

Quiz 2 Solutions

Problem 1. True or False [30 points] (10 parts)

For each of the following questions, circle either T (True) or F (False). There is no penalty for incorrect answers.

- (a) **T F** [3 points] For *all* weighted graphs and all vertices s and t , Bellman-Ford starting at s will *always* return a shortest path to t .

Solution: FALSE. If the graph contains a negative-weight cycle, then no shortest path exists.

- (b) **T F** [3 points] If all edges in a graph have distinct weights, then the shortest path between two vertices is unique.

Solution: FALSE. Even if no two edges have the same weight, there could be two *paths* with the same weight. For example, there could be two paths from s to t with lengths $3 + 5 = 8$ and $2 + 6 = 8$. These paths have the same length (8) even though the edges (2, 3, 5, 6) are all distinct.

- (c) **T F** [3 points] For a directed graph, the absence of back edges with respect to a BFS tree implies that the graph is acyclic.

Solution: FALSE. It is true that the absence of back edges with respect to a *DFS* tree implies that the graph is acyclic. However, the same is not true for a BFS tree. There may be cross edges which go from one branch of the BFS tree to a lower level of another branch of the BFS tree. It is possible to construct a cycle using such cross edges (which decrease the level) and using forward edges (which increase the level).

- (d) **T F** [3 points] At the termination of the Bellman-Ford algorithm, even if the graph has a negative length cycle, a correct shortest path is found for a vertex for which shortest path is well-defined.

Solution: TRUE. If the shortest path is well defined, then it cannot include a cycle. Thus, the shortest path contains at most $V - 1$ edges. Running the usual $V - 1$ iterations of Bellman-Ford will therefore find that path.

- (e) **T F** [3 points] The depth of any DFS tree rooted at a vertex is at least as much as the depth of any BFS tree rooted at the same vertex.

Solution: TRUE. Since BFS finds paths using the fewest number of edges, the BFS depth of any vertex is at least as small as the DFS depth of the same vertex. Thus, the DFS tree has a greater or equal depth.

- (f) **T F** [3 points] In bidirectional Dijkstra, the first vertex to appear in both the forward and backward runs must be on the shortest path between the source and the destination.

Solution: FALSE. When a vertex appears in both the forward and backward runs, it may be that there is another vertex (on a different path) which is further away from the source but substantially closer to the destination. (This was covered in recitation.)

- (g) **T F** [3 points] There is no edge in an undirected graph that jumps more than one level of any BFS tree of the graph.

Solution: TRUE. If such an edge existed, it would provide a shorter path to some node than the path found by BFS (in terms in the number of edges). This cannot happen, as BFS always finds the path with the fewest edges.

- (h) **T F** [3 points] In an unweighted graph where the distance between any two vertices is at most T , any BFS tree has depth at most T , but a DFS tree might have larger depth.

Solution: TRUE. Since all vertices are connected by a path with at most T edges, and since BFS always finds the path with the fewest edges, the BFS tree will have depth at most T . A DFS tree may have depth up to $V - 1$ (for example, in a complete graph).

- (i) **T F** [3 points] BFS takes $O(V + E)$ time irrespective of whether the graph is presented with an adjacency list or with an adjacency matrix.

Solution: FALSE. With an adjacency matrix representation, visiting each vertex takes $O(V)$ time, as we must check all N possible outgoing edges in the adjacency matrix. Thus, BFS will take $O(V^2)$ time using an adjacency matrix.

- (j) **T F** [3 points] An undirected graph is said to be *Hamiltonian* if it has a cycle containing all the vertices. Any DFS tree on a Hamiltonian graph must have depth $V - 1$.

Solution: FALSE. If a graph has a Hamiltonian cycle, then it is *possible*, depending on the ordering of the graph, that DFS will find that cycle and that the DFS tree will have depth $V - 1$. However, DFS is not guaranteed to find that cycle. (Indeed, finding a Hamiltonian cycle in a graph is NP-complete.) If DFS does not find that cycle, then the depth of the DFS tree will be less than $V - 1$.

Problem 2. Neighborhood Finding in Low-Degree Graphs [20 points]

Suppose you are given an adjacency-list representation of an N -vertex graph undirected G with non-negative edge weights in which every vertex has at most 5 incident edges. Give an algorithm that will find the K closest vertices to some vertex v in $O(K \log K)$ time.

Solution: We use a modified version of Dijkstra's algorithm for shortest paths. Suppose that we were to run Dijkstra's algorithm from v until we visited the $(K + 1)$ -st vertex (i.e. v plus K more). Then, these K vertices (not including v) would be the vertices we want.

However, we must make a modification. In the version of Dijkstra presented in class, we create a binary heap (or Fibonacci heap) and initialize the distance of all N vertices to ∞ . We can't do that here, as that would require $O(N)$ time to initialize and as subsequent heap operations would take $O(\log N)$ time instead of $O(\log K)$ time. Thus, we start with an empty heap. Then, when we relax an edge, we insert the destination of the edge into the heap if it isn't already there.

The total time for this algorithm can be determined by the number of operations we perform. As each vertex has degree at most 5 and as we visit K vertices, we perform at most $5K$ Inserts, $5K$ DecreaseKeys, and K ExtractMins. Since we perform at most $5K$ Inserts, the size of the heap is at most $5K$ and all heap operations take $O(\log 5K) = O(\log K)$ time. Thus, our modified Dijkstra takes $O(5K \log K + 5K \log K + K \log K) = O(K \log K)$. (Using a Fibonacci heap results in the same asymptotic runtime.)

[Note: Many students lost points on this problem for not explaining how the heap needs to start empty and how it never grows beyond $5K$ elements.]

Problem 3. Word Chain [15 points] (3 parts)

A word chain is a simple word game, the object of which is to change one word into another through the smallest possible number of steps. At each step a player may perform one of four specific actions upon the word in play — either *add a letter*, *remove a letter* or *change a letter* without switching the order of the letters in play, or *create an anagram* of the current word (an anagram is a word with exactly the same number of each letter). The trick is that each new step must create a valid, English-language word. A quick example would be FROG → FOG → FLOG → GOLF.

- (a) [5 points] Give an $O(L)$ -time algorithm for deciding if two English words of length L are anagrams.

Solution: Iterate through each of the L letters in each word, tracking the frequency of each letter. If both words have the same counts, the words are anagrams. This is similar to counting sort and runs in $O(L + S)$ time where S is the size of the alphabet.

- (b) [2 points] Give an $O(L)$ -time algorithm for deciding whether two words differ by one letter (added/removed/changed).

Solution: Iterate through each of the words comparing each letter as you go. If the letters do not match and one word is longer, then move to the next letter in that word. If a mismatch is found twice, return false, otherwise return true at the end.

Where the previous part asked about identifying anagrams, this asks about the one-off changes other than anagram listed above. Solutions which looked for a one-off anagram were not accepted.

- (c) [8 points] Suppose you are given a dictionary containing N English words of length at most L and two particular words. Give an $O(N^2 \cdot L)$ -time algorithm for deciding whether there is a word chain connecting the two words.

Solution: Construct a graph with each word in the dictionary being a node. For each node, create an edge to another node if the function from either a or b return true. Then use BFS on this graph to determine if one word can be reached from the other. Building the graph takes L time for each comparison, done comparing each node to each other node, N^2 time. This takes longer than BFS, so total time is $O(N^2 \cdot L)$.

Problem 4. Approximate Diameter [15 points]

The *diameter* of a weighted undirected graph $G = (V, E)$ is the maximum distance between any two vertices in G , i.e. $\Delta(G) = \max_{u,v \in V} \delta(u, v)$ where $\Delta(G)$ is the diameter of G and $\delta(u, v)$ is the weight of a shortest path between vertices u and v in G . Assuming that all edge weights in G are non-negative, give an $O(E + V \log V)$ -time algorithm to find a value D that satisfies the following relation: $\Delta(G)/2 \leq D \leq \Delta(G)$. You must prove that the value of D output by your algorithm indeed satisfies the above relation.

Hint: For any arbitrary vertex u , what can you say about $\max_{v \in V} \delta(u, v)$?

Solution: We run Dijkstra's algorithm for single source shortest paths (using a Fibonacci heap) with an arbitrarily selected vertex u as the source. Since the vertices are removed from the heap in non-decreasing order of distance from u , the distance from u to the last vertex in the heap is $\max_{v \in V} \delta(u, v)$. Thus, we can find $\max_{v \in V} \delta(u, v)$ in $O(m + n \log n)$ time. We output $\max_{v \in V} \delta(u, v)$ as D . Since $\Delta \geq \delta(u, v)$ for all $u, v \in V$, $D \leq \Delta$. Further,

$$\begin{aligned}
D &= \max_{v \in V} \delta(u, v) \\
&\geq \max_{v_1, v_2 \in V} \frac{\delta(u, v_1) + \delta(u, v_2)}{2} \\
&= \max_{v_1, v_2 \in V} \frac{\delta(v_1, u) + \delta(u, v_2)}{2} \quad (\text{since the graph is undirected}) \\
&\geq \max_{v_1, v_2 \in V} \frac{\delta(v_1, v_2)}{2} \quad (\text{by triangle inequality of } \delta) \\
&= \frac{\Delta}{2}.
\end{aligned}$$

Problem 5. Triple Testing [20 points]

Consider the following problem: given sets A, B, C , each comprising N integers in the range $-N^k \dots N^k$ for some constant $k > 1$, determine whether there is a triple $a \in A, b \in B, c \in C$ such that $a + b + c = 0$. Give a *deterministic* (e.g. no hashing) algorithm for this problem that runs in time $O(N^2)$.

Solution: Perhaps the simplest solution involved Radix-Sort. We start by generating a set D of all pairs $a + b, a \in A, b \in B$; this takes $O(N^2)$ -time. Then we sort D in $O(N^2)$ time using Radix Sort; this is possible since after adding $2N^k$ to each element in D , all elements in D become integers in the range $0 \dots 4N^k$, where k is constant. Let $D'[1 \dots N^2]$ be the sorted array. Now, for each $c \in C$, we check whether $-c \in D$; this can be done in $O(\log N)$ time per element c using binary search on D' . Since $|C| = N$, we can perform all checks in $O(N \log N)$ time. Overall, the running time is $O(N^2)$.

There are many variants of the above solution. Here are common examples:

- Instead of searching for $-c, c \in C$, in D' , one can search for $-d, d \in D$, in a sorted version of C . This solution is correct, but takes $O(N^2 \log N)$ time, so it received only a partial credit.
- To find collisions between elements in D and the inverses of elements in C , one can (i) label each element to denote whether it comes from D or is the inverse of an element in C , (ii) perform Radix Sort on the union of D and the inverses of the elements in C and (iii) check if the sorted array contains any consecutive elements that are equal, and have different labels. This takes $O(N^2)$ time.
- One can use hashing to check whether an element $-c, c \in C$, belongs to D . Unfortunately, this involves using hash functions, which are either randomized (as in universal hashing) or heuristic (there are some sets on which the hash function has bad performance). As such, hashing was explicitly disallowed by the problem statement. Still, solutions involving hashing received some partial credit.

Overall, the best way to approach this was problem was to use Radix Sort (or some other form of sorting). Some people attempted to map the problem into some shortest paths problem, where the elements of A, B and C are used as edge weights. However, finding a, b, c such that $a + b + c = 0$ would typically require finding a path of length 0, which is quite different from finding the *shortest* path.

Problem 6. Number of Shortest Paths [20 points]

You are at an airport in a foreign city and would like to choose a hotel that has the maximum number of shortest paths from the airport (so that you reduce the risk of getting lost). Suppose you are given a city map with unit distance between each pair of directly connected locations. Design an $O(V + E)$ -time algorithm that finds the number of shortest paths between the airport (the source vertex s) and the hotel (the target vertex t).

Solution: First we view the map as an (unweighted) undirected graph with locations as vertices and two locations are connected if and only if there is a unit-distance connection between the two locations. Then we do a breadth-first search (BFS) starting from the airport. We augment the data structure so that each vertex has an additional field paths to count the number of shortest paths from the root to that vertex. Initially we set $\text{paths}(s) = 1$ for the root vertex s (that is, the airport) and $\text{paths}(v) = 0$ for all other vertices. After each vertex (say v) is explored during BFS, we check all the neighbors of v and set $\text{paths}(v)$ to be the sum of the paths of its neighbor nodes whose level is one less than the level of v . That is (recall that, for every $v \in V(G)$, $N(v)$ denotes the set of vertices adjacent to the vertex v),

$$\text{paths}(v) = \sum_{\substack{w \in N(v) \text{ and } \text{level}(w) = \text{level}(v) - 1}} \text{paths}(w).$$

The correctness of the algorithm follows from the fact that all the shortest paths from the root node s to a node t at level k must have length k , and each of the shortest path is of the form $\langle s, v_1, \dots, v_k = t \rangle$, where node v_i is some node at level i in the BFS tree and $1 \leq i \leq k$. The only modification to the original BFS is about the counter $\text{paths}(v)$ for every vertex v , it is clear that initialization takes $O(V)$ time and updating the counter at any vertex v takes $O(|N(v)|)$ time. Note that $\sum_{v \in V(G)} |N(v)| = 2E$ and an ordinary BFS takes time $O(V + E)$, therefore the total running time of our modified BFS is $O(V + E) + O(V) + O(E) = O(V + E)$.

1) 20 pts

Mark the following statements as **TRUE** or **FALSE**. No need to provide any justification.

FALSE [TRUE/FALSE]

For every node in a network, the total flow into a node equals the total flow out of a node.

TRUE [TRUE/FALSE]

Ford Fulkerson works on both directed and undirected graphs.

Because at every augmentation step you just need to find a path from s to t. This can be done in both direct and undirected graphs.

NO [YES/NO]

Suppose you have designed an algorithm which solves a problem of size n by reducing it to a max flow problem that will be solved with *Ford Fulkerson*, however the edges can have capacities which are $O(2^n)$. Is this algorithm efficient?

YES [YES/NO]

Is it possible for a valid flow to have a flow cycle (that is, a directed cycle in the graph, such that every edge has positive flow)?

positive flow cycles don't cause any problems. The flow can still be valid.

FALSE[TRUE/FALSE]

Dynamic programming and divide and conquer are similar in that in each approach the sub-problems at each step are completely independent of one another.

FALSE [TRUE/FALSE]

Ford Fulkerson has pseudo-polynomial complexity, so any problem that can be reduced to Max Flow and solved using *Ford Fulkerson* will have pseudo-polynomial complexity.

For example the edge disjoint paths problem is solved using FF in polynomial time. This is because for some problems the capacity C becomes a function of n or m.

TRUE [TRUE/FALSE]

In a flow network, the value of flow from S to T can be higher than the number of edge disjoint paths from S to T.

FALSE [TRUE/FALSE]

Complexity of a dynamic programming algorithm is equal to the number of unique sub-problems in the solution space.

TRUE [TRUE/FALSE]

In *Ford-Fulkerson's* algorithm, when finding an augmentation path one can use either BFS or DFS.

TRUE [TRUE/FALSE]

When finding the value of the optimal solution in a dynamic programming algorithm one must find values of optimal solutions for all of its sub-problems.

2) 15 pts

Given a sequence of n real numbers $A_1 \dots A_n$, give an efficient algorithm to find a subsequence (not necessarily contiguous) of maximum length, such that the values in the subsequence form a strictly increasing sequence.

Let $A[i]$ represent the i -th element in the sequence.

```
initialize OPT[i] = 1 for all i.  
best = 1  
for(i = 2..n) {  
    for(j = 1..i-1) {  
        if(A[j] < A[i]) {  
            OPT[i] = max( OPT[i], OPT[j] + 1 )  
        }  
        best = max( best, OPT[i] )  
    }  
}  
return best
```

The runtime of the above algorithm is $O(n^2)$

3) 15 pts

Suppose you are given a table with $N \times M$ cells, each having a certain quantity of apples. You start from the upper-left corner and end at the lower right corner. At each step you can go down or right one cell. Give an efficient algorithm to find the maximum number of apples you can collect.

Let the top-left corner be in row 1 column 1, and the bottom-right corner be in row n column m.

Let $A[i,j]$ represent the number of apples at row i column j.

```
initialize OPT[i,j] = 0 for all i,j.  
for(i = 1...n) {  
    for(j = 1...m) {  
        OPT[i,j] = A[i,j] + max( OPT[i-1,j], OPT[i,j-1] )  
    }  
}  
return OPT[n,m]
```

The runtime of the above algorithm is $O(nm)$.

4) 15 pts

Suppose that you are in charge of a large blood bank, and your job is to match donor blood with patients in need. There are n units of blood, and m patients each in need of one unit of blood. Let us assume that the only factor which matters is that the blood type be compatible according to the following rules:

- (a) Patient with type AB can receive types O, A, B, AB (universal recipient)
- (b) Patient with type A can receive types O, A
- (c) Patient with type B can receive types O, B
- (d) Patient with type O can receive type O

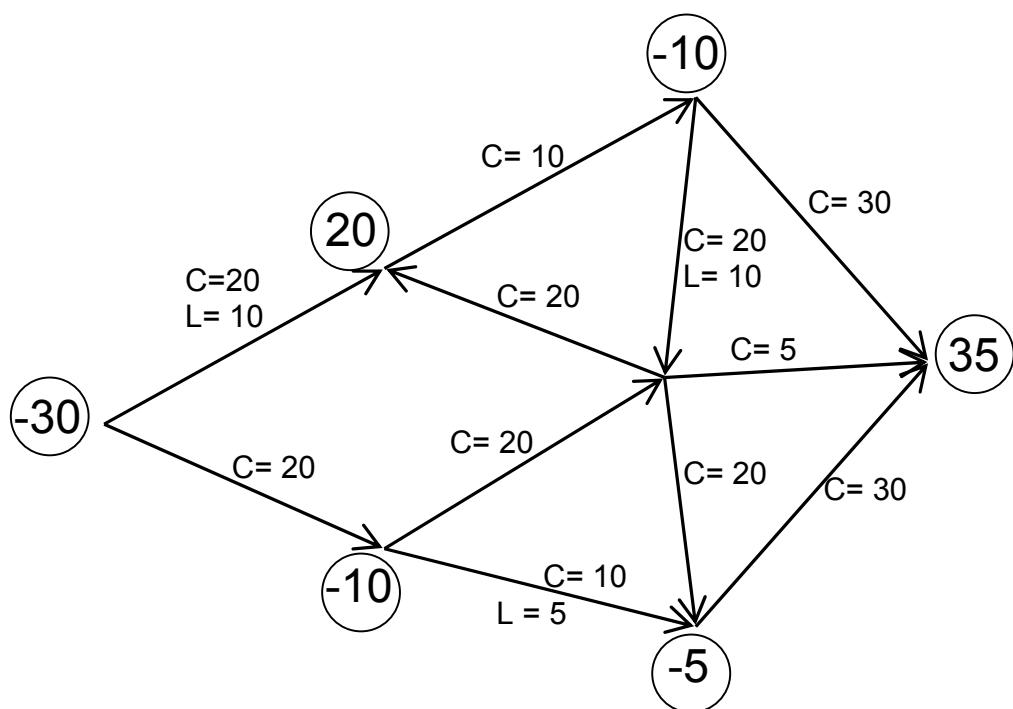
Give a network flow algorithm to find the assignment such that the maximal number of patients receive blood, and prove its correctness.

Given a set of n units of donor blood, and m patients in need of blood, each with some given blood type. We construct a network as follows. There is a source node, and a sink node, n donor nodes d_i , and m patient nodes p_j . We connect the source to every donor node d_i with a capacity of 1. We connect each donor node d_i to every patient node p_j which has blood type compatible with the donor's blood type, with a capacity of 1. We connect each patient node p_j to the sink with capacity 1.

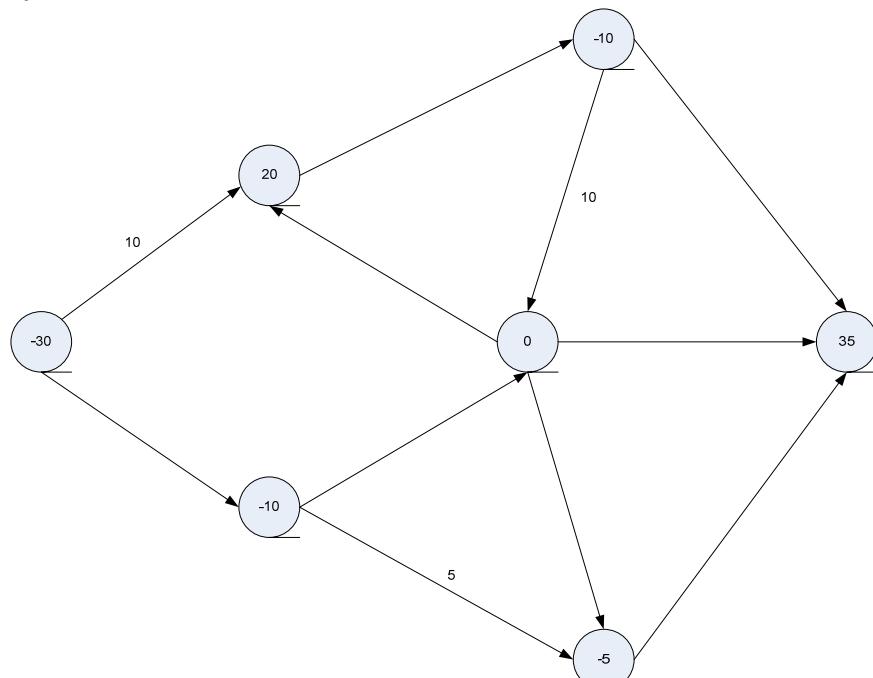
We then find the maximum flow in the network using Ford-Fulkerson. Patient j receives blood from donor i if and only if there is a flow of 1 from d_i to p_j in the maximum flow. The total flow into each donor node d_i is bounded by 1, and the total flow out of each patient node p_j is bounded by capacity 1, and so by conservation of flow, each patient and each donor can only give or receive 1 unit of blood. The types will be compatible by construction of the network.

5) 20 pts

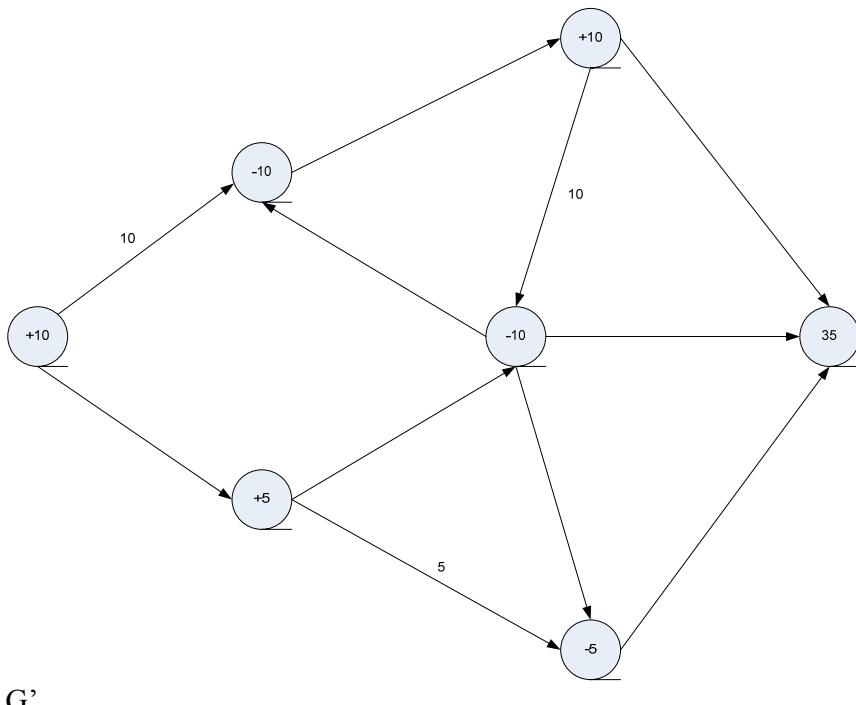
Given the graph below with demands/supplies as indicated below and edge capacities and possible lower bounds on flow marked up on each edge, find a feasible circulation. Show all your work



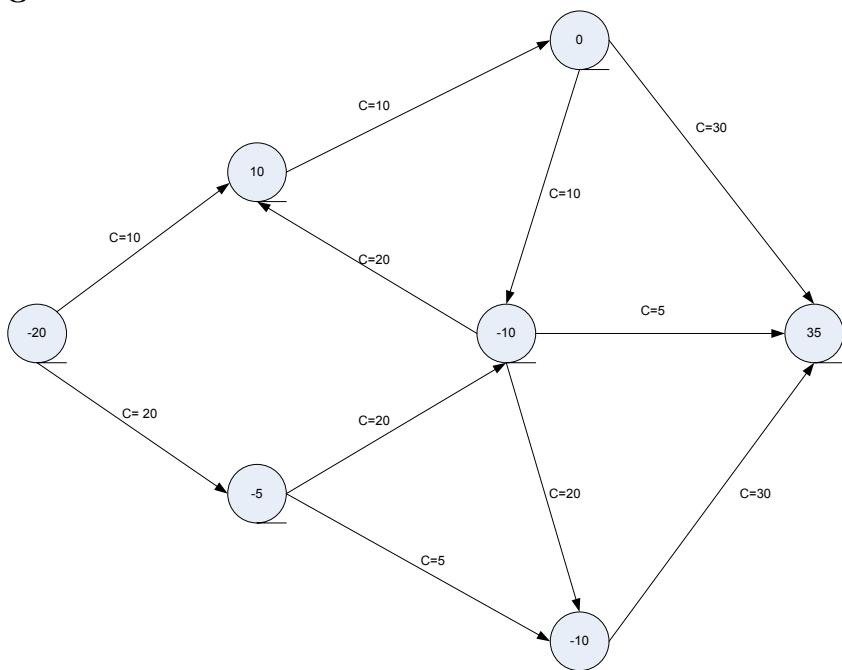
Solution to Q5
f0



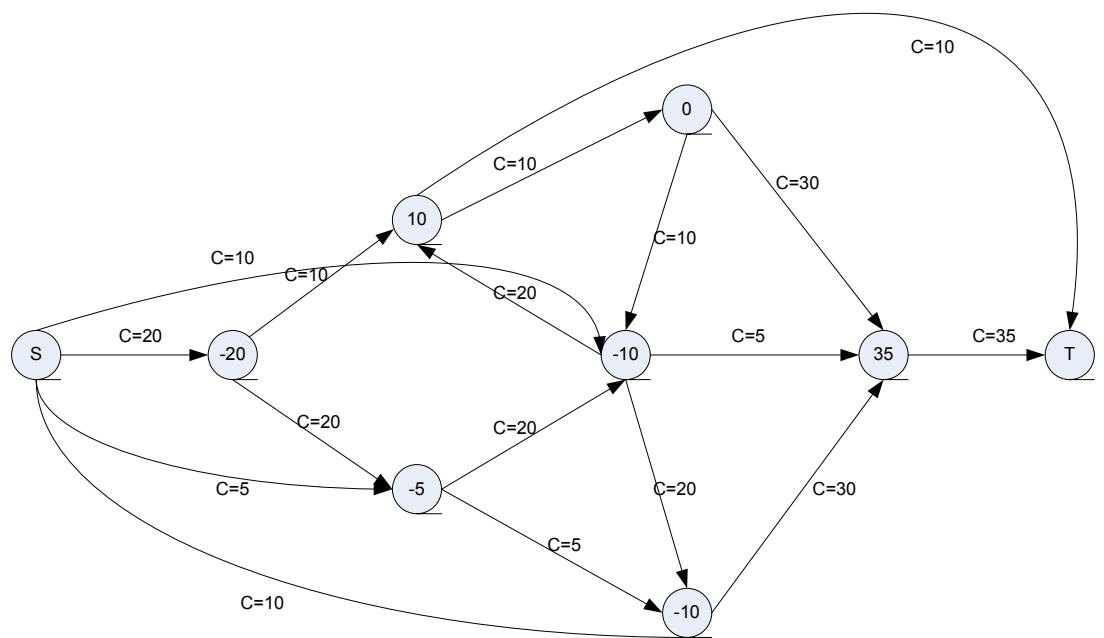
Lv



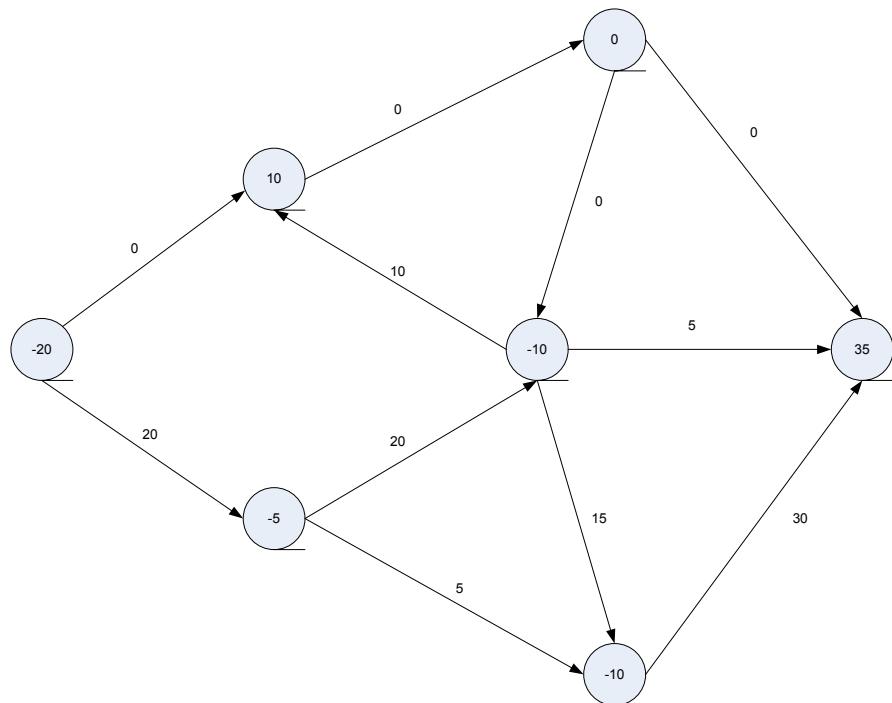
G'



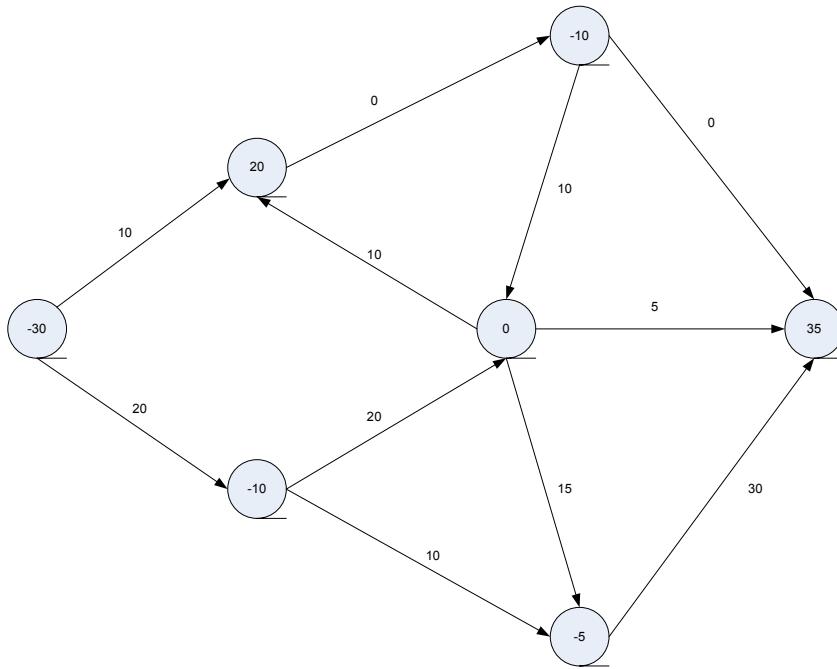
Convert G' to a $s-t$ network



f1



$$\text{Circulation} = f_0 + f_1$$

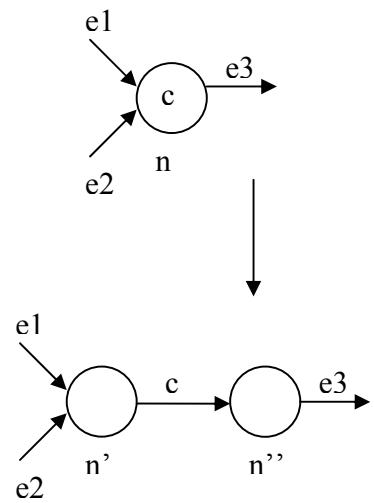


6) 15 pts

Suppose that in addition to each arc having a capacity we also have a capacity on each node (thus if node i has capacity c_i then the maximum total flow which can enter or leave the node is c_i). Suppose you are given a flow network with capacities on both arcs and nodes. Describe how to find a maximum flow in such a network.

Solution:

Assume the initial flow network is G , for any node n with capacity c , decompose it into two nodes n' and n'' , which is connected by the edge (n', n'') with edge capacity c . Next, connect all edges into the node n in G to the node n' , and all edges out of the node n in G out of the node n'' , as shown in the following figure.



After doing this for each node in G , we have a new flow network G' . Just run the standard network flow algorithms to find the maximal flow.

1) 20 pts

Mark the following statements as **TRUE** or **FALSE**. No need to provide any justification.

[**FALSE**]

If you have non integer edge capacities, then you cannot have an integer max flow.

[**TRUE**]

The maximum value of an s-t flow is equal to the minimum capacity of an s-t cut in the network.

[**FALSE**]

For any graph there always exists an edge such that increasing the capacity on that edge will increase the maximum flow from source s to sink t. (Assume that there is at least one path in the graph from source s to sink t.)

[**FALSE**]

If a problem can be solved using both the greedy method and dynamic programming, greedy will always give you a lower time complexity.

[**FALSE**]

There is a feasible circulation with demands $\{d_v\}$ if $\sum_v d_v = 0$.

[**FALSE**]

The best time complexity to solve the max flow problem is $O(Cm)$ where C is the total capacity of the edges leaving the source and m is the number of edges in the network.

[**TRUE**]

In the Ford–Fulkerson algorithm, choice of augmenting paths can affect the number of iterations.

[**FALSE**]

0-1 knapsack problem can be solved using dynamic programming in polynomial time.

[**TRUE**]

Bellman-Ford algorithm solves the shortest path problem in graphs with negative cost edges in polynomial time.

[**FALSE**]

If a dynamic programming solution is set up correctly, i.e. the recurrence equation is correct and the subproblems are always smaller than the original problem, then the resulting algorithm will always find the optimal solution in polynomial time.

2) 20 pts

You are given an n -by- n grid, where each square (i, j) contains $c(i, j)$ gold coins. Assume that $c(i, j) \geq 0$ for all squares. You must start in the upper-left corner and end in the lower-right corner, and at each step you can only travel one square down or right. When you visit any square, including your starting or ending square, you may collect all of the coins on that square. Give an algorithm to find the maximum number of coins you can collect if you follow the optimal path. Analyze the complexity of your solution.

We will solve the following subproblems: let $dp[i, j]$ be the maximum number of coins that it is possible to collect while ending at (i, j) .

We have the following recurrence: $dp[i, j] = c(i, j) + \max(dp[i - 1, j], dp[i, j - 1])$

We also have the base case that when either $i = 0$ or $j = 0$, $dp[i, j] = c(i, j)$.

There are n^2 subproblems, and each takes $O(1)$ time to solve (because there are only two subproblems to recurse on). Thus, the running time is $O(n^2)$.

3) 20 pts

King Arthur's court had n knights. He ruled over m counties. Each knight i had a quota q_i of the number of counties he could oversee. Each county j , in turn, produced a set S_j of the knights that it would be willing to be overseen by. The King sets up Merlin the task of computing an assignment of counties to the knights so that no knight would exceed his quota, while every county j is overseen by a knight from its set S_j . Show how Merlin can employ the Max-Flow algorithm to compute the assignments. Provide proof of correctness and describe the running time of your algorithm. (You may express your running time using function $F(v, e)$, where $F(v, e)$ denotes the running time of the Max-Flow algorithm on a network with v vertices and e edges.)

We make a graph with $n+m+2$ vertices, n vertices k_1, \dots, k_n corresponding to the knights, m vertices c_1, \dots, c_m corresponding to the counties, and two special vertices s and t .

We put an edge from s to k_i with capacity q_i . We put an edge from k_i to c_j with capacity 1 if county j is willing to be ruled by knight i . We put an edge of capacity 1 from c_j to t .

We now find a maximum flow in this graph. If the flow has value m , then there is a way to assign knight to all counties. Since this flow is integral, it will pick one incoming edge for each county c_j to have flow of 1. If this edge comes from knight k_i , then county j is ruled by knight i .

The running time of this algorithm is $F(n+m+2, \sum_j |S_j| + n + m)$.

4) 20 pts

You are going on a cross country trip starting from Santa Monica, west end of I-10 freeway, ending at Jacksonville, Florida, east end of I-10 freeway. As a challenge, you restrict yourself to only drive on I-10 and only on one direction, in other words towards Jacksonville. For your trip you will be using rental cars, which you can rent and drop off at certain cities along I-10. Let's say you have n such cities on I-10, and assume cities are numbered as increasing integers from 1 to n , Santa Monica being 1 and Jacksonville being n . The cost of rental from a city i to city j ($>i$) is known, $C[i,j]$. But, it can happen that cost of renting from city i to j is higher than the total costs of a series of shorter rentals.

- (A) Give a dynamic programming algorithm to determine the minimum cost of a trip from city 1 to n .
- (B) Analyze running time in terms of n .

Denote $OPT[i]$ to be the rental cost for the optimal solution to go from city i to city n for $1 \leq i \leq n$. Then the solution we are looking for is $OPT[1]$ since objective is to go from city 1 to n . Therefore $OPT[i]$ can be recursively defined as follows.

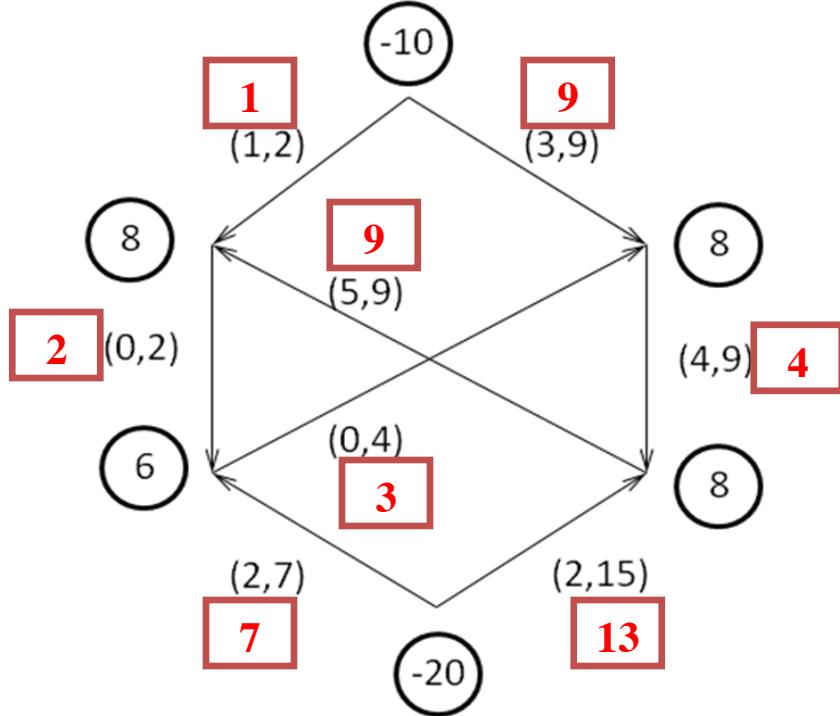
$$OPT[i] = \begin{cases} 0 & \text{if } i = n \\ \min_{i \leq j \leq n} (C[i,j] + OPT[j]) & \text{otherwise} \end{cases}$$

Proof: The car must be rented at the starting city i and then dropped off at another city among $i+1, \dots, n$. In the recurrence we try all possibilities with j being the city where the car is next returned. Furthermore, since $C[i,j]$ is independent from how the subproblem of going from city j, \dots, n is solved, we have the optimal substructure property.

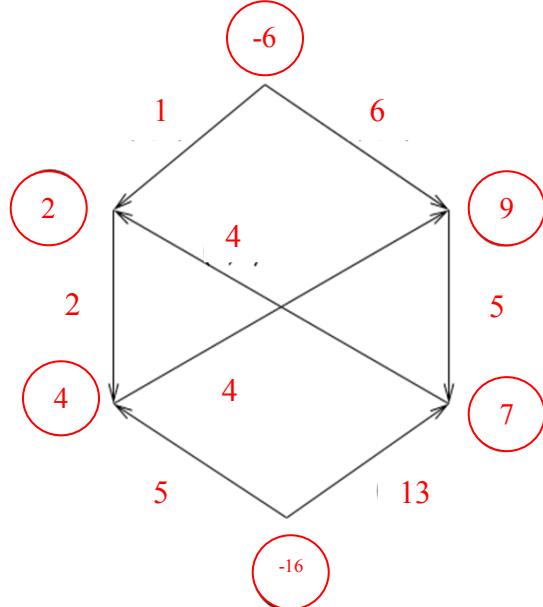
For the time complexity, there are n subproblems to be solved each of which takes linear time, $O(n)$. These subproblems can be computed in the order $OPT[n], OPT[n-1], \dots, OPT[1]$, in a linear fashion. Hence the overall time complexity is $O(n^2)$.

5) 20 pts

Solve the following feasible circulation problem. Determine if a feasible circulation exists or not. If it does, show the feasible circulation. If it does not, show the cut in the network that is the bottleneck. Show all your steps.



First we eliminate the lower bound from each edge:



Then, we attach a super-source s^* to each node with negative demand, and a super-sink t^* to each node with positive demand. The capacities of the edges attached accordingly correspond to the demand of the nodes.

We then seek a maximum s^*-t^* flow. The value of the maximum flow we can get is exactly the summation of positive demands. That is, we have a feasible circulation with the flow values inside boxes shown as above.

1) 20 pts

Mark the following statements as **TRUE** or **FALSE**. No need to provide any justification.

[TRUE/]

$T(n) = 9T\left(\frac{n}{5}\right) + n \log n$ then $T(n) = \theta(n^{\log_5 9})$

[FALSE]

In the sequence alignment problem, the optimal solution can be found in linear time and space by incorporating the divide-and-conquer technique with dynamic programming.

[FALSE]

Master theorem can always be used to find the complexity of $T(n)$, if it can be written as the recurrence relation $T(n) = aT\left(\frac{n}{b}\right) + f(n)$.

[FALSE]

In the divide and conquer algorithm to compute the closest pair among a given set of points on the plane, if the sorted order of the points on both X and Y axis are given as an added input, then the running time of the algorithm improves to $O(n)$.

[TRUE/]

Maximum value of an s-t flow could be less than the capacity of a given s-t cut in a flow network.

[TRUE/]

One can **efficiently** find the maximum number of edge disjoint paths from s to t in a directed graph by reducing the problem to max flow and solving it using the Ford-Fulkerson algorithm.

[TRUE/]

In a network-flow graph, if the capacity associated with every edge is halved, then the max-flow from the source to sink also reduces by half

[TRUE/]

The time complexity to solve the max flow problem can be better than $O(Cm)$ where C is the total capacity of the edges leaving the source and m is the number of edges in the network.

[TRUE/]

Bellman-Ford algorithm can handle negative cost edges even if it runs in a distributed environment using message passing.

[FALSE]

If all edges in a graph have capacity 1, then Ford-Fulkerson runs in linear time.

2) 16 pts

The numbers stored in array $A[1..n]$ represent the values of a function at different points in time. We know that the function behaves the following way:

- $A[1] < A[2] < \dots < A[j]$ $1 < j < n$
- $A[j] > A[j+1] > \dots > A[k]$ $j < k < n$
- $A[k] < A[k+1] < \dots < A[n]$
- $A[n] < A[1]$

Your task is to design a divide-and-conquer algorithm to find the maximum element of the array. Your algorithm must run in better than linear time. You need to provide complexity analysis of your algorithm.

Note: we don't know the exact values of j and k , we only know their range as given above.

Solution: Consider an algorithm $ALG(A, n)$ that takes as input an array A of size n , where array A either satisfies all four conditions above or it satisfies the first two conditions with $k = n$. Also, let $ALG(A, n)$ output the maximum element in A .

Store $l = A[0]$ and $r = A[n]$. $ALG(A, n)$ consists of the following steps

- a) If $n \leq 4$ output the maximum element.
- b) If $A\left[\frac{n}{2}\right] > A\left[\frac{n}{2} + 1\right]$ and $A\left[\frac{n}{2}\right] > A\left[\frac{n}{2} - 1\right]$ then return $A\left[\frac{n}{2}\right]$.
- c) If $A\left[\frac{n}{2}\right] < A\left[\frac{n}{2} + 1\right]$
 - i. If $A\left[\frac{n}{2}\right] > l$ then return $ALG\left(A\left[\frac{n}{2} + 1:n\right], \frac{n}{2}\right)$
 - ii. If $A\left[\frac{n}{2}\right] < r$ then return $ALG\left(A\left[1:\frac{n}{2}\right], \frac{n}{2}\right)$.
- d) If $A\left[\frac{n}{2}\right] < A\left[\frac{n}{2} - 1\right]$ then return $ALG\left(A\left[1:\frac{n}{2}\right], \frac{n}{2}\right)$.

To see why this is correct, observe that step (a) covers the base case for termination of the algorithm, step (b) covers the case when $\frac{n}{2} = j$, step (d) covers the case when $j < \frac{n}{2} \leq k$, step (c)(i) covers the case when $1 < \frac{n}{2} < j$, and finally step (c)(ii) covers the case of $k < \frac{n}{2} < n$.

For complexity analysis, notice that at any stage that is not the final stage of the algorithm, exactly one of step (c)(i), (c)(ii), or (d) is executed so that the associated recurrence is $T(n) = T\left(\frac{n}{2}\right) + O(1)$, implying that $T(n) = O(\log n)$ by Master's Theorem.

3) 16 pts

There are a series of part-time jobs lined up **one after the other**, J_1, J_2, \dots, J_n . For any i^{th} part-time job, you are getting paid M_i amount of money. Also for the i^{th} part-time job, you are also given N_i , which is the number of **immediately** following part-time jobs that you cannot take if you perform that i^{th} job. Give an efficient dynamic programming solution to maximize the amount of money one can make. Also state the runtime of your algorithm.

For $1 \leq i \leq n$, let S_i denote a solution for the set of jobs $\{J_i, \dots, J_n\}$, S_i maximizes the amount of money, and let $\text{opt}(i)$ denote its amount of money.

If S_i chooses to take job J_i , then it has to exclude J_{i+1} through J_{i+N_i} . In this case, its money is M_i plus the money it earns for the set $\{J_{i+N_i+1}, \dots, J_n\}$. Since S_i is optimal for the set $\{J_i, \dots, J_n\}$, S_i restricted to the subset $\{J_{i+N_i+1}, \dots, J_n\}$ has to be optimal. Thus in this case, $\text{opt}(i) = M_i + \text{opt}(i + N_i + 1)$.

If S_i chooses not to take job J_i , then the amount of money one can earn is the money from the set of jobs $\{J_{i+1}, \dots, J_n\}$. Since S_i is optimal for the set $\{J_i, \dots, J_n\}$, S_i restricted to the subset $\{J_{i+1}, \dots, J_n\}$ has to be optimal. Thus in this case, $\text{opt}(i) = \text{opt}(i + 1)$.

We thus have a recurrence for all i , $\text{opt}(i) = \max(\text{opt}(i + 1), \text{opt}(i + N_i + 1) + M_i)$. (We understand that $\text{opt}(\text{index} > n) = 0$).

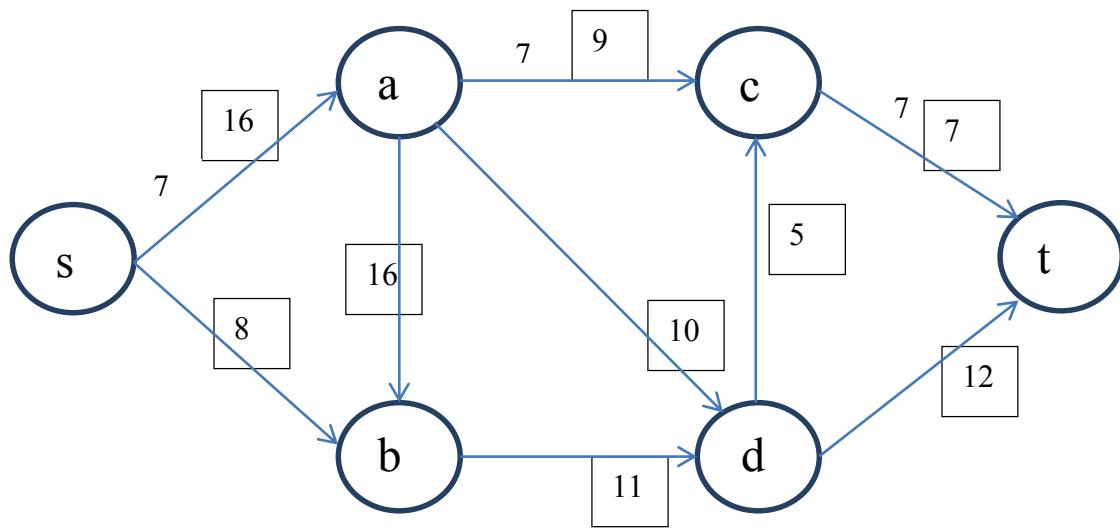
The boundary condition is that $\text{opt}(n) = M_n$ and we start solving recurrence starting with $\text{opt}(n)$, $\text{opt}(n-1)$ and so on until $\text{opt}(1)$. Complexity of this solution is $O(n)$.

4) 16 pts

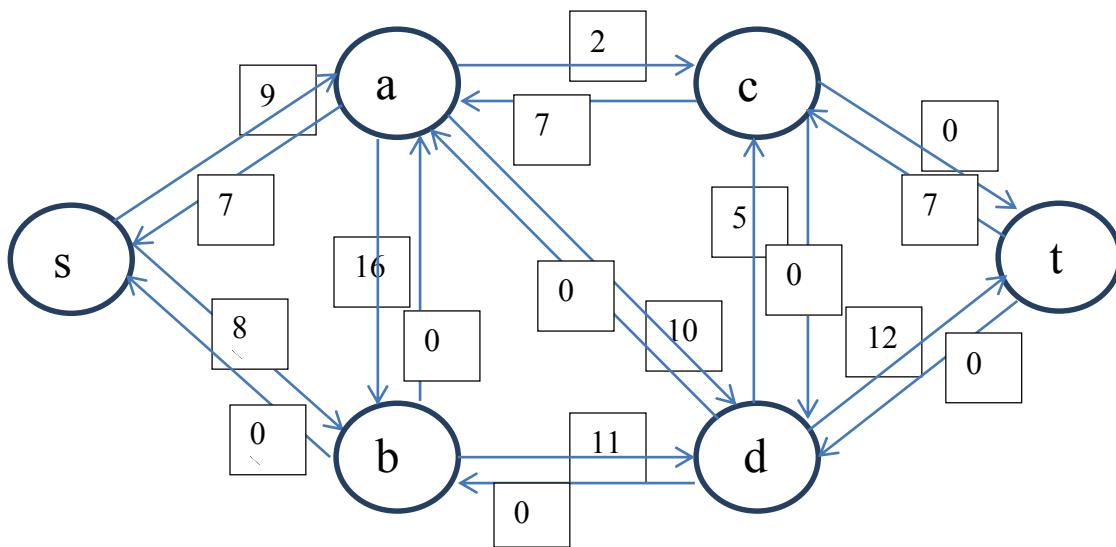
Consider the below flow-network, for which an s-t flow has been computed. The numbers in the boxes give the capacity of each edge, the label next to an edge gives the flow sent on that edge. If no flow is sent on that edge then no label appears in the graph.

- What is the current value of flow? (2 pts)
- Show the residual graph for the corresponding flow-network. (2 pts)
- Calculate the maximum flow in the graph using the Ford-Fulkerson algorithm. You need to show the augmenting path at each step, show the final max flow and a min cut. (12 pts)

Note: extra space provided for this problem on next page/



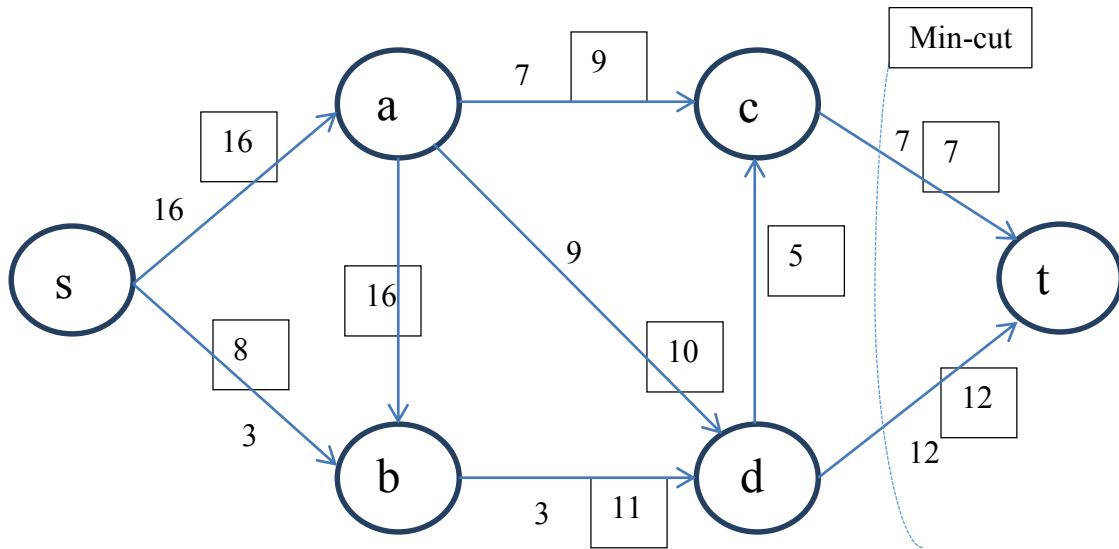
- a) Current value of the flow is 7.
b) Residual graph is shown below



Additional Space for Problem 4

- c) Augmenting paths for one of the maximum flow are
- $S \rightarrow a \rightarrow d \rightarrow t$, increment in flow along these edges is 9.
 - $S \rightarrow b \rightarrow d \rightarrow t$, increment in flow along these edges is 3.

Final network-flow graph is shown below. Maximum flow is 19. Edges in the minimum cut are (c,t) and (d,t) .



5) 16 pts

You are given an n-by-n grid, where each square (i, j) contains $c(i, j)$ gold coins. Assume that $c(i, j) \geq 0$ for all squares. You must start in the upper-left corner and end in the lower-right corner, and at each step you can only travel in one of the following three ways:

- 1- one square down which costs you 1 gold coin, or
- 2- one square to the right which costs you 2 gold coins, or
- 3- one square diagonally down and to the right which costs you 0 gold coins

When you visit any square, including your starting or ending square, you may collect all of the coins on that square. Give an algorithm to end up with the maximum number of coins and show how you can find the optimal path.

Solution:

let $\text{opt}[i, j]$ be the maximum number of coins that it is possible to collect while ending at (i, j) .

We have the following recurrence for $0 \leq i \leq n-1$, $0 \leq j \leq n-1$.

$$\text{opt}[i, j] = c(i, j) + \max(\text{opt}[i-1, j]-2, \text{opt}[i, j-1]-1, \text{opt}[i-1, j-1])$$

If we assume that borrowing, i.e., negative number of golds are fine the initial conditions are

$$\text{opt}(i, -1) = \text{opt}(j, -1) = -\infty, \text{opt}(0, 0) = c(0, 0)$$

if the assumption is that borrowing is not allowed the initial conditions are

$$\text{opt}(i, -1) = \text{opt}(j, -1) = -\infty, \text{opt}(0, 0) = c(0, 0)$$

$$\text{opt}(1, 0) = \begin{cases} -\infty, & c(0, 0) - 2 < 0 \\ c(0, 0) + c(1, 0) - 2, & c(0, 0) - 2 \geq 0 \end{cases}$$

$$\text{opt}(0, 1) = \begin{cases} -\infty, & c(0, 0) - 1 < 0 \\ c(0, 0) + c(0, 1) - 1, & c(0, 0) - 1 \geq 0 \end{cases}$$

There are n^2 subproblems, and each takes $O(1)$ time to solve. Thus, the running time is $O(n^2)$.

To find the optimal path one can trace back through the recursive formula for different i, j values.

6) 16 pts

You need to transport iron-ore from the mine to the factory. We would like to determine how long it takes to transport. For this problem, you are given a graph representing the road network of cities, with a list of k of its vertices (t_1, t_2, \dots, t_k) which are designated as factories, and one vertex S (the iron-ore mine) where all the ore is present. We are also given the following:

- Road Capacities (amount of iron that can be transported per minute) for each road (edges) between the cities (vertices).
- Factory Capacities (amount of iron that can be received per minute) for each factory (at t_1, t_2, \dots, t_k)

Give a polynomial-time algorithm to determine the minimum amount of time necessary to transport and receive all the iron-ore at factories, say C amount.

Here we want to define a network-flow graph with a source, sink, and capacities on the edges. In the given graph we already have a source S (the iron-ore mine) vertex. Add a new sink vertex T to the graph, and add an edge between each of the factory to T . On each of this newly added edge, set the factory capacity given for that factory. On the remaining edges set the capacities given by the road capacities.

We then run any polynomial-time max flow algorithm on the resulting graph; because it is polynomial in size (has only one additional vertex and at most n additional edges), the resulting runtime to compute max-flow (F) is polynomial.

Now we have the maximum rate at which we can transport the iron-ore. So it takes minimum of $\frac{C}{F}$ time to transport and receive all the iron-ore at factories.

1) 20 pts

Mark the following statements as **TRUE** or **FALSE**. No need to provide any justification.

[/FALSE]

If an iteration of the Ford-Fulkerson algorithm on a network places flow 1 through an edge (u, v) , then in every later iteration, the flow through (u, v) is at least 1.

[TRUE/]

For the recursion $T(n) = 4T(n/3) + n$, the size of each subproblem at depth k of the recursion tree is $n/3^{k-1}$.

[/FALSE]

For any flow network G and any maximum flow on G , there is always an edge e such that increasing the capacity of e increases the maximum flow of the network.

[/FALSE]

The asymptotic bound for the recurrence $T(n) = 3T(n/9) + n$ is given by $\Theta(n^{1/2} \log n)$.

[/FALSE]

Any Dynamic Programming algorithm with n subproblems will run in $O(n)$ time.

[/FALSE]

A pseudo-polynomial time algorithm is always slower than a polynomial time algorithm.

[TRUE/]

The sequence alignment algorithm can be used to find the longest common subsequence between two given sequences.

[/FALSE]

If a dynamic programming solution is set up correctly, i.e. the recurrence equation is correct and each unique sub-problem is solved only once (memoization), then the resulting algorithm will always find the optimal solution in polynomial time.

[TRUE/]

For a divide and conquer algorithm, it is possible that the divide step takes longer to do than the combine step.

[TRUE/]

Maximum value of an $s - t$ flow could be less than the capacity of a given $s - t$ cut in a flow network.

2) 16 pts

Recall the Bellman-Ford algorithm described in class where we computed the shortest distance from all points in the graph to t. And recall that we were able to find all shortest distance to t with only $O(n)$ memory.

How would you extend the algorithm to compute both the shortest distance and to find the actual shortest paths from all points to t with only $O(n)$ memory?

We need an array of size n to hold a pointer to the neighbor that gives us the shortest distance to t. Initially all pointers are set to Null. Whenever a node's distance to t is reduced, we update the pointer for that node and point it to the node that is giving us a lower distance to t. Once all shortest distances are computed, to find a path from any node v to t, one can simply follow these pointers to reach t on the shortest path.

3) 16 pts

During their studies, 7 friends (Alice, Bob, Carl, Dan, Emily, Frank, and Geoffrey) live together in a house. They agree that each of them has to cook dinner on exactly one day of the week. However, assigning the days turns out to be a bit tricky because each of the 7 students is unavailable on some of the days. Specifically, they are unavailable on the following days (1 = Monday, 2 = Tuesday, ..., 7 = Sunday):

- Alice: 2, 3, 4, 5
- Bob: 1, 2, 4, 7
- Carl: 3, 4, 6, 7
- Dan: 1, 2, 3, 5, 6
- Emily: 1, 3, 4, 5, 7
- Frank: 1, 2, 3, 5, 6
- Geoffrey: 1, 2, 5, 6

Transform the above problem into a maximum flow problem and draw the resulting flow network. If a solution exists, the flow network should indicate who will cook on each day; otherwise it must show that a feasible solution does not exist

Solution:

I will use the initials of each person's name to refer to them in this solution.

Construct a graph $G = (V, E)$. V consists of 1 node for each person (let us denote this set by $P = \{A, B, C, D, E, F, G\}$), 1 node for each day of the week (let's call this set $D = \{1, 2, 3, 4, 5, 6, 7\}$), a source node s , and a sink node t . Connect s to each node p in P by a directed (s, p) edge of unit capacity. Similarly, connect each node d in D to t by a directed (d, t) edge of unit capacity. Connect each node p in P by a directed edge of unit capacity to those nodes in D when p is **available** to cook. This completes our construction of the flow network. I am omitting the actual drawing of G here.

Finding a max-flow of value 7 in G translates to finding a feasible solution to the allocation of cooking days problem. Since there can be at most unit flow coming into any node p in P , a maximum of unit flow can leave it. Similarly, at most a flow of value 1 can flow into any node d in D because a maximum of unit flow can leave it. Thus, a max-flow of value 7 means that there exists a flow-carrying s - p - d - t path for each p and d . Any (p, d) edge with unit flow indicates that person p will cook on day d .

The following lists one possible max-flow of value 7 in G :

Send unit flow on each (s, p) edge and each (d, t) edge. Also send unit flow on the following (p, d) edges: $(A, 6)$, $(B, 5)$, $(C, 1)$, $(D, 4)$, $(E, 2)$, $(F, 7)$, $(G, 3)$

4) 20 pts

Suppose that there are n asteroids that are headed for earth. Asteroid i will hit the earth in time t_i and cause damage d_i unless it is shattered before hitting the earth, by a laser beam of energy e_i . Engineers at NASA have designed a powerful laser weapon for this purpose. However, the laser weapon needs to charge for a duration ce before firing a beam of energy e . Can you design a dynamic programming based pseudo-polynomial time algorithm to decide on a firing schedule for the laser beam to minimize the damage to earth? Assume that the laser is initially uncharged and the quantities c, t_i, d_i, e_i are all positive integers. Analyze the running time of your algorithm. You should include a brief description/derivation of your recurrence relation. Description of recurrence relation = 8pts, Algorithm = 6pts, Run Time = 6pts

Solution 1 (Assuming that the laser retains energy between firing beams): Sort all asteroids by the time t_i . Label the asteroids from 1 to n and without loss of generality assume $t_1 < t_2 < \dots < t_n$. Also assume that if asteroid i is destroyed then it is done exactly at time t_i (if the laser continuously accumulates energy then the destruction order of the asteroids does not change even if i^{th} asteroid is shot down before time t_i).

Define $OPT(i, T)$ as the minimum possible damage caused to earth due to asteroids $i, i + 1, \dots, n$ if $\frac{T}{c}$ energy is left in the laser just before time t_i . We want the solution corresponding to $OPT(1, t_1)$. If $T \geq ce_i$ and the i^{th} asteroid is destroyed then $OPT(i, T) = OPT(i + 1, T - ce_i + t_{i+1} - t_i)$, otherwise $OPT(i, T) = d_i + OPT(i + 1, T + t_{i+1} - t_i)$. Hence,

$$OPT(i, T) = \begin{cases} d_i + OPT(i + 1, T + t_{i+1} - t_i), & T < ce_i \\ \min\{d_i + OPT(i + 1, T + t_{i+1} - t_i), OPT(i + 1, T - ce_i + t_{i+1} - t_i)\}, & T \geq ce_i \end{cases}$$

Boundary condition: $OPT(n, T) = 0$ if $T \geq ce_n$ and $OPT(n, T) = d_n$ if $T < ce_n$, since if there is enough energy left to destroy the last asteroid then it is always beneficial to do so.

Furthermore, $T \leq t_n$ since a maximum of $\frac{t_n}{c}$ energy is accumulated by the laser before the last asteroid hits the earth and $T \geq 0$ since the left over energy in the laser is always non-negative.

Algorithm:

- i. Initialize $OPT(n, T)$ according to the boundary condition above for $0 \leq T \leq t_n$.
- ii. For each $n - 1 \geq i \geq 1$ and $0 \leq T \leq t_n$ populate $OPT(i, T)$ according to the recurrence defined above.
- iii. Trace forward through the two dimensional OPT array starting at $(1, t_1)$ to determine the firing sequence. For $1 \leq i \leq n - 1$, destroy the i^{th} asteroid with $\frac{T}{c}$ energy left if and only if $OPT(i, T) = OPT(i + 1, T - ce_i + t_{i+1} - t_i)$. Destroy the n^{th} asteroid only if $T \geq ce_n$.

Complexity: Step (i) initializes t_n values each taking constant time. Step (ii) computes $(n - 1)t_n$ values, each taking one invocation of the recurrence and hence is done in $O(nt_n)$ time. Trace back takes $O(n)$ time since the decision for each i takes constant time. Initial sorting takes $O(n \log n)$ time. Thus overall complexity is $O(nt_n + n \log n)$.

Solution 2 (Assuming that the laser does not retain energy left over after firing and if asteroid i is destroyed then it is done exactly at time t_i): Sort all asteroids by the time t_i . Label the asteroids from 1 to n and without loss of generality assume $t_1 < t_2 < \dots < t_n$. In contrast to Solution 1, the destroying sequence will change if we are free to destroy asteroid i before time t_i (this case is not solved here).

Define $OPT(i)$ to be the minimum possible damage caused to earth due to the first i asteroids. We want the solution corresponding to $OPT(n)$. If the i^{th} asteroid is not destroyed either by choice or because $t_i < ce_i$ then $OPT(i) = d_i + OPT(i - 1)$. On the other hand, if the i^{th} asteroid is destroyed ($t_i \geq ce_i$ is necessary) then none of the asteroids arriving between times $t_i - ce_i$ and t_i (both exclusive) can be destroyed. Letting $p[i]$ denote the largest positive integer such that $t_{p[i]} \leq t_i - ce_i$ (and $p[i] = 0$ if no such integer exists), we have $OPT(i) = OPT(p[i]) + \sum_{j=p[i]+1}^{i-1} d_j$ if $p[i] \leq i - 2$ and $OPT(i) = OPT(i - 1)$ if $p[i] = i - 1$. Hence for $i \geq 1$,

$$OPT(i) = \begin{cases} OPT(i - 1), & t_i \geq ce_i \text{ and } p[i] = i - 1 \\ d_i + OPT(i - 1), & t_i < ce_i \\ \min \left\{ d_i + OPT(i - 1), OPT(p[i]) + \sum_{j=p[i]+1}^{i-1} d_j \right\}, & t_i \geq ce_i \text{ and } p[i] \leq i - 2 \end{cases}$$

Boundary condition: $OPT(0) = 0$ since no asteroids means no damage.

Algorithm:

- i. Form the array p element-wise. This is done as follows.
 - a. Set $p[1] = 0$.
 - b. For $2 \leq i \leq n$, binary search for $t_i - ce_i$ in the sorted array $t_1 < t_2 < \dots < t_n$. If $t_i - ce_i < t_1$ then set $p[i] = 0$ else record $p[i]$ as the index such that $t_{p[i]} \leq t_i - ce_i < t_{p[i]+1}$.
- ii. Form array D to store cumulative sum of damages. Set $D[0] = 0$ and $D[j] = D[j - 1] + d_j$ for $1 \leq j \leq n$.
- iii. Set $OPT(0) = 0$ and for $1 \leq i \leq n$ populate $OPT(i)$ according to the recurrence defined above, computing $\sum_{j=p[i]+1}^{i-1} d_j$ as $D[i - 1] - D[p[i]]$.
- iv. Trace back through the one dimensional OPT array starting at $OPT(n)$ to determine the firing sequence. Destroy the first asteroid if and only if $OPT(1) = 0$. For $2 \leq i \leq n$, destroy the i^{th} asteroid if and only if either $OPT(i) = OPT(i - 1)$ with $p[i] = i - 1$ or $OPT(i) = OPT(p[i]) + D[i - 1] - D[p[i]]$ with $p[i] \leq i - 2$.

Complexity: initial sorting takes $O(n \log n)$. Construction of array p takes $O(\log n)$ time for each index and hence a total of $O(n \log n)$ time. Forming array D is done in $O(n)$ time. Using array D , $OPT(i)$ can be populated in constant time for each $1 \leq i \leq n$ and hence step (iii) takes $O(n)$ time. Trace back takes $O(n)$ time since the decision for each i takes constant time. Therefore, overall complexity is $O(n \log n)$.

5) 16 pts

Consider a two-dimensional array $A[1:n, 1:n]$ of integers. In the array each row is sorted in ascending order and each column is also sorted in ascending order. Our goal is to determine if a given value x exists in the array.

- a. One way to do this is to call binary search on each row (alternately, on each column). What is the running time of this approach? [2 pts]
 - b. Design another divide-and-conquer algorithm to solve this problem, and state the runtime of your algorithm. Your algorithm should take strictly less than $O(n^2)$ time to run, and should make use of the fact that each row and each column is in sorted order (i.e., don't just call binary search on each row or column). State the run-time complexity of your solution.
- a) $O(n \log n)$.
- b) Look at the middle element of the full matrix. Based on this, you can either eliminate $A[1.. \frac{n}{2}, 1.. \frac{n}{2}]$ or $A[\frac{n}{2}..n, \frac{n}{2}..n]$. If x is less than middle element then you can eliminate $A[\frac{n}{2}..n, \frac{n}{2}..n]$. If x is greater than middle element then you can eliminate $A[1.. \frac{n}{2}, 1.. \frac{n}{2}]$. You can then recursively search in the remaining three $\frac{n}{2} \times \frac{n}{2}$ matrices. The total runtime is $T(n) = 3T(\frac{n}{2}) + O(1)$, $T(n) = O(n^{\log_2 3})$.

6) 12 pts

Consider a divide-and-conquer algorithm that splits the problem of size n into 4 sub-problems of size $n/2$. Assume that the divide step takes $O(n^2)$ to run and the combine step takes $O(n^2 \log n)$ to run on problem of size n . Use any method that you know of to come up with an upper bound (as tight as possible) on the cost of this algorithm.

Solution: Use the generalized case 2 of Master's Theorem. For $T(n) = aT\left(\frac{n}{b}\right) + \Theta(n^{\log_b a} \log^k n)$ with $k \geq 0$, we have $T(n) = \Theta(n^{\log_b a} \log^{k+1} n)$.

The divide and combine steps together take $O(n^2 \log n)$ time and the worst case is that they actually take $\Theta(n^2 \log n)$ time. Hence the recurrence for the given algorithm is $T(n) = 4T\left(\frac{n}{2}\right) + \Theta(n^2 \log n)$ in the worst case. Comparing with the generalized case, $a = 4, b = 2, k = 1$ and so $T(n) = \Theta(n^2 \log^2 n)$. Since this expression for $T(n)$ is the worst case running time, an upper bound on the running time is $O(n^2 \log^2 n)$.

