HW1

Due Date 11:59pm on Feb. 8, 2023. Submit to Gradescope.

1. Consider an instance I of the stable marriage problem with n men and n women. For a person (man or woman) x with preference list P_x define $P_x(1/2)$ to be the first $n/2$ persons (women/men) in the list P_x (to simplify notation we will assume that n is even). Suppose that we use Gale Shapley algorithm to find a stable matching. If a person (man or woman) x is paired off with another person $y \in P_x(1/2)$, the Gale Shapley matching is said to be *good for* x. Show that there is at least one person x such that Gale-Shapley matching is *good for* x. You may not make any assumptions about the preference lists.

   If we consider that a man proposes to each women in the man's first half, $P_m(1/2)$ but ends up with a $w \notin P_m(1/2)$; we can infer we that we have a net total of $> n * n/2$ rejections (a man makes a proposal and gets rejected because the woman's currennt partner is preferential to the proposer) and evictions (a man proposes to a woman that prefers him to her current partner) combined. More succinctly: $r + e > n * n/2$ where r is the number of rejections and e is the number of evictions. If instead we were to make $r + e \leq n * n/2$, then at least one man would've had a $w \in P_m(1/2)$ and we would be able to show that there is at least man x where GaleShapley is good for x.

   From the women's perspective, all women combined had to have performed $r + e > n * n/2$. And there are two reasons for a woman to perform such operations; she either replaces her current partner (and improves her preference) or she rejects a proposal (she has a better partner already). So, every time a woman is proposed to, she is guaranteed to either improve her partner preference or maintain it. To put it another way, at each proposal, her situation either improves or is maintained.

   Consider a scenario where the the $n * n/2$ rejections and evictions are not evenly distributed across all women. Therefore, there is at least one woman where she has performed $r + e > n/2$. In this case, the woman has had $> n/2$ increases in priority, solidifying her partner in the top half of her priority list, producing a matching that is *good* for at least one person x.

   Consider a more rigorous scenario, the $n * n/2$ rejections are evenly distributed across all women. In this case, each woman has performed $n/2$ rejections/evictions. Each woman has rejected/evicted her partners $n/2$ times. And each woman's final rejection/eviction guarantees that her partner is either already in $P_x(1/2)$ or by evicting and subsequently replacing her partner, she improves her partner into $P_x(1/2)$. This is because the shear number of proposals ensures that we graduate past the latter half of preferable partners into the first half of preferable partners for at least one person.

2. Assume you have functions f and g such that $f(n)$ is $O(g(n))$. For each of the following statements, decide whether you think it is true or false and give a proof or counterexample.

   (a) $\log_2 f(n)$ is $O(log_2 g(n))$

   (b) $2^{f(n)}$ is $O\left(2^{g(n)}\right)$

   (c) $f(n)^2$ is $O\left(g(n)^2\right)$

   The question is stating that we have two functions f and g such that $f(n)$ is $O(g(n))$. This means that there is a constant $c > 0$ and $n_0 \geq 0$ such that $0 \leq f(n) \leq c * g(n)$ for all $n \geq n_0$. And we are asked to determine if this holds for the statements above.

   Something that we should keep in mind is f can be a real or complex valued function while g is strictly a real valued function. Both are defined on an *unbounded subset of the positive real numbers* but only g is strictly positive. So f can be negative but we will only consider it's absolute value.

(a) $\log_2 f(n)$ is $O\left(log_2 g(n)\right)$

For this item, we need to ensure that the relationship $0 \leq \log_2 f(n) \leq c * O\left(log_2 g(n)\right)$ holds. We can use a counterexample here. Consider the functions $f(n) = b$ where $b > 1$ (b is any real-valued constant strictly greater than 1). In this case, $f(n)$ is $O(g(n))$ where $g(n) = 1$. Therefore, we can use some constant (ideally a value c greater than b to satisfy the following statement: $0 \leq f(n) \leq c * g(n)$). But when we perform the log, we find that $\log_2 f(n)$ evaluates to some constant while $\log_2 g(n)$ evaluates to 0. So no values of c nor n_0 will allow us to satisfy the statement. Therefore, this statement is false.

(b) $2^{f(n)}$ is $O\left(2^{g(n)}\right)$

For this item, we need to ensure that the relationship $0 \leq \log_2 2^{f(n)} \leq c * O\left(log_2 2^{g(n)}\right)$ holds. A counterexample we can use here is $f = 2log(n)$ and $g = log(n)$. To satisfy the definition of big O initially, we can just use $c > 2$ and the definition of big O holds for all values of n_0.

But when we evaluate these two functions to the 2nd power, we end up with:

$$2^{f(n)} = 2^{(2*log_2(n))} = n^2$$

$$2^{g(n)} = 2^{log_2(n)} = n$$

So regardless of the values of c and n_0 we chose, we will always fail the check for big O as $2^{f(n)} = n^2$ will outstrip $2^{g(n)} = n$ easily. Therefore, this statement is false.

(c) $f(n)^2$ is $O\left(g(n)^2\right)$

This statement is true. And we can use the product property of big O to reason out our answer. The product property states:

$$f_1 = O(g_1) and f_2 = O(g_2) \implies f_1 f_2 = O(g_1 g_2)$$

And because originally, $f(n)$ is $O(g(n))$. We know from the property above that $f(n)^2$ is $O(g(n)^2)$ holds as well.

3. In class we introduced big-O notation $f(n) \in O(g(n))$, $f(n) \in \Omega(g(n))$ and $f(n) \in \Theta(g(n))$. In computer science it is also common to use “little-o” notation. In particular, we say that $f(n) \in o(g(n))$ if for all positive constants $c > 0$ there exists a number N such that for all $n \geq N$ it holds that $f(n) < c \times g(n)$. Similarly, we say that $f(n) \in \omega(g(n))$ if for all positive constants $c > 0$ there exists a number N such that for all $n \geq N$ it holds that $f(n) > c \times g(n)$.
Changed the above statement from $f(n) \geq c \times g(n)$ **to** $f(n) > c \times g(n)$

Formally prove or disprove each of the following claims. You may assume that for all $n \geq 2^{2^{2019}}$ we have $f(n) \geq 1$ and $g(n) \geq 1$.

(a) $h(n) \in \Theta\left(\max\{f(n)^2, g(n)^2\}\right)$ where $h(n) = f(n) \times g(n)$

The definition of big theta is that for $h(n) \in \Theta\left(f(n)^2\right)$, then it must be both $h(n) \in \Omega(f(n))$ and $h(n) \in O(f(n))$. In this case I believe that I can provide an example that shows the former half to be false. Consider:

$$f(n) = a, g(n) = n$$

Where a is some positive real constant. Thus making:

$$h(n) = a * n$$

We can transform our original statement from:

$$h(n) \in \Theta\left(\max\{f(n)^2, g(n)^2\}\right)$$

to (without loss of generality, we can assume n ¿ a):

$$a * n \in \Theta\left(\max\{a^2, n^2\}\right)$$

$$a * n \in \Theta\left(n^2\right)$$

We can reinterpret this as:

$$a * n \in O\left(n^2\right) \ \& \ a * n \in \Omega\left(n^2\right)$$

But here, $a * n \in \Omega\left(n^2\right)$ is false because,

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} \neq \infty$$

$$\lim_{n\to\infty} \frac{a * n}{n^2} = 0 \neq \infty$$

Therefore, the original statement is false.

(b) $f(n) \in o(g(n))$ if and only if $g(n) \in \omega(f(n))$.

We can argue this by definition by showing how little omega implies little o and vice-versa. The definition of little o is:

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$$

And the definition of little omega is:

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = \infty$$

And we can see how if we take the inverse of either, we end up with the other. Both the following statements are true as they are part of the original statement:

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$$

$$\lim_{n\to\infty} \frac{g(n)}{f(n)} = \infty$$

And we know that

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0 \implies \lim_{n\to\infty} \frac{g(n)}{f(n)} = \infty$$

$$\lim_{n\to\infty} \frac{g(n)}{f(n)} = \infty \implies \lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$$

Because the reciprocal of infinity is 0 and the reciprocal of 0 is infinity. Therefore, the original statement is true.

(c) If $h(n) \in \Theta(\log n)$ where $h(n) = \ln f(n)$ then $f(n) \in O(n)$.

This statement is false. Consider $f(n) = n^2$. We would have $h(n) = \ln f(n) = 2 * \ln n$. $h(n) \in \Theta(\log n)$ holds true because $h(n) \in \Omega(\log n)$ and $h(n) \in O(\log n)$ hold. But, $f(n) = n^2$ and $f(n) \notin O(n)$.

(d) Suppose that $f(n) \in \omega(1)$ then $f(n) \in o\left(f(n)^2\right)$.

This is true. The only case where $f(n) \in o\left(f(n)^2\right)$ is false is in the case that $f(n) \in O(1)$. To put it another way, the only time when

$$\lim_{n\to\infty} \frac{f(n)}{f(n)^2} = 0 \ \& \ f(n) \in o\left(f(n)^2\right)$$

Is true is when f(n) and g(n) are constants. And by stating that $f(n) \in \omega(1)$, we guarantee that $f(n)$ is a function that is dependent on n, therefore making the following statement:

$$\lim_{n\to\infty} \frac{f(n)}{f(n)^2} = 0$$

and therefore, the original statement, true.

(e) If $f(n) \notin O(g(n))$ then $g(n) \in O(f(n))$.

This statement is false. Consider the functions $f(n) = (n^2) * (n \mod 2)$ and $g(n) = n$. In this example, both $f(n) \notin O(g(n))$ and $g(n) \notin O(f(n))$ hold.

(f) $f(n) \in o(g(n))$ implies that $h(n) \in O\big(2^{g(n)}\big)$ where $h(n) = 2^{f(n)}$.

The first part of this statement implies that the following relationship holds:

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$$

We also know that $h(n) = 2^{f(n)}$ and we are trying to see if the following is true:

$$\lim_{n\to\infty} \frac{h(n)}{2^{g(n)}} < \infty$$

Or to put it another way:

$$\lim_{n\to\infty} \frac{2^{f(n)}}{2^{g(n)}} < \infty$$

And we can apply the following:

$$\lim_{n\to\infty} 2^{f(n)-g(n)} < \infty$$

We know from the original statement that $g(n)$ is far larger than $f(n)$ so we know that we can simplify the above statement to:

$$\lim_{n\to\infty} 2^{-\infty} < \infty$$

$$\lim_{n\to\infty} 2^{-\infty} = 0 < \infty$$

So we can say that this statement is true.

(g) If $f(n) \in \Theta(h(n))$ and $g(n) \in o(h(n))$ then $f(n) \in \omega(g(n))$.

From the problem statement, we know that both

$$\lim_{n\to\infty} \frac{f(n)}{h(n)} < \infty$$

and

$$\lim_{n\to\infty} \frac{f(n)}{h(n)} > 0$$

are true because $f(n) \in \Theta(h(n))$. We also know that

$$\lim_{n\to\infty} \frac{g(n)}{h(n)} = 0$$

because $g(n) \in o(h(n))$. We are trying to show that because of the above statements,

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = \infty$$

To do so, we can invert the third statement to get:

$$\lim_{n\to\infty} \frac{h(n)}{g(n)} = \infty$$

And from the first two statements we know that the following must hold:

$$0 < \lim_{n\to\infty} \frac{f(n)}{h(n)} < \infty$$

And if were to multiply infinity by some positive value, we will end up within infinity making the following statement true:

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = \infty$$

To put it another way:

$$\lim_{n\to\infty} \frac{h(n)}{g(n)} * \lim_{n\to\infty} \frac{f(n)}{h(n)} = \lim_{n\to\infty} \frac{f(n)}{g(n)} = \infty * n$$

Where $0 < n < \infty$ which shows that the original statement is true as well. Because the prior statement is a logical extension of the original statement.

4. Suppose that an n-node undirected graph $G = (V, E)$ contains two nodes s and t such that the distance between s and t is strictly greater than $n/2$. Show that there exists some node $v \neq s, t$ such that deleting v from G destroys *all* paths from s to t. Give an algorithm running in time $O(n + m)$ to find such a node.

   Consider an arbitrary graph with nodes s and t. If we perform a BFS traversal from node s, we can generate a spanning tree for the graph G. This spanning tree will encompass all n nodes (including t) and we have at least $n/2$ layers within this tree. This is because the distance between s and t is strictly greater than $n/2$. As s is at the root of this graph and t exists at some distance $n/2$ along a branch of this tree.

   In this tree with n nodes, we have one node with distance $> n/2$ from the root. If we divide the size of the tree n by $n/2$, we find that we have (on average) about 2 nodes per layer. But note that in the 0th layer we have the node s. Therefore, in at least one layer there is exactly one node. Note that no layers will have 0 nodes (because otherwise the BFS tree would be disconnected and no path will exist between s and t). We can have layers with more than 2 nodes but that just means that we have multiple layers with just one node.

