CS570
Analysis of Algorithms
Fall 2013
Exam II

Name: ______________________
Student ID: ______________________

____Tuesday/Thursday Session ____Wednesday Session ____DEN

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

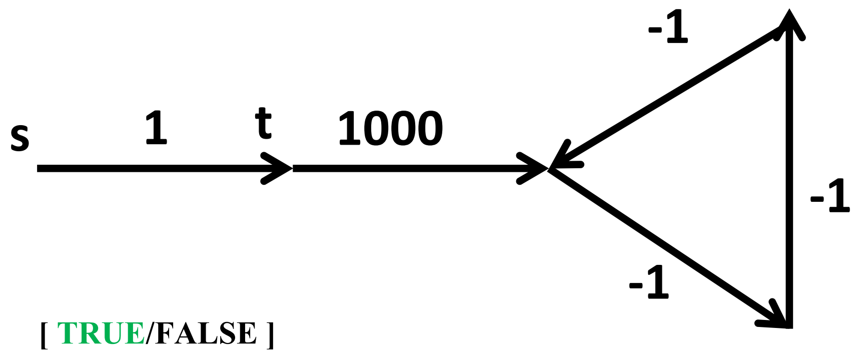
2 hr exam
Close book and notes

If a description to an algorithm is required please limit your description to within 150 words, anything beyond 150 words will not be considered.

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

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

Counter example: Consider this graph. Bellman-Ford will limit the number of edges to at most 4 on the shortest path and terminates (because the number of nodes is 5). Thus, the shortest path from s to t is correct.



[TRUE/FALSE]
A negative cycle in a graph can be detected in polynomial time.

The complexity is O(m*n) while m (number of edges) is upper-bounded by n^2, this this can be done in $O(n^3)$, which is polynomial.

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

Linear in space only, not linear in time.

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

O(Cm) is the worst case. Therefore the best time complexity can be better than O(Cm).

[TRUE/FALSE]
Bellman-Ford algorithm cannot solve the shortest path problem in graphs with negative cost edges in polynomial time.

If there is no negative cycle, the complexity is polynomial $O(n^3)$. If there is a negative cycle, the algorithm can also detect it in polynomial time $O(n^3)$. So the overall complexity is polynomial.

[TRUE/FALSE]
In the Ford–Fulkerson algorithm, choice of augmenting paths can affect the number of iterations.

Choosing the paths with higher bottlenecks may significantly reduce the number of iterations.

[TRUE/FALSE]
For flow networks such that every edge has a capacity of either 0 or 1, the Ford-Fulkerson algorithm terminates in O(n^3) time, where n is the number of vertices.

Complexity is $O(Cm)$. However, if all capacities are either 0 or 1, then $C \leq n$ (number of nodes that are directly connected to source s). Thus, $Cm \leq nm \leq n^3$. Thus, Ford-Fulkerson will terminate in $O(n^3)$.

[TRUE/FALSE]
Every flow network with a non-zero max s-t flow value, has an edge e such that decreasing the capacity of e decreases the maximum s-t flow value.

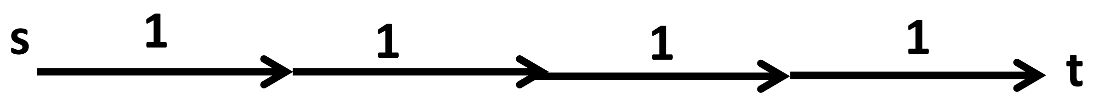
This is obvious from the max flow and min cut theorem.
Max flow $f^* = C(\text{min cut}) = C^*$. Decreasing the capacity of any edge that belongs to a min cut will result in a smaller cut $C' < C^*$, therefore the new max flow will be at most $C' < C^* = f^*$.

[TRUE/FALSE]
Decreasing the capacity of an edge that belongs to a min cut in a flow network will always result in decreasing the maximum flow.

