Problems marked with **E** are graded on effort, which means that they are graded subjectively on the perceived effort you put into them, rather than on correctness. For bonus questions, we will not provide any insight during office hours or Piazza, and we do not guarantee anything about the difficulty of these questions. We strongly encourage you to typeset your solutions in LaTeX.

If you collaborated with someone, you must state their names. You must write your own solution and may not look at any other student's write-up.

0. If applicable, state the name(s) and uniqname(s) of your collaborator(s).

Solution:

1. Conceptually, NP represents the class of decision problems whose "yes" answers can be verified efficiently. In this problem, we will consider its counterpart: $\mathsf{coNP} = \{\overline{L} \mid L \in \mathsf{NP}\}$, representing the class of decision problems whose "no" answers are efficiently verifiable.

   (a) Prove that P is closed under set complement. That is, for any $L \in \mathsf{P}$, we have $\overline{L} \in \mathsf{P}$.

   Solution: If $L \in \mathsf{P}$, there exists an efficient decider M_L for L. We can construct an efficient decider $M_{\overline{L}}$ that, on input x, simply runs $M(x)$ and outputs the opposite answer. (Equivalently, we can just swap the accept and reject states of M.) It is obvious that $M_{\overline{L}}$ is an efficient decider, and $M_{\overline{L}}(x)$ accepts if and only if $M(x)$ rejects, which happens if and only if $x \notin L$. Therefore, $M_{\overline{L}}$ decides $\overline{L}$, as desired.

   (b) Using what you proved in part (a), prove that if $\mathsf{NP} \neq \mathsf{coNP}$, then $\mathsf{P} \neq \mathsf{NP}$.
   Hint: Consider the contrapositive.

   Solution: We prove the contrapositive, that $\mathsf{P} = \mathsf{NP}$ implies $\mathsf{NP} = \mathsf{coNP}$.
   If $\mathsf{P} = \mathsf{NP}$, then because P is closed under set complement, so is NP. So for any $L \in \mathsf{NP}$, we have $\overline{L} \in \mathsf{NP}$ so $\overline{\overline{L}} = L \in \mathsf{coNP}$. Thus, $\mathsf{NP} \subseteq \mathsf{coNP}$. Symmetrically, for any $L \in \mathsf{coNP}$, we have $\overline{L} \in \mathsf{NP}$ and hence $\overline{\overline{L}} = L \in \mathsf{NP}$, so $\mathsf{coNP} \subseteq \mathsf{NP}$. It follows that $\mathsf{NP} = \mathsf{coNP}$.

   (c) Show that $\mathsf{NP} \cap \mathsf{coNP} \neq \varnothing$.

   Solution: Note that there are many languages that would work, including any language in P. One such example is $\emptyset$.
   We know that $\emptyset \in \mathsf{P}$, and since $\mathsf{P} \subseteq \mathsf{NP}$, we know that $\emptyset \in \mathsf{NP}$. Also, we know that $\overline{\emptyset} = \Sigma^* \in \mathsf{P}$, and using similar reasoning, $\Sigma^* \in \mathsf{NP}$. Since $\Sigma^* = \overline{\emptyset} \in \mathsf{NP}$, then by definition $\emptyset \in \mathsf{coNP}$. Since $\emptyset \in \mathsf{NP}$ and $\emptyset \in \mathsf{coNP}$, then $\emptyset \in \mathsf{NP} \cap \mathsf{coNP}$, and so $\mathsf{NP} \cap \mathsf{coNP} \neq \emptyset$.
   Note that this is saying $\emptyset$ is an element in $\mathsf{NP} \cap \mathsf{coNP}$, meaning the intersection is not empty. In fact, many languages are in this intersection, including any language in P, since P is closed under complement.

(d) We say the languages Σ^* and $\varnothing$ are *trivial* languages and all other languages are *non-trivial*. Show that if $\mathsf{P} = \mathsf{NP}$ then every non-trivial language is NP-Hard.