   The algorithm to find this node would be BFS but it terminates in the case that a layer is ever of size 1. If we represent this graph via an adjacency list, we will have a $O(n + m)$ runtime.

```
BFS (G, s)                    //Where G is the graph and s is the source node
  let Q be queue.
  for all neighbors of s:
        Q.enqueu(neighbors)

  size = Q.size
  mark all Q as visited.

  while ( Q is not empty)
       //Removing that vertex from queue,whose neighbour will be visited now
       v  =  Q.dequeue( )
       size--

       if ( size == 0 ):
            size = Q.size
            if ( size == 1 ):
                return Q.dequeue

      //processing all the neighbours of v
      for all neighbours w of v in Graph G
           if w is not visited
                    Q.enqueue( w )            //Stores w in Q to further visit its neighbour
                    mark w as visited.
```

This algorithm will iterate through all edges and all nodes giving us a total runtime of O(n+m). We can also use the fact that this is a trivial expansion of a traditional BFS algorithm to ensure that we have a O(n+m) runtime.

This algorithm returns the single node in the path because it checks when the queue size is 1 which is the same thing as when the BFS tree only has one node in a layer. Which, as indicated above, is the node that if removed destroys the s to t path.

5. We have a connected graph $G = (V, E)$, and a specific vertex $u \in V$. Suppose we compute a depth-first search tree rooted at u, and obtain a tree T that includes all nodes of G. Suppose we then compute a breadth-first search tree rooted at u, and obtain the same tree T. Prove that $G = T$. (In other words, if T is both a depth-first search tree and a breadth-first search tree rooted at u, then G cannot contain any edges that do not belong to T.)

   By definition, if we can deterministically generate the same tree via a BFS and a DFS traversal through a graph, then we are guaranteed to know that there is a unique path from the "specific vertex" $u \in V$ to every other vertex. So, when we generate this BFS/DFS tree, both algorithms only have a single path to pursue to each node. If there is a unique path to each node, then we have none of the following edges:

   (a) back edge
   (b) cross edge
   (c) descendent edge

   Because we have none of the above edges in the graph G we know that G is a tree of $|V| - 1$ edges, a sort of MST. We also know that for the BFS/DFS tree we encompass all the edges in a tree-like structure and take only $|V| - 1$ edges. Therefore, because the max number of edges is $|V| - 1$ and the number of edges total is $|V| - 1$, the BFS/DFS algorithms take all the edges in G because there are no other edges to take to traverse all the edges. Further, the edges G are the only edges available. Therefore BFSTree(G) = DFSTree(G) = G.

6. Show the following using induction. Consider the following graph $H_n = (V_n, E_n)$. Its vertex set is $V_n = \{0,1\}^n$ – that is every bitstring of length n is a vertex. A pair of vertices $u, v \in \{0,1\}^n$ is connected by an edge (i.e., $\{u, v\} \in E_n$) if u and v differ in exactly one bit. For example, if $n = 3$ then $u = (0,0,0)$ and $v = (1,0,0)$ would be connected by an edge, but v and $w = (1,1,1)$ were not connected by an edge. Use the principle of mathematical induction to show that the number of edges in the graph is $|E_n| = n2^{n-1}$.

   Proposition. For the graph H_n described above, $|E_n| = n * 2^{n-1}$ where E_n is equal to the number of edges in the graph.

   Proof. Let E_n be the statement that represents the number of edges in a graph H_n to be equal to $n * 2^{n-1}$. We give a proof by induction on n.

   Base case: We will show that the statement holds for the smallest natural number $n = 0$. Here I assume that the 0-length bitstring is a valid bitstring. As such, H_0 is the graph with a single node (with just the empty bitstring) and no edges. And the statement $|E_n| = n * 2^{n-1}$ holds as $n = 0$ and in this case $|E_n| = 0$.

   Inductive Step: Now I will show that for every $n \geq 0$, if $|E_n|$ holds, then $|E_{n+1}|$ holds as well. We can demonstrate this by thinking about what it means for us to go from $|H_n|$ to $|H_{n+1}|$. Every iteration on n means that we duplicate the current nodes that we have, prepend a 1 to the first set of nodes, a 0 to the second set of nodes, and then create pairs between edges of duplicate nodes. So when we consider how to create H_1 from H_0, we state that $H_0 = (,,)$ where we have exactly one node, the node of the empty string and no edges. But when we create H_1, we take the empty string "" and prepend both a 1 and a 0 to the string to give us two seperate nodes. Further, we can now pair the two nodes that are just slightly different from one another giving us: $H_1 = (1, 0, (1, 0))$. This method of constructing H_{n+1} can be used for all n. Now we can check the value of $|E_n|$ via this construction method.

$$E_n = n * 2^{n-1}$$

when going to n+1, we duplicate the current nodes (maintaining there edges)[$n * 2^{n-1}$] but also add n edges[2^n]. Because we are pairing nodes that had the same original bitstring but were prepended with either a 0 or 1. As such,

$$E_{n+1} = n * 2^{n-1} + n * 2^{n-1} + 2^n$$
$$E_{n+1} = 2(n * 2^{n-1}) + 2^n$$
$$E_{n+1} = n * 2^n + 2^n$$
$$E_{n+1} = (n+1) * 2^n$$

Which is equal to what we expect E_{n+1} to be:

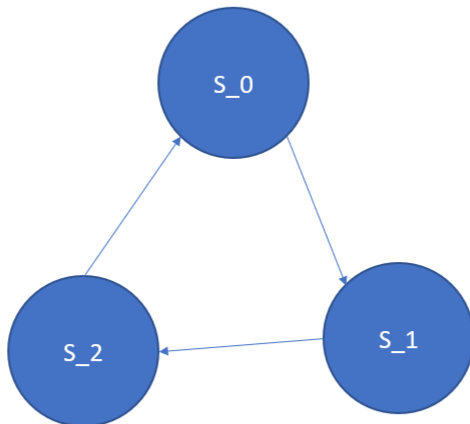
$$E_{n+1} = (n+1) * 2^{(n+1)-1}$$

$$E_{n+1} = (n+1) * 2^n$$

Conclusion: Since both the base case and the induction step have been proven true, by mathematical induction the statement $|E_n| = n * 2^{n-1}$ holds for all n.

7. n friends get together at a bar. It takes exactly 3 friends to participate in a cheer. Various (not necessarily all) triples of friends engage in cheers. For an individual x we use $\texttt{CheerCount}(x)$ to denote the number of times individual x participates in a cheer. For $i \in \{0, 1, 2\}$, let $S_i = \{x : i = \texttt{CheerCount}(x) \mod 3\}$ denote the set of people x that have cheered i times or $i+3$ times or $i+6$ times etc.. For example, if individual x participated in 8 cheers then $x \in S_2$ and if individual y participated in 12 cheers then $y \in S_0$. Show that the number $2|S_2| + |S_1|$ is a multiple of 3.

   We can address this problem by thinking about each of the sets S_0, S_1, S_2 is a state from which when you cheer, you progress to the next state.



   All we have to show is that the statement ($2|S_2| + |S_1|$ is a multiple of 3) holds at both the beginning (prior to any cheers) as well as after all possible transitions. This statement clearly holds at the beginning when both S_2 and S_1 are both empty and all the friends are in the set S_0 making the statement $2|S_2| + |S_1|$ evaluate to 0 and making it a multiple of 3. Now to inspect each of the possible transitions:

   (a) $\{S_0, S_0, S_0\}$ This would give us net change of $\{-3, +3, 0\}$. So we would be changing the initial value of $2|S_2| + |S_1|$ by +3 as we are increasing the number of friends in S_1 by three friends. Or to put it another way, the difference between before and after the transition would be $2 * 0 + 3$. Therefore, we would still be a multiple of 3.

   (b) $\{S_0, S_0, S_1\}$ This would give us a net change of $\{-2, +2-1, +1\}$ or $\{-2, +1, +1\}$. So we would be changing the initial value of $2|S_2| + |S_1|$ by +3. The difference between before and after the transition would be $2 * 1 + 1 = 3$. Therefore, $2|S_2| + |S_1|$ would still be a multiple of 3.

(c) $\{S_0, S_0, S_2\}$ This would give us a net change of $\{-2+1, +2, -1\}$ or $\{-1, +1, +1\}$. So we would be changing the initial value of $2|S_2| + |S_1|$ by +3. The difference between before and after the transition would be $2 * 1 + 1 = 3$. Therefore, $2|S_2| + |S_1|$ would still be a multiple of 3.

(d) $\{S_1, S_1, S_1\}$ This would give us a net change of $\{0, -3, +3\}$ or $\{0, -3, +3\}$. So we would be changing the initial value of $2|S_2| + |S_1|$ by +3. The difference between before and after the transition would be $2 * 3 + (-3) = 3$. Therefore, $2|S_2| + |S_1|$ would still be a multiple of 3.

(e) $\{S_1, S_1, S_0\}$ This would give us a net change of $\{-1, +1-2, +2\}$ or $\{-1, -1, +2\}$. So we would be changing the initial value of $2|S_2| + |S_1|$ by +3. The difference between before and after the transition would be $2 * 2 + (-1) = 3$. Therefore, $2|S_2| + |S_1|$ would still be a multiple of 3.

(f) $\{S_1, S_1, S_2\}$ This would give us a net change of $\{+1, -2, +2-1\}$ or $\{+1, -2, +1\}$. So we would be changing the initial value of $2|S_2| + |S_1|$ by 0. The difference between before and after the transition would be $2 * +1 + -2 = 0$. Therefore, $2|S_2| + |S_1|$ would still be a multiple of 3.

(g) $\{S_2, S_2, S_2\}$ This would give us a net change of $\{+3, 0, -3\}$ or $\{+3, 0, -3\}$. So we would be changing the initial value of $2|S_2| + |S_1|$ by -6. The difference between before and after the transition would be $2 * -3 + 0 = -6$. Therefore, $2|S_2| + |S_1|$ would still be a multiple of 3.

(h) $\{S_2, S_2, S_0\}$ This would give us a net change of $\{+2-1, +1, -2\}$ or $\{+1, +1, -2\}$. So we would be changing the initial value of $2|S_2| + |S_1|$ by -3. The difference between before and after the transition would be $2 * -2 + 1 = -3$. Therefore, $2|S_2| + |S_1|$ would still be a multiple of 3.

(i) $\{S_2, S_2, S_1\}$ This would give us a net change of $\{+2, -1, +1-2\}$ or $\{+2, -1, -1\}$. So we would be changing the initial value of $2|S_2| + |S_1|$ by -3. The difference between before and after the transition would be $2 * -1 + -1 = -3$. Therefore, $2|S_2| + |S_1|$ would still be a multiple of 3.

Note that in the above notation, the inclusion of a value indicates that we are "kicking" a friend out of the given set and into the next set. So S_2 indicates that we lose a friend in S_2 and gain a friend in S_0. The notation to denote this will have +/- a number in one of the three slots. So a transition of $\{S_0, S_0, S_0\}$ would appear as $\{-3, +3, 0\}$

HW2

Due Date 11:59pm on Feb. 22, 2023.

Question 1 (Greedy Gambling)

The local casino is opening up a new game involving n treasure chests $1, \ldots, n$ in a private room. Before a new gambler enters the room all treasure chests are emptied and then single gold bar of value V is *randomly* placed into one of treasure chests, while a worthless rock of equal weight is placed into the other treasure chests. The probability that the gold bar is placed in treasure chest i is given by p_i. Each treasure chest i is labeled with the probability p_i and a cost c_i indicating that the gambler may pay the key master c_i to obtain the key to open treasure chest i. If the treasure chest contains the gold bar then the gambler may keep it, otherwise the gambler may choose to exit the room (quit) or buy another key.

A strategy consists of sequence of intended actions $a_1, \ldots, a_n$ where each action $a \in [n]$ is of the form e.g., buy the key for chest a and open it. All of the a_i values are assumed to be distinct since it does not make sense to buy the same key twice. If box a_i ($i \leq n$) contains the gold bar then the total gain (or loss) is given by $V - \sum_{j=1}^{i} c_{a_j}$ i.e. since the gambler does not need to pay to open boxes $a_{i+1}, \ldots, a_k$. Thus, the expected gain is given by

$$\mathbb{E}\left[\texttt{Gain}\left(a_1 \ldots \hat{a}, a_n\right)\right] = V - \sum_{i=1}^{n} p_{a_i} \left(\sum_{j=1}^{i} c_{a_j} \right)$$

Note that the gambler may also play the special strategy `QUIT` (never open any box) which always has gain 0. However, if the gambler opens one treasure chest then the gambler must continue paying to open treasure chests until the gold bar is found. The task of the greedy gambler is to find a strategy which maximizes the expected gain.

