Exam 1 - Rubric

Q12-21: True/False

12. Finding the *k*-th minimum element in an array of size *n* using a binary min-heap takes O(*k log n*) time. **[False]**

13. We can merge any two arrays each of size *n* into a new sorted array in O(*n*). **[False]**

14. The shortest path in a weighted directed acyclic graph can be found in linear time. **[True]**

15. Given a weighted planar graph. Prim's algorithm using a binary heap implementation will outperform Prim's algorithm using an array implementation. **[True]**

Explanation:

In Planar graph, E <= 3*V - 6. Prim's algorithm using array has the complexity of O(V^2 + E) = O(V^2). Prim's algorithm using binary heap has the complexity of O(V*logV + E*logV) = O(V*logV). Therefore, Prim's algorithm using array will run slower than Prim's algorithm using binary heap.

16. If f(n) = Ω (n log n) and g(n) = O(n^2 log n), then f(n) = O(g(n)). **[False]**

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

18. There are no known polynomial-time algorithms to solve the fractional knapsack problem. **[False]**

19. In a knapsack problem, if one adds another item, one must solve the whole problem again in order to find the new optimal solution. **[False]**

20. Given a dense undirected weighted graph, the time for Prim's algorithm using a Fibonacci heap is O(E). **[True]**

21. In a binomial min-heap with *n* elements, the worst-case runtime complexity of finding the second smallest element is O(1). **[False]**

Q22-31: Multiple Choice

22. The solution to the recurrence relation T(n) = 8 T(n/4) + O(n1.5 log n) by the Master theorem is

$O(n^{1.5} \log^2 n)$

23. There are four alternatives for solving a problem of size n by diving it into subproblems of a smaller size. Assuming that the problem can be divided into subproblems in constant time, which of the following alternatives has the smallest runtime complexity?

we solve 5 subproblems of size n/3, with the combing cost of Θ (n log n)

24. Which of the following statements is true?

If an operation takes O(n) expected time, then it may take O(1) amortized time.

25. Which of the following properties does not hold for binomial heaps?

FINDMIN takes O(log n) worst-case time

26. There are n people who want to get into a movie theater. The theater charges a toll for entrance. The toll policy is the following: the i-th person is charged k2 when i = 0 mod k (this means that i is a multiple of k), or zero otherwise. What is the amortized cost per person for entering the theater? Assume that n > k.

$\Theta(k)$

27. What is the runtime complexity of the following code:

```
void exam1(int n) {
  int count = 0;
  for (int i=n/2; i<=n; i++)
    for (int k=1; k<=n; k = 2 * k)
      for (int j=1; j<=n; j = j * 2)
        count ++;
}
```

$\Theta(n \log^2 n)$

28. Given an order k binomial tree Bk, deleting its root yields

***k* binomial trees**

29. Kruskal's algorithm cannot be applied on

directed and weighted graphs

30. Consider a recurrence T(n) = T(a n) + T(b n) + n, where $0 < a \leq b < 1$ and $a + b = 1$. Which of the following statements is true?

$T(n) = \Theta(n \log n)$

31. Consider an undirected connected graph G with all distinct weights and five vertices {A, B, C, D, E} and. Suppose CD is the minimum weight edge and AB is the maximum weight edge. Which of the following statements is false?

No minimum spanning tree contains edge AB

Long Questions

Q1. Amortized Cost

Consider a singly linked list data structure that stores a sequence of items in any order. To access the item in the i-th position requires i time units. Also, any two contiguous items can be swapped in constant time. The goal is to allow access to a sequence of n items in a minimal amount of time. The TRANSPOSE is a heuristic that improves accessing time if the same is accessed again. TRANSPOSE works in the following way: after accessing x, if x is not at the front of the list, swap it with its next neighbor toward the front. What is the amortized cost of a single access?

Solutions:

Aggregate method

Suppose the singly linked list is:

$$head \rightarrow a\_1 \rightarrow a\_2 \rightarrow \cdots \rightarrow a\_n$$

Considering the worst case of accessing these n items, we start access from a_n to a_1.

We first access to a_n which takes n cost. Then we swap it with a_(n-1) the next step, we want to access to the a_(n-1), again it takes n cost. Accessing a_n and a_(n-1) takes 2*n cost.

Then, we access the a_(n-2) which takes n-2 cost. Then we swap it with a_(n-2). In the next step, we want to access to the a_(n-3), again it takes n-2 cost. Accessing a_(n-2) and a_(n-3) takes 2*(n-2) cost.

Repeatedly do that, we will get the following summation of cost:

2*2 + 2*4 + 2*6 + ... + 2*(n-2) + 2*n + n(cost of swap) = 4*(1+2+3+...+n/2) + n = n(n+2)/2 + n = O(n^2)

Therefore, the amortized cost per access is O(n).

Rubrics:

Please read the textbook about the definition of amortized cost: Amortized analysis gives the average performance of each operation in the worst case.

[12 pts]

- [5 pts] State the worst case scenario is to access from a_n to a_1.We cannot perform swap along
    - The order of accessing a sequence of n elements is not the worst case: -1 pt
    - accessing order is not specified: -2 pts
    - only considering accessing one element: -2 pts
- [6 pts] Analyze and state the summation of total cost which is O(n^2)
    - Wrong explanation or no explanation: - 6pt
        - Explaining using i (this is not the worst case) -4 pt
    - Correct explanations:
        - Swap can not be performed once after each access. It swaps (i-1)th element with i th element. incorrect statement about it will get -1 pt
        - lack of sufficient explanation, for example, how to calculate the total cost: -2 pts
        - The cost of transpose is not considered: -2 pts
            - Transpose actually degrade the performance in the worst case, any statement that transpose increase the performance: - 1 pt
        - the total cost is incorrect: -2 pts
- [1 pts] Analyze and state the amortized cost per access which is O(n).
    - Answer is not O(n): -1 pt

Q2. Shortest Path Problem

Consider a network of computers represented by a directed weighted graph where each vertex is a computer, and each edge is a wire between two computers. An edge weight w(*u*, *v*) represents the probability that a network packet (unit of binary data) going from computer *u* reaches computer *v*. Our goal is to find a network path from a given source *s* to a target computer *t* such that a packet has the highest probability of reaching its destination. In other words, we are looking for a path where the product of probabilities is maximized.

Solutions:

Solution 1:
Use a modification of Dijkstra to find the maximum product of edge weights.
Use a set of unexplored edges like in Dijkstra (this may be a max-heap instead of a min-heap or a priority queue).
Initialize the source node with a 0 and all other nodes with a negative infinity.

```
For every iteration:
    Pick the node x with highest value and add it to a final_path list
    For each of the neighbors n of x:
        If node_value(n) < node_value(x)*weight(x,n):
            node_value(n) = node_value(x)*weight(x,n)
```

Keep iterating and if the picked node happens to be the target node, then we have reached the solution and we return the final_path list (can be a string, set etc as well)

Solution 2:

Take a log of all the edge weights and find the maximum length path using Dijkstra modification where we initialize the same as we did in solution 1 above (i.e with negative infinity and 0 for source node. Then we find the longest path using a modification when picking up the node as above (pick the max node value using a max heap or priority queue). The rest goes, the same as Dijkstra.

Rubric:

[+2 Marks] Using correct initialization of nodes
[+6 Marks] Correct Algorithm Description with correct node updation equation used
[+2 Marks] Returning the entire path and not just the max probability

[-1 Marks] Partially Correct Initialization

[-3 Marks] Incorrect node_value updation
[-1 Marks] Did not mention returning the entire path

Q3-7. Greedy

In this problem you are to find the best order in which to solve your exam questions. Suppose that there are *n* questions with points $p_1, p_2, \ldots, p_n$ and you need time t_k to completely answer the *k*-th question. You are confident that all your completely answered questions will be correct (and get you full credit for them), and you will get a partial credit for an incompletely answered question, in proportion of time spent on that question. Assuming that the total exam duration is T, design a greedy algorithm to decide on the optimal order. Solve the problem in the following five steps.

Question 3 (6 points) Describe your greedy algorithm

This is similar to the fractional knapsack problem. The greedy algorithm is as follows. Calculate $\frac{p_i}{t_i}$ for each $1 \le i \le n$ and sort the questions in descending order of $\frac{p_i}{t_i}$. Let the sorted order of questions denoted by $s(1), s(2), \ldots, s(n)$. Answer questions in this order until $s(j)$ such that $\sum_{k=1}^{j} t_{s(k)} \le T$ 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 questions $s(j+1)$ in the time remaining.

Rubric:

- If the greedy criteria is wrong, 0 point.
- If only stated solving the remaining problems with the highest ratio: 2 points.
- Sorting in Descending order: 4 points (keyword must occur. Not keyword, no points) Other descriptions are fine: largest to smallest, non-increasing, etc.
- Sort without order or wrong order: -2 points
- It's OK not to mention the corner case.
- Some students use the word "order", 'rank' 'arrange', 'choose' instead of "sort", which is also OK as long as they mention the order.
- Some students don't mention order after sorting, but mention in some other places such as the algorithm loop. Give full points for this case.
- Some students do the inverse version. Compute $$\frac{t_i}{p_i}$$ and sort with ascending/increasing order. Get full points for this case.
- Incorrect notations: -2 points.

Question 4 (3 points) Compute the runtime complexity of your algorithm.

Calculate the $\frac{p_i}{t_i}$ for each questions: O(n)

Sorting questions in order of $\frac{p_i}{t_i}$: O(nlogn)

Processing questions: O(n)
Overall: O(2n+nlogn)=O(nlogn)

Rubric:

- considering sorting the pi/ti ratios: 1 point
- considering at least one of calculating pi/ti and processing questions: 1 point
- Final result: 1 point (must be explicitly mentioned)
- If only the final result is given and it is correct without explanation: 3 points.

Question 5 (5 points) State and prove the induction base case (use a proof by contradiction here)
Let P(i) denote an optimal solution where question s(i) is part of the solution. 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 $(\frac{p_{s(1)}}{t_{s(1)}} - \frac{p_a}{t_a})\, min(t_{a,} T, t_{s(1)}) \geq 0$ since $\frac{p_i}{t_i}$ is the 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.

A different way to state the base case: if the optimal solution contains problems p_{i,1}, p_{i, 2}...p_{i, k} and the greedy solution contains problems p_{j, 1}, p_{j, 2},... p_{j, m}. Assume they are all sorted by p/t in the descending order, then we show that p_{i,1}=p_{j,1}, meaning that the problem with the highest p/t ratio must be included in the optimal solution.

Rubric:

- State the base case: 3 points.
- Prove its optimality: 2 points. The idea is that there always exists another problem that can be swapped with s(1) if s(1) is not a part of the optimal solution to improve the total credits.

Common mistakes:

- It is incorrect to perform induction on the time because time is a continuous value. (0 points)
- It is also incorrect to perform induction on the total number of questions. The induction should be performed on the index of questions answered in the p/t order, not the total.
- It is incorrect to prove by showing that after completing the same number of questions, the points earned by greedy must be greater than the points earned by the optimal solution. (0 points)

- It is incorrect to argue that the "efficiency" (defined as p/t) of the greedy solution always stays ahead of the "optimal" solution. This is equivalent to using your approach to prove the optimality of your approach.
- It is incorrect to argue that the first l problems of the greedy and the optimal solution match and prove for the "l+1" problem. The only thing that we can say that there "exists" such an optimal solution that matches the greedy choice.
- When proving the inductive step, many students use "replacement" to show that the credits will increase when replacing the "new" problem with any problem that has less p/t. This is incorrect because when you do "replacement", due to the time constraints, you may replace more than one problem.

Question 6 (3 points) State the inductive hypothesis
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.

Rubric:

- Must state that our algorithm leads to optimality for the first l questions.
- 2 points deduction if not clear that our algorithm is optimal (e.g., there is only talk of an optimal algorithm).
- Hypothesis on time T is incorrect. (0 points)

Question 7 (7 points) State and prove the inductive step
Let $P(1),\ P(2),\ ...,\ P(l)$ be true for some arbitrary l, i.e., the set of questions $\{s(1),\ s(2),\ ...,\ s(l)\ \}$ are part of an optimal solution O. It is clear that Oshould also be optimal with respect to the remaining questions $\{1,\ 2,\ ...,\ n\} \setminus \{s(1),\ s(2),\ ...,\ s(l)\}$ 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 highest 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.

Rubric:

- No partial credit, but equations can be paraphrased in different ways.

