UCSD CSE 101, Winter 2018 FINAL EXAM SOLUTIONS
VERSION A

March 20, 2018

NAME: ____

Student ID:____

Question	Points
1	18
2	12
3	10
4	10
5	10
TOTAL	60

DIRECTIONS.

- There are 16 pages in this exam (Question 5 ends on Page 13), plus three scratch pages. Please verify this now.
- Be clear and concise. Write your answers in the space provided.
- Do NOT write any material that you wish to be graded on the back of pages, since we will scan into Gradescope. Any material written on the back of a page will be disregarded. If you need additional space for your answers, use the scratch pages at the end and/or ask a proctor for extra sheets of paper.
- You may freely use or cite any algorithm from class or from the DPV textbook, e.g., dfs(G), bfs(G, s), dijkstra(G, l, s), etc.
- To avoid rewards for random T/F guessing, in Q1 a correct answer receives +1 points, a wrong answer receives -1 points, and a blank (no) answer receives 0 points.
- The exam duration is 180 minutes. Time checks will be announced after 60, 120 and 150 minutes. At each of these time checks, you will be reminded that a real-time list of clarifications is being projected on-screen.
- As a courtesy to your classmates, do not leave the room during the last 10 minutes of the exam. No turn-ins of exams will be accepted between the 170-minute mark and the 180-minute mark.
- Make sure to check your work, and good luck!

QUESTION 1. *True* or *False* (18 points)

IMPORTANT NOTE: To avoid rewards for random guessing, a correct answer receives +1 points, a wrong answer receives -1 points, and a blank (no) answer receives 0 points.

(a) **A tight big-O bound for the runtime of the procedure $g(n)$ below is $O(n \log n)$.**

```
procedure g(n):
    for (i = 1; i < n; i = i * 2):
        for (j = 1; j < n; j = j + 2):
            for (k = n; k > 1; k = k / 3):
                print("CSE 101")
```

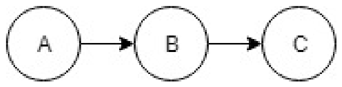
ANSWER: FALSE. Complexity is $O(n \log^2 n)$. (Version B: TRUE for tight $O(n \log^2 n)$ bound.)

(b) **If Algorithm A has runtime that satisfies the recurrence $T_A(n) = 2T_A(n/8) + O(1)$ and Algorithm B has runtime that satisfies the recurrence $T_B(n) = 4T_B(n/2) + O(1)$, then Algorithm A is asymptotically faster than Algorithm B.**

ANSWER: TRUE. You can use Master theorem to obtain this answer. You can also note the number of sub-problems and the height of recursion tree in A is smaller than that of B. Therefore, A should be asymptotically faster than B.

(Version B: labeling is reversed; answer is FALSE.)

(c) **If dfs() is executed on the graph below, vertex A will have the largest *post* number and vertex C will have the largest *pre* number. Note: Break ties in dfs as in lecture and homework.**



ANSWER: TRUE. (Version B: B-C edge is reversed; answer is FALSE.)

(d) **Two SCCs of a directed graph cannot have any vertices in common.**

ANSWER: TRUE. SCCs induce a disjoint partition of the vertex set.

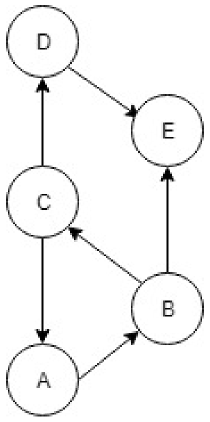
(e) The Meta-graph of a directed graph can have exactly one cycle.

ANSWER: FALSE. The Meta-graph is a DAG, and thus has no cycles.

(f) The time complexity of DFS on a tree is $O(|V|)$.

ANSWER: TRUE. In a tree, $|E|$ is $O(|V|)$.

(g) The number of SCCs in this graph is 2.



ANSWER: FALSE. There are three SCCs: (A,B,C); (D); (E).

(h) The Bellman-Ford algorithm correctly finds shortest-path distances in instances of the single-source shortest path problem which do not contain any negative-weight directed cycles, but the algorithm cannot be used to detect the existence of negative-weight directed cycles.

