Midterm 2

Name:

TA:

Section Time:

Answer all questions. Read them carefully first. Be precise and concise. The number of points indicate the amount of time (in minutes) each problem is worth spending. Write in the space provided, and use the back of the page for scratch. Good luck!

1	
2	
3	
4	
5	
total	

Problem 1 *(25 points)*

$$\begin{aligned} \max \quad & x_1 + x_2 + x_3 \\ \text{subject to:} \quad & x_1 + x_2 \leq 1 \\ & x_1 + x_3 \leq 1 \\ & x_2 + x_3 \leq 1 \\ & x_1, x_2, x_3 \geq 0 \end{aligned}$$

1. Write the dual (use variables y_1 etc.).

   $$\begin{aligned} \min \; & y_1 + y_2 + y_3 \\ \text{subject to } & y_1 + y_2 \geq 1 \\ & y_2 + y_3 \geq 1 \\ & y_1 + y_3 \geq 1 \\ & y_1, y_2, y_3 \geq 0 \end{aligned}$$

2. What are the optimum values of x_1, x_2, and x_3? Of y_1 etc? How do you know they are optimal?

   $(\frac{1}{2}, \frac{1}{2}, \frac{1}{2})$ is a feasible solution for both the primal and the dual, which furthermore achieves the same value of $\frac{3}{2}$ for both. By *weak* duality, this solution is optimal for both the primal and the dual.

3. Simplex (for the original LP) starts at the point $(0, 0, 0)$. In the first step, it increases the variable x_2. What happens next? Rewrite the objective function within the new coordinate system.

   We increase variable x_2 while maintaining $x_1 = x_3 = 0$ until one or more constraints become tight. Here, constraints $x_1 + x_2 \leq 1$ and $x_2 + x_3 \leq 1$ both become simultaneously tight at $x_2 = 1$. At this point, we have reached vertex $(0, 1, 0)$ which can also be defined as the intersection point of the tight constraints $(x_1 = 0), (x_1 + x_2 = 1), (x_3 = 0)$, which collectively define a new basis (alternatively, constraints $(x_1 = 0), (x_2 + x_3 = 1), (x_3 = 0)$ also define the same basis). We now rewrite the program in this new basis.

   A point of coordinates (x_1, x_2, x_3) has coordinates $(x_1, 1 - x_1 - x_2, x_3)$ in the new basis, thus we can directly reuse coordinates x_1, x_3 and need to introduce the new coordinate $w \geq 0$ such that $w = 1 - x_1 - x_2$. Thus $x_2 = 1 - x_1 - w$ and the cost function becomes $x_1 + x_2 + x_3 = 1 - w + x_3$ in the new basis.

4. Suppose now that the third constraint of the original LP is replaced by an equality; that is, by

   $$x_2 + x_3 = 1.$$

   Does the dual change? Does the optimum solution change? Justify briefly.

   The dual change in that constraint $y_3 \geq 0$ disappears. The optimum does not change: you can see this by either reusing the weak duality argument, or more simply by noticing that constraint $x_2 + x_3 \leq 1$ was already tight in the optimal solution.

Problem 2 *(15 points)*

The following is a list of possible properties of a dynamic programming algorithm:

A Runs in linear time.

B Runs in $\Theta(n^2)$ time (n is the size of the input).

C Does not run in polynomial time.

D The dag of subproblems is a "grid": $\Theta(n^2)$ nodes and $\Theta(n^2)$.

E The dag of subproblems has $\Theta(n^2)$ nodes and $\Theta(n^3)$ edges.

F The dag of subproblems is a tree.

Write next to each of the following problems solved in class by dynamic programming **all** and only properties (A, E, etc.) that apply to that algorithm.

- Edit distance (sequence alignment)

  D You have $\Theta(n^2)$ subproblems each of them connected to three other subproblems.

  B Each of the subproblems can be solved in $\Theta(1)$ time. Hence the algorithm is $\Theta(n^2)$

- Traveling salesman problem

  - There are $\Theta(n2^n)$ subproblems.

  C Each of the subproblems takes time $\Theta(n)$. The overall algorithm takes time $\Theta(n^2 2^n)$ which is not polynomial.

- All-pairs shortest path (Floyd-Warshall)

  - We have subproblems $\texttt{dist}(i, j, k)$ standing for the shortest path between i and j using only the vertices $1, 2, \ldots, k$, i.e. there are $\Theta(n^3)$ subproblems.

