# CSC373 Fall'20
# Final Assessment Solutions
Date: December 18, 2020

## Q1 [10 Points] Find the Single Attendee

There are $2n + 1$ attendees at a party, which includes $n$ couples and a single person. At the end of the party, all the attendees form a line in which each person stands next to their partner, except for the single person, who stands somewhere in the line. As an example, for $n = 3$, the seven attendees could be standing in the order $(A_1, A_2, B_1, B_2, C, D_1, D_2)$, where $(A_1, A_2)$, $(B_1, B_2)$, and $(D_1, D_2)$ are couples and $C$ is single.

Your job is to find the position of the single person (this would be 5 in the above example). But you don't know which ones are partners. All you can do is ask questions of the form "Are the $i$-th and $j$-th people in the line partners?" Design a divide-and-conquer algorithm for this problem which finds the position of the single person by asking $O(\log n)$ questions. Justify your answer.

## Solution to Q1

Suppose $A$ is the array of attendees. The key idea is to compare two attendees about half-way in the array. Suppose we compare $A[n]$ and $A[n+1]$ and they are a couple. If $n$ is odd, then $A[1 \dots n+1]$ is of even length, which means it must contain $(n+1)/2$ couples and not the singleton. So we can search $A[n+2 \dots 2n+1]$ for the singleton. Similarly, if $n$ is even, then $A[1 \dots n+1]$ is of odd length, which means it must contain $n/2$ couples and the singleton. So we can search $A[1 \dots n-1]$ for the singleton (since we already know that $A[n]$ and $A[n+1]$ are not the singleton). Similar conclusions hold if $A[n]$ and $A[n+1]$ are not a couple. Also, note that we are careful to always call our algorithm on an array of odd length that contains the singleton.

**Algorithm 1:** Find-the-Singleton

**Input:** Array $A$ of length $2n + 1$

**if** $n = 0$ **then**
    **return** 1
**end**
**if** *If* $A[n]$ *and* $A[n+1]$ *are a couple* **then**
    `// Search the second half if` $n$ `is odd and the first half if` $n$ `is even.`
    **return** $(n+1)$+Find-the-Singleton($A[n+2 \dots 2n+1]$) if $n$ is odd and Find-the-Singleton($A[1 \dots n-1]$) if $n$ is even
**else**
    `// Search the second half if` $n$ `is even and the first half if` $n$ `is odd.`
    **return** $n$+Find-the-Singleton($A[n+1 \dots 2n+1]$) if $n$ is even and Find-the-Singleton($A[1 \dots n]$) if $n$ is odd
**end**

For the worst-case number of questions, note that solving a list of length $2n + 1$ requires solving a list of length at most $n + 1$ and a single additional question. Hence, we have the recurrence relation

$T(2n+1) \le T(n+1) + 1$, which, by the master theorem, gives us $T(n) = O(\log n)$.

**Q2 [15 Points] Event Planner**

There are $n$ events, each takes one unit of time. Each event $i$ will provide a profit of $g_i$ dollars if it is started at or before time $t_i$, but will provide zero profit if it is not started by time $t_i$ (so there is no point in scheduling event $i$ unless it can be scheduled to start by time $t_i$). Here, $g_i, t_i \ge 0$ and $t_i$ may $NOT$ be an integer. An event can start as early as time 0 and no two events can be running simultaneously. The goal is to feasibly schedule a subset of the events to maximize the total profit.

**(a)** [2.5 Points] Prove that there exists an optimal schedule $OPT$ in which every event that is scheduled is scheduled to start at an integral time. Note that in such a solution, each event $i$ is either scheduled to start by time $\lfloor t_i \rfloor$ or not scheduled at all.

**(b)** [5 Points] Design an efficient greedy algorithm which only schedules events at integral start times. [Hint: Let $T = \max_i \lfloor t_i \rfloor$. Think about which event you would schedule to start at time $T$.]

**(c)** [5 Points] Prove that your algorithm always returns an optimal solution.

**(d)** [2.5 Points] Analyze the worst-case running time of your algorithm. Explicitly state the data structures that your algorithm uses.

**Solution to Q2**

**(a)** Consider any optimal schedule $OPT'$ which schedules a subset of the events $S$ and each $i \in S$ is scheduled to start at $s'_i$. Next, consider the schedule $OPT$ which also schedules the same set of events $S$ but schedules each $i \in S$ to start at $s_i = \lfloor s'_i \rfloor$.

