

CS570
Analysis of Algorithms
Fall 2016
Exam I

Name: _____

Student ID: _____

Email Address: _____

_____ **Check if DEN Student**

	Maximum	Received
Problem 1	20	
Problem 2	10	
Problem 3	15	
Problem 4	15	
Problem 5	10	
Problem 6	15	
Problem 7	15	
Total	100	

Instructions:

1. This is a 2-hr exam. Closed book and notes
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.

1) 20 pts

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

[**TRUE**/]

If G is a directed acyclic graph, then G must have a node with no incoming edges.

[**TRUE**/]

Inserting into a binomial heap of n elements has O(1) amortized cost.

[**FALSE**]

In an undirected, connected, weighted graph with at least three vertices and unique edge weights, the heaviest edge in the graph cannot be in any minimum spanning tree.

[**TRUE**/]

If $f=O(g)$ and $g=O(h)$, then $f=O(h)$.

[**FALSE**]

The array [100, 93, 83, 90, 79, 84, 81] corresponds to a binary max-heap.

[**TRUE**/]

Given a graph, suppose we have calculated the shortest path from a source to all other vertices. If we modify the graph such that weights of all edges are doubled, then the shortest path remains the same, only the total weight of the path changes.

[**TRUE**/]

Let $d(u, v)$ denote the minimum distance from node u to node v in a weighted graph G. If $d(s, u) + d(u, t) = d(s, t)$, then u is on at least one shortest path from s to t. (All edge weights in G are positive.)

[**TRUE**/]

Height of a binary heap is $O(n)$

[**FALSE**]

The divide and conquer algorithm to solve the closest pair of points in 2D runs in $O(n \log n)$. But if the two lists of points, one sorted by X-coordinate and the other sorted by Y-coordinate are given to us as input, the rest of the algorithm (skipping the sorting steps) runs in $O(n)$ time.

[**TRUE**/].

The solution of the recurrence $T(n) = 18T(n/3) + O(n^3)$ is $T(n) = O(n^3)$.

Grading:

Each correct answer has 2 points. Wrong answers don't have negative points.

2) 10 pts

Prove or disprove the following statement:

For a given stable matching problem, if m and w appear as a pair when men propose and they also appear as a pair when women propose, then m and w must be paired in all possible stable matchings

Solution:

Let S denote the stable matching obtained from the version where men propose and let S_0 be the stable matching obtained from the version where women propose. From (page 11, statement 1.8 in the text), in S , every woman is paired with her worst valid partner. Applying (page 10, statement 1.7) by symmetry to the version of Gale-Shapley where women propose, it follows that in S_0 , every woman is paired with her best valid partner. Since in both the matchings S and S_0 , w is paired with m, it follows that for w the best valid partner and the worst valid partner are the same. This implies that w has a unique valid partner (which is m), which implies that m and w are paired together in all the stable matchings.

3) 15 pts

Let $G = (V, E)$ be a connected undirected graph with edge-weight function w , and assume all edge weights are distinct. Consider a cycle $v_1, v_2, \dots, v_k, v_{k+1} \geq$ in G , where $v_{k+1} = v_1$, and let (v_i, v_{i+1}) be the edge in the cycle with the largest edge weight. Prove that (v_i, v_{i+1}) does not belong to the minimum spanning tree T of G .

Solution:

Proof by contradiction.

Assume that (v_i, v_{i+1}) does belong to the minimum spanning tree T .

Removing (v_i, v_{i+1}) from T , divides T into two connected components A and B , where some nodes of the given cycle are in A and some are in B .

For any cycle, at least two edges must cross this cut, and therefore there is some other edge (v_j, v_{j+1}) on the cycle, such that adding this edge connects A and B again and creates another spanning tree T' . Since the weight of (v_j, v_{j+1}) is less than (v_i, v_{i+1}) , the weight of T' is less than T and T cannot be a minimum spanning tree.

Contradiction.

Therefore, (v_i, v_{i+1}) doesn't belong to any minimum spanning tree T of G .

4) 15 pts

Each of the following problems suggests a greedy algorithm for a specified task. Prove that the greedy algorithm is optimal, or give a counterexample to show that it is not.

(a) **Problem:** You are given a set of n jobs. Job i starts at a fixed time s_i , ends at a fixed time e_i , and results in a profit q_i . Only one job may run at a time. The goal is to choose a subset of the available jobs to maximize total profit.

Algorithm: First, sort the jobs so that $q_1 \geq q_2 \geq \dots \geq q_n$. Then, try to add each job to the schedule in turn. If job i does not conflict with any jobs schedule so far, add it to the schedule; otherwise, discard it.

This algorithm is not optimal. Consider three jobs with starts $\{1, 1, 3\}$, ends $\{4, 2, 4\}$, and profits $\{3, 2, 2\}$.

Grading:

Part (a) has 5 points. A correct counter example gets all the 5 points. Any other answer (a wrong counter example or trying to prove the optimality of the algorithm) has 0 points.

(b) **Problem:** You are given a set of n jobs, each of which runs in unit time. Job i has an integer-valued deadline time $d_i \geq 0$ and a real-valued penalty $p_i \geq 0$. Jobs may be scheduled to start at any integer time (0, 1, 2, etc), and only one job may run at a time. If job i completes at or before time d_i , then it incurs no penalty; otherwise, it incurs penalty p_i . The goal is to schedule all jobs so as to minimize the total penalty incurred.

Algorithm: Define slot k in the schedule to run from time $k - 1$ to time k . First, sort the jobs so that $p_1 \geq p_2 \geq \dots \geq p_n$. Then, add each job to the schedule in turn. When adding job i , if any time slot between 1 and d_i is available, then schedule i in the latest such slot. Otherwise, schedule job i in the latest available slot $\leq n$.

Solution:

We prove via exchange argument. Let's start with an arbitrary optimal solution O_0 , and prove that we can change this solution to greedy solution without increasing total penalty.

* (Base:) Consider location of job_1, i.e. the job with largest late penalty p_1 . Suppose greedy solution places job_1 at slot_x, and an optimal solution O_0 places job_1 at slot_y, and places job_z at slot_x. job_z is from {job_2, ... job_n}

1. If slot_x==slot_y, job_1==job_z, there exist an optimal solution which places job_1 the same slot to greedy solution, and let's fix this position.

2. If slot_x!=slot_y, (must discuss both case)

2-a. case1, slot_y>slot_x: in optimal solution O_0 , job_1 caused penalty p_1 , and job_z may cause penalty 0 or p_z ($p_z <= p_1$). If we switch job_1 and job_z, the total penalty will decrease by p_1 or p_1-p_z . In any case, the total penalty does not increase, so we can swap job_1 and job_z in O_0 , and the resultant schedule is still optimal.

2-b. case2, slot_y<slot_x: if we swap job_1 and job_z, (1) because job_z is moved forward, it won't cause more penalty, (2) because job_1 is moved to the latest slot before d_1 , a position with 0 penalty, it won't cause more penalty either. Here we assumed $d_1>0$. If $d_1=0$, then job_1 will cause p_1 penalty in any solution, so in this special case, swapping job_1 and job_z in optimal solution won't increase penalty.

From above discussion, we can swap job_1 and job_z in solution O_0 and the result is still an optimal solution. Let's record new optimal solution as O_1 .

* (Inductive:) Suppose we have created an optimal solution O_k through swapping, which place {job_1... job_k} in the same position to greedy solution. Use same notation to base case: suppose greedy solution places job_(k+1) at slot_x, and O_k places it at slot_y, and places job_z at slot_x. Note that because O_k places {job_1... job_k} the same slots to greedy solution, job_z must be one of {job_(k+2), ... job_n}.

1. If slot_x==slot_y, job_(k+1)==job_z, same to base case.

2. If slot_x!=slot_y, (must discuss both case, and must clarify the difference from base case in 2-b)

2-a. case1, same to base case.

2-b. case2, slot_y<slot_x: if we swap job_(k+1) and job_z, (1) because job_z is moved forward, it won't cause more penalty, (2) moving job_(k+1) from slot_y to slot_x won't increase penalty: If slot_x <= b_(k+1), job_(k+1) still has 0 penalty; if slot_x > b_(k+1), according to greedy algorithm, the slots from 1 to b_(k+1) must be filled by jobs from {job_1, ... job_k}, in both greedy solution and O_k , so slot_y > b_(k+1) too, and job_(k+1) causes p_k penalty before and after swapping.

Thus we can conclude for any arbitrary $k \geq 1$, starting with O_k , we can swap job_(k+1) and job_z and the result is still an optimal solution $O_{(k+1)}$.

* (conclusion:) by induction, we can move every job from an optimal solution in the order of {job_1, job_2, ... job_n} to the corresponding position in greedy solution one by one, via swapping, without increasing total penalty. Thus greedy solution is an optimal solution.

Here is a list of typical mistakes in 4b). Please do not argue for more grades from 4b) unless you have strong evidence you did not make above mistakes but penalized for these reasons.

1. Some students try to prove that in greedy solution, if we swap any pair of jobs in the greedy solution the penalty will not decrease. This is incorrect because optimal solution is not greedy solution add one swapping. Some argues that a sequence of swapping can change greedy to global optimal, but after one swapping, the greedy solution is not long a greedy solution, so the property " if we swap any pair of jobs in the greedy solution the penalty will not decrease" no longer holds.
 1. Note: Exchange argument swaps pairs in optimal solution to match greedy solution. It does not swap greedy solution.
2. Similar to 1, some students think that there must exist one pair of job (i,j) s.t. the position of them in greedy and optimal solution is exchangeable. This is incorrect. e.g. order of $(1,2,3)$ and order of $(2,3,1)$.
3. Some students confused optimal and greedy solution, and attempt to guess property of ordering in optimal solution. e.g. they attempted to declare that two jobs cannot be arranged in certain way if penalty and deadlines of these two jobs satisfies some condition. This is invalid. We know nothing about optimal solution except its late penalty is smallest among all factor(n) combinations.
4. Some students explained the intuition of positioning next job in greedy algorithm but did not explain the optimality.
 1. Some students qualitatively analyze why we start from job with large penalty, and why put the next job in "latest" available slot. e.g. starting with hard (with higher late penalty), we use the best slot resourced to avoid suffering from large penalties; using the "latest" available position is friendly for future jobs, maximizing the probability they have on-time slots.
 2. Many students proved in this way: "<1> provided positions of job with 1st, 2nd, ... k-th late penalties, the position of next job (it will have $p_{(k+1)}$ penalty if late, let's call it $(k+1)$ -th job) assigned by greedy algorithm is optimal because it will minimize the penalty from adding this new job. <2> by induction, claim 1 is true for every k, thus the final solution is optimal". However, they did not understand the difference between "local/greedy optimality" and "global optimality". We can only declare that, "if we are already fixed position of 1st ~ k-th jobs, and if we only adding one job", the position of job $(k+1)$ according to greedy algorithm is best. They cannot declare that the fixed position of job $1 \sim k$ are truly optimal (optimal is defined on all n jobs), and they cannot declare the final output of n-jobs' ordering is optimal. In short, this greedy algorithm guarantees one step is best only if we "move only one step, and evaluate only this one step".
5. Many students use induction. They made mistakes in these two ways:
 1. One inductive proof in this way: Provided the total number of jobs is fixed n, (base:) starting with placing first job (which will suffer from most late-penalty if not on-time) into one of n slots, declaring the greedy algorithm provides "optimal first step", then (induction:) suppose we have k jobs have been placed according to greedy algorithm, and their position are optimal, then the position of $(k+1)$ th according to greedy algorithm is optimal. There are at least two mistakes:

- i. Mistake in basis step: We cannot declare position of first job as optimal. Optimality is defined over the order of all n jobs. Knowing position of 1st job, not knowing positions of other (N-1) jobs, we cannot declare if 1st job is optimal or not.
 - ii. Induction error: explained in item 3-2.
6. Another use inductive proof in this way: (base:) suppose we have totally 1 job, then obvious greedy algorithm give optimal solution since only one slot available. Then (induction) suppose for task of arranging n=k jobs, greedy algorithm is optimal, then if there are totally n=(k+1) jobs, greedy algorithm also give optimal solution. In the inductive steps, they declare that the first k of n=k+1 jobs are assigned slots using greedy algorithm, and we have known position of these k jobs are optimal according to inductive assumption.
1. The declaration that "the first k of n=k+1 jobs are assigned slots using greedy algorithm" is incorrect. Placing k jobs and placing k of k+1 jobs are different problems. Actually placing k of k+1 jobs and place k+1 jobs are same problem, because the position of last job provided position of first k jobs is determined.
 2. Some students mixed the basis condition and inductive arguments in above two cases. This is even worse.
7. Key steps and important statements are not explained correctly: Many students do not provide strong evidence when they declare two jobs (either from optimal solution or from greedy solution) can or cannot be swapped. For example,
1. They did not explore all possibilities of properties of job pairs.
 2. The calculation of change of penalty after swapping is incorrect.
8. Some students do not know how to prove optimality (or for some other reasons), and they write many statements or random sentence, random expressions without clean or correct logics.

For any of case 1~5, 3 points will be penalized. If most statements and reasoning steps are clean 1~2 points may be rewarded. For case 6, up to 4 points may be penalized. Usually case 6 together with other mistakes from case 1~5 will cause no more than 6 points penalty. Case 7 itself could cause up to 7 points penalty.

5) 10 pts

Arrange the following functions in increasing order of growth rate with $g(n)$ following $f(n)$ in your list if and only if $f(n) = O(g(n))$

$$n^2 \log n, 2^n, 2^{2n}, n^{\log n}, n^2$$

Solution: $n^2, n^2 \log n, n^{\log n}, 2^n, 2^{2n}$

Grading:

For 5 functions, there are a total of 10 pairwise comparisons. For any wrong pairwise comparison, 1 point is deducted. For example in this answer:

$$n^2, n^{\log n}, n^2 \log n, 2^n, 2^{2n}$$

there's only one incorrect pairwise comparison ($n^{\log n}$ and $n^2 \log n$) so 1 point will be deducted.
However if the answer is:

$$n^{\log n}, n^2, n^2 \log n, 2^n, 2^{2n}$$

there are two incorrect pairwise comparisons (n^2 and $n^{\log n}$ in addition to $n^{\log n}$ and $n^2 \log n$) so 2 points will be deducted.

6) 15 pts

Suppose we perform a sequence of n operations on a data structure in which the i^{th} operation costs i if i is an exact power of 2. If it is not, the i^{th} operation costs 1. Use aggregate analysis to determine the amortized cost per operation.

Solution:

In a sequence of n operations there are $\log n + 1$ exact powers of 2, namely $1, 2, 4, \dots, 2^{\log n}$. Thus the total cost of all operations is $T(n) \leq 2n + n = 3n$, which means $O(3)$ amortized cost per operation

7) 15 pts

A set of cities V is connected by a network of roads $G(V,E)$. The length of road $e \in E$ is weight of that edge. There is a proposal to add one new road to this network, and a set C contains candidate pairs of cities between which the new road may be built. Each of these potential roads has an associated length. Suppose you want to choose the road that would result in maximum decrease in the driving distance between city s and city t . Give an efficient algorithm for solving this problem, analyze its complexity, and express its complexity as a function of $|V|$, $|E|$, and $|C|$.

For full credit, your algorithm should have a run time of $O(|E| \log|V| + |C|)$.

Partial credit (7 pts) will be granted to efficient solutions that don't meet the run time requirement given above.

Solution:

Algorithm:

1. Run Dijkstra algorithm from s to calculate shortest distances from s to all other cities
2. Run Dijkstra algorithm from t to calculate shortest distance from t to all other cities
3. For every candidate pair of cities $\{u,v\}$, the shortest path distance between s and t which covers road $u \rightarrow v$ is $\min(\text{dist}(s,u) + \text{dist}(t,v) + \text{length}(u,v), \text{dist}(s,v) + \text{dist}(s,u) + \text{length}(u,v))$.
4. Choose the shortest distances from loop-3.
 - If the selected distance is longer than original $\text{dist}(s,t)$, any candidate road cannot decrease distance between s and t .
 - Else, choose the $\{u,v\}$ pair that produces shortest new distance. In the case of tie, choose one arbitrarily.

Complexity: Complexity of running Dijkstra's algorithm is $\Theta(2 \times |E| \log|V|)$, the complexity of running step-3 is $\Theta(2 \times |C|)$, thus total complexity is $\Theta(|E| \log|V| + |C|)$.

Additional Space

Additional Space

CS570
Analysis of Algorithms
Fall 2016
Exam II

Name: _____

Student ID: _____

Email Address: _____

_____ **Check if DEN Student**

	Maximum	Received
Problem 1	20	
Problem 2	16	
Problem 3	16	
Problem 4	16	
Problem 5	16	
Problem 6	16	
Total	100	

Instructions:

1. This is a 2-hr exam. Closed book and notes
 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.
- 1) 20 pts
Mark the following statements as **TRUE** or **FALSE**. No need to provide any justification.

[FALSE] The run-time is $O(nW)$ so it's pseudopolynomial

0-1 knapsack problem can be solved using dynamic programming in polynomial time

[FALSE] If neither of the two nodes are reachable by the negative cycle, the distance between them is calculated accurately.

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.

[TRUE] Value of max flow = capacity of min-cut (iterate over all edges of the min-cut)

Given the min-cut, we can find the value of max flow in $O(|E|)$.

[TRUE] Set the penalties to 0 and reward of 1 for every match

The sequence alignment algorithm can be used to find the longest common subsequence between two given sequences.

[FALSE] You only have to do it once.

In dynamic programming you must calculate the optimal value of a sub-problem twice, once during the bottom up pass and once during the top down pass.

[TRUE] For any cut which is not a min-cut this is True.

Maximum value of an s-t flow could be less than the capacity of a given s-t cut in a flow network.

[FALSE] Consider a graph with 4 nodes s, a, b and t and $E = \{ (s,a), (a,t), (s,b), (b,t) \}$. The increase in the flow would be $f + 2$.

Suppose the maximum (s, t)-flow of some graph has value f. Now we increase the capacity of every edge by 1. Then the maximum (s, t)-flow in this modified graph will have value at most $f + 1$.

[FALSE] The Orlin algorithm runs in $O(mn)$

There are no known polynomial-time algorithms to solve maximum flow.

[FALSE] Edges having capacity 1 does not mean $|f| = 1$ so the Ford-Fulkerson is still going to be pseudo-polynomial.

If all edges in a graph have capacity 1, then Ford-Fulkerson runs in linear time.

[FALSE] Choice of augmenting path can still affect the number of iterations of the inner loop.

In the scaled version of the Ford Fulkerson algorithm, choice of augmenting paths cannot affect the number of iterations.

2) 16 pts

Given $N(G(V, E), s, t, c)$, a flow network with source s , sink t , and positive edge capacities c , we are asked to reduce the max flow by removing at most k edges. Prove or disprove the following statement:

Maximum reduction in flow can be achieved by finding a min cut in N , creating a sorted list of the edges going out of this cut in decreasing order and then removing the top k edges in this list.

The Statement is False. (8 points)

Proof (8 points). This requires a counter example where deleting the edges from the min-cut would not result in maximum reduction in max flow.

Many students tried to prove this statement. The statement is true only if all the edges have equal capacities. If the proof was correct based on that assumption, you were given 8 points.

3) 16 pts

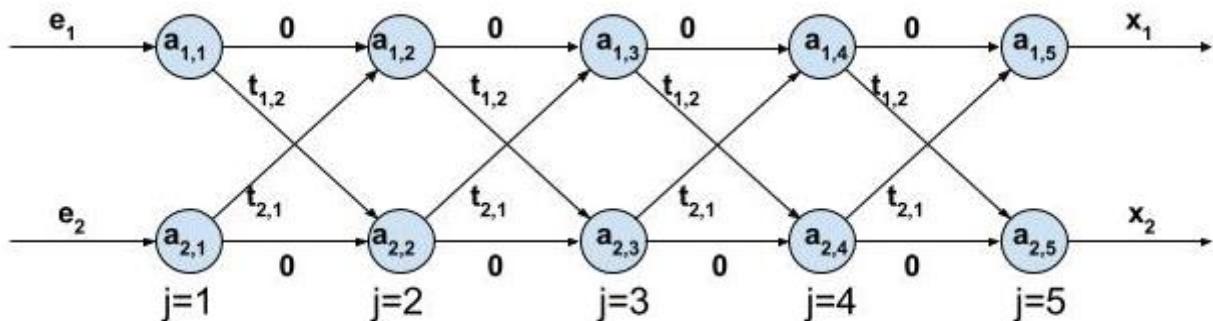
A car factory has k assembly lines, each with n stations. A station is denoted by $S_{i,j}$ where i indicates that the station is on the i -th assembly line ($0 < i \leq k$), and j indicates the station is the j -th station along the assembly line ($0 < j \leq n$). Each station is dedicated to some sort of work like engine fitting, body fitting, painting and so on. So, a car chassis must pass through each of the n stations in that order before exiting the factory. The time taken per station is denoted by $a_{i,j}$. Parallel stations of the k assembly lines perform the same task. After the car passes through station $S_{i,j}$, it will continue to station $S_{i,j+1}$ unless we decide to transfer it to another line. Continuing on the same line incurs no extra cost, but transferring from line x at station $j-1$ to line y at station j takes time $t_{x,y}$. Each assembly line takes an entry time e_i and exit time x_i which may be different for each of these k lines. Give an algorithm for computing the minimum time it will take to build a car chassis.

Below figure provides one example to explain the problem. The example shows $k=2$ assembly lines and 5 stations per line. To illustrate how to evaluate the cost, let's consider two cases: If we always use assembly line 1, the time cost will be:

$$e_1 + a_{1,1} + a_{1,2} + a_{1,3} + a_{1,4} + a_{1,5} + x_1.$$

If we use stations $s_{1,1} \rightarrow s_{2,2} \rightarrow s_{1,3} \rightarrow s_{2,4} \rightarrow s_{2,5}$, the time cost will be

$$e_1 + a_{1,1} + t_{1,2} + a_{2,2} + t_{2,1} + a_{1,3} + t_{1,2} + a_{2,4} + a_{2,5} + x_2.$$



- a) Recursively define the value of an optimal solution (recurrence formula) (6 pts)

Dynamic programming Solution

Subproblem: $\text{minCost}(i,j)$, if we use assembly line- i at station- j , the minimal cost so far.

Initialization: $\text{minCost}(1,1) = e_1 + a_{1,1}$, $\text{minCost}(2,1) = e_2 + a_{2,1}$, ... $\text{minCost}(k,1) = e_k + a_{k,1}$ (1 point)

Recurrence relation: $\text{minCost}(i,j) = \min_{1 \leq l \leq k} \text{minCost}(l,j-1) + t_{l,i} + a_{i,j}$ (4 points)

($t_{i,i} = 0$ in this formulation)

Final cost: $\text{minCost}(i, \text{exit}) = \text{minCost}(i, n) + x_i$ (1 point)

Attention: Some students may assume that the car can only pass to the station (i, j) from adjacent lines $i-1$ and $i+1$. In this case, you will have 3 points off.

- b) Describe (using pseudocode) how the value of an optimal solution is obtained using iteration. Make sure you include any initialization required. (6 pts)

```
// initialization, boundary condition
For i=1:k
    minCost[i][1] = e[i]+a[i][1]           (1 point)

// recurrence
For t=2:n
    For i=1:k
        minCost[i][t] = minCost[1][t-1] + t[1][i]+a[i][t]
        for j=2:k
            minCost[i][t] = min(minCost[j][t-1] + t[j][i]+a[i][t], minCost[i][t]) (4 points)

for i=1:k
    minCost[i][n] = minCost[i][n]+x[i]

// final solution
return min( minCost[1][n], minCost[2][n], ... minCost[k][n] )           (1 point)
```

Attention: some students may ignore the initialization.

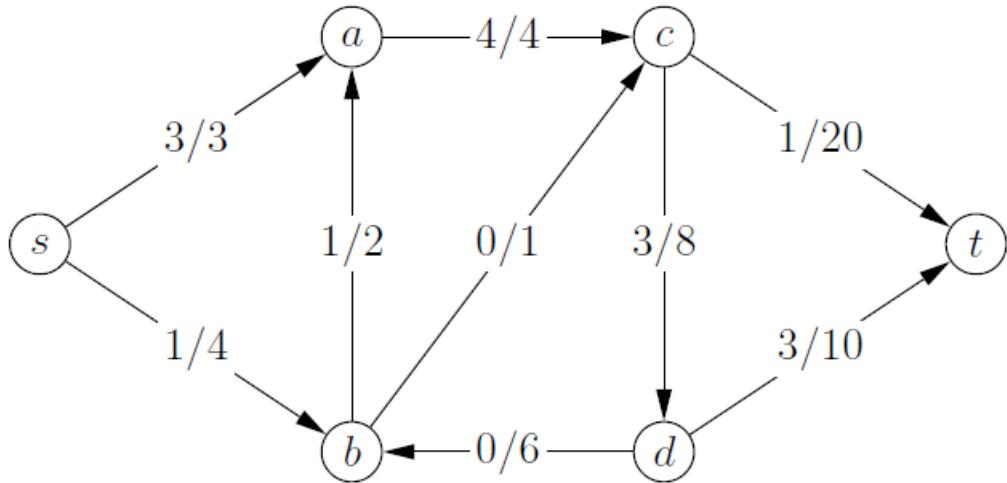
- c) What is the complexity of your solution in part b? (4 pts)

From the recurrence loops, we can tell the complexity is $O(n \times (k^2))$

Attention: If you don't show your work process and your final solution is incorrect, you will have 4 points off.

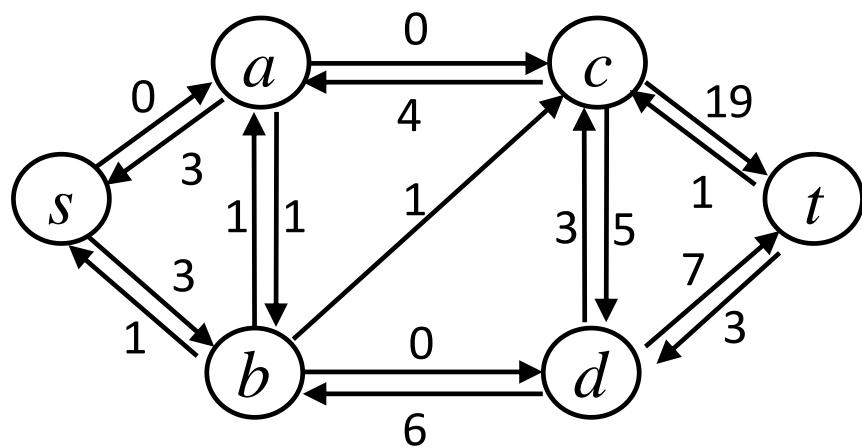
4) 16 pts

You are given a directed graph which after a few iterations of Ford-Fulkerson has the following flow. The labeling of edges indicate flow/capacity:



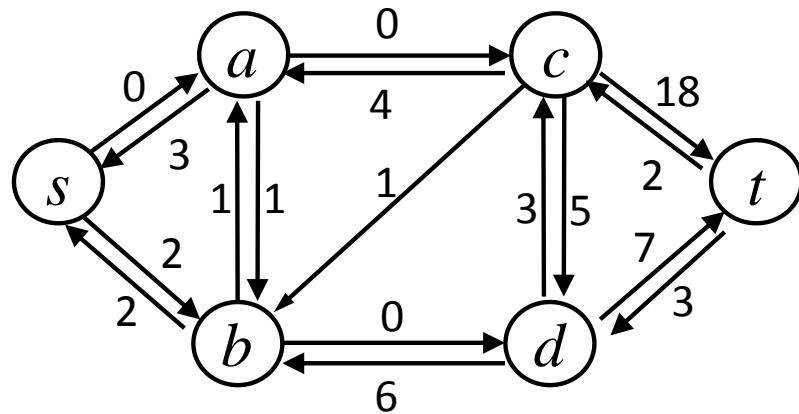
a) Draw the corresponding residual graph. (5 pts)

Each mistake in edge direction or value would cause one-point deduction.



- b) Is this a max flow? If yes, indicate why. If no, find max flow. (6 pts)

No, since there is still path like s-b-c-t that can be augmented with flow 1 (3 points) The max flow is $4+1=5$ that can be obtained from the residual graph shown below (3 points).



- c) What is the min-cut? (5 pts)

Min-cut: $(\{s,a,b\}, \{c,d,t\})$ (4 points). For the min-cut you need to show it on the graph or write it in the form (A,B) where A, B are the two parts of the original graph. The value of min-cut equals to $C_{ac} + C_{bc} = 4 + 1 = 5$. (1 point)

Note that the other cuts like $(\{s,b\}, \{a,c,d,t\})$ with value of 6 etc., have larger values than the min-cut with value of 5 and cannot be considered as min-cut for the network.

5) 16 pts

Given a rod of length n inches and an array of prices that contains prices of all pieces of size smaller than or equal to n . Determine the maximum value obtainable by cutting up the rod and selling the pieces. For example, if the length of the rod is 8 and the values of different pieces are given as in the following table, then the maximum obtainable value is 23 (by cutting the rod into two pieces of lengths 2 and 6). Notation: Price(i) is the price of a rod of length i

Length	1	2	3	4	5	6	7	8
Price	1	5	8	9	10	18	17	20

And if the prices are as given in the following table, then the maximum obtainable value is 24 (by cutting the rod into eight pieces of length 1)

Length	1	2	3	4	5	6	7	8
Price	3	5	8	9	10	19	17	20

a) Recursively define the value of an optimal solution (recurrence formula) (6 pts)

The following four solutions are correct –

Soln 1:

OPT(i): best possible price obtained for a rod of length i

$$\text{OPT}(i) = \max\{p(j) + \text{OPT}(i-j)\} \text{ for all } j \text{ in } \{1..i\}$$

Time complexity: $O(n^2)$

Here, the assumption is that the rod needs to be cut and cannot be left uncut.

Soln 2:

OPT(i): best possible price obtained for a rod of length i

$$\text{OPT}(i) = \max\{\text{OPT}(j) + \text{OPT}(i-j), p[j]\} \text{ for all } j \text{ in } \{1..i\}$$

Time complexity: $O(n^2)$

Here, the assumption is that the rod need not be cut.

Soln 3:

OPT(i, j): best possible price obtained for a rod of length i with j cuts

$$\text{OPT}(i, j) = \max\{\text{OPT}(k, j-1) + p[i-k], \text{OPT}(i, j-1)\} \text{ for all } k \text{ in } \{1..i\}$$

Time complexity: $O(n^3)$

4 points have been deducted for high time complexity.

Many students have used the above definition for OPT (i.e. j cuts as opposed to cuts 1..j) but have not used a third variable k. This is incorrect and marks have been deducted for this since we can have multiple cuts of same length.

Soln 4:

OPT(i, j): best possible price obtained for a rod of length i when the longest cut is no longer than j

$$\text{OPT}(i, j) = \max \{ \text{OPT}(i-j, j) + p[j], \text{OPT}(i, j-1) \}$$

Time complexity: $O(n^2)$

Grading rubric:

(a) Definition of OPT: 2 marks

Recurrence: 3 marks

Final answer: 1 mark

(b) Initialization: 2 marks

Loop indices: 3 marks

Return final answer: 1 marks

(c) No partial marking

Note that no marks have been given for part c if the recurrence formed in part a was incorrect.

b) Describe (using pseudocode) how the value of an optimal solution is obtained using iteration. Make sure you include any initialization required. (6 pts)

Solution 1 can be written using iteration as

```
int cutRod(int price[], int n)
```

```
{
```

```
    int val[n+1];
```

```
    val[0] = 0;
```

```
    int i, j;
```

```
// Build the table val[] in bottom up manner and return the last entry
```

```
// from the table
```

```
for (i = 1; i<=n; i++)
```

```
{
```

```
    int max_val = INT_MIN;
```

```
    for (j = 1; j <= i; j++)
```

```
        max_val = max(max_val, price[j] + val[i-j]);
```

```
    val[i] = max_val;  
}  
  
return val[n];  
}  
// end OR  
  
return cutRod(arr, size)
```

c) What is the complexity of your solution in part b? (4 pts)

$O(n^2)$

6) 16 pts

There are n students in a class. We want to choose a subset of k students as a committee. There has to be m_1 number of freshmen, m_2 number of sophomores, m_3 number of juniors, and m_4 number of seniors in the committee. Each student is from one of k departments, where $k = m_1 + m_2 + m_3 + m_4$. Exactly one student from each department has to be chosen for the committee. We are given a list of students, their home departments, and their class (freshman, sophomore, junior, senior).

Describe an efficient algorithm based on network flow techniques to select who should be on the committee such that the above constraints are all satisfied.

a) Describe how to construct a flow network to solve this problem, including the description of nodes, edges, edge directions and their capacities. (10 pts)

- b) Describe how the solution to the network flow problem you described in part a corresponds to the problem of determining who should be on the committee.
(6 pts)

Solution: Consider the following graph, the source is connected to K departments with capacity one. Each department is connected to a subset of n students, that belongs to the department with capacity one. Each student is connected to the respective group (freshman, sophomore, junior, senior) with capacity one. Each group is connected to the sink with capacity m, for group I. The maximum flow in this problem finds the set of students to be included in the committee. The students' nodes that flow pass through them are shown who should be in the committee

a) The problem has two important constraints :

1. Exactly one student from each department must be selected.
 k nodes and k edges with correct capacity expected. (3 points)
2. Number of selected students from each class must be m_1 for the freshmen, ..
 4 nodes for classes with capacity m_1, m_2, m_3, m_4 expected. (3 points)

For students you can either put some nodes or some edges, But if your graph doesn't give the correct answer you will not get point just for adding n nodes for students. (2 points)

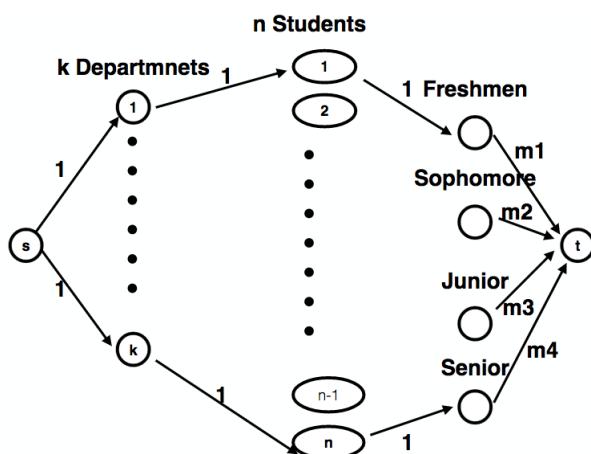
The network flow graph gives the right set of students (2 points)

b) claim: Picking k students for the committee is possible iff The max flow of the given graph in part a is equal to k . (2 points)

Why your graph satisfies the constraint of the problem (2 points)

How the set of students can be determined based on your graph (2 points)

You can either explain these three questions or prove the claim, and you will get 6 points.



Additional Space

Additional Space

CS570
Analysis of Algorithms
Fall 2016
Exam III

Name: _____

Student ID: _____

Email Address: _____

_____ **Check if DEN Student**

	Maximum	Received
Problem 1	20	
Problem 2	18	
Problem 3	18	
Problem 4	18	
Problem 5	18	
Problem 6	8	
Total	100	

Instructions:

1. This is a 2-hr exam. Closed book and notes
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.

1) 20 pts

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

[TRUE] This is the definition of NP.

Every problem in NP has a polynomial time certifier.

[FALSE] Decision problems can be in P.

Every decision problem is in NP-complete.

[TRUE] If 3-SAT reduces to it, it's NP-Hard and we now it's NP so it's NPC.

An NP problem is NP-complete if 3-SAT reduces to it in polynomial time.

[FALSE] It will be $O(m^2 + mn)$ and not linear.

If all edges in a graph have capacity 1, then Ford-Fulkerson runs in linear time.

[FALSE] consider a graph with edges $(a,b) = 4$, $(b,c) = 3$ and $(a, c) = 5$. The shortest path from a to c is 5 but (a, c) is not in the MST.

Let T be a minimum spanning tree of G. Then, for any pair of vertices s and t, the shortest s-t path in G is the path in T.

[TRUE] Master's Theorem

If the running time of a divide-and-conquer algorithm satisfies the recurrence $T(n) = 3 T(n/2) + \Theta(n^2)$, then $T(n) = \Theta(n^2)$.

[FALSE] Consider the problem in the previous item.

In a divide and conquer solution, the sub-problems are disjoint and are of the same size.

[FALSE] Many of them are NP-Complete.

All Integer linear programming problems can be solved in polynomial time.

[TRUE]

If the linear program is feasible and bounded, then there exists an optimal solution.

[FALSE] Each specific operation can be larger than $O(\log n)$.

Suppose we have a data structure where the amortized running time of Insert and Delete is $O(\lg n)$. Then in any sequence of $2n$ calls to Insert and Delete, the worst-case running time for the nth call is $O(\lg n)$.

2) 18 pts

We are given an infinite array where the first n elements contain integers in sorted order and the rest of the elements are filled with ∞ . We are not given the value of n . Describe an algorithm that takes an integer x as input and finds a position in the array containing x , if such a position exists, in $O(\lg n)$ time.

Solution:

- First determine the smallest power of 2 larger than n (8 points)

```
N=1  
while A[i] ≠ ∞  
    N = N*2  
endwhile
```

N will be the smallest power of 2 larger than n

- Use binary search on $A[1..N]$ to find x (8 points)
- Complexity: both components run in $O(\lg n)$ time

Common mistakes:

1. Without finding N , directly do binary search on $A[1..n]$

Note that n is not given. Cases like this get 4 points.

2. Without finding N , directly do binary search on $A[1..2x+1]$

x could be a negative integer; in this case, $2x+1$ is also negative.
Cases like this get 8 points.

3. Time complexity is larger than $O(\log n)$ time. (e.g., linearly scan the array, or build a min heap first)

Cases like this get 0 points

3) 18 pts

For bit strings $A = a_1, \dots, a_m$, $B = b_1, \dots, b_n$ and $C = c_1, \dots, c_{m+n}$, we say that C is an interleaving of A and B if it can be obtained by interleaving the bits in A and B in a way that maintains the left-to-right order of the bits in A and B . For example if $A = 101$ and $B = 01$ then $a_1a_2b_1a_3b_2 = 10011$ is an interleaving of A and B , whereas 11010 is not. Give the most efficient algorithm you can to determine if C is an interleaving of A and B . Analyze the time complexity of your solution as a function of m and n .

Solution: The general form of the subproblem we solve will be: determine if c_1, \dots, c_{i+j} is an interleaving of a_1, \dots, a_i and b_1, \dots, b_j for $0 \leq i \leq |A|$ and $0 \leq j \leq |B|$. Let $c[i, j]$ be true if and only if c_1, \dots, c_{i+j} is an interleaving of a_1, \dots, a_i and b_1, \dots, b_j . We use the convention that if $i = 0$ then $a_i = \Phi$ (the empty string) and if $j = 0$ then $b_j = \Phi$. The subproblem $c[i, j]$ can be recursively defined as shown (where $c[|A|, |B|]$ gives the answer to the optimal problem):

$$c[i, j] = \begin{cases} \text{true,} & \text{if } i = j = 0 \\ \text{false,} & \text{if } a_i \neq c_{i+j} \text{ and } b_j \neq c_{i+j} \\ c[i-1, j], & \text{if } a_i = c_{i+j} \text{ and } b_j \neq c_{i+j} \\ c[i, j-1], & \text{if } a_i \neq c_{i+j} \text{ and } b_j = c_{i+j} \\ c[i-1, j]c[i, j-1], & \text{if } a_i = b_j = c_{i+j} \end{cases}$$

The time complexity is clearly $O(|A||B|)$ since there are $|A|x|B|$ subproblems each of which is solved in constant time. Finally, the $c[i, j]$ matrix can be computed in row major order.

Proof of recurrence not asked for in the problem statement: We now argue this recursive definition is correct. First the case where $i = j = 0$ is when both A and B are empty and then by definition C (which is also empty) is a valid interleaving of A and B . If $a_i \neq c_{i+j}$ and $b_j = c_{i+j}$ then there could only be a valid interleaving in which a_i appears last in the interleaving, and hence $c[i, j]$ is true exactly when c_1, \dots, c_{i+j-1} is a valid interleaving of a_1, \dots, a_{i-1} and b_1, \dots, b_j which is given by $c[i-1, j]$. Similarly, when $a_i \neq c_{i+j}$ and $b_j = c_{i+j}$ then $c[i, j] = c[i-1, j]$.

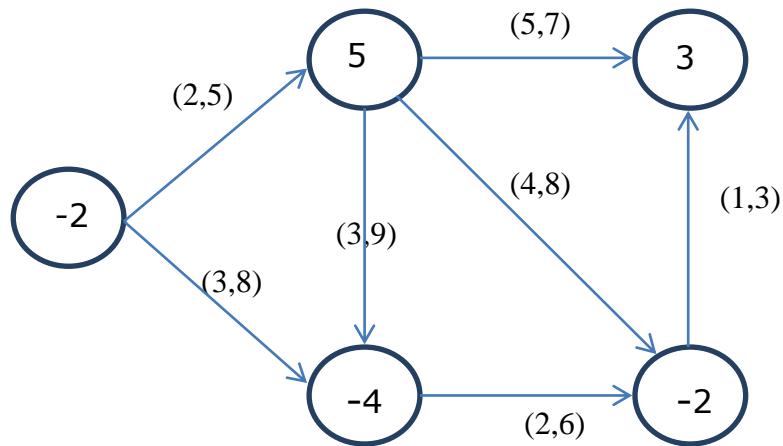
Finally, consider when $a_i = b_j = c_{i+j}$. In this case the interleaving (if it exists) must either end with b_i (in which case $c[i-1, j]$ is true) or must end with a_i (in which case $c[i, j-1]$ is true). Thus returning $c[i-1, j] \vee c[i, j-1]$ gives the correct answer.

Finally, since in all cases the value of $c[i, j]$ comes directly from the answer to one of the subproblems, we have the optimal substructure property.

- Recurrence and definition of OPT (12 pts)
- Algorithm (5 pts)
- Complexity (3 points)

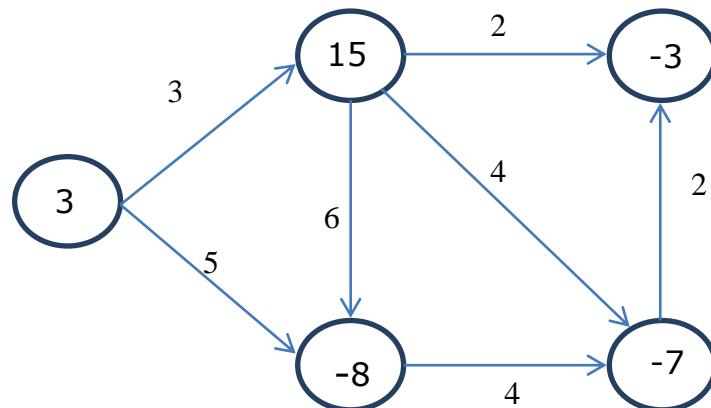
4) 18 pts

In the network 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.

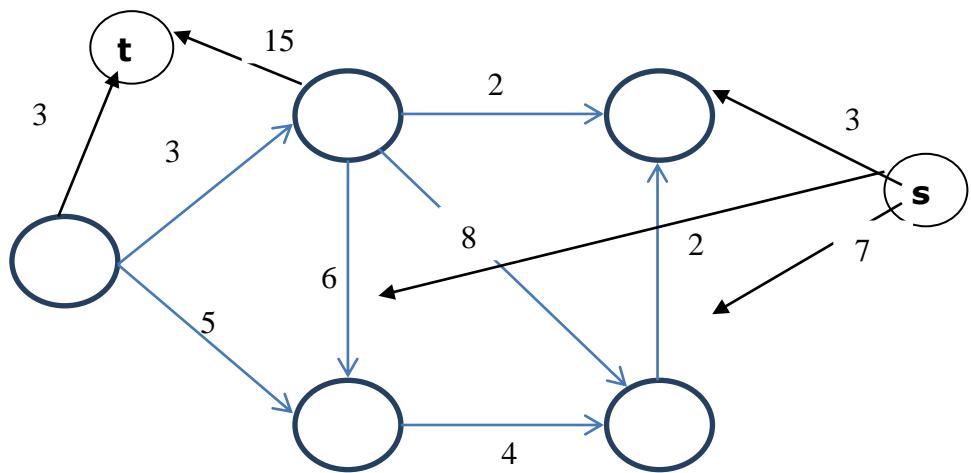


Solution.

Remove lower bounds by changing demands on each vertex



Reduce it to max-flow by adding s and t vertices.



Check if there is a s-t flow $|f|= 18$.

There is no such flow. It follows, no circulation.

6 points - for eliminating lower bounds and converting to problem with only demands

6 points - for eliminating demands and converting to max-flow problem

6 points - Checking if there exists a max-flow of value D

5) 18 pts

The Metric Traveling Salesman Problem (metric-TSP) is a restriction of the Traveling Salesman Problem, defined as follows. Given n cities c_1, \dots, c_n with inter-city distances $d(c_i, c_j)$ that form a metric:

$$d(c_i, c_i) = 0$$

$$d(c_i, c_j) > 0 \text{ for } i \neq j$$

$$d(c_i, c_j) + d(c_j, c_k) \geq d(c_i, c_k) \text{ for } i, j, k \in \{1, \dots, n\}$$

Is there a closed tour that visits each city exactly once and covers a total distance at most C ? Prove that metric-TSP is in NP-complete.

Solution.

1. It's easy to see that is in NP
2. Reduce from HC.

Given a graph $G=(V,E)$ on which we want to solve the HAM-CYCLE problem. We will construct a complete G' with metric d as follows

$$d(u,v) = 0, \text{ if } u = v$$

$$d(u,v) = 1, \text{ if } (u,v) \text{ is in } E$$

$$d(u,v) = 2, \text{ otherwise}$$

Since triangle inequalities hold in G' (edges of any triangle will have sizes of 1 or 2), then $d(c_i, c_j) + d(c_j, c_k) \geq d(c_i, c_k)$ for $i, j, k \in \{1, \dots, n\}$

Let the bound $C = |V|$.

Claim: metric-TSP on G' is solvable if and only if the HC problem on G is solvable.

->) If there is a metric-TSP of length $|V|$, then every distance between successive cities must be 1. Therefore these define as HC on G .

<-) If G has a hamiltonian cycle, then it defines a metric-TSP of size $|V|$.

6) 8 pts

Convert the following linear program

$$\text{maximize } (3x_1 + 8x_2)$$

Subject to

$$x_1 + 4x_2 \leq 20$$

$$x_1 + x_2 \geq 7$$

$$x_1 \geq -1$$

$$x_2 \leq 5$$

to the standard form. You need to show all your steps.

$$x_1 + 1 = z_1 \geq 0$$

$$x_2 - 5 \leq 0 \text{ so } 5 - x_2 = z_2 \geq 0$$

$$x_1 + 4x_2 \leq 0 \rightarrow (z_1 - 1) + 4(5 - z_2) \leq 20 \rightarrow z_1 - 4z_2 \leq 1$$

$$x_1 + x_2 \geq 7 \rightarrow (z_1 - 1) + (5 - z_2) \geq 7 \rightarrow z_2 - z_1 \leq -3$$

finally we get:

$$\text{maximize } 3z_1 - 8z_2 + 37$$

subject to:

$$z_1 - 4z_2 \leq 1$$

$$z_2 - z_1 \leq -3$$

$$z_1 \geq 0$$

$$z_2 \geq 0$$

Additional Space

Additional Space

CS570
Analysis of Algorithms
Spring 2017
Exam II

Name: _____
Student ID: _____
Email Address: _____

_____ **Check if DEN Student**

	Maximum	Received
Problem 1	20	
Problem 2	10	
Problem 3	10	
Problem 4	15	
Problem 5	15	
Problem 6	15	
Problem 7	15	
Total		

1) 20 pts

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

[/FALSE] The value of the max flow is not enough and we need the complete flow

Given the value of max flow, we can find a min-cut in linear time.

[/FALSE]

The Ford-Fulkerson algorithm can compute the maximum flow in polynomial time.

[/FALSE]

A network with unique maximum flow has a unique min-cut.

[TRUE/]

If all of the edge capacities in a graph are an integer multiple of 3, then the value of the maximum flow will be a multiple of 3.

[/FALSE]

The Floyd-Warshall algorithm always fails to find the shortest path between two nodes in a graph with a negative cycle.

[/FALSE] 0/1 knapsack does not have a polynomial time dynamic programming solution—not even a weakly polynomial solution. 0/1 knapsack has a pseudopolynomial time dynamic programming solution which is basically an exponential time solution (with respect to its input size)

0/1 knapsack problem can be solved using dynamic programming in polynomial time, but not **strongly** polynomial time.

[/FALSE]

If a dynamic programming algorithm has n subproblems, then its running time complexity is $O(n)$.

[/FALSE]

The Travelling Salesman problem can be solved using dynamic programming in polynomial time

[/FALSE]

If flow in a network has a cycle, this flow is not a valid flow.

[FALSE/] We can't do this for *undirected* graphs

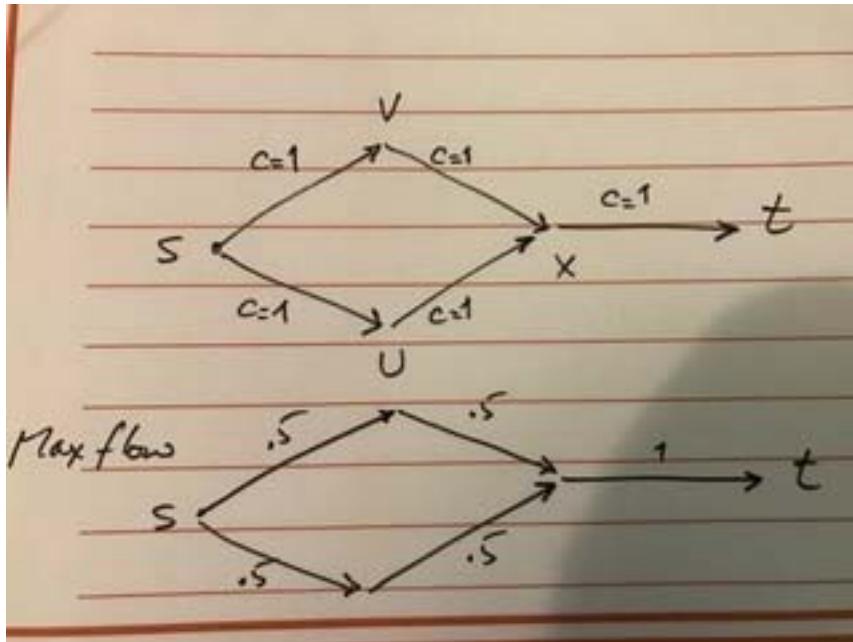
We can use the Bellman-Ford algorithm for undirected graph with negative edge weights.

2) 10 pts

Given $N(G(V, E), s, t, c)$, a flow network with source s , sink t , and positive integer edge capacities $c(e)$ for every $e \in E$. Prove or disprove the following statement:

A maximum flow in an integer capacity graph must have integral (integer) flow on each edge.

Counterexample:



Grading:

A correct counter example gets full 10 points, otherwise 0 points is given.

Common Mistake:

0 is an integer so it will not count as a non-integral flow.

3) 10 pts

The subset sum problem is defined as follows: Given a set of n positive integers $S = \{a_1, a_2, \dots, a_n\}$ and a target value T , does there exist a subset of S such that the sum of the elements in the subset is equal to T ? Let's define a boolean matrix M where $M(i, j)$ is true if there exists a subset of $\{a_1, a_2, \dots, a_i\}$ whose sum equals j . Which one of the following recurrences is valid? (circle one)

1) $M(i, j) = M(i-1, j) \cup M(i, j - a_i)$

2) $M(i, j) = M(i-1, j) \cup M(i-1, j - a_i)$ This is the correct solution

3) $M(i, j) = M(i-1, j-1) \cup M(i-1, j)$

4) $M(i, j) = M(i, j - a_i) \cup M(i-1, j - a_i)$

Solution:

For computing $M(i, j)$ you either include a_i or you don't. If you include a_i then we have $M(i, j) = M(i-1, j-a_i)$. If you don't include a_i , then $M(i, j) = M(i-1, j)$.

Grading:

Recurrence 2 gets full 10 points. Recurrences 1 and 4 get 5 points since they have one of the two cases correct. Recurrence 3 gets 0 points.

4) 15 pts

This problem involves partitioning a given input string into disjoint substrings in the cheapest way. Let $x_1x_2 \dots x_n$ be a string, where particular value of x_i does not matter. Let $C(i,j)$ (for $i \leq j$) be the given precomputed cost of each substring $x_i \dots x_j$. A partition is a decomposition of a string into disjoint substrings. The cost of a partition is the sum of costs of substrings. The goal is to find the min-cost partition.

An example. Let "ab" be a given string. This string can be partitioning in two different ways, such as, "a", "b", and "ab" with the following costs $C(1,1) + C(2,2)$ and $C(1,2)$ respectively.

a) Define (in plain English) subproblems to be solved. (5 pts)

Let subproblem $\text{OPT}(j)$ be the min-cost partition $x_1x_2 \dots x_j$

b) Write the recurrence relation for subproblems. (7 pts)

$$\begin{aligned}\text{OPT}(j) &= \min_{1 \leq k \leq j} (\text{OPT}(k-1) + C(k, j)) \\ \text{OPT}(0) &= 0\end{aligned}$$

c) Compute the runtime of the algorithm. (3 pts)

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

Common mistake:

If you write the recurrence based on two parameters of i, j for the corresponding subsequence $x_i \dots x_j$ like $\text{opt}(i, j)$ you might lose points from 4 to 7 points in part b, depending on your solution. In this case, you probably came up with complexity of $O(n^3)$ instead of $O(n^2)$ which is not optimum compared to the right solution and you might lose other points (2 to 3 points) in part c.

5)

You have two rooms to rent out. There are n customers interested in renting the rooms. The i^{th} customer wishes to rent one room (either room you have) for $d[i]$ days and is willing to pay $\text{bid}[i]$ for the entire stay. Customer requests are non-negotiable in that they would not be willing to rent for a shorter or longer duration. Devise a dynamic programming algorithm to determine the maximum profit that you can make from the customers over a period of D days.

Let $\text{OPT}(d_1, d_2, i)$ be the maximum profit obtainable with d_1 remaining days for room 1 and d_2 remaining days for room 2 using the first i customers.

a) Write the recurrence relation for subproblems. (7 pts)

$$\begin{aligned}\text{OPT}(d_1, d_2, i) = \max(&\text{bid}[i] + \text{OPT}(d_1 - d[i], d_2, i - 1), \\ &\text{bid}[i] + \text{OPT}(d_1, d_2 - d[i], i - 1), \\ &\text{OPT}(d_1, d_2, i - 1))\end{aligned}$$

Initial conditions

$$\text{OPT}(d_1, d_2, 0) = 0$$

$$\text{OPT}(d_1, d_2, i) = -\infty \text{ if } d_1 < d[i] \text{ or } d_2 < d[i]$$

b) Compute the runtime of the algorithm. (4 pts)

$$O(nD^2)$$

Grading Rubric:

Common mistakes:

1. Most common mistake is to model each room separately, so having two recurrence for two rooms. This is not true, and will lead to the same solution for each recurrence.
2. Another common mistake is to model the problem using $2D$ days (simply add up), this is also not true. i.e. If a customer wants to stay for 4 days and there are 2 days left in both rooms, then this approach of combining two rooms into one incorrectly suggests that the customer can stay.
3. Also, there are some students model the problem using three variables, (days, customers, rooms), this is not true.

- Using one dimensional array to define subproblem is also wrong.

Credits:

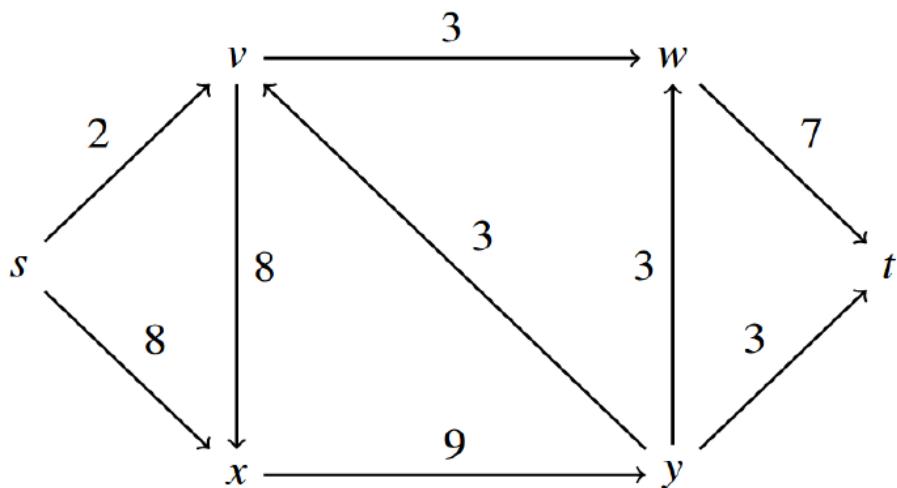
- Without boundary condition, or incorrect boundary condition : -2 points
- Every missing recurrence subproblem : -3 points. i.e. the correct recurrence should be $\max \{f_1, f_2, f_3\}$, but one of them is missing.
- Incorrect time complexity: -4 points (no partial credit)

Partial credits:

For common mistakes mentioned above, give maximum of 6 points (depends on the correctness of recurrence equation and run time complexity, if the recurrence equation is not correct then no credit was given even though the complexity is correct)

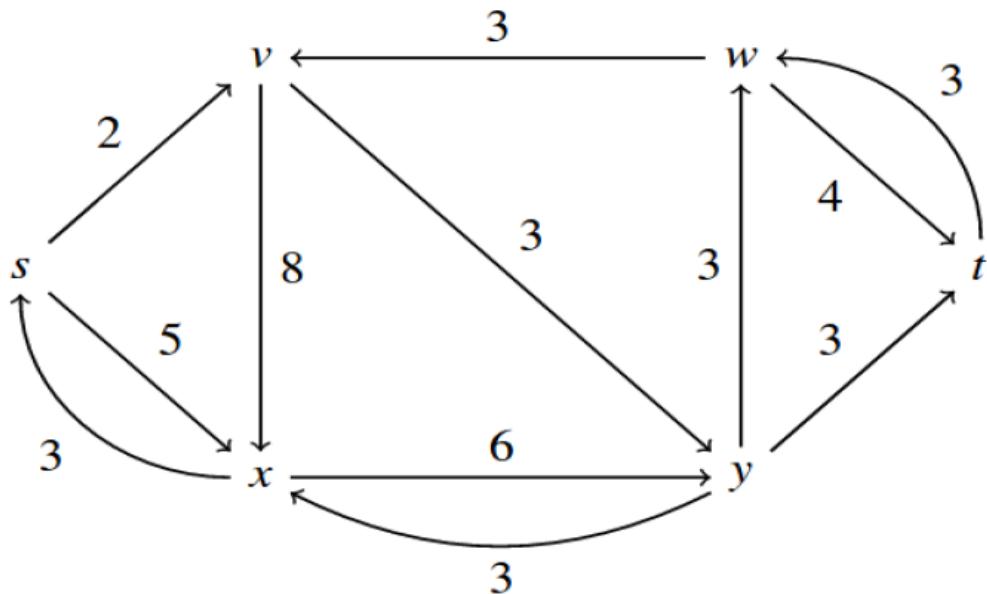
6) 15 pts

You are given the following graph. Each edge is labeled with the capacity of that edge.



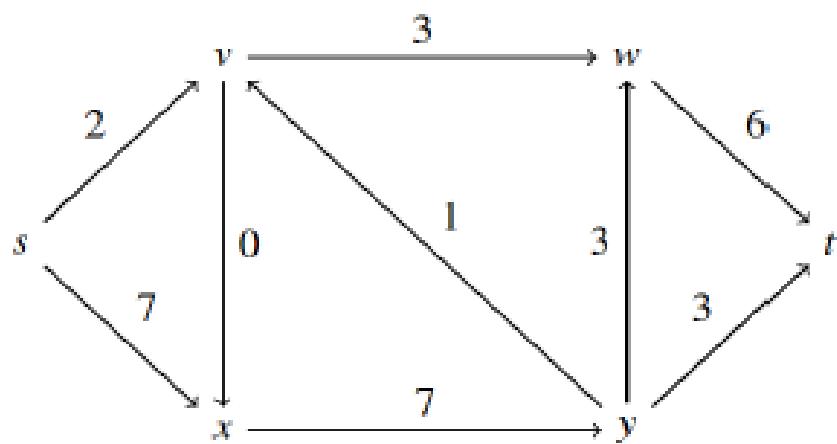
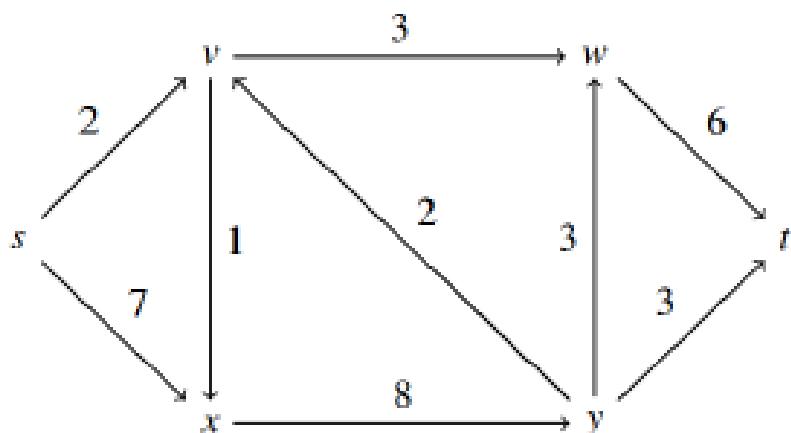
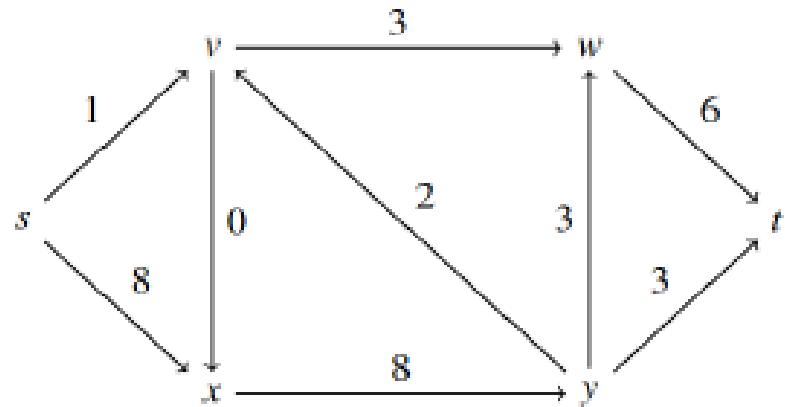
- a) Draw the corresponding residual graph after sending as much flow as possible along the path $s \rightarrow x \rightarrow y \rightarrow v \rightarrow w \rightarrow t$. (5 pts)

3

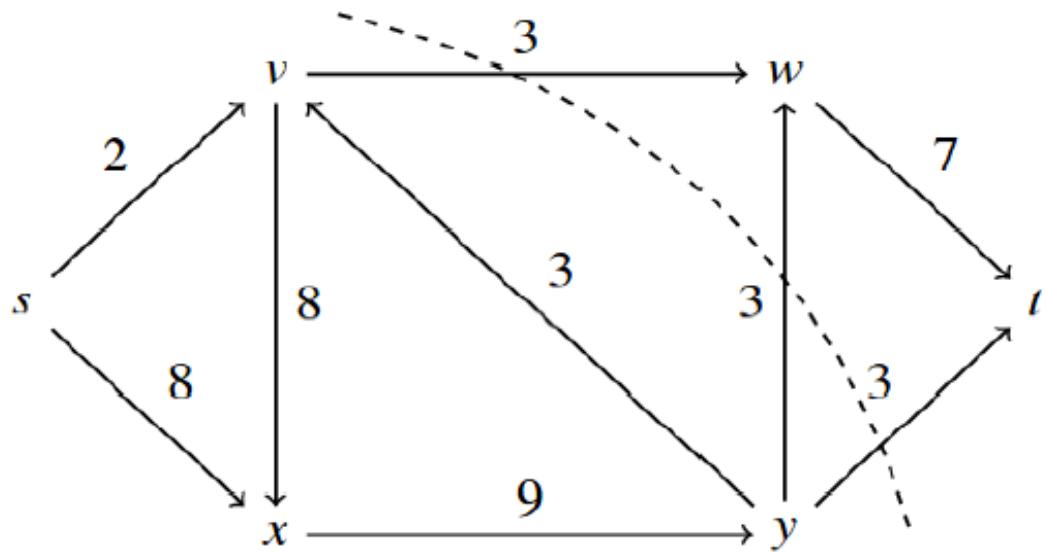


b) Find the value of a max-flow. (5 pts)

9, multiple answers



c) Find a min-cut? (5 pts)



7) 15 pts

Consider a drip irrigation system, which is an irrigation method that saves water and fertilizer by allowing water to drip slowly to the roots of plants. Suppose that the location of all drippers are given to us in terms of their coordinates (x^d, y^d) . Also, we are given locations of plants specified by their coordinates (x^p, y^p) .

A dripper can only provide water to plants within distance l . A single dripper can provide water to no more than n plants. However, we recently got some funding to upgrade our system with which we bought k monster drippers, which can provide water supply to three times the number of plants compared to standard drippers. So, we now have i standard drippers and k monster drippers.

Given the locations of the plants and drippers, as well as the parameters l and n , decide whether every plant can be watered simultaneously by a dripper, subject to the above mentioned constraints. Justify carefully that your algorithm is correct and can be obtained in polynomial time.

Solution:

Create a graph $G=(V, E)$

$V = \{\text{source } s, \text{ nodes } p_1, \dots, p_j \text{ for sets of plants, nodes } d_1, \dots, d_i \text{ for standard droppers, nodes for } d_{i+1}, \dots, d_{i+k}, \text{ and sink } t\}$ (1 pt)

$E = \{$

$c(s, p_x) = 1, \text{ for } x = 1 \text{ to } j$ (1 pt)

$c(p_x, d_y) = 1, \text{ for } x = 1 \text{ to } j, y = 1 \text{ to } i+k$, (1 pt) and p_x, d_y are close enough (1 pt)

$c(d_y, t) = n, \text{ for } y = 1 \text{ to } I$ (2 pt)

$c(d_y, t) = 3n, \text{ for } y = i \text{ to } i+k$ (2 pt)

$\}$

Use Ford-Fulkerson algorithm to find maximum integer flow f in G and justification. (7 pt) #

CS570
Analysis of Algorithms
Spring 2017
Exam III

Name: _____
Student ID: _____
Email Address: _____

_____ **Check if DEN Student**

	Maximum	Received
Problem 1	20	
Problem 2	16	
Problem 3	16	
Problem 4	16	
Problem 5	16	
Problem 6	16	
Total		

1) 20 pts

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

[**TRUE** /]

Given a minimum cut, we could find the maximum flow value in $O(E)$ time.

[**FALSE**]

Any NP-hard problem can be solved in time $O(2^{\text{poly}(n)})$, where n is the input size and $\text{poly}(n)$ is a polynomial.

[**TRUE** /]

Any NP problem can be solved in time $O(2^{\text{poly}(n)})$, where n is the input size and $\text{poly}(n)$ is a polynomial.

[**TRUE** /]

If $3\text{-SAT} \leq_p 2\text{-SAT}$, then $P = NP$.

[**FALSE**]

Assuming $P \neq NP$, there can exist a polynomial-time approximation algorithm for the general Traveling Salesman Problem.

[**FALSE**]

Let $(S, V-S)$ be a minimum (s,t) -cut in the network flow graph G . Let (u,v) be an edge that crosses the cut in the forward direction, i.e., $u \in S$ and $v \in V-S$. Then increasing the capacity of the edge (u, v) necessarily increases the maximum flow of G .

[**FALSE**]

If problem X can be solved using dynamic programming, then X belongs to P .

[**FALSE**]

All instances of linear programming have exactly one optimal solution.

[**FALSE**]

Let $Y \leq_p X$ and there exists a 2-approximation for X , then there must exist a 2-approximation for Y .

[**TRUE** /]

There is no known polynomial-time algorithm to solve an integer linear programming.

2) 16 pts

A set of n space stations need your help in building a radar system to track spaceships traveling between them. The i^{th} space station is located in 3D space at coordinates (x_i, y_i, z_i) . The space stations never move. Each space station i will have a radar with power r_i , where r_i is to be determined. You want to figure how powerful to make each space station's radar transmitter, so that whenever any spaceship travels in a straight line from one space station to another, it will always be in radar range of either the first space station (its origin) or the second space station (its destination). A radar with power r is capable of tracking space ships anywhere in the sphere with radius r centered at itself. Thus, a space ship is within radar range through its trip from space station i to space station j if every point along the line from (x_i, y_i, z_i) to (x_j, y_j, z_j) falls within either the sphere of radius r_i centered at (x_i, y_i, z_i) or the sphere of radius r_j centered at (x_j, y_j, z_j) . The cost of each radar transmitter is proportional to its power, and you want to minimize the total cost of all of the radar transmitters. You are given all of the $(x_1, y_1, z_1), \dots, (x_n, y_n, z_n)$ values, and your job is to choose values for r_1, \dots, r_n . Express this problem as a linear program.

(a) Describe your variables for the linear program. (3 pts)

Solution: $r_i = \text{the power of the } i^{\text{th}} \text{ radar transmitter, } i=1,2,\dots,n$ (3 pts)

(b) Write out the objective function. (5 pts)

Solution: Minimize $r_1 + r_2 + \dots + r_n$.

Defining the objective function without mentioning r_i : -3 pts

(c) Describe the set of constraints for LP. You need to specify the number of constraints needed and describe what each constraint represents. (8 pts)

Solution: $r_i + r_j \geq \sqrt{(x_i - x_j)^2 + (y_i - y_j)^2 + (z_i - z_j)^2}$. Or, $r_i + r_j \geq d_{i,j}$ for each pair of stations i and j , where $d_{i,j}$ is the distance from station i to station j (6 pts)

we need $\sum_{i=1}^{n-1} i = (n^2 - n)/2$ constraints of inequality (2 pts)

3) 16 pts

Given a row of n houses that can each be painted red, green, or blue with a cost $P(i, c)$ for painting house i with color c . Devise a dynamic programming algorithm to find a minimum cost coloring of the entire row of houses such that no two adjacent houses are the same color.

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

Solution: Let $C(i, c)$ be the minimum possible cost of a valid coloring of the first i houses in which the last house gets color c .

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

Solution: The recurrence is $C(i, c) = \min_{c' \neq c} \{C(i - 1, c') + P(i, c)\}$.

The base case: $C(1, c) = P(1, c)$, for all colors c .

c) Describe (using pseudocode) how the value of an optimal solution is obtained using iteration. (4 pts)

```
Set r[1] = P(1,r), g[1] = P(1,g), b[1] = P(1,b)
for i=2 to n
    r[i] = P(i,r) + min(g[i-1],b[i-1])
    g[i] = P(i,g) + min(r[i-1],b[i-1])
    b[i] = P(i,b) + min(r[i-1],g[i-1])
endfor
return min{r[n], g[n], b[n]}
```

d) Compute the runtime of the algorithm. (2 pts)

Solution: $O(n)$

The runtime is $O(n)$ since there are n subproblems each of which takes $O(1)$ time.

Grading rubric:

A. Part (a):

- a. Missing color in subproblem definition -2
- b. Unclear description -2 ~ -3

B. Part (b):

- a. Missing color in recurrence -4

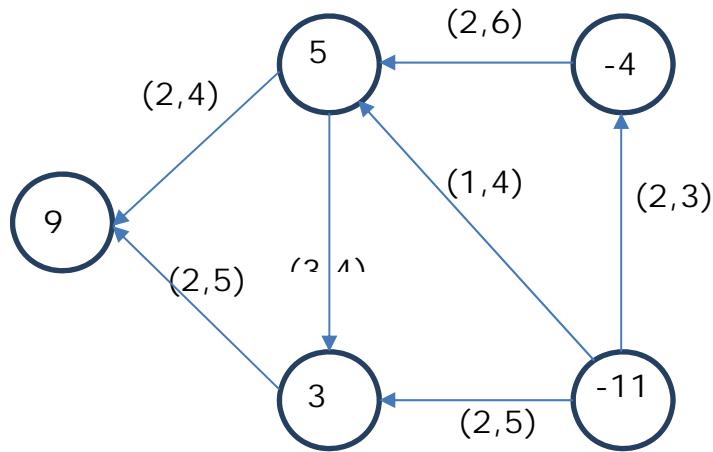
- b. Totally wrong recurrence -6
 - c. Fail to consider no same color in adjacent choice -2
 - d. Wrong symbols or typos in the recurrence -1 ~ -2
- C. Part (c):
- a. If wrong answer in part (b) -2
 - b. If no pseudo code -2
 - c. If the answer is not according to wrong answer in part (b) -4
 - d. If wrong order of for loop, color before house id -2
- D. Part (d):
- a. The question is graded according to recurrence and pseudo on part (b) and (c)
 - b. If there is additional constant in big O notation -1

Common errors:

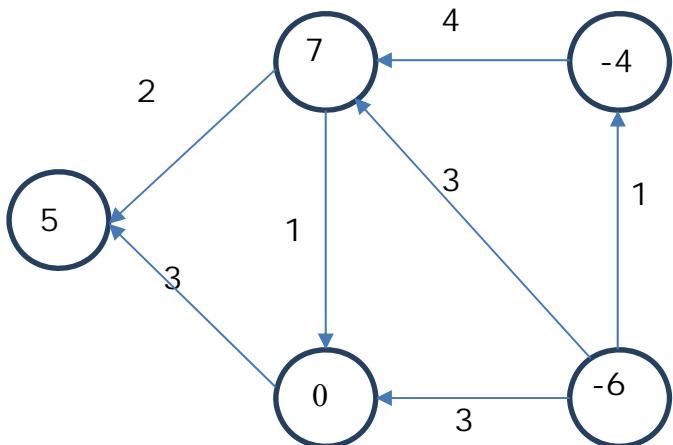
- 1. Failed to include the color of house in the definition of subproblem.
 - a. Score if there are no other mistakes: part (a) -2, part (b) -4, part (c) -2 = 8
- 2. Each house must be painted into one of the three colors.
 - a. Score if there are no other mistakes: part (a) -4, part (b) -4, part (c) -2 = 6
- 3. Use interval (2d state + 2 colors for the ends)
 - a. Score if there are no other mistakes: part (a) -1, part (b) -1, part (c) -1 = 13

4) 16 pts

In the network 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.

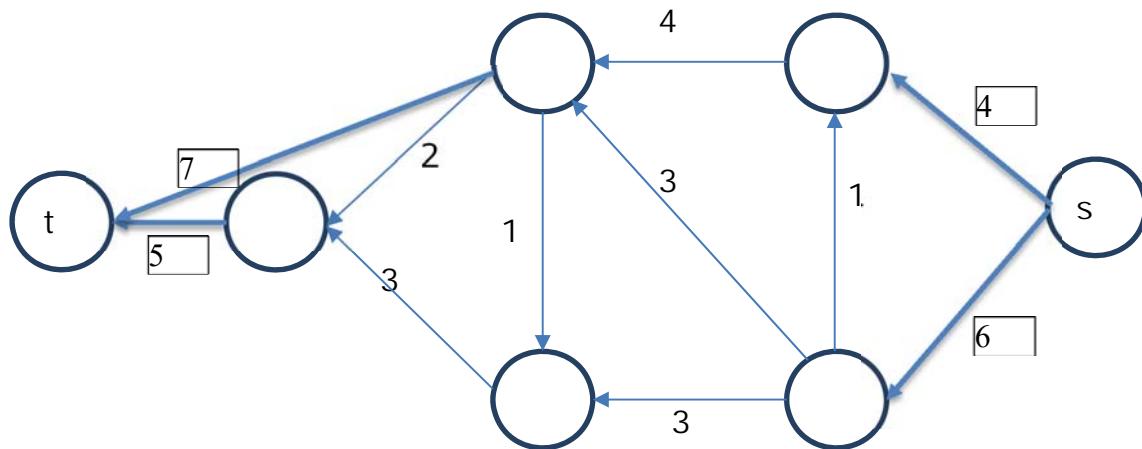


- a) Turn the circulation with lower bounds problem into a circulation problem without lower bounds (6 pts)



Grading rubric

- 1) a) There are total 12 values in the network-flow graph that needs to be written, 5 values (demands) on vertices and 7 values on edges (modified capacities). For each correct value 0.5 marks will be awarded.
- b) 2 marks will be deducted for adding a source and sink vertices and edges.
- b) Turn the circulation with demands problem into the maximum flow problem (6 pts)



Grading rubric:

- a) 2 marks for adding two super nodes, 1 for each node.
 - b) 2 marks for connecting negative demand vertices to supply nodes and positive demands to sink nodes, 0 if they are connected in opposite way.
 - c) 2 marks for placing the capacities on the newly added edges between super nodes and the vertices, will lose 0.5 marks for each edge capacity mistake (maximum deduction here is 2 marks).
 - d) will loose 0.5 marks for any incorrect edge capacity value on the edges in the original network flow.
- c) Does a feasible circulation exist? Explain your answer. (4 pts)

No, a feasible circulation doesn't exist. In the given original network, total demand values on the vertices doesn't match with the total supply value on all the vertices.

Grading rubric:

- a) 2 marks for answering no, 2 marks for explanation.
If you have answered yes and obtained a max-flow value of 10 and show that edges from source are saturated then 1.5 marks are awarded.

5) 16 pts

We want to become celebrity chefs by creating a new dish. There are n ingredients and we'd like to use as many of them as possible. However, some ingredients don't go so well with others: there is $n \times n$ matrix D giving *discord* between any two ingredients, i.e., $D[i,j]$ is a real value between 0 and 1: 0 means i and j go perfectly well together and there is no discord and 1 means they go very badly together. Any dish prepared with these ingredients incurs a *penalty* which is the sum of the discords between all pairs of ingredients in the dish. We would like the total penalty to be small. Consider the decision problem EXPERIMENTAL CUISINE: can we prepare a dish with at least k ingredients and with the total penalty at most p ?

a) Show that Experimental Cuisine is in NP. (4 pts)

Certificate: a dish with at least k ingredients with total penalty of at most p .

Verifier: Go through ingredients and calculate the penalty and the total number of ingredients which takes at most quadratic time.

Grading rubric

-2 points for not mentioning about how to compute the penalty(e.g. add, search, compare, etc.)

b) Show that EXPERIMENTAL CUISINE is NP-complete by giving a reduction from INDEPENDENT SET. (12 pts)

Given an instance of IS, (G,k) , where G is a graph, and k is a parameter. Construct the following: Set D to be an incident matrix of G , i.e., $D[i,j]=1$, iff vertices i and j are connected by an edge, otherwise $D[i,j]=0$.

Set $p=0$. (D,k,p) is an instance of EXPERIMENTAL CUISINE.

Claim: (D,p,k) returns YES for EXPERIMENTAL CUISINE iff (G,k) returns YES for INDEPENDENT SET problem. Proof: Take instances, observe that a collection of vertices form an IS iff corresponding $p = 0$. k parameter stands for size of solution in both cases.

Grading rubric

Construction(4 points)

Since the problem is stated to reduce FROM IS, it should start with Independent Set to construct a discord matrix for EXPERIMENTAL CUISINE.

Claim (4 points)

Correct claim should be made for case of reduction: (D, k, p) returns YES for EXPERIMENTAL CUISINE iff (G, k) returns YES for INDEPENDENT SET problem.

Proof of reduction case (4 points)

The proof should be made in both directions when $p=0$.

6) 16 pts

There are n positive integers. Your goal is to concatenate them to form a single integer which is as large as possible. For example, for the following three integers 111, 222, and 3, the largest integer is 3222111.

a) Develop a greedy algorithm to solve the problem for arbitrary n (10 pts)

1. Sort the n integers using the following rule: If a is larger than b , then put a in front of b . (i) The comparison for number a and b is conducted by scanning through each digit. Need to consider several different cases: a) if the current digit is the same b) if the length of one number is shorter than another. (ii) or the comparison can be easily conduct by concatenating two numbers i.e $\text{concat}(a,b) > \text{concat}(b,a)$
2. Concatenate all the sorted integers together.

b) Provide proof of correctness. (6 pts)

Exchange argument, induction should be the correct way to do it.

Grading Rubric and common mistakes:

For problem (a),

- Your solution is incorrect, but you explain your own solution in a meaningful way, you will get ~4 points.
- the idea is generally correct, but if doesn't consider complete cases (including how to handle different length of numbers? i.e. 67 vs 676, what about the current comparing digit is the same? and etc), will deduct 2-4 points.
- some students write the solution like this: “comparing integers.. “ this is not correct since the comparison should be conducted digit by digit, thus not a right solution
- Some students answered - “appending zero”, this is not correct. i.e 67 vs 676, after appending zero, $676 > 67$, so we get 67667 but the larger one should be 67676

For problem (b),

Common mistakes:

- the idea is in general right, but the proof itself is not a rigorous, get ~2 points deducted
- some students give proof by example, this is not correct. Showing by example is similar to writing a unit-test for programming, which is not a proof at all. deduct ~ 4 points
- some students follow the induction/exchange argument procedure, but does not show any valid evidence among the proof. i.e. use the proof framework and directly get the conclusion without showing any valid evidence. Deduct ~ 2-4 points

Additional Space

Additional Space

Integer Programming

9

The linear-programming models that have been discussed thus far all have been *continuous*, in the sense that decision variables are allowed to be fractional. Often this is a realistic assumption. For instance, we might easily produce $102\frac{3}{4}$ gallons of a divisible good such as wine. It also might be reasonable to accept a solution giving an hourly production of automobiles at $58\frac{1}{2}$ if the model were based upon average hourly production, and the production had the interpretation of *production rates*.

At other times, however, fractional solutions are not realistic, and we must consider the optimization problem:

$$\text{Maximize } \sum_{j=1}^n c_j x_j,$$

subject to:

$$\begin{aligned} \sum_{j=1}^n a_{ij} x_j &= b_i & (i = 1, 2, \dots, m), \\ x_j &\geq 0 & (j = 1, 2, \dots, n), \\ x_j &\text{ integer} & (\text{for some or all } j = 1, 2, \dots, n). \end{aligned}$$

This problem is called the (linear) *integer-programming problem*. It is said to be a *mixed* integer program when some, but not all, variables are restricted to be integer, and is called a *pure* integer program when *all* decision variables must be integers. As we saw in the preceding chapter, if the constraints are of a network nature, then an integer solution can be obtained by ignoring the integrality restrictions and solving the resulting linear program. In general, though, variables will be fractional in the linear-programming solution, and further measures must be taken to determine the integer-programming solution.

The purpose of this chapter is twofold. First, we will discuss integer-programming formulations. This should provide insight into the scope of integer-programming applications and give some indication of why many practitioners feel that the integer-programming model is one of the most important models in management science. Second, we consider basic approaches that have been developed for solving integer and mixed-integer programming problems.

9.1 SOME INTEGER-PROGRAMMING MODELS

Integer-programming models arise in practically every area of application of mathematical programming. To develop a preliminary appreciation for the importance of these models, we introduce, in this section, three areas where integer programming has played an important role in supporting managerial decisions. We do not provide the most intricate available formulations in each case, but rather give basic models and suggest possible extensions.

Capital Budgeting In a typical capital-budgeting problem, decisions involve the selection of a number of potential investments. The investment decisions might be to choose among possible plant locations, to select a configuration of capital equipment, or to settle upon a set of research-and-development projects. Often it makes no sense to consider partial investments in these activities, and so the problem becomes a *go-no-go* integer program, where the decision variables are taken to be $x_j = 0$ or 1 , indicating that the j th investment is rejected or accepted. Assuming that c_j is the contribution resulting from the j th investment and that a_{ij} is the amount of resource i , such as cash or manpower, used on the j th investment, we can state the problem formally as:

$$\text{Maximize } \sum_{j=1}^n c_j x_j,$$

subject to:

$$\begin{aligned} \sum_{j=1}^n a_{ij} x_j &\leq b_i \quad (i = 1, 2, \dots, m), \\ x_j &= 0 \quad \text{or} \quad 1 \quad (j = 1, 2, \dots, n). \end{aligned}$$

The objective is to maximize total contribution from all investments without exceeding the limited availability b_i of any resource.

One important special scenario for the capital-budgeting problem involves cash-flow constraints. In this case, the constraints

$$\sum_{j=1}^n a_{ij} x_i \leq b_i$$

reflect incremental cash balance in each period. The coefficients a_{ij} represent the net cash flow from investment j in period i . If the investment requires additional cash in period i , then $a_{ij} > 0$, while if the investment generates cash in period i , then $a_{ij} < 0$. The righthand-side coefficients b_i represent the incremental exogenous cash flows. If additional funds are made available in period i , then $b_i > 0$, while if funds are withdrawn in period i , then $b_i < 0$. These constraints state that the funds required for investment must be less than or equal to the funds generated from prior investments plus exogenous funds made available (or minus exogenous funds withdrawn).

The capital-budgeting model can be made much richer by including logical considerations. Suppose, for example, that investment in a new product line is contingent upon previous investment in a new plant. This *contingency* is modeled simply by the constraint

$$x_j \geq x_i,$$

which states that if $x_i = 1$ and project i (new product development) is accepted, then necessarily $x_j = 1$ and project j (construction of a new plant) must be accepted. Another example of this nature concerns conflicting projects. The constraint

$$x_1 + x_2 + x_3 + x_4 \leq 1,$$

for example, states that only one of the first four investments can be accepted. Constraints like this commonly are called *multiple-choice constraints*. By combining these logical constraints, the model can incorporate many complex interactions between projects, in addition to issues of resource allocation.

The simplest of all capital-budgeting models has just one resource constraint, but has attracted much attention in the management-science literature. It is stated as:

$$\text{Maximize } \sum_{j=1}^n c_j x_j,$$

subject to:

$$\sum_{j=1}^n a_j x_j \leq b,$$

$$x_j = 0 \quad \text{or} \quad 1 \quad (j = 1, 2, \dots, n).$$

Usually, this problem is called the 0–1 *knapsack* problem, since it is analogous to a situation in which a hiker must decide which goods to include on his trip. Here c_j is the “value” or utility of including good j , which weighs $a_j > 0$ pounds; the objective is to maximize the “pleasure of the trip,” subject to the weight limitation that the hiker can carry no more than b pounds. The model is altered somewhat by allowing more than one unit of any good to be taken, by writing $x_j \geq 0$ and x_j -integer in place of the 0–1 restrictions on the variables. The knapsack model is important because a number of integer programs can be shown to be equivalent to it, and further, because solution procedures for knapsack models have motivated procedures for solving general integer programs.

Warehouse Location In modeling distribution systems, decisions must be made about tradeoffs between transportation costs and costs for operating distribution centers. As an example, suppose that a manager must decide which of n warehouses to use for meeting the demands of m customers for a good. The decisions to be made are which warehouses to operate and how much to ship from any warehouse to any customer. Let

$$y_i = \begin{cases} 1 & \text{if warehouse } i \text{ is opened,} \\ 0 & \text{if warehouse } i \text{ is not opened;} \end{cases}$$

$$x_{ij} = \text{Amount to be sent from warehouse } i \text{ to customer } j.$$

The relevant costs are:

f_i = Fixed operating cost for warehouse i , if opened (for example, a cost to lease the warehouse),

c_{ij} = Per-unit operating cost at warehouse i plus the transportation cost for shipping from warehouse i to customer j .

There are two types of constraints for the model:

- i) the demand d_j of each customer must be filled from the warehouses; and
- ii) goods can be shipped from a warehouse only if it is opened.

The model is:

$$\text{Minimize} \quad \sum_{i=1}^m \sum_{j=1}^n c_{ij} x_{ij} + \sum_{i=1}^m f_i y_i, \quad (1)$$

subject to:

$$\sum_{i=1}^m x_{ij} = d_j \quad (j = 1, 2, \dots, n), \quad (2)$$

$$\sum_{j=1}^n x_{ij} - y_i \left(\sum_{j=1}^n d_j \right) \leq 0 \quad (i = 1, 2, \dots, m), \quad (3)$$

$$x_{ij} \geq 0 \quad (i = 1, 2, \dots, m; j = 1, 2, \dots, n),$$

$$y_i = 0 \quad \text{or} \quad 1 \quad (i = 1, 2, \dots, m).$$

The objective function incorporates transportation and variable warehousing costs, in addition to fixed costs for operating warehouses. The constraints (2) indicate that each customer's demand must be met. The summation over the shipment variables x_{ij} in the i th constraint of (3) is the amount of the good shipped from warehouse i . When the warehouse is not opened, $y_i = 0$ and the constraint specifies that nothing can be shipped from the warehouse. On the other hand, when the warehouse is opened and $y_i = 1$, the constraint simply states that the amount to be shipped from warehouse i can be no larger than the total demand, which is always true. Consequently, constraints (3) imply restriction (ii) as proposed above.

Although oversimplified, this model forms the core for sophisticated and realistic distribution models incorporating such features as:

1. multi-echelon distribution systems from plant to warehouse to customer;
2. capacity constraints on both plant production and warehouse throughput;
3. economies of scale in transportation and operating costs;
4. service considerations such as maximum distribution time from warehouses to customers;
5. multiple products; or
6. conditions preventing splitting of orders (in the model above, the demand for any customer can be supplied from several warehouses).

These features can be included in the model by changing it in several ways. For example, warehouse capacities are incorporated by replacing the term involving y_i in constraint (3) with $y_i K_i$, where K_i is the throughput capacity of warehouse i ; multi-echelon distribution may require triple-subscripted variables x_{ijk} denoting the amount to be shipped, from plant i to customer k through warehouse j . Further examples of how the simple warehousing model described here can be modified to incorporate the remaining features mentioned in this list are given in the exercises at the end of the chapter.

Scheduling The entire class of problems referred to as sequencing, scheduling, and routing are inherently integer programs. Consider, for example, the scheduling of students, faculty, and classrooms in such a way that the number of students who cannot take their first choice of classes is minimized. There are constraints on the number and size of classrooms available at any one time, the availability of faculty members at particular times, and the preferences of the students for particular schedules. Clearly, then, the i th student *is* scheduled for the j th class during the n th time period or *not*; hence, such a variable is either zero or one. Other examples of this class of problems include line-balancing, critical-path scheduling with resource constraints, and vehicle dispatching.

As a specific example, consider the scheduling of airline flight personnel. The airline has a number of routing "legs" to be flown, such as 10 A.M. New York to Chicago, or 6 P.M. Chicago to Los Angeles. The airline must schedule its personnel crews on routes to cover these flights. One crew, for example, might be scheduled to fly a route containing the two legs just mentioned. The decision variables, then, specify the scheduling of the crews to routes:

$$x_j = \begin{cases} 1 & \text{if a crew is assigned to route } j, \\ 0 & \text{otherwise.} \end{cases}$$

Let

$$a_{ij} = \begin{cases} 1 & \text{if leg } i \text{ is included on route } j, \\ 0 & \text{otherwise,} \end{cases}$$

and

$$c_j = \text{Cost for assigning a crew to route } j.$$

The coefficients a_{ij} define the acceptable combinations of legs and routes, taking into account such characteristics as sequencing of legs for making connections between flights and for including in the routes ground time for maintenance. The model becomes:

$$\text{Minimize } \sum_{j=1}^n c_j x_j,$$

subject to:

$$\begin{aligned} \sum_{j=1}^n a_{ij}x_j &= 1 & (i = 1, 2, \dots, m), \\ x_j &= 0 \text{ or } 1 & (j = 1, 2, \dots, n). \end{aligned} \quad (4)$$

The i th constraint requires that one crew must be assigned on a route to fly leg i . An alternative formulation permits a crew to ride as passengers on a leg. Then the constraints (4) become:

$$\sum_{j=1}^n a_{ij}x_j \geq 1 \quad (i = 1, 2, \dots, m). \quad (5)$$

If, for example,

$$\sum_{j=1}^n a_{1j}x_j = 3,$$

then two crews fly as passengers on leg 1, possibly to make connections to other legs to which they have been assigned for duty.

These airline-crew scheduling models arise in many other settings, such as vehicle delivery problems, political districting, and computer data processing. Often model (4) is called a *set-partitioning problem*, since the set of legs will be divided, or partitioned, among the various crews. With constraints (5), it is called a *set-covering problem*, since the crews then will cover the set of legs.

Another scheduling example is the so-called *traveling salesman problem*. Starting from his home, a salesman wishes to visit each of $(n - 1)$ other cities and return home at minimal cost. He must visit each city exactly once and it costs c_{ij} to travel from city i to city j . What route should he select? If we let

$$x_{ij} = \begin{cases} 1 & \text{if he goes from city } i \text{ to city } j, \\ 0 & \text{otherwise,} \end{cases}$$

we may be tempted to formulate his problem as the assignment problem:

$$\text{Minimize } \sum_{i=1}^n \sum_{j=1}^n c_{ij}x_{ij},$$

subject to:

$$\begin{aligned} \sum_{i=1}^n x_{ij} &= 1 & (j = 1, 2, \dots, n), \\ \sum_{j=1}^n x_{ij} &= 1 & (i = 1, 2, \dots, n), \\ x_{ij} &\geq 0 & (i = 1, 2, \dots, n; j = 1, 2, \dots, n). \end{aligned}$$

The constraints require that the salesman must enter and leave each city exactly once. Unfortunately, the assignment model can lead to infeasible solutions. It is possible in a six-city problem, for example, for the assignment solution to route the salesman through two disjoint subtours of the cities instead of on a single trip or tour. (See Fig. 9.1.)

Consequently, additional constraints must be included in order to eliminate subtour solutions. There are a number of ways to accomplish this. In this example, we can avoid the subtour solution of Fig. 9.1 by including the constraint:

$$x_{14} + x_{15} + x_{16} + x_{24} + x_{25} + x_{26} + x_{34} + x_{35} + x_{36} \geq 1.$$

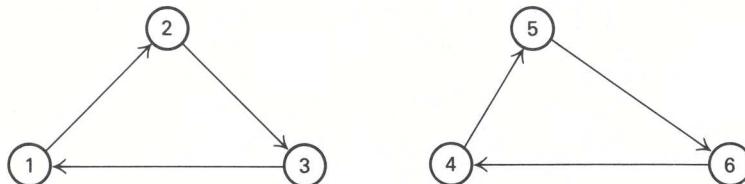


Figure 9.1 Disjoint subtours.

This inequality ensures that at least one leg of the tour connects cities 1, 2, and 3 with cities 4, 5, and 6. In general, if a constraint of this form is included for each way in which the cities can be divided into two groups, then subtours will be eliminated. The problem with this and related approaches is that, with n cities, $(2^n - 1)$ constraints of this nature must be added, so that the formulation becomes a very large integer-programming problem. For this reason the traveling salesman problem generally is regarded as difficult when there are many cities.

The traveling salesman model is used as a central component of many vehicular routing and scheduling models. It also arises in production scheduling. For example, suppose that we wish to sequence $(n - 1)$ jobs on a single machine, and that c_{ij} is the cost for setting up the machine for job j , given that job i has just been completed. What scheduling sequence for the jobs gives the lowest total setup costs? The problem can be interpreted as a traveling salesman problem, in which the “salesman” corresponds to the machine which must “visit” or perform each of the jobs. “Home” is the initial setup of the machine, and, in some applications, the machine will have to be returned to this initial setup after completing all of the jobs. That is, the “salesman” must return to “home” after visiting the “cities.”

9.2 FORMULATING INTEGER PROGRAMS

The illustrations in the previous section not only have indicated specific integer-programming applications, but also have suggested how integer variables can be used to provide broad modeling capabilities beyond those available in linear programming. In many applications, integrality restrictions reflect natural indivisibilities of the problem under study. For example, when deciding how many nuclear aircraft carriers to have in the U.S. Navy, fractional solutions clearly are meaningless, since the optimal number is on the order of one or two. In these situations, the decision variables are inherently integral by the nature of the decision-making problem.

This is not necessarily the case in every integer-programming application, as illustrated by the capital-budgeting and the warehouse-location models from the last section. In these models, integer variables arise from (i) logical conditions, such as *if* a new product is developed, *then* a new plant must be constructed, and from (ii) non-linearities such as fixed costs for opening a warehouse. Considerations of this nature are so important for modeling that we devote this section to analyzing and consolidating specific integer-programming formulation techniques, which can be used as tools for a broad range of applications.

Binary (0–1) Variables

Suppose that we are to determine whether or not to engage in the following activities: (i) to build a new plant, (ii) to undertake an advertising campaign, or (iii) to develop a new product. In each case, we must make a *yes-no* or so-called *go-no-go* decision. These choices are modeled easily by letting $x_j = 1$ if we engage in the j th activity and $x_j = 0$ otherwise. Variables that are restricted to 0 or 1 in this way are termed *binary*, *bivalent*, *logical*, or *0–1 variables*. Binary variables are of great importance because they occur regularly in many model formulations, particularly in problems addressing long-range and high-cost strategic decisions associated with capital-investment planning.

If, further, management had decided that *at most one* of the above three activities can be pursued, the

following constraint is appropriate:

$$\sum_{j=1}^3 x_j \leq 1.$$

As we have indicated in the capital-budgeting example in the previous section, this restriction usually is referred to as a *multiple-choice* constraint, since it limits our choice of investments to be *at most one* of the three available alternatives.

Binary variables are useful whenever variables can assume one of two values, as in batch processing. For example, suppose that a drug manufacturer must decide whether or not to use a fermentation tank. If he uses the tank, the processing technology requires that he make B units. Thus, his production y must be 0 or B , and the problem can be modeled with the binary variable $x_j = 0$ or 1 by substituting Bx_j for y everywhere in the model.

Logical Constraints

Frequently, problem settings impose logical constraints on the decision variables (like timing restrictions, contingencies, or conflicting alternatives), which lend themselves to integer-programming formulations. The following discussion reviews the most important instances of these logical relationships.

Constraint Feasibility

Possibly the simplest logical question that can be asked in mathematical programming is whether a given choice of the decision variables satisfies a constraint. More precisely, *when* is the general constraint

$$f(x_1, x_2, \dots, x_n) \leq b \quad (6)$$

satisfied?

We introduce a binary variable y with the interpretation:

$$y = \begin{cases} 0 & \text{if the constraint is known to be satisfied,} \\ 1 & \text{otherwise,} \end{cases}$$

and write

$$f(x_1, x_2, \dots, x_n) - By \leq b, \quad (7)$$

where the constant B is chosen to be large enough so that the constraint always is satisfied if $y = 1$; that is,

$$f(x_1, x_2, \dots, x_n) \leq b + B,$$

for every possible choice of the decision variables x_1, x_2, \dots, x_n at our disposal. Whenever $y = 0$ gives a feasible solution to constraint (7), we know that constraint (6) must be satisfied. In practice, it is usually very easy to determine a large number to serve as B , although generally it is best to use the smallest possible value of B in order to avoid numerical difficulties during computations.

Alternative Constraints

Consider a situation with the *alternative* constraints:

$$\begin{aligned} f_1(x_1, x_2, \dots, x_n) &\leq b_1, \\ f_2(x_1, x_2, \dots, x_n) &\leq b_2. \end{aligned}$$

At least one, but not necessarily both, of these constraints must be satisfied. This restriction can be modeled by combining the technique just introduced with a multiple-choice constraint as follows:

$$\begin{aligned} f_1(x_1, x_2, \dots, x_n) - B_1 y_1 &\leq b_1, \\ f_2(x_1, x_2, \dots, x_n) - B_2 y_2 &\leq b_2, \\ y_1 + y_2 &\leq 1, \\ y_1, y_2 &\text{ binary.} \end{aligned}$$

The variables y_1 and y_2 and constants B_1 and B_2 are chosen as above to indicate when the constraints are satisfied. The multiple-choice constraint $y_1 + y_2 \leq 1$ implies that at least one variable y_j equals 0, so that, as required, at least one constraint must be satisfied.

We can save one integer variable in this formulation by noting that the multiple-choice constraint can be replaced by $y_1 + y_2 = 1$, or $y_2 = 1 - y_1$, since this constraint also implies that either y_1 or y_2 equals 0. The resulting formulation is given by:

$$\begin{aligned} f_1(x_1, x_2, \dots, x_n) - B_1 y_1 &\leq b_1, \\ f_2(x_1, x_2, \dots, x_n) - B_2(1 - y_1) &\leq b_2, \\ y_1 = 0 \text{ or } 1. \end{aligned}$$

As an illustration of this technique, consider again the custom-molder example from Chapter 1. That example included the constraint

$$6x_1 + 5x_2 \leq 60, \quad (8)$$

which represented the production capacity for producing x_1 hundred cases of six-ounce glasses and x_2 hundred cases of ten-ounce glasses. Suppose that there were an alternative production process that could be used, having the capacity constraint

$$4x_1 + 5x_2 \leq 50. \quad (9)$$

Then the decision variables x_1 and x_2 must satisfy *either* (8) or (9), depending upon which production process is selected. The integer-programming formulation replaces (8) and (9) with the constraints:

$$\begin{aligned} 6x_1 + 5x_2 - 100y &\leq 60, \\ 4x_1 + 5x_2 - 100(1 - y) &\leq 50, \\ y = 0 \text{ or } 1. \end{aligned}$$

In this case, both B_1 and B_2 are set to 100, which is large enough so that the constraint is not limiting for the production process *not* used.

Conditional Constraints

These constraints have the form:

$$f_1(x_1, x_2, \dots, x_n) > b_1 \text{ implies that } f_2(x_1, x_2, \dots, x_n) \leq b_2.$$

Since this implication is not satisfied *only when both* $f_1(x_1, x_2, \dots, x_n) > b_1$ *and* $f_2(x_1, x_2, \dots, x_n) > b_2$, the conditional constraint is logically equivalent to the alternative constraints

$$f_1(x_1, x_2, \dots, x_n) \leq b_1 \text{ and/or } f_2(x_1, x_2, \dots, x_n) \leq b_2,$$

where *at least one* must be satisfied. Hence, this situation can be modeled by alternative constraints as indicated above.

k-Fold Alternatives

Suppose that we must satisfy at least k of the constraints:

$$f_j(x_1, x_2, \dots, x_n) \leq b_j \quad (j = 1, 2, \dots, p).$$

For example, these restrictions may correspond to manpower constraints for p potential inspection systems for quality control in a production process. If management has decided to adopt at least k inspection systems, then the k constraints specifying the manpower restrictions for these systems must be satisfied, and the

remaining constraints can be ignored. Assuming that B_j for $j = 1, 2, \dots, p$, are chosen so that the ignored constraints will not be binding, the general problem can be formulated as follows:

$$f_j(x_1, x_2, \dots, x_n) - B_j(1 - y_j) \leq b_j \quad (j = 1, 2, \dots, p),$$

$$\sum_{j=1}^p y_j \geq k,$$

$$y_j = 0 \text{ or } 1 \quad (j = 1, 2, \dots, p).$$

That is, $y_j = 1$ if the j th constraint is to be satisfied, and at least k of the constraints must be satisfied. If we define $y'_j \equiv 1 - y_j$, and substitute for y_j in these constraints, the form of the resulting constraints is analogous to that given previously for modeling alternative constraints.

Compound Alternatives

The feasible region shown in Fig. 9.2 consists of three disjoint regions, each specified by a system of inequalities. The feasible region is defined by alternative sets of constraints, and can be modeled by the system:

$$\begin{aligned} & \left. \begin{aligned} f_1(x_1, x_2) - B_1 y_1 &\leq b_1 \\ f_2(x_1, x_2) - B_2 y_1 &\leq b_2 \end{aligned} \right\} \text{Region 1} \\ & \left. \begin{aligned} f_3(x_1, x_2) - B_3 y_2 &\leq b_3 \\ f_4(x_1, x_2) - B_4 y_2 &\leq b_4 \end{aligned} \right\} \text{Region 2} \\ & \left. \begin{aligned} f_5(x_1, x_2) - B_5 y_3 &\leq b_5 \\ f_6(x_1, x_2) - B_6 y_3 &\leq b_6 \\ f_7(x_1, x_2) - B_7 y_3 &\leq b_7 \end{aligned} \right\} \text{Region 3} \\ & y_1 + y_2 + y_3 \leq 2, \\ & x_1 \geq 0, \quad x_2 \geq 0, \\ & y_1, y_2, y_3 \text{ binary.} \end{aligned}$$

Note that we use the same binary variable y_j for each constraint defining one of the regions, and that the

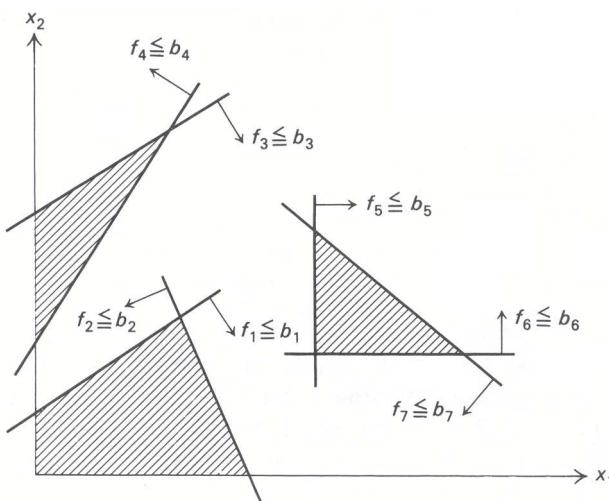


Figure 9.2 An example of compound alternatives.

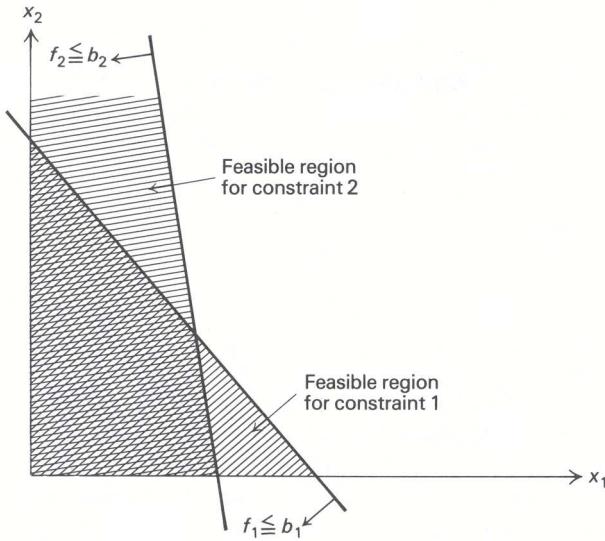


Figure 9.3 Geometry of alternative constraints.

constraint $y_1 + y_2 + y_3 \leq 2$ implies that the decision variables x_1 and x_2 lie in *at least one* of the required regions. Thus, for example, if $y_3 = 0$, then each of the constraints

$$f_5(x_1, x_2) \leq b_5, \quad f_6(x_1, x_2) \leq b_6, \quad \text{and} \quad f_7(x_1, x_2) \leq b_7$$

is satisfied.

The regions do not have to be disjoint before we can apply this technique. Even the simple alternative constraint

$$f_1(x_1, x_2) \leq b_1 \quad \text{or} \quad f_2(x_1, x_2) \leq b_2$$

shown in Fig. 9.3 contains overlapping regions.

Representing Nonlinear Functions

Nonlinear functions can be represented by integer-programming formulations. Let us analyze the most useful representations of this type.

i) Fixed Costs

Frequently, the objective function for a minimization problem contains fixed costs (preliminary design costs, fixed investment costs, fixed contracts, and so forth). For example, the cost of producing x units of a specific product might consist of a fixed cost of setting up the equipment and a variable cost per unit produced on the equipment. An example of this type of cost is given in Fig. 9.4.

Assume that the equipment has a capacity of B units. Define y to be a binary variable that indicates when the fixed cost is incurred, so that $y = 1$ when $x > 0$ and $y = 0$ when $x = 0$. Then the contribution to cost due to x may be written as

$$Ky + cx,$$

with the constraints:

$$\begin{aligned} x &\leq By, \\ x &\geq 0, \\ y &= 0 \quad \text{or} \quad 1. \end{aligned}$$

As required, these constraints imply that $x = 0$ when the fixed cost is not incurred, i.e., when $y = 0$. The constraints themselves do not imply that $y = 0$ if $x = 0$. But when $x = 0$, the minimization will clearly

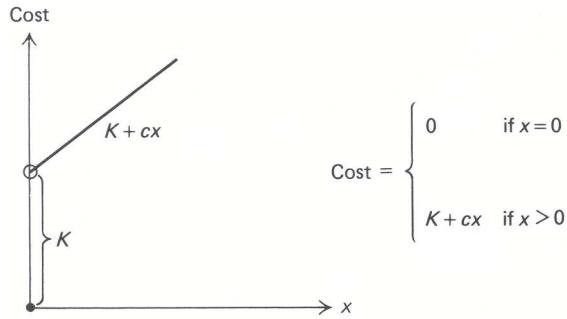


Figure 9.4 A fixed cost.

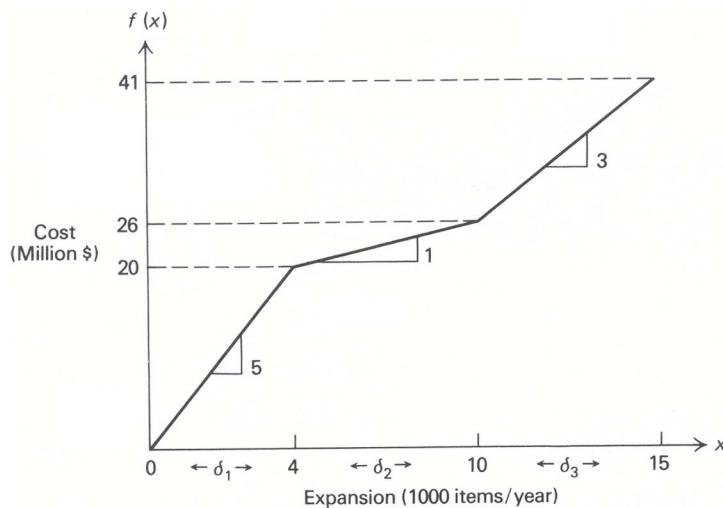


Figure 9.5 Modeling a piecewise linear curve.

select $y = 0$, so that the fixed cost is not incurred. Finally, observe that if $y = 1$, then the added constraint becomes $x \leq B$, which reflects the capacity limit on the production equipment.

ii) Piecewise Linear Representation

Another type of nonlinear function that can be represented by integer variables is a piecewise linear curve. Figure 9.5 illustrates a cost curve for plant expansion that contains three linear segments with variable costs of 5, 1, and 3 million dollars per 1000 items of expansion.

To model the cost curve, we express any value of x as the sum of three variables $\delta_1, \delta_2, \delta_3$, so that the cost for each of these variables is linear. Hence,

$$x = \delta_1 + \delta_2 + \delta_3,$$

where

$$\begin{aligned} 0 &\leq \delta_1 \leq 4, \\ 0 &\leq \delta_2 \leq 6, \\ 0 &\leq \delta_3 \leq 5; \end{aligned} \tag{10}$$

and the total variable cost is given by:

$$\text{Cost} = 5\delta_1 + \delta_2 + 3\delta_3.$$

Note that we have defined the variables so that:

δ_1 corresponds to the amount by which x exceeds 0, but is less than or equal to 4;

δ_2 is the amount by which x exceeds 4, but is less than or equal to 10; and

δ_3 is the amount by which x exceeds 10, but is less than or equal to 15.

If this interpretation is to be valid, we must also require that $\delta_1 = 4$ whenever $\delta_2 > 0$ and that $\delta_2 = 6$ whenever $\delta_3 > 0$. Otherwise, when $x = 2$, say, the cost would be minimized by selecting $\delta_1 = \delta_3 = 0$ and $\delta_2 = 2$, since the variable δ_2 has the smallest variable cost. However, these restrictions on the variables are simply conditional constraints and can be modeled by introducing binary variables, as before.

If we let

$$w_1 = \begin{cases} 1 & \text{if } \delta_1 \text{ is at its upper bound,} \\ 0 & \text{otherwise,} \end{cases}$$

$$w_2 = \begin{cases} 1 & \text{if } \delta_2 \text{ is at its upper bound,} \\ 0 & \text{otherwise,} \end{cases}$$

then constraints (10) can be replaced by

$$\begin{aligned} 4w_1 &\leq \delta_1 \leq 4, \\ 6w_2 &\leq \delta_2 \leq 6w_1, \\ 0 &\leq \delta_3 \leq 5w_2, \\ w_1 \quad \text{and} \quad w_2 &\text{ binary,} \end{aligned} \tag{11}$$

to ensure that the proper conditional constraints hold. Note that if $w_1 = 0$, then $w_2 = 0$, to maintain feasibility for the constraint imposed upon δ_2 , and (11) reduces to

$$0 \leq \delta_1 \leq 4, \quad \delta_2 = 0, \quad \text{and} \quad \delta_3 = 0.$$

If $w_1 = 1$ and $w_2 = 0$, then (11) reduces to

$$\delta_1 = 4, \quad 0 \leq \delta_2 \leq 6, \quad \text{and} \quad \delta_3 = 0.$$

Finally, if $w_1 = 1$ and $w_2 = 1$, then (11) reduces to

$$\delta_1 = 4, \quad \delta_2 = 6, \quad \text{and} \quad 0 \leq \delta_3 \leq 5.$$

Hence, we observe that there are three feasible combinations for the values of w_1 and w_2 :

$$w_1 = 0, \quad w_2 = 0 \quad \text{corresponding to } 0 \leq x \leq 4 \quad \text{since } \delta_2 = \delta_3 = 0;$$

$$w_1 = 1, \quad w_2 = 0 \quad \text{corresponding to } 4 \leq x \leq 10 \quad \text{since } \delta_1 = 4 \text{ and } \delta_3 = 0;$$

and

$$w_1 = 1, \quad w_2 = 1 \quad \text{corresponding to } 10 \leq x \leq 15 \quad \text{since } \delta_1 = 4 \text{ and } \delta_2 = 6.$$

The same general technique can be applied to piecewise linear curves with any number of segments. The general constraint imposed upon the variable δ_j for the j th segment will read:

$$L_j w_j \leq \delta_j \leq L_{j-1} w_{j-1},$$

where L_j is the length of the segment.

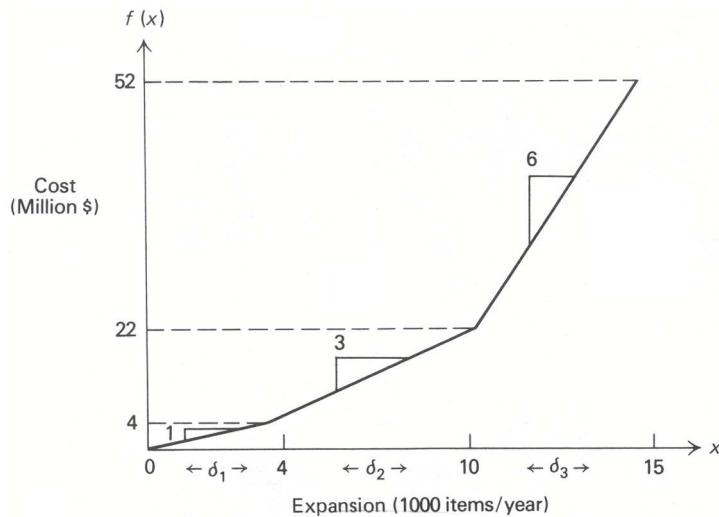


Figure 9.6 Diseconomies of scale.

iii) Diseconomies of Scale

An important special case for representing nonlinear functions arises when only diseconomies of scale apply—that is, when marginal costs are increasing for a minimization problem or marginal returns are decreasing for a maximization problem. Suppose that the expansion cost in the previous example now is specified by Fig. 9.6.

In this case, the cost is represented by

$$\text{Cost} = \delta_1 + 3\delta_2 + 6\delta_3,$$

subject only to the linear constraints without integer variables,

$$\begin{aligned} 0 &\leq \delta_1 \leq 4 \\ 0 &\leq \delta_2 \leq 6, \\ 0 &\leq \delta_3 \leq 5. \end{aligned}$$

The conditional constraints involving binary variables in the previous formulation can be ignored if the cost curve appears in a minimization objective function, since the coefficients of δ_1 , δ_2 , and δ_3 imply that it is always best to set $\delta_1 = 4$ before taking $\delta_2 > 0$, and to set $\delta_2 = 6$ before taking $\delta_3 > 0$. As a consequence, the integer variables have been avoided completely.

This representation without integer variables is not valid, however, if economies of scale are present; for example, if the function given in Fig. 9.6 appears in a maximization problem. In such cases, it would be best to select the third segment with variable δ_3 before taking the first two segments, since the returns are higher on this segment. In this instance, the model requires the binary-variable formulation of the previous section.

iv) Approximation of Nonlinear Functions

One of the most useful applications of the piecewise linear representation is for approximating nonlinear functions. Suppose, for example, that the expansion cost in our illustration is given by the heavy curve in Fig. 9.7.

If we draw linear segments joining selected points on the curve, we obtain a *piecewise linear approximation*, which can be used instead of the curve in the model. The piecewise approximation, of course, is represented by introducing integer variables as indicated above. By using more points on the curve, we can make the approximation as close as we desire.

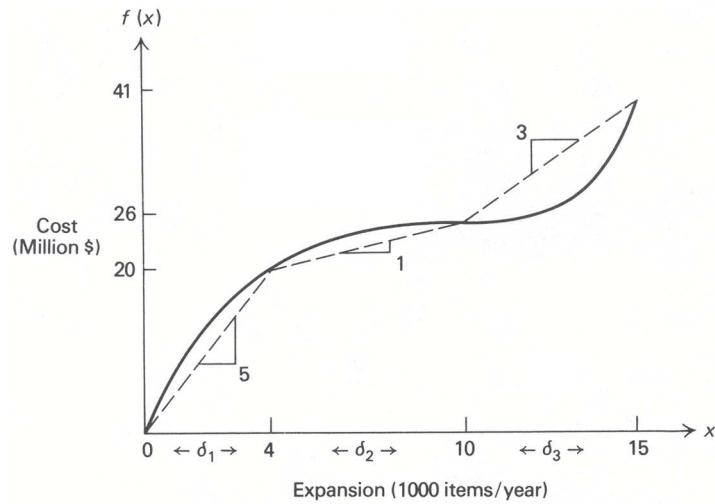


Figure 9.7 Approximation of a nonlinear curve.

9.3 A SAMPLE FORMULATION [†]

Proper placement of service facilities such as schools, hospitals, and recreational areas is essential to efficient urban design. Here we will present a simplified model for firehouse location. Our purpose is to show formulation devices of the previous section arising together in a meaningful context, rather than to give a comprehensive model for the location problem *per se*. As a consequence, we shall ignore many relevant issues, including uncertainty.

Assume that population is concentrated in I districts within the city and that district i contains p_i people. Preliminary analysis (land surveys, politics, and so forth) has limited the potential location of firehouses to J sites. Let $d_{ij} \geq 0$ be the distance from the center of district i to site j . We are to determine the “best” site selection and assignment of districts to firehouses. Let

$$y_j = \begin{cases} 1 & \text{if site } j \text{ is selected,} \\ 0 & \text{otherwise;} \end{cases}$$

and

$$x_{ij} = \begin{cases} 1 & \text{if district } i \text{ is assigned to site } j, \\ 0 & \text{otherwise.} \end{cases}$$

The basic constraints are that every district should be assigned to exactly one firehouse, that is,

$$\sum_{j=1}^J x_{ij} = 1 \quad (i = 1, 2, \dots, I),$$

and that no district should be assigned to an unused site, that is, $y_j = 0$ implies $x_{ij} = 0$ ($i = 1, 2, \dots, I$). The latter restriction can be modeled as alternative constraints, or more simply as:

$$\sum_{i=1}^I x_{ij} \leq y_j I \quad (j = 1, 2, \dots, J).$$

Since x_{ij} are binary variables, their sum never exceeds I , so that if $y_j = 1$, then constraint j is nonbinding. If $y_j = 0$, then $x_{ij} = 0$ for all i .

[†] This section may be omitted without loss of continuity.

Next note that d_i , the distance from district i to its assigned firehouse, is given by:

$$d_i = \sum_{j=1}^J d_{ij} x_{ij},$$

since one x_{ij} will be 1 and all others 0.