Solution: Let L^* be non-trivial. Then there exists $x \in L^*$ and $x' \notin L^*$. Let L be any language in NP; since we are assuming $\mathsf{P} = \mathsf{NP}$, $L \in \mathsf{P}$. Let D be an efficient decider for L. Consider the following reduction $f\colon \Sigma^* \to \Sigma^*$ defined as follows:

```
1: function f(w)
2:     if D(w) accepts then
3:         return x
4:     else
5:         return x'
```

f is efficient because all steps taken are efficient.

f maps correctly because $D(w)$ accepts implies that $w \in L$. By the design of f, $f(w) = x$ which is an element of L^*.

Similarly, $D(w)$ rejects implies that $w \notin L$. By the design of f, $f(w) = x'$ which is not an element of L^*.

2. Suppose you are organizing a face mask distribution operation. You intend to set up a number of centers and you would like every household in the area to be near a mask distribution center. To model this, consider an undirected graph $G = (V, E)$. Represent households by vertices. At each vertex, you may place a distribution center and would like to place at most k centers. You would like to have every household close to a distribution center. More formally, every vertex must be within a constant c number of edges away from a distribution center.

   Define

$$\text{MDC} = \{\langle G = (V, E), k, c\rangle : \exists V' \subseteq V \text{ s.t. } |V'| \leq k, \text{ and } \forall v \in V, \exists F \in V' \text{ s.t. } dist(v, F) \leq c\}$$

   where $dist(a, b)$ when $a, b \in V$ is defined to be the number of edges in the shortest path from a to b. Show that the provided mapping from Vertex-Cover proves MDC is NP-Hard.

Input: Undirected graph $G = (V, E)$, natural number k

```
1: function f(G = (V, E), k)
2:     initialize new graph G' = (V', E') and set V' = V, E' = E
3:     for every edge (u, v) ∈ E do
4:         create a new vertex w_uv in V'
5:         add edges (w_uv, v) and (w_uv, u) to E'
6:     count the number of isolated vertices n_I    // isolated vertices are vertices with no incident edges
7:     set k' = k + n_I
8:     return ⟨G', k', 1⟩
```

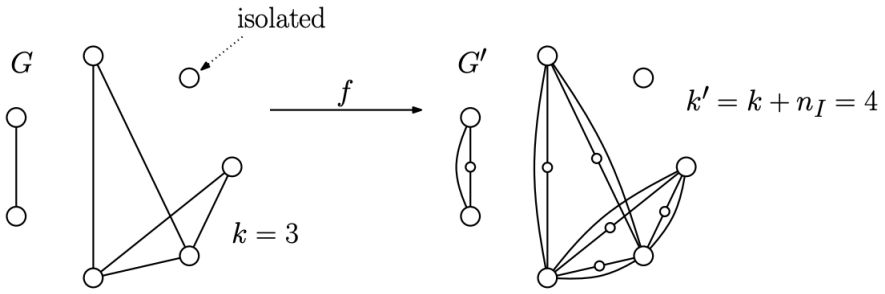


Figure 1: Example of how f maps

(a) Fill in the blanks: f shows that _____ $\leq_p$ _____.

> **Solution:** Vertex-Cover $\leq_p$ MDC

(b) Show that the mapping f is computable in polynomial time.

> **Solution:** To initialize G', we need to copy over G, which would require $O(|V||E|)$ operations. The loop on line 2 does $O(|E|)$ operations. To count the number of isolated vertices requires to look at every vertex and check the lack of their adjacent vertices, which we can infer by looking at the edge set. This step requires at most $O(|V||E|)$ operations. All the other steps are constant time, therefore the entire function runs in $O(|V||E|)$ time - polynomial in the size of input.

(c) Show that $\langle G, k\rangle \in$ Vertex-Cover implies $\langle G', k', 1\rangle \in$ MDC. In other words, show that if a graph G has a vertex cover of size k, then the output graph G' of function f has a network of $\leq k'$ distribution centers such that each household is at most 1 edge away from a member of the network of distribution centers.