Since $s_i \le s'_i$ for each $i \in S$, we know that each event in $S$ is still scheduled profitably. Thus, $OPT$ has the same profit as $OPT'$ and it only schedules events to start at integral times. It remains to show that no two events are overlapping in $OPT$. But since events start at integral times and run for one unit of time, this is equivalent to proving that no two events have the same starting time under $OPT$.

To see this, consider any two events $i, j \in S$. Since $OPT'$ is feasible, $[s'_i, s'_i + 1)$ and $[s'_j, s'_j + 1)$ must not overlap. This directly implies that $\lfloor s'_i \rfloor \ne \lfloor s'_j \rfloor$, i.e., $s_i \ne s_j$, as required.

**(b)** We sort the events by their start deadlines and then, in a single pass, divide them into blocks $E_0, \ldots, E_T$ such that for each $k \in \{0, 1, \ldots, T\}$, $E_k = \{i : k \le t_i < k+1\}$. Note that events in $E_k$ are profitable if started at time $k$ or earlier but not if started at time $k+1$ or later. And note that $T$ is the latest time at which we can start an event profitably.

Only events in $E_T$ can be scheduled at time $T$. Among them, we schedule the most profitable one at time $T$. Then, we consider the unscheduled events in $E_T$ along with the events in $E_{T-1}$, and schedule the most profitable among them at time $T-1$. We continue doing this until we reach time 0. This is explained in the algorithm below.

At time $k$, to find the most profitable event among the unscheduled events in $E_{k+1}, \ldots, E_T$ along with the events in $E_k$, we maintain a priority queue of events from which we can find the most profitable one and delete the scheduled one quickly.

**Algorithm 2:** Greedy-Event-Scheduling

1 Sort the events so that $t_1 \leq \ldots \leq t_n$<br>
2 $T \leftarrow \max_i \lfloor t_i \rfloor = \lfloor t_n \rfloor$<br>
3 Divide the events into buckets $E_0, \ldots, E_T$ such that $E_k = \{i : k \leq t_i < k+1\}$ for each $k \in \{0, 1, \ldots, T\}$<br>
4 $Q \leftarrow$ empty priority queue<br>
5 **for** $t = T, T-1, \ldots, 0$ **do**<br>
6     Add all events in $E_t$ to $Q$ with their profit as the key<br>
7     **if** $Q$ *is empty* **then**<br>
8         **continue**<br>
9     **end**<br>
10     $i \leftarrow$ most profitable event in $Q$<br>
11     Schedule event $i$ to start at time $s_i = t$<br>
12     Delete event $i$ from $Q$<br>
13 **end**

**(c)** We say that a schedule is integral if it only schedules events at integral start times. Part (a) shows that there exists an optimal schedule that is integral. We say that two integral schedules *match* at time $t$ if either both schedule the same event at $t$ or both do not schedule any event at $t$. The *match level* of two integral schedules is the smallest $t$ for which they match at time $t$.

Let $G$ denote our greedy schedule. Among all optimal integral schedules, let $OPT$ be the one with the smallest match level with $G$. If this match level is 0, then the greedy schedule is optimal, so we are done. Otherwise, suppose this match level is $t+1$. Consider time $t$. There are three possibilities:

1. $OPT$ schedules nothing at time $t$ while $G$ schedules some event $i$. Then, $i$ must be scheduled at some other time in $OPT$, otherwise scheduling $i$ at time $t$ would not cause any conflicts and increase the profit, which is impossible. Now, changing the start time of $i$ to $t$ produces an integral optimal schedule $OPT'$ which has the same profit (event $i$ remains profitable when started at time $t$ because $G$ schedules it at time $t$) and has match level $t$ with $G$, a contradiction.
2. $OPT$ schedules some event $i$ at time $t$ while $G$ schedules nothing. Since $G$ and $OPT$ match at times $t+1, \ldots, T$, $i$ must be unscheduled when the greedy algorithm reaches iteration for time $t$. Since $i$ is profitable if scheduled at $t$, $G$ cannot schedule nothing at time $t$, a contradiction.
3. $OPT$ schedules some event $i$ at time $t$ while $G$ schedules a different event $j$. Since $G$ and $OPT$ match at times $t+1, \ldots, T$, neither have $i$ or $j$ scheduled after time $t$. Further, since $i$ must be unscheduled at the iteration for time $t$ in $G$, but it schedules event $j$, the profit of $j$ must be at least as much as the profit of $i$. So if $OPT$ doesn't schedule $j$, then we can replace $i$ by $j$ at time $t$ in $OPT$, and if $OPT$ does schedule $j$ at an earlier time, then we can swap the starting times of $i$ and $j$. Note that $j$ still remains profitable since it has the same starting time as in $OPT$, and $i$ only moves early so remains profitable as well. In either case, we have a new integral optimal schedule with match level $t$ with $G$, a contradiction.