Also, the total population serviced by site j is:

$$s_j = \sum_{i=1}^I p_i x_{ij}.$$

Assume that a central district is particularly susceptible to fire and that either sites 1 and 2 or sites 3 and 4 can be used to protect this district. Then one of a number of similar restrictions might be:

$$y_1 + y_2 \geq 2 \quad \text{or} \quad y_3 + y_4 \geq 2.$$

We let y be a binary variable; then these alternative constraints become:

$$\begin{aligned} y_1 + y_2 &\geq 2y, \\ y_3 + y_4 &\geq 2(1 - y). \end{aligned}$$

Next assume that it costs $f_j(s_j)$ to build a firehouse at site j to service s_j people and that a total budget of B dollars has been allocated for firehouse construction. Then

$$\sum_{j=1}^J f_j(s_j) \leq B.$$

Finally, one possible social-welfare function might be to minimize the distance traveled to the district farthest from its assigned firehouse, that is, to:

Minimize D ,

where

$$D = \max d_i;$$

or, equivalently,[‡] to

Minimize D ,

subject to:

$$D \geq d_i \quad (i = 1, 2, \dots, I).$$

Collecting constraints and substituting above for d_i in terms of its defining relationship

$$d_i = \sum_{j=1}^J d_{ij} x_{ij},$$

we set up the full model as:

Minimize D ,

[‡] The inequalities $D \geq d_i$ imply that $D \geq \max d_i$. The minimization of D then ensures that it will actually be the maximum of the d_i .

subject to:

$$\begin{aligned}
 D - \sum_{j=1}^J d_{ij} x_{ij} &\geq 0 \quad (i = 1, 2, \dots, I), \\
 \sum_{j=1}^J x_{ij} &= 1 \quad (i = 1, 2, \dots, I), \\
 \sum_{i=1}^I x_{ij} &\leq y_j I \quad (j = 1, 2, \dots, J), \\
 S_j - \sum_{i=1}^I p_i x_{ij} &= 0 \quad (j = 1, 2, \dots, J), \\
 \sum_{j=1}^J f_j(s_j) &\leq B, \\
 y_1 + y_2 - 2y &\geq 0, \\
 y_3 + y_4 + 2y &\geq 2, \\
 x_{ij}, y_j, y &\text{ binary} \quad (i = 1, 2, \dots, I; j = 1, 2, \dots, J).
 \end{aligned}$$

At this point we might replace each function $f_j(s_j)$ by an integer-programming approximation to complete the model. Details are left to the reader. Note that if $f_j(s_j)$ contains a fixed cost, then new fixed-cost variables need not be introduced—the variable y_j serves this purpose.

The last comment, and the way in which the conditional constraint “ $y_j = 0$ implies $x_{ij} = 0$ ($i = 1, 2, \dots, I$)” has been modeled above, indicate that the formulation techniques of Section 9.2 should not be applied without thought. Rather, they provide a common framework for modeling and should be used in conjunction with good modeling “common sense.” In general, it is best to introduce as few integer variables as possible.

9.4 SOME CHARACTERISTICS OF INTEGER PROGRAMS—A SAMPLE PROBLEM

Whereas the simplex method is effective for solving linear programs, there is no single technique for solving integer programs. Instead, a number of procedures have been developed, and the performance of any particular technique appears to be highly problem-dependent. Methods to date can be classified broadly as following one of three approaches:

- i) enumeration techniques, including the branch-and-bound procedure;
- ii) cutting-plane techniques; and
- iii) group-theoretic techniques.

In addition, several composite procedures have been proposed, which combine techniques using several of these approaches. In fact, there is a trend in computer systems for integer programming to include a number of approaches and possibly utilize them all when analyzing a given problem. In the sections to follow, we shall consider the first two approaches in some detail. At this point, we shall introduce a specific problem and indicate some features of integer programs. Later we will use this example to illustrate and motivate the solution procedures. Many characteristics of this example are shared by the integer version of the custom-molder problem presented in Chapter 1.

The problem is to determine z^* where:

$$z^* = \max z = 5x_1 + 8x_2,$$

subject to:

$$\begin{aligned}x_1 + x_2 &\leq 6, \\5x_1 + 9x_2 &\leq 45, \\x_1, x_2 &\geq 0 \quad \text{and integer.}\end{aligned}$$

The feasible region is sketched in Fig. 9.8. Dots in the shaded region are feasible integer points.

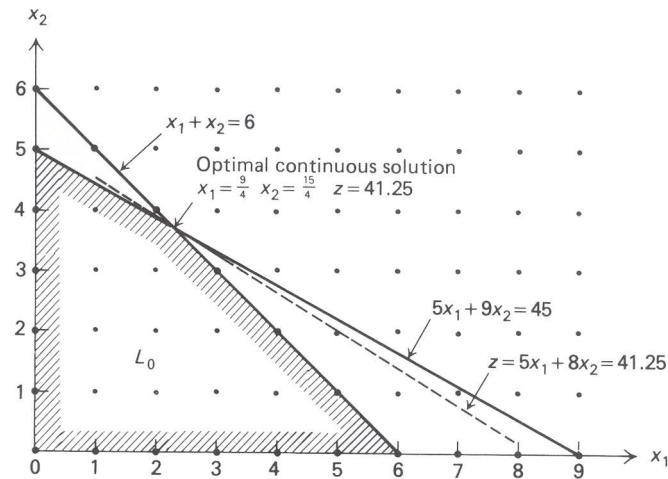


Figure 9.8 An integer programming example.

If the integrality restrictions on variables are dropped, the resulting problem is a linear program. We will call it the *associated linear program*. We may easily determine its optimal solution graphically. Table 9.1 depicts some of the features of the problem.

Table 9.1 Problem features.

	Continuous optimum	Round off	Nearest feasible point	Integer optimum
x_1	$\frac{9}{4} = 2.25$	2	2	0
x_2	$\frac{15}{4} = 3.75$	4	3	5
z	41.25	Infeasible	34	40

Observe that the optimal integer-programming solution is not obtained by *rounding* the linear-programming solution. The closest point to the optimal linear-programmable solution is not even feasible. Also, note that the nearest feasible integer point to the linear-programmable solution is far removed from the optimal integer point. Thus, it is not sufficient simply to round linear-programming solutions. In fact, by scaling the righthand-side and cost coefficients of this example properly, we can construct a problem for which the optimal integer-programming solution lies as far as we like from the rounded linear-programming solution, in either z value or distance on the plane.

In an example as simple as this, almost any solution procedure will be effective. For instance, we could easily enumerate all the integer points with $x_1 \leq 9$, $x_2 \leq 6$, and select the best feasible point. In practice, the number of points to be considered is likely to prohibit such an exhaustive enumeration of potentially feasible points, and a more sophisticated procedure will have to be adopted.

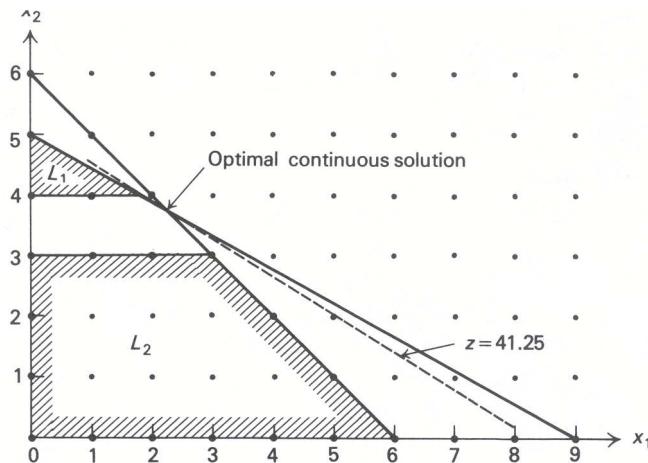


Figure 9.9 Subdividing the feasible region.

9.5 BRANCH-AND-BOUND

Branch-and-bound is essentially a strategy of “divide and conquer.” The idea is to partition the feasible region into more manageable subdivisions and then, if required, to further partition the subdivisions. In general, there are a number of ways to divide the feasible region, and as a consequence there are a number of branch-and-bound algorithms. We shall consider one such technique, for problems with only binary variables, in Section 9.7. For historical reasons, the technique that will be described next usually is referred to as *the* branch-and-bound procedure.

Basic Procedure

An integer linear program is a linear program further constrained by the integrality restrictions. Thus, in a maximization problem, the value of the objective function, at the linear-program optimum, will always be an upper bound on the optimal integer-programming objective. In addition, any integer feasible point is always a lower bound on the optimal linear-program objective value.

The idea of branch-and-bound is to utilize these observations to systematically subdivide the linear-programming feasible region and make assessments of the integer-programming problem based upon these subdivisions. The method can be described easily by considering the example from the previous section. At first, the linear-programming region is not subdivided: The integrality restrictions are dropped and the associated linear program is solved, giving an optimal value z^0 . From our remark above, this gives the upper bound on z^* , $z^* \leq z^0 = 41\frac{1}{4}$. Since the coefficients in the objective function are integral, z^* must be integral and this implies that $z^* \leq 41$.

Next note that the linear-programming solution has $x_1 = 2\frac{1}{4}$ and $x_2 = 3\frac{3}{4}$. Both of these variables must be integer in the optimal solution, and we can divide the feasible region in an attempt to *make* either integral. We know that, in any integer programming solution, x_2 must be either an integer ≤ 3 or an integer ≥ 4 . Thus, our first subdivision is into the regions where $x_2 \leq 3$ and $x_2 \geq 4$ as displayed by the shaded regions L_1 and L_2 in Fig. 9.9. Observe that, by making the subdivisions, we have excluded the old linear-program solution. (If we selected x_1 instead, the region would be subdivided with $x_1 \leq 2$ and $x_1 \geq 3$.)

The results up to this point are pictured conveniently in an *enumeration tree* (Fig. 9.10). Here L_0 represents the associated linear program, whose optimal solution has been included within the L_0 box, and the upper bound on z^* appears to the right of the box. The boxes below correspond to the new subdivisions; the constraints that subdivide L_0 are included next to the lines joining the boxes. Thus, the constraints of L_1 are those of L_0 together with the constraint $x_2 \geq 4$, while the constraints of L_2 are those of L_0 together with the constraint $x_2 \leq 3$.

The strategy to be pursued now may be apparent: Simply treat each subdivision as we did the original problem. Consider L_1 first. Graphically, from Fig. 9.9 we see that the optimal linear-programming solution

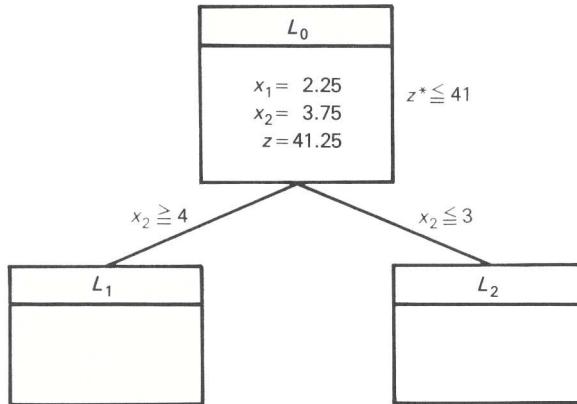
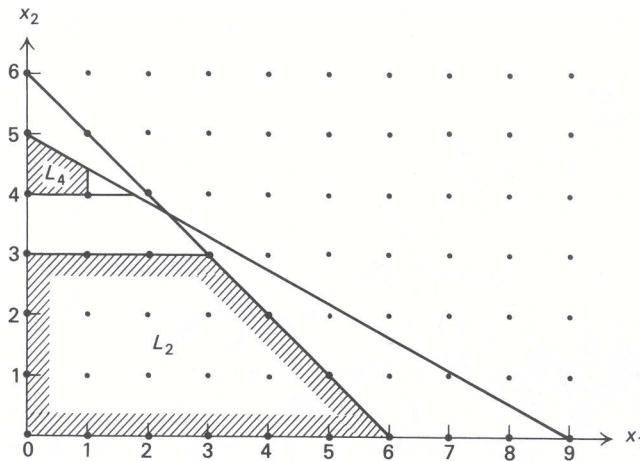


Figure 9.10 Enumeration tree.

Figure 9.11 Subdividing the region L_1 .

lies on the second constraint with $x_2 = 4$, giving $x_1 = \frac{1}{5}(45 - 9(4)) = \frac{9}{5}$ and an objective value $z = 5(\frac{9}{5}) + 8(4) = 41$. Since x_1 is not integer, we subdivide L_1 further, into the regions L_3 with $x_1 \geq 2$ and L_4 with $x_1 \leq 1$. L_3 is an infeasible problem and so this branch of the enumeration tree no longer needs to be considered.

The enumeration tree now becomes that shown in Fig. 9.12. Note that the constraints of any subdivision are obtained by tracing back to L_0 . For example, L_4 contains the original constraints together with $x_2 \geq 4$ and $x_1 \leq 1$. The asterisk (*) below box L_3 indicates that the region need not be subdivided or, equivalently, that the tree will not be extended from this box.

At this point, subdivisions L_2 and L_4 must be considered. We may select one arbitrarily; however, in practice, a number of useful heuristics are applied to make this choice. For simplicity, let us select the subdivision most recently generated, here L_4 . Analyzing the region, we find that its optimal solution has

$$x_1 = 1, \quad x_2 = \frac{1}{9}(45 - 5) = \frac{40}{9}.$$

Since x_2 is not integer, L_4 must be further subdivided into L_5 with $x_2 \leq 4$, and L_6 with $x_2 \geq 5$, leaving L_2 , L_5 and L_6 yet to be considered.

Treating L_5 first (see Fig. 9.13), we see that its optimum has $x_1 = 1$, $x_2 = 4$, and $z = 37$. Since this is the best linear-programming solution for L_5 and the linear program contains every integer solution in L_5 , no integer point in that subdivision can give a larger objective value than this point. Consequently, other points

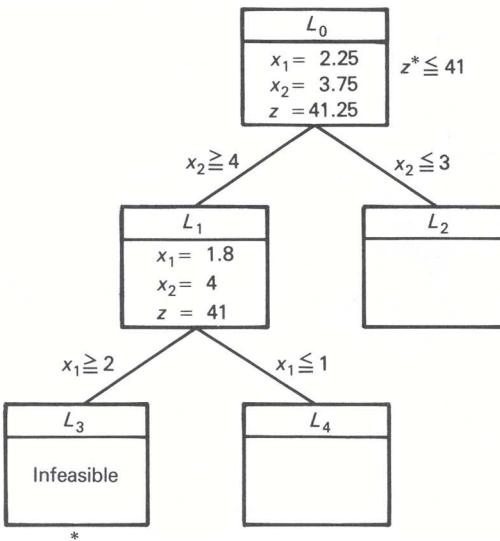


Figure 9.12

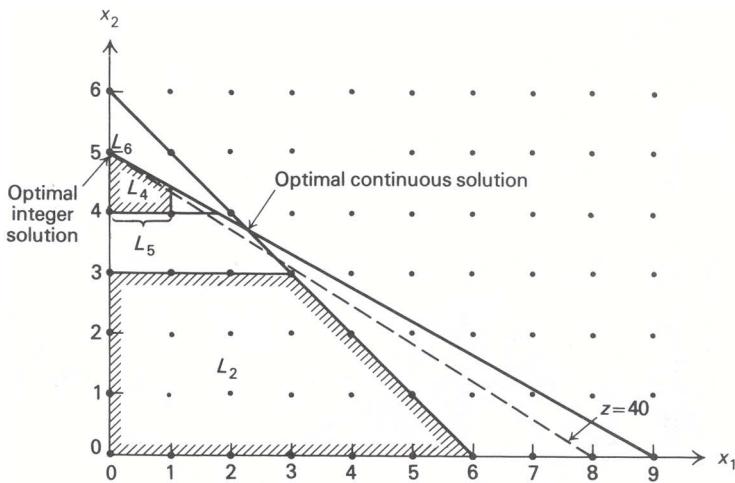


Figure 9.13 Final subdivisions for the example.

in L_5 need never be considered and L_5 need not be subdivided further. In fact, since $x_1 = 1, x_2 = 4, z = 37$, is a feasible solution to the original problem, $z^* \geq 37$ and we now have the bounds $37 \leq z^* \leq 41$. Without further analysis, we could terminate with the integer solution $x_1 = 1, x_2 = 4$, knowing that the objective value of this point is within 10 percent of the true optimum. For convenience, the lower bound $z^* \geq 37$ just determined has been appended to the right of the L_5 box in the enumeration tree (Fig. 9.14).

Although $x_1 = 1, x_2 = 4$ is the best integer point in L_5 , the regions L_2 and L_6 might contain better feasible solutions, and we must continue the procedure by analyzing these regions. In L_6 , the only feasible point is $x_1 = 0, x_2 = 5$, giving an objective value $z = +40$. This is better than the previous integer point and thus the lower bound on z^* improves, so that $40 \leq z^* \leq 41$. We could terminate with this integer solution knowing that it is within 2.5 percent of the true optimum. However, L_2 could contain an even better integer solution.

The linear-programming solution in L_2 has $x_1 = x_2 = 3$ and $z = 39$. This is the best integer point in L_2 but is not as good as $x_1 = 0, x_2 = 5$, so the later point (in L_6) must indeed be optimal. It is interesting to note that, even if the solution to L_2 did not give x_1 and x_2 integer, but had $z < 40$, then no feasible (and, in particular, no integer point) in L_2 could be as good as $x_1 = 0, x_2 = 5$, with $z = 40$. Thus, again $x_1 = 0, x_2 = 5$ would be known to be optimal. This observation has important computational implications,

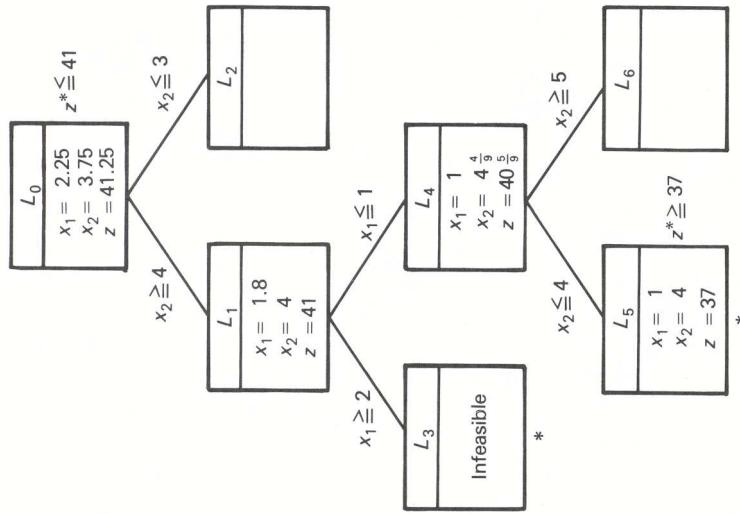


Figure 9.14

since it is not necessary to drive every branch in the enumeration tree to an integer or infeasible solution, but only to an objective value below the best integer solution.

The problem now is solved and the entire solution procedure can be summarized by the enumeration tree in Fig. 9.15.

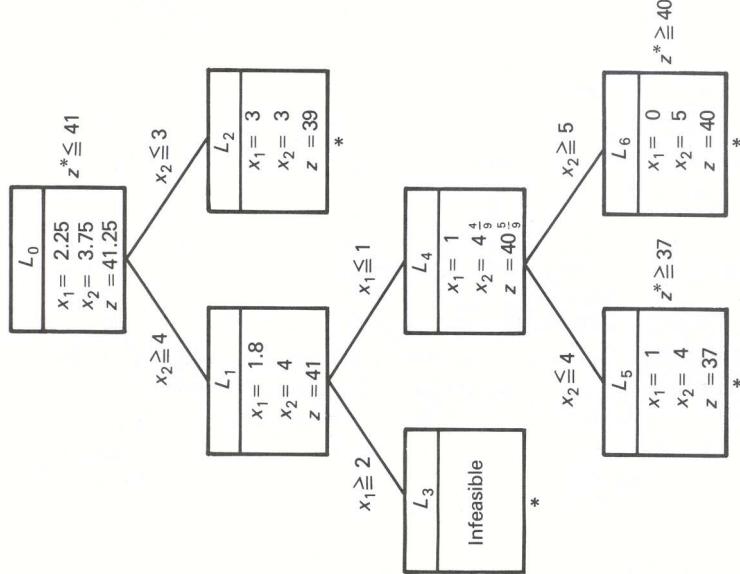


Figure 9.15

Further Considerations

There are three points that have yet to be considered with respect to the branch-and-bound procedure:

- Can the linear programs corresponding to the subdivisions be solved efficiently?
- What is the best way to subdivide a given region, and which unanalyzed subdivision should be considered next?

- iii) Can the upper bound ($z = 41$, in the example) on the optimal value z^* of the integer program be improved while the problem is being solved?

The answer to the first question is an unqualified *yes*. When moving from a region to one of its subdivisions, we add one constraint that is not satisfied by the optimal linear-programming solution over the parent region. Moreover, this was one motivation for the dual simplex algorithm, and it is natural to adopt that algorithm here.

Referring to the sample problem will illustrate the method. The first two subdivisions L_1 and L_2 in that example were generated by adding the following constraints to the original problem:

$$\begin{aligned} \text{For subdivision 1 : } & x_2 \geq 4 \quad \text{or} \quad x_2 - s_3 = 4 \quad (s_3 \geq 0); \\ \text{For subdivision 2 : } & x_2 \leq 3 \quad \text{or} \quad x_2 + s_4 = 3 \quad (s_4 \geq 0). \end{aligned}$$

In either case we add the new constraint to the optimal linear-programming tableau. For subdivision 1, this gives:

$$\left. \begin{array}{rcl} (-z) & -\frac{5}{4}s_1 - \frac{3}{4}s_2 & = -41\frac{1}{4}, \\ x_1 & +\frac{9}{4}s_1 - \frac{1}{4}s_2 & = \frac{9}{4}, \\ x_2 & -\frac{5}{4}s_1 + \frac{1}{4}s_2 & = \frac{15}{4}, \end{array} \right\} \begin{array}{l} \text{Constraints from the} \\ \text{optimal canonical} \\ \text{form} \end{array}$$

$$\begin{array}{rcl} -x_2 & + s_3 & = -4, \\ x_1, x_2, s_1, s_2, s_3 & \geq 0, \end{array} \quad \begin{array}{l} \text{Added constraint} \end{array}$$

where s_1 and s_2 are slack variables for the two constraints in the original problem formulation. Note that the new constraint has been multiplied by -1 , so that the slack variable s_3 can be used as a basic variable. Since the basic variable x_2 appears with a nonzero coefficient in the new constraint, though, we must pivot to isolate this variable in the second constraint to re-express the system as:

$$\begin{array}{rcl} (-z) & -\frac{5}{4}s_1 - \frac{3}{4}s_2 & = -41\frac{1}{4}, \\ x_1 & +\frac{9}{4}s_1 - \frac{1}{4}s_2 & = \frac{9}{4}, \\ x_2 & -\frac{5}{4}s_1 + \frac{1}{4}s_2 & = \frac{15}{4}, \\ \circled{-\frac{5}{4}s_1} + \frac{1}{4}s_2 + s_3 & = & -\frac{1}{4}, \end{array}$$

$$x_1, x_2, s_1, s_2, s_3 \geq 0.$$

These constraints are expressed in the proper form for applying the dual simplex algorithm, which will pivot next to make s_1 the basic variable in the third constraint. The resulting system is given by:

$$\begin{array}{rcl} (-z) & -s_2 - s_3 & = -41, \\ x_1 & +\frac{1}{5}s_2 + \frac{9}{5}s_3 & = \frac{9}{5}, \\ x_2 & -s_3 & = 4, \\ s_1 - \frac{1}{5}s_2 - \frac{4}{5}s_3 & = & \frac{1}{5}, \end{array}$$

$$x_1, x_2, s_1, s_2, s_3 \geq 0.$$

This tableau is optimal and gives the optimal linear-programming solution over the region L_1 as $x_1 = \frac{9}{5}$, $x_2 = 4$, and $z = 41$. The same procedure can be used to determine the optimal solution in L_2 .

When the linear-programming problem contains many constraints, this approach for recovering an optimal solution is very effective. After adding a new constraint and making the slack variable for that constraint basic, we always have a starting solution for the dual-simplex algorithm with only one basic variable negative. Usually, only a few dual-simplex pivoting operations are required to obtain the optimal solution. Using the primal-simplex algorithm generally would require many more computations.

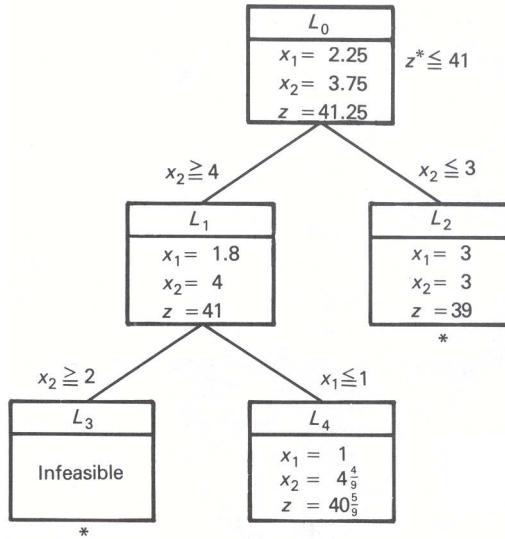


Figure 9.16

Issue (ii) raised above is very important since, if we can make our choice of subdivisions in such a way as to rapidly obtain a good (with luck, near-optimal) integer solution \hat{z} , then we can eliminate many potential subdivisions immediately. Indeed, if any region has its linear programming value $z \leq \hat{z}$, then the objective value of no integer point in that region can exceed \hat{z} and the region need not be subdivided. There is no universal method for making the required choice, although several heuristic procedures have been suggested, such as selecting the subdivision with the largest optimal linear-programming value.[†]

Rules for determining which fractional variables to use in constructing subdivisions are more subtle. Recall that any fractional variable can be used to generate a subdivision. One procedure utilized is to look ahead one step in the dual-simplex method for every possible subdivision to see which is most promising. The details are somewhat involved and are omitted here. For expository purposes, we have selected the fractional variable arbitrarily.

Finally, the upper bound \bar{z} on the value z^* of the integer program can be improved as we solve the problem. Suppose for example, that subdivision L_2 was analyzed before subdivisions L_5 or L_6 in our sample problem. The enumeration tree would be as shown in Fig. 9.16.

At this point, the optimal solution must lie in either L_2 or L_4 . Since, however, the largest value for any feasible point in either of these regions is $40\frac{5}{9}$, the optimal value for the problem z^* cannot exceed $40\frac{5}{9}$. Because z^* must be integral, this implies that $z^* \leq 40$ and the upper bound has been improved from the value 41 provided by the solution to the linear program on L_0 . In general, the upper bound is given in this way as the largest value of any “hanging” box (one that has not been divided) in the enumeration tree.

Summary

The essential idea of branch-and-bound is to subdivide the feasible region to develop bounds $\underline{z} < z^* < \bar{z}$ on z^* . For a maximization problem, the lower bound \underline{z} is the highest value of any feasible integer point encountered. The upper bound is given by the optimal value of the associated linear program or by the largest value for the objective function at any “hanging” box. After considering a subdivision, we must *branch* to (move to) another subdivision and analyze it. Also, if *either*

[†] One common method used in practice is to consider subdivisions on a last-generated-first-analyzed basis. We used this rule in our previous example. Note that data to initiate the dual-simplex method mentioned above must be stored for each subdivision that has yet to be analyzed. This data usually is stored in a list, with new information being added to the top of the list. When required, data then is extracted from the *top* of this list, leading to the last-generated-first-analyzed rule. Observe that when we subdivide a region into two subdivisions, one of these subdivisions will be analyzed next. The data required for this analysis already will be in the computer core and need not be extracted from the list.

- i) the linear program over L_j is infeasible;
- ii) the optimal linear-programming solution over L_j is integer; or
- iii) the value of the linear-programming solution z^j over L_j satisfies $z^j \leq \underline{z}$ (if maximizing),

then L_j need not be subdivided. In these cases, integer-programming terminology says that L_j has been *fathomed*.[†] Case (i) is termed fathoming by infeasibility, (ii) fathoming by integrality, and (iii) fathoming by bounds.

The flow chart in Fig. 9.17 summarizes the general procedure.

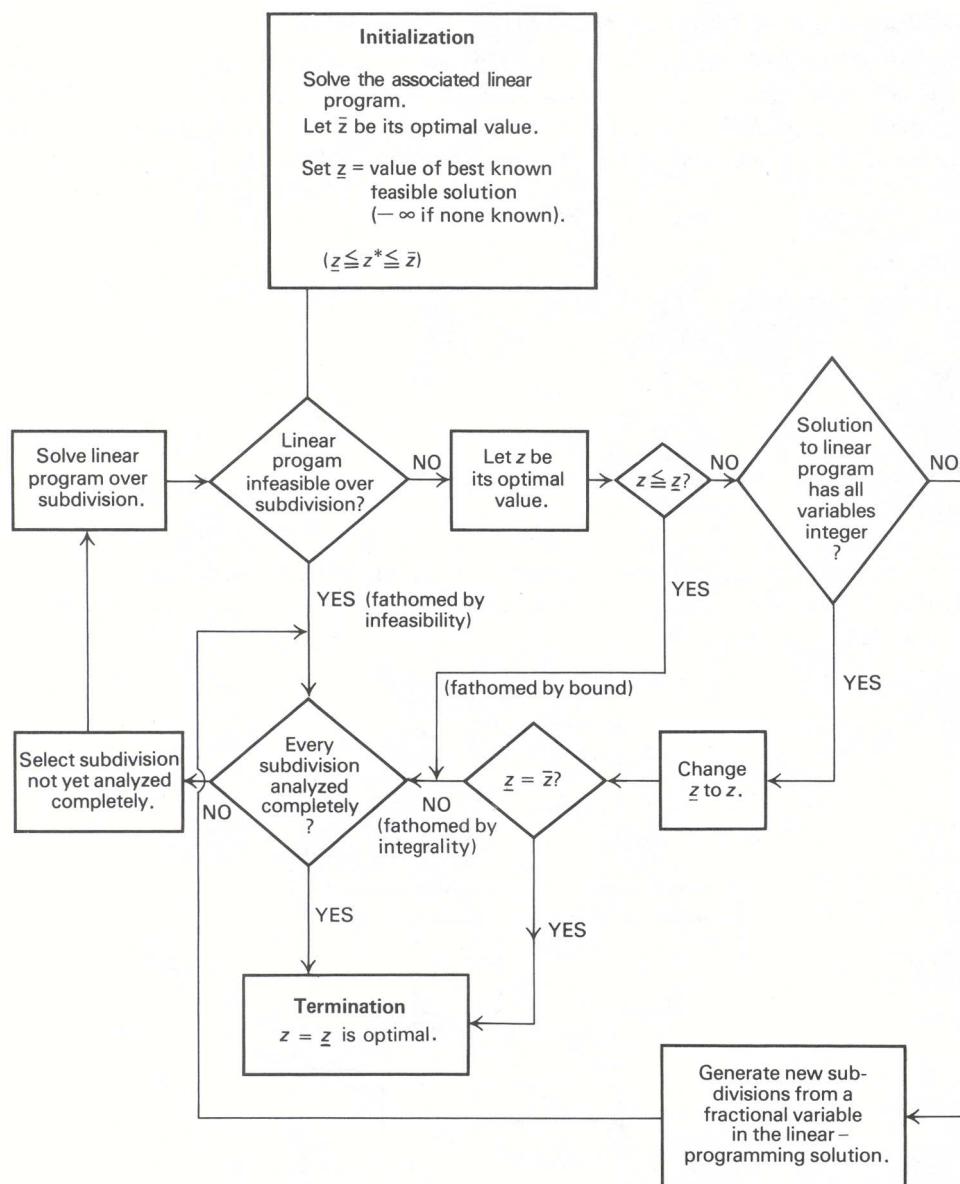


Figure 9.17 Branch-and-bound for integer-programming maximization.

[†] To *fathom* is defined as “to get to the bottom of; to understand thoroughly.” In this chapter, *fathomed* might be more appropriately defined as “understood enough or already considered.”

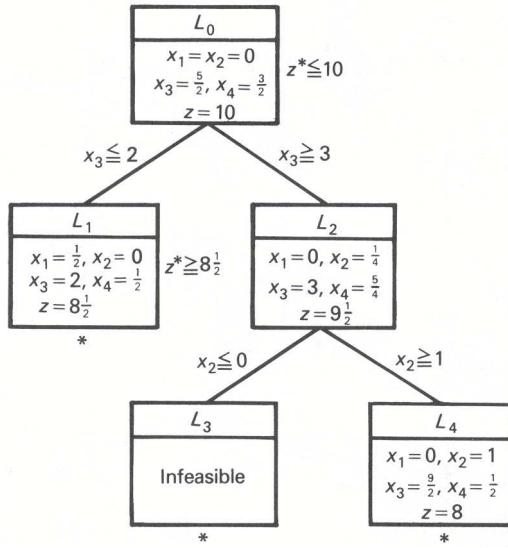


Figure 9.18

9.6 BRANCH-AND-BOUND FOR MIXED-INTEGER PROGRAMS

The branch-and-bound approach just described is easily extended to solve problems in which some, but not all, variables are constrained to be integral. Subdivisions then are generated solely by the integral variables. In every other way, the procedure is the same as that specified above. A brief example will illustrate the method.

$$z^* = \max z = -3x_1 - 2x_2 + 10,$$

subject to:

$$\begin{aligned} x_1 - 2x_2 + x_3 &= \frac{5}{2}, \\ 2x_1 + x_2 + x_4 &= \frac{3}{2}, \\ x_j &\geq 0 \quad (j = 1, 2, 3, 4), \\ x_2 \text{ and } x_3 &\text{ integer.} \end{aligned}$$

The problem, as stated, is in canonical form, with x_3 and x_4 optimal basic variables for the associated linear program.

The continuous variable x_4 cannot be used to generate subdivisions since any value of $x_4 \geq 0$ potentially can be optimal. Consequently, the subdivisions must be defined by $x_3 \leq 2$ and $x_3 \geq 3$. The complete procedure is summarized by the enumeration tree in Fig. 9.18.

The solution in L_1 satisfies the integrality restrictions, so $z^* \geq z = 8\frac{1}{2}$. The only integral variable with a fractional value in the optimal solution of L_2 is x_2 , so subdivisions L_3 and L_4 are generated from this variable. Finally, the optimal linear-programming value of L_4 is 8, so no feasible mixed-integer solution in that region can be better than the value $8\frac{1}{2}$ already generated. Consequently, that region need not be subdivided and the solution in L_1 is optimal.

The dual-simplex iterations that solve the linear programs in L_1, L_2, L_3 , and L_4 are given below in Tableau 1. The variables s_j in the tableaus are the slack variables for the constraints added to generate the subdivisions. The coefficients in the appended constraints are determined as we mentioned in the last section, by eliminating the basic variables x_j from the new constraint that is introduced. To follow the iterations, recall that in the dual-simplex method, pivots are made on negative elements in the generating row; if all elements in this row are *positive*, as in region L_3 , then the problem is infeasible.

Tableau 1

L_1	Basic variables	Current values	x_1	x_2	x_3	x_4	s_1
	($-z$)	-10	-3	-2			
	x_3	$\frac{5}{2}$	1	-2	1		
	x_4	$\frac{3}{2}$	2	1		1	
	s_1	$-\frac{1}{2}$	(-1)	2			1

↑

L_1	Basic variables	Current values	x_1	x_2	x_3	x_4	s_1
	($-z$)	$-8\frac{1}{2}$		-8			-3
	x_3	2		0	1		1
	x_4	$\frac{1}{2}$		5		1	2
	x_1	$\frac{1}{2}$	1	-2			-1

(i.e., $x_3 \leq 2$)

L_2	Basic variables	Current values	x_1	x_2	x_3	x_4	s_2
	($-z$)	-10	-3	-2			
	x_3	$\frac{5}{2}$	1	-2	1		
	x_4	$\frac{3}{2}$	2	1		1	
	s_2	$-\frac{1}{2}$	1	(-2)			1

↑

L_2	Basic variables	Current values	x_1	x_2	x_3	x_4	s_2
	($-z$)	$-9\frac{1}{2}$	-4				-1
	x_3	3	0		1		-1
	x_4	$\frac{5}{4}$	$\frac{5}{2}$			1	$\frac{1}{2}$
	x_2	$\frac{1}{4}$	$-\frac{1}{2}$	1			$-\frac{1}{2}$

(i.e., $x_3 \geq 3$)

9.7 IMPLICIT ENUMERATION

A special branch-and-bound procedure can be given for integer programs with only binary variables. The algorithm has the advantage that it requires no linear-programming solutions. It is illustrated by the following example:

$$z^* = \max z = -8x_1 - 2x_2 - 4x_3 - 7x_4 - 5x_5 + 10,$$

subject to:

$$\begin{aligned} -3x_1 - 3x_2 + x_3 + 2x_4 + 3x_5 &\leq -2, \\ -5x_1 - 3x_2 - 2x_3 - x_4 + x_5 &\leq -4, \end{aligned}$$

$$x_j = 0 \quad \text{or} \quad 1 \quad (j = 1, 2, \dots, 5).$$

One way to solve such problems is complete enumeration. List all possible binary combinations of the variables and select the best such point that is feasible. The approach works very well on a small problem such as this, where there are only a few potential 0–1 combinations for the variables, here 32. In general, though, an n -variable problem contains 2^n 0–1 combinations; for large values of n , the exhaustive approach is prohibitive. Instead, one might implicitly consider every binary combination, just as every integer point was implicitly *considered*, but not necessarily evaluated, for the general problem via branch-and-bound.

Recall that in the ordinary branch-and-bound procedure, subdivisions were analyzed by maintaining the linear constraints and dropping the integrality restrictions. Here, we adopt the opposite tactic of always

Tableau 1 (Continued)

L_3	Basic variables	Current values	x_1	x_2	x_3	x_4	s_2	s_3
	($-z$)	$-9\frac{1}{2}$	-4				-1	
	x_3	3	0		1		-1	
	x_4	$\frac{5}{4}$	$\frac{5}{2}$			1	$\frac{1}{2}$	
	x_2	$\frac{1}{4}$	$-\frac{1}{2}$	1			$-\frac{1}{2}$	
	s_3	$-\frac{1}{4}$	$\frac{1}{2}$				$\frac{1}{2}$	1

(i.e., $x_2 \leq 0$)

L_4	Basic variables	Current values	x_1	x_2	x_3	x_4	s_2	s_4
	($-z$)	$-9\frac{1}{2}$	-4				-1	
	x_3	3	0		1		-1	
	x_4	$\frac{5}{4}$	$\frac{5}{2}$			1	$\frac{1}{2}$	
	x_2	$\frac{1}{4}$	$-\frac{1}{2}$	1			$-\frac{1}{2}$	
	s_4	$-\frac{3}{4}$	$-\frac{1}{2}$				$-\frac{1}{2}$	1

↑

Basic variables	Current values	x_1	x_2	x_3	x_4	s_2	s_4
($-z$)	-8	-3					-2
x_3	$\frac{9}{2}$	1		1			-2
x_4	$\frac{1}{2}$	2			1		1
x_2	1	0	1				-1
s_2	$\frac{3}{2}$	1				1	-2

maintaining the 0–1 restrictions, but ignoring the linear inequalities.

The idea is to utilize a branch-and-bound (or subdivision) process to fix some of the variables at 0 or 1. The variables remaining to be specified are called *free variables*. Note that, if the inequality constraints are ignored, the objective function is maximized by setting the free variables to zero, since their objective-function coefficients are negative. For example, if x_1 and x_4 are fixed at 1 and x_5 at 0, then the free variables are x_2 and x_3 . Ignoring the inequality constraints, the resulting problem is:

$$\max [-8(1) - 2x_2 - 4x_3 - 7(1) - 5(0) + 10] = \max [-2x_2 - 4x_3 - 5],$$

subject to:

$$x_2 \quad \text{and} \quad x_3 \quad \text{binary.}$$

Since the free variables have negative objective-function coefficients, the maximization sets $x_2 = x_3 = 0$. The simplicity of this trivial optimization, as compared to a more formidable linear program, is what we would like to exploit.

Returning to the example, we start with *no* fixed variables, and consequently every variable is free and set to zero. The solution does not satisfy the inequality constraints, and we must subdivide to search for feasible solutions. One subdivision choice might be:

For subdivision 1 : $x_1 = 1$,

For subdivision 2 : $x_1 = 0$.

Now variable x_1 is fixed in each subdivision. By our observations above, if the inequalities are ignored, the optimal solution over each subdivision has $x_2 = x_3 = x_4 = x_5 = 0$. The resulting solution in subdivision 1 gives

$$z = -8(1) - 2(0) - 4(0) - 7(0) - 5(0) + 10 = 2,$$

and happens to satisfy the inequalities, so that the optimal solution to the original problem is *at least* 2, $z^* \geq 2$. Also, subdivision 1 has been fathomed: The above solution is best among all 0–1 combinations with $x_1 = 1$; thus it must be best among those satisfying the inequalities. No other feasible 0–1 combination in subdivision 1 needs to be evaluated explicitly. These combinations have been considered implicitly.

The solution with $x_2 = x_3 = x_4 = x_5 = 0$ in subdivision 2 is the same as the original solution with every variable at zero, and is infeasible. Consequently, the region must be subdivided further, say with $x_2 = 1$ or $x_2 = 0$, giving:

$$\begin{aligned} \text{For subdivision 3 : } & x_1 = 0, x_2 = 1; \\ \text{For subdivision 4 : } & x_1 = 0, x_2 = 0. \end{aligned}$$

The enumeration tree to this point is as given in Fig. 9.19.

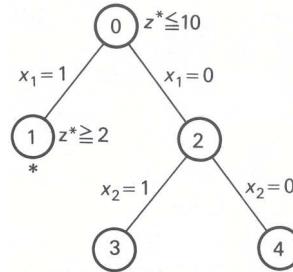


Figure 9.19

Observe that this tree differs from the enumeration trees of the previous sections. For the earlier procedures, the linear-programming solution used to analyze each subdivision was specified explicitly in a box. Here the 0–1 solution (ignoring the inequalities) used to analyze subdivisions is not stated explicitly, since it is known simply by setting free variables to zero. In subdivision ③, for example, $x_1 = 0$ and $x_2 = 1$ are fixed, and the free variables x_3, x_4 and x_5 are set to zero.

Continuing to fix variables and subdivide in this fashion produces the complete tree shown in Fig. 9.20.

The tree is not extended after analyzing subdivisions 4, 5, 7, 9, and 10, for the following reasons.

- i) At ⑤, the solution $x_1 = 0, x_2 = x_3 = 1$, with free variables $x_4 = x_5 = 0$, is feasible, with $z = 4$, thus providing an improved lower bound on z^* .
- ii) At ⑦, the solution $x_1 = x_3 = 0, x_2 = x_4 = 1$, and free variable $x_5 = 0$, has $z = 1 < 4$, so that no solution in that subdivision can be as good as that generated at ⑤.
- iii) At ⑨ and ⑩, every free variable is fixed. In each case, the subdivisions contain only a single point, which is infeasible, and further subdivision is not possible.
- iv) At ④, the second inequality (with fixed variables $x_1 = x_2 = 0$) reads:

$$-2x_3 - x_4 + x_5 \leq -4.$$

No 0–1 values of x_3, x_4 , or x_5 “completing” the fixed variables $x_1 = x_2 = 0$ satisfy this constraint, since the lowest value for the lefthand side of this equation is -3 when $x_3 = x_4 = 1$ and $x_5 = 0$. The subdivision then has no feasible solution and need not be analyzed further.

The last observation is completely general. If, at any point after substituting for the fixed variables, the sum of the remaining negative coefficients in any constraint exceeds the righthand side, then the region defined by these fixed variables has no feasible solution. Due to the special nature of the 0–1 problem, there are a number of other such tests that can be utilized to reduce the number of subdivisions generated. The efficiency of these tests is measured by weighing the time needed to perform them against the time saved by fewer subdivisions.

The techniques used here apply to any integer-programming problem involving only binary variables, so that implicit enumeration is an alternative branch-and-bound procedure for this class of problems. In this case, subdivisions are fathomed if any of three conditions hold:

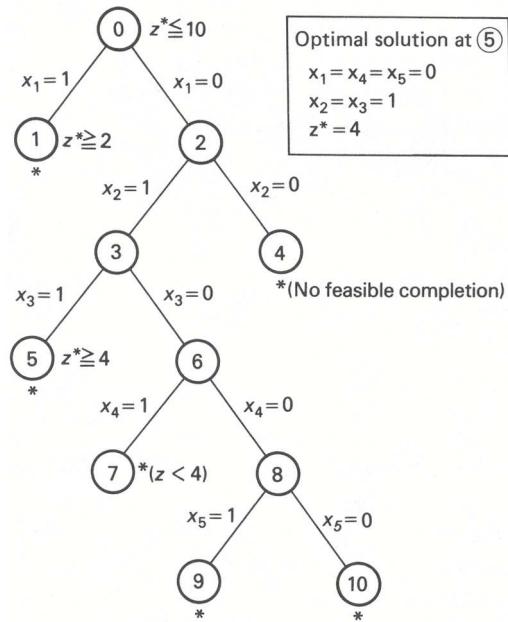


Figure 9.20

- i) the integer program is known to be infeasible over the subdivision, for example, by the above infeasibility test;
- ii) the 0–1 solution obtained by setting free variables to zero satisfies the linear inequalities; or
- iii) the objective value obtained by setting free variables to zero is no larger than the best feasible 0–1 solution previously generated.

These conditions correspond to the three stated earlier for fathoming in the usual branch-and-bound procedure. If a region is not fathomed by one of these tests, implicit enumeration subdivides that region by selecting any free variable and fixing its values to 0 or 1.

Our arguments leading to the algorithm were based upon stating the original 0–1 problem in the following standard form:

1. the objective is a maximization with all coefficients negative; and
2. constraints are specified as “less than or equal to” inequalities.

As usual, minimization problems are transformed to maximization by multiplying cost coefficients by -1 . If x_j appears in the maximization form with a positive coefficient, then the variable substitution $x_j = 1 - x'_j$ everywhere in the model leaves the binary variable x'_j with a negative objective-function coefficient. Finally, “greater than or equal to” constraints can be multiplied by -1 to become “less than or equal to” constraints; and general equality constraints are converted to inequalities by the special technique discussed in Exercise 17 of Chapter 2.

Like the branch-and-bound procedure for general integer programs, the way we choose to subdivide regions can have a profound effect upon computations. In implicit enumeration, we begin with the zero solution $x_1 = x_2 = \dots = x_n = 0$ and generate other solutions by setting variables to 1. One natural approach is to subdivide based upon the variable with highest objective contribution. For the sample problem, this would imply subdividing initially with $x_2 = 1$ or $x_2 = 0$.

Another approach often used in practice is to try to drive toward feasibility as soon as possible. For instance, when $x_1 = 0$, $x_2 = 1$, and $x_3 = 0$ are fixed in the example problem, we could subdivide based upon either x_4 or x_5 . Setting x_4 or x_5 to 1 and substituting for the fixed variables, we find that the constraints become:

$$\begin{array}{ll} x_4 = 1, & x_5(\text{free}) = 0 : \\ -3(0) - 3(1) + (0) + 2(1) + 3(0) \leq -2, & -3(0) - 3(1) + (0) + 2(0) + 3(1) \leq -2, \\ -5(0) - 3(1) - 2(0) - 1(1) + (0) \leq -4, & -5(0) - 3(1) - 2(0) - 1(0) + (1) \leq -4. \end{array}$$

For $x_4 = 1$, the first constraint is infeasible by 1 unit and the second constraint is feasible, giving 1 total unit of infeasibility. For $x_5 = 1$, the first constraint is infeasible by 2 units and the second by 2 units, giving 4 total units of infeasibility. Thus $x_4 = 1$ appears more favorable, and we would subdivide based upon that variable. In general, the variable giving the *least total infeasibilities* by this approach would be chosen next. Reviewing the example problem the reader will see that this approach has been used in our solution.

9.8 CUTTING PLANES

The cutting-plane algorithm solves integer programs by modifying linear-programming solutions until the integer solution is obtained. It does not partition the feasible region into subdivisions, as in branch-and-bound approaches, but instead works with a single linear program, which it refines by adding new constraints. The new constraints successively reduce the feasible region until an integer optimal solution is found.

In practice, the branch-and-bound procedures almost always outperform the cutting-plane algorithm. Nevertheless, the algorithm has been important to the evolution of integer programming. Historically, it was the first algorithm developed for integer programming that could be proved to converge in a finite number of steps. In addition, even though the algorithm generally is considered to be very inefficient, it has provided insights into integer programming that have led to other, more efficient, algorithms.

Again, we shall discuss the method by considering the sample problem of the previous sections:

$$z^* = \max 5x_1 + 8x_2,$$

subject to:

$$\begin{array}{rcl} x_1 + x_2 + s_1 & = & 6, \\ 5x_1 + 9x_2 + s_2 & = & 45, \\ x_1, x_2, s_1, s_2 & \geq & 0. \end{array} \tag{11}$$

s_1 and s_2 are, respectively, slack variables for the first and second constraints.

Solving the problem by the simplex method produces the following optimal tableau:

$$\begin{array}{rcl} (-z) & -\frac{5}{4}s_1 - \frac{3}{4}s_2 & = -41\frac{1}{4}, \\ x_1 & +\frac{9}{4}s_1 - \frac{1}{4}s_2 & = \frac{9}{4}, \\ x_2 - \frac{5}{4}s_1 + \frac{1}{4}s_2 & = & \frac{15}{4}, \\ x_1, x_2, s_1, s_2, s_3 & \geq & 0. \end{array}$$

Let us rewrite these equations in an equivalent but somewhat altered form:

$$\begin{array}{rcl} (-z) & -2s_1 - s_2 + 42 & = \frac{3}{4} - \frac{3}{4}s_1 - \frac{1}{4}s_2, \\ x_1 & +2s_1 - s_2 - 2 & = \frac{1}{4} - \frac{1}{4}s_1 - \frac{3}{4}s_2, \\ x_2 - 2s_1 & - 3 & = \frac{3}{4} - \frac{3}{4}s_1 - \frac{1}{4}s_2, \\ x_1, x_2, s_1, s_2 & \geq & 0. \end{array}$$

These algebraic manipulations have isolated integer coefficients to one side of the equalities and fractions to the other, in such a way that the constant terms on the righthand side are all nonnegative and the slack variable coefficients on the righthand side are all nonpositive.

In any integer solution, the lefthand side of each equation in the last tableau must be integer. Since s_1 and s_2 are nonnegative and appear to the right with negative coefficients, each righthand side necessarily must be less than or equal to the fractional constant term. Taken together, these two observations show that both sides of every equation must be an integer less than or equal to zero (if an integer is less than or equal to a fraction, it necessarily must be 0 or negative). Thus, from the first equation, we may write:

$$\frac{3}{4} - \frac{3}{4}s_1 - \frac{1}{4}s_2 \leq 0 \quad \text{and integer},$$

or, introducing a slack variable s_3 ,

$$\frac{3}{4} - \frac{3}{4}s_1 - \frac{1}{4}s_2 + s_3 = 0, \quad s_3 \geq 0 \quad \text{and integer}. \quad (C_1)$$

Similarly, other conditions can be generated from the remaining constraints:

$$\frac{1}{4} - \frac{1}{4}s_1 - \frac{3}{4}s_2 + s_4 = 0, \quad s_4 \geq 0 \quad \text{and integer} \quad (C_2)$$

$$\frac{3}{4} - \frac{3}{4}s_1 - \frac{1}{4}s_2 + s_5 = 0, \quad s_5 \geq 0 \quad \text{and integer}. \quad (C_3)$$

Note that, in this case, (C_1) and (C_3) are identical.

The new equations (C_1) , (C_2) , and (C_3) that have been derived are called *cuts* for the following reason: Their derivation did not exclude any integer solutions to the problem, so that any integer feasible point to the original problem must satisfy the cut constraints. The linear-programming solution had $s_1 = s_2 = 0$; clearly, these do *not* satisfy the cut constraints. In each case, substituting $s_1 = s_2 = 0$ gives either s_3 , s_4 , or $s_5 < 0$. Thus the net effect of a cut is to cut away the optimal linear-programming solution from the feasible region without excluding any feasible integer points.

The geometry underlying the cuts can be established quite easily. Recall from (11) that slack variables s_1 and s_2 are defined by:

$$\begin{aligned} s_1 &= 6 - x_1 - x_2, \\ s_2 &= 45 - 5x_1 - 9x_2. \end{aligned}$$

Substituting these values in the cut constraints and rearranging, we may rewrite the cuts as:

$$2x_1 + 3x_2 \leq 15, \quad (C_1 \text{ or } C_3)$$

$$4x_1 + 7x_2 \leq 35. \quad (C_2)$$

In this form, the cuts are displayed in Fig. 9.21. Note that they exhibit the features suggested above. In each case, the added cut removes the linear-programming solution $x_1 = \frac{9}{4}$, $x_2 = \frac{15}{4}$, from the feasible region, at the same time including every feasible integer solution.

The basic strategy of the cutting-plane technique is to add cuts (usually only one) to the constraints defining the feasible region and then to solve the resulting linear program. If the optimal values for the decision variables in the linear program are all integer, they are optimal; otherwise, a new cut is derived from the new optimal linear-programming tableau and appended to the constraints.

Note from Fig. 9.21 that the cut $C_1 = C_3$ leads directly to the optimal solution. Cut C_2 does not, and further iterations will be required if this cut is appended to the problem (without the cut $C_1 = C_3$). Also note that C_1 cuts deeper into the feasible region than does C_2 . For problems with many variables, it is generally quite difficult to determine which cuts will be deep in this sense. Consequently, in applications, the algorithm frequently generates cuts that shave very little from the feasible region, and hence the algorithm's poor performance.

A final point to be considered here is the way in which cuts are generated. The linear-programming tableau for the above problem contained the constraint:

$$x_1 + \frac{9}{4}s_1 - \frac{1}{4}s_2 = \frac{9}{4}.$$

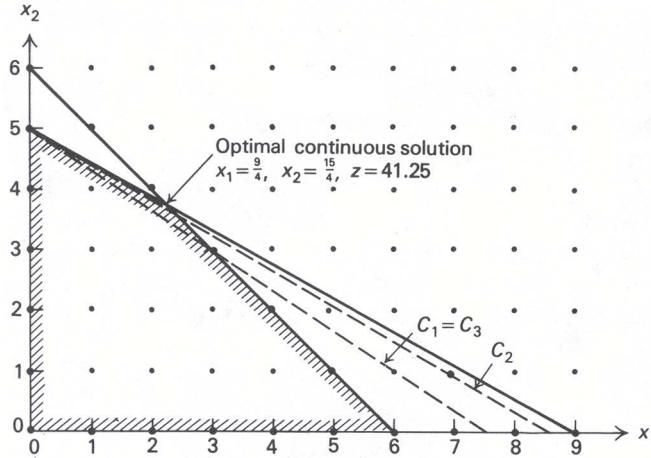


Figure 9.21 Cutting away the linear-programming solution.

Suppose that we round *down* the fractional coefficients to integers, that is, $\frac{9}{4}$ to 2, $-\frac{1}{4}$ to -1, and $\frac{9}{4}$ to 2. Writing these integers to the left of the equality and the remaining fractions to the right, we obtain as before, the equivalent constraint:

$$x_1 + 2s_1 - s_2 - 2 = \frac{1}{4} - \frac{1}{4}s_1 - \frac{3}{4}s_2.$$

By our previous arguments, the cut is:

$$\frac{1}{4} - \frac{1}{4}s_1 - \frac{3}{4}s_2 \leq 0 \quad \text{and integer.}$$

Another example may help to clarify matters. Suppose that the final linear-programming tableau to a problem has the constraint

$$x_1 + \frac{1}{6}x_6 - \frac{7}{6}x_7 + 3x_8 = 4\frac{5}{6}.$$

Then the equivalent constraint is:

$$x_1 - 2x_7 + 3x_8 - 4 = \frac{5}{6} - \frac{1}{6}x_6 - \frac{5}{6}x_7,$$

and the resulting cut is:

$$\frac{5}{6} - \frac{1}{6}x_6 - \frac{5}{6}x_7 \leq 0 \quad \text{and integer.}$$

Observe the way that fractions are determined for negative coefficients. The fraction in the cut constraint determined by the x_7 coefficient $-\frac{7}{6} = -1\frac{1}{6}$ is *not* $\frac{1}{6}$, but rather it is the fraction generated by rounding down to -2; i.e., the fraction is $-1\frac{1}{6} - (-2) = \frac{5}{6}$.

Tableau 2 shows the complete solution of the sample problem by the cutting-plane technique. Since cut $C_1 = C_3$ leads directly to the optimal solution, we have chosen to start with cut C_2 . Note that, if the slack variable for any newly generated cut is taken as the basic variable in that constraint, then the problem is in the proper form for the dual-simplex algorithm. For instance, the cut in Tableau 2(b) generated from the x_1 constraint

$$x_1 + \frac{7}{3}s_1 - \frac{1}{3}s_2 = \frac{7}{3} \quad \text{or} \quad x_1 + 2s_1 - s_2 - 2 = \frac{1}{3} - \frac{1}{3}s_1 - \frac{2}{3}s_2$$

is given by:

$$\frac{1}{3} - \frac{1}{3}s_1 - \frac{2}{3}s_2 \leq 0 \quad \text{and integer.}$$

Letting s_4 be the slack variable in the constraint, we obtain:

$$-\frac{1}{3}s_1 - \frac{2}{3}s_2 + s_4 = -\frac{1}{3}.$$

Tableau 2 Dual Simplex Iterations for the Cutting-Plane Algorithm.

(a)

Basic variables	Current values	x_1	x_2	s_1	s_2	s_3
($-z$)	$-41\frac{1}{4}$			$-\frac{5}{4}$	$-\frac{3}{4}$	
x_1	$\frac{9}{4}$	1		$\frac{9}{4}$	$-\frac{1}{4}$	
x_2	$\frac{15}{4}$		1	$-\frac{5}{4}$	$\frac{1}{4}$	
$\leftarrow s_3$	$-\frac{1}{4}$			$-\frac{1}{4}$	$\textcircled{-}\frac{3}{4}$	1

(cut generated from x_1 constraint)

(b)

Basic variables	Current values	x_1	x_2	s_1	s_2	s_3	s_4
($-z$)	-41			-1		-1	
x_1	$\frac{7}{3}$	1		$\frac{7}{3}$		$-\frac{1}{3}$	
x_2	$\frac{11}{3}$		1	$-\frac{4}{3}$	$\frac{1}{3}$	$\frac{1}{3}$	
s_2	$\frac{1}{3}$			$\frac{1}{3}$	1	$-\frac{4}{3}$	
$\leftarrow s_4$	$-\frac{1}{3}$			$-\frac{1}{3}$		$\textcircled{-}\frac{2}{3}$	1

(cut generated from x_1 constraint)

(c)

Basic variables	Current values	x_1	x_2	s_1	s_2	s_3	s_4	s_5
($-z$)	$-40\frac{1}{2}$			$-\frac{1}{2}$			$-\frac{3}{2}$	
x_1	$\frac{5}{2}$	1		$\frac{5}{2}$			$-\frac{1}{2}$	
x_2	$\frac{7}{2}$		1	$-\frac{3}{2}$	1		$\frac{1}{2}$	
s_2	1			1	1		-2	
s_3	$\frac{1}{2}$			$\frac{1}{2}$		1	$-\frac{3}{2}$	
$\leftarrow s_5$	$-\frac{1}{2}$			$\textcircled{-}\frac{1}{2}$			$-\frac{1}{2}$	1

(cut generated from x_1 constraint)

(d)

Basic variables	Current values	x_1	x_2	s_1	s_2	s_3	s_4	s_5
($-z$)	-40						-1	-1
x_1	0	1					3	5
x_2	5		1				2	-3
s_2	0				1		-3	2
s_3	0			1		1	-2	1
s_1	1						1	-2

Since s_1 and s_2 are nonbasic variables, we may take s_4 to be the basic variable isolated in this constraint (see Tableau 2(b)).

By making slight modifications to the cutting-plane algorithm that has been described, we can show that an optimal solution to the integer-programming problem will be obtained, as in this example, after adding only a finite number of cuts. The proof of this fact by R. Gomory in 1958 was a very important theoretical break-through, since it showed that integer programs can be solved by *some* linear program (the associated linear program plus the added constraints). Unfortunately, the number of cuts to be added, though finite, is usually quite large, so that this result does not have important practical ramifications.

EXERCISES

1. As the leader of an oil-exploration drilling venture, you must determine the least-cost selection of 5 out of 10 possible sites. Label the sites S_1, S_2, \dots, S_{10} , and the exploration costs associated with each as C_1, C_2, \dots, C_{10} .

Regional development restrictions are such that:

- i) Evaluating sites S_1 and S_7 will prevent you from exploring site S_8 .
- ii) Evaluating site S_3 or S_4 prevents you from assessing site S_5 .
- iii) Of the group S_5, S_6, S_7, S_8 , only two sites may be assessed.

Formulate an integer program to determine the minimum-cost exploration scheme that satisfies these restrictions.

2. A company wishes to put together an academic “package” for an executive training program. There are five area colleges, each offering courses in the six fields that the program is designed to touch upon.

The package consists of 10 courses; each of the six fields must be covered.

The tuition (basic charge), assessed when at least one course is taken, at college i is T_i (independent of the number of courses taken). Moreover, each college imposes an additional charge (covering course materials, instructional aids, and so forth) for *each* course, the charge depending on the college and the field of instructions.

Formulate an integer program that will provide the company with the minimum amount it must spend to meet the requirements of the program.

3. The marketing group of A. J. Pitt Company is considering the options available for its next advertising campaign program. After a great deal of work, the group has identified a selected number of options with the characteristics shown in the accompanying table.

	TV	Trade magazine	Newspaper	Radio	Popular magazine	Promotional campaign	Total resource available
<i>Customers reached</i>	1,000,000	200,000	300,000	400,000	450,000	450,000	—
<i>Cost (\$)</i>	500,000	150,000	300,000	250,000	250,000	100,000	1,800,000
<i>Designers needed</i> (man-hours)	700	250	200	200	300	400	1,500
<i>Salesmen needed</i> (man-hours)	200	100	100	100	100	1,000	1,200

The objective of the advertising program is to maximize the number of customers reached, subject to the limitation of resources (money, designers, and salesman) given in the table above. In addition, the following constraints have to be met:

- i) If the promotional campaign is undertaken, it needs either a radio or a popular magazine campaign effort to support it.
- ii) The firm cannot advertise in both the trade and popular magazines.

Formulate an integer-programming model that will assist the company to select an appropriate advertising campaign strategy.

4. Three different items are to be routed through three machines. Each item must be processed first on machine 1, then on machine 2, and finally on machine 3. The sequence of items may differ for each machine. Assume that the times t_{ij} required to perform the work on item i by machine j are known and are integers. Our objective is to minimize the total time necessary to process all the items.

- a) Formulate the problem as an integer programming problem. [Hint. Let x_{ij} be the starting time of processing item i on machine j . Your model must prevent two items from occupying the same machine at the same time; also, an item may not start processing on machine $(j + 1)$ unless it has completed processing on machine j .]

- b) Suppose we want the items to be processed in the same sequence on each machine. Change the formulation in part (a) accordingly.
5. Consider the problem:

$$\text{Maximize } z = x_1 + 2x_2,$$

subject to:

$$\begin{aligned} x_1 + x_2 &\leq 8, \\ -x_1 + x_2 &\leq 2, \\ x_1 - x_2 &\leq 4, \\ x_2 &\geq 0 \quad \text{and integer,} \end{aligned}$$

$$x_1 = 0, 1, 4, \text{ or } 6.$$

- a) Reformulate the problem as an equivalent integer linear program.
 b) How would your answer to part (a) change if the objective function were changed to:

$$\text{Maximize } z = x_1^2 + 2x_2?$$

6. Formulate, but *do not solve*, the following mathematical-programming problems. Also, indicate the type of algorithm used in general to solve each.

- a) A courier traveling to Europe can carry up to 50 kilograms of a commodity, all of which can be sold for \$40 per kilogram. The round-trip air fare is \$450 plus \$5 per kilogram of baggage in excess of 20 kilograms (one way). Ignoring any possible profits on the return trip, should the courier travel to Europe and, if so, how much of the commodity should be taken along in order to maximize his profits?
 b) A copying service incurs machine operating costs of:

$$\begin{aligned} \$0.10 &\text{ for copies 1 to 4,} \\ 0.05 &\text{ for copies 5 to 8,} \\ 0.025 &\text{ for copies 9 and over,} \end{aligned}$$

and has a capacity of 1000 copies per hour. One hour has been reserved for copying a 10-page article to be sold to MBA students. Assuming that all copies can be sold for \$0.50 per article, how many copies of the article should be made?

- c) A petrochemical company wants to maximize profit on an item that sells for \$0.30 per gallon. Suppose that increased temperature increases output according to the graph in Fig. E9.1. Assuming that production costs are directly proportional to temperature as \$7.50 per degree centigrade, how many gallons of the item should be produced?
7. Suppose that you are a ski buff and an entrepreneur. You own a set of hills, any or all of which can be developed. Figure E9.2 illustrates the nature of the cost for putting ski runs on any hill.

The cost includes fixed charges for putting new trails on a hill. For each hill j , there is a limit d_j on the number of trails that can be constructed and a lower limit t_j on the number of feet of trail that must be developed if the hill is used.

Use a piecewise linear approximation to formulate a strategy based on cost minimization for locating the ski runs, given that you desire to have M total feet of developed ski trail in the area.

8. Consider the following word game. You are assigned a number of tiles, each containing a letter a, b, \dots, z from the alphabet. For any letter α from the alphabet, your assignment includes N_α (a nonnegative integer) tiles with the letter α . From the letters you are to construct any of the words w_1, w_2, \dots, w_n . This list might, for example, contain all words from a given dictionary.

You may construct any word at most once, and use any tile at most once. You receive $v_j \geq 0$ points for making word w_j and an additional bonus of $b_{ij} \geq 0$ points for making *both* words w_i and w_j ($i = 1, 2, \dots, n; j = 1, 2, \dots, n$).

- a) Formulate a *linear* integer program to determine your optimal choice of words.

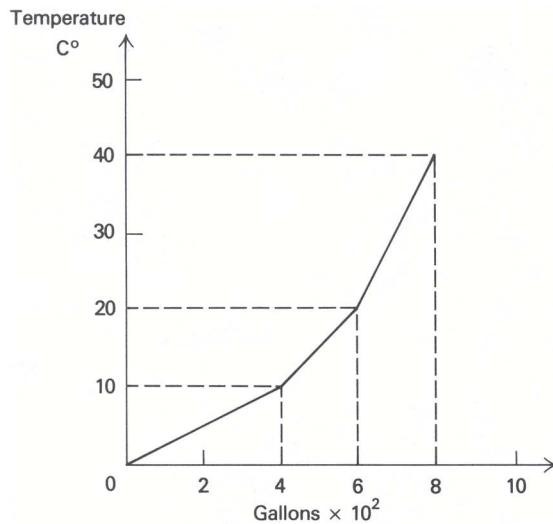


Figure E9.1

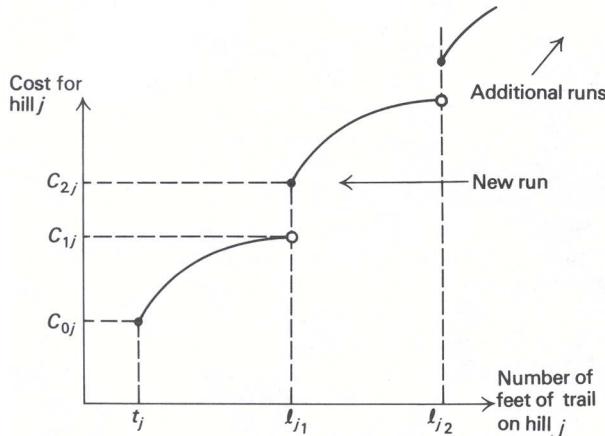


Figure E9.2

- b) How does the formulation change if you are allowed to select 100 tiles with no restriction on your choice of letters?
9. In Section 9.1 of the text, we presented the following simplified version of the warehouse-location problem:

$$\text{Minimize } \sum_i \sum_j c_{ij} x_{ij} + \sum_i f_i y_i,$$

subject to:

$$\begin{aligned} \sum_i x_{ij} &= d_j && (j = 1, 2, \dots, n) \\ \sum_j x_{ij} - y_i \left(\sum_j d_j \right) &\leq 0 && (i = 1, 2, \dots, m) \\ x_{ij} &\geq 0 && (i = 1, 2, \dots, m; j = 1, 2, \dots, n) \\ y_i &= 0 \text{ or } 1 && (i = 1, 2, \dots, m). \end{aligned}$$

- a) The above model assumes only one product and two distribution stages (from warehouse to customer). Indicate how the model would be modified if there were three distribution stages (from plant to warehouse to customer) and L different products. [Hint. Define a new decision variable x_{ijkl} equal to the amount to be sent from plant i , through warehouse j , to customer k , of product ℓ .]

- b) Suppose there are maximum capacities for plants and size limits (both upper and lower bounds) for warehouses. What is the relevant model now?
- c) Assume that each customer has to be served from a single warehouse; i.e., no splitting of orders is allowed. How should the warehousing model be modified? [Hint. Define a new variable z_{jk} = fraction of demand of customer k satisfied from warehouse j .]
- d) Assume that each warehouse i experiences economies of scale when shipping above a certain threshold quantity to an individual customer; i.e., the unit distribution cost is c_{ij} whenever the amount shipped is between 0 and d_{ij} , and c'_{ij} (lower than c_{ij}) whenever the amount shipped exceeds d_{ij} . Formulate the warehouse location model under these conditions.

10. Graph the following integer program:

$$\text{Maximize } z = x_1 + 5x_2,$$

subject to :

$$\begin{aligned} -4x_1 + 3x_2 &\leq 6, \\ 3x_1 + 2x_2 &\leq 18, \end{aligned}$$

$$x_1, x_2 \geq 0 \quad \text{and integer.}$$

Apply the branch-and-bound procedure, graphically solving each linear-programming problem encountered. Interpret the branch-and-bound procedure graphically as well.

11. Solve the following integer program using the branch-and-bound technique:

$$\text{Minimize } z = 2x_1 - 3x_2 - 4x_3,$$

subject to:

$$\begin{aligned} -x_1 + x_2 + 3x_3 &\leq 8, \\ 3x_1 + 2x_2 - x_3 &\leq 10, \end{aligned}$$

$$x_1, x_2, x_3 \geq 0 \quad \text{and integer.}$$

The optimal tableau to the linear program associated with this problem is:

<i>Basic variables</i>	<i>Current values</i>	x_1	x_2	x_3	x_4	x_5
x_3	$\frac{6}{7}$	$-\frac{5}{7}$		1	$\frac{2}{7}$	$-\frac{1}{7}$
x_2	$\frac{38}{7}$	$\frac{8}{7}$	1		$\frac{1}{7}$	$\frac{3}{7}$
($-z$)	$\frac{138}{7}$	$\frac{18}{7}$			$\frac{11}{7}$	$\frac{5}{7}$

The variables x_4 and x_5 are slack variables for the two constraints.

12. Use branch-and-bound to solve the mixed-integer program:

$$\text{Maximize } z = -5x_1 - x_2 - 2x_3 + 5,$$

subject to:

$$\begin{aligned} -2x_1 + x_2 - x_3 + x_4 &= \frac{7}{2}, \\ 2x_1 + x_2 + x_3 + x_5 &= 2, \end{aligned}$$

$$x_j \geq 0 \quad (j = 1, 2, \dots, 5)$$

$$x_3 \text{ and } x_5 \text{ integer.}$$

13. Solve the mixed-integer programming knapsack problem:

$$\text{Maximize } z = 6x_1 + 4x_2 + 4x_3 + x_4 + x_5,$$

subject to:

$$2x_1 + 2x_2 + 3x_3 + x_4 + 2x_5 = 7, \\ x_j \geq 0 \quad (j = 1, 2, \dots, 5),$$

x_1 and x_2 integer.

14. Solve the following integer program using implicit enumeration:

$$\text{Maximize } z = 2x_1 - x_2 - x_3 + 10,$$

subject to:

$$2x_1 + 3x_2 - x_3 \leq 9, \\ 2x_2 + x_3 \geq 4, \\ 3x_1 + 3x_2 + 3x_3 = 6, \\ x_j = 0 \text{ or } 1 \quad (j = 1, 2, 3).$$

15. A college intramural four-man basketball team is trying to choose its starting line-up from a six-man roster so as to maximize average height. The roster follows:

Player	Number	Height*	Position
Dave	1	10	Center
John	2	9	Center
Mark	3	6	Forward
Rich	4	6	Forward
Ken	5	4	Guard
Jim	6	-1	Guard

* In inches over 5' 6".

The starting line-up must satisfy the following constraints:

- i) At least one guard must start.
 - ii) Either John or Ken must be held in reserve.
 - iii) Only one center can start.
 - iv) If John or Rich starts, then Jim cannot start.
- a) Formulate this problem as an integer program.
 b) Solve for the optimal starting line-up, using the implicit enumeration algorithm.

16. Suppose that one of the following constraints arises when applying the implicit enumeration algorithm to a 0–1 integer program:

$$-2x_1 - 3x_2 + x_3 + 4x_4 \leq -6, \tag{1}$$

$$-2x_1 - 6x_2 + x_3 + 4x_4 \leq -5, \tag{2}$$

$$-4x_1 - 6x_2 - x_3 + 4x_4 \leq -3. \tag{3}$$

In each case, the variables on the lefthand side of the inequalities are free variables and the righthand-side coefficients include the contributions of the fixed variables.

- a) Use the feasibility test to show that constraint (1) contains no feasible completion.
- b) Show that $x_2 = 1$ and $x_4 = 0$ in any feasible completion to constraint (2). State a general rule that shows when a variable x_j , like x_2 or x_4 in constraint (2), must be either 0 or 1 in any feasible solution. [Hint. Apply the usual feasibility test after setting x_j to 1 or 0.]
- c) Suppose that the objective function to minimize is:

$$z = 6x_1 + 4x_2 + 5x_3 + x_4 + 10,$$

and that $\underline{z} = 17$ is the value of an integer solution obtained previously. Show that $x_3 = 0$ in a feasible completion to constraint (3) that is a potential improvement upon \underline{z} with $z < \underline{z}$. (Note that either $x_1 = 1$ or $x_2 = 1$ in any feasible solution to constraint (3) having $x_3 = 1$.)

- d) How could the tests described in parts (b) and (c) be used to reduce the size of the enumeration tree encountered when applying implicit enumeration?

17. Although the constraint

$$x_1 - x_2 \leq -1$$

and the constraint

$$-x_1 + 2x_2 \leq -1$$

to a zero-one integer program both have feasible solutions, the system composed of both constraints is infeasible. One way to exhibit the inconsistency in the system is to add the two constraints, producing the constraint

$$x_2 \leq -1,$$

which has no feasible solution with $x_2 = 0$ or 1.

More generally, suppose that we multiply the i th constraint of a system

$$\begin{array}{lll} \text{Multipliers} \\ a_{11}x_1 + a_{12}x_2 + \cdots + a_{1n}x_n \leq b_1, & u_1 \\ \vdots & \vdots & \vdots \\ a_{i1}x_1 + a_{i2}x_2 + \cdots + a_{in}x_n \leq b_i, & u_i \\ \vdots & \vdots & \vdots \\ a_{m1}x_1 + a_{m2}x_2 + \cdots + a_{mn}x_n \leq b_m, & u_m \\ x_j = 0 \quad \text{or} \quad 1 \quad (j = 1, 2, \dots, n), & \end{array}$$

by nonnegative constraints u_i and add to give the composite, or *surrogate*, constraint:

$$\left(\sum_{i=1}^m u_i a_{i1} \right) x_1 + \left(\sum_{i=1}^m u_i a_{i2} \right) x_2 + \cdots + \left(\sum_{i=1}^m u_i a_{in} \right) x_n \leq \sum_{i=1}^m u_i b_i.$$

- a) Show that any feasible solution $x_j = 0$ or 1 ($j = 1, 2, \dots, n$) for the system of constraints must also be feasible to the surrogate constraint.
- b) How might the fathoming tests discussed in the previous exercise be used in conjunction with surrogate constraints in an implicit enumeration scheme?
- c) A “best” surrogate constraint might be defined as one that eliminates as many nonoptimal solutions $x_j = 0$ or 1 as possible. Consider the objective value of the integer program when the system constraints are replaced by the surrogate constraint; that is the problem:

$$v(u) = \text{Min } c_1 x_1 + c_2 x_2 + \cdots + c_n x_n,$$

subject to:

$$\begin{aligned} \left(\sum_{i=1}^m u_i a_{i1} \right) x_1 + \left(\sum_{i=1}^m u_i a_{i2} \right) x_2 + \cdots + \left(\sum_{i=1}^m u_i a_{in} \right) x_n &\leq \sum_{i=1}^m u_i b_i, \\ x_j = 0 \quad \text{or} \quad 1 \quad (j = 1, 2, \dots, n). & \end{aligned}$$

Let us say that a surrogate constraint with multipliers u_1, u_2, \dots, u_m is *stronger* than another surrogate constraint with multipliers $\bar{u}_1, \bar{u}_2, \dots, \bar{u}_m$, if the value $v(u)$ of the surrogate constraint problem with multipliers u_1, u_2, \dots, u_m is larger than the value of $v(\bar{u})$ with multipliers $\bar{u}_1, \bar{u}_2, \dots, \bar{u}_m$. (The larger we make $v(u)$, the more nonoptimal solutions we might expect to eliminate.)

Suppose that we estimate the value of $v(u)$ defined above by replacing $x_j = 0$ or 1 by $0 \leq x_j \leq 1$ for $j = 1, 2, \dots, n$. Then, to estimate the strongest surrogate constraint, we would need to find those values of the multipliers u_1, u_2, \dots, u_m to maximize $v(u)$, where

$$v(u) = \text{Min } c_1x_1 + c_2x_2 + \dots + c_nx_n,$$

subject to:

$$0 \leq x_j \leq 1 \quad (1)$$

and the surrogate constraint.

Show that the optimal shadow prices to the linear program

$$\text{Maximize } c_1x_1 + c_2x_2 + \dots + c_nx_n,$$

subject to:

$$a_{i1}x_1 + a_{i2}x_2 + \dots + a_{in}x_n \leq b_i \quad (i = 1, 2, \dots, m),$$

$$0 \leq x_j \leq 1 \quad (j = 1, 2, \dots, n),$$

solve the problem of maximizing $v(u)$ in (1).

18. The following tableau specifies the solution to the linear program associated with the integer program presented below.

<i>Basic variables</i>	<i>Current values</i>	x_1	x_2	x_3	x_4
x_1	$\frac{16}{5}$	1		$\frac{2}{5}$	$-\frac{1}{5}$
x_2	$\frac{23}{5}$		1	$\frac{1}{5}$	$\frac{2}{5}$
$(-z)$	$-\frac{133}{5}$			$-\frac{11}{5}$	$-\frac{2}{5}$

$$\text{Maximize } z = 4x_1 + 3x_2,$$

subject to:

$$\begin{aligned} 2x_1 + x_2 + x_3 &= 11, \\ -x_1 + 2x_2 + x_4 &= 6, \\ x_j &\geq 0 \text{ and integer} \quad (j = 1, 2, 3, 4). \end{aligned}$$

- a) Derive cuts from each of the rows in the optimal linear-programming tableau, including the objective function.
- b) Express the cuts in terms of the variables x_1 and x_2 . Graph the feasible region for x_1 and x_2 and illustrate the cuts on the graph.
- c) Append the cut derived from the objective function to the linear program, and re-solve. Does the solution to this new linear program solve the integer program? If not, how would you proceed to find the optimal solution to the integer program?

19. Given the following integer linear program:

$$\text{Maximize } z = 5x_1 + 2x_2,$$

subject to:

$$\begin{aligned} 5x_1 + 4x_2 &\leq 21, \\ x_1, x_2 &\geq 0 \text{ and integer,} \end{aligned}$$

solve, using the cutting-plane algorithm. Illustrate the cuts on a graph of the feasible region.

20. The following knapsack problem:

$$\text{Maximize } \sum_{j=1}^n cx_j,$$

subject to:

$$\begin{aligned} \sum_{j=1}^n 2x_j &\leq n, \\ x_j &= 0 \text{ or } 1 \quad (j = 1, 2, \dots, n), \end{aligned}$$

which has the same “contribution” for each item under consideration, has proved to be rather difficult to solve for most general-purpose integer-programming codes when n is an *odd* number.

- a) What is the optimal solution when n is even? when n is odd?
 - b) Comment specifically on why this problem might be difficult to solve on general integer-programming codes when n is odd.
21. Suppose that a firm has N large rolls of paper, each W inches wide. It is necessary to cut N_i rolls of width W_i from these rolls of paper. We can formulate this problem by defining variables

$$x_{ij} = \text{Number of smaller rolls of width } W_i \text{ cut from large roll } j.$$

We assume there are m different widths W_i . In order to cut all the required rolls of width W_i , we need constraints of the form:

$$\sum_{j=1}^N x_{ij} = N_i \quad (i = 1, 2, \dots, m).$$

Further, the number of smaller rolls cut from a large roll is limited by the width W of the large roll. Assuming no loss due to cutting, we have constraints of the form:

$$\sum_{i=1}^m W_i x_{ij} \leq W \quad (j = 1, 2, \dots, N).$$

- a) Formulate an objective function to minimize the number of large rolls of width W used to produce the smaller rolls.
 - b) Reformulate the model to minimize the total trim loss resulting from cutting. Trim loss is defined to be that part of a large roll that is unusable due to being smaller than any size needed.
 - c) Comment on the difficulty of solving each formulation by a branch-and-bound method.
 - d) Reformulate the problem in terms of the collection of possible patterns that can be cut from a given large roll. (*Hint.* See Exercise 25 in Chapter 1.)
 - e) Comment on the difficulty of solving the formulation in (d), as opposed to the formulations in (a) or (b), by a branch-and-bound method.
22. The Bradford Wire Company produces wire screening woven on looms in a process essentially identical to that of weaving cloth. Recently, Bradford has been operating at full capacity and is giving serious consideration to a major capital investment in additional looms. They currently have a total of 43 looms, which are in production all of their available hours. In order to justify the investment in additional looms, they must analyze the utilization of their existing capacity.

Of Bradford's 43 looms, 28 are 50 inches wide, and 15 are 80 inches wide. Normally, one or two widths totaling less than the width of the loom are made on a particular loom. With the use of a “center-tucker,” up to three widths

can be simultaneously produced on one loom; however, in this instance the effective capacities of the 50-inch and 80-inch looms are reduced to 49 inches and 79 inches, respectively. Under no circumstance is it possible to make more than three widths simultaneously on any one loom.

Figure E9.3 gives a typical loom-loading configuration at one point in time. Loom #1, which is 50 inches wide, has two 24-inch widths being produced on it, leaving 2 inches of unused or "wasted" capacity. Loom #12 has only a 31-inch width on it, leaving 19 inches of the loom unused. If there were an order for a width of 19 inches or less, then it could be produced at zero marginal machine cost along with the 31-inch width already being produced on this loom. Note that loom #40 has widths of 24, 26, and 28 inches, totaling 78 inches. The "waste" here is considered to be only 1 inch, due to the reduced effective capacity from the use of the center-tucker. Note also that the combination of widths 24, 26, and 30 is not allowed, for similar reasons.

50" Looms			80" Looms		
Loom number	Widths	Waste	Loom number	Widths	Waste
1	24–24	2	29	48–32	0
2	30	20	30	24–26–28	1
3	40	10	31	48–32	0
4	30 $\frac{1}{4}$	19 $\frac{3}{4}$	32	48–32	0
5	48	2	33	36–36	8
6	32–14 $\frac{1}{2}$	3 $\frac{1}{2}$	34	36–36	8
7	28	22	35	60–18	2
8	26 $\frac{1}{2}$	23 $\frac{1}{2}$	36	36–36	8
9	30	20	37	42–34	4
10	30	20	38	24–26–28	1
11	35	15	39	24–26–28	1
12	31	19	40	24–26–28	1
13	34 $\frac{1}{4}$	15 $\frac{3}{4}$	41	48–32	0
14	36	14	42	36–36	8
15	32	18	43	42–34	4
16	34–16	0	80" Total		46
17	29 $\frac{3}{8}$	20 $\frac{5}{8}$	<u>337$\frac{3}{4}$</u>		<u>337$\frac{3}{4}$</u>
18	30	20	Grand total		<u>383$\frac{3}{4}$</u>
19	36	14			
20	47 $\frac{1}{2}$	2 $\frac{1}{2}$			
21	30	20			
22	20–30	0			
23	48 $\frac{7}{8}$	1 $\frac{1}{8}$			
24	28–22	0			
25	30–14 $\frac{1}{2}$	5 $\frac{1}{2}$			
26	42	8			
27	32–18	0			
28	28 $\frac{1}{2}$	21 $\frac{1}{2}$			
50" Total		<u>337$\frac{3}{4}$</u>			

Figure E9.3

The total of 383 $\frac{3}{4}$ inches of "wasted" loom capacity represents roughly seven and one-half 50-inch looms; and members of Bradford's management are sure that there must be a more efficient loom-loading configuration that would save some of this "wasted" capacity. As there are always numerous additional orders to be produced, any additional capacity can immediately be put to good use.

The two types of looms are run at different speeds and have different efficiencies. The 50-inch looms are operated at 240 pics per second, while the 80-inch looms are operated at 214 pics per second. (There are 16 pics to the inch). Further, the 50-inch looms are more efficient, since their "up time" is about 85% of the total time, as compared to 65% for the 80-inch looms.

The problem of scheduling the various orders on the looms sequentially over time is difficult. However, as a first cut at analyzing how efficiently the looms are currently operating, the company has decided to examine the loom-loading configuration at one point in time as given in Fig. E9-3. If these same orders can be rearranged in

such a way as to free up one or two looms, then it would be clear that a closer look at the utilization of existing equipment would be warranted before additional equipment is purchased.

- a) Since saving an 80-inch loom is not equivalent to saving a 50-inch loom, what is an appropriate objective function to minimize?
 - b) Formulate an integer program to minimize the objective function suggested in part (a).
23. In the export division of Lowell Textile Mills, cloth is woven in lengths that are multiples of the piece-length required by the customer. The major demand is for 18-meter piece-lengths, and the cloth is generally woven in 54-meter lengths.

Cloth cannot be sold in lengths greater than the stipulated piece-length. Lengths falling short are classified into four groups. For 18-meter piece-lengths, the categories and the contribution per meter are as follows:

<i>Category</i>	<i>Technical term</i>	<i>Length (Meters)</i>	<i>Contribution/ Meter</i>
A	First sort	18	1.00
B	Seconds	11–17	0.90
C	Short lengths	6–10	0.75
D	Rags	1–5	0.60
J	Joined parts*	18	0.90

* Joined parts consist of lengths obtained by joining two pieces such that the shorter piece is at least 6 meters long.

The current cutting practice is as follows. Each woven length is inspected and defects are flagged prominently. The cloth is cut at these defects and, since the cloth is folded in exact meter lengths, the lengths of each cut piece is known. The cut pieces are then cut again, if necessary, to obtain as many pieces of first sort as possible. The short lengths are joined wherever possible to make joined parts.

Since the process is continuous, it is impossible to pool all the woven lengths and then decide on a cutting pattern. Each woven length has to be classified for cutting before it enters the cutting room.

As an example of the current practice, consider a woven length of 54 meters with two defects, one at 19 meters and one at 44 meters. The woven length is first cut at the defects, giving three pieces of lengths 19, 25, and 10 meters each. Then further cuts are made as follows: The resulting contribution is

<i>Piece length</i>	<i>Further cuts</i>
25	18 + 7
19	18 + 1
10	10

$$2 \times 18 \times 1.00 + (7 + 10) \times 0.75 + 1 \times 0.60 = 49.35.$$

It is clear that this cutting procedure can be improved upon by the following alternative cutting plan: By joining 7

<i>Piece length</i>	<i>Further cuts</i>
25	18 + 7
19	8 + 11
10	10

+ 11 and 8 + 10 to make two joined parts, the resulting contribution is:

$$18 \times 1.00 + 18 \times 2 \times 0.90 = 50.40.$$

Thus with one woven length, contribution can be increased by \$1.05 by merely changing the cutting pattern. With a daily output of 1000 such woven lengths (two defects on average), substantial savings could be realized by improved cutting procedures.

- a) Formulate an integer program to maximize the contribution from cutting the woven length described above.
 - b) Formulate an integer program to maximize the contribution from cutting a general woven length with no more than four defects. Assume that the defects occur at integral numbers of meters.
 - c) How does the formulation in (b) change when the defects are *not* at integral numbers of meters?
24. Custom Pilot Chemical Company is a chemical manufacturer that produces batches of specialty chemicals to order. Principal equipment consists of eight interchangeable reactor vessels, five interchangeable distillation columns, four large interchangeable centrifuges, and a network of switchable piping and storage tanks. Customer demand comes in the form of orders for batches of one or more (usually not more than three) specialty chemicals, normally to be delivered simultaneously for further use by the customer. Customer Pilot Chemical fills these orders by means of a sort of pilot plant for each specialty chemical formed by inter-connecting the appropriate quantities of equipment. Sometimes a specialty chemical requires processing by more than one of these "pilot" plants, in which case one or more batches of intermediate products may be produced for further processing. There is no shortage of piping and holding tanks, but the expensive equipment (reactors, distillation columns, and centrifuges) is frequently inadequate to meet the demand.

The company's schedules are worked out in calendar weeks, with actual production always taking an integer multiple of weeks. Whenever a new order comes in, the scheduler immediately makes up a precedence tree-chart for it. The precedence tree-chart breaks down the order into "jobs." For example, an order for two specialty chemicals, of which one needs an intermediate stage, would be broken down into three jobs (Fig. E9.4). Each job requires certain equipment, and each is assigned a preliminary time-in-process, or "duration." The durations can always be predicted with a high degree of accuracy.

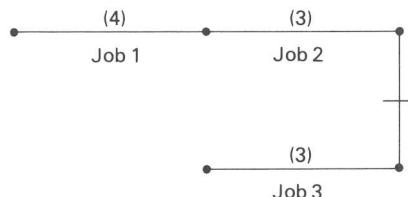


Figure E9.4

Currently, Custom Pilot Chemical has three orders that need to be scheduled (see the accompanying table). Orders can be started at the beginning of their arrival week and should be completed by the end of their *week due*.

Order number	Job number	Precedence relations	Arrival week	Duration in weeks	Resource requirements			
					Week due	Reactors	Distillation columns	Centrifuges
AK14	1	None	15	4	22	5	3	2
	2	(1)	15	3	22	0	1	1
	3	None	15	3	22	2	0	2
AK15	1	None	16	3	23	1	1	1
	2	None	16	2	23	2	0	0
	3	(1)	16	2	23	2	2	0
AK16	1	None	17	5	23	2	1	1
	2	None	17	1	23	1	3	0

For example, AK14 consists of three jobs, where Job 2 cannot be started until Job 1 has been completed. Figure E9-4 is an appropriate precedence tree for this order. Generally, the resources required are tied up for the entire duration of a job. Assume that Custom Pilot Chemical is just finishing week 14.

- a) Formulate an integer program that minimizes the total completion time for all orders.

- b) With a more complicated list of orders, what other objective functions might be appropriate? How would these be formulated as integer programs?
25. A large electronics manufacturing firm that produces a single product is faced with rapid sales growth. Its planning group is developing an overall capacity-expansion strategy, that would balance the cost of building new capacity, the cost of operating the new and existing facilities, and the cost associated with unmet demand (that has to be subcontracted outside at a higher cost).

The following model has been proposed to assist in defining an appropriate strategy for the firm.

Decision variables

Y_{it}	A binary integer variable that will be 1 if a facility exists at site i in period t , and 0 otherwise.
$(IY)_{it}$	A binary integer variable that will be 1 if facility is constructed at site i in period t , and 0 otherwise.
A_{it}	Capacity (sq ft) at site i in period t .
$(IA)_{it}$	The increase in capacity at site i in period t .
$(DA)_{it}$	The decrease in capacity at site i in period t .
P_{it}	Total units produced at site i in period t .
U_t	Total unmet demand (subcontracted) units in period t .

Forecast parameters

D_t	Demand in units in period t .
-------	---------------------------------

Cost parameters—Capacity

s_{it}	Cost of operating a facility at site i in period t .
d_{it}	Cost of building a facility at site i in period t .
a_{it}	Cost of occupying 1 sq ft of fully equipped capacity at site i in period t .
b_{it}	Cost of increasing capacity by 1 sq ft at site i in period t .
c_{it}	Cost of decreasing capacity by 1 sq ft at site i in period t .

Cost parameters—Production

o_t	Cost of unmet demand (subcontracting + opportunity cost) for one unit in period t .
v_{it}	Tax-adjusted variable cost (material + labor + taxes) of producing one unit at site i in period t .

Productivity parameters

$(pa)_{it}$	Capacity needed (sq ft years) at site i in period t to produce one unit of the product.
-------------	---

Policy parameters

$\bar{F}_{it}, \underline{F}_{it}$	Maximum, minimum facility size at site i in period t (if the site exists).
G_{it}	Maximum growth (sq ft) of site i in period t .

Objective function

The model's objective function is to minimize total cost:

$$\begin{aligned} \text{Min } & \sum_{t=1}^T \sum_{i=1}^N s_{it} Y_{it} + d_{it} (IY)_{it} \\ & + \sum_{t=1}^T \sum_{i=1}^N a_{it} A_{it} + b_{it} (IA)_{it} + c_{it} (DA)_{it} \\ & + \sum_{t=1}^T \sum_{i=1}^N v_{it} P_{it} + \sum_{t=1}^T o_t U_t \end{aligned}$$

Description of constraints

1. *Demand constraint*

$$\sum_{i=1}^N P_{it} + U_t = D_t$$

2. *Productivity constraint*

$$(pa)_{it} P_{it} \leq A_{it} \quad \begin{cases} \text{for } i = 1, 2, \dots, N, \\ \text{for } t = 1, 2, \dots, T, \end{cases}$$

3. *Facility-balance constraint*

$$Y_{it-1} + (IY)_{it} = Y_{it} \quad \begin{cases} \text{for } i = 1, 2, \dots, N, \\ \text{for } t = 1, 2, \dots, T, \end{cases}$$

4. *Area-balance constraint*

$$A_{it-1} + (IA)_{it} - (DA)_{it} = A_{it} \quad \begin{cases} \text{for } i = 1, 2, \dots, N, \\ \text{for } t = 1, 2, \dots, T, \end{cases}$$

5. *Facility-size limits*

$$\begin{aligned} A_{it} &\geq Y_{it} \underline{F}_{it} \\ A_{it} &\leq Y_{it} \bar{F}_{it} \end{aligned} \quad \begin{cases} \text{for } i = 1, 2, \dots, N, \\ \text{for } t = 1, 2, \dots, T, \end{cases}$$

6. *Growth constraint*

$$(IA)_{it} - (DA)_{it} \leq G_{it} \quad \begin{cases} \text{for } i = 1, 2, \dots, N, \\ \text{for } t = 1, 2, \dots, T, \end{cases}$$

7. *Additional constraints*

$$\begin{aligned} 0 &\leq Y_{it} \leq 1 \\ Y_{it} &\text{ integer} \end{aligned} \quad \begin{cases} \text{for } i = 1, 2, \dots, N, \\ \text{for } t = 1, 2, \dots, T, \end{cases}$$

All variables nonnegative.

Explain the choice of decision variables, objective function, and constraints. Make a detailed discussion of the model assumptions. Estimate the number of constraints, continuous variables, and integer variables in the model. Is it feasible to solve the model by standard branch-and-bound procedures?

26. An investor is considering a total of I possible investment opportunities ($i = 1, 2, \dots, I$), during a planning horizon covering T time periods ($t = 1, 2, \dots, T$). A total of b_{it} dollars is required to initiate investment i in period t . Investment in project i at time period t provides an income stream $a_{i,t+1}, a_{i,t+2}, \dots, a_{i,T}$ in succeeding time periods. This money is available for reinvestment in other projects. The total amount of money available to the investor at the beginning of the planning period is B dollars.
- Assume that the investor wants to maximize the net present value of the net stream of cash flows (c_t is the corresponding discount factor for period t). Formulate an integer-programming model to assist in the investor's decision.
 - How should the model be modified to incorporate timing restrictions of the form:
 - Project j cannot be initiated until after project k has been initiated; or
 - Projects j and k cannot both be initiated in the same period?
 - If the returns to be derived from these projects are uncertain, how would you consider the risk attitudes of the decision-maker? [Hint. See Exercises 22 and 23 in Chapter 3.]
27. The advertising manager of a magazine faces the following problem: For week t , $t = 1, 2, \dots, 13$, she has been allocated a maximum of n_t pages to use for advertising. She has received requests r_1, r_2, \dots, r_B for advertising, bid r_k indicating:
- the initial week i_k to run the ad,

- ii) the duration d_k of the ad (in weeks),
- iii) the page allocation a_k of the ad (half-, quarter-, or full-page),
- iv) a price offer p_k .

The manager must determine which bids to accept to maximize revenue, subject to the following restrictions:

- i) Any ad that is accepted must be run in consecutive weeks throughout its duration.
- ii) The manager cannot accept conflicting ads. Formally, subsets T_j and \bar{T}_j for $j = 1, 2, \dots, n$ of the bids are given, and she may not select an ad from both T_j and \bar{T}_j ($j = 1, 2, \dots, n$). For example, if $T_1 = \{r_1, r_2\}$, $\bar{T}_1 = \{r_3, r_4, r_5\}$, and bid r_1 or r_2 is accepted, then bids r_3, r_4 , or r_5 must all be rejected; if bid r_3, r_4 , or r_5 is accepted, then bids r_1 and r_2 must both be rejected.
- iii) The manager must meet the Federal Communication Commission's balanced advertising requirements. Formally, subsets S_j and \bar{S}_j for $j = 1, 2, \dots, m$ of the bids are given; if she selects a bid from S_j , she must also select a bid from \bar{S}_j ($j = 1, 2, \dots, m$). For example, if $S_1 = \{r_1, r_3, r_8\}$ and $S_2 = \{r_4, r_6\}$, then either request r_4 or r_6 must be accepted if any of the bids r_1, r_3 , or r_8 are accepted.

Formulate as an integer program.

28. The m -traveling-salesman problem is a variant of the traveling-salesman problem in which m salesmen stationed at a home base, city 1, are to make tours to visit other cities, $2, 3, \dots, n$. Each salesman must be routed through some, but not all, of the cities and return to his (or her) home base; each city must be visited by one salesman. To avoid redundancy, the sales coordinator requires that no city (other than the home base) be visited by more than one salesman or that any salesman visit any city more than once.

Suppose that it cost c_{ij} dollars for any salesman to travel from city i to city j .

- a) Formulate an integer program to determine the minimum-cost routing plan. Be sure that your formulation does not permit subtour solutions for any salesman.
- b) The m -traveling-salesman problem can be reformulated as a single-salesman problem as follows: We replace the home base (city 1) by m fictitious cities denoted $1', 2', \dots, m'$. We link each of these fictitious cities to each other with an arc with high cost $M > \sum_{i=1}^n \sum_{j=1}^n |c_{ij}|$, and we connect each fictitious city i' to every city j at cost $c_{i'j} = c_{ij}$. Figure E9.5 illustrates this construction for $m = 3$.

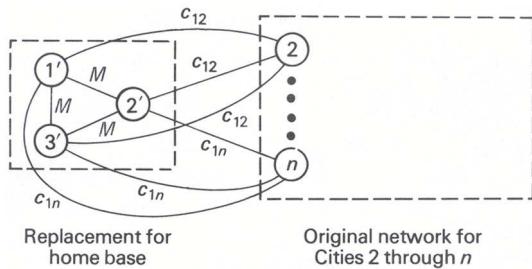


Figure E9.5

Suppose that we solve this reformulation as a single-traveling-salesman problem, but identify each portion of the tour that travels from one of the fictitious cities i' through the original network and back to another fictitious city j' as a tour for one of the m salesmen. Show that these tours solve the m -traveling-salesman problem. [Hint: Can the solution to the single-salesman problem ever use any of the arcs with a high cost M ? How many times must the single salesman leave from and return to one of the fictitious cities?]

29. Many practical problems, such as fuel-oil delivery, newspaper delivery, or school-bus routing, are special cases of the generic vehicle-routing problem. In this problem, a fleet of vehicles must be routed to deliver (or pick up) goods from a depot, node 0, to n drop-points, $i = 1, 2, \dots, n$.

Let

Q_k = Loading capacity of the k th vehicle in the fleet ($k = 1, 2, \dots, m$);

d_i = Number of items to be left at drop-point i ($i = 1, 2, \dots, n$);

t_i^k = Time to unload vehicle k at drop-point i ($i = 1, 2, \dots, n$; $k = 1, 2, \dots, m$);

t_{ij}^k = Time for vehicle k to travel from drop-point i to drop-point

j ($i = 0, 1, \dots, n$; $j = 0, 1, \dots, n$; $k = 1, 2, \dots, m$)

c_{ij}^k = Cost for vehicle k to travel from node i to node j ($i = 0, 1, \dots, n$;

$j = 0, 1, \dots, n$; $k = 1, 2, \dots, m$).

If a vehicle visits drop-point i , then it fulfills the entire demand d_i at that drop-point. As in the traveling or m -traveling salesman problem, only one vehicle can visit any drop-point, and no vehicle can visit the same drop-point more than once. The routing pattern must satisfy the service restriction that vehicle k 's route take longer than T_k time units to complete.

Define the decision variables for this problem as:

$$x_{ij}^k = \begin{cases} 1 & \text{if vehicle } k \text{ travels from drop-point } i \text{ to drop-point } j, \\ 0 & \text{otherwise.} \end{cases}$$

Formulate an integer program that determines the minimum-cost routing pattern to fulfill demand at the drop-points that simultaneously meets the maximum routing-time constraints and loads no vehicle beyond its capacity. How many constraints and variables does the model contain when there are 400 drop-points and 20 vehicles?

ACKNOWLEDGMENTS

Exercise 22 is based on the Bradford Wire Company case written by one of the authors. Exercise 23 is extracted from the Muda Cotton Textiles case written by Munakshi Malya and one of the authors.

Exercise 24 is based on the Custom Pilot Chemical Co. case, written by Richard F. Meyer and Burton Rothberg, which in turn is based on "Multiproject Scheduling with Limited Resources: A Zero–One Programming Approach," *Management Science* 1969, by A. A. B. Pritsker, L. J. Watters, and P. M. Wolfe.

In his master's thesis (Sloan School of Management, MIT, 1976), entitled "A Capacity Expansion Planning Model with Bender's Decomposition," M. Lipton presents a more intricate version of the model described in Exercise 25.

The transformation used in Exercise 28 has been proposed by several researchers; it appears in the 1971 book *Distribution Management* by S. Eilson, C. Watson-Gandy, and N. Christofides, Hafner Publishing Co., 1971.

Integer Programming Formulation

1 Integer Programming Introduction

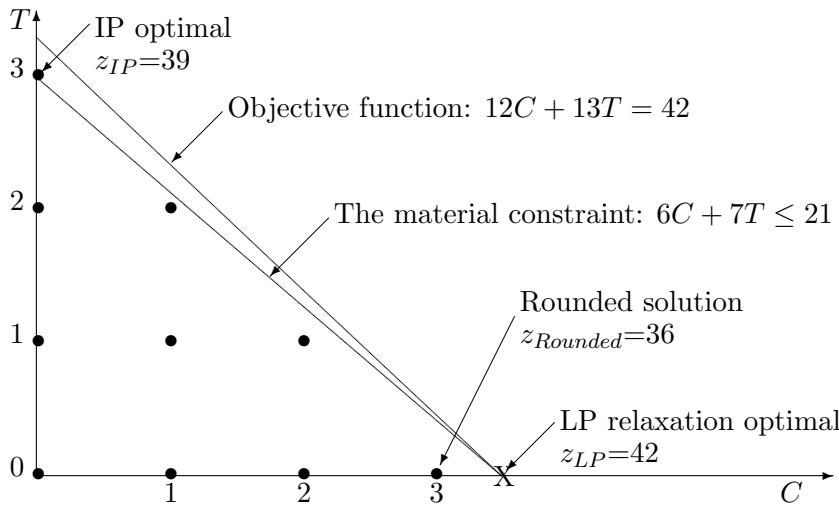
When we introduced linear programs in Chapter 1, we mentioned divisibility as one of the LP assumptions. Divisibility allowed us to consider activities in fractions: We could produce 7.8 units of a product, buy 12500.33 liters of oil, hire 12.123 people for full time, etc. Divisibility assumption is very defensible at times but not always. We can easily buy 12500.33 liters of oil but can not employ 12.123 people. Clearly some activities cannot be done in fractions and must be specified in integers for implementation. As soon as some of the activities are set to be integers, we are in Integer Programming domain. Formally, in an integer program some decision variables are forced to be integers.

We will give a small example here. Suppose we consider producing chairs and tables using only $21\ m^2$ of wood. Each chair (table) requires 6 (7) m^2 of wood. Each chair is sold at \$12 ($\times 10$) and each table is sold at \$13 ($\times 10$). Let C and T denote the number of tables and chairs produced. The IP formulation below maximizes the revenue:

$$\begin{aligned} \text{Maximize : } & 12C + 13T \\ \text{Subject to} \\ & 6C + 7T \leq 21 \quad (1) \\ & C, T \geq 0 \quad (2) \\ & C, T \text{ int} \quad (3) \end{aligned}$$

Note that this formulation differs from the corresponding LP formulation through constraint (3), which says that both C and T are integers.

For a practically oriented mind, solving an IP can be as straightforward as solving the associated LP and rounding the solution. To understand what can go wrong with this approach, we will first solve the IP removing constraint (3) and round down (why not to round up?) the optimal values of C and T to satisfy (3). When the integer constraints are removed from an IP formulation, we obtain an LP formulation. This LP formulation is called the *LP relaxation*. Since we have only two decision variables, we will solve the LP relaxation graphically:



LP solution is $(7/2, 0)$ and is not integer so we round it down to $(3, 0)$. The objective value at $(3, 0)$ is

36. The optimal solution to IP is $(0,3)$ with the objective value 39. 3 units of difference between objective value of the IP optimal and the rounded solution can be significantly higher in more complex problems. As a summary we cannot use rounded solutions of LP relaxations.

The toy example above has illustrated that solving IP's are not straightforward. For toy problems one can evaluate all the integer solutions in the feasible region and pick the best. However, for real problems this approach will take practically infinite amount of time. The solution procedures for IP's are still under development. Two approaches are common: *Branch and Bound technique*, and *Cutting planes*. These techniques are outside the scope of our discussion. Thus, we turn to integer programming formulations.

2 Knapsack Problem

Suppose that Jean Luc (an MBA student) plans to study 40 hours in a week. There are 8 courses he is considering to take in the spring term. They are listed below with the number of hours (per week) required to successfully complete each course.

Operations Research	Accout.	Info. Tech.	Finance	Market.	O. Behav.	Italian Cinema	Russian
9	7	5	8	5	3	7	10

Completing each of these courses increases Jean Luc's chances of finding a job at McKinsey's new Bucharest office. But the contributions of courses towards this dream are different as given below:

Operations Research	Accout.	Info. Tech.	Finance	Market.	O. Behav.	Italian Cinema	Russian
0.10	0.04	0.06	0.12	0.08	0.03	0.04	0.05

Jean Luc wants to choose his spring courses so that he maximizes his chances of getting this job. We will now formulate this situation as an IP. Before going any further, we assume that choice of one course does not affect the choice of any other, i.e., courses can be chosen independently.

2.1 Decision Variables

Jean Luc wants to know the set of the courses he should take. For each course, one by one, if Jean Luc can answer the question whether a course is in the set of courses to be taken, then he will know what he is taking. For example, if OR is in the set, the answer will be a "yes" for OR, otherwise a "no". Let us denote "yes" answers with 1 and "no" answers with 0. Also let x_{OR} be the question. Then, $x_{OR} = 1$ implies a "yes" answer for OR, it is taken. Conversely, $x_{OR} = 0$ implies that it is not taken. With the intuition developed here, we let

$$x_j = \begin{cases} 1 & \text{if course } j \text{ is taken} \\ 0 & \text{otherwise} \end{cases}.$$

2.2 Constraints

There is only one constraint: 40 hours are available for studies. If OR is taken, it requires 9 hours per week. Otherwise, it requires 0 hours. In general, OR requires $9x_{OR}$ hours per week. Writing hourly requirements for each course and summing those up, we obtain the constraint:

$$9x_{OR} + 7x_{Ac} + 5x_{IT} + 8x_{Fi} + 5x_{Ma} + 3x_{OB} + 7x_{IC} + 10x_{Ru} \leq 40$$

2.3 Objective Function

When Jean Luc takes OR, he increases his chance of getting the job by 0.10. Otherwise his chance does not increase. In general, his chance increases by $0.10x_{OR}$. Writing contributions of each course to his chance

and summing, we obtain the objective function:

$$\text{Maximize } 0.10x_{OR} + 0.04x_{Ac} + 0.06x_{IT} + 0.12x_{Fi} + 0.08x_{Ma} + 0.03x_{OB} + 0.04x_{IC} + 0.05x_{Ru}$$

2.4 IP Formulation

We put the objective function and the constraints together to obtain the formulation below:

$$\text{Maximize } 0.10x_{OR} + 0.04x_{Ac} + 0.06x_{IT} + 0.12x_{Fi} + 0.08x_{Ma} + 0.03x_{OB} + 0.04x_{IC} + 0.05x_{Ru}$$

Subject to

$$\begin{aligned} 9x_{OR} + 7x_{Ac} + 5x_{IT} + 8x_{Fi} + 5x_{Ma} + 3x_{OB} + 7x_{IC} + 10x_{Ru} &\leq 40 \\ x_j &\in \{0, 1\} \end{aligned}$$

So if we have a solution where $x_{OR} = 1$, $x_{Ac} = 1$, $x_{IT} = 1$, $x_{Fi} = 1$, $x_{Ma} = 1$ and $x_{OB} = 1$, then Operations Research, Accounting, Information Technology, Finance, Marketing, Organizational Behavior courses are taken. Taking these courses Jean Luc spends 37 hours per week for his course work. He then increases his chances by 0.43.

In this example all decision variables are binary, i.e., they are either 1 or 0. Such IP's are sometimes called Binary Programs. This example actually first originated from a camper considering what to put (food, soaps, magazines, mosquito repellents, etc.) into his fixed capacity knapsack so that he will have a comfortable camping experience. That is where the name knapsack comes from.

3 Facility Location Problem

This problem is very similar to the transportation problem. There are still n retailers receiving materials from warehouses. Only difference is that we want to decide on warehouse locations as well as flows. There are m locations where we can open up warehouses. These locations are known but whether we open up a warehouse at location i ($1 \leq i \leq m$) is not known. To open up a warehouse we pay a cost of F_i . The rest of data is the same as transportation problem data: supplies are b_i , demand are d_j and unit flow costs are c_{ij} . We want to meet customer demand at minimum cost and will build an IP for that purpose.

3.1 Decision Variables

This problem is a generalization of the transportation problem so we still need to decide on flows from warehouse i to retailer j . Thus, it is apparent that amount of flow from i to j must be a decision variable:

$$x_{ij} = \text{Amount of flow from warehouse } i \text{ to retailer } j$$

However, in addition to flows, we have to answer the question whether warehouse i is opened up. If we answer "yes" to this question, it is opened. Let us denote the question with y_i , so that the "yes" answer is $y_i = 1$ and the "no" answer is $y_i = 0$. Clearly, y_i is a decision variable:

$$y_i = \begin{cases} 1 & \text{if warehouse } i \text{ opened up} \\ 0 & \text{otherwise} \end{cases}.$$

3.2 Constraints

Demand Constraints: Just as in the transportation problem, demand at each retailer must be satisfied.

$$\sum_{i=1}^m x_{ij} \geq d_j \quad j = 1..n$$

Supply Constraints: Supply constraints are a little bit more involved this time. First of all, if warehouse i is not opened up, we can not send any units out of it. In notation, we have:

$$y_i = 0 \implies \sum_{j=1}^n x_{ij} = 0$$

On the other hand, if warehouse i is opened, we can send its supply b_i to retailers:

$$y_i = 1 \implies \sum_{j=1}^n x_{ij} \leq b_i$$

When warehouse i is open its supply is b_i , otherwise it is 0. In general, its $y_i b_i$. Thus, we can write the supply constraint as

$$\sum_{j=1}^n x_{ij} \leq y_i b_i \quad i = 1..m .$$

3.3 Objective Function

Without much discussion, we can directly write down the cost of sending materials from warehouses to retailers:

$$\sum_{i=1}^j \sum_{j=1}^n c_{ij} x_{ij}$$

In addition to flow costs, we pay F_i by opening warehouse i , and 0 otherwise. In either case, we pay $F_i y_i$. The total cost of warehouses is:

$$\sum_{i=1}^m F_i y_i$$

Summing up the flow and warehouse costs, we drive the total cost to be minimized:

$$\text{Minimize } \sum_{i=1}^j \sum_{j=1}^n c_{ij} x_{ij} + \sum_{i=1}^m F_i y_i$$

3.4 IP Formulation

Putting all the constraints and the objective function together we obtain the IP formulation:

$$\begin{aligned} & \text{Minimize} && \sum_{i=1}^j \sum_{j=1}^n c_{ij} x_{ij} + \sum_{i=1}^m F_i y_i \\ & \text{Subject to} && \\ & && \sum_{i=1}^m x_{ij} \geq d_j \quad j = 1..n \\ & && \sum_{j=1}^n x_{ij} \leq y_i b_i \quad i = 1..m \\ & && x_{ij} \geq 0 \quad i = 1..m, j = 1..n \\ & && y_i \in \{0, 1\} \quad i = 1..m \end{aligned}$$

Note that the facility location problem formulation has both binary (y_i) variables and continuous (x_{ij}) variables. When this happens the formulation is called a Mixed Integer Linear Program (MILP).

Sudoku table of 9 subtables, each with 3 rows and 3 columns

1,1;1	1,2;1	1,3;1	1,1;2	1,2;2	1,3;2	1,1;3	1,2;3	1,3;3
2,1;1	2,2;1	2,3;1	2,1;2	2,2;2	2,3;2	2,1;3	2,2;3	2,3;3
3,1;1	3,2;1	3,3;1	3,1;2	3,2;2	3,3;2	3,1;3	3,2;3	3,3;3
1,1;4	1,2;4	1,3;4	1,1;5	1,2;5	1,3;5	1,1;6	1,2;6	1,3;6
2,1;4	2,2;4	2,3;4	2,1;5	2,2;5	2,3;5	2,1;6	2,2;6	2,3;6
3,1;4	3,2;4	3,3;4	3,1;5	3,2;5	3,3;5	3,1;6	3,2;6	3,3;6
1,1;7	1,2;7	1,3;7	1,1;8	1,2;8	1,3;8	1,1;9	1,2;9	1,3;9
2,1;7	2,2;7	2,3;7	2,1;8	2,2;8	2,3;8	2,1;9	2,2;9	2,3;9
3,1;7	3,2;7	3,3;7	3,1;8	3,2;8	3,3;8	3,1;9	3,2;9	3,3;9

Subtable k

1,1; k	1,2; k	1,3; k
2,1; k	2,2; k	2,3; k
3,1; k	3,2; k	3,3; k

Within subtable k ,
 i, j, k is the entry
in the i th row and
 j th column

Figure 1: Sudoku table.

4 Sudoku Problem

Sudoku is a number game played on a 9×9 table; 9 rows and 9 columns. We consider this table as 9 subtables, each with 3 rows and 3 columns. The cell in row i , column j of subtable k is denoted by $(i, j; k)$. See Figure 1.

In the sudoku game, each cell in the sudoku table must be filled with a number from 1 to 9. But each number can appear exactly once in each row of the sudoku table, once in each column of the sudoku table and once in each subtable. Some numbers already appear in some cells.

4.1 Decision Variables

Let $y_{i,j;k}^m$ be 1 when the cell $(i, j; k)$ contains number m ; 0 otherwise.

4.2 Constraints

Suppose that some numbers already appear in some cells:

- 3 appears in $(1, 2; 1)$, so $y_{1,2;1}^3 = 1$.
- 7 appears in $(2, 1; 1)$, so $y_{2,1;1}^7 = 1$.
- 6 appears in $(3, 3; 1)$, so $y_{3,3;1}^6 = 1$.
- ...

- 4 appears in $(1, 1; 9)$, so $y_{1,1;9}^4 = 1$.
- 2 appears in $(2, 1; 9)$, so $y_{2,1;9}^2 = 1$.
- 5 appears in $(3, 2; 9)$, so $y_{3,2;9}^5 = 1$.