1) 20 pts

Mark the following statements as **TRUE** or **FALSE**. No need to provide any justification.

[**TRUE/FALSE**]

It is possible for a dynamic programming algorithm to have an exponential running time.

[**TRUE/FALSE**]

In a connected, directed graph with positive edge weights, the Bellman-Ford algorithm runs asymptotically faster than the Dijkstra algorithm.

[**TRUE /FALSE**]

There exist some problems that can be solved by dynamic programming, but cannot be solved by greedy algorithm.

[**TRUE /FALSE**]

The Floyd-Warshall algorithm is asymptotically faster than running the Bellman-Ford algorithm from each vertex.

[**TRUE /FALSE**]

If we have a dynamic programming algorithm with n^2 subproblems, it is possible that the space usage could be $O(n)$.

[**TRUE /FALSE**]

The Ford-Fulkerson algorithm solves the maximum bipartite matching problem in polynomial time.

[**TRUE /FALSE**]

Given a solution to a max-flow problem, that includes the final residual graph G_f . We can verify in a *linear* time that the solution does indeed give a maximum flow.

[**TRUE /FALSE**]

In a flow network, a flow value is upper-bounded by a cut capacity.

[**TRUE/FALSE**]

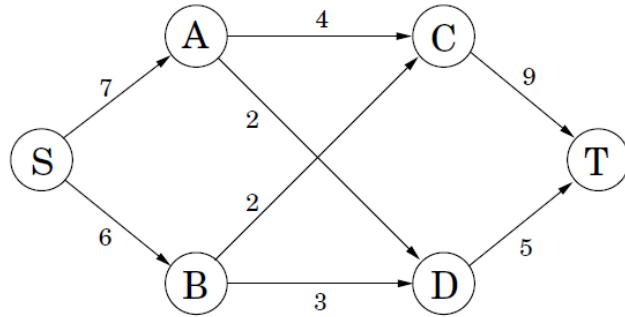
In a flow network, a min-cut is always unique.

[**TRUE/FALSE**]

A maximum flow in an integer capacity graph must have an integer flow on each edge.

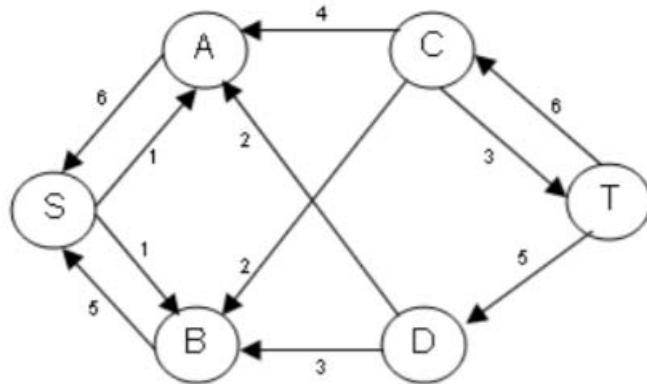
2) 20 pts.

You are given the following graph G . Each edge is labeled with the capacity of that edge.



- a) Find a max-flow in G using the Ford-Fulkerson algorithm. Draw the residual graph G_f corresponding to the max flow. You do not need to show all intermediate steps. (10 pts)

solution



Grading Rubics:

Residual graph: 10 points in total

For each edge in the residual graph, any wrong number marked, wrong direction, or missing will result in losing 1 point.

The total points you lose is equal to the number of edges on which you make any mistake shown above.

b) Find the max-flow value and a min-cut. (6 pts)

Solution: $f = 11$, cut: ($\{S, A, B\}$, $\{C, D, T\}$)

Grading Rubics:

Max-flow value and min-cut: 6 points in total

Max-flow: 2 points.

- If you give the wrong value, you lose the 2 points.

Min-cut: 4 points

- If your solution forms a cut but not a min-cut for the graph, you lose 3 points
- If your solution does not even form a cut, you lose all the 4 points

c) Prove or disprove that increasing the capacity of an edge that belongs to a min cut will always result in increasing the maximum flow. (4 pts)

Solution: increasing (B,D) by one won't increase the max-flow

Grading Rubics:

Prove or Disprove: 4 points in total

- If you judge it "True", but give a structural complete "proof". You get at most 1 point
- If you judge it "False", you get 2 points.
- If your counter example is correct, you get the rest 2 points.

Popular mistake: a number of students try to disprove it by showing that if the min-cut in the original graph is non-unique, then it is possible to find an edge in one min-cut set, such that increasing the capacity of this does not result in max-flow increase.

But they did not do the following thing:

The existence of the network with multiple min-cuts needs to be proved, though it seems to be obvious. The most straightforward way to prove the existence is to give an example-network that has multiple min-cuts. Then it turns out to be giving a counter example for the original question statement.

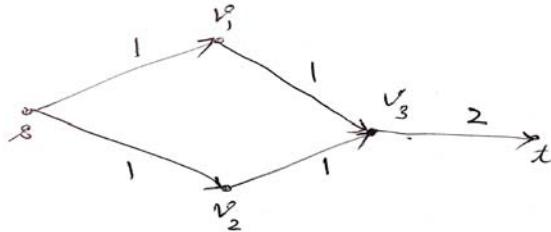
3) 15 pts.

Given a flow network with the source s and the sink t , and positive integer edge capacities c . Let (S, T) be a minimum cut. Prove or disprove the following statement:
If we increase the capacity of every edge by 1, then (S, T) still be a minimum cut.

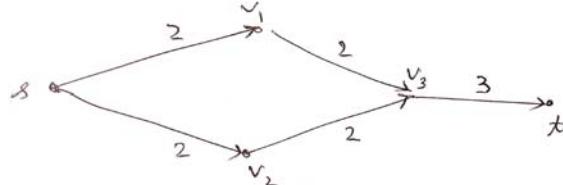
False. Create a counterexample.

An instance of a counter-example:

Initially, a (S, T) min cut is $S : \{s, v_1, v_2\}$ and $T : \{v_3, t\}$



After increasing capacity of every edge by 1, (S, T) is no longer a min-cut. We now have $S' : \{s, v_1, v_2, v_3\}$ and $T' : \{t\}$



Grading rubric:

If you try to prove the statement: -15

For an incorrect counter-example: -9

No credit for simply stating true or false.

4) 15 pts.

Given an unlimited supply of coins of denominations d_1, d_2, \dots, d_n , we wish to make change for an amount C . This might not be always possible. Your goal is to verify if it is possible to make such change. Design an algorithm by *reduction* to the knapsack problem.

- a) Describe reduction. What is the knapsack capacity, values and weights of the items? (10 pts.)

Capacity is C . (2 points)
values = weights = d_k . (4points)

- You need to recognize that the problem should be modeled based on the unbounded knapsack problem with description of the reduction: (5 points)
- Explanation of the verification criteria (4 points)
 $\text{Opt}(j)$: the maximum change equal or less than j that can be achieved with d_1, d_2, \dots, d_n
 $\text{Opt}(0)=0$
 $\text{Opt}(j) = \max[\text{opt}(j-d_i)+d_i] \text{ for } d_i \leq j$
If we obtain $\text{Opt}(C) = C$, it means the change is possible.

- b) Compute the runtime of your verification algorithm. (5 pts)

$O(nC)$ (3 points), Explanation (2 points).

5) 15 pts.

You are considering opening a series of electrical vehicle charging stations along Pacific Coast Highway (PCH). There are n possible locations along the highway, and the distance from the start to location k is $d_k \geq 0$, where $k = 1, 2, \dots, n$. You may assume that $d_i < d_k$ for $i < k$. There are two important constraints:

- 1) at each location k you can open only one charging station with the expected profit p_k , $k = 1, 2, \dots, n$.
- 2) you must open at least one charging station along the whole highway.
- 3) any two stations should be at least M miles apart.

Give a DP algorithm to find the maximum expected total profit subject to the given constraints.

- a) Define (in plain English) subproblems to be solved. (3 pts)

Let $\text{OPT}(k)$ be the maximum expected profit which you can obtain from locations $1, 2, \dots, k$.

Rubrics

Any other definition is okay as long as it will recursively solve subproblems

- b) Write the recurrence relation for subproblems. (6 pts)

$$\text{OPT}(k) = \max \{\text{OPT}(k - 1), p_k + \text{OPT}(f(k))\}$$

where $f(k)$ finds the largest index j such that $d_j \leq d_k - M$, when such j doesn't exist, $f(k)=0$

Base cases:

$$\text{OPT}(1) = p_1$$

$$\text{OPT}(0) = 0$$

Rubrics:

Error in recursion -2pts, if multiple errors, deduction adds up

No base cases: -2pts

Missing base cases: -1pts

Using variables without definition or explanation: -2pts

Overloading variables (re-defining n , p , k , or M): -2pts

- c) Compute the runtime of the above DP algorithm in terms of n . (3 pts)

algorithm solves n subproblems; each subproblem requires finding an index $f(k)$ which can be done in time $O(\log n)$ by binary search.
Hence, the running time is $O(n \log n)$.

Rubric:

$O(n^2)$ is regarded as okay

$O(d_n n)$ pseudo-polynomial is also okay if the recursion goes over all d_n values

$O(n)$ is also okay

$O(n^3), O(nM), O(kn)$: no credit

- d) How can you optimize the algorithm to have $O(n)$ runtime? (3 pts)

Preprocess distances $r_i = d_i - M$. Merge d-list with r-list.

Rubric

Claiming “optimal solution is already found in c ” only gets credit when explanation about pre-process is described in either part b or c.

Without proper explanation (e.g. assume we have, or we can do): no credit

Keeping an array of max profits: no credit, finding the index that is closest to the installed station with M distance away is the bottleneck, which requires pre-processing.

6) 15 pts.

A group of traders are leaving Switzerland, and need to convert their Francs into various international currencies. There are n traders t_1, t_2, \dots, t_n and m currencies c_1, c_2, \dots, c_m . Trader t_k has F_k Francs to convert. For each currency c_j , the bank can convert at most B_j Francs to c_j . Trader t_k is willing to trade as much as S_{kj} of his Francs for currency c_j . (For example, a trader with 1000 Francs might be willing to convert up to 200 of his Francs for USD, up to 500 of his Francs for Japanese's Yen, and up to 200 of his Francs for Euros). Assuming that all traders give their requests to the bank at the same time, describe an algorithm that the bank can use to satisfy the requests (if it can).

a) Describe how to construct a flow network to solve this problem, including the description of nodes, edges, edge directions and their capacities. (8 pts)

Bipartite graph: one partition traders t_1, t_2, \dots, t_n . Other, available currency, c_1, c_2, \dots, c_m .

Connect t_k to c_j with the capacity S_{kj}

Connect source to traders with the capacity F_k .

Connect available currency c_j to the sink with the capacity B_j .

Rubrics:

- Didn't include supersource (-1 point)
- Didn't include traders nodes (-1 point)
- Didn't include currencies nodes (-1 point)
- Didn't include supersink (-1 point)
- Didn't include edge direction (-1 point)
- Assigned no/wrong capacity on edges between source & traders (-1 point)
- Assigned no/wrong capacity on edges between traders & currencies (-1 point)
- Assigned no/wrong capacity on edges between currencies & sink (-1 point)

b) Describe on what condition the bank can satisfy all requests. (4 pts)

If there is a flow f in the network with $|f| = F_1 + \dots + F_n$, then all traders are able to convert their currencies.

Rubrics:

- Wrong condition (-4 points): Unless you explicitly included $|f| = F_1 + \dots + F_n$, no partial points were given for this subproblem.
- In addition to correct answer, added additional condition which is wrong (-2 points)
- No partial points were given for conditions that satisfy only some of the requests, not all requests.
- Note that $\sum S_{kj}$ and $\sum B_j$ can be larger than $\sum F_k$ in some cases where bank satisfy all requests.
- Note that $\sum S_{kj}$ can be larger than $\sum B_j$ in some cases where bank satisfy all requests.
- It is possible that $\sum S_{kj}$ or $\sum B_j$ is smaller than $\sum F_k$. In these cases, bank can never satisfy all requests because max flow will be smaller than $\sum F_k$.

- c) Assume that you execute the Ford-Fulkerson algorithm on the flow network graph in part a), what would be the runtime of your algorithm? (3 pts)

$O(n m |f|)$

Rubrics:

- Flow($|f|$) was not included (-1 point)
- Wrong description (-1 point)
- Computation is incorrect (-1 point)
- Notation error (-1 point)
- Missing big O notation (-1 point)
- Used big Theta notation instead of big O notation (-1 point)
- Wrong runtime complexity (-3 points)
- No runtime complexity is given (-3 points)

1) 20 pts

Mark the following statements as **TRUE** or **FALSE**. No need to provide any justification.

[**TRUE/FALSE**]

Dynamic programming only works on problems with non-overlapping subproblems.

[**TRUE/FALSE**]

If a flow in a network has a cycle, this flow is not a valid flow.

[**TRUE/FALSE**]

Every flow network with a non-zero max s-t flow value, has an edge e such that increasing the capacity of e increases the maximum s-t flow value.

[**TRUE/FALSE**]

A dynamic programming solution always explores the entire search space for all possible solutions.

[**TRUE/FALSE**]

Decreasing the capacity of an edge that belongs to a min cut in a flow network always results in decreasing the maximum flow.

[**TRUE/FALSE**]

Suppose f is a flow of value 100 from s to t in a flow network G . The capacity of the

minimum $s - t$ cut in G is equal to 100.

[**TRUE/FALSE**]

One can **efficiently** find the maximum number of edge disjoint paths from s to t in a directed graph by reducing the problem to max flow and solving it using the Ford-Fulkerson algorithm.

[**TRUE/FALSE**]

If all edges in a graph have capacity 1, then Ford-Fulkerson runs in linear time.

[**TRUE/FALSE**]

Given a flow network where all the edge capacities are even integers, the algorithm will require at most $C/2$ iterations, where C is the total capacity leaving the source s.

[**TRUE/FALSE**]

By combining divide and conquer with dynamic programming we were able to reduce the space requirements for our sequence alignment solution at the cost of increasing the computational complexity of our solution.

2) 20 pts.

Suppose that we have a set of students S_1, \dots, S_s , a set of projects P_1, \dots, P_p , a set of teachers T_1, \dots, T_t . A student is interested in a subset of projects. Each project p_i has an upper bound $K(P_i)$ on the number of the students that can work on it. A teacher T_i is willing to be the leader of a subset of the projects. Furthermore, he/she has an upper bound $H(T_i)$ on the number of students he/she is willing to supervise in total. We assume that no two teachers are willing to supervise the same project. The decision problem here is whether there is really a feasible assignment without violating any of the constraints in K and in H .

a) Solve the decision problem using network flow techniques. (15 pts)

The source node is connected to each student with the capacity of 1. Each student is connected to his/her desired projects with a capacity of 1.

Each project is connected to teachers who are willing to supervise it with a capacity of $K(P_i)$. Each teacher is connected to the sink node with the capacity of $H(T_i)$. If there is a max flow that is equal to the number of students, there is a solution to the problem. You can also convert the order of nodes and edges (source->teachers->projects->students->sink) as long as the edges are correct. You can have 2 sets of nodes for projects (inputs and outputs) as long as the output nodes do not constrain the flow.

b) Prove the correctness of your algorithm. (5 pts)

We need to show that

A – If there is a feasible assignment of students and teachers to projects, we can find a max flow of value s (# of students) in G

B – If we find a max flow of value s in G , we can find a feasible assignment of students and teachers to projects.

Rubric:

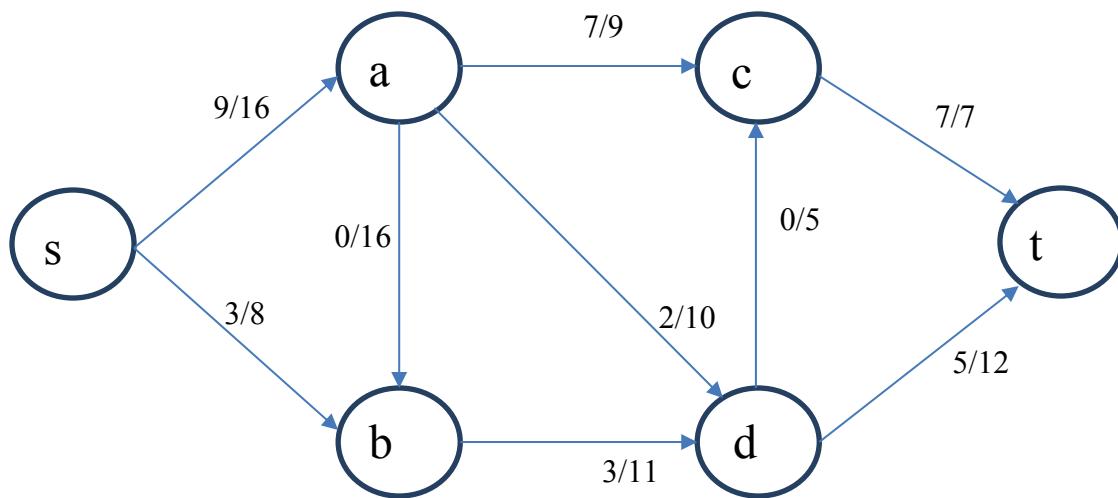
- 1) incorrect/missing nodes (incorrect nodes * -3)
- 2) incorrect/missing edges (incorrect edges * -2)
- 3) extra demand values (extra demands * -1)
- 4) not mentioning that max flow = number of students (-3)
- 5) if the proof is only provided for one side (-2)
- 6) if proof is not sufficient (-1)

3) 16 pts.

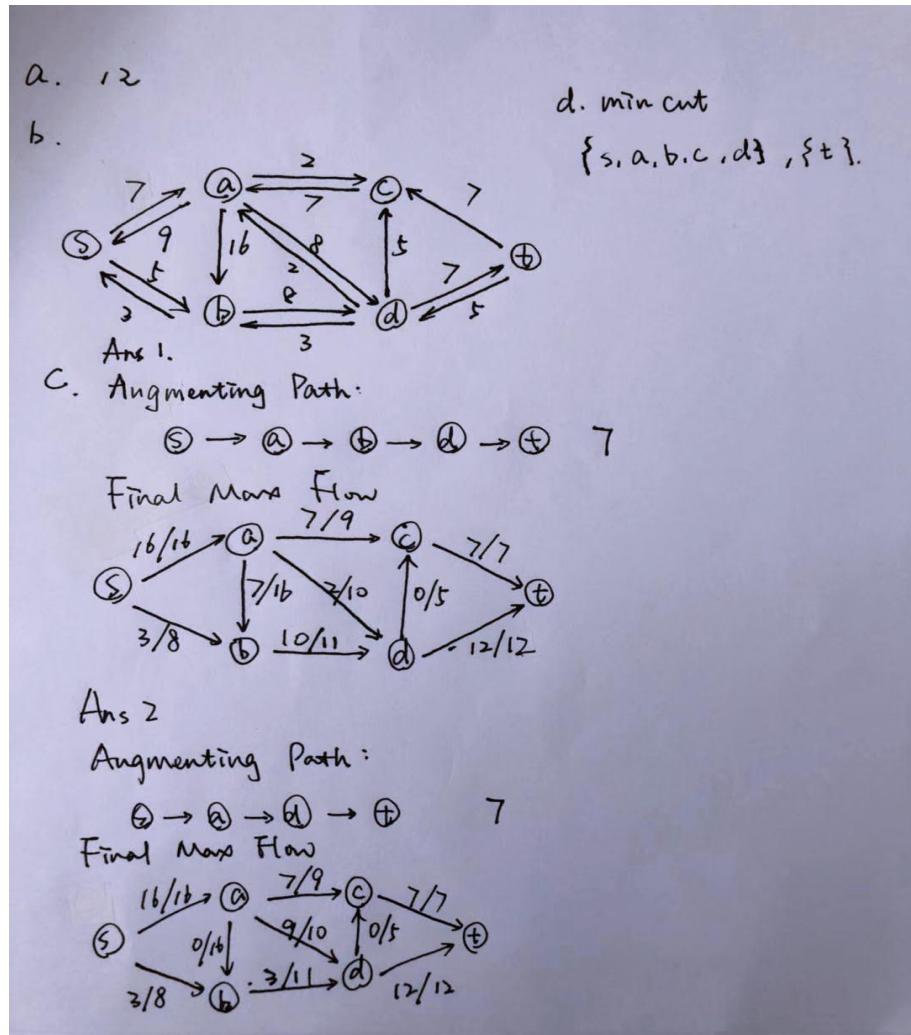
Consider the below flow-network, for which an s-t flow has been computed. The numbers x/y on each edge shows that the capacity of the edge is equal to y , and the flow sent on the edge is equal to x .

- What is the current value of flow? (2 pts)
- Draw the residual graph for the corresponding flow-network. (6 pts)
- Calculate the maximum flow in the graph using the Ford-Fulkerson algorithm. You need to show the augmenting path at each step and final max flow. (6 pts)
- Show the min cut found by the max flow you found in part c. (2 pts)

Note: extra space provided for this problem on next page



Solution:



Rubric:

a) Incorrect value -> 0 point

b) Every incorrect/missing edge or capacity -> -1 point

c) Every possible flow which has a capacity of 19 is correct.

Answers without correct value(other than) 19 will get 0 point.

Answers with correct value/final max flow/final residual graph without augmenting path/detail steps will get 3 points

Answers with correct value/final max flow/final residual graph with augmenting path&detail steps will get 6 points

Answers with a correct augmenting path but incomplete max flow will get 1 point.

d) Incorrect set -> 0 point

4) 16 pts

A symmetric sequence or palindrome is a sequence of characters that is equal to its own reversal; e.g. the phrase “a man a plan a canal panama” forms a palindrome (ignoring the spaces between words).

a) Describe how to use the **sequence alignment algorithm** (described in class) to find a longest symmetric subsequence (longest embedded palindrome) of a given sequence. (14 pts)

Example: CTGACCTAC ‘s longest embedded palindromes are CTCCTC and CACCAC

Solution: given the sequence S , reverse the string to form S^r . Then find the longest common substring between S and S^r by setting all mismatch costs to ∞ and $\delta=1$ (or anything greater than zero) .

b) What is the run time complexity of your solution? (2 pts)

$O(n^2)$

Rubric 4a)

- 10 points for reversing the string and using the sequence alignment algorithm
 - If the recurrence is given without explicitly mentioning the sequence alignment algorithm is fine.
 - Recurrence is discussed in class
- 2 points for allocating the correct mismatch value.
- 2 points for allocating the correct gap score.
- Any other answer which does not use sequence alignment algorithm is wrong. **The question explicitly asks to use the sequence alignment algorithm.** (0 to 2 points)
 - 2 points for correctly mentioning any other algorithm to get palindrome within a string
- Wrong answers
 - Dividing the string into two and reversing the second half.
 - Will not work, as the entire palindrome can be located within the first or the second half
 - Aligning the original string with the copy of the same string
 - This will return the original string and not the palindrome
 - Any other algorithm which finds the palindrome in a string

Rubric 4b)

2 points

0 points for any other answer

5) 20 pts

Let's say you have a dice with m sides numbered from 1 to m , and you are throwing it n times. If the side with number i is facing up you will receive i points. Consider the variable Z which is the summation of your points when you throw the dice n times. We want to find the number of ways we can end up with the summation Z after n throws. We want to do this using dynamic programming.

a) Define (in plain English) subproblems to be solved. (4 pts)

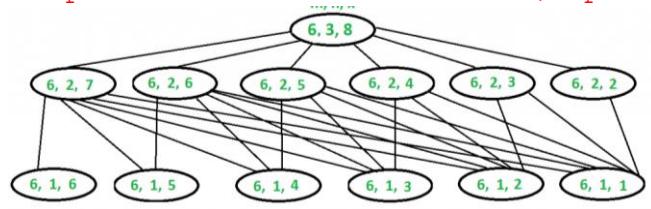
$\text{OPT}(n, Z)$ = number of ways to get the sum Z after n rolls of dice.

b) Write the recurrence relation for subproblems. (6 pts)

$\text{OPT}(n, Z) = \sum_{i=1 \text{ to } m} \text{OPT}(n-1, Z-i)$

Example: Given, $m=6$, Calculate: $\text{OPT}(3, 8)$

Subproblems are as follows (represented as (m, n, Z)):



Rubric:

-6 if not summing over m

-4 if summing over m but OPT is somewhat incorrect.

c) Using the recurrence formula in part b, write pseudocode to compute the number of ways to obtain the value SUM . (6 pts)

Make sure you have initial values properly assigned. (2 pts)

```
OPT(1, i) = 1 for i=1 to MIN(m, SUM)
OPT(1, i) = 0, for i=MIN(m, SUM) to MAX(m, SUM)
For i = 2 to n
    For j = 1 to SUM
        OPT(i, j) = 0
        For k = 1 to MIN(m, i)
            OPT(i, j) = OPT(i, j) + OPT(i-1, j-k)
        Endfor
    Endfor
Endfor
```

Return OPT(n, SUM)

Example, Given, m=6, Calculate: OPT(3, 8)

OPT Table looks as follows:

0	0	0	0	0	0	0	0	0
0	1	1	1	1	1	1	0	0
0	0	1	2	3	4	5	6	5
0	0	0	1	3	6	10	15	21

Alternatively, you can initialize $\text{OPT}(0,0) = 1$ and start the iteration of i from 1.

Rubric:

- 1 for each initialization missed.
- 6 for any incorrect answer. (e.g incorrect recurrence relation, incorrect loops, etc)
i.e. no partial grading for any pseudocode producing incorrect answer.

- d) Compute the runtime of the algorithm described in part c and state whether your solution runs in polynomial time or not (2 pts)

Time Complexity: $O(m * n * Z)$ where m is number of sides, n is the number of times we roll the dice, and Z is given sum. This is pseudo polynomial since the complexity depends on the numerical value of input terms.

Rubric:

- Not graded if part c is incorrect.
- 2 if stated as polynomial time.
- 2 if incorrect complexity equation.

6) 8 pts

- a) Given a flow network $G=(V, E)$ with integer edge capacities, a max flow f in G , and a specific edge e in E , design a linear time algorithm that determines whether or not e belongs to a min cut. (6 pts)

Solution:

Reduce capacity of e by 1 unit. - $O(1)$

Find an s - t path containing e and subtract 1 unit of s - t flow on that path. For this, considering e 's adjacent nodes to be (u, v) , you can run BFS from source to find an s - u path and BFS from v to find an v - t path which gives you an s - u - v - t path crossing e . - $O(|V|+|E|)$

Then construct the new residual graph G_f - $O(|E|)$

If there is an augmenting s - t path in G_f then e does not belong to a min cut, otherwise it does.

- b) Describe why your algorithm runs in linear time.

You can find this using BFS from source(only one iteration of the FF algorithm) - $O(|V| + |E|)$

All the above steps are linear in the size of input, therefore the entire algorithm is linear.

Rubric:

If providing the complete algorithm: full points

- If not decreasing the s - t flow by 1: -1 points
- If running the entire Ford-Fulkerson algorithm to find the flow: -2 points
- If providing the overall idea without description of implementation steps: -3 or -4 points depending on your description

If the provided algorithm is not scalable to graphs having multiple(more than two) min-cuts(for example using BFS to find reachable/unreachable nodes from source/sink): -4 points

- Incomplete or incorrect description of the above algorithm (-1 or -2 additional points depending on your description)

If removing the edge and finding max-flow: -3 or -4 points depending on your description (since this algorithm doesn't run in linear time)

If only considering saturated edges to check for min-cut edges: -7 points

Adding the capacity of e and checking the flow: no points (this approach is not checking for min-cut)

Checking all possible cuts in the graph: no points (this takes exponential time)

Other algorithms: no points

1) 20 pts

Mark the following statements as **TRUE** or **FALSE**. No need to provide any justification.

[**TRUE/FALSE**] True

Max flow problems can in general be solved using greedy techniques.

[**TRUE/FALSE**] False

If all edges have unique capacities, the network has a unique minimum cut.

[**TRUE/FALSE**] True

Flow f is maximum flow if and only if there are no augmenting paths.

[**TRUE/FALSE**] True

Suppose a maximum flow allocation is known. Increase the capacity of an edge by 1 unit. Then, updating a max flow can be easily done by finding an augmenting path in the residual flow graph.

[**TRUE/FALSE**] False

In order to apply divide & conquer algorithm, we must split the original problem into at least half the size.

[**TRUE/FALSE**] True

If all edge capacities in a graph are integer multiples of 5 then the maximum flow value is a multiple of 5.

[**TRUE/FALSE**] False

If all directed edges in a network have distinct capacities, then there is a unique maximum flow.

[**TRUE/FALSE**] True

Given a bipartite graph and a matching pairs, we can determine if the matching is maximum or not in $O(V+E)$ time

[**TRUE/FALSE**] False

Maximum flow problem can be efficiently solved by dynamic programming

[**TRUE/FALSE**] True

The difference between dynamic programming and divide and conquer techniques is that in divide and conquer sub-problems are independent

2) 10pts

Give tight bound for the following recursion

a) $T(n)=T(n/2)+T(n/4)+n$

A simple method is that it is easy to see that $T(n)=4n$ satisfy the recursive relation.
Hence $T(n)=O(n)$

b) $T(n) = 2T(n^{0.5})+\log n$

Let $n=2^k$ and $k=2^m$, we have $T(2^k)=2T(2^{k/2})+k$ and $T(2^{k/2})=2T(2^{k/4})+k/2$
Hence $T(2^k)=2(2T(2^{k/4})+k/2)+k=2^2 T(2^{k/4})+2k=2^3 T(2^{k/8})+3k=\dots=2^m T(1)+mk$
 $=2^m T(1)+m2^m$. Note that $n=2^k$ and $k=2^m$, hence $m=\log \log n$. We have $T(n)=T(1)\log n+(\log \log n)*(\log n)$.
Hence $T(n)=(\log \log n)*(\log n)$

3) 20 pts

Let $A = (a_0, a_1, \dots, a_{n-1})$ be an array of n positive integers.

a- Present a divide-and-conquer algorithm which computes the sum of all the even integers a_i in $A(1..n-1)$ where $a_i > a_{i-1}$. Solutions other than divide and conquer will receive no credit.

Algorithm: Compute-Even-Integers(l, r)

if $r=l+1$

 if(($a_r \% 2 = 0$) and ($a_r > a_l$))

 return a_r

 else

 return 0

else

 let $m = \left\lfloor \frac{l+r}{2} \right\rfloor$

 Let $S = \text{Compute-Even-Integers}(l, m) + \text{Compute-Even-Integers}(m, r)$

 If $a_m \% 2 = 0$ and $a_m > a_{m-1}$

$S = S + a_m$

To computes the sum of all the even integers a_i in $A(1..n-1)$ where $a_i > a_{i-1}$, invoke Compute-Even-Integers($0, n-1$)

b- Show a recurrence which represents the number of additions required by your algorithm.

$$T(n) = 2T(n/2) + c$$

c- Give a tight asymptotic bound for the number of additions required by your algorithm.

By master theorem, it is easy to see that $T(n)=O(n)$

d- Discuss how your approach would compare in practice to an iterative solution of the problem.

In practice, both methods are $O(n)$ algorithm. Their complexity is the same.

4) 20 pts

Families 1,...,N go out for dinner together. To increase their social interaction, no two members of the same family use the same table. Family j has $a(j)$ members.

There are M tables. Table j can seat $b(j)$ people. Find a valid seating assignment if one exists.

We first construct a bipartite graph G whose nodes are the families $f_i (i=1,\dots,N)$ and the tables $t_j (j=1,\dots,M)$. We add edge $e_{ij} = (f_i, t_j)$ in the graph for $i=1,\dots,N$ and $j=1,\dots,M$ and set $e_{ij}=1$. Then we add a source s and sink t in G. For each family f_i , we add $e=(s, f_i)$ in the graph and set $c_e=a(i)$. For each table t_j , we add $e=(t_j, t)$ and set $c_e=b(j)$. After building the graph G for the original problem, we find the maximum s-t-flow value v in graph G by Fulkerson algorithm. If v is $a(1)+\dots+a(N)$, then we make the seating assignment such that no two members of the same family use the same table, otherwise we can not.

To prove the correctness of the algorithm, we prove that no two members of the same family use the same table if and only if the value of the maximum value of an s-t flow in G is $a(1)+\dots+a(N)$:

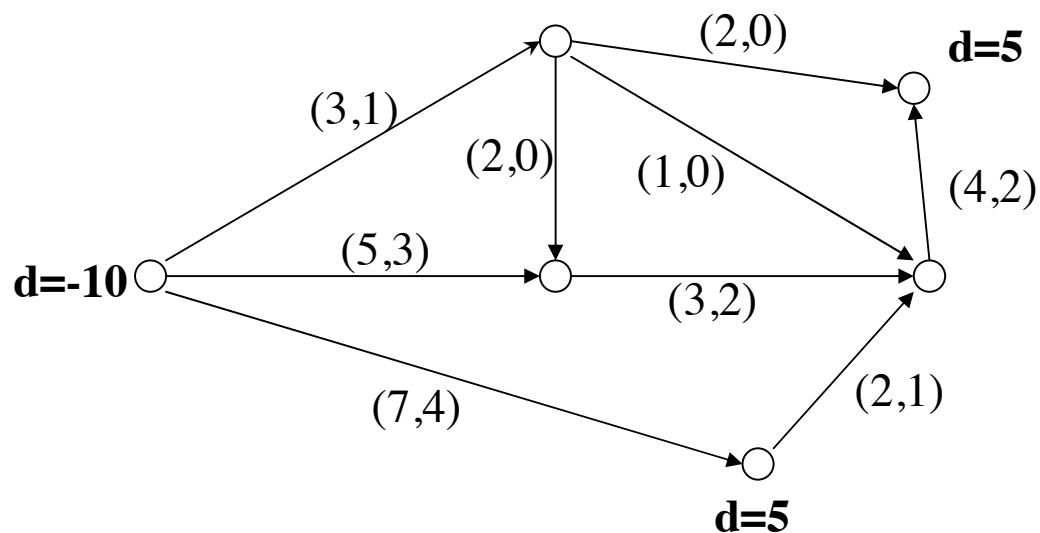
First if no two members of the same family use the same table, it is easy to see that this flow meets all capacity constraints and has value $a(1)+\dots+a(N)$.

For the converse direction, assume that there is an s-t flow of value $a(1)+\dots+a(N)$. The construction given in the algorithm makes sure that there is at most one member in family j sitting in the table j as the capacity of the flow goes from f_i to t_j is 1. So we only need to make sure that all families are seated. Note that $\{s\}, V-\{s\}$ is an s-t cut with value $a(1)+\dots+a(N)$, so the flow must saturate all edges crossing the cut. Therefore, all families must be sitting in some tables. This completes the correctness proof.

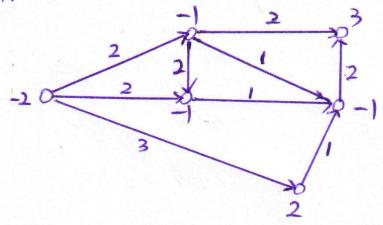
Running time: The time complexity of constructing the graph is $O(MN)$. The number of edges in the graph is $MN+M+N$, and $v(f)=a(1)+\dots+a(N)$. Let $T=a(1)+\dots+a(N)$, then the running time applying Ford-Fulkerson algorithm is $O(MNT)$.

5) 20 pts

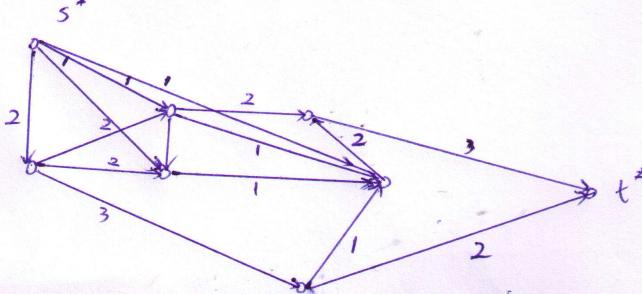
Determine if there is a feasible circulation in the below network. If so, determine the circulation, if not show where the bottlenecks are. The numbers in parentheses are (*lowerbound, upperbound*) on flow.



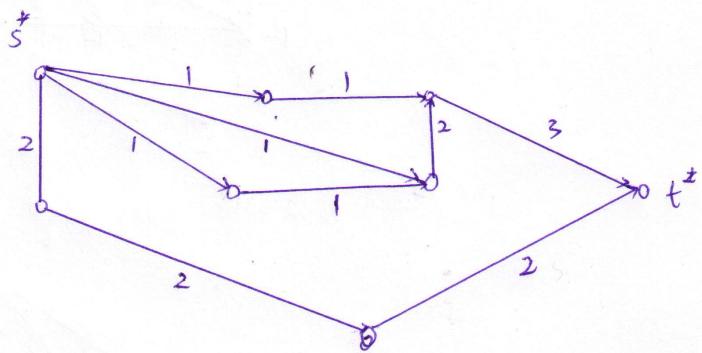
Step 1:



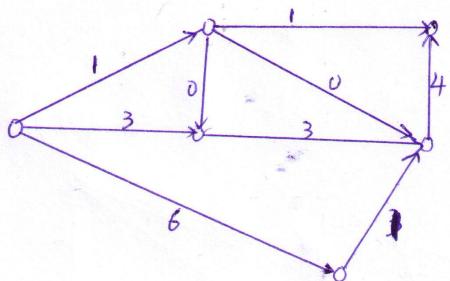
Step 2: the corresponding max-flow problem



Step 3. Solving the max-flow problem and the result is as follows:

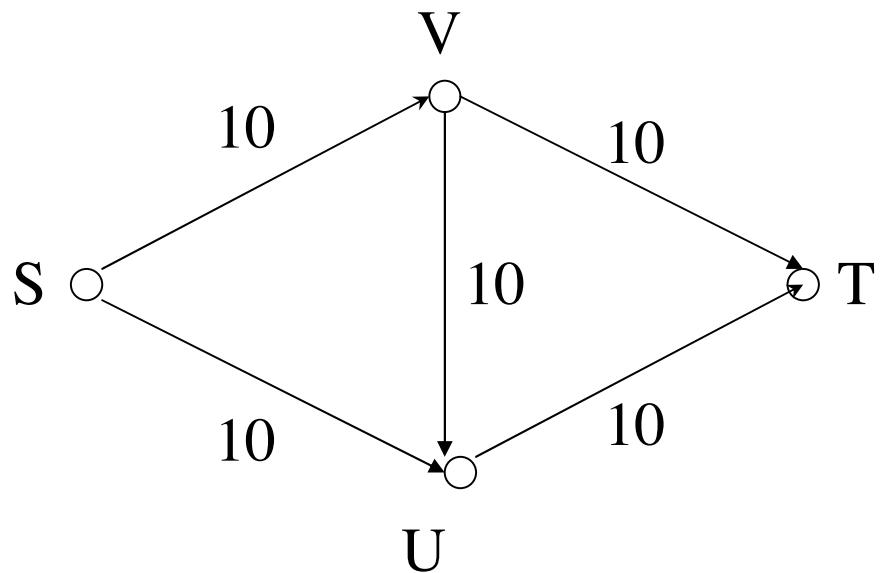


Step 4. The circulation is



6) 5 pts

List all minimum s-t cuts in the flow network pictured below. Capacity of every edge is equal to 10.



$$C1 = \{\{S\}, \{V, U, T\}\}$$

$$C2 = \{\{S, U\}, \{V, T\}\}$$

$$C3 = \{\{S, U, V\}, \{T\}\}$$

7) 5 pts

Show an example of a graph in which poor choices of augmenting paths can lead to exactly C iterations of the Ford-Fulkerson algorithm before achieving max flow. (C is the sum of all edge capacities going out of the source and must be much greater than the number of edges in the network) Specifically, draw the network, mark up edge capacities, and list the sequence of augmenting paths.

Any example satisfying that the condition in this question is fine when you list the sequence of augmenting paths.

1) 20 pts

Mark the following statements as **TRUE**, **FALSE**. No need to provide any justification.

[**TRUE**]

If all capacities in a network flow are rational numbers, then the maximum flow will be a rational number, if exist.

[**TRUE**]

The Ford-Fulkerson algorithm is based on the greedy approach.

[**FALSE**]

The main difference between divide and conquer and dynamic programming is that divide and conquer solves problems in a top-down manner whereas dynamic-programming does this bottom-up.

[**FALSE**]

The Ford-Fulkerson algorithm has a polynomial time complexity with respect to the input size.

[**TRUE**]

Given the Recurrence, $T(n) = T(n/2) + \theta(1)$, the running time would be $O(\log(n))$

[**FALSE**]

If all edge capacities of a flow network are increased by k, then the maximum flow will be increased by at least k.

[**TRUE**]

A divide and conquer algorithm acting on an input size of n can have a lower bound less than $\Omega(n \log n)$.

[**TRUE**]

One can actually prove the correctness of the Master Theorem.

[**TRUE**]

In the Ford Fulkerson algorithm, choice of augmenting paths can affect the number of iterations.

[**FALSE**]

In the Ford Fulkerson algorithm, choice of augmenting paths can affect the min cut.

2) 15 pts

Present a divide-and-conquer algorithm that determines the minimum difference between any two elements of a sorted array of real numbers.

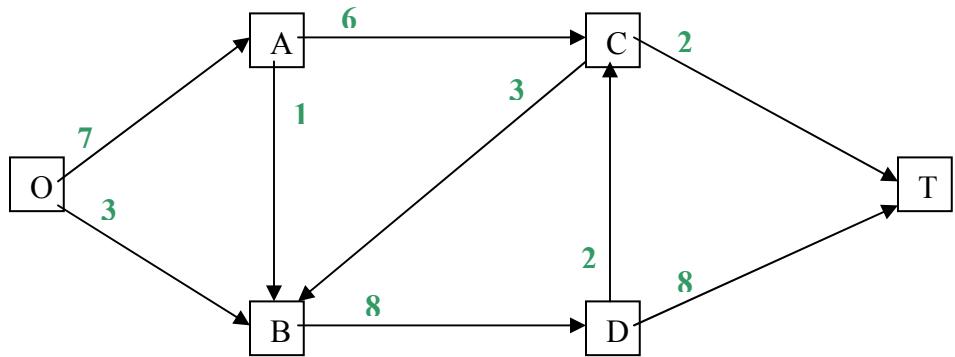
Key feature: The min difference can always been achieved between a pair of neighbors in the array, as the array is sorted.

```
int Min_Diff(first, last)
{
    if (last >= first)
        return inf;
    else
        return min(Min_Diff(first, (first + last)/2), Min_Diff((first + last)/2+1,
last), abs(number[(first + last)/2+1] - number[(first + last)/2]));
}
```

The complexity is liner to the array size.

3) 15 pts

You are given the following directed network with source O and sink T.

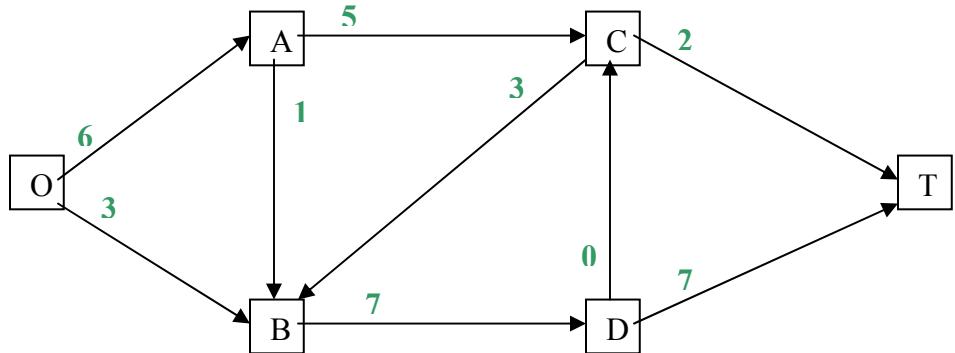


a) Find a maximum flow from O to T in the network.

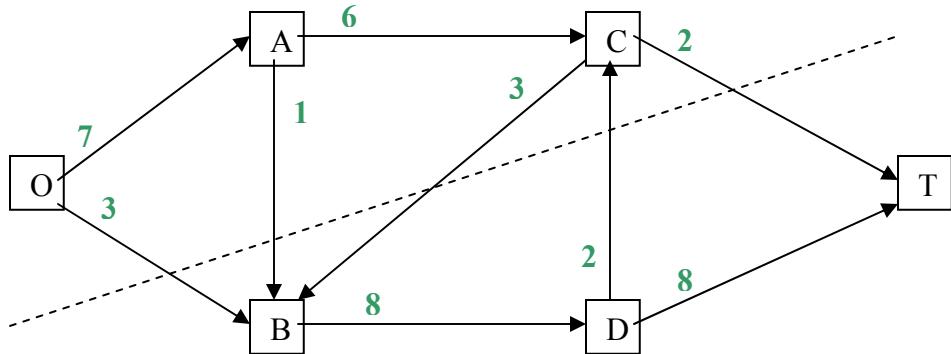
Augmenting paths and flow pushing amount:

OACT	2
OBDT	3
OABDT	1
OACBDT	3

And the maximum flow here is with weight 9:



b) Find a minimum cut. What is its capacity?



Capacity of this min cut is 9.

4) 15 pts

Solve the following recurrences

a) $T(n) = 2T(n/2) + n \log n$

According to the master theorem, $T(n) = \Theta(n \log^2 n)$.

Or we can solve it like this:

$$\begin{aligned}
 T(n) &= 2T\left(\frac{n}{2}\right) + n \log n = 2\left(2T\left(\frac{n}{4}\right) + \frac{n}{2} \log n - \frac{n}{2} \log 2\right) + n \log n \\
 &= 4T\left(\frac{n}{4}\right) + 2n \log n - n \log 2 = \dots = 2^k T\left(\frac{n}{2^k}\right) + kn \log n - \frac{k(k-1)}{2} n \log 2 \\
 &= \dots =_{(k=\log n)} nT(1) + \Theta(n \log^2 n) = \Theta(n \log^2 n)
 \end{aligned}$$

$$b) \quad T(n) = 2T(n/2) + \log n$$

Similar to a), the result is $T(n) = \Theta(n)$.

$$c) \quad T(n) = 2T(n-1) - T(n-2) \text{ for } n \geq 2; \quad T(0) = 3; \quad T(1) = 3$$

It is very easy to find out that for the initial values $T(0)=T(1)$, we always have $T(i)=T(0)$, $i > 0$. Thus $T(n) = 3$.

5) 20 pts

You are given a flow network with integer capacity edges. It consists of a directed graph $G = (V, E)$, a source s and a destination t , both belong to V . You are also given a parameter k . The goal is to delete k edges so as to reduce the maximum flow in G as much as possible. Give an efficient algorithm to find the edges to be deleted. Prove the correctness of your algorithm and show the running time.

We here introduce a straightforward algorithm (assuming $k \leq |E|$, otherwise just return failure):

```
Delete_k_edges()
{
    E' = E;
    for i=1 to k
    {
        curr_Max_Flow = inf;
        for j in E'
            if Max_Flow(V, E'-j) < curr_Max_Flow
            {
                curr_Max_Flow = Max_Flow(V, E'-j);
                index[i] = j;
            }
        E' = E' - index[i];
    }
}
```

Then the final E' is a required edge set, and indices of all k deleted edges are stored in the array $\text{index}[]$.

Running time is $O(k |E| \cdot T(\text{max_flow}))$, depending on the max_flow algorithm used here, the time complexity varies: if Edmonds_Karp is used here the time would be $O(k |V| |E|^3)$; if Dinic or other more advanced algorithm is used here the time complexity can be reduced.

Proof hint:

By induction.

$k = 1$, the algorithm is correct.

Assume $k = i$ the algorithm is correct. Then we prove for $k = i+1$, it is also correct. Here, it is better to divide this $i+1$ into the first step and the following i steps, not vice versa.

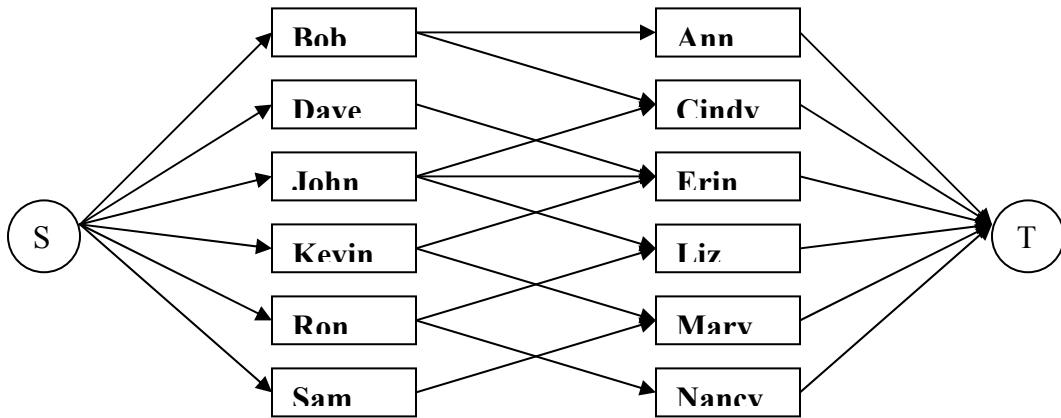
6) 15 pts

Six men and six women are at a dance. The goal of the matchmaker is to match each woman with a man in a way that maximizes the number of people who are matched with compatible mates. The table below describes the compatibility of the dancers.

	Ann	Cindy	Erin	Liz	Mary	Nancy
Bob	C	C	-	-	-	-
Dave	-	-	C	-	-	-
John	-	C	C	C	-	-
Kevin	-	-	C	-	C	-
Ron	-	-	-	C	-	C
Sam	-	-	-	-	C	-

Note: C indicates compatibility.

- a) Determine the maximum number of compatible pairs by reducing the problem to a max flow problem.



All edges are with capacity 1.

Run some maximum flow algorithm like Edmonds-Karp, it would guarantee to return a 0-1 solution within polynomial time, with represents the required match.

- b) Find a minimum cut for the network of part (a).

$A = \{S, \text{Dave}, \text{Kevin}, \text{Sam}, \text{Erin}, \text{Mary}\}$ and $A' = V - A$ constitute a minimum cut, with capacity 5.

- c) Give the list of pairs in the maximum pairs set.

Maximum 5 pairs. One solution:

Bob-Ann, Dave-Erin, John-Cindy, Ron-Nancy, Sam-Mary.

[TRUE/FALSE] TRUE

The problem of deciding whether a given flow f of a given flow network G is maximum flow can be solved in linear time.

[TRUE/FALSE] TRUE

If you are given a maximum $s - t$ flow in a graph then you can find a minimum $s - t$ cut in time $O(m)$.

[TRUE/FALSE] TRUE

An edge that goes straight from s to t is always saturated when maximum $s - t$ flow is reached.

[TRUE/FALSE] FALSE

In any maximum flow there are no cycles that carry positive flow.
(A cycle $\langle e_1, \dots, e_k \rangle$ carries positive flow iff $f(e_1) > 0, \dots, f(e_k) > 0$.)

[TRUE/FALSE] TRUE

There always exists a maximum flow without cycles carrying positive flow.

[TRUE/FALSE] FALSE

In a directed graph with at most one edge between each pair of vertices, if we replace each directed edge by an undirected edge, the maximum flow value remains unchanged.

[TRUE/FALSE] FALSE

The Ford-Fulkerson algorithm finds a maximum flow of a unit-capacity flow network (all edges have unit capacity) with n vertices and m edges in $O(mn)$ time.

[TRUE/FALSE] FALSE

Any Dynamic Programming algorithm with n unique subproblems will run in $O(n)$ time.

[TRUE/FALSE] FALSE

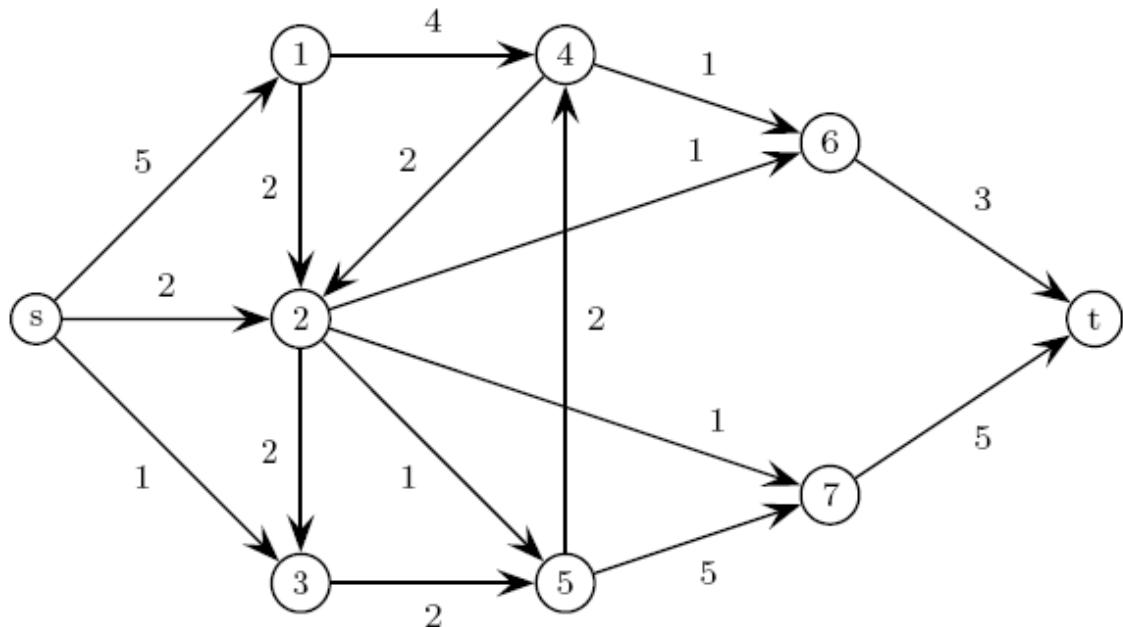
The running time of a pseudo polynomial time algorithm depends polynomially on the size of the input

[TRUE/FALSE] FALSE

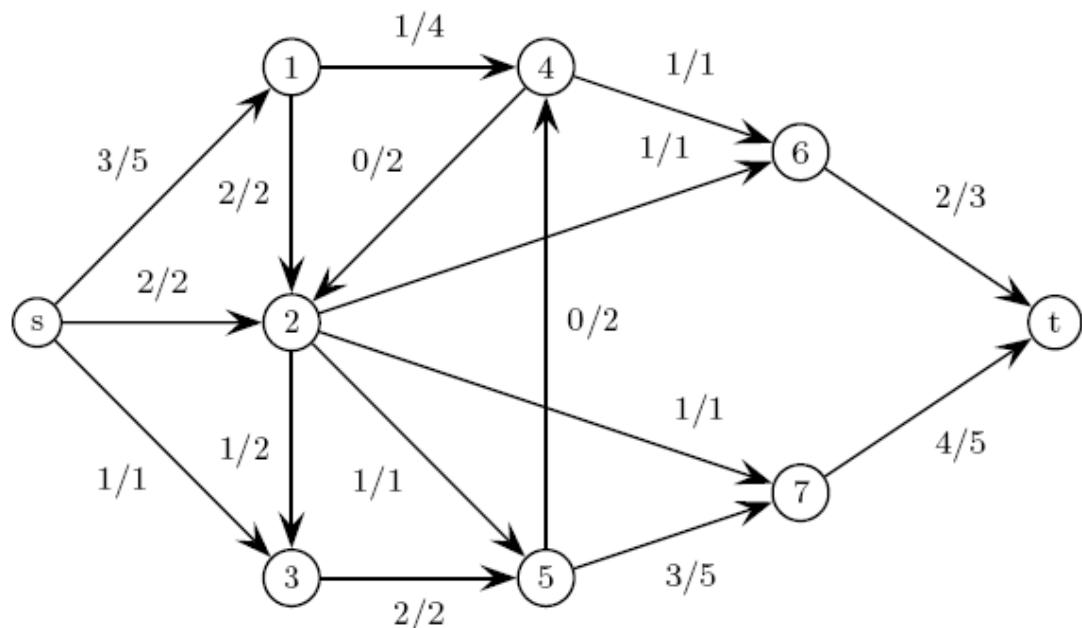
In dynamic programming you must calculate the optimal value of a subproblem twice, once during the bottom up pass and once during the top down pass.

2) 20 pts

- a) Give a maximum s - t flow for the following graph, by writing the flow f_e above each edge e . The printed numbers are the capacities. You may write on this exam sheet.



Solution:



(b) Prove that your given flow is indeed a max-flow.

Solution:

The cut ($\{s: 1, 2, 3, 4\}$, $\{5, 6, 7\}$) has capacity 6. The flow given above has value 6. No flow can have value exceeding the capacity of any cut, so this proves that the flow is a max-flow (and also that the cut is a min-cut).

3) 20 pts

On a table lie n coins in a row, where the i th coin from the left has value $x_i \geq 0$. You get to pick up any set of coins, so long as you never pick up two adjacent coins. Give

a polynomial-time algorithm that picks a (legal) set of coins of maximum total value. Prove that your algorithm is correct, and runs in polynomial time.

We use Dynamic Programming to solve this problem. Let $\text{OPT}(i)$ denote the maximum value that can be picked up among coins $1, \dots, i$.

Base case: $\text{OPT}(0) = 0$, and $\text{OPT}(1) = x_1$.

Considering coin i , there are two options for the optimal solution: either it includes coin i , or it does not. If coin i is not included, then the optimal solution for coins $1, \dots, i$ is the same as the one for coins $1, \dots, i-1$. If coin i is included in the optimal solution, then coin $i-1$ cannot be included, but among coins $1, \dots, i-2$, the optimum subset is included, as a non-optimum one could be replaced with a better one in the solution for i . Hence, the recursion is

$$\text{OPT}(0) = 0$$

$$\text{OPT}(1) = x_1$$

$$\text{OPT}(i) = \max(\text{OPT}(i-1), x_i + \text{OPT}(i-2)) \text{ for } i > 1.$$

Hence, we get the algorithm Coin Selection below: The correctness of the computation follows because it just implements the recurrence which we proved correct above. The output part just traces back the array for the solution constructed previously, and outputs the coins which have to be picked to make the recurrence work out.

The running time is $O(n)$, as for each coin, we only compare two values.

4) 20 pts

We assume that there are n tasks, with time requirements r_1, r_2, \dots, r_n hours. On the project team, there are k people with time availabilities a_1, a_2, \dots, a_k . For each task i

and person j , you are told if person j has the skills to do task i . You are to decide if the tasks can be split up among the people so that all tasks get done, people only execute tasks they are qualified for, and no one exceeds his time availability. Remember that you can split up one task between multiple qualified people. Now, in addition, there are group constraints. For instance, even if each of you and your two roommates can in principle spend 4 hours on the project, you may have decided that between the three of you, you only want to spend 10 hours. Formally, we assume that there are m sets $S_j \subseteq \{1, \dots, k\}$ of people, with set constraints t_j . Then, any valid solution must ensure, in addition to the previous constraints, that the combined work of all people in S_j does not exceed t_j , for all j .

Give an algorithm with running time polynomial in n, m, k for this problem, under the assumption that all the S_j are disjoint, and sketch a proof that your algorithm is correct.

Solution:

We will have one node u_h for each task h , one node v_i for each person i , and one node w_j for each constraint set S_j . In addition, there is a source s and a sink t . As before, the source connects to each node u_h with capacity r_h . Each u_h connects to each node v_i such that person i is able to do task h , with infinite capacity. If a person i is in no constraint set, node v_i connects to the sink t with capacity a_i . Otherwise, it connects to the node w_j for the constraint set S_j with $i \in S_j$, with capacity a_i . (Notice that because the constraint sets are disjoint, each person only connects to one set.) Finally, each node w_j connects to the sink t with capacity t_j .

We claim that this network has an s - t flow of value at least $\sum_h r_h$ if and only if the tasks can be divided between people. For the forward direction, assume that there is such a flow. For each person i , assign him to do as many units of work on task h as the flow from u_h to v_i . First, because the flow saturates all the edges out of the source (the total capacity out of the source is only $\sum_h r_h$), and by flow conservation, each job is fully assigned to people. Because the capacity on the (unique) edge out of v_i is a_i , no person does more than a_i units of work. And because the only way to send on the flow into v_i is to the node w_j for nodes $i \in S_j$, by the capacity constraint on the edge (w_j, t) , the total work done by people in S_j is at most t_j .

Conversely, if we have an assignment that has person i doing x_{ih} units of work on task h , meeting all constraints, then we send x_{ih} units of flow along the path $s-u_h-v_i-t$ (if person i is in no constraint sets), or along $s-u_h-v_i-w_j-t$, if person i is in constraint set S_j . This clearly satisfies conservation and non-negativity. The flow along each edge (s, u_h) is exactly r_h , because that is the total amount of work assigned on job h . The flow along the edge (v_i, t) or (v_i, s_j) is at most a_i , because that is the maximum amount of work assigned to person i . And the total flow along (w_j, t) is at most t_j , because each constraint was satisfied by the assignment. So we have exhibited an s - t flow of total value at least $\sum_h r_h$.

5) 20 pts

There are n trading posts along a river numbered $n, n-1 \dots 3, 2, 1$. At any of the posts you can rent a canoe to be returned at any other post downstream. (It is

impossible to paddle against the river since the water is moving too quickly). For each possible departure point i and each possible arrival point $j (< i)$, the cost of a rental from i to j is known. It is $C[i, j]$. However, it can happen that the cost of renting from i to j is higher than the total costs of a series of shorter rentals. In this case you can return the first canoe at some post k between i and j and continue your journey in a second (and, maybe, third, fourth . . .) canoe. There is no extra charge for changing canoes in this way. Give a dynamic programming algorithm to determine the minimum cost of a trip by canoe from each possible departure point i to each possible arrival point j . Analyze the running time of your algorithm in terms of n . For your dynamic programming solution, focus on computing the minimum cost of a trip from trading post n to trading post 1 , using up to each intermediate trading post.

Solution

Let $\text{OPT}(i, j) =$ The optimal cost of reaching from departure point “ i ” to departure point “ j ”.

Now, lets look at $\text{OPT}(i, j)$. assume that we are at post “ i ”. If we are not making any intermediate stops, we will directly be at “ j ”. We can potentially make the first stop, starting at “ i ” to any post between “ i ” and “ j ” (“ j ” included, “ i ” not included)

This gets us to the following recurrence

$$\text{OPT}(i, j) = \min(\text{over } j \leq k < i)(C[i, k] + \text{OPT}(k, j))$$

The base case is $\text{OPT}(i, i) = C[i, i] = 0$ for all “ i ” from 1 to n

The iterative program will look as follows

Let $\text{OPT}[i, j]$ be the array where you will store the optimal costs. Initialize the 2D array with the base cases

```
for (i=1;i<=n;i++)
{
    for (j=1;j<=i-i;j++)
    {
        calculate OPT(i,j) with the recurrence
    }
}
```

Now, to output the cost from the last post to the first post, will be given by $\text{OPT}[n, 1]$

As for the running time, we are trying to fill up all $\text{OPT}[i, j]$ for $i < j$. Thus, there are $O(n^2)$ entries to fill (which corresponds to the outer loops for “ i ” and “ j ”) In each loop, we could potentially be doing at most k comparisons for the min operation . This is $O(n)$ work. Therefore, the total running time is $O(n^3)$

1) 20 pts

Mark the following statements as **TRUE** or **FALSE**. No need to provide any justification.

In the final residual graph constructed during the execution of the Ford–Fulkerson Algorithm, there's no path from source to sink.

TRUE

In a flow network whose edges all have capacity 1, the maximum flow value equals the maximum degree of a vertex in the flow network.

FALSE.

Memoization is the basis for a top-down alternative to the usual bottom-up approach to dynamic programming solutions.

TRUE

The time complexity of a dynamic programming solution is always lower than that of an exhaustive search for the same problem.

FALSE

If we multiply all edge capacities in a graph by 5, then the new maximum flow value is the original one multiplied by 5.

TRUE.

For any graph G with edge capacities and vertices s and t , there always exists an edge such that increasing the capacity on that edge will increase the maximum flow from s to t . (Assume that there is at least one path in the graph from s to t .)

FALSE.

There might be more than one min-cut, by increasing the edge capacity of one min-cut doesn't have to impact the other one.

Let G be a weighted directed graph with exactly one source s and exactly one sink t . Let (A, B) be a maximum cut in G , that is, A and B are disjoint sets whose union is V , $s \in A, t \in B$, and the sum of the weights of all edges from A to B is the maximum for any two such sets. Now let H be the weighted directed graph obtained by adding 1 to the weight of each edge in G . Then (A, B) must still be a maximum cut in H .

FALSE

There could exist other edge which has many more cross-edges, which was not max-cut previously, to become the new max-cut.

A recursive implementation of a dynamic programming solution is often less efficient in practice than its equivalent iterative implementation.

We give points for both TRUE and FALSE answers due to the ambiguity of "often".

Ford-Fulkerson algorithm will always terminate as long as the flow network G has edges with strictly positive capacities.

FALSE

If some edges are irrational numbers, Ford-Fulkerson may never terminate.

Any problem that can be solved using dynamic programming has a polynomial time worst case time complexity with respect to its input size.

FALSE

2) 20 pts

Judge the following statement is true or false. If true, prove it. If false, give a counter-example.

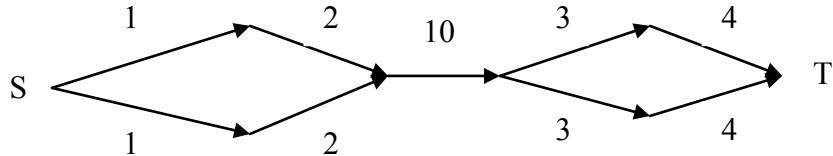
Given a directed graph, if deleting edge e reduces the original maximum flow more than deleting any other edge does, then edge e must be part of a minimum s-t cut in the original graph.

Solution:

False.

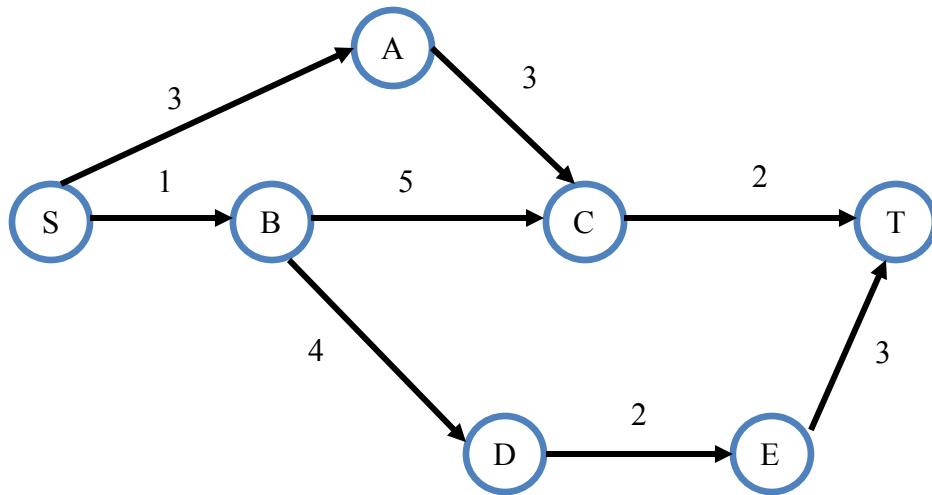
Counter-example:

Apparently deleting the edge with capacity 10 will reduce the flow by the most, while it is not part of minimum s-t cut.



Comment: some of you misunderstood the problem in different ways. Some counter-examples have multiple min s-t cuts and the edge "e" is actually in one of them. And some others failed to satisfy the precondition "more than... any other", and have some equal alternatives, which are also incorrect.

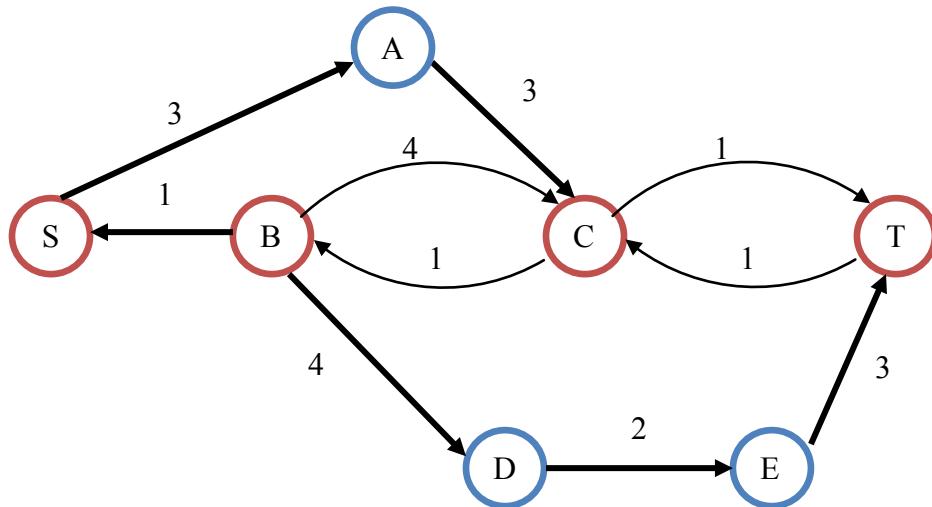
3) 20 pts



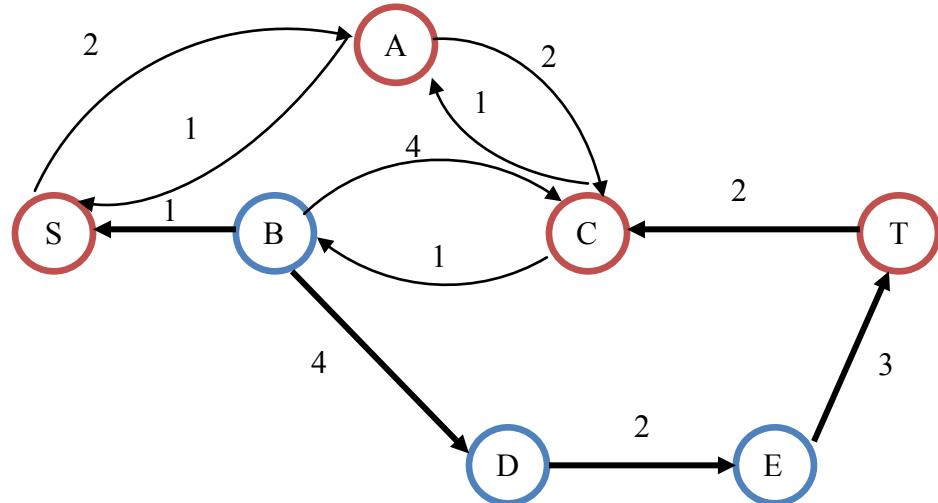
- a- Show all steps involved in finding maximum flow from S to T in above flow network using the Ford-Fulkerson algorithm.
- b- What is the value of the maximum flow?
- c- Identify the minimum cut.

a.