ANSWER: FALSE. Bellman-Ford can detect negative cycles (recall BitVest problem from homework 3).

(i) In an edge-weighted, undirected, connected graph, an edge with minimum weight in a cut must belong to the MST of the graph.

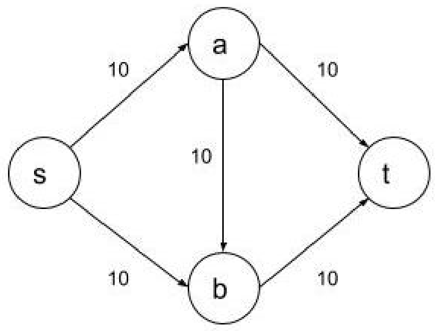
ANSWER: THIS QUESTION IS OMITTED.

(j) During the execution of the Ford-Fulkerson algorithm, once an edge in the flow network has at least one unit of flow, the algorithm will never decrease the flow on this edge.

ANSWER: FALSE. The nature of residual graphs allows the flow to be decreased.

In the following network, if we first send a flow of 10 units through the path $(s - a - b - t)$, we can undo the flow of 10 units on edge (a, b), redirect this flow to the edge (a, t) and increase the

flow by 10 units again on the path $(s - b - t)$, to obtain a max flow of 20.



(k) **Any linear function $y = ax + b$ is both convex and concave.**

ANSWER: TRUE.

(l) **A dynamic programming solution always has space requirement at least as large as the number of unique subproblems.**

ANSWER: FALSE. To obtain the $n - th$ number in the Fibonacci sequence, we only need space of $O(1)$ complexity. The efficient space complexity problems from homework 4 also illustrate the same.

(m) **Prim's MST algorithm is greedy, but Dijkstra's single-source shortest paths algorithm is not greedy.**

ANSWER: FALSE. They are both greedy (they are essentially the same algorithm).

(n) **A greedy approach is optimal for the fractional Knapsack problem.**

ANSWER: TRUE. In the fractional Knapsack, we just use as much as possible of the item type with highest value/weight ratio.

(o) **Computing the longest path in a directed acyclic graph is an NP-Hard problem.**

ANSWER: FALSE. It is solvable using topological ordering, as seen in lecture and in homework 3.

(p) **Because SORTING has an $\Omega(n \log n)$ lower bound in the comparison model of computation, and SORTING reduces to CONVEX HULL in $O(n)$ time, we know that CONVEX HULL has an $\Omega(n \log n)$ lower bound in the comparison model of computation.**

ANSWER: TRUE. This is the example discussed a couple of times in lecture.

(q) **The intersection of NP and NP-Hard is NP-complete.**

ANSWER: TRUE.