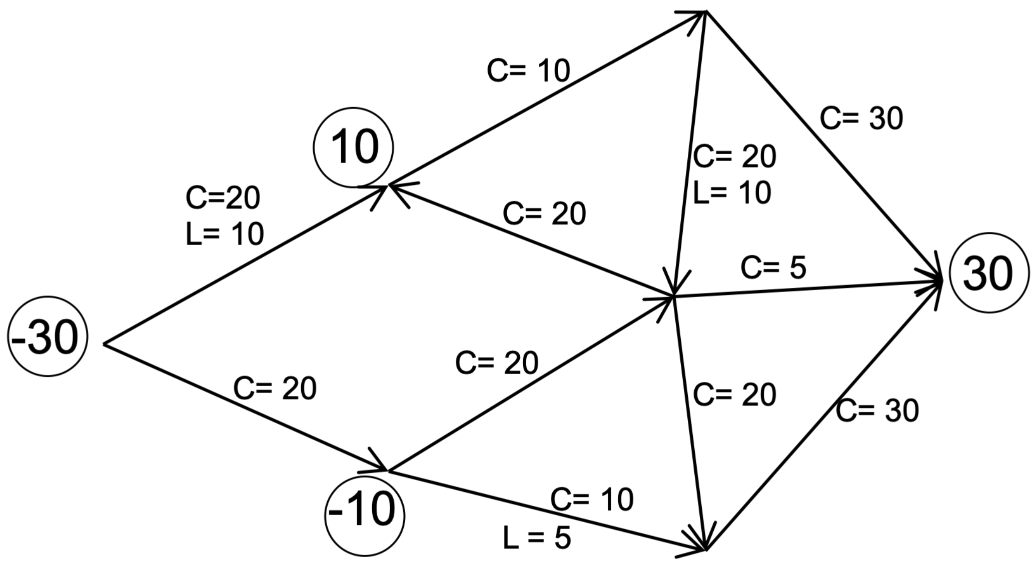
For the same reason as in the previous question.

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

Counter example shown below: Increasing the capacity of any edge will not increase the max flow.



2) 20 pts
Determine if there is a feasible flow in the following network. You need to show all your steps. Demand values are in circles. L denotes the lower bound and C denotes the upper bound of the flow of an edge.