Let us make a list of number m already placed in cell $(i, j; k)$ listed above and call it list \mathcal{L} . A member of the list is denoted by $[m @ (i, j; k)]$. The constraint to specify already given numbers is

$$y_{i,j;k}^m = 1 \text{ for each given number } m \text{ in position } (i, j; k).$$

By using the list, we write these constraints with short-hand notation

$$y_{i,j;k}^m = 1 \text{ for } [m @ (i, j; k)] \in \mathcal{L}. \quad (1)$$

Now we write constraints so that each number can appear exactly once in each row, each column and each subtable. The cell indices in the first row of the table is $(1, 1; 1), (1, 2; 1), (1, 3; 1), (1, 1; 2), (1, 2; 2), (1, 3; 2), (1, 1; 3), (1, 2; 3), (1, 3; 3)$. To place number $m = 1$ once in this row, we write

$$y_{1,1;1}^1 + y_{1,2;1}^1 + y_{1,3;1}^1 + y_{1,1;2}^1 + y_{1,2;2}^1 + y_{1,3;2}^1 + y_{1,1;3}^1 + y_{1,2;3}^1 + y_{1,3;3}^1 = 1.$$

To place number $m = 2$ once in this row, we write

$$y_{1,1;1}^2 + y_{1,2;1}^2 + y_{1,3;1}^2 + y_{1,1;2}^2 + y_{1,2;2}^2 + y_{1,3;2}^2 + y_{1,1;3}^2 + y_{1,2;3}^2 + y_{1,3;3}^2 = 1.$$

In general, to place number m once in this row, we write

$$y_{1,1;1}^m + y_{1,2;1}^m + y_{1,3;1}^m + y_{1,1;2}^m + y_{1,2;2}^m + y_{1,3;2}^m + y_{1,1;3}^m + y_{1,2;3}^m + y_{1,3;3}^m = 1.$$

Consider the second row of the table and cells $(2, 1; 1), (2, 2; 1), (2, 3; 1), (2, 1; 2), (2, 2; 2), (2, 3; 2), (2, 1; 3), (2, 2; 3), (2, 3; 3)$ in there. To place number m once in this row, we write

$$y_{2,1;1}^m + y_{2,2;1}^m + y_{2,3;1}^m + y_{2,1;2}^m + y_{2,2;2}^m + y_{2,3;2}^m + y_{2,1;3}^m + y_{2,2;3}^m + y_{2,3;3}^m = 1.$$

Similarly, to place number m once in the third row, we write

$$y_{3,1;1}^m + y_{3,2;1}^m + y_{3,3;1}^m + y_{3,1;2}^m + y_{3,2;2}^m + y_{3,3;2}^m + y_{3,1;3}^m + y_{3,2;3}^m + y_{3,3;3}^m = 1.$$

We can shorten this constraint as

$$\sum_{j=1}^3 y_{3,j;1}^m + y_{3,j;2}^m + y_{3,j;3}^m = 1 \quad \text{for } m = 1, \dots, 9.$$

Indeed, the constraints for the first 3 rows are

$$\sum_{j=1}^3 y_{i,j;1}^m + y_{i,j;2}^m + y_{i,j;3}^m = 1 \quad \text{for } i = 1, 2, 3, m = 1, \dots, 9.$$

These can also be shortened for the first three rows as

$$\sum_{k=1}^3 \sum_{j=1}^3 y_{i,j;k}^m = 1 \quad \text{for } i = 1, 2, 3, m = 1, \dots, 9.$$

For rows 4, 5, 6, we have

$$\sum_{k=4}^6 \sum_{j=1}^3 y_{i,j;k}^m = 1 \quad \text{for } i = 1, 2, 3, m = 1, \dots, 9.$$

For rows 7, 8, 9, we have

$$\sum_{k=7}^9 \sum_{j=1}^3 y_{i,j;k}^m = 1 \quad \text{for } i = 1, 2, 3, m = 1, \dots, 9.$$

In summary, to make sure that each number appears exactly once in each row, we require

$$\sum_{k=1}^3 \sum_{j=1}^3 y_{i,j;k}^m = 1; \quad \sum_{k=4}^6 \sum_{j=1}^3 y_{i,j;k}^m = 1; \quad \sum_{k=7}^9 \sum_{j=1}^3 y_{i,j;k}^m = 1 \quad \text{for } i = 1, 2, 3, m = 1, \dots, 9. \quad (2)$$

We also need constraints to ensure that each number appears only once in each column. For the first column, we require

$$\sum_{k \in \{1, 4, 7\}} \sum_{i=1}^3 y_{i,1;k}^m = 1 \quad \text{for } m = 1, \dots, 9.$$

For the second and third colun we need

$$\sum_{k \in \{1, 4, 7\}} \sum_{i=1}^3 y_{i,2;k}^m = 1; \quad \sum_{k \in \{1, 4, 7\}} \sum_{i=1}^3 y_{i,3;k}^m = 1 \quad \text{for } m = 1, \dots, 9.$$

Then, the constraints for the first three coluns are

$$\sum_{k \in \{1, 4, 7\}} \sum_{i=1}^3 y_{i,j;k}^m = 1 \quad \text{for } j = 1, 2, 3, m = 1, \dots, 9.$$

The constraints for columns 4,5,6 are

$$\sum_{k \in \{2, 5, 8\}} \sum_{i=1}^3 y_{i,j;k}^m = 1 \quad \text{for } j = 1, 2, 3, m = 1, \dots, 9.$$

The constraints for columns 7,8,9 are

$$\sum_{k \in \{3, 6, 9\}} \sum_{i=1}^3 y_{i,j;k}^m = 1 \quad \text{for } j = 1, 2, 3, m = 1, \dots, 9.$$

In summary, to make sure that each number appears exactly once in each column, we require

$$\sum_{k \in \{1, 4, 7\}} \sum_{i=1}^3 y_{i,j;k}^m = 1; \quad \sum_{k \in \{2, 5, 8\}} \sum_{i=1}^3 y_{i,j;k}^m = 1; \quad \sum_{k \in \{3, 6, 9\}} \sum_{i=1}^3 y_{i,j;k}^m = 1 \quad \text{for } j = 1, 2, 3, m = 1, \dots, 9. \quad (3)$$

We write constraints to make sure that each subtable has only one number m :

$$\sum_{i=1}^3 \sum_{j=1}^3 y_{i,j;k}^m = 1 \quad \text{for } k, m = 1, \dots, 9. \quad (4)$$

Finally, we need to ensure that each cell has a number in it:

$$\sum_{m=1}^9 y_{i,j;k}^m = 1 \quad \text{for } k = 1, \dots, 9, i, j = 1, 2, 3. \quad (5)$$

Now we are ready to put all the constraints in (1-5) together:

$$\begin{aligned}
y_{i,j;k}^m &= 1 \quad \text{for } [m @ (i, j; k)] \in \mathcal{L}, \\
\sum_{k=1}^3 \sum_{j=1}^3 y_{i,j;k}^m &= 1; \quad \sum_{k=4}^6 \sum_{j=1}^3 y_{i,j;k}^m = 1; \quad \sum_{k=7}^9 \sum_{j=1}^3 y_{i,j;k}^m = 1 \quad \text{for } i = 1, 2, 3, m = 1, \dots, 9, \\
\sum_{k \in \{1,4,7\}} \sum_{i=1}^3 y_{i,j;k}^m &= 1; \quad \sum_{k \in \{2,5,8\}} \sum_{i=1}^3 y_{i,j;k}^m = 1; \quad \sum_{k \in \{3,6,9\}} \sum_{i=1}^3 y_{i,j;k}^m = 1 \quad \text{for } j = 1, 2, 3, m = 1, \dots, 9, \\
\sum_{i=1}^3 \sum_{j=1}^3 y_{i,j;k}^m &= 1 \quad \text{for } k, m = 1, \dots, 9, \\
\sum_{m=1}^9 y_{i,j;k}^m &= 1 \quad \text{for } k = 1, \dots, 9, i, j = 1, 2, 3, \\
y_{i,j;k}^m &\in \{0, 1\}.
\end{aligned}$$

4.3 Objective Function

To finish the formulation we need to write an objective function. In Sudoku game, there is no objective as long as numbers are placed as they should according to constraints in (1-5). Thus we do not prefer any $y_{i,j;k}^m = 1$ to $y_{i,j;k}^m = 0$. Thus, the objective can be maximizing $0y_{1,1;1}^1$ or minimizing $0y_{3,3;9}^9$. They will all give us the objective value of zero, but what we care in Sudoku is only the placement of the numbers in the table!

4.4 IP Formulation

Max $0y_{1,1;1}^1$

Subject to

$$\begin{aligned}
y_{i,j;k}^m &= 1 \quad \text{for } [m @ (i, j; k)] \in \mathcal{L}, \\
\sum_{k=1}^3 \sum_{j=1}^3 y_{i,j;k}^m &= 1; \quad \sum_{k=4}^6 \sum_{j=1}^3 y_{i,j;k}^m = 1; \quad \sum_{k=7}^9 \sum_{j=1}^3 y_{i,j;k}^m = 1 \quad \text{for } i = 1, 2, 3, m = 1, \dots, 9, \\
\sum_{k \in \{1,4,7\}} \sum_{i=1}^3 y_{i,j;k}^m &= 1; \quad \sum_{k \in \{2,5,8\}} \sum_{i=1}^3 y_{i,j;k}^m = 1; \quad \sum_{k \in \{3,6,9\}} \sum_{i=1}^3 y_{i,j;k}^m = 1 \quad \text{for } j = 1, 2, 3, m = 1, \dots, 9, \\
\sum_{i=1}^3 \sum_{j=1}^3 y_{i,j;k}^m &= 1 \quad \text{for } k, m = 1, \dots, 9, \\
\sum_{m=1}^9 y_{i,j;k}^m &= 1 \quad \text{for } k = 1, \dots, 9, i, j = 1, 2, 3, \\
y_{i,j;k}^m &\in \{0, 1\}.
\end{aligned}$$

5 Solved Exercises

- Merrill Lynch is considering investments into 6 projects: A, B, C, D, E and F. Each project has an initial cost, an expected profit rate (one year from now) expressed as a percentage of the initial cost,

and an associated risk of failure. These numbers are given in the table below:

	A	B	C	D	E	F
Initial cost (in M)	1.3	0.8	0.6	1.8	1.2	2.4
Profit rate	10%	20%	20%	10%	10%	10%
Failure risk	6%	4%	6%	5%	5%	4%

- a) Provide a formulation to choose the projects that maximize total expected profit, such that Merrill Lynch does not invest more than 4M dollars and its average failure risk is not over 5%. For example, if Merrill Lynch invests only into A and B, it invests only 2.1M dollars and its average failure risk is $(6\%+4\%)/2=5\%$.
- b) Suppose that if A is chosen, B must be chosen. Modify your formulation.
- c) Suppose that if C **and** D are chosen, E must be chosen. Mofdfy your formulation.

Solution: a) Let $y_A = 1$ if project A is chosen, $y_A = 0$ otherwise. Define y_B, y_C, y_D, y_E, y_F similarly.

$$\begin{aligned}
 & \text{Max} \quad 0.13y_A + 0.16y_B + 0.12y_C + 0.18y_D + 0.12y_E + 0.24y_F \\
 & \text{St} \\
 & 1.3y_A + 0.8y_B + 0.6y_C + 1.8y_D + 1.2y_E + 2.4y_F \leq 4 \\
 & 0.06y_A + 0.04y_B + 0.06y_C + 0.05y_D + 0.05y_E + 0.04y_F \leq 0.05(y_A + y_B + y_C + y_D + y_E + y_F) \\
 & y_A, y_B, y_C, y_D, y_E, y_F \in \{0, 1\}.
 \end{aligned}$$

- b) Add : $y_B \geq y_A$.
- c) Add : $y_E \geq y_C + y_D - 1$.

6 Exercises

1. After the bad start to the season and unexpected injuries, Dallas Cowboys wants to sign 3 new players. There are five players under consideration. Player 1, 2 and 4 can play the quarter back position and Cowboys want to bring in at least one new player for this position. Each player has a strength measured in s_j and Cowboys want to add at least S units of strength to the team by hiring some of these 5 players. On the other hand, for public relations purposes Cowboys will be careful while signing people who are prone to be convicted for assaults. Player j is likely to have a_j assaults per season and Cowboys imposes a quota of A assaults per season for these players.
 - a) If player j demands the salary c_j , make up an IP to decide on which players should be signed to minimize the signing budget.
 - b) If player 1 and 2 are bodies and will come to Dallas only together, add a constraint to a) to guarantee that either they are signed together or they are not signed at all.
 - c) If player 1 hates player 3 and will not come if player 3 comes, add an appropriate constraint that does not allow signing player 1 and 3 together.
2. Suppose that you will play a word-construction game. You are assigned n_a, n_b, \dots, n_z copies of letters a, b, \dots, z respectively. That is, if $n_b = 8$ then you can use the letter b at most 8 times in your words. These words must be in an English dictionary, say we use D to denote all the words in the dictionary. Then, $abacus \in D$, $abode \in D$, $zinc \in D$, basically D is the set of all English words. Suppose that for every word you construct, you obtain points equivalent to the length of that word. For example, if you are assigned $n_a = 1, n_e = 2, n_i = 5, n_m = 4, n_n = 1, n_x = 1, n_z = 2$ ($a, e, e, i, i, i, i, m, m, m, n, x, z, z$) and no other letter, you should construct words *minimize* and *maximize* scoring 16 points instead of *zen* and *maze* scoring 7 points. For an assignment of letters (i.e., n_a, n_b, \dots, n_z given and known), provide an IP that will choose words so that your score is

maximized. Hint: Let $x_i = 1$ if the i th word in D is constructed, 0 otherwise. Let a_i be the number of letter “a”’s in the i th word, similarly define b_i to z_i for all $i \in D$.

3. Consider the Market Clearing model presented in the Formulation Chapter. Suppose that there is a fixed cost K_{ij} of using each transportation channel from supplier i to buyer j . This cost is independent of the units that are transported from i to j , it is incurred if there is some, no matter how small or large, shipment. In addition, we want to enforce that each product k is supplied by at most one supplier. Note that this restriction will make the market for each product a monopoly. Provide a formulation to market clearing problem with these additions.
4. Suppose that you are moving to a new apartment and you have only two boxes to pack your stuff in. The first box carries 22 kg and the second carries 28 kg. The weight and the value of your belongings are:

Item	A	B	C	D	E	F	G	H
Weight (kg)	10	9	15	3	11	6	3	4
Value (in \$100)	5	2	7	6	1	6	8	6

- a) Provide a formulation to maximize the value of items you can carry with these two boxes.
- b) Since the box capacity is limited, you cannot fit all your items into boxes. Can you identify an item, which will always (in all optimal solutions) be put in one of the boxes? Justify your answer.
5. Consider a smaller Sudoku game played on 4 subtables each with 2 rows and 2 columns. Following numbers are already placed:

- 2 appears in (1, 2; 1)
- 3 appears in (2, 1; 1)
- 3 appears in (1, 2; 2)
- 1 appears in (1, 1; 3)
- 3 appears in (2, 2; 3)
- 3 appears in (1, 1; 4)
- 1 appears in (2, 2; 4)

We are to place numbers from 1 to 4 into the remaining cells in this smaller Sudoku table. Provide an explicit IP formulation (without using sum notation) to solve this Sudoku problem.

CS570 Fall 2018: Analysis of Algorithms Exam III

	Points		Points
Problem 1	20	Problem 5	10
Problem 2	8	Problem 6	16
Problem 3	14	Problem 7	12
Problem 4	20		
	Total	100	

Instructions:

1. This is a 2-hr exam. Closed book and notes
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.

1) 20 pts

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

[**TRUE/FALSE**]

Let X be a decision problem. If we prove that X is in the class NP and give a poly-time reduction from X to 3-SAT, we can conclude that X is NP-complete.

[**TRUE/FALSE**]

Let A be an algorithm that operates on a list of n objects, where n is a power of two. A spends $\Theta(n^2)$ time dividing its input list into two equal pieces and selecting one of the two pieces. It then calls itself recursively on that list of $n/2$ elements. Then A 's running time on a list of n elements is $O(n)$.

[**TRUE/FALSE**]

If there is a polynomial time algorithm to solve problem A then A is in NP.

[**TRUE/FALSE**]

A pseudo-polynomial time algorithm is always slower than a polynomial time algorithm.

[**TRUE/FALSE**]

In a dynamic programming formulation, the sub-problems must be non-overlapping.

[**TRUE/FALSE**]

A spanning tree of a given undirected, connected graph $G = (V, E)$ can be found in $O(E)$ time.

[**TRUE/FALSE**]

Ford-Fulkerson can return a zero maximum flow for flow networks with non-zero capacities.

[**TRUE/FALSE**]

If $f(n) = \Theta(g(n))$ and $g(n) = \Theta(h(n))$, then $h(n) = \Theta(f(n))$

[**TRUE/FALSE**]

There is a polynomial-time solution for the 0/1 Knapsack problem if all items have the same weight but different values.

[**TRUE/FALSE**]

If there are negative cost edges in a graph but no negative cost cycles, Dijkstra's algorithm still runs correctly.

2) 8 pts

Let $G = (V, E)$ be a simple graph with n vertices. Suppose the weight of every edge of G is one.

Note: In a simple graph there is at most one edge directly connecting any two nodes.

For example, between nodes u and v there will be at most one edge uv .

- (a) What is the weight of a minimum spanning tree of G ? (1 pt)
- (b) Suppose we change the weight of two edges of G to $1/2$. What is the weight of the minimum spanning tree of G ? (2 pts)
- (c) Suppose we change the weight of three edges of G to $1/2$. What is the minimum and maximum possible weights for the the minimum spanning tree of G ? (2 pts)
- (d) Suppose we change the weight of $k < V$ edges of G to $1/2$. What is the minimum and maximum possible weights for the the minimum spanning tree of G ? (3 pts)
1. Any spanning tree of G has exactly $V - 1$ edges, since all the weights are one its total weight is $V - 1$.
 2. To answer this question, consider running Kruskal on G , it will first pick the two lighter edges as they cannot form a cycle (the graph is simple), then $V - 3$ other edge each with weight one. Therefore, the total value is $V - 3 + 2(1/2) = V - 2$.
 3. Use the similar approach as (b). Kruskal first selects two light edges. There are two possibilities for the third edge: (i) it forms a cycle with the first two edges or (ii) it does not form a cycle with the first two edges not. In case (i), Kruskal selects two edges of weight $1/2$ and $V - 3$ edges of weight 1, therefore, the total weight is $V - 3 + 2(1/2) = V - 2$. In case (ii), Kruskal selects three edges of weight $1/2$ and $V - 4$ edges of weight 1, therefore, the total weight is $V - 4 + 3(1/2) = V - 5/2$.
 4. Similar to (c), the minimum weight happens when the k light edges do not form any cycle. In this case, MST contains k edges of weight $1/2$ and $n - 1 - k$ edges of weight 1. Therefore, its weight is $k/2 + n - 1 - k = n - k/2 - 1$. The maximum weight happens if the k edges span as few vertices as possible. This happens when the k edges are part of an almost *complete graph*. Let h be a variable denoting the number of nodes in this subgraph such that the number of edges in a complete graph comprised of these edges nodes exceeds k . In particular, define h to be the smallest number such that, the binomial coefficient $\text{Binomial}[h,2] > k$, ie., $(h \text{ choose } 2) > k$. Accordingly, we can pack all k light edges between h vertices. Consequently, the MST has $h - 1$ edges of weight $1/2$ and $n - 1 - (h - 1)$ edges of weight 1. Therefore, its weight is $n - h/2 - 1/2$.

Rubric:

For each part (a,b,c,d), No partial marking.

This also applies to the parts where minimum and maximum is asked.

If either is wrong then there is no partial marks for that part.

Q3. 14 pts

Recall that in the discussion class we showed that the Bipartite **Undirected** Hamiltonian Cycle problem is NP-complete. Now in the Bipartite **Directed** Hamiltonian Cycle problem, we are given a bipartite directed graph and asked whether there is a simple cycle which visits every node exactly once. Note that this problem might potentially be easier than Hamiltonian Cycle because it assumes a directed bipartite graph. Prove that Bipartite Directed Hamiltonian Cycle is in fact still NP-Complete.

Solution:

- i) This problem is NP. Given a candidate path, we just check the nodes along the given path one by one to see if every node in the network is visited exactly once and if each consecutive node pair along the path are joined by an edge. (3)
- ii) To prove the problem is NP-complete, we reduce Directed Hamiltonian Cycle (DHC) problem to Bipartite Directed Hamiltonian Cycle (BDHC) problem. (2)

Given an arbitrary directed graph G , we split each vertex v in G into two vertex v_{in} and v_{out} . Here v_{in} connects all the incoming edges to v in G ; v_{out} connects all the outgoing edges from v in G . Moreover, we connect one directed edge from v_{in} to v_{out} . After doing these operations for each node in G , we form a new graph G' . (3)

Here G' is bipartite graph, because we can color each v_{in} “blue” and each v_{out} “red” without any coloring conflict.

If there is DHC in G , then there is a BDHC in G' . We can replace each node v on the DHC in G into consecutive nodes v_{in} and v_{out} , and (v_{in}, v_{out}) is an edge in G' . Then the new path is a BDHC in G' . (3)

On the other hand, if there is BDHC in G' , then there is a DHC in G . Note that if v_{in} is on the BDHC, v_{out} must be the successive node on the BDHC. Then we can merge each node pair (v_{in}, v_{out}) on the BDHC in G' into node v and form a DHC in G . (3)

In sum, G has DHC if and only if G' has a BDHC. This completes the reduction, and we confirm that the given problem is NP-complete

Alternate solution:

We can also use Undirected Hamiltonian Cycle for the reduction.

Prove that it's NP as before. (3)

Note that you are using Bipartite Undirected Hamiltonian Cycle for reduction. (2)

Given an arbitrary undirected bipartite graph G , convert G to G' so the vertices in G' stay the same, but all undirected edges are converted to directed edges in **both directions**. G' remains bipartite. (3)

If there is an undirected Hamiltonian cycle in G, you can use the corresponding edges(in either direction) to find a Directed Hamiltonian cycle in in G'. (3)

On the other hand, if there is a directed Hamiltonian cycle in G', you could convert the corresponding edges to undirected edges and find the equivalent cycle in G. (3)

Common mistakes:

Major mistake in checking for NP: -2

Checking for edges instead of vertices: -1

Missing the NP check altogether: -3

Mistake in the conversion step: -2

Missing the two-way proof: -4

Missing either side of the proof: -3

Incomplete explanation as to why the proof holds: -2

If you are proposing to reduce a certain NP-Complete problem, but are going with a solution corresponding a different NP-Complete problem, this shows misunderstanding of the concept: -7

If you are proposing to reduce an incorrect (and irrelevant) NP-complete problem: -8

4) 20 pts.

Suppose you want to sell n roses. You need to group the roses into multiple bouquets with different sizes. The value of each bouquet depends on the number of roses you used. In other words, we know that a bouquet of size i will sell for $v[i]$ dollars. Provide an algorithm to find the maximum possible value of the roses by deciding how to group them into different-sized bouquets.

Here is an example:

Input: $n = 5$, $v[1..n] = [2,3,7,8,10]$ (i.e. a bouquet of size 1 is \$2, bouquet of size 2 is \$3, ..., and finally a single bouquet of size $n=5$ is \$10.

Expected Output = 11 (by grouping the roses into bouquets of size 1, 1, and 3)

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

$OPT(j)$ is the maximum value of j roses.

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

$OPT(j) = \text{MAX } (OPT(j), \{OPT(j-i) + v[i]\} \text{ for all sizes } 0 \leq i < j)$

i.If iteration of i from 0 to j is missing => -1 points

ii.If $OPT(j)$ is missing in the MAX term => -1 points

iii.0 points for wrong recursion

c) Using the recurrence formula in part b, write pseudocode to compute the maximum possible value of the n roses. (6 pts)

Make sure you have initial values properly assigned. (2 pts)

```
OPT[0] = 0
OPT[1] = v[1]
For (int j=2; j < n; j++)
    max = 0
    For (int i=1; i < j; i++)
        If (i<=j && max < OPT[j-i]+v[i] )
            max = OPT[j-i]+v[i]
    OPT[j]=max
```

i) Missing one loop => -1 point

ii) Missing two loops => -2 points

iii) If in b) you got x points, you get x points in the pseudo code. Apart from the case where you missed the iteration in b, where you will get x+1 points in this question

iv) 0 points for no pseudocode or any modification from b.

d) Compute the runtime of the algorithm described in part c and state whether your solution runs in polynomial time or not (2 pts)

e) d) $O(n^2)$

5) 10 pts

Give a tight asymptotic upper bound (O notation) on the solution to each of the following recurrences. You need not justify your answers.

$$T(n) = 2T(n/8) + n$$

Solution: $(n^{1/3} \lg n)$ by Case 2 of the Master Method.

$$T(n) = T(n/3) + T(n/4) + 5n$$

Solution: Use brute force method

$$\begin{aligned} T(n) &= cn + (7/12) cn + (7/12)^2 cn + \dots \\ &= (n) \end{aligned}$$

Rubric:

No partial marks. 5 points for each correct answer.

6) 16 pts

There are n people and m jobs. You are given a payoff matrix, C , where C_{ij} represents the payoff for assigning person i to do job j . Each applicant has a subset of jobs that he/she is interested in. Each job opening can only accept one applicant and a job applicant can be appointed for only one job. You need to find an assignment of jobs to applicants that maximizes the total payoff of your assignment. Write an **Integer Linear Program** (with discrete variables) to solve this problem.

Solution

We are looking for a perfect matching on a bipartite graph of the minimal cost.

Let $x_{ij} = \begin{cases} 1, & \text{if edge}(i,j) \text{ is in matching} \\ 0, & \text{otherwise} \end{cases}$

Objective: $\sum_{\substack{1 \leq i \leq m \\ 1 \leq j \leq n}} c_{ij} \cdot x_{ij}$

Subject to:

1. $\sum_j x_{ij} \leq 1$, where $i = 1, 2, \dots, n$
2. $\sum_i x_{ij} \leq 1$, where $j = 1, 2, \dots, n$
3. $x_{ij} \in \{0,1\}$, $\forall i, j$
4. $x_{ij} = 0$, for all jobs j that applicant i is not interested in.

Rubric:

- 4 if objective function is incorrect.
- 6 if you mention equal to instead of less than equal to sign for constraints 1 and 2
- 2 if constraint 3. is not mentioned, but only is defined to take values 0 or 1.
- 2 if discrete variable is not defined.
- 0 points awarded if you use a max-flow, min-cut formulation

7) 12 pts

The Maximum Acyclic Subgraph is stated as follows: Given a directed graph find an acyclic subgraph of that contains as many arcs (i.e. directed edges) as possible.

Give a 2-approximation algorithm for this problem.

Hint: Consider using an arbitrary ordering of the vertices in G.

Solution:

1. Pick an arbitrary vertex ordering. This partitions the arcs into two acyclic subgraphs and . Here A_1 is the set of arcs where $i < j$ and A_2 is the set of arcs where $i > j$. Thus at least one of A_1 or A_2 contains half the arcs of G . The maximum acyclic subgraph contains at most the total number of arcs in G , so we have a factor 2-approximation algorithm (12)
2. Divide the set of nodes into two nonempty sets, A and B . Consider the set of edges from A to B and the set of edges from B to A . Throw out the edges from the smaller set and keep the edges from the larger set (break ties arbitrarily). Recursively perform the algorithm on A and B individually (12)

Rubric:

If you are only considering the forward or backward edges (-2)

If you start with one node and add the forward/backward edges to/from neighbors, your answer won't work for disconnected graphs (-2)

If your final answer is not 0.5 approximation (-6)

If your answer is not acyclic (-6)

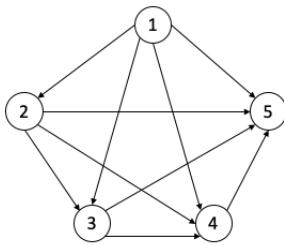
If your algorithm is vague and not implementable (for example if you said, "we find max independent set", or "we find the topological order") (0)

If your algorithm has conceptual flaws (for example if you are adding an edge to the sub graph and also deleting it from the sub graph) (0 pts)

Some of the answers that do not work:

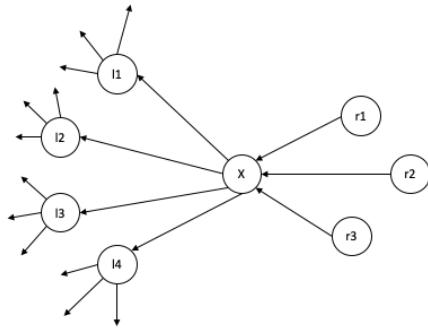
1. If your final answer is a tree (a DFS or a BFS or any other kinds of trees):

You can imagine an acyclic complete directed graph such as below. Your algorithm should return at least half of the edges ($n * (n-1) / 4$) but a tree can have $n-1$ edges. Thus it is not 0.5 approximation (6 pts).



2. If you start adding/removing edges to/from the graph, until it is acyclic. Your algorithm can choose an edge that is repeated in many cycles and end up removing all the other edges in those cycles.

You can try your method on the following graph (All $l(i)$ nodes are connected to all $r(j)$ nodes). If your algorithm chooses all the $(r(j), X)$ and $(X, l(i))$ edges (7 edges), you cannot add any $(l(i), r(j))$ edges (12 edges). Your final answer (7) is less than half of the best answer ($16/2 = 8$) so it is not 0.5 approximation.



CS570 Fall 2018: Analysis of Algorithms Exam III

	Points		Points
Problem 1	20	Problem 5	10
Problem 2	8	Problem 6	16
Problem 3	14	Problem 7	12
Problem 4	20		
	Total	100	

Instructions:

1. This is a 2-hr exam. Closed book and notes
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.

1) 20 pts

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

[**TRUE/FALSE**]

Let X be a decision problem. If we prove that X is in the class NP and give a poly-time reduction from X to 3-SAT, we can conclude that X is NP-complete.

[**TRUE/FALSE**]

Let A be an algorithm that operates on a list of n objects, where n is a power of two. A spends $\Theta(n^2)$ time dividing its input list into two equal pieces and selecting one of the two pieces. It then calls itself recursively on that list of $n/2$ elements. Then A 's running time on a list of n elements is $O(n)$.

[**TRUE/FALSE**]

If there is a polynomial time algorithm to solve problem A then A is in NP.

[**TRUE/FALSE**]

A pseudo-polynomial time algorithm is always slower than a polynomial time algorithm.

[**TRUE/FALSE**]

In a dynamic programming formulation, the sub-problems must be non-overlapping.

[**TRUE/FALSE**]

A spanning tree of a given undirected, connected graph $G = (V, E)$ can be found in $O(E)$ time.

[**TRUE/FALSE**]

Ford-Fulkerson can return a zero maximum flow for flow networks with non-zero capacities.

[**TRUE/FALSE**]

If $f(n) = \Theta(g(n))$ and $g(n) = \Theta(h(n))$, then $h(n) = \Theta(f(n))$

[**TRUE/FALSE**]

There is a polynomial-time solution for the 0/1 Knapsack problem if all items have the same weight but different values.

[**TRUE/FALSE**]

If there are negative cost edges in a graph but no negative cost cycles, Dijkstra's algorithm still runs correctly.

2) 8 pts

Let $G = (V, E)$ be a simple graph with n vertices. Suppose the weight of every edge of G is one.

Note: In a simple graph there is at most one edge directly connecting any two nodes.

For example, between nodes u and v there will be at most one edge uv .

- (a) What is the weight of a minimum spanning tree of G ? (1 pt)
- (b) Suppose we change the weight of two edges of G to $1/2$. What is the weight of the minimum spanning tree of G ? (2 pts)
- (c) Suppose we change the weight of three edges of G to $1/2$. What is the minimum and maximum possible weights for the the minimum spanning tree of G ? (2 pts)
- (d) Suppose we change the weight of $k < V$ edges of G to $1/2$. What is the minimum and maximum possible weights for the the minimum spanning tree of G ? (3 pts)
1. Any spanning tree of G has exactly $V - 1$ edges, since all the weights are one its total weight is $V - 1$.
 2. To answer this question, consider running Kruskal on G , it will first pick the two lighter edges as they cannot form a cycle (the graph is simple), then $V - 3$ other edge each with weight one. Therefore, the total value is $V - 3 + 2(1/2) = V - 2$.
 3. Use the similar approach as (b). Kruskal first selects two light edges. There are two possibilities for the third edge: (i) it forms a cycle with the first two edges or (ii) it does not form a cycle with the first two edges not. In case (i), Kruskal selects two edges of weight $1/2$ and $V - 3$ edges of weight 1, therefore, the total weight is $V - 3 + 2(1/2) = V - 2$. In case (ii), Kruskal selects three edges of weight $1/2$ and $V - 4$ edges of weight 1, therefore, the total weight is $V - 4 + 3(1/2) = V - 5/2$.
 4. Similar to (c), the minimum weight happens when the k light edges do not form any cycle. In this case, MST contains k edges of weight $1/2$ and $n - 1 - k$ edges of weight 1. Therefore, its weight is $k/2 + n - 1 - k = n - k/2 - 1$. The maximum weight happens if the k edges span as few vertices as possible. This happens when the k edges are part of an almost *complete graph*. Let h be a variable denoting the number of nodes in this subgraph such that the number of edges in a complete graph comprised of these edges nodes exceeds k . In particular, define h to be the smallest number such that, the binomial coefficient $\text{Binomial}[h,2] > k$, ie., $(h \text{ choose } 2) > k$. Accordingly, we can pack all k light edges between h vertices. Consequently, the MST has $h - 1$ edges of weight $1/2$ and $n - 1 - (h - 1)$ edges of weight 1. Therefore, its weight is $n - h/2 - 1/2$.

Rubric:

For each part (a,b,c,d), No partial marking.

This also applies to the parts where minimum and maximum is asked.

If either is wrong then there is no partial marks for that part.

Q3. 14 pts

Recall that in the discussion class we showed that the Bipartite **Undirected** Hamiltonian Cycle problem is NP-complete. Now in the Bipartite **Directed** Hamiltonian Cycle problem, we are given a bipartite directed graph and asked whether there is a simple cycle which visits every node exactly once. Note that this problem might potentially be easier than Hamiltonian Cycle because it assumes a directed bipartite graph. Prove that Bipartite Directed Hamiltonian Cycle is in fact still NP-Complete.

Solution:

- i) This problem is NP. Given a candidate path, we just check the nodes along the given path one by one to see if every node in the network is visited exactly once and if each consecutive node pair along the path are joined by an edge. (3)
- ii) To prove the problem is NP-complete, we reduce Directed Hamiltonian Cycle (DHC) problem to Bipartite Directed Hamiltonian Cycle (BDHC) problem. (2)

Given an arbitrary directed graph G , we split each vertex v in G into two vertex v_{in} and v_{out} . Here v_{in} connects all the incoming edges to v in G ; v_{out} connects all the outgoing edges from v in G . Moreover, we connect one directed edge from v_{in} to v_{out} . After doing these operations for each node in G , we form a new graph G' . (3)

Here G' is bipartite graph, because we can color each v_{in} “blue” and each v_{out} “red” without any coloring conflict.

If there is DHC in G , then there is a BDHC in G' . We can replace each node v on the DHC in G into consecutive nodes v_{in} and v_{out} , and (v_{in}, v_{out}) is an edge in G' . Then the new path is a BDHC in G' . (3)

On the other hand, if there is BDHC in G' , then there is a DHC in G . Note that if v_{in} is on the BDHC, v_{out} must be the successive node on the BDHC. Then we can merge each node pair (v_{in}, v_{out}) on the BDHC in G' into node v and form a DHC in G . (3)

In sum, G has DHC if and only if G' has a BDHC. This completes the reduction, and we confirm that the given problem is NP-complete

Alternate solution:

We can also use Undirected Hamiltonian Cycle for the reduction.

Prove that it's NP as before. (3)

Note that you are using Bipartite Undirected Hamiltonian Cycle for reduction. (2)

Given an arbitrary undirected bipartite graph G , convert G to G' so the vertices in G' stay the same, but all undirected edges are converted to directed edges in **both directions**. G' remains bipartite. (3)

If there is an undirected Hamiltonian cycle in G, you can use the corresponding edges(in either direction) to find a Directed Hamiltonian cycle in in G'. (3)

On the other hand, if there is a directed Hamiltonian cycle in G', you could convert the corresponding edges to undirected edges and find the equivalent cycle in G. (3)

Common mistakes:

Major mistake in checking for NP: -2

Checking for edges instead of vertices: -1

Missing the NP check altogether: -3

Mistake in the conversion step: -2

Missing the two-way proof: -4

Missing either side of the proof: -3

Incomplete explanation as to why the proof holds: -2

If you are proposing to reduce a certain NP-Complete problem, but are going with a solution corresponding a different NP-Complete problem, this shows misunderstanding of the concept: -7

If you are proposing to reduce an incorrect (and irrelevant) NP-complete problem: -8

4) 20 pts.

Suppose you want to sell n roses. You need to group the roses into multiple bouquets with different sizes. The value of each bouquet depends on the number of roses you used. In other words, we know that a bouquet of size i will sell for $v[i]$ dollars. Provide an algorithm to find the maximum possible value of the roses by deciding how to group them into different-sized bouquets.

Here is an example:

Input: $n = 5$, $v[1..n] = [2,3,7,8,10]$ (i.e. a bouquet of size 1 is \$2, bouquet of size 2 is \$3, ..., and finally a single bouquet of size $n=5$ is \$10.

Expected Output = 11 (by grouping the roses into bouquets of size 1, 1, and 3)

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

$OPT(j)$ is the maximum value of j roses.

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

$OPT(j) = \text{MAX } (OPT(j), \{OPT(j-i) + v[i]\} \text{ for all sizes } 0 \leq i < j)$

i.If iteration of i from 0 to j is missing => -1 points

ii.If $OPT(j)$ is missing in the MAX term => -1 points

iii.0 points for wrong recursion

c) Using the recurrence formula in part b, write pseudocode to compute the maximum possible value of the n roses. (6 pts)

Make sure you have initial values properly assigned. (2 pts)

$OPT[0] = 0$

$OPT[1] = v[1]$

For (int $j=2$; $j < n$; $j ++$)

 max = 0

 For (int $i=1$; $i < j$; $i ++$)

 If ($i \leq j$ && max < $OPT[j-i] + v[i]$)

 max = $OPT[j-i] + v[i]$

$OPT[j] = \text{max}$

i) Missing one loop => -1 point

ii) Missing two loops => -2 points

iii) If in b) you got x points, you get x points in the pseudo code. Apart from the case where you missed the iteration in b, where you will get x+1 points in this question

iv) 0 points for no pseudocode or any modification from b.

d) Compute the runtime of the algorithm described in part c and state whether your solution runs in polynomial time or not (2 pts)

e) d) $O(n^2)$

5) 10 pts

Give a tight asymptotic upper bound (O notation) on the solution to each of the following recurrences. You need not justify your answers.

$$T(n) = 2T(n/8) + n$$

Solution: $(n^{1/3} \lg n)$ by Case 2 of the Master Method.

$$T(n) = T(n/3) + T(n/4) + 5n$$

Solution: Use brute force method

$$\begin{aligned} T(n) &= cn + (7/12) cn + (7/12)^2 cn + \dots \\ &= (n) \end{aligned}$$

Rubric:

No partial marks. 5 points for each correct answer.

6) 16 pts

There are n people and m jobs. You are given a payoff matrix, C , where C_{ij} represents the payoff for assigning person i to do job j . Each applicant has a subset of jobs that he/she is interested in. Each job opening can only accept one applicant and a job applicant can be appointed for only one job. You need to find an assignment of jobs to applicants that maximizes the total payoff of your assignment. Write an **Integer Linear Program** (with discrete variables) to solve this problem.

Solution

We are looking for a perfect matching on a bipartite graph of the minimal cost.

Let $x_{ij} = \begin{cases} 1, & \text{if edge}(i,j) \text{ is in matching} \\ 0, & \text{otherwise} \end{cases}$

Objective: $\sum_{\substack{1 \leq i \leq m \\ 1 \leq j \leq n}} c_{ij} \cdot x_{ij}$

Subject to:

1. $\sum_j x_{ij} \leq 1$, where $i = 1, 2, \dots, n$
2. $\sum_i x_{ij} \leq 1$, where $j = 1, 2, \dots, n$
3. $x_{ij} \in \{0,1\}$, $\forall i, j$
4. $x_{ij} = 0$, for all jobs j that applicant i is not interested in.

Rubric:

- 4 if objective function is incorrect.
- 6 if you mention equal to instead of less than equal to sign for constraints 1 and 2
- 2 if constraint 3. is not mentioned, but only is defined to take values 0 or 1.
- 2 if discrete variable is not defined.
- 0 points awarded if you use a max-flow, min-cut formulation

7) 12 pts

The Maximum Acyclic Subgraph is stated as follows: Given a directed graph find an acyclic subgraph of that contains as many arcs (i.e. directed edges) as possible.

Give a 2-approximation algorithm for this problem.

Hint: Consider using an arbitrary ordering of the vertices in G.

Solution:

1. Pick an arbitrary vertex ordering. This partitions the arcs into two acyclic subgraphs and . Here A_1 is the set of arcs where $i < j$ and A_2 is the set of arcs where $i > j$. Thus at least one of A_1 or A_2 contains half the arcs of G . The maximum acyclic subgraph contains at most the total number of arcs in G , so we have a factor 2-approximation algorithm (12)
2. Divide the set of nodes into two nonempty sets, A and B . Consider the set of edges from A to B and the set of edges from B to A . Throw out the edges from the smaller set and keep the edges from the larger set (break ties arbitrarily). Recursively perform the algorithm on A and B individually (12)

Rubric:

If you are only considering the forward or backward edges (-2)

If you start with one node and add the forward/backward edges to/from neighbors, your answer won't work for disconnected graphs (-2)

If your final answer is not 0.5 approximation (-6)

If your answer is not acyclic (-6)

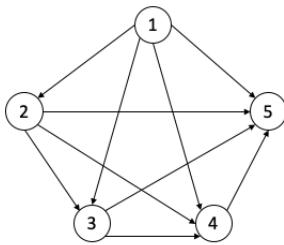
If your algorithm is vague and not implementable (for example if you said, "we find max independent set", or "we find the topological order") (0)

If your algorithm has conceptual flaws (for example if you are adding an edge to the sub graph and also deleting it from the sub graph) (0 pts)

Some of the answers that do not work:

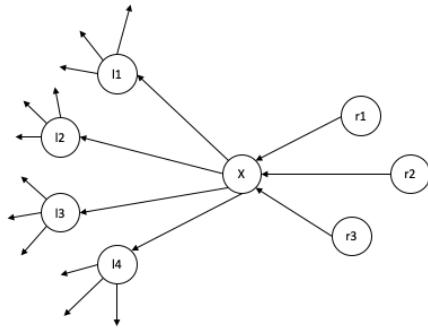
1. If your final answer is a tree (a DFS or a BFS or any other kinds of trees):

You can imagine an acyclic complete directed graph such as below. Your algorithm should return at least half of the edges ($n * (n-1) / 4$) but a tree can have $n-1$ edges. Thus it is not 0.5 approximation (6 pts).



2. If you start adding/removing edges to/from the graph, until it is acyclic. Your algorithm can choose an edge that is repeated in many cycles and end up removing all the other edges in those cycles.

You can try your method on the following graph (All $l(i)$ nodes are connected to all $r(j)$ nodes). If your algorithm chooses all the $(r(j), X)$ and $(X, l(i))$ edges (7 edges), you cannot add any $(l(i), r(j))$ edges (12 edges). Your final answer (7) is less than half of the best answer ($16/2 = 8$) so it is not 0.5 approximation.



CS570
Analysis of Algorithms
Spring 2015
Exam III

Name: _____

Student ID: _____

Email Address: _____

_____ **Check if DEN Student**

	Maximum	Received
Problem 1	20	
Problem 2	16	
Problem 3	16	
Problem 4	16	
Problem 5	16	
Problem 6	16	
Total	100	

Instructions:

1. This is a 2-hr exam. Closed book and notes
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.

1) 20 pts

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

[**TRUE**]

If $\text{SAT} \leq_P A$, then A is NP-hard.

[**FALSE**]

If a problem X can be reduced to a known NP-hard problem, then X must be NP-hard.

[**TRUE**]

If P equals NP, then NP equals NP-complete.

[**FALSE**]

Let X be a decision problem. If we prove that X is in the class NP and give a poly-time reduction from X to Hamiltonian Cycle, we can conclude that X is NP-complete.

[**TRUE**]

The recurrence $T(n) = 2T(n/2) + 3n$, has solution $T(n) = \theta(n \log(n^2))$.

[**FALSE**]

On a connected, directed graph with only positive edge weights, Bellman-Ford runs asymptotically as fast as Dijkstra.

[**TRUE**]

Linear programming is at least as hard as the Max Flow problem in a flow network.

[**TRUE**]

If you are given a maximum s-t flow in a graph then you can find a minimum s-t cut in time $O(m)$ where m is the number of the edges in the graph.

[**TRUE**]

Fibonacci heaps can be used to make Dijkstra's algorithm run in $O(|E| + |V| \log|V|)$ time on a graph $G=(V,E)$

[**FALSE**]

A graph with non-unique edge weights will have at least two minimum spanning trees

2) 16 pts

Given a graph $G=(V, E)$ with an even number of vertices as the input, the HALF-IS problem is to decide if G has an independent set of size $|V|/2$. Prove that HALF-IS is in NP-Complete.

Solution:

Given a graph $G(V, E)$ and a certifier $S \subset V$, $|S| = |V|/2$, we can verify if no two nodes are adjacent in polynomial time ($O(|S|^2) = O(|V|^2)$). Therefore $\text{HALF-IS} \in NP$.

We prove the NP-Hardness using a reduction of the NP-complete problem Independent set problem (IS) to HALF-IS . Consider an instance of IS , which asks for an independent set $A \subset V$, $|A| = k$, for a graph $G(V, E)$, such that no two pair of vertices in A are adjacent to each other.

- (i) If $k = \frac{|V|}{2}$, IS reduces to HALF-IS.
- (ii) If $k < \frac{|V|}{2}$, then add m new nodes such that $k + m = (|V| + m)/2$, i.e., $m = |V| - 2k$. Note that the modified set of nodes V' has even number of nodes. Since the additional nodes are all disconnected from each other, they form a subset of independent set. Therefore, the new graph $G'(V', E')$ where $E' = E$ has an independent-set of size $\frac{|V'|}{2}$ if and only if $G(V, E)$ has an independent set of size k .
- (iii) If $k > \frac{|V|}{2}$, then again add $m = |V| - 2k$ new nodes to form the modified set of nodes V' . Connect these new nodes to all the other $|V| + m - 1$ nodes. Since these m new nodes are connected to every other node of them should belong to an independent set. . Therefore, the new graph $G'(V', E')$ has an independent-set of size $\frac{|V'|}{2}$ if and only if $G(V, E)$ has an independent set of size k .

Hence, any instance of IS $(G(V, E), k)$, can be reduced to an instance of HALF-IS $(G'(V', E'))$.

$$IS \leq_P HALF-IS.$$

$\text{HALF-IS} \in NP$ and $\text{HALF-IS} \in NP\text{-Hard} \Rightarrow \text{HALF-IS} \in NP\text{-complete}$.

3) 16 pts

A variant on the decision version of the subset sum problem is as follows: Given a set of n integer numbers $A = \{a_1, a_2, \dots, a_n\}$ and a target number t . Determine if there is a subset of numbers in A whose product is precisely t . That is, the output is *yes* or *no*. Describe an algorithm (and provide pseudo-code) to solve this problem, and analyze its complexity.

Solution:

Solution: *Use dynamic programming:*

Let $S[i,j]$ shows if there exists a subset of $\{a_1, a_2, \dots, a_i\}$ that add up to j , where $0 \leq j \leq t$.

$S[i,j]$ is true or false.

Initialize: $S[0,0] = \text{true}$; $S[0,j] = \text{false}$ if $j \neq 0$

Recurrence formula:

$S[i,j] = S[i-1,j] \text{ OR } S[i-1, j-a_i]$

Output is $S[n,t]$

Complexity is $O(tn)$

4) 16 pts

We've been put in charge of a phone hotline. We need to make sure that it's staffed by at least one volunteer at all times. Suppose we need to design a schedule that makes sure the hotline is staffed in the time interval $[0, h]$. Each volunteer i gives us an interval $[s_i, f_i]$ during which he or she is willing to work. We'd like to design an algorithm which determines the minimum number of volunteers needed to keep the hotline running. Design an efficient greedy algorithm for this problem that runs in time $O(n \log n)$ if there are n student volunteers. Prove that your algorithm is correct. You may assume that any time instance has at least one student who is willing to work for that time.

Solution:

Algorithm: Initially, select the student who can start at or before time 0 and whose finish time is the latest. For each subsequent volunteer, select the one whose start time is no later than the finish time of the last selected volunteer and whose finish time is the latest. Keep selecting volunteers sequentially in this way until the interval $[0, h]$ is covered.

The running time is $O(n \log n)$:

First, sort the start time of all the volunteers takes time $O(n \log n)$; then, searching and selecting the valid successive volunteers with the latest finish time takes $O(n)$ time in total (each volunteer is checked at most once).

Proof:

Let g_1, g_2, \dots, g_m be the sequence of volunteers we selected according to the greedy algorithm; let p_1, p_2, \dots, p_k be the sequence of selected volunteers of an optimal solution.

Clearly, for any feasible solution, we must have someone who can starts before or at time 0. So in the above two solutions: g_1 and p_1 must start at or before time 0.

According to our algorithm, we have $f_{g1} \geq f_{p1}$. By replacing p_1 by g_1 in the optimal solution, we get another solution: g_1, p_2, \dots, p_k , which uses k volunteers and cover the interval $[0, h]$ and therefore is another optimal solution.

Induction hypothesis: assume that our greedy solution is the same as an optimal solution up to the $r-1$ th selected volunteer, i.e., $g_1, \dots, g_{r-1}, p_r, \dots, p_k$ is an optimal solution. With the same argument: $f_{g_r} \geq f_{p_r}$ by replacing p_r by g_r in the optimal solution, we get another solution $g_1, \dots, g_r, p_{r+1}, \dots, p_k$, which is also optimal.

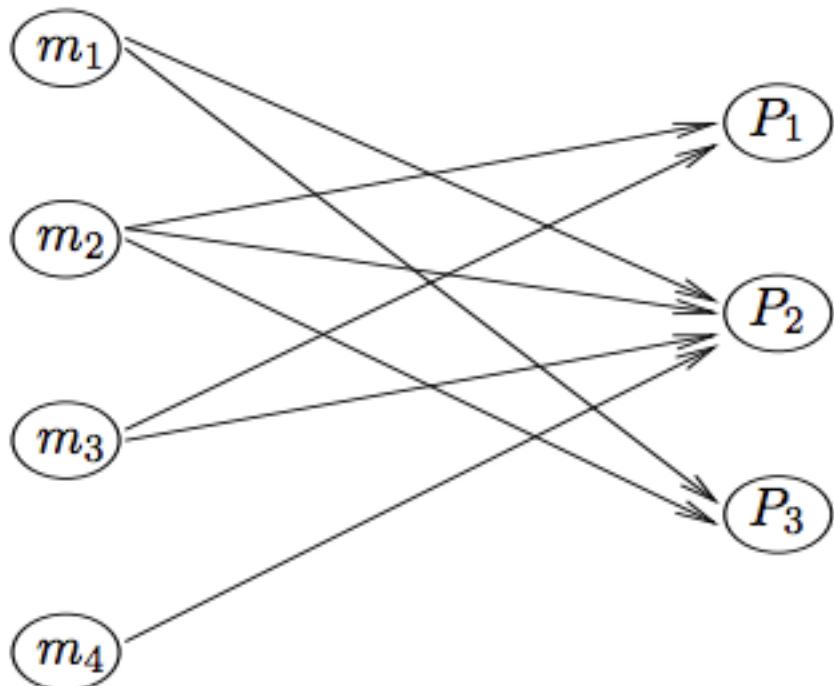
By induction: it follows that g_1, g_2, \dots, g_m is an optimal solution and $m=k$.

5) 16 pts

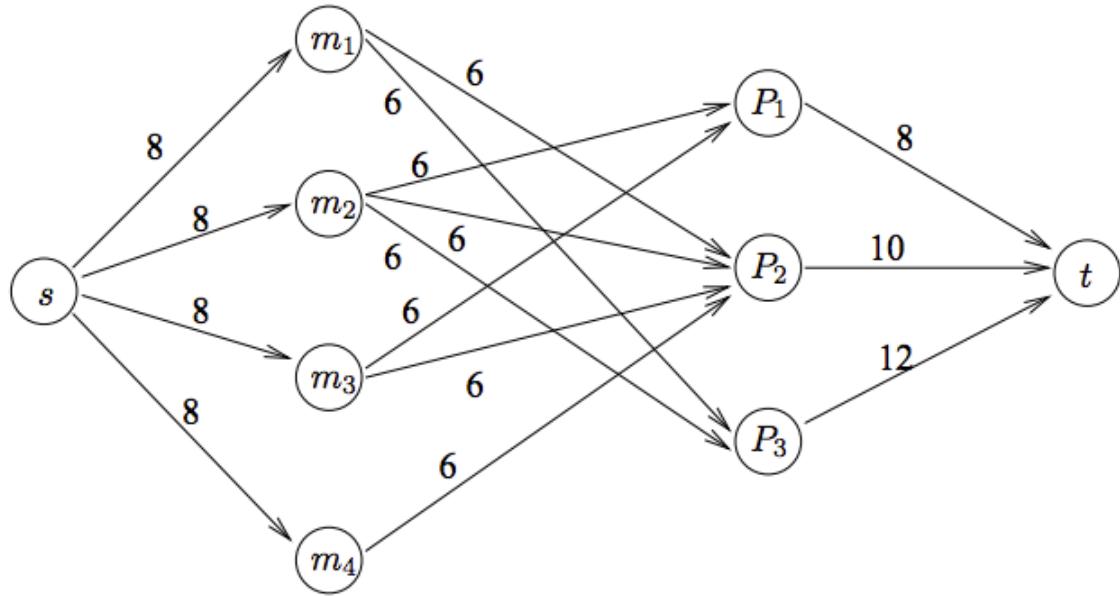
A software house has to handle 3 projects, P_1, P_2, P_3 , over the next 4 months. P_1 can only begin after month 1, and must be completed within month 3. P_2 and P_3 can begin at month 1, and must be completed, respectively, within month 4 and 2. The projects require, respectively, 8, 10, and 12 man-months. For each month, 8 engineers are available. Due to the internal structure of the company, at most 6 engineers can be working, at the same time, on the same project. Determine whether it is possible to complete the projects within the time constraints. Describe how to reduce this problem to the problem of finding a maximum flow in a flow network and justify your reduction.

Solution

We build a product network where months and projects are represented by, respectively, month-nodes m_1, m_2, m_3, m_4 and project-nodes P_1, P_2, P_3 . Each (i, j) edge denotes the possibility of allocating man-hours of month i to project j . For instance, since project P_1 can only begin after month 1 and must be completed before month 3, only arcs outgoing from m_2, m_3 are incident in P_1 .



We add to super nodes s, t , denoting the source and sink of the flow that represents the allocation of men-hours.



All edges outgoing from s have capacity 8, equivalent to the number of available engineers per month. All edges connecting month-nodes to project-nodes have capacity 6, as no more than 6 engineers can work on the same project in the same month. All edges incident in t have capacity equivalent to the number of man-months needed to complete the project. Since all capacities are integer, the maximum flow will be integer as well. To check whether all projects can be completed within the time limits, it suffices to check whether the network admits a feasible flow of value $8 + 10 + 12 = 30$.

6) 16 pts

There are n people and n jobs. You are given a cost matrix, C, where C[i][j] represents the cost of assigning person i to do job j. You want to assign all the jobs to people and also only one job to a person. You also need to minimize the total cost of your assignment. Can this problem be formulated as a linear program? If yes, give the linear programming formulation. If no, describe why it cannot be formulated as an LP and show how it can be reduced to an integer program.

Solution:

We need a 0,1 decision variable to solve the problem and therefore we need to formulate this as an integer program. Below is a formulation of integer program.

Let $x_{ij} = 1$, if job j is assigned to worker i.
 $= 0$, if job j is not assigned to worker i.

Objective function: Minimize

$$\sum_{i=1}^m \sum_{j=1}^n c_{ij} x_{ij}$$

Constraints:

$$\sum_{i=1}^m x_{ij} = 1, \text{ for } j = 1, 2, \dots, n$$

$$\sum_{j=1}^n x_{ij} = 1, \text{ for } i = 1, 2, \dots, m$$

$$x_{ij} = 0 \text{ or } 1$$

Additional Space

Additional Space

CS570
Analysis of Algorithms
Spring 2015
Exam III

Name: _____

Student ID: _____

Email Address: _____

_____ **Check if DEN Student**

	Maximum	Received
Problem 1	20	
Problem 2	16	
Problem 3	16	
Problem 4	16	
Problem 5	16	
Problem 6	16	
Total	100	

Instructions:

1. This is a 2-hr exam. Closed book and notes
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.

1) 20 pts

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

[**TRUE/FALSE**]

If SAT \leq_P A, then A is NP-hard.

[**TRUE/FALSE**]

If a problem X can be reduced to a known NP-hard problem, then X must be NP-hard.

[**TRUE/FALSE**]

If P equals NP, then NP equals NP-complete.

[**TRUE/FALSE**]

Let X be a decision problem. If we prove that X is in the class NP and give a poly-time reduction from X to Hamiltonian Cycle, we can conclude that X is NP-complete.

[**TRUE/FALSE**]

The recurrence $T(n) = 2T(n/2) + 3n$, has solution $T(n) = \theta(n \log(n^2))$.

[**TRUE/FALSE**]

On a connected, directed graph with only positive edge weights, Bellman-Ford runs asymptotically as fast as Dijkstra.

[**TRUE/FALSE**]

Linear programming is at least as hard as the Max Flow problem in a flow network.

[**TRUE/FALSE**]

If you are given a maximum s-t flow in a graph then you can find a minimum s-t cut in time $O(m)$ where m is the number of the edges in the graph.

[**TRUE/FALSE**]

Fibonacci heaps can be used to make Dijkstra's algorithm run in $O(|E| + |V| \log|V|)$ time on a graph $G=(V,E)$

[**TRUE/FALSE**]

A graph with non-unique edge weights will have at least two minimum spanning trees

2) 16 pts

Given a graph $G=(V, E)$ with an even number of vertices as the input, the HALF-IS problem is to decide if G has an independent set of size $|V|/2$. Prove that HALF-IS is in NP-Complete.

3) 16 pts

A variant on the decision version of the subset sum problem is as follows: Given a set of n integer numbers $A = \{a_1, a_2, \dots, a_n\}$ and a target number t . Determine if there is a subset of numbers in A whose product is precisely t . That is, the output is *yes* or *no*. Describe an algorithm (and provide pseudo-code) to solve this problem, and analyze its complexity.

4) 16 pts

We've been put in charge of a phone hotline. We need to make sure that it's staffed by at least one volunteer at all times. Suppose we need to design a schedule that makes sure the hotline is staffed in the time interval $[0, h]$. Each volunteer i gives us an interval $[s_i, f_i]$ during which he or she is willing to work. We'd like to design an algorithm which determines the minimum number of volunteers needed to keep the hotline running. Design an efficient greedy algorithm for this problem that runs in time $O(n \log n)$ if there are n student volunteers. Prove that your algorithm is correct.

You may assume that any time instance has at least one student who is willing to work for that time.

5) 16 pts

A software house has to handle 3 projects, P_1 , P_2 , P_3 , over the next 4 months. P_1 can only begin after month 1, and must be completed within month 3. P_2 and P_3 can begin at month 1, and must be completed, respectively, within month 4 and 2. The projects require, respectively, 8, 10, and 12 man-months. For each month, 8 engineers are available. Due to the internal structure of the company, at most 6 engineers can be working, at the same time, on the same project. Determine whether it is possible to complete the projects within the time constraints. Describe how to reduce this problem to the problem of finding a maximum flow in a flow network and justify your reduction.

6) 16 pts

There are n people and n jobs. You are given a cost matrix, C , where $C[i][j]$ represents the cost of assigning person i to do job j . You want to assign all the jobs to people and also only one job to a person. You also need to minimize the total cost of your assignment. Can this problem be formulated as a linear program? If yes, give the linear programming formulation. If no, describe why it cannot be formulated as an LP and show how it can be reduced to an integer program.

Additional Space

Additional Space

CS570 Spring 2018: Analysis of Algorithms Exam III

	Points		Points
Problem 1	20	Problem 5	15
Problem 2	15	Problem 6	10
Problem 3	15	Problem 7	10
Problem 4	15		
Total: 100 Points			

Instructions:

1. This is a 2-hr exam. Closed book and notes
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.

1) 20 pts

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

[TRUE]

To prove that a problem X is NP-hard, it is sufficient to prove that SAT is polynomial time reducible to X.

[FALSE]

If a problem Y is polynomial time reducible to X, then a problem X is polynomial time reducible to Y.

[TRUE]

Every problem in NP can be solved in polynomial time by a nondeterministic Turing machine.

[TRUE]

Suppose that a divide and conquer algorithm reduces an instance of size n into 4 instances of size $n/5$ and spends $\Theta(n)$ time in the conquer steps. The algorithm runs in $\Theta(n)$ time.

[FALSE]

A linear program with all integer coefficients and constants must have an integer optimum solution.

[FALSE]

Let M be a spanning tree of a weighted graph $G=(V, E)$. The path in M between any two vertices must be a shortest path in G.

[TRUE]

A linear program can have an infinite number of optimal solutions.

[TRUE]

Suppose that a Las Vegas algorithm has expected running time $\Theta(n)$ on inputs of size n . Then there may still be an input on which it runs in time $\Omega(n^2)$.

[FALSE]

The total amortized cost of a sequence of n operations gives a lower bound on the total actual cost of the sequence.

[FALSE]

The maximum flow problem can be efficiently solved by dynamic programming.

2) 15 pts

Consider a uniform hash function h that evenly distributes n integer keys $\{0, 1, 2, \dots, n-1\}$ among m buckets, where $m < n$. Several keys may be mapped into the same bucket. The uniform distribution formally means that $\Pr(h(x)=h(y))=1/m$ for any $x, y \in \{0, 1, 2, \dots, n-1\}$. What is the expected number of total collisions? Namely how many distinct keys x and y do we have such that $h(x) = h(y)$?

Solution:

Let the indicator variable X_{ij} be 1 if $h(k_i) = h(k_j)$ and 0 otherwise for all $i \neq j$. Then let X be a random variable representing the total number of collisions. It can be calculated as the sum of all the X_{ij} 's: $X = \sum_{i < j} X_{ij}$

Now we take the expectation and use linearity of expectation to get:

$$E[X] = E \left[\sum_{i < j} X_{ij} \right] = \sum_{i < j} E[X_{ij}] = \sum_{i < j} P(h(k_i) = h(k_j)) = \sum_{i < j} \frac{1}{m} = \frac{n(n-1)}{2m}$$

Rubrics:

Describe choosing 2 keys from n keys: 10 pts.

Describe $\sum_{i < j} P(h(k_i) = h(k_j))$: 10 pts.

Common mistakes:

1. n/m : The mistake is that it thinks the expected number of collisions for each key is $1/m$. 7pts.
2. $2n/m$: Similar to 1. 7pts.
3. $(n-1)/m$: Similar to 1. 7pts.
4. $(m+1)/2$: Basically, no clue. 3 pts.
5. $n(n-1)/m$: Duplicates. 13 pts.
6. n^2/m : duplicated. 13 pts
7. $(1-(1-1/m)^{(n-1)}) * n$: wrong approach. 5 pts.
8. $n-m$: wrong approach. 3pts.
9. $\frac{1}{m} \sum_{i=0}^{n-1} i$: 15 pts
10. $n(n-1)/2$: The mistake is that it does not consider collision probability. 10 pts.
11. $N(n+1)/2m$: minor mistake. 13 pts.

3) 15 pts.

There are 4 production plants for making cars. Each plant works a little differently in terms of labor needed, materials, and pollution produced per car:

	Labor	Materials	Pollution
Plant 1	2	3	15
Plant 2	3	4	10
Plant 3	4	5	9
Plant 4	5	6	7

The goal is to maximize the number of cars produced under the following constraints:

- There are at most 3300 hours of labor
- There are at most 4000 units of material available.
- The level of pollution should not exceed 12000 units.
- Plant 3 must produce at least 400 cars.

Formulate a linear programming problem, using minimal number of variables, to solve the above task of maximizing the number of cars.

Solution:

We need four variables, to formulate the LP problem: x_1, x_2, x_3, x_4 , where x_i denotes the number of cars at plant-i.

Maximize $x_1 + x_2 + x_3 + x_4$

s.t.

$$x_i \geq 0 \text{ for all } i$$

$$x_3 \geq 400$$

$$2x_1 + 3x_2 + 4x_3 + 5x_4 \leq 3300$$

$$3x_1 + 4x_2 + 5x_3 + 6x_4 \leq 4000$$

$$15x_1 + 10x_2 + 9x_3 + 7x_4 \leq 12000$$

Rubric:

- Identifying 4 variables (2 pts)
- Correct objective function (3 pts)
- Each condition (2 pts)

4) 15 pts.

Given an undirected connected graph $G = (V, E)$ in which a certain number of tokens $t(v) \geq 1$ placed on each vertex v . You will now play the following game. You pick a vertex u that contains at least two tokens, remove two tokens from u and add one token to any one of adjacent vertices. The objective of the game is to perform a sequence of moves such that you are left with exactly one token in the whole graph. You are not allowed to pick a vertex with 0 or 1 token. Prove that the problem of finding such a sequence of moves is NP-complete by reduction from Hamiltonian Path.

Solution:

Construction: given a HP in G , we construct G' as follows. Traverse a HP in G and placed 2 tokens on the starting vertex and one token on each other vertex in the path.

Claim: G has a HP iff G' has a winning sequence.

->) by construction before the last move we will end up with a single vertex having two tokens on it. Making the last move, we will have exactly one token on the board.

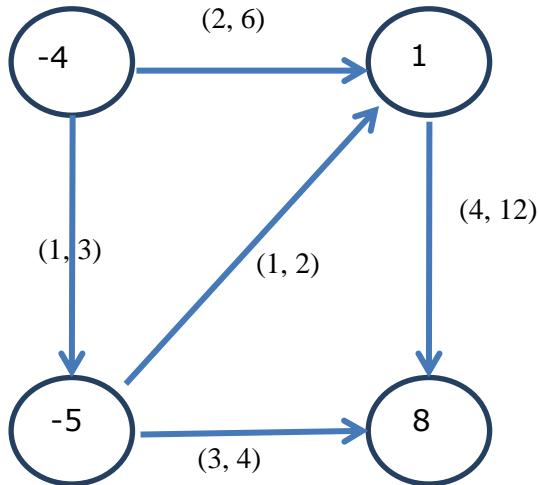
<-) since there is only one vertex with 2 tokens, we will start right there playing the game. Each next move is forced. When we finish the game, we get a sequence moves which represents a HP.

Rubrics:

- Didn't prove it is in NP: -5
- Didn't prove it is NP-hard: -10
- Assigned two tokens on one random vertex instead of the starting vertex of Hamiltonian path: -2 (Since it is a reduction from Hamiltonian path, not from Hamiltonian cycle, 2 tokens should be assigned on the starting vertex)
- Assigned wrong number of tokens on one vertex: -3
- Assigned wrong number of tokens on two vertices: -6
- Assigned wrong number of tokens on three or more vertices (considered as not a valid reduction from Hamiltonian path): -7
- Not a valid reduction from Hamiltonian path: -7

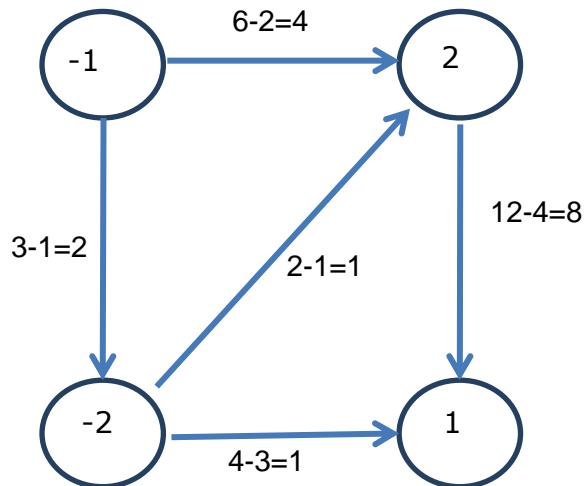
5) 15 pts.

In the network 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.



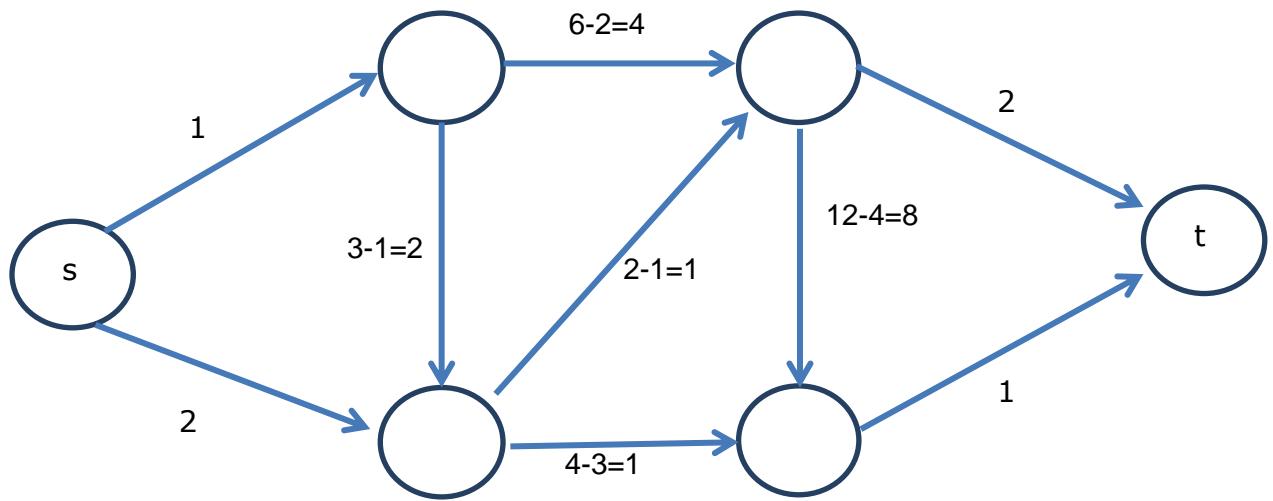
- a) Turn the circulation with lower bounds problem into a circulation problem without lower bounds (6 pts)

- **Each wrong number for nodes: -1**
- **Each wrong number for edges: -0.5**



b) Turn the circulation with demands problem into the max-flow problem (5 pts)

- **s and t nodes on right places: each has 0.5 point**
- **directions of edges from s and t: each direction 0.5 point**
- **values of edges from s and t: each value 0.5 point**

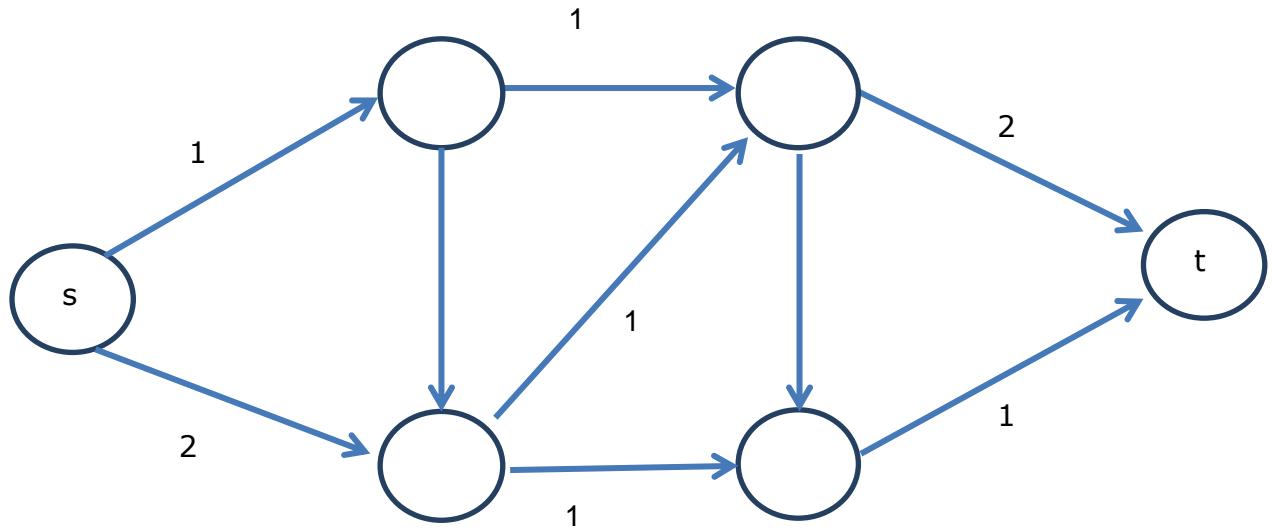


c) Does a feasible circulation exist? Explain your answer. (4 pts)

Solution.

Yes. There is a s-t flow $|f|=2+1=1+2=3$. It follows, there is a feasible circulation.

- **Explanation: 2 points.**
- **Correct Graph/flow: 2 points.**



6) 10 pts.

We wish to determine the most efficient way to deliver power to a network of cities.

Initially, we have n cities that are all connected. It costs $c_i \geq 0$ to open up a power plant at city i . It costs $r_{ij} \geq 0$ to build a cable between cities i and j . A city is said to have power if either it has a power plant, or it is connected by a series of cables to some other city with a power plant (in other words, the cables act as undirected edges). Devise an efficient algorithm for finding the *minimum cost* to power all the cities.

Solution

Consider a graph with the cities at the vertices and with weight r_{ij} on the (undirected) edge between city i and city j . Add a new vertex s and add an edge with weight c_i from s to the i th city, for each i . The minimum spanning tree on this graph gives the best solution (power stations need to be opened at those cities i for which (s,i) is part of the MST.)

Common Mistakes:

- Dynamic Programming: There does not exist a predefined order in which the nodes/edges will be processed and because of this you cannot define the problem based on smaller sub-problems and hence, none of the solutions which were based on Dynamic Programming were correct.
- Max-Flow/Min Cut: It was a surprise to see many ran a Max Flow algorithm to minimize the overall cost of edges. The minimum cut in a network has nothing to do with minimizing flow!! Furthermore, in this problem we want connectivity in the graph and edges on the min cut don't guarantee any level of connectivity.
- ILP: I can't recall anybody correctly formalizing the problem as in ILP. However, even a correct formulation is going to be NP-Hard and far from efficient. A maximum of 3 points was given to a CORRECT ILP solution.
- Dijkstra's algorithm is for finding the shortest path between a single point and every other node in the graph. The resulting Tree from running Dijkstra, is not necessarily an MST so it's not the correct approach to solve this problem.

7) 10 pts.

We want to break up the graph $G = (V, E)$ into two disjoint sets of vertices $V=A \cap B$, such that the number of edges between two partitions is as large as possible. This problem is called a max-cut problem. Consider the following algorithm:

- Start with an arbitrary cut C .
- While there exists a vertex v such that moving v from A to B increases the number of edges crossing cut C , move v to B and update C .

Prove that the algorithm is a 2-approximation.

Hint: after algorithm termination the number of edges in each of two partitions A and B cannot exceed the number of edges on the cut C .

Solution

Let $w(u,v) = 1$, there exists an edge between u and v , and 0 otherwise. Let

$$W = \sum_{(u,v) \in E} w(u,v)$$

By construction, for any node $u \in A$:

$$\sum_{v \in A} w(u, v) \leq \sum_{v \in B} w(u, v)$$

Here, the left hand side is all edges in partition A . The RHS – the crossing edges. If we sum up the above inequality for all $u \in A$, the LHS will contain the twice number of edges in A .

$$2 \sum_{(u,v) \in A} w(u, v) \leq \sum_{u \in A, v \in B} w(u, v) = C$$

We do the same for any node in B :

$$2 \sum_{(u,v) \in B} w(u, v) \leq \sum_{u \in A, v \in B} w(u, v) = C$$

Add them up

$$\sum_{(u,v) \in A} w(u, v) + \sum_{(u,v) \in B} w(u, v) \leq C$$

Next we add edges on the cut C to both sides.

$$\sum_{(u,v) \in A} w(u, v) + \sum_{(u,v) \in B} w(u, v) + \sum_{u \in A, v \in B} w(u, v) = W \leq 2C$$

Clearly the optimal solution $\text{OPT} \leq W$, thus $\text{OPT} \leq W \leq 2C$. It follows, $\text{OPT}/C \leq 2$.

A: # of edges within partition A
B: # of edges within partition B
C: # of edges between partition A and B

Correct direction

inCorrect direction

Prove that $A + B \leq C$

Prove that $A \leq C$ and $B \leq C$
Correct but useless, at most 1 points

Stage 1. 6 points

Prove that $\text{OPT} \leq 2*C$

Prove that $\text{OPT} \leq 3*C$
At most 2 points, if relations between A, B, C, E are used in reasonable way.

Stage 2. 4 points

- The solution can be divided into two parts. **Part 1** is worth 6 points, and **part 2** is worth 4 points.
 - Proving the hint, i.e. "numbers of edges in partition A and partition B cannot exceed the number of edges on the cut C". It is most important and challenging part of this problem.
 - Prove the algorithm is a 1/2-approximation.
- About part 1, here are two acceptable solutions and some solutions not acceptable:
 - If your solution is same to the official solution, using inequalities to rigorously prove the hint, it is perfect, **6 points**.
 - Here is a weaker version. A the end of algorithm, for any node u in the graph, without loss of generality assume it is in part A, the number of edges connecting it to nodes in part B (i.e. edges in the cut) is greater than or equal to the number of edges connecting it to other nodes in part A. So, the total number of edges in the cut is greater than or equal to the total number of edges within each part. This solution did not consider the important fact that both inter-partition and intra-partition edges are counted twice. **3 ~5 points according to the quality of statements.**
 - One incorrect proof: some answers declare that C is increasing and $A + B$ is decreasing when calling steps 2 of the algorithm. This is a correct but useless statement. **0 points**
 - YOU MUST PROVE THE HINT. SIMPLY STATING OR REPHRASING IT GET 0 POINTS IN THIS STEP.** Here is one example that cannot get credits: "According to the algorithm, we move one vertex from A to B if it can increase the number of edges in cut C. Then the number of edges in C is more than edges in partition A and partition B." **0 points**
 - "If move v from A to B can increase the number of edges cross cut C, it means v connect to more vertex in A than in B". It is rephrasing the algorithm, and failed to get the key point. If the above statement is changed to "If move v from A to B can increase the number of edges cross cut C, it means v connect to more vertex in different partition than in the same partition", it will get at least **3 points**.
- About part 2, the correct answer should use relation between OPT, E, A, B, C. Missing key inequalities like $\text{OPT} \leq E$, will lead to loss of grades.
- If the answer is incorrect, you may get some points from some steps that make sense. If the answer is on the incorrect direction in the above diagram:
 - Many students attempted to PROVE the $\text{edges_in_partition_A} \leq C$, and $\text{edges_in_partition_B} \leq C$, but not proving $\text{edges_in_partition_A} + \text{edges_in_partition_B} \leq C$. This is trivial according to step 2 of algorithm, and is not useful for proving the final statement. This will be treated as "correct but useless/irrelevant statements". **AT MOST 1 point.**
 - From above inequalities, $\text{OPT} \leq 3*C$ is a correct conclusion. It is different from the question, but since the logics in this stage is same, you can get at most **2 points out of 4**.

CSCI 570 - Spring 2019 - HW 11

Due: *April 21*

1. State True/False. Let A be *NP-complete*, and B be *NP-hard*. Then, $A \leq_p B$.

True. A is *NP-complete* and thus, $A \in NP$. Then, since B is *NP-hard*, we have $A \leq_p B$ by definition.

2. State True/False. Let $A \leq_p B$ and $B \leq_p A$. Then, either A and B are both *NP-complete*, or neither is.

True. Assume A is *NP-complete*. This means two things:

- (a) $A \in NP$. Then, $B \leq_p A$ implies $B \in NP$. (See Q3 from HW 10)
- (b) A is *NP-hard*. Then, $A \leq_p B$ implies B is *NP-hard*.

Hence, the two together imply B is *NP-complete*. By symmetry, one can show that, if B is *NP-complete*, so is A . QED.

3. State True/False. If $P = NP$, then every *NP-hard* problem can be solved in polynomial time.

False. If $P = NP$, only the *NP-complete* problems will be surely polytime solvable (because they are in NP), but not necessarily all the *NP-hard* problems.

4. Given an undirected graph $G = (V, E)$, a clique is a subset $A \subseteq V$ such that For every pair of vertices $u, v \in A$, if $u = v$, then $(u, v) \in E$. Given a graph and an integer m , the *CLIQUE* problem is to decide if the graph has a clique of size m . The *HALF-CLIQUE* problem is to decide if a given graph $G = (V, E)$ has a clique of size at least $\frac{|V|}{2}$.

First, show that *CLIQUE* is *NP-complete* by showing a reduction from the *INDEPENDENT-SET* problem which is known to be *NP-complete*. Further, show that *HALF-CLIQUE* is *NP-complete* by showing a reduction from *CLIQUE*.

Solution. Given a set of vertices A as the certificate, it is easy to verify (by iterating through pairs of vertices in A) that it is a clique, and that $|A| = k$ or similarly $|A| = \frac{|V|}{2}$. Hence, *CLIQUE* and *HALF-CLIQUE* are both in NP .

First, we prove that *CLIQUE* is *NP-hard*. For a graph $G = (V, E)$, its complement $\bar{G} = (V, \bar{E})$ is defined so that an edge $e \in \bar{E}$ if and only if

$e \notin E$. The key observation is that a set of vertices B is an independent set in G if and only if it is a clique for its complement \bar{G} . Thus an *INDEPENDENT-SET* instance $\langle G, k \rangle$ can be reduced to the *CLIQUE* problem by mapping it to the *CLIQUE* instance $\langle \bar{G}, m = k \rangle$, and the reduction is clearly poly-time and poly-size. Thus, *INDEPENDENT-SET* \leq_p *CLIQUE*.

Next, we show that *HALF-CLIQUE* is *NP-hard*. Given a *CLIQUE* instance $\langle G = (V, E), m \rangle$ we reduce it to a *HALF-CLIQUE* instance. If $m = \frac{|V|}{2}$, then we already have a *HALF-CLIQUE* instance.

If $m < \frac{|V|}{2}$, we add $|V| - 2m$ new vertices to G . Then, we add an edge between every distinct pair of new vertices and also an edge between every new vertex and every existing vertex as well. Call this graph $G' = (V', E')$. G' has $2m$ vertices. Now, if a set of vertices B is a clique in G , then $B \cup (V' \setminus V)$ is a clique in G' and vice versa. Hence, G has a clique of size at least m if and only if G' has a clique of size at least $m + (|V|2m) = |V|m = \frac{|V'|}{2}$.

On the other hand, if $m > \frac{|V|}{2}$, we add $2m - |V|$ new vertices to G and do not introduce any new edges. Call this graph $G' = (V', E')$. G' has $2m$ vertices. Now, G has a clique of size at least m if and only if G' had a clique of size $m = \frac{|V'|}{2}$.

It's easy to see that the reductions are poly-time computable and of poly-size. This shows *CLIQUE* \leq_p *HALF-CLIQUE*.

5. Given an undirected graph with positive edge weights, the *BIG-HAM-CYCLE* problem is to decide if it contains a Hamiltonian cycle C such that the sum of weights of edges in C is at least half of the total sum of weights of edges in the graph. Show that *BIG-HAM-CYCLE* is *NP-complete*. You are allowed to use the fact that deciding if an undirected graph has a Hamiltonian cycle is *NP-complete*.

Solution. The certifier takes as input an undirected graph (the *BIG-HAM-CYCLE* instance) and a sequence of edges (certificate). It verifies that the sequence of edges is a Hamiltonian cycle and that the total weight of the cycle is at least half the total weight of the edges in the graph. Thus *BIG-HAM-CYCLE* is in *NP-complete*.

We claim that *HAM-CYCLE* is polynomial time reducible to *BIG-HAM-CYCLE*. To see this, given an undirected graph $G = (V, E)$ (instance of *HAM-CYCLE*), fix a node u . For a neighbor v of u , set the weight of edge uv to $|E|$ and assign the rest of the edges a weight of 1. This gives a weighted graph G' as an instance of *BIG-HAM-CYCLE*. G' is a yes instance if and only if G has a Hamiltonian cycle containing the edge uv . Now, G has a hamiltonian cycle if and only if it has a hamiltonian cycle containing uv for some 2 neighbors v of the fixed node u . Hence, by repeating the above procedure for every neighbor v of the fixed vertex

u , we can decide if G has a Hamiltonian cycle, (if and only if we find a neighbor v for which the resultant graph G' is a YES instance of *BIG-HAM-CYCLE*. Since these are only $|V|$ calls to the *BIG-HAM-CYCLE* black box and the other steps are also poly-time, *BIG-HAM-CYCLE* must also be NP-hard.

CS570 Fall 2017: Analysis of Algorithms Exam III

	Points
Problem 1	20
Problem 2	15
Problem 3	10
Problem 4	15
Problem 5	15
Problem 6	10
Problem 7	15
Total	100

Instructions:

1. This is a 2-hr exam. Closed book and notes
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.

1) 20 pts

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

[**TRUE/FALSE**]

To prove that a problem X is NP-complete, it is sufficient to prove that 3SAT is polynomial time reducible to X.

[**TRUE/FALSE**]

Finding the minimum element in a binary max heap of n elements takes $O(\log n)$ time

[**TRUE/FALSE**]

We are told that in the worst case, algorithm A runs in $O(n \log n)$ and algorithm B runs in $O(n^2)$. Based on these facts, there must be some N that when $n > N$, algorithm A runs faster than algorithm B.

[**TRUE/FALSE**]

The following recurrence equation $T(n)=3T(n/3) + 0.1 n$ has the solution:

$$T(n)=\Theta(n \log(n^2)).$$

[**TRUE/FALSE**]

Every problem in NP can be solved in exponential time by a deterministic Turing machine

[**TRUE/FALSE**]

In Kruskal's MST algorithm, if we choose edges in decreasing (instead of increasing) order of cost, we will end up with a spanning tree of maximum total cost

[**TRUE/FALSE**]

If all edges in a graph have capacity 1, then Ford-Fulkerson runs in linear time.

[**TRUE/FALSE**]

If problem X can be solved using dynamic programming, then X belongs to P.

[**TRUE/FALSE**]

If Vertex-Cover $\in P$ then SAT $\in P$.

[**TRUE/FALSE**]

Assuming $P \neq NP$, and X is a problem belonging to class NP. There is no polynomial time algorithm for X.

2) 15 pts

We have a directed graph $G = (V, E)$ where each edge (u, v) has a length $l(u, v)$ that is a positive integer. Let n denote the number of vertices in G . Suppose we have previously computed a $n \times n$ matrix $d[\cdot, \cdot]$, where for each pair of vertices $u, v \in V$, $d[u, v]$ stores the length of the shortest path from u to v in G . The $d[\cdot, \cdot]$ matrix is provided for you.

Now we add a single edge (a, b) to get the graph $G' = (V, E')$, where $E' = E \cup \{(a, b)\}$. Let $l(a, b)$ denote the length of the new edge. Your job is to compute a new distance matrix $d'[\cdot, \cdot]$, which should be filled in so that $d'[u, v]$ holds the length of the shortest path from u to v in G' , for each $u, v \in V$.

(a) Write a concise and efficient algorithm to fill in the $d'[\cdot, \cdot]$ matrix. You should not need more than about 3 lines of pseudocode. (12 pts)

Solution:

For each pair of vertices $u, v \in V$:

Set $d'[u, v] = \min(d[u, v], d[u, a] + l(a, b) + d[b, v])$.

Grading:

- Considering all pairs of vertices (6 points)
- Comparing the current shortest path with the length of the path going through edge (a, b) and choosing the smaller path (6 points)

(b) What is the asymptotic worst-case running time of your algorithm? Express your answer in $O(\cdot)$ notation, as a function of n and $|E|$. (3 pts)

Solution:

$O(n^2)$.

Grading:

- Correct Answer (3 points)
- Incorrect answer (0 points)

3) 10 pts.

Recall the 0/1 knapsack problem:

Input: n items, where item i has profit p_i and weight w_i , and knapsack size is W .

Output: A subset of the items that maximizes profit such that the total weight of the set $\leq W$.

You may also assume that all $p_i \geq 0$, and $0 < w_i \leq W$.

We have created the following greedy approximation algorithm for 0/1 knapsack:

Sort items according to decreasing p_i/w_i (breaking ties arbitrarily).

While we can add more items to the knapsack, pick items in this order.

Show that this approximation algorithm has no nonzero constant approximation ratio. In other words, if the value of the optimal solution is P^* , prove that there is no constant ρ ($1 > \rho > 0$), where we can guarantee our greedy algorithm to achieve an approximate solution with total profit ρP^* .

Solution:

The definition of a constant factor is that $P \geq \rho P^*$. Giving an example where $P = P^*$ and saying $\rho = 1$ or giving two different examples and showing the ratio P/P^* is not always the same does not mean there does not exist a constant approximation ratio.

The correct way to show this is not possible is to provide an example with multiple (at least two) items and their respective weights and profits. Then you need to show that the greedy algorithm and the optimal algorithm will generate two different solutions (/profits) and need to show in your example the ratio P/P^* can be unboundedly small.

One example can be as follows:

Consider an item with size 1 and profit 2, and another with size W and profit W . Our greedy algorithm above will take the first item, while the optimal solution is taking the second item, so this algorithm has approximation ratio $2/W$ which can go arbitrarily close to zero.

Grading:

A correct solution has 10 points.

Common Mistakes:

- Giving two different examples and showing that the ratio of greedy to optimal is different doesn't mean there is no constant approximation ratio.
- The existence of a constant approximation factor will not imply $P = NP$.
- Greedy is not optimal!!

4) 15 pts.

The graph five-coloring problem is stated as follows: Determine if the vertices of G can be colored using 5 colors such that no two adjacent vertices share the same color.

Prove that the five-coloring problem is NP-complete.

Hint: You can assume that graph 3-coloring is NP-complete

Solution:

Show that 5-coloring is in NP:

Certificate: a color solution for the network, i.e., each node has a color.

Certifier:

- i. Check for each edge (u,v) , the color of node u is different from the color of node v ;
- ii. Check at most 5 colors are used

This process takes $O(m+n)$ time.

Show that 5-coloring is NP-hard: prove that 3-coloring \leq_p 5-coloring

Graph construction:

Given an arbitrary graph G . Construct G' by adding 2 new nodes u and v to G . Connect u and v to all nodes that existed in G , and to each other.

G' can be colored with 5 colors iff G can be colored with 3 colors.

- i. If there is valid 3-color solution for G , say using colors $\{1,2,3\}$, we want to show there is a valid 5-coloring solution to G' . We can color G' using five colors by assigning colors to G according to the 3-color solution, and then color node u and v by additional two different colors. In this case, node u and v have different colors from all the other nodes in G' , and together with the 3-coloring solution in G , we use at most 5 colors to color G' .
- ii. If there is a valid 5-coloring solution for G' , we want to show there is a valid 3-coloring solution in G . In G' , since node u and v connect to all the other nodes in G and to each other, the 5-coloring solution must assign two different colors to node u and v , say colors 4 and 5. Then the remaining three colors $\{1,2,3\}$ are used to color the remaining graph G and form a valid 3-color solution.

Solving 5-coloring to G' is as hard as solving 3-color to G , then 5-coloring problem is at least as hard as 3-coloring, i.e., 3-coloring \leq_p 5-coloring.

Grading Rubrics:

1. Prove the problem is NP (3 points in total)
 - i) Mentioning the necessity of proving NP (1 point)
 - ii) Checking two connecting nodes (u,v) having different colors (1 point)
 - iii) Checking the total number of colors used is at most (1 point)
2. Prove the problem is NP-hard (12 points in total)
 - i) Graph construction (4 points in total)
 - a. Adding two additional nodes (1 point)
 - b. Connecting to all the nodes in G (2 point)
 - c. Connecting node u and v (1 point)
 - ii) Proof (8 points in total)
 - a. $3\text{color} \rightarrow 5\text{ color}$ (2 points). This is trivial even under other constructions (even incorrect constructions)
 - b. $5\text{color} \rightarrow 3\text{ color}$ (6 points).
 - If your graph construction is incorrect, you will get 0 point in this part.
 - Under the graph construction posted in the standard solution:
 - ✓ Explaining the two additional nodes u and v must have colors different from those used in G (3 points)
 - ✓ Explaining the two additional nodes u and v must have two different colors in between (3 points)

Popular Mistakes:

1. Try to show, if G is 3-colorable, then G is 5-colorable
 - Loss 4 points for construction (actually no construction is given);
 - Lose 6 points for proof (proof for $5\text{ color} \rightarrow 3\text{ color}$ cannot be correct)
2. Without adding additional nodes to G , but try to “manipulate colors” during construction
 - Loss 4 points for construction (actually no construction is given);
 - Lose 6 points for proof (proof for $5\text{ color} \rightarrow 3\text{ color}$ cannot be correct)
3. In the graph construction, someone added additional nodes to G , but assigned additional colors to during the construction, and then in the proof, claim them “using two different colors”.
 - Loss 3 points for construction (wrong construction);
 - Loss 6 points for proof
4. In the graph construction, for each node i in G , someone added two additional nodes u_i and v_i while connecting them to node i , and then connect or not connect u_i and v_i .
 - Loss 2 points for construction (at least they know connecting with somewhere in G);
 - Loss 6 points in proof
5. In the graph construction, someone added two nodes u and v to G and connect them to all the nodes in G , but did not connect them to each other
 - Loss 1 point for construction

- If they can correctly explain that, to prove G has 3-color solution given G' having 5-color solution, their construction can guarantee node u and v take different colors from all the nodes in G , they will only lose 3 points; otherwise, they will lost 6 points.
- 6. In the graph construction, someone added two nodes “into” each edge
 - Loss 2 points for construction
 - Loss 6 points in proof

5) 15 pts.

The price of the recently introduced cryptocurrency, Crypcoint, has been changing rapidly over the last two months. Given the hourly price chart of this currency for the last two months, you are tasked to find out the maximum profit one could have made by buying 100 Crypcoints in one transaction and subsequently selling 100 Crypcoints in one transaction within this period.

Specifically, given the price chart of Crypcoint over this period $P[i]$ for $i = 1$ to n , we're looking for the maximum value of $(P[k] - P[j]) * 100$ for some indices j, k , where $1 \leq j < k \leq n$.

Examples: If the array is [2, 3, 10, 6, 4, 8, 1] then the returned value should be 8 * 100 (Diff between 2 and 10, times 100 Crypcoints). If the array is [7, 9, 5, 6, 3, 2] then the returned value should be 2 * 100 (Diff between 7 and 9, times 100 Crypcoints)

Describe a divide and conquer algorithm to solve the problem in $O(n \log n)$ time. You do not need to prove that your algorithm is correct.

Solution:

CrypMAX($P[1..n]$):

If $n \leq 1$, return 0.

Set $\max_L := \text{CrypMAX}(P[1, \dots, \lfloor n/2 \rfloor])$

Set $\max_R := \text{CrypMAX}(P[\lceil n/2 + 1 \rceil, \dots, n])$

Set $x := \min(P[1, \dots, \lfloor n/2 \rfloor])$.

Set $y := \max(P[\lceil n/2 + 1 \rceil, \dots, n])$.

Return $\max(\max_L, \max_R, (y-x)*100)$.

Complexity Analysis: There are two recursive calls on a subarray of half the size, plus a linear ($\Theta(n)$) scan to find the smallest/largest of either half. Thus, the recurrence relation is $T(n) = 2T(n/2) + \Theta(n)$, which solves to $\Theta(n \log n)$.

Alternate solution:

We can modify the algorithm to also return the smallest and largest elements of the array (i.e. return the triple (best price, min of array, max of array)). This allows the algorithm to avoid the computation of finding the smallest/largest elements of each half of the array, as the recursive calls have already calculated them. (As an additional detail, the base case also needed a minor tweak.) Our runtime would thus be $T(n) = 2T(n/2) + \Theta(1)$, which solves to $\Theta(n)$.

Grading rubric:

1. Basic structure for an efficient divide-and-conquer algorithm (5 pts)
 - a. No ambiguity, e.g., “divide it into several subproblems” or “divide it into two subproblems” will not receive the 5 pts. “... two halves” or “... two subproblems with equal size” are acceptable.
2. Correctly conquer the problem by finding the cross-segment term: $\max(P(mid:end)) - \min(P(1:mid))$. (5 pts)
 - a. The max/min operation are required to be written explicitly, to achieve an $O(n)$ complexity for each step. Finding the maximal different between two halves in brute force requires $O(n^2)$.
3. Complete the entire algorithm (5 pts). Examples:
 - a. -1 pts for missing base case;
 - b. -1 pts for multiplying by 100 more than once inappropriately.
4. Non-divide-and-conquer algorithms:
 - a. Correct $O(n \log(n))$ algorithm receive 10 pts.
 - b. Correct $O(n^2)$ algorithm receive 5 pts.
 - c. Other receive 0 pt.

Note and common mistakes:

1. You cannot buy the coin after your sell. (sell must occur no earlier than buy).
 - a. You cannot simply pick the highest price to sell and the lowest price to buy.
2. If you buy on i -th day, you don't have to sell it on the $i+1$ -th day.
3. You cannot do binary search on unsorted array.
4. [Most common mistake] The buy and sell prices does not have to be the lowest or highest price. For example, [10,4,5,1]. Therefore, returning the best buy/sell dates from the subproblems does not help you with merging the two subproblems, which requires the highest/lowest price from them.

Partially correct example solutions and grades:

1. Brute force search, $O(n^2)$ (5pts)

```
Ret = 0;  
For i in 1 to n:  
    For j in i+1 to n:  
        Ret = max(Ret, P(j)-P(i));
```

Return $100 * Ret$;
Note: the recursive version: $F(i,j) \rightarrow f(i+1,j)$ and $f(i, j-1)$ and $P(j)-P(i)$ also receive 5 pts.

2. Non-divide-and-conquer $O(n)$ (10 pts)

Segment P into fewest monotonically increasing intervals. Calculate the max-difference within these intervals and return the largest difference * 100.

3. Non-divide-and-conquer One-pass $O(n)$ (10 pts)

```
Min = P(1), Ret = 0;
```

For i in 1 to n:
 Ret = max(Ret, P(i)-Min)
 Min = min(Min, P(i))

Return Ret*100

4. Divide-and-conquer O(n^2) (5 pts)

Def Solve(P[1:m]):
 k = argmax(P)
 Return max(P(k)-min(P[1:k]), Solve(P[k+1:m]))

Final solution is 100*Solve(P); Note argmax is O(n) operation on unsorted array.

6) 10 pts.

Recall the Circulation with Lower Bounds problem:

- Each directed edge e has a lower bound l_e and an upper bound c_e on flow
- Node v has demand value d_v , where d_v could be positive, negative, or zero
- Objective is to find a feasible circulation that meets capacity constraints over each edge and satisfies demand conditions for all nodes

We now add the following constraints and objective to this problem:

- For each node v with demand value $d_v = 0$, the flow value through that node should be greater than l_v and less than c_v
- Since we suspect there may be some issues at a specific node x in the network, we want to minimize the flow going through this node as much as possible.

Give a linear programming formulation for this problem.

Solution:

Objective: minimize $\sum_{\text{all } e \text{ into } x} f_e$ (4 points)

Constraints:

1. $l_e \leq f_e \leq c_e \text{ for each edge } e$ (2points)
2. $d_{v_i} = \sum_{\text{all } e \text{ into } v_i} f_e - \sum_{\text{all } e \text{ out of } v_i} f_e \text{ for each vertex } v_i$ (2 points)
3. $l_{v_j} \leq \sum_{\text{all } e \text{ into } v_j} f_e \leq c_{v_j} \text{ for each node } v_j \text{ such that } d_{v_j} = 0$ (2 points)

- 1 point may be deducted from each of your constraints where you have not specified what is the flow
- Up to 3 points may be deducted from your objective if you have not specified what is the flow in your solution. (not defining f for instance)

7) 15 pts

Woody will cut a given log of wood, at any place you choose, for a price equal to the length of the given log. Suppose you have a log of length L , marked to be cut in n different locations labeled $1, 2, \dots, n$. For simplicity, let indices 0 and $n + 1$ denote the left and right endpoints of the original log of length L . Let d_i denote the distance of mark i from the left end of the log, and assume that

$0 = d_0 < d_1 < d_2 < \dots < d_n < d_{n+1} = L$. The wood-cutting problem is the problem of determining the sequence of cuts to the log that will cut the log at all the marked places and minimize Woody's total payment. Remember that for a single cut on a given log, Woody gets paid an amount equal to the length of that log. Give a dynamic programming algorithm to solve this problem.

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

Let $c(i, j)$ be the min cost of cutting a log with left endpoint i and right endpoint j at all its marked locations.

Grading:

Incomplete definition: -2 pts

No definition: -4pts

- b) Write the recurrence relation for subproblems. (7 pts)

(Dynamic programming)

$$c(i, j) = \min_{i < k < j} \{c(i, k) + c(k, j) + d_j - d_i\}$$

Base case: if $j = i+1$, $c(i, j) = 0$

Grading:

Error in recursion: -2pts, if multiple errors, deduction adds up.

Wrong recurrence relation: -7pts

Base case is missing (case of $j = i+1$): -2pts

Wrong base case: -2pts

- c) Compute the runtime of the algorithm in terms of n . (4 pts)

$$O(n^3)$$

Grading:

Used big Theta notation instead of big O notation: -2pts

Missing big O notation: -2pts

Wrong runtime complexity: -4pts

Discussion 11

1. In the *Min-Cost Fast Path* problem, we are given a directed graph $G=(V,E)$ along with positive integer times t_e and positive costs c_e on each edge. The goal is to determine if there is a path P from s to t such that the total time on the path is at most T and the total cost is at most C (both T and C are parameters to the problem). Prove that this problem is **NP**-complete.

Solution:

- 1- Prove that Min-Cost Fast Path is in NP

Certificate: an s - t path with total cost $\leq C$ and total time $\leq T$

Certifier: Can easily check in polynomial time that

- a- Set of edges given are in fact a path from s - t
- b- Total time is $\leq T$ and total cost is $\leq C$

a and b can be easily done in polynomial time. \rightarrow Min-Cost Fast Path \in NP

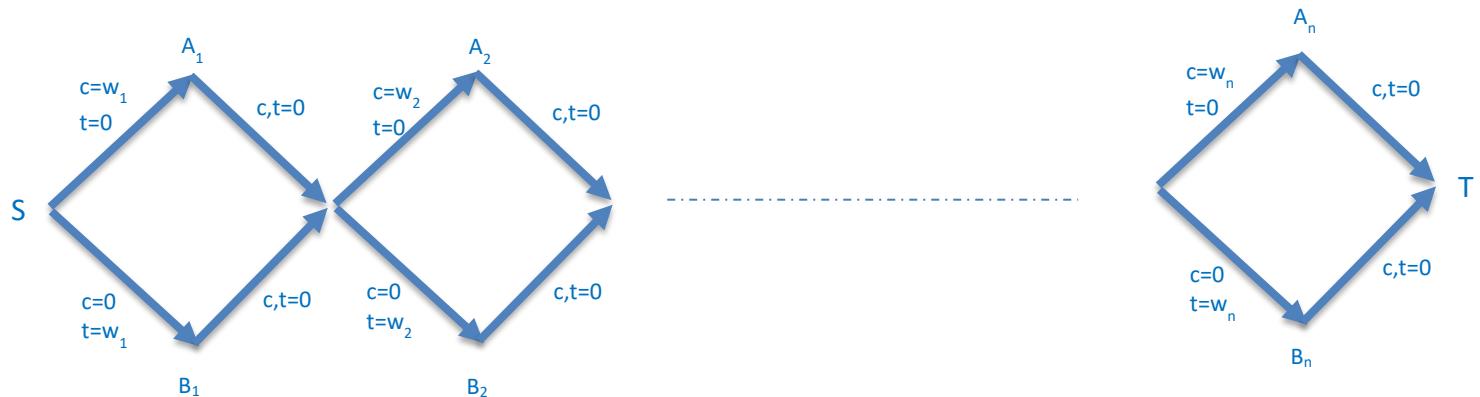
- 2- Choose Subset Sum for our reduction

- 3- Will show that Subset Sum \leq_p Min-Cost Fast Path

Background: Decision version of the Subset Sum problem asks whether given n items where item i has weight w_i , if there is a subset of them whose total weight is less than W and greater than M .

Plan: We build a graph G such that it has an s - t path with total cost $\leq W$ and total time $\leq \sum w_i - M$ iff there is a subset of items whose total weight is between M and W .

We use gadgets to represent each item. Each gadget will offer two paths through the gadget. If the s - t path in G goes through the A node of the gadget we can interpret that as item being selected as part of the set. If the s - t path goes through the B node of the gadget we interpret that as the item not being selected as part of the set. We string up the gadgets and set time t_e and costs c_e to edges as follows:



Proof:

- A- If we are given a set of items with total weight between M and W, we can find a path from S to T with total cost of at most W and total time of at most $\sum w_i - M$ by choosing the path through each gadget based on whether the item is part of the set (Go through the A node) or not (go through the B node). If we go through the A node for object i, the object contributes w_i to the cost of the path and if we go through the B node, the object contributes w_i to the total time for the path. So the total cost for the path will be the total weight of the objects selected which we know is $\leq W$ and the total time of the path is total weight of the objects that are not selected which we know is $\sum w_i - M$ (because we know the total weight of the objects selected is $\geq M$)
- B- If we are given a path from S to T which has a total cost of at most W and a total time of at most $\sum w_i - M$, we can find a set of objects with total weight between M and W. The S-T path can easily select the objects that belong to the set. If the path goes through the A node for an object, we place that object in the set, otherwise not. Since the total cost of the path is at most W then the total weight of the objects selected will be at most W and since the total time for the path is at most $\sum w_i - M$, the total weight of the objects not selected will be $\sum w_i - M$, which means that the total weight of objects selected will be at least M.

2. We saw in lecture that finding a Hamiltonian Cycle in a graph is **NP**-complete. Show that finding a Hamiltonian Path -- a path that visits each vertex exactly once, and isn't required to return to its starting point -- is also **NP**-complete.

Solution:

- 4- Prove that Hamiltonian Path is in NP

Certificate: an ordering of all nodes in G that forms a Hamiltonian Path

Certifier: Can easily check in polynomial time that

- a- There is an edge between each pair of adjacent vertices in the given order
 - b- All nodes in G are visited by the path
- a and b can be easily done in polynomial time. \rightarrow Hamiltonian Path \in NP

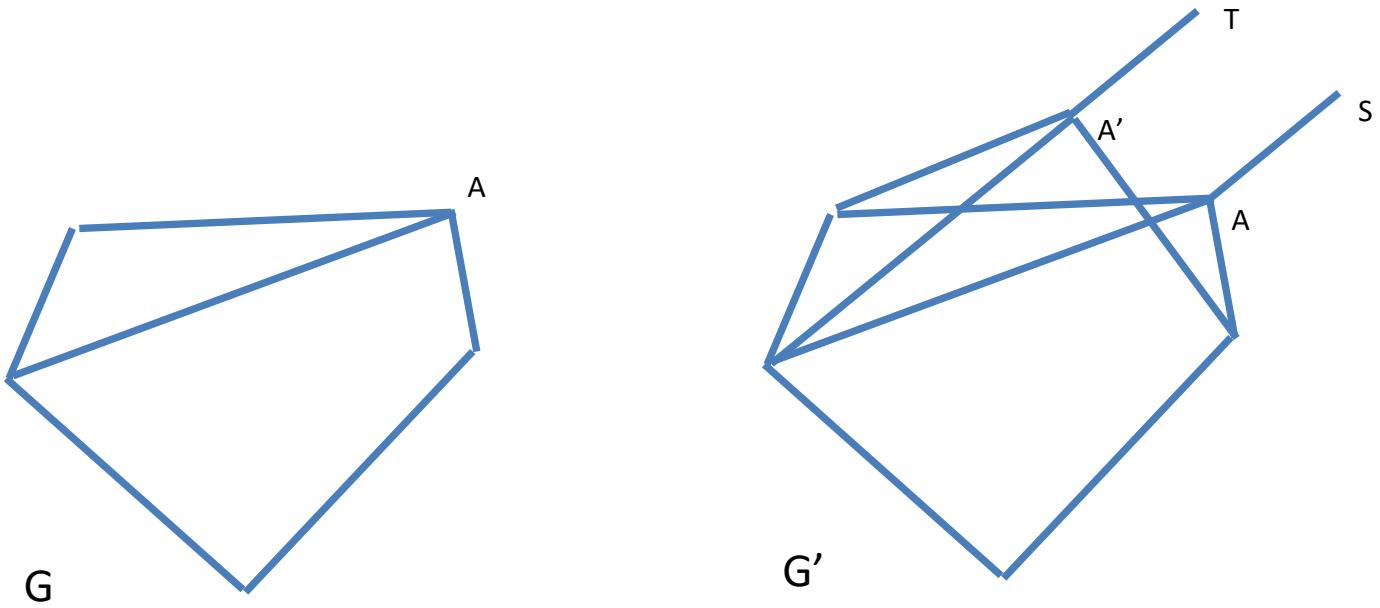
- 5- Choose Hamiltonian Cycle for our reduction

- 6- Will show that Hamiltonian Cycle \leq_p Hamiltonian Path

Plan: Given graph G—an instance of the Hamiltonian Cycle problem, we will construct G' such that G' has a Hamiltonian Path iff G has a Hamiltonian Cycle.

Construction of G'. We will split one of the nodes in G, say node A. Nodes A and A' will have the same connections as the original node A in G. We will then add two new nodes S and T and connect one with A and the other with A'.

Now G' has a Hamiltonian Path from S to T iff there is a Hamiltonian Cycle in G.



Proof:

- A- If we are given a Hamiltonian Path in G' , since S and T have a degree of 1, and can only be the beginning or the end of the path, the path must go from S to T . Ignoring the two new edges SA and TA' , this path will give us a Hamiltonian Cycle in G since A and A' are the same node in G , i.e. the path will start and end at the same node (A).
- B- If we are given a Hamiltonian Cycle in G , we can create a Hamiltonian Path in G' by splitting the Cycle at node A and creating a path from A' to A . We can then form a Hamiltonian Path in G' by starting at S going to A , following the Hamiltonian Cycle to A' and end the Path at T .

3. Some **NP**-complete problems are polynomial-time solvable on special types of graphs, such as bipartite graphs. Others are still **NP**-complete.

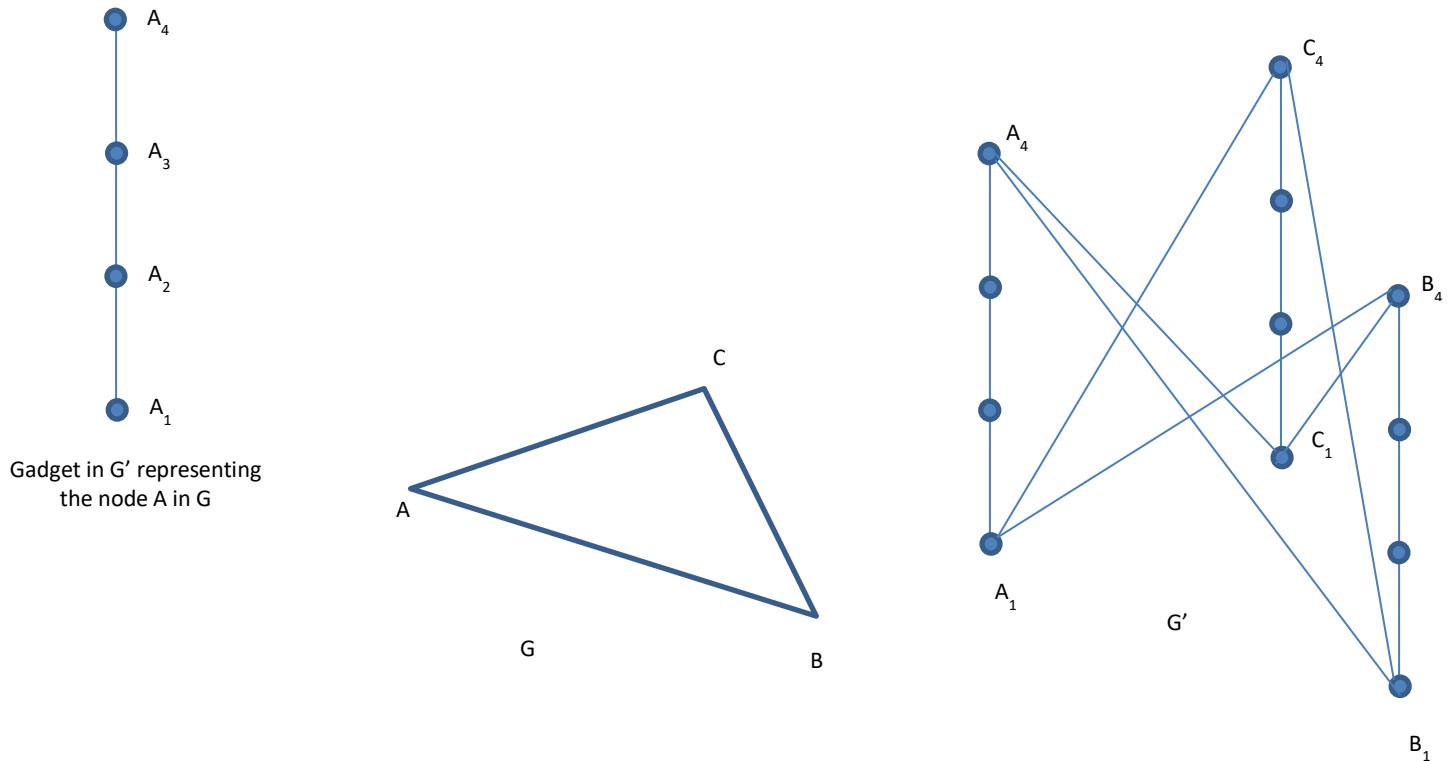
Show that the problem of finding a Hamiltonian Cycle in a bipartite graph is still **NP**-complete.

Solution:

- 7- Prove that Hamiltonian Cycle in a bipartite graph is in NP
 - Certificate: an ordering of all nodes in G that forms a Hamiltonian Cycle
 - Certifier: Can easily check in polynomial time that
 - a- There is an edge between each pair of adjacent vertices in the given order
 - b- All nodes in G are visited by the path
 - c- There is an edge between the last node in the order and the first node
 - a, b and c can be easily done in polynomial time. \rightarrow Hamiltonian Cycle in a bipartite graph \in NP
- 8- Choose Hamiltonian Cycle for our reduction
- 9- Will show that Hamiltonian Cycle \leq_p Hamiltonian Cycle in a bipartite graph

Plan: Given graph G —an instance of the Hamiltonian Cycle problem, we will construct G' such that G' is bipartite and has a Hamiltonian Cycle iff G has a Hamiltonian Cycle.

Construction of G' : For each node A in G we will use a gadget with four nodes as shown below. If A and B are connected in G , we connect nodes A_1 and B_4 and nodes B_1 and A_4 in G' .



G' is bipartite since we place all nodes V_1 and V_3 into the set X and nodes V_2 and V_4 into the set Y , all edges in G' go between sets X and Y . And G' has a Hamiltonian Cycle iff G has a Hamiltonian Cycle.

Proof:

A- If we are given a Hamiltonian Cycle in G' it must be of the form $V_1 V_2 V_3 V_4 U_1 U_2 U_3 U_4 \dots V_1$ since there is no other way for the Hamiltonian Cycle to go through the nodes of each gadget. We can then use the same sequence of nodes V, U, \dots, V to form a Hamiltonian Cycle in G , since if there is a connection between V_4 and U_1 in G' , there must be an edge between V and U in G .

B- If we are given a Hamiltonian Cycle in G that goes through nodes V, U, \dots, V we can form a Hamiltonian Cycle in G' by going through the gadgets corresponding to nodes V, U, \dots, V i.e. nodes $V_1 V_2 V_3 V_4 U_1 U_2 U_3 U_4 \dots V_1$ since if there is a connection between nodes V and U , there must be a connection between nodes V_4 and U_1 in G' .

Discussion 12

1. The *bin packing* problem is as follows. You have an infinite supply of bins, each of which can hold M maximum weight. You also have n objects, each of which has a (possibly distinct) weight w_i (any given w_i is at most M). Our goal is to partition the objects into bins, such that no bin holds more than M total weight, and that we use as few bins as possible. This problem in general is **NP-hard**.

Give a 2-approximation to the *bin packing* problem. That is, give an algorithm that will compute a valid partitioning of objects into bins, such that no bin holds more than M weight, and our algorithm uses at most twice as many bins as the optimal solution. Prove that the approximation ratio of your algorithm is two.

Solution:

Place objects in the bin in the order given starting with bin #1. When there is no more room in bin #1 start placing objects in bin #2, and so on.

Claim: This will give us a 2-approximation

Proof: the total weight of objects in bins 1 and 2 in our solution is greater than M , since the reason why we started placing objects in bin #2 was that we ran out of space in bin #1. So, since the total weight is greater than M , the optimal solution needs at least 1 bin to hold these objects (i.e. 1 bin is a bound on how good the optimal solution can be).

This can be said about every pair of bins in our solution. So if our solution ends up with $2k$ bins, the optimal solution will need at least k bins. Therefore we have 2-approximation.

2. Suppose you are given a set of positive integers $A: a_1 a_2 \dots a_n$ and a positive integer B . A subset $S \subseteq A$ is called *feasible* if the sum of the numbers in S does not exceed B .

The sum of the numbers in S will be called the *total sum* of S . You would like to select a feasible subset S of A whose total sum is as large as possible.

Example: If $A = \{8, 2, 4\}$ and $B = 11$ then the optimal solution is the subset $S = \{8, 2\}$.

Give a linear-time algorithm for this problem that finds a feasible set $S \subseteq A$ whose total sum is at least half as large as the maximum total sum of any feasible set $S' \subseteq A$. Prove that your algorithm achieves this guarantee.

You may assume that each $a_i \leq B$.

Solution:

Starting from a_1 , place integers into the set S for as long as their total does not exceed B until we reach $B/2$. If at any time the total weight of the set goes above $B/2$ we have achieved a .5 approximation (since the optimal solution cannot have a value greater than B). But if at some point in this process, the total has not reached $B/2$ and the next item in the set (a_i) causes the total to exceed B , then we will remove all elements in the set and replace it with a_i . a_i itself will give us a .5 approximation since it must be greater than $B/2$.

Above process can of course be carried out in linear time.

3. A cargo plane can carry a maximum weight of 100 tons and a maximum volume of 60 cubic meters. There are three materials to be transported, and the cargo company may choose to carry any amount of each, up to the maximum available limits given below.

- Material 1 has density 2 tons/cubic meter, maximum available amount 40 cubic meters, and revenue \$1,000 per cubic meter.
- Material 2 has density 1 ton/cubic meter, maximum available amount 30 cubic meters, and revenue \$1,200 per cubic meter.
- Material 3 has density 3 tons/cubic meter, maximum available amount 20 cubic meters, and revenue \$12,000 per cubic meter.

Write a linear program that optimizes revenue within the constraints. You do not need to solve the linear program.

Let M_1 , M_2 , and M_3 denote the cubic meters of the three materials we're going to transport.

We want to maximize the profit. So the objective function will be:

$$\text{Maximize } 1000 * M_1 + 1200 * M_2 + 12000 * M_3$$

Subject to these constraints:

$M_1 + M_2 + M_3 \leq 60$	// 60 cubic meters space available
$2 * M_1 + M_2 + 3 * M_3 \leq 100$	// 100 tons weight capacity
$0 \leq M_1 \leq 40$	
$0 \leq M_2 \leq 30$	// maximum available for these three
$0 \leq M_3 \leq 20$	

4. Recall the 0/1 knapsack problem:

Input: n items, where item i has profit p_i and weight w_i , and knapsack size is W .

Output: A subset of the items that maximizes profit such that the total weight of the set $\leq W$.

You may also assume that all $p_i \geq 0$, and $0 < w_i \leq W$.

We have created the following greedy approximation algorithm for 0/1 knapsack:

Sort items according to decreasing p_i/w_i (breaking ties arbitrarily).
While we can add more items to the knapsack, pick items in this order.

Show that this approximation algorithm has no nonzero constant approximation ratio.

In other words, if the value of the optimal solution is P^* , prove that there is no constant ρ ($1 > \rho > 0$), where we can guarantee our greedy algorithm to achieve an approximate solution with total profit ρP^* .

Solution:

Consider an item with size 1 and profit 2, and another with size W and profit W . Our greedy algorithm above will take the first item, while the optimal solution is taking the second item (for $W>1$). So this algorithm has approximation ratio $2/W$ which can go arbitrarily close to zero as we increase W .

CS570
Analysis of Algorithms
Fall 2014
Exam III

Name: _____
Student ID: _____
Email: _____

Wednesday Evening Section

	Maximum	Received
Problem 1	20	
Problem 2	16	
Problem 3	16	
Problem 4	16	
Problem 5	16	
Problem 6	16	
Total	100	

Instructions:

1. This is a 2-hr exam. Closed book and notes
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.

1) 20 pts

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

[TRUE/FALSE]

All the NP-hard problems are in NP.

[TRUE/FALSE]

Given a weighted graph and two nodes, it is possible to list all shortest paths between these two nodes in polynomial time.

[TRUE/FALSE]

In the memory efficient implementation of Bellman-Ford, the number of iterations it takes to converge can vary depending on the order of nodes updated within an iteration

[TRUE/FALSE]

There is a feasible circulation with demands $\{d_v\}$ if $\sum_v d_v = 0$.

[TRUE/FALSE]

Not every decision problem in P has a polynomial time certifier.

[TRUE/FALSE]

If a problem can be reduced to linear programming in polynomial time then that problem is in P.

[TRUE/FALSE]

If we can prove that $P \neq NP$, then a problem $A \in P$ does not belong to NP.

[TRUE/FALSE]

If all capacities in a flow network are integers, then every maximum flow in the network is such that flow value on each edge is an integer.

[TRUE/FALSE]

In a dynamic programming formulation, the sub-problems must be mutually independent.

[TRUE/FALSE]

In the final residual graph constructed during the execution of the Ford–Fulkerson Algorithm, there's no path from sink to source.

2) 16 pts

In the Bipartite Directed Hamiltonian Cycle problem, we are given a bipartite directed graph $G = (V; E)$ and asked whether there is a simple cycle which visits every node exactly once. Note that this problem might potentially be easier than Directed Hamiltonian Cycle because it assumes a bipartite graph. Prove that Bipartite Directed Hamiltonian Cycle is in fact still NP-Complete.

3) 16 pts

A tourism company is providing boat tours on a river with n consecutive segments. According to previous experience, the profit they can make by providing boat tours on segment i is known as a_i . Here a_i could be positive (they earn money), negative (they lose money), or zero. Because of the administration convenience, the local community of the river requires that the tourism company should do their boat tour business on a contiguous sequence of the river segments, i.e, if the company chooses segment i as the starting segment and segment j as the ending segment, all the segments in between should also be covered by the tour service, no matter whether the company will earn or lose money. The company's goal is to determine the starting segment and ending segment of boat tours along the river, such that their total profit can be maximized. Design an efficient algorithm to achieve this goal, and analyze its run time (Note that brute-force algorithm achieves $\Theta(n^2)$, so your algorithm must do better.)