The expected value is equal to the value of the gold bar minus the expected cost to open all boxes before getting to the correct box. In this case, the cost of getting to the correct box is the sum of the probability of each box times the sum of the costs of the boxes prior to it (including its own cost). Basically, it conditions the the probabilities of later boxes on the costs of needing to unlock all prior boxes. In this case, we should open boxes in order of decreasing p/c.

1. Suppose the gambler sorts the treasure chests such that $p_1 \geq p_2 \geq \ldots \geq p_n$, and then outputs the strategy $1, \ldots, n$ if and only if $\mathbb{E}\left[\texttt{Gain}\left(1, \ldots, n\right)\right] \geq 0$ — otherwise we output the strategy `QUIT`. Provide a short counter-example showing that the gambler might miss the optimal strategy.

   The easiest way to generate a counter example is to solve for a scenario where the reverse order changes the EV significantly. Take for example the following boxes:

   (a) box1: cost = 4, probability = 0.4

   (b) box2: cost = 7, probability = 0.6

   Assuming that the value of the gold bar is 8.4 and we were to take the suggested ordering (highest probability first), we would have a expected cost term to be equal to:

   $$0.6 * 7 + 0.4(4 + 7) = 4.2 + 4.4 = 8.6$$

   But in the alternate ordering, we would have an expected cost of:

$$0.4 * 4 + 0.6(4 + 7) = 1.6 + 6.6 = 8.2$$

And we can find that the latter ordering minimizes cost. So if we were to have a bar of Value 8.4, the former ordering will tell the player to not open any boxes giving an EV of 0. But the latter ordering will tell the player to open boxes in a different order and given an EV of 0.2 demonstrating that ordering box opening by decreasing probability is not the optimal strategy.

2. Suppose the gambler sorts the treasure chests such that $c_n \geq c_{n-1} \geq \ldots \geq c_1$, and then outputs the strategy $1, \ldots, n$ if and only if $\mathbb{E}[\texttt{Gain}(1, \ldots, n)] \geq 0$ — otherwise the gambler output the strategy `QUIT`. Provide a short counter-example showing that the gambler might miss the optimal strategy.

   We will be following a method similar to the prior problem. Take for example the following boxes:

   (a) box1: cost = 4, probability = 0.2

   (b) box2: cost = 7, probability = 0.8

   Assuming that the value of the bar is 8.4, if we were to take the suggested ordering (lowest cost first), we would have a expected cost term to be equal to:

$$0.2 * 4 + 0.8(4 + 7) = 0.8 + 8.8 = 9.6$$

   But in the alternate ordering, we would have an expected cost of:

$$0.8 * 7 + 0.2(4 + 7) = 5.6 + 2.2 = 7.9$$

   And we can find that the latter ordering minimizes cost. So if we were to have a bar of Value 8.4, the former ordering will tell the player to not open any boxes giving an EV of 0. But the latter ordering will tell the player to open boxes in a different order and given an EV of 0.5 demonstrating that ordering the boxes by increasing cost is not the optimal strategy.

3. Provide a greedy algorithm to find the optimal strategy and analyze the performance of your algorithm. You should prove that your algorithm is correct.

   The optimal ordering is the order in decreasing $\frac{p_i}{c_i}$. Note that this ordering is optimal even in the case of boxes with identical ratios. The algorithm to generate such an ordering would first need to find the ratio of price to cost which takes $\Theta(n)$ time and the time it takes to sort/order the ratios is $O(nlog(n))$ therefore, the total runtime of this algorithm would be $O(nlog(n))$.

   I will proceed by contradiction. Assume that the reverse order is more optimal, ie the order of increasing ratios $(\frac{p_i}{c_i})$ produces a more optimal solution than the decreasing order. In the case that two ratios are identical in value, the we don't care about how they are ordering amongst themselves as long as the ordering overall is preserved. So $a < b <= c < d$ and $a < c <= b < d$ are both valid orderings.

   The definition of an optimal ordering is that swapping two distinct ratio-valued boxes will only serve to decrease the EV of the overall ordering and cannot increase it. Consider we have two boxes, one with a larger ratio that is ordered earlier r_l and one with a smaller ratio that is ordered later r_s. I propose that the ordering $Ordering1 = r_1, r_2, ...r_s, r_l, ..., r_n$ has a lower EV than the ordering $Ordering2 = r_1, r_2, ...r_l, r_s, ..., r_n$, suggesting a contradiction. Let us now evaluate the values of both orderings

$$E[Ordering1] = V - (p_1(c_1) + p_2(c_1 + c_2).... + p_s(...c_s) + p_l(....c_s + c_l)...$$

)

$$E[Ordering2] = V - (p_1(c_1) + p_2(c_1 + c_2).... + p_l(...c_l) + p_s(....c_l + c_s)...$$

)

$$E[Ordering2] - E[Ordering1] = -(p_s * c_l) + p_l * c_s$$

If the right side is greater than 0, then we know that the swapped order is better than the original order:

$$0 < -(p_s * c_l) + p_l * c_s$$

$$p_s * c_l < p_l * c_s$$

$$p_s * c_s < p_l * c_l$$

Note that we know that $r_l > r_s$. Which holds true here and shows that Ordering2, a non optimal ordering is better than the "optimal" ordering of sorting by ascending ratio order suggesting a contradiction. Therefore, the optimal ordering is sorting by descending ratio.

Question 2

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

Suppose that the given graph G has two distinct minimum spanning trees $T1$ and $T2$. Let us select an edge that is the minimum edge of all edges that are only in either $T1$ or $T2$. So $e \in T1 \cup T2 - (T1 \cap T2)$ that is the minimum of all e. We can say that this edge ($e1$) is contained only in $T1$. But if were to add $e1$ to $T2$ we would find a cycle because $T2$ is currently a MST. And if we were to restore $T2$'s MST status we would have to remove an edge within the cycle that is not the edge $e1$, the edge that we just added, because $e1$ has a lower value than *all edges* contained in $T1$ and $T2$. So we must remove an edge $e2$ that is part of the cycle. This edge could not have been in $T1$. This is because if it was part of $T1$, then $T1$ would've had a cycle.

Because the edge that we have to remove, $e2$, is not in $T1$ but is in $T2$, it must have originally been in the set that we selected $e1$ from $T1 \cup T2 - (T1 \cap T2)$. So this edge that we must remove from $T2$, $e2$ is a higher weight edge that $e1$. Therefore, we find that $T2$ isn't actually a MST and we find a contradiction. Because this contradiction originated from assuming that we have two unique MSTs (for a graph with distinct edge costs) we realize that we are wrong and therefore can only have one MST (for a graph with distinct edge costs).

Question 3

This year, the Virginia International Raceway will be organizing a "doubles" motor race from Alton, Virginia to Washington D.C. Each team will consist of two drivers, who must alternate driving from city to city (that is, they must switch once they reached the next city, and they do not switch along the way). Furthermore, the person who starts the race must finish the race. Because the speed limits vary from city to city, the CS 6161 staff has assembled a list of n cities and m times that it takes to travel between some pairs of cities (not necessarily in both directions). Note that if there is no given time between a pair of cities, either there are no routes between these cities, or the routes are in terrible condition and it's not advisable to use them during the race. Devise an $O((m + n)logn)$ algorithm to help the CS 6161 team win the race.

We have two conditions to satisfy. The first is that the same driver has to end the journey as they start. Therefore, the path must be of odd length. The other condition is that we need to get the shortest path in time. But it is complicated to apply the dual condition of an odd length in edges while minimizing total edge weight along the path.

Massaging this problem's input into a slightly different format can allow us to simultaneously solve both problems. So what we need to do is duplicate each node. One copy is the odd version of the node while the other is the even version of the node. We duplicate each edge of the original graph between the odd of the source/even of the dst and the even of the source/odd of the dest. Then we can just find the path between the even copy of Alton and the odd copy of DC.

```
findPath(directedGraph Graph, node Source, node Dest):
    dist[even(Source)] = 0

    vertexPQ Q
    directedGraph oddEvenGraph

    for vertex v in Graph.Vertices
        oddEvenGraph.Vertices.append(odd(v))
        oddEvenGraph.Vertices.append(even(v))

    for edge e in Graph.Edges:
        oddEvenGraph.Edges.append(edge(odd(e.source), even(e.dest), e.weight)
        oddEvenGraph.Edges.append(edge(even(e.source), odd(e.dest), e.weight)

    for vertex v in oddEvenGraph.Vertices
        if v != even(Source):
            dist[v] = infin
            pred[v] = NULL

        Q.add(v, infin)

    dist[even(source)] = 0
    pred[even(source)] = NULL
    Q.add(even(source), 0)

    while (!Q.empty):
        u = Q.top()
        Q.pop()

        for each neighbor v of u:
            alt = dist[u] + oddEvenGraph.Edges(u, v).weight
            if alt < dist[v]:
                dist[v] = alt
                prev[v] = u
                Q.decrease_priority(v, alt)


    // we'll probably want to return the path

    retList = []

    node tempNode = odd(Dest)

    while tempNode != even(Source):
        retList.append(tempNode)
        tempNode = prev[tempNode]

    return retList
```

The first for loop (over the input nodes) is of size $\Theta(n)$, the second loop (over the edges) is of size $\Theta(m)$, the third loop (over the vertices) is of size $\Theta(n)$ again. Now we get to the priority queue part. We effectively loop over every edge in the graph (or every node in the graph) and in the case that we have to update its distance, we perform a log(n) operation for a total runtime of O(max(m, n)log(n)). This brings our overall

runtime to O(n + max(m, n)log(n)). And both O(n + mlog(n)) and O(n + nlog(n)) are upperbouned by O(mlog(n) + nlog(n)). Therefore, we can see that this algorithm both solves the problem and runs in $O((m+n)logn)$ time.

Question 4

The wildly popular Spanish-language search engine El Goog needs to do a serious amount of computation every time it recompiles its index. Fortunately, the company has at its disposal a single large supercomputer, together with an essentially unlimited supply of high-end PCs. They've broken the overall computation into n distinct jobs, labeled J_1, J_2, ..., J_n, which can be performed completely independently of one another. Each job consists of two stages: first it needs to be preprocessed on the supercomputer, and then it needs to be finished on one of the PCs. Let's say that job J_i needs p_i seconds of time on the supercomputer, followed by f_i seconds of time on a PC.

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

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

Give a polynomial-time algorithm that finds a schedule with as small a completion time as possible.

We have two things to consider when we want to order our jobs. The first is the time it takes on the supercomputer p_i and the time the job needs on the normal computers f_i. Note that we want to minimize the total time that all the jobs take. To that end, we have to note that we are strictly lower bounded (total time taken) by the sum of all p_i because every job has to be processed by the supercomputer.

Each of these scenarios represents a pair of jobs as tuples. The tuple's first element is the supercomputer time and the second element is the desktop time. For each scenario lets determine the optimal ordering.

1. (2, 5)(5, 2) – (2, 5) first
2. (2, 2)(5, 5) – (5, 5) first
3. (3, 3)(5, 2) – (3, 3) first

Interesting. We seem to pick in order of decreasing desktop computer time. Lets repeat this experiment with a constraint, the fact that we only have a single pc $n = 1$. But lets increase the total number of jobs we have to select so that we are saturating the computers that we have.

1. (2, 5)(5, 2)(3, 3) – (2, 5) first
2. (2, 2)(5, 5)(3, 3) – (5, 5) first
3. (3, 3)(5, 2)(5, 3) – (3, 3) first

So we pick in order of decreasing PC time and when PC time is tied we pick in order of decreasing Supercomputer time.

This makes sense because our lowest possible time is the sum of all the time on the supercomputer + the remaining time of the longest job on the PCs. So if we select the longest time Desktop/PC jobs first, we are able to minimize the remaining time of the longest job on the PCs.