> **Solution:** We argue that if $V_C \subseteq V$ is a vertex cover of size k for G and $V_I \subseteq V$ is the set of all isolated vertices of G, then placing distribution centers at every vertex of $V_C \cup V_I$ will ensure that every household is at most 1 away from a distribution center.
>
> Let $C = V_C \cup V_I$. $|C| = |V_C \cup V_I| \leq k + n_I = k'$. Observe that V_C is a vertex cover of size k, but it is not necessarily a vertex cover of minimal size for G. This means that $V_C \cap V_I$ may not be empty. If $V_C \cap V_I \neq \varnothing$, then we over-count some vertices and obtain an inequality. On the other hand, if $V_C \cap V_I = \varnothing$, then we obtain equality.
>
> We show that placing distribution centers at every vertex in C guarantees that every vertex in G' is either in C or at most a single edge away from C.
>
> First, all the isolated vertices are in C, therefore those vertices are right at a distribution center.
>
> Second, the vertices in V_C are a vertex cover, meaning they cover every edge in G, which means that every vertex in G is either at a distribution center (part of V_C) or

one away from one (vertices on the other end of edges that are covered by vertices in V_C).

Lastly, as for the new vertices added, each edge $(u, v) \in G$ has one of it's vertices in C. Every newly added vertex w_{uv} is connected by an edge to either u or v of some edge in G. Therefore, all the newly added vertices are within one edge of a distribution center vertex.

Therefore all the vertices in G' are at most one edge away from the k' distribution center vertices.

(d) Show that $\langle G', k', 1\rangle \in$ MDC implies $\langle G, k\rangle \in$ Vertex-Cover. In other words, show that if the output graph G' has a network of $\leq k'$ distribution centers such that each household is at most 1 edge away from a member of the network of distribution centers, then the graph G that was passed as input to f has a vertex cover of size at most $k = k' - n_I$.

Solution: We argue that if all vertices of G' are within at most one edge of one of k' distribution center vertices, then G has a vertex cover of size k.

Suppose M is the set of distribution center vertices. All n_I of the isolated vertices in G' are in M. Consider $V_0 = M - V_I$. $|V_0| = k' - n_i = k + n_I - n_I = k$. Consider now V_C, which is exactly V_0 but every vertex w_{uv} that is not in the original graph G is replaced with either u or v from G. First, $|V_C| = |V_0|$. Second, if for some $(u, v) \in G$, a distribution center supplying masks to u, v was located at w_{uv}, then we replaced it with one of u, v to cover edge (u, v) in G. This works out because now the replaced u or v can act as a distribution center giving masks to one of v, u and to w_{uv}, and all three of $\{u, v, w_{uv}\}$ are getting masks from the same number of centers as before.

For all (u, v) in $E(G)$, one of u, v is in V_C, which means it is vertex cover of size k. Therefore, G has a vertex cover of size k.

Note that the purpose of adding the vertices w_{uv} in the mapping f is to force a distribution center to be located at one of the vertices u, v, or w_{uv} in G' for each edge (u, v) in the original graph G. This is what preserves "no" instances of Vertex-Cover being mapped to "no" instances of MDC – if G does not have a vertex cover of size k, any size-k subset of the vertices leaves an edge (u, v) uncovered, and a corresponding size-k' subset of the vertices in G' leaves w_{uv} too far from a distribution center.

3. Having seen that MDC is NP-Hard, a natural question would be to ask if a corresponding search problem is hard. Consider the following restriction of MDC:

$$\text{MDC}_1 = \{\langle G = (V, E), k\rangle : \exists V' \subseteq V \text{ s.t. } |V'| \leq k, \text{ and } \forall v \in V, \exists F \in V' \text{ s.t. } dist(v, F) \leq 1\}$$

where we constrain the maximum distance within which we would like homes to be from distribution centers. In this restriction (which is still NP-Complete by the reduction presented above), a natural question is to find where the distribution centers should be located.

(a) Write an *efficient* algorithm that for a graph $G = (V, E)$ finds a minimal (smallest) set of vertices where one could place distribution centers to ensure that every vertex is within at most 1 edge distance of a distribution center vertex using an *efficient* blackbox decider for MDC_1.

In other words, your algorithm should output $M \subseteq V$ such that

$$\forall v \in V : \exists m \in M : dist(v, m) \leq 1.$$

Hint: One way to make it easier to find where to place the k facilities is to adjust the graph by removing edges "irrelevant" to the problem. For example, is an edge between two vertices that are both outside of V' necessary? What does the graph look like after these edges are removed? What other kinds of edges exist?