**(d)** Sorting and grouping the events by the floor of their start time (Lines 1-3) takes $O(n \log n)$

time. Creating the buckets then takes $O(n + T)$ time. The loop runs for $O(T)$ iterations, and in each iteration, finding the most profitable event and deleting the scheduled event from the priority queue takes $O(\log n)$ time. Hence, the total running time is $O((n + T) \log n)$, which is not polynomial in the input length because $T$ can be quite large.

However, we can slightly modify the algorithm such that we only create and store non-empty buckets, and every time we have an empty $Q$ in Line 7, we reduce $t$ directly to the time of the next non-empty bucket. Thus, we can reduce the number of iterations to the order of the number of events scheduled, which is $O(n)$. This reduces the running time to $O(n \log n)$, which is polynomial. While the loop runs for $O(T)$ steps as stated, we can slightly modify it to skip over all the consecutive trivial steps (i.e. steps in which $Q$ is empty in Line 7), thus

**Q3 [15 Points] Protect the Paintings Again**

Recall the question about protecting paintings from midterm 1. A corridor of a museum is represented by the interval $[a, b]$ (with $a < b$) and contains valuable paintings. There are $n$ guards stationed along the corridor. Guard $i$ can protect the interval $[s_i, f_i]$, where $a \le s_i \le f_i \le b$. We say that a subset of guards $P \subseteq \{1, \ldots, n\}$ is *acceptable* if the guards in $P$ already collectively protect the entire corridor, i.e., $\cup_{i \in P}[s_i, f_i] = [a, b]$. Assume that the set of all guards $\{1, \ldots, n\}$ is acceptable, so there is at least one acceptable set.

In the midterm, we designed a greedy algorithm for finding an acceptable subset $P$ of *minimum cardinality* $|P|$. Instead, suppose that each guard $i$ has an associated non-negative cost $c_i$. Design a dynamic programming solution for finding an acceptable subset $P$ with the smallest total cost $\sum_{i \in P} c_i$. For full credit, your solution must run in $O(n^2)$ time and space.

[Hint: Consider the set of all the "breakpoints": $\{a, b, s_1, f_1, s_2, f_2, \ldots, s_n, f_n\}$. Suppose the *distinct* breakpoints in the ascending order are $a = p_1 < p_2 < \ldots < p_m = b$ for some $m$. It may be useful to think of a subproblem where you want to cover the sub-interval $[p_1, p_j]$ using only some of the guards. Do not forget to bound the maximum number of distinct breakpoints $m$ in terms of $n$.]

**(a)** [5 Points] Define an array storing the necessary information from subproblems. Clearly define what each entry means and how you would compute the desired solution given this array.

**(b)** [5 Points] Write a Bellman equation and briefly justify its correctness.

**(c)** [2.5 Points] In what order would you compute the entries in a bottom-up implementation?

**(d)** [2.5 Points] Analyze the worst-case running time and space complexity of your algorithm.

**Solution to Q3**

**(a)** Sort the guards such that $f_1 \le \ldots \le f_n$. Further, as the hint suggests, sort the distinct breakpoints such that $a = p_1 < \ldots < p_m = b$. Now, for $0 \le i \le n$ and $1 \le j \le m$, define $OPT[i, j]$ to be the smallest cost needed to cover $[a, p_j]$ (with $j = 1$, i.e., $[a, a]$ considered trivially covered) using only the first $i$ guards in the sorted order.

To reconstruct the optimal solution, we look at the Bellman equation below and define $S[i, j]$ to be $Y$ if guard $i$ is used, $N$ if guard $i$ is not used, and $\perp$ in the first two edge cases.