4) 16 pts

Consider the following matching problem. There are m students s_1, s_2, \dots, s_m and a set of n companies $C = \{c_1, c_2, \dots, c_n\}$. Each student can work for only one company, whereas company c_j can hire up to b_j students. Student s_i has a preferred set of companies $\Lambda_i \subseteq C$ at which he/she is willing to work. Your task is to find an assignment of students to companies such that all of the above constraints are satisfied and each student is assigned. Formulate this as a network flow problem and describe any subsequent steps necessary to arrive at the solution. Prove correctness.

5) 16 pts

Consider a directed, weighted graph G where all edge weights are positive. You have one Star, which lets you change the weight of any one edge to zero. In other words, you may change the weight of any one edge to zero. Give an efficient algorithm using Dijkstra's algorithm to find a lowest-cost path between two vertices s and t , given that you may set one edge weight to zero. Note: you will receive 10 pts if your algorithm is efficient. You will receive full points (16 pts) if your algorithm has the same run time complexity as Dijkstra's algorithm.

6) 16 pts

You are given n rods; they are of length l_1, l_2, \dots, l_n , respectively. Our goal is to connect all the rods and form a single rod. The length after connecting two rods and the cost of connecting them are both equal to the sum of their lengths. Give an algorithm to minimize the cost of connecting them to form a single rod. State the complexity of your algorithm and prove that your algorithm is optimal.

Additional Space

CS570
Analysis of Algorithms
Fall 2015
Exam III

Name: _____

Student ID: _____

Email Address: _____

_____ Check if DEN Student

	Maximum	Received
Problem 1	20	
Problem 2	15	
Problem 3	12	
Problem 4	15	
Problem 5	15	
Problem 6	12	
Problem 7	11	
Total	100	

Instructions:

1. This is a 2-hr exam. Closed book and notes
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.

1) 20 pts

Mark the following statements as **TRUE, FALSE**. No need to provide any justification.

[TRUE/FALSE]

If P = NP, then all NP-Hard problems can be solved in Polynomial time.

[TRUE/FALSE]

Dynamic Programming approach only works when used on problems with non-overlapping sub problems.

[TRUE/FALSE]

In a divide & conquer algorithm, the size of each sub-problem must be at most half the size of the original problem.

[TRUE/FALSE]

In a 0-1 knapsack problem, a solution that uses up all of the capacity of the knapsack will be optimal.

[TRUE/FALSE]

If a problem X can be reduced to a known NP-hard problem, then X must be NP-hard.

[TRUE/FALSE]

If SAT \leq_P A, then A is NP-hard.

[TRUE/FALSE]

The recurrence $T(n) = 2T(n/2) + 3n$, has solution $T(n) = \theta(n \log(n^2))$.

[TRUE/FALSE]

Consider two positively weighted graphs $G_1 = (V, E, w_1)$ and $G_2 = (V, E, w_2)$ with the same vertices V and edges E such that, for any edge $e \in E$, we have $w_2(e) = (w_1(e))^2$. For any two vertices $u, v \in V$, any shortest path between u and v in G_2 is also a shortest path in G_1 .

[TRUE/FALSE]

If an undirected graph G=(V,E) has a Hamiltonian Cycle, then any DFS tree in G has a depth $|V| - 1$.

[TRUE/FALSE]

Linear programming is at least as hard as the Max Flow problem.

2) 15 pts

A company makes three models of desks, an executive model, an office model and a student model. Building each desk takes time in the cabinet shop, the finishing shop and the crating shop as shown in the table below:

Type of desk	Cabinet shop	Finishing shop	Crating shop	Profit
Executive	2	1	1	150
Office	1	2	1	125
Student	1	1	.5	50
Available hours	16	16	10	

How many of each type should they make to maximize profit? Use linear programming to formulate your solution. Assume that real numbers are acceptable in your solution.

3) 12 pts

Given a graph $G=(V, E)$ and a positive integer $k < |V|$. The longest-simple-cycle problem is the problem of determining whether a simple cycle (no repeated vertices) of length k exists in a graph. Show that this problem is NP-complete.

4) 15 pts

Suppose there are n steps, and one can climb either 1, 2, or 3 steps at a time. Determine how many different ways one can climb the n steps. E.g. if there are 5 steps, these are some possible ways to climb them: (1,1,1,1,1), (1,2,1,1), (3, 2), (2,3), etc. Your algorithm should run in linear time with respect to n . You need to include your complexity analysis.

5) 15 pts

We'd like to select frequencies for FM radio stations so that no two are too close in frequency (creating interference). Suppose there are n candidate frequencies $\{f_1, \dots, f_n\}$. Our goal is to pick as many frequencies as possible such that no two selected frequencies f_i, f_j have $|f_i - f_j| < e$ (for a given input variable e). Design a greedy algorithm to solve the problem. Prove the optimality of the algorithm and analyze the running time.

6) 12 pts

Let S be an NP-complete problem, and Q and R be two problems whose classification is unknown (i.e. we don't know whether they are in NP, or NP-hard, etc.). We do know that Q is polynomial time reducible to S and S is polynomial time reducible to R. Mark the following statements True or False **based only on the given information, and explain why.**

(i) Q is NP-complete

(ii) Q is NP-hard

(iii) R is NP-complete

(iv) R is NP-hard

7) 11 pts

Consider there are n students and n rooms. A student can only be assigned to one room. Each room has capacity to hold either one or two students. Each student has a subset of rooms as their possible choice. We also need to make sure that there is at least one student assigned to each room.

Give a polynomial time algorithm that determines whether a feasible assignment of students to rooms is possible that meets all of the above constraints. If there is a feasible assignment, describe how your solution can identify which student is assigned to which room.

Additional Space

Additional Space

CS570
Analysis of Algorithms
Fall 2015
Exam III

Name: _____

Student ID: _____

Email Address: _____

_____ Check if DEN Student

	Maximum	Received
Problem 1	20	
Problem 2	15	
Problem 3	12	
Problem 4	15	
Problem 5	15	
Problem 6	12	
Problem 7	11	
Total	100	

Instructions:

1. This is a 2-hr exam. Closed book and notes
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.

1) 20 pts

Mark the following statements as **TRUE**, **FALSE**. No need to provide any justification.

[/FALSE]

If P = NP, then all NP-Hard problems can be solved in Polynomial time.

[/FALSE]

Dynamic Programming approach only works when used on problems with non-overlapping sub problems.

[/FALSE]

In a divide & conquer algorithm, the size of each sub-problem must be at most half the size of the original problem.

[/FALSE]

In a 0-1 knapsack problem, a solution that uses up all of the capacity of the knapsack will be optimal.

[/FALSE]

If a problem X can be reduced to a known NP-hard problem, then X must be NP-hard.

[TRUE/]

If $\text{SAT} \leq_P A$, then A is NP-hard.

[TRUE/]

The recurrence $T(n) = 2T(n/2) + 3n$, has solution $T(n) = \theta(n \log(n^2))$.

[/FALSE]

Consider two positively weighted graphs $G_1 = (V, E, w_1)$ and $G_2 = (V, E, w_2)$ with the same vertices V and edges E such that, for any edge $e \in E$, we have $w_2(e) = (w_1(e))^2$. For any two vertices $u, v \in V$, any shortest path between u and v in G_2 is also a shortest path in G_1 .

[/FALSE]

If an undirected graph $G=(V,E)$ has a Hamiltonian Cycle, then any DFS tree in G has a depth $|V| - 1$.

[TRUE/]

Linear programming is at least as hard as the Max Flow problem.

2) 15 pts

A company makes three models of desks, an executive model, an office model and a student model. Building each desk takes time in the cabinet shop, the finishing shop and the crating shop as shown in the table below:

Type of desk	Cabinet shop	Finishing shop	Crating shop	Profit
Executive	2	1	1	150
Office	1	2	1	125
Student	1	1	.5	50
Available hours	16	16	10	

How many of each type should they make to maximize profit? Use linear programming to formulate your solution. Assume that real numbers are acceptable in your solution.

Solution:

Start by defining your variables:

x = number of executive desks made

y = number of office desks made

z = number of student desks made

Maximize $P=150x+125y+50z$.

Subject to:

$2x + y + z \leq 16$ cabinet hours

$x + 2y + z \leq 16$ finishing hours

$x + y + .5z \leq 10$ crating hours

$x \geq 0, y \geq 0, z \geq 0$

3) 12 pts

Given a graph $G=(V, E)$ and a positive integer $k < |V|$. The longest-simple-cycle problem is the problem of determining whether a simple cycle (no repeated vertices) of length k exists in a graph. Show that this problem is NP-complete.

Solution:

Clearly this problem is in NP. The certificate will be a cycle of the graph, and the certifier will check whether the certificate is really a cycle of length k of the given graph.

We will reduce HAM-CYCLE to this problem. Given an instance of HAM-CYCLE with graph $G=(V,E)$, construct a new graph $G'=(V', E)$ by adding one isolated vertex u to G . Now ask the longest-simple-cycle problem with $k = |V| < |V'|$ for graph G' .

If there is a HAM-CYCLE in G , it will be a cycle of length $k = |V|$ in G' .

If there is no HAM-CYCLE in G' , there must not be no cycle of length $|V|$ in G' . If there is, we know the cycle does not contain u , because it is isolated. So the cycle will contain all vertex in $|V|$, which is a HAM-CYCLE of the G .

The reduction is in polynomial time, so longest-simple-path is NP-Complete.

4) 15 pts

Suppose there are n steps, and one can climb either 1, 2, or 3 steps at a time. Determine how many different ways one can climb the n steps. E.g. if there are 5 steps, these are some possible ways to climb them: (1,1,1,1,1), (1,2,1,1), (3, 2), (2,3), etc. Your algorithm should run in linear time with respect to n . You need to include your complexity analysis.

Solution:

Let $T(n)$ denote the number of different ways for climbing up n stairs.

There are three choices for each first step: either 1, 2, or 3. $T(n) = T(n-1) + T(n-2) + T(n-3)$

The boundary conditions :

when there is only one step, $T(1) = 1$. If only two steps, $T(2) = 2$. $T(3)= 4$. ((1,1,1), (1,2), (2,1), (3))

Pseudo code as below:

```
if n is 1
    return 1
else if n is 2
    return 2
else
    T[1] = 1
    T[2] = 2
    T[3]=4
    for i = 4 to n do
        T[i] = T[i-1]+T[i-2]+ T[i-3]
    return T[n]
```

The time complexity is $\Theta(n)$.

5) 15 pts

We'd like to select frequencies for FM radio stations so that no two are too close in frequency (creating interference). Suppose there are n candidate frequencies $\{f_1, \dots, f_n\}$. Our goal is to pick as many frequencies as possible such that no two selected frequencies f_i, f_j have $|f_i - f_j| < e$ (for a given input variable e). Design a greedy algorithm to solve the problem. Prove the optimality of the algorithm and analyze the running time.

Proof:

I claim that there exists some optimal solution that makes the same first choice as (in terms of which selected frequency has the lowest value) that I do. Consider any optimal solution OPT .

Let p be my first choice and q be OPT 's first choice. If $p = q$, our proof is complete. If it isn't, we know that $p < q$; now consider some other set $OPT1 = OPT - \{q\} + \{p\}$; that is, OPT with its first choice replaced by mine. We know this is a valid set (meaning no frequencies are within e of one another), because $p < q$, and p is the first element in $OPT1$. Because nothing was within e of q , none can be within e of p . We also know that $OPT1$ is an optimal (maximum size) set, because it is the same size as OPT . Therefore, $OPT1$ is an optimal set that includes my first choice. (and therefore, by induction, the second choice will be correct, etc.)

6) 12 pts

Let S be an NP-complete problem, and Q and R be two problems whose classification is unknown (i.e. we don't know whether they are in NP, or NP-hard, etc.). We do know that Q is polynomial time reducible to S and S is polynomial time reducible to R. Mark the following statements True or False **based only on the given information, and explain why.**

(i) Q is NP-complete

False. Because $Q \leq_p S$, Q is at most as hard as S. Because S is in NPC, Q is not necessary to be in NPC, e.g., it could be in P, or could be in NP but not in NPC.

(ii) Q is NP-hard

False. Because $Q \leq_p S$, Q is at most as hard as S. Then Q is not necessary to be in NP-hard, e.g., it could be in P.

(iii) R is NP-complete

False. Because $S \leq_p R$, R is at least as hard as S. Then R is not necessary to be NPC. It is possible to be in NP-hard but not in NPC.

(iv) R is NP-hard

True. Because $S \leq_p R$, R is at least as hard as S. Then R is NP-hard.

7) 11 pts

Consider there are n students and n rooms. A student can only be assigned to one room. Each room has capacity to hold either one or two students. Each student has a subset of rooms as their possible choice. We also need to make sure that there is at least one student assigned to each room.

Give a polynomial time algorithm that determines whether a feasible assignment of students to rooms is possible that meets all of the above constraints. If there is a feasible assignment, describe how your solution can identify which student is assigned to which room.

Solution:

Construct a flow network as follows,

For each student i create a vertex a_i , for each room j create a vertex b_j , if room j is one of student i 's possible choices, add an edge from a_i to b_j with upper bound 1. Create a super source s , connect s to all a_i with an edge of lower bound and upper bound 1.

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

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

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 a_i and b_j , check whether the flow from a_i to b_j is 1, if yes, assign student i to room j .

Additional Space

Additional Space

CS570
Analysis of Algorithms
Fall 2016
Exam III

Name: _____

Student ID: _____

Email Address: _____

_____ **Check if DEN Student**

	Maximum	Received
Problem 1	20	
Problem 2	18	
Problem 3	18	
Problem 4	18	
Problem 5	18	
Problem 6	8	
Total	100	

Instructions:

1. This is a 2-hr exam. Closed book and notes
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.

1) 20 pts

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

[TRUE] This is the definition of NP.

Every problem in NP has a polynomial time certifier.

[FALSE] Decision problems can be in P.

Every decision problem is in NP-complete.

[TRUE] If 3-SAT reduces to it, it's NP-Hard and we now it's NP so it's NPC.

An NP problem is NP-complete if 3-SAT reduces to it in polynomial time.

[FALSE] It will be $O(m^2 + mn)$ and not linear.

If all edges in a graph have capacity 1, then Ford-Fulkerson runs in linear time.

[FALSE] consider a graph with edges $(a,b) = 4$, $(b,c) = 3$ and $(a, c) = 5$. The shortest path from a to c is 5 but (a, c) is not in the MST.

Let T be a minimum spanning tree of G. Then, for any pair of vertices s and t, the shortest s-t path in G is the path in T.

[TRUE] Master's Theorem

If the running time of a divide-and-conquer algorithm satisfies the recurrence $T(n) = 3 T(n/2) + \Theta(n^2)$, then $T(n) = \Theta(n^2)$.

[FALSE] Consider the problem in the previous item.

In a divide and conquer solution, the sub-problems are disjoint and are of the same size.

[FALSE] Many of them are NP-Complete.

All Integer linear programming problems can be solved in polynomial time.

[TRUE]

If the linear program is feasible and bounded, then there exists an optimal solution.

[FALSE] Each specific operation can be larger than $O(\log n)$.

Suppose we have a data structure where the amortized running time of Insert and Delete is $O(\lg n)$. Then in any sequence of $2n$ calls to Insert and Delete, the worst-case running time for the nth call is $O(\lg n)$.

2) 18 pts

We are given an infinite array where the first n elements contain integers in sorted order and the rest of the elements are filled with ∞ . We are not given the value of n . Describe an algorithm that takes an integer x as input and finds a position in the array containing x , if such a position exists, in $O(\lg n)$ time.

Solution:

- First determine the smallest power of 2 larger than n (8 points)

```
N=1  
while A[i] ≠ ∞  
    N = N*2  
endwhile
```

N will be the smallest power of 2 larger than n

- Use binary search on $A[1..N]$ to find x (8 points)
- Complexity: both components run in $O(\lg n)$ time

Common mistakes:

1. Without finding N , directly do binary search on $A[1..n]$

Note that n is not given. Cases like this get 4 points.

2. Without finding N , directly do binary search on $A[1..2x+1]$

x could be a negative integer; in this case, $2x+1$ is also negative.

Cases like this get 8 points.

3. Time complexity is larger than $O(\log n)$ time. (e.g., linearly scan the array, or build a min heap first)

Cases like this get 0 points

3) 18 pts

For bit strings $A = a_1, \dots, a_m$, $B = b_1, \dots, b_n$ and $C = c_1, \dots, c_{m+n}$, we say that C is an interleaving of A and B if it can be obtained by interleaving the bits in A and B in a way that maintains the left-to-right order of the bits in A and B . For example if $A = 101$ and $B = 01$ then $a_1a_2b_1a_3b_2 = 10011$ is an interleaving of A and B , whereas 11010 is not. Give the most efficient algorithm you can to determine if C is an interleaving of A and B . Analyze the time complexity of your solution as a function of m and n .

Solution: The general form of the subproblem we solve will be: determine if c_1, \dots, c_{i+j} is an interleaving of a_1, \dots, a_i and b_1, \dots, b_j for $0 \leq i \leq |A|$ and $0 \leq j \leq |B|$. Let $c[i, j]$ be true if and only if c_1, \dots, c_{i+j} is an interleaving of a_1, \dots, a_i and b_1, \dots, b_j . We use the convention that if $i = 0$ then $a_i = \Phi$ (the empty string) and if $j = 0$ then $b_j = \Phi$. The subproblem $c[i, j]$ can be recursively defined as shown (where $c[|A|, |B|]$ gives the answer to the optimal problem):

$$c[i, j] = \begin{cases} \text{true,} & \text{if } i = j = 0 \\ \text{false,} & \text{if } a_i \neq c_{i+j} \text{ and } b_j \neq c_{i+j} \\ c[i-1, j], & \text{if } a_i = c_{i+j} \text{ and } b_j \neq c_{i+j} \\ c[i, j-1], & \text{if } a_i \neq c_{i+j} \text{ and } b_j = c_{i+j} \\ c[i-1, j]c[i, j-1], & \text{if } a_i = b_j = c_{i+j} \end{cases}$$

The time complexity is clearly $O(|A||B|)$ since there are $|A|x|B|$ subproblems each of which is solved in constant time. Finally, the $c[i, j]$ matrix can be computed in row major order.

Proof of recurrence not asked for in the problem statement: We now argue this recursive definition is correct. First the case where $i = j = 0$ is when both A and B are empty and then by definition C (which is also empty) is a valid interleaving of A and B . If $a_i \neq c_{i+j}$ and $b_j = c_{i+j}$ then there could only be a valid interleaving in which a_i appears last in the interleaving, and hence $c[i, j]$ is true exactly when c_1, \dots, c_{i+j-1} is a valid interleaving of a_1, \dots, a_{i-1} and b_1, \dots, b_j which is given by $c[i-1, j]$. Similarly, when $a_i \neq c_{i+j}$ and $b_j = c_{i+j}$ then $c[i, j] = c[i-1, j]$.

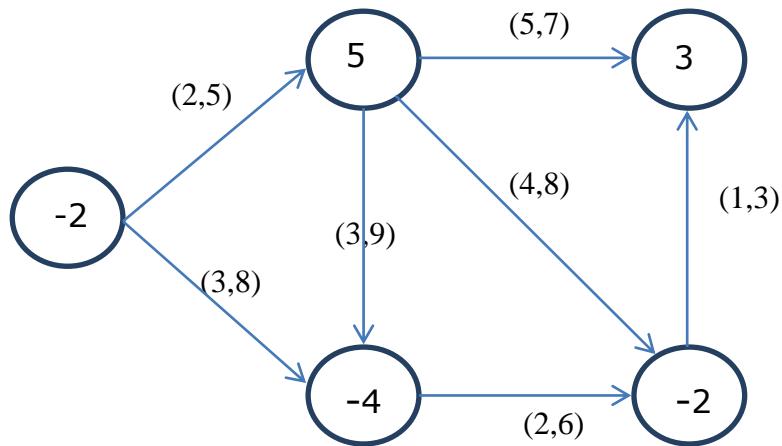
Finally, consider when $a_i = b_j = c_{i+j}$. In this case the interleaving (if it exists) must either end with b_i (in which case $c[i-1, j]$ is true) or must end with a_i (in which case $c[i, j-1]$ is true). Thus returning $c[i-1, j] \vee c[i, j-1]$ gives the correct answer.

Finally, since in all cases the value of $c[i, j]$ comes directly from the answer to one of the subproblems, we have the optimal substructure property.

- Recurrence and definition of OPT (12 pts)
- Algorithm (5 pts)
- Complexity (3 points)

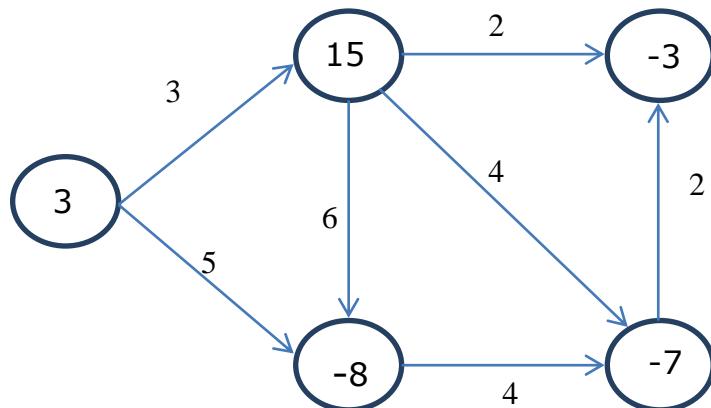
4) 18 pts

In the network 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.

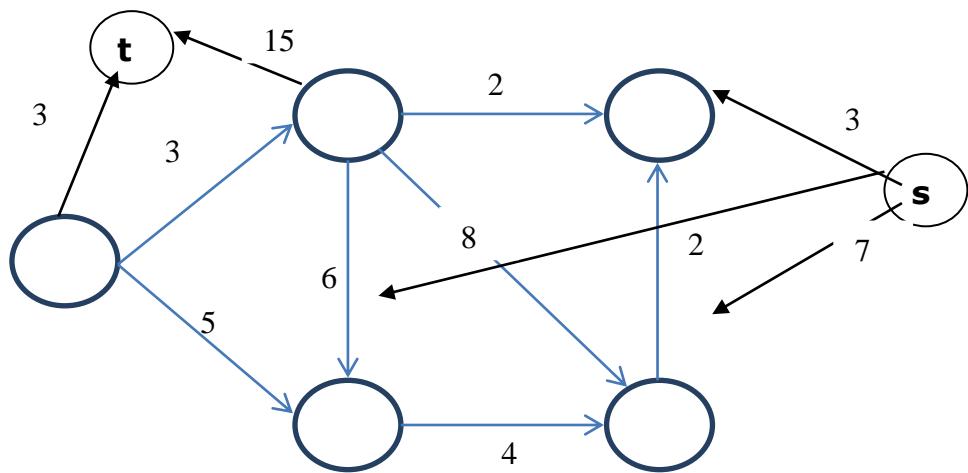


Solution.

Remove lower bounds by changing demands on each vertex



Reduce it to max-flow by adding s and t vertices.



Check if there is a s-t flow $|f|= 18$.

There is no such flow. It follows, no circulation.

6 points - for eliminating lower bounds and converting to problem with only demands

6 points - for eliminating demands and converting to max-flow problem

6 points - Checking if there exists a max-flow of value D

5) 18 pts

The Metric Traveling Salesman Problem (metric-TSP) is a restriction of the Traveling Salesman Problem, defined as follows. Given n cities c_1, \dots, c_n with inter-city distances $d(c_i, c_j)$ that form a metric:

$$d(c_i, c_i) = 0$$

$$d(c_i, c_j) > 0 \text{ for } i \neq j$$

$$d(c_i, c_j) + d(c_j, c_k) \geq d(c_i, c_k) \text{ for } i, j, k \in \{1, \dots, n\}$$

Is there a closed tour that visits each city exactly once and covers a total distance at most C ? Prove that metric-TSP is in NP-complete.

Solution.

1. It's easy to see that is in NP
2. Reduce from HC.

Given a graph $G=(V,E)$ on which we want to solve the HAM-CYCLE problem. We will construct a complete G' with metric d as follows

$$d(u,v) = 0, \text{ if } u = v$$

$$d(u,v) = 1, \text{ if } (u,v) \text{ is in } E$$

$$d(u,v) = 2, \text{ otherwise}$$

Since triangle inequalities hold in G' (edges of any triangle will have sizes of 1 or 2), then $d(c_i, c_j) + d(c_j, c_k) \geq d(c_i, c_k)$ for $i, j, k \in \{1, \dots, n\}$

Let the bound $C = |V|$.

Claim: metric-TSP on G' is solvable if and only if the HC problem on G is solvable.

->) If there is a metric-TSP of length $|V|$, then every distance between successive cities must be 1. Therefore these define as HC on G .

<-) If G has a hamiltonian cycle, then it defines a metric-TSP of size $|V|$.

6) 8 pts

Convert the following linear program

$$\text{maximize } (3x_1 + 8x_2)$$

Subject to

$$x_1 + 4x_2 \leq 20$$

$$x_1 + x_2 \geq 7$$

$$x_1 \geq -1$$

$$x_2 \leq 5$$

to the standard form. You need to show all your steps.

$$x_1 + 1 = z_1 \geq 0$$

$$x_2 - 5 \leq 0 \text{ so } 5 - x_2 = z_2 \geq 0$$

$$x_1 + 4x_2 \leq 0 \rightarrow (z_1 - 1) + 4(5 - z_2) \leq 20 \rightarrow z_1 - 4z_2 \leq 1$$

$$x_1 + x_2 \geq 7 \rightarrow (z_1 - 1) + (5 - z_2) \geq 7 \rightarrow z_2 - z_1 \leq -3$$

finally we get:

$$\text{maximize } 3z_1 - 8z_2 + 37$$

subject to:

$$z_1 - 4z_2 \leq 1$$

$$z_2 - z_1 \leq -3$$

$$z_1 \geq 0$$

$$z_2 \geq 0$$

Additional Space

Additional Space

RCS570 Fall 2017: Analysis of Algorithms Exam II

	Points
Problem 1	20
Problem 2	20
Problem 3	15
Problem 4	15
Problem 5	15
Problem 6	15
Total	100

Instructions:

1. This is a 2-hr exam. Closed book and notes
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.

1) 20 pts

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

[**TRUE/FALSE**]

It is possible for a dynamic programming algorithm to have an exponential running time.

[**TRUE/FALSE**]

In a connected, directed graph with positive edge weights, the Bellman-Ford algorithm runs asymptotically faster than the Dijkstra algorithm.

[**TRUE /FALSE**]

There exist some problems that can be solved by dynamic programming, but cannot be solved by greedy algorithm.

[**TRUE /FALSE**]

The Floyd-Warshall algorithm is asymptotically faster than running the Bellman-Ford algorithm from each vertex.

[**TRUE /FALSE**]

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

[**TRUE /FALSE**]

The Ford-Fulkerson algorithm solves the maximum bipartite matching problem in polynomial time.

[**TRUE /FALSE**]

Given a solution to a max-flow problem, that includes the final residual graph G_f . We can verify in a *linear* time that the solution does indeed give a maximum flow.

[**TRUE /FALSE**]

In a flow network, a flow value is upper-bounded by a cut capacity.

[**TRUE/FALSE**]

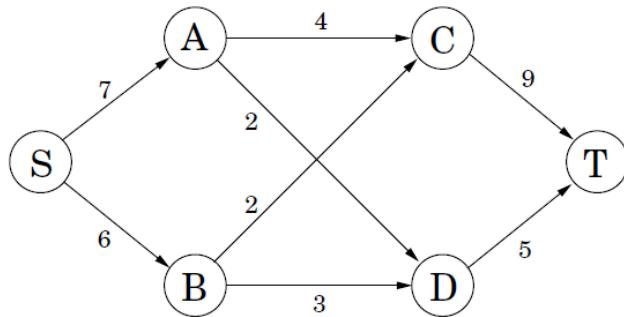
In a flow network, a min-cut is always unique.

[**TRUE/FALSE**]

A maximum flow in an integer capacity graph must have an integer flow on each edge.

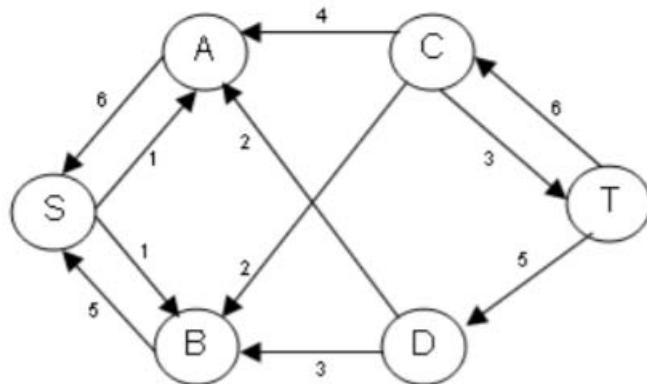
2) 20 pts.

You are given the following graph G . Each edge is labeled with the capacity of that edge.



- a) Find a max-flow in G using the Ford-Fulkerson algorithm. Draw the residual graph G_f corresponding to the max flow. You do not need to show all intermediate steps. (10 pts)

solution



Grading Rubics:

Residual graph: 10 points in total

For each edge in the residual graph, any wrong number marked, wrong direction, or missing will result in losing 1 point.

The total points you lose is equal to the number of edges on which you make any mistake shown above.

b) Find the max-flow value and a min-cut. (6 pts)

Solution: $f = 11$, cut: ($\{S, A, B\}$, $\{C, D, T\}$)

Grading Rubics:

Max-flow value and min-cut: 6 points in total

Max-flow: 2 points.

- If you give the wrong value, you lose the 2 points.

Min-cut: 4 points

- If your solution forms a cut but not a min-cut for the graph, you lose 3 points
- If your solution does not even form a cut, you lose all the 4 points

c) Prove or disprove that increasing the capacity of an edge that belongs to a min cut will always result in increasing the maximum flow. (4 pts)

Solution: increasing (B,D) by one won't increase the max-flow

Grading Rubics:

Prove or Disprove: 4 points in total

- If you judge it "True", but give a structural complete "proof". You get at most 1 point
- If you judge it "False", you get 2 points.
- If your counter example is correct, you get the rest 2 points.

Popular mistake: a number of students try to disprove it by showing that if the min-cut in the original graph is non-unique, then it is possible to find an edge in one min-cut set, such that increasing the capacity of this does not result in max-flow increase.

But they did not do the following thing:

The existence of the network with multiple min-cuts needs to be proved, though it seems to be obvious. The most straightforward way to prove the existence is to give an example-network that has multiple min-cuts. Then it turns out to be giving a counter example for the original question statement.

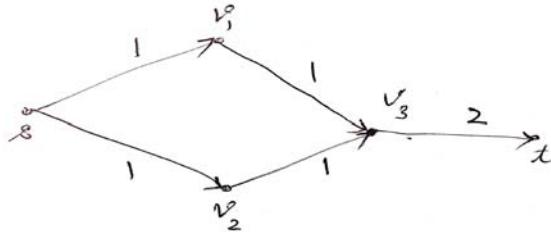
3) 15 pts.

Given a flow network with the source s and the sink t , and positive integer edge capacities c . Let (S, T) be a minimum cut. Prove or disprove the following statement:
If we increase the capacity of every edge by 1, then (S, T) still be a minimum cut.

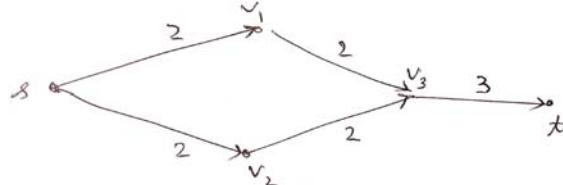
False. Create a counterexample.

An instance of a counter-example:

Initially, a (S, T) min cut is $S : \{s, v_1, v_2\}$ and $T : \{v_3, t\}$



After increasing capacity of every edge by 1, (S, T) is no longer a min-cut. We now have $S' : \{s, v_1, v_2, v_3\}$ and $T' : \{t\}$



Grading rubric:

If you try to prove the statement: -15

For an incorrect counter-example: -9

No credit for simply stating true or false.

4) 15 pts.

Given an unlimited supply of coins of denominations d_1, d_2, \dots, d_n , we wish to make change for an amount C . This might not be always possible. Your goal is to verify if it is possible to make such change. Design an algorithm by *reduction* to the knapsack problem.

- a) Describe reduction. What is the knapsack capacity, values and weights of the items? (10 pts.)

Capacity is C . (2 points)
values = weights = d_k . (4points)

- You need to recognize that the problem should be modeled based on the unbounded knapsack problem with description of the reduction: (5 points)
- Explanation of the verification criteria (4 points)
 $\text{Opt}(j)$: the maximum change equal or less than j that can be achieved with d_1, d_2, \dots, d_n
 $\text{Opt}(0)=0$
 $\text{Opt}(j) = \max[\text{opt}(j-d_i)+d_i] \text{ for } d_i \leq j$
If we obtain $\text{Opt}(C) = C$, it means the change is possible.

- b) Compute the runtime of your verification algorithm. (5 pts)

$O(nC)$ (3 points), Explanation (2 points).

5) 15 pts.

You are considering opening a series of electrical vehicle charging stations along Pacific Coast Highway (PCH). There are n possible locations along the highway, and the distance from the start to location k is $d_k \geq 0$, where $k = 1, 2, \dots, n$. You may assume that $d_i < d_k$ for $i < k$. There are two important constraints:

- 1) at each location k you can open only one charging station with the expected profit p_k , $k = 1, 2, \dots, n$.
- 2) you must open at least one charging station along the whole highway.
- 3) any two stations should be at least M miles apart.

Give a DP algorithm to find the maximum expected total profit subject to the given constraints.

- a) Define (in plain English) subproblems to be solved. (3 pts)

Let $\text{OPT}(k)$ be the maximum expected profit which you can obtain from locations $1, 2, \dots, k$.

Rubrics

Any other definition is okay as long as it will recursively solve subproblems

- b) Write the recurrence relation for subproblems. (6 pts)

$$\text{OPT}(k) = \max \{\text{OPT}(k - 1), p_k + \text{OPT}(f(k))\}$$

where $f(k)$ finds the largest index j such that $d_j \leq d_k - M$, when such j doesn't exist, $f(k)=0$

Base cases:

$$\text{OPT}(1) = p_1$$

$$\text{OPT}(0) = 0$$

Rubrics:

Error in recursion -2pts, if multiple errors, deduction adds up

No base cases: -2pts

Missing base cases: -1pts

Using variables without definition or explanation: -2pts

Overloading variables (re-defining n , p , k , or M): -2pts

- c) Compute the runtime of the above DP algorithm in terms of n . (3 pts)

algorithm solves n subproblems; each subproblem requires finding an index $f(k)$ which can be done in time $O(\log n)$ by binary search.
Hence, the running time is $O(n \log n)$.

Rubric:

$O(n^2)$ is regarded as okay

$O(d_n n)$ pseudo-polynomial is also okay if the recursion goes over all d_n values

$O(n)$ is also okay

$O(n^3), O(nM), O(kn)$: no credit

- d) How can you optimize the algorithm to have $O(n)$ runtime? (3 pts)

Preprocess distances $r_i = d_i - M$. Merge d-list with r-list.

Rubric

Claiming “optimal solution is already found in c ” only gets credit when explanation about pre-process is described in either part b or c.

Without proper explanation (e.g. assume we have, or we can do): no credit

Keeping an array of max profits: no credit, finding the index that is closest to the installed station with M distance away is the bottleneck, which requires pre-processing.

6) 15 pts.

A group of traders are leaving Switzerland, and need to convert their Francs into various international currencies. There are n traders t_1, t_2, \dots, t_n and m currencies c_1, c_2, \dots, c_m . Trader t_k has F_k Francs to convert. For each currency c_j , the bank can convert at most B_j Francs to c_j . Trader t_k is willing to trade as much as S_{kj} of his Francs for currency c_j . (For example, a trader with 1000 Francs might be willing to convert up to 200 of his Francs for USD, up to 500 of his Francs for Japanese's Yen, and up to 200 of his Francs for Euros). Assuming that all traders give their requests to the bank at the same time, describe an algorithm that the bank can use to satisfy the requests (if it can).

a) Describe how to construct a flow network to solve this problem, including the description of nodes, edges, edge directions and their capacities. (8 pts)

Bipartite graph: one partition traders t_1, t_2, \dots, t_n . Other, available currency, c_1, c_2, \dots, c_m .

Connect t_k to c_j with the capacity S_{kj}

Connect source to traders with the capacity F_k .

Connect available currency c_j to the sink with the capacity B_j .

Rubrics:

- Didn't include supersource (-1 point)
- Didn't include traders nodes (-1 point)
- Didn't include currencies nodes (-1 point)
- Didn't include supersink (-1 point)
- Didn't include edge direction (-1 point)
- Assigned no/wrong capacity on edges between source & traders (-1 point)
- Assigned no/wrong capacity on edges between traders & currencies (-1 point)
- Assigned no/wrong capacity on edges between currencies & sink (-1 point)

b) Describe on what condition the bank can satisfy all requests. (4 pts)

If there is a flow f in the network with $|f| = F_1 + \dots + F_n$, then all traders are able to convert their currencies.

Rubrics:

- Wrong condition (-4 points): Unless you explicitly included $|f| = F_1 + \dots + F_n$, no partial points were given for this subproblem.
- In addition to correct answer, added additional condition which is wrong (-2 points)
- No partial points were given for conditions that satisfy only some of the requests, not all requests.
- Note that $\sum S_{kj}$ and $\sum B_j$ can be larger than $\sum F_k$ in some cases where bank satisfy all requests.
- Note that $\sum S_{kj}$ can be larger than $\sum B_j$ in some cases where bank satisfy all requests.
- It is possible that $\sum S_{kj}$ or $\sum B_j$ is smaller than $\sum F_k$. In these cases, bank can never satisfy all requests because max flow will be smaller than $\sum F_k$.

- c) Assume that you execute the Ford-Fulkerson algorithm on the flow network graph in part a), what would be the runtime of your algorithm? (3 pts)

$O(n m |f|)$

Rubrics:

- Flow($|f|$) was not included (-1 point)
- Wrong description (-1 point)
- Computation is incorrect (-1 point)
- Notation error (-1 point)
- Missing big O notation (-1 point)
- Used big Theta notation instead of big O notation (-1 point)
- Wrong runtime complexity (-3 points)
- No runtime complexity is given (-3 points)

CS570
Analysis of Algorithms
Summer 2011
Final Exam

Name: _____

Student ID: _____

_____ Check if DEN student

	Maximum	Received
Problem 1	20	
Problem 2	13	
Problem 3	14	
Problem 4	13	
Problem 5	20	
Problem 6	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 except for the question at the bottom of the page.

[**TRUE/FALSE**]

In a flow network, if all edge capacities are distinct, then the max flow of this network is unique.

[**TRUE/FALSE**]

To find the minimum element in a max heap of n elements, it takes $O(n)$ time.

[**TRUE/FALSE**]

Let T be a spanning tree of graph $G(V, E)$, let k be the number of edges in T, then $k=O(V)$

[**TRUE/FALSE**]

Linear programming problems can be solved in polynomial time.

[**TRUE/FALSE**]

Consider problem A: given a flow network, find the maximum flow from a node s to a node t. problem A is in NP.

[**TRUE/FALSE**]

Given n numbers, it takes $O(n)$ time to construct a binary min heap.

[**TRUE/FALSE**]

Kruskal's algorithm for finding the MST works with positive and negative edge weights.

[**TRUE/FALSE**]

Breadth first search is an example of a divide-and-conquer algorithm.

[**TRUE/FALSE**]

If a problem is not in P, then it must be in NP.

[**TRUE/FALSE**]

L1 can be reduced to L2 in Polynomial time and L1 is in NP, then L2 is in NP

2) 13 pts

Imagine that you constructed an approximation algorithm for the Traveling Salesman Problem that could always calculate a solution that is correct within a factor of $1/k$ of the optimal tour in $O(n^{2k})$ time. Would you be able to use this approximation algorithm to obtain a “good” solution to all other NP-Complete problems? Explain why or why not.

Yes. You can use it.

In the Traveling Salesperson Problem, we are given an undirected graph $G = (V, E)$ and cost $c(e) > 0$ for each edge $e \in E$. Our goal is to find a Hamiltonian cycle with minimum cost. A cycle is said to be Hamiltonian if it visits every vertex in V exactly once.

TSP is known to be NP-complete, and so we cannot expect to exactly solve TSP in polynomial time. What is worse, there is no good approximation algorithm for TSP unless $P = NP$. This is because if one can give a good approximation solution to TSP in polynomial time, then we can exactly solve the NP-Complete Hamiltonian cycle problem (HAM) in polynomial time, which is impossible unless $P = NP$. Recall that HAM is the decision problem of deciding whether the given graph $G = (V, E)$ has a Hamiltonian cycle or not.

Theorem 1 ([2]) The Traveling Salesperson Problem cannot be approximated within any factor, unless $P = NP$.

Proof: For the sake of contradiction, suppose we have an approximation algorithm A on TSP with an approximation ratio C . We show a contradiction by showing that using A , we can exactly solve HAM in polynomial time. Let $G = (V, E)$ be the given instance of HAM. We create a new graph $H = (V, E_0)$ with cost $c(e)$ for each $e \in E_0$ such that $c(e) = 1$ if $e \in E$, otherwise $c(e) = B$, where $B = nC + 1$ and $n = |V|$.

We observe that if G has a Hamiltonian cycle, $OPT = n$, otherwise $OPT \geq n - 1 + B = n(C + 1)$.

(Here, OPT denotes the cost of an optimal TSP solution in H .) Note that there is a “gap” between when G has a Hamiltonian cycle and when it does not. Thus, if A has an approximation ratio of C , we can tell whether G has a Hamiltonian cycle or not: Simply run A on the graph H ; if G has a Hamiltonian cycle, A returns a TSP tour in H of cost at most $C OPT = Cn$. Otherwise, H has no TSP tour of cost less than $n(C + 1)$, and so A must return a tour of at least this cost.

3) 14 pts

Prove the following statement or give a counterexample: "For any weighted graph G there is node v in G such that a Shortest Path Tree (tree found by running Dijkstra's) with v as source is identical to a Minimum Spanning Tree of G ."

False. Note that the MST will always avoid the largest weight edge in a cycle (assuming there is a unique largest), whereas the shortest path tree might use it. Consider the following graph where the diagonal edges

have weight 3 and the other edges have weight 2. A MST consists of three of the weight 2 edges, whereas a shortest path tree will use one of the diagonal edges.

4) 13 pts

Define the capacity of a node as the maximum incoming flow it can have. Briefly show how to reduce the problem of finding the maxflow in a network with capacity limits on the nodes to the usual maxflow problem. Answer in one or two sentences.

Replace the node v with two nodes v' and v'' . Add an edge from v' to v'' with the desired node capacity. Replace all of the edges (u, v) that enter v to (u, v') so that they enter v' . Replace all of the edges (v, w) that leave v to (v'', w) so that they leave v'' .

5) 20 pts

Let $G = (V, E)$ be an undirected graph in which the vertices represent small towns and the edges represent roads between those towns. Each edge e has a positive integer weight $d(e)$ associated with it, indicating the length of that road. The distance between two vertices (towns) in a graph is defined to be the length of the shortest weighted path between those two vertices.

Each vertex v also has a positive integer $c(v)$ associated with it, indicating the cost to build a fire station in that town.

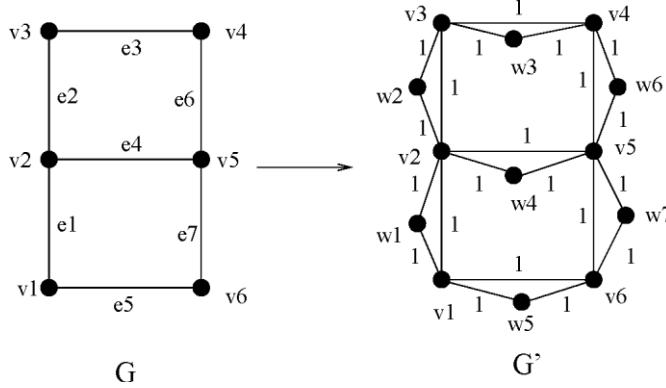
In addition, we are given two positive integer parameters D and C . Our objective is to determine whether there is a way to build fire stations such that the total cost of building the fire stations does not exceed C and the distance from any town to a fire station does not exceed D . This problem is known as the Rural Fire Station (RFS) Problem.

Your company has been hired by the American League of Rural Fire Departments to study this problem. After spending months trying unsuccessfully to find an efficient algorithm for the problem, your boss has a hunch that the problem is NP-complete. Prove that RFS is NP-complete.

We prove that FIRE-STATION-PLACEMENT is NP-complete. To see that it is NP we use the set of vertices for placing the fire stations as a certificate. Clearly, there is a polynomial time verification algorithm.

We now prove that VERTEX-COVER \leq_p FIRE-STATION-PLACEMENT. Let (G, k) be the input for VERTEX-COVER. Without loss of generality, we assume that no vertices in G have degree 0. (If G had any vertices of degree 0, then remove them. Notice that this won't affect the size of a minimum vertex cover of G .)

The graph for the fire-station-placement problem $G' = (V', E')$ is obtained from $G = (V, E)$ as follows. Let $V = \{v_1, \dots, v_n\}$ and $E = \{e_1, \dots, e_m\}$. We will create a new vertex w_i for each edge in E . Let $W = \{w_1, \dots, w_m\}$. Then $V' = V \cup W$ and for each edge $e = (u, v) \in E$ (with associated vertex w) we place the following 3 edges in E' : (u, v) , (u, w) , (v, w) . All edges have a weight of 1. So $|V'| = |V| + |E|$ and $|E'| = 3|E|$. Here is an example of this transformation:



Clearly G' can be constructed in polynomial time. The input for FIRE-STATION-PLACEMENT is $(G', C = k, D = 1)$ and the cost of building each fire station is 1.

We now prove that G has a vertex cover of size k if and only if G' has a placement of k fire stations so that each vertex has distance of at most 1 to its nearest fire station. Suppose that G has a vertex cover C of size k . We show that by placing a fire station in G' at each vertex in C all vertices are directly connected to a fire station (or have a fire station). Suppose not. Let u be such a vertex greater than distance 1 from its nearest fire station. First suppose that $u \in V$ (i.e. an original vertex from G). Since all vertices in G have at least one edge, u must be connected in G' to some vertex $v \in V$ where v is also not in C . This contradicts that C is a vertex cover. Next suppose that $u \in W$ (it is one of the added vertices), then u is connected to two vertices v_1 and v_2 where $(v_1, v_2) \in E$ and there is not a fire station at v_1 or v_2 thus contradicting that C was a vertex cover.

We first prove that if F is the set of vertices to place fire stations so that $|F| = k$ and all vertices have distance at most 1 to some fire station, then there is a vertex cover of size k in G .

We first transform F into a fire station placement solution F_0 in which $|F'| = |F|$ and in which F' is a subset of V (i.e. F' only includes original vertices). We build F' from F by doing the following: for each vertex $w \in W \setminus F$ (in any order). Suppose w is the vertex added corresponding to edge (u_1, u_2) . In this case, we can simply move the fire station at vertex w to u_1 (or anywhere else if u_1 already has a fire station). Notice that the fire station at w could only be a nearest fire station for w, u_1 and u_2 and hence if there is a fire station at u_1 then each of w, u_1 and u_2 can use u_1 as a nearest fire station. Hence we maintain the invariant (started with F) that our solution at each step (and hence F') satisfies the requirements that each vertex in $V' = V \cup W$ is within distance 1 from a fire station.

We now prove that F' is a vertex cover in G . Suppose not. Then there is an edge $(u, v) \in G$ such that neither of u or v belong to F' . Consider the vertex w corresponding to the edge (u, v) (i.e the triangle u, v, w). Notice that w is connected only to u and v and thus w must have a distance of greater than 1 to its nearest fire station thus contradicting the requirements for a proper fire station placement in G' .

6) 20 pts

You are given a array of n positive numbers $A[1] \dots A[n]$. You are not allowed to re-order them. You need to find two indexes i and j, where $1 \leq i < j \leq n$, such that $A[j] - A[i]$ is maximized. Solve this problem in $O(n)$ time using a dynamic programming method.

Let $OPT[i]$ be the max difference of the first i numbers.

$$OPT[i+1] = \max\{OPT[i], A[i+1] - \min[i]\}$$

Where $\min[i]$ is the minimum of the first i numbers.

$$\min[i+1] = \min\{A[i+1], \min[i]\}.$$

To find the indices, we can record the choice $OPT[]$ and $\min[]$ of each step.

Calculate this takes $O(n)$ because each element of the array is visited once.

CS570 Fall 2018: Analysis of Algorithms Exam III

	Points		Points
Problem 1	20	Problem 5	10
Problem 2	8	Problem 6	16
Problem 3	14	Problem 7	12
Problem 4	20		
	Total	100	

Instructions:

1. This is a 2-hr exam. Closed book and notes
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.

1) 20 pts

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

[**TRUE/FALSE**]

Let X be a decision problem. If we prove that X is in the class NP and give a poly-time reduction from X to 3-SAT, we can conclude that X is NP-complete.

[**TRUE/FALSE**]

Let A be an algorithm that operates on a list of n objects, where n is a power of two. A spends $\Theta(n^2)$ time dividing its input list into two equal pieces and selecting one of the two pieces. It then calls itself recursively on that list of $n/2$ elements. Then A 's running time on a list of n elements is $O(n)$.

[**TRUE/FALSE**]

If there is a polynomial time algorithm to solve problem A then A is in NP.

[**TRUE/FALSE**]

A pseudo-polynomial time algorithm is always slower than a polynomial time algorithm.

[**TRUE/FALSE**]

In a dynamic programming formulation, the sub-problems must be non-overlapping.

[**TRUE/FALSE**]

A spanning tree of a given undirected, connected graph $G = (V, E)$ can be found in $O(E)$ time.

[**TRUE/FALSE**]

Ford-Fulkerson can return a zero maximum flow for flow networks with non-zero capacities.

[**TRUE/FALSE**]

If $f(n) = \Theta(g(n))$ and $g(n) = \Theta(h(n))$, then $h(n) = \Theta(f(n))$

[**TRUE/FALSE**]

There is a polynomial-time solution for the 0/1 Knapsack problem if all items have the same weight but different values.

[**TRUE/FALSE**]

If there are negative cost edges in a graph but no negative cost cycles, Dijkstra's algorithm still runs correctly.

2) 8 pts

Let $G = (V, E)$ be a simple graph with n vertices. Suppose the weight of every edge of G is one.

Note: In a simple graph there is at most one edge directly connecting any two nodes.

For example, between nodes u and v there will be at most one edge uv .

- (a) What is the weight of a minimum spanning tree of G ? (1 pt)
- (b) Suppose we change the weight of two edges of G to $1/2$. What is the weight of the minimum spanning tree of G ? (2 pts)
- (c) Suppose we change the weight of three edges of G to $1/2$. What is the minimum and maximum possible weights for the the minimum spanning tree of G ? (2 pts)
- (d) Suppose we change the weight of $k < V$ edges of G to $1/2$. What is the minimum and maximum possible weights for the the minimum spanning tree of G ? (3 pts)
1. Any spanning tree of G has exactly $V - 1$ edges, since all the weights are one its total weight is $V - 1$.
 2. To answer this question, consider running Kruskal on G , it will first pick the two lighter edges as they cannot form a cycle (the graph is simple), then $V - 3$ other edge each with weight one. Therefore, the total value is $V - 3 + 2(1/2) = V - 2$.
 3. Use the similar approach as (b). Kruskal first selects two light edges. There are two possibilities for the third edge: (i) it forms a cycle with the first two edges or (ii) it does not form a cycle with the first two edges not. In case (i), Kruskal selects two edges of weight $1/2$ and $V - 3$ edges of weight 1, therefore, the total weight is $V - 3 + 2(1/2) = V - 2$. In case (ii), Kruskal selects three edges of weight $1/2$ and $V - 4$ edges of weight 1, therefore, the total weight is $V - 4 + 3(1/2) = V - 5/2$.
 4. Similar to (c), the minimum weight happens when the k light edges do not form any cycle. In this case, MST contains k edges of weight $1/2$ and $n - 1 - k$ edges of weight 1. Therefore, its weight is $k/2 + n - 1 - k = n - k/2 - 1$. The maximum weight happens if the k edges span as few vertices as possible. This happens when the k edges are part of an almost *complete graph*. Let h be a variable denoting the number of nodes in this subgraph such that the number of edges in a complete graph comprised of these edges nodes exceeds k . In particular, define h to be the smallest number such that, the binomial coefficient $\text{Binomial}[h,2] > k$, ie., $(h \text{ choose } 2) > k$. Accordingly, we can pack all k light edges between h vertices. Consequently, the MST has $h - 1$ edges of weight $1/2$ and $n - 1 - (h - 1)$ edges of weight 1. Therefore, its weight is $n - h/2 - 1/2$.

Rubric:

For each part (a,b,c,d), No partial marking.

This also applies to the parts where minimum and maximum is asked.

If either is wrong then there is no partial marks for that part.

Q3. 14 pts

Recall that in the discussion class we showed that the Bipartite **Undirected** Hamiltonian Cycle problem is NP-complete. Now in the Bipartite **Directed** Hamiltonian Cycle problem, we are given a bipartite directed graph and asked whether there is a simple cycle which visits every node exactly once. Note that this problem might potentially be easier than Hamiltonian Cycle because it assumes a directed bipartite graph. Prove that Bipartite Directed Hamiltonian Cycle is in fact still NP-Complete.

Solution:

- i) This problem is NP. Given a candidate path, we just check the nodes along the given path one by one to see if every node in the network is visited exactly once and if each consecutive node pair along the path are joined by an edge. (3)
- ii) To prove the problem is NP-complete, we reduce Directed Hamiltonian Cycle (DHC) problem to Bipartite Directed Hamiltonian Cycle (BDHC) problem. (2)

Given an arbitrary directed graph G , we split each vertex v in G into two vertex v_{in} and v_{out} . Here v_{in} connects all the incoming edges to v in G ; v_{out} connects all the outgoing edges from v in G . Moreover, we connect one directed edge from v_{in} to v_{out} . After doing these operations for each node in G , we form a new graph G' . (3)

Here G' is bipartite graph, because we can color each v_{in} “blue” and each v_{out} “red” without any coloring conflict.

If there is DHC in G , then there is a BDHC in G' . We can replace each node v on the DHC in G into consecutive nodes v_{in} and v_{out} , and (v_{in}, v_{out}) is an edge in G' . Then the new path is a BDHC in G' . (3)

On the other hand, if there is BDHC in G' , then there is a DHC in G . Note that if v_{in} is on the BDHC, v_{out} must be the successive node on the BDHC. Then we can merge each node pair (v_{in}, v_{out}) on the BDHC in G' into node v and form a DHC in G . (3)

In sum, G has DHC if and only if G' has a BDHC. This completes the reduction, and we confirm that the given problem is NP-complete

Alternate solution:

We can also use Undirected Hamiltonian Cycle for the reduction.

Prove that it's NP as before. (3)

Note that you are using Bipartite Undirected Hamiltonian Cycle for reduction. (2)

Given an arbitrary undirected bipartite graph G , convert G to G' so the vertices in G' stay the same, but all undirected edges are converted to directed edges in **both directions**. G' remains bipartite. (3)

If there is an undirected Hamiltonian cycle in G, you can use the corresponding edges(in either direction) to find a Directed Hamiltonian cycle in in G'. (3)

On the other hand, if there is a directed Hamiltonian cycle in G', you could convert the corresponding edges to undirected edges and find the equivalent cycle in G. (3)

Common mistakes:

Major mistake in checking for NP: -2

Checking for edges instead of vertices: -1

Missing the NP check altogether: -3

Mistake in the conversion step: -2

Missing the two-way proof: -4

Missing either side of the proof: -3

Incomplete explanation as to why the proof holds: -2

If you are proposing to reduce a certain NP-Complete problem, but are going with a solution corresponding a different NP-Complete problem, this shows misunderstanding of the concept: -7

If you are proposing to reduce an incorrect (and irrelevant) NP-complete problem: -8

4) 20 pts.

Suppose you want to sell n roses. You need to group the roses into multiple bouquets with different sizes. The value of each bouquet depends on the number of roses you used. In other words, we know that a bouquet of size i will sell for $v[i]$ dollars. Provide an algorithm to find the maximum possible value of the roses by deciding how to group them into different-sized bouquets.

Here is an example:

Input: $n = 5$, $v[1..n] = [2,3,7,8,10]$ (i.e. a bouquet of size 1 is \$2, bouquet of size 2 is \$3, ..., and finally a single bouquet of size $n=5$ is \$10.

Expected Output = 11 (by grouping the roses into bouquets of size 1, 1, and 3)

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

$OPT(j)$ is the maximum value of j roses.

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

$OPT(j) = \text{MAX } (OPT(j), \{OPT(j-i) + v[i]\} \text{ for all sizes } 0 \leq i < j)$

i.If iteration of i from 0 to j is missing => -1 points

ii.If $OPT(j)$ is missing in the MAX term => -1 points

iii.0 points for wrong recursion

c) Using the recurrence formula in part b, write pseudocode to compute the maximum possible value of the n roses. (6 pts)

Make sure you have initial values properly assigned. (2 pts)

```
OPT[0] = 0
OPT[1] = v[1]
For (int j=2; j < n; j++)
    max = 0
    For (int i=1; i < j; i++)
        If (i<=j && max < OPT[j-i]+v[i] )
            max = OPT[j-i]+v[i]
    OPT[j]=max
```

i) Missing one loop => -1 point

ii) Missing two loops => -2 points

iii) If in b) you got x points, you get x points in the pseudo code. Apart from the case where you missed the iteration in b, where you will get x+1 points in this question

iv) 0 points for no pseudocode or any modification from b.

d) Compute the runtime of the algorithm described in part c and state whether your solution runs in polynomial time or not (2 pts)

e) d) $O(n^2)$

5) 10 pts

Give a tight asymptotic upper bound (O notation) on the solution to each of the following recurrences. You need not justify your answers.

$$T(n) = 2T(n/8) + n$$

Solution: ~~$\Theta(n^{1/3} \lg n)$ by Case 2 of the Master Method.~~

By case 3, it is $\Theta(n)$

$$T(n) = T(n/3) + T(n/4) + 5n$$

Solution: Use brute force method

$$\begin{aligned} T(n) &= cn + (7/12) cn + (7/12)^2 cn + \dots \\ &= (n) \end{aligned}$$

Rubric:

No partial marks. 5 points for each correct answer.

6) 16 pts

There are n people and m jobs. You are given a payoff matrix, C , where C_{ij} represents the payoff for assigning person i to do job j . Each applicant has a subset of jobs that he/she is interested in. Each job opening can only accept one applicant and a job applicant can be appointed for only one job. You need to find an assignment of jobs to applicants that maximizes the total payoff of your assignment. Write an **Integer Linear Program** (with discrete variables) to solve this problem.

Solution

We are looking for a perfect matching on a bipartite graph of the minimal cost.

Let $x_{ij} = \begin{cases} 1, & \text{if edge}(i,j) \text{ is in matching} \\ 0, & \text{otherwise} \end{cases}$

Objective: $\sum_{\substack{1 \leq i \leq m \\ 1 \leq j \leq n}} c_{ij} \cdot x_{ij}$

Subject to:

1. $\sum_j x_{ij} \leq 1$, where $i = 1, 2, \dots, n$
2. $\sum_i x_{ij} \leq 1$, where $j = 1, 2, \dots, n$
3. $x_{ij} \in \{0,1\}$, $\forall i, j$
4. $x_{ij} = 0$, for all jobs j that applicant i is not interested in.

Rubric:

- 4 if objective function is incorrect.
- 6 if you mention equal to instead of less than equal to sign for constraints 1 and 2
- 2 if constraint 3. is not mentioned, but only is defined to take values 0 or 1.
- 2 if discrete variable is not defined.
- 0 points awarded if you use a max-flow, min-cut formulation

7) 12 pts

The Maximum Acyclic Subgraph is stated as follows: Given a directed graph find an acyclic subgraph of that contains as many arcs (i.e. directed edges) as possible.

Give a 2-approximation algorithm for this problem.

Hint: Consider using an arbitrary ordering of the vertices in G.

Solution:

1. Pick an arbitrary vertex ordering. This partitions the arcs into two acyclic subgraphs and . Here A_1 is the set of arcs where $i < j$ and A_2 is the set of arcs where $i > j$. Thus at least one of A_1 or A_2 contains half the arcs of G . The maximum acyclic subgraph contains at most the total number of arcs in G , so we have a factor 2-approximation algorithm (12)
2. Divide the set of nodes into two nonempty sets, A and B . Consider the set of edges from A to B and the set of edges from B to A . Throw out the edges from the smaller set and keep the edges from the larger set (break ties arbitrarily). Recursively perform the algorithm on A and B individually (12)

Rubric:

If you are only considering the forward or backward edges (-2)

If you start with one node and add the forward/backward edges to/from neighbors, your answer won't work for disconnected graphs (-2)

If your final answer is not 0.5 approximation (-6)

If your answer is not acyclic (-6)

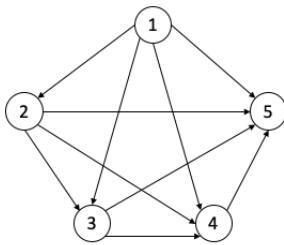
If your algorithm is vague and not implementable (for example if you said, "we find max independent set", or "we find the topological order") (0)

If your algorithm has conceptual flaws (for example if you are adding an edge to the sub graph and also deleting it from the sub graph) (0 pts)

Some of the answers that do not work:

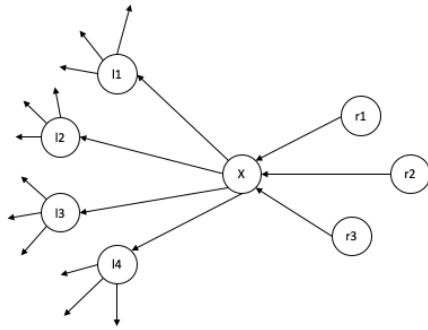
1. If your final answer is a tree (a DFS or a BFS or any other kinds of trees):

You can imagine an acyclic complete directed graph such as below. Your algorithm should return at least half of the edges ($n * (n-1) / 4$) but a tree can have $n-1$ edges. Thus it is not 0.5 approximation (6 pts).



2. If you start adding/removing edges to/from the graph, until it is acyclic. Your algorithm can choose an edge that is repeated in many cycles and end up removing all the other edges in those cycles.

You can try your method on the following graph (All $l(i)$ nodes are connected to all $r(j)$ nodes). If your algorithm chooses all the $(r(j), X)$ and $(X, l(i))$ edges (7 edges), you cannot add any $(l(i), r(j))$ edges (12 edges). Your final answer (7) is less than half of the best answer ($16/2 = 8$) so it is not 0.5 approximation.



CS570 Spring 2018: Analysis of Algorithms Exam III

	Points		Points
Problem 1	20	Problem 5	15
Problem 2	15	Problem 6	10
Problem 3	15	Problem 7	10
Problem 4	15		
Total: 100 Points			

Instructions:

1. This is a 2-hr exam. Closed book and notes
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.

1) 20 pts

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

[TRUE]

To prove that a problem X is NP-hard, it is sufficient to prove that SAT is polynomial time reducible to X.

[FALSE]

If a problem Y is polynomial time reducible to X, then a problem X is polynomial time reducible to Y.

[TRUE]

Every problem in NP can be solved in polynomial time by a nondeterministic Turing machine.

[TRUE]

Suppose that a divide and conquer algorithm reduces an instance of size n into 4 instances of size $n/5$ and spends $\Theta(n)$ time in the conquer steps. The algorithm runs in $\Theta(n)$ time.

[FALSE]

A linear program with all integer coefficients and constants must have an integer optimum solution.

[FALSE]

Let M be a spanning tree of a weighted graph $G=(V, E)$. The path in M between any two vertices must be a shortest path in G.

[TRUE]

A linear program can have an infinite number of optimal solutions.

[TRUE]

Suppose that a Las Vegas algorithm has expected running time $\Theta(n)$ on inputs of size n . Then there may still be an input on which it runs in time $\Omega(n^2)$.

[FALSE]

The total amortized cost of a sequence of n operations gives a lower bound on the total actual cost of the sequence.

[FALSE]

The maximum flow problem can be efficiently solved by dynamic programming.

2) 15 pts

Consider a uniform hash function h that evenly distributes n integer keys $\{0, 1, 2, \dots, n-1\}$ among m buckets, where $m < n$. Several keys may be mapped into the same bucket. The uniform distribution formally means that $\Pr(h(x)=h(y))=1/m$ for any $x, y \in \{0, 1, 2, \dots, n-1\}$. What is the expected number of total collisions? Namely how many distinct keys x and y do we have such that $h(x) = h(y)$?

Solution:

Let the indicator variable X_{ij} be 1 if $h(k_i) = h(k_j)$ and 0 otherwise for all $i \neq j$. Then let X be a random variable representing the total number of collisions. It can be calculated as the sum of all the X_{ij} 's: $X = \sum_{i < j} X_{ij}$

Now we take the expectation and use linearity of expectation to get:

$$E[X] = E \left[\sum_{i < j} X_{ij} \right] = \sum_{i < j} E[X_{ij}] = \sum_{i < j} P(h(k_i) = h(k_j)) = \sum_{i < j} \frac{1}{m} = \frac{n(n-1)}{2m}$$

Rubrics:

Describe choosing 2 keys from n keys: 10 pts.

Describe $\sum_{i < j} P(h(k_i) = h(k_j))$: 10 pts.

Common mistakes:

1. n/m : The mistake is that it thinks the expected number of collisions for each key is $1/m$. 7pts.
2. $2n/m$: Similar to 1. 7pts.
3. $(n-1)/m$: Similar to 1. 7pts.
4. $(m+1)/2$: Basically, no clue. 3 pts.
5. $n(n-1)/m$: Duplicates. 13 pts.
6. n^2/m : duplicated. 13 pts
7. $(1-(1-1/m)^{(n-1)}) * n$: wrong approach. 5 pts.
8. $n-m$: wrong approach. 3pts.
9. $\frac{1}{m} \sum_{i=0}^{n-1} i$: 15 pts
10. $n(n-1)/2$: The mistake is that it does not consider collision probability. 10 pts.
11. $N(n+1)/2m$: minor mistake. 13 pts.

3) 15 pts.

There are 4 production plants for making cars. Each plant works a little differently in terms of labor needed, materials, and pollution produced per car:

	Labor	Materials	Pollution
Plant 1	2	3	15
Plant 2	3	4	10
Plant 3	4	5	9
Plant 4	5	6	7

The goal is to maximize the number of cars produced under the following constraints:

- There are at most 3300 hours of labor
- There are at most 4000 units of material available.
- The level of pollution should not exceed 12000 units.
- Plant 3 must produce at least 400 cars.

Formulate a linear programming problem, using minimal number of variables, to solve the above task of maximizing the number of cars.

Solution:

We need four variables, to formulate the LP problem: x_1, x_2, x_3, x_4 , where x_i denotes the number of cars at plant-i.

Maximize $x_1 + x_2 + x_3 + x_4$

s.t.

$$x_i \geq 0 \text{ for all } i$$

$$x_3 \geq 400$$

$$2x_1 + 3x_2 + 4x_3 + 5x_4 \leq 3300$$

$$3x_1 + 4x_2 + 5x_3 + 6x_4 \leq 4000$$

$$15x_1 + 10x_2 + 9x_3 + 7x_4 \leq 12000$$

Rubric:

- Identifying 4 variables (2 pts)
- Correct objective function (3 pts)
- Each condition (2 pts)

4) 15 pts.

Given an undirected connected graph $G = (V, E)$ in which a certain number of tokens $t(v) \geq 1$ placed on each vertex v . You will now play the following game. You pick a vertex u that contains at least two tokens, remove two tokens from u and add one token to any one of adjacent vertices. The objective of the game is to perform a sequence of moves such that you are left with exactly one token in the whole graph. You are not allowed to pick a vertex with 0 or 1 token. Prove that the problem of finding such a sequence of moves is NP-complete by reduction from Hamiltonian Path.

Solution:

Construction: given a HP in G , we construct G' as follows. Traverse a HP in G and placed 2 tokens on the starting vertex and one token on each other vertex in the path.

Claim: G has a HP iff G' has a winning sequence.

->) by construction before the last move we will end up with a single vertex having two tokens on it. Making the last move, we will have exactly one token on the board.

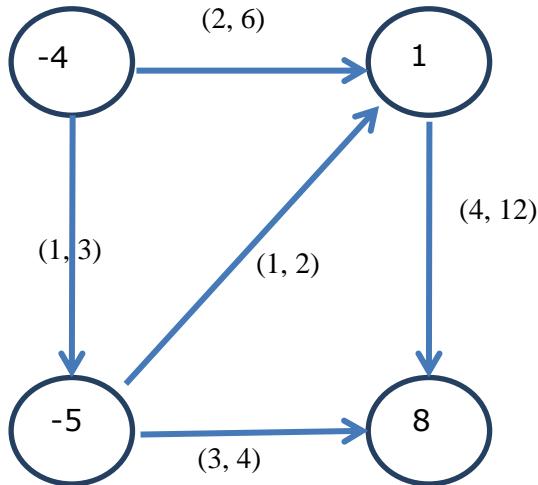
<-) since there is only one vertex with 2 tokens, we will start right there playing the game. Each next move is forced. When we finish the game, we get a sequence moves which represents a HP.

Rubrics:

- Didn't prove it is in NP: -5
- Didn't prove it is NP-hard: -10
- Assigned two tokens on one random vertex instead of the starting vertex of Hamiltonian path: -2 (Since it is a reduction from Hamiltonian path, not from Hamiltonian cycle, 2 tokens should be assigned on the starting vertex)
- Assigned wrong number of tokens on one vertex: -3
- Assigned wrong number of tokens on two vertices: -6
- Assigned wrong number of tokens on three or more vertices (considered as not a valid reduction from Hamiltonian path): -7
- Not a valid reduction from Hamiltonian path: -7

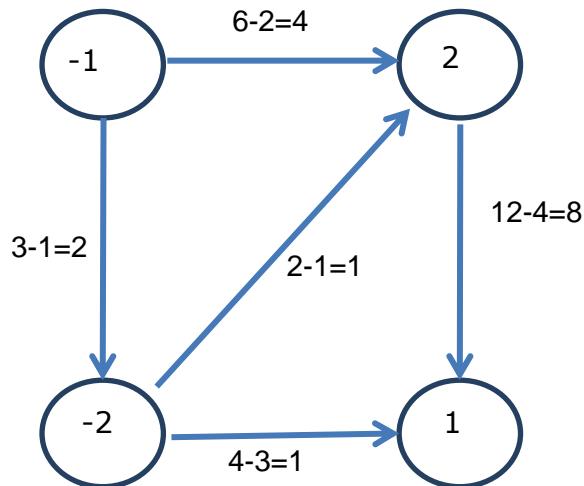
5) 15 pts.

In the network 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.



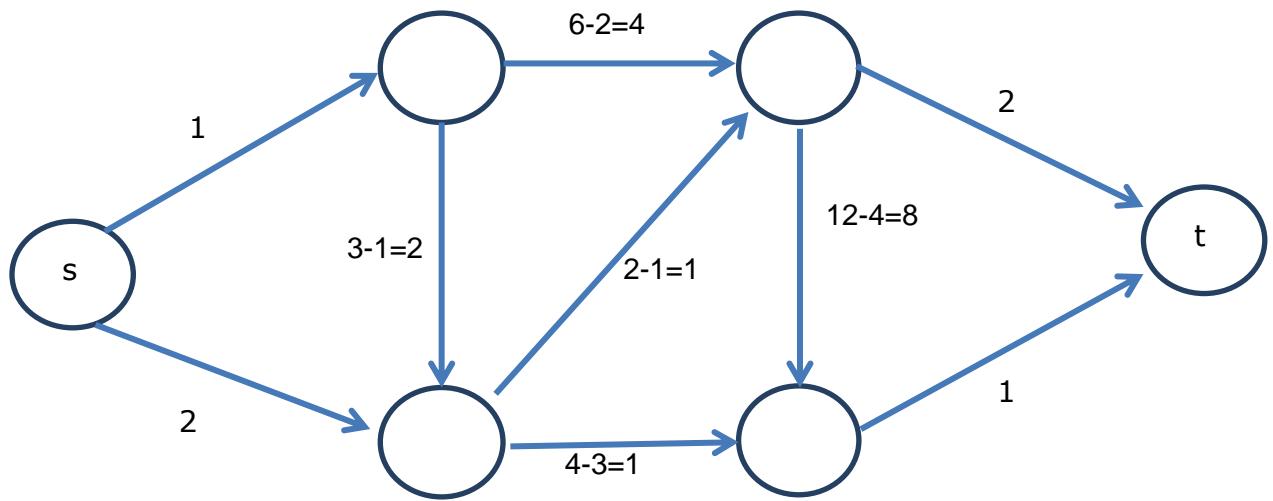
- a) Turn the circulation with lower bounds problem into a circulation problem without lower bounds (6 pts)

- **Each wrong number for nodes: -1**
- **Each wrong number for edges: -0.5**



b) Turn the circulation with demands problem into the max-flow problem (5 pts)

- **s and t nodes on right places: each has 0.5 point**
- **directions of edges from s and t: each direction 0.5 point**
- **values of edges from s and t: each value 0.5 point**

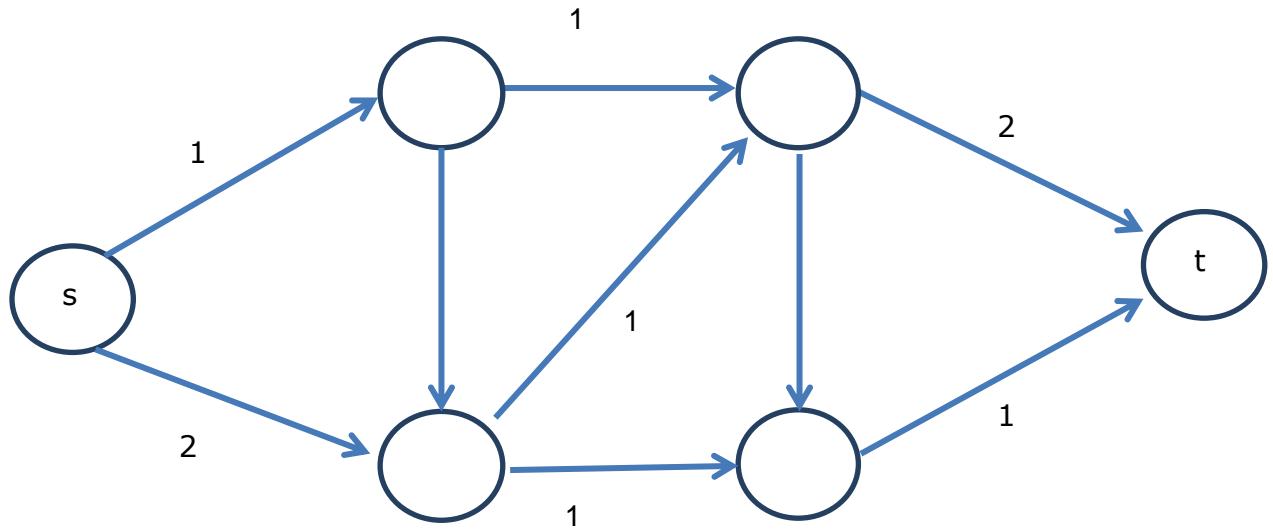


c) Does a feasible circulation exist? Explain your answer. (4 pts)

Solution.

Yes. There is a s-t flow $|f|=2+1=1+2=3$. It follows, there is a feasible circulation.

- **Explanation: 2 points.**
- **Correct Graph/flow: 2 points.**



6) 10 pts.

We wish to determine the most efficient way to deliver power to a network of cities.

Initially, we have n cities that are all connected. It costs $c_i \geq 0$ to open up a power plant at city i . It costs $r_{ij} \geq 0$ to build a cable between cities i and j . A city is said to have power if either it has a power plant, or it is connected by a series of cables to some other city with a power plant (in other words, the cables act as undirected edges). Devise an efficient algorithm for finding the *minimum cost* to power all the cities.

Solution

Consider a graph with the cities at the vertices and with weight r_{ij} on the (undirected) edge between city i and city j . Add a new vertex s and add an edge with weight c_i from s to the i th city, for each i . The minimum spanning tree on this graph gives the best solution (power stations need to be opened at those cities i for which (s,i) is part of the MST.)

Common Mistakes:

- Dynamic Programming: There does not exist a predefined order in which the nodes/edges will be processed and because of this you cannot define the problem based on smaller sub-problems and hence, none of the solutions which were based on Dynamic Programming were correct.
- Max-Flow/Min Cut: It was a surprise to see many ran a Max Flow algorithm to minimize the overall cost of edges. The minimum cut in a network has nothing to do with minimizing flow!! Furthermore, in this problem we want connectivity in the graph and edges on the min cut don't guarantee any level of connectivity.
- ILP: I can't recall anybody correctly formalizing the problem as in ILP. However, even a correct formulation is going to be NP-Hard and far from efficient. A maximum of 3 points was given to a CORRECT ILP solution.
- Dijkstra's algorithm is for finding the shortest path between a single point and every other node in the graph. The resulting Tree from running Dijkstra, is not necessarily an MST so it's not the correct approach to solve this problem.

7) 10 pts.

We want to break up the graph $G = (V, E)$ into two disjoint sets of vertices $V=A \cap B$, such that the number of edges between two partitions is as large as possible. This problem is called a max-cut problem. Consider the following algorithm:

- Start with an arbitrary cut C .
- While there exists a vertex v such that moving v from A to B increases the number of edges crossing cut C , move v to B and update C .

Prove that the algorithm is a 2-approximation.

Hint: after algorithm termination the number of edges in each of two partitions A and B cannot exceed the number of edges on the cut C .

Solution

Let $w(u,v) = 1$, there exists an edge between u and v , and 0 otherwise. Let

$$W = \sum_{(u,v) \in E} w(u,v)$$

By construction, for any node $u \in A$:

$$\sum_{v \in A} w(u, v) \leq \sum_{v \in B} w(u, v)$$

Here, the left hand side is all edges in partition A . The RHS – the crossing edges. If we sum up the above inequality for all $u \in A$, the LHS will contain the twice number of edges in A .

$$2 \sum_{(u,v) \in A} w(u, v) \leq \sum_{u \in A, v \in B} w(u, v) = C$$

We do the same for any node in B :

$$2 \sum_{(u,v) \in B} w(u, v) \leq \sum_{u \in A, v \in B} w(u, v) = C$$

Add them up

$$\sum_{(u,v) \in A} w(u, v) + \sum_{(u,v) \in B} w(u, v) \leq C$$

Next we add edges on the cut C to both sides.

$$\sum_{(u,v) \in A} w(u, v) + \sum_{(u,v) \in B} w(u, v) + \sum_{u \in A, v \in B} w(u, v) = W \leq 2C$$

Clearly the optimal solution $\text{OPT} \leq W$, thus $\text{OPT} \leq W \leq 2C$. It follows, $\text{OPT}/C \leq 2$.

A: # of edges within partition A
B: # of edges within partition B
C: # of edges between partition A and B

Correct direction

inCorrect direction

Prove that $A + B \leq C$

Prove that $A \leq C$ and $B \leq C$
Correct but useless, at most 1 points

Stage 1. 6 points

Prove that $\text{OPT} \leq 2*C$

Prove that $\text{OPT} \leq 3*C$
At most 2 points, if relations between A, B, C, E are used in reasonable way.

Stage 2. 4 points

- The solution can be divided into two parts. **Part 1** is worth 6 points, and **part 2** is worth 4 points.
 - Proving the hint, i.e. "numbers of edges in partition A and partition B cannot exceed the number of edges on the cut C". It is most important and challenging part of this problem.
 - Prove the algorithm is a 1/2-approximation.
- About part 1, here are two acceptable solutions and some solutions not acceptable:
 - If your solution is same to the official solution, using inequalities to rigorously prove the hint, it is perfect, **6 points**.
 - Here is a weaker version. A the end of algorithm, for any node u in the graph, without loss of generality assume it is in part A, the number of edges connecting it to nodes in part B (i.e. edges in the cut) is greater than or equal to the number of edges connecting it to other nodes in part A. So, the total number of edges in the cut is greater than or equal to the total number of edges within each part. This solution did not consider the important fact that both inter-partition and intra-partition edges are counted twice. **3 ~5 points according to the quality of statements.**
 - One incorrect proof: some answers declare that C is increasing and $A + B$ is decreasing when calling steps 2 of the algorithm. This is a correct but useless statement. **0 points**
 - YOU MUST PROVE THE HINT. SIMPLY STATING OR REPHRASING IT GET 0 POINTS IN THIS STEP.** Here is one example that cannot get credits: "According to the algorithm, we move one vertex from A to B if it can increase the number of edges in cut C. Then the number of edges in C is more than edges in partition A and partition B." **0 points**
 - "If move v from A to B can increase the number of edges cross cut C, it means v connect to more vertex in A than in B". It is rephrasing the algorithm, and failed to get the key point. If the above statement is changed to "If move v from A to B can increase the number of edges cross cut C, it means v connect to more vertex in different partition than in the same partition", it will get at least **3 points**.
- About part 2, the correct answer should use relation between OPT, E, A, B, C. Missing key inequalities like $\text{OPT} \leq E$, will lead to loss of grades.
- If the answer is incorrect, you may get some points from some steps that make sense. If the answer is on the incorrect direction in the above diagram:
 - Many students attempted to PROVE the $\text{edges_in_partition_A} \leq C$, and $\text{edges_in_partition_B} \leq C$, but not proving $\text{edges_in_partition_A} + \text{edges_in_partition_B} \leq C$. This is trivial according to step 2 of algorithm, and is not useful for proving the final statement. This will be treated as "correct but useless/irrelevant statements". **AT MOST 1 point.**
 - From above inequalities, $\text{OPT} \leq 3*C$ is a correct conclusion. It is different from the question, but since the logics in this stage is same, you can get at most **2 points out of 4**.

CS570
Analysis of Algorithms
Fall 2007
Exam I

Name: _____
Student ID: _____

	Maximum	Received
Problem 1	20	
Problem 2	20	
Problem 3	12	
Problem 4	12	
Problem 5	12	
Problem 6	12	
Problem 7	12	

Note: The exam is closed book closed notes.

1) 20 pts

Mark the following statements as **TRUE**, **FALSE**. No need to provide any justification.

[**TRUE/FALSE**]

A greedy algorithm is any algorithm that follows the heuristic of making the locally optimum choice at each stage with the hope of finding the global optimum.

T

[**TRUE/FALSE**]

BFS can be used to find the shortest path between any two nodes in a weighted graph.
F

[**TRUE/FALSE**]

DFS can be used to find the shortest path between any two nodes in a non-weighted graph.
F

[**TRUE/FALSE**]

BFS can be used to test whether a graph is bipartite

T

[**TRUE/FALSE**]

If T is a spanning tree of G and e an edge in G which is not in T, then the graph T+e has a unique cycle.

T

[**TRUE/FALSE**]

Let T be a spanning tree of a graph G and e an edge of G which is not in T. For any edge f that is on a cycle in graph T+e, the graph T + e – f is a spanning tree.

T

[**TRUE/FALSE**]

Given a graph $G(V,E)$ with distinct costs on edges and a set $S \subseteq V$, let (u, v) be an edge such that (u, v) is the minimum cost edge between any vertex in S and any vertex in $V-S$. Then, the minimum spanning tree of G must include the edge (u, v) .

T

[**TRUE/FALSE**]

If f , g , and h are positive increasing functions with f in $O(h)$ and g in $\Omega(h)$, then the function $f+g$ must be in $\Theta(h)$.

F

[**TRUE/FALSE**]

If a divide and conquer algorithm divides the problem is half at every step, the $\log(n)$ factor related to the depth of the recursion tree will cause the algorithm to have a lower bound of $\Omega(n \log(n))$

F

[**TRUE/FALSE**]

Suppose that in an instance of the original Stable Marriage problem with n couples, there is a man M who is last on every woman's list and a woman W who is last on every man's list. If the Gale-Shapley algorithm is run on this instance, then M and W will be paired with each other.

T

2) 20 pts

- a) Arrange the following in the order of big oh $4n^2$, $\log_2(n)$, $20n$, 2 , $\log_3(n)$, n^n , 3^n , $n\log(n)$, $2n$, 2^{n+1} , $\log(n!)$

$$2 < \log_2(n) < \log_3(n) < 2n < 20n < \log(n!) < n\log(n) < 4n^2 < 2^{n+1} < 3^n < n^n$$

b) Find the complexity of the following nested loop

```
sum = 0;  
for (i=0; i<3; i++)  
    for (j=0; j<n; j++)  
        sum++;
```

$$O(n)$$

c) Find the complexity of the following code section

```
for (i=0; i<n; i++) {  
    for (j=0; j<n; j++) {  
        A[i] = random(n);  
    } // assume random() is O(1)  
    sort(A, n); // assume sort() is the fastest general sorting algorithm  
}
```

$$O(n^2)$$

d) Find the complexity of the following function

```
int somefunc(int n) {  
    if (n <= 1)  
        return 1;  
    else  
        return somefunc(n-1) + somefunc(n-1);  
}
```

$$O(2^n)$$

e) Find the runtime $T(n)$ in the following recurrence equation:

$$T(n) = 2T(n/4) + n^{0.51}$$

$$T(n) = O(n^{0.51})$$

3) 12 pts

Suppose we are given an instance of the Shortest Path problem with source vertex s on a directed graph G . Assume that all edges costs are positive and distinct. Let P be a minimum cost path from s to t . Now suppose that we replace each edge cost c_e by its square, c_e^2 , thereby creating a new instance of the problem with the same graph but different costs.

Prove or disprove: P still a minimum-cost $s - t$ path for this new instance.

The statement is FALSE

Consider the example:

Vertices: $V=\{A, B, C, D\}$

Edges: $(A \rightarrow B)=100, (A \rightarrow C)=51, (B \rightarrow D)=1, (C \rightarrow D)=51$

Shortest path from A to D is $A \rightarrow B \rightarrow D$ Path length=101

After squaring this path length become $100^2+1^2=10001$

However, $A \rightarrow C \rightarrow D$ has path length $51^2+51^2=5202<10001$

Thus $A \rightarrow C \rightarrow D$ become shortest path from A to D

4) 12 pts

Consider a long country road with houses scattered very sparsely along it. You want to place cell phone base stations at certain points along the road, so that every house is within four miles of one of the base stations. Give an efficient algorithm that achieves this goal, using as few base stations as possible.

Greedy approach

- 1) Start from the beginning of the road
- 2) Find the first uncovered house on the road
- 3) If there is no such a house, terminate this algorithm; otherwise, go to next line
- 4) Locate a base station at 4 miles away after you find this house along the road
- 5) Go to 2)

Proof:

Let the number of base stations used to cover the first n houses in our algorithm be $G(n)$. For any other policy, denote the number of base stations used to cover the first n houses $P(n)$. Optimality of our algorithm can be shown if $G(n) \leq P(n)$ for $n=1, 2, 3, \dots$. We prove this by induction.

It is obvious that $G(1) \leq P(1)$.

Given $G(n-1) \leq P(n-1)$, let's consider $G(n)$ and $P(n)$.

If the n -th house is already covered by $G(n-1)$ base stations, then

$G(n) = G(n-1) \leq P(n-1) \leq P(n)$. This completes the proof.

If the n -th house has not been covered by $G(n-1)$ base stations, then $G(n) = G(n-1) + 1$.

If $G(n-1) < P(n-1)$, then we have $P(n) \geq P(n-1) \geq G(n-1) + 1 = G(n)$, which completes the proof.

Otherwise $G(n-1) = P(n-1)$. In this case the n -th house must have note been covered by $P(n-1)$ base stations in this other policy because we always make sure that our policy covers the longest distance from the beginning to the n -th base station. Thus $P(n) = P(n-1) + 1 \geq G(n)$. This completes the proof.

Running time $O(n)$, where n is the number of houses

5) 12 pts

Given a sorted array A of distinct integers, describe an algorithm that finds i such that $A[i] = i$, if such an i exists. Your algorithm must have a complexity better than $O(n)$.

```
Function(A,n)
{
    i=floor(n/2)
    if A[i]==i
        return TRUE
    if (n==1)&&(A[i]!=i)
        return FALSE
    if A[i]<i
        return Function(A[i+1:n], n-i)
    if A[i]>i
        return Function(A[1:i], i)
}
```

Proof:

The algorithm is based on Divide and Conquer. Every time we break the array into two halves. If the middle element i satisfy $A[i] \leq j$, we can see that for all $j < i$, $A[j] \leq j$. This is because A is a sorted array of DISTINCT integers. To see this we note that

$A[j+1] - A[j] \geq 1$ for all j . Thus in the next round of search we only need to focus on $A[i+1:n]$

Likewise, if $A[i] > i$ we only need to search $A[1:i]$ in the next round.

For complexity $T(n)=T(n/2)+O(1)$

Thus $T(n)=O(\log n)$

6) 12 pts

Consider a max-heap implemented using pointers rather than an array, so that the root has pointers to two smaller heaps, all of whose elements are smaller than the root's element. Give an algorithm to find the smallest element in the heap, and argue that your algorithm always runs in $O(n)$ time on a heap with n elements.

Min=value at the root

Run a BFS or DFG through the heap. Every time a new node is visited compare its value with Min, if it is smaller then Min=this value

Every node visited only once, thus $O(n)$

7) 12 pts

Find all possible stable matchings for the following table of preferences. The women are A,B,C,D and the men are a, b, c, d :

	A	B	C
a	(1,3)	(2,2)	(3,1)
b	(3,1)	(1,3)	(2,2)
c	(2,2)	(3,1)	(1,3)

The content of the box (a,A), which is (1,3), means that man a ranks woman A as his first choice and woman A ranks man a as her third choice.

First we can see that a, b, c all rank D as last choice, A, B, C all rank d as last choice.

Thus d only matches with D, otherwise the man matching with D and the woman matching with d will prefer each other than d and D.

Now we only look at a, b, c and A, B, C we can see that all matches works among them. Thus stable matches include all matching among a, b, c and A, B, C and then (d, D).

Additional Space

Additional Space

Name(last, first): _____

CS570
Analysis of Algorithms
Fall 2005
Midterm Exam

Name: _____
Student ID: _____

	Maximum	Received
Problem 1	14	
Problem 2	2	
Problem 3	5	
Problem 4	10	
Problem 5	10	
Problem 6	15	
Problem 7	20	
Problem 8	15	
Problem 9	9	

Name(last, first): _____

1. 14 pts

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

[TRUE/FALSE]

A dynamic programming algorithm tames the complexity by making sure that no subproblem is solved more than once.

[TRUE/FALSE]

The memoization approach in dynamic programming has the disadvantage that sometimes one may solve subproblems that are not really needed.

[TRUE/FALSE]

A greedy algorithm finds an optimal solution by making a sequence of choices and at each decision point in the algorithm, the choice that seems best at the moment is chosen.

[TRUE/FALSE]

If a problem can be solved correctly using the greedy strategy, there will only be one greedy choice (such as “choose the object with highest value to weight ratio”) for that problem that leads to the optimal solution.

[TRUE/FALSE]

Whereas there could be many optimal solutions to a combinatorial optimization problem, the value associated with them will be unique.

[TRUE/FALSE]

Let G be a directed weighted graph, and let u, v be two vertices. Then a shortest path from u to v remains a shortest path when 1 is added to every edge weight.

[TRUE/FALSE]

Given a connected, undirected graph G with distinct edge weights, then the edge e with the second smallest weight is included in the minimum spanning tree.

2. 2 pts

In our first lecture this semester, which of the following techniques were used to solve the Stable Matching problem? Circle all that may apply.

A- Greedy

B- Dynamic Programming

C- Divide and Conquer

Name(last, first): _____

3. 5 pts

A divide and conquer algorithm is based on the following general approach:

- Divide the problem into 4 subproblems whose size is one third the size of the problem at hand. This is done in $O(\lg n)$
- Solve the subproblems recursively
- Merge the solution to subproblems in linear time with respect to the problem size at hand

What is the complexity of this algorithm?

4. 10 pts

Indicate for each pair of expressions (A,B) in the table below, whether A is **O**, **Ω** , or **Θ** of B. Assume that k and c are positive constants.

A	B	O	Ω	Θ
$(\lg n)^k$	Cn			
2^n	$2^{(n+1)}$			
$2^n n^k$	$2^n n^{2k}$			

Name(last, first): _____

5. 10 pts

The array A below holds a max-heap. What will be the order of elements in array A after a new entry with value 19 is inserted into this heap? Show all your work.

$$A = \{16, 14, 10, 8, 7, 9, 3, 2, 4, 1\}$$

Name(last, first): _____

6. 15 pts

Consider a max-heap H and a given number X. We want to find whether X is larger than the k-th largest element in the list. Design an O(k) time algorithm to do this.

Note: you do not need to prove your algorithm but you need to prove the complexity.

Name(last, first): _____

7. 20 pts total

Consider the following arcade game where the player starts at the lower left corner of an n by n grid to reach the top right corner. At each step the player can either jump to the right (x -axis) or jump up (y -axis). Depending on his/her energy level the player can jump either 1 or 2 squares. If his/her energy level is higher than E he/she can jump 2 squares, otherwise only one square. When landing on square (i,j) the player may gain or lose energy equal to $|X_{ij}|$. Positive X_{ij} indicates energy gain. In addition, every time the player jumps 2 squares he/she loses $E/2$ units of energy.

- a) Present a polynomial time solution to find the highest energy level the player can end the game with. (15 pts)

Name(last, first): _____

- b) Describe how you could build the path that leads to the optimal solution. (5 pts)

Name(last, first): _____

8. 15 pts

T is a spanning tree on an undirected graph $G=(V,E)$. Edge costs in G are NOT guaranteed to be unique. Prove or disprove the following:
If every edge in T belongs to SOME minimum cost spanning tree in G, then T is itself a minimum cost spanning tree.

Name(last, first): _____

9. 9 pts

We have shown in class that the Integer Knapsack problem can be solved in $O(nW)$ where n is the number of items and W is the capacity of the knapsack. The Fractional Knapsack problem is less restrictive in that it allows fractions of items to be placed in the knapsack. Solve the Fractional Knapsack problem (have n objects, weight of object $i = W_i$, value of object $i = V_i$, weight capacity of knapsack= W , Objective: Fill up knapsack with objects to maximize total value of objects in knapsack)

Name(last, first): _____

Additional space.

Name(last, first): _____

Additional space.

CS570 MIDTERM SOLUTION

Q 1:

- 1. True
- 2. False
- 3. True
- 4. False
- 5. True
- 6. False
- 7. True

Grading Policy : 2 points and all or none for each question.

Q 2: The answer is A

Grading Policy : Any other solution is wrong and get zero

Q 3:

$T(n) = 4T(\frac{n}{3}) + \lg n + cn$ by master theorem, $T(n)$ is $O(n^{\log_3 4})$

Grading Policy : 2 points for the correct recursive relation. 3 points for the correct result $O(n^{\log_3 4})$. If the student provide the correct result but doesn't give the recursive relation, get 4 points.

Q 4: Click on (1, 1), (2, 1), (2, 2), (2, 3), (3, 1) and write on others.

Here (i, j) means the $i - th$ row and $j - th$ column in the blank form.

Grading Policy : 1 points for each entry in the blank form

Q 5: The process of a new entry with value 19 inserted is as follows:

(1) $A=\{16,14,10,8,7,9,3,2,4,1,19\}$

(2) $A=\{16,14,10,8,19,9,3,2,4,1,7\}$

(3) $A=\{16,19,10,8,14,9,3,2,4,1,7\}$

(4) $A=\{19,16,10,8,14,9,3,2,4,1,7\}$

Grading Policy : It's OK if represent A by a graph. 2.5 points for each step.

Q 6:

Algorithm: The max head is a tree and we start from the root doing the BFS(or DFS) search. If the incident node is bigger than x , we add

it into the queue. By doing this if we already find k nodes then x is smaller than at least k nodes in the heap, otherwise, x is bigger than $k - th$ biggest node in the heap.

running time: During the BFS(or DFS), we traverse at most k nodes. Thus the running time is $O(k)$

Grading Policy : 9 points for the algorithm and 6 points for the correct running time. A typical mistake is that many students assumed that the max heap was well sorted and just pick the k -th number in the heap. In this case they get 3 points at most

Q 7:

a) The recursive relation is as follows:

$$OPT(m, n) = \max \begin{cases} OPT(m, n - 1) + X_{mn} \\ OPT(m, n - 2) + X_{mn} - E/2, & \text{if } OPT(m, n - 2) > \frac{E}{2} \\ OPT(m - 1, n) + X_{mn} \\ OPT(m - 2, n) + X_{mn} - E/2, & \text{if } OPT(m - 2, n) > \frac{E}{2} \end{cases}$$

And the boundary condition is $OPT(1, 1) = E_0$. Clearly to find the value of $OPT(n, n)$, we need to fill an $n \times n$ matrix. It takes constant time to do the addition and compare the 4 numbers in the recursive relation. Hence the running time of our algorithm is $O(n^2)$

b) After the construction of the $n \times n$ table in a), we start right at the upper right corner (n, n) . We compare the value with the 4 previous value in the recursive relation mentioned in a). If we get $OPT(n, n)$ from the previous position (i, j) in the recursive relation, store the position (i, j) in the path and repeat the process at (i, j) until we get to $(1, 1)$

Grading Policy : For part a), 10 points for the correct recursive relation and 5 points for the correct running time. If the student didn't solve this question by dynamic programming, he gets at most 4 points as the partial credit.

For part b), make sure they know the idea behind the problem.

Q 8: False. The counter example is as follows: Consider the graph $(a, b), (b, c), (a, c)$ where $w(a, b) = 4, w(b, c) = 4, w(c, a) = 2$. Clearly (a, b) is in the MST $((a, b), (c, a))$ and (b, c) is in the MST $((b, c), (c, a))$ but $((a, b), (b, c))$ is not MST of the graph.

Grading Policy : If the answer is "True", at most get 5 points for the whole question. If the answer is "False" but didn't give a counter example, get 9 points. If the answer is "false" and give a correct counter example, but didn't point out the spanning tree in which every edge is in some MST but the spanning tree is not MST, get 12 points.

Q 9: First we calculate $c_i = V_i/W_i$ for $i = 1, 2, \dots, n$, then we sort the objects by decreasing c_i . After that we follows the greedy strategy of always taking as much as possible of the objects remaining which has highest c_i until the knapsack is full. Clearly calculating c_i takes $O(n)$ and sorting c_i takes $O(n \log n)$. Hence the running time of our algorithm is $O(n) + O(n \log n)$, that is, $O(n \log n)$. We prove our algorithm by contradiction. If $S = \{O_1, \dots, O_m\}$ is the objects the optimal solution with weight w_1, \dots, w_m . The total value in the knapsack is $\sum_{i=1}^m (w_i/W_i)V_i$. Assume there is i, j (without loss of generality assume that $i < j$) such that $V_i/W_i > V_j/W_j$ and there is V_i with weight w'_i left outside the knapsack. Then replacing $\min\{w'_i, w_j\}$ weight of O_j with $\min\{w'_i, w_j\}$ weight of O_i we get a solution with total value $\sum_{k=1}^m w_k/W_k V_k + \min\{w'_i, w_j\}/W_i V_i - \min\{w'_i, w_j\}/W_j V_j$. $w'_i > 0, w_j > 0$ and thus $\min\{w'_i, w_j\} > 0$. Note that $V_i/W_i > V_j/W_j$, hence $\sum_{k=1}^m (w_k/W_k)V_k + (\min\{w'_i, w_j\}/W_i)V_i - (\min\{w'_i, w_j\}/W_j)V_j > \sum_{k=1}^m (w_k/W_k)V_k$. we get a better solution than the optimal solution. Contradiction!

Grading Policy : 5 points for the algorithm, 2 points for the proof and 2 points for the running time.

Dynamic Programming will not work here since the fractions of items are allowed to be placed in the knapsack. If the student tried solving this problem by dynamic programming and both the recursive relation and time complexity are correct, they get 4 points at most.

CS570
Analysis of Algorithms
Fall 2006
Exam 1

Name: _____
Student ID: _____

	Maximum	Received
Problem 1	20	
Problem 2	10	
Problem 3	10	
Problem 4	10	
Problem 5	10	
Problem 6	20	
Problem 7	20	

Note: The exam is closed book closed notes.

1) 20 pts

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

[**TRUE/FALSE**] True (By definition it is true)

If $T(n)$ is both $O(f(n))$ and $\Omega(f(n))$, then $T(n)$ is $\Theta(f(n))$.

[**TRUE/FALSE**] True

For a graph G and a node v in that graph, the DFS and BFS trees of G rooted at v always contain the same number of edges

[**TRUE/FALSE**] False (Counter example: Fibonacci Heap)

Complexity of the “Decrease_Key” operation is always $O(\lg n)$ for a priority queue.

[**TRUE/FALSE**] True (See the solution of HW4)

For a graph with distinct edge weights there is a unique MST.

[**TRUE/FALSE**] True (yes and store all the possible solution in a table)

Dynamic programming considers all the possible solutions.

[**TRUE/FALSE**] False(The graph a-b-c-d-a with three edges are 1 and the one left is 4)

Consider an undirected graph $G=(V, E)$ and a shortest path P from s to t in G . Suppose we add one 1 to the cost of each edge in G . P will still remain as a shortest path from s to t .

[**TRUE/FALSE**] True

Consider an undirected graph $G=(V, E)$ and its minimum spanning tree T .

Suppose we add one 1 to the cost of each edge in G . T will still remain as an MST.

[**TRUE/FALSE**] False (Counter example: Shortest Path in a Graph)

Problems solved using dynamic programming cannot be solved thru greedy algorithms.

[**TRUE/FALSE**] False (union-Find data structure is for Kruskal’s algorithm, it is not for the reverse delete. Check the textbook for more details)

The union-Find data structure can be used for an efficient implementation of the reverse delete algorithm to find an MST.

[**TRUE/FALSE**] False (Prim algorithm Vs Kruskal algorithm, return can be different)

While there are different algorithms to find a minimum spanning tree of undirected connected weighted graph G , all of these algorithms produce the same result for a given G .

2) 10 pts

Indicate for each pair of expressions (A,B) in the table below, whether A is **O**, **Ω** , or **Θ** of B. Assume that k and c are positive constants. You can mark each box with Y (yes) and N (no).

A	B	O	Ω	Θ
$n^3 + n^2 + n + c$	n^3	Yes	Yes	Yes
2^n	$2^{(n+k)}$	Yes	Yes	Yes
n^2	$n \cdot 2^{\log(n)}$	No	Yes	No

3) 10 pts

a- What is the minimum and maximum numbers of elements in a heap of height h?

The minimum is $2^{(n-1)}$

The maximum number is $2^n - 1$

b- What is the number of leaves in a heap of size n ?

The smallest integer that no less than $n/2$.

Or

When n is even, the number of leave is $n/2$;

When n is odd; the number of leaver is $(n+1)/2$

c- Is the sequence $< 23, 7, 14, 6, 13, 10, 1, 5, 17, 12 >$ a max-heap? If not, show how to heapify the sequence.

Answer: No. The sequence in heapify is

$< 23, 7, 14, 17, 13, 10, 1, 5, 6, 12 >, < 23, 17, 14, 7, 13, 10, 1, 5, 6, 12 >$

d- Where in a max-heap might the smallest element reside, assuming that all elements are distinct.

The smallest element might reside in any leaves

Grading policy: 2 points for a); 2 points for b); 3 points for c); 3 points for d)

4) 10 pts

Prove or disprove the following:

The shortest path between any two nodes in the minimum spanning tree $T = (V, E')$ of connected weighted undirected graph $G = (V, E)$ is a shortest path between the same two nodes in G . Assume the weights of all edges in G are unique and larger than zero.

False. Consider the graph A-B-C-D-A with weight 1,2,3,4

Grading policy:

- Any one says that the statement is true and try to prove it (of course with wrong proof) may get partial credit up to 3 marks.
- Any one says it is false without correct disproof (counter example) may get partial credit up to 6 marks.
- In Question 5, any one says it is false and give a counter example that has one or more nodes in both G_1 and G_2 may get partial credit up to 6 marks. Because that is a mistake, nodes in G_1 can not exist in G_2 and vice versa.
- Any one says it is false and give correct disproof (counter example) get 10 out of 10.

5) 10 pts

Suppose that you divided a graph $G = (V, E)$ into two sub graphs $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$. And, we can find M_1 which is a MST of G_1 and M_2 which is MST of G_2 . Then, $M_1 \cup M_2 \cup \{\text{minimum weight edge among those connecting two graph } G_1 \text{ and } G_2\}$ always gives MST of G . Prove it or disprove it.

Solution: False.

Counter example: Consider a graph G composed of 4 nodes: A,B,C,D with edge $w(A,B)=1, w(B,C)=1, w(B,D)=1, w(C,D)=2$. Let $V_1=\{A,B\}$ with edge AB, $V_2=\{C,D\}$ with edge CD. Then the weight of $M_1 \cup M_2 \cup$ is 4 while the weight of the MST of G is 3.

Grading policy:

- Any one says that the statement is true and try to prove it (of course with wrong proof) may get partial credit up to 3 marks.
- Any one says it is false without correct disproof (counter example) may get partial credit up to 6 marks.
- In Question 5, any one says it is false and give a counter example that has one or more nodes in both G_1 and G_2 may get partial credit up to 6 marks. Because that is a mistake, nodes in G_1 can not exist in G_2 and vice versa.
- Any one says it is false and give correct disprove (counter example) get 10 out of 10.

6) 20 pts

There are n workers in the factory with heights of p_1, p_2, \dots, p_n , and n working-clothes with height sizes of s_1, s_2, \dots, s_n . The problem is to find best matching strategy such that we minimize the following average differences.

$$\frac{1}{n} \sum |p_i - s_i|$$

Present an algorithm to solve this problem along with its proof of correctness.

Solution: Sort the height of workers in increasing order $p_1 \leq p_2 \leq \dots \leq p_n$

Sort the height size of clothes in increasing order $s_1 \leq s_2 \leq \dots \leq s_n$

Match h_i with s_i is the best matching strategy such that we minimize the following average differences.

$$\frac{1}{n} \sum |p_i - s_i|$$

The algorithm is correct for the problem of minimizing the average difference between the heights of workers and their clothes. The proof is by contradiction. Assume the people and clothes are numbered in increasing order by height. If the greedy algorithm is not optimal, then there is some input $p_1, \dots, p_n, s_1, \dots, s_n$ for which it does not

produce an optimal solution. Let the optimal solution be $T = \{(p_1, s_{(1)}), \dots, (p_n, s_{(n)})\}$, and let the output of the greedy algorithm be $G = \{(p_1, s_1), \dots, (p_n, s_n)\}$. Beginning with p_1 , compare T and G . Let p_i be the first person who is assigned different cloth in G than in T . Let s_j be the pair of cloth assigned to p_i in T . Create solution T' by switching the cloth assignments of p_i and p_j . By the definition of the greedy algorithm, s_i is equal or greater than s_j . The total cost of T' is given by

$$Cost(T') = Cost(T) - \frac{1}{n}(|p_i - s_j| + |p_j - s_i| - |p_i - s_i| - |p_j - s_j|)$$

There are six cases to be considered. For each case, one needs to show that $(|p_i - s_j| + |p_j - s_i| - |p_i - s_i| - |p_j - s_j|) \geq 0$.

Case 1: $p_i \leq p_j \leq s_i \leq s_j$.

$$\begin{aligned} |p_i - s_j| + |p_j - s_i| - |p_i - s_i| - |p_j - s_j| &= \\ (s_j - p_i) + (s_i - p_j) - (s_i - p_i) - (s_j - p_j) &= 0 \end{aligned}$$

Case 2: $p_i \leq s_i \leq p_j \leq s_j$.

$$\begin{aligned} |p_i - s_j| + |p_j - s_i| - |p_i - s_i| - |p_j - s_j| &= \\ (s_j - p_i) + (p_j - s_i) - (s_i - p_i) - (s_j - p_j) &= \\ 2(p_j - s_i) &\geq 0 \end{aligned}$$

Case 3: $p_i \leq s_i \leq s_j \leq p_j$.

$$\begin{aligned} |p_i - s_j| + |p_j - s_i| - |p_i - s_i| - |p_j - s_j| &= \\ (s_j - p_i) + (p_j - s_i) - (s_i - p_i) - (p_j - s_j) &= \\ 2(s_j - s_i) &\geq 0 \end{aligned}$$

Case 4: $s_i \leq s_j \leq p_i \leq p_j$.

$$\begin{aligned} |p_i - s_j| + |p_j - s_i| - |p_i - s_i| - |p_j - s_j| &= \\ (p_i - s_j) + (p_j - s_i) - (p_i - s_i) - (p_j - s_j) &= 0 \end{aligned}$$

Case 5: $s_i \leq p_i \leq s_j \leq p_j$.

$$\begin{aligned} |p_i - s_j| + |p_j - s_i| - |p_i - s_i| - |p_j - s_j| &= \\ (s_j - p_i) + (p_j - s_i) - (p_i - s_i) - (p_j - s_j) &= \\ 2(s_j - p_i) &\geq 0 \end{aligned}$$

Case 6: $s_i \leq p_i \leq p_j \leq s_j$.

$$\begin{aligned} |p_i - s_j| + |p_j - s_i| - |p_i - s_i| - |p_j - s_j| &= \\ (s_j - p_i) + (p_j - s_i) - (p_i - s_i) - (s_j - p_j) &= \\ 2(p_j - p_i) &\geq 0 \end{aligned}$$

Running time: Sorting takes $O(n \log n)$ and hence the running time is $O(n \log n)$

Grading policy: Correct description of algorithm 10 pts

Based on the above correct algorithm, present correct complexity 4pts

Correct proof (all six cases) 6pts

7) 20 pts

Given an unlimited supply of coins of denominations x_1, x_2, \dots, x_n , we wish to make change for a value v , that is, we wish to find a set of coins whose total value is v . This might not be possible: for example, if the denominations are 5 and 10 then we can make change for 15 but not for 12. Give an $O(nv)$ algorithm to determine if it is possible to make change for v using coins of denominations x_1, x_2, \dots, x_n .

Solution: We solve this question by dynamic programming. Denote $\text{OPT}(i)$ as the possibility of paying value i using coins of denominations x_1, x_2, \dots, x_n . Then $\text{OPT}(i)$ is true if and only if we can pay $i - x_1$, or $i - x_2, \dots$ or $i - x_n$ using coins of denominations x_1, x_2, \dots, x_n . In other words,

$$\text{OPT}(i) = \text{OPT}(i - x_1) \vee \text{OPT}(i - x_2) \vee \dots \vee \text{OPT}(i - x_n)$$

Following the recursive relation with initial value $\text{OPT}(x_1) = \dots = \text{OPT}(x_n) = \text{true}$ we compute the value of $\text{OPT}(v)$. If $\text{OPT}(v)$ is true, then we can change for v using coins of denominations x_1, x_2, \dots, x_n , otherwise we can not.

Clearly to compute $\text{OPT}(v)$ we need an array of size v , and each time when we compute $\text{OPT}(i)$, we do n union operations. Hence the running time is $O(nv)$

The following pseudo-code would help you understand the solution:

Data structures: $A[v,n]$: 2-dimensional array with entries T or F;

$\text{OPT}[1..n]$: 1-dimensional array with entries T or F.

```
for (j=1 to v)
    OPT(j) = F;
    for (j=1 to v)
        for (i=1 to n)
            {
                if (j < xi) then A[j,i] = F;
                if (j == xi) then A[j,i] = T;
                if (j > xi) then A[j,i] = OPT(xi) AND OPT(j-xi);
                if (A[j,i] == T) then OPT(j) = T;
            }
    return OPT(v)
```

Additional Space

Additional Space

CS570
Analysis of Algorithms
Fall 2008
Exam I

Name: _____

Student ID: _____

Monday Section Wednesday Section Friday Section

	Maximum	Received
Problem 1	20	
Problem 2	10	
Problem 3	10	
Problem 4	20	
Problem 5	20	
Problem 6	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**] F

Given graph G and a Minimum Spanning Tree T on G , you could find the (weighted) shortest path between arbitrary pair u, v in $V(G)$ using only edges in T .

[**TRUE/FALSE**] F

$V^2 \log V = \theta(E \log E^2)$ whether the graph is dense or sparse.

[**TRUE/FALSE**] T

If DFS and BFS returns different trees, then the original graph is not a tree.

[**TRUE/FALSE**] T

An algorithm with the running time of $n * 2^{\min(n \log n, 10000)}$ runs in polynomial time.

[**TRUE/FALSE**] T

We may need to run Dijkstra's algorithm to compute the shortest path on a directed graph, even if the graph doesn't have a cycle.

[**TRUE/FALSE**] F

Given a graph that contains negative edge weights, we can use Dijkstra's algorithm to find the shortest paths between any two vertexes by first adding a constant weight to all of the edges to eliminate the negative weights.

[**TRUE/FALSE**] F

If $f(n)$ and $g(n)$ are asymptotically positive functions, then $f(n)+g(n) = \theta(\min\{f(n), g(n)\})$.

[**TRUE/FALSE**] F

For a stable matching problem such that m ranks w last and w ranks m last, m and w will never be paired in a stable matching.

[**TRUE/FALSE**] F

Breadth first search is an example of a divide-and-conquer algorithm.

[**TRUE/FALSE**] T

Kruskal's algorithm for finding the MST works with positive and negative edge weights.

2) 10 pts

a) Arrange the following in the increasing order of asymptotic growth. Identify any ties.

$$\lg n^{10}, 3^n, \lg n^{2n}, 3n^2, \lg n^{\lg n}, 10^{\lg n}, n^{\lg n}, n \lg n$$

$$\lg n^{10}, \lg n^{\lg n}, \lg n^{2n}, n \lg n, 3n^2, 10^{\lg n}, n^{\lg n}, 3^n$$

b) Analyze the complexity of the following loops:

```
i- x = 0  
    for i=1 to n  
        x= x + lg n  
    end for
```

O(n)

```
ii- x=0  
    for i=1 to n  
        for j=1 to lg n  
            x = x * lg n  
        endfor  
    endfor
```

O(nlg n)

```
iii- x = 0  
    k = "some constant"  
    for i=1 to max (n, k)  
        x= x + lg n  
    end for
```

O(n)

```
iv- x=0  
    k = "some constant"  
    for i=1 to min(n, k)  
        for j=1 to lg n  
            x = x * lg n  
        endfor  
    endfor
```

O(lgn)

3) 10 pts

- a) For each of the following recurrences, give an expression for the runtime T (n) if the recurrence can be solved with the Master Theorem. Otherwise, indicate that the Master Theorem does not apply.

$$T(n) = T(n/2) + 2^n$$
$$\theta(2^n)$$

$$T(n) = 16T(n/4) + n$$
$$\theta(n^2)$$

$$T(n) = 2T(n/2) + n \log n$$

Master Theorem does not apply.

$$T(n) = 2T(n/2) + \log n$$
$$\theta(n)$$

$$T(n) = 64T(n/8) - n^2 \log n$$

Master Theorem does not apply.

- b) Suppose we have a divide and conquer algorithm which at each step takes time n , and breaks the problem up into $n^{1/2}$ subproblems of size $n^{1/2}$ each. Find the runtime $f(n)$ such that $T(n) = \theta(f(n))$, given the following recurrence. $T(n) = n^{1/2} T(n^{1/2}) + n$

They have to show the full recursion tree.

At the root we spend cn

At the next level we have $n^{1/2}$ nodes each taking $cn^{1/2}$ so again, at this level we spend cn time

Same goes with every other level. There are $\lg n$ levels in the tree because at every we take the square root of n . So the total run time is $\theta(n \lg n)$

4) 20 pts

A key to customer service is to avoid having the customer wait longer than necessary. That's your philosophy anyway when you open a coffee shop shortly after graduation. This philosophy and your CS background have made you wonder about the order in which your server should serve customers whose orders have different levels of complexity. For example, if customer one's order will take 3 minutes while customer two's will take 1 minute, then serving customer one first results in 7 minutes of waiting (3 for customer one and 4 for customer two) while serving customer two first results in only 5 (1 for customer two and 4 for customer one).
(a) Phrase this problem more formally by introducing notation for the service time and precisely define the problem's objective.

[Please refer to solution to Q4](#)

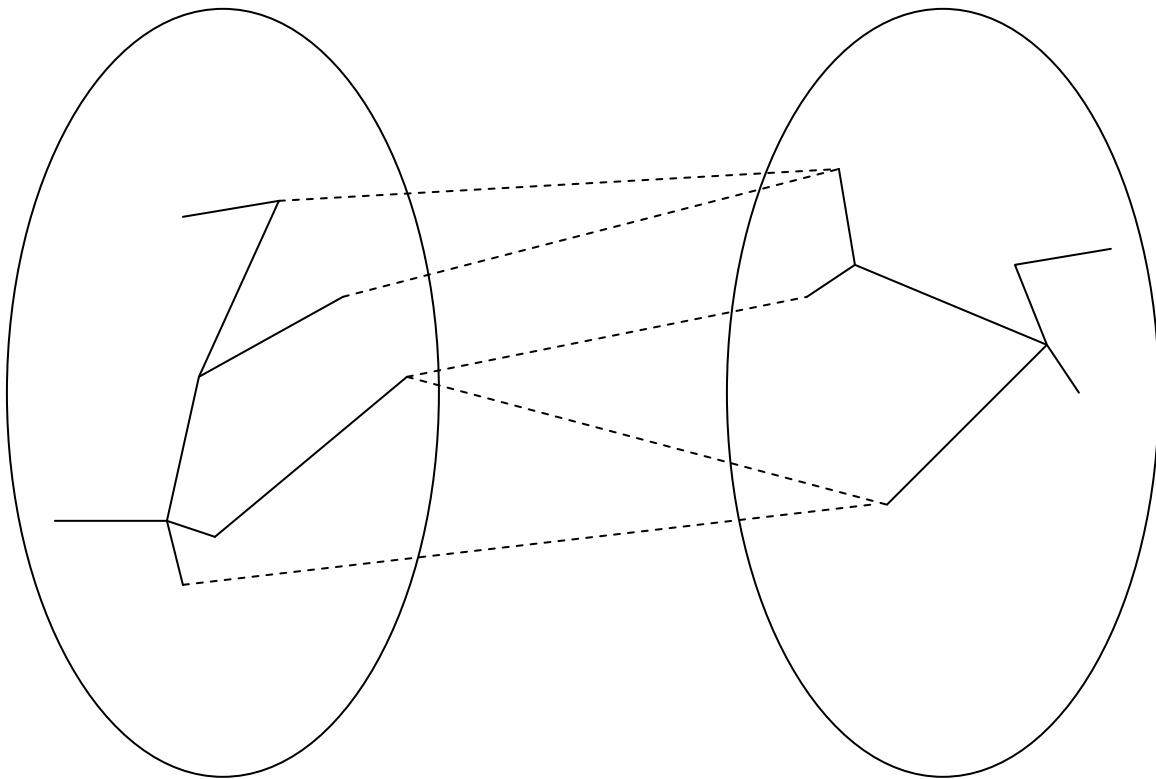
(b) Give a greedy algorithm and use an interchange argument to prove that it works for a single server as long as no new customers arrive.

(c) Show how your algorithm can fail if new customers arrive while the others are being served. (For this problem, assume that you cannot stop serving a customer once you start; coffee equipment is not designed for context switching.)

(d) Show that your algorithm does not necessarily find the best solution if the coffee shop has two servers.

5) 20 pts

Assume we have two graphs $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$. Also assume that we have T_1 which is a MST of G_1 and T_2 which is MST of G_2 . Now consider a new graph $G = (V, E)$ such that $V = V_1 \cup V_2$ and $E = E_1 \cup E_2 \cup E_3$ where E_3 is a new set of edges that all cross the cut (V_1, V_2) . Following is an example of what G might look like.



The dashed edges are E_3 , the solid edges in G_1 (on the left) are T_1 , and the solid edges in G_2 (on the right) are T_2 .