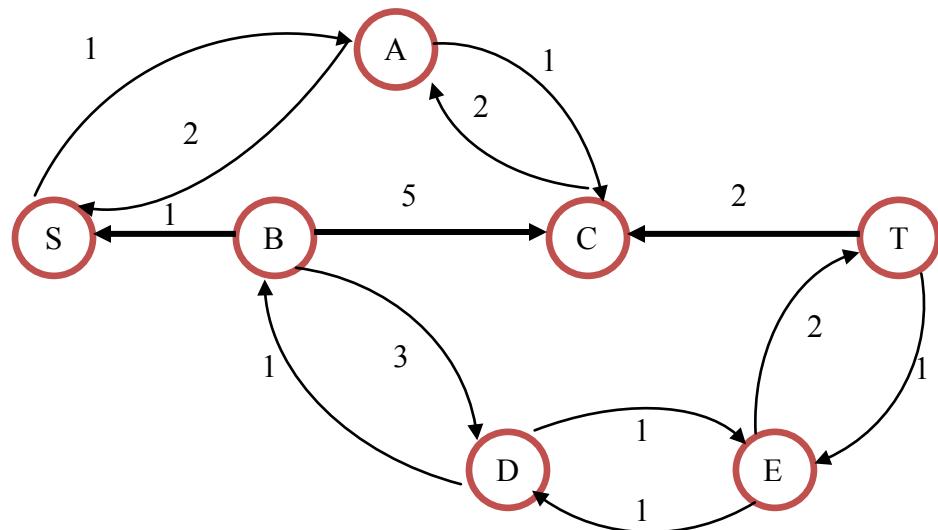
1. Augment path: [S, B, C, T], bottleneck is 1, residual graph:



2. Augment path: [S, A, C, T], bottleneck is 1, residual graph:



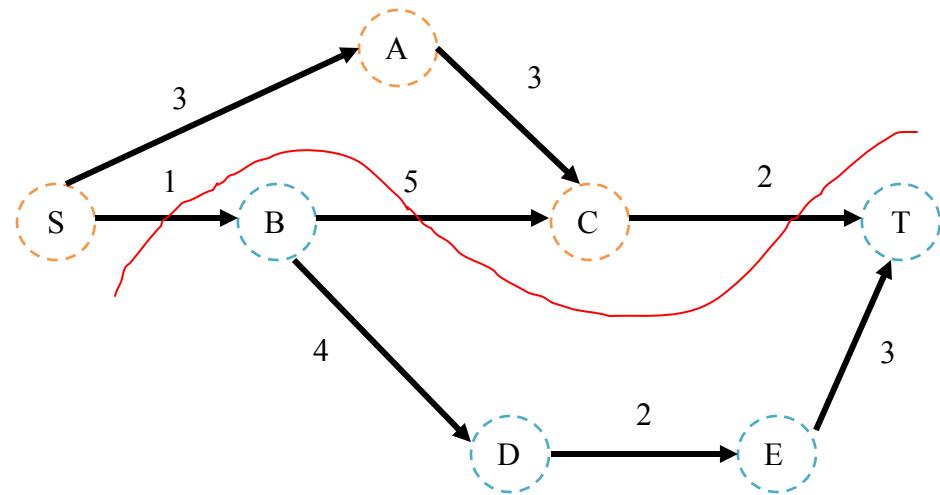
3. Augment path: [S, A, C, B, D, E, T], bottleneck is 1, residual graph:



4. No forward path from S to T, terminate.

b. The maximum flow is 3.

c. The Min-Cut is: [S, A, C] and [B, D, E, T].



4) 20 pts

The words 'computer' and 'commuter' are very similar, and a change of just one letter, p→m will change the first word into the second. The word 'sport' can be changed into 'sort' by the deletion of the 'p', or equivalently, 'sort' can be changed into 'sport' by the insertion of 'p'.

The edit distance of two strings, s_1 and s_2 , is defined as the minimum number of point mutations required to change s_1 into s_2 , where a point mutation is one of:

1. change a letter,
 2. insert a letter or
 3. delete a letter
- a) Design an algorithm using dynamic programming techniques to calculate the edit distance between two strings of size at most n . The algorithm has to take less than or equal to $O(n^2)$ for both time and space complexity.
- b) Print the edits.

The following recurrence relations define the edit distance, $d(s_1, s_2)$, of two strings s_1 and s_2 :

1. $d(., .) = 0$ (for empty strings)
2. $d(s, .) = d(., s) = |s|$ (length of s)
3. $d(s_1 + ch_1, s_2 + ch_2) = \min(d(s_1, s_2) + 0, if ch_1 = ch_2, d(s_1 + ch_1, s_2) + 1, d(s_1, s_2 + ch_2) + 1)$

The first two rules above are obviously true, so it is only necessary consider the last one. Here, neither string is the empty string, so each has a last character, ch_1 and ch_2 respectively. Somehow, ch_1 and ch_2 have to be explained in an edit of $s_1 + ch_1$ into $s_2 + ch_2$.

- If ch_1 equals ch_2 , they can be matched for no penalty, which is 0, and the overall edit distance is $d(s_1, s_2)$.
- If ch_1 differs from ch_2 , then ch_1 could be changed into ch_2 , costing 1, giving an overall cost $d(s_1, s_2) + 1$.
- Another possibility is to delete ch_1 and edit s_1 into $s_2 + ch_2$, $d(s_1, s_2 + ch_2) + 1$.
- The last possibility is to edit $s_1 + ch_1$ into s_2 and then insert ch_2 , $d(s_1 + ch_1, s_2) + 1$.

There are no other alternatives. We take the least expensive, **min**, of these alternatives.

Examination of the relations reveals that $d(s_1, s_2)$ depends only on $d(s'_1, s'_2)$ where s'_1 is shorter than s_1 , or s'_2 is shorter than s_2 , or both. This allows the *dynamic programming* technique to be used.

A two-dimensional matrix, $m[0..|s1|, 0..|s2|]$ is used to hold the edit distance values:

```

m[i,j] = d(s1[1..i], s2[1..j])

m[0,0] = 0
m[i,0] = i,   i=1..|s1|
m[0,j] = j,   j=1..|s2|

m[i,j] = min(m[i-1,j-1] + if s1[i]=s2[j] then 0 else 1,
              m[i-1, j] + 1,
              m[i, j-1] + 1 ),  i=1..|s1|, j=1..|s2|

```

$m[.]$ can be computed *row by row*. Row $m[i,]$ depends only on row $m[i - 1,]$. The time complexity of this algorithm is $O(|s1| * |s2|)$. If $s1$ and $s2$ have a similar length n , this complexity is $O(n^2)$.

b)

Once the algorithm terminates, we have generated the edit distance matrix m , we can do a trace-back for all the edits, e.g. Figure 1:

		S	a	t	u	r	d	a	y
	0	1	2	3	4	5	6	7	8
S	1	0	1	2	3	4	5	6	7
u	2	1	1	2	2	3	4	5	6
n	3	2	2	2	3	3	4	5	6
d	4	3	3	3	3	4	3	4	5
a	5	4	3	4	4	4	4	3	4
y	6	5	4	4	5	5	5	4	3

Figure 1 Edit distance matrix (from Wikipedia)

Here we are using a head recursion so that it prints the path from beginning to the end.

```

Print_Edits(n1, n2)
if n = 0 then
    Output nothing
else
    Find (i,j) ∈ {(n1 - 1, n2), (n1, n2 - 1), (n1 - 1, n2 - 1)} that has the minimum value from
    m[i,j].
    Print_Edits(i,j)
    if m[i,j] = m(n1,n2) then

```

```
    Do nothing
else
    if  $(i, j) = (n_1 - 1, n_2 - 1)$  then
        Print “change s1’s  $n_1^{th}$  letter into s2’s  $n_2^{th}$  letter””
    else if  $(i, j) = (n_1, n_2 - 1)$  then
        Print “insert s2’s  $n_2^{th}$  letter after s1’s  $n_1^{th}$  position””
    else
        Print “delete s1’s  $n_1^{th}$  letter””
```

The trace-back process has the complexity of $O(n)$.

5) 20 pts

Given a 2-dimensional array of size $m \times n$ with only “0” or “1” in each position, starting from position [1, 1] (top left corner), every time you can only move either one cell to the right or one cell down (of course you must remain in the array). Give an $O(mn)$ algorithm to calculate the number of different paths without reaching “0” in any step, starting from [1, 1] and ending in [m, n].

Solution:

We can solve this problem using dynamic programming. We denote $a[i][j]$ as the array, $\text{num}[i][j]$ as the number of different paths without reaching "0" in any step starting from [1, 1] and ending in [i, j]. Then the desired solution is $\text{num}[m][n]$.

We have

$$\text{num}[i][j] = \begin{cases} 0 & i=0 \text{ or } j=0 \text{ or } a[i][j]=0 \\ a[1][1] & i=1 \text{ and } j=1 \\ \text{num}[i-1][j] + \text{num}[i][j-1] & \text{other} \end{cases}$$

This is simply because we can reach cell [i, j] by coming from either [i-1, j] or [i, j-1].

When $a[i][j]=1$, $\text{num}[i][j]$ is just the sum of the two num's from the two different directions.

By calculating num row by row, we can get $\text{num}[m][n]$ at last. This is an $O(mn)$ solution since we use $O(1)$ time to calculate each $\text{num}[i][j]$ and the size of array num is $O(mn)$.

Comment: this is different from the "number of disjoint paths" problem which can be solved by being reduced to max flow problem. And some used max function instead of plus. Some other used recursion, but without memorization which would course the complexity become exponential. And some other tried to find paths and count, which is also an approach with exponential complexity.

1) 20 pts

Mark the following statements as **TRUE** or **FALSE**. No need to provide any justification.

[**FALSE**]

A flow network with unique edge capacities has a unique min cut.

[**TRUE**/]

If a problem can be solved by dynamic programming, then it can always be solved by exhaustive search (Brute Force).

[**TRUE**/]

A divide and conquer algorithm acting on an input size of n can have a lower bound less than $\Theta(n \log n)$.

[**FALSE**]

If a flow in a network has a cycle, this flow is not a valid flow.

[**FALSE**]

In the divide and conquer algorithm to compute the closest pair among a given set of points on the plane, if the sorted order of the points on both X and Y axis are given as an added input, then the running time of the algorithm improves to $O(n)$.

[**TRUE**/]

In a flow network, an edge that goes straight from s to t is always saturated when maximum $s - t$ flow is reached.

[**FALSE**]

The Bellman-Ford algorithm always fails to find the shortest path between two nodes in a graph if there is a negative cycle present in the graph.

[**TRUE**/]

If f is a max $s-t$ flow of a flow network G with source s and sink t , then the capacity of the min $s-t$ cut in the residual graph G_f is 0.

[**FALSE**]

In a dynamic programming solution, the space requirement is always at least as big as the number of unique sub problems.

[**FALSE**]

Decreasing the capacity of an edge that belongs to a min cut in a flow network may not result in decreasing the maximum flow.

2) 20 pts

A city is located at one node x in an undirected network $G = (V, E)$ of channels. There is a big river beside the network. In the rainy season, the flood from the river flows into the network through a set of nodes Y . Assume that the flood can only flow along the edges of the network. Let c_{uv} (integer value) represent the minimum effort (counted in certain effort unit) of building a dam to stop the flood flowing through edge (u, v) . The goal is to determine the minimum total effort of building dams to prevent the flood from reaching the city. Give a pseudo-polynomial time algorithm to solve this problem. Justify your algorithm.

Algorithm:

1. Add node s , and add edges from s to each node in Y .
2. Set the capacity of each of these new edges as infinity.
3. Set city node x as the sink node. Then the flood network G becomes a larger network G' with source s and sink x .
4. Find the min s - x cut on G' by running a polynomial time max-flow algorithm, where you can treat the cost of building a dam on each edge to be the capacity of this edge.
5. Build a dam on each edge in the min-cut's cut-set.

Complexity:

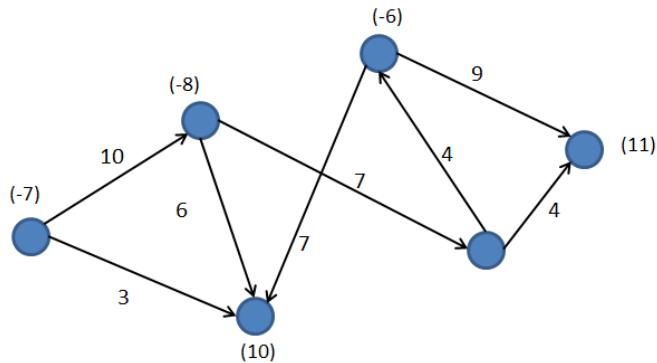
Since the construction of the new edges takes $O(|Y|)$ times, together with a polynomial time max-flow algorithm to find the min-cut, the entire algorithm takes polynomial time.

Justification:

We're simply trying to separate x from Y by choosing edges with the minimum cost, which is a min-cut problem. But min-cut only works when Y is a single node. We fix this by creating a super source s directly connected to Y . But we want to make sure the min-cut's cut-set doesn't include any edge connecting s , because these edges do not belong to G . This is why we put the capacities arbitrarily large on these edges. Min-cut would then find the min-cost way to separate x from Y .

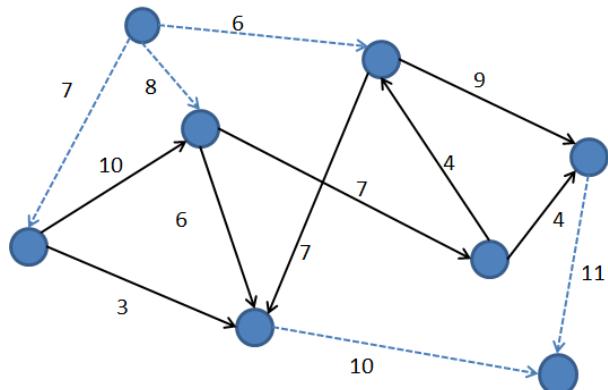
3) 20 pts

The following graph G is an instance of a circulation problem with demands. The edge weights represent capacities and the node weights (in parentheses) represent demands. A negative demand implies source.



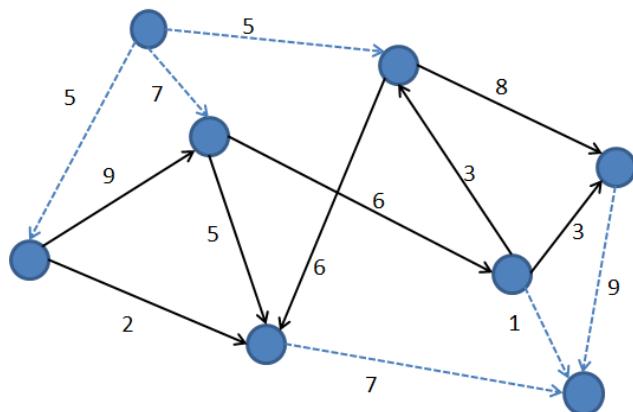
(i) Transform this graph into an instance of max-flow problem.

Solution:



(ii) Now, assume that each edge of G has a constraint of lower bound of 1 unit, i.e., one unit must flow along all edges. Find the new instance of max-flow problem that includes the lower bound constraint.

Solution:



4) 20 pts

There is a series of activities lined up one after the other, J_1, J_2, \dots, J_n . The i^{th} activity takes T_i units of time, and you are given M_i amount of money for it. Also for the i^{th} activity, you are given N_i , which is the number of immediately following activities that you cannot take if you perform that i^{th} activity. Give a dynamic programming solution to maximize the amount of money one can make in T units of time. Note that an activity has to be completed in order to make any money on it. State the runtime of your algorithm.

Solution:

Recurrence formula:

If $T_i > t$ then $\text{opt}(i, t) = \text{opt}(i+1, t)$,

Otherwise, $\text{opt}(i, t) = \max(\text{opt}(i+1, t), \text{opt}(i + N_i + 1, t - T_i) + M_i)$

Boundary conditions: $\text{opt}(i, t) = 0$ for $i > n$ and all t ,

$\text{opt}(i, t) = 0$ for $t < 1$ and all i

To find the value of the optimal solution:

for $i=n$ to 1 by -1

 for $t=1$ to T

 If $T_i > t$ then $\text{opt}(i, t) = \text{opt}(i+1, t)$,

 Otherwise, $\text{opt}(i, t) = \max(\text{opt}(i+1, t), \text{opt}(i + N_i + 1, t - T_i) + M_i)$

 end for

end for

$\text{opt}(1, T)$ will hold the value of the optimal solution (maximum money that can be made). Complexity is $O(nT)$

Given all the values in the two dimensional $\text{opt}()$ array, the optimal set of activities can be found in $O(n)$

Overall complexity of the algorithm is $O(nT)$ which is pseudopolynomial.

5) 20 pts

A polygon is called convex if all of its internal angles are less than 180° and none of the edges cross each other. We represent a convex polygon as an array V with n elements, where each element represents a vertex of the polygon in the form of a coordinate pair (x, y) . We are told that $V[1]$ is the vertex with the least x coordinate and that the vertices $V[1], V[2], \dots, V[n]$ are ordered counter-clockwise. Assuming that the x coordinates (and the y coordinates) of the vertices are all distinct, do the following.

Give a divide and conquer algorithm to find the vertex with the largest x coordinate in $O(\log n)$ time.

Solution:

Since $V[1]$ is known to be the vertex with the minimum x -coordinate (leftmost point), moving counter-clockwise to complete a cycle must first increase the x -coordinates and then after reaching a maximum (rightmost point), should decrease the x -coordinate back to that of $V[1]$. To see this more formally, we claim that there cannot be two distinct local maxima in the x -axis projection of the counter-clockwise tour. For the sake of contradiction, assume otherwise. Then there exists a vertical line that would intersect the boundary of the polygon at more than two points. This is impossible by convexity of the polygon since the line segments between the intersection points must lie completely in the interior of the polygon, thus contradicting our assumption. Thus, $V_x[1 : n]$ is a unimodal array, and the first part of this question is synonymous with detecting the location of the maximum element in this array.

Consider the following algorithm:

- (a) If $n = 1$, return $V_x[1]$.
- (b) If $n = 2$, return $\max\{V_x[1], V_x[2]\}$.
- (c) $k = \lceil \frac{n}{2} \rceil$.
- (d) If $V_x[k] > V_x[k - 1]$ and $V_x[k] > V_x[k + 1]$, then return $V_x[k]$.
- (e) If $V_x[k] < V_x[k - 1]$ then call the algorithm recursively on $V_x[1 : k - 1]$, else call the algorithm recursively on $V_x[k + 1 : n]$.

Complexity: If $T(n)$ is the running time on an input of size n , then beside a constant number of comparisons, the algorithm is called recursively on at most one of $V_x[1 : k - 1]$ (size = $k - 1 = \lceil \frac{n}{2} \rceil - 1 \leq \lfloor \frac{n}{2} \rfloor$) or $V_x[k + 1 : n]$ (size = $n - k = n - \lceil \frac{n}{2} \rceil = \lfloor \frac{n}{2} \rfloor$). Therefore, $T(n) \leq T(\lfloor \frac{n}{2} \rfloor) + \theta(1)$.

Assuming n to be a power of 2, this recurrence simplifies to $T(n) \leq T(n/2) + \theta(1)$ and invoking Master's Theorem gives $T(n) = O(\log n)$.

1) 10 pts

Mark the following statements as **TRUE** or **FALSE**. No need to provide any justification.

True [TRUE/FALSE]

Binary search could be called a divide and conquer technique

False [TRUE/FALSE]

If you have non integer edge capacities, then you cannot have an integer max flow

True [TRUE/FALSE]

The Ford Fulkerson algorithm with real valued capacities can run forever

True [TRUE/FALSE]

If we have a 0-1 valued s-t flow in a graph of value f, then we have f edge disjoint s-t paths in the graph

True [TRUE/FALSE]

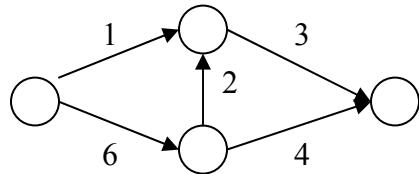
Merge sort works because at each level of the algorithm, the merge step assumes that the two lists are sorted

2) 20pts

Prove or disprove the following for a given graph $G(V,E)$ with integer edge capacities C_i

- a. If the capacities on each edge are unique then there is a unique min cut

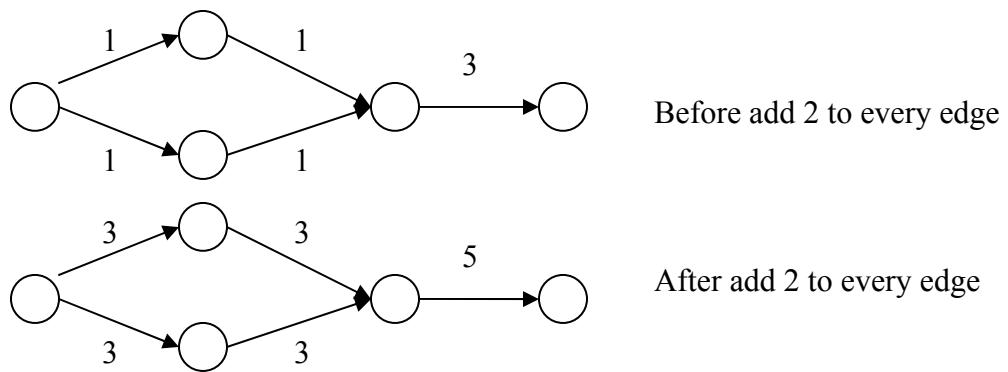
Disprove :



- b. If we multiply each edge with the same multiple “f”, the max flow also gets multiplied by the same factor

Prove: For each cut (A, B) of the graph G , the capacity $c(A, B)$ will be multiplied by “f” if each edge’s capacity is multiplied by “f”, thus the minimal cut will be multiplied by “f”, thus the max flow also gets multiplied by “f”.

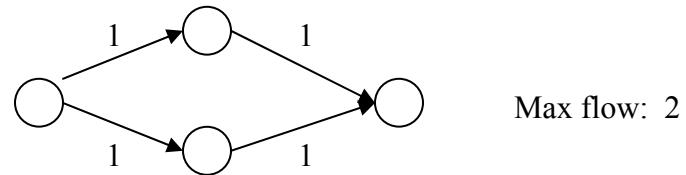
- c. If we add the same amount, say “a” to every edge in the graph, then the min cut stays the same (the edges in the min cut stay the same i.e.)



- d. If the edge costs are all even, then the max flow(min cut) is also even

The capacity of any cut (A, B) is sum of the capacities of all edges out of A, thus the capacity of any cut, including the minimal one, is also even.

- e. If the edge costs are all odd, then the max flow (min cut) is also odd



3) 20 pts

Suppose you are given k sorted arrays, each with n elements, and you want to combine them into a single sorted array of kn elements.

(a) Here's one strategy: merge the first two arrays, then merge in the third, then merge in the fourth, and so on. What is the time complexity of this strategy, in terms of k and n ?

The merge of the first two arrays takes $O(n)$, ... the merge of the last array takes $O(kn)$, totally there are k merge sorts. Thus, it takes $O(k^2n)$ times.

(b) Present a more efficient solution to this problem.

Let the final array to be A , initially A is empty.

Build up a binary heap with the first element from the k given arrays, thus this binary tree consists of k elements.

Extract the minimal element from the binary tree and add it to A , delete it from its original array, and insert the next element from that array into the binary heap.

Each heap operation is $O(\log k)$, and take $O(kn)$ operations, thus, the running time is $O(kn \log k)$

4) 10 pts

Derive a recurrence equation that describes the following code. Then, solve the recurrence equation.

```
COMPOSE (n)
1. for  $i \leftarrow 1$  to  $n$ 
2.       do for  $j \leftarrow 1$  to  $\sqrt{n}$ 
3.             do print( $i, j, n$ )
4. if  $n > 0$ 
5.       then for  $i \leftarrow 1$  to 5
6.             COMPOSE ( $\lfloor n/2 \rfloor$ )
```

$$T(n) = 5 T(n/2) + O(n^{3/2})$$

By the Master theorem, $T(n) = n^{\lg 5}$

5) 20 pts

Let $G=(V,E)$ be a directed graph, with source $s \in V$, sink $t \in V$, and non-negative edge capacities $\{C_e\}$. Give a polynomial time algorithm to decide whether G has a unique minimum s - t cut. (i.e. an s - t cut of capacity strictly less than that of all other s - t cuts.)

Find a max flow f for this graph G , and then construct the residual graph G' based on this max flow f . Start from the source s , perform breadth-first or depth-first search to find the set A that s can reach, define $B = V - A$, the cut (A, B) is a minimum s - t cut. Then, for each node v in B that is connected to A in the original graph G , try the following codes: add v into set A , and then perform breadth-first or depth-first search find a new set A' that s can reach, if the sink t is included in A' , then try next node v' , otherwise, report a new minimum s - t cut. If all possible nodes v in B have been tried but no more minimum s - t cut can be found, then report the (A, B) is the unique minimum s - t cut.

20 pts

Consider you have three courses to study, C1, C2 and C3. For the three courses you need to study a minimum of T1, T2, and T3 hours respectively. You have three days to study for these courses, D1, D2 and D3. On each day you have a maximum of H1, H2, and H3 hours to study respectively. You have only 12 hours total to give to all of these courses, which you could distribute within the three days. On each one of the days, you could potentially study all three courses. Give an algorithm to find the distribution of hours to the three courses on the three days

If $T_1+T_2+T_3 > 12$ or $T_1+T_2+T_3 > H_1 + H_2 + H_3$, then report no feasible distribution can be found.

Else, start C1 with T1 hours, and then C2 with T2 hours, and finally C3 with T3 hours.

1) 15 pts

- a) Suppose that we have a divide-and-conquer algorithm for a solving computational problem that on an input of size n , divides the problem into two independent subproblems of input size $2n/5$ each, solves the two subproblems recursively, and combines the solutions to the subproblems. Suppose that the time for dividing and combining is $O(n)$. What's the running time of this algorithm? Answer the question by giving a recurrence relation for the running time $T(n)$ of the algorithm on inputs of size n , and giving a solution to the recurrence relation.

Solution:

The recurrence relation of $T(n)$:

$$T(n) = 2 T(2n/5) + O(n)$$

To solve the recurrence relation, using the substitution method:

guess that $T(n) \leq cn$

$$\Rightarrow T(n) \leq 2 * 2c/5n + an \leq cn \text{ as long as } c \geq 5a$$

Thus, $T(n) \leq cn \Rightarrow T(n) = O(n)$

b) Characterize each of the following recurrence equations using the master method. You may assume that there exist constants $c > 0$ and $d \geq 1$ such that for all $n < d$, $T(n) = c$.

- a. $T(n) = 2T(n/2) + \log n$
- b. $T(n) = 16T(n/2) + (n \log n)^4$
- c. $T(n) = 9T(n/3) + n^3 \log n$

Solution:

a. Since there is $\epsilon > 0$ such that $\log n = O(n^{\log_2 2-\epsilon}) = O(n^{1-\epsilon})$,
 \Rightarrow case 1 of master method, $T(n) = \Theta(n^{\log_2 2}) = \Theta(n)$

b. $(n \log n)^4 = n^4 \log^4 n = \Theta(n^{\log_2 16} \log^4 n)$,
 \Rightarrow case 2 of master method, $T(n) = \Theta(n^{\log_2 16} \log^{(4+1)} n) = \Theta(n^4 \log^5 n)$

c. $n^3 \log n = \Omega(n^{\log_3 9 + 1})$ and $9 \times \left(\frac{n}{3}\right)^3 \log \frac{n}{3} = \frac{n^3}{3} \log \frac{n}{3} \leq \frac{1}{3} n^3 \log n$,
 \Rightarrow case 3 of master method, $T(n) = \Theta(n^3 \log n)$

2) 15 pts

You are given a sorted array $A[1..n]$ of n distinct integers. Provide an algorithm that finds an index i such that $A[i] = i$, if such exists. Analyze the running time of your algorithm

Solution: (This one is exactly the same as 5) of sample exam 1 of exam I)

Function(A, n)

```
{  
    i=floor(n/2)  
    if A[i]==i  
        return TRUE  
    if (n==1)&&(A[i]!=i)  
        return FALSE  
    if A[i]<i  
        return Function(A[i+1:n], n-i)  
    if A[i>i]  
        return Function(A[1:i], i)  
}
```

Proof:

The algorithm is based on Divide and Conquer. Every time we break the array into two halves. If the middle element i satisfy $A[i] < i$, we can see that for all $j < i$, $A[j] < j$. This is because A is a sorted array of DISTINCT integers. To see this we note that

$A[j+1]-A[j] \geq 1$ for all j . Thus in the next round of search we only need to focus on $A[i+1:n]$

Likewise, if $A[i] > i$ we only need to search $A[1:i]$ in the next round.

For complexity $T(n)=T(n/2)+O(1)$

Thus $T(n)=O(\log n)$

3) 15 pts

Consider a sequence of n distinct integers. Design and analyze a dynamic programming algorithm to find the length of a longest increasing subsequence. For example, consider the sequence:

45 23 9 3 99 108 76 12 77 16 18 4

A longest increasing subsequence is 3 12 16 18, having length 4.

Solution:

Let X be the sequence of n distinct integers.

Denote by $X(i)$ the i th integer in X , and by D_i the length of the longest increasing subsequence of X that ends with $X(i)$.

The recurrence that relates D_i to D_j 's with $j < i$ is as follows:

$$D_i = \max_{j < i, X(j) < X(i)} (D_j + 1)$$

The algorithm is as follows:

for $i = 1 \dots n$

 Compute D_i

end for

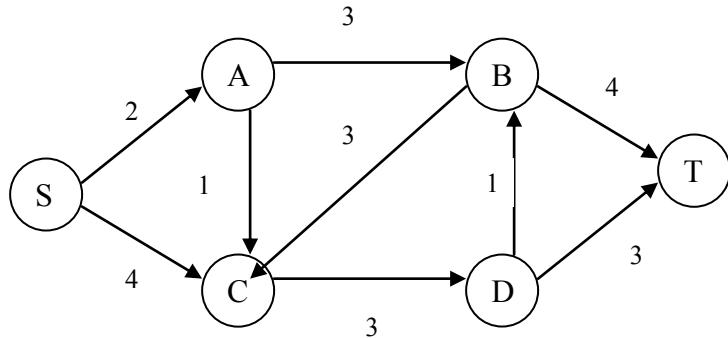
return the largest number among D_1 to D_n .

When computing each D_i , the recurrence finds the largest D_j such that $j < i, X(j) < X(i)$. Thus, each D_i is maximized. The length of the longest increasing subsequence is obviously among D_1 to D_n .

Since computing each D_i costs $O(n)$ and the loop runs for n times, the complexity of the algorithm is $O(n^2)$.

4) 20 pts

In the flow network illustrated below, each directed edge is labeled with its capacity. We are using the Ford-Fulkerson algorithm to find the maximum flow. The first augmenting path is S-A-C-D-T, and the second augmenting path is S-A-B-C-D-T.

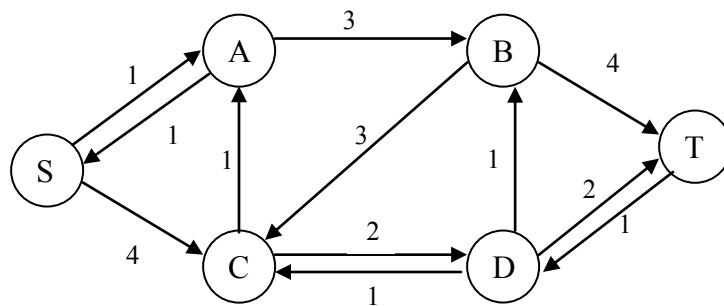


a) 10 pts

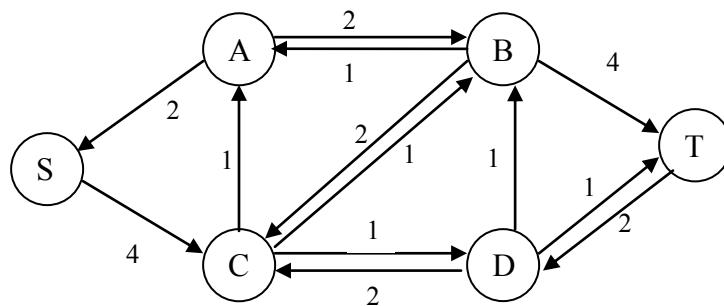
Draw the residual network after we have updated the flow using these two augmenting paths(in the order given)

Solution:

Residual network after S-A-C-D-T:



Residual network after S-A-B-C-D-T:



b) 6pts

List all of the augmenting paths that could be chosen for the third augmentation step.

Solution:

S-C-B-T

S-C-D-T

S-C-A-B-T

S-C-D-B-T

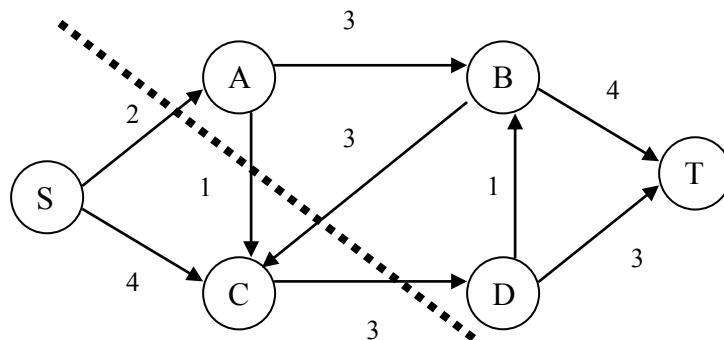
c) 4pts

What is the numerical value of the maximum flow? Draw a dotted line through the original graph to represent a minimum cut.

Solution:

The numerical value of the maximum flow is 5.

A minimum cut is shown in the following figure:



5) 20 pts

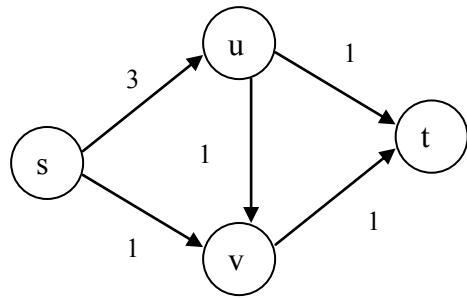
Decide whether you think the following statements are true or false. If true, give a short explanation. If false, give a counterexample.

Let G be an arbitrary flow network, with a source s , a sink t , and a positive integer capacity c_e on every edge e .

a) If f is a maximum s - t flow in G , then for all edges e out of s , we have $f(e) = c_e$.

Solution:

False.



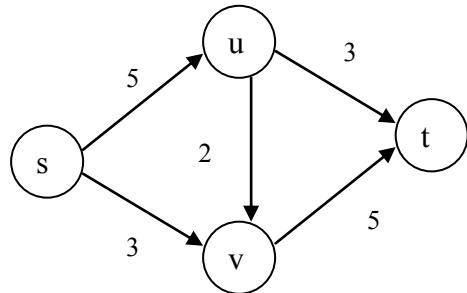
Clearly, the maximum s - t flow in the above graph is 2.

The edge (s, u) does not have $f(e) = c_e$

b) Let (A, B) be a minimum s - t cut with respect to the capacities $\{c_e : e \in E\}$. Now suppose we add 1 to every capacity. Then (A, B) is still a minimum s - t cut with respect to these new capacities $\{1 + c_e : e \in E\}$.

Solution:

False.



Clearly, one of the minimum cuts is $A = \{s, u\}$ and $B = \{v, t\}$. After adding 1 to every capacity, the maximum s - t flow becomes 10 and the cut between A and B is 11.

6) 15 pts

Suppose that in county X, there are only 3 kinds of coins: the first kind are 1-unit coins, the second kind are 4-unit coins, and the third kind are 6-unit coins. You want to determine how to return an amount of K units, where $K \geq 0$ is an integer, using as few coins as possible. For example, if $K=7$, you can use 2 coins (a 1-unit coin and a 6-unit coin), which is better than using three 1-unit coins and a 4-unit coins since in this case, the total number of coins used is 4.

For $0 \leq k \leq K$ and $1 \leq i \leq 3$, let $c(i, k)$ be the smallest number of coins required to return an amount of k using only the first i kinds of coins. So for example, $c(2, 5)$ would be the smallest number of coins required to return 5 units if we can only use 1-unit coins and 4-unit coins. In what follows, you can assume that the denomination of the coins is stored in an array d , i.e., $d[1]=1, d[2]=4, d[3]=6$.

Give a dynamic programming algorithm that returns $c(i, k)$ for $0 \leq k \leq K$ and $1 \leq i \leq 3$

Solution:

Let the coin denominations be d_1, d_2, \dots, d_i .

Because of the optimal substructure, if we knew that an optimal solution for the problem of making change for k cents used a coin of denomination d_j , we would have $c(i, k) = 1 + c(i, k - d_j)$.

As base cases, we have that $c(i, k) = 0$ for all $k \leq 0$.

To develop a recursive formulation, we have to check all denominations, giving

$$c(i, k) = \begin{cases} 0, & \text{if } k \leq 0 \\ 1 + \min_{1 \leq j \leq i} \{c(i, k - d_j)\}, & \text{if } k > 1 \end{cases}$$

The algorithm is as follows:

```
for i = 1...3
    for k = 0...K
        Compute c(i, k)
    end for
end for
```

When $i = j$, to compute each $c(j, k)$ costs $O(j)$. Thus, the complexity is $O(K+2K+3K) = O(6K)$

Q1:

[TRUE/FALSE] **FALSE**

Suppose $f(n) = f\left(\frac{n}{2}\right) + 56$, then $f(n) = \Theta(n)$

[TRUE/FALSE] **TRUE**

Maximum value of an s-t flow could be less than the capacity of a given s-t cut in a flow network.

[TRUE/FALSE] **FALSE**

For edge any edge e that is part of the minimum cut in G, if we increase the capacity of that edge by any integer $k > 1$, then that edge will no longer be part of the minimum cut.

[TRUE/FALSE] **FALSE**

Any problem that can be solved using dynamic programming has a polynomial worst case time complexity with respect to its input size.

[TRUE/FALSE] **FALSE**

In a dynamic programming solution, the space requirement is always at least as big as the number of unique sub problems

[TRUE/FALSE] **FALSE**

In the divide and conquer algorithm to compute the closest pair among a given set of points on the plane, if the sorted order of the points on both X and Y axis are given as an added input, then the running time of the algorithm improves to $O(n)$.

[TRUE/FALSE] **TRUE**

If f is a max s-t flow of a flow network G with source s and sink t , then the value of the min s-t cut in the residual graph G_f is 0.

[TRUE/FALSE] **TRUE**

Bellman-Ford algorithm can solve the shortest path problem in graphs with negative cost edges in polynomial time.

[TRUE/FALSE] **TRUE**

Given a directed graph $G=(V,E)$ and the edge costs, if every edge has a cost of either 1 or -1 then we can determine if it has a negative cost cycle in $O(|V|^3)$ time..

[TRUE/FALSE] **TRUE**

The space efficient version of the solution to the sequence alignment problem (discussed in class), was a divide and conquer based solution where the divide step was performed using dynamic programming.

Q2:

The problem can be formulated as a dynamic programming problem in many different ways
(These are the correct solutions to the three most common formulations used by students in the exam):

- 1) $OPT[v]$ will denote the minimum number of coins required to make change for value “ v ”.

The recursive formula would be:

$$OPT[T] = \min_{1 \leq i \leq n} \{OPT[T - X_i] + 1\}$$

The boundary values will be as follows:

$$OPT[0] = 0$$

$$OPT[v] = \infty \text{ if } v < 0$$

We fill in the $OPT[]$ array in the following order:

```
for (int i = 1; i <= v; i++) {
```

```
    int min = infinity;
```

```
    for (int x : {X1, X2, ..., Xn}) {
```

```
        if (OPT[i - x] + 1 > min) min = OPT[i - x] + 1;
```

```
    }
```

```
    OPT[i] = min;
```

```
}
```

At the end if $OPT[v] \leq k$ we return success, otherwise we return failure.

- 2) $OPT[k, v]$ denotes the minimum number of coins required to make change for v using at most k coins: Then:

$$OPT[k, v] = \min \left\{ OPT[k - 1, v], \min_{1 \leq i \leq n} \{OPT[k - 1, v - X_i] + 1\} \right\}$$

Here the boundary values will be:

$$OPT[k, 0] = 0$$

$$OPT[0, v] = \infty \text{ if } v \neq 0$$

$$OPT[k, v] = \infty \text{ if } v < 0$$

We should fill the $OPT[]$ array in the following order:

```
for (int i = 1; i <= v; i++) {
```

```
    for (int j = 1; j <= k; j++) {
```

```
        int min = OPT[j-1, i];
```

```
        for (int x : {X1, X2, ..., Xn}) {
```

```
            if (OPT[j-1, i - x] + 1 > min) min = OPT[j-1, i - x] + 1;
```

```
        }
```

```
        OPT[j, i] = min;
```

```
    }
```

```
}
```

Like the previous formulation, if $OPT[k, v] \leq k$ we return success, otherwise we return failure.

- 3) $OPT[k, v]$ is a binary value denoting whether it's possible to make change for v using at most k coins. Then:

$$OPT[k, v] = OPT[k - 1, v] \vee \left\{ \bigvee_{1 \leq i \leq n} OPT[k - 1, v - X_i] \right\}$$

The boundary values will be:

$$\begin{aligned} OPT[0, 0] &= true \\ OPT[0, v] &= false \\ OPT[k, v] &= false \text{ if } v < 0 \end{aligned}$$

We should fill the $OPT[]$ array in the following order:

```
for (int i = 1; i <= v; i++) {
    for (int j = 1; j <= k; j++) {
        OPT[i, j] = OPT[j-1, i];
        for (int x : {X1, X2, ..., Xn}) {
            if (OPT[j-1, i - x]) OPT[j, i] = true;
        }
    }
}
```

At the end, we only need to return $OPT[k, v]$ as the result.

Q3:

This problem can be mapped to a network flow problem. First assign a node s_i for each student and node n_i for each night. We also add two nodes S and T . Now we need to assign the edges and capacities. We create an edge between S and each student node with capacity 1 . We also create an edge between each night node and T with capacity 1. Then we need to connect student node s_i with the night node n_j if s_i is capable to cook on night n_i . The capacity for this edge is also 1. Now we only need to run ford-Fulkerson algorithm to find the maximum flow and see if this flow is equal to n or not.

Q4:

Part a) We present a few proofs of the claim. The first two are perhaps the most straightforward but the third leads naturally to an algorithm for part b.

Proof 1: Since A is a finite array, it has a minimum. Let j denote an index where A is minimum. If j is not in the boundary (that is j is neither 1 nor n), then j is a local minimum as well. If j is 1, then since $A[1] \geq A[2]$, $A[2]$ is the minimum values in the array and 2 is a local minimum. Likewise, if j is n, then since $A[n] \geq A[n-1]$, $A[n-1]$ is the minimum value in A and thus n-1 is a local minimum. Thus in every case, we may conclude that there is a local minimum.

Proof 2: Assume that A does not have a local minimum. Since $A[1] \geq A[2]$, this implies that $A[2] > A[3]$ (otherwise, 2 would be a local minimum). Likewise $A[2] > A[3]$ implies that $A[3] > A[4]$ and so on. In particular $A[n-1] > A[n]$, contradicting the fact that $A[n] \geq A[n-1]$. Thus our assumption is incorrect.

There is also an analogous proof that goes from right to left.

Proof 3. We prove the claim by induction. If $n=3$, then 2 is a local minimum. Consider $A[1\dots n]$ with $n>3$, $A[1] \geq A[2]$ and $A[n] \geq A[n-1]$. As the induction hypothesis, assume that all arrays $B[1\dots k]$ with $2 < k < n$, $B[1] \geq B[2]$ and $B[k] \geq B[k-1]$ have a local minimum. Let $j = \text{floor}(n/2)$. If $A[j] \leq A[j+1]$, then $A[1\dots j+1]$ has a local minimum (by the induction hypothesis) and hence $A[1\dots n]$ has a local minimum. Else (that is, $A[j] > A[j+1]$), then $A[j\dots n]$ has a local minimum (by the induction hypothesis) and hence $A[1\dots n]$ has a local minimum.

Part b) A divide and conquer solution for part b follows immediately from the third proof for part a. If $n=3$, then 2 is a local minimum. If $n>3$, set $j=\text{floor}(n/2)$. If $A[j] \leq A[j+1]$, then search for a local minimum in $A[1\dots j+1]$. Else, search for a local minimum in $A[j\dots n]$. Let $T(n)$ denote the number of pairwise comparisons performed by our recursive algorithm for finding a local minimum in $A[1\dots n]$. Then $T(n) \leq T(\text{ceil}(n/2)) + 1$ which implies that $T(n) = O(\log(n))$.

Q5:

- (a) The number of unique ways are shown as follows:

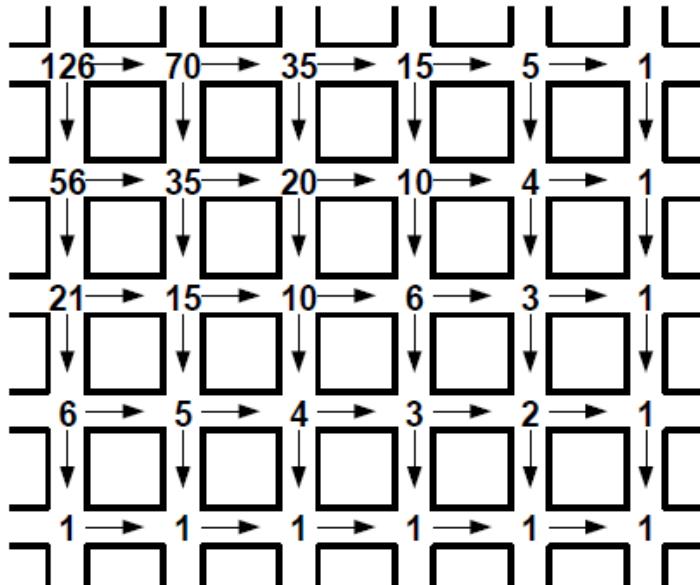


Figure A.

Answer: 126

- (b) Define $\text{OPT}(i,j)$ as the number of unique ways from intersection (i,j) to E: (5,4). The recursive relation is :

$$\text{OPT}(i,j) = \text{OPT}(i+1, j) + \text{OPT}(i,j+1), \text{ for } i < 5, \text{ and } j < 4$$

Boundary condition: $\text{OPT}(5,j) = 1$; $\text{OPT}(i,4) = 1$

Alternative solution:

If you define $\text{OPT}(i,j)$ as the number of unique ways from intersection S: (0,0) to intersection (i,j) , you can also get the correct answer, but the recursive relation and boundary conditions should correspondingly changes.

- (c) With dead ends, the numerical results of the number of unique ways are shown as follows:

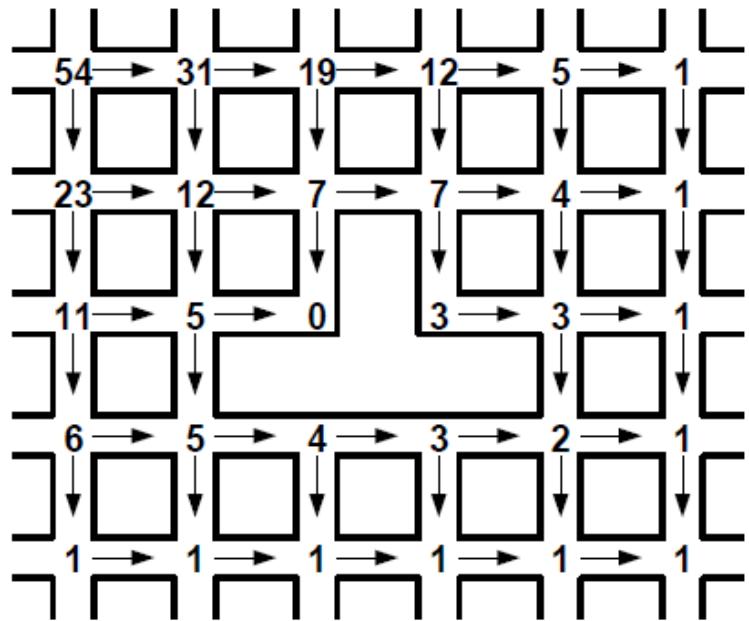


Figure B.

Answer: 54

Q6:

Algorithm:

1. Add node t , and add edges to t from each node in S .
2. Set the capacity of each of these new edges as infinity.
3. Set spammer node q as the source node. Then the communication network G becomes a larger network G' with source q and sink t .
4. Find the min q - t cut on G' by running a polynomial time max-flow algorithm.
5. Install a spam filter on each edge in the min-cut's cut-set.

Complexity:

Since the construction of the new edges takes $O(|S|)$ times, together with the polynomial time of the min-cut algorithm, the entire algorithm takes polynomial time.

Justification:

We're simply trying to separate q from S , but min-cut only works when S is a single node. We fix this by creating t directly connected from S , but we want to make sure the min-cut's cut-set doesn't include any edge connecting t , which is why we put the capacities arbitrarily large on these edges. Min-cut would then find the min-cost way to separate q from S .

Q5:

- (a) The number of unique ways are shown as follows:

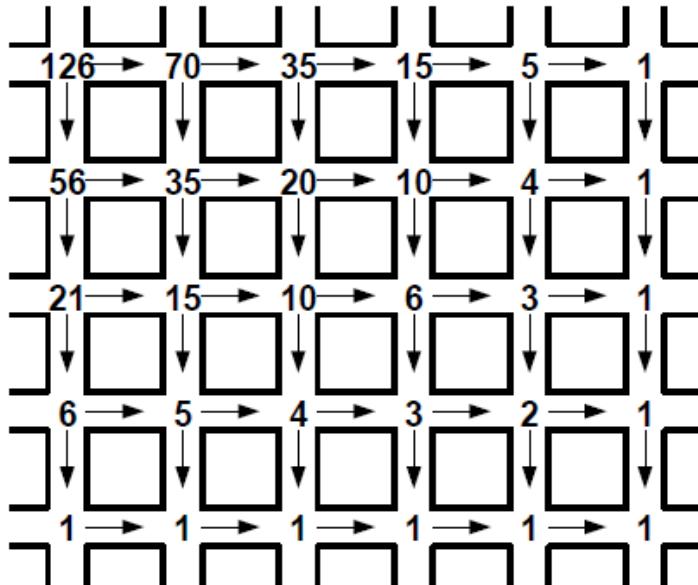


Figure A.

Answer: 126

- (b) Define $\text{OPT}(i,j)$ as the number of unique ways from intersection (i,j) to E: (5,4). The recursive relation is :

$$\text{OPT}(i,j) = \text{OPT}(i+1, j) + \text{OPT}(i,j+1), \text{ for } i < 5, \text{ and } j < 4$$

Boundary condition: $\text{OPT}(5,j) = 1$; $\text{OPT}(i,4) = 1$

Alternative solution:

If you define $\text{OPT}(i,j)$ as the number of unique ways from intersection S: (0,0) to intersection (i,j) , you can also get the correct answer, but the recursive relation and boundary conditions should correspondingly changes.

- (c) With dead ends, the numerical results of the number of unique ways are shown as follows:

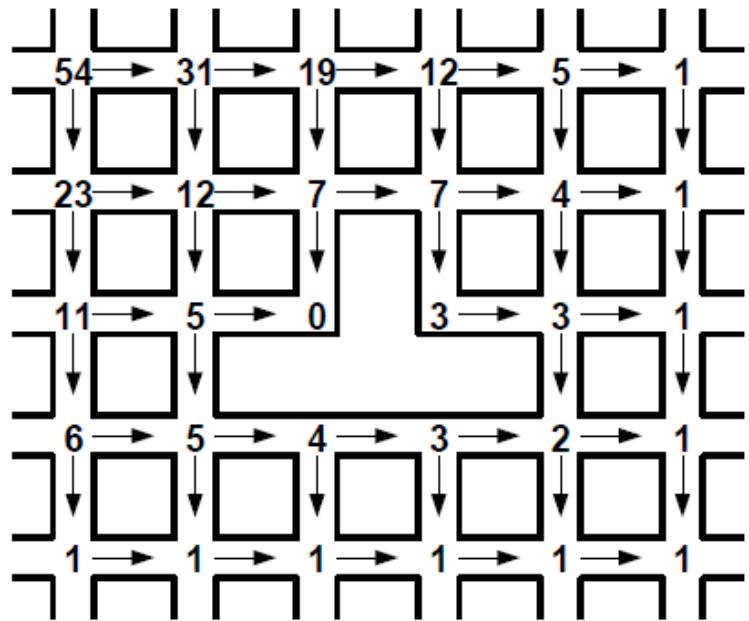


Figure B.

Answer: 54

Q6:

Algorithm:

1. Add node t , and add edges to t from each node in S .
2. Set the capacity of each of these new edges as infinity.
3. Set spammer node q as the source node. Then the communication network G becomes a larger network G' with source q and sink t .
4. Find the min q - t cut on G' by running a polynomial time max-flow algorithm.
5. Install a spam filter on each edge in the min-cut's cut-set.

Complexity:

Since the construction of the new edges takes $O(|S|)$ times, together with the polynomial time of the min-cut algorithm, the entire algorithm takes polynomial time.

Justification:

We're simply trying to separate q from S , but min-cut only works when S is a single node. We fix this by creating t directly connected from S , but we want to make sure the min-cut's cut-set doesn't include any edge connecting t , which is why we put the capacities arbitrarily large on these edges. Min-cut would then find the min-cost way to separate q from S .

1) 20 pts

Mark the following statements as **TRUE** or **FALSE**. No need to provide any justification.

[**TRUE**]

If $NP = P$, then all problems in NP are NP hard

[**FALSE**]

L_1 can be reduced to L_2 in Polynomial time and L_2 is in NP, then L_1 is in NP

[**FALSE**]

The simplex method solves Linear Programming in polynomial time.

[**FALSE**]

Integer Programming is in P.

[**FALSE**]

If a linear time algorithm is found for the traveling salesman problem, then every problem in NP can be solved in linear time.

[**TRUE**]

If there exists a polynomial time 5-approximation algorithm for the general traveling salesman problem then 3-SAT can be solved in polynomial time.

[**FALSE**]

Consider an undirected graph $G=(V, E)$. Suppose all edge weights are different. Then the longest edge cannot be in the minimum spanning tree.

[**FALSE**]

Given a set of demands $D = \{dv\}$ on a directed graph $G(V, E)$, if the total demand over V is zero, then G has a feasible circulation with respect to D .

[**TRUE**]

For a connected graph G , the BFS tree, DFS tree, and MST all have the same number of edges.

[**FALSE**]

Dynamic programming sub-problems can overlap but divide and conquer sub-problems do not overlap, therefore these techniques cannot be combined in a single algorithm.

Grading Criteria: Pretty clear, each has two point. These T/F are designed and answered by Professor Shamsian

2) 10 pts

Demonstrate that an algorithm that consists of a polynomial number of calls to a polynomial time subroutine could run in exponential time.

Suggested Solution:

Suppose X takes an input size of n and returns an output size of n^2 . You call X a polynomial number of times say n. If the size of the original input is n the size of the output will be n^{2n} which is exponential WRT n.

Grading Criteria: Rephrasing the question doesn't get that much. If you show you at least understood polynomial / exponential time, you get some credit.

3) 10 pts

Suppose that we are given a weighted, directed graph $G = (V, E)$ in which edges that leave the source vertex s may have negative weights, all other edge weights are nonnegative, and there are no negative-weight cycles. Argue that Dijkstra's algorithm correctly finds shortest paths from s in this graph.

Suggested solution :

```
DIJKSTRA( $G, w, s$ )
1 INITIALIZE-SINGLE-SOURCE( $G, s$ )
2  $S \leftarrow \emptyset$ 
3  $Q \leftarrow V[G]$ 
4 while  $Q \neq \emptyset$ 
5   do  $u \leftarrow \text{EXTRACT-MIN}(Q)$ 
6    $S \leftarrow S \cup \{u\}$ 
7   for each vertex  $v \in \text{Adj}[u]$ 
8     do RELAX( $u, v, w$ )
  
RELAX( $u, v, w$ )
1 if  $d[v] > d[u] + w(u, v)$ 
2 then  $d[v] \leftarrow d[u] + w(u, v)$ 
3  $\pi[v] \leftarrow u$ 
```

We have the algorithm as shown above.

We show that in each iteration of Dijkstra's algorithm, $d[u] = \delta(s, u)$ when u is added to S (the rest follows from the upper-bound property). Let $N^-(s)$ be the set of vertices leaving s , which have negative weights. We divide the proof to vertices in $N^-(s)$ and all the rest. Since there are no negative loops, the shortest path between s and all $u \in N^-(s)$, is through the edge connecting them to s , hence $\delta(s, u) = w(s, u)$, and it follows that after the first time the loop in lines 7,8 is executed $d[u] = w(s, u) = \delta(s, u)$ for all $u \in N^-(s)$ and by the upper bound property $d[u] = \delta(s, u)$ when u is added to S . Moreover it follows that $S = N^-(s)$ after $|N^-(s)|$ steps of the while loop in lines 4-8.

For the rest of the vertices we argue that the proof of Theorem 24-6 (*Theorem 24.6 - Correctness of Dijkstra's algorithm, Introduction to Algorithem by Cormen*) holds since every

shortest path from s includes at most one negative edge (and if does it has to be the first one). To see why this is true assume otherwise, which mean that the path contains a loop, contradicting the property that shortest paths do not contain loops.

Grading Criteria : You need to argue, so bringing just one example shouldn't get you the whole credit , but it may did! If you showed you at least understood Dijkstra, you got some credit too.

4) 20 pts

Phantasy Airlines operates a cargo plane that can hold up to 30000 pounds of cargo occupying up to 20000 cubic feet. We have contracted to transport the following items.

<u>Item type</u>	<u>Weight</u>	<u>Volume</u>	<u>Number</u>	<u>Cost if not carried</u>
1	4000	1000	3	\$800
2	800	1200	10	\$150
3	2000	2200	4	\$300
4	1500	500	5	\$500

For example, we have contracted 10 items of type 2, each of which weighs 800 pounds and takes up to 1200 cubic feet of space. The last column refers to the cost of subcontracting shipment to another carrier.

For each pound we carry, the cost of flying the plane increases by 5 cents. Which items should we put in the plane, and which should we ship via other carrier, in order to have the lowest shipping cost? Formulate this problem as an integer programming problem.

Suggested Solution:

Assume the data in the table are per item.

Considering x_1, x_2, x_3, x_4 are the items that we will ship for types 1,2,3 and 4 respectively.

It is clear that :

$$0 \leq x_1 \leq 3 \quad \& \quad 0 \leq x_2 \leq 10 \quad \& \quad 0 \leq x_3 \leq 4 \quad \& \quad 0 \leq x_4 \leq 5$$

Weight constraint :

$$x_1 * 4000 + x_2 * 800 + x_3 * 2000 + x_4 * 1500 \leq 30,000$$

Volume constraint:

$$x_1 * 1000 + x_2 * 1200 + x_3 * 2200 + x_4 * 500 \leq 20,000$$

Cost of shipping :

$$C_{Ship} = C_A + (x_1 * 4000 + x_2 * 800 + x_3 * 2000 + x_4 * 1500) * \$0.05$$

Where C_A is the cost of operating empty cargo plane

Cost of sub-contracting

$$C_{Sub} = (3 - x_1) * 800 + (10 - x_2) * 150 + (4 - x_3) * 300 + (5 - x_4) * 500$$

Our objective is to minimize $C_{Ship} + C_{Sub}$

Grading Criteria: Some credit will be deducted if you missed some optimization part. Making this simple question complicated may result deduction.

5) 20 pts

Suppose you are given a set of n integers each in the range $0 \dots K$. Give an efficient algorithm to partition these integers into two subsets such that the difference $|S_1 - S_2|$ is minimized, where S_1 and S_2 denote the sums of the elements in each of the two subsets.

Proposed Solution :

$a[1] \dots a[m]$ = the set of integers

$\text{opt}[i,k]$ = true if and only if it is possible to sum to k using elements $1 \dots i$

```
Initialize opt(i,k) = false for all i,k
for(i=1..n) {
    for(k=1..K) {
        if(opt(i,k) == true) {
            for(j=1..m) {
                opt(i,k + a[j]) = true;
            }
        }
    }
}
for(k = floor(K/2)..1) {
    if(opt(n, k)) {
        Return |K - 2k|; // Returns the difference. Partition can be found by back tracking
    }
}
```

6) 10 pts

Adrian has many, many love interests. Many, many, many love interests. The problem is, however, a lot of these individuals know each other, and the last thing our polyamorous TA wants is fighting between his hookups. So our resourceful TA draws a graph of all of his potential partners and draws an edge between them if they know each other. He wants at least k romantic involvements and none of them must know each other (have an edge between them). Can he create his LOVE-SET in polynomial time? Prove your answer.

Proposed Solution :

This is just Independent Set. Proof of NP-completeness was shown in class lecture. We show the correctness as follows: 1) Any Independent Set of size k on the graph will satisfy the requirement that no two love interests know each other (share an edge). 2) If there is a set of k love interests none of which know each other, then there must be a corresponding set of k vertices, such that no two share an edge (know each other).

7) 10 pts

An edge in a flow network is called a bottleneck if increasing its capacity increases the max flow in the network. Give an efficient algorithm for finding all the bottlenecks in the network.

Proposed Solution

Find max flow and the corresponding residual graph, then for each edge increase its capacity by one unit and see if you find a new path from s to t in the residual graph. (Of course you undo this increase before trying the next edge.) If the flow increases for any such edge then that edge must be a bottleneck.

1) 20 pts

Mark the following statements as **TRUE** or **FALSE**. No need to provide any justification.

[/FALSE]

All integer programming problems can be solved in polynomial time.

[TRUE]

Fractional knapsack problem has a polynomial time greedy solution.

[/FALSE]

For any cycle in a graph, the cheapest edge in the cycle is in a minimum spanning tree.

[/FALSE]

Every decision problem is in NP.

[TRUE/]

If P=NP then P=NP=NP-complete.

[/FALSE]

Ford-Fulkerson algorithm can always terminate if the capacities are real numbers.

[/FALSE]

A flow network with unique edge capacities has a unique min cut.

[/FALSE]

A graph with non-unique edge weights will have at least two minimum spanning trees.

[TRUE/]

A sequence of $O(n)$ priority queue operations can be used to sort a set of n numbers.

[/FALSE]

If a problem X is polynomial time reducible to an NP-complete problem Y, then X is NP-complete.

2) 16 pts

Consider the **complete** weighted graph $G = (V, E)$ with the following properties

- a. The vertices are points on the X-Y plane on a regular $n \times n$ grid, i.e. the set of vertices is given by $V = \{(p, q) | 1 \leq p, q \leq n\}$, where p and q are integer numbers.
- b. The edge weights are given by the usual Euclidean distance, i.e. the weight of the edge between the nodes (i, j) and (k, l) is $\sqrt{(k - i)^2 + (l - j)^2}$.

Prove or disprove: There exists a minimum spanning tree of G such that every node has degree at most two.

Solution:

There does exist an MST of G with every node having degree ≤ 2 . One such MST is obtained by the edges joining **nodes** in the following order: $(1,1) \rightarrow (1,2) \rightarrow (1,3) \rightarrow \dots \rightarrow (1,n-1) \rightarrow (1,n) \rightarrow (2,n) \rightarrow (2,n-1) \rightarrow \dots \rightarrow (2,2) \rightarrow (2,1) \rightarrow (3,1) \rightarrow \dots \rightarrow (n,n)$.

Let us denote the above set of edges by E . To prove that E indeed forms an MST, we need to show that it forms a spanning tree and that it is of minimal weight.

E forms a spanning tree: It is clear that all nodes in the above sequence are distinct. This implies that there are no cycles since any cycle, when written out as a sequence of connected nodes, would necessarily repeat the starting node at the end. It is also clear that all nodes of the graph are covered in the above sequence. In particular, nodes $(i, 1)$, $(i, 2)$, ..., (i, n) are connected in that order for odd integers i and in the reverse order for even integers i . Hence, E forms a spanning tree and has $n^2 - 1$ edges.

E has minimal weight: It is easy to see that every edge in G has at least unit weight, since weight of edge between distinct nodes is minimal for edges between node pairs of the form $\{(i, j), (i+1, j)\}$ or $\{(i, j), (i, j+1)\}$, evaluating to an edge weight of 1. Since all edges in E are of this form, all edges in E have unit weight and total weight of E is equal to $n^2 - 1$. Finally, any spanning tree of G has exactly $n^2 - 1$ edges, so the weight of the MST must be at least $n^2 - 1$, thus proving that weight of E is minimal.

3) 16 pts

You are trying to decide the best order in which to answer your CS-570 exam questions within the duration of the exam. Suppose that there are n questions with points p_1, p_2, \dots, p_n and you need time t_i to completely answer the i^{th} question. You are confident that all your completely answered questions will be correct (and get you full credit for them), and the TAs will give you partial credit for an incompletely answered question, in proportion to the time you spent on that question. Assuming that the total exam duration is T , give a greedy algorithm to decide the order in which you should attempt the questions and prove that the algorithm gives you an optimal answer.

Solution: This is similar to the fractional knapsack problem. The greedy algorithm is as follows. Calculate $\frac{p_i}{t_i}$ for each $1 \leq i \leq n$ and sort the questions in descending order of $\frac{p_i}{t_i}$. Let the sorted order of questions be denoted by $s(1), s(2), \dots, s(n)$. Answer questions in the order $s(1), s(2), \dots$ until $s(j)$ such that $\sum_{k=1}^j t_{s(k)} \leq T$ and $\sum_{k=1}^{j+1} t_{s(k)} > T$. If $\sum_{k=1}^j t_{s(k)} = T$ then stop, otherwise partially solve the question $s(j + 1)$ in the time remaining.

We'll use induction to prove optimality of the above algorithm. The induction hypothesis $P(l)$ is that there exists an optimal solution that agrees with the selection of the first l questions by the greedy algorithm.

Inductive Step: Let $P(1), P(2), \dots, P(l)$ be true for some arbitrary l , i.e. the set of questions $\{s(1), s(2), \dots, s(l)\}$ are part of an optimal solution O . It is clear that O should also be optimal with respect to the remaining questions $\{1, 2, \dots, n\} \setminus \{s(1), s(2), \dots, s(l)\}$ in the remaining time $T - \sum_{k=1}^l t_{s(k)}$. Since $P(1)$ is true, there exists an optimal solution, not necessarily distinct from O , that selects the question with the largest value of $\frac{p_i}{t_i}$ in the remaining set of questions, i.e. the question $s(l + 1)$ is selected. Since $s(l + 1)$ is also the choice of the proposed greedy algorithm, $P(l + 1)$ is proved to be true.

Induction Basis: To show that $P(1)$ is true, we proceed by contradiction. Assume $P(1)$ to be false. Then $s(1)$ is not part of any optimal solution. Let $a \neq s(1)$ be a partially solved question in some optimal solution O' (if O' does not contain any partially solved questions then we take a to be any arbitrary question in O'). Taking time $\min(t_a, T, t_{s(1)})$ away from question a and devoting to question $s(1)$ gives the improvement in points equal to $\left(\frac{p_{s(1)}}{t_{s(1)}} - \frac{p_a}{t_a}\right) \min(t_a, T, t_{s(1)}) \geq 0$ since $\frac{p_i}{t_i}$ is largest for $i = s(1)$. If the improvement is strictly positive then it contradicts optimality of O' and implies the truth of $P(1)$. If improvement is 0, then a can be switched out for $s(1)$ without affecting optimality and thus implying that $s(1)$ is part of an optimal solution which in turn means that $P(1)$ is true.

4) 16 pts

An Edge Cover on a graph $G = (V; E)$ is a set of edges $X \subseteq E$ such that every vertex in V is incident to an edge in X . In the **Bipartite** Edge Cover problem, we are given a bipartite graph and wish to find an Edge Cover that contains $\leq k$ edges. Design a polynomial-time algorithm based on network flow (max flow or circulation) to solve it and justify your algorithm.

Solution:

If the network contains isolated node, then the problem is trivial since there is no way to do edge cover. Now we only consider the case that each node has at least 1 incident edge.

- i) The goal of edge cover is to choose as many edges as possible which cover 2 nodes. You can find this subset of edges by running Bipartite Matching on the original graph, and taking exactly the edges which are in the matching. (Equivalently, you can set capacity of each edge in the graph as 1. Set a super source node s connecting each “blue” node with edge capacity 1 and a super destination t connecting each “red” node with edge capacity 1. Then run max-flow to get the subset of edges connecting 2 nodes in G)
- ii) What remains is to cover the remaining nodes. Since you can only cover a single node (of those remaining) with each selected edge, simply choose an arbitrary incident edge to each uncovered node.
- iii) Set the set of edges you choose in the above two steps as set X . Count the total number of edges in X and compare the size with k .

Proof:

It is obvious that X is an edge cover. The remaining part is to show that X contains the minimum number of edges among all possible edge covers.

Denote the number of edges we find in step i) as x_1 ; denote the number of edges we find in step ii) as x_2 .

Then we have $x_1 * 2 + x_2 = |V|$.

Consider an arbitrary edge cover set Y . Suppose Y contains y_1 edges, each of which is counted as the one covering 2 nodes. Suppose Y contains y_2 edges, each of which is counted as the one covering 1 nodes. (We can ignore the edges covering zero nodes, because we can delete those edges from Y without affecting the coverage)

Then we have $y_1 * 2 + y_2 = |V|$.

Here x_1 must be the maximum number of edges that covers 2 nodes in the bipartite graph, because we do bipartite matching in G (max-flow in G' including s and t). Therefore, we have $x_1 \geq y_1$.

Then we have:

$$x_1 + x_2 = |V| - x_1 = y_1 * 2 + y_2 - x_1 = y_1 + y_2 - (x_1 - y_1) \leq y_1 + y_2.$$

The above algorithm gives the minimum number of edges for covering the nodes.

5) 16 pts

Imagine starting with the given decimal number n , and repeatedly chopping off a digit from one end or the other (your choice), until only one digit is left. The square-depth $\text{SQD}(n)$ of n is defined to be the maximum number of perfect squares you could observe among all such sequences. For example, $\text{SQD}(32492) = 3$ via the sequence

$$32492 \rightarrow 3249 \rightarrow 324 \rightarrow 24 \rightarrow 4$$

since 3249, 324, and 4 are perfect squares, and no other sequence of chops gives more than 3 perfect squares. Note that such a sequence may not be unique, e.g.

$$32492 \rightarrow 3249 \rightarrow 249 \rightarrow 49 \rightarrow 9$$

also gives you 3 perfect squares, viz. 3249, 49, and 9.

Describe an efficient algorithm to compute the square-depth $\text{SQD}(n)$, of a given number n , written as a d -digit decimal number $a_1a_2 \dots a_d$. Analyze your algorithm's running time. Your algorithm should run in time polynomial in d . You may assume the availability of a function `IS_SQUARE(x)` that runs in constant time and returns 1 if x is a perfect square and 0 otherwise.

Solution:

We can solve this using dynamic programming.

First, we define some notation: let $n_{ij} = a_i \dots a_j$. That is, n_{ij} is the number formed by digits i through j of n . Now, define the subproblems by letting $D[i, j]$ be the square-depth of n_{ij} .

The solution to $D[i, j]$ can be found by solving the two subproblems that result from chopping off the left digit and from chopping off the right digit, and adding 1 if n_{ij} itself is square. We can express this as a recurrence:

$$D[i, j] = \max(D[i + 1, j], D[i, j - 1]) + \text{IS-SQUARE}(n_{ij})$$

The base cases are $D[i, i] = \text{IS-SQUARE}(a_i)$, for all $1 \leq i \leq d$. The solution to the general problem is $\text{SQD}(n) = D[1, d]$.

There are $\Theta(d^2)$ subproblems, and each takes $\Theta(1)$ time to solve, so the total running time is $\Theta(d^2)$.