Then, to construct the final solution, we start $P = \emptyset, i = n, j = m$. Then, until $S[i,j] = \bot$, we do the following:

- If $S[i,j] = Y$, then $P \leftarrow P \cup \{i\}, i \leftarrow i-1, j \leftarrow k$ (where $p_k = s_i$).
- If $S[i,j] = N$, then $i \leftarrow i-1$.

At the end, we return $P$.

**(b)** The Bellman equation is as follows.

$$(OPT[i,j], S[i,j]) =$$
$$\begin{cases} (0, \bot) & \text{if } j = 1, \\ (\infty, \bot) & \text{if } j \geq 2, i = 0, \\ (OPT[i-1,j], N) & \text{if } j \geq 2, i \geq 1, p_j \notin [s_i, f_i], \\ (OPT[i-1,j], N) & \text{if } j \geq 2, i \geq 1, p_j \in [s_i, f_i], OPT[i-1,j] < c_i + OPT[i-1,k], \text{ where } p_k = s_i, \\ (c_i + OPT[i-1,k], Y) & \text{if } j \geq 2, i \geq 1, p_j \in [s_i, f_i], OPT[i-1,j] \geq c_i + OPT[i-1,k], \text{ where } p_k = s_i. \end{cases}$$

Note that choosing guard $i$ can only be helpful if $p_j \in [s_i, f_i]$: if $p_j < s_i$, then the guard doesn't cover any useful portion, and if $p_j > f_i$, then due to the sorted order, none of guards $1, \ldots, i$ can cover point $p_j$ (so our recursive solution will keep calling $OPT$ with one smaller $i$ until it reaches $i = 0$ and returns $\infty$). If choosing guard $i$ can be helpful, then we want to consider both choosing guard $i$ (in which case we only have interval $[a, s_i]$ left to be covered) and not choosing guard $i$ (in which case we still need to cover $[a, p_j]$ with only guards $1, \ldots, i-1$).

**(c)** Since $(OPT[i,j], S[i,j])$ only depends on $OPT[i-1, \cdot]$, we compute them in the following order: loop over $i = 0, \ldots, n$, and for each $i$, loop over $j = 1, \ldots, m$.

**(d)** Since there are at most $2n$ breakpoints, we have $m = O(n)$. Hence, both arrays require $O(n^2)$ space. Further, computing each array entry requires $O(1)$ times given previous entries. Hence, the worst-case running time is $O(n^2)$ as well.

### Q4 [15 Points] Divide the Workload

You are the CEO of a company which employs $n$ workers to perform $m$ tasks. Each worker $i$ is supposed to work a total of $w_i$ hours and each task $j$ requires a total of $t_j$ hours of work. Assume that $\sum_{i=1}^{n} w_i = \sum_{j=1}^{m} t_j$. The floor supervisor has come up with an ideal work schedule represented as matrix $A$, where row $i$ represents worker $i$, column $j$ represents task $j$, and $A_{i,j}$ is the number of hours worker $i$ will spend on task $j$. Matrix $A$ has the property that the sum along each row $i$ is exactly $w_i$ and the sum along each column $j$ is exactly $t_j$.

There is just one problem. The floor supervisor has taken the liberty of using fractional values for $A_{i,j}$-s, forgetting the recent company policy that a worker must spend an *integral* number of hours on a task. Luckily, all the $w_i$-s and $t_j$-s are integral. Your goal is to prove that it is always possible to "round" matrix $A$ into some matrix $B$ while preserving the row and column sums (i.e. set each $B_{i,j}$ to be either $\lfloor A_{i,j} \rfloor$ or $\lceil A_{i,j} \rceil$ such that each row $i$ of $B$ still sums to $w_i$ and each column $j$ of $B$ still sums to $t_j$). The example below shows such a rounding of a $3 \times 3$ matrix.

$$A = \left[\begin{array}{ccc|c} 2.6 & 0 & 0.4 & 3 \\ 0.8 & 2.9 & 1.3 & 5 \\ 1.6 & 0.1 & 5.3 & 7 \\ \hline 5 & 3 & 7 & \end{array}\right] \longrightarrow B = \left[\begin{array}{ccc|c} 2 & 0 & 1 & 3 \\ 1 & 3 & 1 & 5 \\ 2 & 0 & 5 & 7 \\ \hline 5 & 3 & 7 & \end{array}\right]$$

**(a)** [2.5 Points] Consider the matrix $A'$ obtained by replacing each entry of $A$ with its fractional part (e.g. replacing 1.3 with 0.3, 2.6 with 0.6, 0.1 with 0.1, etc). First, argue that $A'$ must also have *integral* row and column sums. Next, argue that if $A'$ can be rounded while preserving the row and column sums, then $A$ can be as well.