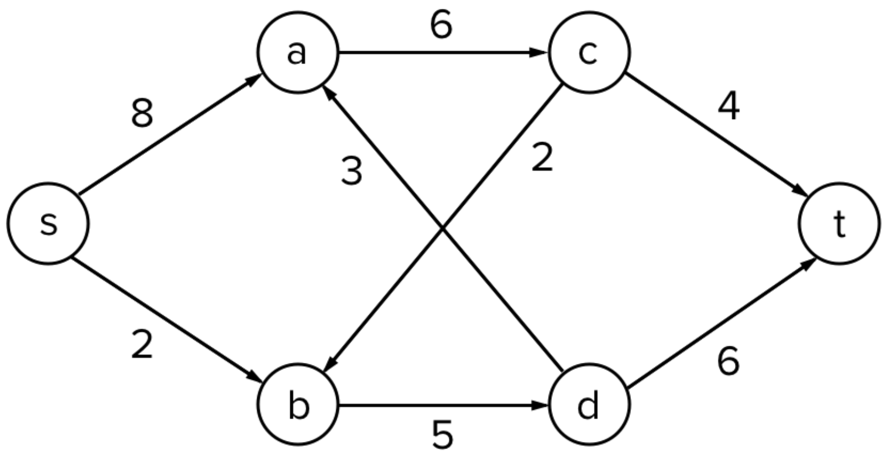
(r) **This is a free point.**

ANSWER: T

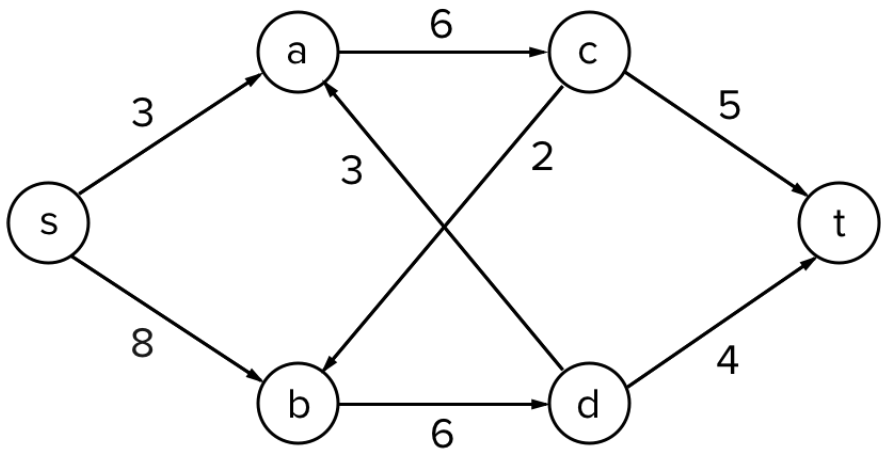
QUESTION 2. Network Flow/Linear Programming (12 points)

For this question, refer to the flow network below:

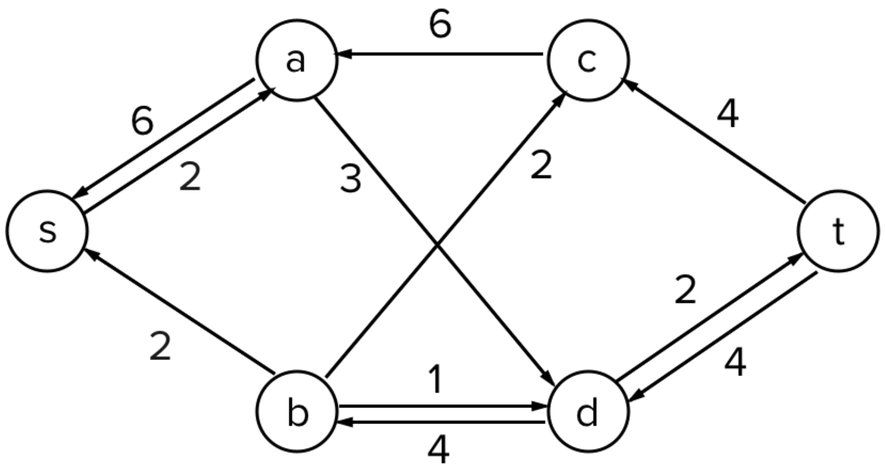
Version A:



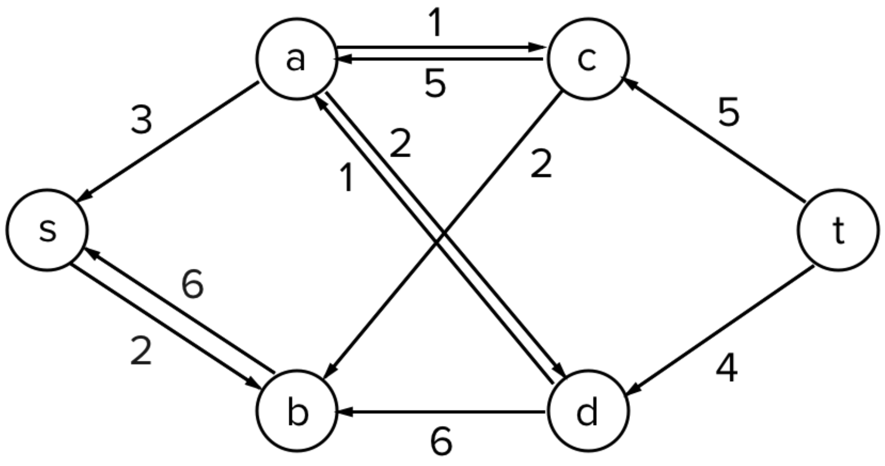
Version B:



(a) *(3 points)* Run the Ford-Fulkerson algorithm on the given network. Draw the final residual graph G' into the figure provided below. Extra copies of the network are given in the last sheet (page 16) of the exam.
Version A:



Version B:



(b) ***(2 points)*** **What is the value of the maximum flow?**

ANSWER: 8 (Version A), 9 (Version B)

(c) ***(3 points)*** **List the edges in the minimum cut that correspond to the maximum flow.**

Version A:
$e_{sb}, e_{ac}, (e_{da})$ **OR** $e_{sb}, e_{cb}, e_{ct}, (e_{da})$

Version B:
e_{ct}, e_{dt} **OR** $e_{sa}, e_{bd}, (e_{cb})$

(d) ***(4 points)*** **Write the problem of determining the maximum s-t flow in the given network as a linear program. (Don't leave out any constraints!)**

Version A:

Maximize $f_{sa} + f_{sb}$ **subject to**

$$\begin{aligned}
0 \leq & f_{sa} \leq 8 \\
0 \leq & f_{sb} \leq 2 \\
0 \leq & f_{ac} \leq 6 \\
0 \leq & f_{da} \leq 3 \\
0 \leq & f_{bd} \leq 5 \\
0 \leq & f_{cb} \leq 2 \\
0 \leq & f_{ct} \leq 4 \\
0 \leq & f_{dt} \leq 6 \\
f_{sa} + f_{da} &= f_{ac} \\
f_{sb} + f_{cb} &= f_{bd} \\
f_{ac} &= f_{cb} + f_{ct} \\
f_{bd} &= f_{da} + f_{dt}
\end{aligned}$$

Version B:

Maximize $f_{sa} + f_{sb}$ **subject to**

$$\begin{aligned}
0 \leq & f_{sa} \leq 3 \\
0 \leq & f_{sb} \leq 8 \\
0 \leq & f_{ac} \leq 6 \\
0 \leq & f_{da} \leq 3 \\
0 \leq & f_{bd} \leq 6 \\
0 \leq & f_{cb} \leq 2 \\
0 \leq & f_{ct} \leq 5 \\
0 \leq & f_{dt} \leq 4 \\
f_{sa} + f_{da} &= f_{ac} \\
f_{sb} + f_{cb} &= f_{bd} \\
f_{ac} &= f_{cb} + f_{ct} \\
f_{bd} &= f_{da} + f_{dt}
\end{aligned}$$

QUESTION 3. Greed. (10 points)

You are babysitting n children and have $m \geq n$ cookies. You must give each child exactly one whole cookie (it is not possible for multiple children to share a cookie).

For $1 \leq j \leq m$, cookie j has a size s_j.
For $1 \leq i \leq n$, child i has a greed factor g_i, which is the minimum size of a cookie that child i will be content with.
(All greed factors and cookie sizes are positive.)

We define child i to be *contented* if he/she receives a cookie j, where $g_i \leq s_j$.

Your goal is to maximize the number of contented children.

(a) *(3 points)* There are $n = 5$ children with greed factors tabulated below. There are $m = 5$ cookies whose sizes are tabulated below as well.

Greed factor table

g_1	g_2	g_3	g_4	g_5
7	3	5	6	8

Cookie size table

s_1	s_2	s_3	s_4	s_5
8	2	4	2	3

(i) What is the maximum number of contented children?

The maximum number of contented children is 2 (Version B example answer is 3)

(ii) What is a corresponding maximum matching of cookies to children?

[For example, if child 2 is given cookie 5, then (g_2, c_5) belongs to the matching set. If there are two cookies with the same size, then you can give either of the cookies to a child, provided all constraints are satisfied.] ***Note that you should write down only one maximum matching of cookies to children, not all possible maximum matchings.***

A possible maximum matching is $\{(g_5, s_1), (g_2, s_3)\}$ Note that adding any unmatched cookie and child as pair in the maximum matching is incorrect. (Version B example would have 3 pairs in the maximum matching).