*** Some students did a greedy approach to solve this problem which is not true for this problem. In each step of the program they explore removing which digit results in a perfect square number, however it might be the other non-removed digit that will produce more squares in the future steps.

6) 16 pts

The Longest Path is the problem of deciding whether a graph has a simple path of length greater or equal to a given number k .

a) Show that the Longest Path problem is in NP (2 pts)

b) Show how the Hamiltonian Cycle problem can be reduced to the Longest Path problem. (14 pts)

Note: You can choose to do the reduction in b either directly, or use transitivity of polynomial time reduction to first reduce Hamiltonian Cycle to another problem (X) and then reduce X to Longest Path.

a) Polynomial length certificate: ordered list of nodes on a path of length $\geq k$

polynomial time Certifier:

check that nodes do not repeat in $O(n \log n)$

check that the length of the path is at least k in $O(n)$

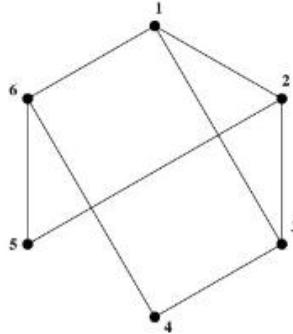
check that there are edges between every two adjacent nodes on the path in $O(n)$

check that the length of the path is at least k in $O(n)$

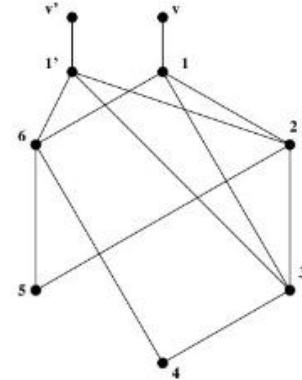
b) Two possible solutions:

I. To reduce directly from Hamiltonian Cycle

Given a graph $G = (V, E)$ we construct a graph G' such that G contains a Ham cycle iff G' contains a simple path of length at least $N+2$. This is done by choosing an arbitrary vertex u in G and adding a copy, u' , of it together with all its edges. Then add vertices v and v' to the graph and connect v with u and v' to u' . We give a cost of 1 to all edges in G' . See figure below:



G



G'

Proof:

A) If there is a HC in G we can find a simple path of length $n+2$ in G' : This path starts at v' goes to u' and follows the HC to u and then to v . The length is $n+2$

B) If there is a simple path of length $n+2$ in G' , there is a HC in G : This path must

include nodes v' and v because there are only n nodes in G and a simple path of length $n+2$ must include v and v' . Moreover, v and v' must be the two ends of this path, otherwise, the path will not be a simple path since there is only one way to get to v and v' . So, to find the HC in G , just follow the path from u' to u .

II. To reduce using Hamiltonian Path

First reduce Ham Cycle to Ham Path (very similar to the above reduction)

Then reduce Ham Path to Longest path. This is very straightforward. To find out if there is a Ham Path in G you can assign weights of 1 to all edges and ask the blackbox if there is a path of length at least n in G .

1) 20 pts

Mark the following statements as **TRUE** or **FALSE**. No need to provide any justification.

[/FALSE]

All the NP-hard problems are in NP.

[/FALSE]

Given a weighted graph and two nodes, it is possible to list all shortest paths between these two nodes in polynomial time.

[TRUE/]

In the memory efficient implementation of Bellman-Ford, the number of iterations it takes to converge can vary depending on the order of nodes updated within an iteration

[/FALSE]

There is a feasible circulation with demands $\{d_v\}$ if $\sum_v d_v = 0$.

[/FALSE]

Not every decision problem in P has a polynomial time certifier.

[TRUE/]

If a problem can be reduced to linear programming in polynomial time then that problem is in P.

[/FALSE]

If we can prove that $P \neq NP$, then a problem $A \in P$ does not belong to NP.

[/FALSE]

If all capacities in a flow network are integers, then every maximum flow in the network is such that flow value on each edge is an integer.

[/FALSE]

In a dynamic programming formulation, the sub-problems must be mutually independent.

[~~TRUE~~/]

In the final residual graph constructed during the execution of the Ford–Fulkerson Algorithm, there's no path from sink to source.

2) 16 pts

In the Bipartite Directed Hamiltonian Cycle problem, we are given a bipartite directed graph $G = (V; E)$ and asked whether there is a simple cycle which visits every node exactly once. Note that this problem might potentially be easier than Directed Hamiltonian Cycle because it assumes a bipartite graph. Prove that Bipartite Directed Hamiltonian Cycle is in fact still NP-Complete.

Solution:

- i) This problem is NP. Given a candidate path, we just check the nodes along the given path one by one to see if every node in the network is visited exactly once and if each consecutive node pair along the path are joined by an edge.
- ii) To prove the problem is NP-complete, we reduce Directed Hamiltonian Cycle (DHC) problem to Bipartite Directed Hamiltonian Cycle (BDHC) problem.

Given an arbitrary directed graph G , we split each vertex v in G into two vertex v_{in} and v_{out} . Here v_{in} connects all the incoming edges to v in G ; v_{out} connects all the outgoing edges from v in G . Moreover, we connect one directed edge from v_{in} to v_{out} . After doing these operations for each node in G , we form a new graph G' .

Here G' is bipartite graph, because we can color each v_{in} “blue” and each v_{out} “red” without any coloring conflict.

If there is DHC in G , then there is a BDHC in G' . We can replace each node v on the DHC in G into consecutive nodes v_{in} and v_{out} , and (v_{in}, v_{out}) is an edge in G' . Then the new path is a BDHC in G' .

On the other hand, if there is BDHC in G' , then there is a DHC in G . Note that if v_{in} is on the BDHC, v_{out} must be the successive node on the BDHC. Then we can merge each node pair (v_{in}, v_{out}) on the BDHC in G' into node v and form a DHC in G .

In sum, G has DHC if and only if G' has a BDHC. This completes the reduction, and we confirm that the given problem is NP-complete

3) 16 pts

A tourism company is providing boat tours on a river with n consecutive segments. According to previous experience, the profit they can make by providing boat tours on segment i is known as a_i . Here a_i could be positive (they earn money), negative (they lose money), or zero. Because of the administration convenience, the local community of the river requires that the tourism company should do their boat tour business on a contiguous sequence of the river segments, i.e, if the company chooses segment i as the starting segment and segment j as the ending segment, all the segments in between should also be covered by the tour service, no matter whether the company will earn or lose money. The company's goal is to determine the starting segment and ending segment of boat tours along the river, such that their total profit can be maximized. Design an efficient algorithm to achieve this goal, and analyze its run time (Note that brute-force algorithm achieves $\Theta(n^2)$, so your algorithm must do better.)

Solution1:

Using dynamic programming:

Define $\text{OPT}(i)$ as the maximum total profit the company can get when running the boat tours on a contiguous sequence of segments starting at segment i .

Base case: $\text{OPT}(n) = a_n$.

Recursive relation: $\text{OPT}(i) = \max \{\text{OPT}(i+1) + a_i, a_i\}$, for $1 \leq i < n$

Algorithm:

```
For i = n, ...., 1
    Compute OPT(i).
End
```

Maximum profit = $\max_{i=1}^n \{\text{OPT}(i)\}$. Denote i^* as the starting index which achieves the maximum profit, then i^* is the optimal starting segment

```
For i = i*,....., n
    If (OPT(i) == a_i)
        Set j* = i;
        Break;
    End
End
The value j* is the index of the optimal ending segment.
```

Complexity: Computing all the OPT values takes time $O(n)$, comparing all OPT values and find the maximum profit and the corresponding starting segment takes time $O(n)$. Finding the optimal ending segment takes time $O(n)$. In sum, the algorithm takes time $O(n)$

Remark: in this solution, we implicitly assume that the company must provide the boat tour, while providing no boat tour is not a choice. If you consider providing no boat tour also as a choice, it is also treated as a correct solution, however, the step of computing the maximum profit above solution should be modified as follows:

$$\text{Maximum profit} = \max \{0, \max_{i=1}^n \{\text{OPT}(i)\}\}.$$

Correspondingly, if 0 is the best result you can get, you need to claim that providing no boat tour is the best solution, and there is no need to confirm the starting segment or ending segment.

Solution 2 (outline):

Using divide and conquer.

- i) Divide operation: Divide the profit sequences of the n consecutive segments, denoted as $A[1,n]$, as two parts:
 $a_1, \dots, a_{\lfloor n/2 \rfloor}$ and $a_{\lfloor n/2+1 \rfloor}, \dots, a_n$, denoted as $A[1, \lfloor n/2 \rfloor]$ and $A[\lfloor n/2+1, n]$ respectively.
- ii) Merge operation:
 Suppose you successfully find the maximum profit in $A[1, \lfloor n/2 \rfloor]$ and $A[\lfloor n/2+1, n]$, denoted as $P_{\text{left}}(1, n)$, $P_{\text{right}}(1, n)$, and the corresponding contiguous sequence of segments, denoted as $B_{\text{left}}(1, n)$ and $B_{\text{right}}(1, n)$
 Then the optimal contiguous segment can be in one of the three cases:
 - a) $B_{\text{left}}(1, n)$
 - b) $B_{\text{right}}(1, n)$
 - c) An optimal contiguous segment sequence crossing $A[1, \lfloor n/2 \rfloor]$ and $A[\lfloor n/2+1, n]$, denoted as $B_{\text{cross}}(1, n)$.

For $B_{\text{cross}}(1, n)$, denote the corresponding profit as $P_{\text{cross}}(1, n)$. The method to confirm $B_{\text{cross}}(1, n)$ and $P_{\text{cross}}(1, n)$ is as follows:

- Starting from sequence $[a_{\lfloor n/2 \rfloor}, a_{\lfloor n/2+1 \rfloor}]$, compute the summation $S_{\text{left}}(1) = a_{\lfloor n/2 \rfloor} + a_{\lfloor n/2+1 \rfloor}$ as one candidate solution for $P_{\text{cross}}(1, n)$.
- Next, including $a_{\lfloor n/2-1 \rfloor}$ into the above sequence as $[a_{\lfloor n/2-1 \rfloor}, a_{\lfloor n/2 \rfloor}, a_{\lfloor n/2+1 \rfloor}]$, compute the summation of the three elements in the sequence as the second candidate solution: $S_{\text{left}}(2) = S_{\text{left}}(1) + a_{\lfloor n/2-1 \rfloor}$.

- Keep including the element one by one to the left until a_1 is included, during each step, compute and record the summation values.
- Find $S_{\text{opt_left}} = \max_i \{S_{\text{left}}(i)\}$, record the corresponding index of the included element that achieves the maximum, say i_{cross^*} . Then i_{cross^*} is the optimal starting index for $B_{\text{cross}}(1,n)$;
- Starting from $[a_{i_{\text{cross}^*}}, \dots, a_{n/2+1}]$ includes the element to the right one by one until a_n is included, during each step, compute the summation values in the same way as in the left part, denote the value as $S_{\text{right}}(i)$.
- Find $P_{\text{cross}}(1,n) = \max_i \{S_{\text{right}}(i)\}$, record the corresponding index of the included element that achieves the maximum, say j_{cross^*} . Then j_{cross^*} is the optimal ending index for $B_{\text{cross}}(1,n)$;

Then

Maximum Profit = $\max \{P_{\text{left}}(1,n), P_{\text{right}}(1,n), P_{\text{cross}}(1,n)\}$, and the corresponding optimal starting index and ending index are the ones that achieve the maximum.

- Before doing step ii) for $A[1,n]$, recursively run the above procedure in each array $A[1,n/2]$ and $A[n/2+1, n]$ and so on.
- Termination condition: for $A[i,j]$, if $i=j$, return a_i as the maximum profit, return i as both the optimal starting and ending index in $A[i,j]$.

Complexity:

The Divide operation takes time $O(1)$.

The Merge operation takes time $O(n)$.

Define $T(n)$ as the running time,

$$T(n) = 2*T(n/2) + O(n)$$

The complexity is $O(n \log(n))$.

Remark: similar as in solution 1, considering no bout tour as a choice is also treated as a correct solution.

4) 16 pts

Consider the following matching problem. There are m students s_1, s_2, \dots, s_m and a set of n companies $C = \{c_1, c_2, \dots, c_n\}$. Each student can work for only one company, whereas company c_j can hire up to b_j students. Student s_i has a preferred set of companies $\Lambda_i \subseteq C$ at which he/she is willing to work. Your task is to find an assignment of students to companies such that all of the above constraints are satisfied and each student is assigned. Formulate this as a network flow problem and describe any subsequent steps necessary to arrive at the solution. Prove correctness.

Construction of flow network: Represent each student and each company as a separate node. Add one source node s and a sink node t for a total of $m + n + 2$ nodes. Add the following directed edges with capacities:

- i. $s \rightarrow s_i$ with capacity 1 unit, for each $1 \leq i \leq m$,
- ii. For each $1 \leq i \leq m$, $s_i \rightarrow c$ with capacity 1 unit for all nodes $c \in \Lambda_i$.
- iii. $c_j \rightarrow t$ with capacity b_j units, for each $1 \leq j \leq n$.

Constructing the solution: Run any max-flow algorithm on the above network to get the realization of edge flows under a max flow configuration (lets call it O for brevity). If an edge $s_i \rightarrow c_j$ shows a non-zero flow in this configuration, assign student s_i to company c_j . Repeat this process for each edge between the student nodes and the company nodes to get the final assignment.

Proof of correctness: We have to show that the solution so obtained satisfies all constraints of the problem.

- i. Since all outgoing edges from s_i are to the node set Λ_i , it is impossible for s_i to have a flow to a node outside Λ_i in configuration O .
- ii. Since the only outgoing edge from c_j is of capacity b_j , configuration O cannot have more than b_j incoming edges of non-zero flow to node c_j . Thus, not more than b_j students can get assigned to company c_j .
- iii. As s_i has a single incoming edge and multiple outgoing edges of capacity 1, configuration O cannot have more than one outgoing edge from s_i with non-zero flow. Hence, s_i can get assigned to at most one company.

We have proved that our mapping from configuration O to an assignment does not violate any of the constraints except possibly that all students might not be assigned. Note that this may happen in practice if it is impossible to assign all students while satisfying all of the given constraints. Thus, what we need to show is that if there exists a feasible assignment then configuration O will have each $s \rightarrow s_i$ edge carry a non-zero flow. Given a feasible assignment $\{(s_i, c_{\sigma(i)}), 1 \leq i \leq m\}$, by construction of the flow network, it is possible to set each edge $s_i \rightarrow c_{\sigma(i)}$ to carry 1 unit of flow. By feasibility of the assignment, company c_j gets no more than b_j incoming edges with non-zero flow, so the outgoing edge from c_j has enough capacity to carry away all incident flow on node c_j . Finally, since each s_i has an outgoing flow of 1 unit in

this assignment, all $s \rightarrow s_i$ edges can be set to have 1 unit of flow, completing the proof.

5) 16 pts

Consider a directed, weighted graph G where all edge weights are positive. You have one Star, which lets you change the weight of any one edge to zero. In other words, you may change the weight of any one edge to zero. Give an efficient algorithm using Dijkstra's algorithm to find a lowest-cost path between two vertices s and t , given that you may set one edge weight to zero. Note: you will receive 10 pts if your algorithm is efficient. You will receive full points (16 pts) if your algorithm has the same run time complexity as Dijkstra's algorithm.

Solution:

Use Dijkstra's algorithm to find the shortest paths from s to all other vertices. Reverse all edges and use Dijkstra to find the shortest paths from all vertices to t . Denote the shortest path from u to v by $u \sim v$, and its length by $\delta(u, v)$.

Now, try setting each edge to zero. For each edge $(u, v) \in E$, consider the path $s \sim u \rightarrow v \sim t$. If we set $w(u, v)$ to zero, the path length is $\delta(s, u) + \delta(v, t)$. Find the edge for which this length is minimized and set it to zero; the corresponding path $s \sim u \rightarrow v \sim t$ is the desired path. The algorithm requires two invocations of Dijkstra, and an additional $\Theta(E)$ time to iterate through the edges and find the optimal edge to take for free. Thus the total running time is the same as that of Dijkstra:

$O(E + V \lg V)$: based on using Fibonacci heap

$O((|V|+|E|) \log (|V|))$: based on using binary heap

6) 16 pts

You are given n rods; they are of length l_1, l_2, \dots, l_n , respectively. Our goal is to connect all the rods and form a single rod. The length after connecting two rods and the cost of connecting them are both equal to the sum of their lengths. Give an algorithm to minimize the cost of connecting them to form a single rod. State the complexity of your algorithm and prove that your algorithm is optimal.

1st interpretation: At each step we connect a single rod to the set of rods that are already connected together.

Of course, the solution is to sort rods by length and connect them in the order of increasing length. To prove this is optimal, you can assume that there is an optimal solution and compare our solution to the optimal solution: At each step our solution stays ahead of (or at least does no worse than) the optimal solution.

Sort Takes $O(n \log n)$

We can use mathematical induction to prove that we always stay ahead of the optimal solution.

Claim: at each step the cost of connecting rods in our solution is less than or equal to that of the other solution and the length of the rod after this connection is smaller than or equal in size to that of the other solution.

Base case: first two shortest rods – obviously this is the opt solution

Assuming that the cost of connecting k rods in our solution is less than or equal to that of another (optimal) solution, we can show that the cost of connecting the next ($k+1^{\text{st}}$) rod will be less than or equal to the cost of connecting the $k+1^{\text{st}}$ rod in the other solution:

Cost of the last connection in our solution = total length of all rods 1 to $k+1$ (ordered by length)

Cost of the last connection in the other solution = total length of all rods 1 to $k+1$ (not necessarily ordered by length)

It is obvious that the cost of our last connection is smaller or equal to the cost of the other connection because the only way to get the minimum total length of $k+1$ rods is to pick the shortest $k+1$ rods.

2nd interpretation: At each step we can connect any rod or set of already connected rods to another rod or set of already connected rods.

The solution is to keep connecting the smallest two rods or sets of already connected rods.

Implementation: Place all rods in a min heap with their key values representing their length. At each step extract two elements from the set and insert the combined rod back into the min heap.

This takes a total of $O(n \log n)$ time

Fact 1: the cost contribution of a rod i to the total assembly cost is $\text{Length}(i) * \text{Level}(i)$, where $\text{Level}(i)$ is the level at which the rod is first assembled with other rods (level 1=root level, level 2= level below root, etc.).

Proof: By observation of cost of assembly tree

Fact2: There is an optimal assembly tree of rods in which the two smallest rods are leaf nodes at the lowest level of the tree and the children of the same parent.

Proof: let's say the two smallest rods are not leaf nodes at the lowest level of the tree, using fact 1, we can swap these rods with two rods at the lowest level of the tree and thereby reducing the total cost of the assembly tree. If the two rods are leaf nodes at the lowest level but not children of the same parent we can swap two rods to make these rods children of the same parent without changing the total cost of the tree (again based on Fact 1).

Assume that there is an optimal assembly tree T^* and our solution produces tree T . We will show that our tree T is also optimal.

To do this, we apply fact 2 to T^* and move the smallest rods to the lowest level of the tree as children of the same parent—without increasing the cost of T^* . We then eliminate the two smallest rods at the bottom of the two trees(since these are the first two rods that are assembled in our algorithm) and assume that there is a new combined rod in place of their parent node. We will get two new trees $T^{*''}$ and T' , where

$$\begin{aligned}\text{Total assembly cost of } T^* &= \text{Total assembly cost of } T^{*''} + \text{total length of the two smallest rods} \\ \text{Total assembly cost of } T &= \text{Total assembly cost of } T' + \text{total length of the two smallest rods}\end{aligned}$$

Since the costs of the two new trees are reduced by the same amount we can now compare the cost of our new tree T' with the cost of the new optimal tree $T^{*''}$. And show that assembly tree T' is also optimal (as compared to the optimal tree $T^{*''}$).

To do this, we apply fact 2 recursively and place the two smallest rods in $T^{*''}$ at the lowest level of the tree and under the same parent node without increasing the cost of $T^{*''}$. These are the next two rods that are combined in our algorithm. We then eliminate these two rods in both trees T' and $T^{*''}$, etc.

By repeating the same steps (applying fact 2 and eliminating the two smallest rods from the optimal assembly tree and our tree) recursively, we will find an optimal solution that follows the same exact assembly sequence that is found in our algorithm. Therefore we can state that our algorithm also produces an optimal assembly tree.

1) 20 pts

Mark the following statements as **TRUE**, **FALSE**. No need to provide any justification.

[/FALSE]

If P = NP, then all NP-Hard problems can be solved in Polynomial time.

[/FALSE]

Dynamic Programming approach only works when used on problems with non-overlapping sub problems.

[/FALSE]

In a divide & conquer algorithm, the size of each sub-problem must be at most half the size of the original problem.

[/FALSE]

In a 0-1 knapsack problem, a solution that uses up all of the capacity of the knapsack will be optimal.

[/FALSE]

If a problem X can be reduced to a known NP-hard problem, then X must be NP-hard.

[TRUE/]

If $\text{SAT} \leq_P A$, then A is NP-hard.

[TRUE/]

The recurrence $T(n) = 2T(n/2) + 3n$, has solution $T(n) = \theta(n \log(n^2))$.

[/FALSE]

Consider two positively weighted graphs $G_1 = (V, E, w_1)$ and $G_2 = (V, E, w_2)$ with the same vertices V and edges E such that, for any edge $e \in E$, we have $w_2(e) = (w_1(e))^2$. For any two vertices $u, v \in V$, any shortest path between u and v in G_2 is also a shortest path in G_1 .

[/FALSE]

If an undirected graph $G=(V,E)$ has a Hamiltonian Cycle, then any DFS tree in G has a depth $|V| - 1$.

[TRUE/]

Linear programming is at least as hard as the Max Flow problem.

2) 15 pts

A company makes three models of desks, an executive model, an office model and a student model. Building each desk takes time in the cabinet shop, the finishing shop and the crating shop as shown in the table below:

Type of desk	Cabinet shop	Finishing shop	Crating shop	Profit
Executive	2	1	1	150
Office	1	2	1	125
Student	1	1	.5	50
Available hours	16	16	10	

How many of each type should they make to maximize profit? Use linear programming to formulate your solution. Assume that real numbers are acceptable in your solution.

Solution:

Start by defining your variables:

x = number of executive desks made

y = number of office desks made

z = number of student desks made

Maximize $P=150x+125y+50z$.

Subject to:

$2x + y + z \leq 16$ cabinet hours

$x + 2y + z \leq 16$ finishing hours

$x + y + .5z \leq 10$ crating hours

$x \geq 0, y \geq 0, z \geq 0$

3) 12 pts

Given a graph $G=(V, E)$ and a positive integer $k < |V|$. The longest-simple-cycle problem is the problem of determining whether a simple cycle (no repeated vertices) of length k exists in a graph. Show that this problem is NP-complete.

Solution:

Clearly this problem is in NP. The certificate will be a cycle of the graph, and the certifier will check whether the certificate is really a cycle of length k of the given graph.

We will reduce HAM-CYCLE to this problem. Given an instance of HAM-CYCLE with graph $G=(V, E)$, construct a new graph $G'=(V', E)$ by adding one isolated vertex u to G . Now ask the longest-simple-cycle problem with $k = |V| < |V'|$ for graph G' .

If there is a HAM-CYCLE in G , it will be a cycle of length $k = |V|$ in G' .

If there is no HAM-CYCLE in G' , there must not be no cycle of length $|V|$ in G' . If there is, we know the cycle does not contain u , because it is isolated. So the cycle will contain all vertex in $|V|$, which is a HAM-CYCLE of the G .

The reduction is in polynomial time, so longest-simple-path is NP-Complete.

4) 15 pts

Suppose there are n steps, and one can climb either 1, 2, or 3 steps at a time. Determine how many different ways one can climb the n steps. E.g. if there are 5 steps, these are some possible ways to climb them: (1,1,1,1,1), (1,2,1,1), (3, 2), (2,3), etc. Your algorithm should run in linear time with respect to n . You need to include your complexity analysis.

Solution:

Let $T(n)$ denote the number of different ways for climbing up n stairs.

There are three choices for each first step: either 1, 2, or 3. $T(n) = T(n-1) + T(n-2) + T(n-3)$

The boundary conditions :

when there is only one step, $T(1) = 1$. If only two steps, $T(2) = 2$. $T(3)= 4$. ((1,1,1), (1,2), (2,1), (3))

Pseudo code as below:

```
if n is 1  
    return 1  
else if n is 2  
    return 2  
else  
    T[1] = 1  
    T[2] = 2  
    T[3]=4  
    for i = 4 to n do  
        T[i] = T[i-1]+T[i-2]+ T[i-3]  
    return T[n]
```

The time complexity is $\Theta(n)$.

5) 15 pts

We'd like to select frequencies for FM radio stations so that no two are too close in frequency (creating interference). Suppose there are n candidate frequencies $\{f_1, \dots, f_n\}$. Our goal is to pick as many frequencies as possible such that no two selected frequencies f_i, f_j have $|f_i - f_j| < e$ (for a given input variable e). Design a greedy algorithm to solve the problem. Prove the optimality of the algorithm and analyze the running time.

Proof:

I claim that there exists some optimal solution that makes the same first choice as (in terms of which selected frequency has the lowest value) that I do. Consider any optimal solution OPT .

Let p be my first choice and q be OPT 's first choice. If $p = q$, our proof is complete. If it isn't, we know that $p < q$; now consider some other set $OPT1 = OPT - \{q\} + \{p\}$; that is, OPT with its first choice replaced by mine. We know this is a valid set (meaning no frequencies are within e of one another), because $p < q$, and p is the first element in $OPT1$. Because nothing was within e of q , none can be within e of p . We also know that $OPT1$ is an optimal (maximum size) set, because it is the same size as OPT . Therefore, $OPT1$ is an optimal set that includes my first choice. (and therefore, by induction, the second choice will be correct, etc.)

6) 12 pts

Let S be an NP-complete problem, and Q and R be two problems whose classification is unknown (i.e. we don't know whether they are in NP, or NP-hard, etc.). We do know that Q is polynomial time reducible to S and S is polynomial time reducible to R. Mark the following statements True or False **based only on the given information, and explain why.**

(i) Q is NP-complete

False. Because $Q \leq_p S$, Q is at most as hard as S. Because S is in NPC, Q is not necessary to be in NPC, e.g., it could be in P, or could be in NP but not in NPC.

(ii) Q is NP-hard

False. Because $Q \leq_p S$, Q is at most as hard as S. Then Q is not necessary to be in NP-hard, e.g., it could be in P.

(iii) R is NP-complete

False. Because $S \leq_p R$, R is at least as hard as S. Then R is not necessary to be NPC. It is possible to be in NP-hard but not in NPC.

(iv) R is NP-hard

True. Because $S \leq_p R$, R is at least as hard as S. Then R is NP-hard.

7) 11 pts

Consider there are n students and n rooms. A student can only be assigned to one room. Each room has capacity to hold either one or two students. Each student has a subset of rooms as their possible choice. We also need to make sure that there is at least one student assigned to each room.

Give a polynomial time algorithm that determines whether a feasible assignment of students to rooms is possible that meets all of the above constraints. If there is a feasible assignment, describe how your solution can identify which student is assigned to which room.

Solution:

Construct a flow network as follows,

For each student i create a vertex a_i , for each room j create a vertex b_j , if room j is one of student i 's possible choices, add an edge from a_i to b_j with upper bound 1. Create a super source s , connect s to all a_i with an edge of lower bound and upper bound 1.

Create a super sink t , connect all b_j to t with an edge of lower bound 1 and upper bound 2.

Connect t to s with an edge of lower bound and upper bound n .

Find an integral circulation of the graph, because all edges capacity and lower bound are integer, this can be done in polynomial time.

If there is one, for each a_i and b_j , check whether the flow from a_i to b_j is 1, if yes, assign student i to room j .

1) 20 pts

Mark the following statements as **TRUE** or **FALSE**. No need to provide any justification.

[**TRUE/FALSE**]

Let X be a decision problem. If we prove that X is in the class NP and give a poly-time reduction from X to 3-SAT, we can conclude that X is NP-complete.

[**TRUE/FALSE**]

Let A be an algorithm that operates on a list of n objects, where n is a power of two. A spends $\Theta(n^2)$ time dividing its input list into two equal pieces and selecting one of the two pieces. It then calls itself recursively on that list of $n/2$ elements. Then A 's running time on a list of n elements is $O(n)$.

[**TRUE/FALSE**]

If there is a polynomial time algorithm to solve problem A then A is in NP.

[**TRUE/FALSE**]

A pseudo-polynomial time algorithm is always slower than a polynomial time algorithm.

[**TRUE/FALSE**]

In a dynamic programming formulation, the sub-problems must be non-overlapping.

[**TRUE/FALSE**]

A spanning tree of a given undirected, connected graph $G = (V, E)$ can be found in $O(E)$ time.

[**TRUE/FALSE**]

Ford-Fulkerson can return a zero maximum flow for flow networks with non-zero capacities.

[**TRUE/FALSE**]

If $f(n) = \Theta(g(n))$ and $g(n) = \Theta(h(n))$, then $h(n) = \Theta(f(n))$

[**TRUE/FALSE**]

There is a polynomial-time solution for the 0/1 Knapsack problem if all items have the same weight but different values.

[**TRUE/FALSE**]

If there are negative cost edges in a graph but no negative cost cycles, Dijkstra's algorithm still runs correctly.

2) 8 pts

Let $G = (V, E)$ be a simple graph with n vertices. Suppose the weight of every edge of G is one.

Note: In a simple graph there is at most one edge directly connecting any two nodes.

For example, between nodes u and v there will be at most one edge uv .

(a) What is the weight of a minimum spanning tree of G ? (1 pt)

(b) Suppose we change the weight of two edges of G to $1/2$. What is the weight of the minimum spanning tree of G ? (2 pts)

(c) Suppose we change the weight of three edges of G to $1/2$. What is the minimum and maximum possible weights for the the minimum spanning tree of G ? (2 pts)

(d) Suppose we change the weight of $k < V$ edges of G to $1/2$. What is the minimum and maximum possible weights for the the minimum spanning tree of G ? (3 pts)

1. Any spanning tree of G has exactly $V - 1$ edges, since all the weights are one its total weight is $V - 1$.
2. To answer this question, consider running Kruskal on G , it will first pick the two lighter edges as they cannot form a cycle (the graph is simple), then $V - 3$ other edge each with weight one. Therefore, the total value is $V - 3 + 2(1/2) = V - 2$.
3. Use the similar approach as (b). Kruskal first selects two light edges. There are two possibilities for the third edge: (i) it forms a cycle with the first two edges or (ii) it does not form a cycle with the first two edges not. In case (i), Kruskal selects two edges of weight $1/2$ and $V - 3$ edges of weight 1, therefore, the total weight is $V - 3 + 2(1/2) = V - 2$. In case (ii), Kruskal selects three edges of weight $1/2$ and $V - 4$ edges of weight 1, therefore, the total weight is $V - 4 + 3(1/2) = V - 5/2$.
4. Similar to (c), the minimum weight happens when the k light edges do not form any cycle. In this case, MST contains k edges of weight $1/2$ and $n - 1 - k$ edges of weight 1. Therefore, its weight is $k/2 + n - 1 - k = n - k/2 - 1$. The maximum weight happens if the k edges span as few vertices as possible. This happens when the k edges are part of an almost *complete graph*. Let h be a variable denoting the number of nodes in this subgraph such that the number of edges in a complete graph comprised of these edges nodes exceeds k . In particular, define h to be the smallest number such that, the binomial coefficient $\text{Binomial}[h,2] > k$, ie., $(h \text{ choose } 2) > k$. Accordingly, we can pack all k light edges between h vertices. Consequently, the MST has $h - 1$ edges of weight $1/2$ and $n - 1 - (h - 1)$ edges of weight 1. Therefore, its weight is $n - h/2 - 1/2$.

Rubric:

For each part (a,b,c,d), No partial marking.

This also applies to the parts where minimum and maximum is asked.

If either is wrong then there is no partial marks for that part.

Q3. 14 pts

Recall that in the discussion class we showed that the Bipartite **Undirected** Hamiltonian Cycle problem is NP-complete. Now in the Bipartite **Directed** Hamiltonian Cycle problem, we are given a bipartite directed graph and asked whether there is a simple cycle which visits every node exactly once. Note that this problem might potentially be easier than Hamiltonian Cycle because it assumes a directed bipartite graph. Prove that Bipartite Directed Hamiltonian Cycle is in fact still NP-Complete.

Solution:

- i) This problem is NP. Given a candidate path, we just check the nodes along the given path one by one to see if every node in the network is visited exactly once and if each consecutive node pair along the path are joined by an edge. (3)
- ii) To prove the problem is NP-complete, we reduce Directed Hamiltonian Cycle (DHC) problem to Bipartite Directed Hamiltonian Cycle (BDHC) problem. (2)

Given an arbitrary directed graph G , we split each vertex v in G into two vertex v_{in} and v_{out} . Here v_{in} connects all the incoming edges to v in G ; v_{out} connects all the outgoing edges from v in G . Moreover, we connect one directed edge from v_{in} to v_{out} . After doing these operations for each node in G , we form a new graph G' . (3)

Here G' is bipartite graph, because we can color each v_{in} “blue” and each v_{out} “red” without any coloring conflict.

If there is DHC in G , then there is a BDHC in G' . We can replace each node v on the DHC in G into consecutive nodes v_{in} and v_{out} , and (v_{in}, v_{out}) is an edge in G' . Then the new path is a BDHC in G' . (3)

On the other hand, if there is BDHC in G' , then there is a DHC in G . Note that if v_{in} is on the BDHC, v_{out} must be the successive node on the BDHC. Then we can merge each node pair (v_{in}, v_{out}) on the BDHC in G' into node v and form a DHC in G . (3)

In sum, G has DHC if and only if G' has a BDHC. This completes the reduction, and we confirm that the given problem is NP-complete

Alternate solution:

We can also use Undirected Hamiltonian Cycle for the reduction.

Prove that it's NP as before. (3)

Note that you are using Bipartite Undirected Hamiltonian Cycle for reduction. (2)

Given an arbitrary undirected bipartite graph G , convert G to G' so the vertices in G' stay the same, but all undirected edges are converted to directed edges in **both directions**. G' remains bipartite. (3)

If there is an undirected Hamiltonian cycle in G, you can use the corresponding edges(in either direction) to find a Directed Hamiltonian cycle in in G'. (3)

On the other hand, if there is a directed Hamiltonian cycle in G', you could convert the corresponding edges to undirected edges and find the equivalent cycle in G. (3)

Common mistakes:

Major mistake in checking for NP: -2

Checking for edges instead of vertices: -1

Missing the NP check altogether: -3

Mistake in the conversion step: -2

Missing the two-way proof: -4

Missing either side of the proof: -3

Incomplete explanation as to why the proof holds: -2

If you are proposing to reduce a certain NP-Complete problem, but are going with a solution corresponding a different NP-Complete problem, this shows misunderstanding of the concept: -7

If you are proposing to reduce an incorrect (and irrelevant) NP-complete problem: -8

4) 20 pts.

Suppose you want to sell n roses. You need to group the roses into multiple bouquets with different sizes. The value of each bouquet depends on the number of roses you used. In other words, we know that a bouquet of size i will sell for $v[i]$ dollars. Provide an algorithm to find the maximum possible value of the roses by deciding how to group them into different-sized bouquets.

Here is an example:

Input: $n = 5$, $v[1..n] = [2,3,7,8,10]$ (i.e. a bouquet of size 1 is \$2, bouquet of size 2 is \$3, ..., and finally a single bouquet of size $n=5$ is \$10.

Expected Output = 11 (by grouping the roses into bouquets of size 1, 1, and 3)

a) Define (in plain English) subproblems to be solved. (4 pts)

OPT(j) is the maximum value of j roses.

b) Write the recurrence relation for subproblems. (6 pts)

OPT(j) = MAX (OPT(j), {OPT(j-i) + v[i]} for all sizes $0 \leq i \leq j$)

i.If iteration of i from 0 to j is missing => -1 points

ii.If OPT(j) is missing in the MAX term => -1 points

iii.0 points for wrong recursion

c) Using the recurrence formula in part b, write pseudocode to compute the maximum possible value of the n roses. (6 pts)

Make sure you have initial values properly assigned. (2 pts)

OPT[0] = 0

OPT[1] = v[1]

For (int j=2; j < n; j ++)

 max = 0

 For (int i=1; i < j; i++)

 If (i<=j && max < OPT[j-i]+v[i])

 max = OPT[j-i]+v[i]

OPT[j]=max

i) Missing one loop => -1 point

ii) Missing two loops => -2 points

iii) If in b) you got x points, you get x points in the pseudo code. Apart from the case where you missed the iteration in b, where you will get x+1 points in this question

iv) 0 points for no pseudocode or any modification from b.

d) Compute the runtime of the algorithm described in part c and state whether your solution runs in polynomial time or not (2 pts)

e) d) O(n^2)

5) 10 pts

Give a tight asymptotic upper bound (O notation) on the solution to each of the following recurrences. You need not justify your answers.

$$T(n) = 2T(n/8) + n$$

Solution: $(n^{1/3} \lg n)$ by Case 2 of the Master Method.

$$T(n) = T(n/3) + T(n/4) + 5n$$

Solution: Use brute force method

$$\begin{aligned} T(n) &= cn + (7/12) cn + (7/12)^2 cn + \dots \\ &= (n) \end{aligned}$$

Rubric:

No partial marks. 5 points for each correct answer.

6) 16 pts

There are n people and m jobs. You are given a payoff matrix, C , where C_{ij} represents the payoff for assigning person i to do job j . Each applicant has a subset of jobs that he/she is interested in. Each job opening can only accept one applicant and a job applicant can be appointed for only one job. You need to find an assignment of jobs to applicants that maximizes the total payoff of your assignment. Write an **Integer Linear Program** (with discrete variables) to solve this problem.

Solution

We are looking for a perfect matching on a bipartite graph of the minimal cost.

Let $x_{ij} = \begin{cases} 1, & \text{if edge}(i,j) \text{ is in matching} \\ 0, & \text{otherwise} \end{cases}$

Objective: $\sum_{\substack{1 \leq i \leq m \\ 1 \leq j \leq n}} c_{ij} \cdot x_{ij}$

Subject to:

1. $\sum_j x_{ij} \leq 1$, where $i = 1, 2, \dots, n$
2. $\sum_i x_{ij} \leq 1$, where $j = 1, 2, \dots, n$
3. $x_{ij} \in \{0,1\}$, $\forall i, j$
4. $x_{ij} = 0$, for all jobs j that applicant i is not interested in.

Rubric:

- 4 if objective function is incorrect.
- 6 if you mention equal to instead of less than equal to sign for constraints 1 and 2
- 2 if constraint 3. is not mentioned, but only is defined to take values 0 or 1.
- 2 if discrete variable is not defined.
- 0 points awarded if you use a max-flow, min-cut formulation

7) 12 pts

The Maximum Acyclic Subgraph is stated as follows: Given a directed graph find an acyclic subgraph of that contains as many arcs (i.e. directed edges) as possible.

Give a 2-approximation algorithm for this problem.

Hint: Consider using an arbitrary ordering of the vertices in G.

Solution:

1. Pick an arbitrary vertex ordering. This partitions the arcs into two acyclic subgraphs and . Here A_1 is the set of arcs where $i < j$ and A_2 is the set of arcs where $i > j$. Thus at least one of A_1 or A_2 contains half the arcs of G . The maximum acyclic subgraph contains at most the total number of arcs in G , so we have a factor 2-approximation algorithm (12)
2. Divide the set of nodes into two nonempty sets, A and B . Consider the set of edges from A to B and the set of edges from B to A . Throw out the edges from the smaller set and keep the edges from the larger set (break ties arbitrarily). Recursively perform the algorithm on A and B individually (12)

Rubric:

If you are only considering the forward or backward edges (-2)

If you start with one node and add the forward/backward edges to/from neighbors, your answer won't work for disconnected graphs (-2)

If your final answer is not 0.5 approximation (-6)

If your answer is not acyclic (-6)

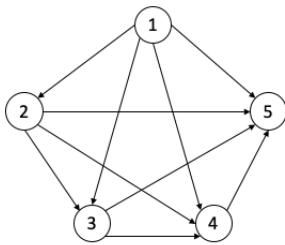
If your algorithm is vague and not implementable (for example if you said, "we find max independent set", or "we find the topological order") (0)

If your algorithm has conceptual flaws (for example if you are adding an edge to the sub graph and also deleting it from the sub graph) (0 pts)

Some of the answers that do not work:

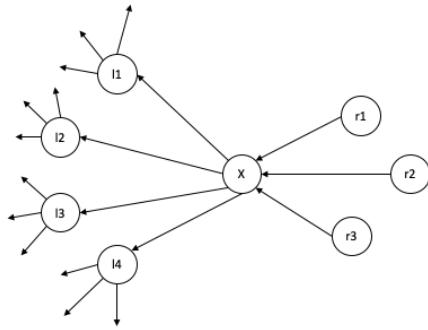
1. If your final answer is a tree (a DFS or a BFS or any other kinds of trees):

You can imagine an acyclic complete directed graph such as below. Your algorithm should return at least half of the edges ($n * (n-1) / 4$) but a tree can have $n-1$ edges. Thus it is not 0.5 approximation (6 pts).



2. If you start adding/removing edges to/from the graph, until it is acyclic. Your algorithm can choose an edge that is repeated in many cycles and end up removing all the other edges in those cycles.

You can try your method on the following graph (All $l(i)$ nodes are connected to all $r(j)$ nodes). If your algorithm chooses all the $(r(j), X)$ and $(X, l(i))$ edges (7 edges), you cannot add any $(l(i), r(j))$ edges (12 edges). Your final answer (7) is less than half of the best answer ($16/2 = 8$) so it is not 0.5 approximation.



1) 20 pts

Mark the following statements as **TRUE** or **FALSE**. No need to provide any justification.

[**TRUE**]

If $\text{SAT} \leq_P A$, then A is NP-hard.

[**FALSE**]

If a problem X can be reduced to a known NP-hard problem, then X must be NP-hard.

[**TRUE**]

If P equals NP, then NP equals NP-complete.

[**FALSE**]

Let X be a decision problem. If we prove that X is in the class NP and give a poly-time reduction from X to Hamiltonian Cycle, we can conclude that X is NP-complete.

[**TRUE**]

The recurrence $T(n) = 2T(n/2) + 3n$, has solution $T(n) = \theta(n \log(n^2))$.

[**FALSE**]

On a connected, directed graph with only positive edge weights, Bellman-Ford runs asymptotically as fast as Dijkstra.

[**TRUE**]

Linear programming is at least as hard as the Max Flow problem in a flow network.

[**TRUE**]

If you are given a maximum s-t flow in a graph then you can find a minimum s-t cut in time $O(m)$ where m is the number of the edges in the graph.

[**TRUE**]

Fibonacci heaps can be used to make Dijkstra's algorithm run in $O(|E| + |V| \log|V|)$ time on a graph $G=(V,E)$

[**FALSE**]

A graph with non-unique edge weights will have at least two minimum spanning trees

2) 16 pts

Given a graph $G=(V, E)$ with an even number of vertices as the input, the HALF-IS problem is to decide if G has an independent set of size $|V|/2$. Prove that HALF-IS is in NP-Complete.

Solution:

Given a graph $G(V, E)$ and a certifier $S \subset V$, $|S| = |V|/2$, we can verify if no two nodes are adjacent in polynomial time ($O(|S|^2) = O(|V|^2)$). Therefore $\text{HALF-IS} \in NP$.

We prove the NP-Hardness using a reduction of the NP-complete problem Independent set problem (IS) to HALF-IS . Consider an instance of IS , which asks for an independent set $A \subset V$, $|A| = k$, for a graph $G(V, E)$, such that no two pair of vertices in A are adjacent to each other.

- (i) If $k = \frac{|V|}{2}$, IS reduces to HALF-IS.
- (ii) If $k < \frac{|V|}{2}$, then add m new nodes such that $k + m = (|V| + m)/2$, i.e., $m = |V| - 2k$. Note that the modified set of nodes V' has even number of nodes. Since the additional nodes are all disconnected from each other, they form a subset of independent set. Therefore, the new graph $G'(V', E')$ where $E' = E$ has an independent-set of size $\frac{|V'|}{2}$ if and only if $G(V, E)$ has an independent set of size k .
- (iii) If $k > \frac{|V|}{2}$, then again add $m = |V| - 2k$ new nodes to form the modified set of nodes V' . Connect these new nodes to all the other $|V| + m - 1$ nodes. Since these m new nodes are connected to every other node of them should belong to an independent set. . Therefore, the new graph $G'(V', E')$ has an independent-set of size $\frac{|V'|}{2}$ if and only if $G(V, E)$ has an independent set of size k .

Hence, any instance of IS $(G(V, E), k)$, can be reduced to an instance of HALF-IS $(G'(V', E'))$.

$$IS \leq_P HALF-IS.$$

$\text{HALF-IS} \in NP$ and $\text{HALF-IS} \in NP\text{-Hard} \Rightarrow \text{HALF-IS} \in NP\text{-complete}$.

3) 16 pts

A variant on the decision version of the subset sum problem is as follows: Given a set of n integer numbers $A = \{a_1, a_2, \dots, a_n\}$ and a target number t . Determine if there is a subset of numbers in A whose product is precisely t . That is, the output is *yes* or *no*. Describe an algorithm (and provide pseudo-code) to solve this problem, and analyze its complexity.

Solution:

Solution: *Use dynamic programming:*

Let $S[i,j]$ shows if there exists a subset of $\{a_1, a_2, \dots, a_i\}$ that add up to j , where $0 \leq j \leq t$.

$S[i,j]$ is true or false.

Initialize: $S[0,0] = \text{true}$; $S[0,j] = \text{false}$ if $j \neq 0$

Recurrence formula:

$S[i,j] = S[i-1,j] \text{ OR } S[i-1, j/a_i]$

Output is $S[n,t]$

Complexity is $O(tn)$

4) 16 pts

We've been put in charge of a phone hotline. We need to make sure that it's staffed by at least one volunteer at all times. Suppose we need to design a schedule that makes sure the hotline is staffed in the time interval $[0, h]$. Each volunteer i gives us an interval $[s_i, f_i]$ during which he or she is willing to work. We'd like to design an algorithm which determines the minimum number of volunteers needed to keep the hotline running. Design an efficient greedy algorithm for this problem that runs in time $O(n \log n)$ if there are n student volunteers. Prove that your algorithm is correct. You may assume that any time instance has at least one student who is willing to work for that time.

Solution:

Algorithm: Initially, select the student who can start at or before time 0 and whose finish time is the latest. For each subsequent volunteer, select the one whose start time is no later than the finish time of the last selected volunteer and whose finish time is the latest. Keep selecting volunteers sequentially in this way until the interval $[0, h]$ is covered.

The running time is $O(n \log n)$:

First, sort the start time of all the volunteers takes time $O(n \log n)$; then, searching and selecting the valid successive volunteers with the latest finish time takes $O(n)$ time in total (each volunteer is checked at most once).

Proof:

Let g_1, g_2, \dots, g_m be the sequence of volunteers we selected according to the greedy algorithm; let p_1, p_2, \dots, p_k be the sequence of selected volunteers of an optimal solution.

Clearly, for any feasible solution, we must have someone who can starts before or at time 0. So in the above two solutions: g_1 and p_1 must start at or before time 0.

According to our algorithm, we have $f_{g1} \geq f_{p1}$. By replacing p_1 by g_1 in the optimal solution, we get another solution: g_1, p_2, \dots, p_k , which uses k volunteers and cover the interval $[0, h]$ and therefore is another optimal solution.

Induction hypothesis: assume that our greedy solution is the same as an optimal solution up to the $r-1$ th selected volunteer, i.e., $g_1, \dots, g_{r-1}, p_r, \dots, p_k$ is an optimal solution. With the same argument: $f_{g_r} \geq f_{p_r}$ by replacing p_r by g_r in the optimal solution, we get another solution $g_1, \dots, g_r, p_{r+1}, \dots, p_k$, which is also optimal.

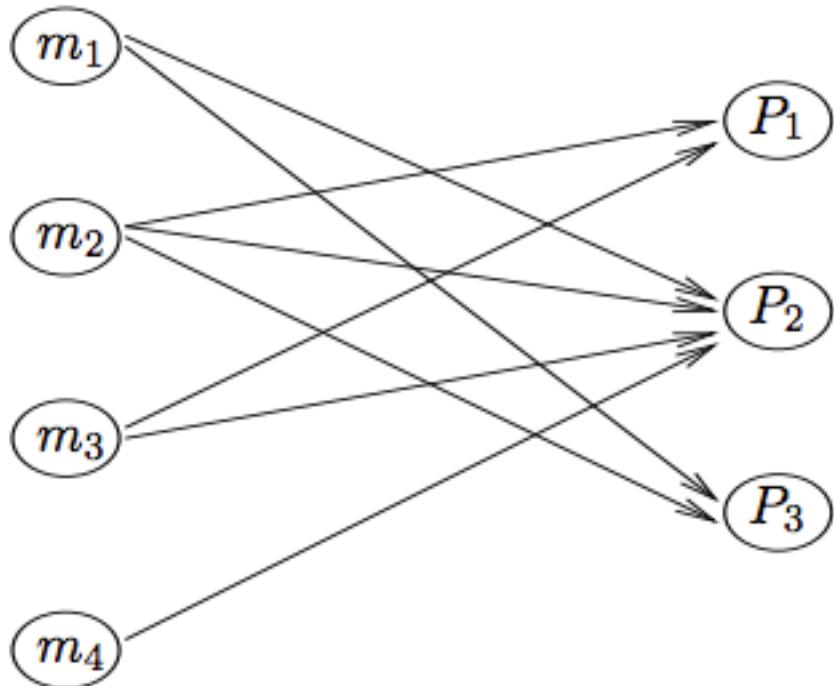
By induction: it follows that g_1, g_2, \dots, g_m is an optimal solution and $m=k$.

5) 16 pts

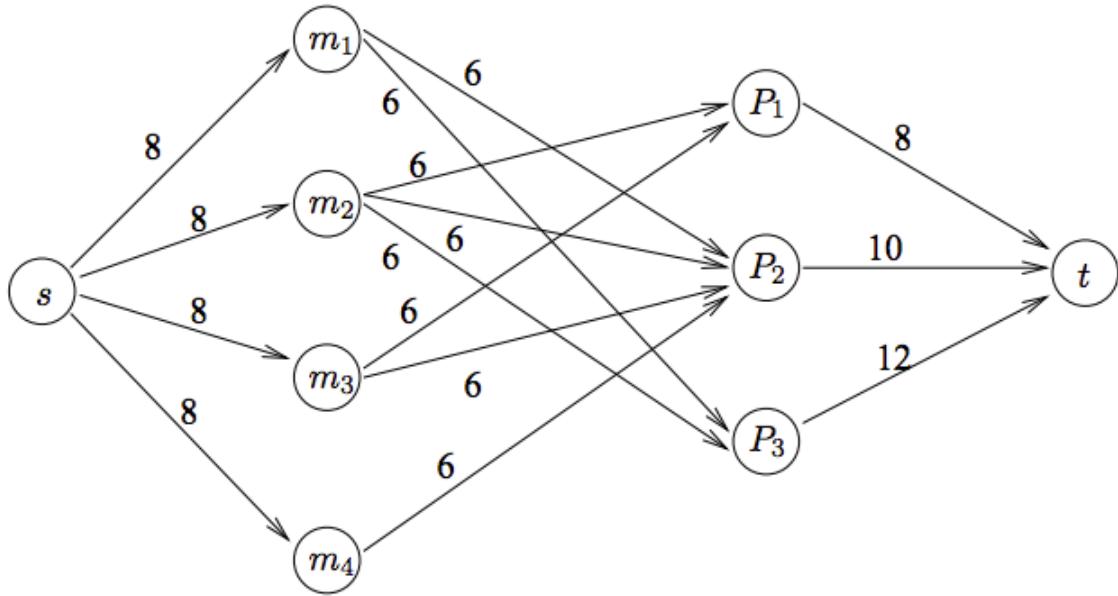
A software house has to handle 3 projects, P_1, P_2, P_3 , over the next 4 months. P_1 can only begin after month 1, and must be completed within month 3. P_2 and P_3 can begin at month 1, and must be completed, respectively, within month 4 and 2. The projects require, respectively, 8, 10, and 12 man-months. For each month, 8 engineers are available. Due to the internal structure of the company, at most 6 engineers can be working, at the same time, on the same project. Determine whether it is possible to complete the projects within the time constraints. Describe how to reduce this problem to the problem of finding a maximum flow in a flow network and justify your reduction.

Solution

We build a product network where months and projects are represented by, respectively, month-nodes m_1, m_2, m_3, m_4 and project-nodes P_1, P_2, P_3 . Each (i, j) edge denotes the possibility of allocating man-hours of month i to project j . For instance, since project P_1 can only begin after month 1 and must be completed before month 3, only arcs outgoing from m_2, m_3 are incident in P_1 .



We add to super nodes s, t , denoting the source and sink of the flow that represents the allocation of men-hours.



All edges outgoing from s have capacity 8, equivalent to the number of available engineers per month. All edges connecting month-nodes to project-nodes have capacity 6, as no more than 6 engineers can work on the same project in the same month. All edges incident in t have capacity equivalent to the number of man-months needed to complete the project. Since all capacities are integer, the maximum flow will be integer as well. To check whether all projects can be completed within the time limits, it suffices to check whether the network admits a feasible flow of value $8 + 10 + 12 = 30$.

6) 16 pts

There are n people and n jobs. You are given a cost matrix, C, where C[i][j] represents the cost of assigning person i to do job j. You want to assign all the jobs to people and also only one job to a person. You also need to minimize the total cost of your assignment. Can this problem be formulated as a linear program? If yes, give the linear programming formulation. If no, describe why it cannot be formulated as an LP and show how it can be reduced to an integer program.

Solution:

We need a 0,1 decision variable to solve the problem and therefore we need to formulate this as an integer program. Below is a formulation of integer program.

Let $x_{ij} = 1$, if job j is assigned to worker i.
= 0, if job j is not assigned to worker i.

Objective function: Minimize

$$\sum_{i=1}^m \sum_{j=1}^n c_{ij} x_{ij}$$

Constraints:

$$\sum_{i=1}^m x_{ij} = 1, \text{ for } j = 1, 2, \dots, n$$

$$\sum_{j=1}^n x_{ij} = 1, \text{ for } i = 1, 2, \dots, m$$

$$x_{ij} = 0 \text{ or } 1$$

1) 20 pts

Mark the following statements as **TRUE** or **FALSE**. No need to provide any justification.

[**TRUE**]

To prove that a problem X is NP-hard, it is sufficient to prove that SAT is polynomial time reducible to X.

[**FALSE**]

If a problem Y is polynomial time reducible to X, then a problem X is polynomial time reducible to Y.

[**TRUE**]

Every problem in NP can be solved in polynomial time by a nondeterministic Turing machine.

[**TRUE**]

Suppose that a divide and conquer algorithm reduces an instance of size n into 4 instances of size $n/5$ and spends $\Theta(n)$ time in the conquer steps. The algorithm runs in $\Theta(n)$ time.

[**FALSE**]

A linear program with all integer coefficients and constants must have an integer optimum solution.

[**FALSE**]

Let M be a spanning tree of a weighted graph $G=(V, E)$. The path in M between any two vertices must be a shortest path in G.

[**TRUE**]

A linear program can have an infinite number of optimal solutions.

[**TRUE**]

Suppose that a Las Vegas algorithm has expected running time $\Theta(n)$ on inputs of size n . Then there may still be an input on which it runs in time $\Omega(n^2)$.

[**FALSE**]

The total amortized cost of a sequence of n operations gives a lower bound on the total actual cost of the sequence.

[**FALSE**]

The maximum flow problem can be efficiently solved by dynamic programming.

2) 15 pts

Consider a uniform hash function h that evenly distributes n integer keys $\{0, 1, 2, \dots, n-1\}$ among m buckets, where $m < n$. Several keys may be mapped into the same bucket. The uniform distribution formally means that $\Pr(h(x)=h(y))=1/m$ for any $x, y \in \{0, 1, 2, \dots, n-1\}$. What is the expected number of total collisions? Namely how many distinct keys x and y do we have such that $h(x) = h(y)$?

Solution:

Let the indicator variable X_{ij} be 1 if $h(k_i) = h(k_j)$ and 0 otherwise for all $i \neq j$. Then let X be a random variable representing the total number of collisions. It can be calculated as the sum of all the X_{ij} 's: $X = \sum_{i < j} X_{ij}$

Now we take the expectation and use linearity of expectation to get:

$$E[X] = E \left[\sum_{i < j} X_{ij} \right] = \sum_{i < j} E[X_{ij}] = \sum_{i < j} P(h(k_i) = h(k_j)) = \sum_{i < j} \frac{1}{m} = \frac{n(n-1)}{2m}$$

Rubrics:

Describe choosing 2 keys from n keys: 10 pts.

Describe $\sum_{i < j} P(h(k_i) = h(k_j))$: 10 pts.

Common mistakes:

1. n/m : The mistake is that it thinks the expected number of collisions for each key is $1/m$. 7pts.
2. $2n/m$: Similar to 1. 7pts.
3. $(n-1)/m$: Similar to 1. 7pts.
4. $(m+1)/2$: Basically, no clue. 3 pts.
5. $n(n-1)/m$: Duplicates. 13 pts.
6. n^2/m : duplicated. 13 pts
7. $(1-(1-1/m)^{(n-1)}) * n$: wrong approach. 5 pts.
8. $n-m$: wrong approach. 3pts.
9. $\frac{1}{m} \sum_{i=0}^{n-1} i$: 15 pts
10. $n(n-1)/2$: The mistake is that it does not consider collision probability. 10 pts.
11. $N(n+1)/2m$: minor mistake. 13 pts.

3) 15 pts.

There are 4 production plants for making cars. Each plant works a little differently in terms of labor needed, materials, and pollution produced per car:

	Labor	Materials	Pollution
Plant 1	2	3	15
Plant 2	3	4	10
Plant 3	4	5	9
Plant 4	5	6	7

The goal is to maximize the number of cars produced under the following constraints:

- There are at most 3300 hours of labor
- There are at most 4000 units of material available.
- The level of pollution should not exceed 12000 units.
- Plant 3 must produce at least 400 cars.

Formulate a linear programming problem, using minimal number of variables, to solve the above task of maximizing the number of cars.

Solution:

We need four variables, to formulate the LP problem: x_1, x_2, x_3, x_4 , where x_i denotes the number of cars at plant-i.

Maximize $x_1 + x_2 + x_3 + x_4$

s.t.

$$x_i \geq 0 \text{ for all } i$$

$$x_3 \geq 400$$

$$2x_1 + 3x_2 + 4x_3 + 5x_4 \leq 3300$$

$$3x_1 + 4x_2 + 5x_3 + 6x_4 \leq 4000$$

$$15x_1 + 10x_2 + 9x_3 + 7x_4 \leq 12000$$

Rubric:

- Identifying 4 variables (2 pts)
- Correct objective function (3 pts)
- Each condition (2 pts)

4) 15 pts.

Given an undirected connected graph $G = (V, E)$ in which a certain number of tokens $t(v) \geq 1$ placed on each vertex v . You will now play the following game. You pick a vertex u that contains at least two tokens, remove two tokens from u and add one token to any one of adjacent vertices. The objective of the game is to perform a sequence of moves such that you are left with exactly one token in the whole graph. You are not allowed to pick a vertex with 0 or 1 token. Prove that the problem of finding such a sequence of moves is NP-complete by reduction from Hamiltonian Path.

Solution:

Construction: given a HP in G, we construct G' as follows. Traverse a HP in G and placed 2 tokens on the starting vertex and one token on each other vertex in the path.

Claim: G has a HP iff G' has a winning sequence.

\Rightarrow) by construction before the last move we will end up with a single vertex having two tokens on it. Making the last move, we will have exactly one token on the board.

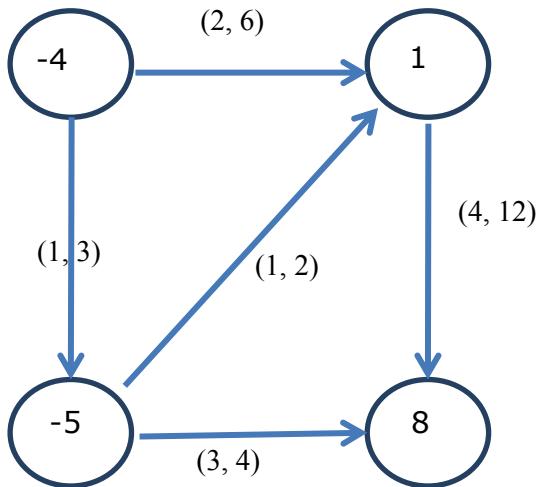
\Leftarrow) since there is only one vertex with 2 tokens, we will start right there playing the game. Each next move is forced. When we finish the game, we get a sequence moves which represents a HP.

Rubrics:

- Didn't prove it is in NP: -5
- Didn't prove it is NP-hard: -10
- Assigned two tokens on one random vertex instead of the starting vertex of Hamiltonian path: -2 (Since it is a reduction from Hamiltonian path, not from Hamiltonian cycle, 2 tokens should be assigned on the starting vertex)
- Assigned wrong number of tokens on one vertex: -3
- Assigned wrong number of tokens on two vertices: -6
- Assigned wrong number of tokens on three or more vertices (considered as not a valid reduction from Hamiltonian path): -7
- Not a valid reduction from Hamiltonian path: -7

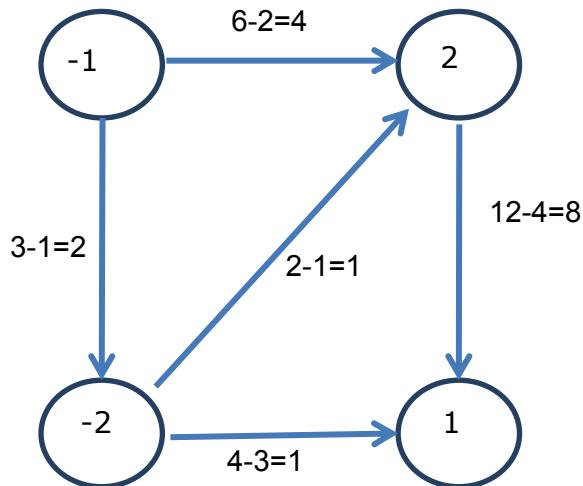
5) 15 pts.

In the network below, the demand values are shown on vertices (supply value if negative). Lower bounds on flow and edge capacities are shown as (lower bound, capacity) for each edge. Determine if there is a feasible circulation in this graph. You need to show all your steps.



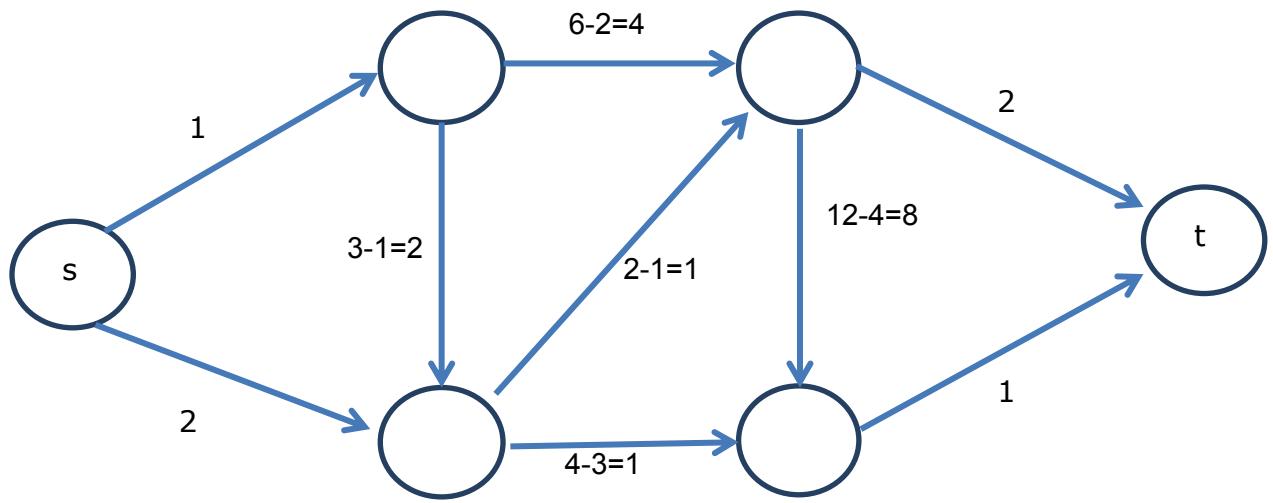
- a) Turn the circulation with lower bounds problem into a circulation problem without lower bounds (6 pts)

- **Each wrong number for nodes: -1**
- **Each wrong number for edges: -0.5**



b) Turn the circulation with demands problem into the max-flow problem (5 pts)

- **s and t nodes on right places: each has 0.5 point**
- **directions of edges from s and t: each direction 0.5 point**
- **values of edges from s and t: each value 0.5 point**

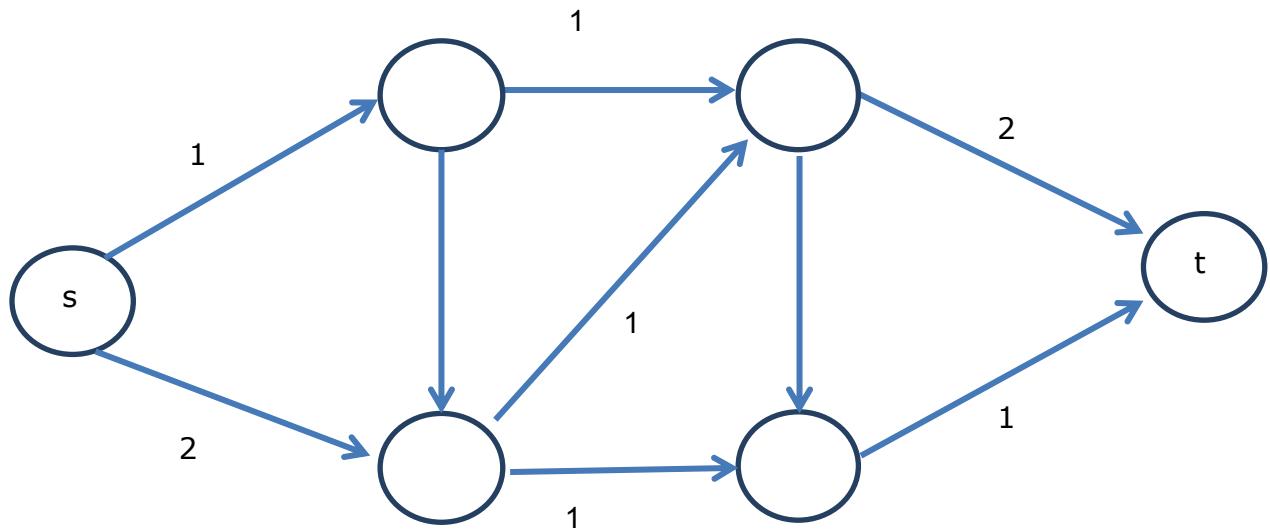


c) Does a feasible circulation exist? Explain your answer. (4 pts)

Solution.

Yes. There is a s-t flow $|f|=2+1=1+2=3$. It follows, there is a feasible circulation.

- **Explanation: 2 points.**
- **Correct Graph/flow: 2 points.**



6) 10 pts.

We wish to determine the most efficient way to deliver power to a network of cities.

Initially, we have n cities that are all connected. It costs $c_i \geq 0$ to open up a power plant at city i . It costs $r_{ij} \geq 0$ to build a cable between cities i and j . A city is said to have power if either it has a power plant, or it is connected by a series of cables to some other city with a power plant (in other words, the cables act as undirected edges). Devise an efficient algorithm for finding the *minimum cost* to power all the cities.

Solution

Consider a graph with the cities at the vertices and with weight r_{ij} on the (undirected) edge between city i and city j . Add a new vertex s and add an edge with weight c_i from s to the i th city, for each i . The minimum spanning tree on this graph gives the best solution (power stations need to be opened at those cities i for which (s,i) is part of the MST.)

Common Mistakes:

- Dynamic Programming: There does not exist a predefined order in which the nodes/edges will be processed and because of this you cannot define the problem based on smaller sub-problems and hence, none of the solutions which were based on Dynamic Programming were correct.
- Max-Flow/Min Cut: It was a surprise to see many ran a Max Flow algorithm to minimize the overall cost of edges. The minimum cut in a network has nothing to do with minimizing flow!! Furthermore, in this problem we want connectivity in the graph and edges on the min cut don't guarantee any level of connectivity.
- ILP: I can't recall anybody correctly formalizing the problem as in ILP. However, even a correct formulation is going to be NP-Hard and far from efficient. A maximum of 3 points was given to a CORRECT ILP solution.
- Dijkstra's algorithm is for finding the shortest path between a single point and every other node in the graph. The resulting Tree from running Dijkstra, is not necessarily an MST so it's not the correct approach to solve this problem.

7) 10 pts.

We want to break up the graph $G = (V, E)$ into two disjoint sets of vertices $V = A \cap B$, such that the number of edges between two partitions is as large as possible. This problem is called a max-cut problem. Consider the following algorithm:

- Start with an arbitrary cut C .
- While there exists a vertex v such that moving v from A to B increases the number of edges crossing cut C , move v to B and update C .

Prove that the algorithm is a 2-approximation.

Hint: after algorithm termination the number of edges in each of two partitions A and B cannot exceed the number of edges on the cut C .

Solution

Let $w(u, v) = 1$, there exists an edge between u and v , and 0 otherwise. Let

$$W = \sum_{(u,v) \in E} w(u, v)$$

By construction, for any node $u \in A$:

$$\sum_{v \in A} w(u, v) \leq \sum_{v \in B} w(u, v)$$

Here, the left hand side is all edges in partition A . The RHS – the crossing edges. If we sum up the above inequality for all $u \in A$, the LHS will contain the twice number of edges in A .

$$2 \sum_{(u,v) \in A} w(u, v) \leq \sum_{u \in A, v \in B} w(u, v) = C$$

We do the same for any node in B :

$$2 \sum_{(u,v) \in B} w(u, v) \leq \sum_{u \in A, v \in B} w(u, v) = C$$

Add them up

$$\sum_{(u,v) \in A} w(u, v) + \sum_{(u,v) \in B} w(u, v) \leq C$$

Next we add edges on the cut C to both sides.

$$\sum_{(u,v) \in A} w(u, v) + \sum_{(u,v) \in B} w(u, v) + \sum_{u \in A, v \in B} w(u, v) = W \leq 2C$$

Clearly the optimal solution $\text{OPT} \leq W$, thus $\text{OPT} \leq W \leq 2C$. It follows, $\text{OPT}/C \leq 2$.

A: # of edges within partition A
B: # of edges within partition B
C: # of edges between partition A and B

Correct direction

inCorrect direction

Prove that $A + B \leq C$

Prove that $A \leq C$ and $B \leq C$
Correct but useless, at most 1 points

Stage 1. 6 points

Prove that $\text{OPT} \leq 2*C$

Prove that $\text{OPT} \leq 3*C$
At most 2 points, if relations between A, B, C, E are used in reasonable way.