  - Each of the subproblems is solved in time $\Theta(1)$. The overall algorithm runs in time $\Theta(n^3)$.

- Knapsack (one copy of each item)

  - We have subproblems $K(w, j)$ which stand for the maxium value achievable using a knapsack of capacity w and items $1, 2, \ldots, j$. We need to solve $K(W, k)$. Note that W can be represented $\Theta(\log(W))$ bits. As a consequence, as n denotes the size of the input, we have $\Omega(2^n)$ subproblems.

  C As we solve an exponential number of subproblems, the algorithm has exponential runtime.

- Chain matrix multiplication

  E Subproblems $C(i, j)$ correspond to the minimum cost of multiplying matrices A_i through A_j. Hence, there are $\Theta(n^2)$ subproblems. For evaluating $C(i, j)$, we attempt to compose it using $C(i, k)$ and $C(k+1, j)$ for all $i \leq k < j$. This implies that there are $\Theta(n^3)$ edges. Also note that even though every subproblem corresponds to a binary tree representing the multiplication order, the DAG of subproblems is not a tree, because subproblems are used again and again to compose larger subproblems.

  - The algorithm takes time $\Theta(n^3)$

- Independent set on trees

  F A subproblem corresponds to a subtree rooted at a chosen vertex.

  A For solving a subproblem, it is necessary to iterate over all children and grandchildren of the vertex for which the subproblem is calculated. However, every subproblem is only referenced at most twice, once as a child and once as a grandchild during the entire execution of the algorithm, hence the algorithm has running time $\Theta(|V|)$.

Grading: 2.5 points each problem. For each problem, 2.5 points were awarded for the correct solution and 1 point if the correct solution could be derived from the given solution by one deletion or insertion. 0 points otherwise.

Problem 3 *(30 points)*

True or false? Circle the right answer. No justification is needed. No points will be subtracted for wrong answers, so it is to your best interest to guess all you want.

T **F** The recurrence $T(n) = 3T(\frac{n}{3}) + n^2$ has solution $\Theta(n^2 \log n)$.

The solution is $\Theta(n^2)$ (Master Theorem)

T **F** In a weighted graph, a node has exactly three edges adjacent to it, with weights 2, 3 and 4, The minimum spanning tree must contain the edge with weight 2.

To see this, consider a cut $(\{v\}, V \setminus \{v\})$ where v is the node with the three adjacent edges. If the minimum spanning tree did not contain the edge with weight 2 we could substitute this edge with the edge with weight 2 to get a better minimum spanning tree, which is a contradiction.

T **F** If a directed graph has at least two nodes and its depth-first search has no back edges, then the graph has at least two strongly connected components.

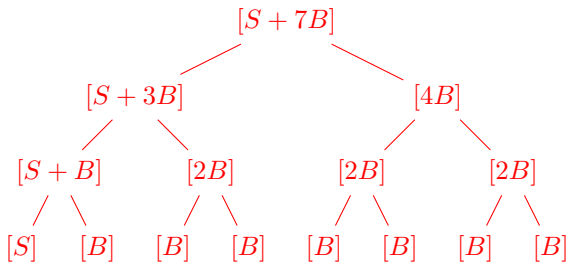
Without loss of generality, assume that we start the DFS traversal at node v and that u is another node in the graph. Either there is a path from v to u or there is none. In the former case, we can conclude that v is in different SCCs because otherwise there would be a back edge. In the latter case, we know that they are in different SCCs because there is no path form v to u.

T **F** In a dynamic programming algorithm, the running time is always proportional to the number of nodes in the dag of subproblems.

Though this is true for many problems, this is not always the case. As an example, consider the example of longest increasing subsequences in the textbook. A simple analysis would yield that it runs in time $O(|V||E|)$, a more careful analysis which takes into account that each edge is only considered once over the entire execution of the algorithm gives the better time bound $O(|E|)$.

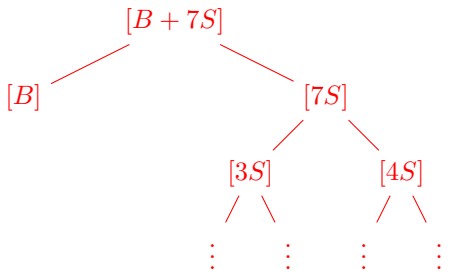
T **F** In Huffman encoding there are 8 symbols, and seven of them have the same frequency, while the eighth frequency is different, **smaller** than the other. Then all leaves must be at the same depth.