Q8-11. Dynamic Programming

Columbus wants to travel the Venice river as fast as possible. There are cities 1, 2, ..., n located by the river. There are some boats connecting two cities with different speeds. For each i < j, there is a boat from city i to city j which takes T[i, j] minutes to travel. Starting from city 1, help Columbus to calculate the shortest time possible to reach city n. Your proposed algorithm should have time complexity O(n^2). For example, if n = 4 and the set of boats are { T[1, 2] = 3, T[1, 3] = 2, T[1, 4] = 7, T[2, 3] = 1, T[2, 4] = 1, T[3, 4] = 3 }, then your algorithm should return 4 minutes (for the route 1 -> 2 -> 4).
Design a dynamic programming algorithm for solving this problem. Please complete the following steps.

1. Define (in plain English) the subproblems to be solved.
2. Write the recurrence relation for the subproblems. Also write the base cases.
3. Use plain English to explain how you use that recursive relation to obtain a solution. You don't need to prove the correctness.
4. State the runtime of the algorithm. Explain your answer

Solution

1. Defining sub-problem If Columbus reaches city i, then the next move is to choose the city $j > i$ to jump to after city i. So, let's define $d[i]$ as the shortest time possible to pass from city i to city n.
2. Recurrence Relation The recurrence relation is the minimum total time among all the possible selection of city $i < j$ to jump to. The base case is $d[n] = 0$.

$$d[n] = 0$$
$$d[i] = min(\{T_{i,j} + d[j] | i < j \leq n\}), 1 \leq i < n$$

3. Explanation: Based on the definition of the sub-problem, the value of $d[i]$ tells us what would be the minimum time (fastest route) to reach city n starting from city i. The base case is $d[n] = 0$, since if Columbus is at city n already, they he does not need to ride any boat and the minimum time is 0. Now assume we know the best answer for any $i + 1 \leq j \leq n$, and we want to find the best answer for i. If Columbus is located at city i, he can move to any city j after city i. If he jumps to city $j > i$, since we already know the best answer for city j and it is $d[j]$, so the total time of travel would be $T_{i,j} + d[j]$. Now, for city i for should select the next city j such that the total travel time would be minimum. That is why we have $d[i] = min(\{T_{i,j} + d[j] | i < j \leq n\}), 1 \leq i < n$.
4. Time Complexity There are n number of unique sub-problems and the update rule for each sub-problem takes $O(n)$ time. So, the total time complexity is $O(n^2)$.

```
d[n] = 0
for i=n-1 to 1:
    best_jump = t[i][n]
    for j=i+1 to n-1:
        best_jump = min(best_jump, t[i][j] + d[j])
    d[i] = best_jump
return d[1]
```

Rubric (20 points):

- Part 1 (6 points): Clear definition of subproblems
    - If they defined a 2D table for dp like OPT[i, j] -3 points (since they cannot achieve O(n^2) using this).
    - If they explain the idea but did not define OPT -4 points
- Part 2 (9 points): Correct recurrence formula (7 points) and correct base case (2 points).
- Part 3 (5 points): Explaining the reason for the recurrence relation.
- Part 4 (4 points): Correct runtime complexity.
    - -4 points if they just say time complexity is O(n^2). This is part of the question not the solution!
    - -2 points if they only mention the number of sub-problem or the time it takes to update.
    - -2 points if they use wrong reasoning of why O(n^2) (while their algorithm is O(n^3))
    - -1 point if explanation is incorrect or not enough
    - -2 point if runtime is not O(n^2)
- If they have a O(n^3) algorithm and every part of the solution is correct, they can get 15/20 points (-3 points for part 1 and -2 points for part 4).
- If they use the following idea and it was correct no reduction of points:
    - Define D[i] as the minimum time to pass from city 1 to i. And then the recurrence formula would be:

$D[1] = 0$

$D[i] = \min\{T_{j, i} + D[j] \mid 1 <= j < i \}$

CSCI 570 - Spring 2021 - Main Exam 2

May 7, 2021

1 Long Questions

Q17 You have a company, and there are n projects and n workers. A project can only be assigned to one worker. Each worker has capacity to work on either one or two projects. Each project has a subset of worker as their possible assignment choice. We also need to make sure that there is at least one project assigned to each worker.

Give a polynomial time algorithm that determines whether a feasible assignment of project to workers is possible that meets all the requirements above. If there is a feasible assignment, describe how your solution can identify which project is assigned to which worker.

Solution:

Method 1:

You can consider this problem as a Bipartite matching problem:

Construct a flow network as follows:

- Create source s and sink t; for each project i create a vertex p_i, for each worker j create a vertex w_j;

- Connect s to all p_i with an edge, make the edge capacity equal to 1; connect all w_j to t with an edge, and the edge capacity is 1 .

- If worker j is one of project i's possible assignment choices, add an edge from p_i to w_j with edge capacity 1.

(10 points)

Algorithm:

If we can find a max-flow, $f = n$, then there is a feasible assignment. Since for each edge, the edge capacity is 1, this algorithm can be done in polynomial time. If there is one, for each p_i and w_j, check whether the flow from p_i to w_j is 1, if yes, assign project i to worker j.

(5 points)

Method 2:

You can use circulation to solve this problem.

Construct a flow network as follows:

- For each project i create a vertex p_i, for each worker j create a vertex w_j;

- If worker j is one of project i's possible assignment choices, add an edge from p_i to w_j with lower bound and upper bound 1.

- Create a super source s, connect s to all p_i with an edge of lower bound and upper bound 1.

- Create a super sink t, connect all w_j to t with an edge of lower bound 1 and upper bound 2.

- Connect t to s with an edge of lower and upper bound n.

(10 points)

Algorithm:

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 p_i and w_j, check whether the flow from p_i to w_j is 1, if yes, assign project i to worker j.

(5 points)

Grading:

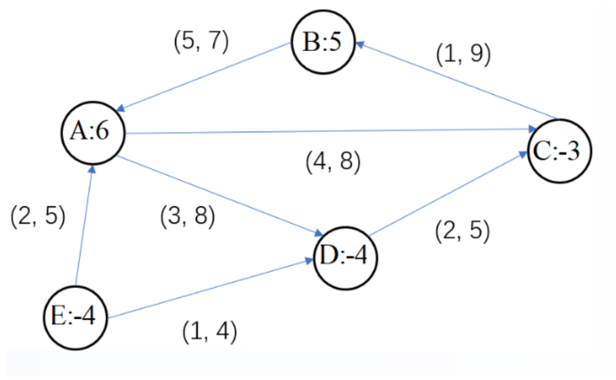
In this question, we need to make sure that there is at least one project assigned to each worker. If you remove all the redundant information, you will find it's a Bipartite matching problem.

- Construct the correct flow network: vertices, edges, and edges capacity (with lower bound and upper bound): 10 points; If the edge capacity equals to 2 (without lower bound), deduct 4 points.

- If the algorithm includes finding an integral circulation of the graph, or finding the max-flow which equals to n: 3 points.

- Describe how the solution can identify which project is assigned to which worker (check whether the flow from p_i to w_j is 1): 2 points

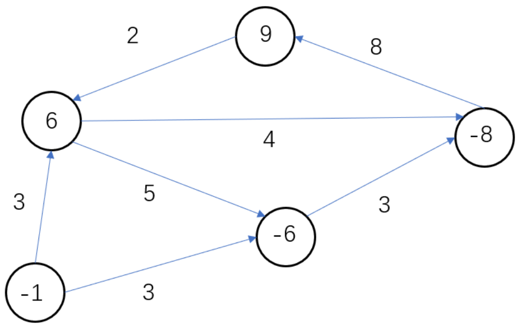
Q18 In the network below, the demand values are shown on vertices (supply values are negative). Lower bounds on flow and edge capacities are shown as (lower bound, capacity) for each edge. Determine if there is a feasible circulation in this graph. Please complete the following steps.



(1) Remove the lower bounds on each edge. Write down the new demands on each vertex A, B, C, D, E,in this order.

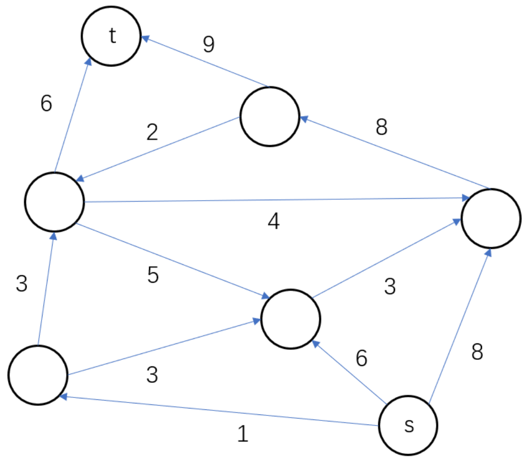
A: 6, B: 9, C: -8, D: -6, E: -1. (5 points)

(2) Solve the circulation problem without lower bounds. Write down the max-flow value.



The max-flow value is: 9. (5 points)

(3) Is there is a feasible circulation in the original graph? Explain your answer.



Check if there is a $s-t$ flow $|f|= 15$. The max-flow of this flow network is 9, so there is no feasible circulation. (5 points)

Grading:

(1) If all values are correct: 5 points, otherwise, deduct 1 point for an incorrect value.

(2) If the max-flow is correct: 5 points, otherwise, 0 points.

(3) If the answer shows that there is no feasible circulation and the explanation is reasonable: 5 points.

Q22 In a country, there are N cities, and there are some undirected roads between them. For every city there is an integer value (may be positive, negative, or zero) on it. You want to know, if there exists a cycle (the cycle cannot visit a city or a road twice), and the sum of values of the cities on the cycle is equal to 0. Note, a single vertex is not a cycle. Prove this problem is NP-Complete. Use a reduction from the Hamiltonian cycle problem. Complete the following five steps.

(1) Show that the problem belongs to NP (2 points).

The solution of the problem can be easily verified in polynomial time, just check if the sum of the values on the cycle is equal to 0, and the cycle doesn't visit a city or a road twice. Thus it is in NP.

(2) Show a polynomial time construction using a reduction from Hamiltonian cycle (6 points).

Given a Hamiltonian cycle problem. It asks if the graph exists a cycle go through all the vertices of the graph and doesn't visit a vertex twice. Let the number of vertices be N. Then we construct an instance of the zero-cycle problem (our problem), as, we set the value of one vertex to be $N-1$, and the value of any other vertices ($N-1$ vertices) to be -1.

(3) Write down the claim that the Hamiltonian cycle problem is polynomially reducible to the original problem. (2 points)

The given graph G has a Hamiltonian cycle if and only if the there is a cycle in the zero-cycle problem's graph G' and the sum of values of the cities on the cycle is equal to 0.

(4) Prove the claim in the direction from the Hamiltonian cycle problem to the reduced problem. (3 points)

If we have a solution of the Hamiltonian cycle problem, then the total value of the cycle will be $(N-1)+(-1)*(N-1)=0$, so the zero-cycle problem can also find a solution.

(5) Prove the claim in the direction from the reduced problem to the Hamiltonian cycle problem. (3 points)

If we have a solution of the zero-cycle problem, we want to prove that it's indeed a solution of the Hamiltonian cycle problem, i.e. we want to prove it will visit all vertices. If the vertex doesn't visit the vertex that has value $N-1$, then all the value on the cycle will be -1, so the total value will be negative, will not be 0, so it's impossible. So the cycle must visit the vertex that has value $N-1$. Then, in order to make the total value to be 0, it must visit all the rest of the vertices ($N-1$ vertices), since the value of them are all -1. So it will visit all the vertices. So it is a solution of the Hamiltonian cycle problem.

Grading: For (2), there may have various value assignment plans for the vertices. If the plan can make sure that, any cycle that sums up to 0 must pass all the vertices, it's correct.

Q27 The edge-coloring problem is to color the edges of a graph with the fewest number of colors in such a way any two edges that share a vertex have different colors . You are given the algorithm that colors a graph with at most $d+1$ colors if the graph has a vertex with maximum degree d. You do not need to know how the algorithm works. Prove that this algorithm is a 2-approximation to the edge coloring problem. You may assume that $d \geq 1$.

Maximum degree of the graph is d. This implies there exists at least one vertex (say v_1) connected to d other vertices. All the d edges connected to v_1 would have to be of different colors to satisfy the edge coloring. Therefore, the lower bound on the optimal solution is $OPT \geq d$.

7 points for making the above claim.

Our algorithm returns a coloring of at-most $d+1$ colors. Hence, the upper bound of the algorithm is $ALG \leq d+1$.

The approximation ratio ρ would be the upper-bound of algorithm divided by the lower-bound of the optimal algorithm which gives:

$$\rho = \frac{d+1}{d}$$

5 points for arriving at above equation.

Since, $d \geq 1$ and ρ gets largest value with $d = 1$, we have:

$$\rho = 1 + \frac{1}{d} \leq 1 + 1 = 2$$

That is, in the worst case we have a 2-approximation ration. Hence, it is a 2-approximation algorithm.