Stage 2. 4 points

- The solution can be divided into two parts. **Part 1** is worth 6 points, and **part 2** is worth 4 points.
 - Proving the hint, i.e. "numbers of edges in partition A and partition B cannot exceed the number of edges on the cut C". It is most important and challenging part of this problem.
 - Prove the algorithm is a 1/2-approximation.
- About part 1, here are two acceptable solutions and some solutions not acceptable:
 - If your solution is same to the official solution, using inequalities to rigorously prove the hint, it is perfect, **6 points**.
 - Here is a weaker version. A the end of algorithm, for any node u in the graph, without loss of generality assume it is in part A, the number of edges connecting it to nodes in part B (i.e. edges in the cut) is greater than or equal to the number of edges connecting it to other nodes in part A. So, the total number of edges in the cut is greater than or equal to the total number of edges within each part. This solution did not consider the important fact that both inter-partition and intra-partition edges are counted twice. **3 ~5 points according to the quality of statements.**
 - One incorrect proof: some answers declare that C is increasing and $A + B$ is decreasing when calling steps 2 of the algorithm. This is a correct but useless statement. **0 points**
 - YOU MUST PROVE THE HINT. SIMPLY STATING OR REPHRASING IT GET 0 POINTS IN THIS STEP.** Here is one example that cannot get credits: "According to the algorithm, we move one vertex from A to B if it can increase the number of edges in cut C. Then the number of edges in C is more than edges in partition A and partition B." **0 points**
 - "If move v from A to B can increase the number of edges cross cut C, it means v connect to more vertex in A than in B". It is rephrasing the algorithm, and failed to get the key point. If the above statement is changed to "If move v from A to B can increase the number of edges cross cut C, it means v connect to more vertex in different partition than in the same partition", it will get at least **3 points**.
- About part 2, the correct answer should use relation between OPT, E, A, B, C. Missing key inequalities like $\text{OPT} \leq E$, will lead to loss of grades.
- If the answer is incorrect, you may get some points from some steps that make sense. If the answer is on the incorrect direction in the above diagram:
 - Many students attempted to PROVE the $\text{edges_in_partition_A} \leq C$, and $\text{edges_in_partition_B} \leq C$, but not proving $\text{edges_in_partition_A} + \text{edges_in_partition_B} \leq C$. This is trivial according to step 2 of algorithm, and is not useful for proving the final statement. This will be treated as "correct but useless/irrelevant statements". **AT MOST 1 point.**
 - From above inequalities, $\text{OPT} \leq 3*C$ is a correct conclusion. It is different from the question, but since the logics in this stage is same, you can get at most **2 points out of 4**.

1) 20 pts

For each of the following statements, answer whether it is TRUE or FALSE, and briefly justify your answer.

- a) If a connected undirected graph G has the same weights for every edge, then every spanning tree of G is a minimum spanning tree, but such a spanning tree cannot be found in linear time.

T

- b) Given a flow network G and a maximum flow of G that has already been computed, one can compute a minimum cut of G in linear time.

F

- c) The Ford-Fulkerson Algorithm finds a maximum flow of a unit-capacity flow network with n vertices and m edges in time $O(mn)$ if one uses depth-first search to find an augmenting path in each iteration.

T

- d) Unless $P = NP$, 3-SAT has no polynomial-time algorithm.

F

- e) The problem of deciding whether a given flow f of a given flow network G is a maximum flow can be solved in linear time.

F

- f) If a decision problem A is polynomial-time reducible to a decision problem B (i.e., $A \leq_p B$), and B is NP-complete, then A must be NP-complete.

F

- g) If a decision problem B is polynomial-time reducible to a decision problem A (i.e., $B \leq_p A$), and B is NP-complete, then A must be NP-complete.

T

- h) Integer max flow (where flows and capacities are integers) is polynomial time reducible to linear programming .

F

- i) It has been proved that NP-complete problems cannot be solved in polynomial time.

F

- j) NP is a class of problems for which we do not have polynomial time solutions.

T

2) 16 pts

Suppose that we are given a weighted undirected graph $G = (V, E)$, a source $s \in V$, a sink $t \in V$, and a subset W of vertices V , and want to find a shortest path from s to t that must go through at least one vertex in W . Give an efficient algorithm that solves the problem by invoking any given single-source shortest path algorithm as a subroutine a small number (how many?) times. Show that your algorithm is correct.

Min (pathLength(s, w) + pathLength(w, t)) where w belongs to W

pathLength(i, j) finds the shortest path between i and j . It can be implemented using any existing shortest path algorithm. It will be used $2 * \text{size}(W)$ times

16 pts

Suppose that we have n files F_1, F_2, \dots, F_n of lengths l_1, l_2, \dots, l_n respectively, and want to concatenate them to produce a single file $F = F_1 \circ F_2 \circ \dots \circ F_n$ of length $l_1 + l_2 + \dots + l_n$. Suppose that the only basic operation available is one that concatenates two files. Moreover, the cost of this basic operation on two files of length l_i and l_j , is determined by a cost function $c(l_i, l_j)$ which depends only on the lengths of the two files. Design an efficient dynamic programming algorithm that given n files F_1, F_2, \dots, F_n and a cost function $c(\cdot, \cdot)$, computes a sequence of basic operations that optimizes total cost of concatenating the n input files.

The sub structure of this problem is the time to merge a set of files S . $\text{time}(S)$ – the time needed to merge the files and the files in S should be kept for repeated use

```
if size (S) == 2:  
    # S1 and S2 are the two files in S  
    time (S) = c (length(S1), length(S2))  
else:  
    # f is a file in S  
    # S-f : take f out from S  
    time (S) = min (c (length(f), total length of all other files in S)+time (S-f))
```

Return $\text{time}(S)$ where S contains all the files

4) 16 pts

Define the language

Double-SAT = { ψ : ψ is a Boolean formula with at least 2 distinct satisfying assignments }.

For instance, the formula $\psi: (x \vee y \vee z) \wedge (x' \vee y' \vee z') \wedge (x' \vee y' \vee z)$ is in Double-SAT, since the assignments $(x = 1, y = 0, z = 0)$ and $(x = 0, y = 1, z = 1)$ are two distinct assignments that both satisfy ψ .

Prove that Double-SAT is NP-Complete.

Various methods can be used for reducing SAT to Double-SAT. Since SAT is NP-Complete, Double-SAT is NP complete.

Following is an example for the reduction.

On input $\psi(x_1, \dots, x_n)$:

1. Introduce a new variable y .

2. Output formula $\psi'(x_1, \dots, x_n, y) = \psi(x_1, \dots, x_n) \wedge (y \mid \text{not } y)$.

If $\psi(x_1, \dots, x_n)$ belongs to SAT, then ψ' has at least 1 satisfying assignment, and therefore $\psi'(x_1, \dots, x_n, y)$ has at least 2 satisfying assignments as we can satisfy the new clause $(y \mid \text{not } y)$ by assigning either

$y = 1$ or $y = 0$ to the new variable y , so $\psi'(x_1, \dots, x_n, y)$ belongs to Double-SAT. On the other hand, if $\psi(x_1, \dots, x_n)$ does not belong to SAT, then clearly $\psi'(x_1, \dots, x_n, y) = \psi(x_1, \dots, x_n) \wedge (y \mid \text{not } y)$ has no satisfying assignment either, so $\psi'(x_1, \dots, x_n, y)$ does not belong to Double-SAT. Therefore, SAT can be reduced to Double-SAT. Since the above reduction clearly can be done in P time, Double-SAT is NP-Complete.

5) 16 pts

Let X be a set of n intervals on the real line. A subset of intervals $Y \subset X$ is called a tiling path if the intervals in Y cover the intervals in X , that is, any real value that is contained in some interval in X is also contained in some interval in Y . The size of a tiling cover is just the number of intervals.

Describe and analyze an algorithm to compute the smallest tiling path of X as quickly as possible. Assume that your input consists of two arrays $X_L[1 .. n]$ and $X_R [1 .. n]$, representing the left and right endpoints of the intervals in X .



A set of intervals. The seven shaded intervals form a tiling path

Sort the intervals from left to right by the start position, if the start positions are same, then sort it from the left to right by the end position;

```

FOR (i=1;i<=n; i++) do
    Result[i] = positive unlimited ;
    FOR (j = 1;j < i; j ++) do
        IF (interval i overlap with interval j) then
            IF (Result[i]> result[j]+1) then
                Result[i] = result[j]+1;
Return result [n]

```

The complexity is $O(n^2)$

6) 16 pts

An edge of a flow network is called *critical* if decreasing the capacity of this edge results in a decrease in the maximum flow. Give an efficient algorithm that finds a critical edge in a network.

Find a min cut of the network which has the same capacity as the maximum flow.
Every outgoing edge from the min cut is a critical edge

1) 10 pts

For each of the following sentences, state whether the sentence is known to be TRUE, known to be FALSE, or whether its truth value is still UNKNOWN.

- (a) If a problem is in P, it must also be in NP.
TRUE.
- (b) If a problem is in NP, it must also be in P.
UNKNOWN.
- (c) If a problem is NP-complete, it must also be in NP.
TRUE.
- (d) If a problem is NP-complete, it must not be in P.
UNKNOWN.
- (e) If a problem is not in P, it must be NP-complete.
FALSE.

If a problem is NP-complete, it must also be NP-hard.

TRUE.

If a problem is in NP, it must also be NP-hard.

FALSE.

If we find an efficient algorithm to solve the Vertex Cover problem we have proven that $P=NP$

TRUE.

If we find an efficient algorithm to solve the Vertex Cover problem with an approximation factor $\rho \geq 1$ (a single constant) then we have proven that $P=NP$

FALSE.

If we find an efficient algorithm that takes as input an approximation factor $\rho \geq 1$ and solves the Vertex Cover problem with that approximation factor, we have proven that $P=NP$.

TRUE.

2) 20 pts

Let $G = (V, E)$ be a weighted, directed graph with weight function $w : E \rightarrow \{0, 1, \dots, W\}$ for some nonnegative integer W . Modify Dijkstra's algorithm to compute the shortest paths from a given source vertex s in $O(WV + E)$ time.

Consider running Dijkstra's algorithm on a graph, where the weight function is $w : E \rightarrow \{1, \dots, W - 1\}$. To solve this efficiently, implement the priority queue by an array A of length $WV + 1$. Any node with shortest path estimate d is kept in a linked list at $A[d]$. $A[WV + 1]$ contains the nodes with ∞ as estimate.

EXTRACT-MIN is implemented by searching from the previous minimum shortest path estimate until a new is found. DECREASE-KEY simply moves vertices in the array. The EXTRACT-MIN operations takes a total of $O(VW)$ and the DECREASE-KEY operations take $O(E)$ time in total. Hence the running time of the modified algorithm will be $O(VW + E)$.

3) 15 pts

At a dance, we have n men and n women. The men have height $g(1), \dots, g(n)$, and women $h(1), \dots, h(n)$. For the dance, we want to match up men with women of roughly the same height. Here are the precise rules:

- a. Each man i is matched up with exactly one woman w_i , and each woman with exactly one man.
- b. For each couple (i, w_i) , the mismatch is height difference $|g(i)-h(w_i)|$.
- c. Our goal is to find a matching minimizing the maximum mismatch,
 $\text{MAX}_i |g(i)-h(w_i)|$.

Give an algorithm that runs in $O(n \log n)$ and achieves the desired matching. Provide proof of correctness.

We use an exchange argument. Let w_i denote the optimal solution. If there is any pair i, j such that $i < j$ in our ordering, but $w_i > w_j$ (also with respect to our ordering), then we evaluate the effect of switching to $w'_i := w_j, w'_j := w_i$. The mismatch of no other couple is affected. The couple including man i now has mismatch $|g(i) - h(w'_i)| = |g(i) - h(w_j)|$. Similarly, the other switched couple now has mismatch $|g(j) - h(w_i)|$.

We first look at man i , and distinguish two cases: if $h(w_j) \leq g(i)$ (i.e., man i is at least as tall as his new partner), then the sorting implies that $|g(i) - h(w_j)| = g(i) - h(w_j) \leq g(j) - h(w_j) = |g(j) - h(w_j)|$. On the other hand, if $h(w_j) > g(i)$, then $|g(i) - h(w_j)| = h(w_j) - g(i) \leq h(w_i) - g(i) = |h(w_i) - g(i)|$. In both cases, the mismatch between man i and his new partner is at most the previous maximum mismatch.

We use a similar argument for man j . If $h(w_i) \leq g(j)$, then $|g(j) - h(w_i)| = g(j) - h(w_i) \leq g(j) - h(w_j) = |g(j) - h(w_j)|$. On the other hand, if $h(w_i) > g(j)$, then $|g(j) - h(w_i)| = h(w_i) - g(j) \leq h(w_i) - g(i) = |h(w_i) - g(i)|$. Hence, the mismatch between j and his partner is also no larger than the previous maximum mismatch.

Hence, in all cases, we have that both of the new mismatches are bounded by the larger of the original mismatches. In particular, the maximum mismatch did not increase by the swap. By making such swaps while there are inversions, we gradually transform the optimum solution into ours. This proves that the solution found by the greedy algorithm is in fact optimal.

4) 20 pts

You are given integers p_0, p_1, \dots, p_n and matrices A_1, A_2, \dots, A_n where matrix A_i has dimension $(p_{i-1}) * p_i$

(a) Let $m(i, j)$ denote the minimum number of scalar multiplications needed to evaluate the matrix product $A_i A_{i+1} \dots A_j$. Write down a recursive algorithm to compute $m(i, j)$, $1 \leq i \leq j \leq n$, that runs in $O(n^3)$ time.

Hint: You need to consider the order in which you multiply the matrices together and find the optimal order of operations.

Initialize $m[i, j] = -1 \quad 1 \leq i \leq j \leq n$

Output $mcr(1, n)$

```
mcr(i, j) {  
    if m[i, j] >= 0 return m[i, j]  
    else if i == j m[i, j] = 0  
    else m[i, j] = min (i <= k < j) { mcr(i, k) + mcr(k+1, j) + P_{i-1}*P_k*P_k }  
    return m[i, j]  
}
```

(b) Use this algorithm to compute $m(1,4)$ for $p_0=2$, $p_1=5$, $p_2=3$, $p_3=6$, $p_4=4$

$m[i, j]$

i\j	2	3	4
1	30	66	114
2		90	132
3			72

$m[1, 4] = 114$

5) 20 pts

In a certain town, there are many clubs, and every adult belongs to at least one club. The townspeople would like to simplify their social life by disbanding as many clubs as possible, but they want to make sure that afterwards everyone will still belong to at least one club.

Prove that the Redundant Clubs problem is NP-complete.

First, we must show that Redundant Clubs is in NP, but this is easy: if we are given a set of K clubs, it is straightforward to check in polynomial time whether each person is a member of another club outside this set.

Next, we reduce from a known NP-complete problem, Set Cover. We translate inputs of Set Cover to inputs of Redundant Clubs, so we need to specify how each Redundant Clubs input element

is formed from the Set Cover instance. We use the Set Cover's elements as our translated list of people,

and make a list of clubs, one for each member of the Set Cover family. The members of each club are just the elements of the corresponding family. To finish specifying the Redundant Clubs input,

we need to say what K is: we let $K = F - K_{SC}$ where F is the number of families in the Set Cover instance and K_{SC} is the value K from the set cover instance. This translation can clearly be done in polynomial time (it just involves copying some lists and a single subtraction).

Finally, we need to show that the translation preserves truth values. If we have a yes-instance of Set Cover, that is, an instance with a cover consisting of K_{SC} subsets, the other K subsets form a solution to the translated Redundant Clubs problem, because each person belongs to a club in the

cover. Conversely, if we have K redundant clubs, the remaining K_{SC} clubs form a cover. So the answer

to the Set Cover instance is yes if and only if the answer to the translated Redundant Clubs instance
is yes.

6) 15 pts

A company makes two products (X and Y) using two machines (A and B). Each unit of X that is produced requires 50 minutes processing time on machine A and 30 minutes processing time on machine B. Each unit of Y that is produced requires 24 minutes processing time on machine A and 33 minutes processing time on machine B.

At the start of the current week there are 30 units of X and 90 units of Y in stock. Available processing time on machine A is forecast to be 40 hours and on machine B is forecast to be 35 hours.

The demand for X in the current week is forecast to be 75 units and for Y is forecast to be 95 units—these demands must be met. In addition, company policy is to maximize the combined sum of the units of X and the units of Y in stock at the end of the week.

- a. Formulate the problem of deciding how much of each product to make in the current week as a linear program.

Solution

Let

- x be the number of units of X produced in the current week
- y be the number of units of Y produced in the current week

then the constraints are:

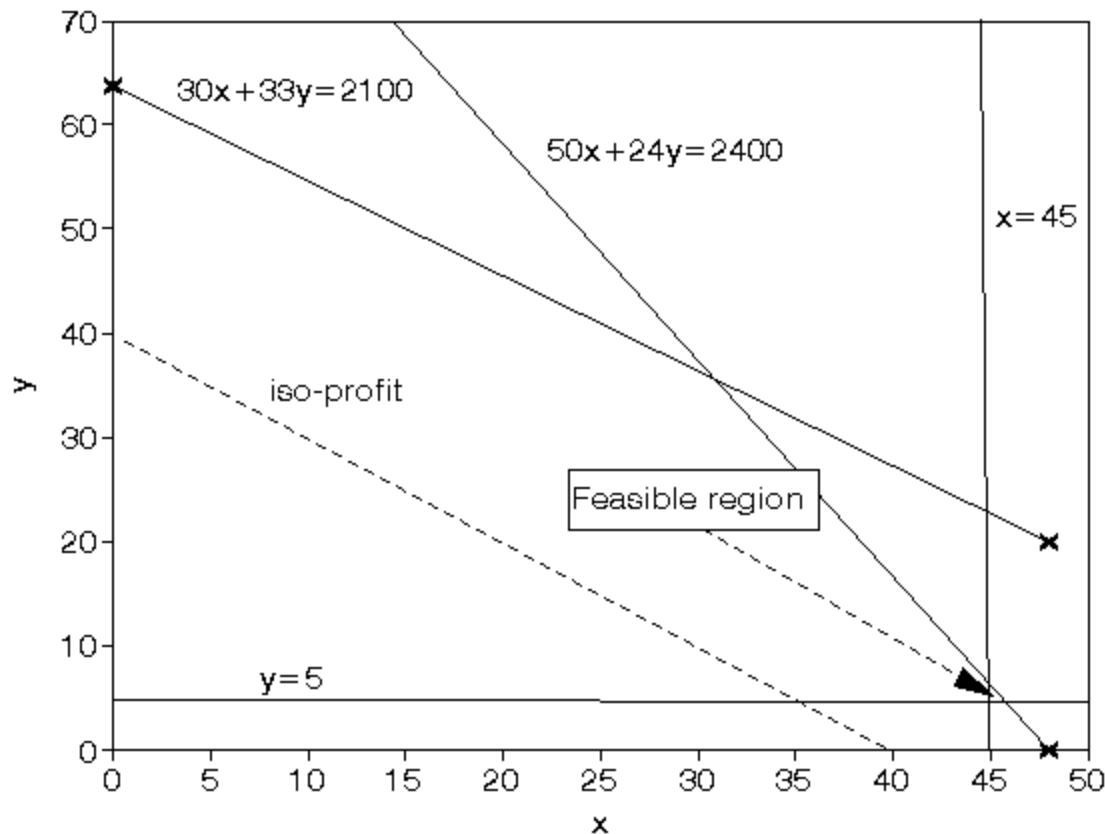
- $50x + 24y \leq 40(60)$ machine A time
- $30x + 33y \leq 35(60)$ machine B time
- $x \geq 75 - 30$
- i.e. $x \geq 45$ so production of X \geq demand (75) - initial stock (30), which ensures we meet demand
- $y \geq 95 - 90$
- i.e. $y \geq 5$ so production of Y \geq demand (95) - initial stock (90), which ensures we meet demand

The objective is: maximise $(x+30-75) + (y+90-95) = (x+y-50)$

i.e. to maximise the number of units left in stock at the end of the week

b. Solve this linear program graphically.

It can be seen in diagram below that the maximum occurs at the intersection of $x=45$ and $50x + 24y = 2400$



Solving simultaneously, rather than by reading values off the graph, we have that $x=45$ and $y=6.25$ with the value of the objective function being 1.25

Final Examination Solutions

Problem 1. Miscellaneous True/False [18 points] (6 parts)

For each of the following questions, circle either T (True) or F (False). **Explain your choice.** (No credit if no explanation given.)

- (a) **T F** If the load factor of a hash table is less than 1, then there are no collisions.
Explain:

Solution: False. The table contains fewer elements than it has slots, but that doesn't prevent elements from hashing to the same slot.

- (b) **T F** If $SAT \leq_P A$, then A is NP-hard.
Explain:

Solution: True. SAT is NP-complete. This follows from the definitions of NP-hard and NP-complete.

- (c) **T F** The longest common subsequence problem can be solved using an algorithm for finding the longest path in a weighted DAG.
Explain:

Solution: True. Shown in lecture.

- (d) **T F** Applying a Givens rotation to a matrix changes at most one row of the matrix.

Explain:

Solution: **False.** It changes two rows.

- (e) **T F** The problem of finding the shortest path from s to t in a directed, weighted graph exhibits optimal substructure.

Explain:

Solution: **True.** The shortest path to t can be easily computed given the shortest paths to all vertices from which there is an edge to t .

- (f) **T F** A single rotation is sufficient to restore the AVL invariant after an insertion into an AVL tree.

Explain:

Solution: **False.** Restoring the invariant may require $\Theta(\lg n)$ rotations.

Problem 2. More True/False [18 points] (6 parts)

For each of the following questions, circle either T (True) or F (False). **Explain your choice.** (No credit if no explanation given.)

- (a) **T F** Using hashing, we can create a sorting algorithm similar to COUNTING-SORT that sorts a set of n (unrestricted) integers in linear time. The algorithm works the same as COUNTING-SORT, except it uses the hash of each integer as the index into the counting sort table.

Explain:

Solution: False. Counting sort requires that the elements in its table are ordered according their indices into the table. Hashing the integers breaks this property

- (b) **T F** There exists a comparison-based algorithm to construct a BST from an unordered list of n elements in $O(n)$ time.

Explain:

Solution: False. Because we can create a sorted list from a BST in $O(n)$ time via an in-order walk, this would violate the $\Omega(n \lg n)$ lower bound on comparison-based sorting.

- (c) **T F** It is possible for a DFS on a directed graph with a positive number of edges to produce no tree edges.

Explain:

Solution: True. Consider a graph with two vertices, a and b , and a single edge (a, b) . Starting the DFS at b produces no tree edges. (When the search then starts at a , the only edge is a cross edge.)

- (d) **T F** A max-heap can support both the INCREASE-KEY and DECREASE-KEY operations in $\Theta(\lg n)$ time.

Explain:

Solution: True.

INCREASE-KEY: Change the node's key, then swap it with its parent until the heap invariant is restored.

DECREASE-KEY: Change the node's key, then MAX-HEAPIFY on the changed node.

- (e) **T F** In a top-down approach to dynamic programming, the larger subproblems are solved before the smaller ones.

Explain:

Solution: False. The larger problems depend on the smaller ones, so the smaller ones need to be solved first. The smaller problems get solved (and memoized) recursively as part of a larger problem.

- (f) **T F** Running a DFS on an undirected graph $G = (V, E)$ always produces the same number of cross edges, no matter what order the vertex list V is in and no matter what order the adjacency lists for each vertex are in.

Explain:

Solution: True. DFS in an undirected graph never produces cross edges.

Problem 3. Miscellaneous Short Answer [24 points] (4 parts)

You should be able to answer each question in no more than a few sentences.

- (a) Two binary search trees t_1 and t_2 are *equivalent* if they contain exactly the same elements. That is, for all $x \in t_1$, $x \in t_2$, and for all $y \in t_2$, $y \in t_1$. Devise an efficient algorithm to determine if BSTs t_1 and t_2 are equivalent. What is the running time of your algorithm, assuming $|t_1| = |t_2| = n$?

Solution: One solution is to read the elements of each tree into a sorted list using an in-order walk, and compare the resulting lists for equality. This algorithm runs in $\Theta(n)$.

Another $\Theta(n)$ solution is to put all of the elements of each tree into a hash table, and compare the resulting tables.

- (b) You are given a very long n -letter string, S , containing transcripts of phone calls obtained from a wiretap of a prominent politician. Describe, at a high level, an efficient algorithm for finding the 4-letter substring that appears most frequently within S .

Solution: Iterate through S , hashing each four-letter substring and keeping track of frequencies in a hash table. Either a normal hash or a rolling hash works here; each requires $\Theta(n)$ time to process the entire string ($\Theta(1)$ per substring).

- (c) A *word ladder* is a sequence of words such that each word is an English word, and each pair of consecutive words differs by replacing exactly one letter with another. Here is a simple example:

TREE, FREE, FRET, FRAT, FEAT, HEAT, HEAP

Assume that you are supplied with a function $\text{GET-WORDS}(w)$ that returns, in $\Theta(1)$ time, a list of all English words that differ from w by exactly one letter (perhaps using a preprocessed lookup table). Describe an efficient algorithm that finds a word ladder, beginning at w_1 and ending at w_2 , of minimum length.

Solution: The GET-WORDS function defines the edges in an implicit graph where words are vertices. Use a (bidirectional) BFS on this graph.

- (d) What procedure would you use to find a longest path from a given vertex s to a given vertex t in a weighted directed acyclic graph, when negative edge weights are present?

Solution: Dynamic programming. The fact that some edges have negative edge weights doesn't affect anything.

Problem 4. Changing Colors [15 points]

Consider a directed graph G where each edge $(u, v) \in E$ has both a weight $w(u, v)$ (not necessarily positive) as well as a color $\text{color}(u, v) \in \{\text{red}, \text{blue}\}$.

The weight of a path $p = v_1 \rightarrow v_2 \rightarrow \dots \rightarrow v_k$ is equal to the sum of the weights of the edges, *plus* 5 for each pair of adjacent edges that are not the same color. That is, when traversing a path, it costs an *additional* 5 units to switch from one edge to another of a different color.

Give an efficient algorithm to find a lowest-cost path between two vertices s and t , and analyze its running time. (You may assume that there exists such a path.) For full credit, your algorithm should have a running time of $O(VE)$, but partial credit will be awarded for slower solutions.

Solution: Construct a new graph G' as follows: for each vertex $v \in V$, create *two* vertices, a red vertex v_R and a blue vertex v_B , in G' . For each edge $(u, v) \in E$, if $\text{color}(u, v) = \text{red}$, create (u_R, v_R) in G' . Otherwise, create (u_B, v_B) . In either case, the new edge should have the same weight as the original. Finally, for each vertex $v \in V$, create the edges (v_R, v_B) and (v_B, v_R) in G' , each with weight 5.

This works because all the red edges run between red vertices in G' , and all the blue edges run between blue vertices. Switching colors requires traversing an edge in G' between a red and a blue vertex, incurring the extra cost of 5 units.

Now, run Bellman-Ford twice: once starting from s_R and once starting from s_B . Examine the paths ending at t_R and t_B . Determine which of the four paths $(s_R \rightsquigarrow t_R, s_R \rightsquigarrow t_B, s_B \rightsquigarrow t_R, s_B \rightsquigarrow t_B)$ is shortest. Strip the subscripts to recover the desired path in G .

Constructing G' takes $O(V + E)$ time. G' has $2V$ vertices and $V + E$ edges. Running Bellman-Ford on G' takes $O(V'E') = O((2V)(V + E)) = O(VE)$ time. Thus the total running time is $O(VE)$.

Problem 5. DAG Paths [20 points]

Consider two vertices, s and t , in some directed acyclic graph $G = (V, E)$. Give an efficient algorithm to determine whether the number of paths in G from s to t is odd or even. Analyze its running time in terms of $|V|$ and $|E|$.

Solution: Here we present a dynamic programming algorithm that finds the number of paths from s to t . Determining whether that number is odd or even is easy.

Produce a topological ordering v_1, v_2, \dots, v_n of the vertices of G , and without loss of generality, let $v_1 = s$ and $v_n = t$ (because that is the only part of the graph we care about).

Define subproblems as follows: let $S[i]$ be the number of paths from v_i to $v_n = t$. We can define a recurrence expressing the solution to these subproblems in terms of smaller subproblems:

$$S[i] = \sum_{j \in \text{adj}(i)} S[j]$$

where $\text{adj}(i)$ is the set of all vertices v s.t. $(i, v) \in E$. (That is, all vertices to which there is an edge from i .) The base case is $S[n] = 1$.

The solution to the general problem is the number of paths from $v_1 = s$ to $v_n = t$, or $S[1]$. The total time required to solve the $O(V)$ subproblems is $O(E)$, and the topological sort requires $O(V + E)$ time, so the total running time is $O(V + E)$.

Problem 6. Square Depth [25 points]

Imagine starting with the given number n , and repeatedly chopping off a digit from one end or the other (your choice), until only one digit is left. The *square-depth* $S(n)$ of n is defined to be the maximum number of square numbers you can arrange to see along the way. For example, $S(32492) = 3$ via the sequence

$$32492 \rightarrow 3249 \rightarrow 324 \rightarrow 24 \rightarrow 4$$

since 3249, 324, and 4 are squares, and there is no better sequence of chops giving a larger score (although you can do as well other ways, since 49 and 9 are both squares).

Describe an efficient algorithm to compute the *square-depth*, $S(n)$, of a given number n , written as a d -digit decimal number $a_1a_2\dots a_d$. Analyze your algorithm's running time.

Your algorithm should run in time polynomial in d . You may assume a function `IS-SQUARE(x)` that returns, in constant time, 1 if x is square, and 0 if not.

Solution: We can solve this using dynamic programming.

First, we define some notation: let $n_{ij} = a_i \dots a_j$. That is, n_{ij} is the number formed by digits i through j of n . Now, define the subproblems by letting $D[i, j]$ be the square-depth of n_{ij} .

The solution to $D[i, j]$ can be found by solving the two subproblems that result from chopping off the left digit and from chopping off the right digit, and adding 1 if n_{ij} itself is square. We can express this as a recurrence:

$$D[i, j] = \max(D[i + 1, j], D[i, j - 1]) + \text{IS-SQUARE}(n_{ij})$$

The base cases are $D[i, i] = \text{IS-SQUARE}(a_i)$, for all $1 \leq i \leq d$. The solution to the general problem is $S(n) = D[1, d]$.

There are $\Theta(d^2)$ subproblems, and each takes $\Theta(1)$ time to solve, so the total running time is $\Theta(d^2)$.

Problem 7. Least-Squares Minimization [30 points] (2 parts)

- (a) Some least-squares minimizations can be solved by substitution even though the coefficient matrix is not upper triangular. For example, the following problem can be solved by substitution:

$$\begin{pmatrix} 1 & 0 & 0 \\ 0 & -7 & -2 \\ 0 & 3 & 0 \end{pmatrix} \begin{pmatrix} x_0 \\ x_1 \\ x_2 \end{pmatrix} = \begin{pmatrix} 5 \\ 8 \\ 4 \end{pmatrix}.$$

One way to solve such problems is to find an ordering of the rows and an ordering of the columns: find x_1 from the third constraint, then substitute x_1 and find x_2 from the second constraint, and so on.

Describe an efficient algorithm to find suitable orderings for the unknowns and the constraints. For simplicity, the algorithm is allowed to assume that there is such a reordering.

The input matrix is given to the algorithm as a Python list of m rows, where each row is a list of tuples; each tuple contains a column index (0 to $n - 1$) and a nonzero value. Zero matrix entries are not represented at all. The output should be a list of n (unknown index, constraint index) pairs, giving the substitution ordering. The algorithm should run in time $\Theta(n + m + k)$, where k the number of nonzeros in the entire matrix (which is also the number of tuples in the input).

Solution: We can perform a substitution for any row in which only one unsolved unknown remains. Start an empty queue S . This queue contains the rows for which there is exactly one unsolved unknown, so we can solve for that unknown and substitute the result into the other constraints.

Allocate two arrays: let $C[i]$ be the number of unknowns in constraint i , and $U[j]$ be a list of constraint/row numbers in which the unknown x_j appears.

Read in the constraints (rows). For each column index j in constraint i , increment $C[i]$ and append i to $U[j]$. For each row i that contains only one entry, append i to S ; we can immediately perform the substitution in this row.

Now repeat the following until all constraints are resolved (i.e., S becomes empty): extract the first row number, i , from S . Solve for the unknown in constraint i and save the result. Now that we have solved for this unknown, we are one step closer to solving each constraint in which it appears: for each i in $U[j]$, decrement $C[i]$. If $C[i]$ is now equal to 1, there is only one unsolved unknown left, so append i to S .

Because there are m constraints to solve, n unknown values to solve, and k substitutions to perform, and each of these operations takes $O(1)$ time, the total time is $O(n + m + k)$.

- (b) For many problems, substitution alone is not sufficient; Givens rotations are necessary.

One interesting challenge in performing a sequence of rotations efficiently is to compute unions of pairs of integer sets.

Describe an algorithm that repeatedly takes two sets of integers represented as lists of integers and generates a list of the union of the two sets. Also describe any data structures that your algorithm uses.

The integers are all between 0 and $n - 1$, there are no duplicates in the input lists, and there should be no duplicates in the output list. The pairs of sets are unrelated.

The algorithm can use $O(n)$ preprocessing time before computing the first union. After the preprocessing phase, computing each union should take $O(k + k')$ worst-case time, where k and k' are the sizes of the two input lists.

Solution: Take the first set and append just those elements from the second set that are not already present, then return the resulting set.

To do this, use a list, A , of n booleans, to keep track of the elements in the first list. Initialize them all to `False`. This preprocessing requires $O(n)$ time.

For i in the first input list, set $A[i]$ to `True`. Now, for j in the second input list, if $A[j]$ is `False`, append j to the first list. Return the resulting list.

Using hashing to perform the lookups is not correct because it requires $O(k + k')$ expected time, when worst-case time was required.

Problem 8. Sorting on Disks [30 points] (3 parts)

You need to sort a large array of n elements that is stored on disk (in a file). Files support only two operations:

1. $\text{READ}(f)$ returns the next number stored in file f .
2. $\text{WRITE}(f, p)$ writes the number p to the end of file f .

These operations are slow, so we wish to sort the given array into another file while incurring as few READs and WRITEs as possible. Your algorithm may create new files if you desire.

If we had unlimited RAM, we could solve this problem by simply reading the entire array into memory, sorting it, then writing it to disk. Unfortunately, the computer can only store a limited number of numbers in memory, so this simple approach is infeasible.

- (a) If our computer can store at most $\frac{n}{2}$ numbers in memory, how would you sort the file? Assume the computer has enough working memory to store any auxiliary information your algorithm needs. How many READs and WRITEs does your algorithm incur?

Solution: READ the first $\frac{n}{2}$ elements of the file, sort them, then WRITE the sorted array to a new file. READ and sort the remaining $\frac{n}{2}$ elements, then merge the two sorted arrays. This process incurs $\frac{3n}{2}$ READs and $\frac{3n}{2}$ WRITEs.

- (b) How would you sort the file if the computer can store at most $\frac{n}{4}$ numbers in memory? Again assume adequate working memory. How many READs and WRITEs does your algorithm incur?

Solution: Repeat the first step of part (a) 3 times, then perform a 4-way merge. This incurs $\frac{7n}{4}$ READs and $\frac{7n}{4}$ WRITEs.

- (c) Now imagine the computer uses some technology (e.g., flash memory) where the READ operation is fast, but the WRITE operation is expensive. Modify your algorithm from part (b) to minimize the number of WRITES. The computer can still store at most $\frac{n}{4}$ numbers in memory.

Solution: Perform a selection sort: select the smallest $\frac{n}{4}$ elements from the file, sort them, then WRITE them to the new file. Repeat this process for each successively larger set of $\frac{n}{4}$ elements until all the elements of the original array are sorted. This incurs n WRITES.

Problem 9. Star Power! [20 points]

Consider a directed, weighted graph G where all edge weights are positive. You have one Star, which (along with making you temporarily invincible) lets you traverse the edge of your choice for free. In other words, you may change the weight of any one edge to zero.

Give an efficient algorithm to find a lowest-cost path between two vertices s and t , given that you may set one edge weight to zero. Analyze your algorithm's running time. For full credit, your algorithm should have a running time of $O(E + V \lg V)$, but partial credit will be awarded for slower solutions.

Solution 1: Use Dijkstra's algorithm to find the shortest paths from s to all other vertices. Reverse all edges and use Dijkstra to find the shortest paths from all vertices to t . Denote the shortest path from u to v by $u \rightsquigarrow v$, and its length by $\delta(u, v)$.

Now, try setting each edge to zero. For each edge $(u, v) \in E$, consider the path $s \rightsquigarrow u \rightarrow v \rightsquigarrow t$. If we set $w(u, v)$ to zero, the path length is $\delta(s, u) + \delta(v, t)$. Find the edge for which this length is minimized and set it to zero; the corresponding path $s \rightsquigarrow u \rightarrow v \rightsquigarrow t$ is the desired path.

The algorithm requires two invocations of Dijkstra, and an additional $\Theta(E)$ time to iterate through the edges and find the optimal edge to take for free. Thus the total running time is the same as that of Dijkstra: $\Theta(E + V \lg V)$.

Solution 2: Construct a new graph G' as follows: for each vertex $v \in V$, create *two* vertices, v_0 and v_1 , in G' . The v_0 vertices represent being at v in G and having not yet used the Star. The v_1 vertices represent being at v , and having already used the Star.

For each edge $(u, v) \in E$, create three edges in G' : (u_0, v_0) and (u_1, v_1) with the original weight $w(u, v)$, and (u_0, v_1) with a weight of zero. An edge, (u_0, v_1) , crossing from the "0" component to the "1" component of the graph represents using the Star to traverse (u, v) .

Now, run Dijkstra's algorithm with source s_0 and destination t_1 . Drop the subscripts from the resulting path to recover the desired answer. (Observe that, because all edge weights are positive, it is always optimal to use the Star somewhere.)

G' has $2V$ vertices and $3E$ edges. Thus, constructing G' takes $\Theta(V + E)$ time, and running Dijkstra on G' takes $\Theta(E + V \lg V)$ time. Thus the total running time is $\Theta(E + V \lg V)$.



BPCPB4F4-1288-45E4-911D-6DD9B50D5B47

csci570-sp19-exam3

#68 2 of 10

Q1

18

1) 20 pts

Mark the following statements as **TRUE**, **FALSE**, or **UNKNOWN** (if unknown is given as a choice). No need to provide any justification.

[TRUE/FALSE/UNKNOWN] **UNKNOWN**

Let ODD denote the problem of deciding if a given integer is odd. Then ODD is polynomial time reducible to Vertex Cover.

[TRUE/FALSE] **FALSE**

If we find a quadratic time solution to an NP-Complete problem, then all NP-Complete problems can be solved in quadratic time.

[TRUE/FALSE/UNKNOWN] **TRUE**

Independent Set problem has a polynomial run time on certain types of graphs.

[TRUE/FALSE] **TRUE**

Consider two decision problems Q1, Q2 such that Q1 reduces in polynomial time to Hamiltonian Cycle and Hamiltonian Cycle reduces in polynomial time to Q2. Then it is possible for both Q1 and Q2 to be in NP.

[TRUE/FALSE] **TRUE**

The problem of determining whether there exists a cycle in an undirected graph is in NP.

[TRUE/FALSE] **FALSE**

If the edge weights in a connected undirected graph G are NOT all distinct, then there will be more than one minimum spanning tree in G.

[TRUE/FALSE] **FALSE**

Dijkstra's algorithm works correctly on graphs with negative-weight edges, as long as there are no negative-weight cycles.

[TRUE/FALSE/UNKNOWN] **UNKNOWN**

If Integer Linear Programming $\leq_p A$, then A cannot be solved in polynomial time.

[TRUE/FALSE] **TRUE**

Suppose that the capacities of the edges of an s-t flow network are integers that are all evenly divisible by 3. (E.g., 3, 6, 9, 12, etc.) Then there exists a maximum flow, such that the flow on each edge is also evenly divisible by 3.

[TRUE/FALSE] **FALSE**

Suppose that G is a directed, acyclic graph and has a topological ordering with v_1 the first node in the ordering and v_n the last node in the ordering. Then there is a path in G from v_1 to v_n .

CA3FA32E-FB2F-48A5-B669-698E2CD137DB

csc1570-sp19-exam3

#68 3 of 10



Q2

11



2) 16 pts

Recall the Interval Scheduling Problem. A set of n requests is given, each with a given start and finish time, $[s_i, f_i]$. The objective is to compute the maximum number of activities whose corresponding intervals do not overlap. In class we presented a greedy algorithm that solves this problem. We will consider some alternatives here.

(a) Earliest Activity First (EAF): Schedule the activity with the earliest start time.

Remove all activities that overlap it. Repeat until no more activities remain. Give an example to show that EAF is not optimal. Your example should show not only that it is not optimal, but its approximation ratio can be arbitrarily high. (6 pts)

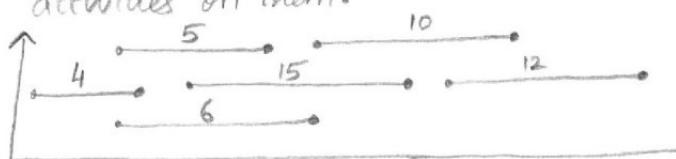
Consider the following counter example: Activity 1 : $[0, 15]$ and 14 activities i where $i=1 \text{ to } 14$ $[i, i+1]$. EAF algorithm selects 1 activity: activity 1. However the optimal solution chooses activities 1 to 14. Greedy value = 1 Optimal value = 14. Hence this algorithm is clearly not optimal. Considering 1 request $[0, w]$ and $w-1$ smaller requests $[i, i+1]$ we get an approximation ratio of $\frac{1}{w-1}$ we can make w as large as possible to get an arbitrarily high approximation ratio.

(b) Shortest Activity First (SAF): Schedule the activity with the smallest duration ($f_i - s_i$). Remove all activities that overlap it. Repeat until no more activities remain. Give an example to show that SAF is not optimal. Show that it is a $\frac{1}{2}$ approximation.

Wrong proof

-5 pts

Consider the following situation; where the graphs have lengths of the activities on them.



The optimal solution would be to schedule activities of length 4, 5, 12. However, SAF schedules

the activities of length 4 and 10. Hence it is NOT optimal.

The optimal greedy algorithm for this problem is Earliest finish time first. Consider a set of 3 jobs J_1, J_2 and J_3 where $f_1 < f_2 < f_3$ and $s_1 < s_2 < s_3$. $f_1 > s_2$, $f_1 - s_1 > f_3 - s_3 > f_2 - s_2$. For every such set of 3 jobs optimal greedy algorithm picks 2 jobs J_1 and J_3 , whereas SAF picks one job J_2 . Divide the entire set of jobs into sets of 3 each as mentioned above sorted according to finish times. We get $n/3$ sets where no of jobs = n . Max no of jobs picked by SAF = $n/3$. Max no of jobs picked by optimal algo = $2n/3$. We get an approximation ratio of $\frac{1}{2}$.



BB8A38A1-161F-41DB-811A-F6E124A1B24C

csci570-sp19-exam3

#68 4 of 10

3) 16 pts

Q3

14

There are n piles of dirt scattered over a large construction site. Pile number i weighs w_i tons and is at position p_i in the plane. The dirt in these piles has to be moved into m new piles at positions q_j (for $j = 1, \dots, m$) with weights v_j (also in tons).

The good news:

You have a bulldozer.

The bad news:

You have to figure out the cheapest way to move the dirt.

The cost of moving t tons of dirt from position p to positions q is $t \times \text{dist}(p, q)$ where $\text{dist}(p, q)$ is the distance from p to q . Note that t need not be an integer. Your goal is to minimize the total cost of moving the dirt from its current locations to the new ones. The input consists of the location (X and Y coordinates) of points $p_1, \dots, p_n, q_1, \dots, q_m$ and weights $w_1, \dots, w_n, v_1, \dots, v_m$. You may ignore the cost of driving around the construction site when not carrying dirt; you only need to account for the cost of moving dirt from pile to pile. Also, because no nuclear reactions will be involved, you may assume that mass is conserved, that is, $\sum_{i=1}^n w_i = \sum_{j=1}^m v_j$.

Give a linear program that minimizes the cost. No need to turn it into the standard form.

:D

Variables involved.

move_{ij} = weight of dirt moved from location p_i to q_j
in tonnes

Objective function :

$$\text{minimize} : \sum_{\substack{i=1 \\ j=1}}^{i=n \\ j=m} \text{move}_{ij} * \text{dist}(p_i, q_j)$$

Subject to constraints

$$\text{move}_{ij} \geq 0$$

$$\sum_{i=1}^n \text{move}_{ij} \leq v_j \quad \forall j = 1 \dots m$$

$$\sum_{j=1}^m \text{move}_{ij} = w_i \quad \forall i = 1 \dots n$$



Q4 14

4) 16 pts

Consider an instance of the Satisfiability Problem, specified by clauses C_1, \dots, C_m , over a set of Boolean variables x_1, \dots, x_n . We say that the instance is **monotone** if each term in each clause consists of non-negated variables; that is, each term is equal to x_i , for some i , rather than $\neg x_i$ (not x_i). For example, suppose we have the three clauses: $(x_1 \vee x_2), (x_1 \vee x_2 \vee x_3), (x_3 \vee x_4)$. This is monotone since we don't have any negated terms, and obviously the assignment that sets all four variables to 1 satisfies all the clauses. But we can observe that this is not the only satisfying truth assignment; we could also have set x_1 and x_4 to 1, and x_2 and x_3 to 0. Indeed, for any monotone instance, it is natural to ask how few variables we need to set to 1 in order to satisfy it.

Given a monotone instance of Satisfiability, together with a number k , the problem of Monotone Satisfiability with Few True Variables (MSFTV) asks: Provided m clauses, is there a satisfying assignment for the instance in which at most k variables are set to 1?

(1) Prove that the MSFTV problem is in NP. (4 pts)

Certificate: Check if MSFTV assignment satisfies the instance (all clauses are satisfied)

Certificate: 1) Check if every clause evaluates to 1.

2) Check if almost K variables are set to 1.

This can be done in polynomial time in the number of clauses.

(2) Prove that the MSFTV problem is NP-complete. (12 pts)

Say certificate =
assignment; poly-size (-0.5)

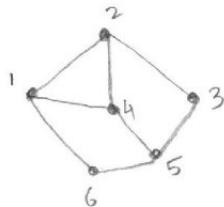
We choose Vertex cover. *Vertex Cover \leq_p MSFTV*

consider an instance of vertex cover problem with graph $G_1(V, E)$.

Set each vertex i in G_1 correspond to a variable x_i . For shown instance, we have

$x_1, x_2, x_3, x_4, x_5, x_6$.

Say construction clearly.



Add a clause for every edge in G_1 .

We get a set of clauses,

There exists a vertex cover of size almost K in G_1 if there exists a satisfying assignment where almost K variables are set to 1 in MSFTV.

$$\begin{aligned} C_1 &= x_1 \vee x_2 & C_3 &= \\ C_2 &= x_1 \vee x_4 & C_4 &= \\ C_5 &= x_3 \vee x_5 & C_6 &= \\ C_7 &= x_5 \vee x_6 & C_8 &= \end{aligned}$$

VC NPC =>
MSFTV NPH;
part 1 further
implies MSFTV
NPC. (-0.5)

Every variable set to 1 \rightarrow add its corresponding vertex to the vertex cover set. Every clause has to be satisfied so every clause will have atleast 1 variable set to 1. \Rightarrow every edge is covered. Good! :) Almost K variables are set to 1.

There exists a satisfying assignment with almost K true values if there exists a VC of size K . For every vertex in the vertex cover set, make its corresponding variable 1. Since every edge is covered atleast one variable in every clause is set to 1. Almost K variables in VC set \Rightarrow almost K variables set to 1. //



B6022C75-E728-40E2-B117-8507FC477176

csci570-sp19-exam3

#68 6 of 10

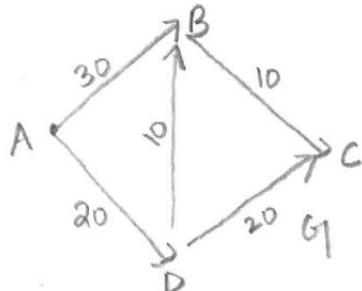
Q5 16

16 pts

Given a flow network (G, c, s, t) , John wants to solve the Max-cut problem, i.e. find an $s-t$ cut with maximum capacity. He proposes to do this by reducing it to the Min-cut problem. So, he first finds the edge with maximum capacity w . Then he considers a transformed network (G', c', s, t) where G' has the same nodes and edges as in G , and where $c'(e) = w - c(e)$ for every edge e . Thus, this new network has all non-negative capacities, and the edges having small capacities in the new network must originally have large capacities, and vice versa. John claims that a min-cut in the new network corresponds to a max-cut in the original network. Prove or disprove whether this is a valid reduction.

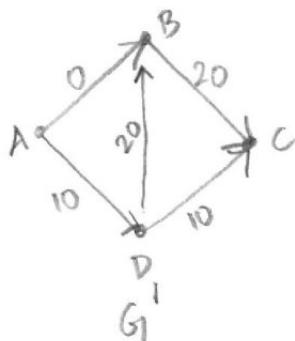
FALSE. Not a valid reduction.

Consider the following flow network.



The maxcut in this network is $\{A, D\} \cup \{B, C\}$ with value 60.

Applying the mentioned reduction,



The mincut in this reduced graph is $\{A\} \cup \{B, C, D\}$ with weight 10.
 However, $\{A\}, \{B, C, D\}$ is NOT the max cut in G' .

72784503-2603-4F40-9D0B-F90167B8BD01

csci570-sp19-exam3

#68 7 of 10



Q6 16

6) 16 pts

Consider a weighted complete bipartite graph G , with each partition having n vertices in it. (In a complete bipartite graph, there is an edge between each node in one partition to all nodes within the other partition, and vice versa). Let M be a maximum matching in G , i.e. a matching containing n disjoint edges.

Given a matching M , we want to find a spanning tree in G that contains M (i.e. all the edges in M) and has the minimum weight among all the spanning trees that contain M . Find an efficient algorithm for this and analyze its complexity.

- 1) Create a partial Tree T^* with n edges of M .
- 2) Create $E' = E - M$
- 3) Sort E' in ascending order of edges.
- 4) Perform Kruskal algorithm by adding $e \in E'$ to T^* if it does not create a cycle.

[Kruskal starts with n disjoint sets with n vertices in each set in the union find data structure]

- 5) Return when T^* has $2n-1$ edges.

Complexity: E has n^2 edges E' has n^2-n edges.

Sorting according to ascending order takes $O(n^2 \log n^2)$

$$= O(n^2 \log n)$$

+ complexity of Kruskal's algorithm with n^2-n edges and n nodes = $O(n \log n^2)$

overall complexity: $O(n^2 \log n)$



5BD9BFC3-F923-41BC-AC6A-E554D1865EC0

csci570-sp19-exam2

#668 2 of 10

Q1

20

1) 20 pts

Mark the following statements as **TRUE** or **FALSE**. No need to provide any justification.[TRUE/FALSE] *TRUE*

The Ford-Fulkerson algorithm can be used to find the maximum flow through a graph with cycles.

[TRUE/FALSE] *FALSE*

In a flow network where all edge capacities are unique, the min cut will be unique.

[TRUE/FALSE] *FALSE*A Dynamic Programming algorithm with n^3 unique sub-problems will run in $O(n^4)$ time.[TRUE/FALSE] *TRUE*

It is possible for a dynamic programming algorithm to have exponential running time.

[TRUE/FALSE] *FALSE*

Let G be a weighted directed graph with exactly one source s and exactly one sink t . Let (A, B) be a maximum cut in G , that is, A and B are disjoint sets whose union is V , $s \in A$, $t \in B$, and the sum of the weights of all edges from A to B is the maximum for any two such sets. Now let H be the weighted directed graph obtained by adding 1 to the weight of each edge in G . Then (A, B) must still be a maximum cut in H .

[TRUE/FALSE] *FALSE*

The recurrence $OPT(j) = \max_{1 \leq i \leq j} \{A[i] - B[j-i] + OPT(j-i)\}$ where A and B are fixed input arrays, will lead to an $O(n)$ dynamic programming algorithm.

[TRUE/FALSE] *FALSE*

In a graph with more than one minimum cut, if we take any edge e that is part of the (A, B) minimum cut (i.e. edge e goes from A into B) and increase the capacity of that edge by any integer $k \geq 1$, then (A, B) will no longer be a minimum cut.

[TRUE/FALSE] *TRUE*

The Ford-Fulkerson algorithm (without scaling) can be used to efficiently solve the maximum matching problem in a bipartite graph.

[TRUE/FALSE] *TRUE*

The space efficient version of the sequence alignment algorithm (which uses a combination of divide and conquer and dynamic programming) has the same running time complexity as the basic dynamic programming algorithm for this problem.

[TRUE/FALSE] *TRUE*

Suppose, increasing the capacity of edge e by 1, increases value of max flow f by 1. Then it must be that $f(e) = \text{capacity}(e)$.

F258F87D-5EAC-4B6A-AA78-C0B0603CEAFF

csci570-sp19-exam2

#668 3 of 10



Q2 6

2) 6 pts

Let $G = (V, E)$ be a flow network with source s , sink t , and integer capacities. Suppose that we are given a maximum flow f in G . Now, we increase the capacity of a single edge e by 1.

a) If $f(e) < \text{Capacity}(e)$, can we say that f will still be a maximum flow in G ? (2pts)

Yes.

b) If $f(e) = \text{Capacity}(e)$, can we say that f will no longer be a maximum flow in G ? (2 pts)

No.

c) Give an $O(V + E)$ -time algorithm to find a new maximum flow in G . (2 pts)

In the residual graph G_f add 1 to the capacity of the edge e . Now try to find an augmenting path from $s-t$. If an augmenting path exists, push flow through this path and update flow to f' . [ONE ITERATION OF FORD FULKERSON]. If no augmenting path exists, conclude that $f = \text{Max flow}$.
One iteration of Ford Fulkerson takes $O(V+E)$ time.



D4A246EB-C953-44F5-9DD2-C3BB6F80AC09

csci570-sp19-exam2

#668 4 of 10

Q3 20

3) 20 pts

We are given a string $S = s_1s_2s_3\dots s_n$, and we want to insert some characters such that the resulting string is a palindrome. Give an efficient dynamic programming algorithm to determine the minimum number of insertions.

Definition: A palindrome is a string that reads the same forward and backward such as ACCBCCA.

Example: Input: ACB Output: 2 (inserting C and A will create the palindrome ACBCA)

a) Define (in plain English) subproblems to be solved. (5 pts)
 We can reduce this problem to sequence alignment with gap costs = 1 and mismatch costs = 0 if $x_i \neq y_j$ and 0 otherwise. Compute alignment of S and reverse of S^R .
 $OPT(i, j) = \min \text{ cost of aligning } s_1\dots s_i \text{ with } s_1^R\dots s_j^R$.

b) Write the recurrence relation for subproblems. (7 pts)

$$OPT(i, j) = \begin{cases} OPT[i-1][j-1] & \text{if } s_i = s_j \\ \min(OPT[i-1][j], OPT[i][j-1]) + 1 & \text{otherwise} \end{cases}$$

where $s_i^R = S_{\text{reverse}}$

c) Using the recurrence formula in part b, write pseudocode to compute the minimum number of insertions. (5 pts)

Make sure you have initial values properly assigned. (3 pts)

```

Create  $S^R = \text{Reverse}(S)$ 
Create  $OPT[n+1][n+1]$ 
for i from 0 to n
     $OPT[i, 0] = i$ 
     $OPT[0, i] = i$ 
end for
for i from 1 to n
    for j from 1 to n
        calculate  $OPT(i, j)$  as per above recurrence
    end for
end for
min.no.of.insertions =  $OPT[n][n]/2$ .
    
```

0802A87C-59B3-49B6-8F9B-C479147CE917

csci570-sp19-exam2

#668 5 of 10



Q4

16

4) 20 pts

The Laver Cup is a tennis tournament that takes place every year between Team Europe and Team US, each consisting of k players. Each player participating has a world ranking. A match between players of ranking r_i and r_j is fair if $|r_i - r_j| \leq d$. The tournament must consist of n matches, each between a player from Team EU and a player from Team US. Describe an efficient algorithm that, given the rankings of all the players, determines if it is possible to schedule n fair matches between the two teams such that each player plays at least one match but no more than m matches, and no two players play each other more than once.

This problem can be reduced to a circulation problem with lower bounds and demands. Create $2k$ nodes in the flow network each for a player. Create edges between the nodes with capacity=1 if for player i and player j $|r_i - r_j| \leq d$. Create a source node s on one side, with a demand value $d_s = -n$ (no of matches) and connect s to every European player with an ^{edge of} lower bound=1 and capacity=m. Create a sink node t and connect every US player to t with lowerbound=1 and capacity=n. Create a demand $d_t = -n$ at node t . For remaining nodes demand=0. It is possible to schedule n matches if and only if the above network has a feasible circulation.

Missing/Incorrect edge value

Correctness

If the network has a feasible circulation, then we can schedule n matches. Given a feasible circulation, we can schedule matches such that if $f(e) = 1$ between US and Europe nodes, there is a match played and if $f(e) = 0$, there is no match. There are n matches because the sink has a demand of n , and every player plays a game because of the lower bound.

If we can schedule n matches, then the network has a feasible circulation. For every match we schedule give a flow of $f_e = 1$ to the edge connecting the 2 players playing the match. For every edge connecting source and Europe players we give a flow of $\frac{f_e}{\sum f(e)}$ and similarly for US and sink. $\sum f(e) = n$ because ^{out of 8th player} n matches are played and every $f(e) = 1$.



C7C8CD9E-4067-4CSB-9341-BEA8B3B17FC7

csci570-sp19-exam2

#668 6 of 10

Q5

19

5) 20 pts

Your company, Stuckbars Coffee and Toffee is planning to open several stores on Main Street. Main Street has n blocks 1, 2, ..., n from East to West, and for each block i you know the revenue $r_i > 0$ you expect from a store on this block. But if you open a store on block i , you cannot open another store in block i , or within the two blocks to its East or within the two blocks to its West. You have designed a dynamic programming algorithm for finding the store locations that maximizes total revenue, but you spilled coffee on it and now you have to reconstruct it. Here are the steps you need to follow.

a) Define (in plain English) subproblems to be solved. (4 pts)

$OPT(i)$ = maximum revenue that can be obtained till the i^{th} block.

b) Write the recurrence relation for subproblems. (6 pts)

$$OPT(i) = \max(OPT(i-3) + r_i, OPT(i-1))$$

c) Using the recurrence formula in part b, write pseudocode to compute the maximum revenue possible. (4 pts)

Make sure you have initial values properly assigned. (2 pts)

create OPT array of size $n+1$

$$OPT[0] = 0$$

$$OPT[1] = r_1$$

$$OPT[2] = \max(r_1, r_2)$$

for $i=3$ to n

$$\text{Compute } OPT[i] = \max(OPT[i-3] + r_i, OPT[i-1])$$

end for

$$\text{max revenue} = OPT[n]$$

09526247-3B83-4B09-823B-BE17C897112D

csci570-sp19-exam2

#668 7 of 10



- d) Using the results in part c, write pseudocode to determine the store locations.
(2 pts)

```

1/p% Computed OPT array
i=n
while (i>=0)
    if OPT[i] = OPT[i-3] + r_i :
        print i //STORE LOCATIONS
        i=i-3
    otherwise
        i=i-1

```

- e) Compute the runtime of the algorithm described in part c and state whether your solution runs in strongly, weakly, or pseudo polynomial time (2 pts)

There are n blocks and n entries in the OPT array. Computation of each $OPT(i)$ using recurrence, takes constant time. Hence runtime

$$= \sum_{i=1}^n O(1) = O(n)$$

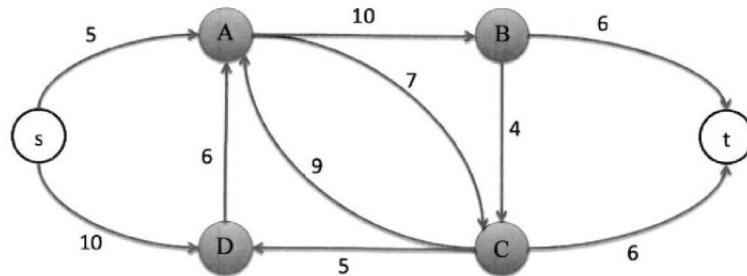
Solution runs in polynomial time $O(n)$.



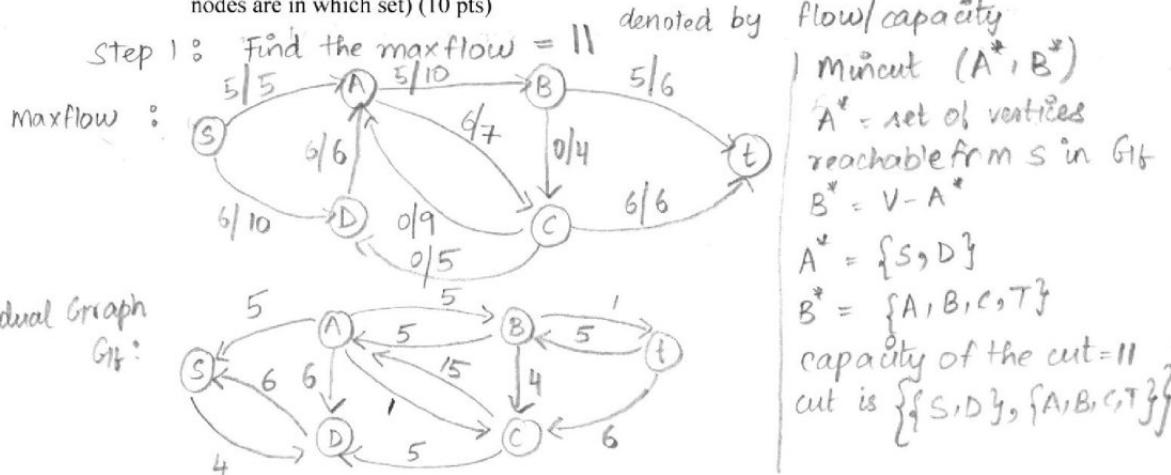
SC40B7EE-622F-48A0-8CD9-FD649696102F
csci570-sp19-exam2
#668 8 of 10

Q6 14

- 6) 14 pts
For the following flow network, with edge capacities as shown,



- a) Find a minimum s-t cut in the network and indicate what its capacity is.
(Clearly show all your steps in finding the minimum cut, and indicate which nodes are in which set) (10 pts)



- b) Describe an algorithm that would determine if this is the only minimum cut in G. (4 pts)

Reverse the edges in G_f and try to find a mincut from t. Let the new mincut be (A', B') . Let the graph obtained on reversing G_f be G_f' . $A' =$ set of vertices reachable from t in G_f' . $B' = V - A'$. If $(A', B') = (A^*, B^*)$ then A^*, B^* is a unique mincut. Otherwise (A', B') is not a unique mincut.

1BB5B6A2-C7F0-4959-A171-AA356685FA4A

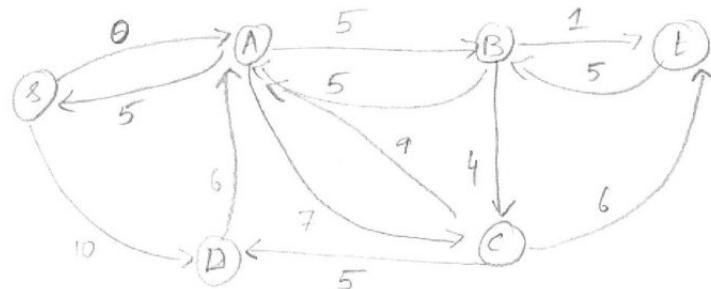
csc1570-spl19-exam2

#668 9 cf 10



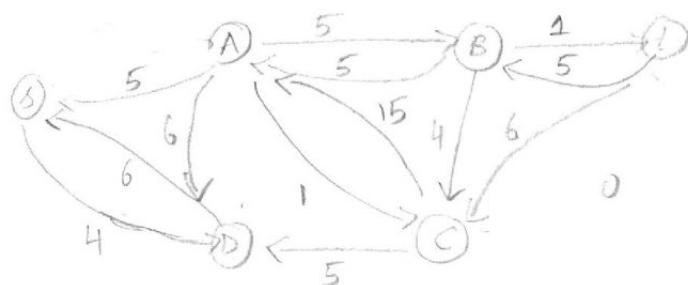
5+6

Additional Space



SAB+

S-D-A C-T



$E \checkmark$	1	2	3	4	5	6	7 ^w	8
\checkmark	4	2	3	6	\times 1	5	\$	

$OPT(i) = \max$ revenue that can be obtained with i blocks

$$OPT(0) = 0$$

for $i \in \text{range}(3, n)$

$$OPT(1) = r_1$$

$$OPT(2) = \max(r_1, r_2)$$

$$OPT(i) = \max_{r_i} \{ OPT(i-3) + r_i, OPT(i-1) \}$$

$$OPT(3) = \max(r_1, r_2, r_3)$$

$$\checkmark 1 \quad 2 \quad 3 \quad 4 \quad \checkmark \checkmark \quad 6 \quad \cancel{7} \quad \checkmark 8$$

0	5	15	5	8	11	11	13	13
---	---	----	---	---	----	----	----	----

For a long time it puzzled me how something so expensive, so leading edge, could be so useless, and then it occurred to me that a computer is a stupid machine with the ability to do incredibly smart things, while computer programmers are smart people with the ability to do incredibly stupid things. They are, in short, a perfect match.

— Bill Bryson, *Notes from a Big Country* (1999)

17 Applications of Maximum Flow

17.1 Edge-Disjoint Paths

One of the easiest applications of maximum flows is computing the maximum number of edge-disjoint paths between two specified vertices s and t in a directed graph G using maximum flows. A set of paths in G is *edge-disjoint* if each edge in G appears in at most one of the paths; several edge-disjoint paths may pass through the same vertex, however.

If we give each edge capacity 1, then the maxflow from s to t assigns a flow of either 0 or 1 to every edge. Since any vertex of G lies on at most two saturated edges (one in and one out, or none at all), the subgraph S of saturated edges is the union of several edge-disjoint paths and cycles. Moreover, the number of paths is exactly equal to the value of the flow. Extracting the actual paths from S is easy—just follow any directed path in S from s to t , remove that path from S , and recurse.

Conversely, we can transform any collection of k edge-disjoint paths into a flow by pushing one unit of flow along each path from s to t ; the value of the resulting flow is exactly k . It follows that the maxflow algorithm actually computes the largest possible set of edge-disjoint paths. The overall running time is $O(VE)$, just like for maximum bipartite matchings.

The same algorithm can also be used to find edge-disjoint paths in *undirected* graphs. We simply replace every undirected edge in G with a pair of directed edges, each with unit capacity, and compute a maximum flow from s to t in the resulting directed graph G' using the Ford-Fulkerson algorithm. For any edge uv in G , if our max flow saturates both directed edges $u \rightarrow v$ and $v \rightarrow u$ in G' , we can remove *both* edges from the flow without changing its value. Thus, without loss of generality, the maximum flow assigns a direction to every saturated edge, and we can extract the edge-disjoint paths by searching the graph of directed saturated edges.

17.2 Vertex Capacities and Vertex-Disjoint Paths

Suppose we have capacities on the vertices as well as the edges. Here, in addition to our other constraints, we require that for any vertex v other than s and t , the total flow into v (and therefore the total flow out of v) is at most some non-negative value $c(v)$. How can we compute a maximum flow with these new constraints?

One possibility is to modify our existing algorithms to take these vertex capacities into account. Given a flow f , we can define the *residual capacity* of a vertex v to be its original capacity minus the total flow into v :

$$c_f(v) = c(v) - \sum_u f(u \rightarrow v).$$

Since we cannot send any more flow into a vertex with residual capacity 0 we remove from the residual graph G_f every edge $u \rightarrow v$ that appears in G whose head vertex v is saturated. Otherwise, the augmenting-path algorithm is unchanged.

But an even simpler method is to transform the input into a traditional flow network, with only edge capacities. Specifically, we replace every vertex v with two vertices v_{in} and v_{out} , connected by an edge $v_{\text{in}} \rightarrow v_{\text{out}}$ with capacity $c(v)$, and then replace every directed edge $u \rightarrow v$ with the edge $u_{\text{out}} \rightarrow v_{\text{in}}$ (keeping the same capacity). Finally, we compute the maximum flow from s_{out} to t_{in} in this modified flow network.

It is now easy to compute the maximum number of vertex-disjoint paths from s to t in any directed graph. Simply give every vertex capacity 1, and compute a maximum flow!

17.3 Maximum Matchings in Bipartite Graphs

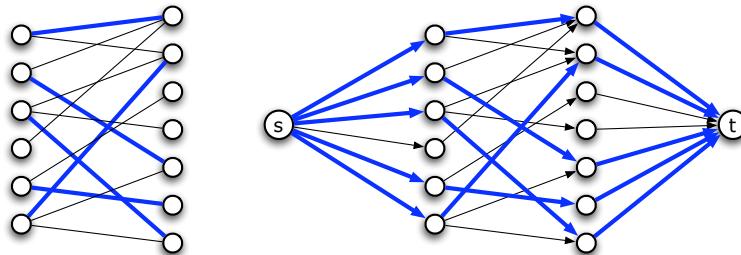
Another natural application of maximum flows is finding large *matchings* in bipartite graphs. A matching is a subgraph in which every vertex has degree at most one, or equivalently, a collection of edges such that no two share a vertex. The problem is to find the matching with the maximum number of edges in a given bipartite graph.

We can solve this problem by reducing it to a maximum flow problem as follows. Let G be the given bipartite graph with vertex set $U \cup W$, such that every edge joins a vertex in U to a vertex in W . We create a new *directed* graph G' by (1) orienting each edge from U to W , (2) adding two new vertices s and t , (3) adding edges from s to every vertex in U , and (4) adding edges from each vertex in W to t . Finally, we assign every edge in G' a capacity of 1.

Any matching M in G can be transformed into a flow f_M in G' as follows: For each edge uw in M , push one unit of flow along the path $s \rightarrow u \rightarrow w \rightarrow t$. These paths are disjoint except at s and t , so the resulting flow satisfies the capacity constraints. Moreover, the value of the resulting flow is equal to the number of edges in M .

Conversely, consider any (s, t) -flow f in G' computed using the Ford-Fulkerson augmenting path algorithm. Because the edge capacities are integers, the Ford-Fulkerson algorithm assigns an integer flow to every edge. (This is easy to verify by induction, hint, hint.) Moreover, since each edge has *unit* capacity, the computed flow either saturates ($f(e) = 1$) or avoids ($f(e) = 0$) every edge in G' . Finally, since at most one unit of flow can enter any vertex in U or leave any vertex in W , the saturated edges from U to W form a matching in G . The size of this matching is exactly $|f|$.

Thus, the size of the maximum matching in G is equal to the value of the maximum flow in G' , and provided we compute the maxflow using augmenting paths, we can convert the actual maxflow into a maximum matching. The maximum flow has value at most $\min\{|U|, |W|\} = O(V)$, so the Ford-Fulkerson algorithm runs in **$O(VE)$ time**.



A maximum matching in a bipartite graph G , and the corresponding maximum flow in G' .

17.4 Binary Assignment Problems

Maximum-cardinality matchings are a special case of a general family of so-called *assignment* problems.¹ An unweighted *binary* assignment problem involves two disjoint finite sets X and Y , which typically represent two different kinds of resources, such as web pages and servers, jobs and machines, rows and columns of a matrix, hospitals and interns, or customers and pints of ice cream. Our task is to choose the largest possible collection of pairs (x, y) as possible, where $x \in X$ and $y \in Y$, subject to several constraints of the following form:

- Each element $x \in X$ can appear in at most $c(x)$ pairs.
- Each element $y \in Y$ can appear in at most $c(y)$ pairs.
- Each pair $(x, y) \in X \times Y$ can appear in the output at most $c(x, y)$ times.