Algorithm:

```
getOptimalOrdering(arrayOfTuples):
    if ( size(arrayOfTuples) == 1 ):
        return arrayOfTuples
```

```
    mid = size(arrayOfTuples) / 2
    arrayOfTuples = merge(getOptimalOrdering(arrayOfTuples[:mid]),
    getOptimalOrdering(arrayOfTuples[mid:]))

    return arrayOfTuples


merge(arrayOfTuples1, arrayOfTuples2):
    ind1 = 0
    ind2 = 0
    ret = []

    while ( ind1 < size(arrayOfTuples1) and ind2 < size(arrayOfTuples2) ):
        if ( arrayOfTuples1[ind1][1] > arrayOfTuples2[ind2][1] ):
            ret.append(arrayOfTuples1[ind1])
            ind1++
        else if ( arrayOfTuples1[ind1][1] > arrayOfTuples2[ind2][1] ):
            ret.append(arrayOfTuples2[ind2])
            ind2++
        else:
            if ( arrayOfTuples1[ind1][0] > arrayOfTuples2[ind2][0] ):
                ret.append(arrayOfTuples1[ind1])
                ind1++
            else :
                ret.append(arrayOfTuples2[ind2])
                ind2++

    if ( ind1 < size(arrayOfTuples1) ):
        ret.append(arrayOfTuples1[ind1:])
    else:
        ret.append(arrayOfTuples2[ind2:]

    return ret
```

This runs in polynomial time because this algorithm is a variation of mergesort. And it is known that mergesort is O(n log n). But I can show that my algorithm is O(n log n) as well here. So at every "step" we split the input into two and then when recombining we do n total work on it. And we keep splitting until we have an array of size 1 so we effectively have log(n) layers of splits. And at each layer we do n work when merging. So our run time is O(n log n) which is polynomial time.

Question 5

To get started on this problem you need to have reviewed Quicksort. Suppose you are given a database containing n people in alphabetical order, and their respective incomes (you may assume the incomes are all different). Someone promises you a large sum of money if you can answer who is the person exactly at the pth percentile (for an arbitrary given value of p), using a linear in n number of comparisons between incomes. Can you get the money?

I believe that it is possible to get the money in this scenario because Quickselect with a pivot selection that runs in linear time is indeed a linear time algorithm. I will now detail and analyze an algorithm to do just that.

```
    wrapper(input, percentile):
        return quickselect(input, percentile * len(input))
```

```
quickselect(input, k):
    medians = []

    for i in range(5, len(input), 5):
        medians.append(getMedian(input[i-5:i]))

    medianOfMedians = quickSelect(medians, 0.5)

    flag = true
    leftSide = []
    rightSide = []
    for i in input:
        if ( i == medianOfMedians and flag ):
            flag = false
            continue

        if ( i > medianOfMedians ):
            rightSide.append(i)
        else:
            leftSide.append(i)

    if len(leftSide) == k:
        return medianOfMedians
    else if ( len(leftSide) > k ):
        return quickselect(leftSide, k)
    else:
        return quickselect(rightSide, k - len(leftSide))
```

Here I assume that the function getMedian runs in constant time on five elements. And note that we recursively call quickselect to get the median of the median of medians. But lets ignore that for now. For this algorithm, we perform n comparisons at every level. But at each level we are doing a fraction of the comparisons of the prior level. This is because via the medianOfMedians algorithm, we are guaranteed to eliminate a fractional value of the total search space at every step of this function. Because the median of medians algorithm is guaranteed to select a value that is in the middle 40% of values we can ensure that our quickselect algorithm diminishes our search space fractionally. And because our recursion is of the form:

$$T(n) = \begin{cases} \Theta(1) & k = 1 \\ T(7/10n) + T(3/10n) + \Theta(n) & otherwise \end{cases} \tag{1}$$

The 7/10 part is the overall work we do on all the elements, the 2/10 part is the median of medians selection, and the $\Theta(n)$ is just stating that we do linear work while partitioning. We can now show that this is linear time via substitution. Lets try substituting cn for $T(n)$:

$$\begin{aligned} T(n) &\leq c * 7/10n + c * 3/10 + \Theta(n) \\ &\leq c * n + \Theta(n) \\ &\leq c * n \end{aligned}$$

Therefore, we can show that this algorithm is indeed a O(n) algorithm as T(n) is indeed upperbounded by a linear form equation cn satisfying the definition of a O(n) algorithm.

HW3

Due Date 11:59pm on Mar. 8, 2023. Submit to Gradescope.

1. In class we talked about the problem of finding the number of inversions in an array. In this problem the input is a sequence of n distinct numbers $a_1, \ldots a_n$ and we defined an inversion to be a pair $i < j$ such that $a_i > a_j$. Suppose instead that we are given an n by n matrix $a_{1,1}, \ldots a_{n,n}$ of n^2 numbers. We say that the pair of points $p_1 = (i_1, j_1)$ and $p_2 = (i_2, j_2)$ are inverted if $i_1 < i_2$, $j_1 < j_2$ AND $a_{i_1,j_1} > a_{i_2,j_2}$. Give an $O(n^2 log^3 n)$ time algorithm to count the number of inversions in the n by n matrix. You should prove that your algorithm is correct and analyze its running time.

   Somehow, we need to sort the matrix via diagonals or sort the matrix by squares (originating from the top left). This would make it easiest to find pairs of locations where the inversion conditions would be satisfied.

i_1, j_1	i_2, j_1	i_3, j_1	
i_1, j_2	i_2, j_2	i_3, j_2	
i_1, j_3	i_2, j_3	i_3, j_3	
			i_4, j_4

   For i_4, j_4, we would care about all the values in the red. For each point, we care about all the cells above and to the left of the point. A naive method of computing the number of inversions is to store all the values in the red cells, sort them, and then binary search the yellow value to find out where to insert the yellow value. If we were to perform this for each n^2 number, we gather n^2 elements, perform a $nlog(n)$ sort operation, and then perform a $log(n)$ binary search to find out where to insert the element. This turns out to be a $(n^2) * (n^2 + n^2 log(n^2) + log(n^2))$ algorithm. Which is a $O(n^4 log(n^2))$ algorithm overall which is worse than the $O(n^2 log^3 n)$ limit that was prescribed for this problem. Therefore, we must employ some craftiness in solving this problem.

   I found a interesting data structure that I think can help us here: https://www.geeksforgeeks.org/binary-indexed-tree-or-fenwick-tree-2/. By using a fenwick tree, we can easily complete this problem. A Fenwick tree is traditionally used to sum up numbers which can be transformed to count inversions in a 1D array. It is particularly special because we can both update values of the tree and query calculations in $\Theta(log(n))$. If we slightly modify a fenwick tree to adapt to two dimensions, we can take advantage of this speed and be able to stay within the speed limit prescribed by this problem.

   A Fenwick tree functions as such, each node only keeps track of the number of values "in it" less the number of values in the parent node. So when we are doing an update operation, we only need to update values at some boundary value within each tree, taking $\Theta(log(n))$ time. And when we are counting values that satisfy a condition, we only need to add "up" the tree taking $\Theta(log(n))$ time as well. So what we can do is sort the values in descending order. And for each value, we take its corresponding coordinates in the matrix, feed it to the Fenwick tree and ask the tree "sum up the values that are relevant to this value". The Fenwick tree iterates on the tree branch this value belongs to and adds up all the values and gets the number of inversions (we are going in descending order so larger values would have been marked first). To mark values, we tell the Fenwick tree to "add 1 to

all the values that correspond to these coordinates". So that when we query the tree and add up the values, we are counting all the values that are larger than the current value. Note that for duplicate values we perform all the queries at once and then we do all the updates this prevents conflicts for duplicates.

```
#define N 4

void update(int l, int r, int val, int bit[][N + 1])
{
    for (int i = l; i <= N; i += i & -i) {
        for (int j = r; j <= N; j += j & -j){
            bit[i][j] += val;

}

// function to find cumulative sum upto
// index (l, r) in the 2D BIT
long long query(int l, int r, int bit[][N + 1])
{
    long long ret = 0;
    for (int i = l; i > 0; i -= i & -i)
        for (int j = r; j > 0; j -= j & -j)
            ret += bit[i][j];

    return ret;
}

// function to count and return the number
// of inversion pairs in the matrix
long long countInversionPairs(int mat[][N])
{
    // the 2D bit array and initialize it with 0.
    int bit[N+1][N+1] = {0};

    // v will store the tuple (-mat[i][j], i, j)
    vector<pair<int, pair<int, int> > > v;

    // store the tuples in the vector v
    for (int i = 0; i < N; ++i)
        for (int j = 0; j < N; ++j)
            v.push_back(make_pair(-mat[i][j], make_pair(i+1, j+1)));

    sort(v.begin(), v.end()); // sorting by the value in the matrix at the point
    long long inv_pair_cnt = 0;
    int i = 0;
    vector<pair<int, int> > pairs;
    vector<pair<int, int> >::iterator it;
    int curr;

    while (i < v.size()) {
        pairs.clear();
        int curr = i;

        while (curr < v.size() && (v[curr].first == v[i].first)) {
```

```
            // collecting all duplicate values
            pairs.push_back(make_pair(v[curr].second.first, v[curr].second.second));
            inv_pair_cnt += query(v[curr].second.first - 1, v[curr].second.second - 1, bit);
            curr++;
        }

        for (it = pairs.begin(); it != pairs.end(); ++it)
        {
            int x = it->first;
            int y = it->second;
            update(x, y, 1, bit);
        }

        i = curr;

    }

    return inv_pair_cnt;

}

int main()
{
    int mat[N][N] = { { 4, 7, 2, 9 },
                      { 6, 4, 1, 7 },
                      { 5, 3, 8, 1 },
                      { 3, 2, 5, 6 } };

    long long inv_pair_cnt = countInversionPairs(mat);
    cout << "The number of inversion pairs are : "<< inv_pair_cnt << endl;
    return 0;
}
```

Note that this code is copied from https://www.geeksforgeeks.org/count-inversion-pairs-matrix/. But I do understand it and this would be similar to my own implementation if I were to make one. It has some efficiency modifications as well as one key modification that makes this work. Instead of querying for larger values at the value i, j, we instead query at $i - 1, j - 1$. Doing so maintains the $i_1 < i_2$ condition in this problem.

If we look at the code, we can see that we iterate over each element, this is n^2 time. And for each of these elements we perform one update operation and one query operation both of which are $\Theta(log(n^2))$-$>\Theta(log(n))$ time each. Overall our runtime is $O(n^2 log(n))$.

In terms of a proof of correctness, if we consider the Fenwick tree as a datastructure for counting elements, we only have to show that the update/query operations work so that they don't double count an inversion and also don't miss any inversions. Lets take for example the pair of inversions [4,4]: 6 and [3,3]: 8. Because 8 is larger than 6, the Fenwick tree nodes of [3, 3], [3, 4], [4, 3], and [4, 4] will have the value of 1 added to them. So, when we query on the element 6, (coords [3, 3]). We take the following path in the Fenwick tree: [3, 3] -¿ [3, 2] -¿ [3, 0] and [2, 3] -¿ [2, 2] -¿ [2, 0]. And in this set, we only see the updated node once so we only count one inversion. This shows how we don't miss or double count inversions. This relationship holds for all inversions and is the basis of a proof by induction.

2. Solve the recurrences below by any method you know (but do not use Master's theorem, except to double check your answer, if possible). You may use any base cases you find reasonable.

(a) $T(n) = 16T(\frac{n}{2}) + \sqrt{n}$

We can solve this by unrolling the recurrence; substituting into itself:

$$\begin{aligned} T(n) &= 16T(\frac{n}{2}) + \sqrt{n} \\ &= 16(16T(\frac{n}{4}) + \sqrt{\frac{n}{2}}) + \sqrt{n} \\ &= 256T(\frac{n}{4}) + 16 * \sqrt{\frac{n}{2}} + \sqrt{n} \\ &= 256(16T(\frac{n}{8}) + \sqrt{\frac{n}{4}}) + 16 * \sqrt{\frac{n}{2}} + \sqrt{n} \\ &= 4096T(\frac{n}{8}) + 256\sqrt{\frac{n}{4}} + 16 * \sqrt{\frac{n}{2}} + \sqrt{n} \end{aligned}$$

We are dealing with an infinite series of some sort combined with a more traditional recurrence relationship.

$$T(n) = 16^k * T(\frac{n}{2^k}) + \sum_{i=0}^{k-1}(16^i * \sqrt{\frac{n}{2^i}})$$

Where k is the number of times we are unrolling (in this case $log_2(n)$) because we divide n in half every step. Simplifying the sum gets us:

$$\sum_{i=0}^{k-1}(16^i * \sqrt{\frac{n}{2^i}} = \frac{(2^{4k}\sqrt{2^{-k}n} - \sqrt{n})}{(8\sqrt{2} - 1)}$$

Lets substitute in our value for k, $log_2(n)$, so we can get the equation entirely in terms of n:

$$\begin{aligned} &= \frac{(2^{4log_2(n)}\sqrt{2^{-log_2(n)}n} - \sqrt{n})}{(8\sqrt{2} - 1)} \\ &= \frac{(n^4\sqrt{n^{-1}n} - \sqrt{n})}{(8\sqrt{2} - 1)} \\ &= \frac{(n^4\sqrt{1} - \sqrt{n})}{(8\sqrt{2} - 1)} \\ &= \frac{(n^4 - \sqrt{n})}{(8\sqrt{2} - 1)} \\ &\in O(n^4) \end{aligned}$$