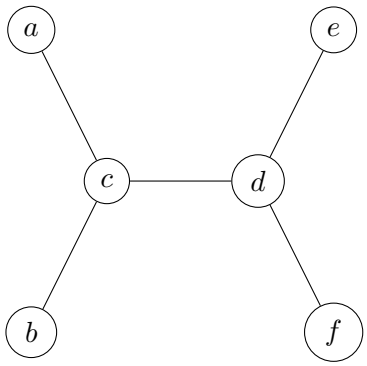
Solution: First note that the MDC_1 is a formulation of the language which is called DOMINATING-SET.

Suppose $MD(\langle G, k\rangle)$ decides MDC_1.

Let's first consider an incorrect attempt in the spirit of the vertex cover search-to-decision reduction.

```
Input: Undirected graph G = (V, E)
function A(G)
    l ← 1, r ← |V|
   // Binary search for the least number of distribution centers
    while l ≠ r do
        k_min ← (l+r)/2
        if MD(G, k_min) accepts then
            r ← k_min
        else
            l ← k_min
      // k_min is the least number of distribution centers necessary
    M ← ∅
    for every vertex v ∈ V do
        Remove v and its neighbors (and their edges) from G
        if MD(G, k_min − 1) accepts then
            M ← M ∪ {v}
            k_min ← k_min − 1
        else
            Replace v and its neighbors (and their edges) to G
    return M
```

This does not work. Consider the following graph for which $k_{min} = 2$:



Running the algorithm proposed, we will eventually attempt to remove c. Removing c will also remove the other center location which we need. The problem is that distribution centers may be located next to each other, so removing one prevents the algorithm from finding the other. In vertex cover, we do not run into this problem because we do not remove other vertices when testing a vertex. One way to resolve this is to remove edges between distribution centers. We employ this approach in the following algorithm.

We can apply the black-box decider to tell us what edges are necessary. Edges which go between vertices in V' and edges which go between $\overline{V'}$ are not necessary for the placement of distribution centers. Additionally, it is not necessary for a vertex to be supplied by multiple distribution centers.

Input: Undirected graph $G = (V, E)$

```
1:  function B(G)
2:      l ← 1, r ← |V|
3:    // Binary search for the least number of distribution centers
4:      while l ≠ r do
5:          k_min ← (l+r)/2
6:          if MD(G, k_min) accepts then
7:              r ← k_min
8:          else
9:              l ← k_min
          // k_min is the least number of distribution centers necessary
10:     for every edge e ∈ E do
11:         Remove e from G
12:         if MD(G, k_min) rejects then
13:             Add e back to G
14:     M ← ∅
15:     for every vertex v ∈ V do
16:         if v is isolated then
17:             add v to M
18:         if v has one adjacent vertex u and degree of u is 1 then
19:             add v to M and remove u from V
20:         if v has more than one adjacent vertex then
21:             add v to M
22:     return M
```

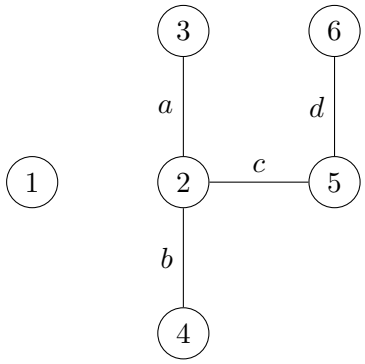
Analysis:

First, the binary search sets k_{min} to be the minimum possible number of distribution centers that will "cover" our graph. The algorithm then removes all edges that don't change the behavior of the decider: edges connecting non-center vertices, center-vertices and for vertices connected to several center vertices, it only leaves one such connection. The graph is then a collection of stars (isolated vertices, isolated pairs of vertices and center vertices connected to several other vertices that are only connected to it). The later loop counts these stars and adds their centers to output (note that for isolated pairs, an arbitrary vertex is chosen as its center).

Efficiency:

The binary search will make $O(\log |V|)$ loops, and each loop is polynomial in G, since MD is an efficient decider. The efficient decider MD will be run $|E|$ times, which is efficient. The loop on line 13 makes $O(|V| \cdot |E|))$ to check every vertex and all adjacent vertices. All parts of the algorithm are efficient, and so the algorithm is efficient.

