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:

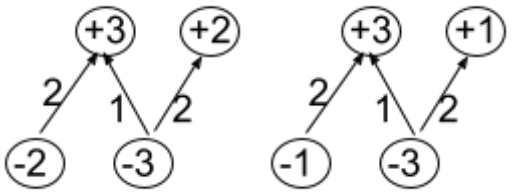
1. This is a 2-hr exam. Open book and notes and internet access. But no internet communications through social media, chat, or any other form is allowed.
2. If a description to an algorithm or a proof is required, please limit your description or proof to within 150 words, preferably not exceeding the 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.

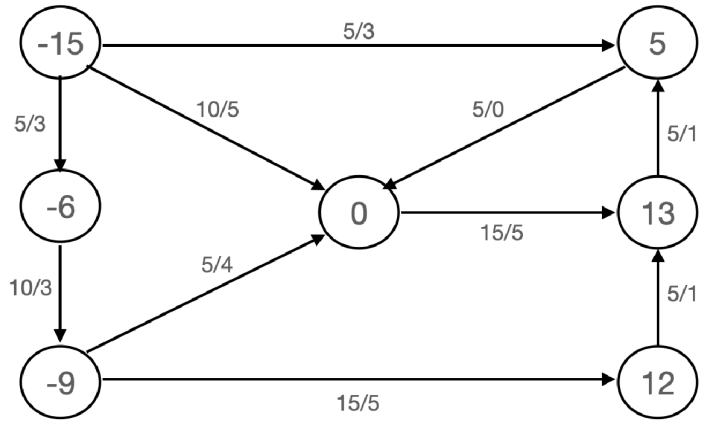
Dynamic Programming (4 pts)

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.

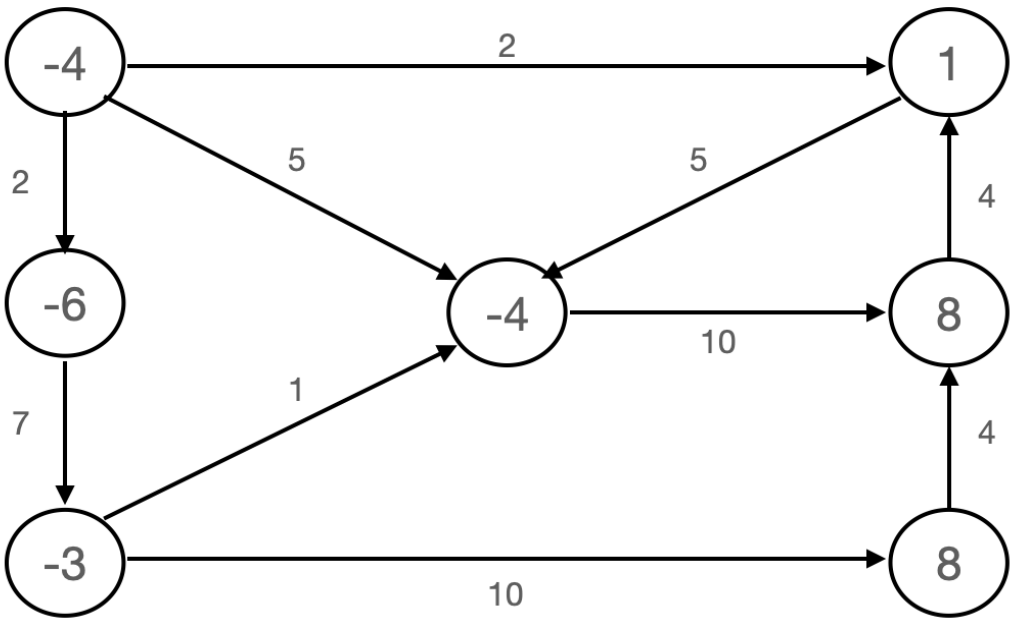


a. Reduce the Feasible Circulation with Lower Bounds problem to a Feasible Circulation problem without lower bounds. (8 pts)

Draw G'

