $A[i + 1 : n]$ are greater than or equals to any of the integers in $A[1 : i]$, $A[i + 1]$ is greater than or equals to any of the integers in $A[1 : i]$.
- Furthermore, the inner loop implies that $A[i + 1] \leq$ any of the elements in $A[i + 2 :]$.

Combining the two facts together, this implies that $A[1 : i + 1]$ contains the first $(i + 1)$ integers from the original array in increasing order. Intuitively, by abusing notation[2], this is equivalent to showing that

$$A[1 : i] \leq A[i + 1] \leq A[i + 2 :]$$

which is then equivalent to

$$A[1 : i + 1] \leq A[i + 2 :]$$

and hence $P(i + 1)$ would be true.

By the principle of mathematical induction, we have that $P(0), P(1), \cdots P(n - 1)$ will be true. In other words, $P(n - 1)$ is true, implying that the array $A[1 : n - 1]$ contains the first $n(= \text{length}(A))$ integers in increasing order from the original array A. Automatically, this implies that $A[n] \geq A[i]$ for any $i = 1, \cdots, n - 1$. In other words, $A[n]$ is the largest element of the array. Henceforth, the array is sorted in increasing order. □

Remark: Note that in our induction argument, we could only assume that $i = 0, \cdots$ up to $n - 2$. This is because in order for us to show that $P(i + 1)$ is true, $i = n - 2$ corresponds to "completing the outer loop for index $i + 1 = n - 1$", which is the final index/iteration for the outer loop.

[2] Inequality on an array implying that is must be true element-wise.

Example 10. For each algorithm below, find a function $f(n)$ such that the algorithm runs in time $\Theta(f(n))$. You should explain the reasoning behind your answer but you do not need to give a formal proof.

```
Foo_1($n$):
if $n > 1$ then
    Foo_1($\lfloor n/2 \rfloor$);
    Foo_1($\lfloor n/2 \rfloor$);
end
```

```
Foo_2($n$):
if $n > 1$ then
    Foo_2($\lfloor n/3 \rfloor$);
    Foo_2($\lfloor n/3 \rfloor$);
    Foo_2($\lfloor n/3 \rfloor$);
end
```

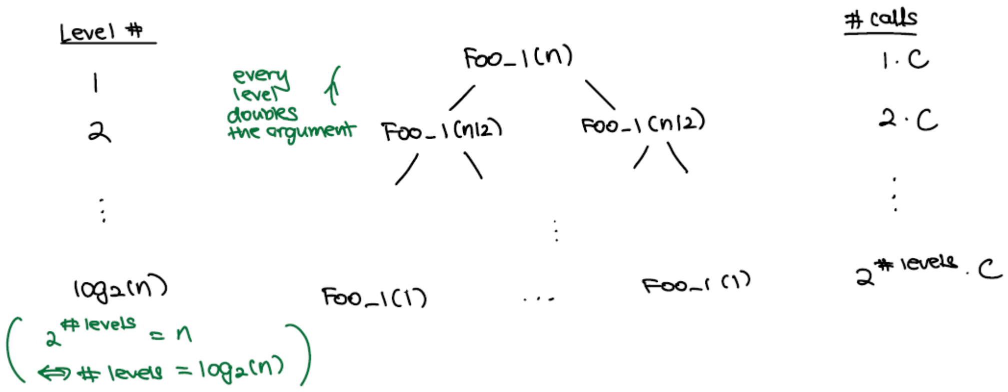
```
Foo_3($n$):
if $n > 1$ then
    Foo_3($\lfloor 2n/3 \rfloor$);
    Foo_3($\lfloor 2n/3 \rfloor$);
end
```

```
Foo_4($n$):
if $n > 1$ then
    Foo_4($\lceil 2n/3 \rceil$);
    Foo_4($\lceil 2n/3 \rceil$);
end
```

Before we begin to analyze these algorithms, recall the formula for a geometric series:

$$\sum_{k=0}^{N} r^k = \frac{r^{N+1} - 1}{r - 1}.$$

Foo_1(n).



$$
\begin{aligned}
\therefore \text{ Total runtime } &= C \cdot \sum_{k=0}^{\#\text{ levels}} 2^k \\
&= C \cdot \sum_{k=0}^{\log_2(n)} 2^k \qquad \text{geometric series } r=2,\ N=\log_2(n)+1 \\
&= C \cdot \frac{2^{\log_2(n)+1}-1}{2-1} \\
&= C \cdot \left(2^{\log_2(n)} \cdot 2 - 1\right) \\
&= C \cdot (2n-1) \\
&= \Theta(n).
\end{aligned}
$$

Foo_2(n).
We repeat a similar argument as for Foo_2(n). This time, we observe that the number of levels would be $\log_3(n)$, with each call branching (recursively calling) out (itself) three times. The total runtime would be

$$\Theta\left(\frac{3^{\log_3(n)+1}-1}{3-1}\right) = \Theta\left(\frac{3 \cdot n - 1}{2}\right) = \boxed{\Theta(n)}.$$

Foo_3(n).
Similarly, we repeat a similar argument as for Foo_1(n) and Foo_2(n). Observe that the number of levels, L, solves the following equation

$$\left(\frac{3}{2}\right)^L = n$$

and hence

$$L = \log_{3/2} n.$$

Each call of the function branches out twice. This implies that the total runtime would be given by

$$\Theta\left(\sum_{k=0}^{L} 2^k\right) = \Theta\left(\frac{2^{\log_{3/2}(n)+1}-1}{2-1}\right).$$

Since

$$\log_{3/2}(n) = \frac{\log_2(n)}{\log_2(3/2)} = \log_2\left(n^{\log_2(3/2)}\right),$$

we have

$$2^{\log_{3/2}(n)} = 2^{\log_2\left(n^{\log_2(3/2)}\right)} = n^{\log_2(3/2)}$$

and hence the total runtime would be

$$\Theta\left(\sum_{k=0}^{L} 2^k\right) = \Theta\left(n^{\log_2(3/2)}\right) \approx \boxed{\Theta(n^{0.58493})}.$$

Foo_4(n).

For Foo_4(n), observe that when we arrive at $n = 2$, we have that $2n/3 = 4/3$ and thus $\lceil 2n/3 \rceil = \lceil 4/3 \rceil = 2$. This implies that Foo_4(2) calls two copies of itself. This process then repeats indefinitely, raising a **runtime/recursion error**. This does not happen for Foo_3(n) since $\lfloor 4/3 \rfloor = 1$ rather than 2.

Remark: In Python, running Foo_4(n) for $n = 2$ will raise “RecursionError: maximum recursion depth exceeded”.

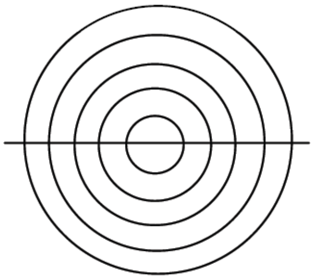
Remark: Later on in this class, we will learn how to derive these powers using what is known as the master theorem for these divide and conquer problems.

3 Discussion 3

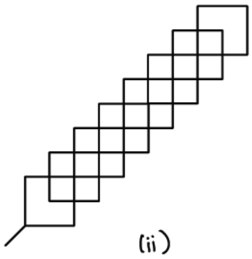
Graph Theory
Terminologies/Definition:

- $G = (V, E)$; $G :=$ graph, which is a collection of vertices V and a collection of edges E.
- Each edge is either an unordered pair $\{u, v\}$ (undirected edge; viewed as a set) or an ordered pair (u, v) (directed edge; viewed as a 2-tuple).
- The **degree** of a vertex v in a graph G is the number of edges connected to the vertex v within that graph.
- A path in a graph G is a sequence of vertices $v_1, \cdots, v_n$ such that each pair v_i and v_{i+1} is connected by an edge in G.
- A **cycle** is a path $v_1, \cdots, v_k$ if $v_1 = v_k$ and the intermediate vertices are distinct.
- An undirected graph is **connected** if for all vertices u and v, there is a path from u to v.
- A directed graph is **strongly connected** if for all vertices u and v, there is a path from u to v and a path from v to u.
- A **connected component** of a graph is a maximum set of vertices such that each vertex is connected to another via a path.
- For a (/an unweighted) graph, the **distance** from u to v is the minimum number of edges in a path from u to v.
- A graph G is a **tree** if it is connected and does not contain a tree.
- It is often useful to pick a node as the **root** of a graph.

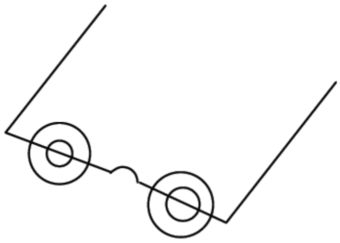
Example 11. (One-Line Puzzles.) Consider the following pictures below.



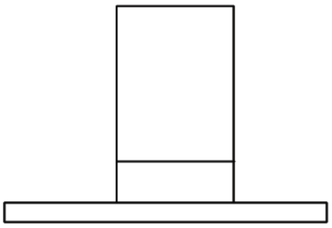
(i)



(ii)



(iii)

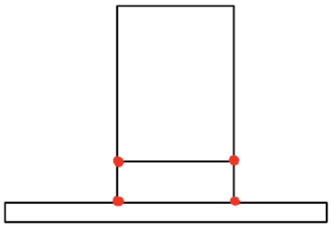


(iv)

One of them cannot be drawn with a pen on a piece of paper without lifting your pen from the pad or retracing any part of the line (though you are allowed to cross lines). Which would that be? Would you be able to come up with a general rule that could be used to determine this quickly?

Suggested Solution: (iv). The trick is to notice that to complete the drawing, you must "enter" and "exit" each intersection point, except at the starting and the ending point. You are allowed to "enter" and "exit" each intersection point as many times as you want.

Equivalently, we can model this using graphs, with vertices as the intersection points between lines, and edges as lines drawn between vertices. The question reduces to finding a path that visits every edge exactly once. It turns out that this is possible if and only if each vertex has an even degree (each "enter" and ”exiting” pair constitutes to adding the degree of the vertex by 2), except for either zero or exactly two vertices having odd degrees (corresponding to the start and end points).



For the problem above, each of the four red vertices has degree 3 (odd), and hence would not satisfy the requirements.

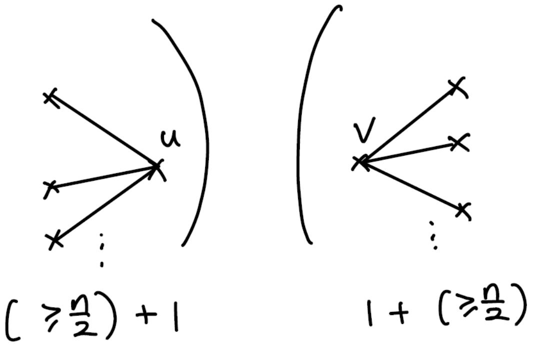
Remark: Look up Eulerian paths/trials/circuits. I've personally encountered this version of the problem from one of the puzzles from"Professor Layton"!

Example 12. (A necessary condition for connectedness - Question 6 of Homework 2.) Prove that if a graph G has n vertices, all of which have degree at least $n/2$, then G is connected.

Suggested Solution: Suppose for a contradiction that G has n vertices, all of which have degree at least $n/2$, and that G is **not connected**. This implies that there are at least two (disjoint) connected components of the graph. Pick vertices u and v, each from a different connected component. Since the degree of each vertex is at least $n/2$, there are at least $\frac{n}{2}+1$ vertices (including the vertex of interest u or v) in each connected component, giving us a total of at least

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

vertices. This contradicts the fact that the number of vertices in the graph G is only n. □



Remark: If $n=1$, then the graph is "trivially"/vacuously connected.

Graph Traversal Problem.

Problem: Given vertices u and v, is there a path from u to v? In fact, we can even do better - obtaining the shortest path from u to v.

Breadth-First Search (BFS).

Idea: One layer at a time, each new layer consists of vertices of distance 1 from the previous layer (if they are connected by an edge). This process also produces a tree, with root at the starting point. In other words, the tree gives us information on all the vertices for which there is a path from the root (say u) to. If we set the initial layer to be layer # 0 (with just u), we could also read off the distance of u from a vertex of interest, v, by looking up which layer v is in.

Runtime Complexity: $\Theta(|V| + |E|)$, where $|V|$ and $|E|$ refers to the number of vertices and edges respectively.

Implementation: Using a queue. See Example 13 below. Recall from the notes from Lecture 3 that by implementing a queue and stack as a linked list, we get $\Theta(1)$ time complexity for initializing, pushing, and popping (removing the leftmost element for a queue and the rightmost element for a stack).

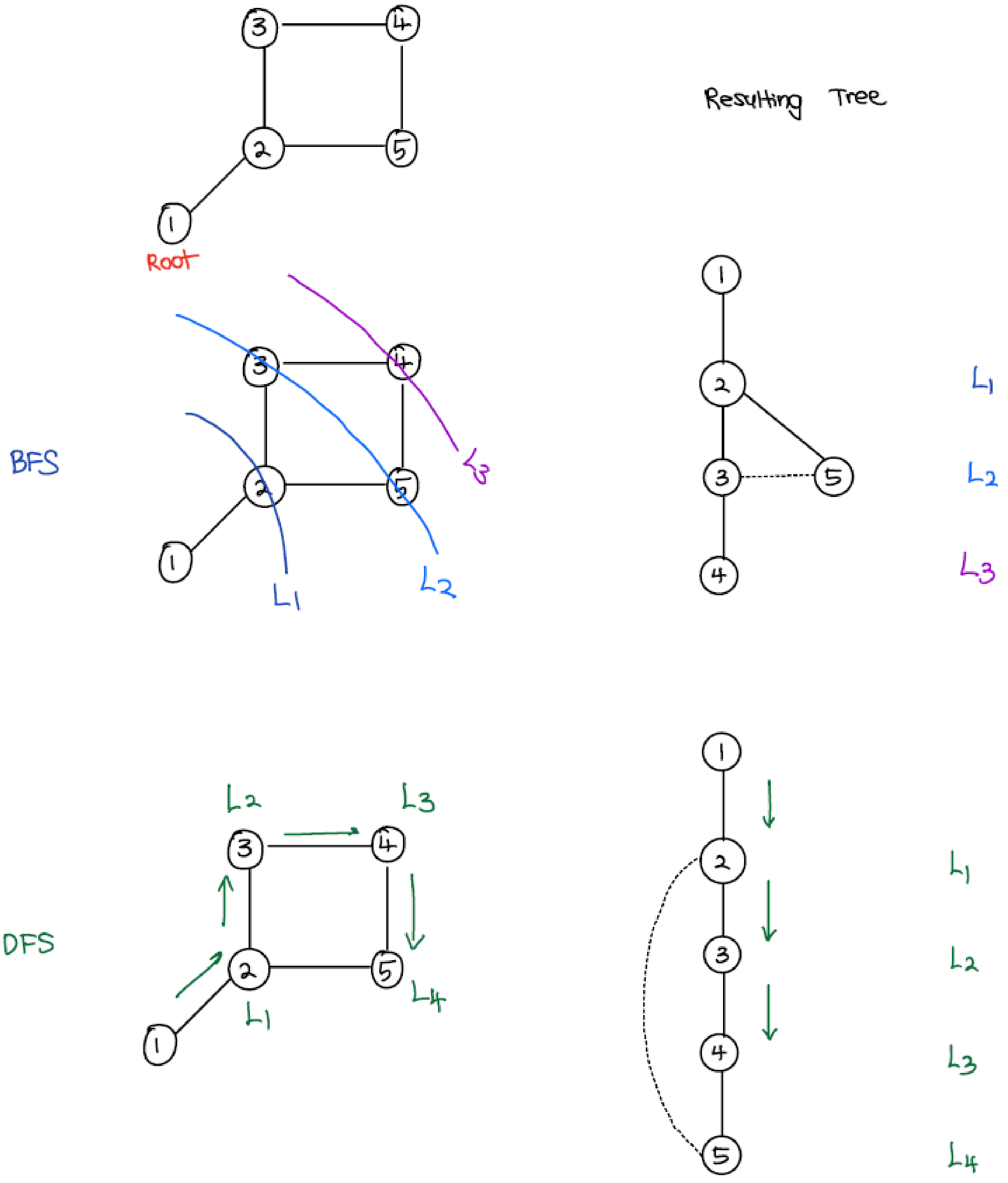
Depth-First Search (DFS).

Idea: Rather than slowly exploring outwards, one attempts to explore as “deep-as-possible” until we hit a dead end. We then backtrack until we hit a node with an unexplored neighbor and repeat the process. Refer to the lecture notes for a pseudo-codes style implementation of this.

Runtime Complexity: $\Theta(|V| + |E|)$, where $|V|$ and $|E|$ refers to the number of vertices and edges respectively.

Implementation: Using recursion. Alternatively, this could be done iteratively using a stack. See Example 13 below.

Example:



Example 13. (Number of Islands.)

Given an $m \times n$ 2D binary grid G that represents a map of '1's (land) and '0's (water), design an algorithm to compute the number of islands. Here, an island is surrounded by water and is formed by connecting adjacent lands horizontally or vertically. You may assume that all four edges of the grid are all surrounded by water.

Assumptions:

- The grid is non-empty ($m, n \geq 1$).

Examples:

$[$
$[1, 1, 1, 1, 0]$,
$[1, 1, 0, 1, 0]$,
$[1, 1, 0, 0, 0]$,
$[0, 0, 0, 0, 0]$,
$]$
yields 1 island.

$[$
$[1, 1, 0, 0, 0]$,
$[1, 1, 0, 0, 0]$,
$[0, 0, 1, 0, 0]$,
$[0, 0, 0, 1, 0]$,
$]$
yields 3 islands.

Source: https://leetcode.com/problems/number-of-islands/description/.
Recall that when designing an algorithm, you must include the following:

1. A short (1-2 sentences) sketch of the main idea of this algorithm.
2. The algorithm itself, written in pseudo-code.
3. The runtime of the algorithm.
4. A proof that the algorithm is correct.
5. A proof that the runtime is correct.

Suggested Solution:

1. For each position that we did not visit, we run a BFS, recording all the positions traversed by the BFS algorithm (in a set). The output will be the number of times we have executed the BFS algorithm.

2. Pseudocode: For an array G,

   num_Islands(G):

```
1  num_rows, num_cols ← length(G), length(G[1]);
2  visited ← set();
3  island ← $0$;
4  directions ← $[[1,0],[-1,0],[0,1],[0,-1]]$;
5  Function BFS($r$,$c$):
6      q ← queue();
7      Add $(r,c)$ to visited;
8      Push $(r,c)$ to q;
9      while q is non-empty do
10         row, col ← q.popleft();
11         for $dr, dc$ in directions do
12             r,c ← row + dr, col + dc;
13             if $1 \le r \le$ num_rows and $1 \le c \le$ num_cols then
14                 if G[r][c] == 1 and $(r,c)$ not in visited then
15                     Push $(r,c)$ to q;
16                     Add $(r,c)$ to (the set) visited;
17                 end
18             end
19         end
20     end
21 for $r = 1, 2, \cdots,$ num_rows do
22     for $c = 1, 2, \cdots,$ num_cols do
23         if G[r][c] == 1 and (r,c) not in visited then
24             BFS($r$,$c$);
25             island ← island $+ 1$;
26         end
27     end
28 end
29 return island
```

3. Runtime: $\Theta(n \cdot m)$.

4. We first map the problem into a graph traversal problem. Consider a graph with nm vertices corresponding to the positions on the array, and an undirected edge connecting two vertices if they are both labeled as '1's and lies directly horizontal/vertical of each other. The equivalent graph question would be to find the number of connected components to a graph. Hence, the number of trees (connected components) produced by a global BFS algorithm would corresponds to the number of islands. In our code above, this corresponds to the number of "local" BFS defined from lines 5 to 20 as each run of the "local" BFS corresponds to constructing a tree consisting of all vertices with a path from the root vertex, and lines 24 and 25 implies that every instance of running the "local" BFS corresponds to adding the number of islands by 1.

   Remark: Here, we are assuming that the BFS algorithm is correct. Thus, we don't have to prove again that BFS algorithm is correct, and instead use the fact that it is correct to prove that the function we are defining gives the desired outcome.

   Remark: Here, we notice that the proof technique is different as from Discussion 2. This is because there is no obvious loop-invariant (ie $P(i)$ for our induction proof). Instead, we are arguing that every instance that we add an island, we are doing it in the right way (BFS) and we satisfy the requirements of the question.

5. The runtime complexity of BFS is $\Theta(|V| + |E|)$ of the graph as described above. Observe that $|V| = nm$, while $0 \le |E| \le 4nm$ (the upper bound is obtained by assuming that each vertex is connected to all four of its neighboring vertices horizontally/vertically). Hence,

$$nm \le |V| + |E| \le 5nm,$$

implying that

$$\Theta(|V| + |E|) = \Theta(nm).$$

Remark: Note that the runtime complexity result above is derived from the fact that all operations involving queue (initialize, push, and pop) are $\Theta(1)$.

Remark: Suppose that we do not use the fact that the entire algorithm is just a BFS search over the entire graph (for connected components). We can analyze the time complexity directly as follows.

- Base Cost: Running the double for loop from lines 21 to 23. $\Theta(num\_rows \cdot num\_cols) = \Theta(nm)$. Here, we assume that line 23 is a constant time operation (ie checking if (r, c) is in $visited$).
- Additional Cost: Running "local" BFS(r,c). On each connected components (island), each vertex is visited exactly once. This is because pushing (r, c) to the queue is equivalent to adding it to the set of $visited$. Furthermore, for each element (r, c) in the queue, lines 10 pops it, and runs a search in all four cardinal directions (lines 11 - 19). Observe that lines 11 - 19 is of a constant time operation, and this runs for the number of times equivalent to the number of new coordinates added to $visited$.
- Hence, over all possible runs of the "local" BFS (over all the islands), the number of times we run lines 10 to 19 would be equivalent to the number of vertices labeled '1'. (Recall that each of these runs are $\Theta(1)$.) Hence, the additional runtime is equivalent to $\Theta(\text{\# of '1's})$. The lower bound for this is 0, and the upper bound for this is nm (the entire array is filled with '1's).
- Total runtime complexity:
$$\Theta(nm) + \underbrace{\Theta(\text{\# of '1's})}_{1 \lesssim \cdot \lesssim n \cdot m} = \Theta(nm).$$

Remark: Instead of using a set $visited$ as initialize in line 2, there is another popular way of doing this by "flooding islands" that you have found. For example, once you have found an island, we set all the '1's on that island to '0's.