Substituting $log_2(n)$ into the entire original equation for k, we get:

$$\begin{aligned} T(n) &= 16^{log_2(n)} * T(\frac{n}{2^{log_2(n)}}) + O(n^4) \\ T(n) &= 16^{log_2(n)} * T(\frac{n}{n}) + O(n^4) \\ T(n) &= 16^{log_2(n)} * T(1) + O(n^4) \end{aligned}$$

We can assume that T(1) occurs in constant time so overall we evaluate to:

$$T(n) = 16^{log_2(n)} + O(n^4)$$

$$T(n) = 2^{4*log_2(n)} + O(n^4)$$
$$T(n) = n^4 + O(n^4)$$
$$T(n) = \Theta(n^4)$$
$$T(n) \in \Theta(n^4)$$

(b) $T(n) = 5T(\frac{n}{5}) + 3n + 2$

This looks like it would take $O(nlog(n))$ time because if we think about it like a tree, we have five problems each of which is a fifth of the size of the original problem and then at each "layer" we perform $O(n)$ work signified by the (3n + 2). So lets run a guess and check with $O(nlog(n))$.

$$T(n) = 5T(\frac{n}{5}) + 3n + 2$$
$$= 5(5T(\frac{n}{25}) + 3\frac{n}{5} + 2) + 3n + 2$$
$$= 25T(\frac{n}{25}) + 6n + 12$$
$$= 25(5T(\frac{n}{125}) + 3\frac{n}{25} + 2) + 6n + 12$$
$$= 125T(\frac{n}{125}) + 9n + 62$$

We can rewrite the above equation in terms of k, the number of levels of our tree:

$$T(n) = 5^k T(\frac{n}{5^k}) + 3*k*n + \sum_{i=0}^{k-1}(2*k^i)$$
$$= 5^k T(\frac{n}{5^k}) + 3*k*n + \frac{2*(k^k - 1)}{k-1}$$

And we know that $k = log_5(n)$ so if we make that substitution:

$$= 5^{log_5(n)} T(\frac{n}{5^{log_5(n)}}) + 3*log_5(n)*n + \frac{2*(log_5(n)^{log_5(n)} - 1)}{log_5(n) - 1}$$
$$= nT(\frac{n}{n}) + 3*log_5(n)*n + \frac{2*(log_5(n)^{log_5(n)} - 1)}{log_5(n) - 1}$$
$$= nT(1) + 3*log_5(n)*n + \frac{2*(log_5(n)^{log_5(n)} - 1)}{log_5(n) - 1}$$
$$= n + 3*log_5(n)*n + \frac{2*(log_5(n)^{log_5(n)} - 1)}{log_5(n) - 1}$$
$$\in \Theta(nlog_5(n))$$
$$\in \Theta(n\frac{log_2(n)}{log_2(5)})$$
$$\in \Theta(nlog_2(n))$$

(c) $T(n) = T(\frac{n}{5}) + T(\frac{3n}{4}) + n$

Let's massage the recurrence via substitution.

$$T(n) = T(\frac{n}{5}) + T(\frac{3n}{4}) + n$$

$$T(n) = (T(\frac{n}{25}) + T(\frac{3n}{20}) + \frac{n}{5}) + (T(\frac{3n}{20}) + T(\frac{9n}{16}) + \frac{3n}{4}) + n$$

On second thought, that seems to be the wrong strategy, the recurrence relationship being composed of two recurrences (as opposed to one) makes the substitution strategy ineffective. Instead lets just use a nifty trick from slide 55 of the D&C1 slides. We know that T(n) is at least linear in n. This implies that $T(n)$ is super-additive.

$$T(n) \leq T(\frac{n}{5} + \frac{3n}{4}) + n$$

$$T(\frac{19n}{20}) + n$$

Another strategy switch. Lets use strong induction. I'm going to assume that for some base case $n = 1$, we can solve the problem in constant time. So for the inductive hypothesis, lets assume that this is true for 1, 2,..., n-1. So lets show that this holds for all values $n > n_0$. I will claim that $T(n) \leq 50n$.

$$T(n) = T(\frac{n}{5}) + T(\frac{3n}{4}) + n$$

$$\leq 50(\frac{n}{5}) + 50(\frac{3n}{4}) + n$$

$$= 10n + \frac{150n}{4} + n$$

$$= 10n + 37.5n + n$$

$$= 48.5n$$

$$\leq 50n$$

We can see that $T(n) \leq 50n$. Therefore, we can see that $T(n) \in 50n$ and subsequently, $T(n) \in \Theta(n)$.

3. Given a string x of length m, design a dynamic programing algorithm that uses linear space and returns the longest contiguous substring of x that occurs more than once. For example, if $x =$ ABABAB, then the answer is ABAB. Analyze the running time and prove the correctness of your algorithm.

Basically we need to use binary search to constrain the length of the substring and we can use Rabin-Karp to check if there is a substring of a given length. With Rabin-Karp telling us whether or not it found a repeating substring of a given length, we can then better inform the length of the substring that we are looking for.

Here is the code:

```
class Solution {
public:
    string longestDupSubstring(const string& s) {
        int low = 0;
        int high = s.size();
        string retString = "";
        string temp;
```

```
        while ( low <= high ) {
            temp = robin_karp(s, (high + low) / 2);

            if ( temp.size() > retString.size() ) {
                retString = temp;
                low = ((high + low) / 2) + 1;

            } else {
                high = ((high + low) / 2) - 1;

            }

        }

        return retString;

    }

    string robin_karp(const string& s, int size) {
        if ( size == 0 ) {
            return "";
        }
        unordered_set<string> u_set;
        string temp;
        temp.reserve(size);

        u_set.insert(s.substr(0, size));

        for ( int i = 1; i <= s.size() - size; i++ ) {
            // cout << s.substr(i, size) << endl;;
            temp = s.substr(i, size);

            if ( u_set.count( temp ) ) {
                return temp;
            } else {
                u_set.insert( temp );
            }

        }

        return "";

    }

};
```

The running time of the is algorithm is $O(nlog(n))$ becuase robin-karp takes linear time as it computes a rolling hash of any given substring. And the driver method (longestDupSubstring) calls robin-karp log(n) number of times as it is using binary search.

The reason that robin-karp takes linear time is because it generates size(s) - size number of substrings, a linear time operation overall. And then for each of those substrings, it generates a hash of said substring. If the substring is already in the set, it has found a duplicate and returns otherwise it inserts the substring into the set. The hashing of the substring, the duplicate detection, and set insertion all take constant time. Therefore, overall robin-karp takes a linear amount of time. The

handler function longestDupSubstring is guaranteed to take $O(log(n))$ time because it is just a version of binary search. But in each of those operations, longestDupSubstring calls robin-karp a linear time algorithm, therefore the complete runtime is $O(nlog(n))$.

In the case that the above question is not considered DP, I suggest a more traditional DP solution. This solution takes advantage of the fact that we build equivalent substrings up from smaller values of the same substrings. So if we see that $x_i == x_j$ we check the prior pair x_{i-1} and x_{j-1} to see if they are equal and so on. But if we were to simply cache the length of the paired matching substrings that end at x_{i-1} and x_{j-1}, we could simply add 1 to that value to get the length of the substring that ends at x_i and x_j. So for a DP solution, we iterate through a matrix and for each corresponding coordinate in the matrix, we inspect if x_j (where j is the row) and x_i (where i is the column) are equivalent. If they are equivalent, we add 1 to the diagonal element preceding i and j. If not equal we set the cell to 0. The implementation can be seen below:

```
string longestDupSubstring(const string& s) {
    int ssize = s.size();
    vector<int> dp1(ssize + 1, 0);
    vector<int> dp2(ssize + 1, 0);

    int ret = 0;
    string retString = "";


    for ( int i = 0; i < ssize; i++ ) {
        for ( int j = i + 1; j < ssize; j++ ) {
            if ( s[i] != s[j] ) {
                dp1[j] = 0;
            } else if ( s[i] == s[j] ){
                if ( i == 0 || j == 0 ) {
                    dp1[j] = 1;
                } else {
                    dp1[j] = 1 + dp2[j - 1];
                }

                if ( dp1[j] > ret ) {
                    ret = dp1[j];
                    retString = s.substr(j - ret + 1, ret);
                }

            }

        }
        dp1.swap(dp2);

    }

    return retString;

}
```

In terms of time and space complexity, this algorithm has a linear space complexity because we only allocate two lists of the size of the input, swapping them as we iterate through "rows" of the matrix. In terms of time complexity, we are $\in O(n^2)$. Because we iterate through every possible pair of characters.

To prove this algorithm is correct, I will proceed by induction. The base cases are anywhere dp[i][j] = 0. This is wherever in the matrix $s_i! = s_j$ or we are out of bounds of the matrix. Assume that a

matrix cell [i][j] stores the length of the longest substring that ends at [i][j]. So when we compute [i + 1][j + 1], we only have to check if $s_{i+1} == s_{j+1}$. If they are equal, we set [i + 1][j + 1] to 1 + [i][j]. Otherwise, we set it to 0. Building on this, we can guarantee that every cell is valid and correctly stores the length of the longest substring that ends at its corresponding index in the string.

4. Giggle, Inc. is organizing a holiday party and you are in charge of deciding what employees are invited. You know the 'fun' value of each employee, which is a real number, possibly negative. The goal is to maximize the fun value of the party, which is the sum of the fun values of the invited people. There are a few constraints to keep in mind. The company has a strict organizational hierarchy which is a rooted tree, with the president sitting at the root. Since it's no fun to socialize with a direct supervisor, an employee will not attend the party if his/her direct supervisor is present. The final constraint is that the president insists to be present, and so he has to be on the guest list, even though his fun value is negative. Give an efficient algorithm to throw the most fun Giggle party.

   This problem is asking for the selection of employees that is both valid (we don't have direct supervisor selections) and is optimally fun (the selection has a max fun value). So the dumb way to do this is just to consider all the binary strings associated with the nodes, check them for the root node being included, check for the string to satisfy the supervisor restriction, and we can just record the binary string that is maximal in fun. Unfortunately, the proposed solution is $O(2^n)$. So we need to be a little smarter. Thankfully, this is just https://leetcode.com/problems/house-robber-iii/ problem, with the condition that the root must be included. So I'll solve the Leetcode problem first and then come back to this.

   I can base my algorithm here off the LeetCode solution, but I need to modify the LeetCode code so that it forces the president to be selected every time. Also note that I have robHelp get a flag passed in that lets us know if we are checking for the grandchild of a node. By passing in a flag, we can avoid null pointer dereferences and can more securely handle for inspecting the child of a null node.

   Also, we need a little modification so that we can actually know if we picked a node or not. safeDeref is a helper method that will return NULL if a node doesn't exist otherwise, it returns a pointer to the node.

```
class Solution {
public:
    // lets hope that we can use a map to check the value of the root
    unordered_map<TreeNode*, int> m;
    unordered_map<TreeNode*, vector<TreeNode*>> selected;
    int rob(TreeNode* root) {
        if ( !root ) {
            return 0;
        }

        return root->val + robHelp(root->right, 'r') + robHelp(root->right, 'l')
             + robHelp(root->left, 'r') + robHelp(root->left, 'l');

    }

    int robHelp(TreeNode* root, char c) {
        if ( !root ) {
            return 0;
        }

        if ( c == 'r' ) {
            return rob(root->right);
        }
```

```
        if ( c == 'l' ) {
            return rob(root->left);
        }

        if ( m.count(root) ) {
            return m[root];
        }

        int selectSelf = root->val + robHelp(root->right, 'r') + robHelp(root->right, 'l')
                                   + robHelp(root->left, 'r') + robHelp(root->left, 'l');
        int SelectChild = robHelp(root->right, 'm') + robHelp(root->left, 'm');

        if ( selectSelf > selectChild ) {
            selected[root].insert(root);
            selected[root].insert(safeDeref[root->right->right]);
            selected[root].insert(safeDeref[root->right->left]);
            selected[root].insert(safeDeref[root->left->right]);
            selected[root].insert(safeDeref[root->left->left]);
        } else {
            selected[root].insert(safeDeref[root->right]);
            selected[root].insert(safeDeref[root->left]);
        }

        m[root] = max( root->val + robHelp(root->right, 'r') + robHelp(root->right, 'l')
                                   + robHelp(root->left, 'r') + robHelp(root->left, 'l'),
                    robHelp(root->right, 'm') + robHelp(root->left, 'm')
        );

        return m[root];

    }

};
```

Note that our solution is stored in selected and we can follow pointers within the map to see what nodes we selected at each level. This algorithm is efficient because we only perform an action at each node one time and don't need to duplicate actions because we use memoization. Our overall runtime in this problem is O(n).