(b) Run your algorithm on the following graph, noting the size of the minimal network that it computes and any intermediate results and the vertices it puts the distribution centers at. You don't have to write down the value of every component for every step of the program - what happens after whole loops/blocks of code run will suffice.



Solution: There are a few different possible executions of the algorithm depending on the order in which edges are considered.

The size of the minimal network is 3, so the binary search will find that the value of $k_{min} = 3$.

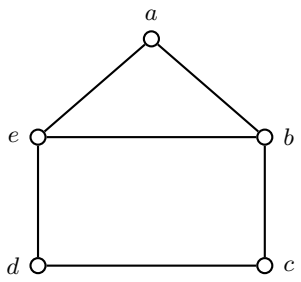
For an arbitrary enumeration of all edges $a - d$, the first loop will not take out a because that will isolate vertex 3 and MD will reject. It will not take out b for the same reason. It will take out c as the decider will still accept the modified graph with the fixed value of k_{min}. It will not remove d for the same reason as a, b.

This leaves stars $\{1, 234, 56\}$.

The second loop will pick vertices $\{1, 2, 5\}$, which are the minimal network distribution center vertices for the graph.

E 4. In this problem, you will work with the "double cover" (also known as "cover twice and remove") approximation algorithm for VERTEX-COVER.

(a) Run the algorithm on the following graph. State, with proof, the smallest approximation ratio that can be obtained in this case.



Solution: Running the vertex cover approximation algorithm:

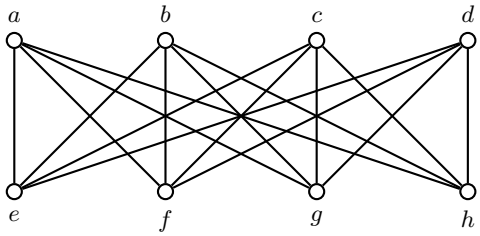
i. We first choose edge (a, b), adding a, b to the cover to get $C = \{a, b\}$, which covers all edges incident to a or b: these are $(a, b), (a, e), (b, e), (b, c)$.

ii. Next, we choose edge (d, e), adding d, e to the cover to get $C = \{a, b, d, e\}$, and additionally covering edges (d, c).

iii. Now all the edges are covered, so the output cover is $C = \{a, b, d, e\}$.

Analysis: the cover C produced by this run of the algorithm has size 4. We show below that the optimal cover size for this graph is 3. Because the algorithm always outputs a cover of even size, the smallest approximation ratio the algorithm can achieve for this graph is 4/3.
It is easy to verify that $C^* = \{b, c, e\}$ is a vertex cover, so there exists one of size 3. (There are others as well.) There is no cover of size 1, because no single vertex is incident to every edge. There is no cover of size 2: because there are 6 edges overall, any cover of size 2 would have to consist of the two nodes b, e that are each incident to 3 edges, but $\{b, e\}$ is not a vertex cover.

(b) Run the algorithm on the following graph. State, with proof, the smallest approximation ratio that can be obtained in this case.



Solution: Running the vertex cover approximation algorithm:

i. We first choose (a, e), updating the cover to $C = \{a, e\}$ and covering edges $(a, e), (a, f), (a, g), (a, h), (e, b), (e, c), (e, d)$.

ii. Next we choose uncovered edge (b, f), updating the cover to $C = \{a, b, e, f\}$ and additionally covering edges $(b, f), (b, g), (b, h), (f, c), (f, d)$.

iii. Next we choose uncovered edge (c, g), updating the cover to $C = \{a, b, c, e, f, g\}$, and additionally covering edges $(c, g), (c, h), (g, d)$.

iv. Finally, we choose edge (d, h), updating the cover to $C = \{a, b, c, d, e, f, g, h\}$ and additionally covering edge (d, h).

v. Now all the edges are covered, so the output cover is $C = \{a, e, b, f, c, g, d, h\}$.