Remark: If we instead want to consider using DFS, this can be done using the same algorithm but only changing the parts labelled in red for two of the lines in the DFS algorithm.

```
num_Islands($G$):
num_rows, num_cols ← length(G), length(G[1]);
visited ← set();
island ← 0;
directions ← $[[1,0],[-1,0],[0,1],[0,-1]]$;
Function DFS($r$,$c$):
    q ← *stack*();
    Add $(r,c)$ to visited;
    Push $(r,c)$ to q;
    while *q is non-empty* do
        row, col ← q.pop();
        for $dr, dc$ *in directions* do
            r,c ← row + dr, col + dc;
            if $1 \leq r \leq$ *num_rows* and $1 \leq c \leq$ *num_cols* then
                if *G[r][c] == 1* and $(r,c)$ *not in visited* then
                    Push $(r,c)$ to q;
                    Add $(r,c)$ to (the set) visited;
                end
            end
        end
    end
for $r = 1, 2, \cdots,$ *num_rows* do
    for $c = 1, 2, \cdots,$ *num_cols* do
        if *G[r][c] == 1* and *(r,c) not in visited* then
            DFS($r$,$c$);
            island ← island + 1;
        end
    end
end
return island
```

Here, pop refers to removing the element ("last element") that was just added, consistent with the intuition behind the DFS algorithm.

4 Discussion 4

Greedy Algorithm.

Goal: Find an optimal solution. “Optimality” is usually measured by some variable (maximizing/minimizing distance, time, etc...).

Optimal solution is too expensive (usually requires checking over all permutation).

Greedy Algorithm: “Approximate” it by being greedy - attempt to find a “locally optimal” solution. This is usually done by optimizing over the “next” step rather than optimizing over all possible permutation.

Implication: This is usually faster.

Key concept: Proof of Correctness. There are usually two strategies to this:

1. “Staying ahead” arguments. We would want to argue that the greedy algorithm is always no worse (at least as good as) than the optimal algorithm. See Example 14.
2. “Exchange” arguments. For these arguments, the idea is to conclude that given the optimal solution, we are able to “modify” (usually in the form of “exchanging”, etc) the optimal solution a finite number of times to arrive at the greedy algorithm such that each modification will always no worse (at least as good as) the algorithm prior to modification. See Example 15.

Example 14. (Can Place Flowers.)
You have a long flowerbed in which some of the plots are planted, and some are not. However, flowers cannot be planted in adjacent plots.

Given an integer array `flowerbed` containing `0`'s and `1`'s, where `0` means empty and `1` means not empty, and an integer `n`, return `True` if `n` new flowers can be planted in the `flowerbed` without violating the no-adjacent flowers rule and `False` otherwise.

Examples:

Input: `flowerbed = [1,0,0,0,1], n = 1`
Output: `True`

Input: `flowerbed = [1,0,0,0,1], n = 2`
Output: `False`

Source: https://leetcode.com/problems/can-place-flowers/description/.

Note that the idea behind the algorithm is simple. We go through the flowerbed. If we encounter an empty plot `0`, we check to see if the adjacent plots are empty (ie `0`). If they both are, we plant a flower there (convert that to `1`) and keep track of the number of flowers we have planted. If we planted more than the flower that are required, we return `True`; else we return `False`.

Here, one has to realize that this algorithm is **greedy**. In other words, we plant a flower at the first possible instance as we run through the `flowerbed` starting from the left (index 0). There could be other valid ways to plant the `flowerbed`. For example, if

`flowerbed = [1,0,0,0,0,1]`,

then both `flowerbed = [1,0,1,0,0,1]` and `flowerbed = [1,0,0,1,0,1]` are valid ways to plant a single flower onto the `flowerbed`. Hence, as we attempt to prove that our algorithm (a.k.a, a **greedy** algorithm) is optimal, we would like to show that any other possible ways to plant the flowers onto the `flowerbed` would give the right answer too.

1. See above for the sketch of the main idea of this algorithm.

2. def can_place_flowers(flowerbed,n):

```
m ← length(flowerbed);
num_flowers ← 0;
for i = 1,2,··· ,m do
    if i == 1 then
        if flowerbed[i] == 0 and flowerbed[i+1] == 0 then
            flowerbed[i] = 1;
            num_flowers += 1;
        end
    end
    else if i == m then
        if flowerbed[i] == 0 and flowerbed[i-1] == 0 then
            flowerbed[i] = 1;
            num_flowers += 1;
        end
    end
    else
        if flowerbed[i] == 0 and flowerbed[i-1] == 0 and flowerbed[i+1] == 0 then
            flowerbed[i] = 1;
            num_flowers += 1;
        end
    end
end
return num_flowers >= n
```

Remark: One can optimize this code for speed and all but this is sufficient to get $\Theta(m)$ (ie handling all the boundary cases together and/or breaking out of the while loop anytime we have num_flowers $\geq$ n).

3. Runtime: $\Theta(m)$, where m is the size of the array flowerbed.

4. Proof of Correctness: Here, we use the "stay-ahead" argument. Let x_k be the index of the k-th flower placed, and the primed variables be that of the optimal solution (which corresponds to maximizing num_flower). In other words, if we can show that:

Claim 1: $x_k \leq x'_k$ for all k and

Claim 2: num_flower $\geq$ num_flowers',

then the second property would imply num_flowers == num_flowers$'$. In other words, our algorithm will output the same Boolean value as the optimal algorithm.

Proof of Claim 1: We proceed by induction on k.

For the base case, since our algorithm plants a flower as soon as it gets the chance to, any other algorithm would plant the first flower either at the same index or some later index.

For the induction step, we assume that $x_k \leq x'_k$ for some k. To show that $x_{k+1} \leq x'_{k+1}$, we first observe that we have 0,1,0 with the middle 1 at position x_k. Our algorithm then attempts to find the next set of 0,0,0 that comes up immediately after our planted 1. In other words, we have $x_{k+1} - x_k \leq x'_{k+1} - x'_k$. This thus implies that $x_{k+1} \leq x'_{k+1} + (x_k - x'_k) \leq x'_{k+1}$ since by the induction hypothesis, $x_k \leq x'_k$ implies that $x_k - x'_k \leq 0$.

Proof of Claim 2: This follows from Claim 1 and substituting k = num_flowers', which implies that $x_{\texttt{num\_flowers}} \leq x_{\texttt{num\_flowers}'}$, implying that there are more room for possibly more flowers to be planted by our algorithm.

Remark: To be more rigorous, note that if `num_flower'` is zero, then we must have `num_flower` to be zero too (since it is the maximum value of that), and our algorithm is automatically correct. Hence, we assume that `num_flower'` ≥ 1. This then implies that there are three consecutive `0` (ie. $\cdots$ `,0,0,0,` $\cdots$) in which our algorithm would also pick up.

Remark: To be even more rigorous, we are assuming that the `1`'s are not planted at the boundary (index 1 or m). However, we can either argue separately for that case using an analogous argument, or pad the `flowerbed` with `0` on the left and right and argue without loss of generality.

5. The runtime complexity follows from the fact that lines 4 - 21 run in constant time for a total of m iterations, as seen from the for loop in line 3.

Remark: The exact same argument holds for correctness if we instead ask for the maximum number of flowers that can be planted onto the `flowerbed`.

Example 15. (Assign Cookies.)

Assume you are an awesome parent and want to give your children some cookies. However, you should give each child at most one cookie.

Each child `i` has a greed factor `g[i]`, which is the minimum size of a cookie that the child will be content with; and each cookie `j` has a size `s[j]`. If `s[j] >= g[i]`, we can assign the cookie `j` to the child `i`, and the child `i` will be content. Your goal is to maximize the number of your content children and output the maximum number. Here, `g` and `s` are given as integer arrays with positive entries.

Examples:

Input: `g = [1,2,3], s = [1,1]`
Output: 1
Explanation: Since the greed factors of the children are 1, 2, and 3, and we only have cookies of size 1, we can only satisfy the child with greed factor 1.

Input: `g = [1,2], s = [1,2,3]`
Output: 2
Explanation: Here, we have 2 children and 3 cookies. We can satisfy the child with greed 1 using the cookie of size 1, and the child with greed 2 using the cookie of size 2. Hence, both children would be satisfied.

A student attempted to code this out using the following pseudocode:

Source: https://leetcode.com/problems/assign-cookies/description/.

Suggested Solution:

1. We first sort the arrays `g` and `s`. We then proceed with the two pointer method, assigning the biggest cookie to the child with the largest greed factor if it satisfies the child. If the cookie is large enough, move on to the smaller cookie and the child with a lower greed. Else, move on to the child with a lower greed (only).

2. `def assign_cookies(g,s):`

```
Sort s and g is increasing order;
maxNum ← 0;
cookie_idx ← length(s);
child_idx ← length(g);
while cookie_idx ≥ 1 and child_idx ≥ 1 do
    if s[cookie_idx] ≥ g[child_idx] then
        maxNum += 1;
        cookie_idx -= 1;
        child_idx -= 1;
    end
    else
        child_idx -= 1;
    end
end
return maxNum
```

3. Runtime: $\Theta(n\log(n) + m\log(m))$, where n and m are the length of arrays g and s respectively.

4. Proof of Correctness: In this example, we will argue by using an "exchange" argument. To see how this works, we consider the following `g` and `s`:

   `g = [1,2,3,4,4,5]` and `s = [1,2,3,3,4]`.

   Here, we consider that the following assignments (given as `(g,s)` pairs):

- `(1,1), (2,2), (3,3), (4,4)`. Observe that no other assignments are possible. This is an **optimal** assignment but it **cannot be an output** of our algorithm.
- `(1,2), (2,3), (3,3), (4,4)`. Observe that no other assignments are possible. This is an **optimal** assignment (though we would not know beforehand), and **is an output** of our algorithm.
- `(1,4)`, `(2,3)` and `(3,3)` only. This implies that we are left with `g = [4,4,5]` and `s = [1,2]`, implying that no further assignments are possible. This output is **not optimal**, but it **cannot be an output** of our algorithm too.

Hence, the idea of the proof is to argue that we can reassign the cookies from the optimal algorithm to the output of our greedy algorithm but this reassign cannot decrease the number of satisfied children.

The strategy here is to observe that the output of our (greedy) algorithm corresponds to the subarray `[1,2,3,3,4]` (ie the subarray starting from the right; this is because the right pointer on the cookie size array will only move one step to the left if an assignment is successful). If we have an assignment like in the first bullet point given by `[1,2,3,3,4]`, we want to argue that starting from an optimal solution,

"Exchange": We can always assign a bigger cookie to the same child whenever a bigger cookie is available (defined as an "exchange").

Claim 1: Each "exchange" will not decrease `maxNum` (in fact, that should be kept constant).

Claim 2: There are only a finite number of "exchanges" before we arrive at our (greedy) algorithm.

Rigorously,

"Exchange": If for some $s_i[k]$ is assigned such that $s_i[k+1]$ is not assigned, we assign $s_i[k+1]$ to the child instead. Note that this is compatible since if $s_i[k]$ was, then $g_i[k] \leq s_i[k] \leq s_{i+1}[k]$, and hence $g_i[k] \leq s_{i+1}[k]$.

Remark: Note that if no such "exchange" exists from the start, it means that it must be the output of our algorithm.

Claim 1: Each "exchange" will not decrease `maxNum` (in fact, that should be kept constant).

Proof (Argument) of Claim 1: From our definition of an "exchange" above, the number of satsified children remains unchanged for each instance of an "exchange" conducted.

Claim 2: There are only a finite number of "exchanges" before we arrive at our (greedy) algorithm.

Proof (Argument) of Claim 2: This should be clear from our construct. If a rigorous proof is necessary, we can go through each assigned cookie $s_i[k]$ starting from the right-most (largest) assigned cookie, conduct as many "exchanges" as possible, and end up with $s_i[m]$. Repeat this argument for the second largest assigned cookie, and so on. The output will be exactly the same as the output from our (greedy) algorithm, and this is done in finite number of "exchanges".

Remark: This kind of sounds like a "bubble" sort, bubbling the "largest" cookie assignment to the right and so on.

Remark: Note that this only deals with the assumption that we are looking at assigning cookies to the same children. To account for the fact that the algorithm assigns cookies to different children too, we might have to repeat an exchange argument for the array `g` too based on a similar argument.

5. Observe that it takes $\Theta(n\log(n))$ and $\Theta(m\log(m))$ time to sort the arrays `s` and `g` respectively. The while loop from lines 5 - 14 only runs at most $\max\{n,m\} = \Theta(n+m)$ (see Discussion 3 Example 1), with the operations inside of each iteration running in constant time. Hence, the time complexity is given by $\Theta(n\log(n) + m\log(m)) + \Theta(n+m) = \Theta(n\log(n) + m\log(m))$.

Example 16. (Assign Cookies II). Refer to the previous example for the task. If we instead consider the following algorithm:

```
def assign_cookies_mod(g,s):
Sort s and g is increasing order;
maxNum ← 0;
cookie_idx ← 1;
child_idx ← 1;
while cookie_idx ≤ length(s) and child_idx ≤ length(g) do
    if s[cookie_idx] ≥ g[child_idx] then
        maxNum += 1;
        cookie_idx += 1;
        child_idx += 1;
    end
    else
        child_idx += 1;
    end
end
return maxNum
```

Is this algorithm correct? If it is, prove it. Else, give arrays g and s such that the algorithm fails to give an optimal solution.

Suggested Solution: It in fact is too. To prove this, we have to argue that

"Exchange": We can always assign a smaller cookie to the same child whenever a smaller cookie is available and is compatible (i.e the child is still satisfied, defined as an "exchange")

Claim 1: Each "exchange" will not decrease maxNum (in fact, that should be kept constant).

Claim 2: There are only a finite number of "exchanges" before we arrive at our (greedy) algorithm.

The written argument will be similar, but understanding that the "lower bound" for the indices of our assigned cookies is the output from our greedy algorithm. In other words, if the optimal assignment is given by [a,b,c,d,e] and the output of our algorithm is [a,b,c,d,e], there is a finite number of "exchanges" to obtain the output from our algorithm. Note that in this example, d with e is a valid exchange since it is an output assigned by our algorithm, implying that it must also be compatible.

Consider instead the following scenario - the optimal assignment is given by [a,b,c,d,e] and the output of our algorithm is [a,b,c,d,e]. This will not happen since optimality implies that the cookie b is enough to satisfy the child with the second smallest greed (compatible). This implies that our algorithm would not have terminated like that, and instead attempts to assign b instead of c to that child.

Remark: Observe that it is "more difficult" to argue that it is correct. Hence, in some other algorithm design problems, it might be possible that the algorithm ends up being wrong and you will be able to see this if you try to prove it but ended up getting stuck!

5 Discussion 5

Dijkstra's Algorithm

Goal: Given a weighted graph with non-negative/positive weights, we want to find the shortest path from a given node `s` to any other (terminal) nodes `t`. To do so, we make use of a priority queue (ie a complete binary tree with the heap property; a heap). Recall the time complexity for the following operations:

- `Insert(a,p)` - insert a node with value `a` and priority `p` in the right position (via heapification).
  Time: $\mathcal{O}(\log(n))$.
- `DeleteMin()` - swap the root with the rightmost node on the bottom level, remove it, and heapify accordingly.
  Time: $\mathcal{O}(\log(n))$.
- `ChangePriority(a,p)` - change the priority of a node, and heapify accordingly.
  Time: $\mathcal{O}(\log(n))$.
- `Peek()` - look at the minimum element (root).
  Time: $\mathcal{O}(1)$.
- Initiating a priority queue.
  Time: $\mathcal{O}(n)$.

Here, $n =$ number of nodes in the priority queue, and $\log(n)$ here corresponds to the number of levels in the binary tree, which also corresponds to the maximum number of "swaps" required to heapify the priority queue if there is a single change made to it.

The psuedo-code for this algorithm is given in class, which we will repeat here just for convenience.

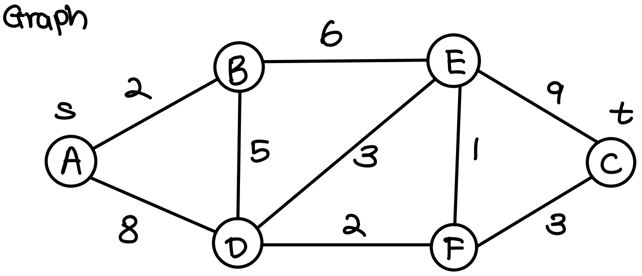
```
Dijkstra(G = (V,E),w,s,t):
    %% w = weights along each edge e in E
    d = new length |V| array
    for v in V:
        d[v] = infty
    d[s] = 0
    P = new priority queue containing each v in V with priority d[v]
    while P is nonempty:
        v = DeleteMin(P)
        for u adjacent to v:
            if d[v] + w[v,u] < d[u]:
                d[u] = d[v] + w[v,u]
                ChangePriority(u,d[u])
    return d[t]
```

Runtime complexity of Dijkstra: $\mathcal{O}\left((|V| + |E|)\log|V|\right)$.

- $\mathcal{O}(|V|)$ to initialize the priority queue.
- $\mathcal{O}(|V|)$ calls of DeleteMin, $\mathcal{O}(|E|)$ calls of ChangePriority (since each edge is only traversed once on each "side" of it), and each takes a maximum time of $\mathcal{O}(\log|V|)$.
- $\mathcal{O}(|V|) + \mathcal{O}((|V| + |E|)\log|V|) = \mathcal{O}((|V| + |E|)\log|V|)$.

An example of how to code unfolds can be seen below.

Target Graph

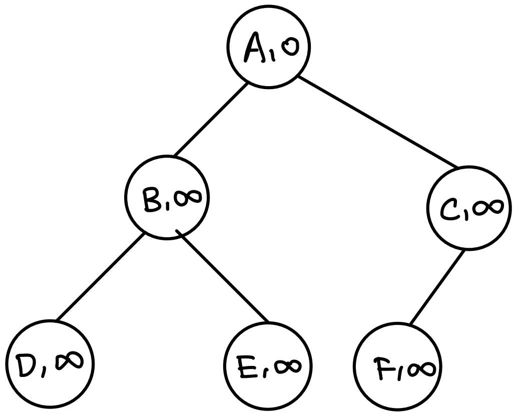


```
Dijkstra(G,w,s,t):
  d = new length |V| array
  for v in V:
    d[v] = infty
  d[s] = 0
  P = new priority queue containing each v in V with priority d[v]
  while P is nonempty:
    v = DeleteMin(P)
    for u adjacent to v:
      if d[v]+w[v,u]<d[u]:
        d[u]=d[v]+w[v,u]
        ChangePriority(u, d[u])
  return d[t]
```

node	d
A	0
B	∞
C	∞
D	∞
E	∞

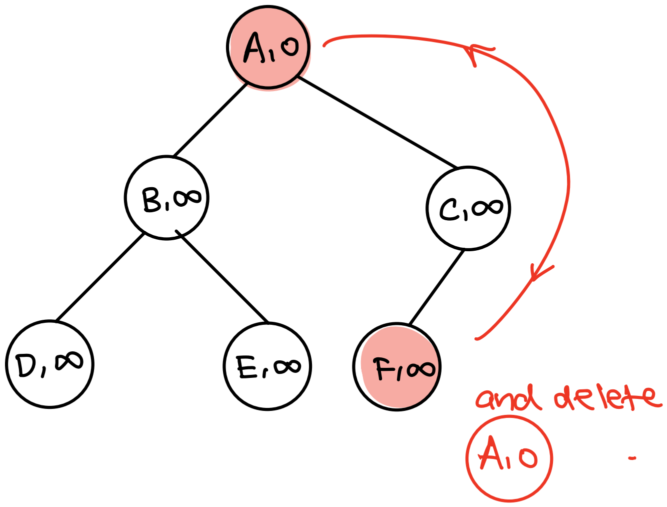
⇩

```
Dijkstra(G,w,s,t):
  d = new length |V| array
  for v in V:
    d[v] = infty
  d[s] = 0
  P = new priority queue containing each v in V with priority d[v]
  while P is nonempty:
    v = DeleteMin(P)
    for u adjacent to v:
      if d[v]+w[v,u]<d[u]:
        d[u]=d[v]+w[v,u]
        ChangePriority(u, d[u])
  return d[t]
```

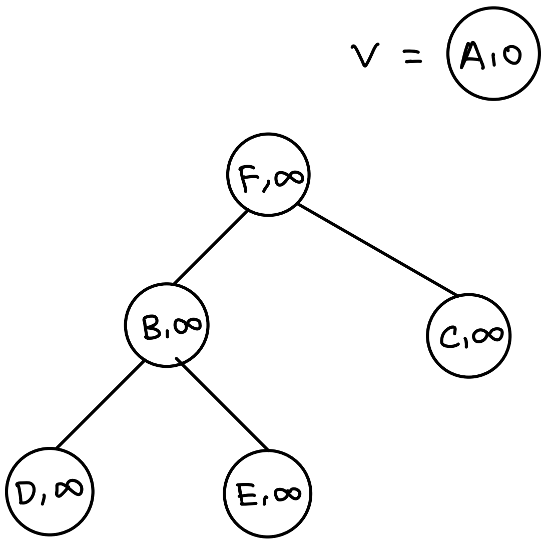


⇩

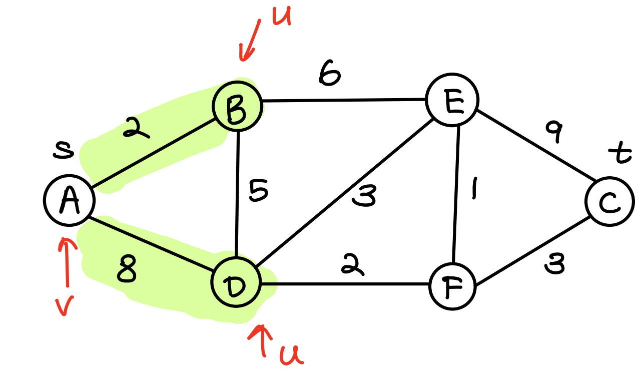
```
Dijkstra(G,w,s,t):
  d = new length |V| array
  for v in V:
    d[v] = infty
  d[s] = 0
  P = new priority queue containing each v in V with priority d[v]
  while P is nonempty:
    v = DeleteMin(P)
    for u adjacent to v:
      if d[v]+w[v,u]<d[u]:
        d[u]=d[v]+w[v,u]
        ChangePriority(u, d[u])
  return d[t]
```



```
Dijkstra(G,w,s,t):
  d = new length |V| array
  for v in V:
    d[v] = infty
  d[s] = 0
  P = new priority queue containing each v in V with priority d[v]
  while P is nonempty:
    v = DeleteMin(P)
    for u adjacent to v:
      if d[v]+w[v,u]<d[u]:
        d[u]=d[v]+w[v,u]
        ChangePriority(u, d[u])
  return d[t]
```



```
Dijkstra(G,w,s,t):
  d = new length |V| array
  for v in V:
    d[v] = infty
  d[s] = 0
  P = new priority queue containing each v in V with priority d[v]
  while P is nonempty:
    v = DeleteMin(P)
    for u adjacent to v:
      if d[v]+w[v,u]<d[u]:
        d[u]=d[v]+w[v,u]
        ChangePriority(u, d[u])
  return d[t]
```



