Name : **Roll No.:**

Analysis and Design of Algorithms CSE222 Winter 2022

End-Semester Examination
Time 120 mins. Full Marks : 80

Please write solutions independent of each other. This is a closed book test. You can not use books or lecture notes. Please note that your solution must fit in the space provided. Extra sheet will be provided only *for roughwork . So try to be precise and brief. Meaningless blabber fetches negative credit. Also, you can use anything as a subroutine that was taught in the lectures or tutorials.*

Part A

Question	1	2	3	4	Total
Points					

Part B

Question	1	2	3	Total
Points				

Total Marks:

Part A

1. (20 points) Write whether the following statements are true (**T**)/false (**F**) (no negative marking)

   (a) Let $f, g : \mathbb{N} \to \mathbb{N}$. Then, $f(n) + g(n) = \Theta(\min\{f(n), g(n)\})$.
   Answer: False

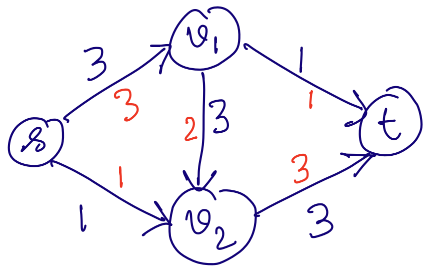
   (b) Let F_n denotes the n'th fibonacci number, i.e. $F_0 = 0, F_1 = 1$, and for all $n \geq 2$, we have $F_n = F_{n-1} + F_{n-2}$. Then, F_n can be computed in $O(n)$-time.
   Answer: True.

   (c) Consider an edge weighted undirected connected graph G such that all edge weights are positive and distinct. Let e^* be the edge with smallest weight in G. Then, G has a minimum spanning tree that does not contain the edge e^*.
   Answer: False

(d) Let $G = (V, E)$ be a flow network, with a source s, a sink t, and a positive integer capacity c_e on every edge e. If f is a maximum (s,t)-flow in G, then for all $e \in E$ that goes out of s, $f(e) = c_e$.
Answer: False. Very simple examples show this is false. Suppose the graph is just a path $s - v_1 - v_2 - t$ with capacity of $(sv1) = 2$ and capacity of other two being 1. Then clearly the flow is only 1 and hence (sv_1) is unsaturated.

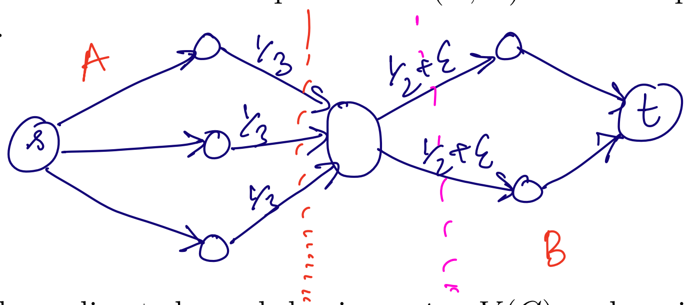
(e) Let $G = (V, E)$ be a flow network with source s, sink t and all the edges of G have odd capacities. Then for any maximum (s,t)-flow f and for any edge $e \in E$, $f(e)$ is either zero or odd.
Answer: False (See example below)

(v_1, v_2) has flow value 2.



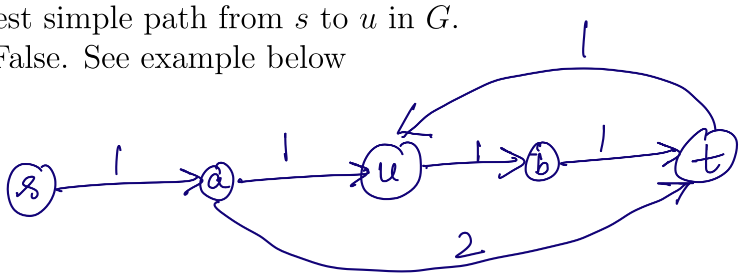
(f) Let $G = (V, E)$ be a flow network, with a source s, a sink t, and a positive capacity c_e on every edge e. Suppose A, B is the *unique* $s - t$ minimum cut in G w.r.t these capacities. Consider a modified graph G' where capacity of every edge is increased by 1. Then A, B is still a minimum cut for G'.