5. You are in charge with placing a number of cleaning robots on a road system connecting a company's buildings at the headquarter campus. Each road is not necessarily bidirected and each directed road is labeled 'clean' or 'dirty'. A robot receives reward points for cleaning a dirty road and receives penalty points for driving on an already clean road. Assume that none of the robots can completely clean a dirty road, so the labels for the roads will stay the same throughout the release of the robots. The number of reward points associated to each dirty road and the number of penalty points associated to each clean road is known in advance. The road system is designed so that if a robot moved along a cycle it must have accrued more penalty points than reward points along the cycle. Given a start and a finish building locations, you are supposed to place a maximum number of robots such that 1) each robot drives between these locations 2) no two robots take the same overall route 3) each robot accumulates the maximum number points as possible (the final score is the number of award points minus the number of penalty points). Give an efficient algorithm to find the number of robots needed.

   This is just Bellman-Ford. But where Bellman-Ford tries to decrease the overall distance, here we instead want to maximize the overall distance. If we make some small adjustments to Bellman-Ford, we can do this problem fairly easily.

```
int BellmanFord(vector<int> vertices, vector<edge> edges, int source, int dest){
    // we get the vertices via a list of integers 0...n-1
    // each edge has a source, a dest and a weight (cost)

    vector<int> distance(INT_MIN, size(vertices));
    vector<vector<int>> predecessors(vector<int>(), size(vertices));
    unordered_set<vector<int>> paths();

    distance[source] = 0; // the distance to the source is 0 always

    for ( int i = 0; i < size(vertices); i++ ) {
        for ( const auto& edge: edges ) {
            if ( distance[edge.source] + edge.weight == distance[edge.dest] ) {
                distance[edge.dest] = distance[edge.source] + edge.weight;
                predecessor[edge.dest].append(edge.source);

            } else if ( distance[edge.source] + edge.weight > distance[edge.dest] ) {
                distance[edge.dest] = distance[edge.source] + edge.weight;
                predecessor[edge.dest] = {edge.source};

            }

        }

    }


    // at this point, bellman ford would try to find negative-weight cycles
    // here we care about positive-weight cycles (as we are trying to find
    // paths that maximize our distance) and we know that no positive-weight
    // cycles exist as stated in the problem statement

    return num(paths(predecessor, dest));


}
```

The modifications that we make to Bellman-Ford are two-fold. First, we try to increase the length of the path between nodes. Second, we need to keep track of *multiple* predecessors for each node, as long as each predecessor offers a path that is as competitive as all the predecessors in the set. Afterwards, we just perform BFS from the destination to the source and count the possible paths. Note that each of these paths are equally as good as one another because they are only generated if they offer the most profitable path to a given node. And we know that we have the most profitable path because we only update predecessor sets/distance values when we find more competitive paths.

Why we are left with the most profitable paths is as follows. We effectively readjust the weights to find the most profitable paths of length at most i every time. And by the end we have found the most profitable path with length at most —n— - 1 every time which is the maximum length a path can take (hits all nodes).

This algorithm is efficient because it runs in $\Theta(|V||E|)$ time. This is because we have a for loop that runs for the number of nodes times and each iteration of that loop runs over all the edges so that is a $|V| * |E|$ algorithm.

HW4

Sidhardh Burre(ssb3vk) **Due Date** 11:59pm on Mar. 29, 2023. Submit to Gradescope.

1. Given a network $G(V, E, s, t)$, give a polynomial time algorithm to determine whether G has a unique minimum s-t cut (i.e., an $s - t$ of capacity strictly less than that of all other s-t cuts).

   Here we want to generate all min cuts and count them. First we need to know how to find the minimum cut, second we have to identify and count the cuts. The trivial way to do this is to generate all possible sets of nodes, compute the max flow between them and examine if two of the sets have the same value. But this algorithm is easily $\Omega(2^n)$. Another method is to compute the max flow and examine the flow graph if we can find two distinct cuts where along both cuts all edges are completely saturated, then we have a graph with unique minimum cuts. Even though the prior idea is a little tedious, it exposes a unique component of the problem, we should be able to find two distinct cuts where the minimum flow is identical or that we have *two* places where the flow is saturated. So if we look at this from a set cut perspective, we should have two distinct pairs of sets s, t, s', t' of nodes such that:

   (a) maxflow(s, t) = maxflow(s', t')
   (b) $s \subseteq s'$
   (c) $t' \subseteq t$
   (d) $s \cap t = s' \cap t' = \emptyset$

   This means that we are double constrained and regardless of if we increase the minimum flow across either cut, we find that the overall flow doesn't increase. An algorithm can be constructed off of this thinking. Essentially, we compute the flow of a graph, storing the remaining capacities across all edges. Then we increase the capacity of a single saturated edge and recompute the flow. If we find that the flow has increased, we have a unique minimum s-t cut. Otherwise, we have a graph that contains a non-unique minimum s-t cut.

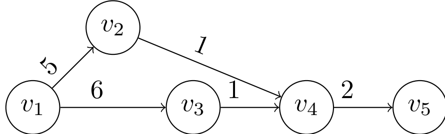
```
uniquemincut(G):
    for each edge in E:
        f(e) = 0

    G_f = residual network of V, E with respect to flow f
    cutedge = NULL

    while( there exists s->t path P in G_f ):
        f_1, cutedge = augment(f, c, P) // also returns the edge with capacity 0
        Update G_f

    c(cutedge)++

    while( there exists s->t path P in G_f ):
        f_2, _ = augment(f, c, P) // also returns the edge with capacity 0
        Update G_f

    return !(f_1 == f_2)
```

We can see that this a polynomial time algorithm because we simply run Ford-Fulkerson twice and it is known that Ford-Fulkerson has a O(m*C) runtime which is maintained here. We have shown a polynomial time algorithm to determine whether or not G has a minimum cut.

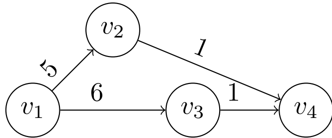
2. Consider a flow network $G = (V, E)$, with positive integer capacities on edge $C : E \to Z^+$. Let v be the value of the maximum flow in this network. Call an edge e a limiting edge if when assigning it a new capacity $c'(e) > c(e)$, the new flow network has a maximum flow $> v$.

   (a) Give an example of a network with no limiting edge. Give an example of a network with some limiting edge.

   I believe this is an extension of the prior problem. The example with no limiting edge:

   In the above example, the limiting edges are E(2, 4), E(3, 4), and E(4, 5). Increasing the capacity of any of one those edges won't increase the capacity of the overall graph. Take for example increasing the capacity of E(2, 4) or E(3, 4), E(4,5) still constrains the max flow and vice versa. The example with a limiting edge:

   Here the limiting edges are E(2, 4) and E(3, 4) increasing the capacity of either would increase the capacity of the overall graph.

   (b) Suppose that you have an algorithm that runs in time $T(G, c)$ that computes a maximum flow in a flow network. Give an $O(|V| + |E|) + T$ time algorithm that determines all the limiting edges in the graph. Prove the correctness of your algorithm.

   Once we perform the algorithm that computes the maximum flow network, we then perform BFS from both the source and the sink and capture the outer layer of edges that are reachable by taking edges that have remaining capacity. The captures from both directions should be equal: $capture_s \cap capture_t = capture_s = capture_t = capture_s \cup capture_t$. And these should be the limiting edges. In the case that the two captures are not equal, then there are no limiting edges.

   If we consider the first of the two examples above, (source) $capture_s = ((v2, v4), (v3, v4))$ and (sink) $capture_t = ((v4, v5))$. We can see that we have no intersect and therefore no limiting edges. For the second example we find that $capture_s = ((v2, v4), (v3, v4))$ and (sink) $capture_t = ((v2, v4), (v3, v4))$ and we have two limiting edges $((v2, v4), (v3, v4))$

   This takes $O(|V| + |E|) + T$ time because we perform BFS twice (from the source and the sink) and then we perform an intersect between two sets and we only need to verify that at least one element intersects. The BFS takes $O(|V| + |E|)$ time and the intersect verification takes $O(|V|)$ time. Therefore, we have $O(|V| + |E|) + T$ time algorithm to find the limiting edges in a graph.

   This algorithm is correct because we only select edges that are reachable via non-saturated edges. Further, to find a limiting edge we ensure that they are limiting from both the source and sink sides. Therefore, we satisfy the definition of a limiting edge and can show that the edges we do select (when we have an intersect of even one edge) are all limiting edges.

3. You've periodically helped the medical consulting firm Doctors Without Weekends on various hospital scheduling issues, and they've just come to you with a new problem. For each of the next n days, the hospital has determined the number of doctors they want on hand; thus, on day i, they have a requirement that exactly p_i doctors be present.

   There are k doctors, and each is asked to provide a list of days on which he or she is willing to work. Thus doctor j provides a set L_j of days on which he or she is willing to work.

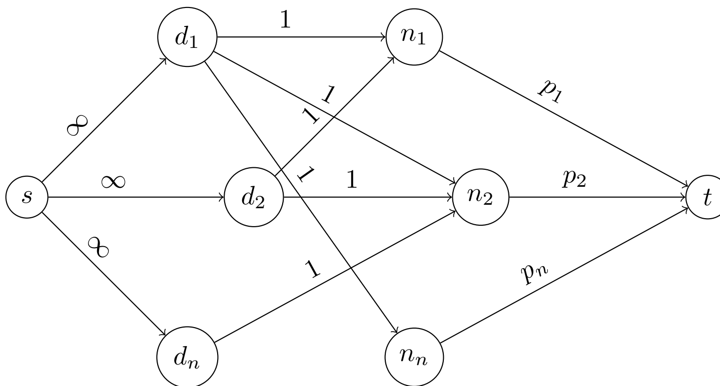
The system produced by the consulting firm should take these lists and try to return to each doctor j a list $L_j^{'}$ with the following properties. (A) $L_j^{'}$ is a subset of L_j, so that doctor j only works on days he or she finds acceptable. (B) If we consider the whole set of lists $L_1^{'}, \ldots, L_k^{'}$, it causes exactly p_i doctors to be present on day i, for $i = 1, 2, \ldots, n$.

(a) Describe a polynomial-time algorithm that implements this system. Specifically, give a polynomial-time algorithm that takes the numbers $p_1, p_2, \ldots, p_n$, and the lists $L_1, \ldots, L_k$, and does one of the following two things.

– Return lists $L_1^{'}, \ldots, L_k^{'}$ satisfying properties (A) and (B); or

– Report (correctly) that there is no set of lists $L_1^{'}, \ldots, L_k^{'}$ that satisfies both properties (A) and (B).

This is essentially Ford-Fulkerson but before doing that we have to do some processing on the provided input. The algorithm, after inputting the L's and p's must construct the following graph:



Edges from the source to the doctor's nodes d_n are of infinite capacity and are unconstrained. Edges between doctors and days are unit capacity, a flow in that edge represents that a doctor can attend that day. And edges between days and the sink are of p_n capacity showing that each day can only have a maximum of p_n doctors.

```
doctorAssignment(L, p):
    G = buildGraph(L, p) // described above

    for each edge in E:
        f(e) = 0

    G_f = residual network of V, E with respect to flow f

    while( there exists s->t path P in G_f ):
        f = augment(f, c, P)
        Update G_f

    if ( flow(G_f) == sum(p) ):
        collect all saturated edges from each node in d
        add each set of edges to a unique set based on origin node
        return set of sets
    else:
        return {}
```

In the algorithm above, we compute the flow of the graph that we've constructed. And if the flow is equal to the sum of the required doctors on each days, then we know that we have satisfied the requirement that exactly p_i doctors are present on day n_i. To get the days that doctors are

assigned, we simply look at the saturated edges between d and n, and for each doctor, we add the set of saturated edges to the corresponding day to determine their schedule. In the case that the flow is not equal to the sum of required doctors on each day, we know that the requirement cannot be satisfied and we return an empty set.

(b) The hospital finds that the doctors tend to submit lists that are much too restrictive, and so it often happens that the system reports (correctly, but unfortunately) that no acceptable set of lists $L_1^{'}, \ldots, L_k^{'}$ exists.