**(b)** [10 Points] Note that each $A'_{i,j} \in [0, 1]$; hence, rounding it means setting it to either 0 or 1 (except, if $A'_{i,j} \in \{0, 1\}$ then the rounding must not change its value). Using network flow techniques, show that $A'$ can be rounded while preserving row and column sums. Justify your answer.

[Hint: Construct a network with integral edge capacities, use $A'$ to construct a max flow with fractional flow values on edges, and then use the integrality property of the Ford-Fulkerson algorithm (i.e. that it finds a max flow in which each edge carries an integral amount of flow).]

**(c)** [2.5 Points] What is the worst-case running time of the naïve Ford-Fulkerson on your network?

**Solution to Q4**

**(a)** Let $F$ be the matrix where $F_{i,j} = \lfloor A_{i,j} \rfloor$. Then, $A' = A - F$. Since row/column sums of both $A$ and $F$ are integral, the row/column sums of $A'$ are also integral.
If $A'$ can be rounded into $B'$ while preserving the row/column sums, then note that $B = F + B'$ gives a rounding of $A$: it has the same row/column sums as that of $F + A' = A$ and each of its entries is either $\lfloor A_{i,j} \rfloor + 0$ or $\lfloor A_{i,j} \rfloor + 1$ (except it must be equal to $\lfloor A_{i,j} \rfloor = A_{i,j}$ if $A_{i,j}$ is an integer, i.e., if $A'_{i,j} = 0$).

**(b)** Construct a network as follows.

- Add a source node $s$ and a target node $t$.
- Add a vertex $r_i$ for each row $i$ and a vertex $c_j$ for each column $j$.
- Add an edge $s \to r_i$ for each $i$ with capacity equal to the sum of row $i$ in $A'$.
- Add an edge $c_j \to t$ for each $j$ with capacity equal to the sum of column $j$ in $A'$.
- Add a unit-capacity edge $r_i \to c_j$ for each $(i, j)$ with $A'_{i,j} > 0$ (i.e. those entries which can be rounded to 1).

Consider the following flow $f$. Every $s \to r_i$ and $c_j \to t$ edge is saturated, and $f_{r_i \to c_j} = A'_{i,j}$ for each $(i, j)$. Note that because the capacities of $s \to r_i$ and $c_j \to t$ edges are the sums of entries of row $i$ and column $j$ in $A'$, flow conservation constraints are satisfied. Edge capacity constraints are also trivially satisfied. Hence, $f$ is a valid flow. Further, since all edges leaving $s$ are saturated, it must be a max flow. Hence, max flow value is $\sum_{i,j} A'_{i,j}$.

However, since the network has edges of integral capacity, the Ford-Fulkerson algorithm must return a flow $f^*$ with integral flow values on edges. Define $B'$ such that $B'_{i,j} = 1$ if $f^*_{r_i \to c_j} = 1$ and $B'_{i,j} = 0$ otherwise. Then, due to the construction of the network, $B'$ must be a rounding of $A'$.

**(c)** The network in part (b) has at most $n+m+n\cdot m = O(n\cdot m)$ edges, $n+m+2 = O(n+m)$ nodes, and max flow value of $\sum_{i,j} A'_{i,j} \leq n \cdot m$. Hence, the worst-case running time of the Ford-Fulkerson algorithm is $O(n^2 m^2)$.

## Q5 [15 Points] Linear Programming

**(a)** [5 Points] Convert the following linear program to the standard form. You only need to write the final answer; no justification is needed.

$$\begin{array}{ll} \max & 3x + 5y + 2z \\ \text{s.t.} & 5y + 10z \leq 3 - 2x \\ & 2x \leq 2 - 3y - z \\ & x, y \geq 0, z \in \mathbb{R} \end{array}$$

**(b)** [5 Points] Write the dual of the linear program from part (a). You do *not* need to write this in the standard form and no justification is needed.