Answer: False. See example below. (A, B) is the unique minimum cut in G but not in G'. .



(A, B) is minimum cut (unique) in G. The pink cut is minimum cut in G'.

(g) Let G be a directed graph having $s, t \in V(G)$ and positive edge weights. Consider a longest simple path P (i.e. a path P with maximum total edge weight) from s to t in G. Let $u \in P$ be an internal vertex in this path P. Then, the subpath of P from s to u is a longest simple path from s to u in G.
Answer: False. See example below



(h) Let u be a vertex in a connected undirected graph G such that removal of u from G creates three connected components. What it means is that $G - \{u\}$ has three connected components. Now consider the DFS-tree T when the depth-first search algorithm is executed starting from u. Then, u will have *exactly* two different subtrees in T.
Answer: False. Suppose C_1, C_2, C_3 are three connected components in $G - \{u\}$. Its clear that DFS(u) will visit each component, backtrack to u and visit the others.

(i) Let G be a connected directed graph with positive edge weights $w : E(G) \to \mathbb{R}$. Suppose that we modify the graph G into G' as follows. For every edge $e \in E(G)$, we set $w'(e) = w(e)/2$ to be the modified weights in G'. Then, every shortest path from s to t in G is a shortest path from s to t in G'.
Answer: True

(j) Let G be a connected directed graph with edge weights $w : E(G) \to \mathbb{R}^+$. Suppose that we modify the graph G into G' as follows. For every edge $e \in E(G)$, we set $w'(e) = (w(e))^2$ to be the modified weights in G'. Then, at least one shortest path from s to t in G is still a shortest path from s to t in G'.
Answer: False. Simple examples exists

2. (4 points each, total 16 points, no negative marking, we want only the answer, no calculation has to be shown) For the first three questions assume that for an input instance of small size $1, 2, 3$ etc, the problem can be solved in $\Theta(1)$-time.

(i) Suppose that an algorithm A partitions a problem instance of size n into 6 subproblems of size $n/6$ each. Then, it recursively solves these subproblems and combines the solutions in $O(n \log n)$-time. Then, what is the tightest asymptotic running time of algorithm A in big-Oh notation?
Answer: $O(n \log^2 n)$.

(ii) Suppose that an algorithm A partitions a problem instance of size n into 2 subproblems, one of size $n/3$ and one of size $2n/3$. Then, it recursively solves these subproblems, and combines the solutions in $O(n)$-time. Then, what is the tightest asymptotic running time of algorithm A in big-Oh notation?
Answer: $O(n \log n)$

(iii) Suppose that an algorithm A partitions the problem instance of size n into 5 subproblems each of size $n/2$, and then combines the solutions in $O(n^3)$-time. Then, what is the tightest asymptotic running time of algorithm A in big-Oh notation?
Answer: $O(n^3)$.

(iv) Given n matrices $A_1, \ldots, A_n$ where the dimension of A_i is $p_{i-1} \times p_i$, we want to compute the product $A_1 \cdot A_2 \cdot \cdots \cdot A_n$ with fewest number of multiplications. Then, a recurrence relation for a dynamic program to solve the problem is given by (tick the correct answer)

(a) $M(i,j) = 1 + \min_{i \le k < j} M(i,k) + M(k,j)$.

(b) $M(i,j) = \min_{i \le k < j} M(i,k) + M(k,j)$.

(c) $M(i,j) = \min_{i \le k < j} p_{k-1}p_kp_{k+1} + M(i,k) + M(k,j)$.