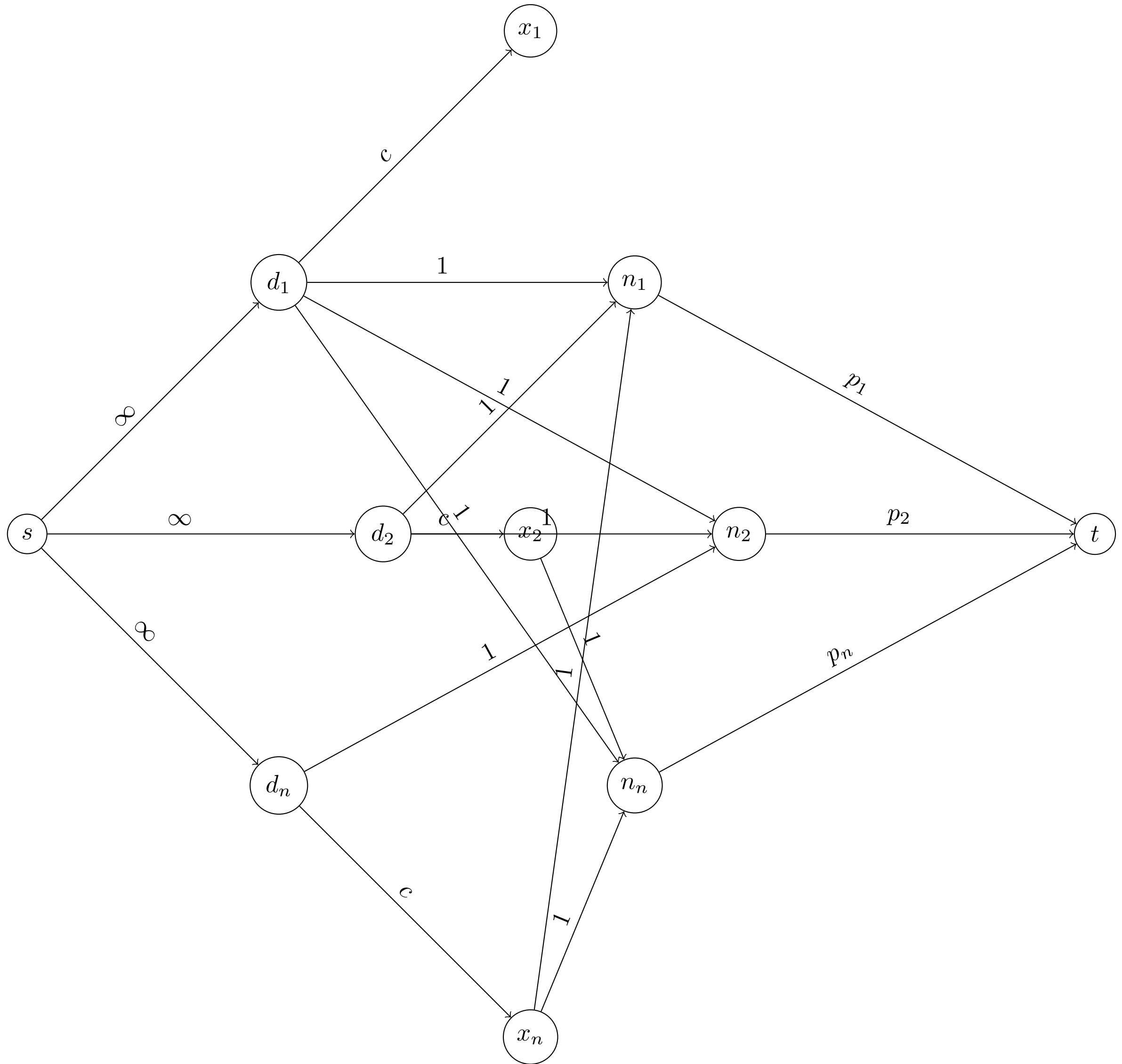
Thus the hospital relaxes the requirements as follows. They add a new parameter $c > 0$, and the system now should try to return to each doctor j a list $L_j^{'}$ with the following properties. (A*) $L_j^{'}$ contains at most c days that do not appear on the list L_j. (B) (Same as before) If we consider the whole set of lists $L_1^{'}, \ldots, L_k^{'}$, it causes exactly p_i doctors to be present on day i, for $i = 1, 2, \ldots, n$.

Describe a polynomial-time algorithm that implements this revised system. It should take the numbers $p_1, p_2, \ldots, p_n$, the lists $L_1, \ldots, L_k$ and the parameter $c > 0$, and do one of the following two things.

– Return lists $L_1^{'}, \ldots, L_k^{'}$ satisfying properties (A*) and (B); or

– Report (correctly) that there is no set of lists $L_1^{'}, \ldots, L_k^{'}$ that satisfies both properties (A*) and (B).

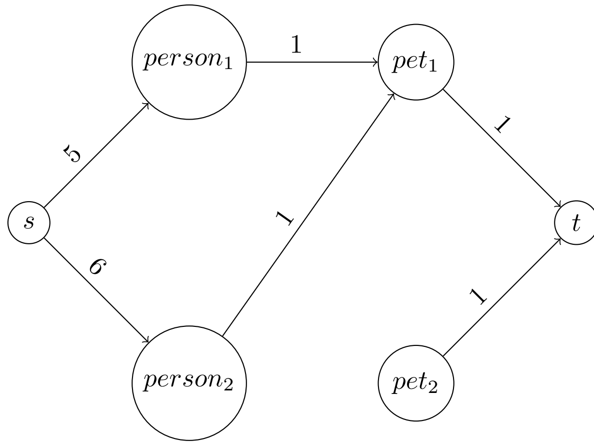
To solve this version of the problem, we do the same thing as before but have a slight modification to the graph we created before. We add dummy nodes to each doctor node and add unit capacity edges between these dummy nodes to node that were not in the original list that the doctor submitted. And we add a capacity of c on the edges between doctors and their dummy nodes.



Now, we can perform the same algorithm above with a slight change. Instead of simply getting saturated edges from the doctor's nodes in the case of where total flow is equal to the sum of all p's, we also need to get saturated edges from the dummy nodes attached to each of the doctor's nodes. This will enforce both the c overload limit and the prior list limit for each doctor.

4. A group of n people in Charlottesville are all interested in adopting a pet from a local shelter. Fortunately there are n pets available, but they can only be adopted if, after spending enough time with a potential owner, the pet shows some interest in the person. Assume also that if there interest between people and pets then the interest is mutual. At the end of a full day of socializing, each of the n people makes a list of p pets that they like, and each of the n pets must have showed interest in exactly p people. Does there always exist an assignment that matches people with pets such that they mutually like each other? If not, show a counterexample. If yes, show a proof.

   In this problem, I assume that the algorithm is not afforded the ability to leave pets/people unmatched. Additionally, I assume that the n people and n pets submit lists of exactly length p where $p > 0$. These conditions ensure that the following type of graph is impossible:



This problem essentially states that if interest exists between either human to pet or pet to human, then the interest is mutual. We can use Hall's marriage theorem to show that this problem will always result in a perfect matching, where both person and pet have mutual liking for one another. Hall's marriage theorem states that a bipartite graph (same number of left and right nodes) has a perfect matching if and only if $|N(S)| \geq |S|$ for all subsets of the left side of the graph.

We can use a proof by contradiction here. Suppose that we do not have a perfect matching. We convert this to a max flow problem and we let the source node + people be set A and the sink node + pets be B. We can now generate a min cut. By max-flow min-cut theorem cap(A,B) ¡ —L—. Lets now define the following:

(a) $L_A = L \cap A$

(b) $L_B = L \cap B$

(c) $R_A = R \cap A$

We can also formulate capacity to be the following $cap(A, B) = |L_B| + |R_A| - - > |R_A| < |L_A|$. Because the min cut doesn't use infinite edges, we know that the number of nodes adjacent to the L_A are a subset of R_A nodes. Therefore we can say that $|N(L_A)| \leq |R_A| < |L_A|$. This means that we should instead expand the source set to all values in L_A. This is a contradiction and therefore shows that hall's marriage theorem is true. Basically, given the conditions of the problem, where each people/pets node has p connections towards pets/people nodes respectively, we can gaurantee that we have a perfect matching.

HW5

Due Date 11:59pm on Apr. 12, 2023. Submit to Gradescope.

1. For each of the two questions below, decide whether the answer is (i) "Yes," (ii) "No," or (iii) "Unknown, because it would resolve the question of whether P = NP." Give a brief explanation of your answer.

   (a) Let's define the decision version of the Interval Scheduling Problem from Chapter 4 as follows: Given a collection of intervals on a time-line, and a bound k, does the collection contain a subset of non-overlapping intervals of size at least k?

   Question: Is it the case that Interval Scheduling $\leq_p$ Vertex Cover?

   Instead of reducing Interval Scheduling to Vertex Cover directly, we can instead show that Interval Scheduling reduces in polynomial time to Independent Set which in turn reduces in polynomial time to Vertex cover.

   $$IntervalScheduling \leq_p IndependentSet \equiv_p VertexCover$$

   Yes. We can reduce Interval Scheduling to Independent Set by doing the following. We can create a node for each of the intervals and then add edges between those nodes if and only if the nodes that the intervals correspond to overlap. This transformation takes polynomial time because all we're doing is duplicating the given intervals and then checking for overlap which can be done in $O(n^2)$ time. Then we pass this graph to the Independent Set oracle and ask it if there are k independent vertices. Because we already know that Independent Set reduces in polynomial time to Vertex Cover, we have shown that Interval Scheduling reduces to Vertex Cover in polynomial time.

   (b) Question: Is it the case that Independent Set $\leq_p$ Interval Scheduling?

   Unknown. This is because it would resolve the question of P = NP. We know that Interval Scheduling can be solved in Polyomial time and we know that Independent Set is NP-Hard. So demonstrating a polynomial time reduction between Independent Set and Interval Scheduling would show that P = NP.

2. You are given a directed graph $G = (V, E)$ with weights w on its edges $e \in E$. The weights can be negative or positive. The Zero-Weight-Cycle Problem is to decide if there is a simple cycle in G so that the sum of the edge weights on this cycle is exactly 0. Prove that this problem is NP-complete.

   This problem has two components. We have to show that the problem is NP and that it is also NP-Hard. To show that a problem is NP we have to show that given a potential solution to the problem, we can verify whether that solution is a real solution to the problem in polynomial time. Given a path cycle C and a graph G, we just need an algorithm A that can determine whether or not the cycle C has a zero-weight sum on the graph G. The algorithm A simply needs to traverse the nodes in G in the order given by C, sum up the weights along the edges (given that there are valid edges between the nodes) and return whether or not the sum is equal to 0. Thus we have shown that the Zero-Weight-Cycle problem is indeed NP.

   Now to show that the problem is NP-hard we have to get a known NP-complete problem and reduce it to our problem in polynomial time. I believe that the Subset sum problem can be reduced to the Zero-Weight-Cycle problem in polynomial time. We can do this by constructing a node for each number in the multiset of integers provided by the problem. We then add directed edges with the value of the node that are directed toward the node from all other nodes. And add a node z that has incoming

edges equal to the negative of the target sum and with the normal outgoing edges. Then we can pass this graph to the Zero-Weight-Cycle problem and query it for a Zero-Weight-Cycle that passes through our node z. If a Zero-Weight-Cycle is found, then the corresponding path excluding node z must have a sum equal to the target sum. Thus we have shown that Zero-Weight-Cycle is both NP and NP-hard and therefore is NP-complete.

If the above reduction fails, we can instead show that the Hamiltonian Path problem, a NP-complete problem, reduces to our Zero-Weight-Cycle problem in polynomial time. A Hamiltonian Path is a path in a graph that visits each vertex exactly once. We can convert the graph given by a Hamiltonian Path problem to a directed graph with all edge weights equal to 1 and set all of the inbound edges to an arbitrary node equal to $-(|N|-1)$. And pass this graph to the Zero-Weight-Cycle problem. Being able to find a zero-weight-cycle indicates that we have a cycle in the graph that visits every node. Which is also a solution to the Hamiltonian Path problem.

3. A combinatorial auction is a particular mechanism developed by economists for selling a collection of items to a collection of potential buyers. (The Federal Communications Commission has studied this type of auction for assigning stations on the radio spectrum to broadcasting companies.)

   Here's a simple type of combinatorial auction. There are n items for sale, labeled $I_1, \ldots, I_n$. Each item is indivisible and can only be sold to one person. Now, m different people place bids: The i-th bid specifies a subset S_i of the items, and an offering price x_i that the bidder is willing to pay for the items in the set S_i, as a single unit. (We'll represent this bid as the pair (S_i, x_i).)

   An auctioneer now looks at the set of all m bids; she chooses to accept some of these bids and to reject the others. Each person whose bid i is accepted gets to take all the items in the corresponding set S_i. Thus the rule is that no two accepted bids can specify sets that contain a common item, since this would involve giving the same item to two different people.

   The auctioneer collects the sum of the offering prices of all accepted bids. (Note that this is a "one-shot" auction; there is no opportunity to place further bids.) The auctioneer's goal is to collect as much money as possible.

   Thus, the problem of Winner Determination for Combinatorial Auctions asks: Given items $I_1, \ldots, I_n$, bids $(S_1, x_1), \ldots, (S_m, x_m)$, and a bound B, is there a collection of bids that the auctioneer can accept so as to collect an amount of money that is at least B?