Assume that S is the smaller frequency, and that B is the larger frequency. You will get the tree



T **F** In Huffman encoding there are 8 symbols, and seven of them have the same frequency while the eighth frequency is different, **larger** than the other. Then all leaves must be at the same depth.

Assume again that S is the smaller frequency, and that B is the larger frequency. As a counterexample, if $B > 4S$ you would get the tree



T **F** Huffman encoding is the method of choice for compressing English text.

There are other compression schemes such as the Lempel-Ziv-Weich algorithm which is widely used to compress English text.

T **F** A set of Horn clauses, in which there is no clause with only one literal, is always satisfiable.

To see why this is true, assign false to all variables. This is a satisfying assignment, because all pure negative clauses are trivially fulfilled and no implications are triggered as we don't have singleton implications and all other implications must have at least a positive variable as condition.

T **F** The greedy algorithm for set cover produces a solution that is always at most a constant factor times the optimum.

The approximation factor is $\ln(n)$ which is not a constant factor.

T **F** Simplex is a polynomial-time algorithm.

There are examples for which it iterates over an exponential number of vertices before finding an optimal assignment.

T **F** Every iteration of simplex in a linear program with m constraints and n variables can be done in $O((m+n)^3)$ time.

Every iteration of the simplex iteration can be done in time $O(mn)$.

T **F** Every iteration of simplex in a linear program with m constraints and n variables can be done in $O((m+n)^2)$ time.

Every iteration of the simplex iteration can be done in time $O(mn)$.

T **F** There is a polynomial-time algorithm for linear programming.

Both the ellipsoid algorithm and the interior point method are polynomial time algorithms. Note that this does not imply that they have necessarily better runtime behavior in practice.

T **F** There is a reduction from finding the value of a zero-sum game to max flow.

There is a reduction from finding the value of a zero-sum game to linear programming but not to max flow.

T **F** The dual of every linear program always has a feasible solution.

As counterexample, consider the following linear program and its dual

primal	dual
$\max\{x\}$	$\min\{-y\}$
$-x \leq -1$	$-y \geq 1$
$x \geq 0$	$y \geq 0$

Note that the feasible region of the dual is empty.

Grading: 2 points each problem. For each problem, 2 points were awarded for the correct solution and 0 points for an incorrect or no solution.

Problem 4 *(25 points)*

(a) You are given a directed graph (V, E), and two vertices $u, v \in V$ such that $(u, v) \notin E$. You want to find the smallest subset $S \subseteq V - \{u, v\}$ (that is, the smallest set of nodes excluding u and v) such that all paths from u to v go through one of the vertices of S. Describe a reduction from this problem to max flow.

Solution:

We will produce a flow network which models removing vertices instead of edges by splitting each original node w (other than u and v) into w_1 that is the head of the in-arcs and the another w_2 that is the tail of the out-arcs, and add a capacity 1 arc between w_1 and w_2. We make the original capacities be $|V|$.

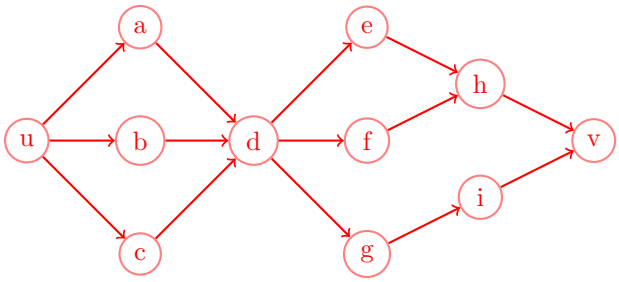
Any u to v path in the original graph corresponds to a path in the new graph and vice versa. Any set S for the original problem there is a set of edges of the form (w_1, w_2) for $w \in S$ which intersects all u to v paths in the new network and for any u-v cut of total capacity less than $|V|$ corresponds to a set of nodes S required in the original problem.

Thus, the maximum flow from u to v in the new graph has the same value as the minimum set S that intersects all u to v paths in the original graph.

Comment.

Many people ran some version of maximum flow on the original network (or various modifications.) Some tried to modify the edges that were output into a set of vertices.

The example below shows that the edges in the minimum edge cut do not really give much of a guide to the set S.



The minimum sized set S consists of node d. The edges adjacent to v are the minimum cut and do not have much to do with the minimum node cut.