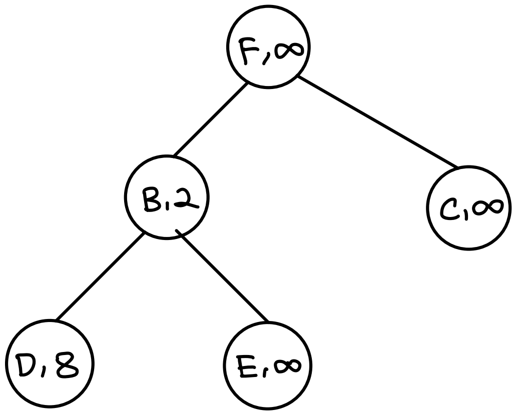
$$d[A] + w[A,B] < \underset{\infty}{d[B]}$$

$$\therefore\ d[B] = d[A] + w[A,B]$$

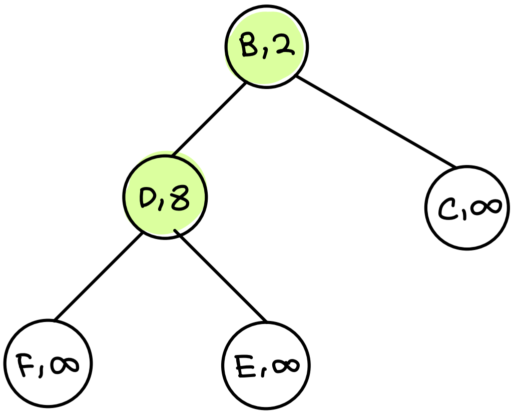
$$= 0 + 2 = 2$$

Similarly, $d[D] = 8$.

```
Dijkstra(G,w,s,t):
  d = new length |V| array
  for v in V:
    d[v] = infty
  d[s] = 0
  P = new priority queue containing each v in V with priority d[v]
  while P is nonempty:
    v = DeleteMin(P)
    for u adjacent to v:
      if d[v]+w[v,u]<d[u]:
        d[u]=d[v]+w[v,u]
        ChangePriority(u, d[u])
  return d[t]
```



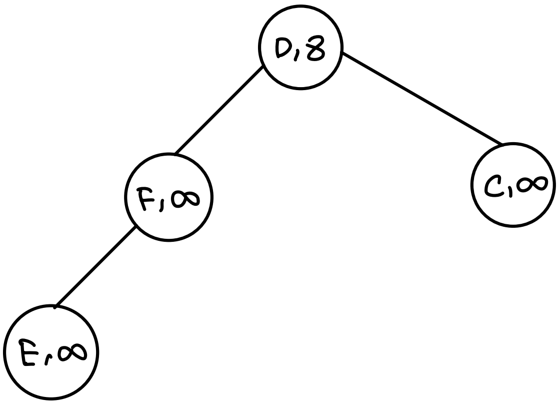
↓ heapify
(under ChangePriority)



⇩

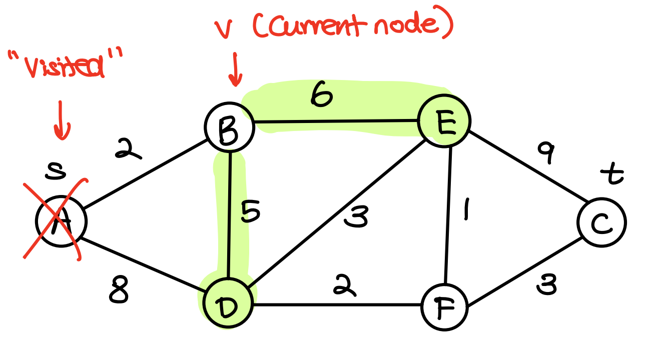
```
Dijkstra(G,w,s,t):
  d = new length |V| array
  for v in V:
    d[v] = infty
  d[s] = 0
  P = new priority queue containing each v in V with priority d[v]
  while P is nonempty:
    v = DeleteMin(P)
    for u adjacent to v:
      if d[v]+w[v,u]<d[u]:
        d[u]=d[v]+w[v,u]
        ChangePriority(u, d[u])
  return d[t]
```

v = B,2



⇩

```
Dijkstra(G,w,s,t):
  d = new length |V| array
  for v in V:
    d[v] = infty
  d[s] = 0
  P = new priority queue containing each v in V with priority d[v]
  while P is nonempty:
    v = DeleteMin(P)
    for u adjacent to v:
      if d[v]+w[v,u]<d[u]:
        d[u]=d[v]+w[v,u]
        ChangePriority(u, d[u])
  return d[t]
```



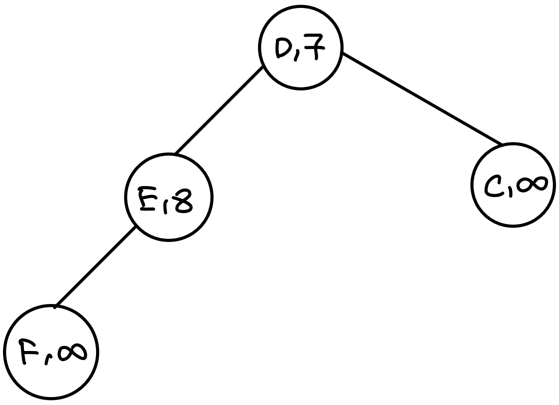
$d[E] = \infty$, ∴ assign $\underbrace{d[B]}_{\text{"shortest to B"}} + \underbrace{w[B,E]}_{\text{distance } B \to E}$

-- 2 + 6 = 8

$d[B] + w[B,D] < d[D]$

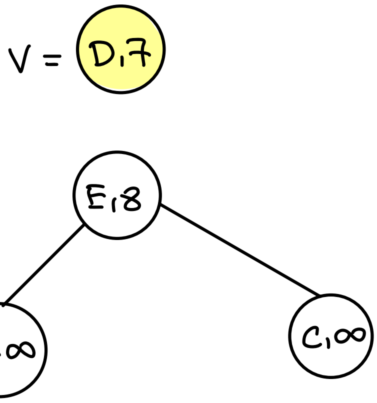
$\underbrace{\text{going by B, then B} \to \text{D}}$ $\underset{\text{shorter}}{<}$ directly to D.

2 + 5 ⟶ assign as d[D]



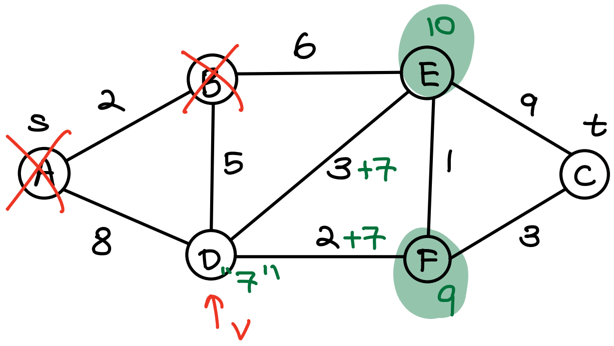
⇩

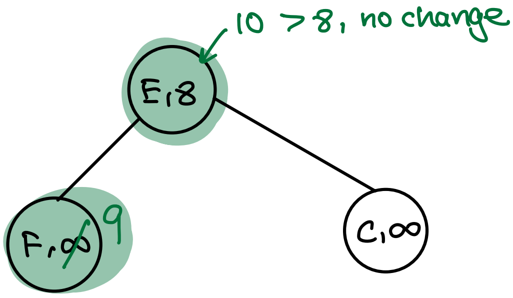
```
Dijkstra(G,w,s,t):
  d = new length |V| array
  for v in V:
    d[v] = infty
  d[s] = 0
  P = new priority queue containing each v in V with priority d[v]
  while P is nonempty:
    v = DeleteMin(P)
    for u adjacent to v:
      if d[v]+w[v,u]<d[u]:
        d[u]=d[v]+w[v,u]
        ChangePriority(u, d[u])
  return d[t]
```

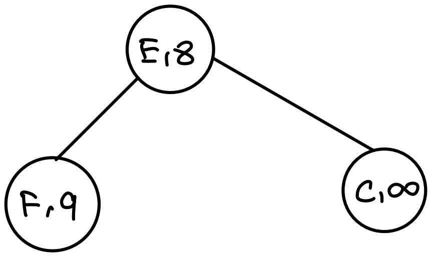


⇩

```
Dijkstra(G,w,s,t):
  d = new length |V| array
  for v in V:
    d[v] = infty
  d[s] = 0
  P = new priority queue containing each v in V with priority d[v]
  while P is nonempty:
    v = DeleteMin(P)
    for u adjacent to v:  → (and not visited)
      if d[v]+w[v,u]<d[u]:
        d[u]=d[v]+w[v,u]
        ChangePriority(u, d[u])
  return d[t]
```

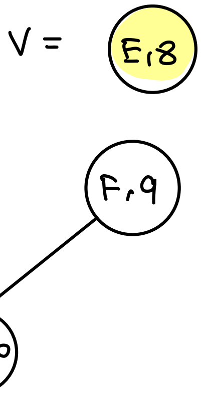






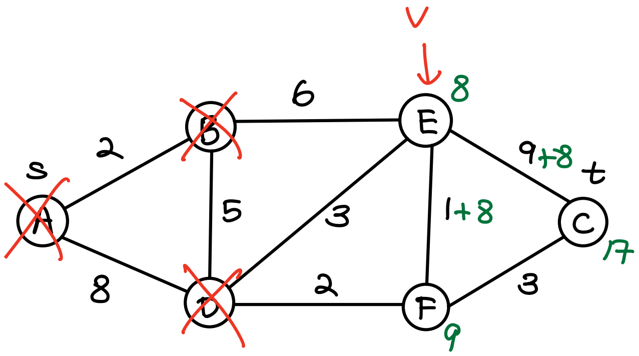
⇩

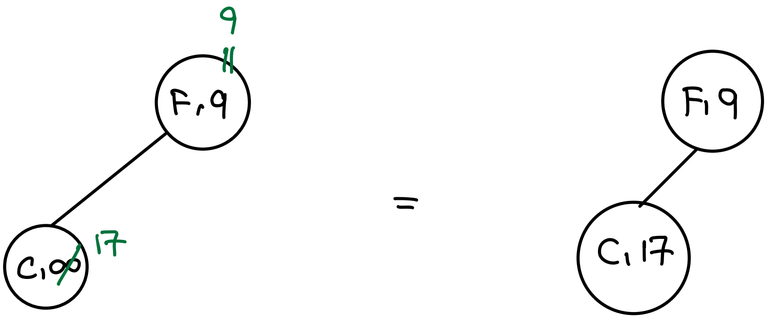
```
Dijkstra(G,w,s,t):
  d = new length |V| array
  for v in V:
    d[v] = infty
  d[s] = 0
  P = new priority queue containing each v in V with priority d[v]
  while P is nonempty:
    v = DeleteMin(P)
    for u adjacent to v:
      if d[v]+w[v,u]<d[u]:
        d[u]=d[v]+w[v,u]
        ChangePriority(u, d[u])
  return d[t]
```



⇩

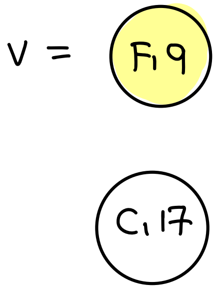
```
Dijkstra(G,w,s,t):
  d = new length |V| array
  for v in V:
    d[v] = infty
  d[s] = 0
  P = new priority queue containing each v in V with priority d[v]
  while P is nonempty:
    v = DeleteMin(P)
    for u adjacent to v:      (and not visited)
      if d[v]+w[v,u]<d[u]:
        d[u]=d[v]+w[v,u]
        ChangePriority(u, d[u])
  return d[t]
```





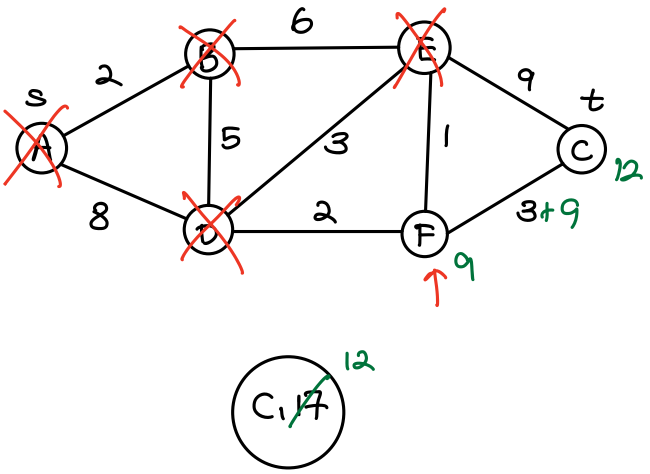
⇩

```
Dijkstra(G,w,s,t):
  d = new length |V| array
  for v in V:
    d[v] = infty
  d[s] = 0
  P = new priority queue containing each v in V with priority d[v]
  while P is nonempty:
    v = DeleteMin(P)
    for u adjacent to v:
      if d[v]+w[v,u]<d[u]:
        d[u]=d[v]+w[v,u]
        ChangePriority(u, d[u])
  return d[t]
```



⇩

```
Dijkstra(G,w,s,t):
  d = new length |V| array
  for v in V:
    d[v] = infty
  d[s] = 0
  P = new priority queue containing each v in V with priority d[v]
  while P is nonempty:
    v = DeleteMin(P)   → (and not visited)
    for u adjacent to v:
      if d[v]+w[v,u]<d[u]:
        d[u]=d[v]+w[v,u]
        ChangePriority(u, d[u])
  return d[t]
```



⇩

DeleteMin(C); do nothing.

⇩

```
Dijkstra(G,w,s,t):
  d = new length |V| array
  for v in V:
    d[v] = infty
  d[s] = 0
  P = new priority queue containing each v in V with priority d[v]
  while P is nonempty:
    v = DeleteMin(P)
    for u adjacent to v:
      if d[v]+w[v,u]<d[u]:
        d[u]=d[v]+w[v,u]
        ChangePriority(u, d[u])
  return d[t]
```

← referencing d[·] changes the distance value stored here.

The value here = when deleteMin of the node happens, since no further updates are available

node	d
A,0	0
B,2	2
C,12	12
D,7	7
E,8	8
F,9	9

∴ d[C] = 12
(for example)

Divide and Conquer Algorithms

Common divide and conquer algorithms:

- Do $\mathcal{O}(n)$ work, divide the problem in half, and recurse on each half. Total work $\approx \mathcal{O}(n\log(n))$. (ie Mergesort, etc.)
- Do $\mathcal{O}(1)$ work, divide in half, and only recurse on one half (ie “decrease and conquer”). (ie. Binary search).

Master Theorem: Suppose that $T : \mathbb{N} \to \mathbb{R}$ is a function (representing runtime complexity) satisfying

$$T(n) = aT\left(\left\lceil \frac{n}{b} \right\rceil\right) + \mathcal{O}(n^d)$$

for some constants $a > 0, b > 1, d \geq 0$, then

$$T(n) = \begin{cases} \Theta(n^d) & \text{if } d > c \quad \text{i.e. “top-heavy”,} \\ \Theta(n^d \log(n)) & \text{if } d = c \quad \text{i.e. “balanced”,} \\ \Theta(n^{\log_b(a)}) & \text{if } d < c \quad \text{i.e. “bottom-heavy”,} \end{cases}$$

for a critical exponent $c := \log_b(a)$.

Example 17. (Midterm Practice Question 4 - Unique Paths.)

Given a weighted graph $G = (V, E)$ and vertices s, v, it is possible that there is no unique shortest path: that is, there are multiple paths from s to v. Given a weighted undirected graph G with positive edge weights and a vertex $s \in V$, we wish to build an array Unique such that for each $v \in V$, `Unique[v] = True` if and only if the shortest path from s to t is unique, `Unique[v] = False` otherwise.

We use the following modification of Dijkstra's algorithm for this. On top of the original Dijkstra's algorithm, if the path is equal in length to one we've already found, then we set `Unique[u] = False`. Alterations from the standard algorithm are in bold and underlined. The code is available below.

(i) What is the runtime of this algorithm?

(ii) Is this algorithm correct? If so, prove that it is correct. If not, give an example of a graph where it fails.

```
Dijkstra(G = (V,E),w,s,t):
    d = new length |V| array
    unique = new length |V| array
    for v in V:
        d[v] = infty
    d[s] = 0
    unique[s] = True
    P = new priority queue containing each v in V with priority d[v]
    while P is nonempty:
        v = DeleteMin(P)
        for u adjacent to v:
            if d[v] + w[v,u] < d[u]:
                d[u] = d[v] + w[v,u]
                ChangePriority(u,d[u])
                unique[u] = unique[v]
            else if d[v] + w[v,u] = d[u]:
                unique[u] = False
    return d[t]
```

Suggested solution:

(i) $\mathcal{O}((|V| + |E|) \log |V|)$. Note that we take an additional $\mathcal{O}(|V|)$ to initialize the array for `unique`, and the additional operations are $\mathcal{O}(1)$.

(ii) This algorithm is correct. One can prove this by inducting on the size of the explored set of vertices, showing that false positive and false negative will not occur. To understand how this algorithm is actually correct, try to run this by hand as per the example above.

Example 18. (h-index, modified.) Given an array of (non-negative) integers `citations` where `citations[i]` is the number of citations a researcher received for their `i`-th paper and `citations` is sorted in **descending** order, return the researcher's h-index.

According to the definition of h-index on Wikipedia: The h-index is defined as the maximum value of `h` such that the given researcher has published at least `h` papers that have each been cited at least `h` times.

To obtain full credit, you must write an algorithm that runs in logarithmic time.

Examples:

Input: `citations = [6,5,3,1,0]`
Output: `3`
Explanation: `[6,5,3,1,0]` means the researcher has papers in total and each of them had received 0, 1, 3, 5, 6 citations respectively. Since the researcher has 3 papers with at least 3 citations each and the remaining two with no more than 3 citations each, their h-index is `3`.

Input: `citations = [100,2,1]`
Output: `2`

Source: https://leetcode.com/problems/h-index-ii/description.

Suggested Solution:

1. (and 4.) Mathematically, we are optimizing (indexing starts from 1):

$$\underset{h}{\text{argmax}} \quad \mathbb{I}\{\texttt{citations}[h] - h \geq 0\}$$

where $\mathbb{I}(\texttt{True}) = 1$ and $\mathbb{I}(\texttt{False}) = 0$. The maximum here makes sense since we are looking at the maximum `h` satisfying the requirement. A candidate `h` satisfies the requirement if there are at least `h` papers cited at least `h` times. If `citations[h]` $\geq$ `h` and the fact that `citations` is sorted in descending order is equivalent to `citations[1], citations[2], ... , citations[h]` (the `h` papers up to the current one) are all $\geq$ `h`. Hence, it suffices to run a binary search by referencing `citations[h] - h`.

2. `def hIndex(citations):`

```
n ← length(citations);
left ← 1; right ← n;
while left <= right do
    mid = ⌈left + (right-left)/2⌉;
    if citations[mid] >= mid then
        left = mid + 1;
    end
    else
        right = mid - 1;
    end
end
return right
```

Remark: Here, we should be returning `right` instead of `left` to ensure that we did not overshoot from our maximum candidate `h`. (Furthermore, the while loop terminating implies that `right < left` (by 1), which further supports our choice of returning `right`.)

3. Runtime complexity: $\Theta(\log(n))$, where $n =$ `length(citations)`.

4. Recall that a binary search algorithm only works on a sorted array and is guaranteed to converge to a solution if that is the case. Here, we are essentially doing a binary search on the modified array `citations[h] - h` that is monotonically decreasing with `h` (which also guarantees the uniqueness of h-index).

5. Each iteration of the while loop cuts the search space by at least half. Hence, we have $2^L = n$, implying that $L = \log_2(n)$ number of iterations of the while loop would be performed. Since each operation within the while loop (lines 4 to 10) are of $\Theta(1)$, this algorithm would take $\Theta(log(n))$ time.

   Remark: Alternatively, using the Master Theorem, since

   $$T(n) = 1 \cdot T\left(\left\lceil \frac{n}{2} \right\rceil\right) + \mathcal{O}(1 = n^0),$$

   then the critical exponent is given by $c = \log_2(1) = 0 = 0$, then this corresponds to the "balanced" case. Thus, we have a runtime of $\Theta(n^0 \log(n)) = \Theta(\log(n))$.

Midterm Review: Additional Problems.

Example 19. (Proof of Correctness - Greedy Algorithms.)

Recall the problem from Homework 3 Question 1 as follows. Suppose that you are in some country where there are n types of coins, which have values $v_1, \cdots, v_n$ and that you have access to an unlimited suply of each type of coins. One day, you decide to buy some item with cost v and you want to do so using as few coins as possible (assume that there is some choice of coins whose values add up to v). Hence, an algorithm was suggested in that problem, in which we attempt to repeatedly find the coin with the highest value less than of equal to the currently remaining amount you need to pay and subtract its value from the balance until you hit 0.

Furthermore, recall that we gave a counterexample to that. Suppose instead that another student thinks that the algorithm is correct, and proceed to using a "staying ahead" argument to prove that it is indeed correct, as follows:

Student's Proof:
To show that the argument is corret, we proceed by proving the following two claims:

Claim 1: Let x_k be the balance after k coins. Then, for our greedy algorithm, we have $x_k \leq x'_k$, where x'_k is the balance after k coins for the optimal algorithm.

Proof: This follows (by induction) from the fact that our algorithm always attempt to pick the largest coin to drop the balance to 0 as fast as possible.

Claim 2: $\texttt{NumCoins} \leq \texttt{NumCoins}'$.

Proof: This follows from Claim 1, since Claim 1 implies that $x_{\texttt{NumCoins}} \leq x_{\texttt{NumCoins}'}$. This implies that with a lower balance, this is attained with a lower number of coins, and thus $\texttt{NumCoins} \leq \texttt{NumCoins}'$. □

Where is the error in the proof above? Explain your answer.

Suggested Solutions:

Recall that a counterexample to the algorithm can be given by trying to fulfill a cost of 110 with coins of values $1, 55,$ and 100. The greedy algorithm would output that we need 11 coins, corresponding to 1×100 coin, and 10×1 coins, while the optimal solution is given by 2×55 coins. Hence, we have $x_1 = 10, x'_1 = 55$, but $x_2 = 9$, while $x'_2 = 0$. This implies that Claim 1 is wrong in the first place! The thing to note is that we could always show that $x_1 \leq x'_1$ but this might not be true for $k \geq 2$. This is because there is a possibility to "skip coins" if they are not admissible, implying that the drop in balance for the second coin by the greedy algorithm might not be as big as that of the optimal algorithm.

Remark: A better intuition would be that picking the bigger coin as fast as possible does not directly minimizes the "objective function" - which is the number of coins required. The subtlety usually comes from understanding if "being greedy in a sense" directly "optimizes" the algorithm. Once you have this newfound perspective, try Question 3 of the Midterm Practice.