**(c)** [5 Points] Consider the optimization problem from part (a), but change the objective function to maximizing $f(x, y, z)$, where

$$f(x,y,z) = \begin{cases} 3x + 5y, & \text{if } z \geq 0, \\ 3x + 5y + 2z, & \text{if } z < 0. \end{cases}$$

Note that $f(x, y, z)$ is *not* linear, and hence, the new optimization problem is not linear as well. Nonetheless, show that it can be converted into an equivalent *linear* program. Provide this equivalent linear program in its standard form and justify the equivalence.

## Solution to Q5

**(a)**

$$\begin{array}{ll} \max & 3x + 5y + 2z' - 2z'' \\ \text{s.t.} & 2x + 5y + 10z' - 10z'' \leq 3 \\ & 2x + 3y + z' - z'' \leq 2 \\ & x, y, z', z'' \geq 0 \end{array}$$

**(b)** Both the dual of the original LP and the dual of the LP in the standard form would be acceptable in this part.

Dual of the original LP:

$$\begin{array}{ll} \min & 3a + 2b \\ \text{s.t.} & 2a + 2b \geq 3 \\ & 5a + 3b \geq 5 \\ & 10a + b = 2 \\ & a, b \geq 0 \end{array}$$

In the dual of the standard form LP, the $10a+b=2$ constraint would be replaced by two constraints: $10a+b \geq 2$ and $-10a-b \geq -2$.

**(c)** We can use the trick from part (a) where we replace the unrestricted $z$ by $z'-z''$ with non-negative variables $z'$ and $z''$, and then optimize the linear objective function $3x+5y-2z''$.

$$\begin{array}{ll} \max & 3x+5y-2z'' \\ \text{s.t.} & 2x+5y+10z'-10z'' \leq 3 \\ & 2x+3y+z'-z'' \leq 2 \\ & x,y,z',z'' \geq 0 \end{array}$$

The idea is that if the optimal solution of the original program has $z \geq 0$, then the corresponding optimal solution in the new LP will set $z''=0$, making the objective $3x+5y$. And if the optimal solution of the original program has $z<0$, then the corresponding optimal solution in the new LP will set $z''=-z$, making the objective $3x+5y-2z''=3x+5y+2z$.

Formally, we can show that the optimal values of the two programs are equal by showing that each is at least the other. If $(x,y,z)$ is an optimal solution of the original program, then note that $(x,y,z',z'')$ is a feasible solution of the new LP with the same objective value, where, if $z \geq 0$ then $z'=z$ and $z''=0$, and if $z<0$, then $z'=0$ and $z''=-z$. Similarly, if $(x,y,z',z'')$ is an optimal solution of the new LP, then $(x,y,z=z'-z'')$ is a feasible solution of the original program. To claim that it has the same objective value, we need to show that $z''>0$ implies $z'=0$. This is true because if both are positive, then reducing both by a small amount $\delta$ yields a feasible solution with a better objective value, a contradiction.

### Q6 [20 Points] SAT

Recall that a CNF formula $\varphi = C_1 \wedge \ldots \wedge C_m$ is a conjunction of clauses, where each clause is a disjunction of (any number of) literals. Recall the NP-complete problem SAT.

**SAT:**
**Input:** A CNF formula $\varphi$.
**Question:** Does $\varphi$ have a satisfying assignment?

Now, consider the following two variants of it.

**TripleSAT:**
**Input:** A CNF formula $\varphi$.
**Question:** Does $\varphi$ have at least *three* different satisfying assignments?

**TwoThirdsSAT:**
**Input:** A CNF formula $\varphi$.
**Question:** Is there an assignment satisfying at least two-thirds ($2/3$) of the clauses of $\varphi$?

**(a)** [3 Points] Prove that TripleSAT is in NP.

**(b)** [7 Points] Prove that TripleSAT is NP-hard through a reduction from SAT.

**(c)** [3 Points] Prove that TwoThirdsSAT is in NP.

**(d)** [7 Points] Prove that TwoThirdsSAT is NP-hard through a reduction from SAT.

**Solution to Q6**

**(a)** We can provide, as advice, three different satisfying assignments of $\varphi$.

**(b)** Given an instance $\varphi$ of SAT, construct an instance $\varphi'$ of TripleSAT by adding two fresh variables $x_1, x_2$ and adding a clause $(x_1 \vee x_2)$. Note that satisfying assignments of $\varphi'$ are formed by taking satisfying assignments of $\varphi$ and appending $(x_1, x_2) = (T, T)$, $(T, F)$, or $(F, T)$. Hence, the number of satisfying assignments of $\varphi'$ is exactly three times the number of satisfying assignments of $\varphi$, which implies that $\varphi'$ has at least three satisfying assignments if and only if $\varphi$ has a satisfying assignment, implying that both instances have the same answer.