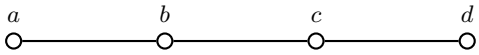
Analysis: the cover C produced by this run of the algorithm has size 8. We show below that the optimal cover size for this graph is 4. We also show that *any* run of the algorithm for this graph outputs a cover of size 8. Therefore, the smallest approximation ratio that the algorithm can obtain is $8/4 = 2$.
Observe that the above graph is the complete bipartite graph $K_{4,4}$, i.e., the graph with 4 vertices on each side, and an edge between every pair of top/bottom vertices (and no other edges). It is easy to see that $\{a, b, c, d\}$ is a cover of size 4. There is no cover of size smaller than 4, because any set of at most 3 vertices leaves at least

one vertex from each of the top and bottom sides unselected, so the edge between those two vertices is uncovered.

The algorithm always returns a cover of size 8 for this graph. This is because with each iteration it selects an edge, and therefore adds exactly one vertex from each of the top and bottom sides to its cover. As just argued, it must run for 4 iterations because otherwise there are unselected nodes from both the top and bottom, and hence an uncovered edge. So, the final cover size is always 8.

(c) Notice that the order of the edges considered in the algorithm is not specified. Prove or disprove: for any graph G, the size of the cover produced by the algorithm is the same regardless of the edge ordering.

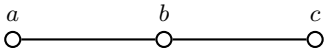
Solution: Here is a very simple counterexample to the statement:



Notice that the minimum vertex cover size is 2 (e.g., $C^* = \{b, c\}$); one vertex is clearly not sufficient. When using the double-cover algorithm, if we first pick the edge (b, c), then b, c are added to the cover and we are done. But if we first pick either (a, b) or (c, d), then we have not covered the other one. So that other edge will be picked next, and the output cover will have size 4.

(d) Show that the approximation ratio obtained for the minimal vertex cover search problem is tight. In other words, show a graph for which the "double cover" algorithm *always* outputs twice the optimal.

Solution: The minimal vertex cover $\{b\}$ has size 1. The double cover algorithm must take two vertices, so it will always take twice as many vertices as the size of the minimal vertex cover.



However, one might protest that this graph is of no issue because we could exhaustively search for a minimum vertex cover for this graph. A family of graphs for which we obtain twice the minimum value and for which we can make the size of the minimum vertex cover arbitrarily large is the family of complete bipartite graphs $K_{a,b}$ (where $a \leq b$ without loss of generality) of which this graph is a member. $K_{a,b}$ refers to the bipartite graphs where there are a vertices "on the left", b vertices "on the right", and edges between every pair (α, β) with α drawn from the left and β drawn from the right. In this case, similar to the reasoning from part (b) above, we have that the minimum vertex cover has size a, but because a edges are taken, the size of the vertex cover is $2a$ when using the "double-cover" algorithm.

5. One minimization problem which you may be familiar interacting with is *job scheduling* - allocating jobs to servers such that the time that the last server finishes its work is minimized.

One way to formulate this problem is to assume that there are m jobs total and n identical servers and the m jobs may be allocated in any way among the n servers subject to the constraint that jobs are not split over servers. The job j takes time t_j to complete on any server. It turns out that there is a 2-approximation for this formulation!

Input: List of jobs $J = [j_1, j_2, \ldots, j_m]$
1: **function** 2-APPROXIMATION(J)
2: **while** there are unallocated jobs **do**
3: allocate a job to a server which would finish the jobs it has been previously allocated first (or tied for first)
4: **return** the computed allocation

(a) Suppose that the input jobs $j_1, \ldots, j_6$ take 1 hour to complete, j_7 takes 3 hours to complete, and there are 3 servers to allocate jobs to. Determine an optimal allocation and state the time it takes to complete all the jobs for the optimal allocation. Show that, depending on the order which jobs are input the to the algorithm, the value of the allocation returned by the greedy algorithm differs. In particular, compare the value(s) which you found to the optimal by giving the ratio $\left(\frac{\text{value of the computed allocation}}{\text{value of the optimal allocation}}\right)$.

Solution: One way to allocate jobs is to to allocate j_7 first and $j_1, \ldots, j_6$ afterwards. This would yield the optimal allocation, where each machine finishes at 3 hours – this is optimal because the jobs require a total of 9 work hours to complete, so the minimal case is when all three machines work for 3 hours each. In this case, the ratio is 1 because we have the optimal allocation.