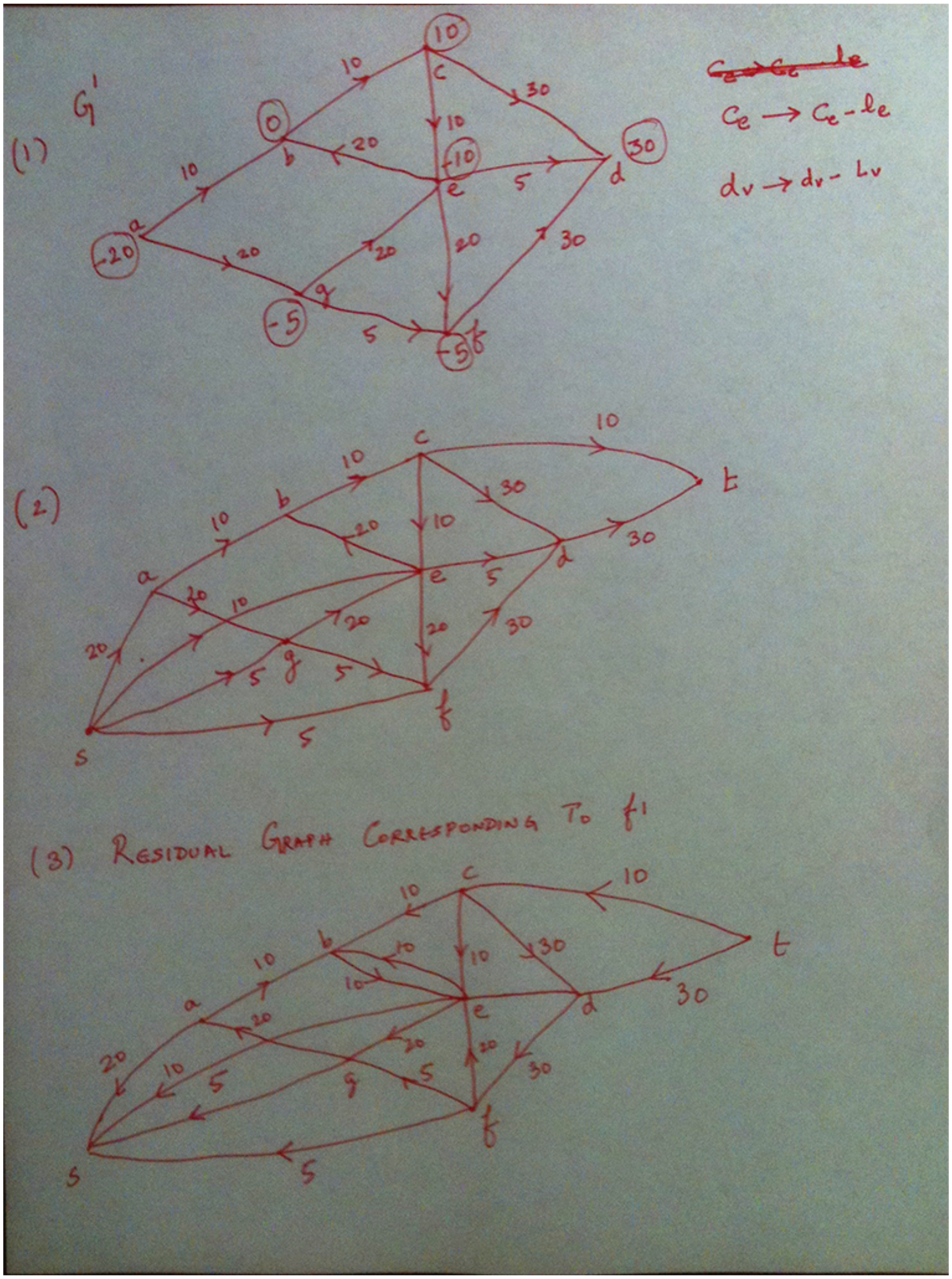


The question tests if you understand the general procedure in finding a circulation in a network with demands and lower bounds. To this end, you are asked to show the steps that you took in arriving at your answer (It is not sufficient if you merely state the correct answer.)

You first need to compute an initial circulation f_0, where for all edges e, f_0(e)=l_e to account for the lower bound constraints. Then, the problem reduces to finding a circulation in the network below (Fig 1 in the image attached) where the demands and capacities are adjusted (d_v → d_v – L_v and c_e→c_e – l_e) to clear the imbalance created by f_0.

Then introduce a source s and a sink t and arrive at the network shown below (Fig 2 in the image attached) by connecting the source to all the vertices with negative demands and from all vertices with demands to the sink.

Then compute a max-flow f_1 for this network by using Ford-Fulkerson's algorithm (Fig 3 in the image attached shows the residual graph corresponding to one such max flow. Please show the augmenting steps and residual graph computations that you performed to arrive at f_1). Then observe that the value of the max-flow computed equals 40 which is the sum of all the positive demands and hence conclude that the original network has a feasible circulation (which is obtained by adding f_0 and f_1).



3) 20 pts

A rope has length of N units, where N is integer. You are asked to cut the rope (at least once) into different smaller pieces of integer lengths so that the product of the lengths of those new smaller ropes is maximized.