Each upper bound $c(x)$, $c(y)$, and $c(x, y)$ is either a (typically small) non-negative integer or ∞ . Intuitively, we create each pair in our output by *assigning* an element of X to an element of y .

The maximum-matching problem is a special case, where $c(z) = 1$ for all $z \in X \cup Y$, and each $c(x, y)$ is either 0 or 1, depending on whether the pair xy defines an edge in the underlying bipartite graph.

Here is a slightly more interesting example. A nearby school, famous for its onerous administrative hurdles, decides to organize a dance. Every pair of students (one boy, one girl) who wants to dance must register in advance. School regulations limit each boy-girl pair to at most three dances together, and limits each student to at most ten dances overall. How can we maximize the number of dances? This is a binary assignment problem for the set X of girls and the set Y of boys, where for each girl x and boy y , we have $c(x) = c(y) = 10$ and either $c(x, y) = 3$ (if x and y registered to dance) or $c(x, y) = 0$ (if they didn't register).

Every binary assignment problem can be reduced to a standard maximum flow problem as follows. We construct a flow network $G = (V, E)$ with vertices $X \cup Y \cup \{s, t\}$ and the following edges:

- an edge $s \rightarrow x$ with capacity $c(x)$ for each $x \in X$,
- an edge $y \rightarrow t$ with capacity $c(y)$ for each $y \in Y$.
- an edge $x \rightarrow y$ with capacity $c(x, y)$ for each $x \in X$ and $y \in Y$, and

Because all the edges have integer capacities, the Ford-Fulkerson algorithm constructs an integer maximum flow f^* . This flow can be decomposed into the sum of $|f^*|$ paths of the form $s \rightarrow x \rightarrow y \rightarrow t$ for some $x \in X$ and $y \in Y$. For each such path, we report the pair (x, y) . (Equivalently, the pair (x, y) appears in our output collection $f(x \rightarrow y)$ times.) It is easy to verify (hint, hint) that this collection of pairs satisfies all the necessary constraints. Conversely, any legal collection of r pairs can be transformed into a feasible integer flow with value r in G . Thus, the largest legal collection of pairs corresponds to a maximum flow in G . So our algorithm is correct.

17.5 Baseball Elimination

Every year millions of baseball fans eagerly watch their favorite team, hoping they will win a spot in the playoffs, and ultimately the World Series. Sadly, most teams are “mathematically eliminated” days or even weeks before the regular season ends. Often, it is easy to spot when a team is eliminated—they can’t win enough games to catch up to the current leader in their division. But sometimes the situation is more subtle.

For example, here are the actual standings from the American League East on August 30, 1996.

¹Most authors refer to finding a maximum-weight matching in a bipartite graph as *the* assignment problem.

Team	Won-Lost	Left	NYY	BAL	BOS	TOR	DET
New York Yankees	75–59	28		3	8	7	3
Baltimore Orioles	71–63	28	3		2	7	4
Boston Red Sox	69–66	27	8	2		0	0
Toronto Blue Jays	63–72	27	7	7	0		0
Detroit Lions	49–86	27	3	4	0	0	

Detroit is clearly behind, but some die-hard Lions fans may hold out hope that their team can still win. After all, if Detroit wins all 27 of their remaining games, they will end the season with 76 wins, more than any other team has now. So as long as every other team loses every game... but that's not possible, because some of those other teams still have to play each other. Here is one complete argument:²

By winning all of their remaining games, Detroit can finish the season with a record of 76 and 86. If the Yankees win just 2 more games, then they will finish the season with a 77 and 85 record which would put them ahead of Detroit. So, let's suppose the Tigers go undefeated for the rest of the season and the Yankees fail to win another game.

The problem with this scenario is that New York still has 8 games left with Boston. If the Red Sox win all of these games, they will end the season with at least 77 wins putting them ahead of the Tigers. Thus, the only way for Detroit to even have a chance of finishing in first place, is for New York to win exactly one of the 8 games with Boston and lose all their other games. Meanwhile, the Sox must loss all the games they play agains teams other than New York. This puts them in a 3-way tie for first place....

Now let's look at what happens to the Orioles and Blue Jays in our scenario. Baltimore has 2 games left with with Boston and 3 with New York. So, if everything happens as described above, the Orioles will finish with at least 76 wins. So, Detroit can catch Baltimore only if the Orioles lose all their games to teams other than New York and Boston. In particular, this means that Baltimore must lose all 7 of its remaining games with Toronto. The Blue Jays also have 7 games left with the Yankees and we have already seen that for Detroit to finish in first place, Toronto must will all of these games. But if that happens, the Blue Jays will win at least 14 more games giving them at final record of 77 and 85 or better which means they will finish ahead of the Tigers. So, no matter what happens from this point in the season on, Detroit can not finish in first place in the American League East.

There has to be a better way to figure this out!

Here is a more abstract formulation of the problem. Our input consists of two arrays $W[1..n]$ and $G[1..n, 1..n]$, where $W[i]$ is the number of games team i has already won, and $G[i, j]$ is the number of upcoming games between teams i and j . We want to determine whether team n can end the season with the most wins (possibly tied with other teams).³

We model this question as an assignment problem: We want to **assign a winner to each game**, so that team n comes in first place. We have an assignment problem! Let $R[i] = \sum_j G[i, j]$ denote the number of remaining games for team i . We will assume that team n wins all $R[n]$ of its remaining games. Then team n can come in first place if and only if every other team i wins at most $W[n] + R[n] - W[i]$ of its $R[i]$ remaining games.

Since we want to **assign** winning teams to games, we start by building a bipartite graph, whose nodes represent the games and the teams. We have $\binom{n}{2}$ game nodes $g_{i,j}$, one for each pair $1 \leq i < j < n$, and $n - 1$ team nodes t_i , one for each $1 \leq i < n$. For each pair i, j , we add edges $g_{i,j} \rightarrow t_i$ and $g_{i,j} \rightarrow t_j$ with infinite capacity. We add a source vertex s and edges $s \rightarrow g_{i,j}$ with capacity $G[i, j]$ for each pair i, j . Finally, we add a target node t and edges $t_i \rightarrow t$ with capacity $W[n] - W[i] + R[n]$ for each team i .

Theorem: Team n can end the season in first place if and only if there is a feasible flow in this graph that saturates every edge leaving s .

Proof: Suppose it is possible for team n to end the season in first place. Then every team $i < n$ wins at most $W[n] + R[n] - W[i]$ of the remaining games. For each game between team i and team j that team i

²Both the example and this argument are taken from <http://riot.ieor.berkeley.edu/~baseball/detroit.html>.

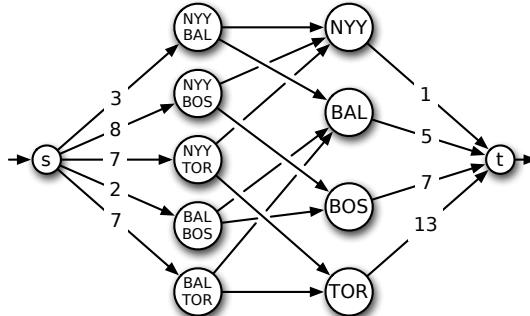
³We assume here that no games end in a tie (always true for Major League Baseball), and that every game is actually played (not always true).

wins, add one unit of flow along the path $s \rightarrow g_{i,j} \rightarrow t_i \rightarrow t$. Because there are exactly $G[i, j]$ games between teams i and j , every edge leaving s is saturated. Because each team i wins at most $W[n] + R[n] - W[i]$ games, the resulting flow is feasible.

Conversely, Let f be a feasible flow that saturates every edge out of s . Suppose team i wins exactly $f(g_{i,j} \rightarrow t_i)$ games against team j , for all i and j . Then teams i and j play $f(g_{i,j} \rightarrow t_i) + f(g_{i,j} \rightarrow t_j) = f(s \rightarrow g_{i,j}) = G[i, j]$ games, so every upcoming game is played. Moreover, each team i wins a total of $\sum_j f(g_{i,j} \rightarrow t_i) = f(t_i \rightarrow t) \leq W[n] + R[n] - W[i]$ upcoming games, and therefore at most $W[n] + R[n]$ games overall. Thus, if team n win all their upcoming games, they end the season in first place. \square

So, to decide whether our favorite team can win, we construct the flow network, compute a maximum flow, and report whether than maximum flow saturates the edges leaving s . The flow network has $O(n^2)$ vertices and $O(n^2)$ edges, and it can be constructed in $O(n^2)$ time. Using Dinitz's algorithm, we can compute the maximum flow in $O(VE^2) = O(n^6)$ time.

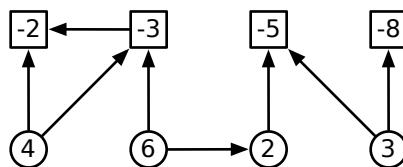
The graph derived from the 1996 American League East standings is shown below. The total capacity of the edges leaving s is 27 (there are 27 remaining games), but the total capacity of the edges entering t is only 26. So the maximum flow has value at most 26, which means that Detroit is mathematically eliminated.



The flow graph for the 1996 American League East standings. Unlabeled edges have infinite capacity.

17.6 Project Selection

In our final example, suppose we are given a set of n projects that we could possibly perform; for simplicity, we identify each project by an integer between 1 and n . Some projects cannot be started until certain other projects are completed. This set of dependencies is described by a directed acyclic graph, where an edge $i \rightarrow j$ indicates that project i depends on project j . Finally, each project i has an associated *profit* p_i which is given to us if the project is completed; however, some projects have negative profits, which we interpret as positive costs. We can choose to finish any subset X of the projects that includes all its dependents; that is, for every project $x \in X$, every project that x depends on is also in X . Our goal is to find a valid subset of the projects whose total profit is as large as possible. In particular, if all of the jobs have negative profit, the correct answer is to do nothing.



A dependency graph for a set of projects. Circles represent profitable projects; squares represent costly projects.

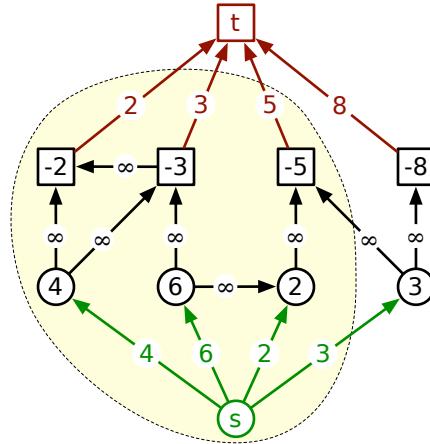
At a high level, our task is to partition the projects into two subsets S and T , the jobs we *Select* and the jobs we *Turn down*. So intuitively, we'd like to model our problem as a minimum cut problem in a certain graph. But in which graph? How do we enforce prerequisites? We want to *maximize* profit, but we only know how to find *minimum* cuts. And how do we convert negative profits into positive capacities?

We define a new graph G by adding a source vertex s and a target vertex t to the dependency graph, with an edge $s \rightarrow j$ for every profitable job (with $p_j > 0$), and an edge $i \rightarrow t$ for every costly job (with $p_i < 0$). Intuitively, we can think of s as a new job ("To the bank!") with profit/cost 0 that we must perform last. We assign edge capacities as follows:

- $c(s \rightarrow j) = p_j$ for every profitable job j ;
- $c(i \rightarrow t) = -p_i$ for every costly job i ;
- $c(i \rightarrow j) = \infty$ for every dependency edge $i \rightarrow j$.

All edge-capacities are positive, so this is a legal input to the maximum cut problem.

Now consider an (s, t) -cut (S, T) in G . If the capacity $\|S, T\|$ is finite, then for every dependency edge $i \rightarrow j$, projects i and j are on the same side of the cut, which implies that S is a valid solution. Moreover, we claim that selecting the jobs in S earns us a total profit of $C - \|S, T\|$, where C is the sum of all the positive profits. This claim immediately implies that we can *maximize* our total profit by computing a *minimum* cut in G .



The flow network for the example dependency graph, along with its minimum cut.
The cut has capacity 13 and $C = 15$, so the total profit for the selected jobs is 2.

We prove our key claim as follows. For any subset A of projects, we define three functions:

$$\begin{aligned} cost(A) &:= \sum_{i \in A: p_i < 0} -p_i = \sum_{i \in A} c(i \rightarrow t) \\ benefit(A) &:= \sum_{j \in A: p_j > 0} p_j = \sum_{j \in A} c(s \rightarrow j) \\ profit(A) &:= \sum_{i \in A} p_i = benefit(A) - cost(A). \end{aligned}$$

By definition, $C = benefit(S) + benefit(T)$. Because the cut (S, T) has finite capacity, only edges of the form $s \rightarrow j$ and $i \rightarrow t$ can cross the cut. By construction, every edge $s \rightarrow j$ points to a profitable job and each edge $i \rightarrow t$ points from a costly job. Thus, $\|S, T\| = cost(S) + benefit(T)$. We immediately conclude that $C - \|S, T\| = benefit(S) - cost(S) = profit(S)$, as claimed.

Exercises

1. Given an undirected graph $G = (V, E)$, with three vertices u , v , and w , describe and analyze an algorithm to determine whether there is a path from u to w that passes through v .
2. Let $G = (V, E)$ be a directed graph where for each vertex v , the in-degree and out-degree of v are equal. Let u and v be two vertices G , and suppose G contains k edge-disjoint paths from u to v . Under these conditions, must G also contain k edge-disjoint paths from v to u ? Give a proof or a counterexample with explanation.
3. A *cycle cover* of a given directed graph $G = (V, E)$ is a set of vertex-disjoint cycles that cover all the vertices. Describe and analyze an efficient algorithm to find a cycle cover for a given graph, or correctly report that no cycle cover exists. [Hint: Use bipartite matching!]
4. Consider a directed graph $G = (V, E)$ with multiple source vertices $s_1, s_2, \dots, s_\sigma$ and multiple target vertices t_1, t_2, \dots, t_τ , where no vertex is both a source and a target. A *multiterminal flow* is a function $f : E \rightarrow \mathbb{R}_{\geq 0}$ that satisfies the flow conservation constraint at every vertex that is neither a source nor a target. The value $|f|$ of a multiterminal flow is the total excess flow out of *all* the source vertices:

$$|f| := \sum_{i=1}^{\sigma} \left(\sum_w f(s_i \rightarrow w) - \sum_u f(u \rightarrow s_i) \right)$$

As usual, we are interested in finding flows with maximum value, subject to capacity constraints on the edges. (In particular, we don't care how much flow moves from any particular source to any particular target.)

- (a) Consider the following algorithm for computing multiterminal flows. The variables f and f' represent flow functions. The subroutine $\text{MAXFLOW}(G, s, t)$ solves the standard maximum flow problem with source s and target t .

```

MAXMULTIFLOW( $G, s[1.. \sigma], t[1.. \tau]$ ):
   $f \leftarrow 0$                                 «Initialize the flow»
  for  $i \leftarrow 1$  to  $\sigma$ 
    for  $j \leftarrow 1$  to  $\tau$ 
       $f' \leftarrow \text{MAXFLOW}(G_f, s[i], t[j])$ 
       $f \leftarrow f + f'$                          «Update the flow»
  return  $f$ 

```

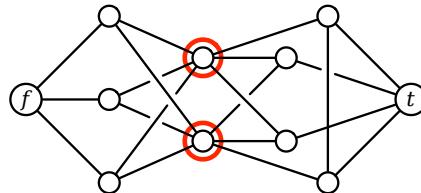
Prove that this algorithm correctly computes a maximum multiterminal flow in G .

- (b) Describe a more efficient algorithm to compute a maximum multiterminal flow in G .

5. The Island of Sodor is home to a large number of towns and villages, connected by an extensive rail network. Recently, several cases of a deadly contagious disease (either swine flu or zombies; reports are unclear) have been reported in the village of Ffarquhar. The controller of the Sodor railway plans to close down certain railway stations to prevent the disease from spreading to Tidmouth, his home town. No trains can pass through a closed station. To minimize expense (and public notice), he wants to close down as few stations as possible. However, he cannot close the Ffarquhar station, because that would expose him to the disease, and he cannot close the Tidmouth station, because then he couldn't visit his favorite pub.

Describe and analyze an algorithm to find the minimum number of stations that must be closed to block all rail travel from Ffarquhar to Tidmouth. The Sodor rail network is represented by an undirected graph, with a vertex for each station and an edge for each rail connection between two stations. Two special vertices f and t represent the stations in Ffarquhar and Tidmouth.

For example, given the following input graph, your algorithm should return the number 2.



6. The UIUC Faculty Senate has decided to convene a committee to determine whether Chief Illiniwek should become the official *maseet symbol* of the University of Illinois Global Campus. Exactly one faculty member must be chosen from each academic department to serve on this committee. Some faculty members have appointments in multiple departments, but each committee member will represent only one department. For example, if Prof. Blagojevich is affiliated with both the Department of Corruption and the Department of Stupidity, and he is chosen as the Stupidity representative, then someone else must represent Corruption. Finally, University policy requires that any committee on virtual *maseots symbols* must contain the same number of assistant professors, associate professors, and full professors. Fortunately, the number of departments is a multiple of 3.

Describe an efficient algorithm to select the membership of the Global Illiniwek Committee. Your input is a list of all UIUC faculty members, their ranks (assistant, associate, or full), and their departmental affiliation(s). There are n faculty members and $3k$ departments.

7. The University of Southern North Dakota at Hoople has hired you to write an algorithm to schedule their final exams. Each semester, USNDH offers n different classes. There are r different rooms on campus and t different time slots in which exams can be offered. You are given two arrays $E[1..n]$ and $S[1..r]$, where $E[i]$ is the number of students enrolled in the i th class, and $S[j]$ is the number of seats in the j th room. At most one final exam can be held in each room during each time slot. Class i can hold its final exam in room j only if $E[i] < S[j]$.

Describe and analyze an efficient algorithm to assign a room and a time slot to each class (or report correctly that no such assignment is possible).

8. Suppose we are given an array $A[1..m][1..n]$ of non-negative real numbers. We want to *round A* to an integer matrix, by replacing each entry x in A with either $\lfloor x \rfloor$ or $\lceil x \rceil$, without changing the sum of entries in any row or column of A . For example:

$$\begin{bmatrix} 1.2 & 3.4 & 2.4 \\ 3.9 & 4.0 & 2.1 \\ 7.9 & 1.6 & 0.5 \end{bmatrix} \longrightarrow \begin{bmatrix} 1 & 4 & 2 \\ 4 & 4 & 2 \\ 8 & 1 & 1 \end{bmatrix}$$

Describe an efficient algorithm that either rounds A in this fashion, or reports correctly that no such rounding is possible.

9. Suppose you are running a web site that is visited by the same set of people every day. Each visitor claims membership in one or more *demographic groups*; for example, a visitor might describe himself as male, 35–45 years old, a resident of Illinois, an academic, a blogger, a Joss Whedon fan⁴, and a Sports Racer.⁵ Your site is supported by advertisers. Each advertiser has told you which demographic groups should see its ads and how many of its ads you must show each day. Altogether, there are n visitors, k demographic groups, and m advertisers.

Describe an efficient algorithm to determine, given all the data described in the previous paragraph, whether you can show each visitor exactly *one* ad per day, so that every advertiser has its desired number of ads displayed, and every ad is seen by someone in an appropriate demographic group.

10. *Ad-hoc networks* are made up of low-powered wireless devices. In principle⁶, these networks can be used on battlefields, in regions that have recently suffered from natural disasters, and in other hard-to-reach areas. The idea is that a large collection of cheap, simple devices could be distributed through the area of interest (for example, by dropping them from an airplane); the devices would then automatically configure themselves into a functioning wireless network.

These devices can communicate only within a limited range. We assume all the devices are identical; there is a distance D such that two devices can communicate if and only if the distance between them is at most D .

We would like our ad-hoc network to be reliable, but because the devices are cheap and low-powered, they frequently fail. If a device detects that it is likely to fail, it should transmit its information to some other *backup* device within its communication range. We require each device x to have k potential backup devices, all within distance D of x ; we call these k devices the *backup set* of x . Also, we do not want any device to be in the backup set of too many other devices; otherwise, a single failure might affect a large fraction of the network.

So suppose we are given the communication radius D , parameters b and k , and an array $d[1..n, 1..n]$ of distances, where $d[i, j]$ is the distance between device i and device j . Describe an algorithm that either computes a backup set of size k for each of the n devices, such that no device appears in more than b backup sets, or reports (correctly) that no good collection of backup sets exists.

- *11. A *rooted tree* is a directed acyclic graph, in which every vertex has exactly one incoming edge, except for the *root*, which has no incoming edges. Equivalently, a rooted tree consists of a root vertex, which has edges pointing to the roots of zero or more smaller rooted trees. Describe a polynomial-time algorithm to compute, given two rooted trees A and B , the largest common rooted subtree of A and B .

[Hint: Let $LCS(u, v)$ denote the largest common subtree whose root in A is u and whose root in B is v . Your algorithm should compute $LCS(u, v)$ for all vertices u and v using dynamic programming. This would be easy if every vertex had $O(1)$ children, and still straightforward if the children of each node were ordered from left to right and the common subtree had to respect that ordering. But for unordered trees with large degree, you need another trick to combine recursive subproblems efficiently. Don't waste your time trying to reduce the polynomial running time.]

⁴Har har har! Mine is an evil laugh! Now die!

⁵It's Ride the Fire Eagle Danger Day!

⁶but not really in practice

6. If using a pencil to write the answers, make sure you apply enough pressure so your answers are readable in the scanned copy of your exam.
7. Do not write your answers in cursive scripts.

1) 20 pts

Mark the following statements as **TRUE** or **FALSE**. No need to provide any justification.

[**TRUE/FALSE**]

For any edge e that is part of a minimum cut in a flow network G , if we increase the capacity of that edge by any integer $k > 1$, then that edge will no longer be part of a minimum cut.

[**TRUE/FALSE**]

The sequence alignment algorithm described in class can be used to find the longest common subsequence between two given sequences.

[**TRUE/FALSE**]

The scaled version of the Ford-Fulkerson algorithm can compute the maximum flow in a flow network in polynomial time.

[**TRUE/FALSE**]

Given a set of demands $D = \{dv\}$ on a circulation network $G(V, E)$, if the total demand over V is zero, then G has a feasible circulation with respect to D .

[**TRUE/FALSE**]

In a flow network, the maximum value of an $s - t$ flow could be less than the capacity of a given $s - t$ cut in that network.

[**TRUE/FALSE**]

If f is a max $s - t$ flow of a flow network G with source s and sink t , then the capacity of the min $s - t$ cut in the residual graph G_f is 0.

[**TRUE/FALSE**]

In a graph with negative weight cycles, one such cycle can be found in $O(nm)$ time where n is the number of vertices and m is the number of edges in the graph.

[**TRUE/FALSE**]

An algorithm runs in weakly polynomial time if the number of operations is bounded by a polynomial in the number of bits in the input, but not in the number of integers in the input.

[**TRUE/FALSE**]

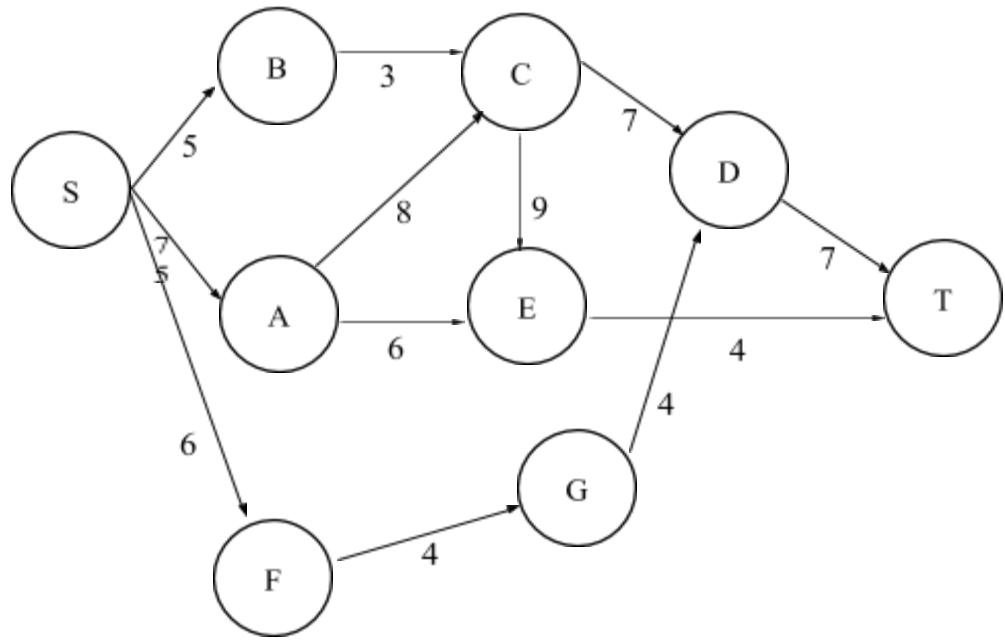
Let $G(V,E)$ be an arbitrary flow network, with a source s , a sink t . Given a flow f of maximum value in G , we can compute an s - t cut of minimum capacity in $O(|E|)$ time.

[**TRUE/FALSE**]

The basic Ford-Fulkerson algorithm can be used to compute a maximum matching in a given bipartite graph in strongly polynomial time.

1) 20 pts

Perform two iterations (i.e. two augmentation steps) of the scaled version of the Ford-Fulkerson algorithm on the flow network given below. You need to show the value of Δ and the augmentation path for each iteration, and the flow f and $G_f(\Delta)$ after each iteration. (Note: iterations may or may not belong to the same scaling phase)



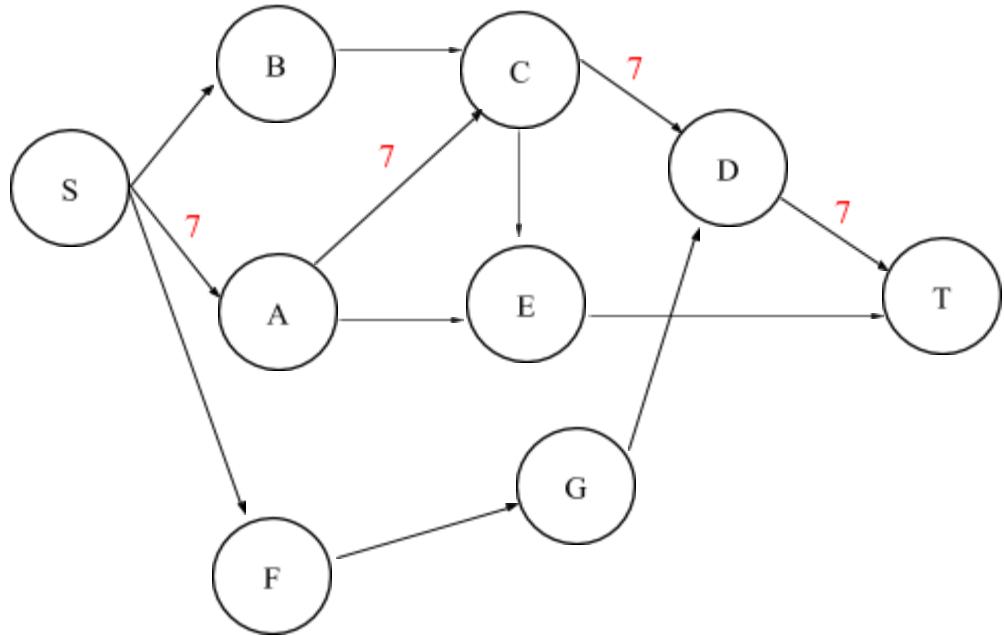
See next page...

(a) Iteration 1: (8 pts)

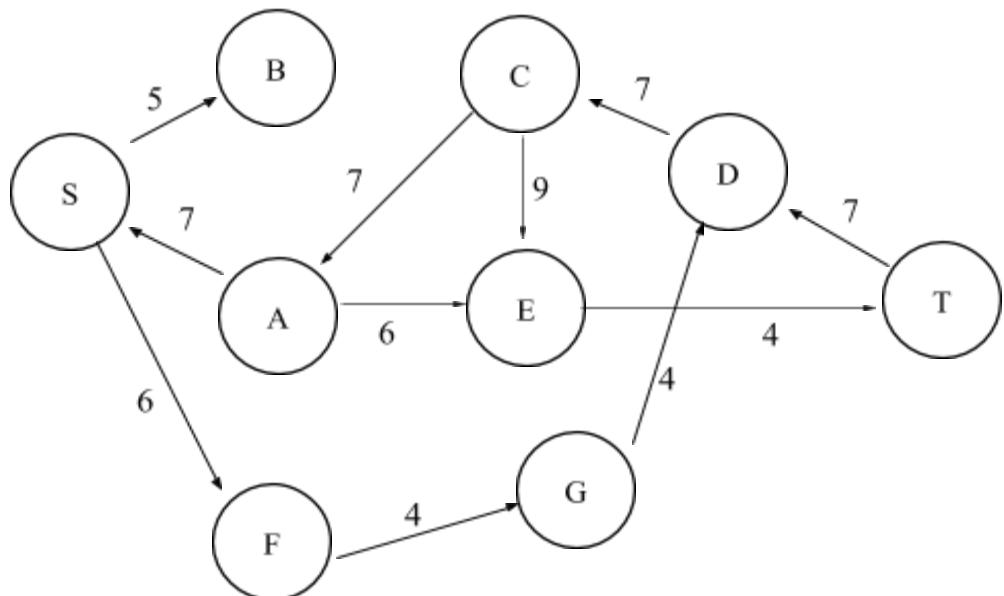
Δ is 4

Augmentation Path: SACDT (there are a few different choices of augmentation paths)

Flow after the first iteration (you can write flow values over each edge carrying flow):



$G_f(\Delta)$ after the first iteration (you can write flow values over each edge carrying flow):

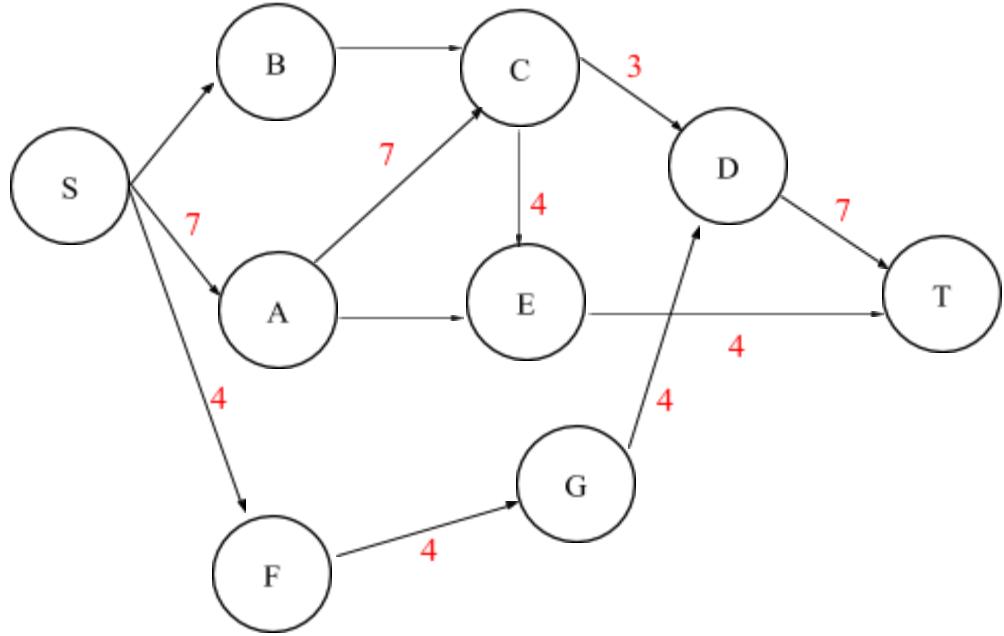


(b) Iteration 2: (8 pts)

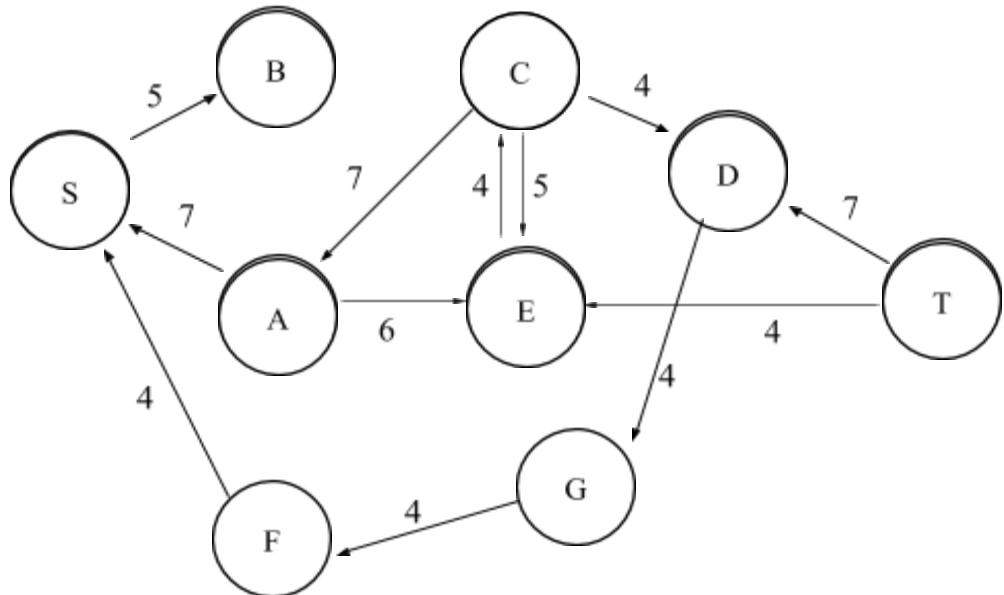
Δ is 4

Augmentation Path: SFGDCET (there are a few different choices of augmentation paths)

Flow after the second iteration (you can write flow values over each edge carrying flow):



$G_f(\Delta)$ after the first iteration (you can write flow values over each edge carrying flow):



(c) Can the choice of augmentation paths in the scaled version of Ford-Fulkerson affect the number of iterations? Explain why. (4 pts)

Yes, for example, if we had chosen paths with bottleneck values of 4 (SAET and SFGDT) in the first two iterations, then we should have needed more than two iterations to get to max flow. But with the choice of augmentation paths given above, we were able to get to max flow in 2 iterations.

An intuitive argument: Yes, there are still different choices for augmentation paths within a given scaling phase. The different augmentation paths could have different bottleneck values. This can affect how quickly the value of flow goes up and therefore affect how many iteration we will have to perform. In particular, using aug paths with higher bottleneck capacity can lead to fewer iterations

Common mistakes:

- 1) Using an example that proves this for the unscaled version but doesn't work for the scaled one.
- 2) Referencing Edmond Karp and the idea of using 'shortest paths'

Rubric:

Parts a) + b)

Computing the Delta values: $3+3 = 6$

Full points for (4,4).

For any Delta in (7/8/9), (Delta, round(Delta/2)) gets -2 penalty.

Most other cases get 0. 1 or 2 partial credit in VERY RARE cases.

Aug path + flow: $2+2 = 4$

Aug path flow not equal to the bottleneck edge: -2

Gf(Delta): $3+3 = 6$

-1 for 1 edge missing/wrong. -2 for more.

In particular, not filtering edges higher than Delta gets -2. A total of -3 if the both a) and b) have this mistake.

Part c)

0 If the answer is No.

If Yes, 1 for the answer, 3 for the explanation. Partial credit 1/2 if the explanation is incomplete/unclear.

3) 20 pts

The Levenshtein distance between two words is the minimum number of single-character edits (i.e. insertions, deletions or substitutions) required to change one word into the other. Each of these operations has unit cost.

For example, the Levenshtein distance between “kitten” and “sitting” is 3. A minimal edit script that transforms the former into the latter is:

kitten -> sitten

sitten -> sittin

sittin -> sitting

We want to design a dynamic programming algorithm that calculates the Levenshtein distance between a string X of length m and another string Y of length n. An edit can be adding, removing, or changing a character in X.

- a) Define (in plain English) subproblems to be solved. (4 pts)

$\text{OPT}(i, j)$ is the Levenshtein distance between $X_1..X_i$ and $Y_1..Y_j$

Rubrics:

- (3 pts) subproblem is the distance between two substrings of X and Y
 - (1 pt) specify two substrings are $X_1 \dots X_i$ and $Y_1 \dots Y_j$

- b) Write the recurrence relation for subproblems. (6 pts)

Rubrics:

- (2 pt) if specify there are two cases (a) $X_i = Y_j$ and (b) $X_i \neq Y_j$
 - (1 pts) if write correct answer for the case $X_i = Y_j$
 - (3 pts) if write correct answer for the case $X_i \neq Y_j$

- c) Using the recurrence formula in part b, write pseudocode (using iteration) to compute the Levenshtein distance between strings X and Y. (6 pts)
Make sure you have initial values properly assigned. (2 pts)

Initialize OPT to be a $(m + 1) * (n + 1)$ 2d array.

$$OPT(0,j) = j \text{ for all } j=1..n$$

$$\text{OPT}(i, 0) = i \text{ for all } i=1..m$$

For i=1 to m

For i=1 to n

```

    If X[i] == Y[j]
        OPT(i, j) = OPT(i - 1, j - 1)
    ELSE
        OPT(i, j) = min{OPT(i-1, j-1), OPT(i, j-1), OPT(i-1,j)} + 1
    endfor
endfor
RETURN OPT(m, n)

```

Rubrics:

2 pts for Initialization

- (1 pt) specify the first column and the first row need to be initialized specifically
- (1 pt) correct initialization of $\text{OPT}(i, 0)$, $\text{OPT}(j, 0)$ for $i=1\dots m$
- if use extra matrix (other than OPT), get 1pt off if no pseudocode of initialization

6 pts for pseudo code

- implement the correct recurrence formula using nested for loop
 - if the recurrence formula get 3 points off, then this part will also get 3 points off
 - get 2pts off if mess up $(i-1,j-1)$, $(i-1,j)$, $(i,j-1)$
- get 2pts off if the nested for loop is not implemented correctly
 - get 1pt off if the for loop's order is in-compatible with the recurrence formula

- d) Compute the runtime of the algorithm described in part c and state whether your solution runs in polynomial time or not (2 pts)

Runtime: $O(mn)$. Polynomial time

Rubrics:

- (1 pt) $O(mn)$
- (1 pt) polynomial/strong polynomial

4) 20 pts

Suppose a concert has just ended and C cars are parked at the event. We would like to determine how long it takes for all of them to leave the area. For this problem, we are given a graph representing the road network where all cars start at a particular vertex s (the parking lot) and several vertices (t_1, t_2, \dots, t_k) are designated as exits. We are also given capacities (in cars per minute) for each road (directed edges). Give a polynomial-time algorithm to determine the amount of time necessary to get all cars out of the area.

Solution:

We already have a source s (the parking lot) and capacities; however, we have multiple sinks (the exits). We can define a new network based on the road network by adding a sink T and connect each exit to it with a directed edge; the capacity can be any number that is at least as much as the total incoming capacity to that exit. We then run any polynomial-time max flow algorithm on the resulting graph; because it is polynomial in size (has only one additional vertex and at most n additional edges), the resulting runtime to compute $v(f^*)$ is polynomial. However, we don't want to stop at computing $v(f^*)$, because that's just the rate at which cars leave the parking lot. It takes $c/v(f^*)$ minutes for the parking lot to empty.

Rubric:

1) The defined work is fully correct. (6 points)

- a) If there is no sink T , deduct 3 points
- b) If the given road network is not used, deduct 2 points

2) The capacity on the new edges from each exit to the sink is correct. (4 points)

Note there can be multiple edges connected to the exit. And the edges are newly defined, whose capacities are not given. The capacity should be at least as much as the total incoming capacity to that exit or infinity.

3) Run any polynomial-time max flow algorithm on the resulting graph to compute the $v(f^*)$. (6 points)

The implementation of max flow is not right, deduct 2 points

4) The value of max flow $v(f^*)$ is the rate at which cars leave the parking lot. It takes $c/v(f^*)$ minutes for the parking lot to empty. (4 points)

- a) If the min time $c/v(f^*)$ is not clearly shown, deduct 2 points

For any attempt. (4 points)

5) 20 pts

There are n companies participating in a trade show. Company i gives away goodies worth g_i at their booth to each person visiting their booth. But you might have to wait in a line to get to the booth.

- a) Knowing that the wait time at booth i is $w_i \geq 0$, formulate a solution that will earn you a minimum of G dollars' worth of goodies without spending more than a total of H hours waiting in lines. Your solution should also indicate, given G and H , if this objective is not possible to achieve. (18 pts)

This is a 0-1 knapsack problem.

The solution can either show a reduction to 0/1 Knapsack or show a direct solution (recurrence formula, pseudocode, etc.)

For a reduction, it is enough to say that:

- H is the capacity of the knapsack
- w_i and g_i are the weight and value of the items respectively
- If the maximum value that we can pack into the knapsack is greater than G , we have a solution, otherwise there is no solution that satisfies the bounds H and G

For the direct solution, we need to

- Define the value of the optimal solution for a unique subproblem
- Present a recurrence formula
- Write pseudocode to show how we can find the value of the optimal solution
- Compare the value of the optimal solution to G and determine if there exists a solution that satisfies the bounds H and G or not.

Rubrics:

- No subproblem definition; - 3 points
- No pseudocode; - 5 points
- Whether efficient; - 1 point
- How to decide whether is possible; - 2 points
- Wrong recursive function; - 5 points
- Wrong algorithm; - 18 points
- Incorrect solution based on the assumption; - 5 points
- Miss the boundary case; - 3 points

- b) Analyze the complexity of your solution and determine if it is an efficient solution. (2 pts)

Solution runs in $O(Hn)$. This is pseudopolynomial and therefore not efficient.

6. If using a pencil to write the answers, make sure you apply enough pressure so your answers are readable in the scanned copy of your exam.
 7. Do not write your answers in cursive scripts.
- 1) 20 pts
- Mark the following statements as **TRUE** or **FALSE**. No need to provide any justification.

[**TRUE/FALSE**]

The residual graph of a maximum flow f can be strongly connected.

Note: A directed graph is strongly connected if there is a path from every node to every other node in the graph.

[**TRUE/FALSE**]

The Ford-Fulkerson algorithm can be used to **efficiently** compute a maximum matching of a given bipartite graph.

[**TRUE/FALSE**]

In successive iterations of the Ford-Fulkerson algorithm, the total flow passing through a vertex in the graph never decreases.

[**TRUE/FALSE**]

In a dynamic programming solution, the space requirement is always at least as big as the number of unique sub problems.

[**TRUE/FALSE**]

Any problem that can be solved with a greedy algorithm can also be solved with dynamic programming.

[**TRUE/FALSE**]

In a flow network G , if we increase the capacity of one edge by a positive amount x and we observe that the value of max flow also increases by x , then that edge must belong to **every** min-cut in G .

[**TRUE/FALSE**]

If we have a dynamic programming algorithm with n^2 subproblems, it is possible that the running time could be asymptotically strictly greater than $\Theta(n^2)$.

[**TRUE/FALSE**]

If we modify the Ford-Fulkerson algorithm by the following heuristic, then the time complexity of the resulting algorithm will be strongly polynomial:

At each step, choose the augmenting path with fewest edges.

[**TRUE/FALSE**]

The dynamic programming solution (presented in class) to the 0/1 knapsack problem has a polynomial run time.

[TRUE/FALSE]

Bellman-Ford algorithm runs in strongly polynomial time.

2) 20 pts.

Given a list of n integers v_1, \dots, v_n the “product-sum” of the list is the largest sum that can be formed by multiplying adjacent elements in the list, with the limitation that each element can only be multiplied with at most one of its neighbors. For example, given the list 1,2,3,1 the product sum is $8 = 1 + (2 \times 3) + 1$, and given the list

2,2,1,3,2,1,2,2,1,2 the product sum is

$$19 = (2 \times 2) + 1 + (3 \times 2) + 1 + (2 \times 2) + 1 + 2.$$

a) Verify that the product-sum of 1, 4, 3, 2, 3, 4, 2 is 29. (2pts)

$$29 = 1 + (4 \times 3) + 2 + (3 \times 4) + 2.$$

Give a dynamic programming algorithm for computing the value of the product-sum of a list of integers.

b) Define (in plain English) subproblems to be solved. (5 pts)

Let $OPT[j]$ be the product-sum for elements 1..j in the list.

Rubric:

1. Mention $OPT[j]$ is the product-sum: 3pts
2. The range should be “from v_1 to v_j ”: 2pts

Note: Some students incorrectly define the sub-problem as “ $OPT[i, j]$ is product-sum of v_i to v_j ”: 2pts deducted

c) Write the recurrence relation for subproblems. (7 pts)

$$OPT[j] = \begin{cases} \max\{OPT[j - 1] + v_j, OPT[j - 2] + v_j * v_{j-1}\} & \text{if } j \geq 2 \\ v_1 & \text{if } j = 1 \\ 0 & \text{if } j = 0 \end{cases}$$

Rubric:

1. Give a correct form of the recurrence, say, “ $OPT[j] = \max(\dots, \dots)$ ”. In the max function there are two entries, “ $OPT[j-1] + v_j$ ” and “ $OPT[j-2] + v_j * v_{j-1}$ ”: each for 3pts
2. Boundary case arranged correct as “ $OPT[0] = 0$, $OPT[1] = v_1$ or $OPT[1] = v_1$, $OPT[2] = \max(v_1 + v_2, v_1 * v_2)$ ”: 1pt

- d) Write pseudo-code to compute the value of the product-sum (You do not need to determine which pairs are multiplied together – we just need the value of the optimal solution.) (4 pts)

```
Prod-Sum(int[ ]v, n)
if (n == 0)
    return 0;
int [ ] OPT = new int [n + 1];
OPT[0] = 0; OPT[1] = v[1];
for int j = 2 to n
    OPT[j] = max(OPT[j - 1] + v[j], OPT[j - 2] + v[j] * v[j - 1]);
endfor
return OPT[n];
```

Rubric:

1. If your recurrence in Part(b) is incorrect and the pseudo-code is formed on top of that: 2pts
2. If your Part(b) is correct but the pseudo-code is wrong, e.g., additional for-loop: 3pt
3. Boundary case is correct: 1pt

Compute the runtime of the algorithm in terms of n . (2 pts)

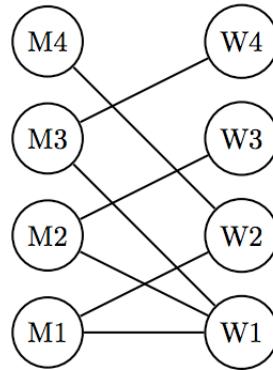
$O(n)$

Rubric:

1. If you leave Part(c) blank and simply write an answer here: no point
2. If your Part(c) is incorrect and you derive the complexity based on that: 1pt
3. If your Part(c) is correct but the complexity is wrong: 1pt

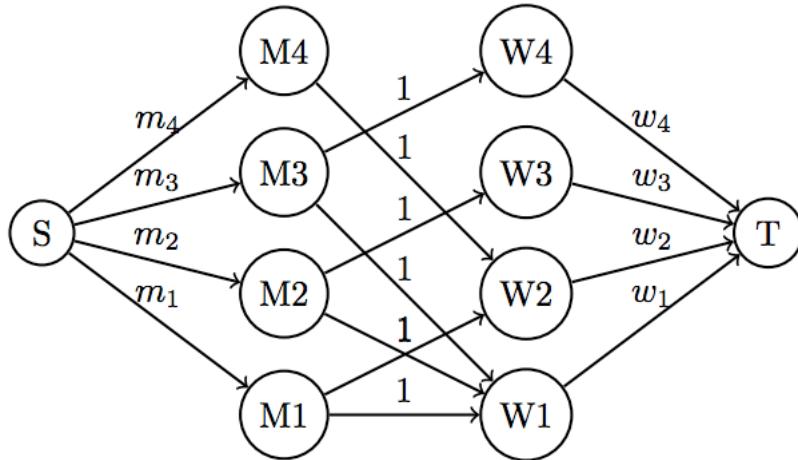
3) 20 pts

A dating website has used complicated algorithms to determine the compatibility between the profiles of men and women on their website. The following graph shows the set of compatible pairs. The website is trying to setup meetings between the men and the women. The i^{th} man has indicated a preference of meeting exactly m_i women, while the j^{th} woman prefers to meet at most w_j men. All meetings must be between compatible pairs. How would you use network flow to set up the meetings?



a) Construct a network flow solution, i.e. draw the network with which you can solve this problem, and explain your construction. (10 pts)

Solution:



We design a flow network, shown above, such that each unit of flow indicates a meeting between a man and a women. Observe that:

- Edges (S, M_i) have capacity m_i , enabling man i to be matched with up to m_i women.

- Since each man-woman pair should only be matched at most once, edges (M_i, W_j) have unit capacity.
- Edges (W_j, T) have capacity w_j , preventing woman j from being matched with more than w_j men.

Rubrics:

Add source and sink linking to all men and all women: 2pt

Edge capacity (S, M_i) should have m_i capacity 2 pts

Edge (M_i, W_j) should have unit capacity, directed from M_i to W_j : 4 pts

Edge (W_j, T) should have w_j capacity 2 pts

Or as with the lower bounds:

Add source and sink linking to all men and all women: 2pt

Edge capacity (S, M_i) should be m_i with m_i lower bound: 2pts

Edge capacity (M_i, W_j) should be 1 with 0 lowerbound:4 pts

Edge capacity (W_j, T) should be w_j with 0 lowerbound:2 pts

Small mistakes -1pt (can add up)

b) Explain how the solution to the network flow problem you constructed in part a will correspond to the solution to our dating problem, and prove the correctness of your solution. (10 pts)

We find max flow f . If the value of max flow $v(f)$ equals $\sum_i m_i$ then we have a feasible solution. Unit flow on edge (M_i, W_j) indicates that meeting between M_i and W_j takes place in the schedule.

Proof of solution:

Prove that G has a max flow of $\sum_i m_i$ iff there is a feasible schedule of meetings for all men and women.

A – If we have a max flow of $\sum_i m_i$ in the network we will have a schedule of meetings between men and women that meets all constraints:

- Man i is scheduled to meet with exactly m_i women since all edges out of S must be saturated, so there is a flow of exactly m_i going into the node corresponding to man i and therefore there are m_i edges carrying unit flows from i to m_i women, i.e. m_i meetings scheduled
- Woman j is scheduled to meet with at most w_j men since flow over edge connecting woman j to sink t has a capacity of w_j , so there cannot be more than w_j unit flows coming into j , i.e. j can have up to w_j meetings scheduled.

B - If we have a schedule of meetings between men and women that meets all constraints, we will have a max flow of $\sum_i m_i$ in the network:

- Send a flow of exactly m_i from S to man i
- Send a unit flow from man i to woman j if there is a meeting scheduled between them
- Send a flow equal to the number of meetings scheduled for woman j , from j to T

This will be a max flow since all edges out of S are saturated, and it will be a valid flow since conservation of flow is satisfied at all nodes.

Solution 2: Circulation

Build a network similar to that in solution 1, but instead place a demand value of $\sum_i m_i$ at T and $-\sum_i m_i$ at S

Proof will follow the same argument as in solution 1

Solution 3: Circulation with lower bounds

Build a network similar to that in solution 2, but also add lower bounds of m_i to those edges between S and many i. These lower bounds are basically redundant in this particular problem and the solution works with or without them.

Proof will follow the same argument as in solution 1

Rubric:

Mention the matching is possible when we have a max flow 3 pts
(Will not be credited if graph in part a is not correct)

The value of max flow should be explicitly written as exactly $F = \sum_i m_i$ to satisfy the condition: 3pts

(The value of max flow can be omitted only when the graph was constructed with lower bounds, and claimed a solution with valid flow/circulation)

Saying max flow $F = \sum_i m_i = \sum_j w_j$ -2pts

Correct proof of correctness: 4pts

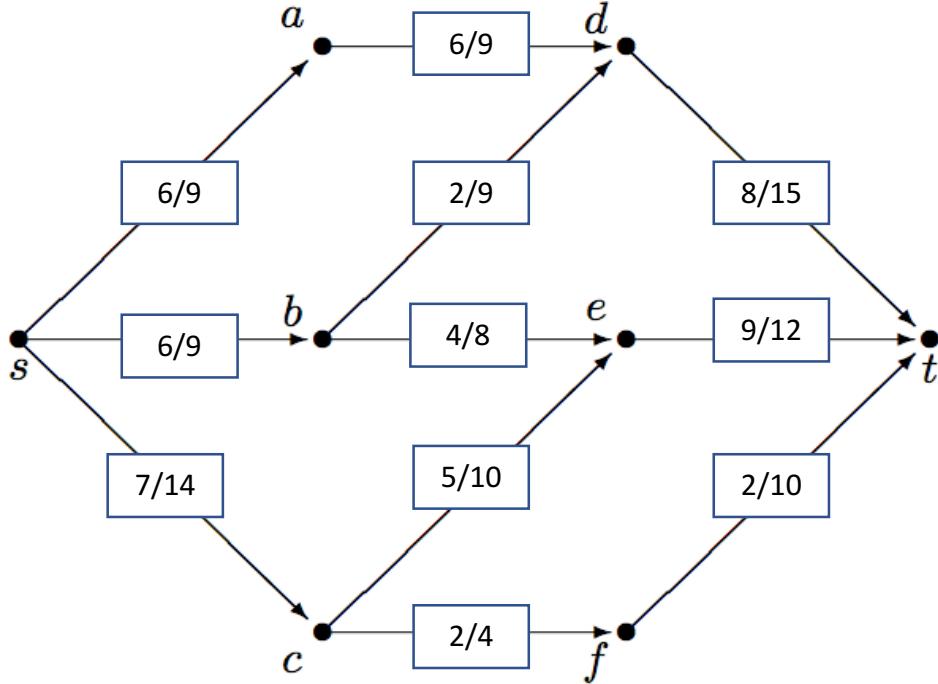
(In order to prove the correspondence, if and only if condition between max-flow \leftrightarrow dating solution should be proved. This was handled in the lecture.)

Case of weak proof: -2pts for one direction(e.g. stating only “if” statement) proof

Not enough explanation about proof: -2pts (e.g. writing statement only)

4) 20 pts.

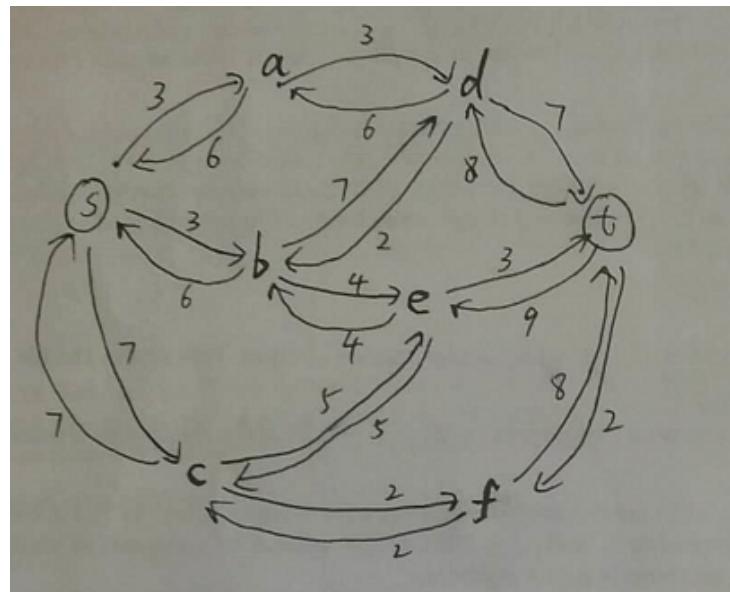
Consider the following flow graph G , with an assigned flow f . The pair x/y indicates that the edge is carrying a flow x and has capacity y .



a) Draw the residual Graph G_f for the flow f . (10 pts)

Solution:

We apply the standard construction for the residual graph. If there is a flow of f and capacity c on the edge (u, v) , we create an edge of capacity $c - f$ from u to v , and an edge of capacity f from v to u .
As shown in the graph below.



Rubric:

Wrong directions of the edges (-2 ~ -4)

Only draw one direction of the connected edges (-4 ~ -5)

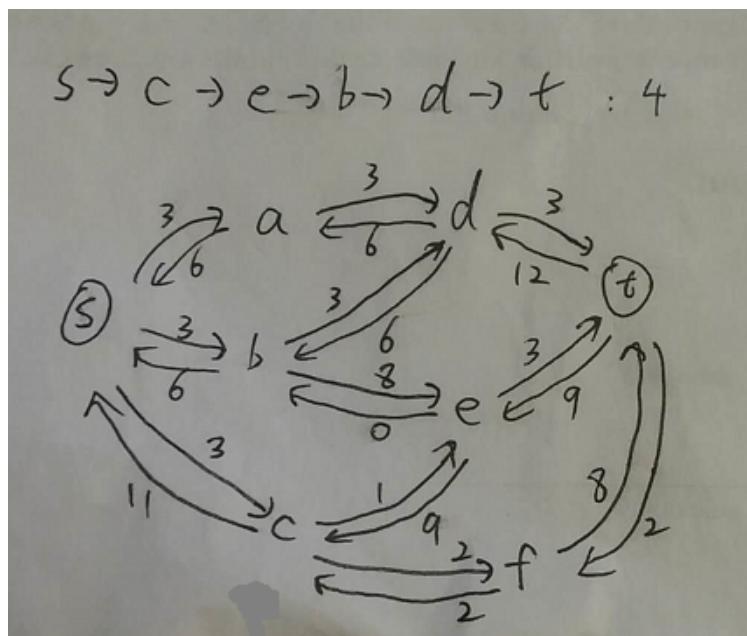
Miscalculated the value of edges (-1 ~ -5, depends on how many mistakes)

- b) Perform one additional step of the Ford-Fulkerson algorithm by choosing an augmenting path with the maximum bottleneck value. Show the residual graph at the end of this step of augmentation. (10 pts)

Solution:

Choose path $S \rightarrow c \rightarrow e \rightarrow b \rightarrow d \rightarrow t$ with bottleneck value of 4. Any other augmenting path cannot reach this value.

As shown in the graph below.



Rubric:

Correct residual graph, but doesn't tell us which is the path (-0.5 ~ -1)
 Miscalculate the value of edges or the directions (-1 ~ -3, depends on how many mistakes. Notice the direction of the edge between b and e)

Wrote a wrong augmentation path (value is not 4), and drew the residual graph (-5 ~ -6)

Only drew the wrong residual graph (-6 ~ -8)

5) 20 pts

We want to solve the problem of determining the set of states with the smallest total population that can provide the votes to win the election (electoral college). Formally, the problem can be described as:

Let p_i be the population of state i , and v_i the number of electoral votes for state i . If a candidate wins state i , he/she receives v_i electoral votes, and the winning candidate is the one who receives at least V electoral votes, where $V = \lfloor (\sum_i v_i)/2 \rfloor + 1$. Our goal is to find a set of states S that minimizes the value of $\sum_{i \in S} p_i$ subject to the constraint that $\sum_{i \in S} v_i \geq V$.

Give a dynamic programming algorithm for finding the set S .

Hint: One way to solve this problem is to turn it into a 0/1 knapsack problem. If you choose this route, you still need to fill out all sections **a** through **e** as they relate to solving this particular problem. In other words, you still need to define subproblems, your recurrence formula, etc. in the context of this problem.

a) Define (in plain English) subproblems to be solved. (4 pts)

Missing some information (-1 pt, each)

Solution 1: use 0/1 knapsack to solve the problem

$\text{OPT}(i,w) = \max$ population of states $1..i$ such that their total v_i **does not exceed** w .

We are basically trying to find those states with the highest total population whose total v_i does not exceed $W = \sum v_i - V$. This set is the complement of the set that the problem statement is asking for.

b) Write the recurrence relation for subproblems. (7 pts)

If one part of the recurrence is wrong (-2 pt, each)

We accept answer if only mention $\text{Max}(\text{OPT}(i-1,w-vi) + pi, \text{OPT}(i-1,w))$.

Same as in knapsack:

$$\begin{aligned} \text{OPT}(i,w) &= \text{OPT}(i-1, w) \text{ if } w < v_i , \\ &\text{Otherwise} \quad \text{Max}(\text{OPT}(i-1,w-vi) + pi, \text{OPT}(i-1,w)) \end{aligned}$$

c) Write pseudo-code to compute the optimal value of all unique subproblems.

(4 pts)

If the base cases / initialization are wrong (-2 pt)

If you do not consider the case if $W < vi$ (-2 pt)

If one part of the pseudo-code is wrong (-2 pt, each)

$$W = \sum v_i - V$$

Initialize $\text{OPT}(0,w) = 0$ for $w=0$ to W

```

For i=1 to n
    For w=0 to W
        If W < vi
            OPT(i,w)      = OPT(i-1, w)
        Otherwise
            OPT(i,w)      = Max( OPT(i-1,w-vi) + pi, OPT(i-1,w))
        Endif
    Endfor
Endfor
Return OPT(n,W)

```

- d) Write pseudo-code to find the set of states S. (3 pts)
If one part of the pseudo-code is wrong (-2 pt, each)

First find those states with the highest total population whose total v_i does not exceed
Initialize set S to NULL

$W = \sum v_i - V$
For i=n to 1 by -1
If ($OPT(i-1, W-vi) + pi > OPT(i-1, W)$) then
//or

If ($OPT(i-1, W-vi) + pi = OPT(i, W)$) then
Add i to the set of states S
 $W = W-vi$
Endif
Endfor

Our Solution $S' = \text{complement of the set } S$

- e) Is the solution you have provided an efficient solution? Give a brief explanation.
(2 pt)
The answer is correct (1 pt). The explanation is correct (1 pt)

No the solution runs in $O(nW)$ which is pseudo-polynomial with respect to the input size, since W represents the numerical value of the input and not the size of the input.

Solution 2: Solve directly

- a) Define (in plain English) subproblems to be solved. (4 pts)
Missing some information (-1 pt, each)

$OP T[i, v]$ = minimum populations of a set of states from $1, 2, \dots, i$ such that their votes sum to **exactly** v electoral votes.

- b) Write the recurrence relation for subproblems. (7 pts)

If one part of the recurrence is wrong (-2 pt, each)

We accept answer if only mention $\min\{OP T[i - 1, v], OP T[i - 1, v - vi] + pi\}$

$$OP T[i, v] = \begin{cases} OP T[i - 1, v] & \text{if } v < vi, \\ \min\{OP T[i - 1, v], OP T[i - 1, v - vi] + pi\} & \text{Otherwise} \end{cases}$$

- c) Write pseudo-code to compute the optimal value of all unique subproblems. (4 pts)

If the base cases / initialization are wrong (-2 pt)

If you do not consider the case if $v < vi$ (-2 pt)

If one part of the pseudo-code is wrong (-2 pt, each)

$$W = \sum v_i$$

The important thing here is the initialization:

$$OP T[0, 0] = 0$$

$$OP T[0, v] = \infty \text{ for } v \geq 1$$

For $i=1$ to n

 For $v=0$ to W

 If $v < vi$

$$OPT[i, v] = OPT[i - 1, v]$$

 Otherwise

$$OPT[i, v] = \min\{OPT[i - 1, v], OP T[i - 1, v - vi] + pi\}$$

 Endfor

Endfor

Return smallest $OPT[i, v]$ where $v \geq V$ and $i=n$.

- d) Write pseudo-code to find the set of states S . (3 pts)

If one part of the pseudo-code is wrong (-2 pt, each)

Initialize set S to NULL

Let $W = v$, where $OPT[n, v]$ was the optimal value of the solution found in part c

For $i=n$ to 1 by -1

```
If ( OPT(i-1,W-vi) + pi < OPT(i-1,W) ) then  
//or  
If ( OPT(i-1, W-vi) + pi = OPT(i,W) ) then  
    Add i to the set of states S  
    W = W-vi  
Endif  
Endfor
```

Our solution is the set S

- e) Is the solution you have provided an efficient solution? Give a brief explanation. (2 pt)
The answer is correct (1 pt). The explanation is correct (1 pt)

No, similar to solution 1.

No the solution runs in $O(nW)$ which is pseudo-polynomial with respect to the input size, since W represents the numerical value of the input and not the size of the input.

CSCI 570 - Spring 2019 - HW 6

Due Feb 28 at 11:59 p.m.

Note This homework assignment covers dynamic programming. It is recommended that you read chapter 6.1 to 6.4 from Klienberg and Tardos.

1 Graded Problems

1. The 0-1 knapsack problem where each item is unique and only one of each kind is available. Now let us consider knapsack problem where you have limited many items of each kind. Namely, there are n different types of items. All the items of the same type i have equal size w_i and value v_i . You are offered with m_i many items of each type. Design a dynamic programming algorithm to compute the optimal value you can get from a knapsack with capacity W .

Solution 1. Transform to 0-1 knapsack problem:

We can treat each allowed item under one type as m_i different kinds of item. Then we use the 0-1 knapsack's DP solution.

Solution 2. Similar to what is taught in the lecture but add another dimension, let $OPT(k, w, u_k)$ be the maximum value achievable using a knapsack of capacity $0 \leq w \leq W$ and with k types of items $1 \leq k \leq n$ and u_k numbers of item k ($1 \leq u_k \leq m_k$). We find the recurrence relation of $OPT(k, w, u_k)$ as follows. Since we have infinitely many items of each type, we choose between the following two cases:

- when $u_k \geq 1$ We may include another item of type k and solve the sub-problem $OPT(k, w - v_k, u_k - 1)$.
- We do not include any item of type k and move to consider next type of item this solving the sub-problem $OPT(k - 1, w, m_{k-1})$.

Therefore, we have

$$OPT(k, w) = \max\{OPT(k - 1, w, m_{k-1}), OPT(k, w - w_k, u_k - 1) + v_k\}.$$

Moreover, we have the initial condition $OPT(0, 0, 0) = 0$.

2. This is a famous puzzle involving $n=2$ eggs and a building with $k=36$ floors. Suppose that we wish to know which stories in a 36-story building are safe to drop eggs from, and which will cause the eggs to break on landing. We make a few assumptions:

- *An egg that survives a fall can be used again.
- *A broken egg must be discarded.
- *The effect of a fall is the same for all eggs.
- *If an egg breaks when dropped, then it would break if dropped from a higher floor.
- *If an egg survives a fall then it would survive a shorter fall.
- *It is not ruled out that the first-floor windows break eggs, nor is it ruled out that the 36th-floor do not cause an egg to break.

If only one egg is available and we wish to be sure of obtaining the right result, the experiment can be carried out in only one way. Drop the egg from the first-floor window; if it survives, drop it from the second floor window. Continue upward until it breaks. In the worst case, this method may require 36 droppings. Suppose 2 eggs are available. What is the least number of egg-droppings that is guaranteed to work in all cases? The problem is not actually to find the critical floor, but merely to decide floors from which eggs should be dropped so that total number of trials are minimized.

Solution:

When we drop an egg from a floor x , there can be two cases (1) The egg breaks (2) The egg doesn't break.

1) If the egg breaks after dropping from x th floor, then we only need to check for floors lower than x with remaining eggs; so the problem reduces to $x-1$ floors and $n-1$ eggs 2) If the egg doesn't break after dropping from the x th floor, then we only need to check for floors higher than x ; so the problem reduces to $k-x$ floors and n eggs.

Since we need to minimize the number of trials in worst case, we take the maximum of two cases. We consider the max of above two cases for every floor and choose the floor which yields minimum number of trials.

$k \Rightarrow$ Number of floors

$n \Rightarrow$ Number of Eggs

$\text{eggDrop}(n, k) \Rightarrow$ Minimum number of trials needed to find the critical floor in worst case.

$\text{eggDrop}(n, k) = 1 + \min\max(\text{eggDrop}(n - 1, x - 1), \text{eggDrop}(n, k - x))$:
 x in $1, 2, \dots, k$

3. You are given a set of n types of rectangular 3-D boxes, where the i^{th} box has height $h(i)$, width $w(i)$ and depth $d(i)$ (all real numbers). You want to

create a stack of boxes which is as tall as possible, but you can only stack a box on top of another box if the dimensions of the 2-D base of the lower box are each strictly larger than those of the 2-D base of the higher box. Of course, you can rotate a box so that any side functions as its base. It is also allowable to use multiple instances of the same type of box.

Following are the key points to note in the problem statement:

- 1) A box can be placed on top of another box only if both width and depth of the upper placed box are smaller than width and depth of the lower box respectively.
- 2) We can rotate boxes such that width is smaller than depth. For example, if there is a box with dimensions 1x2x3 where 1 is height, 23 is base, then there can be three possibilities, 1x2x3, 2x1x3 and 3x1x2
- 3) We can use multiple instances of boxes. What it means is, we can have two different rotations of a box as part of our maximum height stack.

Solution:

- 1) Generate all 3 rotations of all boxes. The size of rotation array becomes 3 times the size of original array. For simplicity, we consider depth as always smaller than or equal to width.
- 2) Sort the above generated $3n$ boxes in decreasing order of base area.
- 3) After sorting the boxes, the problem is same as LIS with following optimal substructure property. $MSH(i)$ = Maximum possible Stack Height with box i at top of stack $MSH(i) = \max(MSH(j)) + \text{height}(i)$ where $j < i$ and $\text{width}(j) > \text{width}(i)$ and $\text{depth}(j) > \text{depth}(i)$. If there is no such j then $MSH(i) = \text{height}(i)$
- 4) To get overall maximum height, we return $\max(MSH(i))$ where $0 < i < n$
4. Solve Kleinberg and Tardos, Chapter 6, Exercise 5.

Let $Y_{i,k}$ denote the substring $y_i y_{i+1} \dots y_k$. Let $Opt(k)$ denote the quality of an optimal segmentation of the substring $Y_{1,k}$. An optimal segmentation of this substring $Y_{1,k}$ will have quality equalling the quality last word (say $y_i \dots y_k$) in the segmentation plus the quality of an optimal solution to the substring $Y_{1,i}$. Otherwise we could use an optimal solution to $Y_{1,i}$ to improve $Opt(k)$ which would lead to a contradiction.

$$Opt(k) = \max_{0 < i < k} Opt(i) + \text{quality}(Y_{i+1,k})$$

We can begin solving the above recurrence with the initial condition that $Opt(0) = 0$ and then go on to compute $Opt(k)$ for $k = 1, 2, \dots, n$ keeping

track of where the segmentation is done in each case. The segmentation corresponding to $Opt(n)$ is the solution and can be computed in $O(n^2)$ time.

2 Practice Problems

5. Solve Kleinberg and Tardos, Chapter 6, Exercise 6.

Let $W = \{w_1, w_2, \dots, w_n\}$ be the set of ordered words which we wish to print. In the optimal solution, if the first line contains k words, then the rest of the lines constitute an optimal solution for the sub problem with the set $\{w_{k+1}, \dots, w_n\}$. Otherwise, by replacing with an optimal solution for the rest of the lines, we would get a solution that contradicts the optimality of the solution for the set $\{w_1, w_2, \dots, w_n\}$.