(b) ***(7 points)*** **In this part, we will design an efficient greedy algorithm to maximize the number of contented children. Write a high-level description of your algorithm describing all the key ideas of the solution. Prove the optimality of the algorithm and analyze its time complexity.**

(i) ***(3 points)*** **High-level description of your algorithm:**

Let the sorted array of greed factors of n children be represented by G.
Let the sorted array of sizes of m cookies be represented by S. Both of the arrays are sorted in non-increasing order.
Set *count*, representing the number of matchings, to 0.
We will use two pointers, i and j to traverse G and S respectively. Scanning both arrays left to right,

- **If $G[i] > S[j]$, increment i until $G[i] \leq S[j]$**
- **If $G[i] \leq S[j]$, we have found a match i.e. we greedily match the first pair $(G[i], S[j])$ that satisfies $G[i] \leq S[j]$. We then increment *count*, i and j.**

We stop the scan when we complete traversing either of the arrays, indicating no more matchings is possible. The variable *count* therefore represents the maximum matching possible.

(ii) ***(1 point)*** **Time complexity analysis:**

The sorting of the two arrays can be executed in $O(n \log n + m \log m)$ time.
The scan to find the matchings is linear in $O(max(n, m))$.
Therefore the overall run time is $O(n \log n + m \log m)$

(iii) ***(3 points)*** **Proof of correctness:**
We prove the correctness of the greedy algorithm using two claims:

Claim 1: If i is the first index $1 \leq i \leq n$ in the sorted G such that $G[i] \leq s_j$ for a cookie j, then any previously unmatched child is correctly unmatched.

Proof: Our greedy algorithm uses the sorted order to *greedily* find a child who can be given a cookie of size s_j. If i is the lowest index such that $G[i] \leq s_j$, and if k was the index of the previous child assigned a cookie, then $G[o] > s_j$ for any $k < o < i$. By the sorted order, $G[o] > s_j$ i.e. the greed of all of these children is more than the size of the next available cookie j, until we reach child i. Therefore, all of these children $k < o < i$ should remain unmatched.

Claim 2: The greedy algorithm A returns the maximum number of contented children, and therefore is optimal.

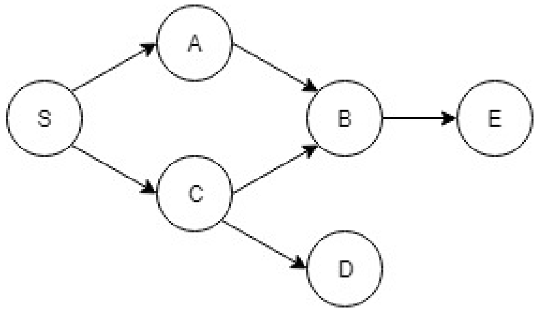
Proof: Suppose there is an optimal solution O which returns C_O as the maximum number of contented children. Let the number of contented children returned by our algorithm A be C_A. We will assume $C_O > C_A$ i.e. there are more number of contented children than what greedy algorithm could find.
Say a cookie j was assigned to a more greedy child k by algorithm O (i.e. a matching (g_k, s_j) was added), while our greedy algorithm matched it with child i (i.e. a matching (g_i, s_j) was added).

We know that $g_k > g_i$ and $k < i$. We also know that our greedy algorithm matches the first i, such that $g_i \leq s_j$. However, in the above case, both $g_k \leq s_j, k < i$ and $g_i \leq s_j$ hold, which implies $k = i$. Thus, the optimal solution at this point reduces to the matching found by greedy algorithm. By induction on i, j, we prove that our assumption is incorrect. We can easily also prove that $C_O < C_A$ is not possible.
Thus, $C_O = C_A$, and therefore our algorithm is optimal, and returns the maximum number of contented children.