Another way to allocate jobs would be to allocate $j_1, \ldots, j_6$ first and j_7 after. This would yield 2 machines which take 2 hours to complete, and another machine which takes 5 hours to complete its jobs. In this case, the ratio is $\frac{5}{3} < 2$. This is quite close to the guarantee given by the algorithm.

(b) Prove that when $m \leq n$, the algorithm provided returns an optimal allocation.

Solution: We cannot split jobs across servers, so we must assign a server for every job. We have more servers than jobs, so the optimal allocation is to assign every job a server with no job allocated. This is exactly what the greedy algorithm does, so when $m \leq n$, the greedy algorithm returns an optimal allocation.

More formally, suppose an optimal allocation assigns more than one job to some machines when there are more machines than jobs. Then we can obtain an allocation that takes no more time than the optimal by taking one of the jobs from a machine with more than one job and allocating that to a machine with no job allocated. This process can be repeated until every machine is allocated at most a single job. The final allocation produced by this process is exactly the output of the greedy algorithm (up to renaming of the machines).

(c) Describe the relationship of the quantities $\max\{t_1, t_2, \ldots, t_m\}$ and $\frac{1}{n} \cdot \sum_{j=1}^{m} t_j$ to the time which an optimal allocation takes to complete all tasks when $m > n$.

Solution: The soonest that any allocation can finish is at least $\max\{t_1, t_2, \ldots, t_m\}$. This is because the longest job must be assigned to some machine - that machine then takes time at least $\max\{t_1, t_2, \ldots, t_m\}$ to complete, so the optimal allocation takes at least time $\max\{t_1, t_2, \ldots, t_m\}$ to complete. This means that

$$\mathsf{OPT} \geq \max\{t_1, t_2, \ldots, t_m\}.$$

Similarly, the best case is when every machine does exactly $\frac{1}{n} \cdot \sum_{j=1}^{m} t_j$. It turns out that OPT is lower bounded by this quantity as well. To see this, suppose $\mathsf{OPT} < \frac{1}{n} \cdot \sum_{j=1}^{m} t_j$. Then every machine takes less than $\frac{1}{n} \sum_{j=1}^{m} t_j$ time to finish its work. But if we sum up the time for every machine (of which there are n), the total time used is less than the total time $\sum_{j=1}^{m} t_j$ required to complete all the jobs, which is an impossibility. This means that

$$\mathsf{OPT} \geq \frac{1}{n} \sum_{j=1}^{m} t_j.$$

(d) Prove that the server which finishes its allocated jobs last takes at most twice the optimal. Conclude that the algorithm is a 2-approximation.
Hint: Consider how the jobs must have been allocated before this server's last job was allocated.

Solution: By the hint, consider how the allocation must have looked before this server's last job was allocated.

It is not possible for this machine to have needed to spent strictly more time than $\frac{1}{n} \cdot \sum_{j=1}^{m} t_j$ for its currently allocated jobs. To see this, suppose this was the case. Then, by the design of the algorithm, we know that every machine takes strictly more than $\frac{1}{n} \cdot \sum_{j=1}^{m} t_j$ time, since the other machines take at least as much time as the server that was selected. Summing over all machines, the total time taken is more than the total time the jobs should take collectively - which is not possible.

So we know that the machine has at most $\frac{1}{n} \cdot \sum_{j=1}^{m} t_j$ work assigned to it before its final job is allocated. When the final job is allocated, at most, it obtains $\max\{t_1, t_2, \ldots, t_m\}$ work. We can bound the time this machine takes as follows:

$$\text{time} \leq \frac{1}{n} \cdot \sum_{j=1}^{m} t_j + \max\{t_1, t_2, \ldots, t_m\} \leq 2\mathsf{OPT}$$

Applying the results from (c), we have that the machine takes at most twice OPT. Combining this result and the result of (b), we obtain that the algorithm is a 2-approximation.

E 6. NP-Complete problems are everywhere, so it is likely you have already encountered one in your other coursework. Go to

`https://piazza.com/class/ke0kpz5jy57110?cid=1510`

and post a follow-up note describing an NP-Complete problem you have encountered previously. Describe the context or motivation for the problem, and how its NP-Completeness can be addressed, if possible (e.g., approximation, heuristics, restricting to special inputs, etc).