Similar to the rental car example shown in the review session:
Going bottom-up: the base cases are N=1,2. OPT(N)=1 (no possible cut). OPT(2) =1 since it has to be cut at least once and there is only one way to cut.
Assume we know all the OPT values from 1 to N-1: OPT(1), OPT(2),…, OPT(N-1). Now we have a rope of length N > 1. The rope can be cut multiple times (or rounds), but in the first round there are N-1 ways to cut it at points i = 1, 2, 3,... ,N-1 (One needs not worry about how to cut the smaller ropes in the further rounds at the moment because those are handled by OPT(i) that i < N.).
Since there are N-1 ways to cut the rope, so the OPT(N) will be the max of all N-1 possible points of cutting: OPT(N) = max {OPT(i) * OPT(N-i), i*OPT(N-i), OPT(i)*(N-i), i*(N-i) } for all i = 1,2,…, N-1. i*OPT(N-i) means we do not want to cut i unit further and only want to further cut (N-i) units. As mentioned earlier, OPT(i) and OPT(N-i) will further handle how to cut smaller ropes of length < N. This explanation is to help students understand the solution. The exam does not ask students to explain this, just simply write the recurrence equation and explain the symbols they use.

a- **(9 points)** Write a recurrence formula to solve this problem using dynamic programming.
OPT(1)=1;
OPT(i) = max {OPT(j)*OPT(i-j), j*OPT(i-j), OPT(j)*(i-j), j*(i-j)} for all j =1,2,…, i-1, for i>1.
Where OPT(i) is the maximum product of the lengths of smaller ropes in the optimal cut.

b- **(9 points)** Present your solution in pseudo code. Use iteration and describe the symbols used.
First, one needs to write pseudo-code to find all OPT(i) for i = 1,…, N (bottom up), Then trace back the OPT(i) to find the cut points (top-down)

```
Create array OPT[i]
Initialize OPT(1)=1; OPT[i]=0 for all i=2,3,...N;
For i = 2 to N {
        For j = 1 to i-1
                If OPT[i] < OPT(j)*OPT(i-j)
                        OPT[i] = OPT(j)*OPT(i-j);
                If OPT[i] < j*OPT(i-j)
                        OPT[i] = j*OPT(i-j);
```

```
            If OPT[i] < OPT(j)* (i-j)
                OPT[i] = OPT(j)* (i-j);

            If OPT[i] < j* (i-j)
                OPT[i] = OPT(j)*OPT(i-j);

            Endif;
        Endfor;
Endfor;
```

The previous code only provides the optimal value, but does not tell how to cut, thus one needs to trace back to find points to cut;

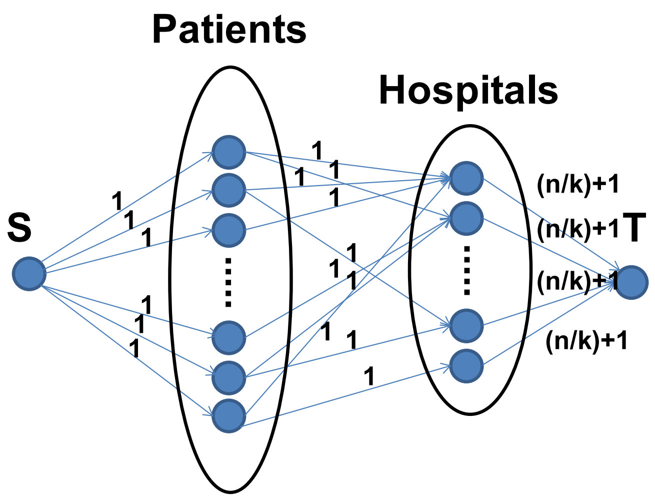
```
For i=N to 2
    For j = 1 to i-1
        If OPT[i] = OPT(j)*OPT(i-j)
            Report: length i, cut at j, further cut j and i-j;
            Break;
        If OPT[i] = j*OPT(i-j)
            Report: length i, cut at j, keep(j), further cut i-j;
            Break;
        If OPT[i] = OPT(j)* (i-j)
            Report: length i, cut at j, further cut j, keep (i-j)
            Break;
        If OPT[i] = j* (i-j)
            Report: length i, cut at j, keep(j) and keep(i-j);
            Break;
        Endif;
    Endfor;
Endfor;
```

c- **(2 points)** What is the complexity of your solution
$O(n^{2})$

4) 20 pts
Due to large-scale flooding in a region, paramedics have identified a set of n injured people distributed across the region who need to be rushed to hospitals. There are k hospitals in the region, and each of the n people needs to be brought to a hospital that

is within a half-hour's drive to their current location.(So different patients will be able to be served by different hospitals depending upon the patients' locations.) However, overloading one hospital with too many patients at the same time is undesirable, so we would like to distribute the patients as evenly as possible across all the hospitals. So the paramedics (or a centralized service advising the paramedics) would like to work out whether they can choose a hospital for each of the injured people in such a way that each hospital receives at most (n/k+1) patients.