Let $Opt(i)$ denote the sum of squares of slacks for the optimal solution with the words $\{w_i, \dots, w_n\}$. Say we can put at most the first p words from w_i to w_n in a line, that is, $\sum_{t=i}^{p+i-1} c_t + p - 1 \leq L$ and $\sum_{t=1}^{p+i} w_t + p > L$. Suppose the first k words are put in the first line, then the number of extra space characters is

$$s(i, k) := L - k + 1 - \sum_{t=i}^{i+k-1} c_t$$

So we have the recurrence

$$Opt(i) = \begin{cases} 0 & \text{if } p \geq n - i + 1 \\ \min_{1 \leq k \leq p} \{(s(i, k))^2 + Opt(i + k)\} & \text{if } p < n - i + 1 \end{cases}$$

Trace back the value of k for which $Opt(i)$ is minimized to get the number of words to be printed on each line. We need to compute $Opt(i)$ for n different values of i . At each step p may be asymptotically as big as L . Thus the total running time is $O(nL)$.

6. Solve Kleinberg and Tardos, Chapter 6, Exercise 10.

- (a) Consider the following example: there are totally 4 minutes, the numbers of steps that can be done respectively on the two machines in the 4 minutes are listed as follows (in time order):

- Machine A: 2, 1, 1, 200
- Machine B: 1, 1, 20, 100

The given algorithm will choose A then move, then stay on B for the final two steps. The optimal solution will stay on A for the four steps.

- (b) An observation is that, in the optimal solution for the time interval from minute 1 to minute i , you should not move in minute i , because otherwise, you can keep staying on the machine where you are and get a better solution ($a_i > 0$ and $b_i > 0$). For the time interval from minute 1 to minute i , consider that if you are on machine A in minute i , you either (i) stay on machine A in minute $i - 1$ or (ii) are in the process of moving from machine B to A in minute $i - 1$. Now let $OPT_A(i)$ represent the maximum value of a plan in minute 1 through i that ends on machine A, and define $OPT_B(i)$ analogously for B. If case (i) is the best action to make for minute i , we have $OPT_A(i) = a_i + OPT_A(i - 1)$; otherwise, we have $OPT_A(i) = a_i + OPT_B(i - 2)$. In sum, we have

$$OPT_A(i) = a_i + \max\{OPT_A(i - 1), OPT_B(i - 2)\} :$$

Similarly, we get the recursive relation for $OPT_B(i)$:

$$OPT_B(i) = b_i + \max\{OPT_B(i - 1), OPT_A(i - 2)\} :$$

The algorithm initializes $OPT_A(0) = 0$, $OPT_B(0) = 0$, $OPT_A(1) = a_1$ and $OPT_B(1) = b_1$. Then the algorithm can be written as follows:

```

 $OPT_A(0) = 0; OPT_B(0) = 0;$ 
 $OPT_A(1) = a_1; OPT_B(1) = b_1;$ 
for  $i = 2, \dots, n$  do
     $OPT_A(i) = a_i + \max\{OPT_A(i - 1), OPT_B(i - 2)\};$ 
    Record the action (either stay or move) in minute  $i - 1$  that achieves the
    maximum.
     $OPT_B(i) = b_i + \max\{OPT_B(i - 1), OPT_A(i - 2)\};$ 
    Record the action in minute  $i - 1$  that achieves the maximum.
end for
Return  $\max\{OPT_A(n), OPT_B(n)\}$ ;
Track back through the arrays  $OPT_A$  and  $OPT_B$  by checking the action
records from minute  $n - 1$  to minute 1 to recover the optimal solution.

```

It takes $O(1)$ time to complete the operations in each iteration; there are $O(n)$ iterations; the tracing backs takes $O(n)$ time. Thus, the overall complexity is $O(n)$.

7. Solve Kleinberg and Tardos, Chapter 6, Exercise 24.

The basic idea is to ask: How should we gerrymander precincts 1 through j , for each j ? To make this work, though, we have to keep track of a few extra things, by adding some variables. For brevity, we say that A-votes in a precinct are the voters for part A and B-voter are the votes for part B. We keep track of the following information about a partial solution.

- How many precincts have been assigned to district 1 so far?
- How many A-votes are in district 1 so far?
- How many A-votes are in district 2 so far?

So let $M[j, p, x, y] = \text{true}$ if it is possible to achieve at least x A-votes in distance 1 and y A-votes in district 2, while allocating p of the first j precincts to district 1. Now suppose precinct $j + 1$ has z A-votes. To compute $M[j+1, p, x, y]$, you either put precinct $j+1$ in district 1 (in which case you check the results of sub-problem $M[j, p-1, x-z, y]$) or in district 2 (in which case you check the results of sub-problem $M[j, p, x, y-z]$). Now to decide if there's a solution to the while problem, you scan the entire table at the end, looking for a value of *true* in any entry from $M[n, n/2, x, y]$ where each of x and y is greater than $mn/4$. (Since each district gets $mn/2$ votes total).

We can build this up in the order of increasing j , and each sub-problem takes constant time to compute, using the values of smaller sub-problems. Since there are n^2, m^2 sub-problems, the running time is $O(n^2m^2)$.

1) 20 pts

Mark the following statements as **TRUE** or **FALSE**. No need to provide any justification.

[**TRUE/FALSE**]

It is possible for a dynamic programming algorithm to have an exponential running time.

[**TRUE/FALSE**]

In a connected, directed graph with positive edge weights, the Bellman-Ford algorithm runs asymptotically faster than the Dijkstra algorithm.

[**TRUE /FALSE**]

There exist some problems that can be solved by dynamic programming, but cannot be solved by greedy algorithm.

[**TRUE /FALSE**]

The Floyd-Warshall algorithm is asymptotically faster than running the Bellman-Ford algorithm from each vertex.

[**TRUE /FALSE**]

If we have a dynamic programming algorithm with n^2 subproblems, it is possible that the space usage could be $O(n)$.

[**TRUE /FALSE**]

The Ford-Fulkerson algorithm solves the maximum bipartite matching problem in polynomial time.

[**TRUE /FALSE**]

Given a solution to a max-flow problem, that includes the final residual graph G_f . We can verify in a *linear* time that the solution does indeed give a maximum flow.

[**TRUE /FALSE**]

In a flow network, a flow value is upper-bounded by a cut capacity.

[**TRUE/FALSE**]

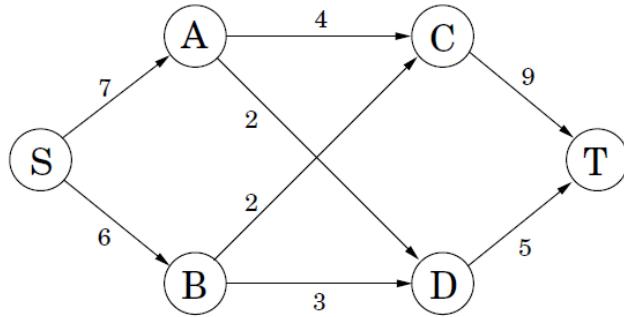
In a flow network, a min-cut is always unique.

[**TRUE/FALSE**]

A maximum flow in an integer capacity graph must have an integer flow on each edge.

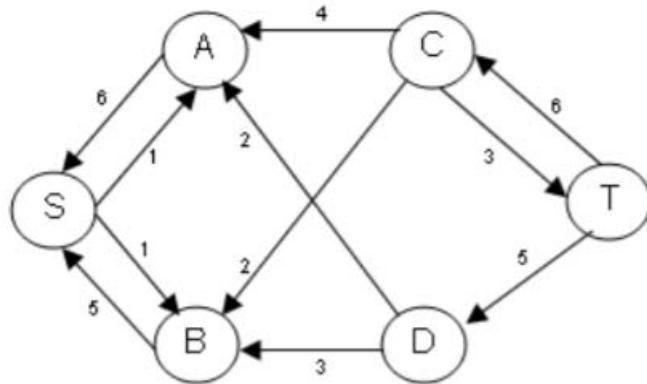
2) 20 pts.

You are given the following graph G . Each edge is labeled with the capacity of that edge.



- a) Find a max-flow in G using the Ford-Fulkerson algorithm. Draw the residual graph G_f corresponding to the max flow. You do not need to show all intermediate steps. (10 pts)

solution



Grading Rubics:

Residual graph: 10 points in total

For each edge in the residual graph, any wrong number marked, wrong direction, or missing will result in losing 1 point.

The total points you lose is equal to the number of edges on which you make any mistake shown above.

- b) Find the max-flow value and a min-cut. (6 pts)

Solution: $f = 11$, cut: ($\{S, A, B\}$, $\{C, D, T\}$)

Grading Rubics:

Max-flow value and min-cut: 6 points in total

Max-flow: 2 points.

- If you give the wrong value, you lose the 2 points.

Min-cut: 4 points

- If your solution forms a cut but not a min-cut for the graph, you lose 3 points
- If your solution does not even form a cut, you lose all the 4 points

- c) Prove or disprove that increasing the capacity of an edge that belongs to a min cut will always result in increasing the maximum flow. (4 pts)

Solution: increasing (B,D) by one won't increase the max-flow

Grading Rubics:

Prove or Disprove: 4 points in total

- If you judge it "True", but give a structural complete "proof". You get at most 1 point
- If you judge it "False", you get 2 points.
- If your counter example is correct, you get the rest 2 points.

Popular mistake: a number of students try to disprove it by showing that if the min-cut in the original graph is non-unique, then it is possible to find an edge in one min-cut set, such that increasing the capacity of this does not result in max-flow increase.

But they did not do the following thing:

The existence of the network with multiple min-cuts needs to be proved, though it seems to be obvious. The most straightforward way to prove the existence is to give an example-network that has multiple min-cuts. Then it turns out to be giving a counter example for the original question statement.

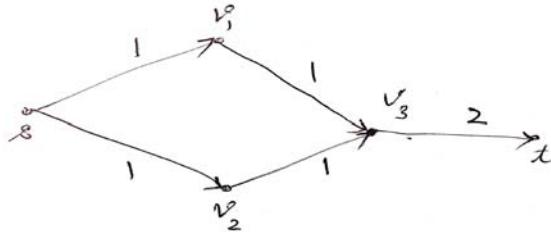
3) 15 pts.

Given a flow network with the source s and the sink t , and positive integer edge capacities c . Let (S, T) be a minimum cut. Prove or disprove the following statement:
If we increase the capacity of every edge by 1, then (S, T) still be a minimum cut.

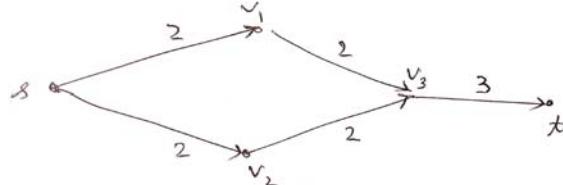
False. Create a counterexample.

An instance of a counter-example:

Initially, a (S, T) min cut is $S : \{s, v_1, v_2\}$ and $T : \{v_3, t\}$



After increasing capacity of every edge by 1, (S, T) is no longer a min-cut. We now have $S' : \{s, v_1, v_2, v_3\}$ and $T' : \{t\}$



Grading rubric:

If you try to prove the statement: -15

For an incorrect counter-example: -9

No credit for simply stating true or false.

4) 15 pts.

Given an unlimited supply of coins of denominations d_1, d_2, \dots, d_n , we wish to make change for an amount C . This might not be always possible. Your goal is to verify if it is possible to make such change. Design an algorithm by *reduction* to the knapsack problem.

- a) Describe reduction. What is the knapsack capacity, values and weights of the items? (10 pts.)

Capacity is C . (2 points)
values = weights = d_k . (4points)

- You need to recognize that the problem should be modeled based on the unbounded knapsack problem with description of the reduction: (5 points)
- Explanation of the verification criteria (4 points)
 $\text{Opt}(j)$: the maximum change equal or less than j that can be achieved with d_1, d_2, \dots, d_n
 $\text{Opt}(0)=0$
 $\text{Opt}(j) = \max[\text{opt}(j-d_i)+d_i] \text{ for } d_i \leq j$
If we obtain $\text{Opt}(C) = C$, it means the change is possible.

- b) Compute the runtime of your verification algorithm. (5 pts)

$O(nC)$ (3 points), Explanation (2 points).

5) 15 pts.

You are considering opening a series of electrical vehicle charging stations along Pacific Coast Highway (PCH). There are n possible locations along the highway, and the distance from the start to location k is $d_k \geq 0$, where $k = 1, 2, \dots, n$. You may assume that $d_i < d_k$ for $i < k$. There are two important constraints:

- 1) at each location k you can open only one charging station with the expected profit p_k , $k = 1, 2, \dots, n$.
- 2) you must open at least one charging station along the whole highway.
- 3) any two stations should be at least M miles apart.

Give a DP algorithm to find the maximum expected total profit subject to the given constraints.

- a) Define (in plain English) subproblems to be solved. (3 pts)

Let $\text{OPT}(k)$ be the maximum expected profit which you can obtain from locations $1, 2, \dots, k$.

Rubrics

Any other definition is okay as long as it will recursively solve subproblems

- b) Write the recurrence relation for subproblems. (6 pts)

$$\text{OPT}(k) = \max \{\text{OPT}(k - 1), p_k + \text{OPT}(f(k))\}$$

where $f(k)$ finds the largest index j such that $d_j \leq d_k - M$, when such j doesn't exist, $f(k)=0$

Base cases:

$$\text{OPT}(1) = p_1$$

$$\text{OPT}(0) = 0$$

Rubrics:

Error in recursion -2pts, if multiple errors, deduction adds up

No base cases: -2pts

Missing base cases: -1pts

Using variables without definition or explanation: -2pts

Overloading variables (re-defining n , p , k , or M): -2pts

- c) Compute the runtime of the above DP algorithm in terms of n . (3 pts)

algorithm solves n subproblems; each subproblem requires finding an index $f(k)$ which can be done in time $O(\log n)$ by binary search.
Hence, the running time is $O(n \log n)$.

Rubric:

$O(n^2)$ is regarded as okay

$O(d_n n)$ pseudo-polynomial is also okay if the recursion goes over all d_n values

$O(n)$ is also okay

$O(n^3), O(nM), O(kn)$: no credit

- d) How can you optimize the algorithm to have $O(n)$ runtime? (3 pts)

Preprocess distances $r_i = d_i - M$. Merge d-list with r-list.

Rubric

Claiming “optimal solution is already found in c ” only gets credit when explanation about pre-process is described in either part b or c.

Without proper explanation (e.g. assume we have, or we can do): no credit

Keeping an array of max profits: no credit, finding the index that is closest to the installed station with M distance away is the bottleneck, which requires pre-processing.

6) 15 pts.

A group of traders are leaving Switzerland, and need to convert their Francs into various international currencies. There are n traders t_1, t_2, \dots, t_n and m currencies c_1, c_2, \dots, c_m . Trader t_k has F_k Francs to convert. For each currency c_j , the bank can convert at most B_j Francs to c_j . Trader t_k is willing to trade as much as S_{kj} of his Francs for currency c_j . (For example, a trader with 1000 Francs might be willing to convert up to 200 of his Francs for USD, up to 500 of his Francs for Japanese's Yen, and up to 200 of his Francs for Euros). Assuming that all traders give their requests to the bank at the same time, describe an algorithm that the bank can use to satisfy the requests (if it can).

a) Describe how to construct a flow network to solve this problem, including the description of nodes, edges, edge directions and their capacities. (8 pts)

Bipartite graph: one partition traders t_1, t_2, \dots, t_n . Other, available currency, c_1, c_2, \dots, c_m .

Connect t_k to c_j with the capacity S_{kj}

Connect source to traders with the capacity F_k .

Connect available currency c_j to the sink with the capacity B_j .

Rubrics:

- Didn't include supersource (-1 point)
- Didn't include traders nodes (-1 point)
- Didn't include currencies nodes (-1 point)
- Didn't include supersink (-1 point)
- Didn't include edge direction (-1 point)
- Assigned no/wrong capacity on edges between source & traders (-1 point)
- Assigned no/wrong capacity on edges between traders & currencies (-1 point)
- Assigned no/wrong capacity on edges between currencies & sink (-1 point)

b) Describe on what condition the bank can satisfy all requests. (4 pts)

If there is a flow f in the network with $|f| = F_1 + \dots + F_n$, then all traders are able to convert their currencies.

Rubrics:

- Wrong condition (-4 points): Unless you explicitly included $|f| = F_1 + \dots + F_n$, no partial points were given for this subproblem.
- In addition to correct answer, added additional condition which is wrong (-2 points)
- No partial points were given for conditions that satisfy only some of the requests, not all requests.
- Note that $\sum S_{kj}$ and $\sum B_j$ can be larger than $\sum F_k$ in some cases where bank satisfy all requests.
- Note that $\sum S_{kj}$ can be larger than $\sum B_j$ in some cases where bank satisfy all requests.
- It is possible that $\sum S_{kj}$ or $\sum B_j$ is smaller than $\sum F_k$. In these cases, bank can never satisfy all requests because max flow will be smaller than $\sum F_k$.

- c) Assume that you execute the Ford-Fulkerson algorithm on the flow network graph in part a), what would be the runtime of your algorithm? (3 pts)

$O(n m |f|)$

Rubrics:

- Flow($|f|$) was not included (-1 point)
- Wrong description (-1 point)
- Computation is incorrect (-1 point)
- Notation error (-1 point)
- Missing big O notation (-1 point)
- Used big Theta notation instead of big O notation (-1 point)
- Wrong runtime complexity (-3 points)
- No runtime complexity is given (-3 points)

1) 20 pts

Mark the following statements as **TRUE** or **FALSE**. No need to provide any justification.

[/FALSE]

If an iteration of the Ford-Fulkerson algorithm on a network places flow 1 through an edge (u, v) , then in every later iteration, the flow through (u, v) is at least 1.

[TRUE/]

For the recursion $T(n) = 4T(n/3) + n$, the size of each subproblem at depth k of the recursion tree is $n/3^{k-1}$.

[/FALSE]

For any flow network G and any maximum flow on G , there is always an edge e such that increasing the capacity of e increases the maximum flow of the network.

[/FALSE]

The asymptotic bound for the recurrence $T(n) = 3T(n/9) + n$ is given by $\Theta(n^{1/2} \log n)$.

[/FALSE]

Any Dynamic Programming algorithm with n subproblems will run in $O(n)$ time.

[/FALSE]

A pseudo-polynomial time algorithm is always slower than a polynomial time algorithm.

[TRUE/]

The sequence alignment algorithm can be used to find the longest common subsequence between two given sequences.

[/FALSE]

If a dynamic programming solution is set up correctly, i.e. the recurrence equation is correct and each unique sub-problem is solved only once (memoization), then the resulting algorithm will always find the optimal solution in polynomial time.

[TRUE/]

For a divide and conquer algorithm, it is possible that the divide step takes longer to do than the combine step.

[TRUE/]

Maximum value of an $s - t$ flow could be less than the capacity of a given $s - t$ cut in a flow network.

2) 16 pts

Recall the Bellman-Ford algorithm described in class where we computed the shortest distance from all points in the graph to t. And recall that we were able to find all shortest distance to t with only $O(n)$ memory.

How would you extend the algorithm to compute both the shortest distance and to find the actual shortest paths from all points to t with only $O(n)$ memory?

We need an array of size n to hold a pointer to the neighbor that gives us the shortest distance to t. Initially all pointers are set to Null. Whenever a node's distance to t is reduced, we update the pointer for that node and point it to the node that is giving us a lower distance to t. Once all shortest distances are computed, to find a path from any node v to t, one can simply follow these pointers to reach t on the shortest path.

3) 16 pts

During their studies, 7 friends (Alice, Bob, Carl, Dan, Emily, Frank, and Geoffrey) live together in a house. They agree that each of them has to cook dinner on exactly one day of the week. However, assigning the days turns out to be a bit tricky because each of the 7 students is unavailable on some of the days. Specifically, they are unavailable on the following days (1 = Monday, 2 = Tuesday, ..., 7 = Sunday):

- Alice: 2, 3, 4, 5
- Bob: 1, 2, 4, 7
- Carl: 3, 4, 6, 7
- Dan: 1, 2, 3, 5, 6
- Emily: 1, 3, 4, 5, 7
- Frank: 1, 2, 3, 5, 6
- Geoffrey: 1, 2, 5, 6

Transform the above problem into a maximum flow problem and draw the resulting flow network. If a solution exists, the flow network should indicate who will cook on each day; otherwise it must show that a feasible solution does not exist

Solution:

I will use the initials of each person's name to refer to them in this solution.

Construct a graph $G = (V, E)$. V consists of 1 node for each person (let us denote this set by $P = \{A, B, C, D, E, F, G\}$), 1 node for each day of the week (let's call this set $D = \{1, 2, 3, 4, 5, 6, 7\}$), a source node s , and a sink node t . Connect s to each node p in P by a directed (s, p) edge of unit capacity. Similarly, connect each node d in D to t by a directed (d, t) edge of unit capacity. Connect each node p in P by a directed edge of unit capacity to those nodes in D when p is **available** to cook. This completes our construction of the flow network. I am omitting the actual drawing of G here.

Finding a max-flow of value 7 in G translates to finding a feasible solution to the allocation of cooking days problem. Since there can be at most unit flow coming into any node p in P , a maximum of unit flow can leave it. Similarly, at most a flow of value 1 can flow into any node d in D because a maximum of unit flow can leave it. Thus, a max-flow of value 7 means that there exists a flow-carrying s - p - d - t path for each p and d . Any (p, d) edge with unit flow indicates that person p will cook on day d .

The following lists one possible max-flow of value 7 in G :

Send unit flow on each (s, p) edge and each (d, t) edge. Also send unit flow on the following (p, d) edges: $(A, 6)$, $(B, 5)$, $(C, 1)$, $(D, 4)$, $(E, 2)$, $(F, 7)$, $(G, 3)$

4) 20 pts

Suppose that there are n asteroids that are headed for earth. Asteroid i will hit the earth in time t_i and cause damage d_i unless it is shattered before hitting the earth, by a laser beam of energy e_i . Engineers at NASA have designed a powerful laser weapon for this purpose. However, the laser weapon needs to charge for a duration ce before firing a beam of energy e . Can you design a dynamic programming based pseudo-polynomial time algorithm to decide on a firing schedule for the laser beam to minimize the damage to earth? Assume that the laser is initially uncharged and the quantities c, t_i, d_i, e_i are all positive integers. Analyze the running time of your algorithm. You should include a brief description/derivation of your recurrence relation. Description of recurrence relation = 8pts, Algorithm = 6pts, Run Time = 6pts

Solution 1 (Assuming that the laser retains energy between firing beams): Sort all asteroids by the time t_i . Label the asteroids from 1 to n and without loss of generality assume $t_1 < t_2 < \dots < t_n$. Also assume that if asteroid i is destroyed then it is done exactly at time t_i (if the laser continuously accumulates energy then the destruction order of the asteroids does not change even if i^{th} asteroid is shot down before time t_i).

Define $OPT(i, T)$ as the minimum possible damage caused to earth due to asteroids $i, i + 1, \dots, n$ if $\frac{T}{c}$ energy is left in the laser just before time t_i . We want the solution corresponding to $OPT(1, t_1)$. If $T \geq ce_i$ and the i^{th} asteroid is destroyed then $OPT(i, T) = OPT(i + 1, T - ce_i + t_{i+1} - t_i)$, otherwise $OPT(i, T) = d_i + OPT(i + 1, T + t_{i+1} - t_i)$. Hence,

$$OPT(i, T) = \begin{cases} d_i + OPT(i + 1, T + t_{i+1} - t_i), & T < ce_i \\ \min\{d_i + OPT(i + 1, T + t_{i+1} - t_i), OPT(i + 1, T - ce_i + t_{i+1} - t_i)\}, & T \geq ce_i \end{cases}$$

Boundary condition: $OPT(n, T) = 0$ if $T \geq ce_n$ and $OPT(n, T) = d_n$ if $T < ce_n$, since if there is enough energy left to destroy the last asteroid then it is always beneficial to do so.

Furthermore, $T \leq t_n$ since a maximum of $\frac{t_n}{c}$ energy is accumulated by the laser before the last asteroid hits the earth and $T \geq 0$ since the left over energy in the laser is always non-negative.

Algorithm:

- i. Initialize $OPT(n, T)$ according to the boundary condition above for $0 \leq T \leq t_n$.
- ii. For each $n - 1 \geq i \geq 1$ and $0 \leq T \leq t_n$ populate $OPT(i, T)$ according to the recurrence defined above.
- iii. Trace forward through the two dimensional OPT array starting at $(1, t_1)$ to determine the firing sequence. For $1 \leq i \leq n - 1$, destroy the i^{th} asteroid with $\frac{T}{c}$ energy left if and only if $OPT(i, T) = OPT(i + 1, T - ce_i + t_{i+1} - t_i)$. Destroy the n^{th} asteroid only if $T \geq ce_n$.

Complexity: Step (i) initializes t_n values each taking constant time. Step (ii) computes $(n - 1)t_n$ values, each taking one invocation of the recurrence and hence is done in $O(nt_n)$ time. Trace back takes $O(n)$ time since the decision for each i takes constant time. Initial sorting takes $O(n \log n)$ time. Thus overall complexity is $O(nt_n + n \log n)$.

Solution 2 (Assuming that the laser does not retain energy left over after firing and if asteroid i is destroyed then it is done exactly at time t_i): Sort all asteroids by the time t_i . Label the asteroids from 1 to n and without loss of generality assume $t_1 < t_2 < \dots < t_n$. In contrast to Solution 1, the destroying sequence will change if we are free to destroy asteroid i before time t_i (this case is not solved here).

Define $OPT(i)$ to be the minimum possible damage caused to earth due to the first i asteroids. We want the solution corresponding to $OPT(n)$. If the i^{th} asteroid is not destroyed either by choice or because $t_i < ce_i$ then $OPT(i) = d_i + OPT(i - 1)$. On the other hand, if the i^{th} asteroid is destroyed ($t_i \geq ce_i$ is necessary) then none of the asteroids arriving between times $t_i - ce_i$ and t_i (both exclusive) can be destroyed. Letting $p[i]$ denote the largest positive integer such that $t_{p[i]} \leq t_i - ce_i$ (and $p[i] = 0$ if no such integer exists), we have $OPT(i) = OPT(p[i]) + \sum_{j=p[i]+1}^{i-1} d_j$ if $p[i] \leq i - 2$ and $OPT(i) = OPT(i - 1)$ if $p[i] = i - 1$. Hence for $i \geq 1$,

$$OPT(i) = \begin{cases} OPT(i - 1), & t_i \geq ce_i \text{ and } p[i] = i - 1 \\ d_i + OPT(i - 1), & t_i < ce_i \\ \min \left\{ d_i + OPT(i - 1), OPT(p[i]) + \sum_{j=p[i]+1}^{i-1} d_j \right\}, & t_i \geq ce_i \text{ and } p[i] \leq i - 2 \end{cases}$$

Boundary condition: $OPT(0) = 0$ since no asteroids means no damage.

Algorithm:

- i. Form the array p element-wise. This is done as follows.
 - a. Set $p[1] = 0$.
 - b. For $2 \leq i \leq n$, binary search for $t_i - ce_i$ in the sorted array $t_1 < t_2 < \dots < t_n$. If $t_i - ce_i < t_1$ then set $p[i] = 0$ else record $p[i]$ as the index such that $t_{p[i]} \leq t_i - ce_i < t_{p[i]+1}$.
- ii. Form array D to store cumulative sum of damages. Set $D[0] = 0$ and $D[j] = D[j - 1] + d_j$ for $1 \leq j \leq n$.
- iii. Set $OPT(0) = 0$ and for $1 \leq i \leq n$ populate $OPT(i)$ according to the recurrence defined above, computing $\sum_{j=p[i]+1}^{i-1} d_j$ as $D[i - 1] - D[p[i]]$.
- iv. Trace back through the one dimensional OPT array starting at $OPT(n)$ to determine the firing sequence. Destroy the first asteroid if and only if $OPT(1) = 0$. For $2 \leq i \leq n$, destroy the i^{th} asteroid if and only if either $OPT(i) = OPT(i - 1)$ with $p[i] = i - 1$ or $OPT(i) = OPT(p[i]) + D[i - 1] - D[p[i]]$ with $p[i] \leq i - 2$.

Complexity: initial sorting takes $O(n \log n)$. Construction of array p takes $O(\log n)$ time for each index and hence a total of $O(n \log n)$ time. Forming array D is done in $O(n)$ time. Using array D , $OPT(i)$ can be populated in constant time for each $1 \leq i \leq n$ and hence step (iii) takes $O(n)$ time. Trace back takes $O(n)$ time since the decision for each i takes constant time. Therefore, overall complexity is $O(n \log n)$.

5) 16 pts

Consider a two-dimensional array $A[1:n, 1:n]$ of integers. In the array each row is sorted in ascending order and each column is also sorted in ascending order. Our goal is to determine if a given value x exists in the array.

- a. One way to do this is to call binary search on each row (alternately, on each column). What is the running time of this approach? [2 pts]
 - b. Design another divide-and-conquer algorithm to solve this problem, and state the runtime of your algorithm. Your algorithm should take strictly less than $O(n^2)$ time to run, and should make use of the fact that each row and each column is in sorted order (i.e., don't just call binary search on each row or column). State the run-time complexity of your solution.
- a) $O(n \log n)$.
- b) Look at the middle element of the full matrix. Based on this, you can either eliminate $A[1.. \frac{n}{2}, 1.. \frac{n}{2}]$ or $A[\frac{n}{2}..n, \frac{n}{2}..n]$. If x is less than middle element then you can eliminate $A[\frac{n}{2}..n, \frac{n}{2}..n]$. If x is greater than middle element then you can eliminate $A[1.. \frac{n}{2}, 1.. \frac{n}{2}]$. You can then recursively search in the remaining three $\frac{n}{2} \times \frac{n}{2}$ matrices. The total runtime is $T(n) = 3T(\frac{n}{2}) + O(1)$, $T(n) = O(n^{\log_2 3})$.

6) 12 pts

Consider a divide-and-conquer algorithm that splits the problem of size n into 4 sub-problems of size $n/2$. Assume that the divide step takes $O(n^2)$ to run and the combine step takes $O(n^2 \log n)$ to run on problem of size n . Use any method that you know of to come up with an upper bound (as tight as possible) on the cost of this algorithm.

Solution: Use the generalized case 2 of Master's Theorem. For $T(n) = aT\left(\frac{n}{b}\right) + \Theta(n^{\log_b a} \log^k n)$ with $k \geq 0$, we have $T(n) = \Theta(n^{\log_b a} \log^{k+1} n)$.

The divide and combine steps together take $O(n^2 \log n)$ time and the worst case is that they actually take $\Theta(n^2 \log n)$ time. Hence the recurrence for the given algorithm is $T(n) = 4T\left(\frac{n}{2}\right) + \Theta(n^2 \log n)$ in the worst case. Comparing with the generalized case, $a = 4, b = 2, k = 1$ and so $T(n) = \Theta(n^2 \log^2 n)$. Since this expression for $T(n)$ is the worst case running time, an upper bound on the running time is $O(n^2 \log^2 n)$.

1) 20 pts

Mark the following statements as **TRUE** or **FALSE**. No need to provide any justification.

[**TRUE/FALSE**]

It is possible for a dynamic programming algorithm to have an exponential running time.

[**TRUE/FALSE**]

In a connected, directed graph with positive edge weights, the Bellman-Ford algorithm runs asymptotically faster than the Dijkstra algorithm.

[**TRUE /FALSE**]

There exist some problems that can be solved by dynamic programming, but cannot be solved by greedy algorithm.

[**TRUE /FALSE**]

The Floyd-Warshall algorithm is asymptotically faster than running the Bellman-Ford algorithm from each vertex.

[**TRUE /FALSE**]

If we have a dynamic programming algorithm with n^2 subproblems, it is possible that the space usage could be $O(n)$.

[**TRUE /FALSE**]

The Ford-Fulkerson algorithm solves the maximum bipartite matching problem in polynomial time.

[**TRUE /FALSE**]

Given a solution to a max-flow problem, that includes the final residual graph G_f . We can verify in a *linear* time that the solution does indeed give a maximum flow.

[**TRUE /FALSE**]

In a flow network, a flow value is upper-bounded by a cut capacity.

[**TRUE/FALSE**]

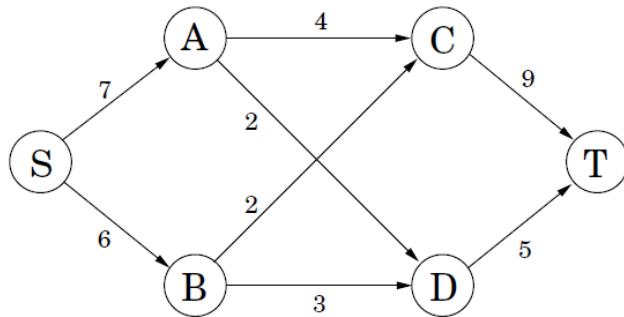
In a flow network, a min-cut is always unique.

[**TRUE/FALSE**]

A maximum flow in an integer capacity graph must have an integer flow on each edge.

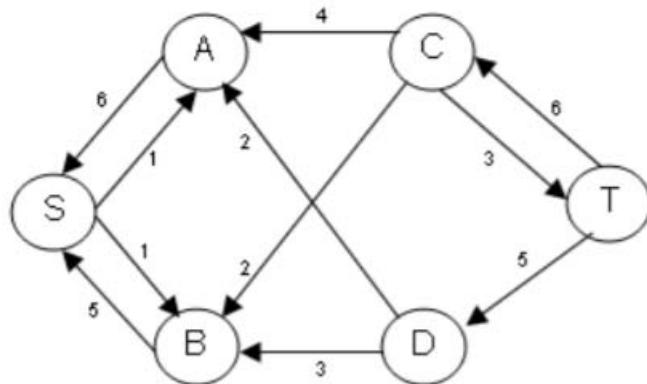
2) 20 pts.

You are given the following graph G . Each edge is labeled with the capacity of that edge.



- a) Find a max-flow in G using the Ford-Fulkerson algorithm. Draw the residual graph G_f corresponding to the max flow. You do not need to show all intermediate steps. (10 pts)

solution



Grading Rubics:

Residual graph: 10 points in total

For each edge in the residual graph, any wrong number marked, wrong direction, or missing will result in losing 1 point.

The total points you lose is equal to the number of edges on which you make any mistake shown above.

- b) Find the max-flow value and a min-cut. (6 pts)

Solution: $f = 11$, cut: ($\{S, A, B\}$, $\{C, D, T\}$)

Grading Rubics:

Max-flow value and min-cut: 6 points in total

Max-flow: 2 points.

- If you give the wrong value, you lose the 2 points.

Min-cut: 4 points

- If your solution forms a cut but not a min-cut for the graph, you lose 3 points
- If your solution does not even form a cut, you lose all the 4 points

- c) Prove or disprove that increasing the capacity of an edge that belongs to a min cut will always result in increasing the maximum flow. (4 pts)

Solution: increasing (B,D) by one won't increase the max-flow

Grading Rubics:

Prove or Disprove: 4 points in total

- If you judge it "True", but give a structural complete "proof". You get at most 1 point
- If you judge it "False", you get 2 points.
- If your counter example is correct, you get the rest 2 points.

Popular mistake: a number of students try to disprove it by showing that if the min-cut in the original graph is non-unique, then it is possible to find an edge in one min-cut set, such that increasing the capacity of this does not result in max-flow increase.

But they did not do the following thing:

The existence of the network with multiple min-cuts needs to be proved, though it seems to be obvious. The most straightforward way to prove the existence is to give an example-network that has multiple min-cuts. Then it turns out to be giving a counter example for the original question statement.

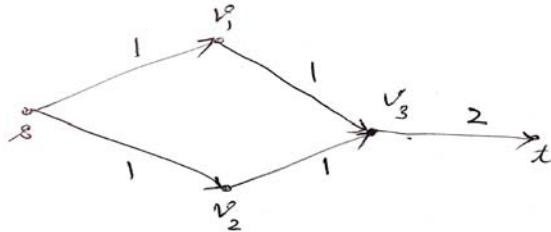
3) 15 pts.

Given a flow network with the source s and the sink t , and positive integer edge capacities c . Let (S, T) be a minimum cut. Prove or disprove the following statement:
If we increase the capacity of every edge by 1, then (S, T) still be a minimum cut.

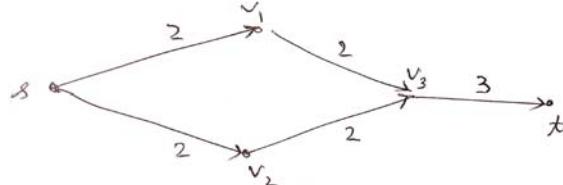
False. Create a counterexample.

An instance of a counter-example:

Initially, a (S, T) min cut is $S : \{s, v_1, v_2\}$ and $T : \{v_3, t\}$



After increasing capacity of every edge by 1, (S, T) is no longer a min-cut. We now have $S' : \{s, v_1, v_2, v_3\}$ and $T' : \{t\}$



Grading rubric:

If you try to prove the statement: -15

For an incorrect counter-example: -9

No credit for simply stating true or false.

4) 15 pts.

Given an unlimited supply of coins of denominations d_1, d_2, \dots, d_n , we wish to make change for an amount C . This might not be always possible. Your goal is to verify if it is possible to make such change. Design an algorithm by *reduction* to the knapsack problem.

- a) Describe reduction. What is the knapsack capacity, values and weights of the items? (10 pts.)

Capacity is C . (2 points)
values = weights = d_k . (4points)

- You need to recognize that the problem should be modeled based on the unbounded knapsack problem with description of the reduction: (5 points)
- Explanation of the verification criteria (4 points)
 $\text{Opt}(j)$: the maximum change equal or less than j that can be achieved with d_1, d_2, \dots, d_n
 $\text{Opt}(0)=0$
 $\text{Opt}(j) = \max[\text{opt}(j-d_i)+d_i] \text{ for } d_i \leq j$
If we obtain $\text{Opt}(C) = C$, it means the change is possible.

- b) Compute the runtime of your verification algorithm. (5 pts)

$O(nC)$ (3 points), Explanation (2 points).

5) 15 pts.

You are considering opening a series of electrical vehicle charging stations along Pacific Coast Highway (PCH). There are n possible locations along the highway, and the distance from the start to location k is $d_k \geq 0$, where $k = 1, 2, \dots, n$. You may assume that $d_i < d_k$ for $i < k$. There are two important constraints:

- 1) at each location k you can open only one charging station with the expected profit p_k , $k = 1, 2, \dots, n$.
- 2) you must open at least one charging station along the whole highway.
- 3) any two stations should be at least M miles apart.

Give a DP algorithm to find the maximum expected total profit subject to the given constraints.

- a) Define (in plain English) subproblems to be solved. (3 pts)

Let $\text{OPT}(k)$ be the maximum expected profit which you can obtain from locations $1, 2, \dots, k$.

Rubrics

Any other definition is okay as long as it will recursively solve subproblems

- b) Write the recurrence relation for subproblems. (6 pts)

$$\text{OPT}(k) = \max \{\text{OPT}(k - 1), p_k + \text{OPT}(f(k))\}$$

where $f(k)$ finds the largest index j such that $d_j \leq d_k - M$, when such j doesn't exist, $f(k)=0$

Base cases:

$$\text{OPT}(1) = p_1$$

$$\text{OPT}(0) = 0$$

Rubrics:

Error in recursion -2pts, if multiple errors, deduction adds up

No base cases: -2pts

Missing base cases: -1pts

Using variables without definition or explanation: -2pts

Overloading variables (re-defining n , p , k , or M): -2pts

- c) Compute the runtime of the above DP algorithm in terms of n . (3 pts)

algorithm solves n subproblems; each subproblem requires finding an index $f(k)$ which can be done in time $O(\log n)$ by binary search.
Hence, the running time is $O(n \log n)$.

Rubric:

$O(n^2)$ is regarded as okay

$O(d_n n)$ pseudo-polynomial is also okay if the recursion goes over all d_n values

$O(n)$ is also okay

$O(n^3), O(nM), O(kn)$: no credit

- d) How can you optimize the algorithm to have $O(n)$ runtime? (3 pts)

Preprocess distances $r_i = d_i - M$. Merge d-list with r-list.

Rubric

Claiming “optimal solution is already found in c ” only gets credit when explanation about pre-process is described in either part b or c.

Without proper explanation (e.g. assume we have, or we can do): no credit

Keeping an array of max profits: no credit, finding the index that is closest to the installed station with M distance away is the bottleneck, which requires pre-processing.

6) 15 pts.

A group of traders are leaving Switzerland, and need to convert their Francs into various international currencies. There are n traders t_1, t_2, \dots, t_n and m currencies c_1, c_2, \dots, c_m . Trader t_k has F_k Francs to convert. For each currency c_j , the bank can convert at most B_j Francs to c_j . Trader t_k is willing to trade as much as S_{kj} of his Francs for currency c_j . (For example, a trader with 1000 Francs might be willing to convert up to 200 of his Francs for USD, up to 500 of his Francs for Japanese's Yen, and up to 200 of his Francs for Euros). Assuming that all traders give their requests to the bank at the same time, describe an algorithm that the bank can use to satisfy the requests (if it can).

a) Describe how to construct a flow network to solve this problem, including the description of nodes, edges, edge directions and their capacities. (8 pts)

Bipartite graph: one partition traders t_1, t_2, \dots, t_n . Other, available currency, c_1, c_2, \dots, c_m .

Connect t_k to c_j with the capacity S_{kj}

Connect source to traders with the capacity F_k .

Connect available currency c_j to the sink with the capacity B_j .

Rubrics:

- Didn't include supersource (-1 point)
- Didn't include traders nodes (-1 point)
- Didn't include currencies nodes (-1 point)
- Didn't include supersink (-1 point)
- Didn't include edge direction (-1 point)
- Assigned no/wrong capacity on edges between source & traders (-1 point)
- Assigned no/wrong capacity on edges between traders & currencies (-1 point)
- Assigned no/wrong capacity on edges between currencies & sink (-1 point)

b) Describe on what condition the bank can satisfy all requests. (4 pts)

If there is a flow f in the network with $|f| = F_1 + \dots + F_n$, then all traders are able to convert their currencies.

Rubrics:

- Wrong condition (-4 points): Unless you explicitly included $|f| = F_1 + \dots + F_n$, no partial points were given for this subproblem.
- In addition to correct answer, added additional condition which is wrong (-2 points)
- No partial points were given for conditions that satisfy only some of the requests, not all requests.
- Note that $\sum S_{kj}$ and $\sum B_j$ can be larger than $\sum F_k$ in some cases where bank satisfy all requests.
- Note that $\sum S_{kj}$ can be larger than $\sum B_j$ in some cases where bank satisfy all requests.
- It is possible that $\sum S_{kj}$ or $\sum B_j$ is smaller than $\sum F_k$. In these cases, bank can never satisfy all requests because max flow will be smaller than $\sum F_k$.

- c) Assume that you execute the Ford-Fulkerson algorithm on the flow network graph in part a), what would be the runtime of your algorithm? (3 pts)

$O(n m |f|)$

Rubrics:

- Flow($|f|$) was not included (-1 point)
- Wrong description (-1 point)
- Computation is incorrect (-1 point)
- Notation error (-1 point)
- Missing big O notation (-1 point)
- Used big Theta notation instead of big O notation (-1 point)
- Wrong runtime complexity (-3 points)
- No runtime complexity is given (-3 points)

1) 20 pts

Mark the following statements as **TRUE** or **FALSE**. No need to provide any justification.

[**TRUE/FALSE**]

Dynamic programming only works on problems with non-overlapping subproblems.

[**TRUE/FALSE**]

If a flow in a network has a cycle, this flow is not a valid flow.

[**TRUE/FALSE**]

Every flow network with a non-zero max s-t flow value, has an edge e such that increasing the capacity of e increases the maximum s-t flow value.

[**TRUE/FALSE**]

A dynamic programming solution always explores the entire search space for all possible solutions.

[**TRUE/FALSE**]

Decreasing the capacity of an edge that belongs to a min cut in a flow network always results in decreasing the maximum flow.

[**TRUE/FALSE**]

Suppose f is a flow of value 100 from s to t in a flow network G . The capacity of the

minimum $s - t$ cut in G is equal to 100.

[**TRUE/FALSE**]

One can **efficiently** find the maximum number of edge disjoint paths from s to t in a directed graph by reducing the problem to max flow and solving it using the Ford-Fulkerson algorithm.

[**TRUE/FALSE**]

If all edges in a graph have capacity 1, then Ford-Fulkerson runs in linear time.

[**TRUE/FALSE**]

Given a flow network where all the edge capacities are even integers, the algorithm will require at most $C/2$ iterations, where C is the total capacity leaving the source s.

[**TRUE/FALSE**]

By combining divide and conquer with dynamic programming we were able to reduce the space requirements for our sequence alignment solution at the cost of increasing the computational complexity of our solution.

2) 20 pts.

Suppose that we have a set of students S_1, \dots, S_s , a set of projects P_1, \dots, P_p , a set of teachers T_1, \dots, T_t . A student is interested in a subset of projects. Each project p_i has an upper bound $K(P_i)$ on the number of the students that can work on it. A teacher T_i is willing to be the leader of a subset of the projects. Furthermore, he/she has an upper bound $H(T_i)$ on the number of students he/she is willing to supervise in total. We assume that no two teachers are willing to supervise the same project. The decision problem here is whether there is really a feasible assignment without violating any of the constraints in K and in H .

a) Solve the decision problem using network flow techniques. (15 pts)

The source node is connected to each student with the capacity of 1. Each student is connected to his/her desired projects with a capacity of 1.

Each project is connected to teachers who are willing to supervise it with a capacity of $K(P_i)$. Each teacher is connected to the sink node with the capacity of $H(T_i)$. If there is a max flow that is equal to the number of students, there is a solution to the problem. You can also convert the order of nodes and edges (source->teachers->projects->students->sink) as long as the edges are correct. You can have 2 sets of nodes for projects (inputs and outputs) as long as the output nodes do not constrain the flow.

b) Prove the correctness of your algorithm. (5 pts)

We need to show that

A – If there is a feasible assignment of students and teachers to projects, we can find a max flow of value s (# of students) in G

B – If we find a max flow of value s in G , we can find a feasible assignment of students and teachers to projects.

Rubric:

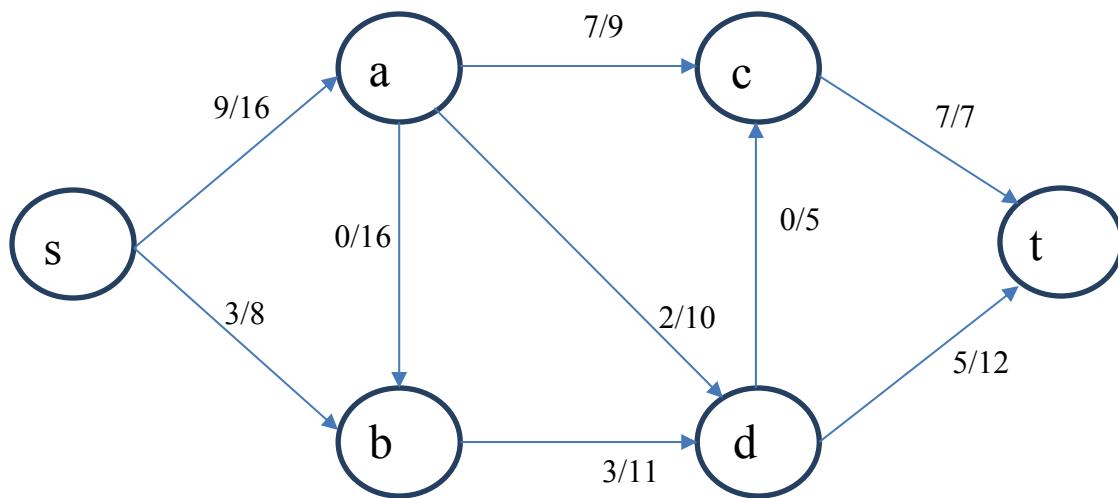
- 1) incorrect/missing nodes (incorrect nodes * -3)
- 2) incorrect/missing edges (incorrect edges * -2)
- 3) extra demand values (extra demands * -1)
- 4) not mentioning that max flow = number of students (-3)
- 5) if the proof is only provided for one side (-2)
- 6) if proof is not sufficient (-1)

3) 16 pts.

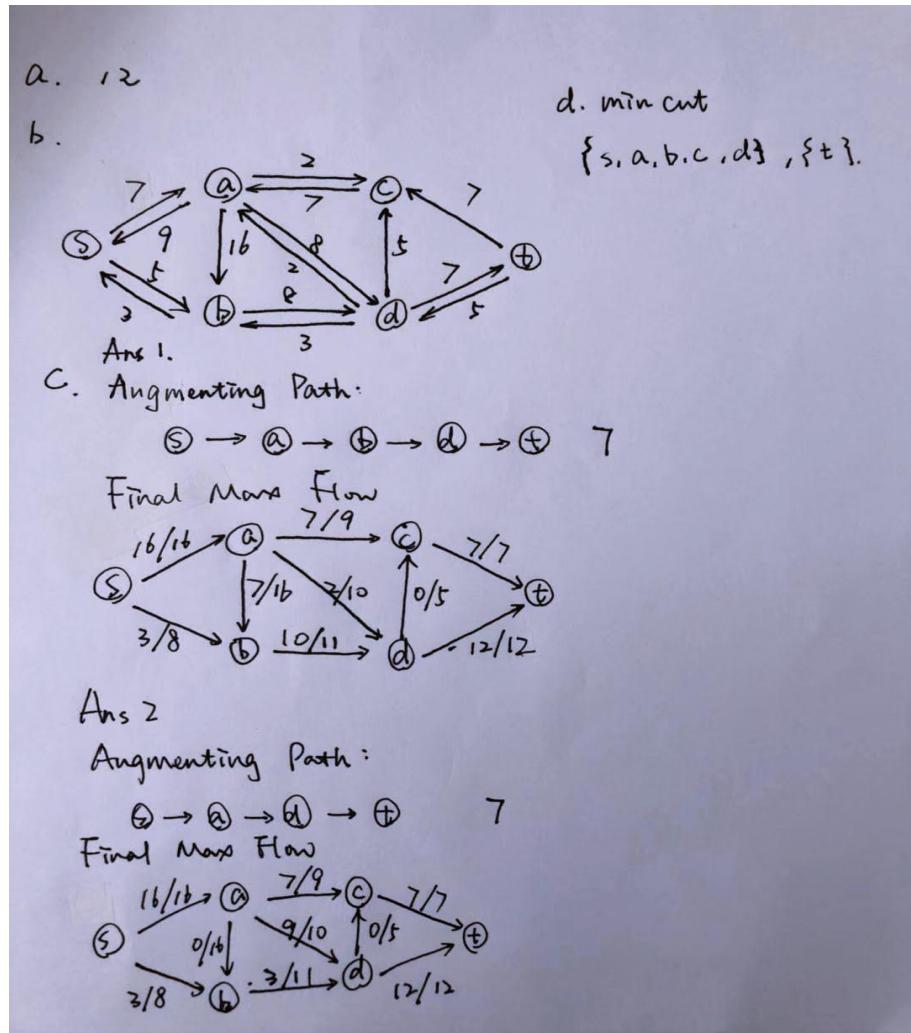
Consider the below flow-network, for which an s-t flow has been computed. The numbers x/y on each edge shows that the capacity of the edge is equal to y , and the flow sent on the edge is equal to x .

- What is the current value of flow? (2 pts)
- Draw the residual graph for the corresponding flow-network. (6 pts)
- Calculate the maximum flow in the graph using the Ford-Fulkerson algorithm. You need to show the augmenting path at each step and final max flow. (6 pts)
- Show the min cut found by the max flow you found in part c. (2 pts)

Note: extra space provided for this problem on next page



Solution:



Rubric:

a) Incorrect value -> 0 point

b) Every incorrect/missing edge or capacity -> -1 point

c) Every possible flow which has a capacity of 19 is correct.

Answers without correct value(other than) 19 will get 0 point.

Answers with correct value/final max flow/final residual graph without augmenting path/detail steps will get 3 points

Answers with correct value/final max flow/final residual graph with augmenting path&detail steps will get 6 points

Answers with a correct augmenting path but incomplete max flow will get 1 point.

d) Incorrect set -> 0 point

4) 16 pts

A symmetric sequence or palindrome is a sequence of characters that is equal to its own reversal; e.g. the phrase “a man a plan a canal panama” forms a palindrome (ignoring the spaces between words).

a) Describe how to use the **sequence alignment algorithm** (described in class) to find a longest symmetric subsequence (longest embedded palindrome) of a given sequence. (14 pts)

Example: CTGACCTAC ‘s longest embedded palindromes are CTCCTC and CACCAC

Solution: given the sequence S , reverse the string to form S^r . Then find the longest common substring between S and S^r by setting all mismatch costs to ∞ and $\delta=1$ (or anything greater than zero) .

b) What is the run time complexity of your solution? (2 pts)

$O(n^2)$

Rubric 4a)

- 10 points for reversing the string and using the sequence alignment algorithm
 - If the recurrence is given without explicitly mentioning the sequence alignment algorithm is fine.
 - Recurrence is discussed in class
- 2 points for allocating the correct mismatch value.
- 2 points for allocating the correct gap score.
- Any other answer which does not use sequence alignment algorithm is wrong. **The question explicitly asks to use the sequence alignment algorithm.** (0 to 2 points)
 - 2 points for correctly mentioning any other algorithm to get palindrome within a string
- Wrong answers
 - Dividing the string into two and reversing the second half.
 - Will not work, as the entire palindrome can be located within the first or the second half
 - Aligning the original string with the copy of the same string
 - This will return the original string and not the palindrome
 - Any other algorithm which finds the palindrome in a string

Rubric 4b)

2 points

0 points for any other answer

5) 20 pts

Let's say you have a dice with m sides numbered from 1 to m , and you are throwing it n times. If the side with number i is facing up you will receive i points. Consider the variable Z which is the summation of your points when you throw the dice n times. We want to find the number of ways we can end up with the summation Z after n throws. We want to do this using dynamic programming.

a) Define (in plain English) subproblems to be solved. (4 pts)

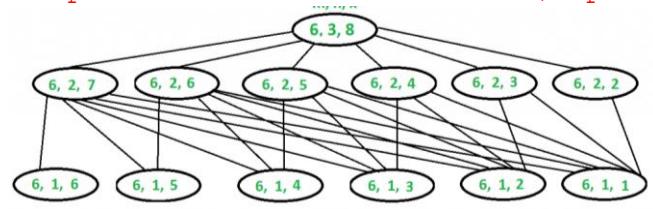
$\text{OPT}(n, Z)$ = number of ways to get the sum Z after n rolls of dice.

b) Write the recurrence relation for subproblems. (6 pts)

$\text{OPT}(n, Z) = \sum_{i=1 \text{ to } m} \text{OPT}(n-1, Z-i)$

Example: Given, $m=6$, Calculate: $\text{OPT}(3, 8)$

Subproblems are as follows (represented as (m, n, Z)):



Rubric:

-6 if not summing over m

-4 if summing over m but OPT is somewhat incorrect.

c) Using the recurrence formula in part b, write pseudocode to compute the number of ways to obtain the value SUM . (6 pts)

Make sure you have initial values properly assigned. (2 pts)

```
OPT(1, i) = 1 for i=1 to MIN(m, SUM)
OPT(1, i) = 0, for i=MIN(m, SUM) to MAX(m, SUM)
For i = 2 to n
    For j = 1 to SUM
        OPT(i, j) = 0
        For k = 1 to MIN(m, i)
            OPT(i, j) = OPT(i, j) + OPT(i-1, j-k)
        Endfor
    Endfor
Endfor
```

Return OPT(n, SUM)

Example, Given, m=6, Calculate: OPT(3, 8)

OPT Table looks as follows:

0	0	0	0	0	0	0	0	0
0	1	1	1	1	1	1	0	0
0	0	1	2	3	4	5	6	5
0	0	0	1	3	6	10	15	21

Alternatively, you can initialize $\text{OPT}(0,0) = 1$ and start the iteration of i from 1.

Rubric:

- 1 for each initialization missed.
- 6 for any incorrect answer. (e.g incorrect recurrence relation, incorrect loops, etc)
i.e. no partial grading for any pseudocode producing incorrect answer.

- d) Compute the runtime of the algorithm described in part c and state whether your solution runs in polynomial time or not (2 pts)

Time Complexity: $O(m * n * Z)$ where m is number of sides, n is the number of times we roll the dice, and Z is given sum. This is pseudo polynomial since the complexity depends on the numerical value of input terms.

Rubric:

- Not graded if part c is incorrect.
- 2 if stated as polynomial time.
- 2 if incorrect complexity equation.

6) 8 pts

- a) Given a flow network $G=(V, E)$ with integer edge capacities, a max flow f in G , and a specific edge e in E , design a linear time algorithm that determines whether or not e belongs to a min cut. (6 pts)

Solution:

Reduce capacity of e by 1 unit. - $O(1)$

Find an s - t path containing e and subtract 1 unit of s - t flow on that path. For this, considering e 's adjacent nodes to be (u, v) , you can run BFS from source to find an s - u path and BFS from v to find an v - t path which gives you an s - u - v - t path crossing e . - $O(|V|+|E|)$

Then construct the new residual graph G_f - $O(|E|)$

If there is an augmenting s - t path in G_f then e does not belong to a min cut, otherwise it does.

- b) Describe why your algorithm runs in linear time.

You can find this using BFS from source(only one iteration of the FF algorithm) - $O(|V| + |E|)$

All the above steps are linear in the size of input, therefore the entire algorithm is linear.

Rubric:

If providing the complete algorithm: full points

- If not decreasing the s - t flow by 1: -1 points
- If running the entire Ford-Fulkerson algorithm to find the flow: -2 points
- If providing the overall idea without description of implementation steps: -3 or -4 points depending on your description

If the provided algorithm is not scalable to graphs having multiple(more than two) min-cuts(for example using BFS to find reachable/unreachable nodes from source/sink): -4 points

- Incomplete or incorrect description of the above algorithm (-1 or -2 additional points depending on your description)

If removing the edge and finding max-flow: -3 or -4 points depending on your description (since this algorithm doesn't run in linear time)

If only considering saturated edges to check for min-cut edges: -7 points

Adding the capacity of e and checking the flow: no points (this approach is not checking for min-cut)

Checking all possible cuts in the graph: no points (this takes exponential time)

Other algorithms: no points

CSCI 570 - Spring 2019 - HW 9 Solution

1. At a dinner party, there are n families a_1, a_2, \dots, a_n and m tables b_1, b_2, \dots, b_m . The i^{th} family a_i has g_i members and the j^{th} table b_j has h_j seats. Everyone is interested in making new friends and the dinner party planner wants to seat people such that no two members of the same family are seated in the same table. Design an algorithm that decides if there exists a seating assignment such that everyone is seated and no two members of the same family are seated at the same table.

Construct the following network $G = (V;E)$. For every family introduce a vertex and for every table introduce a vertex. Let a_i denote the vertex corresponding to the i^{th} family and let b_j denote the vertex corresponding to the j^{th} table. From every family vertex a_i to every table vertex b_j , add an edge $(a_i; b_j)$ of capacity 1. Add two more vertices s and t . For every family vertex a_i add an edge $(s; a_i)$ of capacity g_i . For every table vertex b_j add an edge $(b_j; t)$ of capacity h_j . Claim: There exists a valid seating if and only if the value of max flow from s to t in the above network equals $\sum_{n \geq i \geq 1} g_i$. Proof of Claim: Assume there exists a valid seating, that is a seating where every one is seated and no two members in a family are seated at a table. We construct a flow f to the network as follows. If a member of the i^{th} family is seated at the j^{th} table in the seating assignment, then assign a flow of 1 to the edge $(a_i; b_j)$. Else assign a flow of 0 to the edge $(a_i; b_j)$. The edge $(s; a_i)$ is assigned a flow equaling the number of members in the i^{th} family that are seated (which since the seating is valid equals g_i). Likewise the edge $(b_j; t)$ is assigned a flow equaling the number of seats taken in the table b_j (which since the seating is valid is at most h_j). Clearly the assignment is valid since by construction, the capacity and conservation constraints are satisfied. Further, the value of the flow equals $g_1 + g_2 + \dots + g_n$. Conversely, assume that the value of the max $s-t$ flow equals $g_1 + g_2 + \dots + g_n$. Since the capacities are integers, by the correctness of the Ford-Fulkerson algorithm, there exists a max flow (call f) such that the flow assigned to every edge is an integer. In particular, every edge between the family vertices and table vertices has a flow of either 0 or 1 (since these edges are of capacity 1). Construct a seating assignment as follows: seat a person of the i^{th} family at the j^{th} table if and only if $f(a_i, b_j)$ is 1. By construction at most one member of a family is seated at a table. Since the value of f equals the

capacity of the cut (s, Vs) , every edge out of s is saturated. Thus by flow conservation at a_i , for every a_i the number of edges out of a_i with a flow of 1 is g_i . Thus in the seating assignment, every one is seated. Further, since the flow $f(b_j, t)$ out of b_j is at most h_j , so at most h_j people are seated at table b_j . Thus we have a valid seating.

2. There is a precious diamond that is on display in a museum at m disjoint time intervals. There are n security guards who can be deployed to protect the precious diamond. Each guard has a list of intervals for which he/she is available to be deployed. Each guard can be deployed to at most A time slots and has to be deployed to at least B time slots. Design an algorithm that decides if there is a deployment of guards to intervals such that each interval has either exactly one or exactly two guards deployed.

We create a circulation network as follows. For the i^{th} guard, introduce a vertex g_i and for the j^{th} time interval, introduce a vertex t_j . If the i^{th} guard is available for the j^{th} interval, then introduce an edge from g_i to t_j of capacity 1. Add a source s and a sink t . To every guard vertex add an edge from s of capacity A and lower bound B . From every interval vertex add an edge to t of capacity 2 and lower bound 1. Add an edge from t to s of infinite capacity. We claim that there exists a valid deployment if and only if the above network has a valid circulation. The proof of the claim is virtually identical to the proof in section 7.8 of the text for the survey design problem. The algorithm proceeds by determining if the network has a circulation (by reducing it to a flow problem and then applying Ford-Fulkerson) and answers yes if and only if there is a circulation. The number of vertices and number of edges in the resulting flow problem are bounded by $\mathcal{O}(n)$ and $\mathcal{O}(n^2)$ respectively. The running time of our algorithm is dominated by the flow computation which takes $\mathcal{O}(n^3)$.

3. Solve Kleinberg and Tardos, Chapter 7, Exercise 28.

We create a circulation network as follows. For the i^{th} TA, introduce a vertex p_i and for the j^{th} time interval, introduce a vertex t_j . If the i^{th} TA is available for the j^{th} interval, then introduce an edge from p_i to t_j of capacity 1. Add a source s and a sink t . To every TA vertex add an edge from s of capacity b and lower bound a . From every interval vertex add an edge to t of capacity 1. Add an edge from t to s of capacity c and lower bound c . Claim: there exists a valid assignment if and only if the above network has a valid circulation. Proof of Claim: We first show that if there is a valid assignment, then the above network has a valid circulation. Assume there is a valid assignment. If in the assignment, the i^{th} TA is assigned to the j^{th} interval, then assign a flow of 1 to the edge (p_i, t_j) . Extend this flow to the rest of network using flow conservation laws to get a valid flow. That is, the flow on the edge (t_j, t) is 1 if a TA is assigned to t_j and 0 otherwise, the flow on edge (t, s) is the number of

intervals with TAs assigned to them and the flow on (s, p_i) is the number of intervals the i^{th} TA is assigned to. The validity of the assignment implies that flow conservation holds at every vertex and capacity/lower bound constraints hold at every edge (check this!). We next prove the converse, that is if there is a valid circulation, then there is a valid assignment. Assume that the network has a valid circulation. Since the capacities and lower bounds are integers, when we convert the circulation problem into a flow problem, we get a network with integer capacities. The correctness of Ford-Fulkerson algorithm implies that the resulting flow network has a max flow where the flows are integers. Thus we can conclude that our original network has a valid circulation where the flow on each edge is an integer. Let $f(e)$ denote the flow on edge e in this integer valued circulation. We next construct a TA assignment. Since the capacity of each edge from TA vertices to interval vertices is 1, the flow on these vertices is either 1 and 0. Assign the i^{th} TA to the j^{th} interval if and only if $f(p_i, t_j) = 1$. The assignment is valid for the following reason. The i^{th} TA is assigned to $f(s, p_i)$ intervals (which is between a and b), every interval is assigned to at most one TA (since the flow out of an interval vertex is at most 1 due to the capacity constraint) and the number of intervals with TAs assigned is $f(t, s)$ which is exactly c due to the lower bound/capacity constraint. The algorithm proceeds by determining if the network has a circulation (by reducing it to a flow problem and then applying Ford-Fulkerson) and if so finding one with integer flows (denote by f). If there is a valid circulation, then Assign the i^{th} TA to the i^{th} interval if and only if $f(p_i, t_j) = 1$. Else, return no assignment. As in problem 2, the running time is $\mathcal{O}(n^3)$.

4. The computer science department course structure is represented as a directed acyclic graph $G = (V, E)$ where the vertices correspond to courses and a directed edge $(u, v) \in E$ exists if and only if the course u is a prerequisite of the course v . By taking a course $w \in V$, you gain a benefit of b_w which could be a positive or negative number. Design an algorithm that picks a subset $A \subset V$ of courses to take such that the total benefit $\sum_{w \in A} b_w$ is maximized. Remember that if $v \in A$ and $(u, v) \in E$, then u has to be in A . That is, to take a course, you have to take all its prerequisites. The running time should be polynomial in $|V|$.

The solution to this problem is similar to that of the Project Selection Problem in the text book (section 7.11).

CSCI 570 - Spring 2019 - HW 7 solution

1 Graded Problems

- Given a sequence $\{a_1, a_2, \dots, a_n\}$ of n numbers, describe an $O(n^2)$ algorithm to find the longest monotonically increasing sub-sequence.

Solution: Let l_i denote the length of the longest monotonically increasing sub-sequence that ends with a_i ($l_1 = 1$). Compute the sequences S_i, S_{ij} using the following recurrences.

- Initialize $S_1 = a_1$.
- For $1 \leq j < i$, if $a_j > a_i$ then $S_{ij} = a_i$. Otherwise, S_{ij} is set to $\{S_j$ concatenated with $a_i\}$.
- S_i is set to the longest sequence among all the sequences S_{ij} , $1 \leq j < i$.

Claim: The length of S_i , $l(S_i) = l_i$.

Assume otherwise. Let k be the smallest index such that $l(S_k) < l_k$. Let O_k be a sequence of length l_i ending with a_k . Let a_j be the second last element of the sequence O_k . As $j < k$, $l_j = l(S_j)$

$$l(S_k) < l_k = l_j + 1 \Rightarrow l(S_j) < l_j$$

This is a contradiction as $j < k$ and k is the smallest index such that $l(S_k) < l_k$. Thus our claim is true. Clearly the longest monotonically increasing sub-sequence is by definition the longest of the sequences S_i , $1 \leq i \leq n$.

2. You are given n points $\{(x_1, y_1), (x_2, y_2), \dots, (x_n, y_n)\}$ on the real plane. They have been sorted from left to right and no two points have the same x-coordinate. That means $x_1 < x_2 < \dots < x_n$.

A bitonic tour is defined as follows. The tour starts from (x_1, y_1) , goes through some intermediate points and reaches (x_n, y_n) . Then it goes back to (x_1, y_1) through every one of the rest of the points. All points except (x_1, y_1) are thus visited exactly once. Further, from (x_1, y_1) to (x_n, y_n) , you have to keep going right at every step. Similarly, you have to keep going left from (x_n, y_n) to (x_1, y_1) . Describe an $O(n^2)$ algorithm to compute the shortest bitonic tour.

Solution: Let $p_j = (x_j, y_j)$ denote the j^{th} point. A shortest bitonic tour can be thought of as a cycle where the vertices are points. Edges connect the points if they are visited one after another. Consider the shortest bitonic tour on the first i points. This tour must contain the edge (p_{i-1}, p_i) . (It doesn't matter whether the edge is used in the forward or the backward path). Observe that such a tour must also contain an edge (p_k, p_i) with $k < i - 1$. For $k < i - 1$, a shortest bitonic tour on the points p_1, \dots, p_i that contains (p_k, p_i) must be a shortest bitonic tour on the points p_1, \dots, p_{k+1} minus the edge (p_k, p_{k+1}) plus the edge (p_k, p_i) and plus the path $\{(p_{k+1}, p_{k+2}), \dots, (p_{i-1}, p_i)\}$ (because there is no other way to visit these nodes). Consequently k can be chosen such that we end up with the shortest bitonic tour on p_1, \dots, p_i .

The fact that the subproblem on p_1, \dots, p_{k+1} exhibits the optimal substructure property can be proved by a simple replacement strategy. Suppose we have an optimal solution on p_1, \dots, p_i points which uses a solution on p_1, \dots, p_{k+1} and is not optimal. Now by replacing the non-optimal solution on the p_1, \dots, p_{k+1} points by an optimal solution into the " p_1, \dots, p_i " problem, we obtain a solution which is better than the optimal solution on p_1, \dots, p_i , resulting in a contradiction.

Let $OPT(i)$ be the length of the shortest bitonic tour on the first i points, we can write the following recursion.

$$OPT(i) = \min_{1 \leq k \leq i-2} \{OPT(k+1) + D(k+1, i) + d(k, i) - d(k, k+1)\}$$

for all $3 \leq i \leq n$, where

$$d(i, j) = \sqrt{(x_i - x_j)^2 + (y_i - y_j)^2}$$

$$D(i, j) = \sum_{k=i}^{j-1} d(k, k+1)$$

The base cases are: $OPT(1) = 0, OPT(2) = 2 \times d(1, 2)$. $OPT(n)$ is the length of the shortest bitonic tour.

Each step of the iterations costs $O(n)$ and we need to compute n values in total, so the total running time is $O(n^2)$.

3. (Chapter 6, Exercise 10)

Solution:

- (a) Consider the following example: there are totally 4 minutes, the numbers of steps that can be done on the two machines in the 4 minutes are listed as follows (in time order):

- Machine A: 2, 1, 1, 200
- Machine B: 1, 1, 20, 100

The given algorithm will choose A then move, then stay on B for the final two steps. However, the optimal solution will stay on A for the four steps.

- (b) An observation is that, in the optimal solution for the time interval from minute 1 to minute i , you should not move in minute i ; because otherwise, you can keep staying on the machine where you are and get a better solution ($a_i > 0$ and $b_i > 0$). For the time interval from minute 1 to minute i , consider that if you are on machine A in minute i , you either (i) stay on machine A in minute $i - 1$ or (ii) are in the process of moving from machine B to A in minute $i - 1$. Now let $OPT_A(i)$ represent the maximum value of a plan in minute 1 through i that ends on machine A, and define $OPT_B(i)$ analogously for B. If case (i) is the best action to make for minute $i - 1$, we have $OPT_A(i) = a_i + OPT_A(i - 1)$; otherwise, we have $OPT_A(i) = a_i + OPT_B(i - 2)$. Thus, we have

$$OPT_A(i) = a_i + \max\{OPT_A(i - 1), OPT_B(i - 2)\}$$

$$OPT_B(i) = b_i + \max\{OPT_B(i - 1), OPT_A(i - 2)\}$$

Then the algorithm is illustrated in Algorithm 1.