3 points for arriving at the final ratio of 2

Detailed Rubric:

(a) If only lower-bound is correct (i.e. $OPT \geq d$) but approximation ratio computation is not, award 7 points.

(b) If only upper-bound is correctly used, but lower-bound is incorrect, give 5 points. Since approximation ratio cannot be computed, that part receives 0/3.

Q21 A clique in a graph $G = (V, E)$ is a subset of nodes $C \subseteq V$ s.t. each pair of nodes in C is adjecent, i.e., $\forall u, v \in C, (u, v) \in E$. We are interested in computing the largest clique (i.e., a clique with max. no. of nodes) in a given graph. Write an integer linear program that computes this.

Let x_u be an integer variable that gets a value 1 if u is included in the clique, and 0 otherwise. The ILP formulation is as follows:

$$\begin{array}{ll} \text{max:} & \sum_{u \in V} x_u \\ \text{subject to:} & x_u + x_v \leq 1 \quad \forall\ (u, v) \notin E \\ & x_u \in \{0, 1\} \quad \forall\ u \in V \end{array}$$

Alternatively, you could associate the integer variables with the exclusion of the variables instead of inclusion, i.e., each x_u is 1 if u is excluded, and 0 if included. Then, the ILP is

$$\begin{array}{ll} \text{min:} & \sum_{u \in V} x_u \\ \text{subject to:} & x_u + x_v \geq 1 \quad \forall\ (u, v) \notin E \\ & x_u \in \{0, 1\} \quad \forall\ u \in V \end{array}$$

Grading:

- **2.5 points for correctly defining the variables and mentioning they're binary.**
- **5 points for the objective.**
- **7.5 points for the constraint.**
- **Note that having variables/constraints in addition to the ones required, often makes the solution incorrect.**
- **Partial points for any cases not confirming to the standard solution will be handled on a per-case basis.**

Question 1 - True/False (20 pts)

(In-person exams have first 10 problems, Online exams have a randomized set)

1- **[TRUE/[✓ FALSE]]** A graph G has a unique MST if and only if all of G's edges have different weights.

2- **[TRUE/[✓ FALSE]]** The runtime complexity of merge sort can be improved asymptotically by recursively splitting an array into three parts (rather than into two parts)

3- **[[✓ TRUE]/FALSE]** In the interval scheduling problem, if all intervals are of equal size, a greedy algorithm based on the earliest start time will always select the maximum number of compatible intervals.

4- **[TRUE/[✓ FALSE]]** Amortized analysis is used to determine the average runtime complexity of an algorithm.

5- **[[✓ TRUE]/FALSE]** Function $f(n) = 5n^2 4^n + 6n^4 3^n$ is $O(n^2 4^n)$.

6- **[[✓ TRUE]/FALSE]** If an operation takes O(1) time in the worst case, it is possible for it to take O(log n) amortized time.

7- **[TRUE/[✓ FALSE]]** If you consider every man and woman to be a node and the matchings between them to be represented by edges, the Gale-Shapley algorithm returns a connected graph

8- **[TRUE/[✓ FALSE]]** A Fibonacci Heap extract-min operation has a worst case run time of O(log n).

9- **[[✓ TRUE]/FALSE]** Consider a binary heap A whose elements have positive or negative key values. If we randomly square the key values for some elements, the heap property can be restored in A in linear time.

10- **[[✓ TRUE]/FALSE]** The order in which nodes in the set V-S are added to the set S can be the same for Prim's and Dijkstra's algorithms when running these two algorithms from the same starting point on a given graph.

11- **[[✓ TRUE]/FALSE]** In the stable matching problem discussed in class, suppose Luther prefers Ema to others, and Ema prefers Luther to others. Then Ema only has one valid partner.

12- **[TRUE/[✓ FALSE]]** Algorithm A has a worst case running time of Θ(n^3) and algorithm B has a worst case running time of Θ(n^2 logn). Then Algorithm A can never run faster than algorithm B on the same input set.

13- [TRUE/[✓ **FALSE**] **]** Bipartite graphs cannot have any cycles

14- [[✓ **TRUE**]**/FALSE]** In the stable matching problem, it is possible for a woman to end up with her highest ranked man when men are proposing.

15- [TRUE/[✓ **FALSE**] **]** If we double the number of nodes in a binomial heap the number of binomial trees in the heap will go up by one.

16- [[✓ **TRUE**]**/FALSE]** Consider items A, B, and C where items A and B are more similar to each other than items A and C. The K-clustering algorithm (discussed in lecture) involving n items that include A, B, and C could results in a clustering where A and C are in the same cluster, but B is in a separate cluster.

17- [[✓ **TRUE**]**/FALSE]** The first edge added by Kruskal's can be the last edge added by Prim's algorithm.

Question 2 - Divide and Conquer (15 pts)

Consider a full binary tree representing k generations of the Vjestica family tree (i.e. family tree with k levels). It so happens that the Vjestica family is generally very tall, and they have kept records of all its member's heights which has now sparked a researcher's interest in this data. The researcher wants to find out if there is anyone in this family tree who is taller than its parent (if the parent exists in the tree) AND his/her two children (if the children exist in the tree). In other words, the member we are looking for should be taller than anyone they are immediately related to in the family tree.

a) How many Vjestica's are there in this family tree? (2 pts)

$n = 2^k - 1$ (2pts for this answer)

b) Show that if heights are unique, we can always find such a family member. (5 pts)

Solution 1 (THE SIMPLEST)
The tallest Vjestica must be one as required since he/she is strictly taller than its parent (if the parent exists in the tree) AND his/her two children (if the children exist in the tree) - since the heights are unique.

Solution 2 (Traversal + stopping guarantee)
If the root is taller than the two children, we are done. Otherwise, pick the taller of the two children and see if this child satisfies the criteria.
Do this recursively (which ensures each node thus visited is taller than the parent). If we stop at an internal node, it must satisfy the criteria by having both children smaller, or if we reach a leaf node, then that satisfies the criteria due to having no children (and being taller than the parent as aforementioned).

Solution 3 (Induction)
Base case: property due to having no children or the parents.
Ind Hyp: For some n >= 1, suppose such a member always exists in a tree with n levels.
Ind step: For a tree T with n+1 levels, first check if root is a solution - i.e. if it's taller than both the children. If it's not, there's a taller child, say left child L. Consider the subtree T' rooted at L. By ind hyp, since T' has all distinct heights, it has a member M as desired (relative to T'). If $M \neq L$, then M has both children and parents and it satisfies the property in T as well. if M = L, then L satisfies the property in T' by being taller than the children and having no parent.

Solution 4:
Proof by contradiction. Assume there is no such family member.
So for leaf nodes at level k, they must be smaller than their parents in level k-1. And since nodes in level k-1 are not the members (with the desired property) despite being taller than the children, they must be smaller than their respective parents in level k-2… Continuing up to level

1 the root node must be larger than both the children, ie. root is the member we are looking for, a contradiction.

5pts for any correct proof
~3 pts for correct proof but missing some relatively unimportant steps
0pts for wrong proof.
* Incomplete & Insufficient Proof are not count as correct proof.

c) Assuming that the heights are unique, design a divide and conquer solution to find such a member in no more than O(k) time. (5 pts)

You can assume full binary tree is saved into an array for easy notation.

```
Locally tallest (i) \\Assumes (ensures by definition) that i is taller than its parent IF IT EXISTS

        \\Check to see if we found what we are looking for
        if children exist and person at node i is taller than them, return i

        Else if no children exist, return i

        Else if child at 2i (left child) is taller than child at 2i+1 (right child) then return Locally
tallest (2i)

        Else return Locally tallest (2i+1)
```

Notes: If both children are taller, you can choose either child, no need to choose the taller one of the two children. The fastest approach is you compare with the left child first, if the left child is taller, just go with the left child. If not, compare the node with the right child, if the right is smaller, then you return the node. If the right child is taller, you go with the right child.

So each time in the iteration or recursion, If node is not left node, you do:

```
if node.left > node.val:

        func(node.left)

else if node.right > node.val:

        fucn(node.right)

else:

        return node
```

<u>Call Locally tallest (root)</u>

if the algo does not find the answer: 0pt,

if the algo finds the answer but slow and not using D&C on Tree Structure: 1pt,

if the algo finds the answer but slow and using D&Q on Tree Structure: 2pt, if the algo finds the answer and O(k) and not using D&Q on Tree Structure: 2pt,

if using pseudo code have minor errors using O(k) and D&Q on Tree Structure: 2-4 pt,

if all correct: +5pt,

else: return 0pts;

d) Show your complexity analysis using the Master Method. (3 pts)

F(n) = θ (1)
T(n) = T(n/2) + f(n)
n^(log base b of a) = n^(log base 2 of 1) = n^0 = θ (1)
Case 2 - > T(n) = θ (log n) = θ (k)

if C is not D&C: 0pts.

if C) is D&C:

-1pts for wrong f(n)

-1 pts for wrong T(n) (for your answer in part c) or not clearly shown

-1pts for wrong log b base a,

-1pts for wrong case(chose case 2) or give the right comparison (n^{log_b a} = f(n))

-1pts missing master method,

-1pts for wrong or missing answers.

Question 3 - Greedy (15 pts)

There are n distinct jobs, labeled J_1, J_2, ...,J_n, which can be performed completely independently of one another. Each job consists of two stages: first it needs to be preprocessed on the supercomputer, and then it needs to be finished on one of the PCs.
Let's say that job J_i needs p_i seconds of time on the supercomputer, followed by f_i seconds of time on a PC.

Since there are at least n PCs available on the premises, the finishing of the jobs can be performed on PCs at the same time. However, the supercomputer can only work on a single job a time without any interruption. For every job, as soon as the preprocessing is done on the supercomputer, it can be handed off to a PC for finishing.
Let's say that a schedule is an ordering of the jobs for the supercomputer, and the completion time of the schedule is the earliest time at which all jobs have finished processing on the PCs.

a) Design an algorithm that finds a schedule with as small a completion time as possible. Your algorithm should not take more than O(n^2) time in the worst case. (6 pts)

Sort jobs in order of decreasing f_i

b) Prove that your algorithm produces an optimal solution. (7 pts)

Proof is identical to discussion problem with athletes (swimming, then biking+running).

We define an inversion here as a job i with a higher f_i being scheduled after job j with lower f_j.
We can then show that if we take an optimal algorithm which has inversions, then we can remove the said inversions without increasing the overall completion time for all the jobs, until the optimal solution turns into our solution of scheduling jobs in decreasing order of f_i.

We can remove inversions without increasing the completion time of the schedule:

1) If there is an inversion between two non-adjacent jobs i & j, we can always find two adjacent jobs somewhere between i & j such that they have an inversion between them. Therefore, let us assume the case where we have two adjacent jobs which are an inversion. Let job i with higher f_i be scheduled immediately after job j with f_j as its regular PC processing time.
2) We remove the inversion by scheduling job i before job j. Now doing this does not increase the completion time for job i as not only does the supercomputer portion of job i finish faster, but consequently so does its PC processing finish time.
3) Now that job j is scheduled *after* job i, the total time completion for the schedule until job j would change from the initial time of $p_i + p_j + f_i$ to max($p_i + p_j + f_j$, $p_i + f_i$). We know that $f_i \geq f_j$, therefore our total completion time will never increase with the resolution of this inversion.

4) We can then remove every inversion in the optimal algorithm without increasing the total completion time of the schedule.
5) Once all inversions have been removed, it is easy to see that the optimal solution after removing inversions will be the same as our greedy solution of decreasing f_i times

-2 points for not defining what the inversions in this case are.