- **Include your post in your homework submission as well.**
- **Do not use a problem encountered in EECS 281.**

Bonus points will be awarded to the most original NP-Complete problem!

Solution:

Optional bonus questions

For bonus questions, we will not provide any insight during office hours or Piazza, and we do not guarantee anything about the difficulty of these questions. *Only attempt these questions **after** you have finished the rest of the homework.*

1. Consider the following language:

$$\text{SmallerFactor} = \{(x, y) : \ x \text{ has a factor, } z, \text{ such that } 1 < z < y\}.$$

Show that SmallerFactor $\in$ coNP.
Hint: You can assume that it is possible to verify if a given number is prime in polynomial time.

Submit your answer to this question on Gradescope.

Solution: In order to show that SmallerFactor $\in$ coNP, we need to show that $\overline{\text{SmallerFactor}} \in$ NP. Thus, the language we are trying to reason about is the following:

$$\overline{\text{SmallerFactor}} = \{(x, y) : \ x \text{ does NOT have a factor, } z, \text{ such that } 1 < z < y\}.$$

In order to show that $\overline{\text{SmallerFactor}} \in$ NP, we must construct an efficient verifier. Our certificate P will be a list of integers that contain the prime factorization of x.

$D =$ "On input $((x, y), P)$:
1. If $x = 0$ and $y > 2$, $REJECT$.
2. If $x = 0$ and $y \leq 2$, $ACCEPT$.
3. Verify that the elements of P multiply to form x. If not, $REJECT$.
4. For each element $p \in P$:
5. Verify that p is prime. If not, $REJECT$
6. Verify that $p \geq y$. If not, $REJECT$
7. $ACCEPT$"

Of important note is the fact that each number has a unique prime factorization. Furthermore, if all the prime factors are larger than y, then any of the composite factors of x must also be larger than y. This is because the composite factors of x can be further broken down into the product of primes. Also note that negative numbers, 0, and 1 are all not prime.

Suppose $x \neq 0$. We will handle the case where $x = 0$ separately as all integers divide 0 - in other words, all integers are a factor of 0.

$((x, y), P) \in \overline{\text{SmallerFactor}} \implies$ all factors of P are prime and are $\geq y \implies$ each prime factor of x is greater than $y \implies$ V $ACCEPTS$.

$((x, y), P) \notin \overline{\text{SmallerFactor}} \implies$ one of the factors of P is between 1 and y or is not prime $\implies$ V $REJECTS$.

Suppose $x = 0$. We will not need the certificate in this case. Because every integer is a factor of x, any integer which satisfies $1 < z < y$ is a factor of x. Then it is sufficient to

reject as soon as we know that there is some integer z satisfying $1 < z < y$. This occurs exactly when $y > 2$. Suppose instead $y \leq 2$. Then no integer satisfies $1 < z < y$, so x has no factor which satisfies the inequality and our verifier should accept.

We must also verify that our algorithm runs in polynomial time. First let us explain why our certificate must be polynomial in size. We know that the prime factorization of x will have at most $\log_2(x)$ numbers because the longest prime factorization would be $2 \cdot 2 \cdot 2 ... \log_2(x)$ times. Thus, with respect to the size of our input x, the size of our certificate is $log_2(2^{|x|}) = |x|$. Thus, the certificate is polynomial in size in terms of the input so reading through it will be efficient. Now let's analyze the verifier to ensure it must also be efficient. Multiplying and comparing all the values of P is also polynomial in time as we have already specified that P is polynomial in size and we have previously proven that multiplication is efficient. Thus, our verifier is efficient.

2. Pick any NP-hard optimization problem (could be a problem that we didn't cover) and suggest a **new** approximation algorithm for that problem, supplemented with a formal analysis.

   For this question, in order to earn the points, your algorithm has to be **new** (i.e. do not just use an existing algorithm), although a **different** algorithm with the same (or even better) approximation ratio for the problem you chose may already exist. In this case, include a reference to the other algorithm(s). Submit everything in a single file via e-mail directly to Professor Volkovich at ilyavol@umich.edu.