Algorithm 1

- ```

1: $OPT_A(0) = 0$
2: $OPT_B(0) = 0$
3: $OPT_A(1) = a_1$
4: $OPT_B(1) = b_1$
5: for i from 2 to n do
6: $OPT_A(i) = a_i + \max\{OPT_A(i - 1), OPT_B(i - 2)\}$
7: Record the action (either stay or move) in minute $i - 1$ that achieves the maximum
8: $OPT_B(i) = b_i + \max\{OPT_B(i - 1), OPT_A(i - 2)\}$
9: Record the action (either stay or move) in minute $i - 1$ that achieves the maximum
10: end for
11: return $\max\{OPT_A(n), OPT_B(n)\}$
12: Track back through the arrays OPT_A and OPT_B by checking the action records
 from minute $n - 1$ to minute 1 to recover the optimal solution

```
- 

It takes  $O(1)$  time to complete the operations in each iteration; there are  $O(n)$  iterations; the tracing backs takes  $O(n)$  time. Thus, the overall complexity is  $O(n)$ .

4. (Chapter 6, Exercise 20)

Solution: Let the  $(i, h)$ -subproblem be the problem in which one wants to maximize one's grade on the first  $i$  courses, using at most  $h$  hours. Let  $OPT(i, h)$  be the maximum total grade that can be achieved for this subproblem. Then  $OPT(0, h) = 0$  for all  $h$ , and  $OPT(i, 0) = \sum_{j=1}^i f_j(0)$ . Now, in the optimal solution to the  $(i, h)$  subproblem, one spends  $k$  hours on course  $i$  for some value of  $k \in \{0, 1, \dots, h\}$ ; thus, we have

$$OPT(i, h) = \max_{0 \leq k \leq h} \{f_i(k) + OPT(i - 1, h - k)\}$$

Then the algorithm is illustrated in Algorithm 2.

---

**Algorithm 2**

---

```

1: for h from 1 to H do
2: $OPT(0, h) = 0$
3: end for
4: $OPT(1, 0) = f_1(0)$
5: for i from 2 to n do
6: $OPT(i, 0) = f_i(0) + OPT(i - 1, 0)$
7: end for
8: for i from 1 to n do
9: for h from 1 to H do
10: $OPT(i, h) = \max_{0 \leq k \leq h} \{f_i(k) + OPT(i - 1, h - k)\}$
11: Record the k that results in the maximum, denoted as $k^*(i, h)$
12: end for
13: end for
14: return $OPT(n, H)$
15: Having obtained the $(n + 1) \times (H + 1)$ table with each entry filled with $OPT(i, h)$;
 in order to produce the optimal distribution of time, track back from the $(n + 1, H + 1)$ th entry using $\{k^*(i, h)\}$ until a boundary entry of the table is reached

```

---

The total time to fill in each entry  $OPT(i, h)$  is  $O(H)$ , and there are  $nH$  entries, resulting in a total time of  $O(nH^2)$ . Tracking back according to the records takes  $O(n)$  time. Thus, the overall time is  $O(nH^2)$ .

## 2 Practice Problems

1. Lets consider the vertices in topological order. Suppose  $LP(v)$  represents the value of the most expensive path from  $s$  to  $v$ . Base case:

$$LP(s) = 0$$

$$LP(v) = -\infty$$

Our recurrence becomes:

$$LP(v) = \max_{u:v \in adj[u]} \{LP(u) + c_{(u,v)}\}$$

We can fill in this table in increasing order of  $v$ , with the above defined base cases to ensure that any that aren't reachable from  $s$  aren't considered to be on the maximum path. We want the largest  $LP(v)$  value; if we track the value of the maximal  $u$ , we can use that to backtrack and find the longest path. Our total runtime is  $O(m + n)$ .

# CSCI 570 - Spring 2019 - HW 8

## 1 Graded Problems

1. *The edge connectivity of an undirected graph is the minimum number of edges whose removal disconnects the graph. Describe an algorithm to compute the edge connectivity of an undirected graph with  $n$  vertices and  $m$  edges in  $\mathcal{O}(m^2n)$  time.*

For a cut  $(S, \bar{S})$ , let  $c(S, \bar{S})$  denote the number of edges crossing the cut. By definition, the edge connectivity

$$k = \min_{S \subset V} c(S, \bar{S})$$

Fix a vertex  $u \in V$ . For every cut  $(S, \bar{S})$ , there is a vertex  $v \in V$  such that  $u$  and  $v$  are on either side of the cut. Let  $C_{u,v}$  denote the value of the min  $u$ - $v$  cut. Thus

$$k = \min_{v \in V, v \neq u} C_{u,v}$$

For each  $v \neq u$ ,  $C_{u,v}$  can be determined by computing the max flow from  $u$  to  $v$ . Since  $G$  is undirected, we need to implement an undirected variant of the max flow algorithm. Set all edge capacities to 1. During each step of the flow computation, search for an undirected augmenting path using breadth first search and send a flow of 1 through this path.

There are  $n - 1$  flow computations and each flow computation takes at most  $\mathcal{O}(m^2)$  time.

2. *Solve Kleinberg and Tardos, Chapter 7, Exercise 7.*

Let  $s$  be a source vertex and  $t$  a sink vertex. Introduce a vertex for every base station and introduce a vertex for every client.

For every base station vertex  $b$ , add an edge  $(s, b)$  of capacity  $L$ . For every client vertex  $v$ , add an edge  $(v, t)$  of unit capacity. For every base station vertex  $b$ , add a unit capacity edge from  $b$  to every client  $c$  within its range.

We claim that every client can be connected to a base station subject to load and range conditions if and only if the max flow of the above network is at least  $n$ .

If every client can be connected to a base station subject to load and range conditions, then if a client  $c$  is served by base station  $b$ , assign a unit flow to the edge  $(b, c)$ . Assign the flows to the edges leaving the source (respectively the edges leaving the sink) so that the conservation constraints are satisfied at every base station (respectively client) vertex is satisfied. (note that such an assignment is unique). Further, since every client can be connected to a base station subject to load conditions, the capacity constraints at the edges leaving the source are also satisfied. Hence we are left with a valid flow assignment for the network. Since every client is connected, the flow entering the sink is at least the number of clients  $n$  (in fact, it is exactly  $n$ ).

Conversely, if the max flow of the network is at least  $n$  then there exists a integer flow of flow value at least  $n$  (since all capacities are integral). We will work with this integral flow. For every edge  $(b, c)$  with a flow of 1, assign  $c$  to the base station  $b$ . Since a client vertex can at most contribute one unit of flow entering  $t$ , and the total flow entering  $t$  is at least  $n$ , every client vertex has flow entering it. This implies that every client is serviced by exactly one station. Since the flow entering a base station vertex  $b$  is at most  $L$ , a base station is assigned to at most  $L$  clients. Thus every client is connected to a base station subject to load and range conditions.

Hence our claim is true and all that is left to do is to compute the max-flow of the network using Ford-Fulkerson and to output YES if and only if the max-flow value is at least  $n$ .

3. *Solve Kleinberg and Tardos, Chapter 7, Exercise 9.*

This problem is virtually identical to the previous problem. Let  $s$  be a source vertex and  $t$  a sink vertex. Introduce a vertex for every hospital and introduce a vertex for every injured-person.

For every hospital vertex  $h$ , add an edge  $(s, h)$  of capacity  $\lceil n/k \rceil$ . For every injured-person vertex  $c$ , add an edge  $(c, t)$  of unit capacity. For every hospital vertex  $h$ , add a unit capacity edge from  $h$  to every injured-person vertex  $c$  within half-hour's driving.

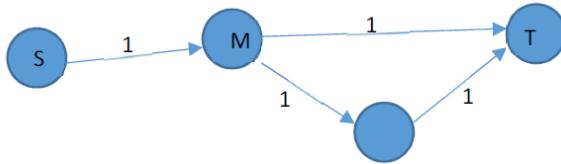
We claim that every injured person can be admitted to a hospital (within 1/2 hour driving) such that the load on the hospitals are balanced if and only if the max flow of the above network is at least  $n$ . The proof is identical to the previous problem.

## 2 Practice Problems

1. An edge of a flow network  $G$  is called critical if decreasing the capacity of this edge results in a decrease in the maximum flow. Is it true that with respect to a maximum flow of  $G$ , any edge whose flow is equal to its capacity is a critical edge? Give an efficient algorithm that finds a critical edge in a flow network.

Not true.

As shown below. The maximum flow is 1, and an alternative path from  $M$  to  $T$  can be found if one path gets blocked.



Get the maximum flow  $f^*$ , we can find a cut  $(A, B)$ , such that  $\text{value}(f^*) = \text{cap}(A, B)$ .

Assume that  $A$  contains source and  $B$  contains target. An edge  $e_c$  from  $A$  to  $B$  can be found, such that  $f(e_c) = c(e_c)$  in  $f^*$ . Then  $e_c$  is a critical edge.

### Proof:

Since  $\text{cap}(A, B) = \sum_{e_i \text{ out of } A} c(e_i)$ , and  $e_c \in \{e_i\}$ , reducing  $c(e_c)$  will then reduce the capacity of cut  $A, B$  from  $\text{cap}(A, B)$  to  $\text{cap}'(A, B)$ .

Originally,  $\text{value}(f^*) = \text{cap}(A, B)$ . After the reduction, the modified maximum flow  $f^{**}$  will satisfy this relation:  $\text{value}(f^{**}) \leq \text{cap}'(A, B) < \text{cap}(A, B) = \text{value}(f^*)$ .

Thus  $e_c$  is a critical edge.

### Implementation:

After finding the maximum flow  $f^*$ , do DFS from source node on  $G_{f^*}$ , and label the reachable nodes as  $A$ . Label the rest of the nodes as  $B$ .

Scan all the edges in  $G$ . If an edge goes from  $A$  to  $B$ , then that edge is an critical edge.

Complexity: DFS is  $O(n + m)$ , and scanning is  $O(m)$ . So the final complexity is the same as that of finding the maximum flow  $f^*$ .

## Practice Final Solutions

- Do not open this exam booklet until you are directed to do so. Read all the instructions first.
- When the exam begins, write your name on every page of this exam booklet.
- The exam contains seven multi-part problems. You have 180 minutes to earn 180 points.
- This exam booklet contains 17 pages, including this one. An extra sheet of scratch paper is attached. Please detach it before turning in your exam.
- This exam is closed book. You may use three handwritten A4 or  $8\frac{1}{2}'' \times 11''$  crib sheets. No calculators or programmable devices are permitted.
- Write your solutions in the space provided. If you need more space, write on the back of the sheet containing the problem. Do not put part of the answer to one problem on the back of the sheet for another problem, since the pages may be separated for grading.
- Do not waste time and paper rederiving facts that we have studied. It is sufficient to cite known results.
- Do not spend too much time on any one problem. Read them all through first, and attack them in the order that allows you to make the most progress.
- Show your work, as partial credit will be given. You will be graded not only on the correctness of your answer, but also on the clarity with which you express it. Be neat.
- Good luck!

| Problem | Points | Grade | Initials |
|---------|--------|-------|----------|
| 1       | 12     |       |          |
| 2       | 56     |       |          |
| 3       | 20     |       |          |
| 4       | 25     |       |          |
| 5       | 27     |       |          |
| 6       | 20     |       |          |
| 7       | 20     |       |          |
| Total   | 180    |       |          |

Name: **Solutions** \_\_\_\_\_  
Circle your recitation letter and the name of your recitation instructor:

David      A      B      Steve      C      D      Hanson      E      F

**Problem 1. Algorithm Design Techniques [12 points]**

The following are a few of the design strategies we followed in class to solve several problems.

- 1.Dynamic programming.
- 2.Greedy strategy.
- 3.Divide-and-conquer.

For each of the following problems, mention which of the above design strategies was used (in class) in the following algorithms.

- 1.Longest common subsequence algorithm

**Solution:** Dynamic Programming

- 2.Minimum spanning tree algorithm (Prim's algorithm)

**Solution:** Greedy

3.Select

**Solution:** Divide-and-Conquer

4.Fast Fourier Transform

**Solution:** Divide-and-Conquer

**Problem 2. True or False, and Justify [56 points]**

Circle **T** or **F** for each of the following statements, and briefly explain why. The better your argument, the higher your grade, but be brief. Your justification is worth more points than your true-or-false designation. Each part is 4 points.

- (a) **T F** An inorder traversal of a Min Heap will output the values in sorted order.

**Solution:** False. Consider the Min Heap with 1 at the root and 3 as left child and 2 as right child.

- (b) **T F** A Monte-Carlo algorithm is a randomized algorithm that always outputs the correct answer and runs in expected polynomial time.

**Solution:** False. This is definition of a Las Vegas Algorithm.

- (c) **T F** Two distinct degree- $d$  polynomials with integer coefficients can evaluate to the same value in as many as  $d + 1$  distinct points.

**Solution:** False. They have same value on at most  $d$  distinct points.

- (d) **T F** The right subtree of an  $n$ -node 2-3-4 tree contains  $\Omega(n)$  nodes.

**Solution:** False. A tree with all degree-3 nodes on one subtree and degree-2 nodes on the other will have depth  $h = \log_3 n$ . There will be  $2^{\log_3 n} = n^{\log_3 2} = o(n)$  nodes on the sparse subtree. *Note: The prior version of the solutions incorrectly stated that this problem was true.*

- (e) **T F** If a problem in NP can be solved in polynomial time, then all problems in NP can be solved in polynomial time.

**Solution:** False. The decision version of MST is in NP, but this doesn't mean that all problems in NP can be solved in polynomial time.

- (f) **T F** If an NP-complete problem can be solved in linear time, then all NP-complete problems can be solved in linear time.

**Solution:** False. The reductions are *polynomial-time* but not necessarily *linear* time.

- (g) **T F** If single digit multiplication can be done in  $O(1)$  time, then multiplying two  $k$ -digit numbers can be done in  $O(k \log k)$  time.

**Solution:** True. Use FFT.

- (h) **T F** In a  $k$ -bit binary counter (that is initialized to zero and is always non-negative), any sequence of  $n < 2^k$  increments followed by  $m \leq n$  decrements takes  $O(m + n)$  total bit flips in the worst case, where a bit flip changes one bit in the binary counter from 0 to 1 or from 1 to 0.

**Solution:** True. As shown in lecture the amortized cost of an increment in a sequence of increments is  $O(1)$  and by the same argument, the amortized cost of a decrement in a sequence of decrements is  $O(1)$ .

- (i) **T F** Consider a directed graph  $G$  in which every edge has a positive edge weight. Suppose you create a new graph  $G'$  by replacing the weight of each edge by the negation of its weight in  $G$ . For a given source vertex  $s$ , you compute all shortest path from  $s$  in  $G'$  using Dijkstra's algorithm.

True or false: the resulting paths in  $G'$  are the longest (i.e., highest cost) simple paths from  $s$  in  $G$ .

**Solution:** False. Dijkstra's algorithm may not necessarily return the minimum weight path because this new graph may contain a negative weight cycle.

- (j) **T F** A spanning tree of a given undirected, connected graph  $G = (V, E)$  can be found in  $O(E)$  time.

**Solution:** True. Perform a walk on the graph starting from an arbitrary node.

- (k) T F Consider the following algorithm for computing the square root of a number  $x$ :

```
SQUARE-ROOT(x)
For $i = 1, \dots, x/2$:
 if $i^2 = x$ then output i .
```

True or False: This algorithm runs in polynomial time.

**Solution:** False. To run in polynomial-time, this algorithm would have to run in time polynomial in  $\lg x$ .

- (l) T F An efficient max-flow algorithm can be used to efficiently compute a maximum matching of a given bipartite graph.

**Solution:** True. Add a “supersource” and a “supersink”, each connected to a partition of the graph by unit capacity edges.

- (m) **T F** The following recurrence has solution  $T(n) = \Theta(n \lg(n^2))$ .

$$T(n) = 2T(n/2) + 3 \cdot n$$

**Solution:** True.  $\Theta(n \lg n^2) = \Theta(n \lg n)$ .

- (n) **T F** Computing the convolution of two vectors, each with  $n$  entries, requires  $\Omega(n^2)$  time.

**Solution:** False. Use the standard FFT algorithm.

**Problem 3. Placing Gas Stations Along a Highway [20 points]**

Give a dynamic programming algorithm that on input  $S$ , where  $S = \{s_0 = 0 \leq s_1 \leq \dots \leq s_n = m\}$  is a finite set of positive integers, determines whether it is possible to place gas stations along an  $m$ -mile highway such that:

1. A gas station can only be placed at a distance  $s_i \in S$  from the start of the highway.
2. There must be a gas station at the beginning of the highway ( $s_0 = 0$ ) and at the end of the highway ( $s_n = m$ ).
3. The distance between every two consecutive gas stations on the highway is between 15 and 25 miles.

For example, suppose the input is  $\{0, 15, 40, 50, 60\}$ . Then your algorithm should output “yes”, because we can place gas stations at distances  $\{0, 15, 40, 60\}$  from the beginning of the highway.

However, if the input is  $\{0, 25, 30, 55, 70\}$ , then your algorithm should output “no”, because there is no subset of the distances that satisfies the conditions listed above.

Remember to analyze the running time of your algorithm.

**Solution:** Check to see if 0 and  $m$  are in  $S$ . If not, output “no”.

Let  $G[0] = \text{“yes”}$  if  $s_0 = 0$ .

For  $i$  from 1 to  $n$ , let  $j = s_i$ :

Let  $G[j]_{j \in S, j > 0} = \text{“yes”}$  if  $G[k] = \text{“yes”}$ , for some  $k \in S$ ,  $k < j$ , and  $15 \leq |s_k - s_j| \leq 25$ .

Return  $G[m]$ .

**Problem 4. Independent Set and Vertex Cover [25 points]**

For a graph  $G = (V, E)$ , we say  $S \subseteq V$  is an independent set in  $G$  if there are no edges between any two vertices in  $S$ .

We say a subset  $T \subseteq V$  is a vertex cover of  $G$  if for every edge  $(u, v)$  in  $E$  at least one of  $u$  or  $v$  is in  $T$ .

- (a) **[10 points]** Show  $S$  is an independent set in  $G$  if and only if  $V - S$  is a vertex cover of  $G$ .

**Solution:**  $\Rightarrow$  Assume  $S$  is an Independent Set, but that  $V - S$  is not a Vertex Cover. Then there exists an edge whose endpoints are both not in  $V - S$ , namely, there is an edge whose endpoints are in  $S$ . That is a contradiction, so  $V - S$  must be a Vertex Cover.

$\Leftarrow$  Now assume that  $V - S$  is a Vertex Cover, but that  $S$  is not an Independent Set. Then there exists an edge with both endpoints in  $S$ . But then that edge would not be touched by  $V - S$ , so  $V - S$  could not be a Vertex Cover. This contradicts our assumption, so  $S$  must be an Independent Set.

Therefore,  $S$  is an Independent Set iff  $V - S$  is a Vertex Cover.

- (b) [5 points] Show that the decision problem  $\text{Independent Set} = \{(G, k) \mid G \text{ contains an independent set of size at least } k\}$  is in NP.

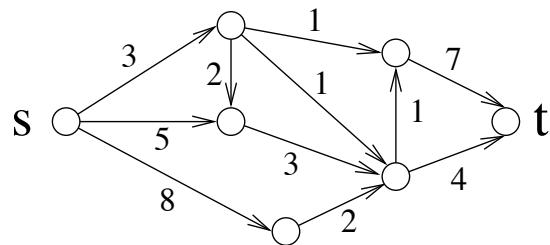
**Solution:** A set of  $k$  vertices forming an independent set is a proof that a given graph is an instance of Independent Set language. This proof can be verified trivially in polynomial time.

- (c) [10 points] We showed in class that the decision problem  $\text{Vertex Cover} = \{(G, k) | G \text{ contains a vertex cover of size at most } k\}$  is NP-complete. Use this to show that *Independent Set* is NP-complete.

**Solution:** We need to show that Vertex Cover reduces to Independent Set. By part (a), if a graph has a Vertex Cover of size  $k$ , then it has an Independent Set of size  $n - k$ . So, given an instance  $(G, k)$ , we can trivially make an instance of Independent Set  $(G, n - k)$ .

**Problem 5. Flows [27 points]**

Consider the following graph:



- (a) [10 points] What is the maximum flow in this graph? Give the actual flow as well as its value. Justify your answer.

**Solution:** The maximum flow is 6. From S, we route 3 along both the 3-capacity edge and the 5-capacity edge. The resultant flow will inundate the 1, 1 and 4-capacity edges that form a min-cut for the graph. By the Max-Flow/Min-Cut algorithm, this is a Max-Flow.

- (b) [5 points] True or false: For any flow network  $G$  and any maximum flow on  $G$ , there is always an edge  $e$  such that increasing the capacity of  $e$  increases the maximum flow of the network. Justify your answer.

**Solution:** False. A counterexample is a graph with two unit capacity edges in a chain. Increasing the capacity of a single edge will not increase the max flow, since the other edge is at capacity.

- (c) [5 points] Suppose you have a flow network  $G$  with integer capacities, and an integer maximum flow  $f$ . Suppose that, for some edge  $e$ , we increase the capacity of  $e$  by one. Describe an  $O(|E|)$ -time algorithm to find a maximum flow in the modified graph.

**Solution:** Add the new flow to the residual flow graph in  $O(|E|)$  time. Perform a tree traversal from the source node to detect whether a path now exists to the sink. If so, augment along that path and increase the maximum flow by one.

- (d) [7 points] Consider the decision problem:  $\text{Flow} = \{(G, s, t, k) \mid G = (V, E) \text{ is a flow network, } s, t \in V, \text{ and the value of an optimal flow from } s \text{ to } t \text{ in } G \text{ is } k\}$ .

Is  $\text{Flow}$  in NP? Why or why not?

**Solution:** Yes. We can explicitly compute the max-flow using Edmonds-Karp or another polynomial time max-flow algorithm. Since  $P \subseteq NP$ ,  $\text{Flow}$  must be in NP.

**Problem 6. Comparing Sets** [20 points]

Given two sets of integers  $S_1$  and  $S_2$ , each of size  $n$ , your job is to determine if  $S_1$  and  $S_2$  are identical.

Give a  $O(n)$  time randomized algorithm that always outputs “yes” if  $S_1 = S_2$  and outputs “no” with probability at least  $1 - 1/n$  if  $S_1 \neq S_2$ .

You may assume that we have a probabilistic model of computation in which generating a random number takes  $O(1)$  time and a comparison, a multiplication, and an addition of two numbers each takes  $O(1)$  time, regardless of the size of the two numbers involved.

**Hint:** Reduce the problem of comparing sets to the problem of comparing polynomials.

**Solution:** Reduce the problem of comparing sets to that of comparing two degree- $n$  polynomials  $P_1(x) = \prod_{r_1 \in M_1} (x - r_1)$  and  $P_2(x) = \prod_{r_2 \in M_2} (x - r_2)$  with  $n$  integer roots, where each polynomial is specified as a list of these  $n$  integer roots, i.e.  $P_1(x) = \{r_{11}, r_{12}, \dots, r_{1n}\}$ . The following randomized algorithm for comparing  $P_1(x)$  and  $P_2(x)$  which runs in  $O(n)$  time.

```
COMPARE-POLYNOMIALS($P_1(x), P_2(x)$)
1 Choose a random integer $a \in \{1, \dots, n^2\}$
2 Compute $P_1(a) = \prod_{i=1}^n (a - r_{1i})$ and $P_2(a) = \prod_{i=1}^n (a - r_{2i})$
3 if $P_1(a) = P_2(a)$
4 then output $M_1 = M_2$
5 else output $M_1 \neq M_2$
```

Line 2 of COMPARE-POLYNOMIALS( $P_1(x), P_2(x)$ ) runs in  $O(n)$  time, since each multiplication takes  $O(1)$  time. All other lines take  $O(1)$  time under the assumptions stated above. Now we will analyze the probability that COMPARE-POLYNOMIALS( $P_1(x), P_2(x)$ ) outputs the correct answer. If  $P_1(x) = P_2(x)$ , then COMPARE-POLYNOMIALS( $P_1(x), P_2(x)$ ) outputs the correct answer with probability 1.

A degree- $n$  polynomial  $P(x)$  has at most  $n$  distinct roots. Furthermore,  $P(a) \neq 0$  if  $a$  is not a root of  $P(x)$ . Thus,  $P(x)$  has at most  $n$  integer zeroes in the range  $\{1, \dots, n^2\}$ .

Let  $P_3(x) = P_1(x) - P_2(x)$ . Note that  $P_1(a) = P_2(a)$  exactly when  $P_3(a) = 0$ . Since the maximum degree of  $P_3(x)$  is  $n$ ,  $P_3(x)$  has at most  $n$  distinct roots. Thus, if we choose a random integer  $x$  from the range  $\{1, \dots, n^2\}$ , the probability that  $P_3(x)$  evaluates to 0 is at most  $1/n$ .

Therefore, if  $P_1(x) \neq P_2(x)$ , the probability that we choose  $a$  such that  $P_1(a) - P_2(a) = 0$  is at most  $\frac{1}{n}$ . Thus, the probability that COMPARE-POLYNOMIALS( $P_1(x), P_2(x)$ ) outputs the correct answer is at least  $1 - \frac{1}{n}$ .

**Problem 7. Finding the Topological Sort of a Complete Directed Acyclic Graph** [20 points]

Let  $G = (V, A)$  be a directed acyclic graph that has an edge between every pair of vertices and whose vertices are labeled  $1, 2, \dots, n$ , where  $n = |V|$ . To determine the direction of an edge between two vertices in  $V$ , you are only allowed to ask a *query*. A query consists of two specified vertices  $u$  and  $v$  and is answered with:

- “from  $u$  to  $v$ ” if  $(u, v)$  is in  $A$ , or
- “from  $v$  to  $u$ ” if  $(v, u)$  is in  $A$ .

Give matching upper and lower bounds (as functions of  $n$ ) for the number of queries required to find a topological sort of  $G$ .

**Solution:** This problem reduces to sorting. A query establishes an ordering between two nodes, i.e.  $(u, v)$ ’s existence can be interpreted as  $u > v$  and  $(v, u)$  as  $v < u$ . Since the graph is complete, there exists an ordering between every pair of nodes, in other words, we have a total ordering on the graph.

We can simply run a comparison-based sort to output a topological sort. The “max” element will be a source node with out-going edges to all other nodes. Similarly, the “min” element will be a sink node. Since *query* is effectively a comparison operator, there is a tight  $\Omega(n \log n)$  lower bound on performing a topological sort in this model.

1) 20 pts

Mark the following statements as **TRUE** or **FALSE**. No need to provide any justification.

[REDACTED]  
It is possible for a dynamic programming algorithm to have an exponential running time.

[REDACTED]  
In a connected, directed graph with positive edge weights, the Bellman-Ford algorithm runs asymptotically faster than the Dijkstra algorithm.

[REDACTED]  
There exist some problems that can be solved by dynamic programming, but cannot be solved by greedy algorithm.

[REDACTED]  
The Floyd-Warshall algorithm is asymptotically faster than running the Bellman-Ford algorithm from each vertex.

[REDACTED]  
If we have a dynamic programming algorithm with  $n^2$  subproblems, it is possible that the space usage could be  $O(n)$ .

[REDACTED]  
The Ford-Fulkerson algorithm solves the maximum bipartite matching problem in polynomial time.

[REDACTED]  
Given a solution to a max-flow problem, that includes the final residual graph  $G_f$ . We can verify in a *linear* time that the solution does indeed give a maximum flow.

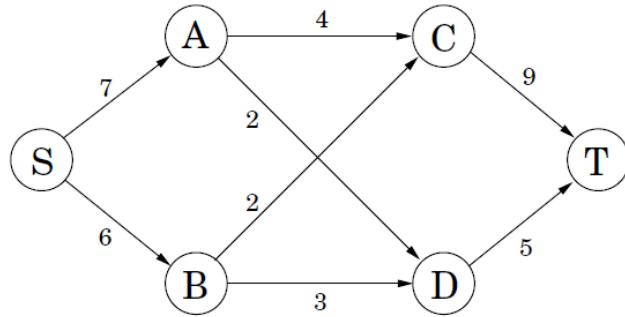
[REDACTED]  
In a flow network, a flow value is upper-bounded by a cut capacity.

[REDACTED]  
In a flow network, a min-cut is always unique.

[REDACTED]  
A maximum flow in an integer capacity graph must have an integer flow on each edge.

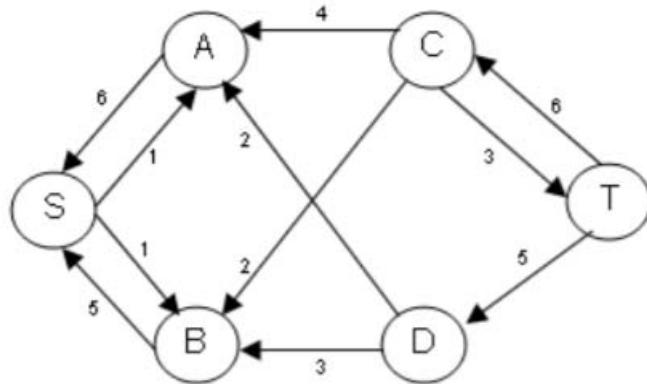
2) 20 pts.

You are given the following graph  $G$ . Each edge is labeled with the capacity of that edge.



- a) Find a max-flow in  $G$  using the Ford-Fulkerson algorithm. Draw the residual graph  $G_f$  corresponding to the max flow. You do not need to show all intermediate steps. (10 pts)

solution



**Grading Rubics:**

Residual graph: 10 points in total

For each edge in the residual graph, any wrong number marked, wrong direction, or missing will result in losing 1 point.

The total points you lose is equal to the number of edges on which you make any mistake shown above.

- b) Find the max-flow value and a min-cut. (6 pts)

Solution:  $f = 11$ , cut: ( $\{S, A, B\}$ ,  $\{C, D, T\}$ )

**Grading Rubics:**

Max-flow value and min-cut: 6 points in total

Max-flow: 2 points.

- If you give the wrong value, you lose the 2 points.

Min-cut: 4 points

- If your solution forms a cut but not a min-cut for the graph, you lose 3 points
- If your solution does not even form a cut, you lose all the 4 points

- c) Prove or disprove that increasing the capacity of an edge that belongs to a min cut will always result in increasing the maximum flow. (4 pts)

Solution: increasing (B,D) by one won't increase the max-flow

**Grading Rubics:**

Prove or Disprove: 4 points in total

- If you judge it "True", but give a structural complete "proof". You get at most 1 point
- If you judge it "False", you get 2 points.
- If your counter example is correct, you get the rest 2 points.

**Popular mistake:** a number of students try to disprove it by showing that if the min-cut in the original graph is non-unique, then it is possible to find an edge in one min-cut set, such that increasing the capacity of this does not result in max-flow increase.

But they did not do the following thing:

The existence of the network with multiple min-cuts needs to be proved, though it seems to be obvious. The most straightforward way to prove the existence is to give an example-network that has multiple min-cuts. Then it turns out to be giving a counter example for the original question statement.

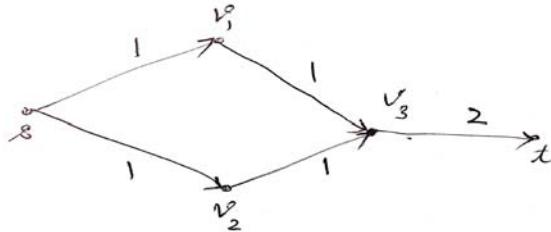
3) 15 pts.

Given a flow network with the source  $s$  and the sink  $t$ , and positive integer edge capacities  $c$ . Let  $(S, T)$  be a minimum cut. Prove or disprove the following statement:  
*If we increase the capacity of every edge by 1, then  $(S, T)$  still be a minimum cut.*

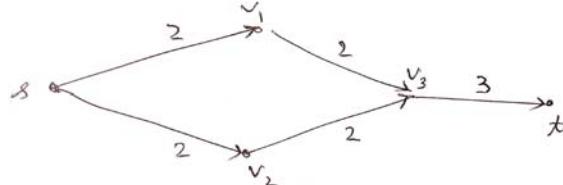
*False. Create a counterexample.*

An instance of a counter-example:

Initially, a  $(S, T)$  min cut is  $S : \{s, v_1, v_2\}$  and  $T : \{v_3, t\}$



After increasing capacity of every edge by 1,  $(S, T)$  is no longer a min-cut. We now have  $S' : \{s, v_1, v_2, v_3\}$  and  $T' : \{t\}$



Grading rubric:

If you try to prove the statement: -15

For an incorrect counter-example: -9

No credit for simply stating true or false.

4) 15 pts.

Given an unlimited supply of coins of denominations  $d_1, d_2, \dots, d_n$ , we wish to make change for an amount  $C$ . This might not be always possible. Your goal is to verify if it is possible to make such change. Design an algorithm by *reduction* to the knapsack problem.

- a) Describe reduction. What is the knapsack capacity, values and weights of the items? (10 pts.)

Capacity is  $C$ . (2 points)  
values = weights =  $d_k$ . (4points)

- You need to recognize that the problem should be modeled based on the unbounded knapsack problem with description of the reduction: (5 points)
- Explanation of the verification criteria (4 points)  
 $\text{Opt}(j)$ : the maximum change equal or less than  $j$  that can be achieved with  $d_1, d_2, \dots, d_n$   
 $\text{Opt}(0)=0$   
 $\text{Opt}(j) = \max[\text{opt}(j-d_i)+d_i] \text{ for } d_i \leq j$   
If we obtain  $\text{Opt}(C) = C$ , it means the change is possible.

- b) Compute the runtime of your verification algorithm. (5 pts)

$O(nC)$  (3 points), Explanation (2 points).

5) 15 pts.

You are considering opening a series of electrical vehicle charging stations along Pacific Coast Highway (PCH). There are  $n$  possible locations along the highway, and the distance from the start to location  $k$  is  $d_k \geq 0$ , where  $k = 1, 2, \dots, n$ . You may assume that  $d_i < d_k$  for  $i < k$ . There are two important constraints:

- 1) at each location  $k$  you can open only one charging station with the expected profit  $p_k$ ,  $k = 1, 2, \dots, n$ .
- 2) you must open at least one charging station along the whole highway.
- 3) any two stations should be at least  $M$  miles apart.

Give a DP algorithm to find the maximum expected total profit subject to the given constraints.

- a) Define (in plain English) subproblems to be solved. (3 pts)

Let  $\text{OPT}(k)$  be the maximum expected profit which you can obtain from locations  $1, 2, \dots, k$ .

Rubrics

Any other definition is okay as long as it will recursively solve subproblems

- b) Write the recurrence relation for subproblems. (6 pts)

$$\text{OPT}(k) = \max \{\text{OPT}(k - 1), p_k + \text{OPT}(f(k))\}$$

where  $f(k)$  finds the largest index  $j$  such that  $d_j \leq d_k - M$ , when such  $j$  doesn't exist,  $f(k)=0$

Base cases:

$$\text{OPT}(1) = p_1$$

$$\text{OPT}(0) = 0$$

Rubrics:

Error in recursion -2pts, if multiple errors, deduction adds up

No base cases: -2pts

Missing base cases: -1pts

Using variables without definition or explanation: -2pts

Overloading variables (re-defining  $n$ ,  $p$ ,  $k$ , or  $M$ ): -2pts

- c) Compute the runtime of the above DP algorithm in terms of  $n$ . (3 pts)

algorithm solves  $n$  subproblems; each subproblem requires finding an index  $f(k)$  which can be done in time  $O(\log n)$  by binary search.  
Hence, the running time is  $O(n \log n)$ .

Rubric:

$O(n^2)$  is regarded as okay

$O(d_n n)$  pseudo-polynomial is also okay if the recursion goes over all  $d_n$  values

$O(n)$  is also okay

$O(n^3), O(nM), O(kn)$  : no credit

- d) How can you optimize the algorithm to have  $O(n)$  runtime? (3 pts)

Preprocess distances  $r_i = d_i - M$ . Merge d-list with r-list.

Rubric

Claiming “optimal solution is already found in  $c$ ” only gets credit when explanation about pre-process is described in either part b or c.

Without proper explanation (e.g. assume we have, or we can do): no credit

Keeping an array of max profits: no credit, finding the index that is closest to the installed station with  $M$  distance away is the bottleneck, which requires pre-processing.

6) 15 pts.

A group of traders are leaving Switzerland, and need to convert their Francs into various international currencies. There are  $n$  traders  $t_1, t_2, \dots, t_n$  and  $m$  currencies  $c_1, c_2, \dots, c_m$ . Trader  $t_k$  has  $F_k$  Francs to convert. For each currency  $c_j$ , the bank can convert at most  $B_j$  Francs to  $c_j$ . Trader  $t_k$  is willing to trade as much as  $S_{kj}$  of his Francs for currency  $c_j$ . (For example, a trader with 1000 Francs might be willing to convert up to 200 of his Francs for USD, up to 500 of his Francs for Japanese's Yen, and up to 200 of his Francs for Euros). Assuming that all traders give their requests to the bank at the same time, describe an algorithm that the bank can use to satisfy the requests (if it can).

a) Describe how to construct a flow network to solve this problem, including the description of nodes, edges, edge directions and their capacities. (8 pts)

**Bipartite graph:** one partition traders  $t_1, t_2, \dots, t_n$ . Other, available currency,  $c_1, c_2, \dots, c_m$ .

Connect  $t_k$  to  $c_j$  with the capacity  $S_{kj}$

Connect source to traders with the capacity  $F_k$ .

Connect available currency  $c_j$  to the sink with the capacity  $B_j$ .

Rubrics:

- Didn't include supersource (-1 point)
- Didn't include traders nodes (-1 point)
- Didn't include currencies nodes (-1 point)
- Didn't include supersink (-1 point)
- Didn't include edge direction (-1 point)
- Assigned no/wrong capacity on edges between source & traders (-1 point)
- Assigned no/wrong capacity on edges between traders & currencies (-1 point)
- Assigned no/wrong capacity on edges between currencies & sink (-1 point)

b) Describe on what condition the bank can satisfy all requests. (4 pts)

If there is a flow  $f$  in the network with  $|f| = F_1 + \dots + F_n$ , then all traders are able to convert their currencies.

Rubrics:

- Wrong condition (-4 points): Unless you explicitly included  $|f| = F_1 + \dots + F_n$ , no partial points were given for this subproblem.
- In addition to correct answer, added additional condition which is wrong (-2 points)
- No partial points were given for conditions that satisfy only some of the requests, not all requests.
- Note that  $\sum S_{kj}$  and  $\sum B_j$  can be larger than  $\sum F_k$  in some cases where bank satisfy all requests.
- Note that  $\sum S_{kj}$  can be larger than  $\sum B_j$  in some cases where bank satisfy all requests.
- It is possible that  $\sum S_{kj}$  or  $\sum B_j$  is smaller than  $\sum F_k$ . In these cases, bank can never satisfy all requests because max flow will be smaller than  $\sum F_k$ .

- c) Assume that you execute the Ford-Fulkerson algorithm on the flow network graph in part a), what would be the runtime of your algorithm? (3 pts)

$O(n m |f|)$

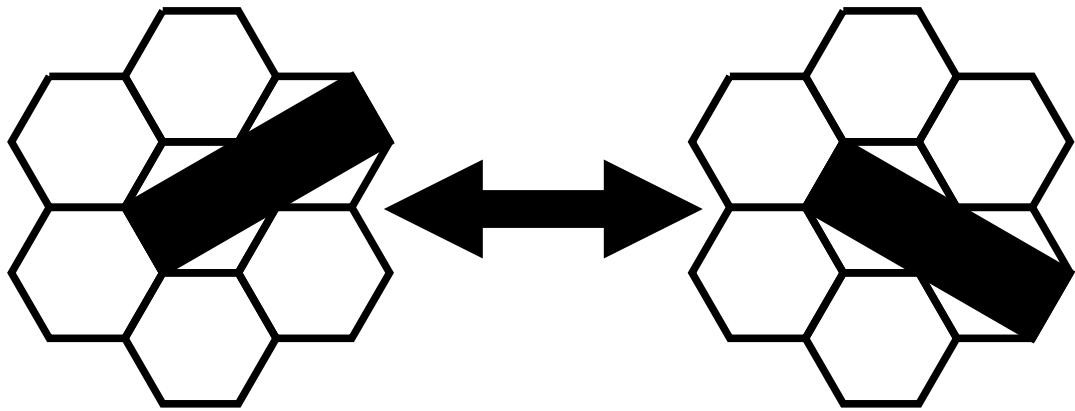
Rubrics:

- Flow( $|f|$ ) was not included (-1 point)
- Wrong description (-1 point)
- Computation is incorrect (-1 point)
- Notation error (-1 point)
- Missing big O notation (-1 point)
- Used big Theta notation instead of big O notation (-1 point)
- Wrong runtime complexity (-3 points)
- No runtime complexity is given (-3 points)

# CSE101 Homework 1 Solutions

Winter 2015

**Question 1** (Moving, 20 points). *Alice is attempting to move a table from one side of her apartment to the other. The apartment is laid out on a hexagonal grid. The table is large enough that it takes up two adjacent hexagons at any given time. Alice can move the table by leaving one end fixed and rotating the other end either clockwise or counterclockwise one unit as shown below:*



*Unfortunately, Alice's apartment has several immovable obstacles, each taking up a full hexagon so that the table cannot be moved to overlap any obstacle. Produce an efficient algorithm that given the layout of Alice's apartment, the location of obstacles and the desired starting and ending configurations of the table, determines whether or not it is possible for Alice to move the table from the starting configuration to the ending one.*

**Solution 1.** The problem of rearranging Alice's table can be abstracted as an undirected graph reachability problem. Each node in the graph, which we'll call  $G_T = (V_T, E_T)$ , corresponds to a unique table position, and an edge between two nodes means that the two positions can be reached from each other by a single 60-degree rotation of the table. (Since any move can be undone, the edges are undirected.) Since the table occupies two adjacent cells, each pair of adjacent cells in the apartment gets a unique node  $V_T$ , unless one of those cells contains an immovable object, in which case the node is not included. (Alternatively, we could leave the node in  $V_T$ , but since the table cannot occupy that space, it would have no edges with any other table positions.)

Once we have constructed  $G_T$ , we can simply run a partial depth-first-search on the graph, starting at the node corresponding to the table's starting position (call it  $s$ ), and ending when  $s$  is popped off of the stack. (It is a "partial" search because it is not guaranteed to visit all nodes or edges, only the ones reachable from  $s$ .) If the destination configuration ( $t$ ) is visited before this partial depth-first-search terminates, then there is a way to move the table to the destination. Otherwise, there is no configuration path which moves the table from  $s$  to  $t$ .

So we need an efficient algorithm that, given Alice's apartment layout, generates the graph  $G_T$ . We'll assume that Alice's apartment is given as a graph  $G_A = (V_A, E_A)$ , where each node in  $V_A$  is a cell, and an edge  $\{x, y\}$  indicates that cells  $x$  and  $y$  are adjacent. Assume we have  $G_A$  in adjacency list form, so that  $A(x)$  denotes the set of vertices adjacent to vertex  $x$ . The following algorithm has time complexity  $O(|V_A|)$ :

```

procedure makeGT(G_A):
 (V_A, E_A) <- G_A // parse G_A as V_A, E_A tuple
 V_T = E_A // every edge in G_A is a node in G_T
 E_T = {}

 for u in V_A:
 if u contains immovable object, skip u
 // for each pair of nodes adjacent to cell u
 for v in A(u):
 if v contains immovable object, skip v
 for w in A(u):
 if w contains immovable object, skip w

 if w in A(v): // can rotate table around u
 E_T <- E_T union { {u, v}, {u, w} } // constant time in Adjacency-list form

 return G_T = (V_T, E_T)

```

Even though there are several nested loops in this algorithm, we end up doing a constant amount of work per node  $u$ . This is because each cell is adjacent to at most 6 cells, and each table configuration is adjacent to at most 4 other configurations (2 per side of the table; a clockwise rotation and a counterclockwise one). We could technically improve this algorithm, since it visits each edge in  $E_A$  multiple times, but this will not improve its big- $O$  complexity.

Once we run this procedure and get  $G_T$ , we run the partial DFS beginning at  $s$ . Ordinarily, we would say that DFS has running time  $O(|V_T| + |E_T|)$ , but ultimately we want to get the time complexity as a function of  $|V_A|$  and  $|E_A|$ . This will take a few steps.

First, we observe that in a given configuration, the table can be moved to at most 4 neighboring configurations. This means that the degree of each vertex in  $V_T$  is at most 4. Therefore, the sum of the degrees of each node in  $V_T$  is less than  $4|V_T|$ , so that  $|E_T|$  is bound by  $2|V_T|$ , and therefore the DFS has time complexity  $O(|V_T| + 2|V_T|) = O(|V_T|)$ .

Next, we note that by the definition of  $G_T$ ,  $V_T = E_A$ , since each configuration of the table can be mapped to an edge in  $G_A$ , i.e., a pair of adjacent cells in the apartment. Therefore the DFS has time complexity  $O(|E_A|)$ .

Finally, we note that  $|E_A|$  is bound by the number of cells in the apartment, since each cell is adjacent to at most 6 other cells. Therefore, the DFS has time complexity  $O(|V_A|)$ .

So, generating  $G_T$  has complexity  $O(|V_A|)$ , and running the partial DFS takes time  $O(|V_A|)$ , so our overall algorithm has complexity  $O(|V_A|)$ .

**Question 2** (DAG Detection, 20 points). Give a linear time algorithm that given a directed graph determines whether or not it is a DAG.

**Solution 2.** The key question we must answer about the given graph is whether or not it contains a cycle. To answer this question, we can modify the standard depth-first-search algorithm to assign pre and post numbers to each node, as described in class. During the depth-first-search, if an edge  $(s, p)$  is found such that  $s$  is the node currently at the top of the stack, and  $p$  a node which has been assigned a pre number but no post number, then the algorithm reports a cycle. If the depth-first-search completes without detecting such an edge, then it reports that there are no cycles, and therefore the graph is a DAG.

This algorithm is guaranteed to find a cycle if one exists in the graph. One way to see this is that if there is a cycle, then each node in the cycle is reachable from itself. Therefore, if at some point the depth-first-search is exploring the paths out of a node  $s$  which is in a cycle, eventually the algorithm will visit  $s$  again before  $s$  is popped off the stack, i.e., before  $s$  has been given a post number. One subtle point

is that this event is only guaranteed to be occur if  $s$  is the first node in the cycle that has been visited, but since depth-first-search eventually visits all nodes, there is guaranteed to be one node in a cycle which is visited first.

If on the other hand there is no cycle, then there is no way for a node to be visited a second time before it is popped off the stack, i.e., before a post number is affixed to it. Therefore, our algorithm reports a cycle exactly when there is a cycle, without false positives or false negatives.

Since DFS (with pre/post number assignment) is  $O(|V| + |E|)$ , and checking a node to see if it has a pre or post number is a constant amount of work per edge, this algorithm has time complexity  $O(|V| + |E|)$ .

**Alternate Solution:** Another way to do it is as follows. Run DFS on the graph computing pre- and post- numbers, then check whether or not  $\text{post}(u) > \text{post}(w)$  for each edge  $(u, w) \in E$ . We claim that this holds for all edges if and only if the graph is a DAG. On the one hand, if  $G$  is a DAG, then we know from class, that the above relation holds for all edges. On the other hand, if all edges go from vertices with larger post numbers to smaller post numbers, there cannot be a cycle, because the vertex in the cycle with the smallest post number could not have an edge leading to another vertex in the cycle. The runtime of the algorithm is  $O(|V| + |E|)$  for the DFS and then  $O(|E|)$  to run the check over all edges, for a total runtime of  $O(|V| + |E|)$ .

Essentially, what we are doing here is pretending that our graph is a DAG, running topological sort, and then seeing if we found an actual linear order.

**Question 3** (Longest Path, 20 points). Find an efficient algorithm that given a DAG,  $G$ , finds the length of the longest path in  $G$ . Hint: topologically sort and compute the lengths of the longest path starting from  $v$  for each  $v$  in some order.

**Solution 3.** To compute the longest path in a DAG, we will keep track of the length of the longest path from each vertex in the graph. After each of these path lengths have been computed, the maximum value can be output.

Specifically, let  $\text{lenpath}[\cdot]$  be an array of length  $n = |V|$  with an index corresponding to each node in the graph. Define the array by  $\text{lenpath}[v] =$  the length of a longest path starting at  $v$ . Then each entry can be computed by the following:

$$\text{lenpath}[v] = 1 + \max_{u \in V : (v, u) \in E} \text{lenpath}[u].$$

First, it is clear that the second vertex on any path from  $v$  of non-zero length must be a neighbor vertex  $u$  such that  $(v, u) \in E$ . Let  $u_0$  be a vertex which maximizes the above expression (i.e.  $u_0$  is the destination of some edge from  $v$  such that  $\text{lenpath}[u]$  is maximum among all such  $u$ 's). Then the length of a longest path from  $v$  is exactly  $\text{lenpath}[u_0] + 1$  for the edge  $(v, u_0)$ . We will show this by a simple contradiction: assume there is a path from  $v$  of length greater than  $\text{lenpath}[u_0] + 1$ . Then the next vertex on this path must be some other neighbor  $u'$  of  $v$ , and the length of the path would be the length of a longest path from  $u'$  plus 1 for the edge  $(v, u')$ . But this is exactly  $\text{lenpath}[u'] + 1$ . Therefore it must be that  $\text{lenpath}[u'] > \text{lenpath}[u_0]$ , which contradicts  $u_0$  being the vertex that maximizes  $\text{lenpaths}$ .

Now, since computing  $\text{lenpath}[v]$  involves the values for vertices that are later in a topological ordering of the DAG, we will fill in the array in a reverse topological order. The base cases will be vertices with no out-going edges, for whom the longest path is necessarily of length 0. Below is the full algorithm:

```

LongestPath(G):
//input: DAG G=(V,E) in adjacency list format
//output: the length of a longest path in G

let n = |V|
initialize an array lenpath[] of length n

find a topological ordering of G
for every v in V in reverse topological order:
 if v has no outgoing edges:
 lenpath[v] = 0
 else:
 for every u in V such that (v,u) in E:

```

```

 if lenpath[v] < lenpath[u] + 1:
 lenpath[v] = lenpath[u] + 1

return max(lenpath)

```

*Topological sort takes  $O(|V| + |E|)$  time. Testing if a vertex has no outgoing edges is a simple look-up in the adjacency list, and then for every edge in the graph we do an addition and a comparison. Thus the total running time is  $O(|V| + |E|)$ .*

**Question 4** (Cheapest Reachability, 20 points). *Bob is planning a trip to Digraphia. Digraphia has many cities, but for unfathomable reasons travel between them is restricted, with certain pairs of cities connected by one-way roads. Bob has determined the cheapest available flights into and out of each city, and the available travel routes. Provide an efficient algorithm to determine the pair of cities  $s$  and  $t$  so that  $t$  is reachable from  $s$  and so that the cost of flying into  $s$  plus the cost of flying out of  $t$  is minimized. If there are  $n$  cities and  $m$  accessible one-way roads, you should aim for a runtime of  $O(n \log(n) + m)$  or better. Hint: Try running depth first search exploring vertices in increasing order of entry cost.*

*Can you do better if the roads are all bi-directional?*

**Solution 4.** *Assume that the land of Digraphia is abstracted adjacency list form, where nodes are cities and directed edges are one-way roads. Our algorithm is as follows:*

```

procedure find_min_cost(G):
 sorted_nodes = [sort nodes in ascending order of fly-in cost] // O(nlogn)
 min_cost = infinity

 for n in sorted_nodes:
 n.visited = false

 for n in sorted_nodes:
 min_out = modified_explore(n)
 min_cost = minimum((n.in_cost + min_out), min_cost)

 return min_cost

procedure modified_explore(n):
 if n.visited:
 return infinity
 n.visited = true

 min_out = n.out_cost
 for x in A(n): // for immediate reachable neighbors
 min_out = minimum(min_out, modified_explore(x))

 return min_out

```

*Sorting the nodes requires time  $O(n \log n)$ , and running the modified depth first search requires time  $O(n + m)$ , so the overall algorithm has complexity  $O((n \log n) + n + m) = O(n(\log n + 1) + m) = O(n \log n + m)$ .*

*Proving the correctness of this algorithm is a little tricky, because it does not explore all possible pairs  $(s, t)$  such that  $t$  is reachable from  $s$ . We must show that the pairs ignored by our algorithm cannot produce a minimum-cost pair of flights.*

*We can prove this through a crude kind of induction. We explore nodes in order of increasing fly-in cost, so the first node we explore,  $n_1$ , has the cheapest fly-in cost. Let  $R_x$  denote the set of nodes reachable from node  $x$ . Our algorithm will explore all pairs  $(n_1, t)$  such that  $t \in R_{n_1}$ , and find the minimum*

fly-out cost for each of these,  $o_{\min} = \min_{m \in R_{n_1}} \{m.\text{outCost}\}$ .

In the simplest case (the “base” case), all nodes in the graph are in  $R_{n_1}$ . In this case, the minimum cost and solution to the algorithm will be  $n_1.\text{inCost} + o_{\min}$ , since  $o_{\min}$  is the cheapest fly-out cost in the entire graph, and  $n_1$  has the cheapest fly-in cost, so no better solution is possible.

In the “inductive” case, some nodes have already been explored, but others have not. Since we explore the nodes in order of increasing fly-in cost, the next node we (non-trivially) explore,  $n_k$  will have the smallest fly-in cost of all unvisited nodes. Let  $T$  be the set of nodes which have already been visited. This exploration will visit the nodes  $R_{n_k} - T$ , the nodes reachable from  $n_k$  which have not already been visited. Thus, we now have  $o_{\min} = \min_{m \in R_{n_k} - T} \{m.\text{outCost}\}$ . So  $o_{\min}$  is no longer guaranteed to be the minimum fly-out cost of all nodes reachable from  $n_k$ , since it is possible that the node with the smallest fly-out cost,  $n_{\min}$ , has already been visited. However, if  $n_{\min}$  has already been visited, then that means that it was reachable from a node with a smaller fly-in cost than  $n_k.\text{inCost}$ , so the pair  $(n_k, n_{\min})$  cannot be the optimal solution.

Therefore, the pairs of nodes which are not examined by our algorithm cannot produce a minimum flight cost, and so our algorithm finds the optimal flight cost.

If the graph is undirected, things are easier. Then we only need to compute the connected component, and compare for each component the minimum fly-in cost plus the minimum fly-out cost. Since any city in the component is reachable from any other city in the component, this cost is both achievable and optimal. This algorithm takes  $O(|V| + |E|)$  time to compute the connected components of the graph. For each component  $C$ , it takes  $O(|V_C|)$  time to find the minimum fly-in and fly-out costs in the component, so the total time over all components is  $O(|V|)$ . Thus, the total runtime is only  $O(n + m)$ .

**Question 5** (Pre- and Post- Orderings, 20 points). For each of the following tables either demonstrate a graph so that the given values could be the pre- and post- numbers for the vertices of that graph under a depth first search, or show that no such graph exists.

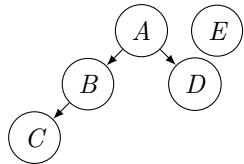
|  | Vertex | Pre | Post |
|--|--------|-----|------|
|  | A      | 1   | 10   |
|  | B      | 2   | 4    |
|  | C      | 3   | 5    |
|  | D      | 6   | 7    |
|  | E      | 8   | 9    |

|  | Vertex | Pre | Post |
|--|--------|-----|------|
|  | A      | 1   | 8    |
|  | B      | 2   | 5    |
|  | C      | 3   | 4    |
|  | D      | 6   | 7    |
|  | E      | 9   | 10   |

|  | Vertex | Pre | Post |
|--|--------|-----|------|
|  | A      | 1   | 10   |
|  | B      | 2   | 3    |
|  | C      | 4   | 8    |
|  | D      | 5   | 7    |

**Solution 5.** (a) Since vertex  $B$  has a pre- value of 2 and vertex  $C$  has a pre-value of 3, it must be that  $\text{explore}(G, C)$  was called from  $\text{explore}(G, B)$ , which in turn means that  $(B, C)$  is an edge in the graph. However,  $\text{pre}(B) < \text{pre}(C) < \text{post}(B) < \text{post}(C)$  is an impossible relationship since the pre- and post- numbers are interleaved. Therefore no graph could be given the pre- and post- values.

(b) The following is an example of a graph whose vertices could be given the pre- and post- values during a depth-first search:



- (c) No graph can be given the pre- and post- value. In particular,  $\text{pre}(D) = 5, \text{post}(D) = 7$  is impossible. Consider the following two cases: either we explore a vertex from  $D$  or we don't. If we explore a vertex, call it  $U$ , then  $\text{pre}(U) = \text{pre}(D) + 1 = 6$  and the clock must be updated at least once more for the post-visit of  $U$ , so  $\text{post}(U) \geq 7$ , but  $\text{post}(D) = 7$  so this is a contradiction. If we don't explore a vertex from  $U$ , immediately after we pre-visit  $D$ , we increment the clock and post-visit  $D$ , so  $\text{post}(D) = \text{pre}(D) + 1 = 6$ , another contradiction.

**Question 6** (Extra credit, 1 point). Approximately how much time did you spend working on this homework?

1) 20 pts

Mark the following statements as **TRUE** or **FALSE**. No need to provide any justification.

[**FALSE**]

A flow network with unique edge capacities has a unique min cut.

[**TRUE**/]

If a problem can be solved by dynamic programming, then it can always be solved by exhaustive search (Brute Force).

[**TRUE**/]

A divide and conquer algorithm acting on an input size of  $n$  can have a lower bound less than  $\Theta(n \log n)$ .

[**FALSE**]

If a flow in a network has a cycle, this flow is not a valid flow.

[**FALSE**]

In the divide and conquer algorithm to compute the closest pair among a given set of points on the plane, if the sorted order of the points on both X and Y axis are given as an added input, then the running time of the algorithm improves to  $O(n)$ .

[**TRUE**/]

In a flow network, an edge that goes straight from  $s$  to  $t$  is always saturated when maximum  $s - t$  flow is reached.

[**FALSE**]

The Bellman-Ford algorithm always fails to find the shortest path between two nodes in a graph if there is a negative cycle present in the graph.

[**TRUE**/]

If  $f$  is a max  $s-t$  flow of a flow network  $G$  with source  $s$  and sink  $t$ , then the capacity of the min  $s-t$  cut in the residual graph  $G_f$  is 0.

[**FALSE**]

In a dynamic programming solution, the space requirement is always at least as big as the number of unique sub problems.

[**FALSE**]

Decreasing the capacity of an edge that belongs to a min cut in a flow network may not result in decreasing the maximum flow.

2) 20 pts

A city is located at one node  $x$  in an undirected network  $G = (V, E)$  of channels. There is a big river beside the network. In the rainy season, the flood from the river flows into the network through a set of nodes  $Y$ . Assume that the flood can only flow along the edges of the network. Let  $c_{uv}$  (integer value) represent the minimum effort (counted in certain effort unit) of building a dam to stop the flood flowing through edge  $(u, v)$ . The goal is to determine the minimum total effort of building dams to prevent the flood from reaching the city. Give a pseudo-polynomial time algorithm to solve this problem. Justify your algorithm.

Algorithm:

1. Add node  $s$ , and add edges from  $s$  to each node in  $Y$ .
2. Set the capacity of each of these new edges as infinity.
3. Set city node  $x$  as the sink node. Then the flood network  $G$  becomes a larger network  $G'$  with source  $s$  and sink  $x$ .
4. Find the min  $s$ - $x$  cut on  $G'$  by running a polynomial time max-flow algorithm, where you can treat the cost of building a dam on each edge to be the capacity of this edge.
5. Build a dam on each edge in the min-cut's cut-set.

Complexity:

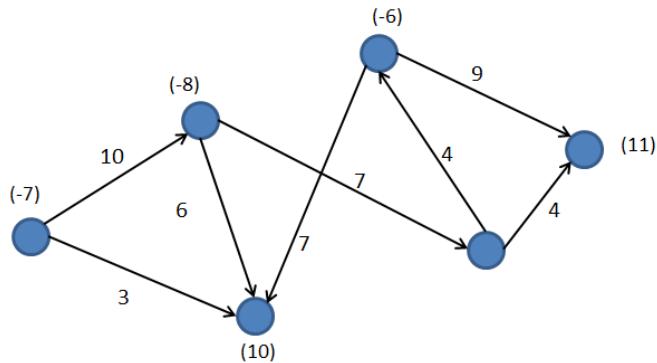
Since the construction of the new edges takes  $O(|Y|)$  times, together with a polynomial time max-flow algorithm to find the min-cut, the entire algorithm takes polynomial time.

Justification:

We're simply trying to separate  $x$  from  $Y$  by choosing edges with the minimum cost, which is a min-cut problem. But min-cut only works when  $Y$  is a single node. We fix this by creating a super source  $s$  directly connected to  $Y$ . But we want to make sure the min-cut's cut-set doesn't include any edge connecting  $s$ , because these edges do not belong to  $G$ . This is why we put the capacities arbitrarily large on these edges. Min-cut would then find the min-cost way to separate  $x$  from  $Y$ .

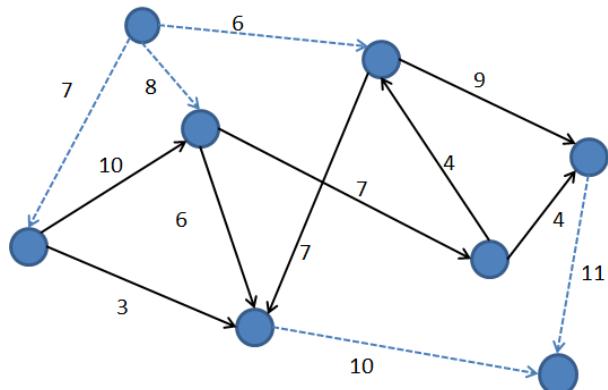
3) 20 pts

The following graph  $G$  is an instance of a circulation problem with demands. The edge weights represent capacities and the node weights (in parentheses) represent demands. A negative demand implies source.



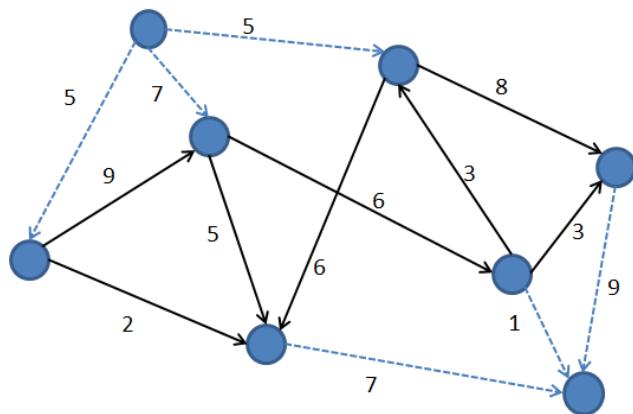
- (i) Transform this graph into an instance of max-flow problem.

**Solution:**



- (ii) Now, assume that each edge of  $G$  has a constraint of lower bound of 1 unit, i.e., one unit must flow along all edges. Find the new instance of max-flow problem that includes the lower bound constraint.

**Solution:**



4) 20 pts

There is a series of activities lined up one after the other,  $J_1, J_2, \dots, J_n$ . The  $i^{th}$  activity takes  $T_i$  units of time, and you are given  $M_i$  amount of money for it. Also for the  $i^{th}$  activity, you are given  $N_i$ , which is the number of immediately following activities that you cannot take if you perform that  $i^{th}$  activity. Give a dynamic programming solution to maximize the amount of money one can make in  $T$  units of time. Note that an activity has to be completed in order to make any money on it. State the runtime of your algorithm.

Solution:

Recurrence formula:

If  $T_i > t$  then  $\text{opt}(i, t) = \text{opt}(i+1, t)$ ,

Otherwise,  $\text{opt}(i, t) = \max(\text{opt}(i+1, t), \text{opt}(i + N_i + 1, t - T_i) + M_i)$

Boundary conditions:  $\text{opt}(i, t) = 0$  for  $i > n$  and all  $t$ ,

$\text{opt}(i, t) = 0$  for  $t < 1$  and all  $i$

To find the value of the optimal solution:

for  $i=n$  to 1 by -1

    for  $t=1$  to  $T$

        If  $T_i > t$  then  $\text{opt}(i, t) = \text{opt}(i+1, t)$ ,

        Otherwise,  $\text{opt}(i, t) = \max(\text{opt}(i+1, t), \text{opt}(i + N_i + 1, t - T_i) + M_i)$

    end for

end for

$\text{opt}(1, T)$  will hold the value of the optimal solution (maximum money that can be made). Complexity is  $O(nT)$

Given all the values in the two dimensional  $\text{opt}( )$  array, the optimal set of activities can be found in  $O(n)$

Overall complexity of the algorithm is  $O(nT)$  which is pseudopolynomial.

5) 20 pts

A polygon is called convex if all of its internal angles are less than  $180^\circ$  and none of the edges cross each other. We represent a convex polygon as an array  $V$  with  $n$  elements, where each element represents a vertex of the polygon in the form of a coordinate pair  $(x, y)$ . We are told that  $V[1]$  is the vertex with the least  $x$  coordinate and that the vertices  $V[1], V[2], \dots, V[n]$  are ordered counter-clockwise. Assuming that the  $x$  coordinates (and the  $y$  coordinates) of the vertices are all distinct, do the following.

Give a divide and conquer algorithm to find the vertex with the largest  $x$  coordinate in  $O(\log n)$  time.

**Solution:**

Since  $V[1]$  is known to be the vertex with the minimum  $x$ -coordinate (leftmost point), moving counter-clockwise to complete a cycle must first increase the  $x$ -coordinates and then after reaching a maximum (rightmost point), should decrease the  $x$ -coordinate back to that of  $V[1]$ . To see this more formally, we claim that there cannot be two distinct local maxima in the  $x$ -axis projection of the counter-clockwise tour. For the sake of contradiction, assume otherwise. Then there exists a vertical line that would intersect the boundary of the polygon at more than two points. This is impossible by convexity of the polygon since the line segments between the intersection points must lie completely in the interior of the polygon, thus contradicting our assumption. Thus,  $V_x[1 : n]$  is a unimodal array, and the first part of this question is synonymous with detecting the location of the maximum element in this array.

Consider the following algorithm:

- (a) If  $n = 1$ , return  $V_x[1]$ .
- (b) If  $n = 2$ , return  $\max\{V_x[1], V_x[2]\}$ .
- (c)  $k = \lceil \frac{n}{2} \rceil$ .
- (d) If  $V_x[k] > V_x[k - 1]$  and  $V_x[k] > V_x[k + 1]$ , then return  $V_x[k]$ .
- (e) If  $V_x[k] < V_x[k - 1]$  then call the algorithm recursively on  $V_x[1 : k - 1]$ , else call the algorithm recursively on  $V_x[k + 1 : n]$ .

**Complexity:** If  $T(n)$  is the running time on an input of size  $n$ , then beside a constant number of comparisons, the algorithm is called recursively on at most one of  $V_x[1 : k - 1]$  (size =  $k - 1 = \lceil \frac{n}{2} \rceil - 1 \leq \lfloor \frac{n}{2} \rfloor$ ) or  $V_x[k + 1 : n]$  (size =  $n - k = n - \lceil \frac{n}{2} \rceil = \lfloor \frac{n}{2} \rfloor$ ). Therefore,  $T(n) \leq T(\lfloor \frac{n}{2} \rfloor) + \theta(1)$ .

Assuming  $n$  to be a power of 2, this recurrence simplifies to  $T(n) \leq T(n/2) + \theta(1)$  and invoking Master's Theorem gives  $T(n) = O(\log n)$ .

- (23) **T F** The Bellman-Ford algorithm is not suitable if the input graph has negative-weight edges.

**Answer:** False. Bellman Ford is used when there are negative weight edges, it's Dijkstra's algorithm that cannot be used.

- (24) **T F** Memoization is the basis for a top-down alternative to the usual bottom-up version of dynamic programming.

**Answer:** True

- (25) **T F** Given a weighted, directed graph  $G = (V, E)$  with no negative-weight cycles, the shortest path between every pair of vertices  $u, v \in V$  can be determined in  $O(V^3)$  worst-case time.

**Answer:** True. Floyd-Warshall's algorithm performs exactly this in  $O(V^3)$  time.

- (26) **T F** For hashing an item into a hash table in which collisions are resolved by chaining, the worst-case time is proportional to the load factor of the table.

**Answer:** False. Even with a low load factor (say  $\frac{1}{2}$ ) in the worst case you can get all  $n$  elements to hash to the same slot

- (27) **T F** A red-black tree on 128 keys must have at least 1 red node.

**Answer:** True

- (28) **T F** The move-to-front heuristic for self-organizing lists runs no more than a constant factor slower than any other reorganization strategy.

**Answer:** True. In recitation we've seen that it's *4-competitive*, which means that the best algorithm that can exist ("God's algorithm") could only do better by a factor of 4.

- (32) **T F** The Floyd-Warshall algorithm solves the all-pairs shortest-paths problem using dynamic programming.

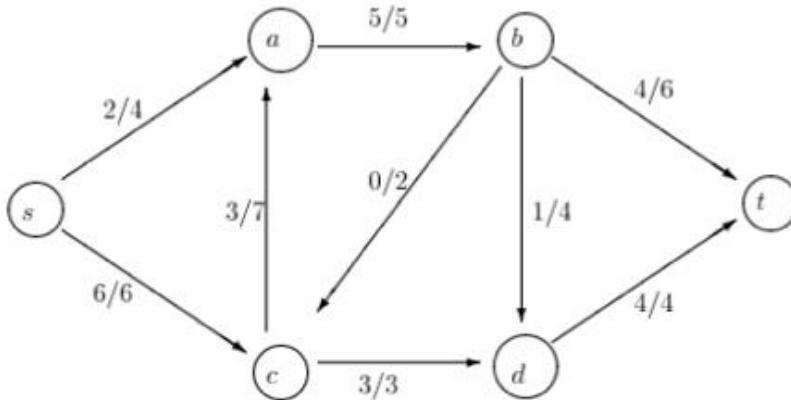
**Answer:** True

- (33) **T F** A maximum matching in a bipartite graph can be found using a maximum-flow algorithm.

**Answer:** True. We've seen this in recitation.

- (10) **T F** The figure below describes a flow assignment in a flow network. The notation  $a/b$  describes  $a$  units of flow in an edge of capacity  $b$ .

True or False: The following flow is a maximal flow.



**Answer:** True. The flow pushes 8 units across the cut  $\{s, a, c\}$  vs.  $\{b, d, t\}$  has capacity 8. The cut must be maximum by the Max-Flow Min-cut theorem.

- (43) **T F** For any network and any maximal flow on this network there always exists an edge such that increasing the capacity on that edge will increase the network's maximal flow.

**Answer:** False. There may be *two* min-cuts, with the edge in question increasing the capacity of only one of them. The other one will remain as it is, preventing the max-flow from increasing any further

- (36) **T F** If all edge capacities in a graph are integer multiples of 5 then the maximum flow value is a multiple of 5.

**Answer:** True. Consider the minimum cut. It is made up of edges with capacities that are multiples of 5, so the capacity of the cut (sum of capacities of edges in the cut) must be a multiple of 5. By the maxflow-mincut theorem, the maximum flow has the same value.

- (37) **T F** For any graph  $G$  with edge capacities and vertices  $s$  and  $t$ , there always exists an edge such that increasing the capacity on that edge will increase the maximum flow from  $s$  to  $t$  in  $G$ . (Assume that there is at least one path in the graph from  $s$  to  $t$ .)

**Answer:** False. There may be more than one minimum cut. For example, consider the following graph. There is no single edge you can increase to increase the flow.