(d) $M(i,j) = \min_{i \le k < j} p_{i-1}p_kp_j + M(i,k) + M(k,j)$.

(e) $M(i,j) = \min_{i \le k < j} p_k + M(i,k) + M(k,j)$.

(f) $M(i,j) = p_ip_j + \min_{i \le k < j} M(i,k) + M(k,j)$.

Answer: Option (d).

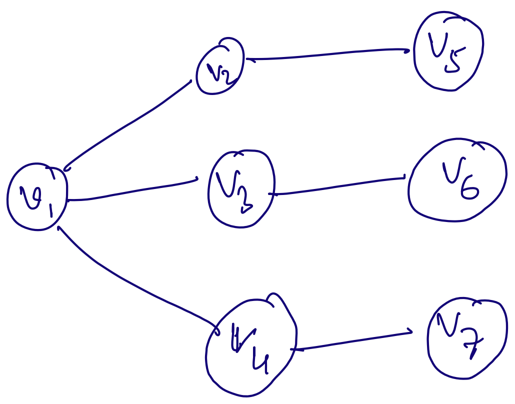
3. (5 points) Given a sequence of n weeks, a plan is specified by a choice of 'low-stress job', 'high-stress job', or 'none' for each of the n weeks. If a 'high-stress' job is chosen for $i > 1$, then 'none' should be chosen for the week $i-1$. But, a high-stress job can be chosen in week 1. However, if a low-stress job can be chosen in a week i, then either of low-stress or high-stress job can be chosen in week $i-1$. A low stress job gives revenue of a_i and a high-stress job gives a revenue of b_i for week i. The goal is to find a plan for n consecutive weeks that maximizes the revenue. Let $Rev[i]$ denotes the value of an optimal revenue for the first i weeks. Then, write a complete recurrence relation and solution to the actual problem for a dynamic programming formulation of this problem.

Solution:
Base Cases: $Rev[0] = 0$ and $Rev[1] = \max(a_1, b_1)$.
Rub: +2. -1 for missing either.
Recurrence: For all $i \ge 2$, $Rev[i] = max\{b_i + Rev[i-2], a_i + Rev[i-1]\}$.
Rub: +2. -1 for missing the range of indices.
Solution to the original problem: $Rev[n]$.
Rub: +1

No credit for changing the subproblem definition.

4. (4 points) Suppose that you are given an undirected graph G and you want to find a subset of vertices $S \subseteq V(G)$ of minimum size such that for every edge (u,v), either $u \in S$, or $v \in S$ (or both). Consider a greedy algorithm as follows. Initialize a set $Q = \emptyset$. Pick a vertex u with maximum degree (the number of neighbors) in the graph and add u into Q. Remove u from G, and then repeat this process for the rest of the graph. Give an example where this algorithm *does not construct* an optimal solution. Just draw the example, show what optimal is and what greedy does.

Answer: See example below. **Rub:** +3 for constructing the correct counter example. +0.5 each for showing what optimal is and what greedy does.



Optimal : $\{v_2, v_3, v_4\}$

greedy : $\{v_1, v_2, v_4, v_6\}$

Part B

For each of the following questions, you get 10% for writing "I do not know the solution"

1. (9 points) Let G be a connected undirected graph having n vertices and $n + 15$ edges with all edges having positive weights. Design an $O(n)$-time algorithm to detect the minimum weight edge that is present in a cycle. Give only the most important steps and argue runtime.

   Answer: Approach 1: (We did not see this approach earlier but got it after looking at some of the endsem solutions. This is perhaps the most elegant and clean solution to the problem).

   First we use modified DFS to determine all the *bridge edges* in the graph. Using the algorithm done in lectures, this can be done in $\mathcal{O}(|E| + |V|)$ which is just $\mathcal{O}(n)$ in this case. The important thing to observe is that an edge is a bridge edge *if and only if* it is not part of any cycle. So the remaining task is to determine the minimum weight edge among the remaining edges which clearly can be done by a simple traversal of the adjacency lists.