-2 points for not generalizing the proof

-1 points for not showing that the algorithm is now the same as our greedy algorithm

c) Analyze the complexity of your solution (2 pts)

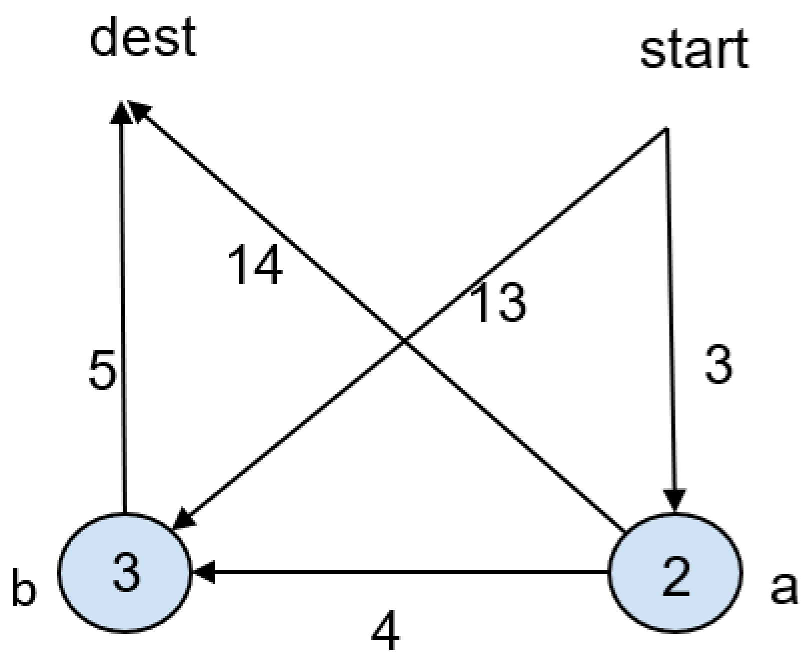
Only requires a Sort which takes θ (nlog n)

- If you analyze your algorithm (which may be incorrect) incorrectly, all points will be deducted. Conversely, if you correctly analyze your incorrect algorithm, you will be awarded full points for this sub-question.

Question 4 - Shortest Path (12 pts)

You are in the city called $start$, and you want to drive your new electric car to the city called $dest$. There are many possible paths to take for this trip, but the good news is that if you take any of these paths, the recharging stations along the way are designed such that if you stop at every recharging station on your way, you will always have enough charge to reach $dest$. You can consider the map as a **directed weighted** graph with recharging stations at some of its nodes. The edge weights C_e represent the time it takes to drive through edge e. You are currently in the city $start$ with a full charge and you want to optimize your trip to get to $dest$ as fast as possible. What you also need to consider is that some charging stations are faster than others. In fact, for each charging station i, we know that we need to wait a fixed amount of time C_i to recharge. Give an efficient algorithm to find out the fastest path from city $start$ to city $dest$ (minimum total time) assuming that you must stop and recharge at every charging station on that path.

Example: Assume charging in cities a and b takes 2 and 3 units of time respectively.



The optimal path is start -> a -> b -> dest. The total time is 17:

3 hrs driving between start to a

2 hours charging at a

4 hours driving from a to b

3 hours charging at b

5 hours driving from b to dest

Solution: Create a new graph G' where we split each node *u* with a charging station into two nodes u_1 to u_2 with an edge going from u_1 to u_2. connect all incoming edges into u to u_1. All outgoing edges from u will go out of u_2. Assign edge costs to new edges equal to the charging time at that charging station. Run Dijkstra's to find shortest path.

graph modification (8 pt):

- solution I: split charging station node into two nodes, one receiving incoming edges and one sending outgoing edges (*as described above*)
- solution II: adding the charging node weight (charging time) to the cost (traveling time) of all coming or all outgoing edges weights
- solution III: run Dijkstras and add each node value into account during each step

no modification to graph (-8 points)

modification not specified (-6 points)

completely incorrect modification (-6 points)

other errors such as not accounting for edge weights and incomplete modification (-4 points)

add node values to both incoming and outgoing edges or not clear what is added to what(-3 points)

path finding (4 pt):

- Use some shortest path finding algorithm on weighted graphs (like Dijkstras)

modified version of Dijkstras has errors (-2 points)

incorrect path finding or exponential time (-4 points)

Question 5 - Heaps (12 pts)

The United States Commission of Southern California Universities (USCSCU) is researching the impact of class rank on student performance. For this research, they want to find a list of students ordered by GPA containing every student in California. However, each school only has an ordered list of its own students by GPA and the commission needs an algorithm to combine all the lists. There are a few hundred colleges of interest, but each college can have thousands of students, and the USCSCU is terribly underfunded so the only computer they have on hand is an old vacuum tube computer that can do about a thousand operations per second. They are also on a deadline to produce a report so every second counts. Find the fastest algorithm for yielding the combined list and give its runtime in terms of the total number of students (m) and the number of colleges (n).

```
Use a minheap H of size n.
Insert the first elements of each sorted array into the heap. The objects entered into the heap
will consist of the pair (GPA, college ID) with GPA as the key value.
Set pointers into all n arrays to the second element of the array CP(j) = 2 for j=1 to n
Loop over all students (i= 1 to m)
        S = Extract_min(H)
        CombinedSort (i) = S.GPA
        j = S. college_ID
        Insert element at CP(j) from college j into the heap
        Increment CP(j)
endloop
```

Build min heap (4 points)
Extract the min element (2 points)
Keep pointer to insert the next element in array for extracted element. (2 points)
Recursively/ In Loop do it for all the students (2 points)
Runtime complexity - O(mlog n) (2 points)

SOLUTION 2: Divide & Conquer works too:
https://leetcode.com/problems/merge-k-sorted-lists/solution/#approach-5-merge-with-divide-and-conquer

Question 6 - Gale Shapley (10 pts)

Consider the Gale-Shapley algorithm operating on n men and n women, with women proposing.

a) What is the maximum number of times a woman may be rejected (with respect to the problem size n)? Give an example where this can happen. (5 pts)

Maximum number of rejections can be n-1 since nobody will be rejected by all n men.
correct n-1 (2pts)
correct explanation (3pts)

Sample Example: Some woman has the least preference for all men.

b) Now, consider the following modification to the G-S algorithm: At each iteration, we always pick the free woman with the highest average preference among men, i.e. the most "popular" remaining woman (when taking an average across all men's preference lists).
Prove or disprove: this will help reduce the number of rejections for some women. (5 pts)

Since all women will end up with their best valid partners regardless of the order in which we pick women from the list of free women, the number of rejections will not change.

no change (2pts)

Two acceptable interpretations for the claim due to slight ambiguity:

"sometimes change for some women" -> NO; proof with reasoning

"Always change for some women" -> NO; proof with a counterexample

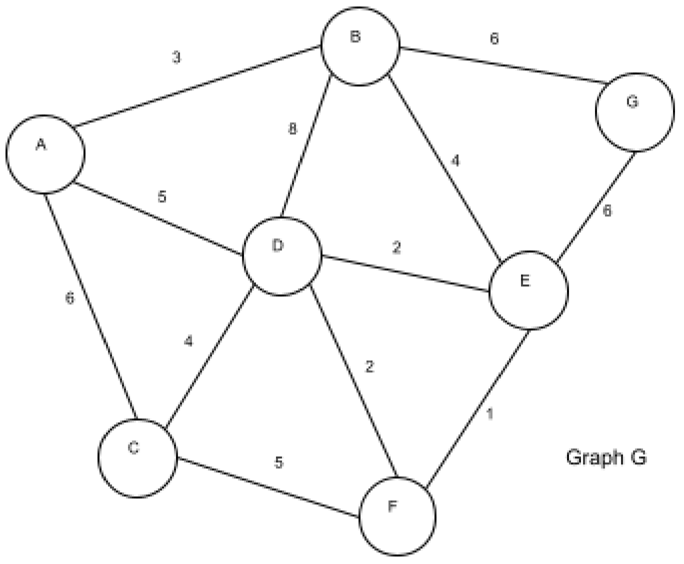
valid proof (3pts)

reasoning: All women end up with their best valid partners regardless of the order.

counterexample: any example with no change.

Question 7 - Multiple Choice questions (16 pts)

For the next 3 questions consider the following graph G.



1. In graph G, if we use Kruskal's Algorithm to find the MST, what is the third edge added to the solution? Select all correct answers (4 pts)

   a. E-F

   b. D-E

   O c. A-B

   d. C-F

   e. D-F

2. In graph G, if we use Prim's Algorithm to find MST starting at A, what is the second edge added to the solution? (4 pts)

   a. B-G

   O b. B-E

   c. D-E

   d. A-D

   e. E-F

3. What is the cost of the MST in the Graph? (4 pts)

a. 18

b. 19

O c. 20

d. 21

e. 22

4. Which of the following Asymptotic notation is O(n^2)? Select all apply. (4 pts)

O a. O(7 log n^3)

b. O(n log n^n)

c. O(3^n)

O d. O(log 10^n)

O e. O(5 n^2 + n log n)

O f. O(2000)

CS570 Fall 2021: Analysis of Algorithms Exam II

	Points		Points
Problem 1	20	Problem 4	20
Problem 2	20	Problem 5	20
Problem 3	20		
	Total	**100**	

Instructions:

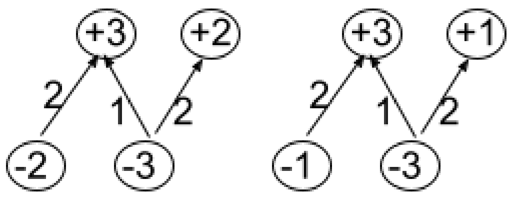
1. This is a 2-hr exam. Open book and notes and internet access. But no internet communications through social media, chat, or any other form is allowed.
2. If a description to an algorithm or a proof is required, please limit your description or proof to within 150 words, preferably not exceeding the space allotted for that question.
3. No space other than the pages in the exam booklet will be scanned for grading.
4. If you require an additional page for a question, you can use the extra page provided within this booklet. However please indicate clearly that you are continuing the solution on the additional page.
5. Do not detach any sheets from the booklet. Detached sheets will not be scanned.
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.
8. This exam is printed double sided. Check and use the back of each page.

True/False Questions

Network flow (2 pts)

1. We are guaranteed to find max flow using Ford-Fulkerson as long as our edge capacities are all positive.
   False. Not guaranteed to terminate or converge to max flow for arbitrary non-integral capacities.
2. If a flow network G contains an edge that goes directly from source S to sink T, then this edge will always be saturated due to any max flow in G.
   True. This edge is a part of every min-cut, thus must be saturated at max flow.
3. The capacity of an s-t cut in a Flow Network G could be greater than the value of max flow in G.
   True. There could always be a cut of size greater than that of a min-cut (= max flow value).
4. In a Circulation Network with no lower bound constraints and a feasible circulation F, if we decrease the demand at a sink node by one unit and decrease the supply at a source node by one, we will still have a feasible circulation in the resulting Circulation Network.

   False. Consider the network shown on the side with capacities and demands, initially as on the left, and the changed ones as on the right. Initially there's a feasible circulation (obtained by saturating all the edges). After the changed demands, there is no feasible circulation.

5. If f and f' are two feasible s-t flows in a Flow Network such that $|f'| > |f|$, then there is always a feasible s-t flow in the network with value $|f'| - |f|$
   True. If f is a feasible flow, there is always a feasible flow of any lower value (this is a simple flow network, not a circulation with lower bounds/demands).

Dynamic Programming (2 pts)

1. The memory space required for any dynamic programming algorithm with n^2 unique subproblems is $\Omega(n^2)$
   False. We have seen examples where only very few subproblems (and not all) need to be stored at any point during the DP computation of all the sub-problems. In such cases, the memory needed can be much less.
2. The 0-1 knapsack problem can be solved using dynamic programming in O(N * V) runtime, where N is the number of items and V is the **sum of the weights of all items**
   True.
3. In a 0/1 knapsack problem with n items, suppose the value of the optimal solution for all unique subproblems has been found. If one adds a new item to the list now, one must re-compute all values of the optimal solutions for all unique subproblems in order to find the value of the optimal solution for the n+1 items.
   False. We can simply compute Opt(W, n+1) = Opt(W, n) + Opt(W - W_{n+1} , n) with problems on the right already computed.

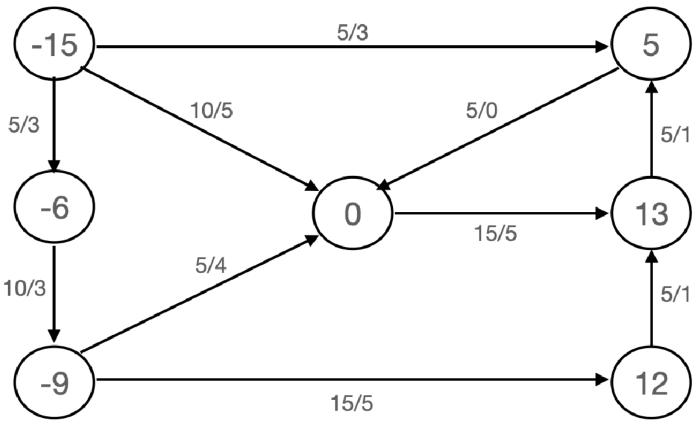
Dynamic Programming (4 pts)

4. Recall the solution to the 0/1 knapsack problem presented in lecture. If our objective were to only find the **value of the optimal solution** (and not the actual set of items in the optimal solution), O(n) memory space will be sufficient to solve this problem.
   False. To compute all the values Opt(w,i) for all w, for a given i, we need all the values Opt(w, i-1) for all w. Thus, we need to store O(W) memory, which is not necessarily O(n).