a- Describe a procedure that takes the given information about the patients' locations (hence specifying which hospital each patient could go to) and determines whether a balanced allocation of patients is possible (i.e. each hospital receives at most (n/k+1) patients).



If the hospital is within the patient half-hour's drive we connect them

b- Provide proof of correctness for your procedure

Each unit flow from S to T is equivalent to assigning a patient to a hospital with the following restriction:

Each hospital get less than (n/k)+1 patients.
Each patient will be assign to only one hospital which is located in the half-hour's drive to the patient

We use the ford-Fulkerson algorithm to find the maximum flow. The maximum flow is the maximum of this assignment and if we can assign all patients (max flow = n) we can do the balance allocation.

c- What is the asymptotic running time of your procedure (in terms of n and k)?

We use the Ford-Fulkerson algorithm to solve this problem and in for this algorithm the running time is O(Cm) in which the C is the maximum possible flow and m is the number of edges. Thus, C= n and m = n + nk + k and the complexity will be O(n(n+k+nk)) =O(n^2k)

5) 20 pts

A matching of an undirected graph is a subset of the edges of the graph such that no two edges in the subset share a vertex. Design an algorithm that given a tree T finds a matching of T with the maximum number of edges. The running time should be bonded by a polynomial in the number of vertices in the tree.

There are at least two ways of approaching this problem. One is through dynamic programming and the other through network flows.

Perhaps the easier solution is through network flows.

The first observation is that every tree is bipartite (There are no cycles in a tree which implies that there are no odd cycles in a tree which implies that a tree is bipartite). But from class (or Sec 7.5 in text) we know how to compute a maximum matching of bipartite graph! All that remains is to represent our tree as a bipartite graph (that is, find a partition of its vertices such that there are no edges contained within a single partition). This is accomplished by rooting the tree at an arbitrary vertex and considering all the odd level vertices on one side of the partition and the even level vertices on the other side. (This can be performed efficiently by BFS).

There is a very natural dynamic programming solution which we next briefly sketch. Root the tree T=(V,E) at an arbitrary vertex r. For a vertex x, let T_x denote the subtree rooted at x and let OPT(x) denote the size of the maximum matching of the subtree rooted at x.

Now consider a maximal matching M_v for the tree rooted at v. Either M_v does not contain an edge incident on v or it does. If it does, then M_v contains an edge (u,v) where u is a child of v. Further, M_v restricted to T_w (that is, the intersection of M_v with the edge set of T_w) where w is a child of v other than u is a maximum matching for T_w. Likewise, M_v restricted to T_z where z is a child of u is a maximum matching of T_z. Hence, we can deduce the following recurrence,

OPT(v) equals the minimum of the following two terms:

(i) Sum of OPT(u), where the summation is over children of v.
(ii) Min_{u is a child of v} {sum of OPT(w) where w ranges over the children of v other than u + sum of OPT(z) where z ranges over the children of u}.

The above recurrence can be solved bottom up (that is starting from the leaves). Keep track of the choice made at each iteration (that is, was an edge from v to a child selected in the matching ? and if so which one) and trace back the Maximum matching M_r from the computed OPT() values.

The initial condition is that OPT(y)=0 for all leaves y. When implemented naively the running time is O(n^2) since we have n subproblems and to compute OPT(v) given the OPT values of the descendants of v we at most need to spend O(n) time). Here n denotes the number of vertices in the tree.