Example 20. (Water and Jug Problem.) You are given two jugs with capacities x liters and y liters. You have an infinite water supply. Return whether the total amount of water in both jugs may reach target using the following operations:

- Fill either jug completely with water.
- Completely empty either jug.
- Pour water from one jug into another until the receiving jug is full, or the transferring jug is empty.

Examples:

Input: `x = 3, y = 5, target = 4`
Output: `True`
Explanation:

1. Fill the 5-liter jug (0, 5).
2. Pour from the 5-liter jug into the 3-liter jug, leaving 2 liters (3, 2).
3. Empty the 3-liter jug (0, 2).
4. Transfer the 2 liters from the 5-liter jug to the 3-liter jug (2, 0).
5. Fill the 5-liter jug again (2, 5).
6. Pour from the 5-liter jug into the 3-liter jug until the 3-liter jug is full. This leaves 4 liters in the 5-liter jug (3, 4).
7. Empty the 3-liter jug. Now, you have exactly 4 liters in the 5-liter jug (0, 4).

Remark: See https://www.youtube.com/watch?v=BVtQNK_ZUJg&ab_channel=notnek01.

Input: `x = 2, y = 6, target = 5`
Output: `False`
Explanation: No matter what we do, the total amount of water in all the jugs would be an even number. Hence, we will never get to `target`, an odd number.

Source: https://leetcode.com/problems/water-and-jug-problem/description/.

Suggested Solution:

1. Idea: Run a BFS with the root corresponding to $(x, y) = (0, 0)$, with an edge connected if it satisfies any of the requirements. Check if any of the nodes visited by BFS would satisfy $x + y =$ `target`. (Without loss of generality, we can assume that such a graph would be constructed beforehand or we can do it manually with time complexity $\mathcal{O}(|E|)$ - though this will not change the time complexity as in 5.)

2. `def canMeasureWater(x,y,target):`

```
if x + y > target then
    return False
end
else
    Run BFS with root at (0,0), keeping track of the nodes visited by BFS under visited.
    for all nodes (a,b) in visited do
        if a + b == target then
            return True
        end
    end
    return False
end
```

3. Runtime = $\Theta(xy)$.
4. Since the BFS keep tracks of all possible tuples (x, y) that could be reached by the rules stated in the question, then the `target` is reachable if it corresponds to the sum of the two components in any of the nodes.
5. $|V|$ = Number of nodes = $\mathcal{O}(xy)$, since we consider all possible pairs of non-negative integers (a, b) such that $0 \leq a \leq x$ and $0 \leq b \leq y$.
   $|E|$ = Number of edges $\leq 6\times$ Number of nodes $= \mathcal{O}(xy)$.
   Time complexity = $\mathcal{O}(|V| + |E|) = \mathcal{O}(xy)$.

Remark: Note that there is an alternative method that combines the jugs into a single large jug. Here, the possible moves are $+x, -x, +y, -y$ of the amount of water in the jug. (Transferring water to an alternative jug does not change the total amount of water in the jug. Emptying a jug always corresponds to either $-x$, $-y$, or $-\max\{x, y\} + \min\{x, y\}$, while filling a jug always corresponds to either $+x$, $+y$, or $+\max\{x, y\} - \min\{x, y\}$.) BFS can then be done by tracking the total amount of water in this single jug, giving a runtime complexity of $\Theta(x + y)$ instead as there are now only a maximum number of $x + y + 1$ nodes (and $4(x + y + 1)$ edges).

Remark: There is a much faster way to solve this problem using methods from Math 61. Equivalently, we can map this problem to finding non-negative integers a and b such that $ax + by =$ `target` (see Bézout's Identity). In other words, we could have instead check if $gcd(x, y)$ divides `target`. For the first example, observe that $gcd(3, 5) = 1$, which does divide 4. The runtime complexity of $gcd(x, y) = \Theta(\min(x, y))$, corresponding to the Euclidean algorithm, which is significantly faster than $\mathcal{O}(xy)$.

Remark: In contrast to the mathematical method, the BFS method gives you the exact path and steps such that one would be able to get to the `target` amount of water. Furthermore, using the idea of BFS tree, this automatically gives us the solution that takes the least amount of steps/moves. (This is not something that can be done via DFS or the mathematical method too, which also serves as a significant advantage of BFS over DFS!)

Example 21. (Runtime Complexity.) For each `foo` algorithm below, find a function $f(n)$ such that the algorithm runs in time $\Theta(f(n))$. You should explain the reasoning behind your answer but you do not need to give a formal proof.

```
Foo_3($n$):
1 if $n > 1$ then
2     Foo_3($\lfloor 2n/3 \rfloor$);
3     Foo_3($\lfloor 2n/3 \rfloor$);
4 end
```

```
Foo_5($n$):
1 if $n > 1$ then
2     Foo_5($\lfloor n/2 \rfloor$);
3     Foo_5($\lfloor n/2 \rfloor$);
4     Foo_5($\lfloor n/2 \rfloor$);
5     Foo_5($\lfloor n/2 \rfloor$);
6     for $i = 1, \cdots, n^2$ do
7         pass
8     end
9 end
```

Suggested Solutions:

foo_3:

For `foo_3`, we have covered this in Discussion 2. For simplicity, I will copy the solution over as follows. Observe that the number of levels, L, solves the following equation

$$\left(\frac{3}{2}\right)^L = n$$

and hence

$$L = \log_{3/2} n.$$

Each call of the function branches out twice. This implies that the total runtime would be given by

$$\Theta\left(\sum_{k=0}^{L} 2^k\right) = \Theta\left(\frac{2^{\log_{3/2}(n)+1} - 1}{2 - 1}\right).$$

Since

$$\log_{3/2}(n) = \frac{\log_2(n)}{\log_2(3/2)} = \log_2\left(n^{\log_2(3/2)}\right),$$

we have

$$2^{\log_{3/2}(n)} = 2^{\log_2\left(n^{\log_2(3/2)}\right)} = n^{\log_2(3/2)}$$

and hence the total runtime would be

$$\Theta\left(\sum_{k=0}^{L} 2^k\right) = \Theta\left(n^{\log_2(3/2)}\right) \approx \boxed{\Theta(n^{0.58493})}.$$

foo_5: On top of doing two recursive calls (to create new levels), lines 4 to 6 require an additional $\Theta(n^2)$ runtime for `foo_5`(n). Hence, for the first level, we do

- Four recursive calls of `foo_5`$(n/2^1)$.
- Each of these calls will spend an additional $\Theta\left(\left(\frac{n}{2}\right)^2\right) = \Theta\left(\frac{n^2}{4}\right)$ doing lines 4 - 6.
- Total time spent on this level $= \Theta\left(4 \cdot \frac{n^2}{4}\right) = \Theta(n^2)$.

For the k-th level,

- 4^k recursive calls of `foo_5`$(n \cdot \left(\frac{1}{2}\right)^k)$.
- Each of these calls will spend an additional $\Theta\left(\left(n \cdot \left(\frac{1}{2}\right)^k\right)^2\right) = \Theta\left(\frac{n^2}{4^k}\right)$ doing lines 4 - 6.
- Total time spent on this level = $\Theta\left(4^k \cdot \frac{n^2}{4^k}\right) = \Theta(n^2)$.

Since each level takes the same amount of time independent of the level, the total runtime complexity is given by

$$\Theta\left(\sum_{k=0}^{L} n^2\right) = \Theta(n^2 L) = \boxed{\Theta(n^2 \log(n))}$$

since the number of levels is given by $L = \log_2(n)$ as each level divides the argument of `foo_5`(n) by 2.

Remark: Alternatively, one can apply the Master Theorem here. The runtime complexity follows the recurrence relation:

$$T(n) = 4 \cdot T\left(\left\lceil \frac{n}{2} \right\rceil\right) + \Theta(n^2).$$

Here, since $c = \log_2(4) = 2$, then it belongs to the "balanced" case, and we immediately obtain

$$T(n) = \Theta(n^2 \log(n)).$$

6 Discussion 6

Example 22. (Tower of Hanoi.) The Tower of Hanoi is a mathematical game or puzzle consisting of three rods and a number of disks of differing diameters can slide onto any rod. The puzzle begins with the disks stacked on one rod in order of decreasing size, the smallest at the top, thus approximating a conical shape ("tower"). The objective of the puzzle is to move the entire stack to one of the other rods, obeying the following rules:

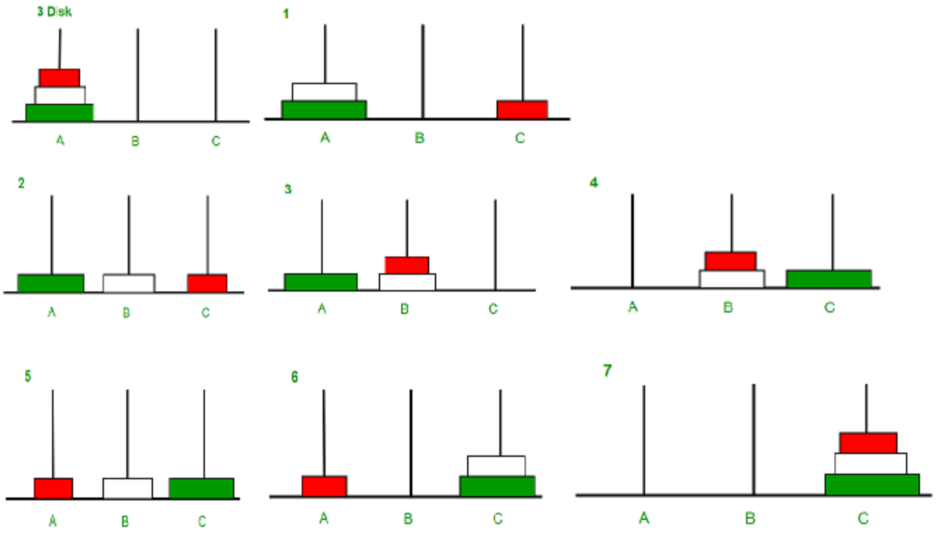
- Only one disk may be moved at a time.
- Each move consists of taking the upper disk from one of the stacks and placing it on top of another stack or on an empty rod.
- No disk may be placed on top of a disk that is smaller than it.

Suppose that the rods are labeled as A, B, and C respectively. Write a function `TowerOfHanoi(n)` that prints out a sequence of steps such that we can move a "tower" of size `n` from 'A' to 'C'. Here, we assume that disk i is of size i, and we start from a disk of sizes $n, n-1, \cdots, 1$ on rod 'A'.

Example:
Input: `TowerOfHanoi(3)`
Output:

```
Move disk 1 from rod A to rod C.
Move disk 2 from rod A to rod B.
Move disk 1 from rod C to rod B.
Move disk 3 from rod A to rod C.
Move disk 1 from rod B to rod A.
Move disk 2 from rod B to rod C.
Move disk 1 from rod A to rod C.
```

Explanation: This is the correct sequence of steps to move a tower of size 3 from rod A to C.



Suggested solution:

1. Idea: Write a helper function `TowerOfHanoi_help(n,from,to,aux)` corresponding the choice of the origin, destination, and the auxilary rods. Then, `TowerOfHanoi(n)` would be `TowerOfHanoi_help(n,‘A’,‘C’,‘B’)`. Here, we use the fact that to move a tower of size `n` from A to C, we

   - Move the tower of size `n-1` from A to B,
   - Move the disk of size n to disk C, and finally,
   - Move the tower of size `n-1` at B to C.

   This method of solving the problem by recursion would give us the correct printed sequence of steps!

2. Hence, it suffices to define `TowerOfHanoi_help(n,from,to,aux)`. Here,
   `def TowerOfHanoi_help(n,from,to,aux):`

```
1 if n = 0 then
2     return (or pass)
3 end
4 TowerOfHanoi_help(n-1,from,aux,to);
5 print(Move disk $n$ from rod from to to);
6 TowerOfHanoi_help(n-1,aux,to,from);
```

3. Runtime: $\mathcal{O}(2^n)$.

4. Proof of correctness: By induction on n for any choice of `from != to != aux` $\in \{A, B, C\}$. Here, the base case corresponds to $n = 0$, which we know that if there are no discs, there is no need to print anything.

   For the induction step, we assume that `TowerOfHanoi_help(k-1,from,to,aux)` is true for `from != to != aux` $\in \{A, B, C\}$. To show that this is true for `TowerOfHanoi_help(k,from,to,aux)` is true for `from != to != aux` $\in \{A, B, C\}$, we let the choice of `from != to != aux` be given. Then, lines 4 - 6 together with the sketch of the recursion logic in 1 and the fact that lines 4 and 6 prints out the right sequences (from the induction hypothesis) allows us to deduce that this is true for `TowerOfHanoi_help(k,from,to,aux)` and we are done.

5. Proof of Runtime - To runtime for the helper function for size n would be given by

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

   Hence, by recursively applying the formula, we have

$$T(n) = \mathcal{O}(2T(n-1)) = \mathcal{O}(2^2T(n-2) = \cdots = \mathcal{O}(2^nT(0)) = \mathcal{O}(2^n).$$

   Remark: Here, you **cannot** use the Master's theorem, and you must analyze the runtime from scratch!

Remark: The etymology of the Tower of Hanoi is believed to have various religious origins. In summary, priests carried a tower with $n = 64$ disks and were asked to “solve the puzzle”. If the legend were true, and if the priests were able to move disks at a rate of one per second, using the smallest number of moves, it would take them $2^{64} - 1$ seconds or roughly 585 billion years to finish, which is about 42 times the estimated current age of the universe. For more information, visit https://en.wikipedia.org/wiki/Tower_of_Hanoi.

Example 23. (Search a 2D Matrix.)
You are given an $m \times n$ integer matrix matrix with the following two properties:

- Each row is sorted in non-decreasing order.
- The first integer of each row is greater than the last integer of the previous row.

Write a function `searchMatrix(matrix,target)` that attempts to search for an integer `target` in the nested array `matrix`; return `True` if target is in else `False`. You must write a solution in $\mathcal{O}(\log(mn))$ time complexity.

Example:

Input: `matrix = [[1,3,5,7],[10,11,16,20],[23,30,34,60]]`, `target = 3`.
Output: `True`.
Explanation: 3 is in `matrix`.

Source: https://leetcode.com/problems/search-a-2d-matrix/.

Suggested solution:

1. Do a binary search for the right row number, followed by a binary search along that row to search for the right column number.
2. Pseudocode: (Iterative)

```
def searchMatrix(matrix,target):
top = 1; btm = length(matrix);
m = length(matrix); n = length(matrix[1]);
while top < btm do
    mid = floor((top+btm)/2);
    if matrix[mid][1] = target then
        return True
    end
    else if matrix[mid][1] > target then
        btm = mid - 1
    end
    else if matrix[mid][n] < target then
        top = mid + 1
    end
    else
        top = mid; Break
    end
end
row = top; Note that this could also be btm, since the while loop terminates if they are equal.
left = 1; right = length(matrix[1]);
while left <= right do
    mid = floor((left+right)/2);
    if matrix[row][mid] = target then
        return True
    end
    else if matrix[row][mid] > target then
        right = mid - 1
    end
    else if matrix[row][mid] < target then
        left = mid + 1
    end
end
return False
```

On the next page, we will consider a recursive version of this code.

```
def searchMatrix(matrix,target):
1  def searchMatrix_row(matrix,target,top,btm):
2      if top = btm then
3          return top
4      end
5      mid = floor((top+btm)/2);
6      if matrix[mid][1] = target then
7          return True
8      end
9      else if matrix[mid][1] > target then
10         return searchMatrix_row(matrix,target,top,mid-1)
11     end
12     else if matrix[mid][n] < target then
13         return searchMatrix_row(matrix,target,mid+1,btm)
14     end
15 row = searchMatrix_row(matrix,target,1,length(matrix));
16 Return True if row is True.
17 def searchMatrix_col(matrix,target,left,right):
18     mid = floor((left,right)/2);
19     if left = right then
20         return matrix[row][mid] = target
21     end
22     else if matrix[row][mid] > target then
23         return searchMatrix_col(matrix,target,left,mid-1)
24     end
25     else if matrix[row][mid] < target then
26         return searchMatrix_col(matrix,target,mid+1,right)
27     end
28 return searchMatrix_col(matrix,target,1,length(matrix[1]));
```

3. Runtime: $\mathcal{O}(\log(m) + \log(n)) = \mathcal{O}(\log(mn))$.

4. Proof of correctness (Iterative). To do so, we claim that if `target` is in the matrix,

   - `row` in line 14 will give the right row number if `target` is not in the first column, and
   - The output of `searchmatrix(matrix,target)` is correct.

   To argue for the first point, we consider the following loop invariant: `target` is always in the range `[matrix[top][1],matrix[btm][n]]` `(n = length(matrix[1]))` for given indices `top` and `btm` in the while loop from lines 2 to 13 if it is not found yet.

   Base Case: This follows from our assumption that `target` is in `matrix`.

   Induction Case: Assume that `target` is in `[matrix[top][1],matrix[btm][n]]`. We would like to show that the new interval would also contain `target` (if it is not found yet).

   - Lines 4 - 5: `target` is found, and we return the right Boolean value.
   - Lines 7 - 9: `target` not found; line 7 and the fact that the "matrix column is sorted" implies that it must lie between `[matrix[top][1],matrix[mid][1])` or in other words, `[matrix[top][1],matrix[mid-1][n]]` (since `target` is in `matrix`).
   - Lines 10 - 12: Similarly, this implies that it must lie `(matrix[mid][n],matrix[btm][n]]` or in other words, `[matrix[mid+1][1],matrix[btm][n]]`.
   - Lines 14 - 16: Else, if both line 8 and 11 are False, then we have `target` in `[matrix[mid][1],matrix[mid][n]]`. By setting `top = mid` and hence `row = top` in Line 18, we are in the right row to do a binary search in the later parts of the code.

   Hence, the conclusion holds by induction.

Next, we would like to argue for the second point. From the first point, we've deduced that if `target` is in `matrix`, then `target` is in `[matrix[row][1],matrix[row][n]]`. We then repeat the usual binary search argument as follows.

Loop invariant: `target` is always in the range `[matrix[row][left],matrix[row][right]]` if not found yet.

Base Case: This follows from our assumption that `target` is in `matrix`.

Induction Case: Assume that `target` is in `[matrix[row][left],matrix[row][right]]`.

- Lines 22 - 24: `target` is found, and we return the right Boolean value.
- Lines 25 - 27: `target` not found; line 8 and the fact that the "matrix column is sorted" implies that it must lie between `[matrix[row][left],matrix[row][mid])` or in other words, `[matrix[row][left],matrix[row][mid-1]]` (since `target` is in `matrix`).
- Lines 28 - 30: Similarly, this implies that it must lie `(matrix[row][mid],matrix[row][right]]` or in other words, `[matrix[row][mid+1],matrix[row][right]]`.

Hence, the conclusion holds by induction.

On the other hand, we have to argue that there are no false positives - if `target` is not in `matrix`, then we should return False. Here, consider any output for `row` in line 18. We would like to show that the code that follows always return False. This follows from the fact that lines 22 - 24 will never be activated, and the search interval always shrinks by at least one, implying that the while loop must terminate. Hence, line 31 activates, returning False.

Remark: One can argue similarly for the recursive version.

5. Proof of runtime: $\mathcal{O}(\log(m))$ to search for the row in which the target is in via binary search, and $\mathcal{O}(\log(n))$ to search for the target along that row.

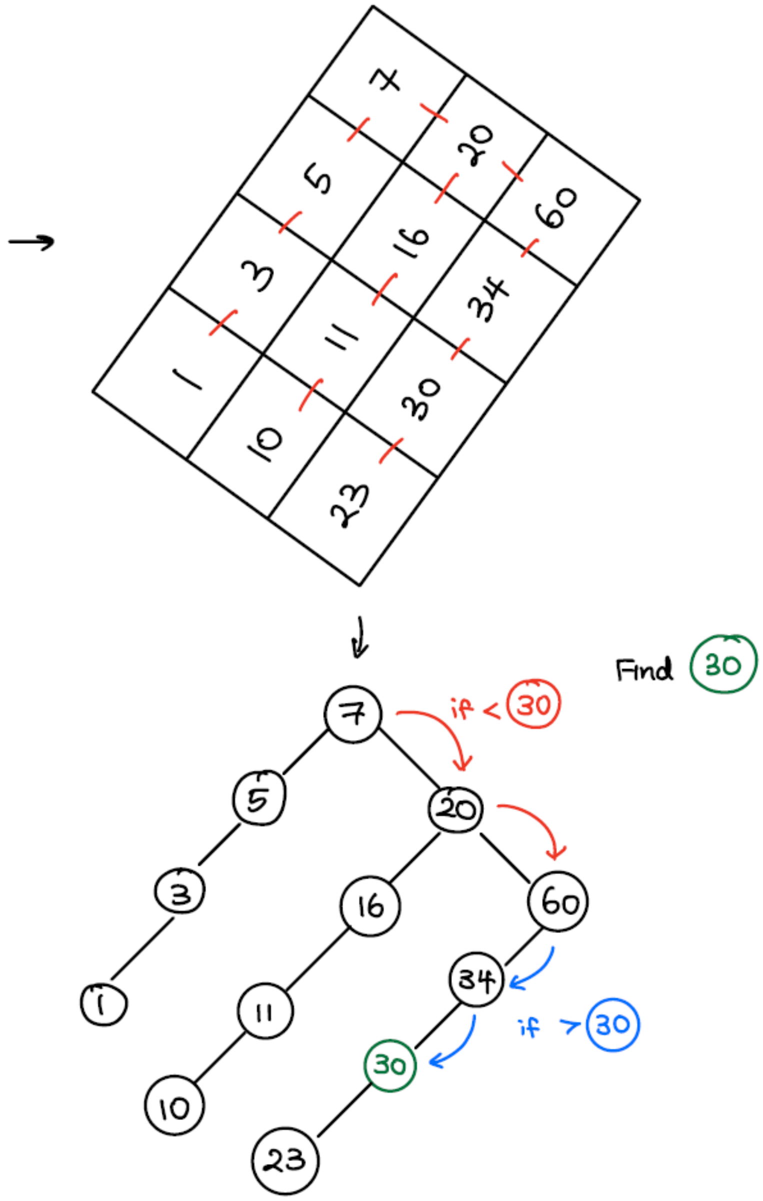
Remark: Note that lines 19 to 32 are exactly the lines for the binary search algorithm. Personally, I think it is okay to just say to run the binary search algorithm along the row `row` and not provide too much details here.