   Approach 2: This is more elaborate (although essentially uses ideas of the bridge edge finding algorithm)

```
Algorithm 1: Find-Min-Edge-One-Cyle
   Data: Graph $G = (V, E)$ as adjacency list, weights $w(e), \forall e \in E$, a source vertex $v$
1  Mark $v$ as visited
2  for all $u$ in adjacency list of $v$ do
3      if $u$ is already visited then
4          $\min\_edge = w(vu)$
5          $v' = v$
6          while $v' \neq u$ do
7              $\min\_edge = \min\{\min\_edge, w(v', parent[v'])\}$
8              $v' = parent[v']$
9          return min_edge and abort recursion (abort is not precise but just for
           pseudocode)
10     else
11         $parent[u] = v$
12         Find-Min-Edge-One-Cyle($G, w, u$) (usual DFS)
```

   Algorithm 1 detects the minimum weight edge of one cycle. As soon as a back edge (vu) is detected (Line 2), the algorithm traces back the parent pointers all the way to u and finds the minimum weight edge. This is accomplished in Lines 2-8. The rest is just DFS.

   Algorithm 2 calls Algorithm 1. Once it finds the minimum weight edge of one cycle, it deletes that edge from the graph and continues the process from scratch. The process is repeated 16 times since after that there won't be any more cycles.

Algorithm 2: Find-Min-Edge-All-Cycles

```
  Data: Graph $G = (V, E)$ as adjacency list, weights $w(e), \forall e \in E$
1 Mark $v$ as unvisited for all vertices $v$
2 for $i$=1 to 16 do
3     Mark $v$ as unvisited for all vertices $v$
4     Mark $parent[v]$ as NULL for all vertices except $s$ (some arbitrary source)
5     Initialize $cur\_min = \infty$
6     Let $(uv)$ = Find-Min-Edge-One-Cycle($G, w, s$)
7     $cur\_min = \min\{cur\_min, w(uv)\}$
8 return $cur\_min$
```

Rubric: +7 alloted for pseudocode. Just realizing that cycle detection/dfs is useful : +2. Finding the minimum cost edge of one cycle : +3. Repeating the process 16 times : +2. Some other algorithm might also work but it has to revolve around this basic framework. Similar rubric has to be followed in those cases.

Runtime: Algorithm 1 is essentially cycle detection which can happen in $\mathcal{O}(n)$ time (as done in tutorial). However, the tracing back to the 'beginning' of the cycle (Lines 2-8) needs additional work which can be at most the length of the cycle which is at most n.

Algorithm 2 calls Algorithm 1 16 times and hence the total runtime is $\mathcal{O}(n)$.

Rubric: +3 for runtime. +2 for saying minimum of one cycle can be done in $\mathcal{O}(n)$. +1 for realizing we need only 16 DFS runs.

Additional Explanation. (Not for grading)

Every step detects a cycle, and deletes exactly one edge. So, this preserves the fact that G remains connected after every step of edge deletion that the algorithm does. After 16 steps, the graph will have only $n - 1$ edges, hence cannot contain a cycle. All the steps (i), (ii), and (iii) can be performed in $O(n)$-time each separately. At every step, we keep track of the deleted edges along with its weights. In this process, we can find out a cycle explicitly.

The correctness arguments (sketch): We have to prove the following lemmas as we keep deleting one edge once we detect a cycle. In the rest of the parts of the proof, we use *optimal solution* or *optimal edge* to denote an edge with minimum weight that is present in at least one cycle.

Lemma B.1.1: As the algorithm deletes an edge participating in some cycle, the graph continues to remain connected after every instance of edge deletions.

Proof: Clearly, the edge deleted is present in a cycle, and is not a bridge (or a cut-edge). Hence, the deletion of such an edge does not disconnect the graph. Hence, the graph continues to remain connected after every edge deletion. (end of proof)

Lemma B.1.2: If an optimal edge (u, v) is present in a unique cycle, then the algorithm will delete (u, v) in one of the steps.