(b) Suppose that you want to find the smallest set of nodes of a strongly connected directed graph whose deletion renders the graph not strongly connected. Show how you can do this by running a max-flow algorithm $O(|V|)$ times.

Intended Solution:

From an arbitrary node u, use part a to find the minimum cut to and from each other node v. Output the minimum sized set.

Removing this set leaves a graph that is no longer strongly connected as there are no paths from u to v nor any from v to u.

It is minimal in the case that u is not in the optimal solution for this problem, as u must be in a difference SCC than some node v, and we search over all possibilities.

What if u is a node in the optimal solution? This problem makes us have to search over all possibilities to for u, resulting in a $|V|^2$ algorithm. We missed this case! Thus, the problem does not have a solution as state. Thus, we graded this answer very generously.

Comment. One common method for solution though was to ensure that at least one node was not in a cycle using part (a). This one also does not work as the final graph with nodes in the optimal solution removed may consist of disjoint SCCS with more than one node. That is, every node is in a cycle in the resulting graph.

(c) Argue that the algorithm in (c) above, where you solve max flow by the augmenting path algorithm, runs in time $O(|V|^2|E|)$.

Solution. Observe that only $|V|$ augmenting paths need be computed in applying Ford Fulkerson to the instance in part (a) before determining that there is no set S. Since each augmenting path takes $O(|E|)$ time, the running time of each application of part (a) is $O(|V||E|)$. Since part (a) is invoked $|V|$ times the total time becomes $O(|V|^2|E|)$.

Problem 5 *(25 Points)*

You are working for a construction company, and you are in charge of scheduling the work of a crew for the next n weeks. You have many more customers ("jobs") than you can accommodate. The i-th job is described by a triple (d_j, t_j, p_j), where d_j is the deadline (the week by the end of which the job must be done), $t_j > 0$ is the number of consecutive weeks required for the job (typically between one and three, but occasionally larger), and $p_j > 0$ is the company's profit from this job.

It so happens that for every one of the n weeks in your scheduling horizon, you have exactly one job with a deadline that week. That is, you can assume that the jobs are $(1, t_1, p_1), (2, t_2, p_2), (3, t_3, p_3), \ldots, (n, t_n, p_n)$. Also assume that $t_j \leq j$ (so any job could be scheduled if you start it right now).

You want to select a set of jobs and schedule them so that the total profit of the company is as large as possible, and

- each scheduled job (j, t_j, p_j) is assigned t_j consecutive weeks no later than its deadline j;
- no two scheduled jobs overlap.

(a) **(5 points)** Argue that if there is a way to schedule a set of jobs for the first $i \leq n$ weeks, and if one of the scheduled jobs has deadline i, then the same jobs can be scheduled so that the job with deadline i ends precisely at week i. (Assume that you have a schedule for the first i weeks and the job with deadline i finishes earlier than week i. Rearrange.)

proof: Suppose that you have a feasible schedule and the job i finishes before week i. Then rearrange the same jobs as follows: Those scheduled before i, stay put. Job i occupies the slots $i-t_i+1, i-t_i+2, \ldots, i-1, i$. Finally, all other jobs are pushed t_i weeks forward, Since no job is pushed ahead in time, except for i that now finishes exactly at week i, all deadlines are satisfied, and the proof is complete.

(b) Based on (a), you write a dynamic programming algorithm for finding the best schedule, as follows (fill the blanks (...) below):

For each week $i = 0, 1, 2, \ldots, n$, define $P[i]$ to be the maximum total profit ... that can be obtained by scheduling some of the jobs from $1, 2, \ldots i$ so that the deadlines of the scheduled jobs are respected and all jobs finish exactly at their deadline.

$P[0] = \ldots 0$

For $i = 1, 2, \ldots n : P[i] = \max\{\ldots P[i-1], P[i-t_i] + p_i\}$ (that is, either you do not schedule job i, or you do).

(c) What is the running time of your algorithm? Justify briefly.

$O(n)$, because there are n subproblems, and each takes constant time to compute.

PS: This is fine, except that **it is wrong! We goofed!** This answers a different question, "what is the highest profit if each job scheduled must finish at exactly its deadline?" And this is a different, more restricted problem, yielding smaller profit: To see why, imagine you have three jobs: $(1, 1, 1), (2, 1, 10), (3.2.10)$. Then the optimum schedule includes jobs 2 and 3 (profit 20), while the algorithm return 1 and 3 (profit 11).

Sorry about that! We gave everybody 20 points for parts (b) and (c).