Network Flow Problem 1

(20 pts)

In the network G below, the demand values are shown on vertices (supply value if negative). Lower bounds on flow and edge capacities are shown as (capacity/lower bound) for each edge. Determine if there is a feasible circulation in this graph. You need to show all your steps.

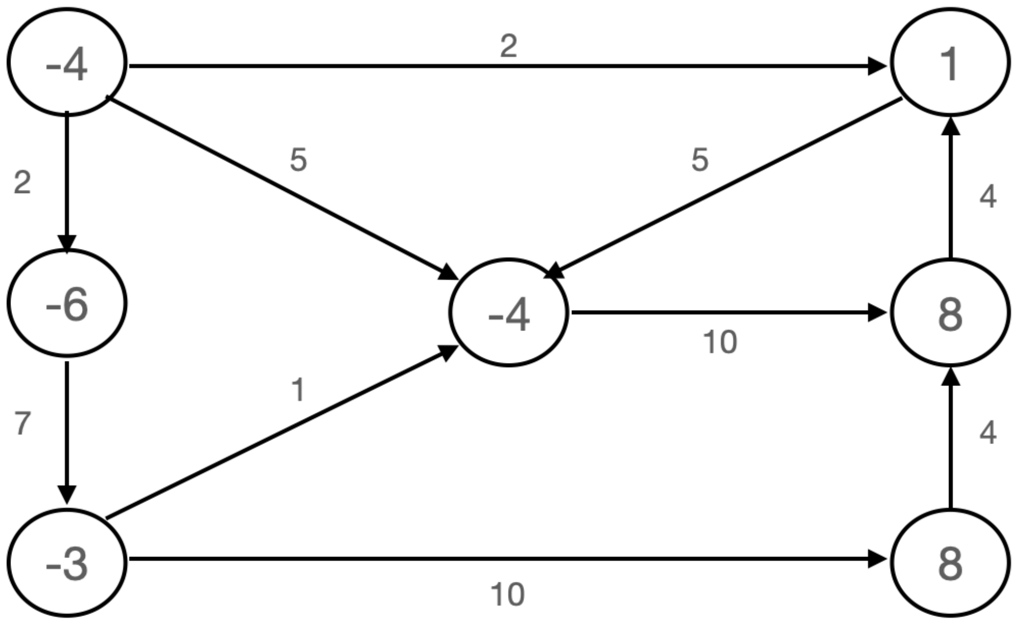


a. Reduce the Feasible Circulation with Lower Bounds problem to a Feasible Circulation problem without lower bounds. (8 pts)

Draw G'

If demand is correct, get 4 pts

If flow on edges are correct, get 4 pts



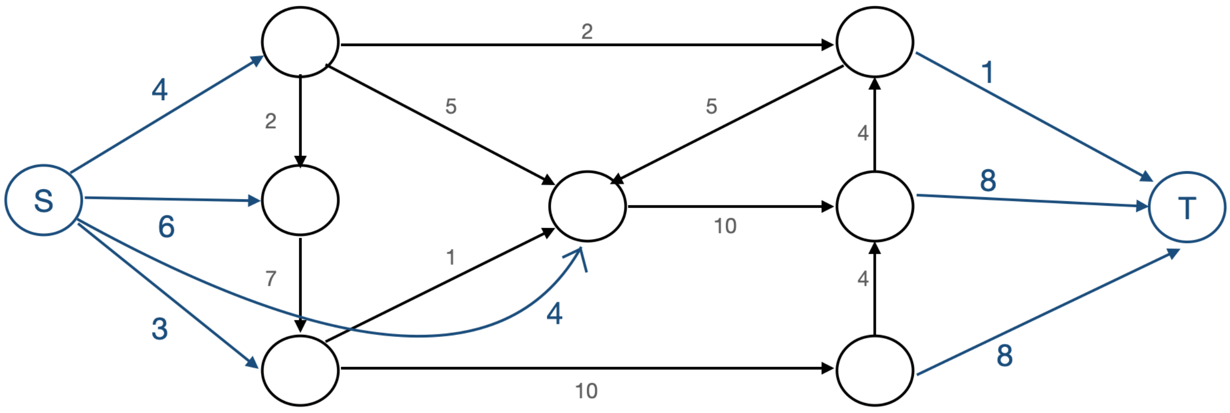
b. Reduce the Feasible Circulation problem obtained in *part a* to a Maximum Flow problem in a Flow Network.(8 pts)

Add S and T (2 pts)

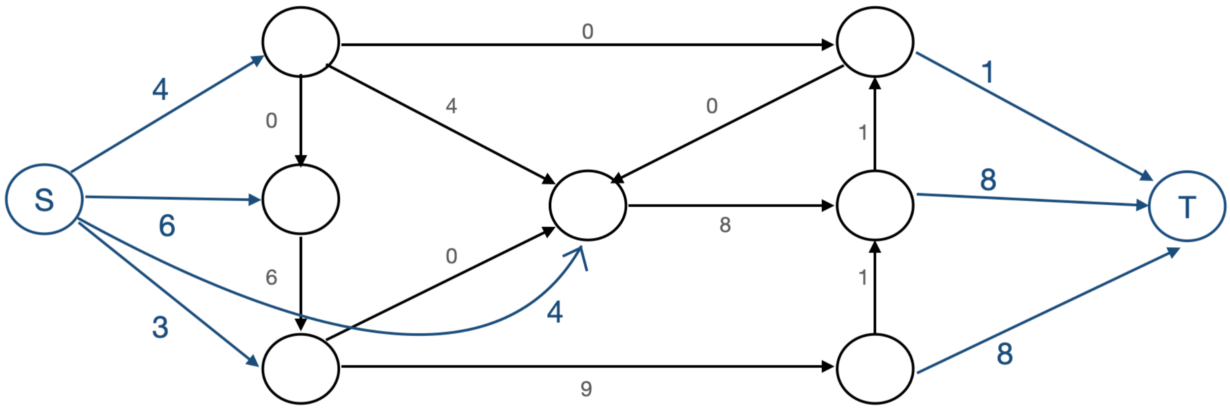
Connect S with correct nodes (2 pts)

Connect T with correct nodes (2 pts)

Give correct capacities to edges (2pts)



c. Using the solution to the resulting Max Flow problem explain whether there is a Feasible Circulation in G. (4 pts)

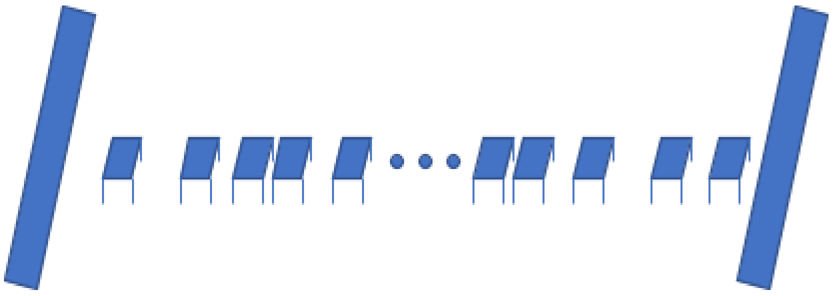


This is the max flow in Flow network. (2 pts)

Since the flow saturates all edges from S, so there is feasible circulation in G. (2 pts)

Network Flow Problem 2 (20 pts)

A number of people have gotten themselves involved in a very dangerous game called the octopus game. At this stage of the game, they need to pass a river. The river is 100 feet wide, but contestants can only jump k feet at most, where k is obviously less than 100. To help contestants cross the bridge there are platforms placed along a straight line at integer distances from one end of the river to the other end at various distances where the distance between any two consecutive platform is less than k. But the problem is that each platform can only be used once. If another contestant tries to use it for the second time it will break.



a) Design a network flow based solution to determine the maximum number of contestants that can safely cross this river and live to play in the next stage of the game. (14 pts)

Solution 3 a)

The construction of the network can be done as follows:

1. Create a source 'S' and sink node 'T' and create nodes that would represent the platforms (lets assume we call these nodes {A…….N})
2. For every platform node {A….N} create a dummy node {A',B'.....N'} respectively. Connect all the platform nodes {A…….N} to the corresponding dummy nodes {A'…….N'} so that the edge A -> A' represents platform A and so on. Set the capacity to **1** for all these edges. This part is the most crucial in the construction to make sure that a platform can not be used by more than one contestants.
3. Connect the source 'S' to the platform nodes that are at max **k** feet distance from it. The capacity of any such edge could be set to anything >= 1. This is to make sure that all the inflow to the nodes is at platform node {A,B...N}.
4. Connect the dummy nodes {A',B'...N'} to **<u>all</u>** the platform nodes (or the sink) that are at max **k** feet distance from it. HOWEVER, it suffices to add such edges in only the forward direction. This also makes sure the outflow at each platform is from the dummy nodes {A',B',....N'}

Final answer: Once the network is constructed, run a max flow algorithm on it (say FF) and the max flow value of the network obtained will be the maximum number of contestants that can cross the river.

Rubrics 3a:

- -2 points for each incorrect/missing capacity/node/edge
- -2 pts: if the final answer is missing
- AT MOST half the points if the fundamental structure of the network is wrong (partial credit subjective to how close the proposed solution is)

b) Prove that your solution is correct (6 pts)

Claim : The max flow of the network will give the number of contestants that can get across the river

Forward Claim : If there is a flow of value V, we can send V people across

Proof : Since k<100, any S-T path must pass via some platform (say p), and thereby has a bottleneck of 1 due to the edge p - > p'. Thus any s-t path can have a flow of at most 1. Thus, if the flow value is V, there will be V paths each having a flow of 1. Further, these cannot share any edge p -> p' due to its capacity of 1. Thus, we can assign each such path to a contestant and none of them will use the same platform, thus allowing us to send V people safely.

Backward Claim : If we can send V people across, we can have a flow of value V

Proof : For 1 person crossing the river, we get a path going from s to t such that the person jumps at most k feet at a time. These V paths must be node-disjoint, since no platforms can be repeated. If we send a flow of 1 unit down each corresponding path in the network, we won't exceed any capacity constraints and this flow will have a value of V.

Rubrics 3b:

- -2 pts: If the claim is incorrect
- -2 pts: If forward claim is not satisfactory/incorrect.
- -2 pts: If backward claim is not satisfactory/incorrect.

Dynamic Programming Problem 1 (20 pts)

Tommy has just joined the boy scouts and he's eager to earn some badges. He's got summer holidays coming up which will last for *N* days. Each day, he can earn points by doing any one activity among *hiking*, *swimming*, or *camping*. The camp instructor has posted the amount of points each activity can earn you on each day. For example, on the *ith* day, camping is worth c*[i]* points, hiking h[i] points and swimming s[i] points. However, Tommy gets bored doing the same activity really easily. He cannot do the same activity two or more days in a row.
Based on this information, devise a Dynamic Programming algorithm to maximize the number of points Tommy can earn. You may assume that all point values are positive, and remember he can only do one activity per day.

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

$OPT_{i,j}$ = the maximum number of points Tommy can earn from the start until the i^{th} day, given he does activity j on the i^{th} day.

- -2 points if it is mentioned: "points earned **ON** the ith day" instead of **until** the ith day.
- -4 points for incorrectly defined subproblems

OR

$OPT_{i,j}$ = the maximum number of points Tommy can earn from the i^{th} day, until the end given he does activity j on the i^{th} day.

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

$$OPT_{i,j} = max(OPT_{i-1,k} + act[j][i] \; where \; j \neq k, \; j, k \in \{c, \; h, \; s\}, act = \{c, \; h, \; s\})$$

In the second variant, i-1 is replaced by i+1 in each of the terms on the RHS

c) Using the recurrence formula in part b, write pseudocode using iteration to compute the maximum number of points Tommy can earn. (5 pts)
Make sure you specify

- base cases and their values (2 pts)
- where the final answer can be found (e.g. opt(n), or opt(0,n), etc.) (1 pt)

For variant 1:

```
function POINTS_EARNED(N, c, h, s):
    act = [c, h, s]
    OPT = [[0, 0, 0] N times]
    OPT[0][0] = c[0], OPT[0][1] = h[0], OPT[0][2] = s[0]
    for i in 1, 2, ..., N - 1:
        for j in 0, 1, 2:
            for k in 0, 1, 2:
                if j != k:
                    OPT[i][j] = max(OPT[i][j], OPT[i - 1][k] +
act[j][i])
            END-FOR
        END-FOR
    END-FOR

    return max(OPT[N - 1])
```

-2 points for not specifying base cases/values, or for incorrect base cases/values
-1 point for each mistake in pseudocode
-1 point for incorrect return value or incorrect final answer

d) What is the complexity of your solution? (1 pt)
   Is this an efficient solution? (1 pt)

$O(N)$
Yes.