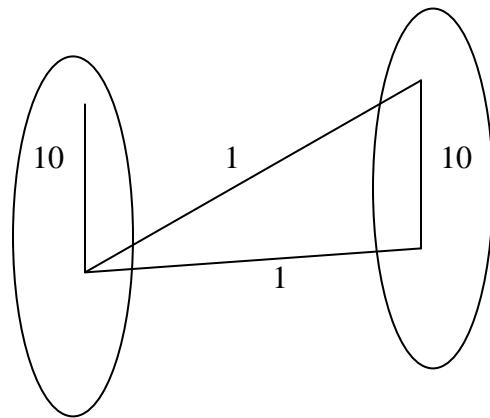
Now assume we want to find a MST of the new graph. Consider the following algorithm:

```
Maybe-MST(T1, T2, E3)
emin = a minimum weight edge in E3
T = T1 U T2 U { emin }
return T
```

Prove or disprove this algorithm (space provided on next page)

Additional space

Can show a simple counter example.



6) 20 pts

For an unsorted array x of size n , give a divide and conquer algorithm that runs in $O(n)$ time and computes the maximum sum M of two adjacent elements in an array.
 $M = \text{Max } (x_i + x_{i+1}); i=1 \text{ to } n-1$

Divide: divide problem into 2 equal size subproblems at each step

Conquer: solve the subproblems recursively until the subproblem become trivial to solve.

In this case problem size of 2 is trivial. In that case the solution is the sum of the 2 numbers.

Combine: We need to merge the solutions to the two subproblems S1 and S2. At each step we need to evaluate the following 3 cases:

Case 1: Max is in S1 (subproblem to the left)

Case 2: Max is in S2 (subproblem to the right)

Cases 3: Max is on the border of S1 and S2

To achieve this. In addition to finding the Max and sending it up the recursion tree at each step we need to send the two numbers at the ends of the subarrays S1 and S2 at each step. So using the following convention:

F1 = first element of S1

L1 = last element of S1

MAX1 = maximum sum of two adjacent elements in S1

F2 = first element of S2

L2 = last element of S2

MAX2 = maximum sum of two adjacent elements in S2

Then we will combine the two subproblems as follows:

If ($L1+F2 > MAX1$ and $L1+F2 > MAX2$) then

 F = F1

 L = L2

 MAX = L1+F2

Else if ($MAX1 > MAX2$) then

 F = F1

 L = L2

 MAX = MAX1

Else

 F = F1

 L = L2

 MAX = MAX2

endif

Additional Space

Additional Space

CS570
Analysis of Algorithms
Fall 2009
Exam I

Name: _____

Student ID: _____

____ Monday ____ Friday 2-5 ____ Friday 5-8 ____ DEN

	Maximum	Received
Problem 1	20	
Problem 2	15	
Problem 3	15	
Problem 4	15	
Problem 5	15	
Problem 6	20	
Total	100	

2 hr exam

Close book and notes

If a description of an algorithm is required, please limit your description within 200 words, anything beyond 200 words will not be considered. Proof/justification or complexity analysis will only be considered if the algorithm is either correct or provided by the problem.

1) 20 pts

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

[TRUE]

Given a min-heap with n elements, the time complexity of select the smallest element from the n elements is $O(1)$.

[FALSE] But we give credit to both true and false

Given a min-heap with n elements, the time complexity of select the second smallest element from the n elements is $O(1)$.

[FALSE] Algorithms (e.g., bubble sort) with $O(n^2)$ time complexity could cost $O(n)$ in some instances.

Given a problem with input of size n , a solution with $O(n)$ time complexity always costs less in computing time than a solution with $O(n^2)$ time complexity.

[FALSE]

By using a heap, we can sort any array with n integers in $O(n)$ time.

[TRUE] m is at most $n(n-1)$

For any graph with n vertices and m edges we can say that $m = O(n^2)$.

[FALSE] m could be $n(n-1)$

For any graph with n vertices and m edges we can say that $O(m+n) = O(m)$.

[FALSE] Consider the following counter-example:

G(V, E). V={A,B,C}, (A,B)=2, (A,C)=1, (B,C)=1. P=AB is the shortest path between A and B, but it is not in the MST.

Given the shortest path P between two nodes A and B in graph $G(V,E)$, then there exists a minimum spanning tree T of G , such that all edges of path P is contained in T .

[FALSE] Consider the following counter-example:

w1 prefers m1 to m2; w2 prefers m2 to m1; m1 prefer w1 to w2; m2 prefer w2 to w1

Consider an instance of the Stable Matching Problem in which there exists a man m and a woman w such that m is ranked last on the preference list of w and w is ranked last on the preference list of m , then in every stable matching S for this instance, the pair (w, m) belongs to S .

[FALSE] A graph could have multiple MSTs.

While there are many algorithms to find the minimum spanning tree in a graph, they all produce the same minimum spanning tree.

[TRUE]

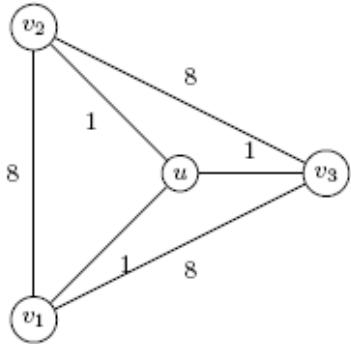
A BFS tree is a spanning tree

2) 15 pts

Here is a divide-and-conquer algorithm that aims at finding a minimum spanning tree. Given a graph $G = (V, E)$, partition the set V of vertices into two sets V_1 and V_2 such that $|V_1|$ and $|V_2|$ differ by at most 1. Let E_1 be the set of edges that are incident only on vertices in V_1 , and let E_2 be the set of edges that are incident only on vertices in V_2 . Recursively solve a minimum spanning tree problem on each of the two subgraphs $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$. Finally, select the minimum-weight edge in E that crosses the cut (V_1, V_2) , and use this edge to unite the resulting two minimum spanning trees into a single spanning tree.

Either argue that the algorithm correctly computes a minimum spanning tree of G , or provide an example for which the algorithm fails.

This algorithm fails. Take the following simple graph:



Never mind how the algorithm first divides the graph, the MST in one subgraph (the one containing u) will be one edge of cost 1, and in the other subgraph, it will be one edge of cost 8. Adding another edge of cost 1 will give a tree of cost 10. But clearly, it would be better to connect everyone to u via edges of cost 1.

3) 15 pts

Suppose you are given two sets A and B , each containing n positive integers. You can choose to reorder each set however you like. After reordering, let a_i be the i th element of set A , and let b_i be the i th element of set B . You then receive a payoff of $\prod_{i=1}^n a_i^{b_i}$. Give an algorithm that will maximize your payoff. Prove that your algorithm maximizes the payoff, and state its running time.

Solution:

Algorithm:

Sort A and B into monotonically decreasing order.

Proof:

Consider any indices i, j, p, q such that $i < j$ and $p < q$, and consider the terms a_i^{bp} and a_j^{bq} . We want to show that it is no worse to include these terms in the payoff than to include a_i^{bq} and a_j^{bp} , that is, $a_i^{bp} a_j^{bq} \geq a_i^{bq} a_j^{bp}$.

Since A and B are sorted into monotonically decreasing order and $i < j, p < q$, we have $a_i \geq a_j$ and $b_p \geq b_q$. Since a_i and a_j are positive and $b_p - b_q$ is nonnegative, we have $a_i^{bp-bq} \geq a_j^{bp-bq}$.

Multiplying both sides by $a_i^{bq} a_j^{bq}$ yields $a_i^{bp} a_j^{bq} \geq a_i^{bq} a_j^{bp}$.

Since the order of multiplication does not matter, sorting A and B into monotonically increasing order works as well.

Complexity:

Sorting 2 sets of n positive integers is $O(n \log n)$.

4) 15 pts

Indicate, for each pair of expressions (A, B) in the table below, whether A is O, Ω or Θ of B. Assume that $k \geq 1$, and $\varepsilon > 0$ are constants. Answer “yes” or “no” in each box in the following form. No explanations required.

A	B	O	Ω	Θ
$\log^k n$	n^ε	Yes	No	No
\sqrt{n}	$n^{\sin n}$	No	No	No
$n^{\log m}$	$m^{\log n}$	Yes	Yes	Yes

The explanation is not required:

- (1) $\log(\log n) = O(\log n) \Rightarrow k \log(\log n) = O(\varepsilon \log n) \Rightarrow \log(\log^k n) = O(\log^\varepsilon n) \Rightarrow \log^k n = O(n^\varepsilon)$
- (2) $-1 \leq \sin n \leq 1$, so we can not determine which one is larger;
- (3) $n^{\log m} = \Theta(n^{\lg m})$, $m^{\log n} = \Theta(m^{\lg n})$, let $m = 10^x$, and $n = 10^y$, then $n^{\lg m} = m^{\lg n} = 2^{xy}$.

5) 15 pts

Suppose you are given a number x and an array A with n entries, each being a distinct number. Also it is known that the sequence of values $A[1]; A[2]; \dots; A[n]$ is unimodal. In other words for some unknown index p between 1 and n , we have

$A[1] < A[2] < \dots < A[p]$ and $A[p] > A[p+1] > \dots > A[n]$.

Give an algorithm with running time $O(\log n)$ to find out if x belongs to A , if yes the algorithm should return the index j such that $A[j] = x$. You should justify both your algorithm and the running time.

Solution: The idea is to first find out p and then break A into two separated sorted arrays, then use binary search on these two arrays to check if x is belong to A .

Let $\text{FindPeak}()$ be the function of finding the peak in A . Then $\text{FindPeak}(A[1:n])$ works as follows:

Look at $A[n/2]$, there are 3 cases:

(1) If $A[n/2-1] < A[n/2] < A[n/2+1]$, then the peak must come strictly after $n/2$.

Run $\text{FindPeak}(A[n/2:n])$.

(2) If $A[n/2-1] > A[n/2] > A[n/2+1]$, then the peak must come strictly before $n/2$.

Run $\text{FindPeak}(A[1:n/2])$.

(3) If $A[n/2-1] < A[n/2] > A[n/2+1]$, then the peak is $A[n/2]$, return $n/2$.

Now we know the peak index(p value). Then we can run binary search on $A[1:p]$ and $A[p+1:n]$ to see if x belong to them because they are both sorted.

In the procedure of finding p , we halve the size of the array in each recurrence. The running time $T(n)$ satisfies $T(n) = T(n/2) + O(1)$. Thus $T(n) = O(\log n)$. Also both binary search has running time at most $O(\log n)$, so total running time is $O(\log n)$.

Explanations:

Note that trying to get x in one shot (in one divide and conquer recursion) usually does not work. Binary search certainly cannot work because the array is not sorted. Modified binary search can hardly work either. The problem is that in each round you need to abandon half the array. However, this decision is hard to made. For example, the array is $[1 2 3 4 5 -1]$, $x=-1$. When divide we have $\text{mid}=3$. We find $A[\text{mid}-1] < A[\text{mid}] < A[\text{mid}+1]$. A common but wrong practice here is to continue search only in the $A[1:\text{mid}]$. We can see clearly $x=-1$ is in the second half of the array. On the other hand you cannot throw away the first half of the array either. The counter example is $[1 2 3 4 5 -1]$ where $x=1$.

One other mistake is to search p in a linear fashion. This take $O(n)$ in the worst case.

6) 20 pts

Given a string S with m digits (digits are from 1 to 9), you are required to delete n ($n \leq m$) digits from S . After the operation, you have a string S' with $m-n$ digits, let N denote the number that S' represents. Design a greedy algorithm to minimize N . For example, $S = "456298111"$, $n = 3$, after deleting 4, 5 and 6, you get the minimal $N = 298111$. Prove the correctness of your algorithm and analyze its time complexity.

Solution:

The greedy strategy: every step we delete a digit, make sure the number represented by the rest of the digits is minimal. To achieve this, in each step, we do the following operation:

Find the first i such that $S[i] > S[i+1]$, then we delete $S[i]$; if such i does not exist, which means the digits are in increasing order, then we simply delete the last digit. (*) We call the position i “peak”.

Proof by induction

(1) We first prove (*) can achieve minimal N when $n = 1$. We have

$$S' = a_1 a_2 \dots a_{i-1} a_{i+1} \dots a_{n-1}$$

Without loss of generality, suppose we delete $S[j]$ rather than $S[i]$, then we have

$S_1 = a_1 a_2 \dots a_{j-1} a_{j+1} \dots a_{n-1}$ ($j \neq i$), and N_1 is the number represented by S_1 .

If $j < i$, since $a_j < a_{j+1}$, we have $N < N_1$; if $j > i$, since $a_{i+1} < a_i$, we have $N < N_1$.

Therefore, $N < N_1$ for all $j \neq i$.

(2) Next, we assume (*) can achieve minimal N when $n=k$, now we prove (*) can achieve minimal N when $n=k+1$.

Suppose the first deletion, we delete p' , then for the rest of k deletions, we need to do (*) k times (deleting the first k peaks) in order to get the minimal result, which is denoted by N' ;

Let N denote the number generated by deleting the first $k+1$ peaks, p_1, p_2, \dots, p_{k+1} , in S .

If $p' = p_i$ ($1 \leq i \leq k+1$), then $N' = N$; if $p' < p_1$ or $p_i < p' < p_{i+1}$ or $p' > p_{k+1}$, similar to (1), we can prove $N' > N$; Therefore, we show that $N \leq N'$.

(3) With (1) and (2), we prove the correctness of the algorithm.

Complexity:

$O(n)$, but $O(mn)$ is also acceptable.

CS570
Analysis of Algorithms
Fall 2010
Exam I

Name: _____
Student ID: _____

____ Monday ____ Friday ____ DEN

	Maximum	Received
Problem 1	20	
Problem 2	15	
Problem 3	15	
Problem 4	15	
Problem 5	15	
Problem 6	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**]

The number of spanning trees in a fully connected graph with n vertices goes up exponentially with respect to n .

[**FALSE**]

BFS can be used to find the shortest path between any two nodes in a weighted graph.

[**FALSE**]

DFS can be used to find the shortest path between any two nodes in a non-weighted graph.

[**TRUE**]

While there are different algorithms to find a minimum spanning tree of an undirected connected weighted graph G , all of these algorithms produce the same result for a given graph with unique edge costs.

[**TRUE**]

If $T(n)$ is $\Theta(f(n))$, then $T(n)$ is both $O(f(n))$ and $\Omega(f(n))$.

[**TRUE**]

The array [20 15 18 7 9 5 12 3 6 2] forms a max-heap.

[**TRUE**]

Suppose that in an instance of the original Stable Marriage problem with n couples, there is a man M who is first on every woman's list and a woman W who is first on every man's list. If the Gale-Shapley algorithm is run on this instance, then M and W will be paired with each other.

[**TRUE**]

The complexity of the recursion given by $T(n) = 4T(n/2) + cn^2$, for some positive constant c , is $O(n^2 \log n)$.

[**FALSE**]

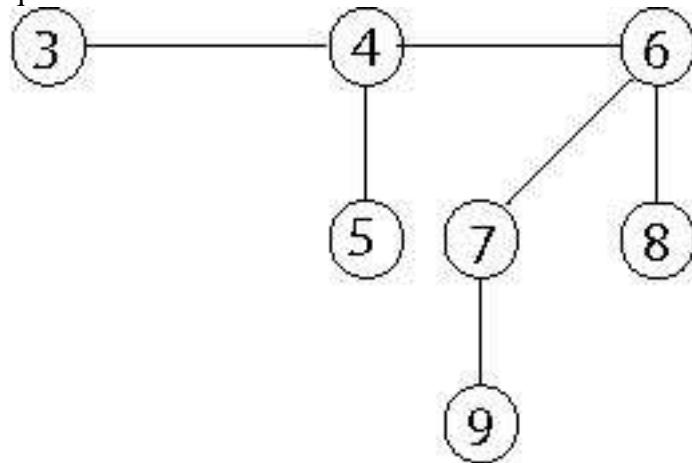
Consider the interval scheduling problem. A greedy algorithm, which is designed to always select the available request that starts the earliest, returns an optimal set A.

[**FALSE**]

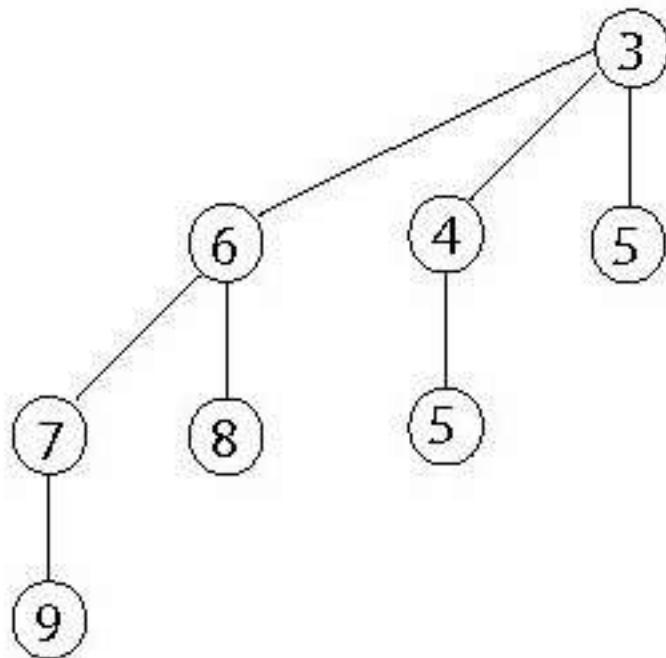
Any divide and conquer algorithm will run in best case $\Omega(n \log n)$ time because the height of the recursion tree is at least $\log n$.

2) 15 pts

You are given the below binomial heap. Show all your work as you answer the questions below.



a- Insert a new node with key value 5. Show the resulting tree and intermediate steps if any. Is the resulting heap also a binary heap and/or a Fibonacci heap?



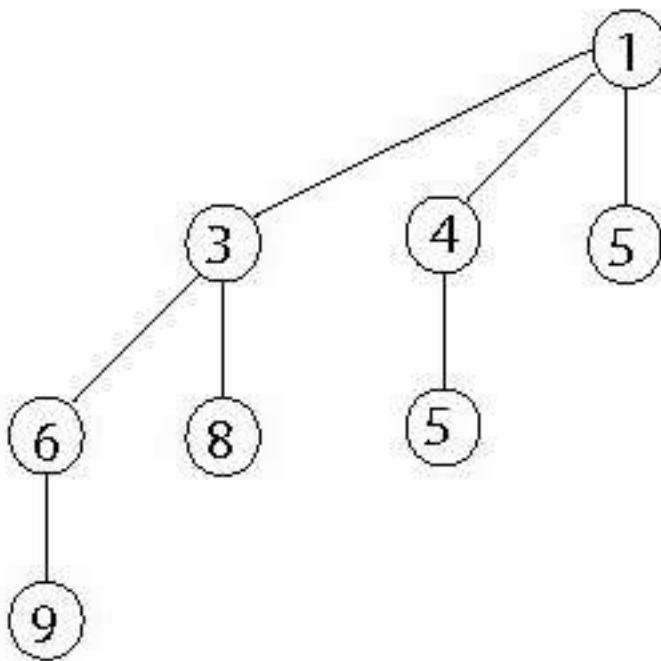
This is also a Fibonacci heap, but not binary heap.

b- Analyze the complexity of your insertion algorithm.

O(logn)

c- Now decrease the key of node 7 to 1. Is the minimum-heap property violated? If so, rearrange the heap. Show the resulting tree and intermediate steps if any.

When key value is changed to 1, min-heap property is violated since 1 is the smallest key value and it has to be at the root node. The rearrangement of key values results in the following heap.



d- Analyze the complexity of the operation in C.

O(logn)

3) 15 pts

A polygon is convex if all of its internal angles are less than 180° (and none of the edges cross each other). We represent a convex polygon as an array $V[1..n]$ where each element of the array represents a vertex of the polygon in the form of a coordinate pair (x, y) . We are told that $V[1]$ is the vertex with the minimum x coordinate and that the vertices $V[1..n]$ are ordered counterclockwise. You may also assume that the x coordinates of the vertices are all distinct, as are the y coordinates of the vertices.

a- Give a divide and conquer algorithm to find the vertex with the maximum x coordinate in $O(\log n)$ time.

Note that for each $1 \leq i < n$ either $V[i] < V[i + 1]$ or $V[i] > V[i + 1]$ (Such an array is called a unimodal array). The main idea is to distinguish these two cases:

1. if $V[i] < V[i + 1]$, then the maximum element of $V[1..n]$ occurs in $V[i + 1..n]$.
 2. In a similar way, if $V[i] > V[i + 1]$, then the maximum element of $V[1..n]$ occurs in $V[1..i]$.
- This leads to the following divide and conquer solution (note its resemblance to binary search):

```
1 a, b ← 1, n
2 while a < b
3   do mid ← ⌊(a + b)/2⌋
4     if V[mid] < V[mid + 1]
5       then a ← mid + 1
6     if V[mid] > V[mid + 1]
7       then b ← mid
8 return V[a]
```

The precondition is that we are given a unimodal array $V[1..n]$. The postcondition is that $V[a]$ is the maximum element of $V[1..n]$. For the loop we propose the invariant “The maximum element of $V[1..n]$ is in $V[a..b]$ and $a \leq b$ ”.

When the loop completes, $a \geq b$ (since the loop condition failed) and $a \leq b$ (by the loop invariant). Therefore $a = b$, and by the first part of the loop invariant the maximum element of $V[1..n]$ is equal to $V[a]$.

We use induction to prove the correctness of the invariant. Initially, $a = 1$ and $b = n$, so, the invariant trivially holds. Suppose that the invariant holds at the start of the loop. Then, we know that the maximum element of $V[1..n]$ is in $V[a..b]$. Notice that $V[a..b]$ is unimodal as well. If $V[mid] < V[mid + 1]$, then the maximum element of $V[a..b]$ occurs in $V[mid+1..b]$ by case 1. Hence, after $a \leftarrow mid+1$ and b remains unchanged in line 4, the maximum element is again in $V[a..b]$. The other case is symmetric.

To complete the proof, we need to show that the second part of the invariant $a \leq b$ is also true. At the start of the loop $a < b$. Therefore, $a \leq ⌊(a + b)/2⌋ < b$. This means that $a \leq mid < b$ such that after line 4 or line 5 in which a and b get updated $a \leq b$ holds once more.

The divide and conquer approach leads to a running time of $T(n) = T(n/2) + \Theta(1) = \Theta(\log n)$.

b- Give a divide and conquer algorithm to find the vertex with the maximum y coordinate in $O(\log n)$ time.

After finding the vertex $V[max]$ with the maximum x -coordinate, notice that the y -coordinates in $V[max], V[max + 1], \dots, V[n - 1], V[n], V[1]$ form a unimodal array and the maximum y -coordinate of $V[1..n]$ lies in this array. Thus the divide and conquer solution in part a can be used to find the vertex with the maximum y -coordinate. The total running time is $\Theta(\log n)$.

4) 15 pts

You are given a weighted directed graph $G = (V, E, w)$ and the shortest path distances $\delta(s, u)$ from a source vertex s to every other vertex in G . However, you are not given $\pi(u)$ (the predecessor pointers). With this information, give an algorithm to find a shortest path from s to a given vertex t in $O(|V| + |E|)$ time.

Start at u . Of the edges that point to u , at least one of them will come from a vertex v that satisfies $\delta(s, v) + w(v, u) = \delta(s, u)$. Such a v is on the shortest path. Recursively find the shortest path from s to v .

This algorithm hits every vertex and edge at most once, for a running time of $O(|V| + |E|)$.

5) 15 pts

Suppose you are choosing between the following three algorithms:

- Algorithm A solves problems of size n by dividing them into five subproblems of half the size, recursively solving each subproblem, and then combining the solutions in linear time.
- Algorithm B solves problems of size n by recursively solving two subproblems of size $n - 1$ and then combining the solutions in constant time.
- Algorithm C solves problems of size n by dividing them into nine subproblems of size $n/3$, recursively solving each subproblem, and then combining the solutions in $O(n^2)$ time.

What are the running times of each of these algorithms (in big-O notation), and which would you choose?

Algorithm A solves problems by dividing them into five subproblems of half the size, recursively solving each subproblem, and then combining the solutions in linear time.

$$T(n) = 5 T(n/2) + c n$$

Applying master theorem, $a=2$, $b=5$, $f(n)=c n$, $\text{degree}(f(n))=1$

$$\text{Since } \log_2 5 > 1, T(n) = O(n^{\log_2 5}) = O(n^{\log_2 5})$$

Algorithm B solves problems of size n by recursively solving two subproblems of size $n-1$ and then combining the solutions in constant time.

$$T(n) = 2 T(n-1) + c = 2^2 T(n-2) + 2c + c = (2^n - 1)c$$

$$T(n) = O(2^n)$$

Algorithm C solves problems of size n by dividing them into nine subproblems of size $n/3$, recursively solving each subproblems and then combining the solution in $O(n^2)$ time.

$$T(n) = 9 T(n/3) + c n^2$$

Applying master theorem, $a=3$, $b=9$, $f(n)=c n^2$, $\text{degree}(f(n))=2$

$$\text{Since } \log_3 9 = 2, T(n) = O(n^2 \log n)$$

From above three algorithms, we can see that time complexity of the third algorithm is best. Thus, we will choose algorithm C.

6) 20 pts

- a- Suppose we are given an instance of the Shortest Path problem with source vertex s on a directed graph G . Assume that all edges costs are positive and distinct. Let P be a minimum cost path from s to t . Now suppose that we replace each edge cost c_e by its square root, $c_e^{1/2}$, thereby creating a new instance of the problem with the same graph but different costs.

Prove or disprove: P still a minimum-cost $s - t$ path for this new instance.

The statement can be disproved by giving a counterexample as follows.

$G=(V, E); V=\{s, a, t\}; E=\{(s, a), (a, t), (s, t)\};$

$\text{cost}((s, a))=9; \text{cost}((a, t))=16; \text{cost}((s, t))=36.$

It is obvious that the minimum-cost $s - t$ path is $s - a - t$.

By replacing each edge cost c_e by its square root, $c_e^{1/2}$, the costs become:

$\text{cost}((s, a))=3; \text{cost}((a, t))=4; \text{cost}((s, t))=6.$

Now the the minimum-cost $s - t$ path is $s - t$, not $s - a - t$ anymore.

- b- Suppose we are given an instance of the Minimum Spanning Tree problem on an undirected graph G . Assume that all edges costs are positive and distinct. Let T be an MST in G . Now suppose that we replace each edge cost c_e by its square root, $c_e^{1/2}$, thereby creating a new instance of the problem (G') with the same graph but different costs.

Prove or disprove: T is still an MST in G' .

The statement is true due to that replacing each edge cost c_e by its square root, $c_e^{1/2}$ does not change the order of the cost, i.e. for positive real numbers a and b , if a is greater than b , $a^{1/2}$ is greater than $b^{1/2}$.

CS570
Analysis of Algorithms
Fall 2014
Exam I

Name: _____
Student ID: _____

Wednesday Evening Section

	Maximum	Received
Problem 1	20	
Problem 2	16	
Problem 3	16	
Problem 4	16	
Problem 5	16	
Problem 6	16	
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**]

Let T be a complete binary tree with n nodes. Finding a path from the root of T to a given vertex $v \in T$ using breadth-first search takes $O(\log n)$.

False :

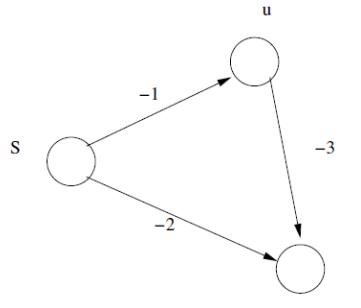
BFS requires $O(n)$ time. BFS examines each node in the tree in breadth-first order. The vertex v could well be the last vertex explored (Also, notice that T is not necessarily stored).

[**TRUE/FALSE**]

Dijkstra's algorithm works correctly on a directed acyclic graph even when there are negative-weight edges.

False:

Consider Dijkstra's algorithm run from source s in the following graph. The vertex v will be removed from the queue with $d[v] = -2$ even though the shortest path to it is -4 .



[**TRUE/FALSE**]

If the edge e is not part of any MST of G , then it must be the maximum weight edge on some cycle in G .

True:

Take any MST T . Since e is not part of it, e must complete some cycle T . e must be the heaviest edge on that cycle. Otherwise a smaller weight tree could be constructed by swapping the heavier edge on the cycle with e and thus T cannot be MST.

[**TRUE/FALSE**]

If $f(n) = O(g(n))$ and $g(n) = O(f(n))$ then $f(n) = g(n)$.

False:

$f(n) = n$ and $g(n) = n + 1$.

[TRUE/FALSE]

The following array is a max heap: [10; 3; 5; 1; 4; 2].

False: The element 3 is smaller than its child 4, violating the maxheap property.

[TRUE/FALSE]

There are at least 2 distinct solutions to the stable matching problem: one that is preferred by men and one that is preferred by women.

False:

Consider the example: two men X and Y; two women A and B.

Preference list (decreasing order):

X's list: A, B ; A's list: X, Y;

Y's list: A, B; B's list: X, Y.

The matching is unique: X-A and Y-B

[TRUE/FALSE]

In a binary max-heap with n elements, the time complexity of finding the second largest element is O(1).

True:

The second largest one should be the larger (or equal) one of the two children of the root node. Finding them takes time O(1). Note: it is possible that several elements have the same value including the first two layer of elements, then in this case, the largest element is equal to the second largest element, and returning one of the second layer elements is also fine.

[TRUE/FALSE]

Given a binary max-heap with n elements, the time complexity of finding the smallest element is O($\lg n$).

False:

The smallest element should be among the leaf nodes. Consider a full binary tree of n nodes. It has $(n+1)/2$ leafs (you can think of why). Then the worst case of finding the smallest element of a full binary tree (heap) is $\Theta(n)$

[TRUE/FALSE]

Kruskal's algorithm can fail in the presence of negative cost edges.

False:

The MST problem does not change even with the negative cost edges. So the Kruskal algorithm still works.

[TRUE/FALSE]

If a weighted undirected graph has two MSTs, then its vertex set can be partitioned into two, such that the minimum weight edge crossing the partition is not unique.

True: Let T1 and T2 be two distinct MSTs. Let e1 be an edge that is in T1 but not in T2. Removing e1 from T1 partitions the vertex set into two connected components. There is a unique edge (call it e2) that is in T2 that crosses this partition. By minimum-weight property of T1 and T2, both e1 and e2 should be of the same weight and further should be of the minimum weight among all edges crossing this partition.

2) 16 pts

The diameter of a graph is the maximum of the shortest paths' lengths between all pairs of nodes in graph G . Design an algorithm which computes the diameter of a connected, undirected, unweighted graph in $O(mn)$ time, and explain why it has that runtime.

Solution:

Suppose we suspected that a particular vertex v was an endpoint of the diameter of the graph. Since this graph is unweighted graph, we can use BFS to find the shortest path from any particular node v and report the largest layer number reached.

However, while we don't know any particular vertex v to be part of this, we do know that there is at least one vertex in the graph for which this is true. Accordingly, we can run BFS from every vertex v , and report the maximum layer reached in all of these runs.

The total time taken is $O(n(m + n))$; $O(m + n)$ for a single breadth-first search, which is run $O(n)$ times.

But notice that, when the graph is connected, n is $O(m)$. Accordingly, $O(m + n)$ is $O(m)$, for a total runtime of $O(mn)$, as desired.

3) 16 pts

Consider a stable marriage problem where the set of men is given by $M = \{m_1, m_2, \dots, m_N\}$ and the set of women is $W = \{w_1, w_2, \dots, w_N\}$. Consider their preference lists to have the following properties:

$$\begin{aligned}\forall w_i \in W: w_i \text{ prefers } m_i \text{ over } m_j, \forall j > i \\ \forall m_i \in M: m_i \text{ prefers } w_i \text{ over } w_j, \forall j > i\end{aligned}$$

Prove that a unique stable matching exists for this problem.

Note: symbol \forall means “for all”.

We will prove that matching S where m_i is matched to w_i for all $i = 1, 2, \dots, N$ is the unique stable matching in this case. We will use the notation $S(a) = b$ to denote that in matching S , a is matched to b .

It is evident that S is a matching because every man is paired with exactly one woman and vice versa. If any man m_j prefers w_k to w_j where $k < j$, then such a higher ranked woman w_k prefers her current partner to m_j . Thus, there are no instabilities and S is a stable matching. Now let's prove that this stable matching is unique.

By way of contradiction, let's assume that another stable matching S' , which is different from S , exists. Therefore, there must exist some i for which $S'(w_i) = m_k$, $k \neq i$. Let x be the minimum value of such an i . Similarly, there must exist some j for which $S'(m_j) = w_l$, $j \neq l$. Let y be the minimum value of such a j . Since $S'(w_i) = m_i$ for all $i < x$, and $S'(m_j) = w_j$ for all $j < y$, $x = y$. $S'(w_x) = m_k$ implies $x < k$. Similarly, $S'(m_y) = w_l$ implies that $y = x < l$. Given the preference lists, $m_y = m_x$ prefers w_x to w_l , and w_x prefers m_x to m_k . This is an instability and hence, S' cannot be a stable matching.

4) 16 pts

Ivan is a businessman and he has several large containers of fruits to ship. As he has only one ship he can transport only one container at a time and also it takes certain fixed amount of time per trip. As there are several varieties of fruits in the containers, the cost and depreciation factor associated with each container is different. He has n containers, a_1, a_2, \dots, a_n . Initial value of each container is v_1, v_2, \dots, v_n and depreciation factor of each container is d_1, d_2, \dots, d_n . So, if container a_i happens to be the j^{th} shipment, its value will be $v_i / (d_i \times j)$. Can you help Ivan maximize total value of containers after depreciation ($\sum v_i / (d_i \times j)$) by providing an efficient algorithm to ship the containers? Provide proof of correctness and state the complexity of your algorithm.

Solution: Ship the containers in decreasing order of v_i / d_i . We prove the optimality of this algorithm by an exchange argument.

Thus, consider any other schedule. This schedule should consist an inversion. Further, in fact, there must be an adjacent such pair i, j . Note that for this pair we have

$v_i / d_i <= v_j / d_j$, for $i < j$. If we can show that swapping this pair i, j does not decrease the total value, then we can iteratively do this until there are no more inversions, arriving at the greedy schedule without having decreased the value we are trying to maximize. It will then follow that our greedy algorithm is optimal.

Consider the effect of swapping i and j . The schedule of shipping all the other containers remains the same. Before the swap, the contribution of i and j to the total value was $(v_i / (d_i * i) + v_j / (d_j * j))$, while after the swap the total value is $(v_i / (d_i * j) + v_j / (d_j * i))$. The difference between value after the swap is $(v_j / d_j - v_i / d_i) * (1/i - 1/j)$. As both, $(v_j / d_j - v_i / d_i)$ and $(1/i - 1/j)$ terms are positive, the total value is not decreased by swapping as desired.

The complexity of this solution is $O(n \log n)$, for sorting in decreasing order.

5) 16 pts

Consider an undirected graph $G = (V, E)$ with distinct nonnegative edge weights $w_e \geq 0$. Suppose that you have computed a minimum spanning tree of G . Now suppose each edge weight is increased by 1: the new weights are $w'_e = w_e + 1$. Does the minimum spanning tree change? Give an example where it changes or prove it cannot change.

Solution:

This question is similar to 6b question in your hw4. Monotonicity is preserved in $(.+)$ function as in $(.)^2$

6) 16 pts

Mark all the correct statements and provide a brief explanation for each.

a. Consider these two statements about a connected undirected graph with V vertices and E edges:

- I. $O(V) = O(E)$
- II. $O(E) = O(V^2)$

(a) I and II are both false.

(b) Only I is true.

(c) Only II is true.

(d) I and II are both true.

(d) If there are V vertices, there can be at most $V C_2$ edges in the graph. For the graph to be connected, there must be at least $V - 1$ edges.

$$E \leq V(V-1)/2 = O(V^2).$$

Also, $V \leq 2(V-1) \leq 2E$

$$\Rightarrow V = O(E)$$

b. Suppose the shortest path from node i to node j goes through node k and that the cost of the subpath from i to k is D_{ik} . Consider these two statements:

I. Every shortest path from i to j must go through k .

II. Every shortest path from i to k has cost D_{ik} .

(a) I and II are both true.

(b) Only I is true.

(c) Only II is true.

(d) I and II are both false.

(c) I. is false because there can be multiple shortest paths from i to j , not necessarily going through k . For example, there could be an edge between i and j that has the same cost as the path through k .

II. is true because if there were a path P' from i to k that had cost less than D_{ik} , then the path from i to j consisting of P' and the path from k to j would be shorter than the shortest i - j path, which is a contradiction.

c. Consider the execution times of two algorithms I and II:

- I. $O(n \log n)$

II. $O(\log(n^n))$

- (a) Only I is polynomial.
 - (b) Only II is polynomial.
 - (c) I and II are both polynomial.
 - (d) Neither I nor II is polynomial.
- (c) $\log(n^n) = n \log n \leq n^2$

d. Suppose $f(n) = 3n^3 + 2n^2 + n$. Consider these statements:

- I. $f(n) = O(n^3)$
- II. $f(n) = O(n^4)$

- (a) Only I is true.
- (b) Only II is true.
- (c) I and II are both true.
- (d) I and II are both false.

(c) For all $n \geq 1$,
$$f(n) \leq 3n^3 + 2n^3 + n^3 = 6n^3 \leq 6n^4$$

Additional Space

CS570
Analysis of Algorithms
Fall 2014
Exam I

Name: _____
Student ID: _____

Thursday Evening Section **DEN Yes / No**

	Maximum	Received
Problem 1	20	
Problem 2	15	
Problem 3	15	
Problem 4	20	
Problem 5	15	
Problem 6	15	
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]

Adding a number w on the weight of every edge of a graph might change the shortest path between two vertices u and v .

True:

Consider $G = (V, E)$ with $V = \{u, x, v\}$ and $E = \{ux, xv, uv\}$. Let weight of edge $ux = w_{ux} = w_{xv} = 3$ and $w_{uv} = 7$. Then the shortest path from u to v is given by $u \rightarrow x \rightarrow v$ with a total weight of 6. However, if we now add 2 to every edge weight, the path $u \rightarrow x \rightarrow v$ will have a total weight of 10 while the path $u \rightarrow v$ will have a weight of 9, making it the new shortest path.

[TRUE/FALSE]

Suppose that for some graph G we have that the average edge weight is A . Then a minimum spanning tree of G will have weight at most $(n - 1) \cdot A$.

False:

We may be forced to select edges with weight much higher than average. For example, consider a graph G consisting of a complete graph G' on 4 nodes, with all edges having weight 1 and another vertex u , connected to one of the vertices of G' by an edge of weight 8. The average weight is $(8 + 6)/7 = 2$. Therefore, we would expect the spanning tree to have weight at most $4 * 2 = 8$. But the spanning tree has weight more than 8 because the unique edge incident on u must be selected.

[TRUE/FALSE]

DFS finds the longest paths from start vertex s to each vertex v in the graph.

False:

Depends on the order in which the nodes are traversed

[TRUE/FALSE]

If one can reach every vertex from a start vertex s in a directed graph, then the graph is strongly connected.

False

[TRUE/FALSE]

$F(n) = 4n + 3\sqrt{n}$ is both $O(n)$ and $\Omega(n)$.

True:

The dominant term is $4n$, which is obviously both $O(n)$ and $\Omega(n)$

[TRUE/FALSE]

In Fibonacci heaps, the decrease-key operation takes $O(1)$ time.

True:

Just as definition of decrease-key operation in Fibonacci heaps

[TRUE/FALSE]

If the edge weights of a weighted graph are doubled, then the number of minimum spanning trees of the graph remains unchanged.

True:

With edge weights doubled, the weights of all possible spanning trees in the graph are doubled. So any MST in the original graph is also the MST in the new graph; any spanning tree that is not the MST in the original graph is still not the MST in the new graph.

[TRUE/FALSE]

Given a binary max-heap with n elements, the time complexity of finding the smallest element is $O(\lg n)$.

False:

The smallest element should be among the leaf nodes. Consider a full binary tree of n nodes. It has $(n+1)/2$ leafs (you can think of why). Then the worst case of finding the smallest element of a full binary tree (heap) is $\Theta(n)$

[TRUE/FALSE]

An undirected graph $G = (V, E)$ must be connected if $|E| > |V| - 1$

False:

Consider a graph having nodes: a, b, c, d, e. {a, b, c, d} forms a fully connected subgraph, while e is isolated from other nodes. Now the fully connected subgraph has 6 edges, and there are only 5 nodes in total. But this graph is not a connected graph.

[TRUE/FALSE]

If all edges in a connected undirected graph have unit cost, then you can find the MST using BFS.

True:

Any spanning tree of a graph having only unit cost edges is also a MST, because the weight is always $n-1$ units. Of course, BFS gives a spanning tree in the connected graph.

2) 16 pts

At the Perfect Programming Company, programmers program in pairs in order to ensure that the highest quality code is produced. The productivity of each pair of programmers is the speed of the slower programmer. For an even number of programmers, give an efficient algorithm for pairing them up so that the sum of the productivity of all pairs is maximized. Analyze the running time and prove the correctness of your algorithm.

Solution:

A simple greedy algorithm works for this problem. Sort the speeds of the programmers in decreasing order using an optimal sorting algorithm such as merge sort. Consecutive sorted programmers are then paired together starting with pairing the fastest programmer with the second fastest programmer.

Sorting takes $O(n \lg n)$ time while pairing the programmers takes $O(n)$ time giving a total running time of $O(n \lg n)$.

Correctness: Let P be the set of programmers. The problem exhibits an optimal substructure.

Assume the optimal pairing. Given any pair of programmers $(i; j)$ in the optimal pairing, the optimal sum of productivity is just the sum of the productivity of $(i; j)$ with the optimal sum of the productivity of the all pairs in $P - \{i, j\}$.

We now show that the greedy choice works by showing that there exists an optimal pairing such that the two fastest programmers are paired together. Assume an optimal pairing where fastest programmer i is not paired with the second faster programmer j . Instead let i be paired with k and j be paired with l . Let p_i, p_j, p_k and p_l be the programming speeds of i, j, k and l respectively. We now change the pairings by pairing i with j and k with l . The change in the sum of productivities is

$$(p_i + \min(p_k, p_l)) - (p_k + p_l) = p_i \cdot \max(p_k, p_l) \geq 0$$

since p_i is at least as large as the larger of p_k and p_l . We now have an optimal pairing where the fastest programmer is paired with the second fastest programmer. Hence to find the optimal solution, we can keep pairing the two fastest remaining programmers together.

3) 16 pts

The graph K_n is defined to be an undirected graph with n vertices and all possible edges (a fully connected graph). That is, the vertices are named $\{1, \dots, n\}$ and for any numbers i and j with $i \neq j$, there is an edge between vertex i and vertex j . Describe the result of a breadth-first search and a depth-first search of K_n . For each search, describe the resulting search tree.

Solution:

For the BFS, the algorithm looks at all the neighbors of the root node before going to the second level. Since every node in the graph is a neighbor of every other node, every node other than the root is put at level 1. Thus the tree has one node at level 0 and $n-1$ nodes at level 1. The non-tree edges are exactly those edges between different nodes on level 1 -- there are $(n-1 \text{ choose } 2)$ of these.

For the DFS, the search of the root node (call it 1) will find another node 2, and then the recursive call on node 2 will find node 3, and so forth. None of the recursive calls can terminate while any undiscovered nodes remain. So the tree edges form a single path of $n-1$ edges, with one node on each level and thus a single leaf. There are back edges, $(n-1 \text{ choose } 2)$ of them, as each node has a back edge to each of its ancestors except its parent.

4) 16 pts

A *d*-ary heap is like a binary heap, but instead of 2 children, nodes have *d* children.

(a) How would you represent a d-ary heap in an array? [4 points]

If we know the maximum number, *N*, of nodes that we may store in the d-ary heap, we can allocate an array *H* of size *N* indexed by $i = 1, 2, \dots, N$. The heap nodes will correspond to positions in *H*. $H[1]$ will be the root node and for any node at position *i* in *H*, its children will be at positions $d*(i-1) + 2, d*(i-1) + 3, \dots, d*(i-1) + d + 1$, and the parent position at $\lfloor (i-2)/d \rfloor + 1$. If there are $n < N$ elements in the heap at any time, we will use the first *n* indices of the array to store the heap elements.

(b) What is the height of a d-ary heap of *n* elements in terms of *n* and *d*? [4 points]

There is 1 node at level 0 (L_0), *d* at L_1 , d^2 at L_2 and so on. Let *h* be the height of the d-ary heap. Then there are d^h nodes at the last level of the heap. Thus,

$$n = 1 + d + d^2 + d^3 + \dots + d^h$$

Solving this, we get

$$h = \log_d(n(d-1)+1) - 1$$

(c) Give an efficient implementation of ExtractMin. Analyze its running time in terms of *d* and *n*. [8 points]

Similar to the ExtractMin operation for a binary heap, for a d-ary heap with *n* elements, we find the minimum (the root node) in $O(1)$ time and to delete it, we replace $H[1]$ with $H[n] = w$. If the resulting array is not a heap, we use Heapify-up or Heapify-down to fix the heap in $O(h) = O(\log_d n)$ time.

5) 16 pts

You are given a directed graph representing several career paths available in the industry. Each node represents a position and there is an edge from node v to node u if and only if v is a pre-requisite for u. Starting positions are the ones which have no pre-requisite positions. Except starting positions, we can only perform a position only if any of its pre-requisite positions are performed. Top positions are the ones which are not pre-requisites for any positions. Ivan wants to start a career at any of the starting positions and achieve a top position by going through the minimum number of positions. Using the given graph provide a linear time algorithm to help Ivan choose his desired career path. Note that this graph has no cycles.

Solution: Add a node s and connect an edge from s to all nodes that have no incoming edges (starting positions). Add a node t and connect an edge from all nodes with no outgoing edges (top positions) to t. Do a BFS from s and find the shortest path to t. This will give you the shortest path from a starting position to a top position in $O(m+n)$.

6) 16 pts

Suppose we are given an instance of the Minimum Spanning Tree problem on a graph G .

Assume that all edges costs are distinct. Let T be a minimum spanning tree for this instance. Now suppose that we replace each edge cost c_e by its square, c_e^2 thereby creating a new instance of the problem with the same graph but different costs.

Prove or disprove: T is still a MST for this new instance.

Solution:

The claim is false.

Note that the edge cost can be negative. (This is the key difference between this problem and a homework problem in HW4)

Consider the following example:

The original graph: $G = (V, E)$ being an undirected graph, where $V = \{a, b, c\}$, $E = \{(a,b), (b,c), (a,c)\}$. Weights: $w(a,b) = 1$, $w(b,c) = -3$, $w(a,c) = 2$.

The MST is : $T = (T_v, T_E)$, where $T_v = \{a, b, c\}$, $T_E = \{(a,b), (b,c)\}$

After squaring the edge weights, we get a new graph G' with $w'(a,b) = 1$, $w'(b,c) = 9$, $w'(a,c) = 4$.

The MST is: $T' = (T'_v, T'_E)$, where $T'_v = \{a, b, c\}$, $T'_E = \{(a,b), (a,c)\}$.

So the MST changes..

Additional Space

CS570
Analysis of Algorithms
Spring 2007
Exam 1

Name: _____
Student ID: _____

	Maximum	Received
Problem 1	14	
Problem 2	6	
Problem 3	15	
Problem 4	20	
Problem 5	15	
Problem 6	20	
Problem 7	10	

Note: The exam is closed book closed notes.

1) 14 pts

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

[**TRUE/FALSE**] False

Function $f(n)=5n^32^n + 6n^23^n$ is $O(n^32^n)$.

[**TRUE/FALSE**] True

$n\log^3 n$ is NOT $O(n\log n)$

[**TRUE/FALSE**] True

In an undirected graph, the shortest path between two nodes lies on some minimum spanning tree.

[**TRUE/FALSE**] False

Max base heap can be converted into Min based heap by reversing the order of the elements in the heap array.

[**TRUE/FALSE**] False

Kruskal's algorithm for finding a MST of a weighted, undirected graph is a form of a dynamic programming technique.

[**TRUE/FALSE**] True

In the case of applying Dijkstra's algorithm for dense graph, Fibonacci implementation is the best one among Binary, Binomial, and Fibonacci.

[**TRUE/FALSE**] False

If all of the weights from a connected and weighted graph are distinct, then distinct spanning trees of the graph have distinct weights.

2) 6pts

What is the worst-case complexity of the each of the following code fragments?

a) for ($i = 0; i < 2N; i++$) {
 sequence of statements
}
for ($j = 0; j < 3M; j++$) {
 sequence of statements
}
 $O(M+N)$

b) for ($i = 0; i < N; i++$) {
 for ($j = 0; j < 2N^2; j++$) {
 sequence of statements
 }
}
for ($k = 0; k < 3M; k++$) {
 sequence of statements
}
 $O(N^3+M)$

c) for ($i = 0; i < N^3; i++$) {
 for ($j = i; j < 2\lg(M); j++$) {
 sequence of statements
 }
}
for ($k = 0; k < M; k++$) {
 sequence of statements
}

$O(N^3 \log M + M)$

3) 15 pts

The maximum spanning tree problem is to find a spanning tree of a graph whose edge weights have the largest sum possible. Give an algorithm to find a maximum spanning tree and give a proof that the algorithm is correct.

Solution: We can reduce finding the maximum spanning tree problem into finding minimum spanning tree problem. The reduction is as follows: Let $w(e)$ denote the weight of edge e in the graph and let M be the maximum weight of all the edges in G . Now consider re-weighting each edge e in G as $M-w(e)$. Now consider two spanning tree of G , T and T' and note that they both contain exactly the same number of edges, i.e., $n-1$ edges, where G has n nodes. The key point is that $\sum_{e \in T} w(e) \geq \sum_{e' \in T'} w(e')$ if and only if

$$\sum_{e \in T} [M - w(e)] = (n-1)M - \sum_{e \in T} w(e) \leq (n-1)M - \sum_{e' \in T'} w(e') = \sum_{e' \in T'} [M - w(e')]$$

Hence, if we solve the minimum spanning tree problem using the new edge weights, the optimal tree found will be the maximum spanning tree using the original weights. Therefore we can reduce the maximum spanning tree problem into the minimum spanning tree problem. Hence we can use either Kruskal's algorithm or Prim's algorithm after the reduction and the running time is

$O(m \log n)$

4) 20 pts

Suppose you are given two sets A and B , each containing n positive integers. You can choose to reorder each set however you like. After reordering, let a_i be the i -th element of set A , and let b_i be the i -th element of set B . You then receive a payoff of $\sum_{i=1}^n b_i \log a_i$. Give an algorithm that will maximize your payoff. Prove that your algorithm maximizes the payoff, and state its running time.

Algorithm: Sort A and B in increasing order and the payoff is maximized

Proof: Suppose $\{a_1, \dots, a_n\}$ and $\{b_1, \dots, b_n\}$ is an optimal solution but $(a_i > a_j)$ and $(b_i < b_j)$. Then switch a_i and a_j we can get a better solution. The reason is as follows:

$$\begin{aligned} a_i > a_j, b_i < b_j &\Leftrightarrow a_i^{b_j - b_i} > a_j^{b_j - b_i} \Leftrightarrow a_i^{b_j} a_j^{b_i} > a_j^{b_j} a_i^{b_i} \Leftrightarrow \log(a_i^{b_j} a_j^{b_i}) > \log(a_j^{b_j} a_i^{b_i}) \\ &\Leftrightarrow b_j \log a_i + b_i \log a_j > b_j \log a_j + b_i \log a_i \end{aligned}$$

which contradicts that $\{a_1, \dots, a_n\}$ and $\{b_1, \dots, b_n\}$ is an optimal solution

Running time: Sorting takes $O(n \log n)$. Hence the running time of our algorithm is $O(n \log n)$

5) 15 pts

Given a connected graph $G = (V, E)$ with positive edge weights and two nodes s, t in V , prove or disprove:

- a. If all edge weights are unique, then there is a single shortest path between any two nodes in V .
 - b. If each edge's weight is increased by k , the shortest path cost will increase by a multiple of k .
 - c. If the weight of some edge e decreases by k , then the shortest path cost will decrease by at most k .
- a) False. Counter Example: (s,a) with weight 1, (a,t) with weight 2 and (s,t) with weight 3. There are two shortest path from s to t though the edge weights are unique.
- b) **False. Example: suppose the shortest path $s \rightarrow t$ consist of two edges, each with cost 1, and there is also an edge $e=(s,t)$ in G with $\text{cost}(e)=3$. If now we increase the cost of each edge by 2, e will become the shortest path (with the total cost of 5).**
- c) True. For any two nodes s, t , assume that P_1, \dots, P_k are all the paths from s to t . If e belongs to P_i then the path cost decrease by k , otherwise the path cost unchanged. Hence all paths from s to t will decrease by at most k . As shortest path is among them, then the shortest path cost will decrease by at most k .

Grading policy: 5 points for each item. 2 points for TRUE/FALSE part and 3 points for prove/disprove part.

6) 20 pts

Sam is shocked to find out that his word processor has corrupted his text document and has eliminated all spacing between words and all punctuation so his final term paper looks something like: “anawardwinningalgorithmto...”. Luckily he has access to a Boolean function *dictionary()* that takes a string *s* and returns true if *s* is a valid word and false otherwise. Considering that he has limited time to turn in his paper before the due date, find an algorithm that reconstructs his paper into a sequence of valid words with no more than $O(n^2)$ calls to the *dictionary()* function.

Solution: We denote that the string as $s[1], \dots, s[n]$. For the sub string $s[1], \dots, s[m]$, we construct the table P as follows: if $s[1], \dots, s[m]$ is composed of a sequence of valid words, then $P[m]=\text{true}$; otherwise $P[m]=\text{false}$. The recursive relation is as follows:

$$P[0] = \text{true}$$

$$P[m] = \vee_{0 \leq i \leq m-1} (P[i] \wedge \text{dictionary } (s[i+1], \dots, s[m]))$$

If $P[m]$ is true, we denote $q[m]=j$ where j is such that $P[j] \wedge \text{dictionary}(s[j+1], \dots, s[m])$ is true. By the definition of $P[m]$, it is obvious that such a j must exist when $P[m]$ is true. To reconstruct the paper into a sequence of valid words, we just need to output $n, q[n], q[q[n]], \dots, 0$ as segmentation points.

Running time: The length of table of P and q is n and each step when we compute $P[m]$ we need at most m calls of function *dictionary()*. Note the $m \leq n$. Hence the running time is bounded by $O(n^2)$

Remark: Actually this question is exactly the same as the first problem in HW4: Problem 5 in Chapter 6 if we set $\text{quality}(Y_{i+1}, k) = 0$ if $\text{dictionary}(Y_{i+1}k) = \text{false}$ and $\text{quality}(Y_{i+1}, k) = 1$ if $\text{dictionary}(Y_{i+1}k) = \text{true}$.

7) 10 pts

An $n \times n$ array A consists of 1's and 0's such that, in any row of A, all the 1's come before any 0's in that row. Give the most efficient algorithm you can for counting the number of 1's in A.

Algorithm: For each row i, we use binary search to find the index of last element of 1 in the array $A_i[1, \dots, n]$. Suppose the index is a_i . Then the sum of $a_i (0 < i < n+1)$ is the number of 1's in A.

Proof: Assume that in i^{th} row k is the index of last element of 1. Then we have the property that $a_{ij}=1$ for $j < k+1$ and $a_{ij}=0$ for $j > k$ as all 1's come before any 0's in each row. During the binary search, if $a_{im}=1$, then we must have $k \geq m$; $a_{im}=0$, then we must have $k < m$. By doing this, we can finally find the value of k for each row.

Running time: There are n rows and for each row the running time to find the last element which is 1 by binary search is $\log n$. Hence the running time of our algorithm is $O(n \log n)$

Grading policy: 5 points for algorithm. 3 points for proof and 2 points for running time.

Additional Space

Additional Space

CS570

Analysis of Algorithms

Spring 2009

Exam I- Solution

1) 20 pts

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

[TRUE/FALSE]

Given a weighted graph and two nodes, it is possible to list all shortest paths between these two nodes in polynomial time.

False. Not valid for graph with negative weights.

[TRUE/FALSE]

Given a graph $G = (V, E)$ with cost on edges and a set S which is a subset of V , let (u, v) be an edge such that (u, v) is the minimum cost edge between any vertex in S and any vertex in $V - S$. Then, the minimum spanning tree of G must include the edge (u, v) .

True. Refer to textbook, page 145, theorem 4.17.

[TRUE/FALSE]

Let $G = (V, E)$ be a weighted graph and let M be a minimum spanning tree of G . The path in M between any pair of vertices v_1 and v_2 must be a shortest path in G .

False. Counter example: a loop with three nodes A, B and C. The edge costs are A-B: 1, B-C: 1, A-C: 1.5. Edge A-C is not included in M , but the shortest path between A and C is through edge A-C.

[TRUE/FALSE]

$$2n^2 + \theta(n) = \theta(n^2)$$

True.

[TRUE/FALSE]

Kruskal's algorithm can fail in the presence of negative cost edges.

This is False.

[TRUE/FALSE]

For any cycle in a graph, the cheapest edge in the cycle is in a minimum spanning tree.

False. If every edge in a cycle C is very expensive then it is possible that the MST doesn't use any edge in C (including the cheapest edge in C).

[TRUE/FALSE]

In every instance of the stable marriage problem there is a stable matching containing a pair (m, w) such that m is ranked first on the preference list of w and w is ranked first on the preference list of m .

False. Consider the following scenario where there are two men and women each in M and W with the following preferences:

m prefers w to w'

m' prefers w' to w

w prefers m' to m

w' prefers m to m'

[TRUE/FALSE]

Let $T(n)$ be a function obeying the recurrence $T(n) = 5T(n/5) + a$ with initial condition $T(1) = b$, where a and b are positive numbers. Then $T(n) = \Theta(n \log n)$.

False. By the Master Theorem $T(n) = \Theta(n)$. We compare $n \log_5^5$ with the overhead term of $a = O(n^0)$ and find that we are in the case where the former dominates.

[TRUE/FALSE]

If the edges in a connected undirected graph are unit cost, then you can find the MST using BFS.

This is true.

[TRUE/FALSE]

Asymptotically, a running time of n^d is better than $10n^d$

This is False. Asymptotically, both of them are the same Basically within theta of each other

2) 10 pts

Give an algorithm that given as input a sequence of n numbers and a natural number $k < n$, outputs the smallest k numbers of the sequence in $O(n + k \log n)$ time.

Solution: Building a MIN-HEAP takes $O(n)$ time and doing EXTRACT-MIN k times takes $O(k \log n)$ time and gives us the k smallest numbers of the sequence. Hence running time is $O(n + k \log n)$.

3) 10 pts

Arrange the following in the increasing order of asymptotic growth (x is a constant which is greater than 1).

$$x^{\lceil \log n \rceil} \quad x^n \quad n^{10} \quad n^{\lfloor \log n \rfloor^4} \quad n^{\log n} \quad x^{x^n} \quad x^{n^x}$$

Solution:

$$x^{\lceil \log n \rceil} \quad n^{\lfloor \log n \rfloor^4} \quad n^{10} \quad n^{\log n} \quad x^n \quad x^{n^x} \quad x^{x^n}$$

4) 20 pts

You have a sufficient supply of coins, and your goal is to be able to pay any amount using as few coins as possible.

(a) Describe a greedy algorithm to solve the problem when the coins used are of values 1, 5, 10 and 25 cents. Prove the optimality of the algorithm.

Solution: Let k be the number of coins of denomination d_i just enough to be greater than equal to d_{i+1} , where d_{i+1} is the next higher denomination. Then k coins of denomination d_i can be replaced by d_{i+1} and some other coin(s) using less than k coins. One can verify this observation by taking into account the fact that 5 pennies can be replaced by a nickel, 2 nickels can be replaced by a dime and 3 dimes can be replaced by a quarter and a nickel (2 coins).

Let the money to change for be n , $n = 25a + 10b + 5c + d$. a, b, c, d are the numbers of quarters, dimes, nickels and pennies. We have $b < 3$, $c < 2$ and $d < 5$ (or it will violate the conclusion in the observation, e.g., if $b \geq 3$, we can replace 3 dimes by a quarter and a nickel which results in less coins). We prove that when $n \geq 25$, we should always pick a quarter. As $10b+5c+d \leq 10*2+5*1+1*4 = 29$, it means in the optimal solution, the amount of the money of the dimes, nickels and pennies will be no more than 29 cents. So we only need to prove the cases when n is 25, 26, 27, 28 or 29. One can easily verify that in such cases it is always better to choose a quarter. Similarly we can prove that when $10 \leq n < 25$, we should always pick a dime and when $5 \leq n < 10$ we should always pick a nickel.

(b) Show that a greedy algorithm does not yield optimal results if you are using only coins of value 1 cent, 7 cents and 10 cents.

Solution: Counter example: Consider making change for $x = 14$. The greedy algorithm yields $1 \times (10 \text{ cents}) + 4 \times (1 \text{ cents})$. This is 5 coins. However, we can see that $2 \times (7 \text{ cents}) = 14 = x$ also. Here we use 2 coins. Obviously $2 < 5$, so Greedy is sub-optimal in the case of coin denominations 1, 7, and 10.

5) 10 pts

Let T be a minimum spanning tree for G with edge weights given by weight function w . Choose one edge $(x, y) \in T$ and a positive number k , and define the weight function w' by

$$\begin{aligned}w'(u, v) &= w(u, v), \text{ if } (u, v) \neq (x, y) \\w'(u, v) &= w(u, v) - k, \text{ if } (u, v) = (x, y)\end{aligned}$$

Show that T is also a minimum spanning tree for G with edge weights given by w' .

Solution: Proof by contradiction. Note T is a spanning tree for G with weight function w' and $w'(T) = w(T) - K$.

Suppose T is not a MST for G with w' and hence there exists a tree T' which is an MST for G with w' . So $w'(T') < w'(T)$ (*).

We have two cases to analyze. First, if $(x, y) \notin T'$ then $w'(T') = w(T') - k$ and from the above we have $w(T') < w(T)$. Now T' is a spanning tree for G with w which is better than T which is a MST for G with w . A contradiction. Case 2, suppose $(x, y) \in T'$ then $w'(T') = w(T')$ and so by (*) we have $w(T') < w'(T) = w(T) - k$. A similar contradiction again.

Hence T is a MST for G with w' .

6) 15 pts

Stable Matching: Prove that, if all boys have the same list of preferences, and all the girls have the same list preferences, there can be only one stable marriage.

Solution: Proof by contradiction. Suppose there are n boys b_1, b_2, \dots, b_n ; and n girls g_1, g_2, \dots, g_n . Without loss of generality, assume that all the boys prefer g_1 over g_2, g_2 over g_3 , and so on. Likewise all the girls most prefer b_1 and least prefer b_n . It is easy to verify that the pairing $(b_1, g_1), (b_2, g_2), \dots, (b_n, g_n)$ is a stable matching. Suppose that there is another stable matching, including couple (b_i, g_j) for $i \neq j$. If there are multiple such pairs, let i be the smallest such value. Therefore $j > i$, because g_1 to g_{i-1} are engaged to b_1 through b_{i-1} respectively: so j cannot be less than i . By the definition of the problem, everyone is engaged, including g_i , and thus for some $k > i$, our alternate pairing includes the couple (b_k, g_i) . Note that b_i is a more popular boy than b_k , since we chose i to be the smallest possible value. Therefore g_i prefers b_i over her fiance. And because $i < j$, we know b_i prefers g_i over his fiancee. Thus b_i and g_i constitute an unstable couple. But this contradicts our supposition that this alternate matching is stable. So we conclude that there is no alternate stable matching.

7) 15 pts

Give a divide-and-conquer algorithm to find the average of n real numbers. You

should clearly show all steps (divide, conquer, and combine) in your pseudo code.

Analyze complexity of your solution.

Solution:

Divide: divide array up in 2 equal pieces

Conquer: solve the two subproblems recursively and return the average

Combine: Compute the average for the original problem by:

Average = (number of items in sub problem 1 * average for subproblem 1 + number of items in sub problem 2 * average for subproblem 2) / (size of the original problem)

Complexity of Divide is O(1), complexity of Combine is O(1). Master theorem gives you a complexity of O(n) for the solution.

Question 1:

1. False. A bipartite graph cannot contain cycle of odd length, however the number of cycles could be odd. For instance, consider the graph with the edge set $\{(a,b),(b,c),(c,d),(d,a)\}$. The number of edges is 1, which is odd.
2. True. A tree by definition does not contain cycles, and hence does not contain cycles of odd length and is bipartite.
3. True. Let T_1 and T_2 be two distinct MSTs. Hence there exists an edge e that is in T_1 but not T_2 . Add e to T_2 thereby inducing a (unique) cycle C . There are at least two edges of maximum weight in C (for if there were a unique maximum weight edge in C , then it cannot be in an MST thereby contradicting the fact that it is either in T_1 or T_2).
4. True. The claim in question follows from the following two facts: (i) the weight of a tree is linear in its edge weights and (ii) every spanning tree has exactly $|V|-1$ edges. Hence the weight of a spanning tree under the modification goes from $w(T)$ to $2w(T)$. Thus an MST of the original graph remains an MST even after the modification.
5. False. The max element in a min heap with n elements is one of the leafs. Since there are $\Theta(n)$ leafs and the structure of min heap does not carry any information on how the leaf values compare, to find a max element takes at least $\Omega(n)$ time.
6. False. Consider for instance $f(n) = n$ and $g(n)=n^2$. Clearly $n+n^2$ is not in $\Theta(n)$.
7. True.
8. True. The number of spanning trees of a completely connected graph with n vertices is n^{n-2} (Cayley's formula) which grows at least exponentially in n . Even without the knowledge of Cayley's formula, it is easy to derive $\Omega(2^n)$ lower bound.
9. True. If the edge lengths are identical, then BFS does find shortest paths from the source. BFS takes $O(|V|+|E|)$ where as Dijkstra even with a Fibonacci heap implementation takes $O(|E| + |V| \log |V|)$
10. False. The $O()$ notation merely gives an upper bound on the running time. To claim one is faster than other, you need an upper bound for the former and a lower bound (Ω) for the latter.

Question 2:**Algorithm1:**

1. Construct the adjacent list of G^{rev} in order to facilitate the access to the incoming edges to each node.

2. Starting from node t , go through the incoming edges to t one by one to check if the following condition is satisfied:

$$\delta(s, t) == e(u, t) + \delta(s, u), (u, t) \in E$$

3. There must be at least one node u satisfying the above condition, and this u must be a predecessor of node t .
 4. Keep doing the same checking operation recursively on the predecessor node to get the further predecessor node until node s is reached.

Algorithm2:

Run a modified version of Dijkstra algorithm without using the priority queue. Specifically,

1. Set $S=\{s\}$.
 2. While (S does not include t)
 - Check each node u satisfying $u \notin S, (v, u) \in E, v \in S$, if the following condition is satisfied:
$$\delta(s, u) == e(v, u) + \delta(s, v)$$
 - If yes, add u into S , and v is the predecessor of u .
- endWhile

Complexity Analysis:

In either of the above algorithms, each directed edge is checked at most once, the complexity is $O(|V|+|E|)$.

Question 3:

Algorithm:

Sort jobs by starting time so that $s_1 \leq s_2 \leq \dots \leq s_n$.

Define f_1, f_2, \dots, f_n as the corresponding finishing times of the jobs.
 Define d as the number of allocated processor.

Initialize $d = 0$;

For $j = 1$ to n
 If (job j is compatible with processor)
 Allocate processor k to job j .
 Else
 Allocate a new processor $d + 1$;
 Schedule job j to processor $d + 1$;

```
        Update  $d = d + 1$ ;  
EndFor
```

Implementation:

Sort starting time takes time $O(n \log n)$.

Keep the allocated processors in a priority queue (Min-heap), accessed by minimum finishing time.

For each processor k , maintain the finishing time of the last job added.

To check the compatibility: compare the job j 's starting time with the finishing time (key value of the root of the Min-heap) of processor k with last allocated job i . If $s_i > f_i$, allocate processor k to job j ; otherwise, allocate a new processor. The operation corresponds to either changing key value or adding a new node. Either operation takes time $O(\log n)$.

For n jobs, the total time is $O(n \log n)$.

Proof:

Definition: define the “depth” d of a set of jobs to be the maximum number that pass over any single point on the time-line.

Then we have the follow observations: All schedules use $\geq d$ (the depth) processors

- Assume that greedy uses D processors where $D > d$.
- Processor D is used because we needed to schedule a job, say j , that is incompatible with all $D - 1$ other processors.
- Since we sorted by starting time, all these incompatibilities are caused by jobs with start times $\geq s_j$
- The number of such incompatible processors is at most $d - 1$.
- Therefore, $D - 1 \leq d - 1$, a contradiction

Question 4:

- i) The mayor is merely contending that the graph of the city is a strongly-connected graph, denoted as G . Form a graph of the city (intersections become nodes, one-way streets become directed edges). Suppose there are n nodes and m edges. The algorithm is:
 - (1) Choose an arbitrary node s in G , and run BFS from s . If all the nodes are reached the BFS tree rooted at s , then go to step (2), otherwise, the mayor's statement must be wrong. This operation takes time $O(m + n)$

- (2) Reverse the direction of all the edges to get the graph G^{inv} , this step takes time $O(m + n)$
- (3) Do BFS in G^{inv} starting from s . If all the nodes are reached, then the mayor's statement is correct; otherwise, the mayor's statement is wrong. This step takes time $O(m + n)$.

Explanation: BFS on G tests if s can reach all other nodes; BFS on G^{inv} tests if all other nodes reach s . If these two conditions are not satisfied, the mayor's statement is wrong obviously; if these two conditions are satisfied, any node v can reach any node u by going through s .

- ii)** Now the mayor is contending that the intersections which are reachable from the city form a strongly-connected component. Run the first step of the previous algorithm (test to see which nodes are reachable from town hall). Remove any nodes which are unreachable from the town hall, and this is the component which the mayor is claiming is strongly connected. Run the previous algorithm on this component to verify it is strongly connected.

Question 5:

No, the modification does not work. Consider the following directed graph with edge set $\{(a,b),(b,c),(a,c)\}$ all of whose edge weights are -1 . The shortest path from a to c is $((a,b),(b,c))$ with path cost -2 . In this case, $w=1$ and if 2 is added to every edge length before running Dijkstra starting from a , then Dijkstra outputs (a,c) as the shortest path from a to c which is incorrect.

Another way to reason why the modification does not work is that if there were a directed cycle in the graph (reachable from the chosen source) whose total weight is negative, then the shortest paths are not well defined since you could traverse the negative cycle multiple times and make the length arbitrarily small. Under the modification, all edge lengths are positive and hence clearly the shortest paths in the original graph are not preserved.

Remark: Many students who claimed that the modification works made this common mistake. They claimed that since the transformation preserved the relative ordering of the edges by their weights, the algorithm should work. This is incorrect. Even though the edge lengths are increased by the same amount, the path lengths change as a function of the number of edges in them.

Question 6:

- a) We use the $\lceil \frac{n}{2} \rceil$ smaller elements to build a max heap. We use the remaining $\lfloor \frac{n}{2} \rfloor$ elements to build a min heap. The median will be at the root of max heap (We

assume the case of even n , median is $\frac{n}{2}^{th}$ element when sorted in increasing order (5 pts).

Part (a) grading break down:

- Putting the smaller half elements in max heap and the other half in min heap (4 pts).
 - Mention where the median will be at (1 pts).
- b) For a new element x , we compare x with current median (root of max heap). If $x < \text{median}$ we insert x into max heap. Otherwise we insert x into min heap. If $\text{size}(\text{maxHeap}) > \text{size}(\text{minHeap}) + 1$, then we ExtractMax() on max heap and insert the extracted value into the min heap. Also if $\text{size}(\text{minHeap}) > \text{size}(\text{maxHeap})$ we ExtractMin() on min heap and insert the extracted value in max heap. (6 pts).

Part (b) grading break down:

- Correctly identifying which heap to insert the value into (2 pts).
 - Maintaining the size of the heaps correctly (4 pts).
 - If you only got the idea that you have to maintain the size of the heaps equal but don't do it correctly (1 pt).
- c) ExtractMax() on max heap. If after extraction $\text{size}(\text{maxHeap}) < \text{size}(\text{minHeap})$ then we ExtractMin() on min heap and insert the extracted value in max heap (5 pts).

Part (c) grading break down:

- Correctly identifying which root to extract and retaining heap property (1 pt).
- Maintaining the size of heaps correctly (4 pts).
- If you only got the idea that you have to maintain the size of the heaps equal but don't do it correctly (1 pt).

CS570
Analysis of Algorithms
Summer 2008
Exam I

Name: _____
Student ID: _____

____ 4:00 - 5:40 Section ____ 6:00 – 7:40 Section

	Maximum	Received
Problem 1	20	
Problem 2	20	
Problem 3	10	
Problem 4	15	
Problem 5	10	
Problem 6	10	
Problem 7	15	
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]

There exist a perfect matching and a corresponding preference list such that every man is part of an instability, and every woman is part of an instability.

FALSE

[TRUE/FALSE]

A greedy algorithm always returns the optimal solution.

FALSE

[TRUE/FALSE]

The function $100n + 3$ is $O(n^2)$

FALSE

[TRUE/FALSE]

You are given n elements to insert into an empty heap. You could directly call heap insert n times. Or you could first sort the elements and then call heap insert n times. In either case, the asymptotic time complexity is the same.

TRUE

[TRUE/FALSE]

If a problem can be solved correctly using the greedy strategy, there will only be one greedy choice (e.g. “choose the object with highest value to weight ratio”) for that problem that leads to the optimal solution.

FALSE

[TRUE/FALSE]

The Depth First Search algorithm can not be used to test whether a directed graph is strongly-connected.

FALSE

[TRUE/FALSE]

Consider an undirected graph $G=(V, E)$ with non-negative edge weights. Suppose all edge weights are different. Then the edge of maximum weight can be in the minimum spanning tree.

TRUE

[TRUE/FALSE]

Consider a perfect matching S where Mike is matched to Susan. Suppose Mike prefers Winona to Rachel. Then the pair (Mike, Rachel) can never be an instability with respect to S .

FALSE

[TRUE/FALSE]

Consider an undirected graph $G=(V, E)$. Suppose all edge weights are different. Then the shortest path from A to B must be unique.

FALSE

[TRUE/FALSE]

The number of elements in a heap must always be an integer power of 2.

FALSE

2) 20 pts

Given two graphs G and G' that have the same sets of vertices V and edges E , however different weight functions (W and W' respectively) on their edges. Suppose for each graph the weights on the edges are distinct and satisfy the following relation: $W'(e)=W(e)^2$ (Note: all edge weights in G and G' are positive integers) for every edge e of E . Decide whether each of the following statements is true. Give either a short proof or a counter example.

a) The minimum spanning tree of G is the same as the minimum spanning tree of G' .

Solution

This statement is true

Now, since the edges costs are all distinct, the order of the sorted lists of the edges will be the same in G and G' . This is because if $a > b$, then $a^2 > b^2$.

Now, if we run, Prims algorithm, it will consider the edges in the same order in the case of both G and G' . Therefore, since the structure of G and G' are same and also the ordering of the edges (with respect to edge cost) is the same, the same MST will be produced.

b) For a pair of vertices a and b in V , a shortest path between them in G is also a shortest path in G' .

Solution

This statement is false. Hint : Pythagoras's theorem

3) 10 pts

Prove or give a counterexample: An array that is sorted in ascending order is a min-heap.

Solution

Let's assume that the statement is false. Suppose H is a binary tree for the array sorted in ascending order. By the statement, for every element v at a node i in H ,

1. the element w at i 's parent satisfies $\text{key}(w) \leq \text{key}(v)$.
2. the element w at node i and its right sibling w' satisfies $\text{key}(w) \leq \text{key}(w')$.

Min-heap property: for every element v at a node i and the element w at i 's parent satisfies $\text{key}(w) \leq \text{key}(v)$.

H satisfies the min-heap property. Contradiction. Thus, the statement is true.

4) 15 pts

Let $G = (V, E)$ be an undirected graph with maximum degree d . A coloring of G is an assignment to each vertex of G of a “color” such that adjacent vertices have distinct colors. Consider the greedy algorithm that colors as many vertices as possible with color “ j ” before moving to color “ $j+1$.”

Prove or give a counterexample: This greedy algorithm never requires more than $d+1$ colors to color G .

Solution:

Proof: Let $\chi(G)$ denotes number of colors this greedy algorithm requires to color G .

Prove by contradiction.

Assume there exists a graph G' with maximum degree d such that $\chi(G') >= d+2$. Let v be the first vertex in G' colored with color “ $d+2$ ” by the algorithm. Hence all vertices adjacent to v must have used up color “1” through “ $d+1$ ” (otherwise by the algorithm color “ $d+2$ ” would not be used), which means v has at least $d+1$ adjacent vertices, that is, the degree of v is at least $d+1$. But the maximum degree in G' is d . Contradiction. Therefore, this greedy algorithm never requires more than $d+1$ colors to color G .

5) 10 pts

You and your eight-year-old nephew Shrek decide to play a simple card game. At the

beginning of the game, the cards are dealt face up in a long row. Each card is worth a different number of points. After all the cards are dealt, you and Shrek take turns removing either the leftmost or rightmost card from the row, until all the cards are gone. At each turn, you can decide which of the two cards to take. The winner of the game is the player that has collected the most points when the game ends.

Having never taken an algorithms class, Shrek follows the obvious greedy strategy—when it's his turn, Shrek always takes the card with the higher point value. Your task is to find a strategy that will beat Shrek whenever possible. (It might seem mean to beat up on a little kid like this, but Shrek absolutely hates it when grown-ups let him win.)

Prove that you should not also use the greedy strategy. That is, show that there is a game that you can win, but only if you do not follow the same greedy strategy as Shrek.

Solution

Let S be shrek's total point and S' be my total point at i th turn. Let $p_k, p_{k+1}, p_{k+2}, p_{k+3}, p_{k+4}, p_{k+5}$ be the sums of each pair of cards that remained, where $k \geq 1$ and $p_{k+2} > p_{k+3} > p_{k+4} > p_{k+1} > p_k$. By the greedy strategy, the order that Shrek picks the pair of cards would be $p_{k+4}, p_{k+2}, p_{k+1}$ and my order would be p_{k+3} then p_k . The total point at the final turn is that $S + p_{k+4} + p_{k+2} + p_{k+1} > S' + p_{k+3} + p_k$. If I don't use the greedy strategy, the result at the final turn would be $S + p_{k+4} + p_{k+3} + p_{k+1} < S' + p_k + p_{k+2}$. Contradiction. Thus, the greedy strategy is not an optimal solution.

6) 10 pts

Arrange the following functions in increasing order of asymptotic complexity. If $f(n)=\Theta(g(n))$ then put $f=g$. Else, if $f(n)=O(g(n))$, put $f < g$.

$4n^2, \log_2(n), 20n, 2, \log_3(n), n^n, 3^n, n\log(n), 2n, 2^{n+1}, \log(n!)$

Solution: $2 < \log_2(n) = \log_3(n) < 2n = 20n < n\log(n) = \log(n!) < 4n^2 < 2^{n+1} < 3^n < n^n$

7) 15 pts

Prove or give a counterexample: Let G be an undirected, connected, bipartite, weighted graph. If the weight of each edge in G is $+1$, and for every pair of vertices (u,v) in G there is exactly one shortest path, then G is a tree.

Solution

The statement is true.

It is same as the following statement:

G has loop(s) \Rightarrow there exists at least one pair of vertices (u,v) in G with more than one shortest path.

G is a bipartite graph \Rightarrow it can not contain an odd cycle \Rightarrow all the cycle(s) have even number of nodes \Rightarrow all the cycle(s) have length $2n$ (since edge weight is 1)

Take a smallest loop with $2m$ nodes and length $2m$ in G , then any pair of nodes (u,v) that are farthest away have two paths with equal length m . Also, since the loop is smallest, there doesn't not exist any path $< m$ between (u,v) .

Proved.

CS570
Analysis of Algorithms
Summer 2009
Exam I

Name: _____
Student ID: _____

	Maximum	Received
Problem 1	20	
Problem 2	10	
Problem 3	10	
Problem 4	20	
Problem 5	15	
Problem 6	10	
Problem 7	15	
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**] **T**

An array with following sequence of terms [20, 15, 18, 7, 9, 5, 12, 3, 6, 2] is a max-heap.

[**TRUE/FALSE**] **T**

Complexity of the following shortest path algorithms are the same:

- Find shortest path between S and T
- Find shortest path between S and all other points in the graph

[**TRUE/FALSE**] **T**

In an undirected weighted graph with distinct edge weights, both the lightest and the second lightest edge are in the MST.

[**TRUE/FALSE**] **F**

Dijkstra's algorithm works correctly on graphs with negative-cost edges, as long as there are no negative-cost cycles in the graph.

[**TRUE/FALSE**] **T**

Not all recurrence relations can be solved by Master theorem.

[**TRUE/FALSE**] **F**

Mergesort does not need any additional memory space other than that held by the array being sorted.

[**TRUE/FALSE**] **T**

An algorithm with a complexity of $O(n^2)$ could run faster than one with complexity of $O(n)$ for a given problem.

[**TRUE/FALSE**] **F**

There are at least 2 distinct solutions to the stable matching problem--one that is preferred by men and one that is preferred by women.

[**TRUE/FALSE**] **F**

A divide and conquer algorithm has a minimum complexity of $O(n \log n)$ since the height of the recursion tree is always $O(\log n)$.

[**TRUE/FALSE**] **T**

Stable matching algorithm presented in class is based on the greedy technique.

2) 10 pts

a) Arrange the following in the increasing order of asymptotic growth. Identify any ties.

$$\lg n^{10}, 3^n, \lg n^{2n}, 3n^2, \lg n^{\lg n}, 10^{\lg n}, n^{\lg n}, n \lg n$$

If \lg is \log_2 (convention),

$$\lg n^{10} < \lg n^{2n} < \lg n^{2n} = n \lg n < 3n^2 < 10^{\lg n} < n^{\lg n} < 3^n$$

If \lg is \log_{10} (from ISO specification)

$$\lg n^{10} < \lg n^{2n} < 10^{\lg n} < \lg n^{2n} = n \lg n < 3n^2 < n^{\lg n} < 3^n$$

b) Analyze the complexity of the following loops:

```
i- x = 0
  for i=1 to n
    x= x + lg n
  end for
```

O(n)

```
ii- x=0
  for i=1 to n
    for j=1 to lg n
      x = x * lg n
    endfor
  endfor
```

O(n log n)

```
iii- x = 0
  k = "some constant"
  for i=1 to max (n, k)
    x= x + lg n
  end for
```

O(n)

```
iv- x=0
  k = "some constant"
  for i=1 to min(n, k)
    for j=1 to lg n
      x = x * lg n
    endfor
  endfor
```

O(log n)

3) 10 pts

- a) For each of the following recurrences, give an expression for the runtime $T(n)$ if the recurrence can be solved with the Master Theorem. Otherwise, indicate that the Master Theorem does not apply.

$$T(n) = T(n/2) + 2^n$$

$\Theta(2^n)$

$$T(n) = 16T(n/4) + n$$

$\Theta(n^2)$

$$T(n) = 2T(n/2) + n \log n$$

More general case 2

If $F(n) = \Theta((n^{\log_b a})^k (\log n)^k)$ with $k \geq 0$, then $T(n) = \Theta((n^{\log_b a})^k (\log n)^{k+1})$

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

$$T(n) = 2T(n/2) + \log n$$

$\Theta(n)$

$$T(n) = 64T(n/8) - n^2 \log n$$

Does not apply ($f(n)$ is not positive)

4) 20 pts

You work for a small manufacturing company and have recently been placed in charge of shipping items from the factory, where they are produced, to the warehouse, where they are stored. Every day the factory produces n items which we number from 1 to n in the order that they arrive at the loading dock to be shipped out. As the items arrive at the loading dock over the course of the day they must be packaged up into boxes and shipped out. Items are boxed up in contiguous groups according to their arrival order; for example, items 1... 6 might be placed in the first box, items 7...10 in the second, and 11... 42 in the third.

Items have two attributes, *value* and *weight*, and you know in advance the values $v_1 \dots v_n$ and weights $w_1 \dots w_n$ of the n items. There are two types of shipping options available to you:

Limited-Value Boxes: One of your shipping companies offers insurance on boxes and hence requires that any box shipped through them must contain no more than V units of value. Therefore, if you pack items into such a “limited-value” box, you can place as much weight in the box as you like, as long as the total value in the box is at most V .

Limited-Weight Boxes: Another of your shipping companies lacks the machinery to lift heavy boxes, and hence requires that any box shipped through them must contain no more than W units of weight. Therefore, if you pack items into such a “limited-weight” box, you can place as much value in the box as you like, as long as the total weight inside the box is at most W .

Please assume that every individual item has a value at most V and a weight at most W . You may choose different shipping options for different boxes. Your job is to determine the optimal way to partition the sequence of items into boxes with specified shipping options, so that shipping costs are minimized.

Suppose limited-value and limited-weight boxes each cost \$1 to ship. Describe an $O(n)$ greedy algorithm that can determine a minimum-cost set of boxes to use for shipping the n items. Justify why your algorithm produces an optimal solution.

We use a greedy algorithm that always attempts to pack the largest possible prefix of the remaining items that still fits into some box, either limited-value or limited weight. The algorithm scans over the items in sequence, maintaining a running count of the total value and total weight of the items encountered thus far. As long as the running value count is at most V or the running weight count is at most W , the items encountered thus far can be successfully packed into some type of box. Otherwise, if we reach a item j whose value and weight would cause our counts to V and W , then prior to processing item j we first package up the items scanned thus far (up to item $j-1$) into an appropriate box and zero out both counters. Since the algorithm spends only a constant amount of work on each item, its running time is $O(n)$.

Why does the greedy algorithm generate an optimal solution (minimizing the total number of boxes)? Suppose that it did not, and that there exists an optimal solution different from the greedy solution that uses fewer boxes. Consider, among all optimal solutions, one which agrees with the greedy solution in a maximal prefix of its boxes. Let us now examine the sequence of boxes produced by both solutions, and consider the first box where the greedy and optimal solutions differ. The greedy box includes items $i \dots j$ and the optimal box includes items $i \dots k$, where $k < j$ (since the greedy algorithm always places the maximum possible number of items into a box). In the optimal solution, let us now remove items $k+1 \dots j$ from the boxes in which they currently reside and place them in the box we are considering, so now it contains the same set of items as the corresponding greedy box. In so doing, we clearly still have a feasible packing of items into boxes and since the number of boxes has not changed, this must still be an optimal solution; however, it now agrees with the greedy solution in one more box, contradicting the fact that we started with an optimal solution agreeing maximally with the greedy solution.

5) 15 pts

For two unsorted arrays x and y , each of size n , give a divide and conquer algorithm that runs in $O(n)$ time and computes the maximum sum M , as given below:

$$M = \text{Max } (x_i + x_{i+1} - y_i); \quad i=1 \text{ to } n-1$$

Divide: divide problem into 2 equal size subproblems at each step.

Conquer: solve the subproblems recursively until the subproblem become trivial to solve. In this case problem size of 2 is trivial. In that case the solution is the sum of the 2 numbers.

Combine: We need to merge the solutions to the two subproblems S1 and S2. At each step we need to evaluate the following 3 cases:

Case 1: Max is in S1 (subproblem to the left)

Case 2: Max is in S2 (subproblem to the right)

Case 3: Max is on the border of S1 and S2

In case 3, you need to calculate $X(\text{the last element of S1}) + X(\text{the first element of S2}) - Y(\text{the last element of S1})$

6) 10 pts

Suppose we are given an instance of the Shortest Path problem with source vertex s on a directed graph G . Assume that all edges costs are positive and distinct. Let P be a minimum cost path from s to t . Now suppose that we replace each edge cost c_e by its square, c_e^2 , thereby creating a new instance of the problem with the same graph but different costs.

Prove or disprove: P still a minimum-cost $s - t$ path for this new instance.

Let G have edges (s,v) , (v,t) , and (s,t) , when the first two of these edges have cost 3 and the third has cost 5. Then the shortest path is the single edge (s,t) , but after squaring the costs the shortest path would go through v .

7) 15 pts

We are given two arrays of integers $A[1..n]$ and $B[1..n]$, and a number X . Design an algorithm which decides whether there exist $i, j \in \{1, \dots, n\}$ such that $A[i] + B[j] = X$. Your algorithm should run in time $O(n \log n)$.

Sort array B

For int $i = 0$ to n

 Temp = $X - A[i]$

 Do binary search to find an element whose value is equal to $X - A[i]$ in array B

 If it finds return true

End of For

CS570
Analysis of Algorithms
Fall 2008
Exam II

Name: _____
Student ID: _____

____ Monday Section ____ Wednesday Section ____ Friday Section

	Maximum	Received
Problem 1	20	
Problem 2	15	
Problem 3	15	
Problem 4	15	
Problem 5	20	
Problem 6	15	
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.

FALSE [TRUE/FALSE]

For every node in a network, the total flow into a node equals the total flow out of a node.

TRUE [TRUE/FALSE]

Ford Fulkerson works on both directed and undirected graphs.

Because at every augmentation step you just need to find a path from s to t. This can be done in both direct and undirected graphs.

NO [YES/NO]

Suppose you have designed an algorithm which solves a problem of size n by reducing it to a max flow problem that will be solved with *Ford Fulkerson*, however the edges can have capacities which are $O(2^n)$. Is this algorithm efficient?

YES [YES/NO]

Is it possible for a valid flow to have a flow cycle (that is, a directed cycle in the graph, such that every edge has positive flow)?

positive flow cycles don't cause any problems. The flow can still be valid.

FALSE[TRUE/FALSE]

Dynamic programming and divide and conquer are similar in that in each approach the sub-problems at each step are completely independent of one another.

FALSE [TRUE/FALSE]

Ford Fulkerson has pseudo-polynomial complexity, so any problem that can be reduced to Max Flow and solved using *Ford Fulkerson* will have pseudo-polynomial complexity.

For example the edge disjoint paths problem is solved using FF in polynomial time. This is because for some problems the capacity C becomes a function of n or m.

TRUE [TRUE/FALSE]

In a flow network, the value of flow from S to T can be higher than the number of edge disjoint paths from S to T.

FALSE [TRUE/FALSE]

Complexity of a dynamic programming algorithm is equal to the number of unique sub-problems in the solution space.

TRUE [TRUE/FALSE]

In *Ford-Fulkerson's* algorithm, when finding an augmentation path one can use either BFS or DFS.

TRUE [TRUE/FALSE]

When finding the value of the optimal solution in a dynamic programming algorithm one must find values of optimal solutions for all of its sub-problems.

2) 15 pts

Given a sequence of n real numbers $A_1 \dots A_n$, give an efficient algorithm to find a subsequence (not necessarily contiguous) of maximum length, such that the values in the subsequence form a strictly increasing sequence.

Let $A[i]$ represent the i -th element in the sequence.

```
initialize OPT[i] = 1 for all i.  
best = 1  
for(i = 2..n) {  
    for(j = 1..i-1) {  
        if(A[j] < A[i]) {  
            OPT[i] = max( OPT[i], OPT[j] + 1 )  
        }  
        best = max( best, OPT[i] )  
    }  
}  
return best
```

The runtime of the above algorithm is $O(n^2)$

3) 15 pts

Suppose you are given a table with $N \times M$ cells, each having a certain quantity of apples. You start from the upper-left corner and end at the lower right corner. At each step you can go down or right one cell. Give an efficient algorithm to find the maximum number of apples you can collect.

Let the top-left corner be in row 1 column 1, and the bottom-right corner be in row n column m.

Let $A[i,j]$ represent the number of apples at row i column j.

```
initialize OPT[i,j] = 0 for all i,j.  
for(i = 1...n) {  
    for(j = 1...m) {  
        OPT[i,j] = A[i,j] + max( OPT[i-1,j], OPT[i,j-1] )  
    }  
}  
return OPT[n,m]
```

The runtime of the above algorithm is $O(nm)$.

4) 15 pts

Suppose that you are in charge of a large blood bank, and your job is to match donor blood with patients in need. There are n units of blood, and m patients each in need of one unit of blood. Let us assume that the only factor which matters is that the blood type be compatible according to the following rules:

- (a) Patient with type AB can receive types O, A, B, AB (universal recipient)
- (b) Patient with type A can receive types O, A
- (c) Patient with type B can receive types O, B
- (d) Patient with type O can receive type O

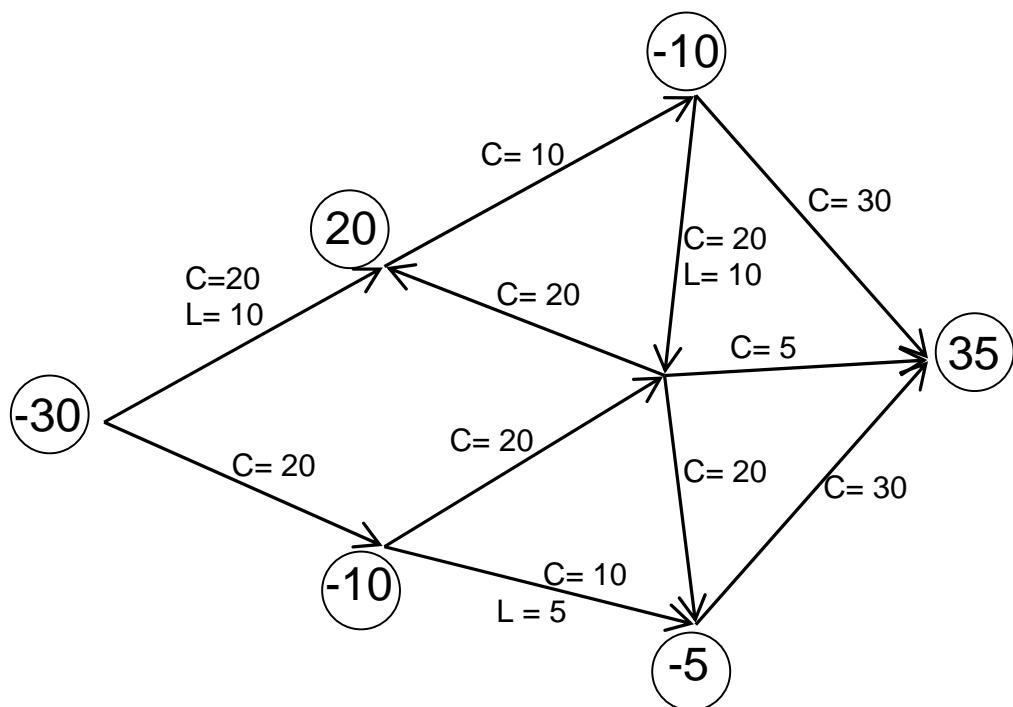
Give a network flow algorithm to find the assignment such that the maximal number of patients receive blood, and prove its correctness.

Given a set of n units of donor blood, and m patients in need of blood, each with some given blood type. We construct a network as follows. There is a source node, and a sink node, n donor nodes d_i , and m patient nodes p_j . We connect the source to every donor node d_i with a capacity of 1. We connect each donor node d_i to every patient node p_j which has blood type compatible with the donor's blood type, with a capacity of 1. We connect each patient node p_j to the sink with capacity 1.

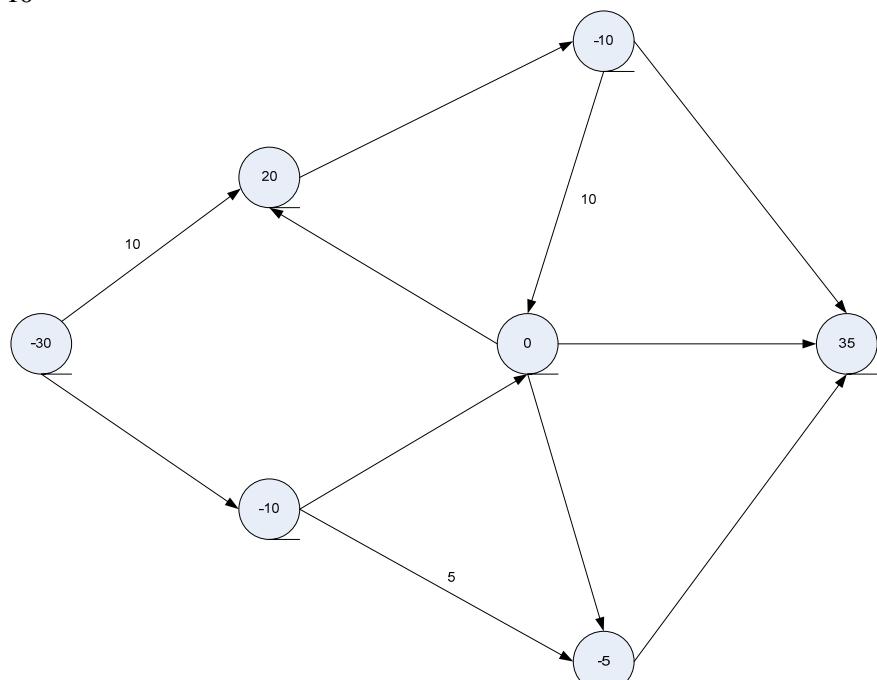
We then find the maximum flow in the network using Ford-Fulkerson. Patient j receives blood from donor i if and only if there is a flow of 1 from d_i to p_j in the maximum flow. The total flow into each donor node d_i is bounded by 1, and the total flow out of each patient node p_j is bounded by capacity 1, and so by conservation of flow, each patient and each donor can only give or receive 1 unit of blood. The types will be compatible by construction of the network.

5) 20 pts

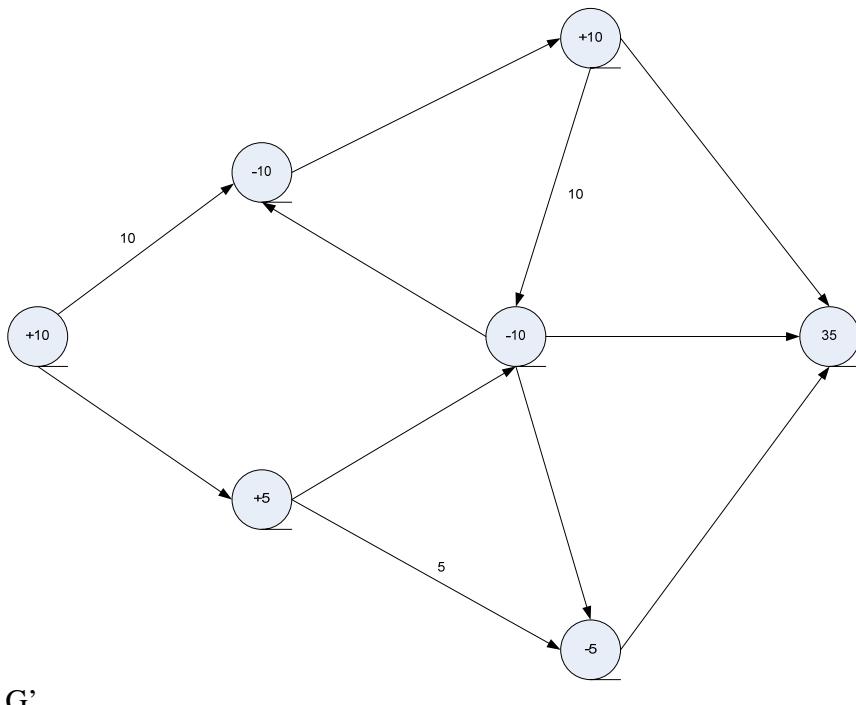
Given the graph below with demands/supplies as indicated below and edge capacities and possible lower bounds on flow marked up on each edge, find a feasible circulation. Show all your work



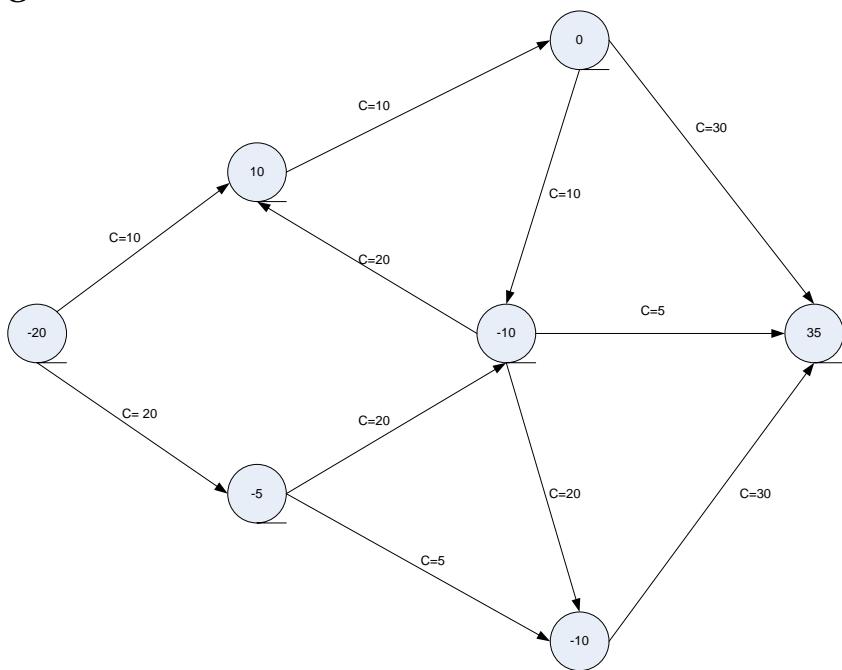
Solution to Q5
f0



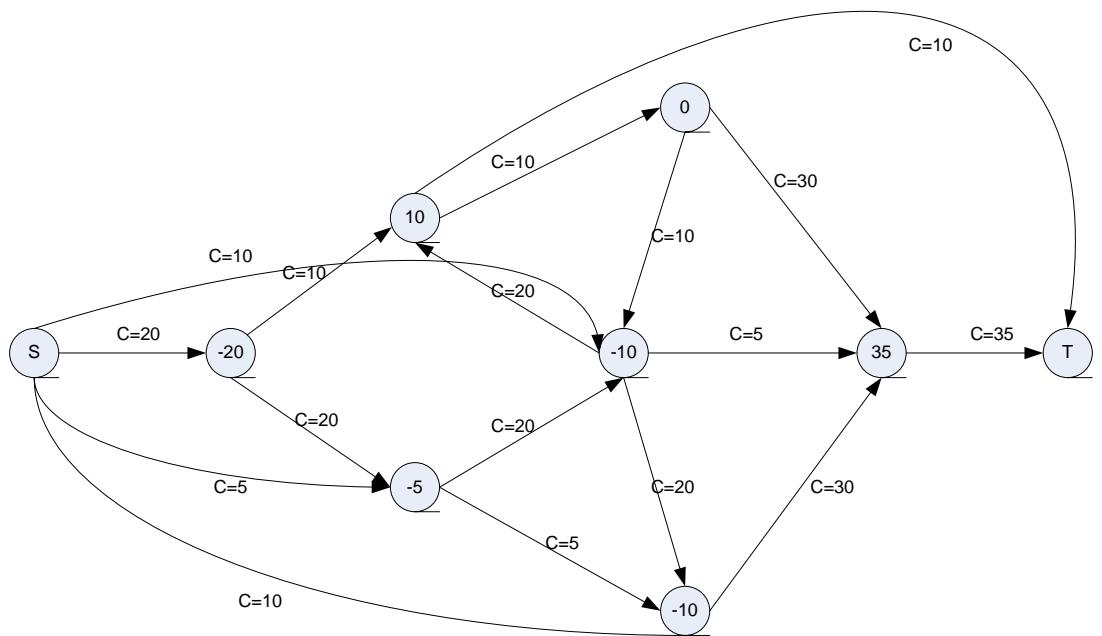
Lv



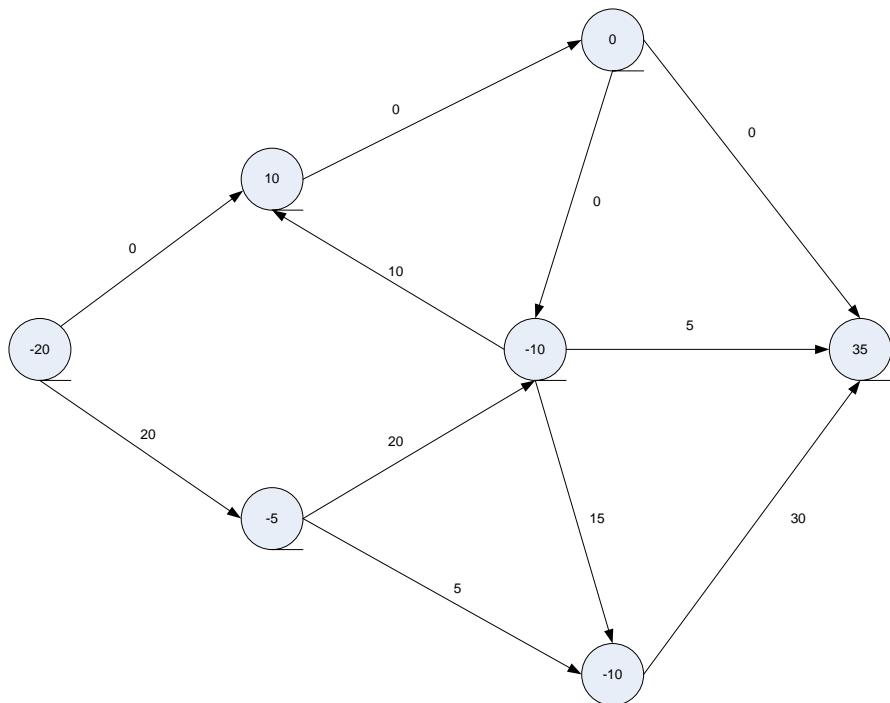
G'



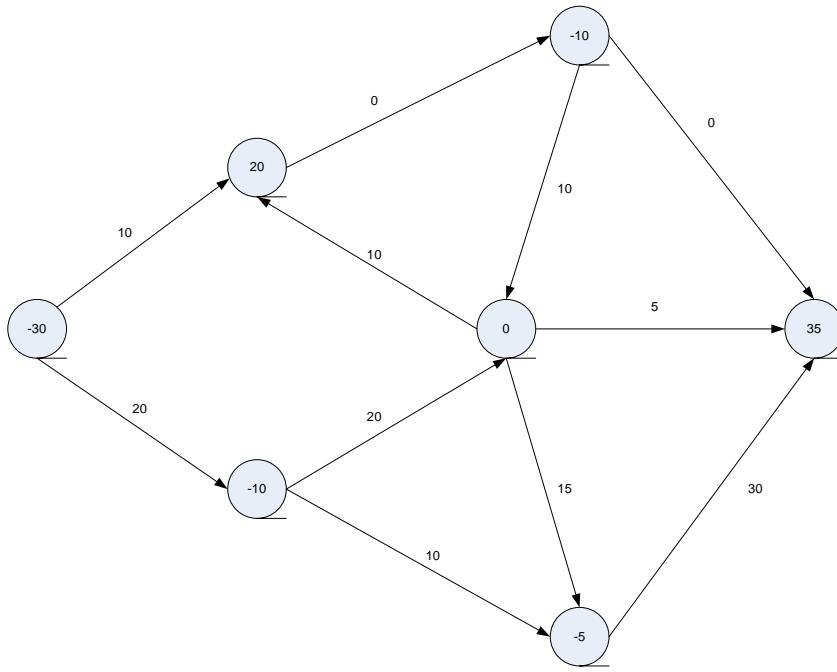
Convert G' to a st network



f1



$$\text{Circulation} = f_0 + f_1$$

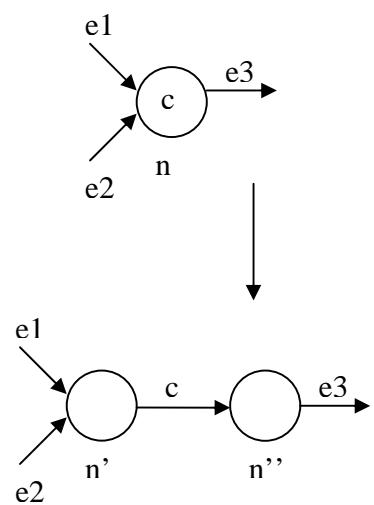


6) 15 pts

Suppose that in addition to each arc having a capacity we also have a capacity on each node (thus if node i has capacity c_i then the maximum total flow which can enter or leave the node is c_i). Suppose you are given a flow network with capacities on both arcs and nodes. Describe how to find a maximum flow in such a network.

Solution:

Assume the initial flow network is G , for any node n with capacity c , decompose it into two nodes n' and n'' , which is connected by the edge (n', n'') with edge capacity c . Next, connect all edges into the node n in G to the node n' , and all edges out of the node n in G out of the node n'' , as shown in the following figure.



After doing this for each node in G , we have a new flow network G' . Just run the standard network flow algorithms to find the maximal flow.

CS570
Analysis of Algorithms
Fall 2010
Exam II

Name: _____
Student ID: _____

____ Monday ____ Friday ____ 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.

[**FALSE**]

If you have non integer edge capacities, then you cannot have an integer max flow.

[**TRUE**]

The maximum value of an s-t flow is equal to the minimum capacity of an s-t cut in the network.

[**FALSE**]

For any graph there always exists an edge such that increasing the capacity on that edge will increase the maximum flow from source s to sink t. (Assume that there is at least one path in the graph from source s to sink t.)

[**FALSE**]

If a problem can be solved using both the greedy method and dynamic programming, greedy will always give you a lower time complexity.

[**FALSE**]

There is a feasible circulation with demands $\{d_v\}$ if $\sum_v d_v = 0$.

[**FALSE**]

The best time complexity to solve the max flow problem is $O(Cm)$ where C is the total capacity of the edges leaving the source and m is the number of edges in the network.

[**TRUE**]

In the Ford–Fulkerson algorithm, choice of augmenting paths can affect the number of iterations.

[**FALSE**]

0-1 knapsack problem can be solved using dynamic programming in polynomial time.

[**TRUE**]

Bellman-Ford algorithm solves the shortest path problem in graphs with negative cost edges in polynomial time.

[**FALSE**]

If a dynamic programming solution is set up correctly, i.e. the recurrence equation is correct and the subproblems are always smaller than the original problem, then the resulting algorithm will always find the optimal solution in polynomial time.

2) 20 pts

You are given an n -by- n grid, where each square (i, j) contains $c(i, j)$ gold coins. Assume that $c(i, j) \geq 0$ for all squares. You must start in the upper-left corner and end in the lower-right corner, and at each step you can only travel one square down or right. When you visit any square, including your starting or ending square, you may collect all of the coins on that square. Give an algorithm to find the maximum number of coins you can collect if you follow the optimal path. Analyze the complexity of your solution.

We will solve the following subproblems: let $dp[i, j]$ be the maximum number of coins that it is possible to collect while ending at (i, j) .

We have the following recurrence: $dp[i, j] = c(i, j) + \max(dp[i - 1, j], dp[i, j - 1])$

We also have the base case that when either $i = 0$ or $j = 0$, $dp[i, j] = c(i, j)$.

There are n^2 subproblems, and each takes $O(1)$ time to solve (because there are only two subproblems to recurse on). Thus, the running time is $O(n^2)$.

3) 20 pts

King Arthur's court had n knights. He ruled over m counties. Each knight i had a quota q_i of the number of counties he could oversee. Each county j , in turn, produced a set S_j of the knights that it would be willing to be overseen by. The King sets up Merlin the task of computing an assignment of counties to the knights so that no knight would exceed his quota, while every county j is overseen by a knight from its set S_j . Show how Merlin can employ the Max-Flow algorithm to compute the assignments. Provide proof of correctness and describe the running time of your algorithm. (You may express your running time using function $F(v, e)$, where $F(v, e)$ denotes the running time of the Max-Flow algorithm on a network with v vertices and e edges.)

We make a graph with $n+m+2$ vertices, n vertices k_1, \dots, k_n corresponding to the knights, m vertices c_1, \dots, c_m corresponding to the counties, and two special vertices s and t .

We put an edge from s to k_i with capacity q_i . We put an edge from k_i to c_j with capacity 1 if county j is willing to be ruled by knight i . We put an edge of capacity 1 from c_j to t .

We now find a maximum flow in this graph. If the flow has value m , then there is a way to assign knight to all counties. Since this flow is integral, it will pick one incoming edge for each county c_j to have flow of 1. If this edge comes from knight k_i , then county j is ruled by knight i .

The running time of this algorithm is $F(n+m+2, \sum_j |S_j| + n + m)$.

4) 20 pts

You are going on a cross country trip starting from Santa Monica, west end of I-10 freeway, ending at Jacksonville, Florida, east end of I-10 freeway. As a challenge, you restrict yourself to only drive on I-10 and only on one direction, in other words towards Jacksonville. For your trip you will be using rental cars, which you can rent and drop off at certain cities along I-10. Let's say you have n such cities on I-10, and assume cities are numbered as increasing integers from 1 to n , Santa Monica being 1 and Jacksonville being n . The cost of rental from a city i to city j ($>i$) is known, $C[i,j]$. But, it can happen that cost of renting from city i to j is higher than the total costs of a series of shorter rentals.

- (A) Give a dynamic programming algorithm to determine the minimum cost of a trip from city 1 to n .
- (B) Analyze running time in terms of n .

Denote $OPT[i]$ to be the rental cost for the optimal solution to go from city i to city n for $1 \leq i \leq n$. Then the solution we are looking for is $OPT[1]$ since objective is to go from city 1 to n . Therefore $OPT[i]$ can be recursively defined as follows.

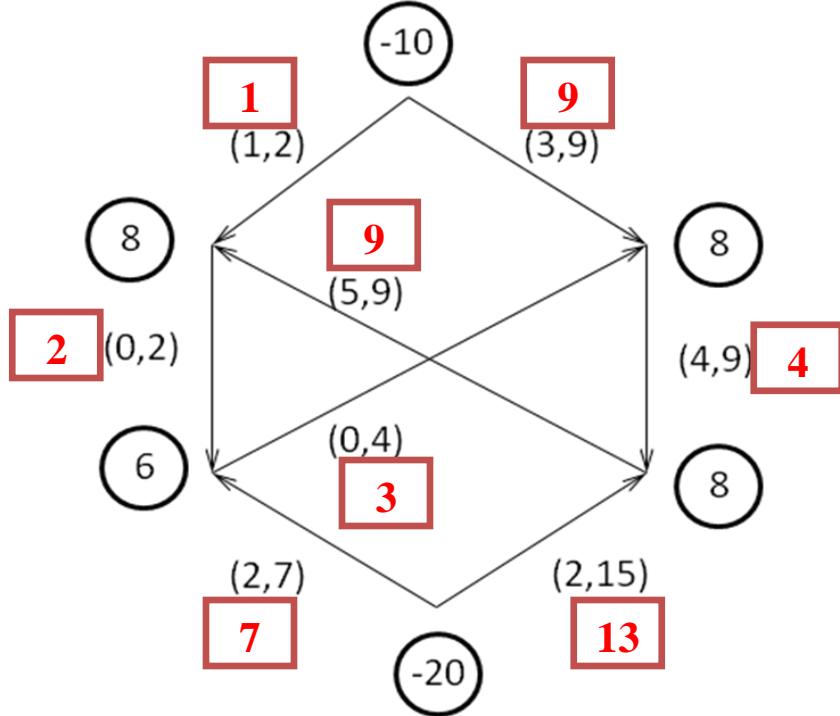
$$OPT[i] = \begin{cases} 0 & \text{if } i = n \\ \min_{i \leq j \leq n} (C[i,j] + OPT[j]) & \text{otherwise} \end{cases}$$

Proof: The car must be rented at the starting city i and then dropped off at another city among $i+1, \dots, n$. In the recurrence we try all possibilities with j being the city where the car is next returned. Furthermore, since $C[i,j]$ is independent from how the subproblem of going from city j, \dots, n is solved, we have the optimal substructure property.

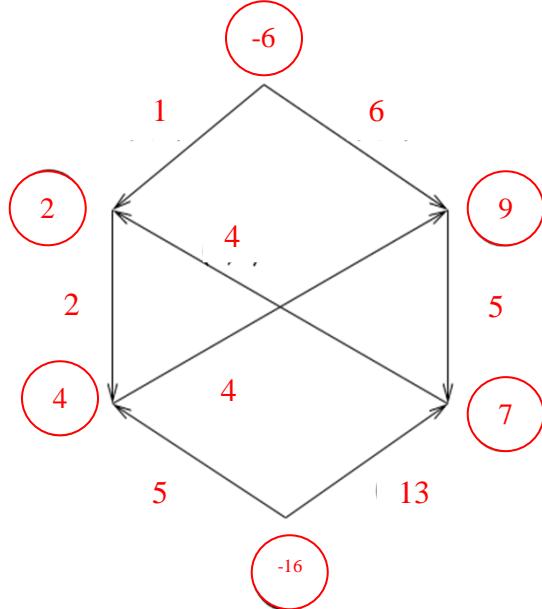
For the time complexity, there are n subproblems to be solved each of which takes linear time, $O(n)$. These subproblems can be computed in the order $OPT[n], OPT[n-1], \dots, OPT[1]$, in a linear fashion. Hence the overall time complexity is $O(n^2)$.

5) 20 pts

Solve the following feasible circulation problem. Determine if a feasible circulation exists or not. If it does, show the feasible circulation. If it does not, show the cut in the network that is the bottleneck. Show all your steps.



First we eliminate the lower bound from each edge:



Then, we attach a super-source s^* to each node with negative demand, and a super-sink t^* to each node with positive demand. The capacities of the edges attached accordingly correspond to the demand of the nodes.

We then seek a maximum s^*-t^* flow. The value of the maximum flow we can get is exactly the summation of positive demands. That is, we have a feasible circulation with the flow values inside boxes shown as above.

CS570
Analysis of Algorithms
Fall 2014
Exam II

Name: _____
Student ID: _____
Email: _____

Thursday Evening Section

	Maximum	Received
Problem 1	20	
Problem 2	16	
Problem 3	16	
Problem 4	16	
Problem 5	16	
Problem 6	16	
Total	100	

Instructions:

1. This is a 2-hr exam. Closed book and notes
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.

1) 20 pts

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

[TRUE/]

$$T(n) = 9T\left(\frac{n}{5}\right) + n \log n \text{ then } T(n) = \theta(n^{\log_5 9})$$

[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.

[FALSE]

Master theorem can always be used to find the complexity of $T(n)$, if it can be written as the recurrence relation $T(n) = aT\left(\frac{n}{b}\right) + f(n)$.

[FALSE]

In the divide and conquer algorithm to compute the closest pair among a given set of points on the plane, if the sorted order of the points on both X and Y axis are given as an added input, then the running time of the algorithm improves to $O(n)$.

[TRUE/]

Maximum value of an s-t flow could be less than the capacity of a given s-t cut in a flow network.

[TRUE/]

One can **efficiently** find the maximum number of edge disjoint paths from s to t in a directed graph by reducing the problem to max flow and solving it using the Ford-Fulkerson algorithm.

[TRUE/]

In a network-flow graph, if the capacity associated with every edge is halved, then the max-flow from the source to sink also reduces by half

[TRUE/]

The 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.

[TRUE/]

Bellman-Ford algorithm can handle negative cost edges even if it runs in a distributed environment using message passing.

[FALSE]

If all edges in a graph have capacity 1, then Ford-Fulkerson runs in linear time.

2) 16 pts

The numbers stored in array $A[1..n]$ represent the values of a function at different points in time. We know that the function behaves the following way:

- $A[1] < A[2] < \dots < A[j]$ $1 < j < n$
- $A[j] > A[j+1] > \dots > A[k]$ $j < k < n$
- $A[k] < A[k+1] < \dots < A[n]$
- $A[n] < A[1]$

Your task is to design a divide-and-conquer algorithm to find the maximum element of the array. Your algorithm must run in better than linear time. You need to provide complexity analysis of your algorithm.

Note: we don't know the exact values of j and k , we only know their range as given above.

Solution: Consider an algorithm $ALG(A, n)$ that takes as input an array A of size n , where array A either satisfies all four conditions above or it satisfies the first two conditions with $k = n$. Also, let $ALG(A, n)$ output the maximum element in A .

Store $l = A[0]$ and $r = A[n]$. $ALG(A, n)$ consists of the following steps

- a) If $n \leq 4$ output the maximum element.
- b) If $A\left[\frac{n}{2}\right] > A\left[\frac{n}{2} + 1\right]$ and $A\left[\frac{n}{2}\right] > A\left[\frac{n}{2} - 1\right]$ then return $A\left[\frac{n}{2}\right]$.
- c) If $A\left[\frac{n}{2}\right] < A\left[\frac{n}{2} + 1\right]$
 - i. If $A\left[\frac{n}{2}\right] > l$ then return $ALG\left(A\left[\frac{n}{2} + 1:n\right], \frac{n}{2}\right)$
 - ii. If $A\left[\frac{n}{2}\right] < r$ then return $ALG\left(A\left[1:\frac{n}{2}\right], \frac{n}{2}\right)$.
- d) If $A\left[\frac{n}{2}\right] < A\left[\frac{n}{2} - 1\right]$ then return $ALG\left(A\left[1:\frac{n}{2}\right], \frac{n}{2}\right)$.