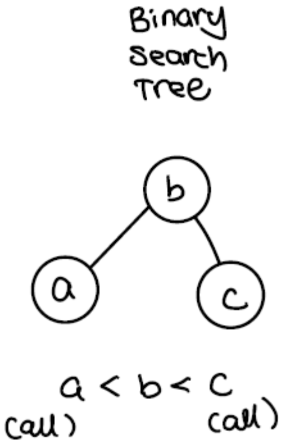
Remark: If we tilt the `matrix` in the example a little, we can view this as a **binary search tree** (BST)! This means that we just have to traverse from the root of the tree downwards, and the time complexity will be of the order of the height of the tree. In this case, that would be $\mathcal{O}(n+m)$. Even though this would not be as good as a double binary search, this is still at least better than a brute-force approach and serves as an interesting perspective to the problem!

Remark: Note that I've corrected for an error in the representation of the BST in the diagram below (I had the wrong edges connected in the previous version). Thanks to Haotian for pointing it out!

Credits: Leetcode community contributer here.

1	3	5	7
10	11	16	20
23	30	34	60





7 Discussion 7

Dynamic Programming (DP)

- Main Idea: Naively solve a problem, but optimize the code in a certain way.
- Usually contains an "optimal substructure" structure - ie given that the solution is optimal up till step n (and saved somewhere), we want to obtain a solution that is optimal up till step $n+1$.

The usual sequence of reducing time/space complexity includes:

1. Formulate the problem recursively.
2. **Memoization** (a.k.a top-down approach): Record each output in an array somewhere - if it was previously computed, we just have to access its value (and not go on with recursion).
3. **Tabulation** (a.k.a bottom-up approach): Convert the recursive code into an iterative one via means of "tabulating", computing the solution "upwards" starting from the base cases.

   Remark: This is the version that was mostly covered in lectures, so I will be focusing on that. Nonetheless, it is good to know what the memoization approach is too since sometimes it might be easier to come up with the memoization approach.

Example 24. (House Robber I.)
You are a professional robber planning to rob houses along a street. Each house has a certain amount of money stashed, the only constraint stopping you from robbing each of them is that adjacent houses have security systems connected and it will automatically contact the police if two adjacent houses were broken into on the same night.

Given an (positive) integer array `nums` representing the amount of money of each house, return the maximum amount of money you can rob tonight without alerting the police.
Example:

Input: `nums = [2,7,9,3,1]`.
Output: `12`.
Explanation: Rob house 1 (`money = 2`), 3 (`money = 9`), and 5 (`money = 1`).

Source: https://leetcode.com/problems/house-robber/description/.

Suggested solution:
Here, the idea is to perform dynamic programming. The recursive step here follows the following logic:

- If I do not rob house `n`, then I continue to rob from houses `n + 1` onwards.
- If I rob house `n`, then I am not allowed to rob house `n + 1`. I can only continue to rob from houses `n + 2` onwards.

This will serve as the code for the recursive version and the memoized version of the problem. For the tabulation version, we need to shift our perspective as follows. If I am at house n,

- If I had robbed house `n - 1`, then I am not allowed to rob house `n`. Hence, the maximum amount at this step would be `rob(n-1)`.
- If I did not rob house `n - 1`, then I need to rob house `n` (since this is a better solution than not robbing both of these houses). Hence, the maximum amount at this step would be equivalent to whatever combinations that gives me the maximum I get from robbing up till house `n - 2`, plus the amount I get from robbing house `n`.

Recursive Code

1. See above.
2. Pseudocode: (Assuming indices start from 1)
   `def rob(nums):`
   ```
   1 def rob_right(i):
   2     if i > length(nums): then
   3         return 0
   4     end
   5     return max(rob_right(i+1), rob_right(i+2) + nums[i])
   6 return rob_right(1)
   ```
3. Runtime: $\Theta(F_n)$.
4. Proof of correctness: Here, we induct on i, the argument for `rob_right`. (Assume without loss of generality that `length(nums) >= 3`.)
   Base Case: `i = length(nums)`. By line 5, since `rob_right(length(nums)+1)` and `rob_right(length(nums)+2)` are both zero, we have that `rob_right(length(nums)) = nums[length(nums)]`, consistent with the solution that having only one house, we should just rob that. One can argue similarly for `i = length(nums) - 1`.
   Induction Case: We proceed by strong induction (actually, assuming that it is true for $i+1$ and $i+2$ would be sufficient). We would like to show that `rob_right(i)` is correct. This follows from the logic described above.

   By induction, this is true for all `i <= length(nums)`, and is thus true for `i = 1`, corresponding to robbing the entire `nums` array.

5. Proof of runtime: The runtime follows $T(n) = T(n-1) + T(n-2) + \mathcal{O}(1)$, which was proved during lectures that this corresponds to the n-th Fibonacci number.

Memoization:

The only difference is in the psuedocode, and hence the time complexity, which we will provide below.

```
def rob(nums):
 1 Initialize memo as an array with size length(nums).
 2 def rob_right(i):
 3     if i > length(nums): then
 4         return 0
 5     end
 6     if memo[i] != -1 then
 7         return memo[i]
 8     end
 9     memo[i] = max(rob_right(i+1), rob_right(i+2) + nums[i])
10     return memo[i]
11 return rob_right(1)
```

The new time complexity is at $\Theta(n)$, where n is `length(nums)`. This follows from the fact that we only call line 9 for n times (since if it was a pre-computed `rob_right` value, we would just access it from `memo` with $\mathcal{O}(1)$ time).

Tabulation:

1. See above.

2. Pseudocode:

```
def rob(nums):
 1 n ← length(nums);
 2 if n = 1 then
 3     return nums[1]
 4 end
 5 if n = 2 then
 6     return max(nums[1],nums[2])
 7 end
 8 return rob_right(1)
 9 Initialize array dp of size $n$ with dp[1] = nums[1] and dp[2] = max(nums[1],nums[2])
10 for i = 3,...,n do
11     dp[i] = max(dp[i-1], dp[i-2] + nums[i])
12 end
13 return dp[n]
```

3. Runtime: $\Theta(n)$.

4. Proof of correctness. We induct on the fact that `dp[i]` gives the maximum money that the robber can get if we have houses from `num[1:i]`. The base case can be argued easily. For the induction step, we proceed by strong induction, and (similar to the recursive version) argue based on the logic from the sketch above.

5. Proof of runtime. This follows from the fact that initializing array of size n is of $\Theta(n)$ and we run line 11 a total of $n-2$ times (with each run taking up $\Theta(1)$ time).

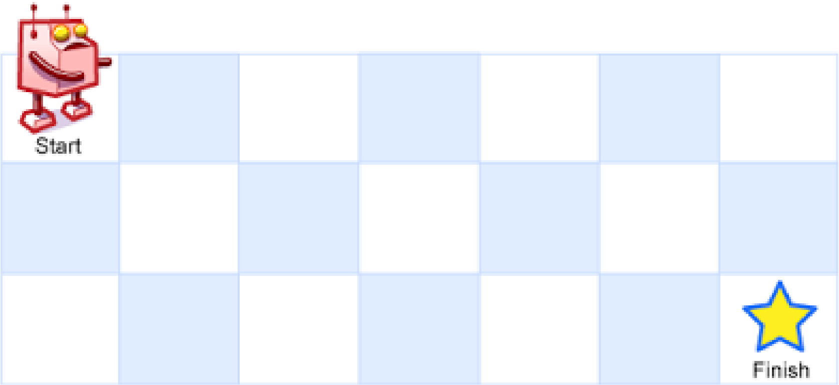
Remark: A more general version for houses with distance `k` apart is basically the version that you see in Question 2 of Homework 5.

Example 25. (Unique Path.) There is a robot on an $m \times n$ grid. The robot is initially located at the top-left corner (i.e., `grid[1][1]`). The robot tries to move to the bottom-right corner (i.e., `grid[m][n]`). The robot can only move either down or right at any point in time.

Given the two integers `m` and `n`, return the number of possible unique paths that the robot can take to reach the bottom-right corner.

Examples:
Input: `m = 3, n = 2`.
Output: 3.
Explanation: Three ways - RDD, DRD, and DDR (D = Down, R = Right).

Input: `m = 3, n = 7`.
Output: 28.



Source: https://leetcode.com/problems/unique-paths/description/.
Suggested solution:

1. Here, we conduct a 2D DP, and log the number of ways to go to a grid `grid[i][j]`. Here, the recursive step involves `grid[i][j] = grid[i][j-1] + grid[i-1][j]`.

2. Pseudocode:
   `def unique_paths(m,n):`

```
Initialize an array grid of size m x n with grid[1][j] = grid[i][1] = 1 for all valid i and j.
for $i$ = 2, ..., $m$ do
    for $j$ = 2, ..., $n$ do
        grid[i][j] = grid[i][j-1] + grid[i-1][j];
    end
end
return grid[m][n]
```

3. Runtime: $\Theta(mn)$.

4. Proof of correctness: We consider the proposition

   P(i) = "`grid[i][j]` records the correct number of ways to go to that grid, for all `1 <= j <= m`".

   Base Case: $i = 1$. This is true since we have initialized the first row to be all rows, since the only way to reach any of the grid points on the first row is to always move to the right.
   Induction Case: We assume that $P(i)$ is true and attempt to show that $P(i+1)$ is true. Let `1 <= j <= m` be given, and this is then equivalent to showing that `grid[i][j]` is true for any arbitrarily given `j`.

   At this point in time, the rigorous thing to do is to do a second induction on `j`. We will not go over the details but instead sketch the idea here. Recall that we always get to use the fact that $P(i)$ is true (from the outer induction). Hence, this inner indunction argument corresponds to showing that `grid[i+1][j]` for `1 <= j <= m`. The base case corresponds to `grid[i+1][1]`, where we have already set this to be one (since the only way for this to happen is to always go down for any `i`). For the induction case, we get to also assume that `grid[i+1][j’]` is true for all `1 <= j’ <= j`. To show that `grid[i+1][j+1]` is true then follows from the fact that `grid[i+1][j]` is true (inner induction assumption) and `grid[i][j+1]` (outer induction assumption) and that the number of ways to get to `grid[i+1][j+1]` is precisely the sum of these two values (ie see the logic in 1)!

Remark: In general, an induction proof with two variables is a little tricky since there are 2 variables to induct on. To resolve this, we usually only induct on one of the variables, and suppose that it is true on all other values of the variables. To show that it is true on all other values of the variables, this usually involves a second induction argument.

5. Proof of runtime: Initializing an array of size $m \times n$ is of $\Theta(mn)$, and we run line 4 a total of $\Theta(mn)$ times with each run taking $\Theta(1)$.

Remark: This is an example of a 2D dynamic programming problem. In other words, sometimes, it might be useful to think of tabulation in 2D, as compared to be just along an array. See Question 4 of Homework 5 for a generalization of this.

Remark: Space optimization: We could instead use a single array of size n, and update the values on the go by line 4. This reduces the space complexity to just $\Theta(m)$. (This is just for your enrichment, since we don’t focus on space complexity in this class.)

Remark: Time optimization: Mathematical method. The solution to this uses a basic idea in combinatorics, that is, rearranging the letters D and R in a row. In general, this problem reduces to arranging $m-1$ copies of D and $n-1$ copies of R, which yields a solution

$$\frac{(m+n-2)!}{(m-1)!(n-1)!}.$$

The time complexity would thus corresponds to the time it takes to compute $(m+n-2)!$, which is of $\Theta(m+n)$.

Remark: Further optimization - optimizing the constants! Here, to compute the expression above, we could use memoization to record the values of factorial on the go. This implies that we will really only spend at most $m+n-2$ operations computing all the required factorials, rather than recomputing the factorials again with a separate calls for each of the three factorials.

Example 26. (Coin Change II.) You are given an integer array `coins` representing coins of different denominations and an integer `amount` representing a total amount of money.

Return the number of combinations that make up that amount. If that amount of money cannot be made up by any combination of the coins, return 0. You may assume that you have an infinite number of each kind of coin.

Example:
Input: `amount = 5`, `coins = [1,2,5]`.
Output: 4
Explanation: There are four ways to make up the amount: 5, 2+2+1, 2+1+1+1, 1+1+1+1+1.

Source: https://leetcode.com/problems/coin-change-ii/description/.
Suggested solution:

1. Initialize a 2D `dp` array to tabulate the number of ways - i.e. `dp[amount]` is the number of ways to make up `amount` (index starting from zero). The base case here would be `dp[i][0] = 1` for all `i`, since there is a way to make up an amount of zero, and `dp[0][j] = 0` for all `j` (the value at `[0][0]` doesn't matter) since an empty `coins` means that there are no ways in which one can make up these amounts.

   The recursive step involves the change in the array `dp` as you introduce a new coin `c`. For each new coin, we go through the `dp` array and do:

   `dp[i][j] = dp[i-1][j] + dp[i][j-c].`

   In other words, the number of ways to make this amount with this new coin added into `coin` is equivalent to the number of ways we can do that amount without this coin, plus the number of ways we can do it with the coin in but for a smaller amount and we just have to add (use) this coin. (These are mutually exclusive events that makes up what we need to compute.)

2. Pseudocode:

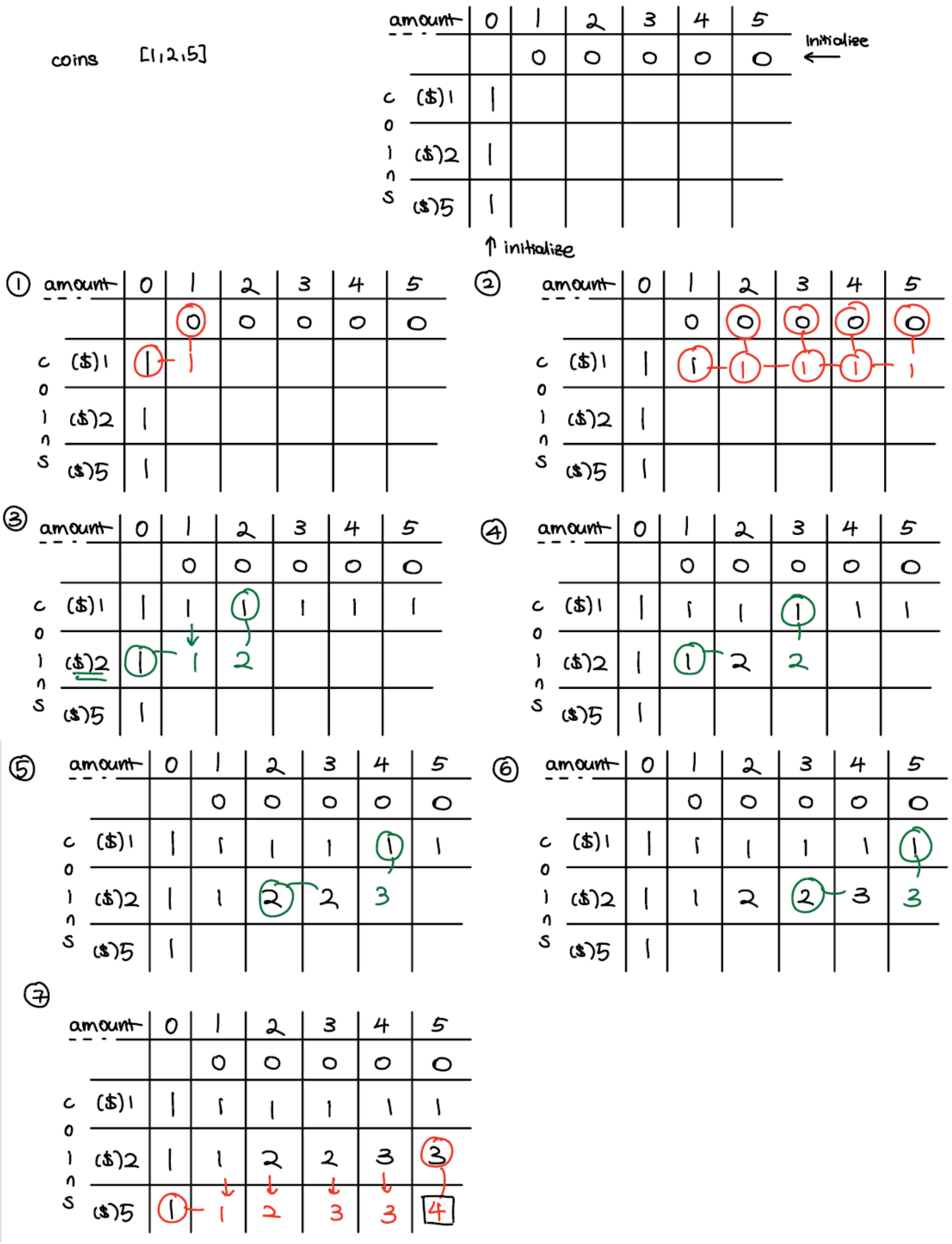
```
def change(coins,amount):
 1 n ← length(coins);
 2 Initialize an array dp of size (n+1) x (amount+1) with grid[0][j] = 0 and grid[i][0] = 1 for all
     valid i and j.
 3 for i = 1, ..., n do
 4     c ← coin[i-1];
 5     Set dp[i][j] = dp[i-1][j] for j from 1 to c-1;
 6     for j = c, ..., amount do
 7         dp[i][j] = dp[i-1][j] + dp[i][j-c];
 8     end
 9 end
10 return dp[n][amount]
```

3. Runtime: $\Theta($`length(coins)` $\times$ `amount`$)$.

4. The proof follows a similar logic as the previous example. I will skip the details since this discussion will probably be getting a little too long.

5. Line 2: $\Theta($`length(coins)` $\times$ `amount`$)$, and we are basically doing line 7 a total of $\Theta($`length(coins)` $\times$ `amount`$)$ times, with each iteration taking $\Theta(1)$ time.

The diagram below would help you to understand the logic behind the 2D DP tabulation approach above.



coins [1,2,5]

	amount	0	1	2	3	4	5
			0	0	0	0	0
coins	($)1	1					
	($)2	1					
	($)5	1					

Initialize ←

↑ initialize

①

	amount	0	1	2	3	4	5
			0	0	0	0	0
coins	($)1	1	1				
	($)2	1					
	($)5	1					

②

	amount	0	1	2	3	4	5
			0	0	0	0	0
coins	($)1	1	1	1	1	1	1
	($)2	1					
	($)5	1					

③

	amount	0	1	2	3	4	5
			0	0	0	0	0
coins	($)1	1	1	1	1	1	1
	($)2	1	1	2			
	($)5	1					

④

	amount	0	1	2	3	4	5
			0	0	0	0	0
coins	($)1	1	1	1	1	1	1
	($)2	1	1	2	2		
	($)5	1					

⑤

	amount	0	1	2	3	4	5
			0	0	0	0	0
coins	($)1	1	1	1	1	1	1
	($)2	1	1	2	2	3	
	($)5	1					

⑥

	amount	0	1	2	3	4	5
			0	0	0	0	0
coins	($)1	1	1	1	1	1	1
	($)2	1	1	2	2	3	3
	($)5	1					

⑦

	amount	0	1	2	3	4	5
			0	0	0	0	0
coins	($)1	1	1	1	1	1	1
	($)2	1	1	2	2	3	3
	($)5	1	1	2	3	3	4

Remark: Space Optimization: Observe that as per the previous example, the new values along a row only depends on the values from the previous row and the entries to the left of it. This implies that it suffices to keep an array and update it for each coin, cutting the space complexity of our solution.

Pseudocode:

```
def change(coins,amount):
Initialize an array dp of size (amount+1) with grid[0] = 1.
for c in coins do
    for j = c, ..., amount do
        dp[j] = dp[j] + dp[j-c];
    end
end
return dp[amount]
```

Note that this code is a lot shorter. Albeit with the same runtime complexity, the space complexity is now at $\Theta($amount$)$.

Remark: This is also known as the $0-\infty$ knapsack problem, since we have unlimited supplies of each coin! For a different objective function, see Question 1 of Homework 5, involving the original coin change problem from Homework 3! Here, we note some of the variations of the knapsack problem:

- $0-1$ knapsack problem - this corresponds to the fact that you can only use the coin once. This might have been covered in class. If not, there are a ton of resources that you can consult for this.
- Fractional knapsack problem - this corresponds to the fact that you might have to break up the coins (with values pegged on the broken “pieces” of the coin). For an example, see Question 3 of Homework 5.
- Most knapsack problems can be done using a 2D tabulation approach, which one can further optimize for space complexity if needed (as described in the previous example and in this example)!

8 Discussion 8

Types of DP Problems + Problem Solving Template:
Template: A `dp` array would be tabulated and fill by iteration, depending on the previous entries.

1D DP:

- Depends on $\underline{k}$ previous entries - `dp[i]` depends on `dp[i-k], dp[i-k+1], ..., dp[i-1]`.

  Runtime: $\mathcal{O}(nk)$ or $\mathcal{O}(n)$ if k is fixed beforehand (and not as an input to the function).

  Examples:

  - Fibonacci ($k = 2$, Week 6 Notes).
  - House Robber (Discussion 7 Example 1).
  - Maximum Subarray (Week 6 Notes).

- Depends on all previous entries - `dp[i]` depends on `dp[1], ..., dp[i-1]`.

  Runtime: $\mathcal{O}(n^2)$, since it requires $\mathcal{O}(i)$ step for the computation of the i-th entry of `dp`, and hence gives a triangular sum sort of time complexity (ie $\sum_{i=1}^{n} i = \frac{n(n+1)}{2} = \mathcal{O}(n^2)$).

  Examples:

  - Longest Increasing Subsequence a.k.a LIS (Week 6 Notes).
  - Russian Dolls (Discussion 8/This Discussion, Example 2 - Variation of LIS).
  - Some problem(s) in HW 6.

2D DP:

- Has a general matrix structure given (ie input array is 2D, etc).

  Runtime: $\mathcal{O}(mnf(m,n))$, where m and n are the number of rows and columns of the given array, and $f(i,j)$ is usually the time it takes to compute `dp[i][j]`.

  Examples:

  - Unique Paths and Block Paths (Discussion 7 Example 2, HW 5 Question 4).
  - Fractional Knapsack/Selling a Rectangular Cloth (HW 5 Question 3).

- Two inputs (or associated properties) to the function, each of which are realized with a 1D array. The strategy is to "cross" the two inputs in a 2D `dp` array.

  Runtime: $\mathcal{O}(mnf(m,n))$, where m and n are the sizes of the two given array, and $f(i,j)$ is usually the time it takes to compute `dp[i][j]`.

  Examples:

  - 0-1 Knapsack (Week 7 Notes).
  - 0-∞ Knapsack Problems like Coin Change (HW 5 Question 1), and Coin Change II (Discussion 7 Example 3).
  - Edit Distance (Week 7 Notes).
  - Longest Common Subsequence a.k.a LCS (HW 5 Question 5/Discussion 8 Example 1).

- Usually, if `dp[i][j]` only depends on the values on the current row `dp[i][:j-1]` and its value on the same column but right before this index `dp[i-1][j]`, then we can cut down space complexity by only using a single 1D `dp` array (see Unique Paths and Coin Change II in Discussion 7).