1 point for correct runtime complexity analysis of **YOUR** solution. 0 points if analysis is incorrect.
0 points for saying it is not efficient.

Dynamic Programming Problem 2 (20 pts)

Assume a truck with capacity W is loading. There are n package types with different weights, i.e. [w1, w2, … wn], and all the weights are integers. Packages of the same type have the same weight. The company's rule requires that the truck needs to take packages with exactly weight W to maximize profit, but the workers like to save their energies for after work activities and want to load as few packages as possible. Assuming that there are infinite packages for each package type and that there will always be combinations of packages with a total weight of W, design an algorithm to find out the minimum number of packages the workers need to load.

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

OPT(w,k) = min packages needed to make up a capacity of exactly w, by considering only the first k package types
OR
Same except, considering package types k onwards up to n (add details below)

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

If $w >= w_k$
$OPT(w,k) = \min\{ 1+OPT(w- w_k , k) , OPT(w, k-1) \}$
Else
$OPT(w,k) = OPT(w, k-1)$

For variant 2, replace k-1 by k+1.

c) Using the recurrence formula in part b, write pseudocode using iteration to compute the minimum number of packages to meet the objective. (5 pts)
Make sure you specify

```
1.   Initialize for base cases (mentioned below for clarity). Initialize rest to infinity
2.   For w = 0 to W
         For k = 0 to n          //can switch the inner-outer loops
             If not base case: call recurrence
3.   Return ans (mentioned below for clarity)
```

- base cases and their values (2 pts)

OPT(w,0) = inf for all w >0
OPT(0,0) = 0

- where the final answer can be found (e.g. opt(n), or opt(0,n), etc.) (1 pt)
OPT(W,n)

d) What is the complexity of your solution? (1 pt)
Is this an efficient solution? (1 pt)

$O(nW)$ pseudo-polynomial run time
This is not an efficient solution

Solution 2:

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

   OPT(w,k) = min packages needed to make up a capacity of exactly w, by considering only the first k packages, AND selecting package k for sure.

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

   The recurrence captures all cases of j where j was the highest index used before the kth one:

   If $w >= w_k$
   $OPT(w,k) = 1 + \min_{j=0 \text{ to } k} OPT(w - w_k, j)$
   Else
   $OPT(w,k) = \infty$
   (No penalty for missing the latter case IF the base cases in ‘c)’ include $w < 0$.)

c) Using the recurrence formula in part b, write pseudocode using iteration to compute the minimum number of packages to meet the objective. (5 pts)

```
1. Initialize for base cases (mentioned below for clarity). Initialize rest to infinity
2. For w = 0 to W
       For k = 0 to n          //can switch the inner-outer loops
           If (not base case): //call recurrence
               For (j = 0 to k)
                   $OPT(w,k) = \min\{OPT(w,k), 1 + OPT(w - w_k, j)\}$
3. Return ans (mentioned below for clarity)
```

Make sure you specify
- base cases and their values (2 pts)
  $OPT(w,0) = \inf$ for all $w > 0$
  $OPT(0,0) = 0$
  $OPT(w,k) = \infty$ for $w < 0$ and all k. (Not required if the recurrence has the second case to ensure w never takes negative values)
- where the final answer can be found (e.g. opt(n), or opt(0,n), etc.) (1 pt)
Max_k OPT(W, k)

d) What is the complexity of your solution? (1 pt)
   Is this an efficient solution? (1 pt)

$O(n^2W)$ pseudo-polynomial run time
This is not an efficient solution

Solution3: (1-D) subproblems

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

   OPT(w) = min packages needed to make up a capacity of exactly w, (by considering all packages)

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

$OPT(w) = 1 + min\_\{j=1 \text{ to } n, w >= w_j\}\ OPT(w- w_j)$

(No penalty for missing the second condition under min_ IF the base cases in 'c)' include $w < 0$.)

c) Using the recurrence formula in part b, write pseudocode using iteration to compute the minimum number of packages to meet the objective. (5 pts)

```
1.   Initialize for base cases (mentioned below for clarity). Initialize rest to infinity
2.   For w = -W to W
             If (not base case): //call recurrence
                     For (j = 1 to n)
                             $OPT(w) = min\{OPT(w), 1 + OPT(w- w_j)\}$
3.   Return ans (mentioned below for clarity)
```

Make sure you specify

- base cases and their values (2 pts)

  $OPT(0) = 0$

  $OPT(w)$ = infinity for $w < 0$ (Not required if the recurrence has the second case to ensure w never takes negative values)

- where the final answer can be found (e.g. opt(n), or opt(0,n), etc.) (1 pt)

OPT(W)

CS570 Fall 2021: Analysis of Algorithms Exam II

	Points		Points
Problem 1	20	Problem 4	20
Problem 2	20	Problem 5	20
Problem 3	20		
	Total	**100**	

Instructions:

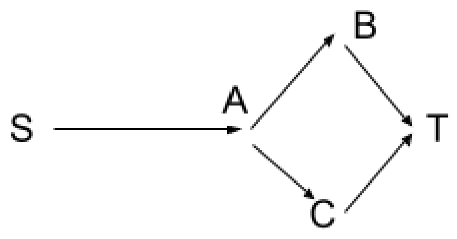
1. This is a 2-hr exam. Open book and notes and internet access. But no internet communications through social media, chat, or any other form is allowed.
2. If a description to an algorithm or a proof is required, please limit your description or proof to within 150 words, preferably not exceeding the space allotted for that question.
3. No space other than the pages in the exam booklet will be scanned for grading.
4. If you require an additional page for a question, you can use the extra page provided within this booklet. However please indicate clearly that you are continuing the solution on the additional page.
5. Do not detach any sheets from the booklet. Detached sheets will not be scanned.
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.
8. This exam is printed double sided. Check and use the back of each page.

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

[TRUE/FALSE] (2 pts)
If all edge capacities in a Flow Network are integer multiples of 5 then considering any max flow F in this network, flow over each edge due to F must be an integer multiple of 5.

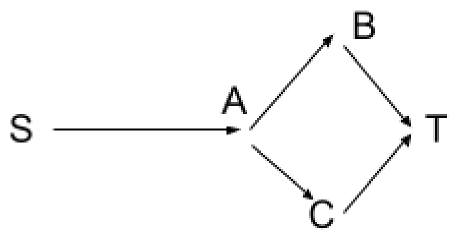
False. There will always exist a flow with flow over each edge being a multiple of 5, but this may not be true of ANY flow.

Counter - example as shown on the side. Let all capacities be 5. Let f(SA) = 5, f(AB) = f(AC) = 2, f(BT) = f(CT) = 3. Then, f is a max flow.



[TRUE/FALSE] (2 pts)
In a Flow Network G, if the capacity of any edge on a min cut is increased by 1 unit, then the value of max flow in that network will also increase by 1 unit.
Note: an edge on the min cut (A, B) refers to an edge that carries flow out of A (where the source S is) and into B (where the sink T is).

False. As counter - example, consider the network as shown on the side. Let all capacities be 5. The max flow here is of value 5. Consider the cut having AB, AC as the cut-edges. Increasing one of these to a capacity of 6 still keeps max flow at 5.

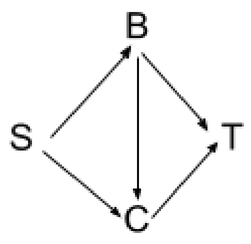


[TRUE/FALSE] (2 pts)
In a Flow Network with maximum flow value v(F)>0, and with more than one min cut, decreasing the capacity of an edge on any of the min cuts will reduce the value of max flow.
Note: an edge on the min cut (A, B) refers to an edge that carries flow out of A (where the source S is) and into B (where the sink T is).

True. Suppose the capacity of an edge in a min-cut (A,B) is decreased from c to c'. Initially, v(F) must have been c. Now, the max flow must be bounded by the new size of cut (A,B), i.e., c', thus, it does decrease.

[TRUE/FALSE] (2 pts)
Consider the Ford Fulkerson algorithm and the residual graph used at each iteration. If the Flow Network has no cycles (i.e. the network is a directed acyclic graph), we will not need to include backward edges in the residual graph to achieve max flow.

False. Consider the counter-example shown on the side. This is a DAG (no cycles). Let all capacities be 5. Suppose FF chooses path SBCT in the first step. If back-edges are not added, the residual graph will have T disconnected from S at this stage and terminate - at flow 5. Allowing the back edges however, we get path SCBT as step 2 making the actual max flow of 10.



[TRUE/FALSE] (2 pts)
Suppose an edge e is not saturated due to max flow f in a Flow Network. Then increasing e's capacity will not increase the max flow value of the network.
True. Since e is not saturated, e is not a cut-edge for any min-cut. Thus, increasing the capacity of e, does not affect the size of any min-cut and hence the max-flow value

remains the same.

[TRUE/FALSE] (2 pts)
The time complexity of any dynamic programming algorithm with n^2 unique subproblems is $\Omega(n^2)$.
True.
[TRUE/FALSE] (2 pts)
The Bellman-ford algorithm may not be able to find the shortest simple path in a graph with negative cost edges.

True.

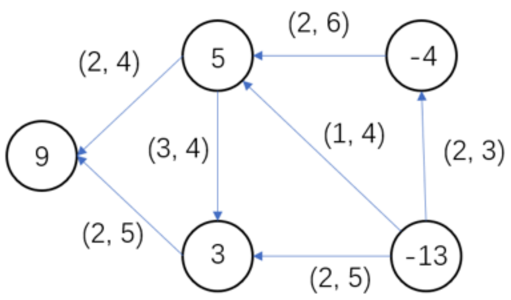
[TRUE/FALSE] (2 pts)
If the running time of an algorithm can be represented as a polynomial in terms of the number of bits in the input, then the algorithm is considered to be efficient.

True.

[TRUE/FALSE] (4 pts)
Recall the coin change problem from lecture where we are trying to pay amount m using the minimum number of coins. We presented two attempts to solve this problem in class, one based on the greedy approach and one using dynamic programming. If coin denominations are limited to 1, 3, and 4 these two approaches will result in the same number of coins to pay any amount m.

False. DP always gives the optimal solution. Greedy does not for $m = 6$.

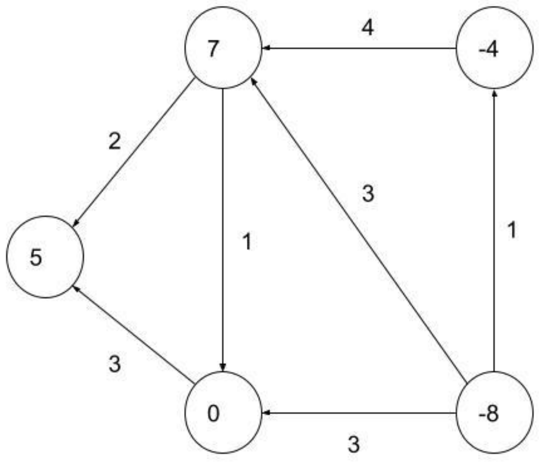
2) 20 pts
In the network G below, the demand values are shown on vertices (supply value if negative). Lower bounds on flow and edge capacities are shown as (lower bound, capacity) for each edge. Determine if there is a feasible circulation in this graph. You need to show all your steps.



a. Reduce the Feasible Circulation with Lower Bounds problem to a Feasible Circulation problem without lower bounds. (8 pts)

Solution:
Steps to follow :- 1) Assign flows to edges to satisfy lower bounds.
2) At each node v, L_v = (f_in - f_out), followed by D' = D - L_v
3) At each edge, cap = cap - LB

Resultant network:

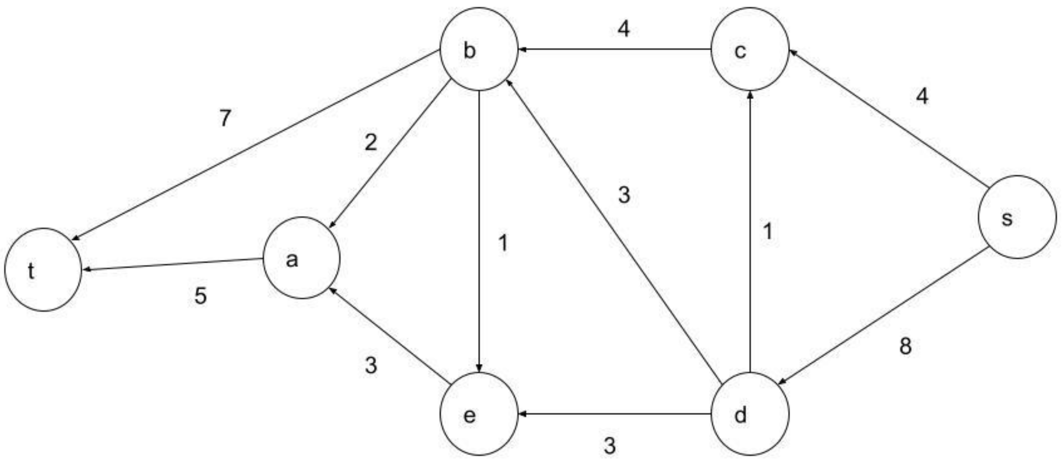


Rubrics:

(4 Pts) : If all edge values: (Upper_bound - Lower_bound) is correctly calculated
(4 Pts): If all Node Values (D' = D - (f_in - f_out)) is correctly calculated
(-2 Pts) if no steps explained.

b. Reduce the Feasible Circulation problem obtained in *part a* to a Maximum Flow problem in a Flow Network. (8 pts)

Solution:
The Max-Flow graph is as follows:



Rubrics:

Add S and T (2 pts)
Connect S with correct nodes (2 pts)
Connect T with correct nodes (2 pts)
Give correct capacities to edges (2pts)

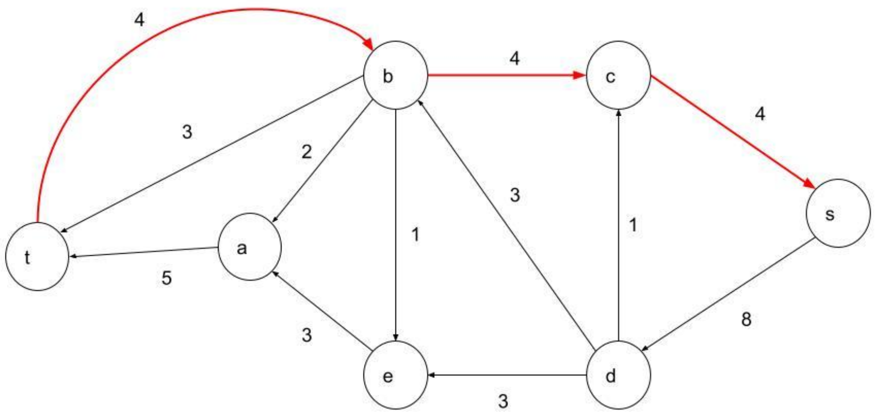
c. Using the solution to the resulting Max Flow problem explain whether there is a Feasible Circulation in G. (4 pts)

Solution:

<u>Candidate Solution 1:</u>

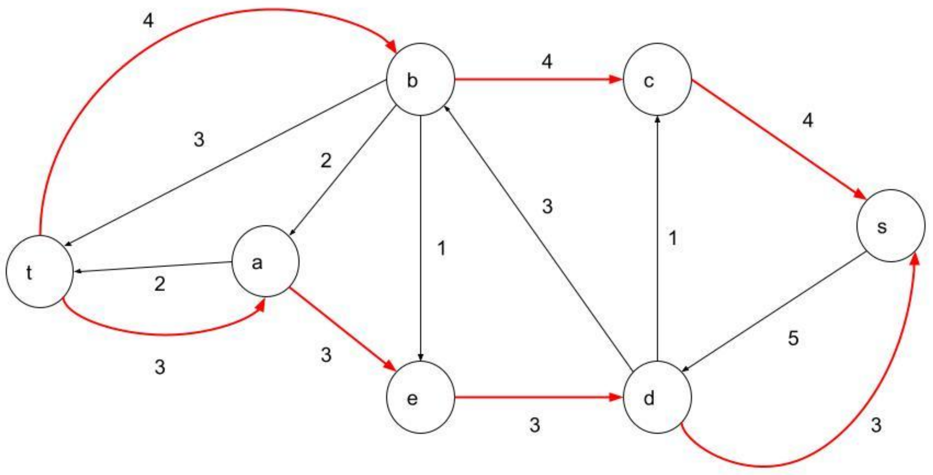
First Augmenting Path: s -> c -> b -> t with flow = 4

Residual Graph 1:



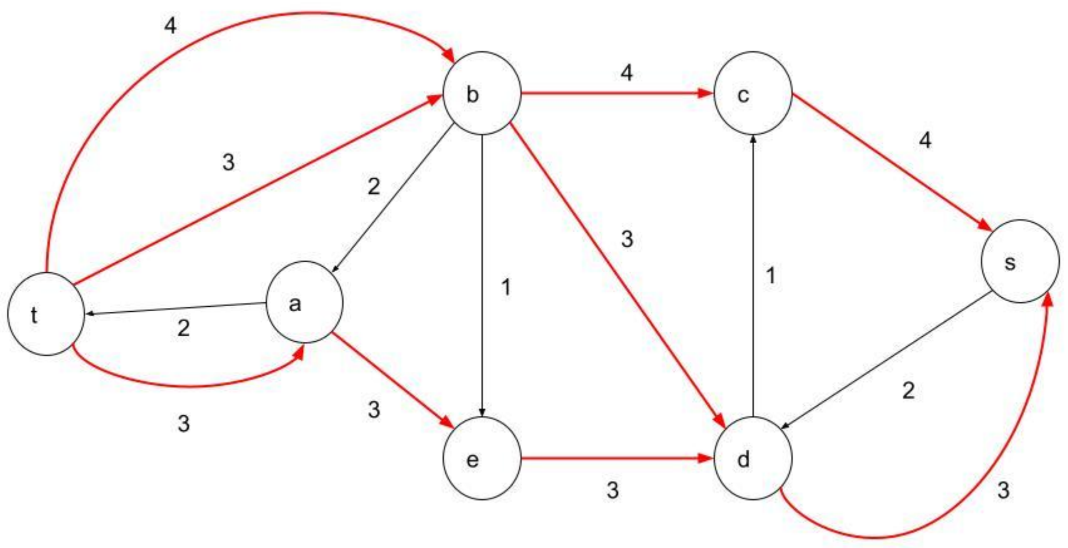
Second Augmenting Path: s -> d -> e -> a -> t with flow = 3

Residual Graph 2:



Third Augmenting path: s -> d -> b -> t with flow 3

Residual Graph 3:



<u>Candidate Solution 2:</u>
First Augmenting Path: s -> c -> b -> t with flow = 4
Second Augmenting Path: s -> d -> b -> a -> t with flow = 2
Third Augmenting Path: s -> d -> b -> t with flow = 1
Fourth Augmenting Path: s -> d -> e -> a -> t with flow = 3

<u>Candidate Solution 3:</u>
First Augmenting Path: s -> d -> e -> a -> t with flow = 3
Second Augmenting Path: s -> d -> b -> a -> t with flow = 2
Third Augmenting Path: s -> d -> c -> b -> t with flow = 1
Fourth Augmenting Path: s -> c -> b -> t with flow = 3
Fifth Augmenting Path: s -> d -> b -> t with flow = 1

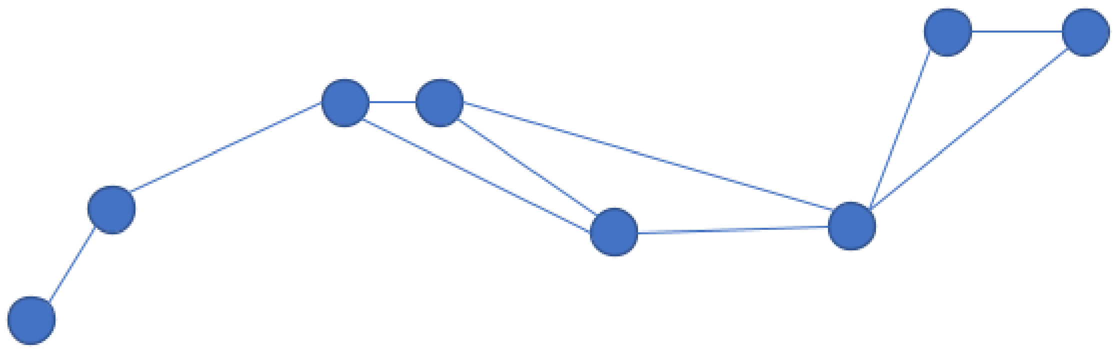
Max- Flow = 10

Since, the value of Max-Flow is less than the total demand value D=12, there is **No Feasible solution in the circulation network, and therefore there is no feasible circulation in the circulation with lower bounds network.**

Rubrics:
2 pts: Finding correct Max-Flow and presenting their appropriate residual graphs according to the sequence of augmenting paths that the student has chosen

1 pt: Correctly concluding “No Feasible Circulation”
1 pt: Reasoning behind “No Feasible Circulation”

3) 20 pts
In the last 570 exam, there were nearly a thousand students that were ready to take the exam in person at 8 different test locations (L1..L8). There were Si students assigned to each room i. The copy center made some big mistakes and instead of delivering Si papers they delivered Ri papers to each room i, where Ri was higher than Si for some rooms and lower for some others. TAs then had to rush and come up with a solution to redistribute the excess papers at each test location to those test locations that had a shortage. And because the time was short, they ruled out sending papers directly between test locations that were far apart from each other on campus. But they did not rule out sending papers indirectly through other test locations. (For example, L1 and L3 in below graph were considered far apart from each other but papers could still be redistributed between them through L2). So with these considerations in mind, they ended up with a connected graph that looked like this:



a) Assuming that the total number of papers delivered was at least equal to the total number of papers required, design a network flow based algorithm to determine how many papers had to be sent (and in which direction) on each edge in the above network in order to supply each room with the required papers. You need to describe exactly how you reduce this problem to a network flow problem. (14 pts)

Solution 3a):
Across all the candidate solutions below, the common steps initially are

1. Use the locations as nodes in the flow network.
2. Convert each undirected edge in the given graph to 2 directed edges in opposite directions in the flow network.
3. No constraints on how many papers MUST be carried along any such edge, so no lower bound. Capacity should be a suitable high value: Infinity (= conservative safest upper bound) OR Summation of Si (= papers needed in total) OR Summation of Ri (= papers available in total) OR $\sum_{\{i\ where\ Si-Ri>0\}} (S_i - R_i)$ (= sum of shortages of papers) OR $\sum_{\{i\ where\ Ri-Si>0\}} (R_i - S_i)$ (= sum of surplus of papers). The flow on an edge between two locations can be at most any of these upper-bounds.

Candidate Solution 1: Circulation with lower bounds:

1. Add a source S and edges S -> L_i for each location. Each such edge corresponds to the papers sent from the copy center. Since this is exactly R_i, set it's LB/cap to R_i/R_i.
2. Add a sink T and edges L_i -> T for each location. Each such edge corresponds to the papers finally retained at location L_i. This needs to be at least S_i , thus that's the LB. It

could be much more; a capacity of as low as max {S_i, R_i} works (a safe upper bound is simply a capacity of infinity)

3. Add a demand value of ∑ R_i at T and $-$ ∑ R_i at S. Alternatively (instead of adding the demands at S,T), adding a T -> S edge suffices. (This is not always equivalent, but works here since the net S-T flow value is fixed due to the LB/cap values of R_i/R_i on the edges out of S). The capacity of this back-edge has to be at least ∑ R_i.

Solution 2: Relative to solution 1, The lower bounds on the edges from S could be removed by adjusting the capacities/demands around. Consequently,, we have no source, but do have a sink, and each L_i has a demand of '- R_i' and they are all connected to the sink with demand ∑ R_i

Solution 3: Relative to solution 2, , we can get rid of the lower bounds on edges L_i -> T. In this case, we have (In addition to the common steps laid out in the beginning) a demand of Si - Ri at location Li and sink T with a demand (sum of Ri - sum of Si). We have edges L_i -> T of capacity R_i - S_i wherever it is positive (but okay to add edges from all the L_i -> T and okay to have bigger capacities).

Solution 4: The circulation network from Solution 3 can be further converted to a simple flow network by getting rid of the demands and adding a super source S* and a super sink T*. S* must connect to all L_i where R_i - S_i is positive, with as much edge capacity. Each L_i with positive S_i - R_i connects to T* with as much capacity and T connects to T* with capacity (sum of Ri - sum of Si). This network must have a max flow of $\sum_{\{i \; where \; Ri-Si > 0\}} (R_i - S_i)$.

Solution 5: A modified version of Solution 4 can get rid of T and all the edges coming in and out of it. This network must have a max flow of value $\sum_{\{i \; where \; Si-Ri > 0\}} (S_i - R_i)$ (note this crucial difference with respect to max flow value of Solution 4).