Proof: It is clear from Lemma B.1.1 that the graph continues to remain connected after every edge deletion that the algorithm does. As an optimal edge is present in a unique cycle C^*, and the graph remains connected after every step, the algorithm will invoke depth-first search (with some modification) at some step when the graph has at least n edges. Hence, in one of the steps, this particular unique cycle C^* will get detected by the algorithm. At that step, the algorithm will delete the edge (u, v) as it has minimum weight. (end of proof)

Lemma B.1.3: Suppose that an optimal edge (u, v) is present in two or more cycles, but the algorithm deletes an edge (w, z) that is not optimal. Then, (u, v) is present in at least one cycle in $G - \{(w, z)\}$.

Proof: Suppose that an optimal edge (u, v) is present in at least two cycles. Let C_1 and C_2 be two such cycles containing the edge (u, v). Then, there is a cycle C_3 such that $(u, v) \notin C_3$. Moreover, $C_1 \neq C_3$ and $C_2 \neq C_3$. Now, suppose that the algorithm deletes an edge (w, z) that is not an optimal edge. Then, there are two cases that can happen. The first case is that (w, z) is neither present in C_1, nor present in C_2 (or in any other cycle where (u, v) is present). In such a case, both C_1 and C_2 are retained in $G - (w, z)$. The second case is that (w, z) is present in one of the cycles in C_1 or in C_2 (or in some cycle where (u, v) is present. In such case, if the algorithm detects C_1 (or C_2) explicitly, and chooses to delete an edge that is of minimum weight in C_1 (or in C_2 respectively). But, then (u, v) will be present in C_1 (or in C_2 respectively). Hence, the algorithm will delete (u, v) as the weight of (u, v) is strictly smaller than the weight of (w, z). Otherwise, in the third case that (w, z) is actually present in C_3 that is distinct from any other cycle where (u, v) is present. In such case, algorithm can detect such a cycle C_3 and deletes (w, z) as it has minimum weight. Then, one of C_1 or C_2 (or one of the cycles that contain the edge (u, v)) is still retained even after deleting (w, z). This completes the proof (end of proof).

Using the above three lemmas, we can prove that the algorithm correctly finds out an optimal edge.

2. (14 points) Suppose you are given a road-network in $G = (V, E)$ where $(u, v) \in E$ models a unidirectional link road between two cities $u, v \in V$. Each link (uv) also has a restriction on the size of the vehicle that is allowed to use that link - we denote this by a non-negative integer c_{uv}. Your job is to chart out a route from city s to city t (of course there may or may not be a direct link between these cities) so that you can drive the biggest vehicle that you own (so that you can take as many friends as possible on the trip!). Design an algorithm to find such a route that runs in time $\mathcal{O}(|E| \log |V|)$ or faster. Remember that the numbers c_{uv} are very very large compared to m, n.

Solution: There are two possible correct solution approaches.

Approach 1: In graph algorithm language, the problem is to find the path from s to t that maximizes the 'bottleneck' capacity - that is the edge along this path which has the minimum capacity. Turns out that the max bottleneck path enjoys an analogous optimal substructure property just like shortest paths. In fact, a simple modification to Dijksta's algorithm does the job as follows.