QUESTION 4. Dynamic Programming (10 points)

You are given a DAG $G = (V, E)$ and a source vertex s. Give an algorithm to find the number of distinct paths in G of odd length, and of even length, that start at s and end at any other vertex in the graph. Your algorithm should have $O(|V| + |E|)$ runtime and return two numbers, NUM_ODDS and NUM_EVENS.

In the following example,



the total numbers of odd-length and even-length paths are

- **NUM_ODDS = 4. (Odd-length paths: $(S \to A)$, $(S \to C)$, $(S \to A \to B \to E)$, $(S \to C \to B \to E)$.)**
- **NUM_EVENS = 3. (Even-length paths: $(S \to A \to B)$, $(S \to C \to B)$, $(S \to C \to D)$.)**

Note that you are only required to report the total number of paths of each type. Your algorithm should NOT write out all the paths.

(i) ***(2 points)*** **Subproblem definition: We define:**

- **$f[v]$ – number of distinct paths of odd length that start at node v.**
- **$g[v]$ – number of distinct paths of even length (including length 0) that start at node v.**
- **The answer for the problem will be $f[s]$, $g[s] - 1$ (we are excluding a path of length 0)**

(ii) ***(3 point)*** **Recursive Formulation:**

$$f[v] = \begin{cases} 0 & \textbf{if outdegree(v) = 0} \\ \sum_u g[u] & \textbf{for all } u : (v, u) \in E \end{cases}$$

$$g[v] = \begin{cases} 1 & \textbf{if outdegree(v) = 0} \\ 1 + \sum_u f[u] & \textbf{for all } u : (v, u) \in E \end{cases}$$

(iii) ***(4 points)*** **Pseudocode:**

```
get_odd_even_paths(G, v):
 if f[v] == inf or g[v] == inf:
     f[v] = 0
     g[v] = 1
     for u: (v, u) in E:
         f[u], g[u] = get_odd_even_paths(G, u)
         f[v] += g[u]
         g[v] += f[u]
 return f[v], g[v]

main(G, s):
 for v in V:
     f[v] = g[v] = inf
 NUM_ODDS, NUM_EVENS = get_odd_even_paths(G, s)
 return NUM_ODDS, NUM_EVENS - 1
```

(iv) ***(1 point)*** **Time complexity analysis**

We compute values $f[v], g[v]$ for each node only once in $\mathcal{O}(1)$ time. To compute values for all possible v we may need to traverse an entire graph. Thus, overall time complexity is $\mathcal{O}(|V| + |E|)$.

QUESTION 5. NP and Reduction (10 points)

(a) (6 points)

The Hamiltonian Circuit problem is stated as:

HC(G): Does graph G = (V,E) contain a Hamiltonian Circuit?

Let us define the Two Circuits problem as:

2CIRCUITS(G): Does graph G = (V,E) contain two disjoint circuits, with sizes n_1 and n_2 respectively, for some n_1 and n_2 satisfying $n_1 + n_2 = |V|$? (Two circuits are disjoint if they have no vertices in common.)

Given that HC(G) is NP-complete, prove that 2CIRCUITS(G) is NP-complete.

Clarifications made during the exam:

- **Q5: "circuit," "tour," and "cycle" are synonymous.**
  - **There are m distinct vertices in a circuit/cycle of length m edges**
- **Q5(a): You can assume that all instances of HC(G) and 2CIRCUITS(G) are undirected graphs.**

— 2CIRCUITS(G) $\in$ NP:

Given a graph $G = (V, E)$ and two circuits that are claimed to exist in G, we can (i) verify that the numbers of vertices in the circuits sum to $|V|$, (ii) verify that all vertices in the circuits are distinct (and, in V), and (iii) verify that all edges in the circuits are in E. Each of these verifications can be performed in time polynomial in the size of G. Thus, 2CIRCUITS(G) is in NP.

— NP-completeness reduction:

We reduce HC to 2CIRCUITS as follows. Given an instance $G = (V, E)$ of HC, create an instance $G' = (V', E')$ of 2CIRCUITS that consists of two separate copies of G. (Thus, $|V'| = 2|V|$ and $|E'| = 2|E|$.) Because G' is disconnected, there is no single circuit in G' with size $|V'|$. We now show that HC(G) = YES if and only if 2CIRCUITS(G') = YES.