Solution 6: Create a source S, and connect S -> L_i nodes with capacity R_i and create a sink T, and connect each L_i -> T with capacity S_i. This network must have a max flow of value sum (Si)

Solution 7: This one has a different structure with two partitions (L and L') EACH having nodes for ALL 8 locations. Create S, and edges from S -> L_i nodes with capacity R_i. Create a sink T, and connect L'_i -> T with capacity S_i. Each L_i and its copy L'_i should have edges both ways. But for i \ne j, L_i -> L'_j can be an edge in just this one direction. ALL these edges between L and L' must have the same high capacity as outlined in the beginning of this solution. This network must have a max flow of value sum (Si)

Final answer (i.e. paper redistribution scheme) required in each solution: Compute the feasible circulation (in solutions 1/2/3) or max flow (in solutions 4/5/6/7) F. From location L_i to an adjacent L_j send F(L_i -> L_j) - F(L_j - L_i) papers (or in the reverse direction, whichever gives a positive number).

Rubrics 3a:

- -2 points for each incorrect/missing demand/LB/capacity/node/edge

- -2 pts: if the final algorithm description of how to compute redistribution solution from the constructed network is missing.
- AT MOST half the points if the fundamental structure of the network is wrong (partial credit subjective to how close the proposed solution is)

a) Prove that you solution is correct (6 pts)

Solution 3b):

Here's the proof for solution 1 (Circulation without lower bounds)

Reduction Claim: **If there is a feasible circulation in the resulting network then it gives a feasible redistribution of papers across the exam rooms. (The solution will actually always exist because we have enough papers and no limits on capacities that stop us from sending papers from one location to another.)**

Proof:

Claim 1) If we have a feasible redistribution of papers to exam rooms we will have a feasible circulation in the circulation network constructed:

- Send flow on each edge (in the appropriate direction) equal to the number of papers that are sent out from room to room.
- Send flow from S to each room of R_i
- Send flow from each room to T equal to the final number of papers in that room.
- This will give us a feasible circulation in the network since
    - All lower bound constraints on flow are met (each room got enough papers)
    - All capacity constraints are met (edge capacities were set high enough to allow for movement of any extra papers between locations)
    - All demand conditions are met (especially at T) since redistribution of papers will not cause us to lose or add any flow at any node.

Claim 2) If we have a feasible circulation we can find a feasible circulation of papers to exam rooms

- Send papers equal to flow sent out from room to room
- This will give us a feasible redistribution of papers since
    - All requirements for papers at exam rooms will be met (since lower bounds on flow from room to T are met)
    - There are no physical limits on how many papers can be transported from room to room

Other proofs follow the same format.
- For candidate solutions 4-7, the claim should say "The max flow value is ___ if and only if …" and not "max flow exists if and only if..."
-A common mistake is an attempt to have a circulation formulation as in Solution 3 but missing the sink therein, and then incorrectly reducing it to max flow which looks as the one in Solution 5 (circulation -> flow conversion is not necessary, i.e., has no credit). (The penalty

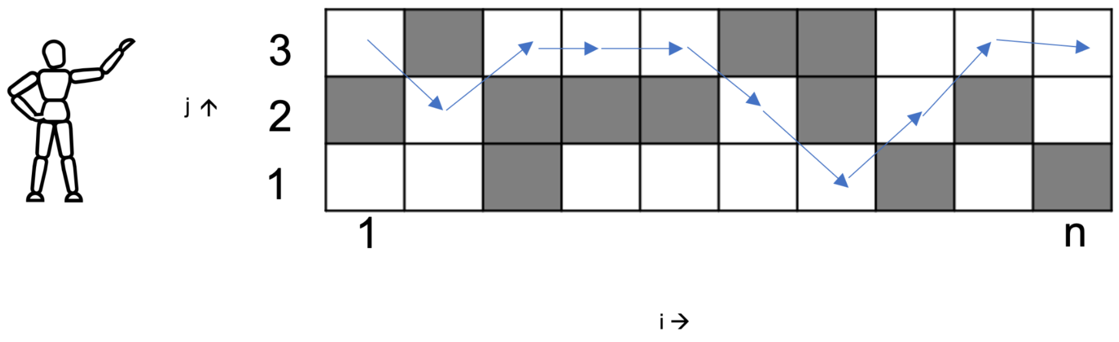
for this error will be reduced from 3.5 to 2.5 IF the claim/final answer is correctly given in terms of the max flow value of the resultant network).

Rubrics 3b:

- **If** the claim is correct,
    - 3 pts: Proof of the Forward claim.
    - 3 pts: Proof of the backward claim.
- Partial credit of at most 2 in total if the claim itself is incorrect (for the proposed solution), but can qualify as a fair attempt..

4) 20 pts

Jack has gotten himself involved in a very dangerous game called the octopus game where he needs to pass a bridge which has some unreliable sections. The bridge consists of 3n tiles as shown below. Some tiles are strong and can withstand Jack's weight, but some tiles are weak and will break if Jack lands on them. Jack has no clue which tiles are strong or weak but we have been given that information in an array called BadTile(3,n) where **BadTile (j, i) = 1 if the tile is weak and 0 if the tile is strong.** At any step Jack can move either to the tile right in front of him (i.e. from tile (j, i) to (j, i+1)), or diagonally to the left or right (if they exist). (No sideways or backward moves are allowed and one cannot go from tile (1,i) to (3, i+1) or from (3, i) to (1, i+1)). Using dynamic programming find out how many ways (if any) there are for Jack to pass this deadly bridge. Figure below shows bad tiles in gray and one of the possible ways for Jack to safely cross the bridge alive.



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

OPT(j, i) = The number of ways Jack can reach the block (j, i) safely from the bridge start

OR

OPT(j, i) = The subproblems are the number of ways Jack can reach the end safely starting from block (j, i)

Rubrics:
Missing from block or to block : -1

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

Variant 1:

OPT(j, i) = 0	if BadTile(j, i) = 1;	
= OPT(j,i-1) + OPT(j+1,i-1)		if j = 1;
= OPT(j,i-1) + OPT(j+1,i-1) + OPT (j-1,i-1)		if j = 2;
= OPT(j,i-1) + OPT (j-1,i-1)		if j = 3;

Variant 2 :
In the second variant, i-1 is replaced by i+1 in each of the terms on the RHS

Variant 3 :

```
OPT1[i] = 0                                  if BadTile = 1
        = 1                                  if(i ==1)
        = opt2[i-1] + opt1[i-1]                  otherwise

OPT2[i] = 0                                  if BadTile = 1
        = 1                                  if(i ==1)
        = opt2[i-1] + opt1[i-1] +opt3[i-1]           otherwise

OPT3[i] = 0                                  if BadTile = 1
        = 1                                  if(i ==1)
        = opt2[i-1] +opt3[i-1]               otherwise
```

Rubrics:
If only j=2 case is written and other cases are not mentioned : -2 (-1 for each case missed)
If recurrence relation is partially correct : -4
Partially wrong recurrence relation -> adding +1 at each step : -2

c) Using the recurrence formula in part b, write pseudocode using iteration to compute the total number of ways to safely cross the bridge. (5 pts)
Make sure you specify
- base cases and their values (2 pts)
- where the final answer can be found (e.g. opt(n), or opt(0,n), etc.) (1 pt)

For Variant 1:

```
Given BadTile (3,n)
Initialize OPT(3,n) with 0 for each element
OPT(1,1) = 1 if BadTile (1,1) equals to 0, else 0
OPT(2,1) = 1 if BadTile (2,1) equals to 0, else 0
OPT(3,1) = 1 if BadTile (3,1) equals to 0, else 0

for i in  2, 3,4, …, n:
    for j in 1, 2, 3:
        if BadTile(j, i) equals to 1:
            OPT(j, i) = 0
        if j equals to 1:
            OPT(j, i) = OPT(j, i-1) + OPT(j+1, i-1)
        if j equals to 2:
            OPT(j, i) = OPT(j,i-1) + OPT(j+1,i-1) + OPT (j-1,i-1)
        if j equals to 3:
            OPT(j, i) = OPT(j,i-1) + OPT (j-1,i-1)

return OPT(1, n)+OPT(2, n)+OPT(3, n)
```

For Variant 2:

Initialize OPT(5,n)

```
OPT(0,i)  = 0 //outside the grid
OPT(4,i) = 0 // outside the grid

For i in 1 to n:
        For j =1 to 3:
                OPT(j, i) = OPT(j,i-1) + OPT(j+1,i-1) + OPT (j-1,i-1)

return OPT(1, n)+OPT(2, n)+OPT(3, n)
```

If variant 2 is written recurrence would be just j = 2 case

For Variant 3:

```
Initialize 3 arrays opt1[], opt2[] opt[3];

For i in 2 to n:
        Above recurrence
return OPT(1, n)+OPT(2, n)+OPT(3, n)
```

If variant 2 is written recurrence would be just j = 2 case

Rubric:
If where final answer can be found is not mentioned : -1
If recurrence relation is not shown or is wrong : -2
If 3 cases are not shown : -2
If base condition are given according to example : -1

d) What is the complexity of your solution? (1 pt)
   Is this an efficient solution? (1 pt)

O(N). Yes.

5) 20 pts
Assume a truck with capacity W is loading. There are n packages with different weights, i.e. [w1, w2, … wn], and all the weights are integers. The company's rule requires that the truck needs to take packages with exactly weight W to maximize profit, but the workers like to save their energies for after work activities and want to load as few packages as possible. Assuming that there are combinations of packages that add up to weight W, design an algorithm to find out the minimum number of packages the workers need to load.

NOTE: For those who assumed n packages to mean "n types of packages", we've still decided to award points for it <u>out of a maximum</u> possible 13 points. (i.e. for any errors in the attempted solution under this mis-interpretation will have reductions out of the 13 max possible score). This variant of the problem is in fact the one asked in the onlinc version of the exam, please refer to it for solutions and rubrics.

Solution 1:

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

OPT(w,k) = min packages needed to make up a capacity of exactly w, by considering only the first k packages
OR
Same except, considering packages k onwards up to n (add details below)

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

If $w >= w_k$
$OPT(w,k) = \min\{ 1+OPT(w- w_k , k-1) , OPT(w, k-1) \}$
Else
$OPT(w,k) = OPT(w, k-1)$

(No penalty for missing the latter case IF the base cases in 'c)' include $w < 0$.)

c) Using the recurrence formula in part b, write pseudocode using iteration to compute the minimum number of packages to meet the objective. (5 pts)

1. Initialize for base cases (mentioned below for clarity)

```
2.  For w = 0 to W
        For k = 0 to n        //can switch the inner-outer loops
            If not base case: call recurrence
3.  Return ans (mentioned below for clarity)
```

Make sure you specify

- base cases and their values (2 pts)

OPT(w,0) = inf for all $w < 0$ (up to -W)

OPT(0,0) = 0

OPT(w,k) = infinity for $w < 0$ and all k. (Not required if the recurrence has the second case to ensure w never takes negative values)

- where the final answer can be found (e.g. opt(n), or opt(0,n), etc.) (1 pt)

OPT[W, n] in the first definition

OPT[W, 1] in the alternate one

d) What is the complexity of your solution? (1 pt)
Is this an efficient solution? (1 pt)

O(nW) pseudo-polynomial run time

This is not an efficient solution

Solution 2:

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

OPT(w,k) = min packages needed to make up a capacity of exactly w, by considering only the first k packages, AND selecting package k for sure.

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

The recurrence captures all cases of j where j was the highest index used before the kth one:

If $w >= w_k$

$OPT(w,k) = 1 + \min\_\{j=0 \text{ to } k-1\}\ OPT(w- w_k , j)$

Else

OPT(w,k) = infinity

(No penalty for missing the latter case IF the base cases in 'c)' include $w < 0$.)

c) Using the recurrence formula in part b, write pseudocode using iteration to compute the minimum number of packages to meet the objective. (5 pts)

```
4.  Initialize for base cases (mentioned below for clarity). Initialize rest to infinity
```

```
5.    For w = 0 to W
          For k = 0 to n          //can switch the inner-outer loops
              If (not base case): //call recurrence
                  For (j = 0 to k-1)
                      OPT(w,k) = min{OPT(w,k), 1 + OPT(w- $w_k$ , j)}
6.    Return ans (mentioned below for clarity)
```

Make sure you specify

- base cases and their values (2 pts)

OPT(w,0) = inf for all w >0

OPT(0,0) = 0

OPT(w,k) = infinity for w < 0 and all k. (Not required if the recurrence has the second case to ensure w never takes negative values)

- where the final answer can be found (e.g. opt(n), or opt(0,n), etc.) (1 pt)

Max_k OPT(W, k)

d) What is the complexity of your solution? (1 pt)
Is this an efficient solution? (1 pt)

$O(n^2W)$ pseudo-polynomial run time

This is not an efficient solution