To see why this is correct, observe that step (a) covers the base case for termination of the algorithm, step (b) covers the case when $\frac{n}{2} = j$, step (d) covers the case when $j < \frac{n}{2} \leq k$, step (c)(i) covers the case when $1 < \frac{n}{2} < j$, and finally step (c)(ii) covers the case of $k < \frac{n}{2} < n$.

For complexity analysis, notice that at any stage that is not the final stage of the algorithm, exactly one of step (c)(i), (c)(ii), or (d) is executed so that the associated recurrence is $T(n) = T\left(\frac{n}{2}\right) + O(1)$, implying that $T(n) = O(\log n)$ by Master's Theorem.

3) 16 pts

There are a series of part-time jobs lined up **one after the other**, J_1, J_2, \dots, J_n . For any i^{th} part-time job, you are getting paid M_i amount of money. Also for the i^{th} part-time job, you are also given N_i , which is the number of **immediately** following part-time jobs that you cannot take if you perform that i^{th} job. Give an efficient dynamic programming solution to maximize the amount of money one can make. Also state the runtime of your algorithm.

For $1 \leq i \leq n$, let S_i denote a solution for the set of jobs $\{J_i, \dots, J_n\}$, S_i maximizes the amount of money, and let $\text{opt}(i)$ denote its amount of money.

If S_i chooses to take job J_i , then it has to exclude J_{i+1} through J_{i+N_i} . In this case, its money is M_i plus the money it earns for the set $\{J_{i+N_i+1}, \dots, J_n\}$. Since S_i is optimal for the set $\{J_i, \dots, J_n\}$, S_i restricted to the subset $\{J_{i+N_i+1}, \dots, J_n\}$ has to be optimal. Thus in this case, $\text{opt}(i) = M_i + \text{opt}(i + N_i + 1)$.

If S_i chooses not to take job J_i , then the amount of money one can earn is the money from the set of jobs $\{J_{i+1}, \dots, J_n\}$. Since S_i is optimal for the set $\{J_i, \dots, J_n\}$, S_i restricted to the subset $\{J_{i+1}, \dots, J_n\}$ has to be optimal. Thus in this case, $\text{opt}(i) = \text{opt}(i + 1)$.

We thus have a recurrence for all i , $\text{opt}(i) = \max(\text{opt}(i + 1), \text{opt}(i + N_i + 1) + M_i)$. (We understand that $\text{opt}(\text{index} > n) = 0$).

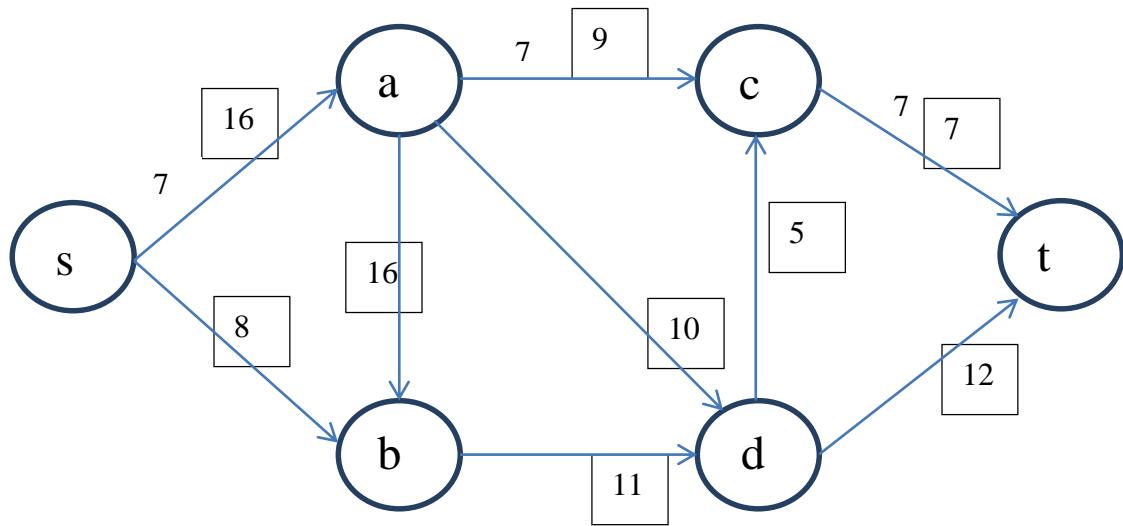
The boundary condition is that $\text{opt}(n) = M_n$ and we start solving recurrence starting with $\text{opt}(n)$, $\text{opt}(n-1)$ and so on until $\text{opt}(1)$. Complexity of this solution is $O(n)$.

4) 16 pts

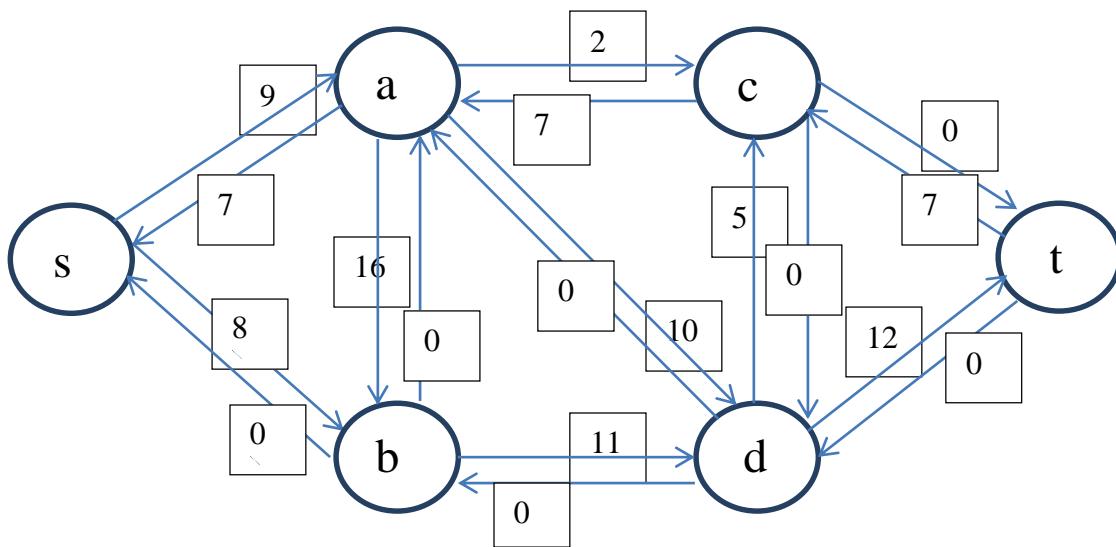
Consider the below flow-network, for which an s-t flow has been computed. The numbers in the boxes give the capacity of each edge, the label next to an edge gives the flow sent on that edge. If no flow is sent on that edge then no label appears in the graph.

- What is the current value of flow? (2 pts)
- Show the residual graph for the corresponding flow-network. (2 pts)
- Calculate the maximum flow in the graph using the Ford-Fulkerson algorithm. You need to show the augmenting path at each step, show the final max flow and a min cut. (12 pts)

Note: extra space provided for this problem on next page/



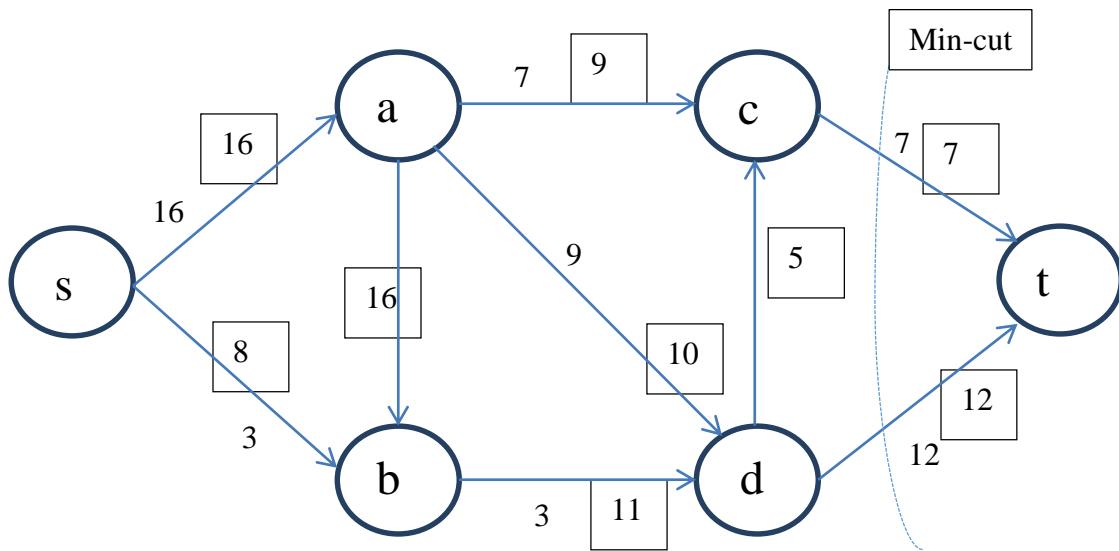
- a) Current value of the flow is 7.
b) Residual graph is shown below



Additional Space for Problem 4

- c) Augmenting paths for one of the maximum flow are
- S->a->d->t, increment in flow along these edges is 9.
 - S->b->d->t, increment in flow along these edges is 3.

Final network-flow graph is shown below. Maximum flow is 19. Edges in the minimum cut are (c,t) and (d,t).



5) 16 pts

You are given an n-by-n grid, where each square (i, j) contains $c(i, j)$ gold coins. Assume that $c(i, j) \geq 0$ for all squares. You must start in the upper-left corner and end in the lower-right corner, and at each step you can only travel in one of the following three ways:

- 1- one square down which costs you 1 gold coin, or
- 2- one square to the right which costs you 2 gold coins, or
- 3- one square diagonally down and to the right which costs you 0 gold coins

When you visit any square, including your starting or ending square, you may collect all of the coins on that square. Give an algorithm to end up with the maximum number of coins and show how you can find the optimal path.

Solution:

let $\text{opt}[i, j]$ be the maximum number of coins that it is possible to collect while ending at (i, j) .

We have the following recurrence for $0 \leq i \leq n-1$, $0 \leq j \leq n-1$.

$$\text{opt}[i, j] = c(i, j) + \max(\text{opt}[i-1, j]-2, \text{opt}[i, j-1]-1, \text{opt}[i-1, j-1])$$

If we assume that borrowing, i.e., negative number of golds are fine the initial conditions are

$$\text{opt}(i, -1) = \text{opt}(j, -1) = -\infty, \text{opt}(0, 0) = c(0, 0)$$

if the assumption is that borrowing is not allowed the initial conditions are

$$\text{opt}(i, -1) = \text{opt}(j, -1) = -\infty, \text{opt}(0, 0) = c(0, 0)$$

$$\text{opt}(1, 0) = \begin{cases} -\infty, & c(0, 0) - 2 < 0 \\ c(0, 0) + c(1, 0) - 2, & c(0, 0) - 2 \geq 0 \end{cases}$$

$$\text{opt}(0, 1) = \begin{cases} -\infty, & c(0, 0) - 1 < 0 \\ c(0, 0) + c(0, 1) - 1, & c(0, 0) - 1 \geq 0 \end{cases}$$

There are n^2 subproblems, and each takes $O(1)$ time to solve. Thus, the running time is $O(n^2)$.

To find the optimal path one can trace back through the recursive formula for different i, j values.

6) 16 pts

You need to transport iron-ore from the mine to the factory. We would like to determine how long it takes to transport. For this problem, you are given a graph representing the road network of cities, with a list of k of its vertices (t_1, t_2, \dots, t_k) which are designated as factories, and one vertex S (the iron-ore mine) where all the ore is present. We are also given the following:

- Road Capacities (amount of iron that can be transported per minute) for each road (edges) between the cities (vertices).
- Factory Capacities (amount of iron that can be received per minute) for each factory (at t_1, t_2, \dots, t_k)

Give a polynomial-time algorithm to determine the minimum amount of time necessary to transport and receive all the iron-ore at factories, say C amount.

Here we want to define a network-flow graph with a source, sink, and capacities on the edges. In the given graph we already have a source S (the iron-ore mine) vertex. Add a new sink vertex T to the graph, and add an edge between each of the factory to T . On each of this newly added edge, set the factory capacity given for that factory. On the remaining edges set the capacities given by the road capacities.

We then run any polynomial-time max flow algorithm on the resulting graph; because it is polynomial in size (has only one additional vertex and at most n additional edges), the resulting runtime to compute max-flow (F) is polynomial.

Now we have the maximum rate at which we can transport the iron-ore. So it takes minimum of $\frac{C}{F}$ time to transport and receive all the iron-ore at factories.

Additional Space

CS570
Analysis of Algorithms
Fall 2014
Exam II

Name: _____
Student ID: _____
Email: _____

Wednesday Evening Section

	Maximum	Received
Problem 1	20	
Problem 2	16	
Problem 3	16	
Problem 4	20	
Problem 5	16	
Problem 6	12	
Total	100	

Instructions:

1. This is a 2-hr exam. Closed book and notes
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.

1) 20 pts

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

[/FALSE]

If an iteration of the Ford-Fulkerson algorithm on a network places flow 1 through an edge (u, v) , then in every later iteration, the flow through (u, v) is at least 1.

[TRUE/]

For the recursion $T(n) = 4T(n/3) + n$, the size of each subproblem at depth k of the recursion tree is $n/3^{k-1}$.

[/FALSE]

For any flow network G and any maximum flow on G , there is always an edge e such that increasing the capacity of e increases the maximum flow of the network.

[/FALSE]

The asymptotic bound for the recurrence $T(n) = 3T(n/9) + n$ is given by $\Theta(n^{1/2} \log n)$.

[/FALSE]

Any Dynamic Programming algorithm with n subproblems will run in $O(n)$ time.

[/FALSE]

A pseudo-polynomial time algorithm is always slower than a polynomial time algorithm.

[TRUE/]

The sequence alignment algorithm can be used to find the longest common subsequence between two given sequences.

[/FALSE]

If a dynamic programming solution is set up correctly, i.e. the recurrence equation is correct and each unique sub-problem is solved only once (memoization), then the resulting algorithm will always find the optimal solution in polynomial time.

[TRUE/]

For a divide and conquer algorithm, it is possible that the divide step takes longer to do than the combine step.

[TRUE/]

Maximum value of an $s - t$ flow could be less than the capacity of a given $s - t$ cut in a flow network.

2) 16 pts

Recall the Bellman-Ford algorithm described in class where we computed the shortest distance from all points in the graph to t. And recall that we were able to find all shortest distance to t with only $O(n)$ memory.

How would you extend the algorithm to compute both the shortest distance and to find the actual shortest paths from all points to t with only $O(n)$ memory?

We need an array of size n to hold a pointer to the neighbor that gives us the shortest distance to t. Initially all pointers are set to Null. Whenever a node's distance to t is reduced, we update the pointer for that node and point it to the node that is giving us a lower distance to t. Once all shortest distances are computed, to find a path from any node v to t, one can simply follow these pointers to reach t on the shortest path.

3) 16 pts

During their studies, 7 friends (Alice, Bob, Carl, Dan, Emily, Frank, and Geoffrey) live together in a house. They agree that each of them has to cook dinner on exactly one day of the week. However, assigning the days turns out to be a bit tricky because each of the 7 students is unavailable on some of the days. Specifically, they are unavailable on the following days (1 = Monday, 2 = Tuesday, ..., 7 = Sunday):

- Alice: 2, 3, 4, 5
- Bob: 1, 2, 4, 7
- Carl: 3, 4, 6, 7
- Dan: 1, 2, 3, 5, 6
- Emily: 1, 3, 4, 5, 7
- Frank: 1, 2, 3, 5, 6
- Geoffrey: 1, 2, 5, 6

Transform the above problem into a maximum flow problem and draw the resulting flow network. If a solution exists, the flow network should indicate who will cook on each day; otherwise it must show that a feasible solution does not exist

Solution:

I will use the initials of each person's name to refer to them in this solution.

Construct a graph $G = (V, E)$. V consists of 1 node for each person (let us denote this set by $P = \{A, B, C, D, E, F, G\}$), 1 node for each day of the week (let's call this set $D = \{1, 2, 3, 4, 5, 6, 7\}$), a source node s , and a sink node t . Connect s to each node p in P by a directed (s, p) edge of unit capacity. Similarly, connect each node d in D to t by a directed (d, t) edge of unit capacity. Connect each node p in P by a directed edge of unit capacity to those nodes in D when p is **available** to cook. This completes our construction of the flow network. I am omitting the actual drawing of G here.

Finding a max-flow of value 7 in G translates to finding a feasible solution to the allocation of cooking days problem. Since there can be at most unit flow coming into any node p in P , a maximum of unit flow can leave it. Similarly, at most a flow of value 1 can flow into any node d in D because a maximum of unit flow can leave it. Thus, a max-flow of value 7 means that there exists a flow-carrying s - p - d - t path for each p and d . Any (p, d) edge with unit flow indicates that person p will cook on day d .

The following lists one possible max-flow of value 7 in G :

Send unit flow on each (s, p) edge and each (d, t) edge. Also send unit flow on the following (p, d) edges: $(A, 6)$, $(B, 5)$, $(C, 1)$, $(D, 4)$, $(E, 2)$, $(F, 7)$, $(G, 3)$

4) 20 pts

Suppose that there are n asteroids that are headed for earth. Asteroid i will hit the earth in time t_i and cause damage d_i unless it is shattered before hitting the earth, by a laser beam of energy e_i . Engineers at NASA have designed a powerful laser weapon for this purpose. However, the laser weapon needs to charge for a duration ce before firing a beam of energy e . Can you design a dynamic programming based pseudo-polynomial time algorithm to decide on a firing schedule for the laser beam to minimize the damage to earth? Assume that the laser is initially uncharged and the quantities c, t_i, d_i, e_i are all positive integers. Analyze the running time of your algorithm. You should include a brief description/derivation of your recurrence relation. Description of recurrence relation = 8pts, Algorithm = 6pts, Run Time = 6pts

Solution 1 (Assuming that the laser retains energy between firing beams): Sort all asteroids by the time t_i . Label the asteroids from 1 to n and without loss of generality assume $t_1 < t_2 < \dots < t_n$. Also assume that if asteroid i is destroyed then it is done exactly at time t_i (if the laser continuously accumulates energy then the destruction order of the asteroids does not change even if i^{th} asteroid is shot down before time t_i).

Define $OPT(i, T)$ as the minimum possible damage caused to earth due to asteroids $i, i + 1, \dots, n$ if $\frac{T}{c}$ energy is left in the laser just before time t_i . We want the solution corresponding to $OPT(1, t_1)$. If $T \geq ce_i$ and the i^{th} asteroid is destroyed then $OPT(i, T) = OPT(i + 1, T - ce_i + t_{i+1} - t_i)$, otherwise $OPT(i, T) = d_i + OPT(i + 1, T + t_{i+1} - t_i)$. Hence,

$$OPT(i, T) = \begin{cases} d_i + OPT(i + 1, T + t_{i+1} - t_i), & T < ce_i \\ \min\{d_i + OPT(i + 1, T + t_{i+1} - t_i), OPT(i + 1, T - ce_i + t_{i+1} - t_i)\}, & T \geq ce_i \end{cases}$$

Boundary condition: $OPT(n, T) = 0$ if $T \geq ce_n$ and $OPT(n, T) = d_n$ if $T < ce_n$, since if there is enough energy left to destroy the last asteroid then it is always beneficial to do so. Furthermore, $T \leq t_n$ since a maximum of $\frac{t_n}{c}$ energy is accumulated by the laser before the last asteroid hits the earth and $T \geq 0$ since the left over energy in the laser is always non-negative.

Algorithm:

- i. Initialize $OPT(n, T)$ according to the boundary condition above for $0 \leq T \leq t_n$.
- ii. For each $n - 1 \geq i \geq 1$ and $0 \leq T \leq t_n$ populate $OPT(i, T)$ according to the recurrence defined above.
- iii. Trace forward through the two dimensional OPT array starting at $(1, t_1)$ to determine the firing sequence. For $1 \leq i \leq n - 1$, destroy the i^{th} asteroid with $\frac{T}{c}$ energy left if and only if $OPT(i, T) = OPT(i + 1, T - ce_i + t_{i+1} - t_i)$. Destroy the n^{th} asteroid only if $T \geq ce_n$.

Complexity: Step (i) initializes t_n values each taking constant time. Step (ii) computes $(n - 1)t_n$ values, each taking one invocation of the recurrence and hence is done in $O(nt_n)$ time. Trace back takes $O(n)$ time since the decision for each i takes constant time. Initial sorting takes $O(n \log n)$ time. Thus overall complexity is $O(nt_n + n \log n)$.

Solution 2 (Assuming that the laser does not retain energy left over after firing and if asteroid i is destroyed then it is done exactly at time t_i): Sort all asteroids by the time t_i . Label the asteroids from 1 to n and without loss of generality assume $t_1 < t_2 < \dots < t_n$. In contrast to Solution 1, the destroying sequence will change if we are free to destroy asteroid i before time t_i (this case is not solved here).

Define $OPT(i)$ to be the minimum possible damage caused to earth due to the first i asteroids. We want the solution corresponding to $OPT(n)$. If the i^{th} asteroid is not destroyed either by choice or because $t_i < ce_i$ then $OPT(i) = d_i + OPT(i - 1)$. On the other hand, if the i^{th} asteroid is destroyed ($t_i \geq ce_i$ is necessary) then none of the asteroids arriving between times $t_i - ce_i$ and t_i (both exclusive) can be destroyed. Letting $p[i]$ denote the largest positive integer such that $t_{p[i]} \leq t_i - ce_i$ (and $p[i] = 0$ if no such integer exists), we have $OPT(i) = OPT(p[i]) + \sum_{j=p[i]+1}^{i-1} d_j$ if $p[i] \leq i - 2$ and $OPT(i) = OPT(i - 1)$ if $p[i] = i - 1$. Hence for $i \geq 1$,

$$OPT(i) = \begin{cases} OPT(i - 1), & t_i \geq ce_i \text{ and } p[i] = i - 1 \\ d_i + OPT(i - 1), & t_i < ce_i \\ \min \left\{ d_i + OPT(i - 1), OPT(p[i]) + \sum_{j=p[i]+1}^{i-1} d_j \right\}, & t_i \geq ce_i \text{ and } p[i] \leq i - 2 \end{cases}$$

Boundary condition: $OPT(0) = 0$ since no asteroids means no damage.

Algorithm:

- i. Form the array p element-wise. This is done as follows.
 - a. Set $p[1] = 0$.
 - b. For $2 \leq i \leq n$, binary search for $t_i - ce_i$ in the sorted array $t_1 < t_2 < \dots < t_n$. If $t_i - ce_i < t_1$ then set $p[i] = 0$ else record $p[i]$ as the index such that $t_{p[i]} \leq t_i - ce_i < t_{p[i]+1}$.
- ii. Form array D to store cumulative sum of damages. Set $D[0] = 0$ and $D[j] = D[j - 1] + d_j$ for $1 \leq j \leq n$.
- iii. Set $OPT(0) = 0$ and for $1 \leq i \leq n$ populate $OPT(i)$ according to the recurrence defined above, computing $\sum_{j=p[i]+1}^{i-1} d_j$ as $D[i - 1] - D[p[i]]$.
- iv. Trace back through the one dimensional OPT array starting at $OPT(n)$ to determine the firing sequence. Destroy the first asteroid if and only if $OPT(1) = 0$. For $2 \leq i \leq n$, destroy the i^{th} asteroid if and only if either $OPT(i) = OPT(i - 1)$ with $p[i] = i - 1$ or $OPT(i) = OPT(p[i]) + D[i - 1] - D[p[i]]$ with $p[i] \leq i - 2$.

Complexity: initial sorting takes $O(n \log n)$. Construction of array p takes $O(\log n)$ time for each index and hence a total of $O(n \log n)$ time. Forming array D is done in $O(n)$ time. Using array D , $OPT(i)$ can be populated in constant time for each $1 \leq i \leq n$ and hence step (iii) takes $O(n)$ time. Trace back takes $O(n)$ time since the decision for each i takes constant time. Therefore, overall complexity is $O(n \log n)$.

5) 16 pts

Consider a two-dimensional array $A[1:n, 1:n]$ of integers. In the array each row is sorted in ascending order and each column is also sorted in ascending order. Our goal is to determine if a given value x exists in the array.

- a. One way to do this is to call binary search on each row (alternately, on each column). What is the running time of this approach? [2 pts]
 - b. Design another divide-and-conquer algorithm to solve this problem, and state the runtime of your algorithm. Your algorithm should take strictly less than $O(n^2)$ time to run, and should make use of the fact that each row and each column is in sorted order (i.e., don't just call binary search on each row or column). State the run-time complexity of your solution.
- a) $O(n \log n)$.
- b) Look at the middle element of the full matrix. Based on this, you can either eliminate $A[1.. \frac{n}{2}, 1.. \frac{n}{2}]$ or $A[\frac{n}{2}..n, \frac{n}{2}..n]$. If x is less than middle element then you can eliminate $A[\frac{n}{2}..n, \frac{n}{2}..n]$. If x is greater than middle element then you can eliminate $A[1.. \frac{n}{2}, 1.. \frac{n}{2}]$. You can then recursively search in the remaining three $\frac{n}{2} \times \frac{n}{2}$ matrices. The total runtime is $T(n) = 3T(\frac{n}{2}) + O(1)$, $T(n) = O(n^{\log_2 3})$.

6) 12 pts

Consider a divide-and-conquer algorithm that splits the problem of size n into 4 sub-problems of size $n/2$. Assume that the divide step takes $O(n^2)$ to run and the combine step takes $O(n^2 \log n)$ to run on problem of size n . Use any method that you know of to come up with an upper bound (as tight as possible) on the cost of this algorithm.

Solution: Use the generalized case 2 of Master's Theorem. For $T(n) = aT\left(\frac{n}{b}\right) + \Theta(n^{\log_b a} \log^k n)$ with $k \geq 0$, we have $T(n) = \Theta(n^{\log_b a} \log^{k+1} n)$.

The divide and combine steps together take $O(n^2 \log n)$ time and the worst case is that they actually take $\Theta(n^2 \log n)$ time. Hence the recurrence for the given algorithm is $T(n) = 4T\left(\frac{n}{2}\right) + \Theta(n^2 \log n)$ in the worst case. Comparing with the generalized case, $a = 4, b = 2, k = 1$ and so $T(n) = \Theta(n^2 \log^2 n)$. Since this expression for $T(n)$ is the worst case running time, an upper bound on the running time is $O(n^2 \log^2 n)$.

Additional Space

CS570
Analysis of Algorithms
Spring 2007
Exam 2

Name: _____
Student ID: _____

	Maximum	Received
Problem 1	20	
Problem 2	10	
Problem 3	20	
Problem 4	20	
Problem 5	20	
Problem 6	5	
Problem 7	5	

Note: The exam is closed book closed notes.

1) 20 pts

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

[**TRUE/FALSE**] True

Max flow problems can in general be solved using greedy techniques.

[**TRUE/FALSE**] False

If all edges have unique capacities, the network has a unique minimum cut.

[**TRUE/FALSE**] True

Flow f is maximum flow if and only if there are no augmenting paths.

[**TRUE/FALSE**] True

Suppose a maximum flow allocation is known. Increase the capacity of an edge by 1 unit. Then, updating a max flow can be easily done by finding an augmenting path in the residual flow graph.

[**TRUE/FALSE**] False

In order to apply divide & conquer algorithm, we must split the original problem into at least half the size.

[**TRUE/FALSE**] True

If all edge capacities in a graph are integer multiples of 5 then the maximum flow value is a multiple of 5.

[**TRUE/FALSE**] False

If all directed edges in a network have distinct capacities, then there is a unique maximum flow.

[**TRUE/FALSE**] True

Given a bipartite graph and a matching pairs, we can determine if the matching is maximum or not in $O(V+E)$ time

[**TRUE/FALSE**] False

Maximum flow problem can be efficiently solved by dynamic programming

[**TRUE/FALSE**] True

The difference between dynamic programming and divide and conquer techniques is that in divide and conquer sub-problems are independent

2) 10pts

Give tight bound for the following recursion

a) $T(n)=T(n/2)+T(n/4)+n$

A simple method is that it is easy to see that $T(n)=4n$ satisfy the recursive relation.
Hence $T(n)=O(n)$

b) $T(n) = 2T(n^{0.5})+\log n$

Let $n=2^k$ and $k=2^m$, we have $T(2^k)=2T(2^{k/2})+k$ and $T(2^{k/2})=2T(2^{k/4})+k/2$
Hence $T(2^k)=2(2T(2^{k/4})+k/2)+k=2^2 T(2^{k/4})+2k=2^3 T(2^{k/8})+3k=\dots=2^m T(1)+mk$
 $=2^m T(1)+m2^m$. Note that $n=2^k$ and $k=2^m$, hence $m=\log \log n$. We have $T(n)=T(1)\log n+(\log \log n)*(\log n)$.
Hence $T(n)=(\log \log n)*(\log n)$

3) 20 pts

Let $A = (a_0, a_1, \dots, a_{n-1})$ be an array of n positive integers.

a- Present a divide-and-conquer algorithm which computes the sum of all the even integers a_i in $A(1..n-1)$ where $a_i > a_{i-1}$. Solutions other than divide and conquer will receive no credit.

Algorithm: Compute-Even-Integers(l, r)

if $r=l+1$

 if(($a_r \% 2 = 0$) and ($a_r > a_l$))

 return a_r

 else

 return 0

else

 let $m = \left\lfloor \frac{l+r}{2} \right\rfloor$

 Let $S = \text{Compute-Even-Integers}(l, m) + \text{Compute-Even-Integers}(m, r)$

 If $a_m \% 2 = 0$ and $a_m > a_{m-1}$

$S = S + a_m$

To computes the sum of all the even integers a_i in $A(1..n-1)$ where $a_i > a_{i-1}$, invoke Compute-Even-Integers($0, n-1$)

b- Show a recurrence which represents the number of additions required by your algorithm.

$$T(n) = 2T(n/2) + c$$

c- Give a tight asymptotic bound for the number of additions required by your algorithm.

By master theorem, it is easy to see that $T(n)=O(n)$

d- Discuss how your approach would compare in practice to an iterative solution of the problem.

In practice, both methods are $O(n)$ algorithm. Their complexity is the same.

4) 20 pts

Families 1.....N go out for dinner together. To increase their social interaction, no two members of the same family use the same table. Family j has $a(j)$ members.

There are M tables. Table j can seat $b(j)$ people. Find a valid seating assignment if one exists.

We first construct a bipartite graph G whose nodes are the families $f_i (i=1, \dots, N)$ and the tables $t_j (j=1, \dots, M)$. We add edge $e_{ij} = (f_i, t_j)$ in the graph for $i=1, \dots, N$ and $j=1, \dots, M$ and set $e_{ij}=1$. Then we add a source s and sink t in G. For each family f_i , we add $e=(s, f_i)$ in the graph and set $c_e=a(i)$. For each table t_j , we add $e=(t_j, t)$ and set $c_e=b(j)$. After building the graph G for the original problem, we find the maximum s-t-flow value v in graph G by Fulkerson algorithm. If v is $a(1)+\dots+a(N)$, then we make the seating assignment such that no two members of the same family use the same table, otherwise we can not.

To prove the correctness of the algorithm, we prove that no two members of the same family use the same table if and only if the value of the maximum value of an s-t flow in G is $a(1)+\dots+a(N)$:

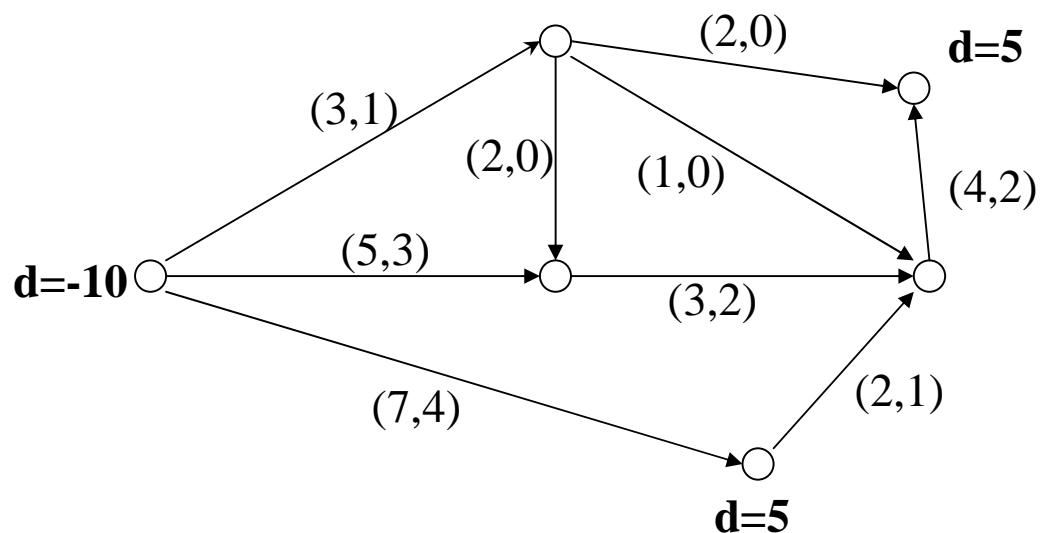
First if no two members of the same family use the same table, it is easy to see that this flow meets all capacity constraints and has value $a(1)+\dots+a(N)$.

For the converse direction, assume that there is an s-t flow of value $a(1)+\dots+a(N)$. The construction given in the algorithm makes sure that there is at most one member in family j sitting in the table j as the capacity of the flow goes from f_i to t_j is 1. So we only need to make sure that all families are seated. Note that $\{s\}$, $V-\{s\}$ is an s-t cut with value $a(1)+\dots+a(N)$, so the flow must saturate all edges crossing the cut. Therefore, all families must be sitting in some tables. This completes the correctness proof.

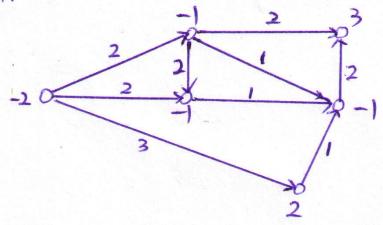
Running time: The time complexity of constructing the graph is $O(MN)$. The number of edges in the graph is $MN+M+N$, and $v(f)=a(1)+\dots+a(N)$. Let $T=a(1)+\dots+a(N)$, then the running time applying Ford-Fulkerson algorithm is $O(MNT)$.

5) 20 pts

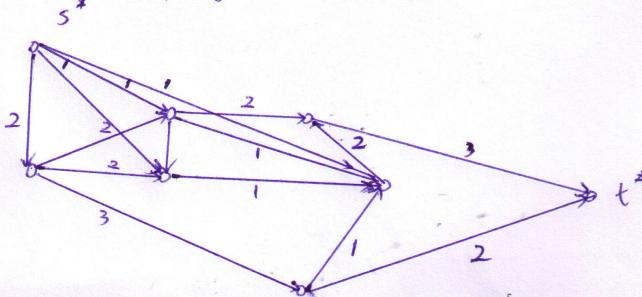
Determine if there is a feasible circulation in the below network. If so, determine the circulation, if not show where the bottlenecks are. The numbers in parentheses are (*lowerbound, upperbound*) on flow.



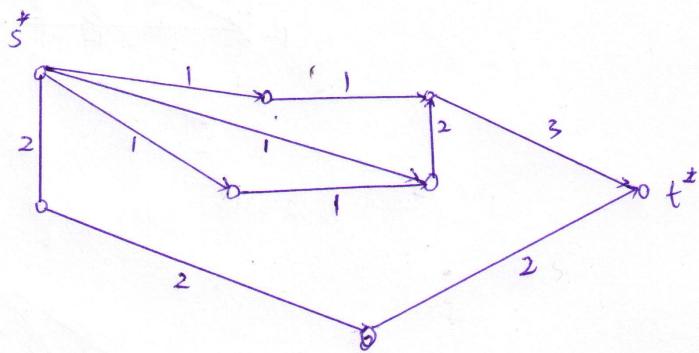
Step 1:



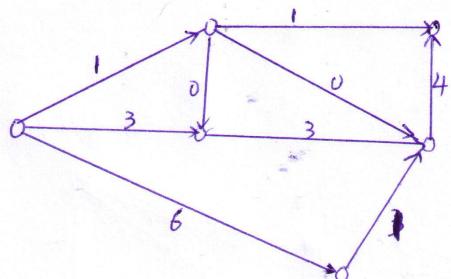
Step 2: the corresponding max-flow problem



Step 3. Solving the max-flow problem and the result is as follows:

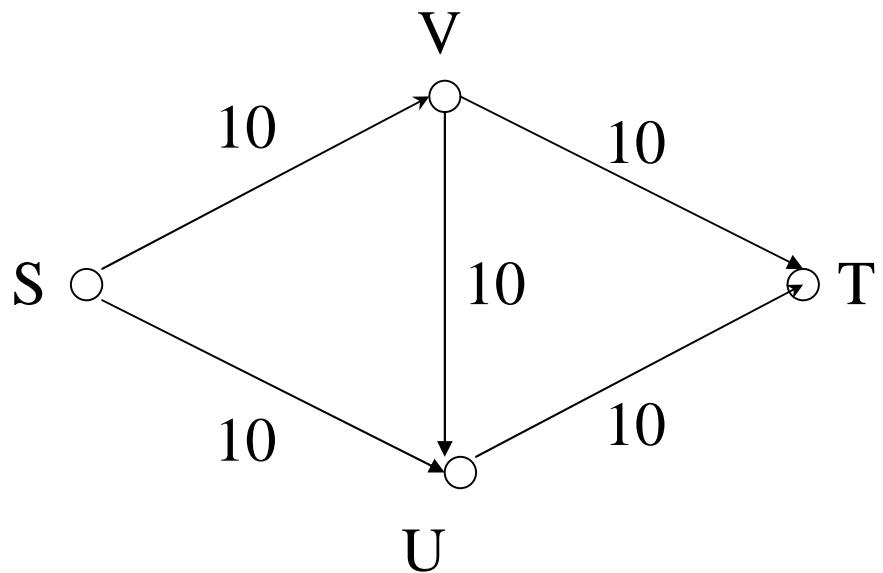


Step 4. The circulation is



6) 5 pts

List all minimum s-t cuts in the flow network pictured below. Capacity of every edge is equal to 10.



$$C1 = \{\{S\}, \{V, U, T\}\}$$

$$C2 = \{\{S, U\}, \{V, T\}\}$$

$$C3 = \{\{S, U, V\}, \{T\}\}$$

7) 5 pts

Show an example of a graph in which poor choices of augmenting paths can lead to exactly C iterations of the Ford-Fulkerson algorithm before achieving max flow. (C is the sum of all edge capacities going out of the source and must be much greater than the number of edges in the network) Specifically, draw the network, mark up edge capacities, and list the sequence of augmenting paths.

Any example satisfying that the condition in this question is fine when you list the sequence of augmenting paths.

CS570
Analysis of Algorithms
Spring 2008
Exam II

Name: _____
Student ID: _____

	Maximum	Received
Problem 1	20	
Problem 2	15	
Problem 3	15	
Problem 4	15	
Problem 5	20	
Problem 6	15	
Total	100	

Note: The exam is closed book closed notes.

1) 20 pts

Mark the following statements as **TRUE**, **FALSE**. No need to provide any justification.

[**TRUE**]

If all capacities in a network flow are rational numbers, then the maximum flow will be a rational number, if exist.

[**TRUE**]

The Ford-Fulkerson algorithm is based on the greedy approach.

[**FALSE**]

The main difference between divide and conquer and dynamic programming is that divide and conquer solves problems in a top-down manner whereas dynamic-programming does this bottom-up.

[**FALSE**]

The Ford-Fulkerson algorithm has a polynomial time complexity with respect to the input size.

[**TRUE**]

Given the Recurrence, $T(n) = T(n/2) + \theta(1)$, the running time would be $O(\log(n))$

[**FALSE**]

If all edge capacities of a flow network are increased by k, then the maximum flow will be increased by at least k.

[**TRUE**]

A divide and conquer algorithm acting on an input size of n can have a lower bound less than $\Omega(n \log n)$.

[**TRUE**]

One can actually prove the correctness of the Master Theorem.

[**TRUE**]

In the Ford Fulkerson algorithm, choice of augmenting paths can affect the number of iterations.

[**FALSE**]

In the Ford Fulkerson algorithm, choice of augmenting paths can affect the min cut.

2) 15 pts

Present a divide-and-conquer algorithm that determines the minimum difference between any two elements of a sorted array of real numbers.

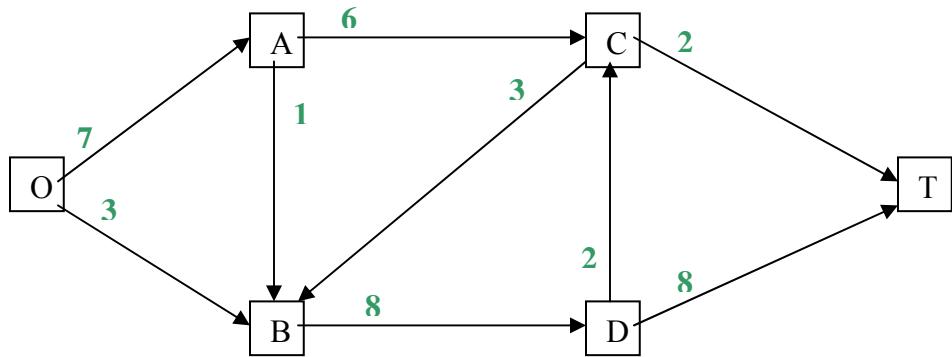
Key feature: The min difference can always been achieved between a pair of neighbors in the array, as the array is sorted.

```
int Min_Diff(first, last)
{
    if (last >= first)
        return inf;
    else
        return min(Min_Diff(first, (first + last)/2), Min_Diff((first + last)/2+1,
last), abs(number[(first + last)/2+1] - number[(first + last)/2]));
}
```

The complexity is liner to the array size.

3) 15 pts

You are given the following directed network with source O and sink T.

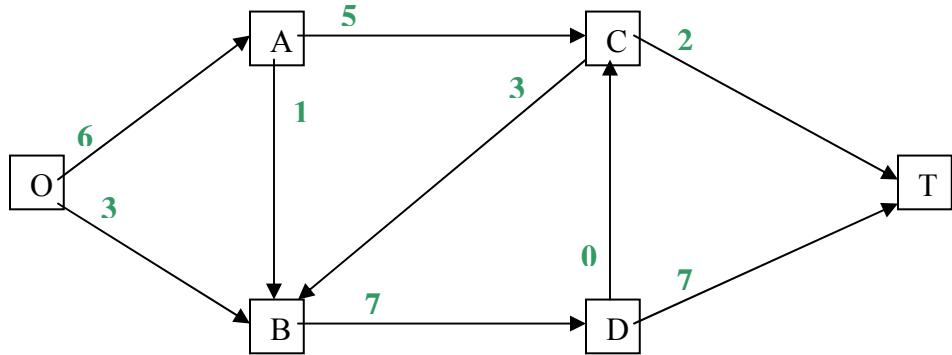


a) Find a maximum flow from O to T in the network.

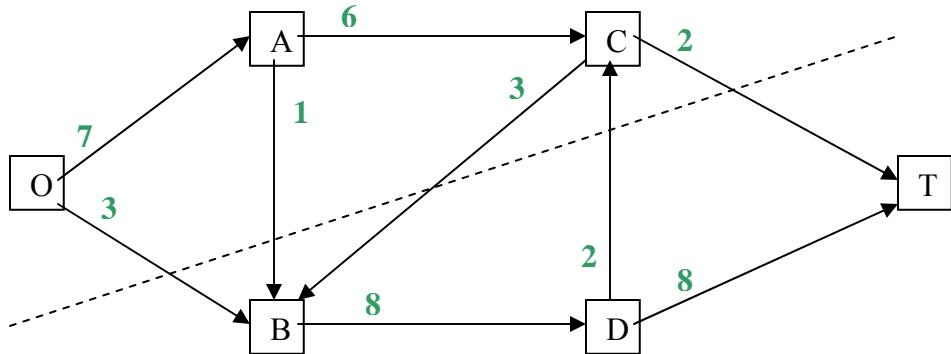
Augmenting paths and flow pushing amount:

OACT	2
OBDT	3
OABDT	1
OACBDT	3

And the maximum flow here is with weight 9:



b) Find a minimum cut. What is its capacity?



Capacity of this min cut is 9.

4) 15 pts

Solve the following recurrences

a) $T(n) = 2T(n/2) + n \log n$

According to the master theorem, $T(n) = \Theta(n \log^2 n)$.

Or we can solve it like this:

$$\begin{aligned}
 T(n) &= 2T\left(\frac{n}{2}\right) + n \log n = 2\left(2T\left(\frac{n}{4}\right) + \frac{n}{2} \log n - \frac{n}{2} \log 2\right) + n \log n \\
 &= 4T\left(\frac{n}{4}\right) + 2n \log n - n \log 2 = \dots = 2^k T\left(\frac{n}{2^k}\right) + kn \log n - \frac{k(k-1)}{2} n \log 2 \\
 &= \dots =_{(k=\log n)} nT(1) + \Theta(n \log^2 n) = \Theta(n \log^2 n)
 \end{aligned}$$

$$b) \quad T(n) = 2T(n/2) + \log n$$

Similar to a), the result is $T(n) = \Theta(n)$.

$$c) \quad T(n) = 2T(n-1) - T(n-2) \text{ for } n \geq 2; \quad T(0) = 3; \quad T(1) = 3$$

It is very easy to find out that for the initial values $T(0)=T(1)$, we always have $T(i)=T(0)$, $i > 0$. Thus $T(n) = 3$.

5) 20 pts

You are given a flow network with integer capacity edges. It consists of a directed graph $G = (V, E)$, a source s and a destination t , both belong to V . You are also given a parameter k . The goal is to delete k edges so as to reduce the maximum flow in G as much as possible. Give an efficient algorithm to find the edges to be deleted. Prove the correctness of your algorithm and show the running time.

We here introduce a straightforward algorithm (assuming $k \leq |E|$, otherwise just return failure):

```
Delete_k_edges()
{
    E' = E;
    for i=1 to k
    {
        curr_Max_Flow = inf;
        for j in E'
            if Max_Flow(V, E'-j) < curr_Max_Flow
            {
                curr_Max_Flow = Max_Flow(V, E'-j);
                index[i] = j;
            }
        E' = E' - index[i];
    }
}
```

Then the final E' is a required edge set, and indices of all k deleted edges are stored in the array $\text{index}[]$.

Running time is $O(k |E| \cdot T(\text{max_flow}))$, depending on the max_flow algorithm used here, the time complexity varies: if Edmonds_Karp is used here the time would be $O(k |V| |E|^3)$; if Dinic or other more advanced algorithm is used here the time complexity can be reduced.

Proof hint:

By induction.

$k = 1$, the algorithm is correct.

Assume $k = i$ the algorithm is correct. Then we prove for $k = i+1$, it is also correct. Here, it is better to divide this $i+1$ into the first step and the following i steps, not vice versa.

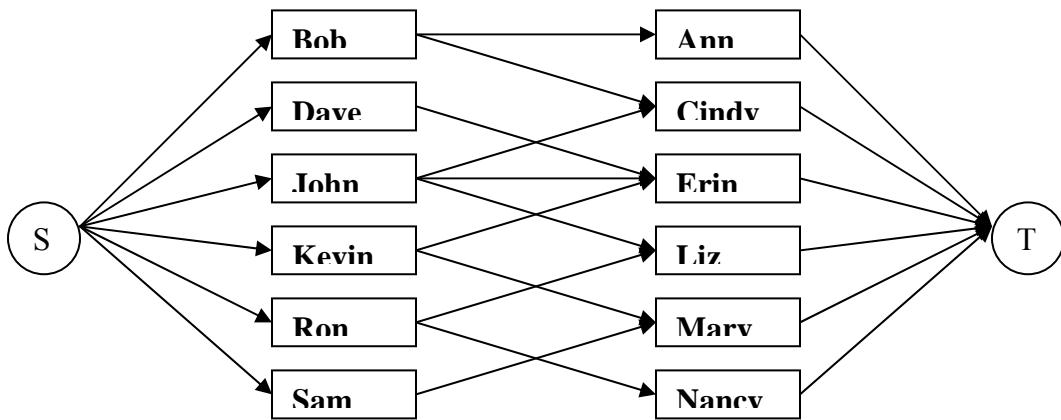
6) 15 pts

Six men and six women are at a dance. The goal of the matchmaker is to match each woman with a man in a way that maximizes the number of people who are matched with compatible mates. The table below describes the compatibility of the dancers.

	Ann	Cindy	Erin	Liz	Mary	Nancy
Bob	C	C	-	-	-	-
Dave	-	-	C	-	-	-
John	-	C	C	C	-	-
Kevin	-	-	C	-	C	-
Ron	-	-	-	C	-	C
Sam	-	-	-	-	C	-

Note: C indicates compatibility.

- a) Determine the maximum number of compatible pairs by reducing the problem to a max flow problem.



All edges are with capacity 1.

Run some maximum flow algorithm like Edmonds-Karp, it would guarantee to return a 0-1 solution within polynomial time, with represents the required match.

- b) Find a minimum cut for the network of part (a).

$A = \{S, Dave, Kevin, Sam, Erin, Mary\}$ and $A' = V - A$ constitute a minimum cut, with capacity 5.

- c) Give the list of pairs in the maximum pairs set.

Maximum 5 pairs. One solution:

Bob-Ann, Dave-Erin, John-Cindy, Ron-Nancy, Sam-Mary.

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 $\langle e_1, \dots, e_k \rangle$ carries positive flow iff $f(e_1) > 0, \dots, f(e_k) > 0$.)

[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

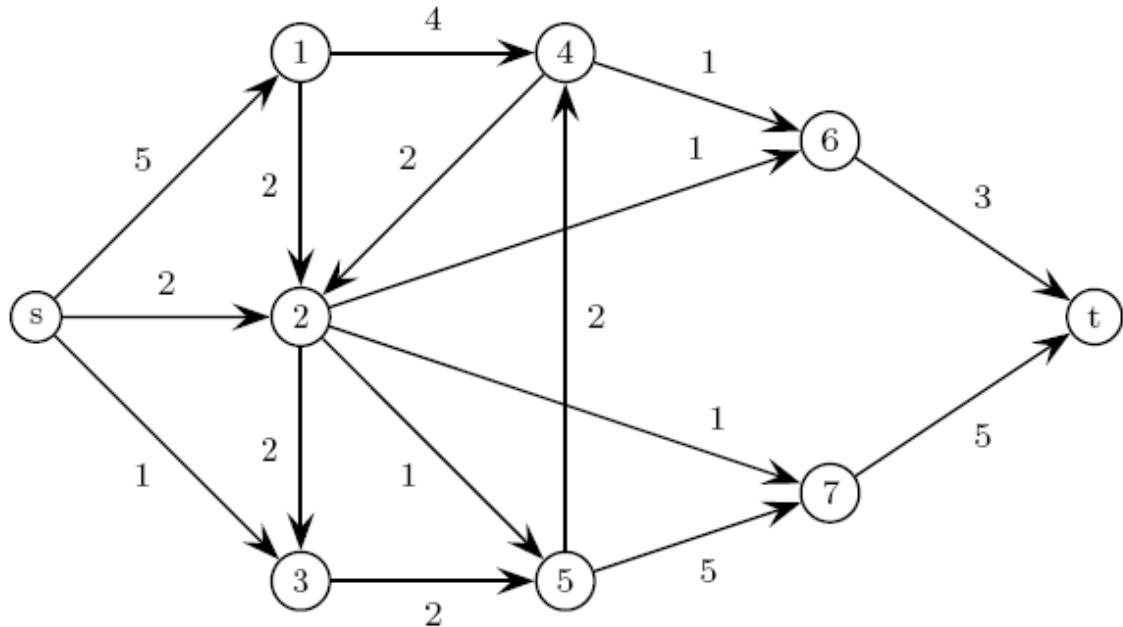
The running time of a pseudo polynomial time algorithm depends polynomially on the size of the input

[TRUE/FALSE] FALSE

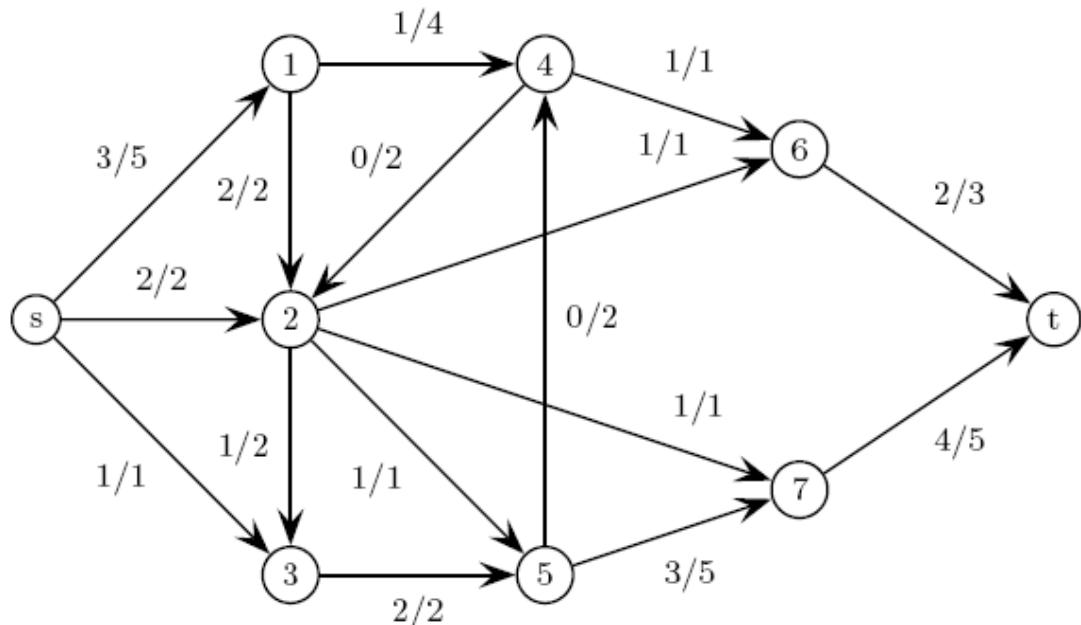
In dynamic programming you must calculate the optimal value of a subproblem twice, once during the bottom up pass and once during the top down pass.

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 $x_i \geq 0$. 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 $\text{OPT}(i)$ denote the maximum value that can be picked up among coins $1, \dots, i$.

Base case: $\text{OPT}(0) = 0$, and $\text{OPT}(1) = x_1$.

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 $1, \dots, i$ is the same as the one for coins $1, \dots, i-1$. If coin i is included in the optimal solution, then coin $i-1$ cannot be included, but among coins $1, \dots, i-2$, 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

$$\text{OPT}(0) = 0$$

$$\text{OPT}(1) = x_1$$

$$\text{OPT}(i) = \max(\text{OPT}(i-1), x_i + \text{OPT}(i-2)) \text{ for } i > 1.$$

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, \dots, r_n hours. On the project team, there are k people with time availabilities a_1, a_2, \dots, 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 $S_j \subseteq \{1, \dots, k\}$ 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 w_j 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 $s-u_h-v_i-t$ (if person i is in no constraint sets), or along $s-u_h-v_i-w_j-t$, 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 $n, n-1 \dots 3, 2, 1$. 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 $\text{OPT}(i, j) =$ The optimal cost of reaching from departure point “ i ” to departure point “ j ”.

Now, lets look at $\text{OPT}(i, j)$. 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

$$\text{OPT}(i, j) = \min(\text{over } j \leq k < i)(C[i, k] + \text{OPT}(k, j))$$

The base case is $\text{OPT}(i, i) = C[i, i] = 0$ for all “ i ” from 1 to n

The iterative program will look as follows

Let $\text{OPT}[i, j]$ 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 $\text{OPT}[n, 1]$

As for the running time, we are trying to fill up all $\text{OPT}[i, j]$ 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 $O(n^3)$

CS570
Analysis of Algorithms
Spring 2010
Exam II

Name: _____
Student ID: _____

DEN Student YES / NO

	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 200 words, anything beyond 200 words will not be considered.

1) 20 pts

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

In the final residual graph constructed during the execution of the Ford–Fulkerson Algorithm, there's no path from source to sink.

TRUE

In a flow network whose edges all have capacity 1, the maximum flow value equals the maximum degree of a vertex in the flow network.

FALSE.

Memoization is the basis for a top-down alternative to the usual bottom-up approach to dynamic programming solutions.

TRUE

The time complexity of a dynamic programming solution is always lower than that of an exhaustive search for the same problem.

FALSE

If we multiply all edge capacities in a graph by 5, then the new maximum flow value is the original one multiplied by 5.

TRUE.

For any graph G with edge capacities and vertices s and t , there always exists an edge such that increasing the capacity on that edge will increase the maximum flow from s to t . (Assume that there is at least one path in the graph from s to t .)

FALSE.

There might be more than one min-cut, by increasing the edge capacity of one min-cut doesn't have to impact the other one.

Let G be a weighted directed graph with exactly one source s and exactly one sink t . Let (A, B) be a maximum cut in G , that is, A and B are disjoint sets whose union is V , $s \in A, t \in B$, and the sum of the weights of all edges from A to B is the maximum for any two such sets. Now let H be the weighted directed graph obtained by adding 1 to the weight of each edge in G . Then (A, B) must still be a maximum cut in H .

FALSE

There could exist other edge which has many more cross-edges, which was not max-cut previously, to become the new max-cut.

A recursive implementation of a dynamic programming solution is often less efficient in practice than its equivalent iterative implementation.

We give points for both TRUE and FALSE answers due to the ambiguity of "often".

Ford-Fulkerson algorithm will always terminate as long as the flow network G has edges with strictly positive capacities.

FALSE

If some edges are irrational numbers, Ford-Fulkerson may never terminate.

Any problem that can be solved using dynamic programming has a polynomial time worst case time complexity with respect to its input size.

FALSE

2) 20 pts

Judge the following statement is true or false. If true, prove it. If false, give a counter-example.

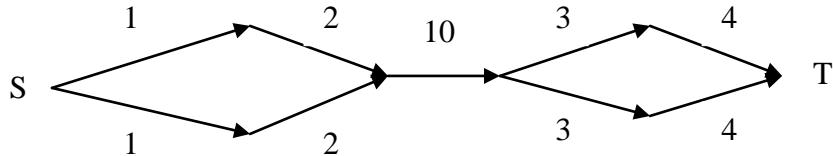
Given a directed graph, if deleting edge e reduces the original maximum flow more than deleting any other edge does, then edge e must be part of a minimum s-t cut in the original graph.

Solution:

False.

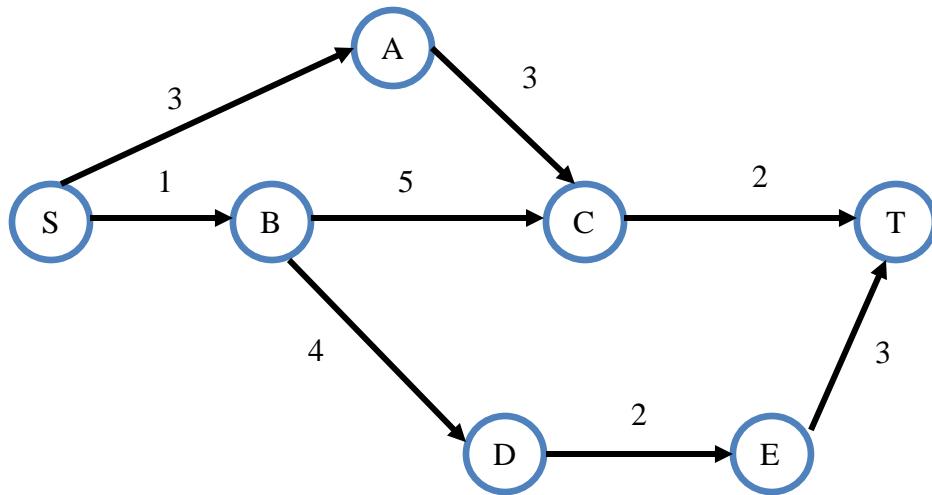
Counter-example:

Apparently deleting the edge with capacity 10 will reduce the flow by the most, while it is not part of minimum s-t cut.



Comment: some of you misunderstood the problem in different ways. Some counter-examples have multiple min s-t cuts and the edge "e" is actually in one of them. And some others failed to satisfy the precondition "more than... any other", and have some equal alternatives, which are also incorrect.

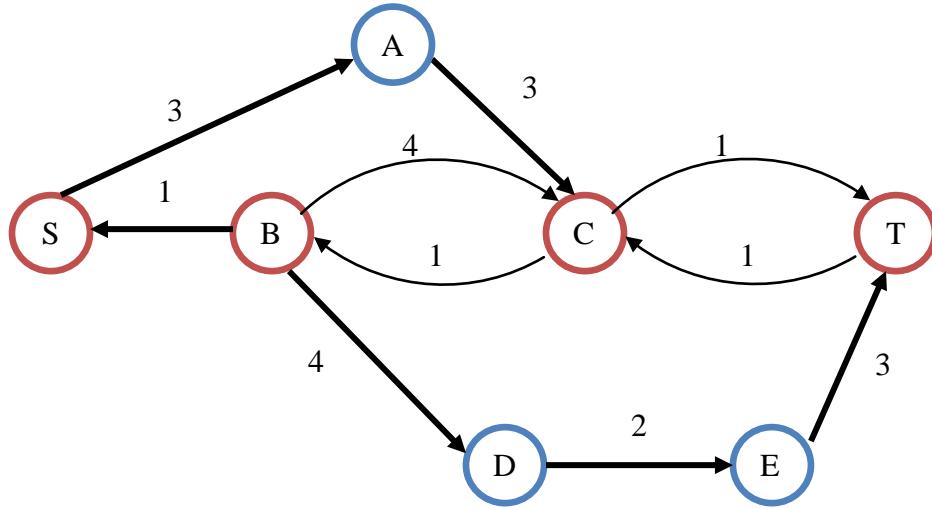
3) 20 pts



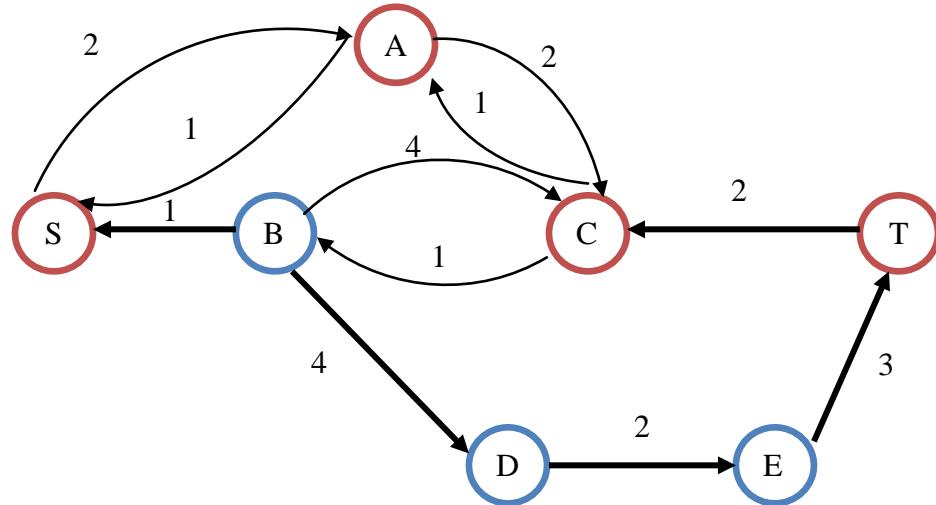
- a- Show all steps involved in finding maximum flow from S to T in above flow network using the Ford-Fulkerson algorithm.
- b- What is the value of the maximum flow?
- c- Identify the minimum cut.

a.

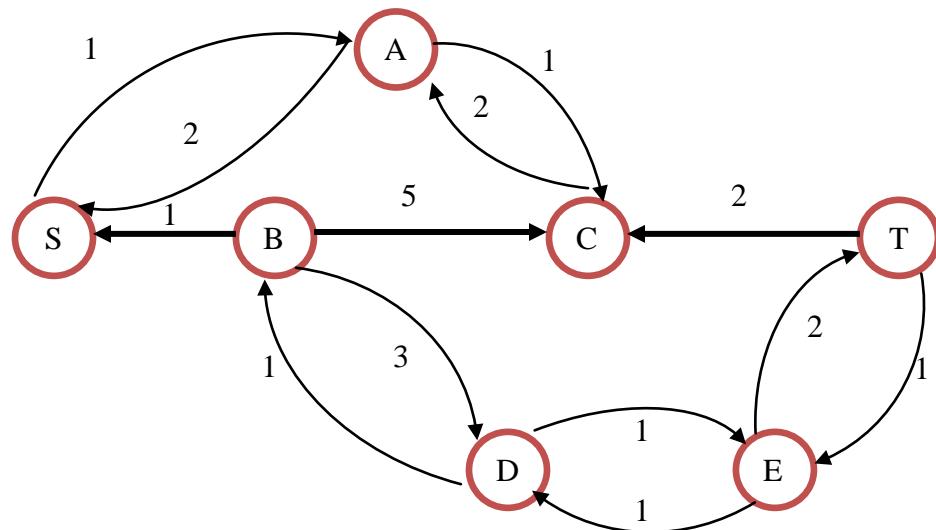
1. Augment path: [S, B, C, T], bottleneck is 1, residual graph:



2. Augment path: [S, A, C, T], bottleneck is 1, residual graph:



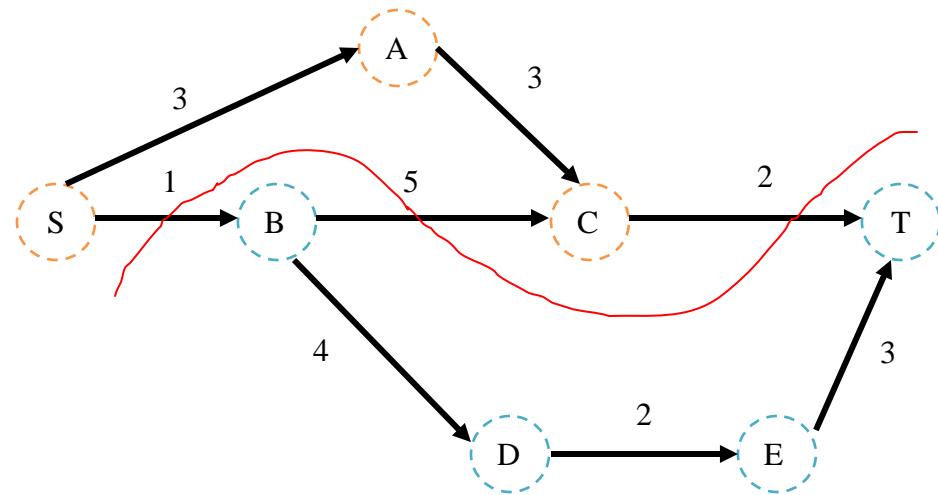
3. Augment path: [S, A, C, B, D, E, T], bottleneck is 1, residual graph:



4. No forward path from S to T, terminate.

b. The maximum flow is 3.

c. The Min-Cut is: [S, A, C] and [B, D, E, T].



4) 20 pts

The words `computer' and `commuter' are very similar, and a change of just one letter, p→m will change the first word into the second. The word `sport' can be changed into `sort' by the deletion of the `p', or equivalently, `sort' can be changed into `sport' by the insertion of `p'.

The edit distance of two strings, s_1 and s_2 , is defined as the minimum number of point mutations required to change s_1 into s_2 , where a point mutation is one of:

1. change a letter,
 2. insert a letter or
 3. delete a letter
- a) Design an algorithm using dynamic programming techniques to calculate the edit distance between two strings of size at most n . The algorithm has to take less than or equal to $O(n^2)$ for both time and space complexity.
- b) Print the edits.

The following recurrence relations define the edit distance, $d(s_1, s_2)$, of two strings s_1 and s_2 :

1. $d(., .) = 0$ (for empty strings)
2. $d(s, .) = d(., s) = |s|$ (length of s)
3. $d(s_1 + ch_1, s_2 + ch_2) = \min(d(s_1, s_2) + 0, if ch_1 = ch_2, d(s_1 + ch_1, s_2) + 1, d(s_1, s_2 + ch_2) + 1)$

The first two rules above are obviously true, so it is only necessary consider the last one. Here, neither string is the empty string, so each has a last character, ch_1 and ch_2 respectively. Somehow, ch_1 and ch_2 have to be explained in an edit of $s_1 + ch_1$ into $s_2 + ch_2$.

- If ch_1 equals ch_2 , they can be matched for no penalty, which is 0, and the overall edit distance is $d(s_1, s_2)$.
- If ch_1 differs from ch_2 , then ch_1 could be changed into ch_2 , costing 1, giving an overall cost $d(s_1, s_2) + 1$.
- Another possibility is to delete ch_1 and edit s_1 into $s_2 + ch_2$, $d(s_1, s_2 + ch_2) + 1$.
- The last possibility is to edit $s_1 + ch_1$ into s_2 and then insert ch_2 , $d(s_1 + ch_1, s_2) + 1$.

There are no other alternatives. We take the least expensive, **min**, of these alternatives.

Examination of the relations reveals that $d(s_1, s_2)$ depends only on $d(s'_1, s'_2)$ where s'_1 is shorter than s_1 , or s'_2 is shorter than s_2 , or both. This allows the *dynamic programming* technique to be used.

A two-dimensional matrix, $m[0..|s1|, 0..|s2|]$ is used to hold the edit distance values:

```

m[i, j] = d(s1[1..i], s2[1..j])

m[0, 0] = 0
m[i, 0] = i,   i=1..|s1|
m[0, j] = j,   j=1..|s2|

m[i, j] = min(m[i-1, j-1] + if s1[i]=s2[j] then 0 else 1,
               m[i-1, j] + 1,
               m[i, j-1] + 1 ),  i=1..|s1|, j=1..|s2|

```

$m[.]$ can be computed *row by row*. Row $m[i, .]$ depends only on row $m[i - 1, .]$. The time complexity of this algorithm is $O(|s1| * |s2|)$. If $s1$ and $s2$ have a similar length n , this complexity is $O(n^2)$.

b)

Once the algorithm terminates, we have generated the edit distance matrix m , we can do a trace-back for all the edits, e.g. Figure 1:

		S	a	t	u	r	d	a	y
	0	1	2	3	4	5	6	7	8
S	1	0	1	2	3	4	5	6	7
u	2	1	1	2	2	3	4	5	6
n	3	2	2	2	3	3	4	5	6
d	4	3	3	3	3	4	3	4	5
a	5	4	3	4	4	4	4	3	4
y	6	5	4	4	5	5	5	4	3

Figure 1 Edit distance matrix (from Wikipedia)

Here we are using a head recursion so that it prints the path from beginning to the end.

```

Print_Edits(n1, n2)
if n = 0 then
    Output nothing
else
    Find (i, j) ∈ {(n1 - 1, n2), (n1, n2 - 1), (n1 - 1, n2 - 1)} that has the minimum value from
    m[i, j].
    Print_Edits(i, j)
    if m[i, j] = m(n1, n2) then

```

```
    Do nothing
else
    if  $(i, j) = (n_1 - 1, n_2 - 1)$  then
        Print “change s1’s  $n_1^{th}$  letter into s2’s  $n_2^{th}$  letter”
    else if  $(i, j) = (n_1, n_2 - 1)$  then
        Print “insert s2’s  $n_2^{th}$  letter after s1’s  $n_1^{th}$  position”
    else
        Print “delete s1’s  $n_1^{th}$  letter”
```

The trace-back process has the complexity of $O(n)$.

5) 20 pts

Given a 2-dimensional array of size $m \times n$ with only “0” or “1” in each position, starting from position [1, 1] (top left corner), every time you can only move either one cell to the right or one cell down (of course you must remain in the array). Give an $O(mn)$ algorithm to calculate the number of different paths without reaching “0” in any step, starting from [1, 1] and ending in [m, n].

Solution:

We can solve this problem using dynamic programming. We denote $a[i][j]$ as the array, $\text{num}[i][j]$ as the number of different paths without reaching "0" in any step starting from [1, 1] and ending in [i, j]. Then the desired solution is $\text{num}[m][n]$.

We have

$$\text{num}[i][j] = \begin{cases} 0 & i=0 \text{ or } j=0 \text{ or } a[i][j]=0 \\ a[1][1] & i=1 \text{ and } j=1 \\ \text{num}[i-1][j] + \text{num}[i][j-1] & \text{other} \end{cases}$$

This is simply because we can reach cell [i, j] by coming from either [i-1, j] or [i, j-1].

When $a[i][j]=1$, $\text{num}[i][j]$ is just the sum of the two num's from the two different directions.

By calculating num row by row, we can get $\text{num}[m][n]$ at last. This is an $O(mn)$ solution since we use $O(1)$ time to calculate each $\text{num}[i][j]$ and the size of array num is $O(mn)$.

Comment: this is different from the "number of disjoint paths" problem which can be solved by being reduced to max flow problem. And some used max function instead of plus. Some other used recursion, but without memorization which would course the complexity become exponential. And some other tried to find paths and count, which is also an approach with exponential complexity.

CS570
Analysis of Algorithms
Summer 2007
Exam 2

Name: _____
Student ID: _____

	Maximum	Received
Problem 1	10	
Problem 2	20	
Problem 3	20	
Problem 4	10	
Problem 5	20	
Problem 6	20	

Note: The exam is closed book closed notes.

1) 10 pts

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

True [TRUE/FALSE]

Binary search could be called a divide and conquer technique

False [TRUE/FALSE]

If you have non integer edge capacities, then you cannot have an integer max flow

True [TRUE/FALSE]

The Ford Fulkerson algorithm with real valued capacities can run forever

True [TRUE/FALSE]

If we have a 0-1 valued s-t flow in a graph of value f, then we have f edge disjoint s-t paths in the graph

True [TRUE/FALSE]

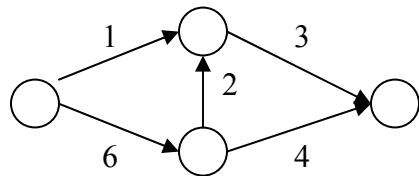
Merge sort works because at each level of the algorithm, the merge step assumes that the two lists are sorted

2) 20pts

Prove or disprove the following for a given graph $G(V,E)$ with integer edge capacities C_i

- a. If the capacities on each edge are unique then there is a unique min cut

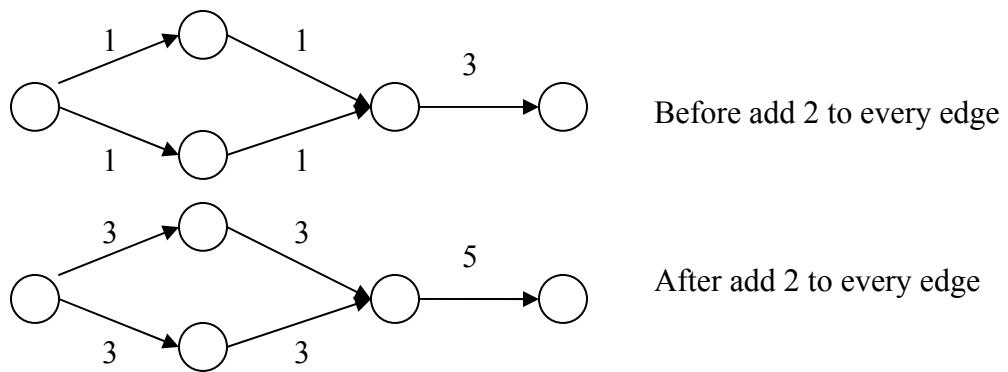
Disprove :



- b. If we multiply each edge with the same multiple “f”, the max flow also gets multiplied by the same factor

Prove: For each cut (A, B) of the graph G , the capacity $c(A, B)$ will be multiplied by “f” if each edge’s capacity is multiplied by “f”, thus the minimal cut will be multiplied by “f”, thus the max flow also gets multiplied by “f”.

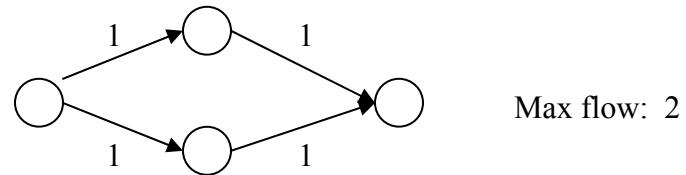
- c. If we add the same amount, say “a” to every edge in the graph, then the min cut stays the same (the edges in the min cut stay the same i.e.)



- d. If the edge costs are all even, then the max flow(min cut) is also even

The capacity of any cut (A, B) is sum of the capacities of all edges out of A, thus the capacity of any cut, including the minimal one, is also even.

- e. If the edge costs are all odd, then the max flow (min cut) is also odd



3) 20 pts

Suppose you are given k sorted arrays, each with n elements, and you want to combine them into a single sorted array of kn elements.

(a) Here's one strategy: merge the first two arrays, then merge in the third, then merge in the fourth, and so on. What is the time complexity of this strategy, in terms of k and n ?

The merge of the first two arrays takes $O(n)$, ... the merge of the last array takes $O(kn)$, totally there are k merge sorts. Thus, it takes $O(k^2 n)$ times.

(b) Present a more efficient solution to this problem.

Let the final array to be A , initially A is empty.

Build up a binary heap with the first element from the k given arrays, thus this binary tree consists of k elements.

Extract the minimal element from the binary tree and add it to A , delete it from its original array, and insert the next element from that array into the binary heap.

Each heap operation is $O(\log k)$, and take $O(kn)$ operations, thus, the running time is $O(kn \log k)$

4) 10 pts

Derive a recurrence equation that describes the following code. Then, solve the recurrence equation.

```
COMPOSE (n)
1. for  $i \leftarrow 1$  to  $n$ 
2.       do for  $j \leftarrow 1$  to  $\sqrt{n}$ 
3.             do print( $i, j, n$ )
4. if  $n > 0$ 
5.       then for  $i \leftarrow 1$  to 5
6.             COMPOSE ( $\lfloor n/2 \rfloor$ )
```

$$T(n) = 5 T(n/2) + O(n^{3/2})$$

By the Master theorem, $T(n) = n^{\lg 5}$

5) 20 pts

Let $G=(V,E)$ be a directed graph, with source $s \in V$, sink $t \in V$, and non-negative edge capacities $\{C_e\}$. Give a polynomial time algorithm to decide whether G has a unique minimum s - t cut. (i.e. an s - t cut of capacity strictly less than that of all other s - t cuts.)

Find a max flow f for this graph G , and then construct the residual graph G' based on this max flow f . Start from the source s , perform breadth-first or depth-first search to find the set A that s can reach, define $B = V - A$, the cut (A, B) is a minimum s - t cut. Then, for each node v in B that is connected to A in the original graph G , try the following codes: add v into set A , and then perform breadth-first or depth-first search find a new set A' that s can reach, if the sink t is included in A' , then try next node v' , otherwise, report a new minimum s - t cut. If all possible nodes v in B have been tried but no more minimum s - t cut can be found, then report the (A, B) is the unique minimum s - t cut.

20 pts

Consider you have three courses to study, C1, C2 and C3. For the three courses you need to study a minimum of T1, T2, and T3 hours respectively. You have three days to study for these courses, D1, D2 and D3. On each day you have a maximum of H1, H2, and H3 hours to study respectively. You have only 12 hours total to give to all of these courses, which you could distribute within the three days. On each one of the days, you could potentially study all three courses. Give an algorithm to find the distribution of hours to the three courses on the three days

If $T_1+T_2+T_3 > 12$ or $T_1+T_2+T_3 > H_1 + H_2 + H_3$, then report no feasible distribution can be found.

Else, start C1 with T1 hours, and then C2 with T2 hours, and finally C3 with T3 hours.

CS570
Analysis of Algorithms
Summer 2008
Exam II

Name: _____
Student ID: _____

____ 4:00 - 5:40 Section ____ 6:00 – 7:40 Section

	Maximum	Received
Problem 1	15	
Problem 2	15	
Problem 3	15	
Problem 4	20	
Problem 5	20	
Problem 6	15	
Total	100	

2 hr exam
Close book and notes

1) 15 pts

- a) Suppose that we have a divide-and-conquer algorithm for a solving computational problem that on an input of size n , divides the problem into two independent subproblems of input size $2n/5$ each, solves the two subproblems recursively, and combines the solutions to the subproblems. Suppose that the time for dividing and combining is $O(n)$. What's the running time of this algorithm? Answer the question by giving a recurrence relation for the running time $T(n)$ of the algorithm on inputs of size n , and giving a solution to the recurrence relation.

Solution:

The recurrence relation of $T(n)$:

$$T(n) = 2 T(2n/5) + O(n)$$

To solve the recurrence relation, using the substitution method:

guess that $T(n) \leq cn$

$$\Rightarrow T(n) \leq 2 * 2c/5n + an \leq cn \text{ as long as } c \geq 5a$$

Thus, $T(n) \leq cn \Rightarrow T(n) = O(n)$

b) Characterize each of the following recurrence equations using the master method. You may assume that there exist constants $c > 0$ and $d \geq 1$ such that for all $n < d$, $T(n) = c$.

- a. $T(n) = 2T(n/2) + \log n$
- b. $T(n) = 16T(n/2) + (n \log n)^4$
- c. $T(n) = 9T(n/3) + n^3 \log n$

Solution:

a. Since there is $\varepsilon > 0$ such that $\log n = O(n^{\log_2 2-\varepsilon}) = O(n^{1-\varepsilon})$,
 \Rightarrow case 1 of master method, $T(n) = \Theta(n^{\log_2 2}) = \Theta(n)$

b. $(n \log n)^4 = n^4 \log^4 n = \Theta(n^{\log_2 16} \log^4 n)$,
 \Rightarrow case 2 of master method, $T(n) = \Theta(n^{\log_2 16} \log^{(4+1)} n) = \Theta(n^4 \log^5 n)$

c. $n^3 \log n = \Omega(n^{\log_3 9 + 1})$ and $9 \times \left(\frac{n}{3}\right)^3 \log \frac{n}{3} = \frac{n^3}{3} \log \frac{n}{3} \leq \frac{1}{3} n^3 \log n$,
 \Rightarrow case 3 of master method, $T(n) = \Theta(n^3 \log n)$

2) 15 pts

You are given a sorted array $A[1..n]$ of n distinct integers. Provide an algorithm that finds an index i such that $A[i] = i$, if such exists. Analyze the running time of your algorithm

Solution: (This one is exactly the same as 5) of sample exam 1 of exam I)

Function(A, n)

```
{  
    i=floor(n/2)  
    if A[i]==i  
        return TRUE  
    if (n==1)&&(A[i]!=i)  
        return FALSE  
    if A[i]<i  
        return Function(A[i+1:n], n-i)  
    if A[i]>i  
        return Function(A[1:i], i)  
}
```

Proof:

The algorithm is based on Divide and Conquer. Every time we break the array into two halves. If the middle element i satisfy $A[i] < i$, we can see that for all $j < i$, $A[j] < j$. This is because A is a sorted array of DISTINCT integers. To see this we note that

$A[j+1]-A[j] \geq 1$ for all j . Thus in the next round of search we only need to focus on $A[i+1:n]$

Likewise, if $A[i] > i$ we only need to search $A[1:i]$ in the next round.

For complexity $T(n)=T(n/2)+O(1)$

Thus $T(n)=O(\log n)$

3) 15 pts

Consider a sequence of n distinct integers. Design and analyze a dynamic programming algorithm to find the length of a longest increasing subsequence. For example, consider the sequence:

45 23 9 3 99 108 76 12 77 16 18 4

A longest increasing subsequence is 3 12 16 18, having length 4.

Solution:

Let X be the sequence of n distinct integers.

Denote by $X(i)$ the i th integer in X , and by D_i the length of the longest increasing subsequence of X that ends with $X(i)$.

The recurrence that relates D_i to D_j 's with $j < i$ is as follows:

$$D_i = \max_{j < i, X(j) < X(i)} (D_j + 1)$$

The algorithm is as follows:

for $i = 1 \dots n$

 Compute D_i

end for

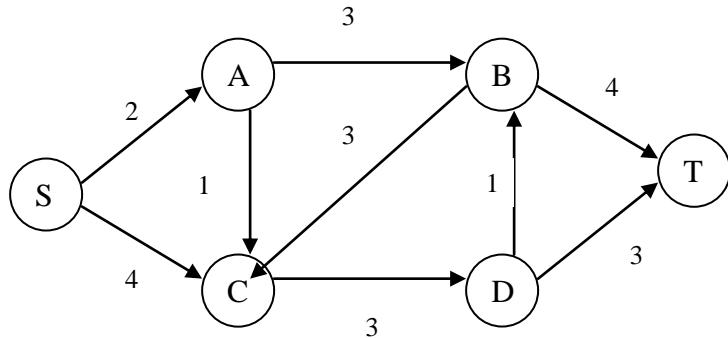
return the largest number among D_1 to D_n .

When computing each D_i , the recurrence finds the largest D_j such that $j < i, X(j) < X(i)$. Thus, each D_i is maximized. The length of the longest increasing subsequence is obviously among D_1 to D_n .

Since computing each D_i costs $O(n)$ and the loop runs for n times, the complexity of the algorithm is $O(n^2)$.

4) 20 pts

In the flow network illustrated below, each directed edge is labeled with its capacity. We are using the Ford-Fulkerson algorithm to find the maximum flow. The first augmenting path is S-A-C-D-T, and the second augmenting path is S-A-B-C-D-T.

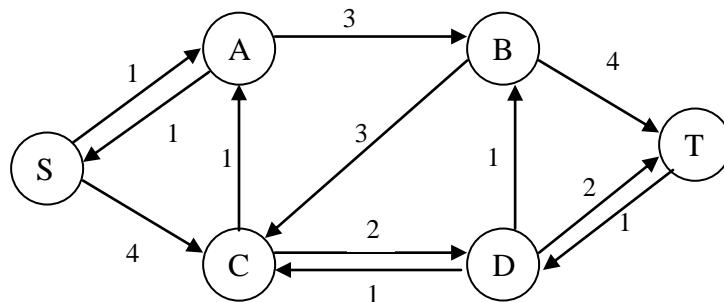


a) 10 pts

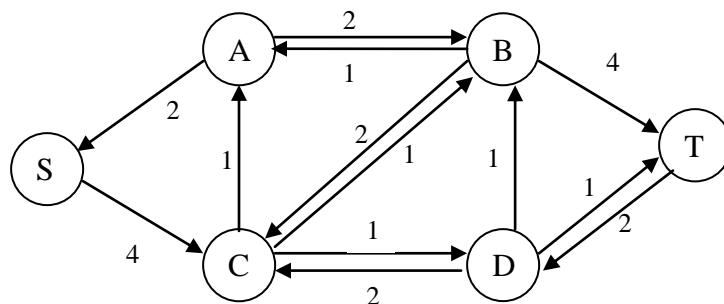
Draw the residual network after we have updated the flow using these two augmenting paths(in the order given)

Solution:

Residual network after S-A-C-D-T:



Residual network after S-A-B-C-D-T:



b) 6pts

List all of the augmenting paths that could be chosen for the third augmentation step.

Solution:

S-C-B-T

S-C-D-T

S-C-A-B-T

S-C-D-B-T

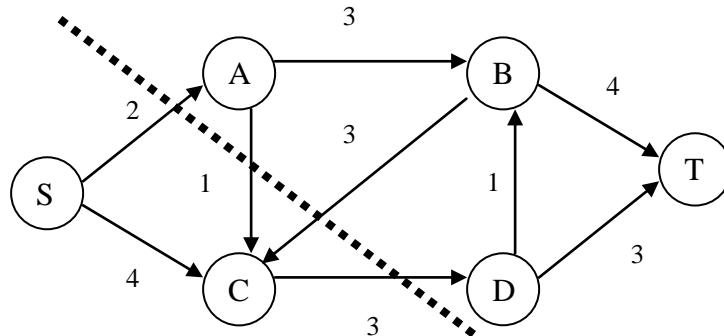
c) 4pts

What is the numerical value of the maximum flow? Draw a dotted line through the original graph to represent a minimum cut.

Solution:

The numerical value of the maximum flow is 5.

A minimum cut is shown in the following figure:



5) 20 pts

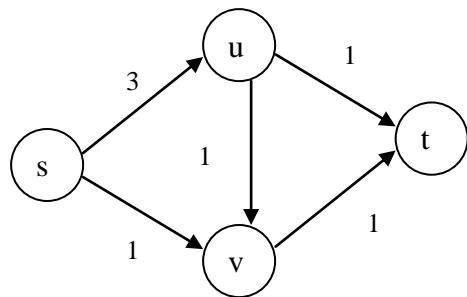
Decide whether you think the following statements are true or false. If true, give a short explanation. If false, give a counterexample.

Let G be an arbitrary flow network, with a source s , a sink t , and a positive integer capacity c_e on every edge e .

a) If f is a maximum s - t flow in G , then for all edges e out of s , we have $f(e) = c_e$.

Solution:

False.



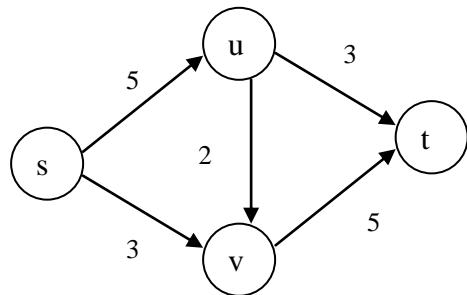
Clearly, the maximum s - t flow in the above graph is 2.

The edge (s, u) does not have $f(e) = c_e$

b) Let (A, B) be a minimum s - t cut with respect to the capacities $\{c_e : e \in E\}$. Now suppose we add 1 to every capacity. Then (A, B) is still a minimum s - t cut with respect to these new capacities $\{1 + c_e : e \in E\}$.

Solution:

False.



Clearly, one of the minimum cuts is $A = \{s, u\}$ and $B = \{v, t\}$. After adding 1 to every capacity, the maximum s - t flow becomes 10 and the cut between A and B is 11.

6) 15 pts

Suppose that in county X, there are only 3 kinds of coins: the first kind are 1-unit coins, the second kind are 4-unit coins, and the third kind are 6-unit coins. You want to determine how to return an amount of K units, where $K \geq 0$ is an integer, using as few coins as possible. For example, if $K=7$, you can use 2 coins (a 1-unit coin and a 6-unit coin), which is better than using three 1-unit coins and a 4-unit coins since in this case, the total number of coins used is 4.

For $0 \leq k \leq K$ and $1 \leq i \leq 3$, let $c(i, k)$ be the smallest number of coins required to return an amount of k using only the first i kinds of coins. So for example, $c(2, 5)$ would be the smallest number of coins required to return 5 units if we can only use 1-unit coins and 4-unit coins. In what follows, you can assume that the denomination of the coins is stored in an array d , i.e., $d[1]=1, d[2]=4, d[3]=6$.

Give a dynamic programming algorithm that returns $c(i, k)$ for $0 \leq k \leq K$ and $1 \leq i \leq 3$

Solution:

Let the coin denominations be d_1, d_2, \dots, d_i .

Because of the optimal substructure, if we knew that an optimal solution for the problem of making change for k cents used a coin of denomination d_j , we would have $c(i, k) = 1 + c(i, k - d_j)$.

As base cases, we have that $c(i, k) = 0$ for all $k \leq 0$.

To develop a recursive formulation, we have to check all denominations, giving

$$c(i, k) = \begin{cases} 0, & \text{if } k \leq 0 \\ 1 + \min_{1 \leq j \leq i} \{c(i, k - d_j)\}, & \text{if } k > 1 \end{cases}$$

The algorithm is as follows:

```
for i = 1...3
    for k = 0...K
        Compute c(i, k)
    end for
end for
```

When $i = j$, to compute each $c(j, k)$ costs $O(j)$. Thus, the complexity is $O(K+2K+3K) = O(6K)$

Q1:

[TRUE/FALSE] **FALSE**

Suppose $f(n) = f\left(\frac{n}{2}\right) + 56$, then $f(n) = \Theta(n)$

[TRUE/FALSE] **TRUE**

Maximum value of an s-t flow could be less than the capacity of a given s-t cut in a flow network.

[TRUE/FALSE] **FALSE**

For edge any edge e that is part of the minimum cut in G, if we increase the capacity of that edge by any integer $k > 1$, then that edge will no longer be part of the minimum cut.

[TRUE/FALSE] **FALSE**

Any problem that can be solved using dynamic programming has a polynomial worst case time complexity with respect to its input size.

[TRUE/FALSE] **FALSE**

In a dynamic programming solution, the space requirement is always at least as big as the number of unique sub problems

[TRUE/FALSE] **FALSE**

In the divide and conquer algorithm to compute the closest pair among a given set of points on the plane, if the sorted order of the points on both X and Y axis are given as an added input, then the running time of the algorithm improves to $O(n)$.

[TRUE/FALSE] **TRUE**

If f is a max s-t flow of a flow network G with source s and sink t , then the value of the min s-t cut in the residual graph G_f is 0.

[TRUE/FALSE] **TRUE**

Bellman-Ford algorithm can solve the shortest path problem in graphs with negative cost edges in polynomial time.

[TRUE/FALSE] **TRUE**

Given a directed graph $G=(V,E)$ and the edge costs, if every edge has a cost of either 1 or -1 then we can determine if it has a negative cost cycle in $O(|V|^3)$ time..

[TRUE/FALSE] **TRUE**

The space efficient version of the solution to the sequence alignment problem (discussed in class), was a divide and conquer based solution where the divide step was performed using dynamic programming.

Q2:

The problem can be formulated as a dynamic programming problem in many different ways
(These are the correct solutions to the three most common formulations used by students in the exam):

- 1) $OPT[v]$ will denote the minimum number of coins required to make change for value “ v ”.

The recursive formula would be:

$$OPT[T] = \min_{1 \leq i \leq n} \{OPT[T - X_i] + 1\}$$

The boundary values will be as follows:

$$OPT[0] = 0$$

$$OPT[v] = \infty \text{ if } v < 0$$

We fill in the $OPT[]$ array in the following order:

```
for (int i = 1; i <= v; i++) {  
    int min = infinity;  
    for (int x : {X1, X2, ..., Xn}) {  
        if (OPT[i - x] + 1 > min) min = OPT[i - x] + 1;  
    }  
    OPT[i] = min;  
}
```

At the end if $OPT[v] \leq k$ we return success, otherwise we return failure.

- 2) $OPT[k, v]$ denotes the minimum number of coins required to make change for v using at most k coins: Then:

$$OPT[k, v] = \min \left\{ OPT[k - 1, v], \min_{1 \leq i \leq n} \{OPT[k - 1, v - X_i] + 1\} \right\}$$

Here the boundary values will be:

$$OPT[k, 0] = 0$$

$$OPT[0, v] = \infty \text{ if } v \neq 0$$

$$OPT[k, v] = \infty \text{ if } v < 0$$

We should fill the $OPT[]$ array in the following order:

```
for (int i = 1; i <= v; i++) {  
    for (int j = 1; j <= k; j++) {  
        int min = OPT[j-1, i];  
        for (int x : {X1, X2, ..., Xn}) {  
            if (OPT[j-1, i - x] + 1 > min) min = OPT[j-1, i - x] + 1;  
        }  
        OPT[j, i] = min;  
    }  
}
```

Like the previous formulation, if $OPT[k, v] \leq k$ we return success, otherwise we return failure.

- 3) $OPT[k, v]$ is a binary value denoting whether it's possible to make change for v using at most k coins. Then:

$$OPT[k, v] = OPT[k - 1, v] \vee \left\{ \bigvee_{1 \leq i \leq n} OPT[k - 1, v - X_i] \right\}$$

The boundary values will be:

$$\begin{aligned} OPT[0, 0] &= true \\ OPT[0, v] &= false \\ OPT[k, v] &= false \text{ if } v < 0 \end{aligned}$$

We should fill the $OPT[]$ array in the following order:

```
for (int i = 1; i <= v; i++) {
    for (int j = 1; j <= k; j++) {
        OPT[i, j] = OPT[j-1, i];
        for (int x : {X1, X2, ..., Xn}) {
            if (OPT[j-1, i - x]) OPT[j, i] = true;
        }
    }
}
```

At the end, we only need to return $OPT[k, v]$ as the result.

Q3:

This problem can be mapped to a network flow problem. First assign a node s_i for each student and node n_i for each night. We also add two nodes S and T . Now we need to assign the edges and capacities. We create an edge between S and each student node with capacity 1 . We also create an edge between each night node and T with capacity 1. Then we need to connect student node s_i with the night node n_j if s_i is capable to cook on night n_i . The capacity for this edge is also 1. Now we only need to run ford-Fulkerson algorithm to find the maximum flow and see if this flow is equal to n or not.

Q4:

Part a) We present a few proofs of the claim. The first two are perhaps the most straightforward but the third leads naturally to an algorithm for part b.

Proof 1: Since A is a finite array, it has a minimum. Let j denote an index where A is minimum. If j is not in the boundary (that is j is neither 1 nor n), then j is a local minimum as well. If j is 1, then since $A[1] \geq A[2]$, $A[2]$ is the minimum values in the array and 2 is a local minimum. Likewise, if j is n, then since $A[n] \geq A[n-1]$, $A[n-1]$ is the minimum value in A and thus n-1 is a local minimum. Thus in every case, we may conclude that there is a local minimum.

Proof 2: Assume that A does not have a local minimum. Since $A[1] \geq A[2]$, this implies that $A[2] > A[3]$ (otherwise, 2 would be a local minimum). Likewise $A[2] > A[3]$ implies that $A[3] > A[4]$ and so on. In particular $A[n-1] > A[n]$, contradicting the fact that $A[n] \geq A[n-1]$. Thus our assumption is incorrect.

There is also an analogous proof that goes from right to left.

Proof 3. We prove the claim by induction. If $n=3$, then 2 is a local minimum. Consider $A[1\dots n]$ with $n>3$, $A[1] \geq A[2]$ and $A[n] \geq A[n-1]$. As the induction hypothesis, assume that all arrays $B[1\dots k]$ with $2 < k < n$, $B[1] \geq B[2]$ and $B[k] \geq B[k-1]$ have a local minimum. Let $j = \text{floor}(n/2)$. If $A[j] \leq A[j+1]$, then $A[1\dots j+1]$ has a local minimum (by the induction hypothesis) and hence $A[1\dots n]$ has a local minimum. Else (that is, $A[j] > A[j+1]$), then $A[j\dots n]$ has a local minimum (by the induction hypothesis) and hence $A[1\dots n]$ has a local minimum.

Part b) A divide and conquer solution for part b follows immediately from the third proof for part a. If $n=3$, then 2 is a local minimum. If $n>3$, set $j = \text{floor}(n/2)$. If $A[j] \leq A[j+1]$, then search for a local minimum in $A[1\dots j+1]$. Else, search for a local minimum in $A[j\dots n]$. Let $T(n)$ denote the number of pairwise comparisons performed by our recursive algorithm for finding a local minimum in $A[1\dots n]$. Then $T(n) \leq T(\text{ceil}(n/2)) + 1$ which implies that $T(n) = O(\log(n))$.

Q5:

- (a) The number of unique ways are shown as follows:

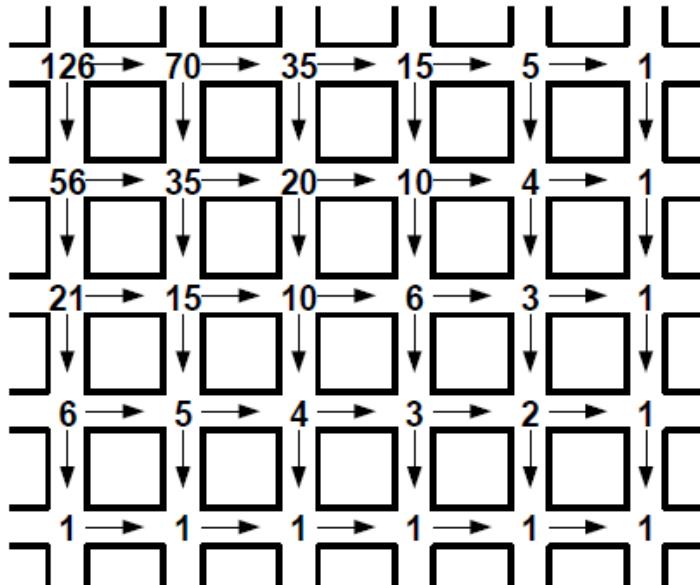


Figure A.

Answer: 126

- (b) Define $\text{OPT}(i,j)$ as the number of unique ways from intersection (i,j) to E: (5,4). The recursive relation is :

$$\text{OPT}(i,j) = \text{OPT}(i+1, j) + \text{OPT}(i,j+1), \text{ for } i < 5, \text{ and } j < 4$$

Boundary condition: $\text{OPT}(5,j) = 1$; $\text{OPT}(i,4) = 1$

Alternative solution:

If you define $\text{OPT}(i,j)$ as the number of unique ways from intersection S: (0,0) to intersection (i,j) , you can also get the correct answer, but the recursive relation and boundary conditions should correspondingly changes.

- (c) With dead ends, the numerical results of the number of unique ways are shown as follows:

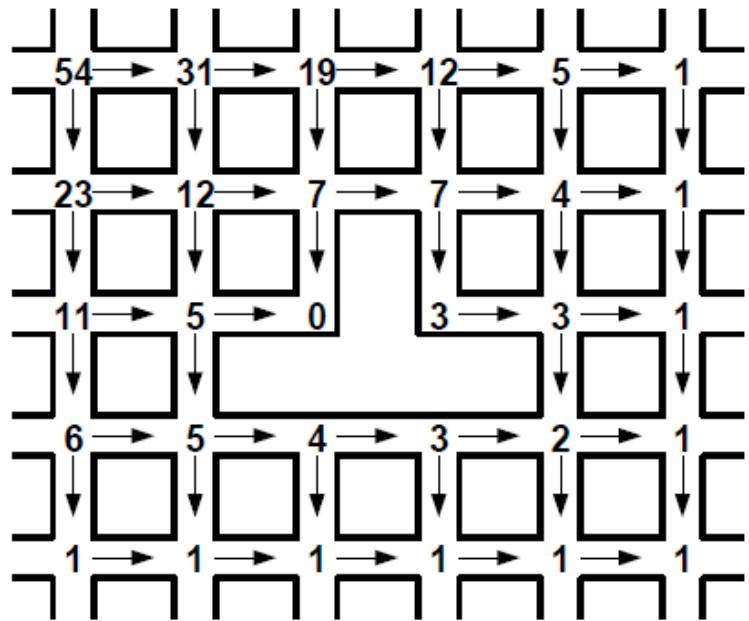


Figure B.

Answer: 54

Q6:

Algorithm:

1. Add node t , and add edges to t from each node in S .
2. Set the capacity of each of these new edges as infinity.
3. Set spammer node q as the source node. Then the communication network G becomes a larger network G' with source q and sink t .
4. Find the min $q-t$ cut on G' by running a polynomial time max-flow algorithm.
5. Install a spam filter on each edge in the min-cut's cut-set.

Complexity:

Since the construction of the new edges takes $O(|S|)$ times, together with the polynomial time of the min-cut algorithm, the entire algorithm takes polynomial time.

Justification:

We're simply trying to separate q from S , but min-cut only works when S is a single node. We fix this by creating t directly connected from S , but we want to make sure the min-cut's cut-set doesn't include any edge connecting t , which is why we put the capacities arbitrarily large on these edges. Min-cut would then find the min-cost way to separate q from S .

Q5:

- (a) The number of unique ways are shown as follows:

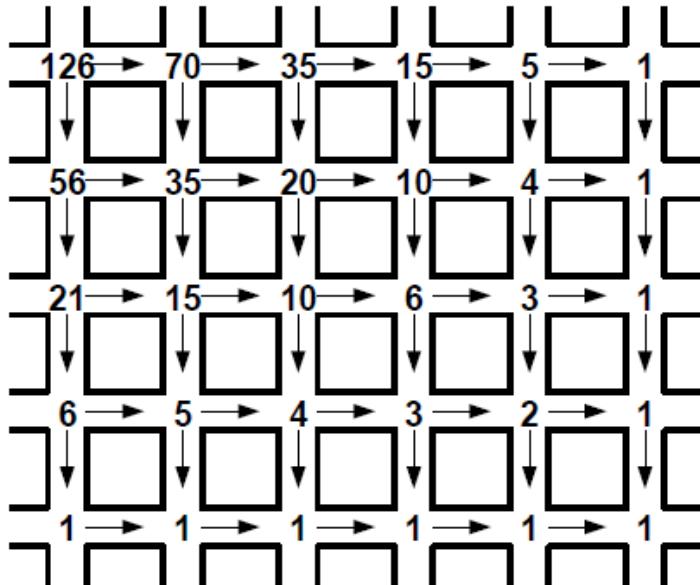


Figure A.

Answer: 126

- (b) Define $\text{OPT}(i,j)$ as the number of unique ways from intersection (i,j) to E: (5,4). The recursive relation is :

$$\text{OPT}(i,j) = \text{OPT}(i+1, j) + \text{OPT}(i,j+1), \text{ for } i < 5, \text{ and } j < 4$$

Boundary condition: $\text{OPT}(5,j) = 1$; $\text{OPT}(i,4) = 1$

Alternative solution:

If you define $\text{OPT}(i,j)$ as the number of unique ways from intersection S: (0,0) to intersection (i,j) , you can also get the correct answer, but the recursive relation and boundary conditions should correspondingly changes.

- (c) With dead ends, the numerical results of the number of unique ways are shown as follows:

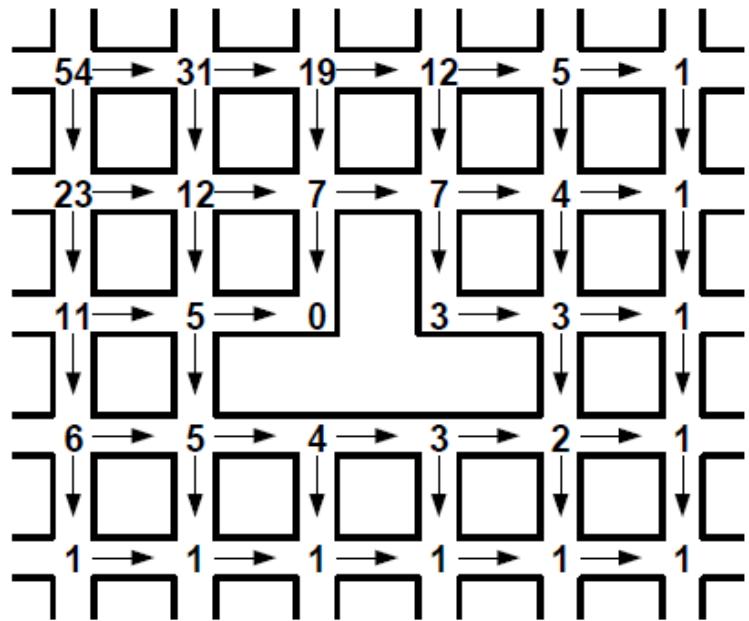


Figure B.

Answer: 54

Q6:

Algorithm:

1. Add node t , and add edges to t from each node in S .
2. Set the capacity of each of these new edges as infinity.
3. Set spammer node q as the source node. Then the communication network G becomes a larger network G' with source q and sink t .
4. Find the min $q-t$ cut on G' by running a polynomial time max-flow algorithm.
5. Install a spam filter on each edge in the min-cut's cut-set.

Complexity:

Since the construction of the new edges takes $O(|S|)$ times, together with the polynomial time of the min-cut algorithm, the entire algorithm takes polynomial time.

Justification:

We're simply trying to separate q from S , but min-cut only works when S is a single node. We fix this by creating t directly connected from S , but we want to make sure the min-cut's cut-set doesn't include any edge connecting t , which is why we put the capacities arbitrarily large on these edges. Min-cut would then find the min-cost way to separate q from S .

CS570
Analysis of Algorithms
Fall 2008
Final Exam

Name: _____
Student ID: _____

____ Monday Section ____ Wednesday Section ____ Friday Section

	Maximum	Received
Problem 1	20	
Problem 2	10	
Problem 3	10	
Problem 4	20	
Problem 5	20	
Problem 6	10	
Problem 7	10	
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]

If $NP = P$, then all problems in NP are NP hard

[FALSE]

L_1 can be reduced to L_2 in Polynomial time and L_2 is in NP, then L_1 is in NP

[FALSE]

The simplex method solves Linear Programming in polynomial time.

[FALSE]

Integer Programming is in P.

[FALSE]

If a linear time algorithm is found for the traveling salesman problem, then every problem in NP can be solved in linear time.

[TRUE]

If there exists a polynomial time 5-approximation algorithm for the general traveling salesman problem then 3-SAT can be solved in polynomial time.

[FALSE]

Consider an undirected graph $G=(V, E)$. Suppose all edge weights are different. Then the longest edge cannot be in the minimum spanning tree.

[FALSE]

Given a set of demands $D = \{d_v\}$ on a directed graph $G(V, E)$, if the total demand over V is zero, then G has a feasible circulation with respect to D .

[TRUE]

For a connected graph G , the BFS tree, DFS tree, and MST all have the same number of edges.

[FALSE]

Dynamic programming sub-problems can overlap but divide and conquer sub-problems do not overlap, therefore these techniques cannot be combined in a single algorithm.

Grading Criteria: Pretty clear, each has two point. These T/F are designed and answered by Professor Shamsian

2) 10 pts

Demonstrate that an algorithm that consists of a polynomial number of calls to a polynomial time subroutine could run in exponential time.

Suggested Solution:

Suppose X takes an input size of n and returns an output size of n^2 . You call X a polynomial number of times say n. If the size of the original input is n the size of the output will be n^{2n} which is exponential WRT n.

Grading Criteria: Rephrasing the question doesn't get that much. If you show you at least understood polynomial / exponential time, you get some credit.

3) 10 pts

Suppose that we are given a weighted, directed graph $G = (V, E)$ in which edges that leave the source vertex s may have negative weights, all other edge weights are nonnegative, and there are no negative-weight cycles. Argue that Dijkstra's algorithm correctly finds shortest paths from s in this graph.

Suggested solution :

```
DIJKSTRA( $G, w, s$ )
1 INITIALIZE-SINGLE-SOURCE( $G, s$ )
2  $S \leftarrow \emptyset$ 
3  $Q \leftarrow V[G]$ 
4 while  $Q \neq \emptyset$ 
5   do  $u \leftarrow \text{EXTRACT-MIN}(Q)$ 
6    $S \leftarrow S \cup \{u\}$ 
7   for each vertex  $v \in \text{Adj}[u]$ 
8     do RELAX( $u, v, w$ )
  
RELAX( $u, v, w$ )
1 if  $d[v] > d[u] + w(u, v)$ 
2 then  $d[v] \leftarrow d[u] + w(u, v)$ 
3  $\pi[v] \leftarrow u$ 
```

We have the algorithm as shown above.

We show that in each iteration of Dijkstra's algorithm, $d[u]=\delta(s,u)$ when u is added to S (the rest follows from the upper-bound property). Let $N^-(s)$ be the set of vertices leaving s , which have negative weights. We divide the proof to vertices in $N^-(s)$ and all the rest. Since there are no negative loops, the shortest path between s and all $u \in N^-(s)$, is through the edge connecting them to s , hence $\delta(s,u)=w(s,u)$, and it follows that after the first time the loop in lines 7,8 is executed $d[u]= w(s,u)=\delta(s,u)$ for all $u \in N^-(s)$ and by the upper bound property $d[u]=\delta(s,u)$ when u is added to S . Moreover it follows that $S= N^-(s)$ after $|N^-(s)|$ steps of the while loop in lines 4-8.

For the rest of the vertices we argue that the proof of Theorem 24-6 (*Theorem 24.6 - Correctness of Dijkstra's algorithm, Introduction to Algorithem by Cormen*) holds since every

shortest path from s includes at most one negative edge (and if does it has to be the first one). To see why this is true assume otherwise, which mean that the path contains a loop, contradicting the property that shortest paths do not contain loops.

Grading Criteria : You need to argue, so bringing just one example shouldn't get you the whole credit , but it may did! If you showed you at least understood Dijkstra, you got some credit too.

4) 20 pts

Phantasy Airlines operates a cargo plane that can hold up to 30000 pounds of cargo occupying up to 20000 cubic feet. We have contracted to transport the following items.

<u>Item type</u>	<u>Weight</u>	<u>Volume</u>	<u>Number</u>	<u>Cost if not carried</u>
1	4000	1000	3	\$800
2	800	1200	10	\$150
3	2000	2200	4	\$300
4	1500	500	5	\$500

For example, we have contracted 10 items of type 2, each of which weighs 800 pounds and takes up to 1200 cubic feet of space. The last column refers to the cost of subcontracting shipment to another carrier.

For each pound we carry, the cost of flying the plane increases by 5 cents. Which items should we put in the plane, and which should we ship via other carrier, in order to have the lowest shipping cost? Formulate this problem as an integer programming problem.

Suggested Solution:

Assume the data in the table are per item.

Considering x_1, x_2, x_3, x_4 are the items that we will ship for types 1,2,3 and 4 respectively.

It is clear that :

$$0 \leq x_1 \leq 3 \quad \& \quad 0 \leq x_2 \leq 10 \quad \& \quad 0 \leq x_3 \leq 4 \quad \& \quad 0 \leq x_4 \leq 5$$

Weight constraint :

$$x_1 * 4000 + x_2 * 800 + x_3 * 2000 + x_4 * 1500 \leq 30,000$$

Volume constraint:

$$x_1 * 1000 + x_2 * 1200 + x_3 * 2200 + x_4 * 500 \leq 20,000$$

Cost of shipping :

$$C_{Ship} = C_A + (x_1 * 4000 + x_2 * 800 + x_3 * 2000 + x_4 * 1500) * \$0.05$$

Where C_A is the cost of operating empty cargo plane

Cost of sub-contracting

$$C_{Sub} = (3 - x_1) * 800 + (10 - x_2) * 150 + (4 - x_3) * 300 + (5 - x_4) * 500$$

Out objective is to minimize $C_{Ship} + C_{Sub}$

Grading Criteria: Some credit will be deducted if you missed some optimization part. Making this simple question complicated may result deduction.

5) 20 pts

Suppose you are given a set of n integers each in the range $0 \dots K$. Give an efficient algorithm to partition these integers into two subsets such that the difference $|S_1 - S_2|$ is minimized, where S_1 and S_2 denote the sums of the elements in each of the two subsets.

Proposed Solution :

$a[1] \dots a[m]$ = the set of integers

$\text{opt}[i,k]$ = true if and only if it is possible to sum to k using elements $1 \dots i$

```
Initialize opt(i,k) = false for all i,k
for(i=1..n) {
    for(k=1..K) {
        if(opt(i,k) == true) {
            for(j=1..m) {
                opt(i,k + a[j]) = true;
            }
        }
    }
}
for(k = floor(K/2)..1) {
    if(opt(n, k)) {
        Return |K - 2k|; // Returns the difference. Partition can be found by back tracking
    }
}
```

6) 10 pts

Adrian has many, many love interests. Many, many, many love interests. The problem is, however, a lot of these individuals know each other, and the last thing our polyamorous TA wants is fighting between his hookups. So our resourceful TA draws a graph of all of his potential partners and draws an edge between them if they know each other. He wants at least k romantic involvements and none of them must know each other (have an edge between them). Can he create his LOVE-SET in polynomial time? Prove your answer.

Proposed Solution :

This is just Independent Set. Proof of NP-completeness was shown in class lecture. We show the correctness as follows: 1) Any Independent Set of size k on the graph will satisfy the requirement that no two love interests know each other (share an edge). 2) If there is a set of k love interests none of which know each other, then there must be a corresponding set of k vertices, such that no two share an edge (know each other).

7) 10 pts

An edge in a flow network is called a bottleneck if increasing its capacity increases the max flow in the network. Give an efficient algorithm for finding all the bottlenecks in the network.

Proposed Solution

Find max flow and the corresponding residual graph, then for each edge increase its capacity by one unit and see if you find a new path from s to t in the residual graph. (Of course you undo this increase before trying the next edge.) If the flow increases for any such edge then that edge must be a bottleneck.

CS570
Analysis of Algorithms
Summer 2008
Final Exam

Name: _____
Student ID: _____

____ 4:00 - 5:40 Section ____ 6:00 – 7:40 Section

	Maximum	Received
Problem 1	20	
Problem 2	16	
Problem 3	16	
Problem 4	16	
Problem 5	16	
Problem 6	16	
Total	100	

2 hr exam
Close book and notes

1) 20 pts

For each of the following statements, answer whether it is TRUE or FALSE, and briefly justify your answer.

- a) If a connected undirected graph G has the same weights for every edge, then every spanning tree of G is a minimum spanning tree, but such a spanning tree cannot be found in linear time.

T

- b) Given a flow network G and a maximum flow of G that has already been computed, one can compute a minimum cut of G in linear time.

F

- c) The Ford-Fulkerson Algorithm finds a maximum flow of a unit-capacity flow network with n vertices and m edges in time $O(mn)$ if one uses depth-first search to find an augmenting path in each iteration.

T

- d) Unless $P = NP$, 3-SAT has no polynomial-time algorithm.

F

- e) The problem of deciding whether a given flow f of a given flow network G is a maximum flow can be solved in linear time.

F

- f) If a decision problem A is polynomial-time reducible to a decision problem B (i.e., $A \leq_p B$), and B is NP-complete, then A must be NP-complete.

F

- g) If a decision problem B is polynomial-time reducible to a decision problem A (i.e., $B \leq_p A$), and B is NP-complete, then A must be NP-complete.

T

- h) Integer max flow (where flows and capacities are integers) is polynomial time reducible to linear programming .

F

- i) It has been proved that NP-complete problems cannot be solved in polynomial time.

F

- j) NP is a class of problems for which we do not have polynomial time solutions.

T

2) 16 pts

Suppose that we are given a weighted undirected graph $G = (V, E)$, a source $s \in V$, a sink $t \in V$, and a subset W of vertices V , and want to find a shortest path from s to t that must go through at least one vertex in W . Give an efficient algorithm that solves the problem by invoking any given single-source shortest path algorithm as a subroutine a small number (how many?) times. Show that your algorithm is correct.

$\text{Min}(\text{pathLength}(s, w) + \text{pathLength}(w, t))$ where w belongs to W

$\text{pathLength}(i, j)$ finds the shortest path between i and j . It can be implemented using any existing shortest path algorithm. It will be used $2 * \text{size}(W)$ times

16 pts

Suppose that we have n files F_1, F_2, \dots, F_n of lengths l_1, l_2, \dots, l_n respectively, and want to concatenate them to produce a single file $F = F_1 \circ F_2 \circ \dots \circ F_n$ of length $l_1 + l_2 + \dots + l_n$. Suppose that the only basic operation available is one that concatenates two files. Moreover, the cost of this basic operation on two files of length l_i and l_j , is determined by a cost function $c(l_i, l_j)$ which depends only on the lengths of the two files. Design an efficient dynamic programming algorithm that given n files F_1, F_2, \dots, F_n and a cost function $c(\cdot, \cdot)$, computes a sequence of basic operations that optimizes total cost of concatenating the n input files.

The sub structure of this problem is the time to merge a set of files S . $\text{time}(S)$ – the time needed to merge the files and the files in S should be kept for repeated use

```
if size (S) == 2:  
    # S1 and S2 are the two files in S  
    time (S) = c (length(S1), length(S2))  
else:  
    # f is a file in S  
    # S-f : take f out from S  
    time (S) = min (c (length(f), total length of all other files in S)+time (S-f))
```

Return $\text{time}(S)$ where S contains all the files

4) 16 pts

Define the language

Double-SAT = { ψ : ψ is a Boolean formula with at least 2 distinct satisfying assignments }.