**(c)** We can provide, as advice, an assignment that satisfies at least two-thirds of the clauses of $\varphi$.

**(d)** Given an instance $\varphi$ of SAT with $n$ variables and $m$ clauses, construct an instance $\varphi'$ of TwoThirdsSAT by adding $m$ fresh variables $x_1, \ldots, x_m$ and $2m$ clauses $x_1, \bar{x}_1, \ldots, x_m, \bar{x}_m$. Note that $\varphi'$ has $3m$ clauses and every assignment satisfies exactly $m$ of the $2m$ newly added clauses. Hence, an assignment of $\varphi'$ satisfies at least $2m$ of its clauses if and only if the corresponding assignment of $\varphi$ satisfies all its $m$ clauses. Hence, both instances have the same answer.

**Q7 [15 Points] Sabotage!**

There is an undirected graph $G = (V, E)$, where nodes in $V$ are servers and edges in $E$ are cables running between pairs of servers. A set of $k > 2$ servers $S = \{v_1, \ldots, v_k\} \subseteq V$ is trying to collaboratively solve a problem and you want to sabotage this!

Specifically, you want to remove a subset of edges $T \subseteq E$ such that all nodes in $S$ become disconnected from one another (i.e. there is no path left between any two of them). You want $|T|$ to be as small as possible. Consider the following greedy algorithm.

---

**Algorithm 3:** Greedy-Sabotage

---

**for** $i = 1, \ldots, k$ **do**
    Let $E_i$ be the smallest subset of edges we need to remove to disconnect $v_i$ from every other node in $S$. (It turns out that $E_i$ can be computed efficiently, but do not worry about this.)
**end**
Remove the union of $k - 1$ smallest $E_i$-s (i.e. the union of all but the largest $E_i$).

---

**(a)** [5 Points] Prove that Greedy-Sabotage returns a feasible solution $T$ (i.e. removing the set of edges $T$ it returns will indeed disconnect every pair of nodes in $S$).

**(b)** [10 Points] Prove that Greedy-Sabotage achieves a $2 - 1/k$ approximation ratio. For partial credit, prove a slightly weaker approximation ratio of 2.

[Hint: Let $T^*$ denote the optimal solution. For each $i \in \{1, \ldots, k\}$, let $V_i \subseteq V$ denote the set of nodes in the connected component containing $v_i$ (but no other node in $S$) that remains after removing $T^*$. Can you relate the number of edges in $T^*$ with one endpoint in $V_i$ with $|E_i|$?]

**Solution to Q7**

**(a)** Let $T$ be the set of edges removed by Greedy-Sabotage. Consider any two nodes $a, b \in S$. Since $T$ is the union of $k-1$ of the $E_i$-s, we have that either $E_a \subseteq T$ or $E_b \subseteq T$. Hence, either $a$ or $b$ must be disconnected from all other nodes in $S$ after removal of $T$, which implies that $a$ and $b$ are not connected to each other after removal of $T$. Hence, Greedy-Sabotage returns a feasible solution.

**(b)** Let $E_i^*$ be the number of edges in $T^*$ with one endpoint in $V_i$. Since removing $E_i^*$ disconnects $V_i$ from the rest of the graph, it also disconnects $v_i$ from the other nodes in $S$. Since $E_i$ is the smallest set that does this, we have $|E_i^*| \geq |E_i|$.

Note that $2T^* \geq \sum_i |E_i^*| \geq \sum_i |E_i|$, where the first transition holds because each edge in $T^*$ is counted at most twice (once for each endpoint) in the sum on the RHS. Since $T$ omits the $E_i$ with the largest cardinality (which must have cardinality at least $(1/k)\sum_i |E_i|$), we have that $T \leq (1-1/k) \cdot \sum_i |E_i|$. Hence, $2(1-1/k) \cdot T^* \geq T$, which means our greedy algorithm achieves an approximation factor of $2 - 2/k$ (which is actually better than $2 - 1/k$).