Example 27. (Longest Common Subsequence.)
Given two strings `text1` and `text2`, return the length of their longest common subsequence. If there is no common subsequence, return `0`. Recall that a subsequence of a string is a new string generated from the original string with some characters (can be none) deleted without changing the relative order of the remaining characters. A common subsequence of two strings is a subsequence that is common to both strings.

Example:

Input: `text1 = "abcde"` and `text2 = "acfe"`
Output: 3.
Explanation: Longest common subsequence would be `"ace"`.

Source: https://leetcode.com/problems/longest-common-subsequence/description/ and Homework 5 Question 5 (Optional Problem).

Suggested solution:
This falls under the scenario of "2 variables", so the idea is to create a 2D `dp` recording the first string along the column and the second string along the row. Then, the dp step involves

- If `text1[i] = text2[j]`, then `dp[i][j] = dp[i-1][j-1]` since the new common subsequence will be the original one (up to `[i-1][j-1]`) with this new common string appended to it.
- Else, `dp[i][j] = max(dp[i-1][j],dp[i][j-1])`. This is because since the appended character from `text1[i]` and that from `text2[j]` are not the same, then the length of the common subsequence will not change (since these added characters would not be common). Hence, the longest common subsequence will the the maximum of a subsequence up to [i][j-1] and then adding `text2[j]` and that up to [i-1][j] and then adding `text1[i]` (note that there are no + 1 because the new "character" at [i][j] is not "common").

1. See above.
2. Pseudocode:

```
def LCS(text1,text2):
1  m,n ← length(text1),length(text2);
2  Initialize a 2D dp array of size (m+1) × (n+1), with dp[1][j] = dp[i][1] = 0 for all i,j.
3  for i = 2, ..., m+1 do
4      for j = 2, ..., n+1 do
5          if text1[i] = text2[j] then
6              dp[i][j] = dp[i-1][j-1] + 1;
7          end
8          else
9              dp[i][j] = max(dp[i-1][j],dp[i][j-1]);
10         end
11     end
12 end
13 return dp[m+1][n+1]
```

3. Runtime: $\mathcal{O}(nm)$.
4. Proof of correctness. Following the double induction argument back in Discussion 7, it suffices to show that the base case makes sense and the induction step holds. For the base case, it follows from the fact that `dp[1][j] = dp[i][j] = 0` since there should be no common subsequence between a string and an empty string (represented by the first row/column). For the induction step, we assume that `dp[i'][j']` is true for all `i' <= i` and `j' <= j` but `i'` and `j'` cannot be both `i` and `j` simultaneously. Then, the induction step holds by following the logic in the sketch of the pseudocode.
5. Proof of runtime. Lines 5 to 9 runs in $\mathcal{O}(1)$ time, for a total of $\mathcal{O}(mn)$ times. This thus gives a total runtime complexity of $\mathcal{O}(mn)$.

text2

text1

		a	c	f	e
	0	0	0	0	0
a	0				
b	0				
c	0				
d	0				
e	0				

$$dp[i][j] = \begin{cases} dp[i-1][j-1]+1 & \text{if } text1[i] = text2[j] \\ \max(dp[i][j-1], dp[i-1][j]) & \text{" } \neq \text{ "} \end{cases}$$

text2

text1

		a	c	f	e
	0	0	0	0	0
a	0	1			
b	0				
c	0				
d	0				
e	0				

text2

text1

		a	c	f	e
	0	0	0	0	0
a	0	1	1	1	1
b	0				
c	0				
d	0				
e	0				

text2

text1

		a	c	f	e
	0	0	0	0	0
a	0	1	1	1	1
b	0	1	1	1	1
c	0				
d	0				
e	0				

text2

text1

		a	c	f	e
	0	0	0	0	0
a	0	1	1	1	1
b	0	1	1	1	1
c	0	1	2	2	2
d	0				
e	0				

longest subsequence up till here = "a" from 0 ↘ 1

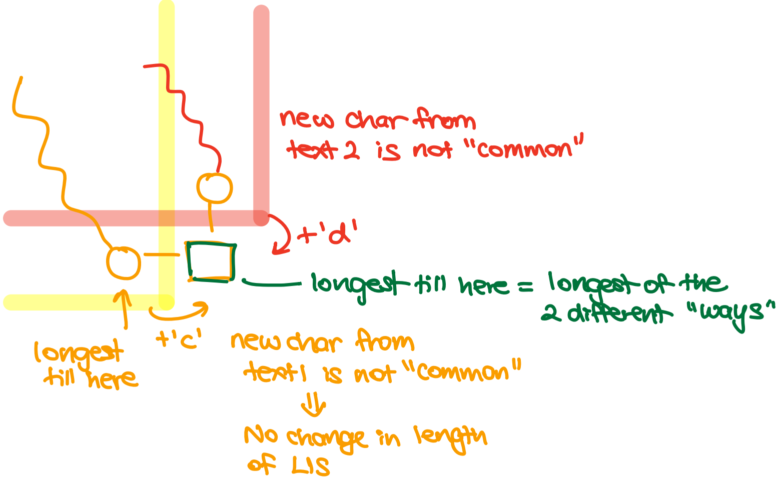
$\underbrace{1}_{\text{subseq for}} + 1$

subseq for + "c" ⇒ "ac"

text2

text1

		a	c	f	e
	0	0	0	0	0
a	0	1	1	1	1
b	0	1	1	1	1
c	0	1	2	2	2
d	0	1	2	2	2
e	0				


new char from
text 2 is not "common"
+'d'
longest till here = longest of the 2 different "ways"
longest till here
+'c'
new char from text1 is not "common"
⇓
No change in length of LIS

text2

text1

		a	c	f	e
	0	0	0	0	0
a	0	1	1	1	1
b	0	1	1	1	1
c	0	1	2	2	2
d	0	1	2	2	2
e	0	1	2	2	3

Example 28. (Russian Doll Envelopes.) You are given a 2D array of positive integers `envelopes` where `envelopes[i] = [wi, hi]` represents the width and the height of an envelope. One envelope can fit into another if and only if both the width and height of one envelope are greater than the other envelope's width and height. Note that you are not allowed to rotate the envelopes (since well, it wouldn't make sense - see the image below!).

Return the maximum number of envelopes you can Russian doll (i.e., put one inside the other).

Example:

Input: `envelopes = [[5,4],[6,5],[6,7],[7,8],[2,3],[10,10]]`.
Output: `5`.
Explanation: The maximum number of envelopes you can Russian doll is `5`. `([2,3] => [5,4] => [6,5] -> [7,8] -> [10,10])`

Source: https://leetcode.com/problems/russian-doll-envelopes/description/.

An image of a set of Russian dolls from
https://www.macalester.edu/russian-studies/about/resources/miscellany/matryoshka/:

Here, the general idea would be to sort the `envelopes` array by the first entry of the pair in increasing order, and the second entry in decreasing order (if the first entries of two pairs are the same). Then, we perform the longest increasing subsequence algorithm using only the height (second entry) on the sorted list. The trick of sorting ties in decreasing order of their second entry actually ensures that the "smallest admissible envelope" will be picked last.

For example, upon sorting, we have `envelopes = [[2,3],[5,4],[6,7],[6,5],[7,6],[10,10]]`. Recall that the LIS algorithm at `i` checks over all indices `j < i` with values that are lower, and take the maximum over those values and add one. By sorting the second entry in decreasing order (for pairs with the same value on its first entry) would result in envelopes of the same width being only picked once! It is easier to see this by looking at what happens if we sort the second entry by increasing and decreasing order.

Decreasing: `envelopes = [[2,3],[5,4],[6,7],[6,5],[7,8],[10,10]]`
or `envelopes = [[2,3],[5,4],[6,7],[6,5],[7,8],[10,10]]`

Increasing: `envelopes = [[2,3],[5,4],[6,5],[6,7],[7,8],[10,10]]`.

Hence, observe that the first type of sort would work, but not the second since we ended up picking envelopes with the same width but of increasing height!

Suggested solution:

1. See above.

2. Pseudocode:

```
def maxEnvelopes(envelopes):
1 Sort the envelopes array by the first entry in increasing order, but for ties, we sort them in decreasing
   order.
2 Initialize an array dp of size n = length(envelopes).
3 for $i$ = 1, ..., $n$ do
4   | dp[i] = max(1,dp[j]+1 over all j < i with envelopes[j][2] < envelopes[i][2])
5 end
6 return dp[n]
```

3. Runtime: $\mathcal{O}(n^2)$.

4. Skipped; see the sketch of the algorithm above. The key is to argue that the LIS created using heights obtained after sorting the envelopes is the sequence of envelopes to be chosen such that we get the maximum number of envelopes.

5. Proof of runtime: Sorting the list $\mathcal{O}(n\log(n))$, while the for loop takes $\mathcal{O}(n^2)$ (triangular sum argument, since each iteration i takes $\mathcal{O}(i)$ time). Hence, we have $\mathcal{O}(n^2)$.

Remark: Lines 2 to 6 is just the LIS algorithm performed on the height of the sorted envelopes array.

Remark: Note that LIS can be performed in $\mathcal{O}(n\log(n))$ by a non-DP approach (ie Greedy + Binary Search). For more information, see https://leetcode.com/problems/longest-increasing-subsequence/solutions/1326308/c-python-dp-binary-search-bit-segment-tree-solutions-picture-explain-o-nlogn/.

Ford-Fulkerson Algorithm - An Introduction.

Objective: Maximum Flow Problem - Compute the maximum flow from one vertex (source) to another (sink).

Key steps of the algorithm:

1. **Initialization:**
   - Construct a residual graph `G'` with a directed edge going in each direction between any pairs of vertices. If the edge does not exist in the original graph `G`, we set the weight of the edge to be `0`.
   - Furthermore, we initialize the flow on each directed edge on the original graph `G` to be `0`.
2. **Maintenence:**
   - Find a path from `s` to `t`. This is done using the following helper function:

     ```
     def FindPath(G',w',s,t):
     Use BFS/DFS to search for an s-t path in G' with all edge weights >0
     return that path
     ```

   - Find the maximum amount of flow that you can send down that path, and modify the residual graph (decrease weights along paths, and increase weights along directed edges in the opposite direction).
   - These are done using the `Augment` helper function below:

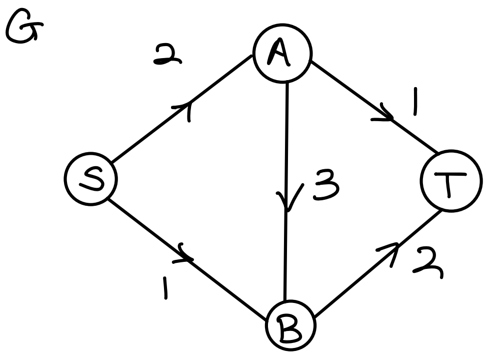
     ```
     def Augment(P,G = (V,E), G',w'):
     Obtain flow from G (as a pointer so that we can modify G as we modify flow);
     b = min(w'(e) for e in P); (Maximum possible flow we can send along that path till capacity is
       reached along one of the edges.)
     for (u,v) in E do
         if (u,v) is an edge in G then
             flow[u,v] += b
         end
         else
             flow[v,u] -= b
         end
         w'[u,v] -= b;
         w'[v,u] += b;
     end
     ```

3. **Termination:**
   - Repeat until there aren't any `s-t` paths on `G'`.
   - Return the flow recorded on the original graph `G`.

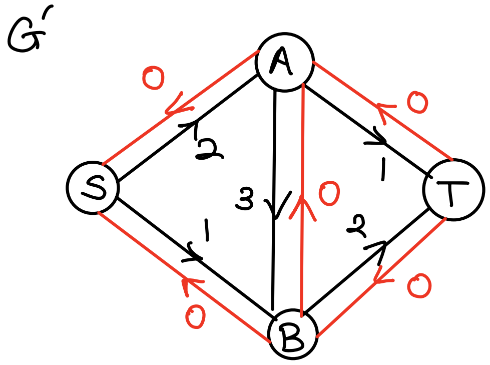
Time complexity: $\mathcal{O}(|E|F)$ where $|E|$ is the number of edges in the graph and F is the maximum flow value (assuming that each weight is an integer).

Full algorithm:

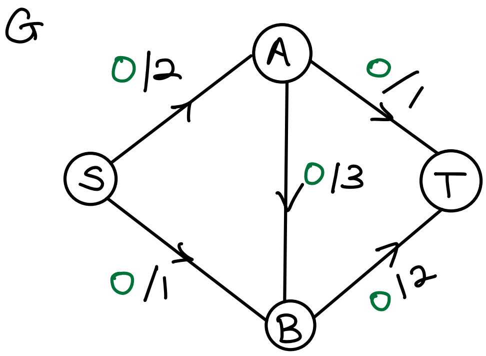
```
def MaxFlow(G=(V,E), w, s ,t):
Initialization
Make a copy of G',w' from G,w;
Obtain flow from G (as a pointer so that we can modify G as we modify flow);
for (u,v) in E do
    if (v,u) not in E then
        Add (v,u) to G' with weight w'(u,v) = 0;
    end
    flow[u,v] = 0
end
Maintenance and Termination
while True do
    p = FindPath(G',w',s,t);
    if p is not empty then
        Augment(p,G,G',w')
    end
    else
        return flow
    end
end
```



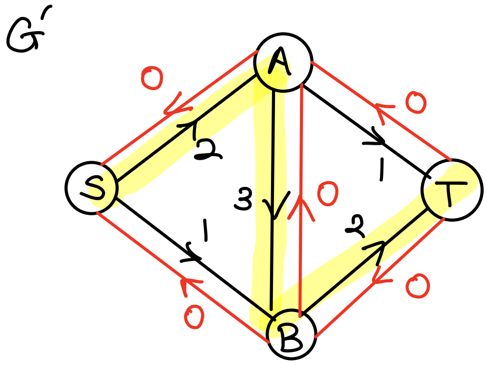
```
MaxFlow(G = (V,E), w,s,t):
//Setup Residual Graph and Flow:
  G’,w’ a copy of G,w
  for (u,v) in E:
      if (v,u) not in E:
          add (v,u) to G’ with weight w’(u,v) = 0
  for (u,v) in E:
      flow[u,v] = 0
  While True:  //Find Path:
      p = FindPath(G’,w’,s,t)
//Modify Graph:
      if p not empty:
          Augment(P, flow, w’)
      else:
          return flow
```



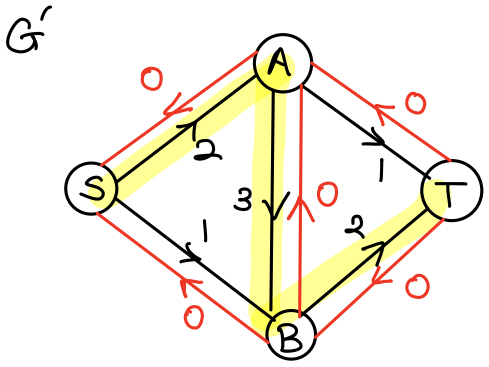
```
MaxFlow(G = (V,E), w,s,t):
//Setup Residual Graph and Flow:
  G’,w’ a copy of G,w
  for (u,v) in E:
      if (v,u) not in E:
          add (v,u) to G’ with weight w’(u,v) = 0
  for (u,v) in E:
      flow[u,v] = 0
  While True:  //Find Path:
      p = FindPath(G’,w’,s,t)
//Modify Graph:
      if p not empty:
          Augment(P, flow, w’)
      else:
          return flow
```



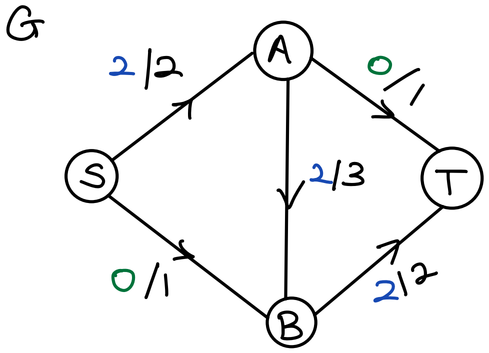
```
MaxFlow(G = (V,E), w,s,t):
//Setup Residual Graph and Flow:
  G’,w’ a copy of G,w
  for (u,v) in E:
      if (v,u) not in E:
          add (v,u) to G’ with weight w’(u,v) = 0
  for (u,v) in E:
      flow[u,v] = 0
  While True:  //Find Path:
      p = FindPath(G’,w’,s,t)
//Modify Graph:
      if p not empty:
          Augment(P, flow, w’)
      else:
          return flow
```



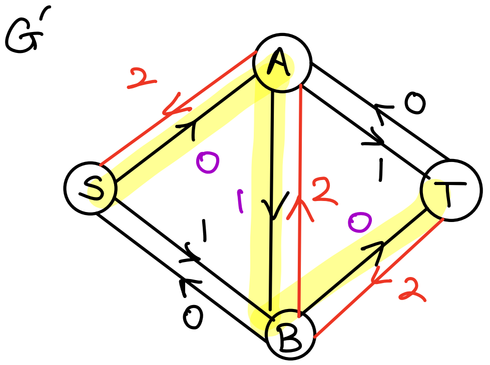
```
Augment(P,G,flow,w'):
  b = min(w'(e) for e in P) = 2
  for (u,v) in P:
      if (u,v) is an edge in G:
          flow[u,v] = flow[u,v]+b
      else:
          flow[v,u] = flow[v,u]-b
      w'[u,v] = w'[u,v]-b
      w'[v,u] = w'[v,u]+b
```

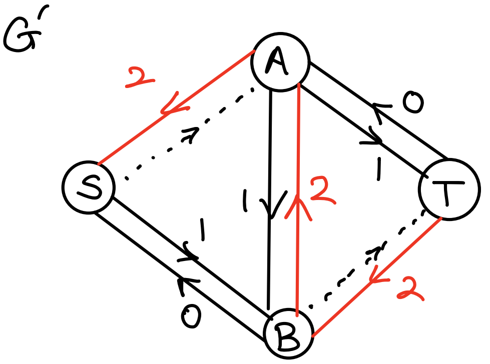


```
Augment(P,G,flow,w'):
  b = min(w'(e) for e in P)
  for (u,v) in P:
      if (u,v) is an edge in G:
          flow[u,v] = flow[u,v]+b
      else:
          flow[v,u] = flow[v,u]-b
      w'[u,v] = w'[u,v]-b
      w'[v,u] = w'[v,u]+b
```



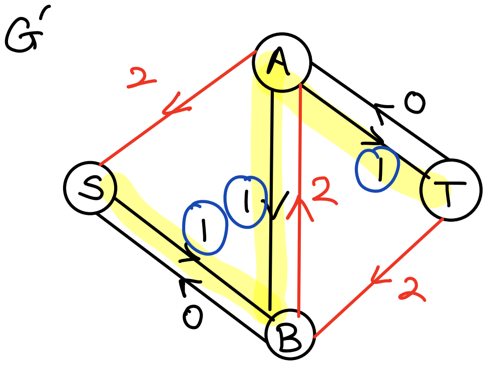
```
Augment(P,G,flow,w'):
  b = min(w'(e) for e in P)
  for (u,v) in P:
      if (u,v) is an edge in G:
          flow[u,v] = flow[u,v]+b
      else:
          flow[v,u] = flow[v,u]-b
      w'[u,v] = w'[u,v]-b
      w'[v,u] = w'[v,u]+b
```



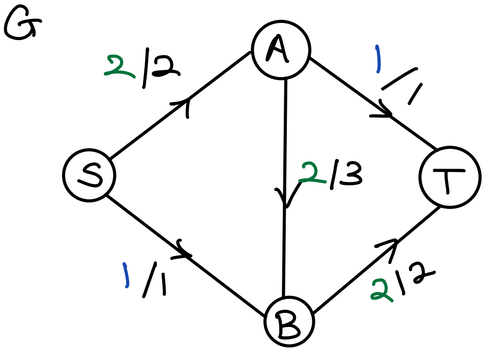


↳ next iteration

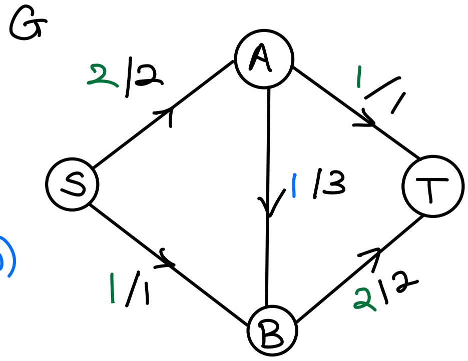
```
MaxFlow(G = (V,E), w,s,t):
//Setup Residual Graph and Flow:
  G',w' a copy of G,w
  for (u,v) in E:
      if (v,u) not in E:
          add (v,u) to G' with weight w'(u,v) = 0
  for (u,v) in E:
      flow[u,v] = 0
  While True:  //Find Path:
      p = FindPath(G',w',s,t)
//Modify Graph:
      if p not empty:
          Augment(P, flow, w')
      else:
          return flow
```



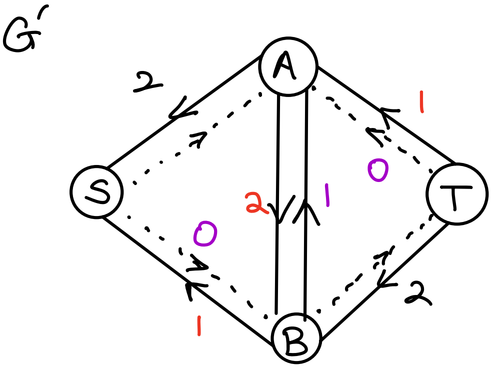
```
Augment(P,G,flow,w'):
  b = min(w'(e) for e in P)  =1
  for (u,v) in P:
      if (u,v) is an edge in G:
          flow[u,v] = flow[u,v]+b
      else:
          flow[v,u] = flow[v,u]-b
      w'[u,v] = w'[u,v]-b
      w'[v,u] = w'[v,u]+b
```



```
Augment(P,G,flow,w'):
  b = min(w'(e) for e in P)  =1
  for (u,v) in P:
      if (u,v) is an edge in G:
          flow[u,v] = flow[u,v]+b
      else:
          flow[v,u] = flow[v,u]-b  (backflow)
      w'[u,v] = w'[u,v]-b
      w'[v,u] = w'[v,u]+b
```



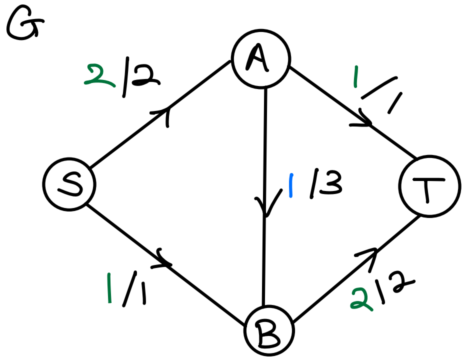
```
Augment(P,G,flow,w'):
  b = min(w'(e) for e in P)
  for (u,v) in P:
      if (u,v) is an edge in G:
          flow[u,v] = flow[u,v]+b
      else:
          flow[v,u] = flow[v,u]-b
      w'[u,v] = w'[u,v]-b
      w'[v,u] = w'[v,u]+b
```



⇒ No more paths from s-t

⇒ Terminate

⇒ Return



9 Discussion 9

Maximum Flow/Minimum Cut Problems & Algorithm:

- Ford-Fulkerson algorithm terminates if all edges have integer weights.
- Integrality Theorem: If all edges have integer weights, then there exists a maximum flow for which the flow along every edge is an integer value. In fact, the output of Ford-Fulkerson is a maximum flow with integer flow values on each edge.
- Runtime complexity:
  - General Ford-Fulkerson: $\mathcal{O}(F|E|)$ where F is the maximum flow value.
  - BFS Fold-Fulkerson (ie BFS will be used to find paths a.k.a Edmonds-Karp): $\mathcal{O}(|V||E|^2)$.