For instance, the formula $\psi: (x \vee y \vee z) \wedge (x' \vee y' \vee z') \wedge (x' \vee y' \vee z)$ is in Double-SAT, since the assignments $(x = 1, y = 0, z = 0)$ and $(x = 0, y = 1, z = 1)$ are two distinct assignments that both satisfy ψ .

Prove that Double-SAT is NP-Complete.

Various methods can be used for reducing SAT to Double-SAT. Since SAT is NP-Complete, Double-SAT is NP complete.

Following is an example for the reduction.

On input $\psi(x_1, \dots, x_n)$:

1. Introduce a new variable y .

2. Output formula $\psi'(x_1, \dots, x_n, y) = \psi(x_1, \dots, x_n) \wedge (y \mid \text{not } y)$.

If $\psi(x_1, \dots, x_n)$ belongs to SAT, then ψ' has at least 1 satisfying assignment, and therefore $\psi'(x_1, \dots, x_n, y)$ has at least 2 satisfying assignments as we can satisfy the new clause $(y \mid \text{not } y)$ by assigning either

$y = 1$ or $y = 0$ to the new variable y , so $\psi'(x_1, \dots, x_n, y)$ belongs to Double-SAT. On the other hand, if $\psi(x_1, \dots, x_n)$ does not belong to SAT, then clearly $\psi'(x_1, \dots, x_n, y) = \psi(x_1, \dots, x_n) \wedge (y \mid \text{not } y)$ has no satisfying assignment either, so $\psi'(x_1, \dots, x_n, y)$ does not belong to Double-SAT. Therefore, SAT can be reduced to Double-SAT. Since the above reduction clearly can be done in P time, Double-SAT is NP-Complete.

5) 16 pts

Let X be a set of n intervals on the real line. A subset of intervals $Y \subset X$ is called a tiling path if the intervals in Y cover the intervals in X , that is, any real value that is contained in some interval in X is also contained in some interval in Y . The size of a tiling cover is just the number of intervals.

Describe and analyze an algorithm to compute the smallest tiling path of X as quickly as possible. Assume that your input consists of two arrays $X_L[1 .. n]$ and $X_R [1 .. n]$, representing the left and right endpoints of the intervals in X .



A set of intervals. The seven shaded intervals form a tiling path

Sort the intervals from left to right by the start position, if the start positions are same, then sort it from the left to right by the end position;

```

FOR (i=1;i<=n; i++) do
    Result[i] = positive unlimited ;
    FOR (j = 1;j < i; j ++) do
        IF (interval i overlap with interval j) then
            IF (Result[i]> result[j]+1) then
                Result[i] = result[j]+1;
Return result [n]

```

The complexity is $O(n^2)$

6) 16 pts

An edge of a flow network is called *critical* if decreasing the capacity of this edge results in a decrease in the maximum flow. Give an efficient algorithm that finds a critical edge in a network.

Find a min cut of the network which has the same capacity as the maximum flow.
Every outgoing edge from the min cut is a critical edge

CS570
Analysis of Algorithms
Summer 2009
Final Exam

Name: _____
Student ID: _____

	Maximum	Received
Problem 1	10	
Problem 2	20	
Problem 3	15	
Problem 4	20	
Problem 5		
Problem 6		
Total	100	

2hr exam, closed books and notes.

1) 10 pts

For each of the following sentences, state whether the sentence is known to be TRUE, known to be FALSE, or whether its truth value is still UNKNOWN.

- (a) If a problem is in P, it must also be in NP.
TRUE.
- (b) If a problem is in NP, it must also be in P.
UNKNOWN.
- (c) If a problem is NP-complete, it must also be in NP.
TRUE.
- (d) If a problem is NP-complete, it must not be in P.
UNKNOWN.
- (e) If a problem is not in P, it must be NP-complete.
FALSE.

If a problem is NP-complete, it must also be NP-hard.
TRUE.

If a problem is in NP, it must also be NP-hard.
FALSE.

If we find an efficient algorithm to solve the Vertex Cover problem we have proven that $P=NP$
TRUE.

If we find an efficient algorithm to solve the Vertex Cover problem with an approximation factor $\rho \geq 1$ (a single constant) then we have proven that $P=NP$
FALSE.

If we find an efficient algorithm that takes as input an approximation factor $\rho \geq 1$ and solves the Vertex Cover problem with that approximation factor, we have proven that $P=NP$.

TRUE.

2) 20 pts

Let $G = (V, E)$ be a weighted, directed graph with weight function $w : E \rightarrow \{0, 1, \dots, W\}$ for some nonnegative integer W . Modify Dijkstra's algorithm to compute the shortest paths from a given source vertex s in $O(WV + E)$ time.

Consider running Dijkstra's algorithm on a graph, where the weight function is $w : E \rightarrow \{1, \dots, W - 1\}$. To solve this efficiently, implement the priority queue by an array A of length $WV + 1$. Any node with shortest path estimate d is kept in a linked list at $A[d]$. $A[WV + 1]$ contains the nodes with ∞ as estimate.

EXTRACT-MIN is implemented by searching from the previous minimum shortest path estimate until a new is found. DECREASE-KEY simply moves vertices in the array. The EXTRACT-MIN operations takes a total of $O(VW)$ and the DECREASE-KEY operations take $O(E)$ time in total. Hence the running time of the modified algorithm will be $O(VW + E)$.

3) 15 pts

At a dance, we have n men and n women. The men have height $g(1), \dots, g(n)$, and women $h(1), \dots, h(n)$. For the dance, we want to match up men with women of roughly the same height. Here are the precise rules:

- a. Each man i is matched up with exactly one woman w_i , and each woman with exactly one man.
- b. For each couple (i, w_i) , the mismatch is height difference $|g(i)-h(w_i)|$.
- c. Our goal is to find a matching minimizing the maximum mismatch,
 $\text{MAX}_i |g(i)-h(w_i)|$.

Give an algorithm that runs in $O(n \log n)$ and achieves the desired matching. Provide proof of correctness.

We use an exchange argument. Let w_i denote the optimal solution. If there is any pair i, j such that $i < j$ in our ordering, but $w_i > w_j$ (also with respect to our ordering), then we evaluate the effect of switching to $w'_i := w_j, w'_j := w_i$. The mismatch of no other couple is affected. The couple including man i now has mismatch $|g(i) - h(w'_i)| = |g(i) - h(w_j)|$. Similarly, the other switched couple now has mismatch $|g(j) - h(w_i)|$.

We first look at man i , and distinguish two cases: if $h(w_j) \leq g(i)$ (i.e., man i is at least as tall as his new partner), then the sorting implies that $|g(i) - h(w_j)| = g(i) - h(w_j) \leq g(j) - h(w_j) = |g(j) - h(w_j)|$. On the other hand, if $h(w_j) > g(i)$, then $|g(i) - h(w_j)| = h(w_j) - g(i) \leq h(w_i) - g(i) = |h(w_i) - g(i)|$. In both cases, the mismatch between man i and his new partner is at most the previous maximum mismatch.

We use a similar argument for man j . If $h(w_i) \leq g(j)$, then $|g(j) - h(w_i)| = g(j) - h(w_i) \leq g(j) - h(w_j) = |g(j) - h(w_j)|$. On the other hand, if $h(w_i) > g(j)$, then $|g(j) - h(w_i)| = h(w_i) - g(j) \leq h(w_i) - g(i) = |h(w_i) - g(i)|$. Hence, the mismatch between j and his partner is also no larger than the previous maximum mismatch.

Hence, in all cases, we have that both of the new mismatches are bounded by the larger of the original mismatches. In particular, the maximum mismatch did not increase by the swap. By making such swaps while there are inversions, we gradually transform the optimum solution into ours. This proves that the solution found by the greedy algorithm is in fact optimal.

4) 20 pts

You are given integers p_0, p_1, \dots, p_n and matrices A_1, A_2, \dots, A_n where matrix A_i has dimension $(p_{i-1}) * p_i$

(a) Let $m(i, j)$ denote the minimum number of scalar multiplications needed to evaluate the matrix product $A_i A_{i+1} \dots A_j$. Write down a recursive algorithm to compute $m(i, j)$, $1 \leq i \leq j \leq n$, that runs in $O(n^3)$ time.

Hint: You need to consider the order in which you multiply the matrices together and find the optimal order of operations.

Initialize $m[i, j] = -1 \quad 1 \leq i \leq j \leq n$

Output $mcr(1, n)$

```
mcr(i, j) {  
    if m[i, j] >= 0 return m[i, j]  
    else if i == j m[i, j] = 0  
    else m[i, j] = min (i <= k < j) { mcr(i, k) + mcr(k+1, j) + P_{i-1} * P_k * P_k }  
    return m[i, j]  
}
```

(b) Use this algorithm to compute $m(1,4)$ for $p_0=2$, $p_1=5$, $p_2=3$, $p_3=6$, $p_4=4$

$m[i, j]$

i\j	2	3	4
1	30	66	114
2		90	132
3			72

$m[1, 4] = 114$

5) 20 pts

In a certain town, there are many clubs, and every adult belongs to at least one club. The townspeople would like to simplify their social life by disbanding as many clubs as possible, but they want to make sure that afterwards everyone will still belong to at least one club.

Prove that the Redundant Clubs problem is NP-complete.

First, we must show that Redundant Clubs is in NP, but this is easy: if we are given a set of K clubs, it is straightforward to check in polynomial time whether each person is a member of another club outside this set.

Next, we reduce from a known NP-complete problem, Set Cover. We translate inputs of Set Cover to inputs of Redundant Clubs, so we need to specify how each Redundant Clubs input element

is formed from the Set Cover instance. We use the Set Cover's elements as our translated list of people,

and make a list of clubs, one for each member of the Set Cover family. The members of each club are just the elements of the corresponding family. To finish specifying the Redundant Clubs input,

we need to say what K is: we let $K = F - K_{SC}$ where F is the number of families in the Set Cover instance and K_{SC} is the value K from the set cover instance. This translation can clearly be done in polynomial time (it just involves copying some lists and a single subtraction).

Finally, we need to show that the translation preserves truth values. If we have a yes-instance of Set Cover, that is, an instance with a cover consisting of K_{SC} subsets, the other K subsets form a solution to the translated Redundant Clubs problem, because each person belongs to a club in the

cover. Conversely, if we have K redundant clubs, the remaining K_{SC} clubs form a cover. So the answer

to the Set Cover instance is yes if and only if the answer to the translated Redundant Clubs instance
is yes.

6) 15 pts

A company makes two products (X and Y) using two machines (A and B). Each unit of X that is produced requires 50 minutes processing time on machine A and 30 minutes processing time on machine B. Each unit of Y that is produced requires 24 minutes processing time on machine A and 33 minutes processing time on machine B.

At the start of the current week there are 30 units of X and 90 units of Y in stock. Available processing time on machine A is forecast to be 40 hours and on machine B is forecast to be 35 hours.

The demand for X in the current week is forecast to be 75 units and for Y is forecast to be 95 units—these demands must be met. In addition, company policy is to maximize the combined sum of the units of X and the units of Y in stock at the end of the week.

- a. Formulate the problem of deciding how much of each product to make in the current week as a linear program.

Solution

Let

- x be the number of units of X produced in the current week
- y be the number of units of Y produced in the current week

then the constraints are:

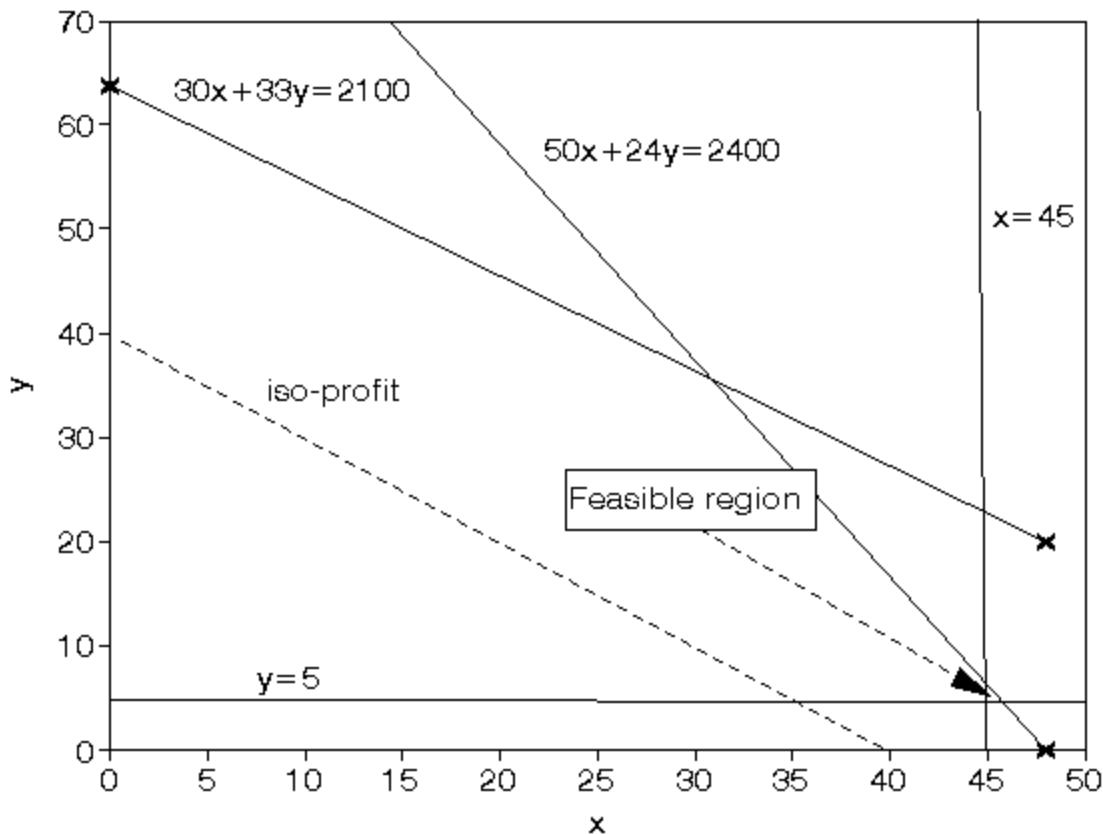
- $50x + 24y \leq 40(60)$ machine A time
- $30x + 33y \leq 35(60)$ machine B time
- $x \geq 75 - 30$
- i.e. $x \geq 45$ so production of X \geq demand (75) - initial stock (30), which ensures we meet demand
- $y \geq 95 - 90$
- i.e. $y \geq 5$ so production of Y \geq demand (95) - initial stock (90), which ensures we meet demand

The objective is: maximise $(x+30-75) + (y+90-95) = (x+y-50)$

i.e. to maximise the number of units left in stock at the end of the week

b. Solve this linear program graphically.

It can be seen in diagram below that the maximum occurs at the intersection of $x=45$ and $50x + 24y = 2400$



Solving simultaneously, rather than by reading values off the graph, we have that $x=45$ and $y=6.25$ with the value of the objective function being 1.25

CS570
Analysis of Algorithms
Fall 2014
Exam III

Name: _____
Student ID: _____
Email: _____

Wednesday Evening Section

	Maximum	Received
Problem 1	20	
Problem 2	16	
Problem 3	16	
Problem 4	16	
Problem 5	16	
Problem 6	16	
Total	100	

Instructions:

1. This is a 2-hr exam. Closed book and notes
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.

1) 20 pts

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

[/FALSE]

All integer programming problems can be solved in polynomial time.

[TRUE]

Fractional knapsack problem has a polynomial time greedy solution.

[/FALSE]

For any cycle in a graph, the cheapest edge in the cycle is in a minimum spanning tree.

[/FALSE]

Every decision problem is in NP.

[TRUE/]

If P=NP then P=NP=NP-complete.

[/FALSE]

Ford-Fulkerson algorithm can always terminate if the capacities are real numbers.

[/FALSE]

A flow network with unique edge capacities has a unique min cut.

[/FALSE]

A graph with non-unique edge weights will have at least two minimum spanning trees.

[TRUE/]

A sequence of $O(n)$ priority queue operations can be used to sort a set of n numbers.

[/FALSE]

If a problem X is polynomial time reducible to an NP-complete problem Y, then X is NP-complete.

2) 16 pts

Consider the **complete** weighted graph $G = (V, E)$ with the following properties

- a. The vertices are points on the X-Y plane on a regular $n \times n$ grid, i.e. the set of vertices is given by $V = \{(p, q) | 1 \leq p, q \leq n\}$, where p and q are integer numbers.
- b. The edge weights are given by the usual Euclidean distance, i.e. the weight of the edge between the nodes (i, j) and (k, l) is $\sqrt{(k - i)^2 + (l - j)^2}$.

Prove or disprove: There exists a minimum spanning tree of G such that every node has degree at most two.

Solution:

There does exist an MST of G with every node having degree ≤ 2 . One such MST is obtained by the edges joining **nodes** in the following order: $(1,1) \rightarrow (1,2) \rightarrow (1,3) \rightarrow \dots \rightarrow (1,n-1) \rightarrow (1,n) \rightarrow (2,n) \rightarrow (2,n-1) \rightarrow \dots \rightarrow (2,2) \rightarrow (2,1) \rightarrow (3,1) \rightarrow \dots \rightarrow (n,n)$.

Let us denote the above set of edges by E . To prove that E indeed forms an MST, we need to show that it forms a spanning tree and that it is of minimal weight.

E forms a spanning tree: It is clear that all nodes in the above sequence are distinct. This implies that there are no cycles since any cycle, when written out as a sequence of connected nodes, would necessarily repeat the starting node at the end. It is also clear that all nodes of the graph are covered in the above sequence. In particular, nodes $(i, 1)$, $(i, 2)$, ..., (i, n) are connected in that order for odd integers i and in the reverse order for even integers i . Hence, E forms a spanning tree and has $n^2 - 1$ edges.

E has minimal weight: It is easy to see that every edge in G has at least unit weight, since weight of edge between distinct nodes is minimal for edges between node pairs of the form $\{(i, j), (i+1, j)\}$ or $\{(i, j), (i, j+1)\}$, evaluating to an edge weight of 1. Since all edges in E are of this form, all edges in E have unit weight and total weight of E is equal to $n^2 - 1$. Finally, any spanning tree of G has exactly $n^2 - 1$ edges, so the weight of the MST must be at least $n^2 - 1$, thus proving that weight of E is minimal.

3) 16 pts

You are trying to decide the best order in which to answer your CS-570 exam questions within the duration of the exam. Suppose that there are n questions with points p_1, p_2, \dots, p_n and you need time t_i to completely answer the i^{th} question. You are confident that all your completely answered questions will be correct (and get you full credit for them), and the TAs will give you partial credit for an incompletely answered question, in proportion to the time you spent on that question. Assuming that the total exam duration is T , give a greedy algorithm to decide the order in which you should attempt the questions and prove that the algorithm gives you an optimal answer.

Solution: This is similar to the fractional knapsack problem. The greedy algorithm is as follows. Calculate $\frac{p_i}{t_i}$ for each $1 \leq i \leq n$ and sort the questions in descending order of $\frac{p_i}{t_i}$.

Let the sorted order of questions be denoted by $s(1), s(2), \dots, s(n)$. Answer questions in the order $s(1), s(2), \dots$ until $s(j)$ such that $\sum_{k=1}^j t_{s(k)} \leq T$ 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 question $s(j + 1)$ in the time remaining.

We'll use induction to prove optimality of the above algorithm. 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.

Inductive Step: Let $P(1), P(2), \dots, P(l)$ be true for some arbitrary l , i.e. the set of questions $\{s(1), s(2), \dots, s(l)\}$ are part of an optimal solution O . It is clear that O should also be optimal with respect to the remaining questions $\{1, 2, \dots, n\} \setminus \{s(1), s(2), \dots, s(l)\}$ 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 largest 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.

Induction Basis: 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 $\left(\frac{p_{s(1)}}{t_{s(1)}} - \frac{p_a}{t_a}\right) \min(t_a, T, t_{s(1)}) \geq 0$ since $\frac{p_i}{t_i}$ is 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.

4) 16 pts

An Edge Cover on a graph $G = (V; E)$ is a set of edges $X \subseteq E$ such that every vertex in V is incident to an edge in X . In the **Bipartite** Edge Cover problem, we are given a bipartite graph and wish to find an Edge Cover that contains $\leq k$ edges. Design a polynomial-time algorithm based on network flow (max flow or circulation) to solve it and justify your algorithm.

Solution:

If the network contains isolated node, then the problem is trivial since there is no way to do edge cover. Now we only consider the case that each node has at least 1 incident edge.

- i) The goal of edge cover is to choose as many edges as possible which cover 2 nodes. You can find this subset of edges by running Bipartite Matching on the original graph, and taking exactly the edges which are in the matching. (Equivalently, you can set capacity of each edge in the graph as 1. Set a super source node s connecting each “blue” node with edge capacity 1 and a super destination t connecting each “red” node with edge capacity 1. Then run max-flow to get the subset of edges connecting 2 nodes in G)
- ii) What remains is to cover the remaining nodes. Since you can only cover a single node (of those remaining) with each selected edge, simply choose an arbitrary incident edge to each uncovered node.
- iii) Set the set of edges you choose in the above two steps as set X . Count the total number of edges in X and compare the size with k .

Proof:

It is obvious that X is an edge cover. The remaining part is to show that X contains the minimum number of edges among all possible edge covers.

Denote the number of edges we find in step i) as x_1 ; denote the number of edges we find in step ii) as x_2 .

Then we have $x_1 * 2 + x_2 = |V|$.

Consider an arbitrary edge cover set Y . Suppose Y contains y_1 edges, each of which is counted as the one covering 2 nodes. Suppose Y contains y_2 edges, each of which is counted as the one covering 1 nodes. (We can ignore the edges covering zero nodes, because we can delete those edges from Y without affecting the coverage)

Then we have $y_1 * 2 + y_2 = |V|$.

Here x_1 must be the maximum number of edges that covers 2 nodes in the bipartite graph, because we do bipartite matching in G (max-flow in G' including s and t).

Therefore, we have $x_1 \geq y_1$.

Then we have:

$$x_1 + x_2 = |V| - x_1 = y_1 * 2 + y_2 - x_1 = y_1 + y_2 - (x_1 - y_1) \leq y_1 + y_2.$$

The above algorithm gives the minimum number of edges for covering the nodes.

5) 16 pts

Imagine starting with the given decimal number n , and repeatedly chopping off a digit from one end or the other (your choice), until only one digit is left. The square-depth $\text{SQD}(n)$ of n is defined to be the maximum number of perfect squares you could observe among all such sequences. For example, $\text{SQD}(32492) = 3$ via the sequence

$$32492 \rightarrow 3249 \rightarrow 324 \rightarrow 24 \rightarrow 4$$

since 3249, 324, and 4 are perfect squares, and no other sequence of chops gives more than 3 perfect squares. Note that such a sequence may not be unique, e.g.

$$32492 \rightarrow 3249 \rightarrow 249 \rightarrow 49 \rightarrow 9$$

also gives you 3 perfect squares, viz. 3249, 49, and 9.

Describe an efficient algorithm to compute the square-depth $\text{SQD}(n)$, of a given number n , written as a d -digit decimal number $a_1 a_2 \dots a_d$. Analyze your algorithm's running time. Your algorithm should run in time polynomial in d . You may assume the availability of a function `IS_SQUARE(x)` that runs in constant time and returns 1 if x is a perfect square and 0 otherwise.

Solution:

We can solve this using dynamic programming.

First, we define some notation: let $n_{ij} = a_i \dots a_j$. That is, n_{ij} is the number formed by digits i through j of n . Now, define the subproblems by letting $D[i, j]$ be the square-depth of n_{ij} .

The solution to $D[i, j]$ can be found by solving the two subproblems that result from chopping off the left digit and from chopping off the right digit, and adding 1 if n_{ij} itself is square. We can express this as a recurrence:

$$D[i, j] = \max(D[i + 1, j], D[i, j - 1]) + \text{IS-SQUARE}(n_{ij})$$

The base cases are $D[i, i] = \text{IS-SQUARE}(a_i)$, for all $1 \leq i \leq d$. The solution to the general problem is $\text{SQD}(n) = D[1, d]$.

There are $\Theta(d^2)$ subproblems, and each takes $\Theta(1)$ time to solve, so the total running time is $\Theta(d^2)$.

*** Some students did a greedy approach to solve this problem which is not true for this problem. In each step of the program they explore removing which digit results in a perfect square number, however it might be the other non-removed digit that will produce more squares in the future steps.

6) 16 pts

The Longest Path is the problem of deciding whether a graph has a simple path of length greater or equal to a given number k .

a) Show that the Longest Path problem is in NP (2 pts)

b) Show how the Hamiltonian Cycle problem can be reduced to the Longest Path problem. (14 pts)

Note: You can choose to do the reduction in b either directly, or use transitivity of polynomial time reduction to first reduce Hamiltonian Cycle to another problem (X) and then reduce X to Longest Path.

a) Polynomial length certificate: ordered list of nodes on a path of length $\geq k$

polynomial time Certifier:

check that nodes do not repeat in $O(n \log n)$

check that the length of the path is at least k in $O(n)$

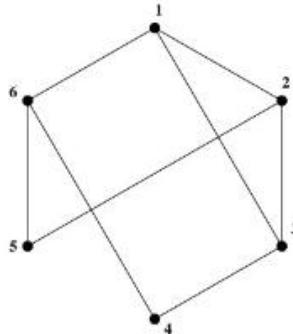
check that there are edges between every two adjacent nodes on the path in $O(n)$

check that the length of the path is at least k in $O(n)$

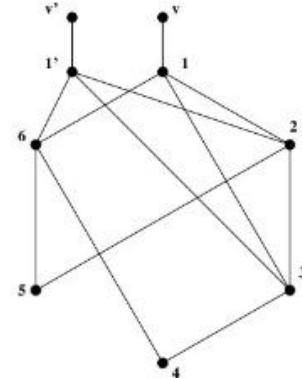
b) Two possible solutions:

I. To reduce directly from Hamiltonian Cycle

Given a graph $G = (V, E)$ we construct a graph G' such that G contains a Ham cycle iff G' contains a simple path of length at least $N+2$. This is done by choosing an arbitrary vertex u in G and adding a copy, u' , of it together with all its edges. Then add vertices v and v' to the graph and connect v with u and v' to u' . We give a cost of 1 to all edges in G' . See figure below:



G



G'

Proof:

A) If there is a HC in G we can find a simple path of length $n+2$ in G' : This path starts at v' goes to U' and follows the HC to u and then to v . The length is $n+2$

B) If there is a simple path of length $n+2$ in G' , there is a HC in G : This path must

include nodes v' and v because there are only n nodes in G and a simple path of length $n+2$ must include v and v' . Moreover, v and v' must be the two ends of this path, otherwise, the path will not be a simple path since there is only one way to get to v and v' . So, to find the HC in G , just follow the path from u' to u .

II. To reduce using Hamiltonian Path

First reduce Ham Cycle to Ham Path (very similar to the above reduction)

Then reduce Ham Path to Longest path. This is very straightforward. To find out if there is a Ham Path in G you can assign weights of 1 to all edges and ask the blackbox if there is a path of length at least n in G .

Additional Space

CS570
Analysis of Algorithms
Fall 2014
Exam III

Name: _____
Student ID: _____
Email: _____

Wednesday Evening Section

	Maximum	Received
Problem 1	20	
Problem 2	16	
Problem 3	16	
Problem 4	16	
Problem 5	16	
Problem 6	16	
Total	100	

Instructions:

1. This is a 2-hr exam. Closed book and notes
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.

1) 20 pts

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

[/FALSE]

All the NP-hard problems are in NP.

[/FALSE]

Given a weighted graph and two nodes, it is possible to list all shortest paths between these two nodes in polynomial time.

[TRUE/]

In the memory efficient implementation of Bellman-Ford, the number of iterations it takes to converge can vary depending on the order of nodes updated within an iteration

[/FALSE]

There is a feasible circulation with demands $\{d_v\}$ if $\sum_v d_v = 0$.

[/FALSE]

Not every decision problem in P has a polynomial time certifier.

[TRUE/]

If a problem can be reduced to linear programming in polynomial time then that problem is in P.

[/FALSE]

If we can prove that $P \neq NP$, then a problem $A \in P$ does not belong to NP.

[/FALSE]

If all capacities in a flow network are integers, then every maximum flow in the network is such that flow value on each edge is an integer.

[/FALSE]

In a dynamic programming formulation, the sub-problems must be mutually independent.

[~~TRUE/~~]

In the final residual graph constructed during the execution of the Ford–Fulkerson Algorithm, there's no path from sink to source.

2) 16 pts

In the Bipartite Directed Hamiltonian Cycle problem, we are given a bipartite directed graph $G = (V; E)$ and asked whether there is a simple cycle which visits every node exactly once. Note that this problem might potentially be easier than Directed Hamiltonian Cycle because it assumes a bipartite graph. Prove that Bipartite Directed Hamiltonian Cycle is in fact still NP-Complete.

Solution:

- i) This problem is NP. Given a candidate path, we just check the nodes along the given path one by one to see if every node in the network is visited exactly once and if each consecutive node pair along the path are joined by an edge.
- ii) To prove the problem is NP-complete, we reduce Directed Hamiltonian Cycle (DHC) problem to Bipartite Directed Hamiltonian Cycle (BDHC) problem.

Given an arbitrary directed graph G , we split each vertex v in G into two vertex v_{in} and v_{out} . Here v_{in} connects all the incoming edges to v in G ; v_{out} connects all the outgoing edges from v in G . Moreover, we connect one directed edge from v_{in} to v_{out} . After doing these operations for each node in G , we form a new graph G' .

Here G' is bipartite graph, because we can color each v_{in} “blue” and each v_{out} “red” without any coloring conflict.

If there is DHC in G , then there is a BDHC in G' . We can replace each node v on the DHC in G into consecutive nodes v_{in} and v_{out} , and (v_{in}, v_{out}) is an edge in G' . Then the new path is a BDHC in G' .

On the other hand, if there is BDHC in G' , then there is a DHC in G . Note that if v_{in} is on the BDHC, v_{out} must be the successive node on the BDHC. Then we can merge each node pair (v_{in}, v_{out}) on the BDHC in G' into node v and form a DHC in G .

In sum, G has DHC if and only if G' has a BDHC. This completes the reduction, and we confirm that the given problem is NP-complete

3) 16 pts

A tourism company is providing boat tours on a river with n consecutive segments. According to previous experience, the profit they can make by providing boat tours on segment i is known as a_i . Here a_i could be positive (they earn money), negative (they lose money), or zero. Because of the administration convenience, the local community of the river requires that the tourism company should do their boat tour business on a contiguous sequence of the river segments, i.e, if the company chooses segment i as the starting segment and segment j as the ending segment, all the segments in between should also be covered by the tour service, no matter whether the company will earn or lose money. The company's goal is to determine the starting segment and ending segment of boat tours along the river, such that their total profit can be maximized. Design an efficient algorithm to achieve this goal, and analyze its run time (Note that brute-force algorithm achieves $\Theta(n^2)$, so your algorithm must do better.)

Solution1:

Using dynamic programming:

Define $\text{OPT}(i)$ as the maximum total profit the company can get when running the boat tours on a contiguous sequence of segments starting at segment i .

Base case: $\text{OPT}(n) = a_n$.

Recursive relation: $\text{OPT}(i) = \max\{\text{OPT}(i+1)+a_i, a_i\}$, for $1 \leq i < n$

Algorithm:

For $i = n, \dots, 1$
 Compute $\text{OPT}(i)$.
End

Maximum profit = $\max\{\text{OPT}(i)\}$. Denote i^* as the starting index which achieves the maximum profit, then i^* is the optimal starting segment

For $i = i^*, \dots, n$
 If ($\text{OPT}(i) = a_i$)
 Set $j^* = i$;
 Break;
 End
End
The value j^* is the index of the optimal ending segment.

Complexity: Computing all the OPT values takes time $O(n)$, comparing all OPT values and find the maximum profit and the corresponding starting segment takes time $O(n)$. Finding the optimal ending segment takes time $O(n)$. In sum, the algorithm takes time $O(n)$

Remark: in this solution, we implicitly assume that the company must provide the boat tour, while providing no boat tour is not a choice. If you consider providing no boat tour also as a choice, it is also treated as a correct solution, however, the step of computing the maximum profit above solution should be modified as follows:

$$\text{Maximum profit} = \max\{0, \max_{\{i\}} \{\text{OPT}(i)\}\}.$$

Correspondingly, if 0 is the best result you can get, you need to claim that providing no boat tour is the best solution, and there is no need to confirm the starting segment or ending segment.

Solution 2 (outline):

Using divide and conquer.

- i) Divide operation: Divide the profit sequences of the n consecutive segments, denoted as $A[1,n]$, as two parts:
 $a_1, \dots, a_{n/2}$ and $a_{n/2+1}, \dots, a_n$, denoted as $A[1,n/2]$ and $A[n/2+1, n]$ respectively.
- ii) Merge operation:
 Suppose you successfully find the maximum profit in $A[1,n/2]$ and $A[n/2+1, n]$, denoted as $P_{left}(1,n)$, $P_{right}(1,n)$, and the corresponding contiguous sequence of segments, denoted as $B_{left}(1,n)$ and $B_{right}(1,n)$
 Then the optimal contiguous segment can be in one of the three cases:
 - a) $B_{left}(1,n)$
 - b) $B_{right}(1,n)$
 - c) An optimal contiguous segment sequence crossing $A[1,n/2]$ and $A[n/2+1, n]$, denoted as $B_{cross}(1,n)$.

For $B_{cross}(1,n)$, denote the corresponding profit as $P_{cross}(1,n)$. The method to confirm $B_{cross}(1,n)$ and $P_{cross}(1,n)$ is as follows:

- Starting from sequence $[a_{n/2}, a_{n/2+1}]$, compute the summation $S_{left}(1) = a_{n/2} + a_{n/2+1}$ as one candidate solution for $P_{cross}(1,n)$.
- Next, including $a_{n/2-1}$ into the above sequence as $[a_{n/2-1}, a_{n/2}, a_{n/2+1}]$, compute the summation of the three elements in the sequence as the second candidate solution: $S_{left}(2) = S_{left}(1) + a_{n/2-1}$.

- Keep including the element one by one to the left until a_1 is included, during each step, compute and record the summation values.
- Find $S_{\text{opt_left}} = \max_i \{S_{\text{left}}(i)\}$, record the corresponding index of the included element that achieves the maximum, say i_{cross^*} . Then i_{cross^*} is the optimal starting index for $B_{\text{cross}}(1,n)$;
- Starting from $[a_{i_{\text{cross}^*}}, \dots, a_{n/2+1}]$ includes the element to the right one by one until a_n is included, during each step, compute the summation values in the same way as in the left part, denote the value as $S_{\text{right}}(i)$.
- Find $P_{\text{cross}}(1,n) = \max_i \{S_{\text{right}}(i)\}$, record the corresponding index of the included element that achieves the maximum, say j_{cross^*} . Then j_{cross^*} is the optimal ending index for $B_{\text{cross}}(1,n)$;

Then

Maximum Profit = $\max \{P_{\text{left}}(1,n), P_{\text{right}}(1,n), P_{\text{cross}}(1,n)\}$, and the corresponding optimal starting index and ending index are the ones that achieve the maximum.

- Before doing step ii) for $A[1,n]$, recursively run the above procedure in each array $A[1,n/2]$ and $A[n/2+1, n]$ and so on.
- Termination condition: for $A[i,j]$, if $i=j$, return a_i as the maximum profit, return i as both the optimal starting and ending index in $A[i,j]$.

Complexity:

The Divide operation takes time $O(1)$.

The Merge operation takes time $O(n)$.

Define $T(n)$ as the running time,

$$T(n) = 2*T(n/2) + O(n)$$

The complexity is $O(n \log(n))$.

Remark: similar as in solution 1, considering no bout tour as a choice is also treated as a correct solution.

4) 16 pts

Consider the following matching problem. There are m students s_1, s_2, \dots, s_m and a set of n companies $C = \{c_1, c_2, \dots, c_n\}$. Each student can work for only one company, whereas company c_j can hire up to b_j students. Student s_i has a preferred set of companies $\Lambda_i \subseteq C$ at which he/she is willing to work. Your task is to find an assignment of students to companies such that all of the above constraints are satisfied and each student is assigned. Formulate this as a network flow problem and describe any subsequent steps necessary to arrive at the solution. Prove correctness.

Construction of flow network: Represent each student and each company as a separate node. Add one source node s and a sink node t for a total of $m + n + 2$ nodes. Add the following directed edges with capacities:

- i. $s \rightarrow s_i$ with capacity 1 unit, for each $1 \leq i \leq m$,
- ii. For each $1 \leq i \leq m$, $s_i \rightarrow c$ with capacity 1 unit for all nodes $c \in \Lambda_i$.
- iii. $c_j \rightarrow t$ with capacity b_j units, for each $1 \leq j \leq n$.

Constructing the solution: Run any max-flow algorithm on the above network to get the realization of edge flows under a max flow configuration (lets call it O for brevity). If an edge $s_i \rightarrow c_j$ shows a non-zero flow in this configuration, assign student s_i to company c_j . Repeat this process for each edge between the student nodes and the company nodes to get the final assignment.

Proof of correctness: We have to show that the solution so obtained satisfies all constraints of the problem.

- i. Since all outgoing edges from s_i are to the node set Λ_i , it is impossible for s_i to have a flow to a node outside Λ_i in configuration O .
- ii. Since the only outgoing edge from c_j is of capacity b_j , configuration O cannot have more than b_j incoming edges of non-zero flow to node c_j . Thus, not more than b_j students can get assigned to company c_j .
- iii. As s_i has a single incoming edge and multiple outgoing edges of capacity 1, configuration O cannot have more than one outgoing edge from s_i with non-zero flow. Hence, s_i can get assigned to at most one company.

We have proved that our mapping from configuration O to an assignment does not violate any of the constraints except possibly that all students might not be assigned. Note that this may happen in practice if it is impossible to assign all students while satisfying all of the given constraints. Thus, what we need to show is that if there exists a feasible assignment then configuration O will have each $s \rightarrow s_i$ edge carry a non-zero flow. Given a feasible assignment $\{(s_i, c_{\sigma(i)}), 1 \leq i \leq m\}$, by construction of the flow network, it is possible to set each edge $s_i \rightarrow c_{\sigma(i)}$ to carry 1 unit of flow. By feasibility of the assignment, company c_j gets no more than b_j incoming edges with non-zero flow, so the outgoing edge from c_j has enough capacity to carry away all incident flow on node c_j . Finally, since each s_i has an outgoing flow of 1 unit in

this assignment, all $s \rightarrow s_i$ edges can be set to have 1 unit of flow, completing the proof.

5) 16 pts

Consider a directed, weighted graph G where all edge weights are positive. You have one Star, which lets you change the weight of any one edge to zero. In other words, you may change the weight of any one edge to zero. Give an efficient algorithm using Dijkstra's algorithm to find a lowest-cost path between two vertices s and t , given that you may set one edge weight to zero. Note: you will receive 10 pts if your algorithm is efficient. You will receive full points (16 pts) if your algorithm has the same run time complexity as Dijkstra's algorithm.

Solution:

Use Dijkstra's algorithm to find the shortest paths from s to all other vertices. Reverse all edges and use Dijkstra to find the shortest paths from all vertices to t . Denote the shortest path from u to v by $u \rightsquigarrow v$, and its length by $\delta(u, v)$.

Now, try setting each edge to zero. For each edge $(u, v) \in E$, consider the path $s \rightsquigarrow u \rightarrow v \rightsquigarrow t$. If we set $w(u, v)$ to zero, the path length is $\delta(s, u) + \delta(v, t)$. Find the edge for which this length is minimized and set it to zero; the corresponding path $s \rightsquigarrow u \rightarrow v \rightsquigarrow t$ is the desired path. The algorithm requires two invocations of Dijkstra, and an additional $\Theta(E)$ time to iterate through the edges and find the optimal edge to take for free. Thus the total running time is the same as that of Dijkstra:

$O(E + V \lg V)$: based on using Fibonacci heap

$O(|V| + |E|) \log (|V|)$: based on using binary heap

6) 16 pts

You are given n rods; they are of length l_1, l_2, \dots, l_n , respectively. Our goal is to connect all the rods and form a single rod. The length after connecting two rods and the cost of connecting them are both equal to the sum of their lengths. Give an algorithm to minimize the cost of connecting them to form a single rod. State the complexity of your algorithm and prove that your algorithm is optimal.

1st interpretation: At each step we connect a single rod to the set of rods that are already connected together.

Of course, the solution is to sort rods by length and connect them in the order of increasing length. To prove this is optimal, you can assume that there is an optimal solution and compare our solution to the optimal solution: At each step our solution stays ahead of (or at least does no worse than) the optimal solution.

Sort Takes $O(n \log n)$

We can use mathematical induction to prove that we always stay ahead of the optimal solution.

Claim: at each step the cost of connecting rods in our solution is less than or equal to that of the other solution and the length of the rod after this connection is smaller than or equal in size to that of the other solution.

Base case: first two shortest rods – obviously this is the opt solution

Assuming that the cost of connecting k rods in our solution is less than or equal to that of another (optimal) solution, we can show that the cost of connecting the next ($k+1^{\text{st}}$) rod will be less than or equal to the cost of connecting the $k+1^{\text{st}}$ rod in the other solution:

Cost of the last connection in our solution = total length of all rods 1 to $k+1$ (ordered by length)

Cost of the last connection in the other solution = total length of all rods 1 to $k+1$ (not necessarily ordered by length)

It is obvious that the cost of our last connection is smaller or equal to the cost of the other connection because the only way to get the minimum total length of $k+1$ rods is to pick the shortest $k+1$ rods.

2nd interpretation: At each step we can connect any rod or set of already connected rods to another rod or set of already connected rods.

The solution is to keep connecting the smallest two rods or sets of already connected rods.

Implementation: Place all rods in a min heap with their key values representing their length. At each step extract two elements from the set and insert the combined rod back into the min heap.

This takes a total of $O(n \log n)$ time

Fact 1: the cost contribution of a rod i to the total assembly cost is $\text{Length}(i) * \text{Level}(i)$, where $\text{Level}(i)$ is the level at which the rod is first assembled with other rods (level 1=root level, level 2= level below root, etc.).

Proof: By observation of cost of assembly tree

Fact2: There is an optimal assembly tree of rods in which the two smallest rods are leaf nodes at the lowest level of the tree and the children of the same parent.

Proof: let's say the two smallest rods are not leaf nodes at the lowest level of the tree, using fact 1, we can swap these rods with two rods at the lowest level of the tree and thereby reducing the total cost of the assembly tree. If the two rods are leaf nodes at the lowest level but not children of the same parent we can swap two rods to make these rods children of the same parent without changing the total cost of the tree (again based on Fact 1).

Assume that there is an optimal assembly tree T^* and our solution produces tree T . We will show that our tree T is also optimal.

To do this, we apply fact 2 to T^* and move the smallest rods to the lowest level of the tree as children of the same parent—without increasing the cost of T^* . We then eliminate the two smallest rods at the bottom of the two trees(since these are the first two rods that are assembled in our algorithm) and assume that there is a new combined rod in place of their parent node. We will get two new trees T^{**} and T' , where

$$\begin{aligned}\text{Total assembly cost of } T^* &= \text{Total assembly cost of } T^{**} + \text{total length of the two smallest rods} \\ \text{Total assembly cost of } T &= \text{Total assembly cost of } T' + \text{total length of the two smallest rods}\end{aligned}$$

Since the costs of the two new trees are reduced by the same amount we can now compare the cost of our new tree T' with the cost of the new optimal tree T^{**} . And show that assembly tree T' is also optimal (as compared to the optimal tree T^{**}).

To do this, we apply fact 2 recursively and place the two smallest rods in T^{**} at the lowest level of the tree and under the same parent node without increasing the cost of T^{**} . These are the next two rods that are combined in our algorithm. We then eliminate these two rods in both trees T' and T^{**} , etc.

By repeating the same steps (applying fact 2 and eliminating the two smallest rods from the optimal assembly tree and our tree) recursively, we will find an optimal solution that follows the same exact assembly sequence that is found in our algorithm. Therefore we can state that our algorithm also produces an optimal assembly tree.

Additional Space

CS570
Analysis of Algorithms
Fall 2014
Exam III

Name: _____
Student ID: _____
Email: _____

Wednesday Evening Section

	Maximum	Received
Problem 1	20	
Problem 2	16	
Problem 3	16	
Problem 4	16	
Problem 5	16	
Problem 6	16	
Total	100	

Instructions:

1. This is a 2-hr exam. Closed book and notes
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.

1) 20 pts

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

[/FALSE]

All integer programming problems can be solved in polynomial time.

[TRUE]

Fractional knapsack problem has a polynomial time greedy solution.

[/FALSE]

For any cycle in a graph, the cheapest edge in the cycle is in a minimum spanning tree.

[/FALSE]

Every decision problem is in NP.

[TRUE/]

If P=NP then P=NP=NP-complete.

[/FALSE]

Ford-Fulkerson algorithm can always terminate if the capacities are real numbers.

[/FALSE]

A flow network with unique edge capacities has a unique min cut.

[/FALSE]

A graph with non-unique edge weights will have at least two minimum spanning trees.

[TRUE/]

A sequence of $O(n)$ priority queue operations can be used to sort a set of n numbers.

[/FALSE]

If a problem X is polynomial time reducible to an NP-complete problem Y, then X is NP-complete.

2) 16 pts

Consider the **complete** weighted graph $G = (V, E)$ with the following properties

- a. The vertices are points on the X-Y plane on a regular $n \times n$ grid, i.e. the set of vertices is given by $V = \{(p, q) | 1 \leq p, q \leq n\}$, where p and q are integer numbers.
- b. The edge weights are given by the usual Euclidean distance, i.e. the weight of the edge between the nodes (i, j) and (k, l) is $\sqrt{(k - i)^2 + (l - j)^2}$.

Prove or disprove: There exists a minimum spanning tree of G such that every node has degree at most two.

Solution:

There does exist an MST of G with every node having degree ≤ 2 . One such MST is obtained by the edges joining **nodes** in the following order: $(1,1) \rightarrow (1,2) \rightarrow (1,3) \rightarrow \dots \rightarrow (1,n-1) \rightarrow (1,n) \rightarrow (2,n) \rightarrow (2,n-1) \rightarrow \dots \rightarrow (2,2) \rightarrow (2,1) \rightarrow (3,1) \rightarrow \dots \rightarrow (n,n)$.

Let us denote the above set of edges by E . To prove that E indeed forms an MST, we need to show that it forms a spanning tree and that it is of minimal weight.

E forms a spanning tree: It is clear that all nodes in the above sequence are distinct. This implies that there are no cycles since any cycle, when written out as a sequence of connected nodes, would necessarily repeat the starting node at the end. It is also clear that all nodes of the graph are covered in the above sequence. In particular, nodes $(i, 1)$, $(i, 2)$, ..., (i, n) are connected in that order for odd integers i and in the reverse order for even integers i . Hence, E forms a spanning tree and has $n^2 - 1$ edges.

E has minimal weight: It is easy to see that every edge in G has at least unit weight, since weight of edge between distinct nodes is minimal for edges between node pairs of the form $\{(i, j), (i+1, j)\}$ or $\{(i, j), (i, j+1)\}$, evaluating to an edge weight of 1. Since all edges in E are of this form, all edges in E have unit weight and total weight of E is equal to $n^2 - 1$. Finally, any spanning tree of G has exactly $n^2 - 1$ edges, so the weight of the MST must be at least $n^2 - 1$, thus proving that weight of E is minimal.

3) 16 pts

You are trying to decide the best order in which to answer your CS-570 exam questions within the duration of the exam. Suppose that there are n questions with points p_1, p_2, \dots, p_n and you need time t_i to completely answer the i^{th} question. You are confident that all your completely answered questions will be correct (and get you full credit for them), and the TAs will give you partial credit for an incompletely answered question, in proportion to the time you spent on that question. Assuming that the total exam duration is T , give a greedy algorithm to decide the order in which you should attempt the questions and prove that the algorithm gives you an optimal answer.

Solution: This is similar to the fractional knapsack problem. The greedy algorithm is as follows. Calculate $\frac{p_i}{t_i}$ for each $1 \leq i \leq n$ and sort the questions in descending order of $\frac{p_i}{t_i}$.

Let the sorted order of questions be denoted by $s(1), s(2), \dots, s(n)$. Answer questions in the order $s(1), s(2), \dots$ until $s(j)$ such that $\sum_{k=1}^j t_{s(k)} \leq T$ 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 question $s(j + 1)$ in the time remaining.

We'll use induction to prove optimality of the above algorithm. 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.

Inductive Step: Let $P(1), P(2), \dots, P(l)$ be true for some arbitrary l , i.e. the set of questions $\{s(1), s(2), \dots, s(l)\}$ are part of an optimal solution O . It is clear that O should also be optimal with respect to the remaining questions $\{1, 2, \dots, n\} \setminus \{s(1), s(2), \dots, s(l)\}$ 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 largest 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.

Induction Basis: 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 $\left(\frac{p_{s(1)}}{t_{s(1)}} - \frac{p_a}{t_a}\right) \min(t_a, T, t_{s(1)}) \geq 0$ since $\frac{p_i}{t_i}$ is 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.

4) 16 pts

An Edge Cover on a graph $G = (V; E)$ is a set of edges $X \subseteq E$ such that every vertex in V is incident to an edge in X . In the **Bipartite** Edge Cover problem, we are given a bipartite graph and wish to find an Edge Cover that contains $\leq k$ edges. Design a polynomial-time algorithm based on network flow (max flow or circulation) to solve it and justify your algorithm.

Solution:

If the network contains isolated node, then the problem is trivial since there is no way to do edge cover. Now we only consider the case that each node has at least 1 incident edge.

- i) The goal of edge cover is to choose as many edges as possible which cover 2 nodes. You can find this subset of edges by running Bipartite Matching on the original graph, and taking exactly the edges which are in the matching. (Equivalently, you can set capacity of each edge in the graph as 1. Set a super source node s connecting each “blue” node with edge capacity 1 and a super destination t connecting each “red” node with edge capacity 1. Then run max-flow to get the subset of edges connecting 2 nodes in G)
- ii) What remains is to cover the remaining nodes. Since you can only cover a single node (of those remaining) with each selected edge, simply choose an arbitrary incident edge to each uncovered node.
- iii) Set the set of edges you choose in the above two steps as set X . Count the total number of edges in X and compare the size with k .

Proof:

It is obvious that X is an edge cover. The remaining part is to show that X contains the minimum number of edges among all possible edge covers.

Denote the number of edges we find in step i) as x_1 ; denote the number of edges we find in step ii) as x_2 .

Then we have $x_1 * 2 + x_2 = |V|$.

Consider an arbitrary edge cover set Y . Suppose Y contains y_1 edges, each of which is counted as the one covering 2 nodes. Suppose Y contains y_2 edges, each of which is counted as the one covering 1 nodes. (We can ignore the edges covering zero nodes, because we can delete those edges from Y without affecting the coverage)

Then we have $y_1 * 2 + y_2 = |V|$.

Here x_1 must be the maximum number of edges that covers 2 nodes in the bipartite graph, because we do bipartite matching in G (max-flow in G' including s and t). Therefore, we have $x_1 \geq y_1$.

Then we have:

$$x_1 + x_2 = |V| - x_1 = y_1 * 2 + y_2 - x_1 = y_1 + y_2 - (x_1 - y_1) \leq y_1 + y_2.$$

The above algorithm gives the minimum number of edges for covering the nodes.

5) 16 pts

Imagine starting with the given decimal number n , and repeatedly chopping off a digit from one end or the other (your choice), until only one digit is left. The square-depth $\text{SQD}(n)$ of n is defined to be the maximum number of perfect squares you could observe among all such sequences. For example, $\text{SQD}(32492) = 3$ via the sequence

$$32492 \rightarrow 3249 \rightarrow 324 \rightarrow 24 \rightarrow 4$$

since 3249, 324, and 4 are perfect squares, and no other sequence of chops gives more than 3 perfect squares. Note that such a sequence may not be unique, e.g.

$$32492 \rightarrow 3249 \rightarrow 249 \rightarrow 49 \rightarrow 9$$

also gives you 3 perfect squares, viz. 3249, 49, and 9.

Describe an efficient algorithm to compute the square-depth $\text{SQD}(n)$, of a given number n , written as a d -digit decimal number $a_1a_2 \dots a_d$. Analyze your algorithm's running time. Your algorithm should run in time polynomial in d . You may assume the availability of a function `IS_SQUARE(x)` that runs in constant time and returns 1 if x is a perfect square and 0 otherwise.

Solution:

We can solve this using dynamic programming.

First, we define some notation: let $n_{ij} = a_i \dots a_j$. That is, n_{ij} is the number formed by digits i through j of n . Now, define the subproblems by letting $D[i, j]$ be the square-depth of n_{ij} .

The solution to $D[i, j]$ can be found by solving the two subproblems that result from chopping off the left digit and from chopping off the right digit, and adding 1 if n_{ij} itself is square. We can express this as a recurrence:

$$D[i, j] = \max(D[i + 1, j], D[i, j - 1]) + \text{IS-SQUARE}(n_{ij})$$

The base cases are $D[i, i] = \text{IS-SQUARE}(a_i)$, for all $1 \leq i \leq d$. The solution to the general problem is $\text{SQD}(n) = D[1, d]$.

There are $\Theta(d^2)$ subproblems, and each takes $\Theta(1)$ time to solve, so the total running time is $\Theta(d^2)$.

*** Some students did a greedy approach to solve this problem which is not true for this problem. In each step of the program they explore removing which digit results in a perfect square number, however it might be the other non-removed digit that will produce more squares in the future steps.

6) 16 pts

The Longest Path is the problem of deciding whether a graph has a simple path of length greater or equal to a given number k .

a) Show that the Longest Path problem is in NP (2 pts)

b) Show how the Hamiltonian Cycle problem can be reduced to the Longest Path problem. (14 pts)

Note: You can choose to do the reduction in b either directly, or use transitivity of polynomial time reduction to first reduce Hamiltonian Cycle to another problem (X) and then reduce X to Longest Path.

a) Polynomial length certificate: ordered list of nodes on a path of length $\geq k$

polynomial time Certifier:

check that nodes do not repeat in $O(n \log n)$

check that the length of the path is at least k in $O(n)$

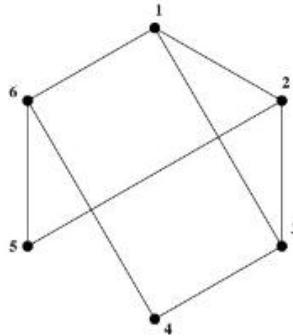
check that there are edges between every two adjacent nodes on the path in $O(n)$

check that the length of the path is at least k in $O(n)$

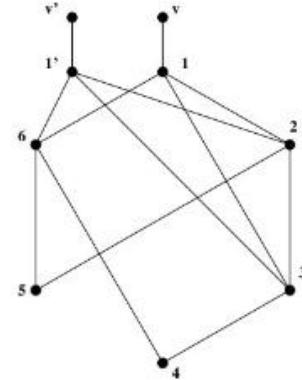
b) Two possible solutions:

I. To reduce directly from Hamiltonian Cycle

Given a graph $G = (V, E)$ we construct a graph G' such that G contains a Ham cycle iff G' contains a simple path of length at least $N+2$. This is done by choosing an arbitrary vertex u in G and adding a copy, u' , of it together with all its edges. Then add vertices v and v' to the graph and connect v with u and v' to u' . We give a cost of 1 to all edges in G' . See figure below:



G



G'

Proof:

A) If there is a HC in G we can find a simple path of length $n+2$ in G' : This path starts at v' goes to U' and follows the HC to u and then to v . The length is $n+2$

B) If there is a simple path of length $n+2$ in G' , there is a HC in G : This path must

include nodes v' and v because there are only n nodes in G and a simple path of length $n+2$ must include v and v' . Moreover, v and v' must be the two ends of this path, otherwise, the path will not be a simple path since there is only one way to get to v and v' . So, to find the HC in G , just follow the path from u' to u .

II. To reduce using Hamiltonian Path

First reduce Ham Cycle to Ham Path (very similar to the above reduction)

Then reduce Ham Path to Longest path. This is very straightforward. To find out if there is a Ham Path in G you can assign weights of 1 to all edges and ask the blackbox if there is a path of length at least n in G .

Additional Space

CS570
Analysis of Algorithms
Fall 2014
Exam III

Name: _____
Student ID: _____
Email: _____

Wednesday Evening Section

	Maximum	Received
Problem 1	20	
Problem 2	16	
Problem 3	16	
Problem 4	16	
Problem 5	16	
Problem 6	16	
Total	100	

Instructions:

1. This is a 2-hr exam. Closed book and notes
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.

1) 20 pts

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

[/FALSE]

All the NP-hard problems are in NP.

[/FALSE]

Given a weighted graph and two nodes, it is possible to list all shortest paths between these two nodes in polynomial time.

[TRUE/]

In the memory efficient implementation of Bellman-Ford, the number of iterations it takes to converge can vary depending on the order of nodes updated within an iteration

[/FALSE]

There is a feasible circulation with demands $\{d_v\}$ if $\sum_v d_v = 0$.

[/FALSE]

Not every decision problem in P has a polynomial time certifier.

[TRUE/]

If a problem can be reduced to linear programming in polynomial time then that problem is in P.

[/FALSE]

If we can prove that $P \neq NP$, then a problem $A \in P$ does not belong to NP.

[/FALSE]

If all capacities in a flow network are integers, then every maximum flow in the network is such that flow value on each edge is an integer.

[/FALSE]

In a dynamic programming formulation, the sub-problems must be mutually independent.

[~~TRUE~~/]

In the final residual graph constructed during the execution of the Ford–Fulkerson Algorithm, there's no path from sink to source.

2) 16 pts

In the Bipartite Directed Hamiltonian Cycle problem, we are given a bipartite directed graph $G = (V; E)$ and asked whether there is a simple cycle which visits every node exactly once. Note that this problem might potentially be easier than Directed Hamiltonian Cycle because it assumes a bipartite graph. Prove that Bipartite Directed Hamiltonian Cycle is in fact still NP-Complete.

Solution:

- i) This problem is NP. Given a candidate path, we just check the nodes along the given path one by one to see if every node in the network is visited exactly once and if each consecutive node pair along the path are joined by an edge.
- ii) To prove the problem is NP-complete, we reduce Directed Hamiltonian Cycle (DHC) problem to Bipartite Directed Hamiltonian Cycle (BDHC) problem.

Given an arbitrary directed graph G , we split each vertex v in G into two vertex v_{in} and v_{out} . Here v_{in} connects all the incoming edges to v in G ; v_{out} connects all the outgoing edges from v in G . Moreover, we connect one directed edge from v_{in} to v_{out} . After doing these operations for each node in G , we form a new graph G' .

Here G' is bipartite graph, because we can color each v_{in} “blue” and each v_{out} “red” without any coloring conflict.

If there is DHC in G , then there is a BDHC in G' . We can replace each node v on the DHC in G into consecutive nodes v_{in} and v_{out} , and (v_{in}, v_{out}) is an edge in G' . Then the new path is a BDHC in G' .

On the other hand, if there is BDHC in G' , then there is a DHC in G . Note that if v_{in} is on the BDHC, v_{out} must be the successive node on the BDHC. Then we can merge each node pair (v_{in}, v_{out}) on the BDHC in G' into node v and form a DHC in G .

In sum, G has DHC if and only if G' has a BDHC. This completes the reduction, and we confirm that the given problem is NP-complete

3) 16 pts

A tourism company is providing boat tours on a river with n consecutive segments. According to previous experience, the profit they can make by providing boat tours on segment i is known as a_i . Here a_i could be positive (they earn money), negative (they lose money), or zero. Because of the administration convenience, the local community of the river requires that the tourism company should do their boat tour business on a contiguous sequence of the river segments, i.e, if the company chooses segment i as the starting segment and segment j as the ending segment, all the segments in between should also be covered by the tour service, no matter whether the company will earn or lose money. The company's goal is to determine the starting segment and ending segment of boat tours along the river, such that their total profit can be maximized. Design an efficient algorithm to achieve this goal, and analyze its run time (Note that brute-force algorithm achieves $\Theta(n^2)$, so your algorithm must do better.)

Solution1:

Using dynamic programming:

Define $\text{OPT}(i)$ as the maximum total profit the company can get when running the boat tours on a contiguous sequence of segments starting at segment i .

Base case: $\text{OPT}(n) = a_n$.

Recursive relation: $\text{OPT}(i) = \max\{\text{OPT}(i+1)+a_i, a_i\}$, for $1 \leq i < n$

Algorithm:

For $i = n, \dots, 1$
 Compute $\text{OPT}(i)$.
End

Maximum profit = $\max_{i=1}^n \{\text{OPT}(i)\}$. Denote i^* as the starting index which achieves the maximum profit, then i^* is the optimal starting segment

For $i = i^*, \dots, n$
 If ($\text{OPT}(i) = a_i$)
 Set $j^* = i$;
 Break;
 End
End
The value j^* is the index of the optimal ending segment.

Complexity: Computing all the OPT values takes time $O(n)$, comparing all OPT values and find the maximum profit and the corresponding starting segment takes time $O(n)$. Finding the optimal ending segment takes time $O(n)$. In sum, the algorithm takes time $O(n)$

Remark: in this solution, we implicitly assume that the company must provide the boat tour, while providing no boat tour is not a choice. If you consider providing no boat tour also as a choice, it is also treated as a correct solution, however, the step of computing the maximum profit above solution should be modified as follows:

$$\text{Maximum profit} = \max\{0, \max_{\{i\}} \{\text{OPT}(i)\}\}.$$

Correspondingly, if 0 is the best result you can get, you need to claim that providing no boat tour is the best solution, and there is no need to confirm the starting segment or ending segment.

Solution 2 (outline):

Using divide and conquer.

- i) Divide operation: Divide the profit sequences of the n consecutive segments, denoted as $A[1,n]$, as two parts:
 $a_1, \dots, a_{n/2}$ and $a_{n/2+1}, \dots, a_n$, denoted as $A[1,n/2]$ and $A[n/2+1, n]$ respectively.
- ii) Merge operation:
 Suppose you successfully find the maximum profit in $A[1,n/2]$ and $A[n/2+1, n]$, denoted as $P_{left}(1,n)$, $P_{right}(1,n)$, and the corresponding contiguous sequence of segments, denoted as $B_{left}(1,n)$ and $B_{right}(1,n)$
 Then the optimal contiguous segment can be in one of the three cases:
 - a) $B_{left}(1,n)$
 - b) $B_{right}(1,n)$
 - c) An optimal contiguous segment sequence crossing $A[1,n/2]$ and $A[n/2+1, n]$, denoted as $B_{cross}(1,n)$.

For $B_{cross}(1,n)$, denote the corresponding profit as $P_{cross}(1,n)$. The method to confirm $B_{cross}(1,n)$ and $P_{cross}(1,n)$ is as follows:

- Starting from sequence $[a_{n/2}, a_{n/2+1}]$, compute the summation $S_{left}(1) = a_{n/2} + a_{n/2+1}$ as one candidate solution for $P_{cross}(1,n)$.
- Next, including $a_{n/2-1}$ into the above sequence as $[a_{n/2-1}, a_{n/2}, a_{n/2+1}]$, compute the summation of the three elements in the sequence as the second candidate solution: $S_{left}(2) = S_{left}(1) + a_{n/2-1}$.

- Keep including the element one by one to the left until a_1 is included, during each step, compute and record the summation values.
- Find $S_{\text{opt_left}} = \max_i \{S_{\text{left}}(i)\}$, record the corresponding index of the included element that achieves the maximum, say i_{cross^*} . Then i_{cross^*} is the optimal starting index for $B_{\text{cross}}(1,n)$;
- Starting from $[a_{i_{\text{cross}^*}}, \dots, a_{n/2+1}]$ includes the element to the right one by one until a_n is included, during each step, compute the summation values in the same way as in the left part, denote the value as $S_{\text{right}}(i)$.
- Find $P_{\text{cross}}(1,n) = \max_i \{S_{\text{right}}(i)\}$, record the corresponding index of the included element that achieves the maximum, say j_{cross^*} . Then j_{cross^*} is the optimal ending index for $B_{\text{cross}}(1,n)$;

Then

Maximum Profit = $\max \{P_{\text{left}}(1,n), P_{\text{right}}(1,n), P_{\text{cross}}(1,n)\}$, and the corresponding optimal starting index and ending index are the ones that achieve the maximum.

- Before doing step ii) for $A[1,n]$, recursively run the above procedure in each array $A[1,n/2]$ and $A[n/2+1, n]$ and so on.
- Termination condition: for $A[i,j]$, if $i=j$, return a_i as the maximum profit, return i as both the optimal starting and ending index in $A[i,j]$.

Complexity:

The Divide operation takes time $O(1)$.

The Merge operation takes time $O(n)$.

Define $T(n)$ as the running time,

$$T(n) = 2*T(n/2) + O(n)$$

The complexity is $O(n \log(n))$.

Remark: similar as in solution 1, considering no bout tour as a choice is also treated as a correct solution.

4) 16 pts

Consider the following matching problem. There are m students s_1, s_2, \dots, s_m and a set of n companies $C = \{c_1, c_2, \dots, c_n\}$. Each student can work for only one company, whereas company c_j can hire up to b_j students. Student s_i has a preferred set of companies $\Lambda_i \subseteq C$ at which he/she is willing to work. Your task is to find an assignment of students to companies such that all of the above constraints are satisfied and each student is assigned. Formulate this as a network flow problem and describe any subsequent steps necessary to arrive at the solution. Prove correctness.

Construction of flow network: Represent each student and each company as a separate node. Add one source node s and a sink node t for a total of $m + n + 2$ nodes. Add the following directed edges with capacities:

- i. $s \rightarrow s_i$ with capacity 1 unit, for each $1 \leq i \leq m$,
- ii. For each $1 \leq i \leq m$, $s_i \rightarrow c$ with capacity 1 unit for all nodes $c \in \Lambda_i$.
- iii. $c_j \rightarrow t$ with capacity b_j units, for each $1 \leq j \leq n$.

Constructing the solution: Run any max-flow algorithm on the above network to get the realization of edge flows under a max flow configuration (lets call it O for brevity). If an edge $s_i \rightarrow c_j$ shows a non-zero flow in this configuration, assign student s_i to company c_j . Repeat this process for each edge between the student nodes and the company nodes to get the final assignment.

Proof of correctness: We have to show that the solution so obtained satisfies all constraints of the problem.

- i. Since all outgoing edges from s_i are to the node set Λ_i , it is impossible for s_i to have a flow to a node outside Λ_i in configuration O .
- ii. Since the only outgoing edge from c_j is of capacity b_j , configuration O cannot have more than b_j incoming edges of non-zero flow to node c_j . Thus, not more than b_j students can get assigned to company c_j .
- iii. As s_i has a single incoming edge and multiple outgoing edges of capacity 1, configuration O cannot have more than one outgoing edge from s_i with non-zero flow. Hence, s_i can get assigned to at most one company.

We have proved that our mapping from configuration O to an assignment does not violate any of the constraints except possibly that all students might not be assigned. Note that this may happen in practice if it is impossible to assign all students while satisfying all of the given constraints. Thus, what we need to show is that if there exists a feasible assignment then configuration O will have each $s \rightarrow s_i$ edge carry a non-zero flow. Given a feasible assignment $\{(s_i, c_{\sigma(i)}), 1 \leq i \leq m\}$, by construction of the flow network, it is possible to set each edge $s_i \rightarrow c_{\sigma(i)}$ to carry 1 unit of flow. By feasibility of the assignment, company c_j gets no more than b_j incoming edges with non-zero flow, so the outgoing edge from c_j has enough capacity to carry away all incident flow on node c_j . Finally, since each s_i has an outgoing flow of 1 unit in

this assignment, all $s \rightarrow s_i$ edges can be set to have 1 unit of flow, completing the proof.

5) 16 pts

Consider a directed, weighted graph G where all edge weights are positive. You have one Star, which lets you change the weight of any one edge to zero. In other words, you may change the weight of any one edge to zero. Give an efficient algorithm using Dijkstra's algorithm to find a lowest-cost path between two vertices s and t , given that you may set one edge weight to zero. Note: you will receive 10 pts if your algorithm is efficient. You will receive full points (16 pts) if your algorithm has the same run time complexity as Dijkstra's algorithm.

Solution:

Use Dijkstra's algorithm to find the shortest paths from s to all other vertices. Reverse all edges and use Dijkstra to find the shortest paths from all vertices to t . Denote the shortest path from u to v by $u \rightsquigarrow v$, and its length by $\delta(u, v)$.

Now, try setting each edge to zero. For each edge $(u, v) \in E$, consider the path $s \rightsquigarrow u \rightarrow v \rightsquigarrow t$. If we set $w(u, v)$ to zero, the path length is $\delta(s, u) + \delta(v, t)$. Find the edge for which this length is minimized and set it to zero; the corresponding path $s \rightsquigarrow u \rightarrow v \rightsquigarrow t$ is the desired path. The algorithm requires two invocations of Dijkstra, and an additional $\Theta(E)$ time to iterate through the edges and find the optimal edge to take for free. Thus the total running time is the same as that of Dijkstra:

$O(E + V \lg V)$: based on using Fibonacci heap

$O(|V| + |E|) \log |V|$): based on using binary heap

6) 16 pts

You are given n rods; they are of length l_1, l_2, \dots, l_n , respectively. Our goal is to connect all the rods and form a single rod. The length after connecting two rods and the cost of connecting them are both equal to the sum of their lengths. Give an algorithm to minimize the cost of connecting them to form a single rod. State the complexity of your algorithm and prove that your algorithm is optimal.

1st interpretation: At each step we connect a single rod to the set of rods that are already connected together.

Of course, the solution is to sort rods by length and connect them in the order of increasing length. To prove this is optimal, you can assume that there is an optimal solution and compare our solution to the optimal solution: At each step our solution stays ahead of (or at least does no worse than) the optimal solution.

Sort Takes $O(n \log n)$

We can use mathematical induction to prove that we always stay ahead of the optimal solution.

Claim: at each step the cost of connecting rods in our solution is less than or equal to that of the other solution and the length of the rod after this connection is smaller than or equal in size to that of the other solution.

Base case: first two shortest rods – obviously this is the opt solution

Assuming that the cost of connecting k rods in our solution is less than or equal to that of another (optimal) solution, we can show that the cost of connecting the next ($k+1^{\text{st}}$) rod will be less than or equal to the cost of connecting the $k+1^{\text{st}}$ rod in the other solution:

Cost of the last connection in our solution = total length of all rods 1 to $k+1$ (ordered by length)

Cost of the last connection in the other solution = total length of all rods 1 to $k+1$ (not necessarily ordered by length)

It is obvious that the cost of our last connection is smaller or equal to the cost of the other connection because the only way to get the minimum total length of $k+1$ rods is to pick the shortest $k+1$ rods.

2nd interpretation: At each step we can connect any rod or set of already connected rods to another rod or set of already connected rods.

The solution is to keep connecting the smallest two rods or sets of already connected rods.

Implementation: Place all rods in a min heap with their key values representing their length. At each step extract two elements from the set and insert the combined rod back into the min heap.

This takes a total of $O(n \log n)$ time

Fact 1: the cost contribution of a rod i to the total assembly cost is $\text{Length}(i) * \text{Level}(i)$, where $\text{Level}(i)$ is the level at which the rod is first assembled with other rods (level 1=root level, level 2= level below root, etc.).

Proof: By observation of cost of assembly tree

Fact2: There is an optimal assembly tree of rods in which the two smallest rods are leaf nodes at the lowest level of the tree and the children of the same parent.

Proof: let's say the two smallest rods are not leaf nodes at the lowest level of the tree, using fact 1, we can swap these rods with two rods at the lowest level of the tree and thereby reducing the total cost of the assembly tree. If the two rods are leaf nodes at the lowest level but not children of the same parent we can swap two rods to make these rods children of the same parent without changing the total cost of the tree (again based on Fact 1).

Assume that there is an optimal assembly tree T^* and our solution produces tree T . We will show that our tree T is also optimal.

To do this, we apply fact 2 to T^* and move the smallest rods to the lowest level of the tree as children of the same parent—without increasing the cost of T^* . We then eliminate the two smallest rods at the bottom of the two trees(since these are the first two rods that are assembled in our algorithm) and assume that there is a new combined rod in place of their parent node. We will get two new trees $T^{*''}$ and T' , where

$$\begin{aligned}\text{Total assembly cost of } T^* &= \text{Total assembly cost of } T^{*''} + \text{total length of the two smallest rods} \\ \text{Total assembly cost of } T &= \text{Total assembly cost of } T' + \text{total length of the two smallest rods}\end{aligned}$$

Since the costs of the two new trees are reduced by the same amount we can now compare the cost of our new tree T' with the cost of the new optimal tree $T^{*''}$. And show that assembly tree T' is also optimal (as compared to the optimal tree $T^{*''}$).

To do this, we apply fact 2 recursively and place the two smallest rods in $T^{*''}$ at the lowest level of the tree and under the same parent node without increasing the cost of $T^{*''}$. These are the next two rods that are combined in our algorithm. We then eliminate these two rods in both trees T' and $T^{*''}$, etc.

By repeating the same steps (applying fact 2 and eliminating the two smallest rods from the optimal assembly tree and our tree) recursively, we will find an optimal solution that follows the same exact assembly sequence that is found in our algorithm. Therefore we can state that our algorithm also produces an optimal assembly tree.

Additional Space

CS570 Spring 2018: Analysis of Algorithms Exam I

	Points
Problem 1	20
Problem 2	10
Problem 3	18
Problem 4	20
Problem 5	15
Problem 6	17
Total	100

Instructions:

1. This is a 2-hr exam. Closed book and notes
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.

1) 20 pts

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

[**TRUE/FALSE**]

If all the edge weights of a given graph are the same, then every spanning tree of that graph is minimum.

[**TRUE/FALSE**]

An in-order traversal of a min-heap outputs the values in sorted order.

[**TRUE/FALSE**]

If all edges in a connected undirected graph have distinct positive weights, the shortest path between any two vertices is unique.

[**TRUE/FALSE**]

If a connected undirected graph $G(V, E)$ has $n = |V|$ vertices and $n + 10$ edges, we can find the minimum spanning tree of G in $O(n)$ runtime.

[**TRUE/FALSE**]

If path P is the shortest path from u to v and w is a node on the path, then the part of path P from u to w is also the shortest path.

[**TRUE/FALSE**]

An amortized cost of insertion into a binomial heap is constant.

[**TRUE/FALSE**]

Gale-Shapley algorithm is a greedy algorithm.

[**TRUE/FALSE**]

For all positive functions $f(n)$, $g(n)$ and $h(n)$, if $f(n) = O(g(n))$ and $f(n) = \Omega(h(n))$, then $g(n) + h(n) = \Omega(f(n))$.

[**TRUE/FALSE**]

Any function which is $\Omega(\log \log n)$ is also $\Omega(\log n)$.

[**TRUE/FALSE**]

The depths of any two leaves in a binomial heap differ by at most 1.

2) 10 pts.

Consider a list data structure that has the following operations defined on it:

- *Append(x)*: Adds the element x to the end of the list
- *DeleteFourth()*: Removes every fourth element in the list i.e. removes the first, fifth, ninth, etc., elements of the list.

Assume that *Append(x)* has a cost 1, and *DeleteFourth()* has a cost equals to the number of elements in the list. What is the amortized cost of *Append* and *DeleteFourth* operations? Consider the worst sequence of operations. Justify your answer using the accounting method.

Solution: *Append* gets charged 5 tokens. When we call *Append*, we spend 1 token immediately to pay for the cost of the call, we then store the remaining 4 tokens with the item added to the list. When we call *DeleteFourth ()*, every element in the list has 4 tokens stored with it. We take the 4 tokens from each item that is deleted to pay for the cost of the call to *DeleteFourth ()*. We can do this since $n/4$ items are deleted and so there are n tokens on these deleted items. Thus, at the end of the call to *DeleteFourth ()* all remaining elements in the list still have 4 tokens stored on them. The amortized cost per operation is thus $O(1)$

3) 18 pts.

For each of the following recurrences, give an expression for the runtime $T(n)$ if the recurrence can be solved with the Master Theorem. Otherwise, indicate that the Master Theorem does not apply.

1. $T(n) = 16 T(n/4) + 5n^3$

solution: $T(n) = \Theta(n^3)$.

2. $T(n) = 4 T(n/2) + n^2 \log n$

solution: $T(n) = \Theta(n^2 \log^2 n)$.

3. $T(n) = 4 T(n/8) - n^2$

solution: does not apply

4. $T(n) = 2^n T(n/2) + n$

solution: does not apply

5. $T(n) = 0.2 T(n/2) + n \log n$

solution: does not apply

6. $T(n) = 4 T(n/2) + n/\log n$

solution: $T(n) = \Theta(n^2)$.

4) 20 pts

You are given a set $X = \{x_1, x_2, \dots, x_n\}$ of points on the real line. Your task is to design a greedy algorithm that finds a smallest set of intervals, each of length-2 that contains all the given points. Linked list is a data structure consisting of a group of nodes which together represent a sequence. Under the simplest form, each node is composed of data and a reference (in other words, a link) to the next node in the sequence.

Example: Suppose that $X = \{1.5, 2.0, 2.1, 5.7, 8.8, 9.1, 10.2\}$. Then the three intervals $[1.5, 3.5]$, $[4, 6]$, and $[8.7, 10.7]$ are length-2 intervals such that every $x \in X$ is contained in one of the intervals. Note that 3 is the minimum possible number of intervals because points 1.5, 5.7, and 8.8 are far enough from each other that they have to be covered by 3 distinct intervals. Also, note that the above solution is not unique.

- a) Describe the steps of your greedy algorithm in plain English. What is its runtime complexity? (10 pts)

```
Sort X = {x1, x2, ... ,xn}. Let it be S.  
Initialize T = {}  
while S not empty:  
    select the smallest x from S  
    add [x, x+2] into T  
    remove all elements within [x, x+2] from S  
return T
```

- b) Argue that your algorithm correctly finds the smallest set of intervals. (10 pts)

Assume there is an optimal solution O. We will prove that the size of our solution A is the same as the size of the optimal solution O.

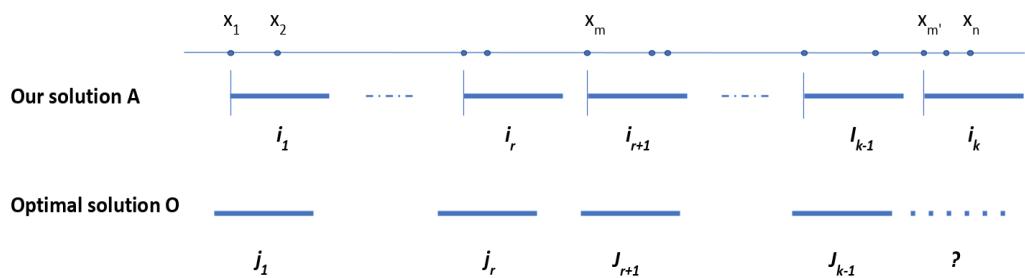
1- We will first prove that intervals in our solution (i_1, i_2, \dots, i_n) are never to the left of the corresponding intervals in the optimal solution (j_1, j_2, \dots, j_p). We will use mathematical induction:

Base Case: i_1 is not to the left of j_1 since if it were then point x_1 will not be covered by j_1
Inductive step: Assume i_r is not to the left of j_r for $r \geq 1$, we will prove that i_{r+1} is not to the left of j_{r+1}

Proof: In above diagram, x_m is the leftmost point covered by interval i_{r+1} . Our solution A uses interval i_{r+1} to cover this point since interval i_r was not covering it. And since i_r is not to the left of j_r then j_r will not be able to cover x_m either. So, if j_{r+1} is any further to right of i_{r+1} then x_m will not be covered in O.

2- Now assume our solution needs k intervals and that the optimal solution O needs fewer intervals. We will prove that this is not possible:

Proof: Our solution needed interval i_k because $x_{m'}$ (the leftmost point covered by i_k) was not covered by i_{k-1} and since in step 1 we proved that our intervals are never to the left of the corresponding intervals in the optimal solution, then $x_{m'}$ will not be covered by j_{k-1} either. So the optimal solution will also need an interval j_k to cover $x_{m'}$. Therefore, the size of our solution is the same as the size of optimal solution.



5) 15 pts.

Given a $n \times n$ matrix where each of the rows and columns are sorted in ascending order, find the k -th smallest element in the matrix using a heap. You may assume that $k < n$. Your algorithm must run in time strictly better than $O(n^2)$.

Solution 1.

1. Build a min heap containing the first element of each row and then run deleteMin to return the smallest element.
2. If the element from the i -th row was deleted, then insert another remaining element in the same row to the min heap. Repeat the procedure k times.
3. The total running time is $O(n + k \log n)$. Minheap is built in $O(n)$ average and worst-time, since the input elements are pre-sorted. A deleteMin operation takes $\Theta(\log n)$, since there are n elements in the . To find the k th smallest elements we require $k-1$ delete and insert operations.

Solution 2. (Using straightforward pruning: for a given integer k , no row or column in range $[k+1,n]$ can have the k th smallest element)

The algorithm is identical to the one in solution 1 except for the following changes.

If $k < n$, for any given k , there is no scenario in which rows $k+1$ to n can have the k th smallest element. This implies that I can prune these rows entirely. Accordingly, I will need to add only k elements (the first elements of the first k rows) to my min heap (in lieu of step 1), giving a total running time of $O(k + k \log k)$ since the heap now contains no more than k elements.

Solution 3. (Using same pruning as above but dumbing-down the solution to near brute force)

1. Insert k elements from each of the k rows into a minheap. Total k^2 elements in min heap => $O(k^2)$,
 2. and output the k th smallest element after k deletemin operations $O(k \log k)$.
- Total running time: $O(k^2)$

Solution 4. (another pruning criterion) (-4 marks)

1. Insert k elements of first row into a max-heap. We will maintain this heap of fixed size k .
2. Iteratively visit each following row, reducing the number of elements visited by 1 in each iteration. (e.g. in second row visit $k-1$ elements in order, then in third row visit $k-2$,...)
3. for each new element encountered, compare with root node of max-heap. If smaller, remove root of max-heap, insert the new element in the heap.

In short, max heap maintains the k smallest elements yet seen.

Total running time: $O(k^2 \log k)$ (Not necessarily better than $O(n^2)$)

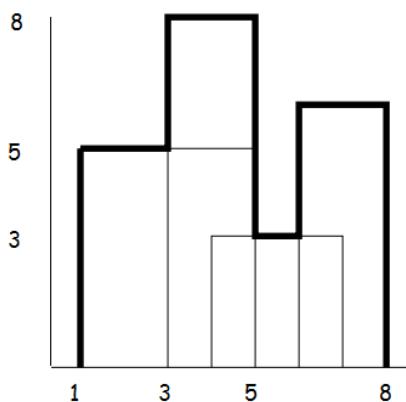
6) 17 pts

Suppose that you are given the exact locations and shapes of several rectangular buildings in a city, and you wish to draw the skyline (in two dimensions) of these buildings, eliminating hidden lines. Assume that the bottoms of all the buildings lie on the x -axis. Each building B_i is represented by a triple (L_i, H_i, R_i) , where L_i and R_i denote the left and right x coordinates of the building, respectively, and H_i denotes the building's height. A skyline is a list of x coordinates and the heights connecting them arranged in order from left to right.

For example, the buildings in the figure below correspond to the following input

$$(1, 5, 5), (4, 3, 7), (3, 8, 5), (6, 6, 8).$$

The skyline is represented as follows: $(1, 5, 3, 8, 5, 3, 6, 6, 8)$. Notice that in the skyline we alternate the x -coordinates and the heights. Also, the x -coordinates are in sorted order.



- a) Given a skyline of n buildings in the form $(x_1, h_1, x_2, h_2, \dots, x_n)$ and another skyline of m buildings in the form $(x'_1, h'_1, x'_2, h'_2, \dots, x'_m)$, show how to compute the combined skyline for the $m + n$ buildings in $O(m + n)$ steps. (5 pts)

Merge the “left” list and the “right” list iteratively by keeping track of the current left height (initially 0), the current right height (initially 0), finding the lowest next x -coordinate in either list; we assume it is the left list. We remove the first two elements, x and h , and set the current left height to h , and output x and the maximum of the current left and right heights.

- b) Assume that we have correctly built a solution to part a), design a divide and conquer algorithm to compute the skyline of a given set of n buildings. Your algorithm should run in $O(n \log n)$ steps. (12 pts)

If there is one building, output it.

Otherwise, split the buildings into two groups, recursively compute skylines, output the result of merging them using part (a).

The runtime is bounded by the recurrence

$T(n) \leq 2T(n/2) + O(n)$, which implies that $T(n) = O(n \log n)$.

CS570
Analysis of Algorithms
Fall 2008
Final Exam

Name: _____
Student ID: _____

____ Monday Section ____ Wednesday Section ____ Friday Section

	Maximum	Received
Problem 1	20	
Problem 2	10	
Problem 3	10	
Problem 4	20	
Problem 5	20	
Problem 6	10	
Problem 7	10	
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]

If $NP = P$, then all problems in NP are NP hard

[FALSE]

L_1 can be reduced to L_2 in Polynomial time and L_2 is in NP, then L_1 is in NP

[FALSE]

The simplex method solves Linear Programming in polynomial time.

[FALSE]

Integer Programming is in P.

[FALSE]

If a linear time algorithm is found for the traveling salesman problem, then every problem in NP can be solved in linear time.

[TRUE]

If there exists a polynomial time 5-approximation algorithm for the general traveling salesman problem then 3-SAT can be solved in polynomial time.

[FALSE]

Consider an undirected graph $G=(V, E)$. Suppose all edge weights are different. Then the longest edge cannot be in the minimum spanning tree.

[FALSE]

Given a set of demands $D = \{d_v\}$ on a directed graph $G(V, E)$, if the total demand over V is zero, then G has a feasible circulation with respect to D .

[TRUE]

For a connected graph G , the BFS tree, DFS tree, and MST all have the same number of edges.

[FALSE]

Dynamic programming sub-problems can overlap but divide and conquer sub-problems do not overlap, therefore these techniques cannot be combined in a single algorithm.

Grading Criteria: Pretty clear, each has two point. These T/F are designed and answered by Professor Shamsian

2) 10 pts

Demonstrate that an algorithm that consists of a polynomial number of calls to a polynomial time subroutine could run in exponential time.

Suggested Solution:

Suppose X takes an input size of n and returns an output size of n^2 . You call X a polynomial number of times say n. If the size of the original input is n the size of the output will be n^{2n} which is exponential WRT n.

Grading Criteria: Rephrasing the question doesn't get that much. If you show you at least understood polynomial / exponential time, you get some credit.

3) 10 pts

Suppose that we are given a weighted, directed graph $G = (V, E)$ in which edges that leave the source vertex s may have negative weights, all other edge weights are nonnegative, and there are no negative-weight cycles. Argue that Dijkstra's algorithm correctly finds shortest paths from s in this graph.

Suggested solution :

```
DIJKSTRA( $G, w, s$ )
1 INITIALIZE-SINGLE-SOURCE( $G, s$ )
2  $S \leftarrow \emptyset$ 
3  $Q \leftarrow V[G]$ 
4 while  $Q \neq \emptyset$ 
5   do  $u \leftarrow \text{EXTRACT-MIN}(Q)$ 
6    $S \leftarrow S \cup \{u\}$ 
7   for each vertex  $v \in \text{Adj}[u]$ 
8     do RELAX( $u, v, w$ )
  
RELAX( $u, v, w$ )
1 if  $d[v] > d[u] + w(u, v)$ 
2 then  $d[v] \leftarrow d[u] + w(u, v)$ 
3  $\pi[v] \leftarrow u$ 
```

We have the algorithm as shown above.

We show that in each iteration of Dijkstra's algorithm, $d[u]=\delta(s,u)$ when u is added to S (the rest follows from the upper-bound property). Let $N^-(s)$ be the set of vertices leaving s , which have negative weights. We divide the proof to vertices in $N^-(s)$ and all the rest. Since there are no negative loops, the shortest path between s and all $u \in N^-(s)$, is through the edge connecting them to s , hence $\delta(s,u)=w(s,u)$, and it follows that after the first time the loop in lines 7,8 is executed $d[u]= w(s,u)=\delta(s,u)$ for all $u \in N^-(s)$ and by the upper bound property $d[u]=\delta(s,u)$ when u is added to S . Moreover it follows that $S= N^-(s)$ after $|N^-(s)|$ steps of the while loop in lines 4-8.

For the rest of the vertices we argue that the proof of Theorem 24-6 (*Theorem 24.6 - Correctness of Dijkstra's algorithm, Introduction to Algorithem by Cormen*) holds since every

shortest path from s includes at most one negative edge (and if does it has to be the first one). To see why this is true assume otherwise, which mean that the path contains a loop, contradicting the property that shortest paths do not contain loops.

Grading Criteria : You need to argue, so bringing just one example shouldn't get you the whole credit , but it may did! If you showed you at least understood Dijkstra, you got some credit too.

4) 20 pts

Phantasy Airlines operates a cargo plane that can hold up to 30000 pounds of cargo occupying up to 20000 cubic feet. We have contracted to transport the following items.

<u>Item type</u>	<u>Weight</u>	<u>Volume</u>	<u>Number</u>	<u>Cost if not carried</u>
1	4000	1000	3	\$800
2	800	1200	10	\$150
3	2000	2200	4	\$300
4	1500	500	5	\$500

For example, we have contracted 10 items of type 2, each of which weighs 800 pounds and takes up to 1200 cubic feet of space. The last column refers to the cost of subcontracting shipment to another carrier.

For each pound we carry, the cost of flying the plane increases by 5 cents. Which items should we put in the plane, and which should we ship via other carrier, in order to have the lowest shipping cost? Formulate this problem as an integer programming problem.

Suggested Solution:

Assume the data in the table are per item.

Considering x_1, x_2, x_3, x_4 are the items that we will ship for types 1,2,3 and 4 respectively.

It is clear that :

$$0 \leq x_1 \leq 3 \quad \& \quad 0 \leq x_2 \leq 10 \quad \& \quad 0 \leq x_3 \leq 4 \quad \& \quad 0 \leq x_4 \leq 5$$

Weight constraint :

$$x_1 * 4000 + x_2 * 800 + x_3 * 2000 + x_4 * 1500 \leq 30,000$$

Volume constraint:

$$x_1 * 1000 + x_2 * 1200 + x_3 * 2200 + x_4 * 500 \leq 20,000$$

Cost of shipping :

$$C_{Ship} = C_A + (x_1 * 4000 + x_2 * 800 + x_3 * 2000 + x_4 * 1500) * \$0.05$$

Where C_A is the cost of operating empty cargo plane

Cost of sub-contracting

$$C_{Sub} = (3 - x_1) * 800 + (10 - x_2) * 150 + (4 - x_3) * 300 + (5 - x_4) * 500$$

Out objective is to minimize $C_{Ship} + C_{Sub}$

Grading Criteria: Some credit will be deducted if you missed some optimization part. Making this simple question complicated may result deduction.

5) 20 pts

Suppose you are given a set of n integers each in the range $0 \dots K$. Give an efficient algorithm to partition these integers into two subsets such that the difference $|S_1 - S_2|$ is minimized, where S_1 and S_2 denote the sums of the elements in each of the two subsets.

Proposed Solution :

$a[1] \dots a[m]$ = the set of integers

$\text{opt}[i,k]$ = true if and only if it is possible to sum to k using elements $1 \dots i$

```
Initialize opt(i,k) = false for all i,k
for(i=1..n) {
    for(k=1..K) {
        if(opt(i,k) == true) {
            for(j=1..m) {
                opt(i,k + a[j]) = true;
            }
        }
    }
}
for(k = floor(K/2)..1) {
    if(opt(n, k)) {
        Return |K - 2k|; // Returns the difference. Partition can be found by back tracking
    }
}
```

6) 10 pts

Adrian has many, many love interests. Many, many, many love interests. The problem is, however, a lot of these individuals know each other, and the last thing our polyamorous TA wants is fighting between his hookups. So our resourceful TA draws a graph of all of his potential partners and draws an edge between them if they know each other. He wants at least k romantic involvements and none of them must know each other (have an edge between them). Can he create his LOVE-SET in polynomial time? Prove your answer.

Proposed Solution :

This is just Independent Set. Proof of NP-completeness was shown in class lecture. We show the correctness as follows: 1) Any Independent Set of size k on the graph will satisfy the requirement that no two love interests know each other (share an edge). 2) If there is a set of k love interests none of which know each other, then there must be a corresponding set of k vertices, such that no two share an edge (know each other).

7) 10 pts

An edge in a flow network is called a bottleneck if increasing its capacity increases the max flow in the network. Give an efficient algorithm for finding all the bottlenecks in the network.

Proposed Solution

Find max flow and the corresponding residual graph, then for each edge increase its capacity by one unit and see if you find a new path from s to t in the residual graph. (Of course you undo this increase before trying the next edge.) If the flow increases for any such edge then that edge must be a bottleneck.

CS570
Analysis of Algorithms
Summer 2009
Final Exam

Name: _____
Student ID: _____

	Maximum	Received
Problem 1	10	
Problem 2	20	
Problem 3	15	
Problem 4	20	
Problem 5		
Problem 6		
Total	100	

2hr exam, closed books and notes.

1) 10 pts

For each of the following sentences, state whether the sentence is known to be TRUE, known to be FALSE, or whether its truth value is still UNKNOWN.

- (a) If a problem is in P, it must also be in NP.
TRUE.
- (b) If a problem is in NP, it must also be in P.
UNKNOWN.
- (c) If a problem is NP-complete, it must also be in NP.
TRUE.
- (d) If a problem is NP-complete, it must not be in P.
UNKNOWN.
- (e) If a problem is not in P, it must be NP-complete.
FALSE.

If a problem is NP-complete, it must also be NP-hard.

TRUE.

If a problem is in NP, it must also be NP-hard.

FALSE.

If we find an efficient algorithm to solve the Vertex Cover problem we have proven that $P=NP$

TRUE.

If we find an efficient algorithm to solve the Vertex Cover problem with an approximation factor $\rho \geq 1$ (a single constant) then we have proven that $P=NP$

FALSE.

If we find an efficient algorithm that takes as input an approximation factor $\rho \geq 1$ and solves the Vertex Cover problem with that approximation factor, we have proven that $P=NP$.

TRUE.

2) 20 pts

Let $G = (V, E)$ be a weighted, directed graph with weight function $w : E \rightarrow \{0, 1, \dots, W\}$ for some nonnegative integer W . Modify Dijkstra's algorithm to compute the shortest paths from a given source vertex s in $O(WV + E)$ time.

Consider running Dijkstra's algorithm on a graph, where the weight function is $w : E \rightarrow \{1, \dots, W - 1\}$. To solve this efficiently, implement the priority queue by an array A of length $WV + 1$. Any node with shortest path estimate d is kept in a linked list at $A[d]$. $A[WV + 1]$ contains the nodes with ∞ as estimate.

EXTRACT-MIN is implemented by searching from the previous minimum shortest path estimate until a new is found. DECREASE-KEY simply moves vertices in the array. The EXTRACT-MIN operations takes a total of $O(VW)$ and the DECREASE-KEY operations take $O(E)$ time in total. Hence the running time of the modified algorithm will be $O(VW + E)$.

3) 15 pts

At a dance, we have n men and n women. The men have height $g(1), \dots, g(n)$, and women $h(1), \dots, h(n)$. For the dance, we want to match up men with women of roughly the same height. Here are the precise rules:

- a. Each man i is matched up with exactly one woman w_i , and each woman with exactly one man.
- b. For each couple (i, w_i) , the mismatch is height difference $|g(i)-h(w_i)|$.
- c. Our goal is to find a matching minimizing the maximum mismatch,
 $\text{MAX}_i |g(i)-h(w_i)|$.

Give an algorithm that runs in $O(n \log n)$ and achieves the desired matching. Provide proof of correctness.

We use an exchange argument. Let w_i denote the optimal solution. If there is any pair i, j such that $i < j$ in our ordering, but $w_i > w_j$ (also with respect to our ordering), then we evaluate the effect of switching to $w'_i := w_j, w'_j := w_i$. The mismatch of no other couple is affected. The couple including man i now has mismatch $|g(i) - h(w'_i)| = |g(i) - h(w_j)|$. Similarly, the other switched couple now has mismatch $|g(j) - h(w_i)|$.

We first look at man i , and distinguish two cases: if $h(w_j) \leq g(i)$ (i.e., man i is at least as tall as his new partner), then the sorting implies that $|g(i) - h(w_j)| = g(i) - h(w_j) \leq g(j) - h(w_j) = |g(j) - h(w_j)|$. On the other hand, if $h(w_j) > g(i)$, then $|g(i) - h(w_j)| = h(w_j) - g(i) \leq h(w_i) - g(i) = |h(w_i) - g(i)|$. In both cases, the mismatch between man i and his new partner is at most the previous maximum mismatch.

We use a similar argument for man j . If $h(w_i) \leq g(j)$, then $|g(j) - h(w_i)| = g(j) - h(w_i) \leq g(j) - h(w_j) = |g(j) - h(w_j)|$. On the other hand, if $h(w_i) > g(j)$, then $|g(j) - h(w_i)| = h(w_i) - g(j) \leq h(w_i) - g(i) = |h(w_i) - g(i)|$. Hence, the mismatch between j and his partner is also no larger than the previous maximum mismatch.

Hence, in all cases, we have that both of the new mismatches are bounded by the larger of the original mismatches. In particular, the maximum mismatch did not increase by the swap. By making such swaps while there are inversions, we gradually transform the optimum solution into ours. This proves that the solution found by the greedy algorithm is in fact optimal.

4) 20 pts

You are given integers p_0, p_1, \dots, p_n and matrices A_1, A_2, \dots, A_n where matrix A_i has dimension $(p_{i-1}) * p_i$

(a) Let $m(i, j)$ denote the minimum number of scalar multiplications needed to evaluate the matrix product $A_i A_{i+1} \dots A_j$. Write down a recursive algorithm to compute $m(i, j)$, $1 \leq i \leq j \leq n$, that runs in $O(n^3)$ time.

Hint: You need to consider the order in which you multiply the matrices together and find the optimal order of operations.

Initialize $m[i, j] = -1 \quad 1 \leq i \leq j \leq n$

Output $mcr(1, n)$

```
mcr(i, j) {  
    if m[i, j] >= 0 return m[i, j]  
    else if i == j m[i, j] = 0  
    else m[i, j] = min (i <= k < j) { mcr(i, k) + mcr(k+1, j) + P_{i-1} * P_k * P_k }  
    return m[i, j]  
}
```

(b) Use this algorithm to compute $m(1,4)$ for $p_0=2$, $p_1=5$, $p_2=3$, $p_3=6$, $p_4=4$

$m[i, j]$

i\j	2	3	4
1	30	66	114
2		90	132
3			72

$m[1, 4] = 114$

5) 20 pts

In a certain town, there are many clubs, and every adult belongs to at least one club. The townspeople would like to simplify their social life by disbanding as many clubs as possible, but they want to make sure that afterwards everyone will still belong to at least one club.

Prove that the Redundant Clubs problem is NP-complete.

First, we must show that Redundant Clubs is in NP, but this is easy: if we are given a set of K clubs, it is straightforward to check in polynomial time whether each person is a member of another club outside this set.

Next, we reduce from a known NP-complete problem, Set Cover. We translate inputs of Set Cover to inputs of Redundant Clubs, so we need to specify how each Redundant Clubs input element

is formed from the Set Cover instance. We use the Set Cover's elements as our translated list of people,

and make a list of clubs, one for each member of the Set Cover family. The members of each club are just the elements of the corresponding family. To finish specifying the Redundant Clubs input,

we need to say what K is: we let $K = F - K_{SC}$ where F is the number of families in the Set Cover instance and K_{SC} is the value K from the set cover instance. This translation can clearly be done in polynomial time (it just involves copying some lists and a single subtraction).

Finally, we need to show that the translation preserves truth values. If we have a yes-instance of Set Cover, that is, an instance with a cover consisting of K_{SC} subsets, the other K subsets form a solution to the translated Redundant Clubs problem, because each person belongs to a club in the

cover. Conversely, if we have K redundant clubs, the remaining K_{SC} clubs form a cover. So the answer

to the Set Cover instance is yes if and only if the answer to the translated Redundant Clubs instance
is yes.

6) 15 pts

A company makes two products (X and Y) using two machines (A and B). Each unit of X that is produced requires 50 minutes processing time on machine A and 30 minutes processing time on machine B. Each unit of Y that is produced requires 24 minutes processing time on machine A and 33 minutes processing time on machine B.

At the start of the current week there are 30 units of X and 90 units of Y in stock. Available processing time on machine A is forecast to be 40 hours and on machine B is forecast to be 35 hours.

The demand for X in the current week is forecast to be 75 units and for Y is forecast to be 95 units—these demands must be met. In addition, company policy is to maximize the combined sum of the units of X and the units of Y in stock at the end of the week.

- a. Formulate the problem of deciding how much of each product to make in the current week as a linear program.

Solution

Let

- x be the number of units of X produced in the current week
- y be the number of units of Y produced in the current week

then the constraints are:

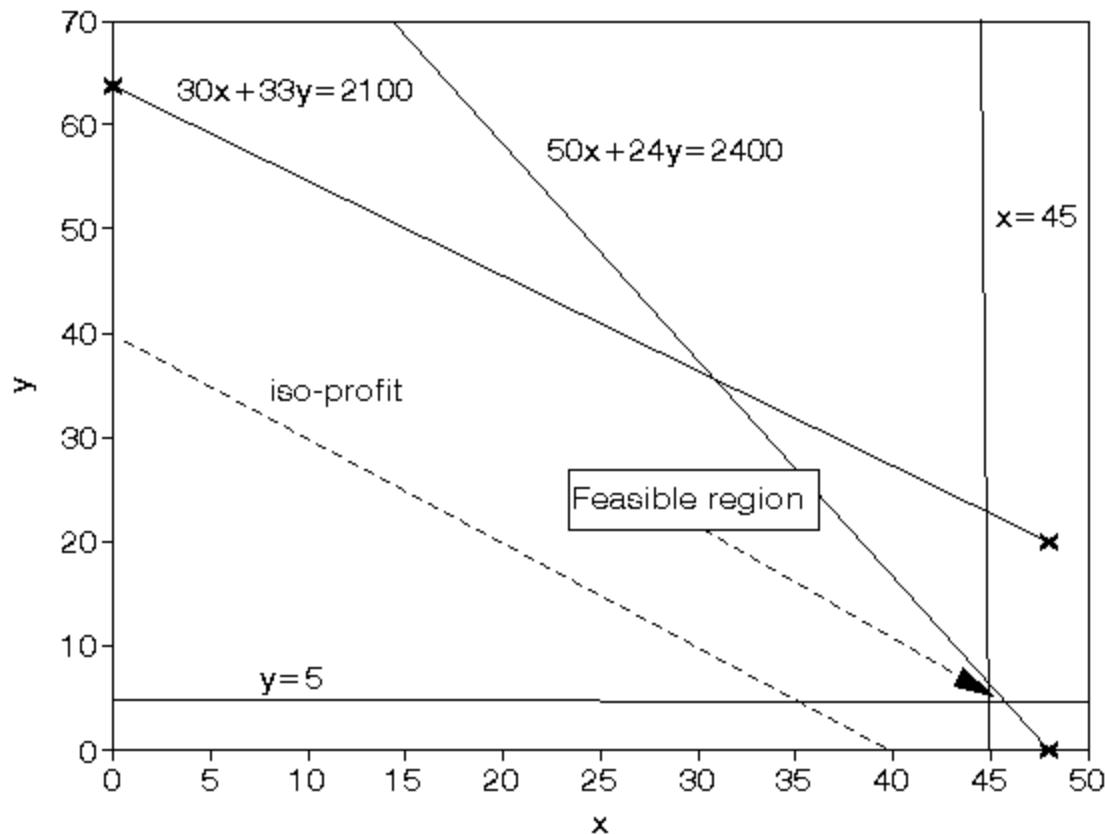
- $50x + 24y \leq 40(60)$ machine A time
- $30x + 33y \leq 35(60)$ machine B time
- $x \geq 75 - 30$
- i.e. $x \geq 45$ so production of X \geq demand (75) - initial stock (30), which ensures we meet demand
- $y \geq 95 - 90$
- i.e. $y \geq 5$ so production of Y \geq demand (95) - initial stock (90), which ensures we meet demand

The objective is: maximise $(x+30-75) + (y+90-95) = (x+y-50)$

i.e. to maximise the number of units left in stock at the end of the week

b. Solve this linear program graphically.

It can be seen in diagram below that the maximum occurs at the intersection of $x=45$ and $50x + 24y = 2400$



Solving simultaneously, rather than by reading values off the graph, we have that $x=45$ and $y=6.25$ with the value of the objective function being 1.25

CS570
Analysis of Algorithms
Fall 2006
Final Exam

Name: _____
Student ID: _____

	Maximum	Received
Problem 1	20	
Problem 2	15	
Problem 3	15	
Problem 4	15	
Problem 5	15	
Problem 6	20	

Note: The exam is closed book closed notes.

1) 20 pts

Mark the following statements as **TRUE**, **FALSE**, or **UNKNOWN**. No need to provide any justification.

[**TRUE/FALSE**]

P is the class of problems that are solvable in polynomial time.

[**TRUE/FALSE**]

NP is the class of problems that are verifiable in polynomial time

[**TRUE/FALSE**]

It is not known whether $P \neq NP$

[**TRUE/FALSE**]

If an NP-complete problem can be solved in linear time, then all the NP complete problems can be solved in linear time

[**TRUE/FALSE**]

Maximum matching problem in a bipartite graph can be efficiently solved thru the solution of an equivalent max flow problem

[**TRUE/FALSE**]

The following recurrence equation has the solution $T(n) = \Theta(n \cdot \log(n^2))$.

$$T(n) = 2T\left(\frac{n}{2}\right) + 3 \cdot n$$

[**TRUE/FALSE/UNKNOWN**]

If a problem is not in P, it should be in NP-complete



[**TRUE/FALSE/UNKNOWN**]

If a problem is in NP, it must also be in P.



[**TRUE/FALSE/UNKNOWN**]

If a problem is NP-complete, it must not be in P.

[**TRUE/FALSE**]

Integer programming is NP-Complete

2) 15 pts

A cargo plane can carry a maximum of 100 tons and a maximum of 60 cubic meters of cargo. There are three materials that need to be carried, and the cargo company may choose to carry any amount of each, up to the maximum amount available of each.

- Material 1 has density 2 tons/cubic meter, maximum available amount 40 cubic meters, and revenue \$1000 per cubic meter.
- Material 2 has density 1 tons/cubic meter, maximum available amount 30 cubic meters, and revenue \$1200 per cubic meter.
- Material 3 has density 3 tons/cubic meter, maximum available amount 20 cubic meters, and revenue \$12000 per cubic meter.

Write a linear program which optimizes revenue within the given constraints.

3) 15 pts

Given two sorted arrays $a[1, \dots, n]$ and $b[1, \dots, n]$, present an $O(\log n)$ algorithm to search the median of their combined $2n$ elements.

4) 15 pts

Present an efficient algorithm for the following problem.

Input: n positive integers $a_1, a_2, a_3, \dots, a_{n-1}, a_n$ and number **t**

Task: Determine if there exists a subset of the a_i 's whose sum equals **t**?

5) 15 pts

There are M faculties and N courses. Every faculty chooses 2 courses based on his/her preference. A faculty can teach zero, one course or two courses and a course can be taught by one faculty only.

Find a feasible course allocation if one exists (A feasible course allocation allows a faculty to teach only the courses he/she prefers).

2) 20 pts

In a weighted graph, the length of path $p = \langle v_0, v_1, \dots, v_k \rangle$ is the sum of the weights of its constituent edges. A path is simple if all vertices in the path are distinct. Let D denote an algorithm for the decision version of the longest path problem. The input consists of a weighted, connected undirected graph $G = (V, E)$ along with an integer k . The output is “yes” if and only if G has a simple path of length k or more.

a) Give an algorithm that uses D to compute the length of the longest path. (You don't have to output the actual path.)

b) (True / False) The longest path problem is NP-complete. If True prove it. If False disprove it.

CS570 FALL 2006 FINAL SOLUTION

Problem 1: 1.True 2.True 3.False 4.False 5.True 6.True 7.False 8.Unknown
9.Unknown 10.True

Problem 2:

Solution 1): Assume that the weight of material 1, material 2 and material 3 carried on the cargo plane are x_1, x_2 and x_3 respectively. Clearly $x_1 \geq 0, x_2 \geq 0, x_3 \geq 0$. Then the total revenue is $1000 * x_1/2 + 1200 * x_2/1 + 12000 * x_3/3$. As the cargo plane can carry a maximum of 100 tons and a maximum of 60 cubic meters of cargo, we have $x_1 + x_2 + x_3 \leq 100, x_1/2 + x_2/1 + x_3/3 \leq 60$; As Material 1 has maximum available amount 40 cubic meters, we have $x_1/2 \leq 40$; As Material 2 has maximum available amount 30 cubic meters, we have $x_2/1 \leq 30$; As Material 3 has maximum available amount 20 cubic meters, we have $x_3/3 \leq 20$; Hence the linear program which optimize revenue is:

$$\max(1000 * x_1/2 + 1200 * x_2/1 + 12000 * x_3/3)$$

subject to:
$$\begin{cases} x_1 \geq 0, \\ x_2 \geq 0, \\ x_3 \geq 0, \\ x_1 + x_2 + x_3 \leq 100, \\ x_1/2 + x_2/1 + x_3/3 \leq 60, \\ x_1/2 \leq 40, \\ x_2/1 \leq 30, \\ x_3/3 \leq 20. \end{cases}$$

Solution 2): Assume that the amount of material 1, material 2 and material 3 carried on the cargo plane are x_1, x_2 and x_3 respectively. Clearly $x_1 \geq 0, x_2 \geq 0, x_3 \geq 0$. Then the total revenue is $1000 * x_1 + 1200 * x_2 + 12000 * x_3$. As the cargo plane can carry a maximum of 100 tons and a maximum of 60 cubic meters of cargo, we have $x_1 + x_2 + x_3 \leq 60, x_1 * 2 + x_2 * 1 + x_3 * 3 \leq 100$; As Material 1 has maximum available amount 40 cubic meters, we have $x_1 \leq 40$; As Material 2 has maximum available amount 30 cubic meters, we have $x_2 \leq 30$; As Material 3 has maximum available amount 20 cubic meters, we have $x_3 \leq 20$; Hence the linear program which optimize revenue is:

$$\max(1000 * x_1 + 1200 * x_2 + 12000 * x_3)$$

subject to:
$$\begin{cases} x_1 \geq 0, \\ x_2 \geq 0, \\ x_3 \geq 0, \\ x_1 + x_2 + x_3 \leq 60, \\ x_1 * 2 + x_2 * 1 + x_3 * 3 \leq 60, \\ x_1 \leq 40, \\ x_2 \leq 30, \\ x_3 \leq 20. \end{cases}$$

Problem 3: The problem is essentially the same as Problem 1 in Chapter 5, which is assigned in HW#5.

We denote $A[1, n], B[1, n]$ as the sorted array in increasing order and $A[k]$ as the $k - th$ element in the array. For brevity of our computation we assume that $n = 2^s$. First we compare $A[n/2]$ to $B[n/2]$. Without loss of generality, assume that $A[n/2] < B[n/2]$, then the elements $A[1], \dots, A[n/2]$ are smaller than n elements, that is, $A[n/2], \dots, A[n]$ and $B[n/2], \dots, B[n]$. Thus $A[1], \dots, A[n/2]$ can't be the median of the two databases. Similarly, $B[n/2], \dots, B[n]$ can't be the median of the two databases either. Note that m is the median value of A and B if and only if m is the median of $A[n/2], \dots, A[n]$ and $B[1], \dots, B[n/2]$ (We delete the same number of numbers that are bigger than m and smaller than m), that is, the $n/2$ smallest number of $A[n/2], \dots, A[n]$ and $B[1], \dots, B[n/2]$. Hence the resulting algorithm is:

```
Algorithm 1 Median-value( $A[1, n], B[1, n], n$ )
if  $n = 1$  then
    return min( $A[n], B[n]$ );
else if  $A[n/2] > B[n/2]$ 
    Median-value( $A[1, n/2], B[n/2, n], n/2$ )
else if  $A[n/2] < B[n/2]$ 
    Median-value( $A[n/2, n], B[1, n/2], n/2$ )
end if
```

For running time, assume that the time for the comparison of two numbers is constant, namely, c , then $T(n) = T(n/2) + c$ and thus $T(n) = c \log n$. Hence our algorithm is $O(\log n)$.

Problem 4: This problem is actually equivalent to solving the Subset Sum problem: Given n items $\{1, \dots, n\}$ and each has a given nonnegative weight a_i , we want to maximize $\sum a_i$ given the condition that $\sum a_i = t$. If the maximum value returned is t , then the output is yes, otherwise false. The recursive relation is:

$$\begin{cases} OPT(i, t) = OPT(i - 1, t), \text{if } t < a_i; \\ OPT(i, t) = \max(OPT(i - 1, t), a_i + OPT(i - 1, t - a_i)), \text{otherwise}; \end{cases}$$

Time complexity: Note the we are building a table of solutions M , and we compute each of the values $M[i, t]$ in $O(1)$ time using the previous values, thus the running time is $O(nt)$.

Or The Subset-Sum algorithm given in the textbook is $O(nt)$, hence the running time of our algorithm is $O(nt)$.

Problem 5: This problem can easily be reduced into max flow problem. We denote the M faculty as $\{x_1, \dots, x_M\}$ and N courses as $\{y_1, \dots, y_N\}$. The reduction is as follows: add source s , sink t in the graph. For each x_i , add edge $s \rightarrow x_i$. The capacity lower bound of these edges is 1 and the capacity upper bound is 2. For each

y_i , add edge $y_i \rightarrow t$. The capacity of each edge is 1. For each faculty x_i , if x_i prefer course y_{i_1} and y_{i_2} , add edges $x_i \rightarrow y_{i_1}$ and $x_i \rightarrow y_{i_2}$ in the graph. The capacity of each edge is 1. To find a feasible allocation, we just need to solve the max flow problem in the constructed graph to find a max flow with value N .

Problem 6:

a)To compute the length of the longest path, we just need to find a integer k such that $D(G, k) = Yes$ and $D(G, k + 1) = No$. Hence the algorithm is:

```

k = 0;
while D(G, k) = Yes
    k++;
return k;

```

b)True. The longest path problem is NP-complete. The proof is as follows: Clearly given a path we can easily check if the length is k in polynomial time, hence the problem is NP . To prove that it is NP-complete, we prove that even the simplest case of longest path problem is NP-complete. The simplest case in our longest path problem is that the weight of each edge is 1. We use a simple reduction from HAMILTON PATH problem: Given an undirected graph, does it have a Hamilton path, i.e, a path visiting each vertex exactly once? HAMILTON PATH is NP-complete. Given an instance $G' = < V', E' >$ for HAMILTON PATH, count the number $|V'|$ of nodes in G' and output the instance $G = G', K = |V'|$ for LONGEST PATH. Obviously, G' has a simple path of length $|V'|$ if and only if G' has a Hamilton path. Therefore, if we can solve LONGEST PATH problem, we can easily solve HAMILTON PATH problem. Hence LONGEST PATH Problem is NP-complete.

CS570
Analysis of Algorithms
Summer 2008
Final Exam

Name: _____
Student ID: _____

____ 4:00 - 5:40 Section ____ 6:00 – 7:40 Section

	Maximum	Received
Problem 1	20	
Problem 2	16	
Problem 3	16	
Problem 4	16	
Problem 5	16	
Problem 6	16	
Total	100	

2 hr exam
Close book and notes

1) 20 pts

For each of the following statements, answer whether it is TRUE or FALSE, and briefly justify your answer.

- a) If a connected undirected graph G has the same weights for every edge, then every spanning tree of G is a minimum spanning tree, but such a spanning tree cannot be found in linear time.

T

- b) Given a flow network G and a maximum flow of G that has already been computed, one can compute a minimum cut of G in linear time.

F



- c) The Ford-Fulkerson Algorithm finds a maximum flow of a unit-capacity flow network with n vertices and m edges in time $O(mn)$ if one uses depth-first search to find an augmenting path in each iteration.

T

- d) Unless $P = NP$, 3-SAT has no polynomial-time algorithm.

F

- e) The problem of deciding whether a given flow f of a given flow network G is a maximum flow can be solved in linear time.

F

- f) If a decision problem A is polynomial-time reducible to a decision problem B (i.e., $A \leq_p B$), and B is NP-complete, then A must be NP-complete.

F

- g) If a decision problem B is polynomial-time reducible to a decision problem A (i.e., $B \leq_p A$), and B is NP-complete, then A must be NP-complete.

T

- h) Integer max flow (where flows and capacities are integers) is polynomial time reducible to linear programming .

F

- i) It has been proved that NP-complete problems cannot be solved in polynomial time.

F

- j) NP is a class of problems for which we do not have polynomial time solutions.

T

2) 16 pts

Suppose that we are given a weighted undirected graph $G = (V, E)$, a source $s \in V$, a sink $t \in V$, and a subset W of vertices V , and want to find a shortest path from s to t that must go through at least one vertex in W . Give an efficient algorithm that solves the problem by invoking any given single-source shortest path algorithm as a subroutine a small number (how many?) times. Show that your algorithm is correct.

Min (pathLength(s, w) + pathLength(w, t)) where w belongs to W

pathLength(i, j) finds the shortest path between i and j . It can be implemented using any existing shortest path algorithm. It will be used $2 * \text{size}(W)$ times

16 pts

Suppose that we have n files F_1, F_2, \dots, F_n of lengths l_1, l_2, \dots, l_n respectively, and want to concatenate them to produce a single file $F = F_1 \circ F_2 \circ \dots \circ F_n$ of length $l_1 + l_2 + \dots + l_n$. Suppose that the only basic operation available is one that concatenates two files. Moreover, the cost of this basic operation on two files of length l_i and l_j , is determined by a cost function $c(l_i, l_j)$ which depends only on the lengths of the two files. Design an efficient dynamic programming algorithm that given n files F_1, F_2, \dots, F_n and a cost function $c(\cdot, \cdot)$, computes a sequence of basic operations that optimizes total cost of concatenating the n input files.

The sub structure of this problem is the time to merge a set of files S . $\text{time}(S)$ – the time needed to merge the files and the files in S should be kept for repeated use

```
if size (S) == 2:  
    # S1 and S2 are the two files in S  
    time (S) = c (length(S1), length(S2))  
else:  
    # f is a file in S  
    # S-f : take f out from S  
    time (S) = min (c (length(f), total length of all other files in S)+time (S-f))
```

Return $\text{time} (S)$ where S contains all the files

4) 16 pts

Define the language

Double-SAT = { ψ : ψ is a Boolean formula with at least 2 distinct satisfying assignments }.

For instance, the formula $\psi: (x \vee y \vee z) \wedge (x' \vee y' \vee z') \wedge (x' \vee y' \vee z)$ is in Double-SAT, since the assignments $(x = 1, y = 0, z = 0)$ and $(x = 0, y = 1, z = 1)$ are two distinct assignments that both satisfy ψ .

Prove that Double-SAT is NP-Complete.

Various methods can be used for reducing SAT to Double-SAT. Since SAT is NP-Complete, Double-SAT is NP complete.

Following is an example for the reduction.

On input $\psi(x_1, \dots, x_n)$:

1. Introduce a new variable y .

2. Output formula $\psi'(x_1, \dots, x_n, y) = \psi(x_1, \dots, x_n) \wedge (y \mid \text{not } y)$.

If $\psi(x_1, \dots, x_n)$ belongs to SAT, then ψ' has at least 1 satisfying assignment, and therefore $\psi'(x_1, \dots, x_n, y)$ has at least 2 satisfying assignments as we can satisfy the new clause $(y \mid \text{not } y)$ by assigning either

$y = 1$ or $y = 0$ to the new variable y , so $\psi'(x_1, \dots, x_n, y)$ belongs to Double-SAT. On the other hand, if $\psi(x_1, \dots, x_n)$ does not belong to SAT, then clearly $\psi'(x_1, \dots, x_n, y) = \psi(x_1, \dots, x_n) \wedge (y \mid \text{not } y)$ has no satisfying assignment either, so $\psi'(x_1, \dots, x_n, y)$ does not belong to Double-SAT. Therefore, SAT can be reduced to Double-SAT. Since the above reduction clearly can be done in P time, Double-SAT is NP-Complete.

