CS570
Analysis of Algorithms
Spring 2009
Exam II

Name: _____

Student ID: _____

____2:00-5:00 Friday Section ____5:00-8:00 Friday Section

	Maximum	Received
Problem 1	20	
Problem 2	20	
Problem 3	20	
Problem 4	20	
Problem 5	20	
Total	100	

2 hr exam
Close book and notes

1) 20 pts
   Mark the following statements as **TRUE** or **FALSE**. No need to provide any justification.

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

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

[TRUE/FALSE] TRUE
An edge that goes straight from s to t is always saturated when maximum s - t flow is reached.

[TRUE/FALSE] FALSE
In any maximum flow there are no cycles that carry positive flow.
(A cycle $<e_1, \ldots, e_k>$ carries positive flow iff $f(e_1) > 0, \ldots, f(e_k) > 0$.)

[TRUE/FALSE] TRUE
There always exists a maximum flow without cycles carrying positive flow.

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

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

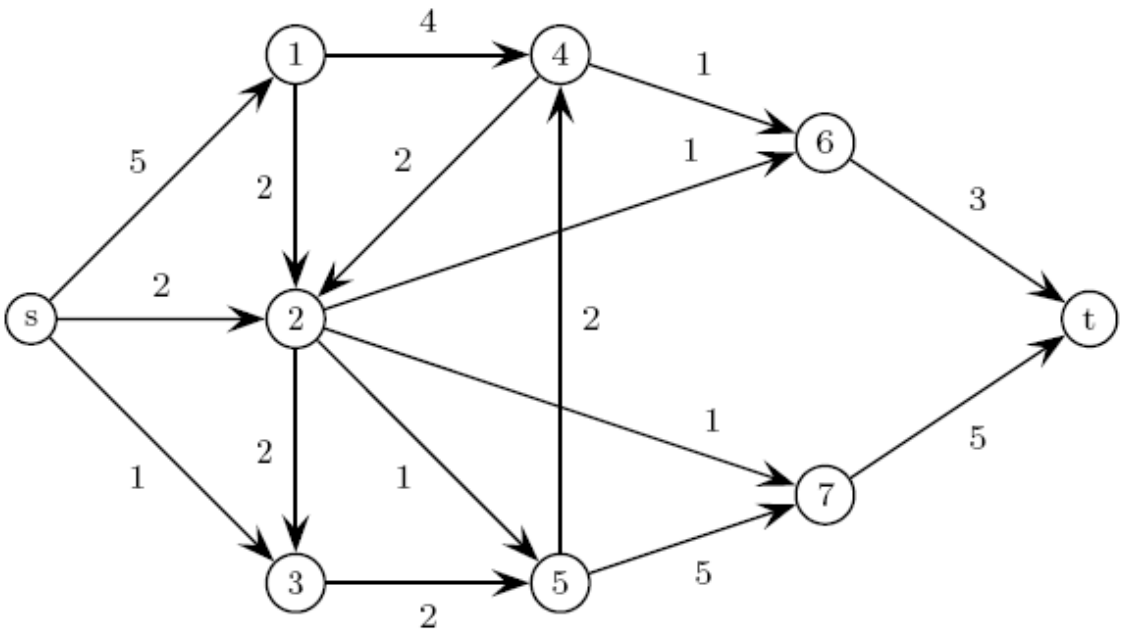
[TRUE/FALSE] FALSE
Any Dynamic Programming algorithm with n unique subproblems will run in $O(n)$ time.

[TRUE/FALSE] FALSE
The running time of a pseudo polynomial time algorithm depends polynomially on the size of of the input

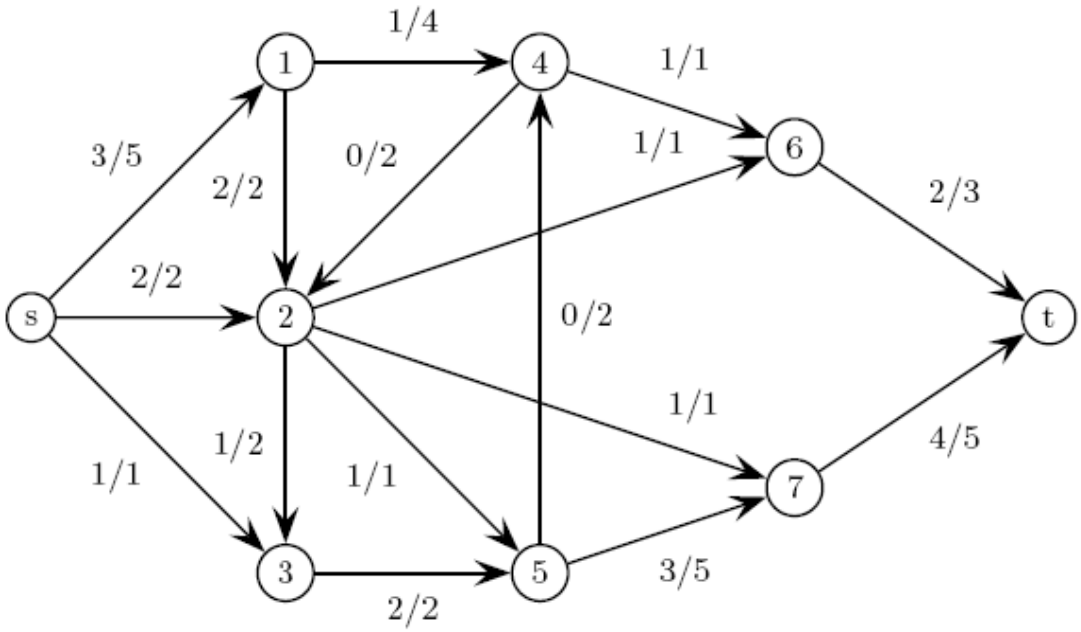
[TRUE/FALSE] FALSE
In dynamic programming you must calculate the optimal value of a subproblem twice, once during the bottom up pass and once during the top down pass.

2) 20 pts

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



Solution:



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

Solution:

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

3) 20 pts
On a table lie *n* coins in a row, where the i*th* coin from the left has value $x_i >= 0$. You get to pick up any set of coins, so long as you never pick up two adjacent coins. Give

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

We use Dynamic Programming to solve this problem. Let OPT(i) denote the maximum value that can be picked up among coins 1, . . . , *i*.
Base case: OPT(0) = 0, and OPT(1) = x1.

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

OPT(0) = 0
OPT(1) = x_1
OPT(i) = max(OPT(i-1), x_i + OPT(i − 2)) for i > 1.

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

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

4) 20 pts
We assume that there are n tasks, with time requirements $r_1, r_2, \ldots, r_n$ hours. On the project team, there are *k* people with time availabilities $a_1, a_2, \ldots, a_k$. For each task *i*

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

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

Solution:

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

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

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

5) 20 pts

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

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

Solution

Let OPT(i,j) = The optimal cost of reaching from departure point "i" to departure point "j".

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

This gets us to the following recurrence

OPT(i,j) = min(over j<= k < i)(C[i,k] + OPT(k,j))

The base case is OPT(i,i) = C[i,i] = 0 for all "i" from 1 to n

The iterative program will look as follows

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

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

Now, to output the cost from the last post to the first post, will be given by OPT[n,1]

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