Cuts. For a given network G, an $s-t$ cut in G is a partition of the vertices V into two disjoint sets A and B, where $s \in A$ and $t \in B$. Furthermore, we consider an $s-t$ flow f.

- The capacity of the cut, written as $w(A, B)$, is the sum of the weights of the edges going out of A.
- The flow out of A, denoted by $f^{\text{out}}(A)$, is the sum of the flow along edges going out of A.
- The flow in of A, denoted by $f^{\text{in}}(A)$, is the sum of the flow along edges going into A.
- The size of a flow f, denoted by size(f), is the total flow from s to t.
- Regardless of whether it is a max flow or not, we always have the following equation for a cut:

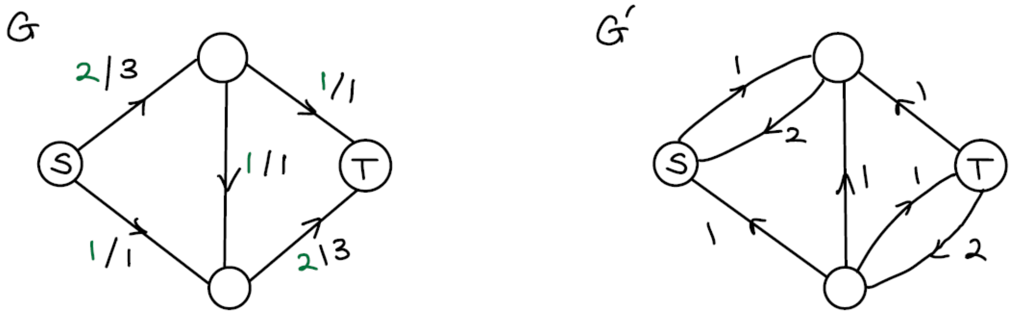
$$\text{size}(f) = f^{\text{in}}(A) - f^{\text{out}}(A). \tag{1}$$

Last but not least, we recall the theorem that we've used to prove that the Ford-Fulkerson algorithm will always terminate and find a flow with the maximum possible size - the **Max-Flow/Min-Cut** theorem.

Theorem (Max-Flow/Min-Cut). The maximum size of an $s-t$ flow on a network G is equal to the minimum weight over $s-t$ cuts of G. The proof is actually useful to know, as it depends on the following two key steps:

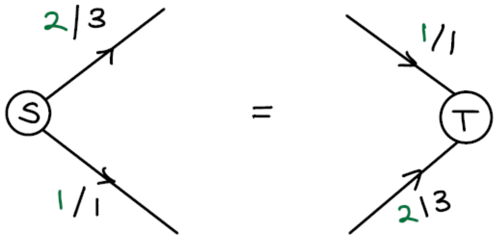
- Max-Flow $\leq$ Cut for any cut. (In particular, Max-Flow $\leq$ Min-Cut.)
- Consider the output of Ford-Fulkerson. Using the residual graph as a reference, we construct a cut to the original network G (ie consisting of all vertices reachable from s in the residual graph), and argue that the max flow (size) from Ford-Fulkerson is equals to the weight of this cut. Thus, Max-Flow $\geq$ Min-Cut.

Output of Fort-Fulkerson:



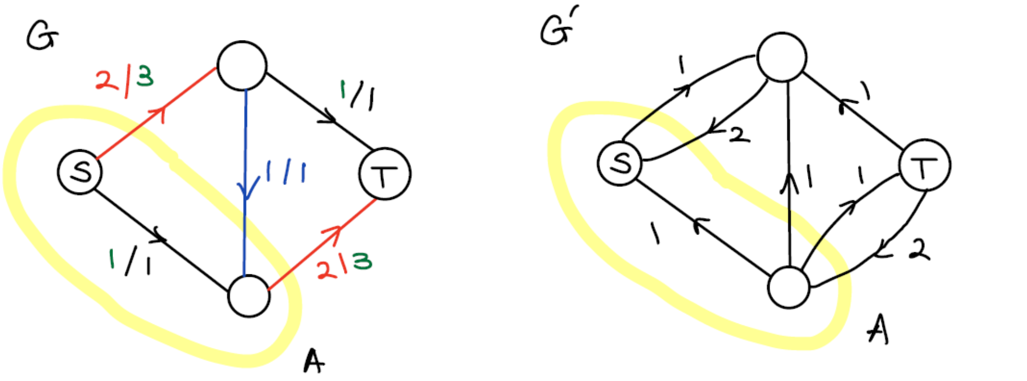
flow f

size(f) =



= 3

Example of cut:



$f^{in}(A) = 1$

$f^{out}(A) = 4$

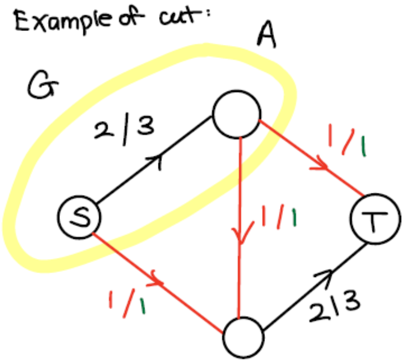
$$\underset{3}{\text{size}(f)} = \underset{4}{f^{out}(A)} - \underset{1}{f^{in}(A)}$$

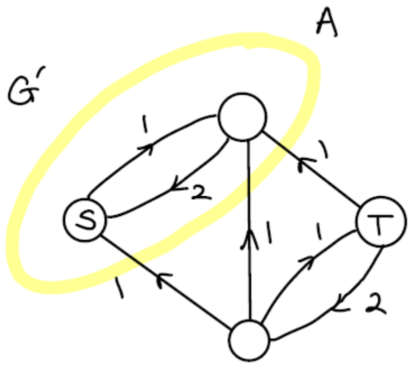
$w(A,B) = 3 + 3 = 6$

In this case,

$$\underset{3}{\text{size}(f)} < \underset{6}{w(A,B)}$$

Example of cut:





Remark: Cut w/ all nodes reachable from ⓢ in G'.

$f^{in}(A) = 0$ (none)

$f^{out}(A) = 3$

$$\underset{3}{\text{size}(f)} = \underset{3}{f^{out}(A)} - \underset{0}{f^{in}(A)}$$

$w(A,B) = 1 + 1 + 1$

In this case,

$$\underset{3}{\text{size}(f)} = \underset{3}{w(A,B)}$$

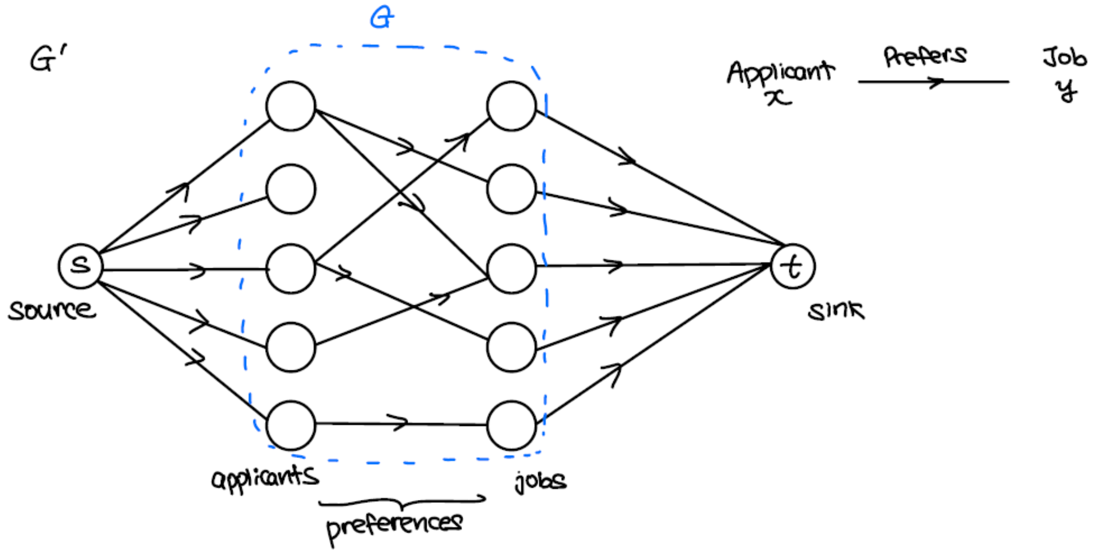
(agrees w/ converse step for proof of max-flow/min-cut theorem)

Example 29. (Bipartite Matching Problem.) There are m job applicants and n jobs. Each applicant has a subset of jobs they are interested in. Each job opening can accept only one applicant, and each applicant can be assigned to at most one job. Design an efficient algorithm to find an assignment of jobs to applicants that maximizes the number of applicants who get jobs.

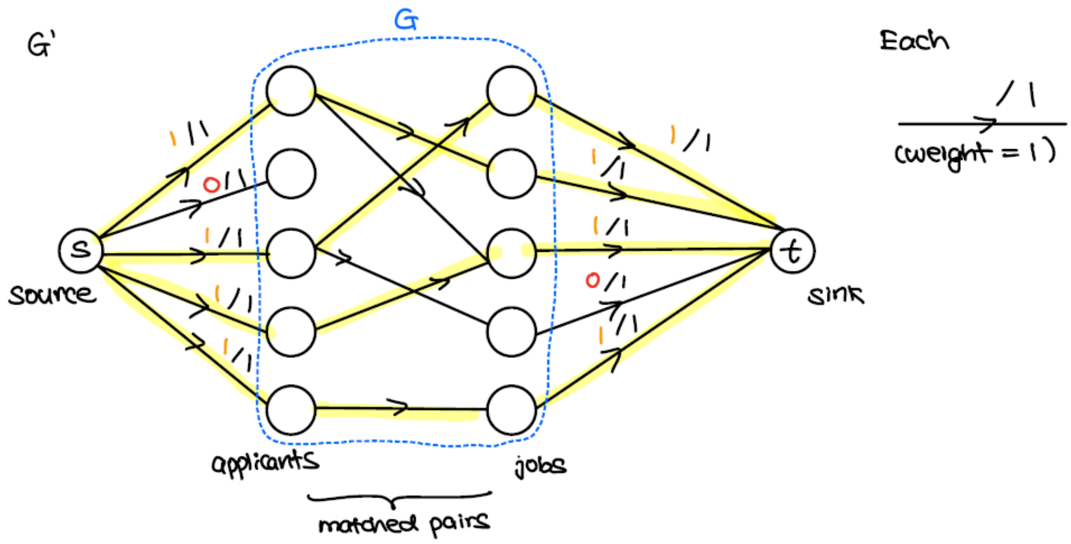
Suggested solution:

Remark: This is also known as the bipartite matching problem and would be useful in solving one of the problems in Homework 7.

1. The idea here is map this to a network flow problem. To do so, we construct auxiliary "source" and "sink" nodes that has a directed edge to all applicants and jobs respectively, together with nodes corresponding to applicants and jobs. Furthermore, a directed edge between an applicant and a job corresponds to an applicant's preference for that job.



Then, the max flow problem from the source to the sink corresponds to the solution to the problem. In other words, the max flow value is the maximum number of applicants who will get a job. (This will be the part that requires more details in the proof of correctness.)



2. Pseudocode: (In plain English)

```
1 Construct nodes for a "source", a "sink", for each applicant and for each jobs.
2 Construct a directed edge with capacity/weight 1 from the source to each node corresponding to an
    applicant.
3 Construct a directed edge with capacity/weight 1 from each node corresponding to a job, to the sink.
4 For each applicant-job pair given, construct a directed edge with capacity/weight 1 from the (node
    representing the) applicant to the (node representing the) job.
5 Run a maximum flow algorithm (ie BFS-based Fork-Fulkerson) to obtain the maximum flow from the
    source to the sink, and store the flow value (size of the flow) as flow.
6 return flow.
```

3. Runtime Complexity = $\mathcal{O}(|V||E|^2) = \mathcal{O}((m+n)\cdot(mn)^2) = \mathcal{O}(m^3n^2 + m^2n^3)$.

Remark: If we are allowed to express time complexity as a function of the number of edges (ie pairings), then $|E| = |E'| + n + m$ (with $|E'|$ represents the total number of applicant-job pairs). This gives $\mathcal{O}((m+n)^3 + (m+n)^2|E| + (m+n)|E|^2) = \mathcal{O}((m+n)^3 + (m+n)|E|^2)$.

4. The proof of correctness lies with proving the following claim:

   The maximum number of matches (in G') is equals to the maximum flow in the network G.

   To do so, we argue as follows. Suppose that the size of the flow is some integer k and that each flow value is either 0 or 1.

   - Take a cut with the source node and applicants in one of the partitions (and call that A). By (1), we have size$(f) = k$ and $f^{\text{in}}(A) = 0$ (since there are no edges point into A). Hence, $f^{\text{out}}(A) = k$.
   - Since each edge in G' carries a flow of 0 or 1, then there must be a total of k edges pointing out of the set A carrying a unit flow. In other words, there will be a total of k edges carrying a unit flow in G'.
   - Argue that each applicant node is the tail of at most one edge in G. (Suppose for a contradiction that it is a tail of two (or more) edges; then the flow value out of the applicant node is 2; while the maximum flow value into the applicant node is 1 by construct - hence a contradiction!)
   - Similarly, argue that each job node is the head of at most one edge in G.
   - Hence, each edge carrying a flow (ie in G') must originate from a unique applicant to a unique job (unique here means "not shared" among any other edges).
   - Hence, the number of matches would be given by k (with a match defined as a unit flow across the edge in G').

   In other words, we have shown that for an integer flow size k, with each flow value that is either 0 or 1, k is also the number of matches.

   Since by the integrality theorem, there is a max flow with flow along every edge to be an integer (in fact, as output of the Ford-Fulkerson algorithm), then the size of the max flow is also an integer. Hence, the hypothesis of the lemma above holds, and k is the maximum number of matches.

5. Note that the number of vertices is given by $m + n + 2$ (number of applicants + number of jobs + source + sink), while the number of edges is of the order $\mathcal{O}(m + mn + n) = \mathcal{O}(mn)$ (since we have m edges from source to applicants, n edges from jobs to sink, and a maximum of mn edges between applicants and jobs since each of the m applicants can be interested in any of the n jobs listed).

   - Time complexity of line 1 = $\mathcal{O}(|V|) = \mathcal{O}(m+n)$.
   - Time complexity of line 2 = $\mathcal{O}(m)$.
   - Time complexity of line 3 = $\mathcal{O}(n)$.
   - Time complexity of line 4 = $\mathcal{O}(|E|) = \mathcal{O}(mn + m + n) = \mathcal{O}(mn)$.
   - Time complexity of line 5 = $\mathcal{O}(|V||E|^2) = \mathcal{O}(|V||E|^2) = \mathcal{O}((m+n)\cdot(mn)^2) = \mathcal{O}(m^3n^2 + m^2n^3)$.

Hence, the dominant term/bottleneck is in line 5, which implies that we have the overall runtime complexity to be at $\mathcal{O}(m^2n^2 \cdot (m+n))$.

Remark: Alternatively, notice that we can use the general time-complexity for Ford-Fulkerson, given by

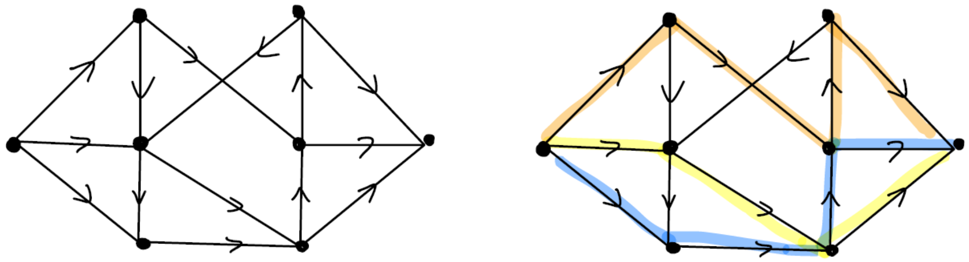
$$\Theta(|E|F).$$

Here, the maximum flow value is given by $\min\{m, n\}$. Together with the fact that $|E| = \mathcal{O}(nm)$, we have a time complexity of

$$\mathcal{O}(\min\{m, n\}nm).$$

Example 30. (Disjoint Paths in Directed Graphs.) Given a directed graph and two vertices in it, labeled as s and t, design an efficient algorithm to determine the maximum number of edge-disjoint paths from s to t. Two paths are edge disjoint if they don't share any edge.

Example:



In the graph above, there are a total of 3 edge disjoint paths. from the leftmost vertex to the rightmost vertex, each highlighted in a different color. Here, observe that the edges cannot be used more than once.

Suggested Solution (Sketch):

Set each edge on the graph to be 1 and run Ford-Fulkerson. The size of the flow is the number of edge-disjoint paths. This is because once the edge is traversed, it can no longer be traversed again; mapping this to a max-flow problem allows us to try multiple paths from the starting to the ending vertex.

Runtime: $\mathcal{O}(|V||E|^2)$ for Edmonds-Karp.

The key for proof of correctness lies in the proving the following lemma:

"There are k edge-disjoint paths in a directed graph from s to t if and only if the size of an $s-t$ flow in G is $\geq k$."

We then conclude by proving a max-flow/min-cut style theorem. For more information, see Section 7.6 of the textbook.

10 Discussion 10

Randomized Algorithms

By including randomness in algorithms, one has to now consider both the worst runtime complexity (which we have been considering) and the expected runtime capacity.

For example, for Quicksort as described in Week 9 notes, which picks a pivot element at random, it still has a worst runtime complexity of $\Theta(n^2)$, but the expected/average runtime is at $\Theta(n\log(n))$. In fact, in practice, this is often more efficient than MergeSort, but that only happens with high probability (and not certainly).

In analyzing expected runtime complexity, we recall the definition of expectation from probability as follows:

$$\mathbb{E}(X) = \sum_x x\mathbb{P}(X = x).$$

Furthermore, by the law of total expectation/towering property, we have

$$\mathbb{E}(X) = \sum_n \mathbb{E}(X|E_n)\mathbb{P}(E_n).$$

In other words, we are looking at the product of the value of the “random variable” X at x, and the probability that we get this x, and summing them over all possible x. For instance, for QuickSort, recall that the expected runtime complexity for an array of size n is given by

$$T(n) = \underbrace{n-1}_{\text{“Number of Comparsions to split the array”}} + \underbrace{\frac{1}{n}}_{\mathbb{P}(\text{Picking } i\text{-th biggest/smallest element as the pivot})} \sum_{i=1}^{n}(T(i) + T(n-i))$$

In the equation above, if we split at the i-th ranked element, then the conditional expectation term given in the law of total probability corresponds to the expected runtime given that we have two arrays of size i and size $n-i$ respectively. Following the arguments given in class, we can then show that

$$T(n) = \Theta(n\log(n)).$$

Hashing

For more information, see Week 9 notes. In summary, we have

- Using a hashing function/hash table, if there are no collisions, then under reasonable assumptions, the runtime complexity for a lookup is $\mathcal{O}(1)$.
- If there is a collision, use chaining.
- Under adversarial attacks, one can consider using a family of hashing functions (see Universal Hashing).

Example 31. Consider the following Las Vegas Algorithm below:

```
def LV_Find(A,a):
found ← 0;
while found == 0 do
    Select an element e from A randomly.
    If e is a, set found ← 1;
end
```

Furthermore, for this problem, we assume that A is of size n for some positive even integer n and n/2 of these elements are a.

(i) What is the worst-case runtime complexity of LV_Find(A,a)?

(ii) What is the expected runtime complexity of LV_Find(A,a)?

Hint: $\sum_{i=1}^{\infty} \frac{i}{2^i}$ converges to a positive finite number.

Suggested Solutions:

(i) $\Theta(\infty)$, since there is chance that the algorithm will never terminate.

(ii) Let $T(n)$ be the runtime complexity for LV_Find. Then,

$$T(n) = \sum_{i=1}^{\infty} i \times \mathbb{P}(\text{a found on the } i\text{-th iteration}) = \sum_{i=1}^{\infty} \frac{i}{2^i} = \Theta(1)$$

since the above series converges by the hint given in the question and is independent of n.

P & NP.

Let X and Y be decision problems.

Some key definitions/terminologies:

- **P** := Class of problems that can be **solved** in polynomial time.
(Rigorously speaking, with worst runtime complexity as $\Theta(p(n))$, where p is a polynomial with non-negative coefficients and n is the number of bits in the input.)

- **NP** := Class of problems that can be **verified** in polynomial time.
(Rigorously speaking, there is an algorithm `A` such that given a "yes" instance `x` to the problem X, we verify it using the algorithm A in polynomial time - ie $\Theta(p(|\mathtt{x}|))$.)

- **Reductions**. A reduction from X to Y is a polynomial time algorithm `A` such that for each instance of `x` in X, then `A(x)` is in Y and that `x` and `A(x)` have the same answer.

Using the notation in the textbook, we say that if there is a reduction from X to Y, then $X \leq_p Y$; ie X is **polynomial-reducible** to Y (ie Y is at least as hard to solve as X). Hence,

- If Y can be solved in polynomial time, then so can X.

- For a given target problem Y in NP, for **any** problems of interest X in NP such that there is a sequence of reductions $X \leq_p X_1 \leq_p \cdots \leq_p X_n \leq_p Y$ from X to Y, then we say that Y is **NP-complete**.

  Examples: 3SAT (true by a theorem by Cook and Levin), Independent Set problem (true since 3SAT $\leq_p$ Independent Set).

Recall that the 3SAT problem refers to a satisfiability problem of the form

$$(x_1 \vee x_2 \vee x_3) \wedge (x_4 \vee x_5 \vee x_6) \wedge .... \wedge (x_{n-2} \vee x_{n-1} \vee x_n)$$

with each x_i being a literal (ie either itself or the negation of itself, and "itself" here is a Boolean variable). The problem is then asking if we can determine if the formula can be satisfied by some values of $x_1, \cdots, x_n$.