If there exists an HC in G (i.e., HC(G) = YES), then this HC is a circuit in each of the copies of G that comprise G'. Therefore, 2CIRCUITS(G') is YES, with $n_1 = n_2 = |V|$, which satisfies $n_1 + n_2 = |V'|$.

If 2CIRCUITS(G') is YES, then because G' is disconnected, there must a circuit (of size $|V|$) in each of the two copies of G that comprise G'. So, HC(G) is YES.

(b) (4 points) Recall the decision problems TSP and TSP-Extension:

— $TSP(G, k)$: Given an edge-weighted undirected graph G and a number k, does there exist a TSP tour in G with cost $\leq k$?

— $TSP_{Extension}(G, P, k)$: Given an edge-weighted undirected graph G, a path P in G, and a number k, can P be extended to a complete TSP tour in G that has cost $\leq k$?

Assume that you are given two "deciders", $DECIDE_{TSP}(G, k)$ and $DECIDE_{TSP_{Extension}}(G, P, k)$, that respectively decide these problems.

— $DECIDE_{TSP}(G, k)$ returns YES if there exists a TSP tour in G with cost $\leq k$.

— $DECIDE_{TSP_{Extension}}(G, P, k)$ returns YES if the path P can be extended into a TSP tour in G with cost $\leq k$.

Now, you are given an arbitrary edge-weighted undirected graph $H = (V, E)$ where $|V| = N$ and all edge weights are positive integers. The sum of all the edge weights in H is bounded by a positive integer, M.

Describe how you would use the given "deciders" to find a *minimum-cost TSP <u>tour</u>* in H. You can describe your method using any combination of English description and pseudocode. The number of calls that your method makes to "deciders" should be polynomial in N and polynomial (logarithmic is doable) in M.

Clarifications made during the exam:

- **Q5(b): You can assume that H always contains at least one TSP tour.**
- **Q5(b): You are asked to find an actual tour with minimum cost.**
  - **The use of TSP-E in class, and the "logarithmic", should help...**

For the given graph $H = (V, E)$, we perform the following.

First, we perform binary search to determine the minimum value of k, call it k^*, for which $TSP(H, k)$ is YES. To do this, we make calls to $DECIDE_{TSP}(H, k)$ for each value of k that we check. We know that k^* is a positive integer $\leq M$, so we will make $O(\log M)$ calls to $TSP(H, k)$.

Second, after determining k^*, we must construct an actual tour that has cost k^*. We do this using $DECIDE_{TSP_{Extension}}(G, P, k)$ as in lecture. Start with any vertex $v_1 \in V$. We know that $DECIDE_{TSP_{Extension}}(H, P_1 = (v_1), k^*) =$ YES, since there is a tour (that visits v_1 at some point) with cost k^* in H. In other words, P_1 can be extended to a tour with cost k^*.
We then invoke $DECIDE_{TSP_{Extension}}(H, P_2 = (P_1, v_j \notin P_1), k^*)$ for all $v_j \neq v_1$. For at least one v_j, $DECIDE_{TSP_{Extension}}(H, P_2 = (P_1, v_j \notin P_1), k^*) =$ YES, since P_1 can be extended to a tour with cost k^*. We add this vertex to P_1 in order to obtain a longer path, P_2. We continue in this way to obtain P_3, P_4, etc. until we have $P_{|V|}$. At each step, i.e., for path length $L = 1, ..., |V| - 1$, there must exist some $v_j \notin P_L$ that grows our partial tour by one more vertex, to obtain a path P_{L+1}. I.e., $DECIDE_{TSP_{Extension}}(H, P_{L+1} = (P_L, v_j \notin P_L), k^*) =$ YES, A final edge from the last vertex in $P_{|V|}$ back to v_1 completes the tour with cost k^*.

SCRATCH PAPER 1 of 3. DO NOT REMOVE.

SCRATCH PAPER 2 of 3. DO NOT REMOVE.

SCRATCH PAPER 3 of 3. DO NOT REMOVE.