If demand is correct, get 4 pts

If flow on edges are correct, get 4 pts



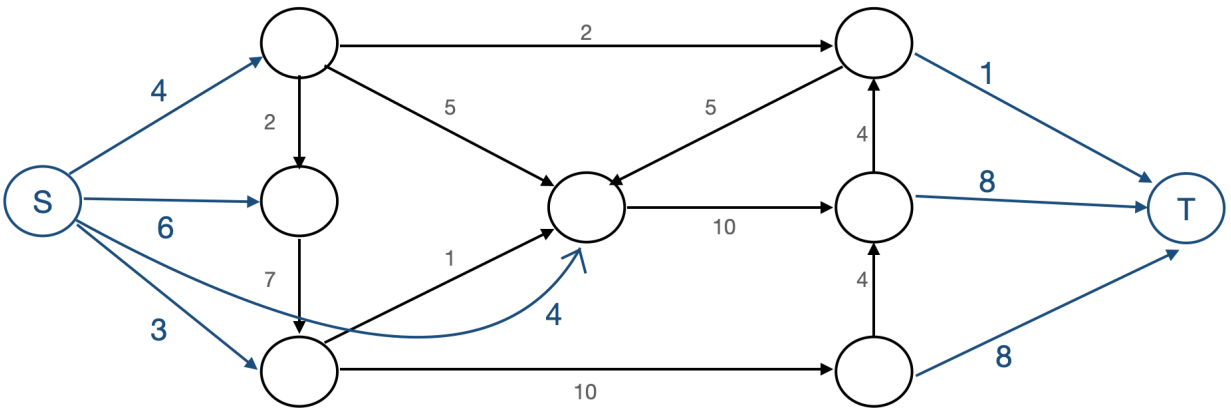
b. Reduce the Feasible Circulation problem obtained in *part a* to a Maximum Flow problem in a Flow Network.(8 pts)

Add S and T (2 pts)

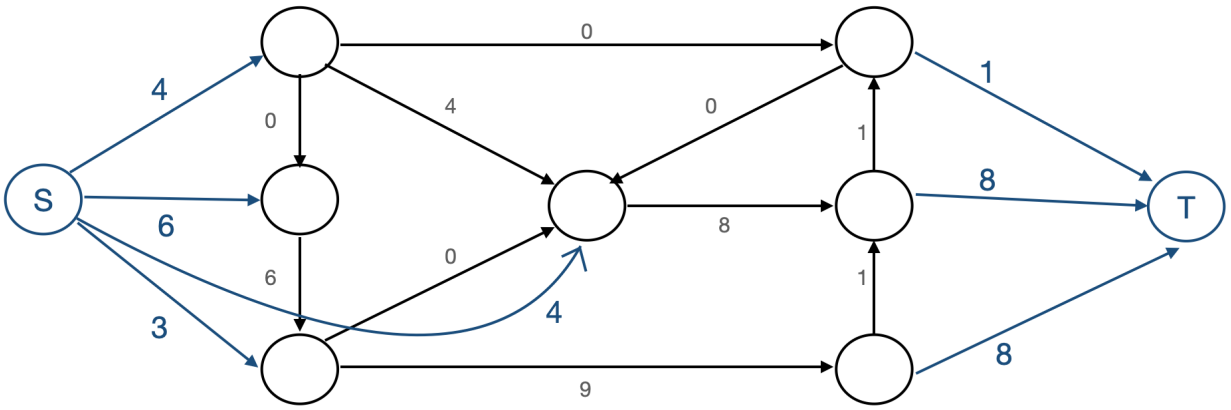
Connect S with correct nodes (2 pts)

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)

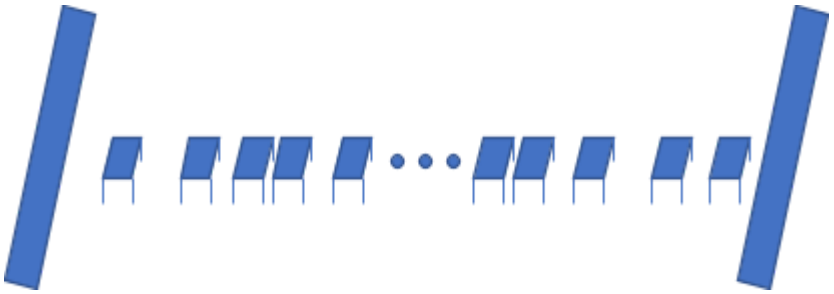


This is the max flow in Flow network. (2 pts)

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 **all** 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 k^{th} one:

If $w >= w_k$
$OPT(w,k) = 1 + \min_{j=0 \text{ to } k} OPT(w- w_k , j)$
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)

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) = 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

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)