Rub : +2 just to be able to realize the above

Algorithm 3: Find-Max-Bottleneck-Path

```
Data: Graph $G = (V, E)$ as adjacency list, capacities $c(e), \forall e \in E$, a source vertex $s$,
      destination $t$
1 Initialize $label[s] = \infty$, $v] = 0, \forall v \neq s$ (+2 for initializations)
2 Create a Max-Heap $H$ with all vertices (+2 for creating the intial heap)
3 while $H$ is non-empty do
4     $v = ExtractMax(H)$ (+2 for this)
5     for all $u$ in the adjacency list of $v$ do
6         if $u$ is in $H$ then
7             if $label[u] < \min\{label[v], c(vu)\}$ then
8                 $label[u] = \min\{label[v], c(vu)\}$ (+2 for this)
9                 $DecreaseKey(H, u)$ (+2 for this)
```

Runtime : The above algorithm is just a modification of Dijkstra's algorithm with only two differences. The *label* array is now intended to contain the maximum bottleneck capacity of paths from s to all other vertices. We use a max-heap instead of a min-heap. Hence the runtime in $\mathcal{O}(|E| \log |V|)$ **Rub : +2 for running time only if it conforms to the written pseudocode and max-heap etc. mentioned properly**

The correctness can be formally proved using the exact induction setup of Dijkstra with minor changes to the inequalities.

Approach 2: Another alternate solution can also be provided. Just compute a maximum spanning tree of the graph. The approach is similar to the Prim's algorithm (with appropriate modifications). Proof of correctness is also similar to the proof of Prim's algorithm.

Rubric: +2 for the running time only if it the written pseudocode computes maximum spanning tree, and max-heap etc. mentioned properly. Also pseudocode has to be written. Otherwise, just stating "maximum spanning tree is equivalent to maximum bottleneck pat" and hence computing maximum spanning tree gives the answer alone will not get credit.

3. (12 points) In the fictional Starling City, Oliver Queen aka the Arrow has his lair (known as the Arrowcave) in the basement of a nightclub owned by his sister. Currently, the entire nightclub and the cave is under attack by Ra's Al Ghul and his League of Assassins. It is only up to the tech genius Felicity Smoak to figure out if it's possible to evacuate everyone inside. Now she has a list of locations - let us call that list X - which has people stranded at. She also has a list of locations S which are safe spots. Assume that the list X and S are disjoint. Felicty has an entire map of the lair and the club which is assumed to be modeled as a set of locations V and a set of corridors/stairwells E connecting pairs of locations $u \in V, v \in V$. Felicity has to chalk out an evacuation plan which is a set of paths such that

   a) Each location in X is the beginning of one path

   b) each path ends at some location in S

   c) Any two paths *do not* share a common corridor/stairwell

   Can you help Felicity design a fast algorithm - which runs in time polynomial in $|V|, |E|$ - to figure out if an evacuation plan is possible ? No pseudocode required. Describe the construction briefly but precisely and argue tightest possible runtime.

   The problem can be solved using a max-flow formulation. We create the flow network as follows (**+2 for writing this**)

   - Create a dummy source s and dummy sink t (**Rub : +1**)
   - Introduce a directed edge from s to all vertices in X with capacity $= 1$ (**: +1 for the edge, +1 for capacity**)
   - Introduce a directed edge from all vertices in S to sink t with capacity $|X|$ (or $|V|$ or ∞) (**: +1 for edge, +1 for capacity**)
   - Introduce directions to all edges in E (assuming it is given which direction the stairwell goes to, we can introduce both directions if needed) and make capacities 1 (it is critical to make all capacities 1 here) (**+1 for the edges, +2 for correct capacity**)

   The answer is YES if max flow is $|X|$, no otherwise.

   Justification (Not required for grading) : Roughly speaking, consider any feasible plan. Since the paths are edge-disjoint, we can send 1 unit of flow along those paths without violating capacities and send a total flow of $|X|$, by the property (a) of a feasible plan.

   On the other hand, suppose you compute a max-flow using FF algorithm. Recall that it is an *integral* flow. This along with the fact that all capacities are 1 gives us that all the paths along which flow is sent by FF will be edge disjoint. The value of the flow gives the number of paths.

   Runtime: Since the capacities are all 1, the runtime is $\mathcal{O}(|X| \cdot |E|)$ since the value of the min-cut is at most $|X|$. ($\mathcal{O}(|S| \cdot |E|)$ is also fine).(**+2 for writing this correctly**)