Hence, intuitively,

- NP-complete ≈ "Non-polynomial" ≈ "Exponential" (ie 3SAT $\to \Theta(2^n)$ corresponding to checking all possible True/False values for each x_i, Subset Sum in the example below $\to \Theta(2^n)$, etc).

- Why are we particular about NP vs NP-complete? This is because some problems in NP are actually in P! This is because P $\subseteq$ NP (any problem in P is a problem in NP - any problem solvable in polynomial time can be verified in polynomial time).

An example of a reduction:

- Bipartite Matching $\leq_p$ Max-Flow. Recall that the algorithm `A` would be to draw source and sink nodes to the "applicants" and "jobs" respectively (see Example 1 of the discussion supplement last week).

- Since Max-Flow is in P (ie $\Theta(|V||E|^2)$, a polynomial function in its argument), then bipartite-maching is in P too.

Example 32. (Subset Sum.) Given N non-negative distinct integers $a_1, \cdots, a_N$ and a target sum K, the task is to design an algorithm to decide if there is a subset having a sum equal to K.

Show that this problem is in NP.

Suggested Solution: To prove that this is in NP, we basically have to check that a given solution is verifiable in polynomial time by an algorithm A.

1. Algorithm A: Take a subset $x_1, \cdots, x_n$ (of $\{a_1, \cdots, a_N\}$) and check if $\sum_{i=1}^{n} x_i = K$.
2. Given any "yes" instance given by $x_1, \cdots, x_n$ (ie a subset that sums to K), then we can check it with algorithm A. The runtime complexity of A is given by $\Theta(n) \in \mathcal{O}(N) = \mathcal{O}(p(N))$, and hence is a polynomial in N.

□

Example 33. (k-coloring.) Given a graph G, the task is to design an algorithm to determine if it is possible to perform a $k-$coloring on the graph G. Recall that a $k-$coloring of G is a function $f : V \to \{1, 2, \cdots, k\}$ such that for every edge (u, v). we have $f(u) \neq f(v)$ (colors of vertices connected by an edge cannot be the same).

(i) Prove that 2-coloring is in NP.

(ii) Prove that 3-coloring is in NP.

(iii) Which of the following reduction enables us to conclude that 3-coloring is NP-complete? Explain your answer.

R1: There is a reduction from 3-coloring to 3SAT.

R2: There is a reduction from 3SAT to 3-coloring.

Suggested Solution:

(i) Recall that we can perform a 2-coloring (bipartite) by running BFS with color propagation depending on the level of the resulting tree from BFS as in Week 2 notes. The BFS algorithm runs in $\Theta(|V| + |E|)$, a polynomial in its arguments. Hence, it is in P. Since P $\subseteq$ NP, then the 2-coloring problem is in NP.

(ii) To do so, one just has to verify that a graph G and a coloring function f, it is a valid $3-$coloring. Indeed, it suffices to traverse each vertex once to see if we have at most 3 colors, and each edge once to check if colors of vertices connected by an edge are not the same. Such an algorithm runs in $\mathcal{O}(|V| + |E|)$ (polynomial) time too. Hence, this is a problem in NP.

Remark: In fact, k-coloring is in NP for any positive integer k by utilizing the same exact argument.

(iii) A reduction from 3SAT to 3-coloring (R2).

Rigorously speaking, we have 3SAT $\leq_p$ 3-coloring. To show that 3-coloring is NP-complete, for any problems of interest NP, we do reductions from $X \leq_p \cdots \leq_p$ 3SAT, and use the fact that 3SAT $\leq_p$ 3-coloring to conclude that any X reduces to the 3-coloring problem. This thus shows that 3-coloring is NP-complete.

Remark: For the exact construction of this reduction, see section 8.7 of the textbook.

11 Final Exam Revision

Main Design Paradigms:

Graph Traversal.

- BFS/DFS; Runtime - $\Theta(|V| + |E|)$.
- Understanding how pseudocode works and how one can modify/apply it accordingly.
- Be able to run the algorithm by hand to produce a corresponding search tree from BFS/DFS. See Discussion 3.
- Other applications:
  - Getting distance from one node to another by accessing the resultant BFS search tree.
  - 2-coloring/bipartite graphs by color propagation. (HW 2 Problem 4.)
  - Cycle detection. (HW 2 Problem 3.)

Greedy Algorithms.

- Idea: Always pick the "locally" optimal solution.
- Paradigm: "Locally" optimal $=$ Globally optimal? Look into proof techniques/counterexamples.
- Proof Techniques:
  - Exchange Argument. (Week 3 notes - Minimize Maximum Lateness, HW 3 Problem 4 - Minimize weighted sum of finishing times.)
  - Staying Ahead. (Week 3 notes - Interval Scheduling, Discussion 4 Example 1 - Can Place Flower.)
  - Counterexample, especially if the greedy aspect of the algorithm does not directly optimize what we are looking for. (HW 4 Problem 1 - Coin Change Counterexample.)

Divide and Conquer.

- Computing worst runtime complexity for divide-and-conquer algorithms using the master theorem.

  Master Theorem: Suppose that $T : \mathbb{N} \to \mathbb{R}$ is a function (representing runtime complexity) satisfying
  $$T(n) = aT\left(\left\lceil \frac{n}{b} \right\rceil\right) + \mathcal{O}(n^d)$$
  for some constants $a > 0, b > 1, d \geq 0$, then
  $$T(n) = \begin{cases} \Theta(n^d) & \text{if } d > c \quad \text{i.e. "top-heavy"}, \\ \Theta(n^d \log(n)) & \text{if } d = c \quad \text{i.e. "balanced"}, \\ \Theta(n^{\log_b(a)}) & \text{if } d < c \quad \text{i.e. "bottom-heavy"}, \end{cases}$$
  for a critical exponent $c := \log_b(a)$.
- Example: MergeSort and understanding how it works. Runtime - $\Theta(n \log(n))$.
- Example: Binary search and understanding how it works. Runtime - $\Theta(\log(n))$. (HW 3 Problem 3 - Adjusting Microscope's Blurriness, Practice Midterm - Searching a Jumbled List.)

Dijsktra's Algorithm.

- Understanding how to run the algorithm using a PriorityQueue/MinHeap by hand. (See Discussion 5.)
- Understanding how to modify the algorithm accordingly and its impact. (See Practice Midterm - Unique Paths, Actual Midterm - TouristTrap.) In addition, also try to understand how to obtain the actual minimum path rather than just returning the minimum distance.
- Understanding how to apply the algorithm accordingly. This includes:
  - HW 4 Problem 2 - Minimize sum of distances while having two parties meet.
- Runtime: $\Theta((|V| + |E|) \log |V|)$.

Dynamic Programming. I will reuse the summary from Discussion 8 as follows.

Template: A `dp` array would be tabulated and fill by iteration, depending on the previous entries.

1D DP:

- Depends on $\underline{k}$ previous entries - `dp[i]` depends on `dp[i-k], dp[i-k+1], ..., dp[i-1]`.

  Runtime: $\mathcal{O}(nk)$ or $\mathcal{O}(n)$ if k is fixed beforehand (and not as an input to the function).

  Examples:

  - Fibonacci ($k = 2$, Week 6 Notes).
  - House Robber (Discussion 7 Example 1).
  - Maximum Subarray (Week 6 Notes).

- Depends on all previous entries - `dp[i]` depends on `dp[1], ..., dp[i-1]`.

  Runtime: $\mathcal{O}(n^2)$, since it requires $\mathcal{O}(i)$ step for the computation of the i-th entry of `dp`, and hence gives a triangular sum sort of time complexity (ie $\sum_{i=1}^{n} i = \frac{n(n+1)}{2} = \mathcal{O}(n^2)$).

  Examples:

  - Longest Increasing Subsequence a.k.a LIS (Week 6 Notes).
  - Russian Dolls (Discussion 8/This Discussion, Example 2 - Variation of LIS).
  - HW 6 Problems 1, 2, and 3.

2D DP:

- Has a general matrix structure given (ie input array is 2D, etc).

  Runtime: $\mathcal{O}(mnf(m,n))$, where m and n are the number of rows and columns of the given array, and $f(i,j)$ is usually the time it takes to compute `dp[i][j]`.

  Examples:

  - Unique Paths and Block Paths (Discussion 7 Example 2, HW 5 Question 4).
  - Fractional Knapsack/Selling a Rectangular Cloth (HW 5 Question 3).

- Two inputs (or associated properties) to the function, each of which are realized with a 1D array. The strategy is to “cross” the two inputs in a 2D `dp` array.

  Runtime: $\mathcal{O}(mnf(m,n))$, where m and n are the sizes of the two given array, and $f(i,j)$ is usually the time it takes to compute `dp[i][j]`.

  Examples:

  - 0-1 Knapsack (Week 7 Notes).
  - 0-∞ Knapsack Problems like Coin Change (HW 5 Question 1), and Coin Change II (Discussion 7 Example 3).
  - Edit Distance (Week 7 Notes).
  - Longest Common Subsequence a.k.a LCS (HW 5 Question 5/Discussion 8 Example 1).

- Usually, if `dp[i][j]` only depends on the values on the current row `dp[i][:j-1]` and its value on the same column but right before this index `dp[i-1][j]`, then we can cut down space complexity by only using a single 1D `dp` array (see Unique Paths and Coin Change II in Discussion 7).

Max Flow Problems.

- Understanding how to run the algorithm by hand. (See Discussion 8.)
- “Assumed Knowledge”: Implementing constraint on flow values on the vertices - HW 6 Problem 4.
- Understand how to modify the algorithm acccordingly and its impact. (See HW 6 Problem 4 - Adding constraints on nodes, HW 7 Problem 4 - Modifying the weight of a single edge.)
- Applications:

  - Bipartite Matching (Discussion 9 Example 1.)
  - Disjoint Paths in Directed Graph (Discussion 9 Example 2.)

- Max-Flow Min-Cut Theorem and applications. Recall that the maximum flow corresponds to the capacity of the cut obtained by a cut including the source s and all nodes reachable from s from the residual graph G. For an application of this, see HW 7 Problem 1.

Other Topics: (Discussion 10.)

- Randomized algorithms.
- Hashing and chaining.
- Given a problem, determine if it is in P, NP, and/or NP-complete. To determine if an algorithm is NP-complete, consider reductions to either the 3SAT or the independent set problem. (Discussion 10 Example 3 - k-coloring.)

Additional Problems.
Here are some additional problems that you can look at to help provide you with "intuition"/"experience" in solving problems from each of the design paradigms. For algorithm design problems, I will only include the runtime complexity, the sketch of the pseudocode, and sometimes the pseudocode itself to the problem itself.

Exercise 1. (Some True/False Problems.)

(i) To access the smallest k elements in an array of size n (where $1 \leq k < n$), the best way to do this is to sort the array first and then access the first k elements with a worst case runtime complexity of $\Theta(n\log(n) + k) = \Theta(n\log(n))$.

(ii) If a problem is in P, then it is in NP.

(iii) Consider the flow output obtained from the Ford-Fulkerson algorithm, and let A be the vertices reachable from the source node s in the final residual graph, and $B = V \setminus A$. Then, it is always true that the sum of the flow value along all the edges going from B to A is zero.

Exercise 2. ($\frac{3}{2}-$approximation of bipartite matching.) Consider a greedy algorithm for the bipartite matching problem between two sets of vertices V_1 and V_2 with the corresponding directed edges from $v \in V_1$ to $v \in V_2$. In other words, we loop through the set of nodes in V_1 for some v_1, and if there is an edge from $v_1 \in V_1$ to some $v_2 \in V_2$ and v_2 is not assigned yet, we then assign v_1 to v_2 (ie choose the directed edge (v_1, v_2)).

Recall that the bipartite matching problem can be solved using the Ford-Fulkerson algorithm by reducing this to a Max-Flow problem.

Prove/Disprove:

($\frac{3}{2}-$approximation of bipartite matching.) The number of matches obtained from the greedy algorithm is at least $\frac{2}{3}$ times of the optimal number of matches.

For the remaining exercises, you will be asked to design an efficient algorithm for a given problem. It suffices to only give a sketch of the algorithm, a pseudocode, and the runtime complexity of the algorithm. You do not need to prove that the algorithm is correct.

Exercise 3. (Search in a Rotated Array.) There is an integer array `nums` sorted in ascending order (with distinct values).

Prior to being passed to your function, `nums` is possibly rotated at an **unknown** pivot index `k (1 <= k < nums.length` such that the resulting array is `[nums[k], nums[k+1], ..., nums[n-1], nums[0], nums[1], ..., nums[k-1]]` (0-indexed). For example, `[0,1,2,4,5,6,7]` might be rotated at pivot index 3 and become `[4,5,6,7,0,1,2]`.

Given the array `nums` after the possible rotation and an integer `target`, return the index of `target` if it is in `nums`, or `-1` if it is not in `nums`.

Exercise 4. (Number of ways to jump.) Consider a frog on a one-dimensional pond with lotus leaves positioned uniformly apart, starting at the 1-st lotus leaf. The frog has the ability to jump up to k lotus leaves ahead, and is only able to jump forward.

Design a function `NumWays(n,k)` to efficiently compute the number of ways in which the frog can reach leaf number n given that it can jump up to k leaves ahead.

Examples:

Input: `n = 3, k = 2`
Output: `2`
Explanation: The frog starts from the leaf # 1, and can either jump to the next leaf twice (ie # 1 → # 2 → # 3, or jump directly to the target leaf (ie # 1 → # 3).

Input: `n = 4, k = 3`
Output: `4`
Explanation: The permutations are:

- $1 \rightarrow 4$,
- $1 \rightarrow 3 \rightarrow 4$,
- $1 \rightarrow 2 \rightarrow 4$, and
- $1 \rightarrow 2 \rightarrow 3 \rightarrow 4$.

Exercise 5. (Minimum Gene Mutation.) A gene string can be represented by an 8-character long string, with choices from `'A'`, `'C'`, `'G'`, and `'T'`.

Suppose we need to investigate a mutation from a gene string `startGene` to a gene string `endGene` where one mutation is defined as one single character changed in the gene string. For example, `"AACCGGTT" --> "AACCGGTA"` is one mutation.

There is also a gene bank `bank` that records all the valid gene mutations. A gene must be in `bank` to make it a valid gene string.

Given the two gene strings `startGene` and `endGene` and the gene bank bank, return the minimum number of mutations needed to mutate from `startGene` to `endGene`. If there is no such a mutation, return `-1`. Note that the starting point is assumed to be valid, so it might not be included in the bank.

Remark: When you are writing down the runtime complexity, it suffices to write it as a function of the `n`, the size of the gene bank `bank`.

Examples:

Input: `startGene = "AACCGGTT", endGene = "AAACGGTA",`
`bank = ["AACCGGTA","AACCGCTA","AAACGGTA"]`
Output: 2
Explanation: Do the following mutations: `"AACCGGTT" --> "AACCGGTA" --> "AAACGGTA"`. Note that we cannot do `"AACCGGTT" --> "AAACGGTT" --> "AAACGGTA"` since `"AAACGGTT"` is not in `bank`.

Input: `startGene = "AAAAAAAT", endGene = "TAAAAAAA",`
`bank = ["AAAAAATT", "AAAAAATA", "AAAAATTA", "AAAAATAA", ..., "TTAAAAAA", "TAAAAAAA"]`
Output: 14
Explanation: Even though the most direct way to perform the mutation is `"AAAAAAAT" -> "AAAAAAAA" -> "TAAAAAAA"`, the intermediate gene `"AAAAAAAA"` is not available in `bank`. We are then forced through a sequence of transformations with two `'T'`s, then the next with just one `'T'` but translated. Hence, we can move the `'T'` by one step after two mutations, resulting in a total of `14` mutations required to go from `startGene` to `endGene`.

Exercise 6. (When to buy/sell stocks.) You are given an integer array `prices` where `prices[i]` is the price of a given stock on the `i`th day, and an integer `k`.

Find the maximum profit you can achieve. You may complete at most `k` transactions: i.e. you may buy at most `k` times and sell at most `k` times.

Note: You may not engage in multiple transactions simultaneously (i.e., you must sell the stock before you buy again the next day).

Example:

Input: `k = 2, prices = [3,2,6,5,0,3]`
Output: 7
Explanation: Buy on day 2 (price = 2) and sell on day 3 (price = 6), profit = $6 - 2 = 4$. Then buy on day 5 (price = 0) and sell on day 6 (price = 3), profit = $3 - 0 = 3$. Sum of profits = $3 + 4 = 7$.

Suggested Solutions:

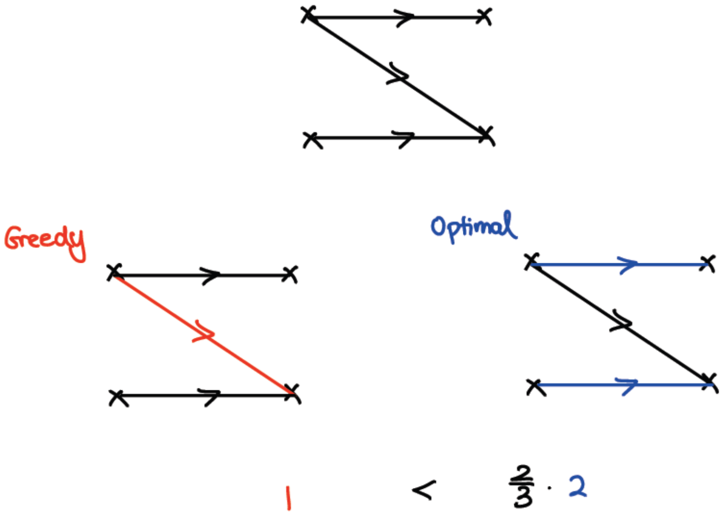
- Exercise 1:

  (i) False. Recall that initializing a `MinHeap` takes $\Theta(n)$ time. Performing k operations of `DeleteMin` only costs $\Theta(k\log(n))$, with a total worst-case runtime complexity of $\Theta(k\log(n)+n)$, which is better/faster than $\Theta(n\log(n))$ since $k<n$.

  (ii) True. If a problem can be solved in polynomial time, it can be verified in polynomial time. (Recall that P $\subseteq$ NP.)

  (iii) True. This is one of the arguments in the proof of Max-Flow Min-Cut Theorem. Suppose for a contradiction that one of the edges going from some node v_B in B to some node in v_A in A has a positive flow value. This implies that there is a directed edge going from v_A to v_B in the residual graph. v_B is now reachable, and should be in the set A, contradicting that $v_B \in B$.

- Exercise 2: False. See the counterexample below.



Remark: In fact, the correct approximation factor is $\frac{1}{2}$. Intuitively, the counterexample illustrates this, since a “greedy-ly” selected edge would block at most two “effective” pairs.

- Exercise 3: Use a modification of the the binary search. I'll illustrate the rough idea as follows.

  Consider `[4,5,6,7,0,1,2]`, we use the usual binary search idea as follows. Have a left and right pointer as per usual. If the middle value is the target, we can return the corresponding index. Else, observe that since the array was **already sorted** and then rotated, we can show that it is always the case that either the left portion of the rotated array (including the middle value) is sorted, or the right portion of the rotated array is sorted. (In our example, the middle pointer is at `7`, and we see that `[4,5,6,7]` is sorted (but `[7,0,1,2]` is not).)

  Hence, if it is sorted, we can check to see if `nums[left] <= target <= nums[middle]`. If it is, we then move the right pointer to the middle index. If not, we search on the other half of the array. Alternatively, if the right part of the array is sorted, then we check to see if `nums[middle] <= target <= nums[right]`; if it is, we search on the right half - else - we search on the left half. If the algorithm terminates and we have not returned anything, then we return `-1`, indicating that the element cannot be found in the array `nums`.

  Runtime complexity = $\Theta(\log(n))$, where $n=$ `length(nums)`.

  Source: https://leetcode.com/problems/search-in-rotated-sorted-array/.

- Exercise 4. This is actually equivalent to the Fibonacci sequence, but we are adding the last k elements before the current element (say i). Hence, the `dp` step is given by

  ```
  dp[i] = dp[i-1] + dp[i-2] + ...  + dp[i-k].
  ```

  Intuitively, the number of ways to reach leaf i is the sum of the number of ways to reach leaf $i-1$ (and then it jumps directly to i), leaf $i-2$ (and then it jumps directly to i; 2 leaves ahead), all the way up to leaf $i-k$ (and then it jumps directly to i; k leaves ahead).

  To deal with the base case, observe that `dp[1] = 0` (since we are starting on leaf 1, assuming indexing starting from 1). Then, for i from 2 up to k, we just sum up all the previous entries whenever possible. (For example, for $i = 4$ with $k = 6$, then we have `dp[4] = dp[3] + dp[2] + dp[1]`.)

  Runtime complexity: $\Theta(nk)$ - a total of n iterations in a single for loop, accessing up to k previous elements in a single iteration.

- Exercise 5. It suffices to run BFS on a graph of vertices as each of the strings in `bank`, together with the starting `startGene`. An edge is connected between two genes if it is one character away, and constructing the edges would take $\Theta(n)$ (to do this in $\Theta(n)$, we look at all possible single-character mutations for each position (ie a total of 3 other characters per position for a total of 8 positions, and check to see if the mutated gene is in `bank`). (This also implies that if we have a total of n vertices, then there are at most $24n$ edges.)

  We then run BFS to obtain the BFS search tree, and return the level in which `endGene` is at. If `endGene` cannot be found, we then return `-1`.

  Runtime Complexity: $\Theta(|V| + |E|) = \Theta(n + n) = \Theta(n)$.
  Source: https://leetcode.com/problems/minimum-genetic-mutation/.

- Exercise 6. This is similar to the DP problem in which it "depends on all previous entries". Here, we look at a 2D DP array, with `dp[t][i]` as the maximum profit if we have prices up to the `i`-th price and up to `t` transactions available. The `dp` step is given by

  ```
  dp[t][i] = max(dp[t][i-1], prices[i] - prices[j] + dp[t-1][j-1] for j = 0,..,i-1).
  ```

  The first term corresponds to what happens if we choose to not buy/sell, while the second term looks at all possible "dates" that we should be buying the stock to sell at time `i`. We then loop accordingly.

  Base Case: `dp[0][i] = 0` for all `i`, and `dp[t][0] = 0` for all `t` (for `0`-indexed `dp` arrays). We then return `dp[k][n]` for `n = length(prices)`.

  Runtime complexity: $\Theta(kn^2)$, since for each `t`, we run the max for a total of `i` times for each `i`, resulting in a triangular sum with time complexity $\Theta(n^2)$.

  Source: https://leetcode.com/problems/best-time-to-buy-and-sell-stock-iv/.

  There is a way to get the time complexity down in the actual pseudo-code by noticing that some terms are repeatedly computed. For more information, see https://leetcode.com/problems/best-time-to-buy-and-sell-stock-iii/solutions/135704/detail-explanation-of-dp-solution/.

References

[1] Jon Kleinberg and Eva Trados. *Algorithm Design*. Addison Wesley, 1st edition, 2005.