   Example. Suppose an auctioneer decides to use this method to sell some excess computer equipment. There are four items labeled "PC," "monitor," "printer", and "scanner"; and three people place bids. Define S_1= PC, monitor, S_2 = PC, printer, S_3 = monitor, printer, scanner and $x1 = x2 = x3 = 1$. The bids are $(S_1, x_1), (S_2, x_2), (S_3, x_3)$, and the bound B is equal to 2.

   Then the answer to this instance is no: The auctioneer can accept at most one of the bids (since any two bids have a desired item in common), and this results in a total monetary value of only 1.

   Prove that the problem of Winner Determination in Combinatorial Auctions is NP-complete.

   To show that a problem is NP-complete, we have to show that it is both NP and NP-hard. To show that the problem is NP, we have to show that given a potential solution to the problem (in this case a set of sets) that produces price I, there is an algorithm that can tell us whether or not the given potential solution is a valid. In this case, the algorithm would need to show that this set of sets is both compatible (no items are duplicated among two or more sets) and that the sum of the prices for all sets is greater than bound B. We know that we can produce such an algorithm that runs in polynomial time because all it must do is verify that no sets have an intersection, a O(n) problem where n is the number of sets, and that the sum of the prices offered for each set is greater than a given bounds B, also a O(n) problem where n is the number of sets, as we just iterate over all the sets, add up their taking price, and return whether or not the sum is greater than B.

   Now to show that this problem is NP-hard, we have to show that a known NP-complete problem reduces to our problem in polynomial time. The NP-complete problem I will be using is the set packing problem. To convert the set packing problem to Winner Determination in Combinatorial Auctions, we only have to assign each subset a price of 1 and pass it into Combinatorial Auctions

asking it whether or not we can collect profit $\geq k$ where k is the number of sets we want to ensure are disjoint. If Combinatorial Auctions says that we can collect profit $\geq k$ then we know that we have $\geq k$ sets that are disjoint. This demonstrates that Winner Determination in Combinatorial Auctions is indeed NP-complete.

HW6

Due Date 11:59pm on Apr. 26, 2023. Submit to Gradescope.

1. Consider the following word game, which we'll call Geography. You have a set of names of places, like the capital cities of all the countries in the world. The first player begins the game by naming the capital city c of the country the players are in; the second player must then choose a city $c^{'}$ that starts with the letter on which c ends; and the game continues in this way, with each player alternately choosing a city that starts with the letter on which the previous one ended. The player who loses is the first one who cannot choose a city that hasn't been named earlier in the game.

   For example, a game played in Hungary would start with "Budapest," and then it could continue (for example), "Tokyo, Ottawa, Ankara, Amsterdam, Moscow, Washington, Nairobi."

   This game is a good test of geographical knowledge, of course, but even with a list of the world's capitals sitting in front of you, it's also a major strategic challenge. Which word should you pick next, to try forcing your opponent into a situation where they'll be the one who's ultimately stuck without a move?

   To highlight the strategic aspect, we define the following abstract version of the game, which we call Geography on a Graph. Here, we have a directed graph $G = (V, E)$, and a designated start node $s \in V$. Players alternate turns starting from s; each player must, if possible, follow an edge out of the current node to a node that hasn't been visited before. The player who loses is the first one who cannot move to a node that hasn't been visited earlier in the game. (There is a direct analogy to Geography, with nodes corresponding to words.) In other words, a player loses if the game is currently at node v, and for edges of the form (v, w), the node w has already been visited.

   Give a polynomial-time algorithm to decide whether a player has a forced win in Geography on a Graph, in the special case when the underlying graph G has no directed cycles (in other words, when G is a DAG).

   I am assuming that the start node s is not a node that player one ($P1$) selects but is arbitrarily selected and $P1$ selects a node that is adjacent (in the directed sense) to the start node s. For this problem, I will assume that the algorithm wants to determine if $P1$ can force a win. So if $P1$ *can* force a win, the algorithm will output true, otherwise it will output false, indicating that $P2$ will be able to force a win. This algorithm seeks to solve the following problem:

   $$\exists P1_1 \forall P2_1 \exists P1_2 \forall P2_2 ... \exists P1_n \nexists P2_n$$

   We only respond with true if the above statement also evaluates to true. This means that for all possible selections of P2, we must have a selection P1 that ensures that P1 has the last word. We can use a variant of recursive DFS to solve this problem.

```
bool recursiveGeography(node* curr, bool p1Turn):
    if ( p1Turn ):
        ret = false

        for n in neighbors(curr)
            ret |= recursiveGeography(n, !p1Turn)
```

```
            return ret

        else:
            ret = true

            for n in neighbors(curr)
                ret &= recursiveGeography(n, !p1Turn)

            return ret
```

This algorithm does exactly what was described in the above write-up. For each action that P2 can take, we ensure that there is at least one action that P1 can take that will ensure that for all actions P2 takes on and on, eventually P2 will not have a remaining action giving P1 the win in the game. Note that we do not need to keep track of visited nodes as there are no cycles so we cannot revisit a previously visited node even if we tried.

2. Suppose you are given a set of positive integers $A = a_1, a_2, \ldots, a_n$ and a positive integer B. A subset $S \subseteq A$ is called feasible if the sum of the numbers in S does not exceed B:

$$\sum_{a_i \in S} a_i \leq B$$

The sum of the numbers in S will be called the total sum of S.

You would like to select a feasible subset S of A whose total sum is as large as possible.

Example. If $A = \{8, 2, 4\}$ and $B = 11$, then the optimal solution is the subset $S = \{8, 2\}$.

   (a) Here is an algorithm (Algorithm 1) for this problem.

Algorithm 1 Algorithm 1

1: **procedure** MyProcedure
2:     Initially $S = \emptyset$
3:     Define $T = 0$
4:     For $i = 1, 2, \ldots, n$
5:         If $T + a_i \leq B$ then
6:             $S \leftarrow S \bigcup a_i$
7:             $T \leftarrow T + a_i$
8:         End if
9:     End for
10:    Return S

Give an instance in which the total sum of the set S returned by this algorithm is less than half the total sum of some other feasible subset of A.

This algorithm iterates through each member of A and checks if adding the value to the solution set will put the sum T of the solution set S, over the bounds B. If adding the value to the solution set does put our sum T over the limit B then we don't add the value to the solution set, otherwise we do add the value to the solution set.

An instance of the input set that could cause the set S returned by this algorithm be less than half of the total sum of some other feasible subset A is the set $A = (1, B)$. According to this algorithm, $S = 1$ but the optimal and feasible subset is $S^* = (B)$. And given that $B > 2$, we can see that the total sum given by the algorithm, $T = 1$, is less than half of the total sum of the feasible subset $S*$ because $T^* = B$. Because $B > 2$, we can see that $T < (1/2) * T^*$.

Algorithm 2 Algorithm 2

```
1: procedure MyProcedure
2:     Initially S = Ø
3:     Define T = 0
4:     A ← sorted(A) // sorts elements in descending order
5:     For i = 1, 2, ..., n
6:         If T + a_i ≤ B then
7:             S ← S ⋃ a_i
8:             T ← T + a_i
9:         End if
10:    End for
11:    Return S
```

(b) Give a polynomial-time approximation algorithm for this problem with the following guarantee: It returns a feasible set $S \subseteq A$ whose total sum is at least half as large as the maximum total sum of any feasible set $S' \subseteq A$. Your algorithm should have a running time of at most $O(nlogn)$.

The algorithm described above does almost exactly the same thing as Algorithm 1 with a key distinction, it sorts the input list A before operating on it. This algorithm is guaranteed to return a feasible set S whose total sum is at least half as large as the maximum total sum of any feasible set in $O(nlogn)$ time. This algorithm runs in $O(nlogn)$ time where n is the number of elements in the input list. This because we have two steps, a sort of the input set A (which takes $O(nlogn)$ time) and a linear scan through the sorted list which takes $O(n)$ time for a total of $O(nlogn)$ time. The proof for the feasible set is below.

Before proceeding, I would like to differentiate the input space into two. One is where the following condition is met:

$$sum(A) \leq B$$

The other is the following:

$$sum(A) > B$$

Note that in the first case, the algorithm will simply set the solution set $S = A$. And we satisfy the condition where $sum(S) \geq \frac{1}{2}S'$ where S' is the feasible set that generates the maximum possible sum. This is because $S' = A = S$. So this proof will only deal with the latter case of this problem, where $sum(A) > B$ and we must select a strict subset of A as our solution S.

As we progress along the list of elements in A (in decreasing order), we only add element A_i if $A_i < B - T$ (note that $\leq$ is pointedly not used as that would cause us to fall into the first case where $sum(A) \leq B$). But eventually, as we scan along elements of A, we come across an element in A, A_i where adding this element to the set S will put us over the limit prescribed to us B. Because we have worked to sort all our values, we can guarantee that $T > A_i$. And because $A_i + T > B$, we can also say that $T + T > A_i + T > B$ we can go even further and state that $T + T > A_i + T > B \geq OPT(S)$. We can extract the relevant components of the inequality to find that $2T \geq OPT(S)$ or that $T \geq \frac{1}{2}OPT(S)$. Therefore, we have provided an algorithm that generates a feasible set $S \subseteq A$ whose total sum is at least half as large as the maximum total sum of any feasible set $S' \subseteq A$.

3. Consider the Traveling-Salesman problem in which the distance function is symmetric, i.e. $d(u, v) = d(v, u)$ for all $u, v \in V$, and it satisfies the triangle inequality. Show an algorithm that produces a 2-approximation to an optimal-length hamiltonian tour in this graph.

   An algorithm that can generate a 2-approximation for the optimal length hamiltonian tour in a graph with symmetric distance functions and satisfies the triangle inequality is as follows. This algorithm will first generate the MST of this graph, the node with the highest indegree will be selected as the start node n of our approximate hamiltonian tour.

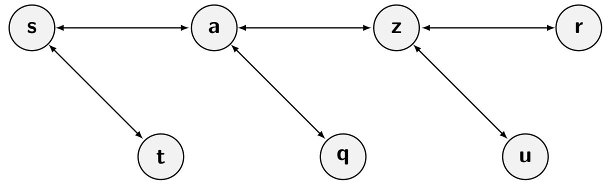
   From this start node n, we perform dfs, marking nodes as visited and touring through the nodes to generate a hamiltonian tour. The algorithm is detailed below:

Algorithm 3 2-approximation for Optimal Hamiltonian Tour

```
1: procedure MyProcedure(G)
2:     G, n ← MST(G)
3:     Initially S = stack
4:     hTour ← ∅
5:     S.push(n)
6:     While S is not empty do:
7:         n = S.pop()
8:         If n is not discovered then
9:             n.discovered ← True
10:            hTour ← hTour + n
11:            For each neighbor of n, v:
12:                S.push(v)
13:    Return hTour
```

The proof as to why this algorithm produces a 2-approximation solution to the optimal length Hamiltonian tour is because of the following. Note that the edge-weight sum of the MST $sum(MST.E) = w$ is upperbounded by the Optimal Weight of a Hamiltonian Path h, $w \leq h$. Via the solution that is given by this algorithm, each node is visited exactly once in a DFS method, so only once we've exhausted an entire branch do we "jump" back up to the child of the original branching location. The distance of this jump is upper-bounded by the path through the MST between these two nodes. As long as all of the jumps/traversals combined only end up traversing/being upper bounded by the MST edges exactly twice, we can produce a 2-approximation solution for the problem.

First, I must show that the solution generated in hTour is indeed a tour. This is by definition, a DFS/pre-order traversal is always a tour, each node appears once. A Hamiltonian Tour H^* is indeed a spanning tree, not a minimum spanning tree, but a spanning tree of some sort. Therefore, the weight of the optimal Hamiltonian Tour is lower bounded on the weight of the MST: $MST \leq H^*$. A full walk of the tree (keeping track of when we remove values from the stack) lists vertices both when they are first encountered *and* when they are returned to (after visiting the subtree rooted at the vertex) essentially traversing each edge *exactly twice*. So the weight of walk W along the edges used to create the MST, is guaranteed to be $W = 2 * MST$. And because we established before that $MST \leq H^*$, we can also state that $2 * MST \leq 2 * H^* \implies W = 2 * MST \leq 2 * H^* \implies W \leq 2 * H^*$. But our walk W is not a proper tour as it includes each vertex twice. So we have the problem of transforming the walk W to our tour generated by our algorithm H. But because we know that the triangle inequality is held here, we know that we can skip the second visit to a node and simply "jump" to the next non-visited node in our walk. And these jumps won't increase the cost of the tour H because the triangle inequality upper bounds the direct distance between (jump) pairs of nodes by the path that would need to be taken via the MST. So we can now say that $H \leq W \leq 2 * H^*$.



What a walk would look like starting from node s: s t s a q a z u z r z a s. What a tour would look like: s t a q z u r