5) 16 pts

Let X be a set of n intervals on the real line. A subset of intervals $Y \subset X$ is called a tiling path if the intervals in Y cover the intervals in X , that is, any real value that is contained in some interval in X is also contained in some interval in Y . The size of a tiling cover is just the number of intervals.

Describe and analyze an algorithm to compute the smallest tiling path of X as quickly as possible. Assume that your input consists of two arrays $X_L[1 .. n]$ and $X_R [1 .. n]$, representing the left and right endpoints of the intervals in X .



A set of intervals. The seven shaded intervals form a tiling path

Sort the intervals from left to right by the start position, if the start positions are same, then sort it from the left to right by the end position;

```

FOR (i=1;i<=n; i++) do
    Result[i] = positive unlimited ;
    FOR (j = 1;j < i; j ++) do
        IF (interval i overlap with interval j) then
            IF (Result[i]> result[j]+1) then
                Result[i] = result[j]+1;
Return result [n]

```

The complexity is $O(n^2)$

6) 16 pts

An edge of a flow network is called *critical* if decreasing the capacity of this edge results in a decrease in the maximum flow. Give an efficient algorithm that finds a critical edge in a network.

Find a min cut of the network which has the same capacity as the maximum flow.
Every outgoing edge from the min cut is a critical edge

CS570
Analysis of Algorithms
Spring 2015
Exam III

Name: _____

Student ID: _____

Email Address: _____

_____ **Check if DEN Student**

	Maximum	Received
Problem 1	20	
Problem 2	16	
Problem 3	16	
Problem 4	16	
Problem 5	16	
Problem 6	16	
Total	100	

Instructions:

1. This is a 2-hr exam. Closed book and notes
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.

1) 20 pts

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

[**TRUE**]

If $\text{SAT} \leq_P A$, then A is NP-hard.

[**FALSE**]

If a problem X can be reduced to a known NP-hard problem, then X must be NP-hard.

[**TRUE**]

If P equals NP, then NP equals NP-complete.

[**FALSE**]

Let X be a decision problem. If we prove that X is in the class NP and give a poly-time reduction from X to Hamiltonian Cycle, we can conclude that X is NP-complete.

[**TRUE**]

The recurrence $T(n) = 2T(n/2) + 3n$, has solution $T(n) = \theta(n \log(n^2))$.

[**FALSE**]

On a connected, directed graph with only positive edge weights, Bellman-Ford runs asymptotically as fast as Dijkstra.

[**TRUE**]

Linear programming is at least as hard as the Max Flow problem in a flow network.

[**TRUE**]

If you are given a maximum s-t flow in a graph then you can find a minimum s-t cut in time $O(m)$ where m is the number of the edges in the graph.

[**TRUE**]

Fibonacci heaps can be used to make Dijkstra's algorithm run in $O(|E| + |V| \log|V|)$ time on a graph $G=(V,E)$

[**FALSE**]

A graph with non-unique edge weights will have at least two minimum spanning trees

2) 16 pts

Given a graph $G=(V, E)$ with an even number of vertices as the input, the HALF-IS problem is to decide if G has an independent set of size $|V|/2$. Prove that HALF-IS is in NP-Complete.

Solution:

Given a graph $G(V, E)$ and a certifier $S \subset V$, $|S| = |V|/2$, we can verify if no two nodes are adjacent in polynomial time ($O(|S|^2) = O(|V|^2)$). Therefore $\text{HALF-IS} \in NP$.

We prove the NP-Hardness using a reduction of the NP-complete problem Independent set problem (IS) to HALF-IS . Consider an instance of IS , which asks for an independent set $A \subset V$, $|A| = k$, for a graph $G(V, E)$, such that no two pair of vertices in A are adjacent to each other.

- (i) If $k = \frac{|V|}{2}$, IS reduces to HALF-IS.
- (ii) If $k < \frac{|V|}{2}$, then add m new nodes such that $k + m = (|V| + m)/2$, i.e., $m = |V| - 2k$. Note that the modified set of nodes V' has even number of nodes. Since the additional nodes are all disconnected from each other, they form a subset of independent set. Therefore, the new graph $G'(V', E')$ where $E' = E$ has an independent-set of size $\frac{|V'|}{2}$ if and only if $G(V, E)$ has an independent set of size k .
- (iii) If $k > \frac{|V|}{2}$, then again add $m = |V| - 2k$ new nodes to form the modified set of nodes V' . Connect these new nodes to all the other $|V| + m - 1$ nodes. Since these m new nodes are connected to every other node of them should belong to an independent set. . Therefore, the new graph $G'(V', E')$ has an independent-set of size $\frac{|V'|}{2}$ if and only if $G(V, E)$ has an independent set of size k .

Hence, any instance of IS $(G(V, E), k)$, can be reduced to an instance of HALF-IS $(G'(V', E'))$.

$$IS \leq_P HALF-IS.$$

$\text{HALF-IS} \in NP$ and $\text{HALF-IS} \in NP\text{-Hard} \Rightarrow \text{HALF-IS} \in NP\text{-complete}$.

3) 16 pts

A variant on the decision version of the subset sum problem is as follows: Given a set of n integer numbers $A = \{a_1, a_2, \dots, a_n\}$ and a target number t . Determine if there is a subset of numbers in A whose product is precisely t . That is, the output is *yes* or *no*. Describe an algorithm (and provide pseudo-code) to solve this problem, and analyze its complexity.

Solution:

Solution: *Use dynamic programming:*

Let $S[i,j]$ shows if there exists a subset of $\{a_1, a_2, \dots, a_i\}$ that add up to j , where $0 \leq j \leq t$.

$S[i,j]$ is true or false.

Initialize: $S[0,0] = \text{true}$; $S[0,j] = \text{false}$ if $j \neq 0$

Recurrence formula:

$S[i,j] = S[i-1,j] \text{ OR } S[i-1, j-a_i]$

Output is $S[n,t]$

Complexity is $O(tn)$

4) 16 pts

We've been put in charge of a phone hotline. We need to make sure that it's staffed by at least one volunteer at all times. Suppose we need to design a schedule that makes sure the hotline is staffed in the time interval $[0, h]$. Each volunteer i gives us an interval $[s_i, f_i]$ during which he or she is willing to work. We'd like to design an algorithm which determines the minimum number of volunteers needed to keep the hotline running. Design an efficient greedy algorithm for this problem that runs in time $O(n \log n)$ if there are n student volunteers. Prove that your algorithm is correct. You may assume that any time instance has at least one student who is willing to work for that time.

Solution:

Algorithm: Initially, select the student who can start at or before time 0 and whose finish time is the latest. For each subsequent volunteer, select the one whose start time is no later than the finish time of the last selected volunteer and whose finish time is the latest. Keep selecting volunteers sequentially in this way until the interval $[0, h]$ is covered.

The running time is $O(n \log n)$:

First, sort the start time of all the volunteers takes time $O(n \log n)$; then, searching and selecting the valid successive volunteers with the latest finish time takes $O(n)$ time in total (each volunteer is checked at most once).

Proof:

Let g_1, g_2, \dots, g_m be the sequence of volunteers we selected according to the greedy algorithm; let p_1, p_2, \dots, p_k be the sequence of selected volunteers of an optimal solution.

Clearly, for any feasible solution, we must have someone who can starts before or at time 0. So in the above two solutions: g_1 and p_1 must start at or before time 0.

According to our algorithm, we have $f_{g1} \geq f_{p1}$. By replacing p_1 by g_1 in the optimal solution, we get another solution: g_1, p_2, \dots, p_k , which uses k volunteers and cover the interval $[0, h]$ and therefore is another optimal solution.

Induction hypothesis: assume that our greedy solution is the same as an optimal solution up to the $r-1$ th selected volunteer, i.e., $g_1, \dots, g_{r-1}, p_r, \dots, p_k$ is an optimal solution. With the same argument: $f_{g_r} \geq f_{p_r}$ by replacing p_r by g_r in the optimal solution, we get another solution $g_1, \dots, g_r, p_{r+1}, \dots, p_k$, which is also optimal.

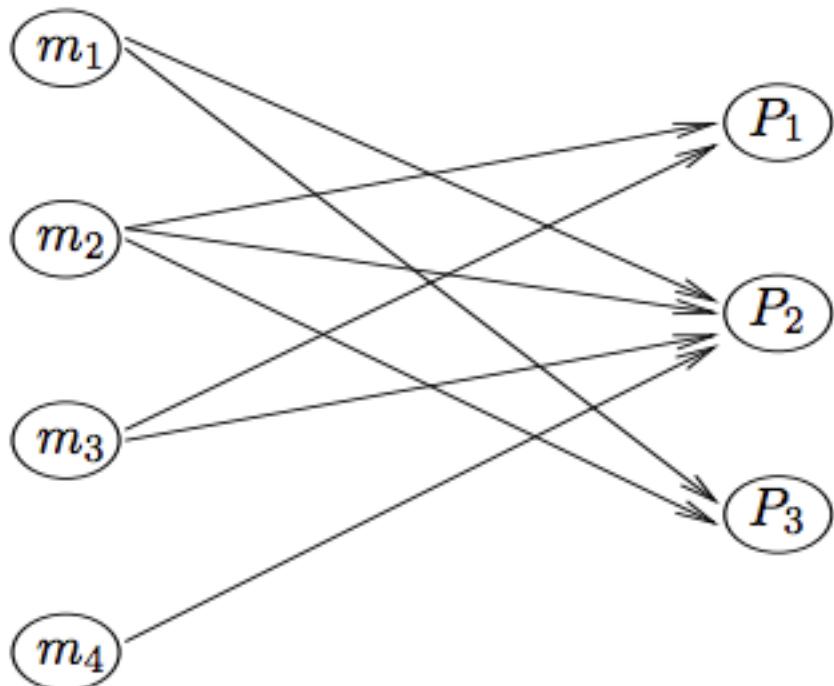
By induction: it follows that g_1, g_2, \dots, g_m is an optimal solution and $m=k$.

5) 16 pts

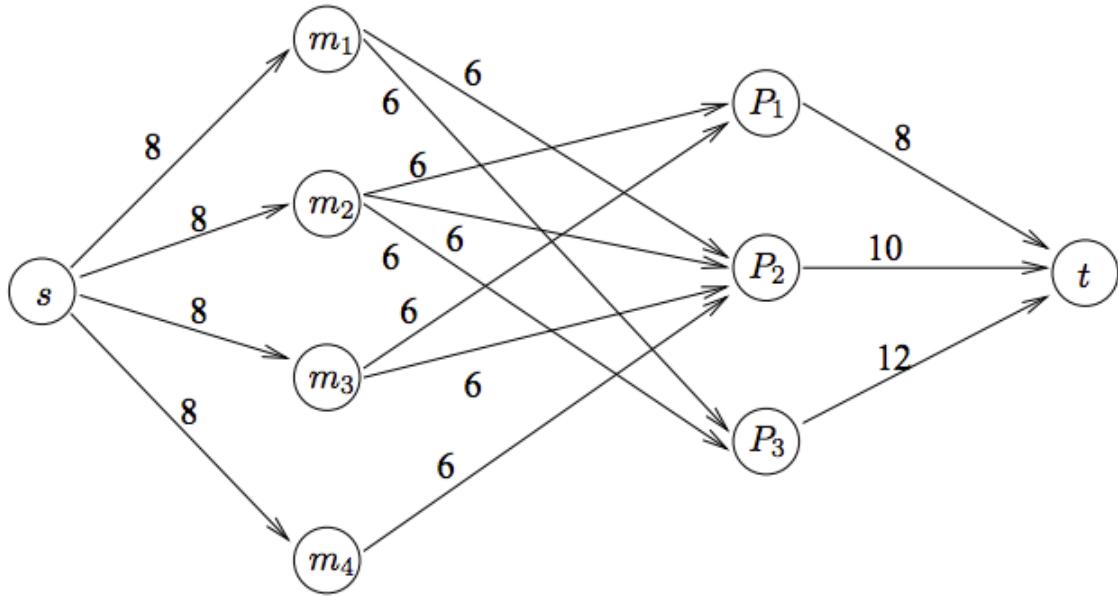
A software house has to handle 3 projects, P_1, P_2, P_3 , over the next 4 months. P_1 can only begin after month 1, and must be completed within month 3. P_2 and P_3 can begin at month 1, and must be completed, respectively, within month 4 and 2. The projects require, respectively, 8, 10, and 12 man-months. For each month, 8 engineers are available. Due to the internal structure of the company, at most 6 engineers can be working, at the same time, on the same project. Determine whether it is possible to complete the projects within the time constraints. Describe how to reduce this problem to the problem of finding a maximum flow in a flow network and justify your reduction.

Solution

We build a product network where months and projects are represented by, respectively, month-nodes m_1, m_2, m_3, m_4 and project-nodes P_1, P_2, P_3 . Each (i, j) edge denotes the possibility of allocating man-hours of month i to project j . For instance, since project P_1 can only begin after month 1 and must be completed before month 3, only arcs outgoing from m_2, m_3 are incident in P_1 .



We add to super nodes s, t , denoting the source and sink of the flow that represents the allocation of men-hours.



All edges outgoing from s have capacity 8, equivalent to the number of available engineers per month. All edges connecting month-nodes to project-nodes have capacity 6, as no more than 6 engineers can work on the same project in the same month. All edges incident in t have capacity equivalent to the number of man-months needed to complete the project. Since all capacities are integer, the maximum flow will be integer as well. To check whether all projects can be completed within the time limits, it suffices to check whether the network admits a feasible flow of value $8 + 10 + 12 = 30$.

6) 16 pts

There are n people and n jobs. You are given a cost matrix, C, where C[i][j] represents the cost of assigning person i to do job j. You want to assign all the jobs to people and also only one job to a person. You also need to minimize the total cost of your assignment. Can this problem be formulated as a linear program? If yes, give the linear programming formulation. If no, describe why it cannot be formulated as an LP and show how it can be reduced to an integer program.

Solution:

We need a 0,1 decision variable to solve the problem and therefore we need to formulate this as an integer program. Below is a formulation of integer program.

Let $x_{ij} = 1$, if job j is assigned to worker i.
 $= 0$, if job j is not assigned to worker i.

Objective function: Minimize

$$\sum_{i=1}^m \sum_{j=1}^n c_{ij} x_{ij}$$

Constraints:

$$\sum_{i=1}^m x_{ij} = 1, \text{ for } j = 1, 2, \dots, n$$

$$\sum_{j=1}^n x_{ij} = 1, \text{ for } i = 1, 2, \dots, m$$

$$x_{ij} = 0 \text{ or } 1$$

Additional Space

Additional Space

CS570
Analysis of Algorithms
Fall 2006
Final Exam

Name: _____
Student ID: _____

	Maximum	Received
Problem 1	20	
Problem 2	15	
Problem 3	15	
Problem 4	15	
Problem 5	15	
Problem 6	20	

Note: The exam is closed book closed notes.

1) 20 pts

Mark the following statements as **TRUE**, **FALSE**, or **UNKNOWN**. No need to provide any justification.

[**TRUE/FALSE**]

P is the class of problems that are solvable in polynomial time.

[**TRUE/FALSE**]

NP is the class of problems that are verifiable in polynomial time

[**TRUE/FALSE**]

It is not known whether $P \neq NP$

[**TRUE/FALSE**]

If an NP-complete problem can be solved in linear time, then all the NP complete problems can be solved in linear time

[**TRUE/FALSE**]

Maximum matching problem in a bipartite graph can be efficiently solved thru the solution of an equivalent max flow problem

[**TRUE/FALSE**]

The following recurrence equation has the solution $T(n) = \Theta(n \cdot \log(n^2))$.

$$T(n) = 2T\left(\frac{n}{2}\right) + 3 \cdot n$$

[**TRUE/FALSE/UNKNOWN**]

If a problem is not in P, it should be in NP-complete

[**TRUE/FALSE/UNKNOWN**]

If a problem is in NP, it must also be in P.

[**TRUE/FALSE/UNKNOWN**]

If a problem is NP-complete, it must not be in P.

[**TRUE/FALSE**]

Integer programming is NP-Complete

2) 15 pts

A cargo plane can carry a maximum of 100 tons and a maximum of 60 cubic meters of cargo. There are three materials that need to be carried, and the cargo company may choose to carry any amount of each, up to the maximum amount available of each.

- Material 1 has density 2 tons/cubic meter, maximum available amount 40 cubic meters, and revenue \$1000 per cubic meter.
- Material 2 has density 1 tons/cubic meter, maximum available amount 30 cubic meters, and revenue \$1200 per cubic meter.
- Material 3 has density 3 tons/cubic meter, maximum available amount 20 cubic meters, and revenue \$12000 per cubic meter.

Write a linear program which optimizes revenue within the given constraints.

3) 15 pts

Given two sorted arrays $a[1, \dots, n]$ and $b[1, \dots, n]$, present an $O(\log n)$ algorithm to search the median of their combined $2n$ elements.

4) 15 pts

Present an efficient algorithm for the following problem.

Input: n positive integers $a_1, a_2, a_3, \dots, a_{n-1}, a_n$ and number **t**

Task: Determine if there exists a subset of the a_i 's whose sum equals **t**?

5) 15 pts

There are M faculties and N courses. Every faculty chooses 2 courses based on his/her preference. A faculty can teach zero, one course or two courses and a course can be taught by one faculty only.

Find a feasible course allocation if one exists (A feasible course allocation allows a faculty to teach only the courses he/she prefers).

2) 20 pts

In a weighted graph, the length of path $p = \langle v_0, v_1, \dots, v_k \rangle$ is the sum of the weights of its constituent edges. A path is simple if all vertices in the path are distinct. Let D denote an algorithm for the decision version of the longest path problem. The input consists of a weighted, connected undirected graph $G = (V, E)$ along with an integer k . The output is “yes” if and only if G has a simple path of length k or more.

a) Give an algorithm that uses D to compute the length of the longest path. (You don't have to output the actual path.)

b) (True / False) The longest path problem is NP-complete. If True prove it. If False disprove it.

CS570 FALL 2006 FINAL SOLUTION

Problem 1: 1.True 2.True 3.False 4.False 5.True 6.True 7.False 8.Unknown
9.Unknown 10.True

Problem 2:

Solution 1): Assume that the weight of material 1, material 2 and material 3 carried on the cargo plane are x_1, x_2 and x_3 respectively. Clearly $x_1 \geq 0, x_2 \geq 0, x_3 \geq 0$. Then the total revenue is $1000 * x_1/2 + 1200 * x_2/1 + 12000 * x_3/3$. As the cargo plane can carry a maximum of 100 tons and a maximum of 60 cubic meters of cargo, we have $x_1 + x_2 + x_3 \leq 100, x_1/2 + x_2/1 + x_3/3 \leq 60$; As Material 1 has maximum available amount 40 cubic meters, we have $x_1/2 \leq 40$; As Material 2 has maximum available amount 30 cubic meters, we have $x_2/1 \leq 30$; As Material 3 has maximum available amount 20 cubic meters, we have $x_3/3 \leq 20$; Hence the linear program which optimize revenue is:

$$\max(1000 * x_1/2 + 1200 * x_2/1 + 12000 * x_3/3)$$

subject to:
$$\begin{cases} x_1 \geq 0, \\ x_2 \geq 0, \\ x_3 \geq 0, \\ x_1 + x_2 + x_3 \leq 100, \\ x_1/2 + x_2/1 + x_3/3 \leq 60, \\ x_1/2 \leq 40, \\ x_2/1 \leq 30, \\ x_3/3 \leq 20. \end{cases}$$

Solution 2): Assume that the amount of material 1, material 2 and material 3 carried on the cargo plane are x_1, x_2 and x_3 respectively. Clearly $x_1 \geq 0, x_2 \geq 0, x_3 \geq 0$. Then the total revenue is $1000 * x_1 + 1200 * x_2 + 12000 * x_3$. As the cargo plane can carry a maximum of 100 tons and a maximum of 60 cubic meters of cargo, we have $x_1 + x_2 + x_3 \leq 60, x_1 * 2 + x_2 * 1 + x_3 * 3 \leq 100$; As Material 1 has maximum available amount 40 cubic meters, we have $x_1 \leq 40$; As Material 2 has maximum available amount 30 cubic meters, we have $x_2 \leq 30$; As Material 3 has maximum available amount 20 cubic meters, we have $x_3 \leq 20$; Hence the linear program which optimize revenue is:

$$\max(1000 * x_1 + 1200 * x_2 + 12000 * x_3)$$

subject to:
$$\begin{cases} x_1 \geq 0, \\ x_2 \geq 0, \\ x_3 \geq 0, \\ x_1 + x_2 + x_3 \leq 60, \\ x_1 * 2 + x_2 * 1 + x_3 * 3 \leq 60, \\ x_1 \leq 40, \\ x_2 \leq 30, \\ x_3 \leq 20. \end{cases}$$

Problem 3: The problem is essentially the same as Problem 1 in Chapter 5, which is assigned in HW#5.

We denote $A[1, n], B[1, n]$ as the sorted array in increasing order and $A[k]$ as the $k - th$ element in the array. For brevity of our computation we assume that $n = 2^s$. First we compare $A[n/2]$ to $B[n/2]$. Without loss of generality, assume that $A[n/2] < B[n/2]$, then the elements $A[1], \dots, A[n/2]$ are smaller than n elements, that is, $A[n/2], \dots, A[n]$ and $B[n/2], \dots, B[n]$. Thus $A[1], \dots, A[n/2]$ can't be the median of the two databases. Similarly, $B[n/2], \dots, B[n]$ can't be the median of the two databases either. Note that m is the median value of A and B if and only if m is the median of $A[n/2], \dots, A[n]$ and $B[1], \dots, B[n/2]$ (We delete the same number of numbers that are bigger than m and smaller than m), that is, the $n/2$ smallest number of $A[n/2], \dots, A[n]$ and $B[1], \dots, B[n/2]$. Hence the resulting algorithm is:

```
Algorithm 1 Median-value( $A[1, n], B[1, n], n$ )
if  $n = 1$  then
    return min( $A[n], B[n]$ );
else if  $A[n/2] > B[n/2]$ 
    Median-value( $A[1, n/2], B[n/2, n], n/2$ )
else if  $A[n/2] < B[n/2]$ 
    Median-value( $A[n/2, n], B[1, n/2], n/2$ )
end if
```

For running time, assume that the time for the comparison of two numbers is constant, namely, c , then $T(n) = T(n/2) + c$ and thus $T(n) = c \log n$. Hence our algorithm is $O(\log n)$.

Problem 4: This problem is actually equivalent to solving the Subset Sum problem: Given n items $\{1, \dots, n\}$ and each has a given nonnegative weight a_i , we want to maximize $\sum a_i$ given the condition that $\sum a_i = t$. If the maximum value returned is t , then the output is yes, otherwise false. The recursive relation is:

$$\begin{cases} OPT(i, t) = OPT(i - 1, t), \text{if } t < a_i; \\ OPT(i, t) = \max(OPT(i - 1, t), a_i + OPT(i - 1, t - a_i)), \text{otherwise}; \end{cases}$$

Time complexity: Note the we are building a table of solutions M , and we compute each of the values $M[i, t]$ in $O(1)$ time using the previous values, thus the running time is $O(nt)$.

Or The Subset-Sum algorithm given in the textbook is $O(nt)$, hence the running time of our algorithm is $O(nt)$.

Problem 5: This problem can easily be reduced into max flow problem. We denote the M faculty as $\{x_1, \dots, x_M\}$ and N courses as $\{y_1, \dots, y_N\}$. The reduction is as follows: add source s , sink t in the graph. For each x_i , add edge $s \rightarrow x_i$. The capacity lower bound of these edges is 1 and the capacity upper bound is 2. For each

y_i , add edge $y_i \rightarrow t$. The capacity of each edge is 1. For each faculty x_i , if x_i prefer course y_{i_1} and y_{i_2} , add edges $x_i \rightarrow y_{i_1}$ and $x_i \rightarrow y_{i_2}$ in the graph. The capacity of each edge is 1. To find a feasible allocation, we just need to solve the max flow problem in the constructed graph to find a max flow with value N .

Problem 6:

a)To compute the length of the longest path, we just need to find a integer k such that $D(G, k) = Yes$ and $D(G, k + 1) = No$. Hence the algorithm is:

```

k = 0;
while D(G, k) = Yes
    k++;
return k;

```

b)True. The longest path problem is NP-complete. The proof is as follows: Clearly given a path we can easily check if the length is k in polynomial time, hence the problem is NP . To prove that it is NP-complete, we prove that even the simplest case of longest path problem is NP-complete. The simplest case in our longest path problem is that the weight of each edge is 1. We use a simple reduction from HAMILTON PATH problem: Given an undirected graph, does it have a Hamilton path, i.e, a path visiting each vertex exactly once? HAMILTON PATH is NP-complete. Given an instance $G' = < V', E' >$ for HAMILTON PATH, count the number $|V'|$ of nodes in G' and output the instance $G = G', K = |V'|$ for LONGEST PATH. Obviously, G' has a simple path of length $|V'|$ if and only if G' has a Hamilton path. Therefore, if we can solve LONGEST PATH problem, we can easily solve HAMILTON PATH problem. Hence LONGEST PATH Problem is NP-complete.

CS570
Analysis of Algorithms
Summer 2008
Final Exam

Name: _____
Student ID: _____

____ 4:00 - 5:40 Section ____ 6:00 – 7:40 Section

	Maximum	Received
Problem 1	20	
Problem 2	16	
Problem 3	16	
Problem 4	16	
Problem 5	16	
Problem 6	16	
Total	100	

2 hr exam
Close book and notes

1) 20 pts

For each of the following statements, answer whether it is TRUE or FALSE, and briefly justify your answer.

- a) If a connected undirected graph G has the same weights for every edge, then every spanning tree of G is a minimum spanning tree, but such a spanning tree cannot be found in linear time.

T

- b) Given a flow network G and a maximum flow of G that has already been computed, one can compute a minimum cut of G in linear time.

F

- c) The Ford-Fulkerson Algorithm finds a maximum flow of a unit-capacity flow network with n vertices and m edges in time $O(mn)$ if one uses depth-first search to find an augmenting path in each iteration.

T

- d) Unless $P = NP$, 3-SAT has no polynomial-time algorithm.

F

- e) The problem of deciding whether a given flow f of a given flow network G is a maximum flow can be solved in linear time.

F

- f) If a decision problem A is polynomial-time reducible to a decision problem B (i.e., $A \leq_p B$), and B is NP-complete, then A must be NP-complete.

F

- g) If a decision problem B is polynomial-time reducible to a decision problem A (i.e., $B \leq_p A$), and B is NP-complete, then A must be NP-complete.

T

- h) Integer max flow (where flows and capacities are integers) is polynomial time reducible to linear programming .

F

- i) It has been proved that NP-complete problems cannot be solved in polynomial time.

F

- j) NP is a class of problems for which we do not have polynomial time solutions.

T

2) 16 pts

Suppose that we are given a weighted undirected graph $G = (V, E)$, a source $s \in V$, a sink $t \in V$, and a subset W of vertices V , and want to find a shortest path from s to t that must go through at least one vertex in W . Give an efficient algorithm that solves the problem by invoking any given single-source shortest path algorithm as a subroutine a small number (how many?) times. Show that your algorithm is correct.

Min (pathLength(s, w) + pathLength(w, t)) where w belongs to W

pathLength(i, j) finds the shortest path between i and j . It can be implemented using any existing shortest path algorithm. It will be used $2 * \text{size}(W)$ times

16 pts

Suppose that we have n files F_1, F_2, \dots, F_n of lengths l_1, l_2, \dots, l_n respectively, and want to concatenate them to produce a single file $F = F_1 \circ F_2 \circ \dots \circ F_n$ of length $l_1 + l_2 + \dots + l_n$. Suppose that the only basic operation available is one that concatenates two files. Moreover, the cost of this basic operation on two files of length l_i and l_j , is determined by a cost function $c(l_i, l_j)$ which depends only on the lengths of the two files. Design an efficient dynamic programming algorithm that given n files F_1, F_2, \dots, F_n and a cost function $c(\cdot, \cdot)$, computes a sequence of basic operations that optimizes total cost of concatenating the n input files.

The sub structure of this problem is the time to merge a set of files S . $\text{time}(S)$ – the time needed to merge the files and the files in S should be kept for repeated use

```
if size (S) == 2:  
    # S1 and S2 are the two files in S  
    time (S) = c (length(S1), length(S2))  
else:  
    # f is a file in S  
    # S-f : take f out from S  
    time (S) = min (c (length(f), total length of all other files in S)+time (S-f))
```

Return $\text{time}(S)$ where S contains all the files

4) 16 pts

Define the language

Double-SAT = { ψ : ψ is a Boolean formula with at least 2 distinct satisfying assignments }.

For instance, the formula $\psi: (x \vee y \vee z) \wedge (x' \vee y' \vee z') \wedge (x' \vee y' \vee z)$ is in Double-SAT, since the assignments $(x = 1, y = 0, z = 0)$ and $(x = 0, y = 1, z = 1)$ are two distinct assignments that both satisfy ψ .

Prove that Double-SAT is NP-Complete.

Various methods can be used for reducing SAT to Double-SAT. Since SAT is NP-Complete, Double-SAT is NP complete.

Following is an example for the reduction.

On input $\psi(x_1, \dots, x_n)$:

1. Introduce a new variable y .

2. Output formula $\psi'(x_1, \dots, x_n, y) = \psi(x_1, \dots, x_n) \wedge (y \mid \text{not } y)$.

If $\psi(x_1, \dots, x_n)$ belongs to SAT, then ψ' has at least 1 satisfying assignment, and therefore $\psi'(x_1, \dots, x_n, y)$ has at least 2 satisfying assignments as we can satisfy the new clause $(y \mid \text{not } y)$ by assigning either

$y = 1$ or $y = 0$ to the new variable y , so $\psi'(x_1, \dots, x_n, y)$ belongs to Double-SAT. On the other hand, if $\psi(x_1, \dots, x_n)$ does not belong to SAT, then clearly $\psi'(x_1, \dots, x_n, y) = \psi(x_1, \dots, x_n) \wedge (y \mid \text{not } y)$ has no satisfying assignment either, so $\psi'(x_1, \dots, x_n, y)$ does not belong to Double-SAT. Therefore, SAT can be reduced to Double-SAT. Since the above reduction clearly can be done in P time, Double-SAT is NP-Complete.

5) 16 pts

Let X be a set of n intervals on the real line. A subset of intervals $Y \subset X$ is called a tiling path if the intervals in Y cover the intervals in X , that is, any real value that is contained in some interval in X is also contained in some interval in Y . The size of a tiling cover is just the number of intervals.

Describe and analyze an algorithm to compute the smallest tiling path of X as quickly as possible. Assume that your input consists of two arrays $X_L[1 .. n]$ and $X_R [1 .. n]$, representing the left and right endpoints of the intervals in X .



A set of intervals. The seven shaded intervals form a tiling path

Sort the intervals from left to right by the start position, if the start positions are same, then sort it from the left to right by the end position;

```

FOR (i=1;i<=n; i++) do
    Result[i] = positive unlimited ;
    FOR (j = 1;j < i; j ++) do
        IF (interval i overlap with interval j) then
            IF (Result[i]> result[j]+1) then
                Result[i] = result[j]+1;
Return result [n]

```

The complexity is $O(n^2)$

6) 16 pts

An edge of a flow network is called *critical* if decreasing the capacity of this edge results in a decrease in the maximum flow. Give an efficient algorithm that finds a critical edge in a network.

Find a min cut of the network which has the same capacity as the maximum flow.
Every outgoing edge from the min cut is a critical edge

CS570
Analysis of Algorithms
Fall 2008
Final Exam

Name: _____
Student ID: _____

____ Monday Section ____ Wednesday Section ____ Friday Section

	Maximum	Received
Problem 1	20	
Problem 2	10	
Problem 3	10	
Problem 4	20	
Problem 5	20	
Problem 6	10	
Problem 7	10	
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]

If $NP = P$, then all problems in NP are NP hard

[FALSE]

L_1 can be reduced to L_2 in Polynomial time and L_2 is in NP, then L_1 is in NP

[FALSE]

The simplex method solves Linear Programming in polynomial time.

[FALSE]

Integer Programming is in P.

[FALSE]

If a linear time algorithm is found for the traveling salesman problem, then every problem in NP can be solved in linear time.

[TRUE]

If there exists a polynomial time 5-approximation algorithm for the general traveling salesman problem then 3-SAT can be solved in polynomial time.

[FALSE]

Consider an undirected graph $G=(V, E)$. Suppose all edge weights are different. Then the longest edge cannot be in the minimum spanning tree.

[FALSE]

Given a set of demands $D = \{d_v\}$ on a directed graph $G(V, E)$, if the total demand over V is zero, then G has a feasible circulation with respect to D .

[TRUE]

For a connected graph G , the BFS tree, DFS tree, and MST all have the same number of edges.

[FALSE]

Dynamic programming sub-problems can overlap but divide and conquer sub-problems do not overlap, therefore these techniques cannot be combined in a single algorithm.

Grading Criteria: Pretty clear, each has two point. These T/F are designed and answered by Professor Shamsian

2) 10 pts

Demonstrate that an algorithm that consists of a polynomial number of calls to a polynomial time subroutine could run in exponential time.

Suggested Solution:

Suppose X takes an input size of n and returns an output size of n^2 . You call X a polynomial number of times say n. If the size of the original input is n the size of the output will be n^{2n} which is exponential WRT n.

Grading Criteria: Rephrasing the question doesn't get that much. If you show you at least understood polynomial / exponential time, you get some credit.

3) 10 pts

Suppose that we are given a weighted, directed graph $G = (V, E)$ in which edges that leave the source vertex s may have negative weights, all other edge weights are nonnegative, and there are no negative-weight cycles. Argue that Dijkstra's algorithm correctly finds shortest paths from s in this graph.

Suggested solution :

```
DIJKSTRA( $G, w, s$ )
1 INITIALIZE-SINGLE-SOURCE( $G, s$ )
2  $S \leftarrow \emptyset$ 
3  $Q \leftarrow V[G]$ 
4 while  $Q \neq \emptyset$ 
5   do  $u \leftarrow \text{EXTRACT-MIN}(Q)$ 
6    $S \leftarrow S \cup \{u\}$ 
7   for each vertex  $v \in \text{Adj}[u]$ 
8     do RELAX( $u, v, w$ )
  
RELAX( $u, v, w$ )
1 if  $d[v] > d[u] + w(u, v)$ 
2 then  $d[v] \leftarrow d[u] + w(u, v)$ 
3  $\pi[v] \leftarrow u$ 
```

We have the algorithm as shown above.

We show that in each iteration of Dijkstra's algorithm, $d[u]=\delta(s,u)$ when u is added to S (the rest follows from the upper-bound property). Let $N^-(s)$ be the set of vertices leaving s , which have negative weights. We divide the proof to vertices in $N^-(s)$ and all the rest. Since there are no negative loops, the shortest path between s and all $u \in N^-(s)$, is through the edge connecting them to s , hence $\delta(s,u)=w(s,u)$, and it follows that after the first time the loop in lines 7,8 is executed $d[u]= w(s,u)=\delta(s,u)$ for all $u \in N^-(s)$ and by the upper bound property $d[u]=\delta(s,u)$ when u is added to S . Moreover it follows that $S= N^-(s)$ after $|N^-(s)|$ steps of the while loop in lines 4-8.

For the rest of the vertices we argue that the proof of Theorem 24-6 (*Theorem 24.6 - Correctness of Dijkstra's algorithm, Introduction to Algorithem by Cormen*) holds since every

shortest path from s includes at most one negative edge (and if does it has to be the first one). To see why this is true assume otherwise, which mean that the path contains a loop, contradicting the property that shortest paths do not contain loops.

Grading Criteria : You need to argue, so bringing just one example shouldn't get you the whole credit , but it may did! If you showed you at least understood Dijkstra, you got some credit too.

4) 20 pts

Phantasy Airlines operates a cargo plane that can hold up to 30000 pounds of cargo occupying up to 20000 cubic feet. We have contracted to transport the following items.

<u>Item type</u>	<u>Weight</u>	<u>Volume</u>	<u>Number</u>	<u>Cost if not carried</u>
1	4000	1000	3	\$800
2	800	1200	10	\$150
3	2000	2200	4	\$300
4	1500	500	5	\$500

For example, we have contracted 10 items of type 2, each of which weighs 800 pounds and takes up to 1200 cubic feet of space. The last column refers to the cost of subcontracting shipment to another carrier.

For each pound we carry, the cost of flying the plane increases by 5 cents. Which items should we put in the plane, and which should we ship via other carrier, in order to have the lowest shipping cost? Formulate this problem as an integer programming problem.

Suggested Solution:

Assume the data in the table are per item.

Considering x_1, x_2, x_3, x_4 are the items that we will ship for types 1,2,3 and 4 respectively.

It is clear that :

$$0 \leq x_1 \leq 3 \quad \& \quad 0 \leq x_2 \leq 10 \quad \& \quad 0 \leq x_3 \leq 4 \quad \& \quad 0 \leq x_4 \leq 5$$

Weight constraint :

$$x_1 * 4000 + x_2 * 800 + x_3 * 2000 + x_4 * 1500 \leq 30,000$$

Volume constraint:

$$x_1 * 1000 + x_2 * 1200 + x_3 * 2200 + x_4 * 500 \leq 20,000$$

Cost of shipping :

$$C_{Ship} = C_A + (x_1 * 4000 + x_2 * 800 + x_3 * 2000 + x_4 * 1500) * \$0.05$$

Where C_A is the cost of operating empty cargo plane

Cost of sub-contracting

$$C_{Sub} = (3 - x_1) * 800 + (10 - x_2) * 150 + (4 - x_3) * 300 + (5 - x_4) * 500$$

Out objective is to minimize $C_{Ship} + C_{Sub}$

Grading Criteria: Some credit will be deducted if you missed some optimization part. Making this simple question complicated may result deduction.

5) 20 pts

Suppose you are given a set of n integers each in the range $0 \dots K$. Give an efficient algorithm to partition these integers into two subsets such that the difference $|S_1 - S_2|$ is minimized, where S_1 and S_2 denote the sums of the elements in each of the two subsets.

Proposed Solution :

$a[1] \dots a[m]$ = the set of integers

$\text{opt}[i,k]$ = true if and only if it is possible to sum to k using elements $1 \dots i$

```
Initialize opt(i,k) = false for all i,k
for(i=1..n) {
    for(k=1..K) {
        if(opt(i,k) == true) {
            for(j=1..m) {
                opt(i,k + a[j]) = true;
            }
        }
    }
}
for(k = floor(K/2)..1) {
    if(opt(n, k)) {
        Return |K - 2k|; // Returns the difference. Partition can be found by back tracking
    }
}
```

6) 10 pts

Adrian has many, many love interests. Many, many, many love interests. The problem is, however, a lot of these individuals know each other, and the last thing our polyamorous TA wants is fighting between his hookups. So our resourceful TA draws a graph of all of his potential partners and draws an edge between them if they know each other. He wants at least k romantic involvements and none of them must know each other (have an edge between them). Can he create his LOVE-SET in polynomial time? Prove your answer.

Proposed Solution :

This is just Independent Set. Proof of NP-completeness was shown in class lecture. We show the correctness as follows: 1) Any Independent Set of size k on the graph will satisfy the requirement that no two love interests know each other (share an edge). 2) If there is a set of k love interests none of which know each other, then there must be a corresponding set of k vertices, such that no two share an edge (know each other).

7) 10 pts

An edge in a flow network is called a bottleneck if increasing its capacity increases the max flow in the network. Give an efficient algorithm for finding all the bottlenecks in the network.

Proposed Solution

Find max flow and the corresponding residual graph, then for each edge increase its capacity by one unit and see if you find a new path from s to t in the residual graph. (Of course you undo this increase before trying the next edge.) If the flow increases for any such edge then that edge must be a bottleneck.

CS570
Analysis of Algorithms
Summer 2009
Final Exam

Name: _____
Student ID: _____

	Maximum	Received
Problem 1	10	
Problem 2	20	
Problem 3	15	
Problem 4	20	
Problem 5		
Problem 6		
Total	100	

2hr exam, closed books and notes.

1) 10 pts

For each of the following sentences, state whether the sentence is known to be TRUE, known to be FALSE, or whether its truth value is still UNKNOWN.

- (a) If a problem is in P, it must also be in NP.
TRUE.
- (b) If a problem is in NP, it must also be in P.
UNKNOWN.
- (c) If a problem is NP-complete, it must also be in NP.
TRUE.
- (d) If a problem is NP-complete, it must not be in P.
UNKNOWN.
- (e) If a problem is not in P, it must be NP-complete.
FALSE.

If a problem is NP-complete, it must also be NP-hard.

TRUE.

If a problem is in NP, it must also be NP-hard.

FALSE.

If we find an efficient algorithm to solve the Vertex Cover problem we have proven that $P=NP$

TRUE.

If we find an efficient algorithm to solve the Vertex Cover problem with an approximation factor $\rho \geq 1$ (a single constant) then we have proven that $P=NP$

FALSE.

If we find an efficient algorithm that takes as input an approximation factor $\rho \geq 1$ and solves the Vertex Cover problem with that approximation factor, we have proven that $P=NP$.

TRUE.

2) 20 pts

Let $G = (V, E)$ be a weighted, directed graph with weight function $w : E \rightarrow \{0, 1, \dots, W\}$ for some nonnegative integer W . Modify Dijkstra's algorithm to compute the shortest paths from a given source vertex s in $O(WV + E)$ time.

Consider running Dijkstra's algorithm on a graph, where the weight function is $w : E \rightarrow \{1, \dots, W - 1\}$. To solve this efficiently, implement the priority queue by an array A of length $WV + 1$. Any node with shortest path estimate d is kept in a linked list at $A[d]$. $A[WV + 1]$ contains the nodes with ∞ as estimate.

EXTRACT-MIN is implemented by searching from the previous minimum shortest path estimate until a new is found. DECREASE-KEY simply moves vertices in the array. The EXTRACT-MIN operations takes a total of $O(VW)$ and the DECREASE-KEY operations take $O(E)$ time in total. Hence the running time of the modified algorithm will be $O(VW + E)$.

3) 15 pts

At a dance, we have n men and n women. The men have height $g(1), \dots, g(n)$, and women $h(1), \dots, h(n)$. For the dance, we want to match up men with women of roughly the same height. Here are the precise rules:

- a. Each man i is matched up with exactly one woman w_i , and each woman with exactly one man.
- b. For each couple (i, w_i) , the mismatch is height difference $|g(i)-h(w_i)|$.
- c. Our goal is to find a matching minimizing the maximum mismatch,
 $\text{MAX}_i |g(i)-h(w_i)|$.

Give an algorithm that runs in $O(n \log n)$ and achieves the desired matching. Provide proof of correctness.

We use an exchange argument. Let w_i denote the optimal solution. If there is any pair i, j such that $i < j$ in our ordering, but $w_i > w_j$ (also with respect to our ordering), then we evaluate the effect of switching to $w'_i := w_j, w'_j := w_i$. The mismatch of no other couple is affected. The couple including man i now has mismatch $|g(i) - h(w'_i)| = |g(i) - h(w_j)|$. Similarly, the other switched couple now has mismatch $|g(j) - h(w_i)|$.

We first look at man i , and distinguish two cases: if $h(w_j) \leq g(i)$ (i.e., man i is at least as tall as his new partner), then the sorting implies that $|g(i) - h(w_j)| = g(i) - h(w_j) \leq g(j) - h(w_j) = |g(j) - h(w_j)|$. On the other hand, if $h(w_j) > g(i)$, then $|g(i) - h(w_j)| = h(w_j) - g(i) \leq h(w_i) - g(i) = |h(w_i) - g(i)|$. In both cases, the mismatch between man i and his new partner is at most the previous maximum mismatch.

We use a similar argument for man j . If $h(w_i) \leq g(j)$, then $|g(j) - h(w_i)| = g(j) - h(w_i) \leq g(j) - h(w_j) = |g(j) - h(w_j)|$. On the other hand, if $h(w_i) > g(j)$, then $|g(j) - h(w_i)| = h(w_i) - g(j) \leq h(w_i) - g(i) = |h(w_i) - g(i)|$. Hence, the mismatch between j and his partner is also no larger than the previous maximum mismatch.

Hence, in all cases, we have that both of the new mismatches are bounded by the larger of the original mismatches. In particular, the maximum mismatch did not increase by the swap. By making such swaps while there are inversions, we gradually transform the optimum solution into ours. This proves that the solution found by the greedy algorithm is in fact optimal.

4) 20 pts

You are given integers p_0, p_1, \dots, p_n and matrices A_1, A_2, \dots, A_n where matrix A_i has dimension $(p_{i-1}) * p_i$

(a) Let $m(i, j)$ denote the minimum number of scalar multiplications needed to evaluate the matrix product $A_i A_{i+1} \dots A_j$. Write down a recursive algorithm to compute $m(i, j)$, $1 \leq i \leq j \leq n$, that runs in $O(n^3)$ time.

Hint: You need to consider the order in which you multiply the matrices together and find the optimal order of operations.

Initialize $m[i, j] = -1 \quad 1 \leq i \leq j \leq n$

Output $mcr(1, n)$

```
mcr(i, j) {  
    if m[i, j] >= 0 return m[i, j]  
    else if i == j m[i, j] = 0  
    else m[i, j] = min (i <= k < j) { mcr(i, k) + mcr(k+1, j) + P_{i-1} * P_k * P_k }  
    return m[i, j]  
}
```

(b) Use this algorithm to compute $m(1,4)$ for $p_0=2$, $p_1=5$, $p_2=3$, $p_3=6$, $p_4=4$

$m[i, j]$

i\j	2	3	4
1	30	66	114
2		90	132
3			72

$m[1, 4] = 114$

5) 20 pts

In a certain town, there are many clubs, and every adult belongs to at least one club. The townspeople would like to simplify their social life by disbanding as many clubs as possible, but they want to make sure that afterwards everyone will still belong to at least one club.

Prove that the Redundant Clubs problem is NP-complete.

First, we must show that Redundant Clubs is in NP, but this is easy: if we are given a set of K clubs, it is straightforward to check in polynomial time whether each person is a member of another club outside this set.

Next, we reduce from a known NP-complete problem, Set Cover. We translate inputs of Set Cover to inputs of Redundant Clubs, so we need to specify how each Redundant Clubs input element

is formed from the Set Cover instance. We use the Set Cover's elements as our translated list of people,

and make a list of clubs, one for each member of the Set Cover family. The members of each club are just the elements of the corresponding family. To finish specifying the Redundant Clubs input,

we need to say what K is: we let $K = F - K_{SC}$ where F is the number of families in the Set Cover instance and K_{SC} is the value K from the set cover instance. This translation can clearly be done in polynomial time (it just involves copying some lists and a single subtraction).

Finally, we need to show that the translation preserves truth values. If we have a yes-instance of Set Cover, that is, an instance with a cover consisting of K_{SC} subsets, the other K subsets form a solution to the translated Redundant Clubs problem, because each person belongs to a club in the

cover. Conversely, if we have K redundant clubs, the remaining K_{SC} clubs form a cover. So the answer

to the Set Cover instance is yes if and only if the answer to the translated Redundant Clubs instance
is yes.

6) 15 pts

A company makes two products (X and Y) using two machines (A and B). Each unit of X that is produced requires 50 minutes processing time on machine A and 30 minutes processing time on machine B. Each unit of Y that is produced requires 24 minutes processing time on machine A and 33 minutes processing time on machine B.

At the start of the current week there are 30 units of X and 90 units of Y in stock. Available processing time on machine A is forecast to be 40 hours and on machine B is forecast to be 35 hours.

The demand for X in the current week is forecast to be 75 units and for Y is forecast to be 95 units—these demands must be met. In addition, company policy is to maximize the combined sum of the units of X and the units of Y in stock at the end of the week.

- a. Formulate the problem of deciding how much of each product to make in the current week as a linear program.

Solution

Let

- x be the number of units of X produced in the current week
- y be the number of units of Y produced in the current week

then the constraints are:

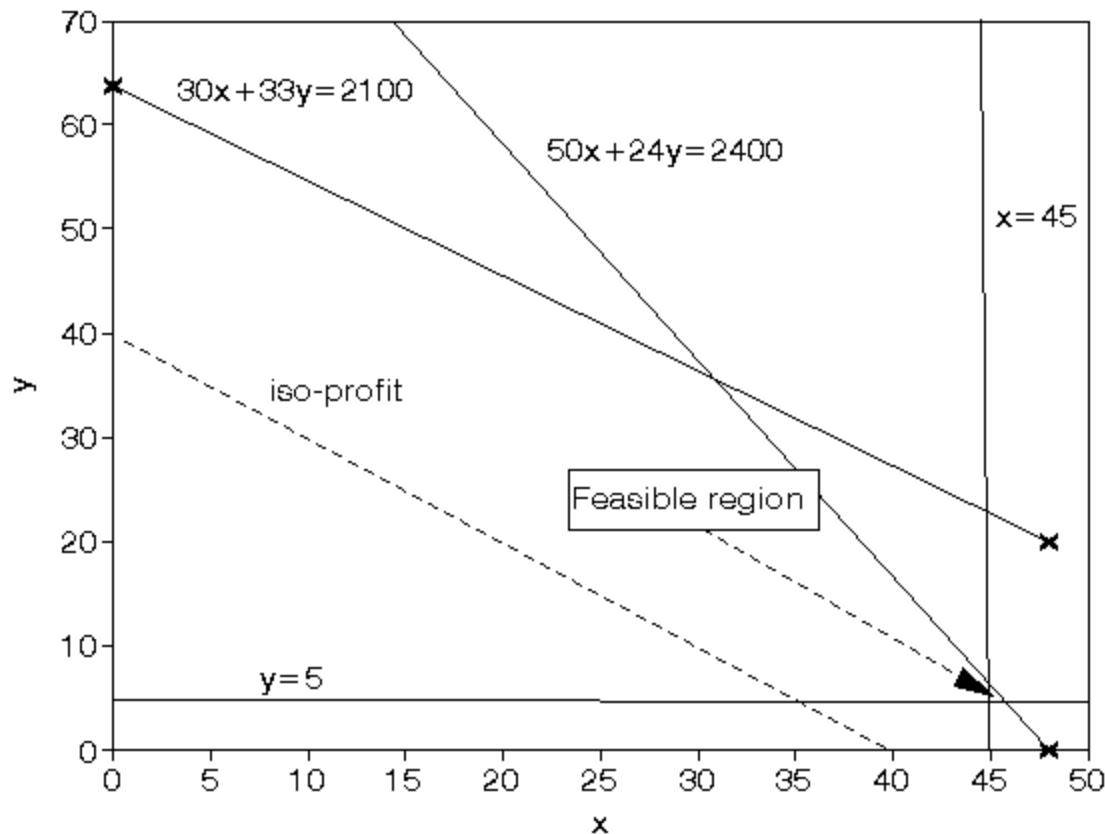
- $50x + 24y \leq 40(60)$ machine A time
- $30x + 33y \leq 35(60)$ machine B time
- $x \geq 75 - 30$
- i.e. $x \geq 45$ so production of X \geq demand (75) - initial stock (30), which ensures we meet demand
- $y \geq 95 - 90$
- i.e. $y \geq 5$ so production of Y \geq demand (95) - initial stock (90), which ensures we meet demand

The objective is: maximise $(x+30-75) + (y+90-95) = (x+y-50)$

i.e. to maximise the number of units left in stock at the end of the week

b. Solve this linear program graphically.

It can be seen in diagram below that the maximum occurs at the intersection of $x=45$ and $50x + 24y = 2400$



Solving simultaneously, rather than by reading values off the graph, we have that $x=45$ and $y=6.25$ with the value of the objective function being 1.25

CS570
Analysis of Algorithms
Fall 2014
Exam III

Name: _____
Student ID: _____
Email: _____

Wednesday Evening Section

	Maximum	Received
Problem 1	20	
Problem 2	16	
Problem 3	16	
Problem 4	16	
Problem 5	16	
Problem 6	16	
Total	100	

Instructions:

1. This is a 2-hr exam. Closed book and notes
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.

1) 20 pts

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

[/FALSE]

All the NP-hard problems are in NP.

[/FALSE]

Given a weighted graph and two nodes, it is possible to list all shortest paths between these two nodes in polynomial time.

[TRUE/]

In the memory efficient implementation of Bellman-Ford, the number of iterations it takes to converge can vary depending on the order of nodes updated within an iteration

[/FALSE]

There is a feasible circulation with demands $\{d_v\}$ if $\sum_v d_v = 0$.

[/FALSE]

Not every decision problem in P has a polynomial time certifier.

[TRUE/]

If a problem can be reduced to linear programming in polynomial time then that problem is in P.

[/FALSE]

If we can prove that $P \neq NP$, then a problem $A \in P$ does not belong to NP.

[/FALSE]

If all capacities in a flow network are integers, then every maximum flow in the network is such that flow value on each edge is an integer.

[/FALSE]

In a dynamic programming formulation, the sub-problems must be mutually independent.

[~~TRUE~~/]

In the final residual graph constructed during the execution of the Ford–Fulkerson Algorithm, there's no path from sink to source.

2) 16 pts

In the Bipartite Directed Hamiltonian Cycle problem, we are given a bipartite directed graph $G = (V; E)$ and asked whether there is a simple cycle which visits every node exactly once. Note that this problem might potentially be easier than Directed Hamiltonian Cycle because it assumes a bipartite graph. Prove that Bipartite Directed Hamiltonian Cycle is in fact still NP-Complete.

Solution:

- i) This problem is NP. Given a candidate path, we just check the nodes along the given path one by one to see if every node in the network is visited exactly once and if each consecutive node pair along the path are joined by an edge.
- ii) To prove the problem is NP-complete, we reduce Directed Hamiltonian Cycle (DHC) problem to Bipartite Directed Hamiltonian Cycle (BDHC) problem.

Given an arbitrary directed graph G , we split each vertex v in G into two vertex v_{in} and v_{out} . Here v_{in} connects all the incoming edges to v in G ; v_{out} connects all the outgoing edges from v in G . Moreover, we connect one directed edge from v_{in} to v_{out} . After doing these operations for each node in G , we form a new graph G' .

Here G' is bipartite graph, because we can color each v_{in} “blue” and each v_{out} “red” without any coloring conflict.

If there is DHC in G , then there is a BDHC in G' . We can replace each node v on the DHC in G into consecutive nodes v_{in} and v_{out} , and (v_{in}, v_{out}) is an edge in G' . Then the new path is a BDHC in G' .

On the other hand, if there is BDHC in G' , then there is a DHC in G . Note that if v_{in} is on the BDHC, v_{out} must be the successive node on the BDHC. Then we can merge each node pair (v_{in}, v_{out}) on the BDHC in G' into node v and form a DHC in G .

In sum, G has DHC if and only if G' has a BDHC. This completes the reduction, and we confirm that the given problem is NP-complete

3) 16 pts

A tourism company is providing boat tours on a river with n consecutive segments. According to previous experience, the profit they can make by providing boat tours on segment i is known as a_i . Here a_i could be positive (they earn money), negative (they lose money), or zero. Because of the administration convenience, the local community of the river requires that the tourism company should do their boat tour business on a contiguous sequence of the river segments, i.e, if the company chooses segment i as the starting segment and segment j as the ending segment, all the segments in between should also be covered by the tour service, no matter whether the company will earn or lose money. The company's goal is to determine the starting segment and ending segment of boat tours along the river, such that their total profit can be maximized. Design an efficient algorithm to achieve this goal, and analyze its run time (Note that brute-force algorithm achieves $\Theta(n^2)$, so your algorithm must do better.)

Solution1:

Using dynamic programming:

Define $\text{OPT}(i)$ as the maximum total profit the company can get when running the boat tours on a contiguous sequence of segments starting at segment i .

Base case: $\text{OPT}(n) = a_n$.

Recursive relation: $\text{OPT}(i) = \max\{\text{OPT}(i+1)+a_i, a_i\}$, for $1 \leq i < n$

Algorithm:

For $i = n, \dots, 1$
 Compute $\text{OPT}(i)$.
End

Maximum profit = $\max_{i=1}^n \{\text{OPT}(i)\}$. Denote i^* as the starting index which achieves the maximum profit, then i^* is the optimal starting segment

For $i = i^*, \dots, n$
 If ($\text{OPT}(i) = a_i$)
 Set $j^* = i$;
 Break;
 End
End
The value j^* is the index of the optimal ending segment.

Complexity: Computing all the OPT values takes time $O(n)$, comparing all OPT values and find the maximum profit and the corresponding starting segment takes time $O(n)$. Finding the optimal ending segment takes time $O(n)$. In sum, the algorithm takes time $O(n)$

Remark: in this solution, we implicitly assume that the company must provide the boat tour, while providing no boat tour is not a choice. If you consider providing no boat tour also as a choice, it is also treated as a correct solution, however, the step of computing the maximum profit above solution should be modified as follows:

$$\text{Maximum profit} = \max\{0, \max_{\{i\}} \{\text{OPT}(i)\}\}.$$

Correspondingly, if 0 is the best result you can get, you need to claim that providing no boat tour is the best solution, and there is no need to confirm the starting segment or ending segment.

Solution 2 (outline):

Using divide and conquer.

- i) Divide operation: Divide the profit sequences of the n consecutive segments, denoted as $A[1,n]$, as two parts:
 $a_1, \dots, a_{n/2}$ and $a_{n/2+1}, \dots, a_n$, denoted as $A[1,n/2]$ and $A[n/2+1, n]$ respectively.
- ii) Merge operation:
 Suppose you successfully find the maximum profit in $A[1,n/2]$ and $A[n/2+1, n]$, denoted as $P_{left}(1,n)$, $P_{right}(1,n)$, and the corresponding contiguous sequence of segments, denoted as $B_{left}(1,n)$ and $B_{right}(1,n)$
 Then the optimal contiguous segment can be in one of the three cases:
 - a) $B_{left}(1,n)$
 - b) $B_{right}(1,n)$
 - c) An optimal contiguous segment sequence crossing $A[1,n/2]$ and $A[n/2+1, n]$, denoted as $B_{cross}(1,n)$.

For $B_{cross}(1,n)$, denote the corresponding profit as $P_{cross}(1,n)$. The method to confirm $B_{cross}(1,n)$ and $P_{cross}(1,n)$ is as follows:

- Starting from sequence $[a_{n/2}, a_{n/2+1}]$, compute the summation $S_{left}(1) = a_{n/2} + a_{n/2+1}$ as one candidate solution for $P_{cross}(1,n)$.
- Next, including $a_{n/2-1}$ into the above sequence as $[a_{n/2-1}, a_{n/2}, a_{n/2+1}]$, compute the summation of the three elements in the sequence as the second candidate solution: $S_{left}(2) = S_{left}(1) + a_{n/2-1}$.

- Keep including the element one by one to the left until a_1 is included, during each step, compute and record the summation values.
- Find $S_{\text{opt_left}} = \max_i \{S_{\text{left}}(i)\}$, record the corresponding index of the included element that achieves the maximum, say i_{cross^*} . Then i_{cross^*} is the optimal starting index for $B_{\text{cross}}(1,n)$;
- Starting from $[a_{i_{\text{cross}^*}}, \dots, a_{n/2+1}]$ includes the element to the right one by one until a_n is included, during each step, compute the summation values in the same way as in the left part, denote the value as $S_{\text{right}}(i)$.
- Find $P_{\text{cross}}(1,n) = \max_i \{S_{\text{right}}(i)\}$, record the corresponding index of the included element that achieves the maximum, say j_{cross^*} . Then j_{cross^*} is the optimal ending index for $B_{\text{cross}}(1,n)$;

Then

Maximum Profit = $\max \{P_{\text{left}}(1,n), P_{\text{right}}(1,n), P_{\text{cross}}(1,n)\}$, and the corresponding optimal starting index and ending index are the ones that achieve the maximum.

- Before doing step ii) for $A[1,n]$, recursively run the above procedure in each array $A[1,n/2]$ and $A[n/2+1, n]$ and so on.
- Termination condition: for $A[i,j]$, if $i=j$, return a_i as the maximum profit, return i as both the optimal starting and ending index in $A[i,j]$.

Complexity:

The Divide operation takes time $O(1)$.

The Merge operation takes time $O(n)$.

Define $T(n)$ as the running time,

$$T(n) = 2*T(n/2) + O(n)$$

The complexity is $O(n \log(n))$.

Remark: similar as in solution 1, considering no bout tour as a choice is also treated as a correct solution.

4) 16 pts

Consider the following matching problem. There are m students s_1, s_2, \dots, s_m and a set of n companies $C = \{c_1, c_2, \dots, c_n\}$. Each student can work for only one company, whereas company c_j can hire up to b_j students. Student s_i has a preferred set of companies $\Lambda_i \subseteq C$ at which he/she is willing to work. Your task is to find an assignment of students to companies such that all of the above constraints are satisfied and each student is assigned. Formulate this as a network flow problem and describe any subsequent steps necessary to arrive at the solution. Prove correctness.

Construction of flow network: Represent each student and each company as a separate node. Add one source node s and a sink node t for a total of $m + n + 2$ nodes. Add the following directed edges with capacities:

- i. $s \rightarrow s_i$ with capacity 1 unit, for each $1 \leq i \leq m$,
- ii. For each $1 \leq i \leq m$, $s_i \rightarrow c$ with capacity 1 unit for all nodes $c \in \Lambda_i$.
- iii. $c_j \rightarrow t$ with capacity b_j units, for each $1 \leq j \leq n$.

Constructing the solution: Run any max-flow algorithm on the above network to get the realization of edge flows under a max flow configuration (lets call it O for brevity). If an edge $s_i \rightarrow c_j$ shows a non-zero flow in this configuration, assign student s_i to company c_j . Repeat this process for each edge between the student nodes and the company nodes to get the final assignment.

Proof of correctness: We have to show that the solution so obtained satisfies all constraints of the problem.

- i. Since all outgoing edges from s_i are to the node set Λ_i , it is impossible for s_i to have a flow to a node outside Λ_i in configuration O .
- ii. Since the only outgoing edge from c_j is of capacity b_j , configuration O cannot have more than b_j incoming edges of non-zero flow to node c_j . Thus, not more than b_j students can get assigned to company c_j .
- iii. As s_i has a single incoming edge and multiple outgoing edges of capacity 1, configuration O cannot have more than one outgoing edge from s_i with non-zero flow. Hence, s_i can get assigned to at most one company.

We have proved that our mapping from configuration O to an assignment does not violate any of the constraints except possibly that all students might not be assigned. Note that this may happen in practice if it is impossible to assign all students while satisfying all of the given constraints. Thus, what we need to show is that if there exists a feasible assignment then configuration O will have each $s \rightarrow s_i$ edge carry a non-zero flow. Given a feasible assignment $\{(s_i, c_{\sigma(i)}), 1 \leq i \leq m\}$, by construction of the flow network, it is possible to set each edge $s_i \rightarrow c_{\sigma(i)}$ to carry 1 unit of flow. By feasibility of the assignment, company c_j gets no more than b_j incoming edges with non-zero flow, so the outgoing edge from c_j has enough capacity to carry away all incident flow on node c_j . Finally, since each s_i has an outgoing flow of 1 unit in

this assignment, all $s \rightarrow s_i$ edges can be set to have 1 unit of flow, completing the proof.

5) 16 pts

Consider a directed, weighted graph G where all edge weights are positive. You have one Star, which lets you change the weight of any one edge to zero. In other words, you may change the weight of any one edge to zero. Give an efficient algorithm using Dijkstra's algorithm to find a lowest-cost path between two vertices s and t , given that you may set one edge weight to zero. Note: you will receive 10 pts if your algorithm is efficient. You will receive full points (16 pts) if your algorithm has the same run time complexity as Dijkstra's algorithm.

Solution:

Use Dijkstra's algorithm to find the shortest paths from s to all other vertices. Reverse all edges and use Dijkstra to find the shortest paths from all vertices to t . Denote the shortest path from u to v by $u \rightsquigarrow v$, and its length by $\delta(u, v)$.

Now, try setting each edge to zero. For each edge $(u, v) \in E$, consider the path $s \rightsquigarrow u \rightarrow v \rightsquigarrow t$. If we set $w(u, v)$ to zero, the path length is $\delta(s, u) + \delta(v, t)$. Find the edge for which this length is minimized and set it to zero; the corresponding path $s \rightsquigarrow u \rightarrow v \rightsquigarrow t$ is the desired path. The algorithm requires two invocations of Dijkstra, and an additional $\Theta(E)$ time to iterate through the edges and find the optimal edge to take for free. Thus the total running time is the same as that of Dijkstra:

$O(E + V \lg V)$: based on using Fibonacci heap

$O(|V| + |E|) \log |V|$): based on using binary heap

6) 16 pts

You are given n rods; they are of length l_1, l_2, \dots, l_n , respectively. Our goal is to connect all the rods and form a single rod. The length after connecting two rods and the cost of connecting them are both equal to the sum of their lengths. Give an algorithm to minimize the cost of connecting them to form a single rod. State the complexity of your algorithm and prove that your algorithm is optimal.

1st interpretation: At each step we connect a single rod to the set of rods that are already connected together.

Of course, the solution is to sort rods by length and connect them in the order of increasing length. To prove this is optimal, you can assume that there is an optimal solution and compare our solution to the optimal solution: At each step our solution stays ahead of (or at least does no worse than) the optimal solution.

Sort Takes $O(n \log n)$

We can use mathematical induction to prove that we always stay ahead of the optimal solution.

Claim: at each step the cost of connecting rods in our solution is less than or equal to that of the other solution and the length of the rod after this connection is smaller than or equal in size to that of the other solution.

Base case: first two shortest rods – obviously this is the opt solution

Assuming that the cost of connecting k rods in our solution is less than or equal to that of another (optimal) solution, we can show that the cost of connecting the next ($k+1^{\text{st}}$) rod will be less than or equal to the cost of connecting the $k+1^{\text{st}}$ rod in the other solution:

Cost of the last connection in our solution = total length of all rods 1 to $k+1$ (ordered by length)

Cost of the last connection in the other solution = total length of all rods 1 to $k+1$ (not necessarily ordered by length)

It is obvious that the cost of our last connection is smaller or equal to the cost of the other connection because the only way to get the minimum total length of $k+1$ rods is to pick the shortest $k+1$ rods.

2nd interpretation: At each step we can connect any rod or set of already connected rods to another rod or set of already connected rods.

The solution is to keep connecting the smallest two rods or sets of already connected rods.

Implementation: Place all rods in a min heap with their key values representing their length. At each step extract two elements from the set and insert the combined rod back into the min heap.

This takes a total of $O(n \log n)$ time

Fact 1: the cost contribution of a rod i to the total assembly cost is $\text{Length}(i) * \text{Level}(i)$, where $\text{Level}(i)$ is the level at which the rod is first assembled with other rods (level 1=root level, level 2= level below root, etc.).

Proof: By observation of cost of assembly tree

Fact2: There is an optimal assembly tree of rods in which the two smallest rods are leaf nodes at the lowest level of the tree and the children of the same parent.

Proof: let's say the two smallest rods are not leaf nodes at the lowest level of the tree, using fact 1, we can swap these rods with two rods at the lowest level of the tree and thereby reducing the total cost of the assembly tree. If the two rods are leaf nodes at the lowest level but not children of the same parent we can swap two rods to make these rods children of the same parent without changing the total cost of the tree (again based on Fact 1).

Assume that there is an optimal assembly tree T^* and our solution produces tree T . We will show that our tree T is also optimal.

To do this, we apply fact 2 to T^* and move the smallest rods to the lowest level of the tree as children of the same parent—without increasing the cost of T^* . We then eliminate the two smallest rods at the bottom of the two trees(since these are the first two rods that are assembled in our algorithm) and assume that there is a new combined rod in place of their parent node. We will get two new trees T^{**} and T' , where

$$\begin{aligned}\text{Total assembly cost of } T^* &= \text{Total assembly cost of } T^{**} + \text{total length of the two smallest rods} \\ \text{Total assembly cost of } T &= \text{Total assembly cost of } T' + \text{total length of the two smallest rods}\end{aligned}$$

Since the costs of the two new trees are reduced by the same amount we can now compare the cost of our new tree T' with the cost of the new optimal tree T^{**} . And show that assembly tree T' is also optimal (as compared to the optimal tree T^{**}).

To do this, we apply fact 2 recursively and place the two smallest rods in T^{**} at the lowest level of the tree and under the same parent node without increasing the cost of T^{**} . These are the next two rods that are combined in our algorithm. We then eliminate these two rods in both trees T' and T^{**} , etc.

By repeating the same steps (applying fact 2 and eliminating the two smallest rods from the optimal assembly tree and our tree) recursively, we will find an optimal solution that follows the same exact assembly sequence that is found in our algorithm. Therefore we can state that our algorithm also produces an optimal assembly tree.

Additional Space

CS570
Analysis of Algorithms
Fall 2015
Exam III

Name: _____

Student ID: _____

Email Address: _____

_____ Check if DEN Student

	Maximum	Received
Problem 1	20	
Problem 2	15	
Problem 3	12	
Problem 4	15	
Problem 5	15	
Problem 6	12	
Problem 7	11	
Total	100	

Instructions:

1. This is a 2-hr exam. Closed book and notes
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.

1) 20 pts

Mark the following statements as **TRUE**, **FALSE**. No need to provide any justification.

[/FALSE]

If P = NP, then all NP-Hard problems can be solved in Polynomial time.

[/FALSE]

Dynamic Programming approach only works when used on problems with non-overlapping sub problems.

[/FALSE]

In a divide & conquer algorithm, the size of each sub-problem must be at most half the size of the original problem.

[/FALSE]

In a 0-1 knapsack problem, a solution that uses up all of the capacity of the knapsack will be optimal.

[/FALSE]

If a problem X can be reduced to a known NP-hard problem, then X must be NP-hard.

[TRUE/]

If $\text{SAT} \leq_P A$, then A is NP-hard.

[TRUE/]

The recurrence $T(n) = 2T(n/2) + 3n$, has solution $T(n) = \theta(n \log(n^2))$.

[/FALSE]

Consider two positively weighted graphs $G_1 = (V, E, w_1)$ and $G_2 = (V, E, w_2)$ with the same vertices V and edges E such that, for any edge $e \in E$, we have $w_2(e) = (w_1(e))^2$. For any two vertices $u, v \in V$, any shortest path between u and v in G_2 is also a shortest path in G_1 .

[/FALSE]

If an undirected graph $G=(V,E)$ has a Hamiltonian Cycle, then any DFS tree in G has a depth $|V| - 1$.

[TRUE/]

Linear programming is at least as hard as the Max Flow problem.

2) 15 pts

A company makes three models of desks, an executive model, an office model and a student model. Building each desk takes time in the cabinet shop, the finishing shop and the crating shop as shown in the table below:

Type of desk	Cabinet shop	Finishing shop	Crating shop	Profit
Executive	2	1	1	150
Office	1	2	1	125
Student	1	1	.5	50
Available hours	16	16	10	

How many of each type should they make to maximize profit? Use linear programming to formulate your solution. Assume that real numbers are acceptable in your solution.

Solution:

Start by defining your variables:

x = number of executive desks made

y = number of office desks made

z = number of student desks made

Maximize $P=150x+125y+50z$.

Subject to:

$2x + y + z \leq 16$ cabinet hours

$x + 2y + z \leq 16$ finishing hours

$x + y + .5z \leq 10$ crating hours

$x \geq 0, y \geq 0, z \geq 0$

3) 12 pts

Given a graph $G=(V, E)$ and a positive integer $k < |V|$. The longest-simple-cycle problem is the problem of determining whether a simple cycle (no repeated vertices) of length k exists in a graph. Show that this problem is NP-complete.

Solution:

Clearly this problem is in NP. The certificate will be a cycle of the graph, and the certifier will check whether the certificate is really a cycle of length k of the given graph.

We will reduce HAM-CYCLE to this problem. Given an instance of HAM-CYCLE with graph $G=(V, E)$, construct a new graph $G'=(V', E)$ by adding one isolated vertex u to G . Now ask the longest-simple-cycle problem with $k = |V| < |V'|$ for graph G' .

If there is a HAM-CYCLE in G , it will be a cycle of length $k = |V|$ in G' .

If there is no HAM-CYCLE in G' , there must not be no cycle of length $|V|$ in G' . If there is, we know the cycle does not contain u , because it is isolated. So the cycle will contain all vertex in $|V|$, which is a HAM-CYCLE of the G .

The reduction is in polynomial time, so longest-simple-path is NP-Complete.

4) 15 pts

Suppose there are n steps, and one can climb either 1, 2, or 3 steps at a time. Determine how many different ways one can climb the n steps. E.g. if there are 5 steps, these are some possible ways to climb them: (1,1,1,1,1), (1,2,1,1), (3, 2), (2,3), etc. Your algorithm should run in linear time with respect to n . You need to include your complexity analysis.

Solution:

Let $T(n)$ denote the number of different ways for climbing up n stairs.

There are three choices for each first step: either 1, 2, or 3. $T(n) = T(n-1) + T(n-2) + T(n-3)$

The boundary conditions :

when there is only one step, $T(1) = 1$. If only two steps, $T(2) = 2$. $T(3)= 4$. ((1,1,1), (1,2), (2,1), (3))

Pseudo code as below:

```
if n is 1  
    return 1  
else if n is 2  
    return 2  
else  
    T[1] = 1  
    T[2] = 2  
    T[3]=4  
    for i = 4 to n do  
        T[i] = T[i-1]+T[i-2]+ T[i-3]  
    return T[n]
```

The time complexity is $\Theta(n)$.

5) 15 pts

We'd like to select frequencies for FM radio stations so that no two are too close in frequency (creating interference). Suppose there are n candidate frequencies $\{f_1, \dots, f_n\}$. Our goal is to pick as many frequencies as possible such that no two selected frequencies f_i, f_j have $|f_i - f_j| < e$ (for a given input variable e). Design a greedy algorithm to solve the problem. Prove the optimality of the algorithm and analyze the running time.

Proof:

I claim that there exists some optimal solution that makes the same first choice as (in terms of which selected frequency has the lowest value) that I do. Consider any optimal solution OPT .

Let p be my first choice and q be OPT 's first choice. If $p = q$, our proof is complete. If it isn't, we know that $p < q$; now consider some other set $OPT1 = OPT - \{q\} + \{p\}$; that is, OPT with its first choice replaced by mine. We know this is a valid set (meaning no frequencies are within e of one another), because $p < q$, and p is the first element in $OPT1$. Because nothing was within e of q , none can be within e of p . We also know that $OPT1$ is an optimal (maximum size) set, because it is the same size as OPT . Therefore, $OPT1$ is an optimal set that includes my first choice. (and therefore, by induction, the second choice will be correct, etc.)

6) 12 pts

Let S be an NP-complete problem, and Q and R be two problems whose classification is unknown (i.e. we don't know whether they are in NP, or NP-hard, etc.). We do know that Q is polynomial time reducible to S and S is polynomial time reducible to R. Mark the following statements True or False **based only on the given information, and explain why.**

(i) Q is NP-complete

False. Because $Q \leq_p S$, Q is at most as hard as S. Because S is in NPC, Q is not necessary to be in NPC, e.g., it could be in P, or could be in NP but not in NPC.

(ii) Q is NP-hard

False. Because $Q \leq_p S$, Q is at most as hard as S. Then Q is not necessary to be in NP-hard, e.g., it could be in P.

(iii) R is NP-complete

False. Because $S \leq_p R$, R is at least as hard as S. Then R is not necessary to be NPC. It is possible to be in NP-hard but not in NPC.

(iv) R is NP-hard

True. Because $S \leq_p R$, R is at least as hard as S. Then R is NP-hard.

7) 11 pts

Consider there are n students and n rooms. A student can only be assigned to one room. Each room has capacity to hold either one or two students. Each student has a subset of rooms as their possible choice. We also need to make sure that there is at least one student assigned to each room.

Give a polynomial time algorithm that determines whether a feasible assignment of students to rooms is possible that meets all of the above constraints. If there is a feasible assignment, describe how your solution can identify which student is assigned to which room.

Solution:

Construct a flow network as follows,

For each student i create a vertex a_i , for each room j create a vertex b_j , if room j is one of student i 's possible choices, add an edge from a_i to b_j with upper bound 1. Create a super source s , connect s to all a_i with an edge of lower bound and upper bound 1.

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

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

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 a_i and b_j , check whether the flow from a_i to b_j is 1, if yes, assign student i to room j .

Additional Space

Additional Space

CS570
Analysis of Algorithms
Fall 2014
Exam III

Name: _____
Student ID: _____
Email: _____

Wednesday Evening Section

	Maximum	Received
Problem 1	20	
Problem 2	16	
Problem 3	16	
Problem 4	16	
Problem 5	16	
Problem 6	16	
Total	100	

Instructions:

1. This is a 2-hr exam. Closed book and notes
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.

1) 20 pts

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

[/FALSE]

All integer programming problems can be solved in polynomial time.

[TRUE]

Fractional knapsack problem has a polynomial time greedy solution.

[/FALSE]

For any cycle in a graph, the cheapest edge in the cycle is in a minimum spanning tree.

[/FALSE]

Every decision problem is in NP.

[TRUE/]

If P=NP then P=NP=NP-complete.

[/FALSE]

Ford-Fulkerson algorithm can always terminate if the capacities are real numbers.

[/FALSE]

A flow network with unique edge capacities has a unique min cut.

[/FALSE]

A graph with non-unique edge weights will have at least two minimum spanning trees.

[TRUE/]

A sequence of $O(n)$ priority queue operations can be used to sort a set of n numbers.

[/FALSE]

If a problem X is polynomial time reducible to an NP-complete problem Y, then X is NP-complete.

2) 16 pts

Consider the **complete** weighted graph $G = (V, E)$ with the following properties

- a. The vertices are points on the X-Y plane on a regular $n \times n$ grid, i.e. the set of vertices is given by $V = \{(p, q) | 1 \leq p, q \leq n\}$, where p and q are integer numbers.
- b. The edge weights are given by the usual Euclidean distance, i.e. the weight of the edge between the nodes (i, j) and (k, l) is $\sqrt{(k - i)^2 + (l - j)^2}$.

Prove or disprove: There exists a minimum spanning tree of G such that every node has degree at most two.

Solution:

There does exist an MST of G with every node having degree ≤ 2 . One such MST is obtained by the edges joining **nodes** in the following order: $(1,1) \rightarrow (1,2) \rightarrow (1,3) \rightarrow \dots \rightarrow (1,n-1) \rightarrow (1,n) \rightarrow (2,n) \rightarrow (2,n-1) \rightarrow \dots \rightarrow (2,2) \rightarrow (2,1) \rightarrow (3,1) \rightarrow \dots \rightarrow (n,n)$.

Let us denote the above set of edges by E . To prove that E indeed forms an MST, we need to show that it forms a spanning tree and that it is of minimal weight.

E forms a spanning tree: It is clear that all nodes in the above sequence are distinct. This implies that there are no cycles since any cycle, when written out as a sequence of connected nodes, would necessarily repeat the starting node at the end. It is also clear that all nodes of the graph are covered in the above sequence. In particular, nodes $(i, 1)$, $(i, 2)$, ..., (i, n) are connected in that order for odd integers i and in the reverse order for even integers i . Hence, E forms a spanning tree and has $n^2 - 1$ edges.

E has minimal weight: It is easy to see that every edge in G has at least unit weight, since weight of edge between distinct nodes is minimal for edges between node pairs of the form $\{(i, j), (i+1, j)\}$ or $\{(i, j), (i, j+1)\}$, evaluating to an edge weight of 1. Since all edges in E are of this form, all edges in E have unit weight and total weight of E is equal to $n^2 - 1$. Finally, any spanning tree of G has exactly $n^2 - 1$ edges, so the weight of the MST must be at least $n^2 - 1$, thus proving that weight of E is minimal.

3) 16 pts

You are trying to decide the best order in which to answer your CS-570 exam questions within the duration of the exam. Suppose that there are n questions with points p_1, p_2, \dots, p_n and you need time t_i to completely answer the i^{th} question. You are confident that all your completely answered questions will be correct (and get you full credit for them), and the TAs will give you partial credit for an incompletely answered question, in proportion to the time you spent on that question. Assuming that the total exam duration is T , give a greedy algorithm to decide the order in which you should attempt the questions and prove that the algorithm gives you an optimal answer.

Solution: This is similar to the fractional knapsack problem. The greedy algorithm is as follows. Calculate $\frac{p_i}{t_i}$ for each $1 \leq i \leq n$ and sort the questions in descending order of $\frac{p_i}{t_i}$.

Let the sorted order of questions be denoted by $s(1), s(2), \dots, s(n)$. Answer questions in the order $s(1), s(2), \dots$ until $s(j)$ such that $\sum_{k=1}^j t_{s(k)} \leq T$ 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 question $s(j + 1)$ in the time remaining.

We'll use induction to prove optimality of the above algorithm. 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.

Inductive Step: Let $P(1), P(2), \dots, P(l)$ be true for some arbitrary l , i.e. the set of questions $\{s(1), s(2), \dots, s(l)\}$ are part of an optimal solution O . It is clear that O should also be optimal with respect to the remaining questions $\{1, 2, \dots, n\} \setminus \{s(1), s(2), \dots, s(l)\}$ 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 largest 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.

Induction Basis: 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 $\left(\frac{p_{s(1)}}{t_{s(1)}} - \frac{p_a}{t_a}\right) \min(t_a, T, t_{s(1)}) \geq 0$ since $\frac{p_i}{t_i}$ is 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.

4) 16 pts

An Edge Cover on a graph $G = (V; E)$ is a set of edges $X \subseteq E$ such that every vertex in V is incident to an edge in X . In the **Bipartite** Edge Cover problem, we are given a bipartite graph and wish to find an Edge Cover that contains $\leq k$ edges. Design a polynomial-time algorithm based on network flow (max flow or circulation) to solve it and justify your algorithm.

Solution:

If the network contains isolated node, then the problem is trivial since there is no way to do edge cover. Now we only consider the case that each node has at least 1 incident edge.

- i) The goal of edge cover is to choose as many edges as possible which cover 2 nodes. You can find this subset of edges by running Bipartite Matching on the original graph, and taking exactly the edges which are in the matching. (Equivalently, you can set capacity of each edge in the graph as 1. Set a super source node s connecting each “blue” node with edge capacity 1 and a super destination t connecting each “red” node with edge capacity 1. Then run max-flow to get the subset of edges connecting 2 nodes in G)
- ii) What remains is to cover the remaining nodes. Since you can only cover a single node (of those remaining) with each selected edge, simply choose an arbitrary incident edge to each uncovered node.
- iii) Set the set of edges you choose in the above two steps as set X . Count the total number of edges in X and compare the size with k .

Proof:

It is obvious that X is an edge cover. The remaining part is to show that X contains the minimum number of edges among all possible edge covers.

Denote the number of edges we find in step i) as x_1 ; denote the number of edges we find in step ii) as x_2 .

Then we have $x_1 * 2 + x_2 = |V|$.

Consider an arbitrary edge cover set Y . Suppose Y contains y_1 edges, each of which is counted as the one covering 2 nodes. Suppose Y contains y_2 edges, each of which is counted as the one covering 1 nodes. (We can ignore the edges covering zero nodes, because we can delete those edges from Y without affecting the coverage)

Then we have $y_1 * 2 + y_2 = |V|$.

Here x_1 must be the maximum number of edges that covers 2 nodes in the bipartite graph, because we do bipartite matching in G (max-flow in G' including s and t). Therefore, we have $x_1 \geq y_1$.

Then we have:

$$x_1 + x_2 = |V| - x_1 = y_1 * 2 + y_2 - x_1 = y_1 + y_2 - (x_1 - y_1) \leq y_1 + y_2.$$

The above algorithm gives the minimum number of edges for covering the nodes.

5) 16 pts

Imagine starting with the given decimal number n , and repeatedly chopping off a digit from one end or the other (your choice), until only one digit is left. The square-depth $\text{SQD}(n)$ of n is defined to be the maximum number of perfect squares you could observe among all such sequences. For example, $\text{SQD}(32492) = 3$ via the sequence

$$32492 \rightarrow 3249 \rightarrow 324 \rightarrow 24 \rightarrow 4$$

since 3249, 324, and 4 are perfect squares, and no other sequence of chops gives more than 3 perfect squares. Note that such a sequence may not be unique, e.g.

$$32492 \rightarrow 3249 \rightarrow 249 \rightarrow 49 \rightarrow 9$$

also gives you 3 perfect squares, viz. 3249, 49, and 9.

Describe an efficient algorithm to compute the square-depth $\text{SQD}(n)$, of a given number n , written as a d -digit decimal number $a_1 a_2 \dots a_d$. Analyze your algorithm's running time. Your algorithm should run in time polynomial in d . You may assume the availability of a function `IS_SQUARE(x)` that runs in constant time and returns 1 if x is a perfect square and 0 otherwise.

Solution:

We can solve this using dynamic programming.

First, we define some notation: let $n_{ij} = a_i \dots a_j$. That is, n_{ij} is the number formed by digits i through j of n . Now, define the subproblems by letting $D[i, j]$ be the square-depth of n_{ij} .

The solution to $D[i, j]$ can be found by solving the two subproblems that result from chopping off the left digit and from chopping off the right digit, and adding 1 if n_{ij} itself is square. We can express this as a recurrence:

$$D[i, j] = \max(D[i + 1, j], D[i, j - 1]) + \text{IS-SQUARE}(n_{ij})$$

The base cases are $D[i, i] = \text{IS-SQUARE}(a_i)$, for all $1 \leq i \leq d$. The solution to the general problem is $\text{SQD}(n) = D[1, d]$.

There are $\Theta(d^2)$ subproblems, and each takes $\Theta(1)$ time to solve, so the total running time is $\Theta(d^2)$.

*** Some students did a greedy approach to solve this problem which is not true for this problem. In each step of the program they explore removing which digit results in a perfect square number, however it might be the other non-removed digit that will produce more squares in the future steps.

6) 16 pts

The Longest Path is the problem of deciding whether a graph has a simple path of length greater or equal to a given number k .

a) Show that the Longest Path problem is in NP (2 pts)

b) Show how the Hamiltonian Cycle problem can be reduced to the Longest Path problem. (14 pts)

Note: You can choose to do the reduction in b either directly, or use transitivity of polynomial time reduction to first reduce Hamiltonian Cycle to another problem (X) and then reduce X to Longest Path.

a) Polynomial length certificate: ordered list of nodes on a path of length $\geq k$

polynomial time Certifier:

check that nodes do not repeat in $O(n \log n)$

check that the length of the path is at least k in $O(n)$

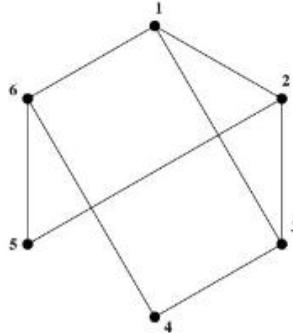
check that there are edges between every two adjacent nodes on the path in $O(n)$

check that the length of the path is at least k in $O(n)$

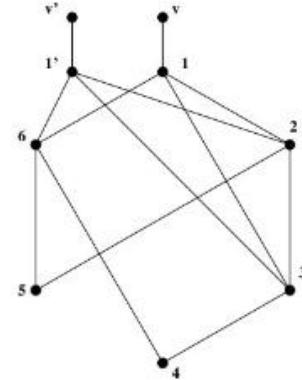
b) Two possible solutions:

I. To reduce directly from Hamiltonian Cycle

Given a graph $G = (V, E)$ we construct a graph G' such that G contains a Ham cycle iff G' contains a simple path of length at least $N+2$. This is done by choosing an arbitrary vertex u in G and adding a copy, u' , of it together with all its edges. Then add vertices v and v' to the graph and connect v with u and v' to u' . We give a cost of 1 to all edges in G' . See figure below:



G



G'

Proof:

A) If there is a HC in G we can find a simple path of length $n+2$ in G' : This path starts at v' goes to U' and follows the HC to u and then to v . The length is $n+2$

B) If there is a simple path of length $n+2$ in G' , there is a HC in G : This path must

include nodes v' and v because there are only n nodes in G and a simple path of length $n+2$ must include v and v' . Moreover, v and v' must be the two ends of this path, otherwise, the path will not be a simple path since there is only one way to get to v and v' . So, to find the HC in G , just follow the path from u' to u .

II. To reduce using Hamiltonian Path

First reduce Ham Cycle to Ham Path (very similar to the above reduction)

Then reduce Ham Path to Longest path. This is very straightforward. To find out if there is a Ham Path in G you can assign weights of 1 to all edges and ask the blackbox if there is a path of length at least n in G .

Additional Space

CS570
Analysis of Algorithms
Fall 2014
Exam III

Name: _____
Student ID: _____
Email: _____

Wednesday Evening Section

	Maximum	Received
Problem 1	20	
Problem 2	16	
Problem 3	16	
Problem 4	16	
Problem 5	16	
Problem 6	16	
Total	100	

Instructions:

1. This is a 2-hr exam. Closed book and notes
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.

1) 20 pts

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

[TRUE/FALSE]

All the NP-hard problems are in NP.

[TRUE/FALSE]

Given a weighted graph and two nodes, it is possible to list all shortest paths between these two nodes in polynomial time.

[TRUE/FALSE]

In the memory efficient implementation of Bellman-Ford, the number of iterations it takes to converge can vary depending on the order of nodes updated within an iteration

[TRUE/FALSE]

There is a feasible circulation with demands $\{d_v\}$ if $\sum_v d_v = 0$.

[TRUE/FALSE]

Not every decision problem in P has a polynomial time certifier.

[TRUE/FALSE]

If a problem can be reduced to linear programming in polynomial time then that problem is in P.

[TRUE/FALSE]

If we can prove that $P \neq NP$, then a problem $A \in P$ does not belong to NP.

[TRUE/FALSE]

If all capacities in a flow network are integers, then every maximum flow in the network is such that flow value on each edge is an integer.

[TRUE/FALSE]

In a dynamic programming formulation, the sub-problems must be mutually independent.

[TRUE/FALSE]

In the final residual graph constructed during the execution of the Ford–Fulkerson Algorithm, there's no path from sink to source.

2) 16 pts

In the Bipartite Directed Hamiltonian Cycle problem, we are given a bipartite directed graph $G = (V; E)$ and asked whether there is a simple cycle which visits every node exactly once. Note that this problem might potentially be easier than Directed Hamiltonian Cycle because it assumes a bipartite graph. Prove that Bipartite Directed Hamiltonian Cycle is in fact still NP-Complete.

3) 16 pts

A tourism company is providing boat tours on a river with n consecutive segments. According to previous experience, the profit they can make by providing boat tours on segment i is known as a_i . Here a_i could be positive (they earn money), negative (they lose money), or zero. Because of the administration convenience, the local community of the river requires that the tourism company should do their boat tour business on a contiguous sequence of the river segments, i.e, if the company chooses segment i as the starting segment and segment j as the ending segment, all the segments in between should also be covered by the tour service, no matter whether the company will earn or lose money. The company's goal is to determine the starting segment and ending segment of boat tours along the river, such that their total profit can be maximized. Design an efficient algorithm to achieve this goal, and analyze its run time (Note that brute-force algorithm achieves $\Theta(n^2)$, so your algorithm must do better.)

4) 16 pts

Consider the following matching problem. There are m students s_1, s_2, \dots, s_m and a set of n companies $C = \{c_1, c_2, \dots, c_n\}$. Each student can work for only one company, whereas company c_j can hire up to b_j students. Student s_i has a preferred set of companies $\Lambda_i \subseteq C$ at which he/she is willing to work. Your task is to find an assignment of students to companies such that all of the above constraints are satisfied and each student is assigned. Formulate this as a network flow problem and describe any subsequent steps necessary to arrive at the solution. Prove correctness.

5) 16 pts

Consider a directed, weighted graph G where all edge weights are positive. You have one Star, which lets you change the weight of any one edge to zero. In other words, you may change the weight of any one edge to zero. Give an efficient algorithm using Dijkstra's algorithm to find a lowest-cost path between two vertices s and t , given that you may set one edge weight to zero. Note: you will receive 10 pts if your algorithm is efficient. You will receive full points (16 pts) if your algorithm has the same run time complexity as Dijkstra's algorithm.

6) 16 pts

You are given n rods; they are of length l_1, l_2, \dots, l_n , respectively. Our goal is to connect all the rods and form a single rod. The length after connecting two rods and the cost of connecting them are both equal to the sum of their lengths. Give an algorithm to minimize the cost of connecting them to form a single rod. State the complexity of your algorithm and prove that your algorithm is optimal.

Additional Space

CS570
Analysis of Algorithms
Fall 2015
Exam III

Name: _____

Student ID: _____

Email Address: _____

_____ Check if DEN Student

	Maximum	Received
Problem 1	20	
Problem 2	15	
Problem 3	12	
Problem 4	15	
Problem 5	15	
Problem 6	12	
Problem 7	11	
Total	100	

Instructions:

1. This is a 2-hr exam. Closed book and notes
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.

1) 20 pts

Mark the following statements as **TRUE, FALSE**. No need to provide any justification.

[TRUE/FALSE]

If P = NP, then all NP-Hard problems can be solved in Polynomial time.

[TRUE/FALSE]

Dynamic Programming approach only works when used on problems with non-overlapping sub problems.

[TRUE/FALSE]

In a divide & conquer algorithm, the size of each sub-problem must be at most half the size of the original problem.

[TRUE/FALSE]

In a 0-1 knapsack problem, a solution that uses up all of the capacity of the knapsack will be optimal.

[TRUE/FALSE]

If a problem X can be reduced to a known NP-hard problem, then X must be NP-hard.

[TRUE/FALSE]

If SAT \leq_P A, then A is NP-hard.

[TRUE/FALSE]

The recurrence $T(n) = 2T(n/2) + 3n$, has solution $T(n) = \theta(n \log(n^2))$.

[TRUE/FALSE]

Consider two positively weighted graphs $G_1 = (V, E, w_1)$ and $G_2 = (V, E, w_2)$ with the same vertices V and edges E such that, for any edge $e \in E$, we have $w_2(e) = (w_1(e))^2$. For any two vertices $u, v \in V$, any shortest path between u and v in G_2 is also a shortest path in G_1 .

[TRUE/FALSE]

If an undirected graph G=(V,E) has a Hamiltonian Cycle, then any DFS tree in G has a depth $|V| - 1$.

[TRUE/FALSE]

Linear programming is at least as hard as the Max Flow problem.

2) 15 pts

A company makes three models of desks, an executive model, an office model and a student model. Building each desk takes time in the cabinet shop, the finishing shop and the crating shop as shown in the table below:

Type of desk	Cabinet shop	Finishing shop	Crating shop	Profit
Executive	2	1	1	150
Office	1	2	1	125
Student	1	1	.5	50
Available hours	16	16	10	

How many of each type should they make to maximize profit? Use linear programming to formulate your solution. Assume that real numbers are acceptable in your solution.

3) 12 pts

Given a graph $G=(V, E)$ and a positive integer $k < |V|$. The longest-simple-cycle problem is the problem of determining whether a simple cycle (no repeated vertices) of length k exists in a graph. Show that this problem is NP-complete.

4) 15 pts

Suppose there are n steps, and one can climb either 1, 2, or 3 steps at a time. Determine how many different ways one can climb the n steps. E.g. if there are 5 steps, these are some possible ways to climb them: (1,1,1,1,1), (1,2,1,1), (3, 2), (2,3), etc. Your algorithm should run in linear time with respect to n . You need to include your complexity analysis.

5) 15 pts

We'd like to select frequencies for FM radio stations so that no two are too close in frequency (creating interference). Suppose there are n candidate frequencies $\{f_1, \dots, f_n\}$. Our goal is to pick as many frequencies as possible such that no two selected frequencies f_i, f_j have $|f_i - f_j| < e$ (for a given input variable e). Design a greedy algorithm to solve the problem. Prove the optimality of the algorithm and analyze the running time.

6) 12 pts

Let S be an NP-complete problem, and Q and R be two problems whose classification is unknown (i.e. we don't know whether they are in NP, or NP-hard, etc.). We do know that Q is polynomial time reducible to S and S is polynomial time reducible to R. Mark the following statements True or False **based only on the given information, and explain why.**

(i) Q is NP-complete

(ii) Q is NP-hard

(iii) R is NP-complete

(iv) R is NP-hard

7) 11 pts

Consider there are n students and n rooms. A student can only be assigned to one room. Each room has capacity to hold either one or two students. Each student has a subset of rooms as their possible choice. We also need to make sure that there is at least one student assigned to each room.

Give a polynomial time algorithm that determines whether a feasible assignment of students to rooms is possible that meets all of the above constraints. If there is a feasible assignment, describe how your solution can identify which student is assigned to which room.

Additional Space

Additional Space

CS570
Analysis of Algorithms
Fall 2015
Exam III

Name: _____

Student ID: _____

Email Address: _____

_____ Check if DEN Student

	Maximum	Received
Problem 1	20	
Problem 2	15	
Problem 3	12	
Problem 4	15	
Problem 5	15	
Problem 6	12	
Problem 7	11	
Total	100	

Instructions:

1. This is a 2-hr exam. Closed book and notes
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.

1) 20 pts

Mark the following statements as **TRUE**, **FALSE**. No need to provide any justification.

[/FALSE]

If P = NP, then all NP-Hard problems can be solved in Polynomial time.

[/FALSE]

Dynamic Programming approach only works when used on problems with non-overlapping sub problems.

[/FALSE]

In a divide & conquer algorithm, the size of each sub-problem must be at most half the size of the original problem.

[/FALSE]

In a 0-1 knapsack problem, a solution that uses up all of the capacity of the knapsack will be optimal.

[/FALSE]

If a problem X can be reduced to a known NP-hard problem, then X must be NP-hard.

[TRUE/]

If $\text{SAT} \leq_P A$, then A is NP-hard.

[TRUE/]

The recurrence $T(n) = 2T(n/2) + 3n$, has solution $T(n) = \theta(n \log(n^2))$.

[/FALSE]

Consider two positively weighted graphs $G_1 = (V, E, w_1)$ and $G_2 = (V, E, w_2)$ with the same vertices V and edges E such that, for any edge $e \in E$, we have $w_2(e) = (w_1(e))^2$. For any two vertices $u, v \in V$, any shortest path between u and v in G_2 is also a shortest path in G_1 .

[/FALSE]

If an undirected graph $G=(V,E)$ has a Hamiltonian Cycle, then any DFS tree in G has a depth $|V| - 1$.

[TRUE/]

Linear programming is at least as hard as the Max Flow problem.

2) 15 pts

A company makes three models of desks, an executive model, an office model and a student model. Building each desk takes time in the cabinet shop, the finishing shop and the crating shop as shown in the table below:

Type of desk	Cabinet shop	Finishing shop	Crating shop	Profit
Executive	2	1	1	150
Office	1	2	1	125
Student	1	1	.5	50
Available hours	16	16	10	

How many of each type should they make to maximize profit? Use linear programming to formulate your solution. Assume that real numbers are acceptable in your solution.

Solution:

Start by defining your variables:

x = number of executive desks made

y = number of office desks made

z = number of student desks made

Maximize $P=150x+125y+50z$.

Subject to:

$2x + y + z \leq 16$ cabinet hours

$x + 2y + z \leq 16$ finishing hours

$x + y + .5z \leq 10$ crating hours

$x \geq 0, y \geq 0, z \geq 0$

3) 12 pts

Given a graph $G=(V, E)$ and a positive integer $k < |V|$. The longest-simple-cycle problem is the problem of determining whether a simple cycle (no repeated vertices) of length k exists in a graph. Show that this problem is NP-complete.

Solution:

Clearly this problem is in NP. The certificate will be a cycle of the graph, and the certifier will check whether the certificate is really a cycle of length k of the given graph.

We will reduce HAM-CYCLE to this problem. Given an instance of HAM-CYCLE with graph $G=(V, E)$, construct a new graph $G'=(V', E)$ by adding one isolated vertex u to G . Now ask the longest-simple-cycle problem with $k = |V| < |V'|$ for graph G' .

If there is a HAM-CYCLE in G , it will be a cycle of length $k = |V|$ in G' .

If there is no HAM-CYCLE in G' , there must not be no cycle of length $|V|$ in G' . If there is, we know the cycle does not contain u , because it is isolated. So the cycle will contain all vertex in $|V|$, which is a HAM-CYCLE of the G .

The reduction is in polynomial time, so longest-simple-path is NP-Complete.

4) 15 pts

Suppose there are n steps, and one can climb either 1, 2, or 3 steps at a time. Determine how many different ways one can climb the n steps. E.g. if there are 5 steps, these are some possible ways to climb them: (1,1,1,1,1), (1,2,1,1), (3, 2), (2,3), etc. Your algorithm should run in linear time with respect to n . You need to include your complexity analysis.

Solution:

Let $T(n)$ denote the number of different ways for climbing up n stairs.

There are three choices for each first step: either 1, 2, or 3. $T(n) = T(n-1) + T(n-2) + T(n-3)$

The boundary conditions :

when there is only one step, $T(1) = 1$. If only two steps, $T(2) = 2$. $T(3)= 4$. ((1,1,1), (1,2), (2,1), (3))

Pseudo code as below:

```
if n is 1
    return 1
else if n is 2
    return 2
else
    T[1] = 1
    T[2] = 2
    T[3]=4
    for i = 4 to n do
        T[i] = T[i-1]+T[i-2]+ T[i-3]
    return T[n]
```

The time complexity is $\Theta(n)$.

5) 15 pts

We'd like to select frequencies for FM radio stations so that no two are too close in frequency (creating interference). Suppose there are n candidate frequencies $\{f_1, \dots, f_n\}$. Our goal is to pick as many frequencies as possible such that no two selected frequencies f_i, f_j have $|f_i - f_j| < e$ (for a given input variable e). Design a greedy algorithm to solve the problem. Prove the optimality of the algorithm and analyze the running time.

Proof:

I claim that there exists some optimal solution that makes the same first choice as (in terms of which selected frequency has the lowest value) that I do. Consider any optimal solution OPT .

Let p be my first choice and q be OPT 's first choice. If $p = q$, our proof is complete. If it isn't, we know that $p < q$; now consider some other set $OPT1 = OPT - \{q\} + \{p\}$; that is, OPT with its first choice replaced by mine. We know this is a valid set (meaning no frequencies are within e of one another), because $p < q$, and p is the first element in $OPT1$. Because nothing was within e of q , none can be within e of p . We also know that $OPT1$ is an optimal (maximum size) set, because it is the same size as OPT . Therefore, $OPT1$ is an optimal set that includes my first choice. (and therefore, by induction, the second choice will be correct, etc.)

6) 12 pts

Let S be an NP-complete problem, and Q and R be two problems whose classification is unknown (i.e. we don't know whether they are in NP, or NP-hard, etc.). We do know that Q is polynomial time reducible to S and S is polynomial time reducible to R. Mark the following statements True or False **based only on the given information, and explain why.**

(i) Q is NP-complete

False. Because $Q \leq_p S$, Q is at most as hard as S. Because S is in NPC, Q is not necessary to be in NPC, e.g., it could be in P, or could be in NP but not in NPC.

(ii) Q is NP-hard

False. Because $Q \leq_p S$, Q is at most as hard as S. Then Q is not necessary to be in NP-hard, e.g., it could be in P.

(iii) R is NP-complete

False. Because $S \leq_p R$, R is at least as hard as S. Then R is not necessary to be NPC. It is possible to be in NP-hard but not in NPC.

(iv) R is NP-hard

True. Because $S \leq_p R$, R is at least as hard as S. Then R is NP-hard.

7) 11 pts

Consider there are n students and n rooms. A student can only be assigned to one room. Each room has capacity to hold either one or two students. Each student has a subset of rooms as their possible choice. We also need to make sure that there is at least one student assigned to each room.

Give a polynomial time algorithm that determines whether a feasible assignment of students to rooms is possible that meets all of the above constraints. If there is a feasible assignment, describe how your solution can identify which student is assigned to which room.

Solution:

Construct a flow network as follows,

For each student i create a vertex a_i , for each room j create a vertex b_j , if room j is one of student i 's possible choices, add an edge from a_i to b_j with upper bound 1. Create a super source s , connect s to all a_i with an edge of lower bound and upper bound 1.

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

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

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 a_i and b_j , check whether the flow from a_i to b_j is 1, if yes, assign student i to room j .

Additional Space

Additional Space

CS570 Fall 2018: Analysis of Algorithms Exam III

	Points		Points
Problem 1	20	Problem 5	10
Problem 2	8	Problem 6	16
Problem 3	14	Problem 7	12
Problem 4	20		
	Total	100	

Instructions:

1. This is a 2-hr exam. Closed book and notes
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.

1) 20 pts

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

[**TRUE/FALSE**]

Let X be a decision problem. If we prove that X is in the class NP and give a poly-time reduction from X to 3-SAT, we can conclude that X is NP-complete.

[**TRUE/FALSE**]

Let A be an algorithm that operates on a list of n objects, where n is a power of two. A spends $\Theta(n^2)$ time dividing its input list into two equal pieces and selecting one of the two pieces. It then calls itself recursively on that list of $n/2$ elements. Then A 's running time on a list of n elements is $O(n)$.

[**TRUE/FALSE**]

If there is a polynomial time algorithm to solve problem A then A is in NP.

[**TRUE/FALSE**]

A pseudo-polynomial time algorithm is always slower than a polynomial time algorithm.

[**TRUE/FALSE**]

In a dynamic programming formulation, the sub-problems must be non-overlapping.

[**TRUE/FALSE**]

A spanning tree of a given undirected, connected graph $G = (V, E)$ can be found in $O(E)$ time.

[**TRUE/FALSE**]

Ford-Fulkerson can return a zero maximum flow for flow networks with non-zero capacities.

[**TRUE/FALSE**]

If $f(n) = \Theta(g(n))$ and $g(n) = \Theta(h(n))$, then $h(n) = \Theta(f(n))$

[**TRUE/FALSE**]

There is a polynomial-time solution for the 0/1 Knapsack problem if all items have the same weight but different values.

[**TRUE/FALSE**]

If there are negative cost edges in a graph but no negative cost cycles, Dijkstra's algorithm still runs correctly.

2) 8 pts

Let $G = (V, E)$ be a simple graph with n vertices. Suppose the weight of every edge of G is one.

Note: In a simple graph there is at most one edge directly connecting any two nodes.

For example, between nodes u and v there will be at most one edge uv .

- (a) What is the weight of a minimum spanning tree of G ? (1 pt)
- (b) Suppose we change the weight of two edges of G to $1/2$. What is the weight of the minimum spanning tree of G ? (2 pts)
- (c) Suppose we change the weight of three edges of G to $1/2$. What is the minimum and maximum possible weights for the the minimum spanning tree of G ? (2 pts)
- (d) Suppose we change the weight of $k < V$ edges of G to $1/2$. What is the minimum and maximum possible weights for the the minimum spanning tree of G ? (3 pts)
1. Any spanning tree of G has exactly $V - 1$ edges, since all the weights are one its total weight is $V - 1$.
 2. To answer this question, consider running Kruskal on G , it will first pick the two lighter edges as they cannot form a cycle (the graph is simple), then $V - 3$ other edge each with weight one. Therefore, the total value is $V - 3 + 2(1/2) = V - 2$.
 3. Use the similar approach as (b). Kruskal first selects two light edges. There are two possibilities for the third edge: (i) it forms a cycle with the first two edges or (ii) it does not form a cycle with the first two edges not. In case (i), Kruskal selects two edges of weight $1/2$ and $V - 3$ edges of weight 1, therefore, the total weight is $V - 3 + 2(1/2) = V - 2$. In case (ii), Kruskal selects three edges of weight $1/2$ and $V - 4$ edges of weight 1, therefore, the total weight is $V - 4 + 3(1/2) = V - 5/2$.
 4. Similar to (c), the minimum weight happens when the k light edges do not form any cycle. In this case, MST contains k edges of weight $1/2$ and $n - 1 - k$ edges of weight 1. Therefore, its weight is $k/2 + n - 1 - k = n - k/2 - 1$. The maximum weight happens if the k edges span as few vertices as possible. This happens when the k edges are part of an almost *complete graph*. Let h be a variable denoting the number of nodes in this subgraph such that the number of edges in a complete graph comprised of these edges nodes exceeds k . In particular, define h to be the smallest number such that, the binomial coefficient $\text{Binomial}[h,2] > k$, ie., $(h \text{ choose } 2) > k$. Accordingly, we can pack all k light edges between h vertices. Consequently, the MST has $h - 1$ edges of weight $1/2$ and $n - 1 - (h - 1)$ edges of weight 1. Therefore, its weight is $n - h/2 - 1/2$.

Rubric:

For each part (a,b,c,d), No partial marking.

This also applies to the parts where minimum and maximum is asked.

If either is wrong then there is no partial marks for that part.

Q3. 14 pts

Recall that in the discussion class we showed that the Bipartite **Undirected** Hamiltonian Cycle problem is NP-complete. Now in the Bipartite **Directed** Hamiltonian Cycle problem, we are given a bipartite directed graph and asked whether there is a simple cycle which visits every node exactly once. Note that this problem might potentially be easier than Hamiltonian Cycle because it assumes a directed bipartite graph. Prove that Bipartite Directed Hamiltonian Cycle is in fact still NP-Complete.

Solution:

- i) This problem is NP. Given a candidate path, we just check the nodes along the given path one by one to see if every node in the network is visited exactly once and if each consecutive node pair along the path are joined by an edge. (3)
- ii) To prove the problem is NP-complete, we reduce Directed Hamiltonian Cycle (DHC) problem to Bipartite Directed Hamiltonian Cycle (BDHC) problem. (2)

Given an arbitrary directed graph G , we split each vertex v in G into two vertex v_{in} and v_{out} . Here v_{in} connects all the incoming edges to v in G ; v_{out} connects all the outgoing edges from v in G . Moreover, we connect one directed edge from v_{in} to v_{out} . After doing these operations for each node in G , we form a new graph G' . (3)

Here G' is bipartite graph, because we can color each v_{in} “blue” and each v_{out} “red” without any coloring conflict.

If there is DHC in G , then there is a BDHC in G' . We can replace each node v on the DHC in G into consecutive nodes v_{in} and v_{out} , and (v_{in}, v_{out}) is an edge in G' . Then the new path is a BDHC in G' . (3)

On the other hand, if there is BDHC in G' , then there is a DHC in G . Note that if v_{in} is on the BDHC, v_{out} must be the successive node on the BDHC. Then we can merge each node pair (v_{in}, v_{out}) on the BDHC in G' into node v and form a DHC in G . (3)

In sum, G has DHC if and only if G' has a BDHC. This completes the reduction, and we confirm that the given problem is NP-complete

Alternate solution:

We can also use Undirected Hamiltonian Cycle for the reduction.

Prove that it's NP as before. (3)

Note that you are using Bipartite Undirected Hamiltonian Cycle for reduction. (2)

Given an arbitrary undirected bipartite graph G , convert G to G' so the vertices in G' stay the same, but all undirected edges are converted to directed edges in **both directions**. G' remains bipartite. (3)

If there is an undirected Hamiltonian cycle in G, you can use the corresponding edges(in either direction) to find a Directed Hamiltonian cycle in in G'. (3)

On the other hand, if there is a directed Hamiltonian cycle in G', you could convert the corresponding edges to undirected edges and find the equivalent cycle in G. (3)

Common mistakes:

Major mistake in checking for NP: -2

Checking for edges instead of vertices: -1

Missing the NP check altogether: -3

Mistake in the conversion step: -2

Missing the two-way proof: -4

Missing either side of the proof: -3

Incomplete explanation as to why the proof holds: -2

If you are proposing to reduce a certain NP-Complete problem, but are going with a solution corresponding a different NP-Complete problem, this shows misunderstanding of the concept: -7

If you are proposing to reduce an incorrect (and irrelevant) NP-complete problem: -8

4) 20 pts.

Suppose you want to sell n roses. You need to group the roses into multiple bouquets with different sizes. The value of each bouquet depends on the number of roses you used. In other words, we know that a bouquet of size i will sell for $v[i]$ dollars. Provide an algorithm to find the maximum possible value of the roses by deciding how to group them into different-sized bouquets.

Here is an example:

Input: $n = 5$, $v[1..n] = [2,3,7,8,10]$ (i.e. a bouquet of size 1 is \$2, bouquet of size 2 is \$3, ..., and finally a single bouquet of size $n=5$ is \$10.

Expected Output = 11 (by grouping the roses into bouquets of size 1, 1, and 3)

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

$OPT(j)$ is the maximum value of j roses.

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

$OPT(j) = \text{MAX } (OPT(j), \{OPT(j-i) + v[i]\} \text{ for all sizes } 0 \leq i < j)$

i.If iteration of i from 0 to j is missing => -1 points

ii.If $OPT(j)$ is missing in the MAX term => -1 points

iii.0 points for wrong recursion

c) Using the recurrence formula in part b, write pseudocode to compute the maximum possible value of the n roses. (6 pts)

Make sure you have initial values properly assigned. (2 pts)

```
OPT[0] = 0
OPT[1] = v[1]
For (int j=2; j < n; j++)
    max = 0
    For (int i=1; i < j; i++)
        If (i<=j && max < OPT[j-i]+v[i] )
            max = OPT[j-i]+v[i]
    OPT[j]=max
```

i) Missing one loop => -1 point

ii) Missing two loops => -2 points

iii) If in b) you got x points, you get x points in the pseudo code. Apart from the case where you missed the iteration in b, where you will get x+1 points in this question

iv) 0 points for no pseudocode or any modification from b.

d) Compute the runtime of the algorithm described in part c and state whether your solution runs in polynomial time or not (2 pts)

e) d) $O(n^2)$

5) 10 pts

Give a tight asymptotic upper bound (O notation) on the solution to each of the following recurrences. You need not justify your answers.

$$T(n) = 2T(n/8) + n$$

Solution: $(n^{1/3} \lg n)$ by Case 2 of the Master Method.

$$T(n) = T(n/3) + T(n/4) + 5n$$

Solution: Use brute force method

$$\begin{aligned} T(n) &= cn + (7/12) cn + (7/12)^2 cn + \dots \\ &= (n) \end{aligned}$$

Rubric:

No partial marks. 5 points for each correct answer.

6) 16 pts

There are n people and m jobs. You are given a payoff matrix, C , where C_{ij} represents the payoff for assigning person i to do job j . Each applicant has a subset of jobs that he/she is interested in. Each job opening can only accept one applicant and a job applicant can be appointed for only one job. You need to find an assignment of jobs to applicants that maximizes the total payoff of your assignment. Write an **Integer Linear Program** (with discrete variables) to solve this problem.

Solution

We are looking for a perfect matching on a bipartite graph of the minimal cost.

Let $x_{ij} = \begin{cases} 1, & \text{if edge}(i,j) \text{ is in matching} \\ 0, & \text{otherwise} \end{cases}$

Objective: $\sum_{\substack{1 \leq i \leq m \\ 1 \leq j \leq n}} c_{ij} \cdot x_{ij}$

Subject to:

1. $\sum_j x_{ij} \leq 1$, where $i = 1, 2, \dots, n$
2. $\sum_i x_{ij} \leq 1$, where $j = 1, 2, \dots, n$
3. $x_{ij} \in \{0,1\}$, $\forall i, j$
4. $x_{ij} = 0$, for all jobs j that applicant i is not interested in.

Rubric:

- 4 if objective function is incorrect.
- 6 if you mention equal to instead of less than equal to sign for constraints 1 and 2
- 2 if constraint 3. is not mentioned, but only is defined to take values 0 or 1.
- 2 if discrete variable is not defined.
- 0 points awarded if you use a max-flow, min-cut formulation

7) 12 pts

The Maximum Acyclic Subgraph is stated as follows: Given a directed graph find an acyclic subgraph of that contains as many arcs (i.e. directed edges) as possible.

Give a 2-approximation algorithm for this problem.

Hint: Consider using an arbitrary ordering of the vertices in G.

Solution:

1. Pick an arbitrary vertex ordering. This partitions the arcs into two acyclic subgraphs and . Here A_1 is the set of arcs where $i < j$ and A_2 is the set of arcs where $i > j$. Thus at least one of A_1 or A_2 contains half the arcs of G . The maximum acyclic subgraph contains at most the total number of arcs in G , so we have a factor 2-approximation algorithm (12)
2. Divide the set of nodes into two nonempty sets, A and B . Consider the set of edges from A to B and the set of edges from B to A . Throw out the edges from the smaller set and keep the edges from the larger set (break ties arbitrarily). Recursively perform the algorithm on A and B individually (12)

Rubric:

If you are only considering the forward or backward edges (-2)

If you start with one node and add the forward/backward edges to/from neighbors, your answer won't work for disconnected graphs (-2)

If your final answer is not 0.5 approximation (-6)

If your answer is not acyclic (-6)

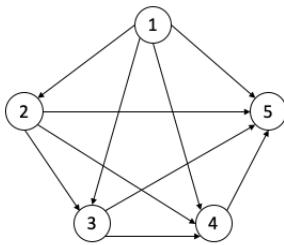
If your algorithm is vague and not implementable (for example if you said, "we find max independent set", or "we find the topological order") (0)

If your algorithm has conceptual flaws (for example if you are adding an edge to the sub graph and also deleting it from the sub graph) (0 pts)

Some of the answers that do not work:

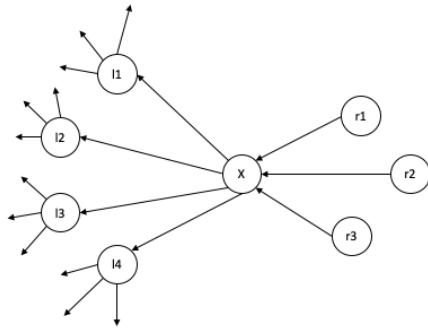
1. If your final answer is a tree (a DFS or a BFS or any other kinds of trees):

You can imagine an acyclic complete directed graph such as below. Your algorithm should return at least half of the edges ($n * (n-1) / 4$) but a tree can have $n-1$ edges. Thus it is not 0.5 approximation (6 pts).



2. If you start adding/removing edges to/from the graph, until it is acyclic. Your algorithm can choose an edge that is repeated in many cycles and end up removing all the other edges in those cycles.

You can try your method on the following graph (All $l(i)$ nodes are connected to all $r(j)$ nodes). If your algorithm chooses all the $(r(j), X)$ and $(X, l(i))$ edges (7 edges), you cannot add any $(l(i), r(j))$ edges (12 edges). Your final answer (7) is less than half of the best answer ($16/2 = 8$) so it is not 0.5 approximation.



CS570
Analysis of Algorithms
Fall 2014
Exam III

Name: _____
Student ID: _____
Email: _____

Wednesday Evening Section

	Maximum	Received
Problem 1	20	
Problem 2	16	
Problem 3	16	
Problem 4	16	
Problem 5	16	
Problem 6	16	
Total	100	

Instructions:

1. This is a 2-hr exam. Closed book and notes
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.

1) 20 pts

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

[TRUE/FALSE]

All the NP-hard problems are in NP.

[TRUE/FALSE]

Given a weighted graph and two nodes, it is possible to list all shortest paths between these two nodes in polynomial time.

[TRUE/FALSE]

In the memory efficient implementation of Bellman-Ford, the number of iterations it takes to converge can vary depending on the order of nodes updated within an iteration

[TRUE/FALSE]

There is a feasible circulation with demands $\{d_v\}$ if $\sum_v d_v = 0$.

[TRUE/FALSE]

Not every decision problem in P has a polynomial time certifier.

[TRUE/FALSE]

If a problem can be reduced to linear programming in polynomial time then that problem is in P.

[TRUE/FALSE]

If we can prove that $P \neq NP$, then a problem $A \in P$ does not belong to NP.

[TRUE/FALSE]

If all capacities in a flow network are integers, then every maximum flow in the network is such that flow value on each edge is an integer.

[TRUE/FALSE]

In a dynamic programming formulation, the sub-problems must be mutually independent.

[TRUE/FALSE]

In the final residual graph constructed during the execution of the Ford–Fulkerson Algorithm, there's no path from sink to source.

2) 16 pts

In the Bipartite Directed Hamiltonian Cycle problem, we are given a bipartite directed graph $G = (V; E)$ and asked whether there is a simple cycle which visits every node exactly once. Note that this problem might potentially be easier than Directed Hamiltonian Cycle because it assumes a bipartite graph. Prove that Bipartite Directed Hamiltonian Cycle is in fact still NP-Complete.

3) 16 pts

A tourism company is providing boat tours on a river with n consecutive segments. According to previous experience, the profit they can make by providing boat tours on segment i is known as a_i . Here a_i could be positive (they earn money), negative (they lose money), or zero. Because of the administration convenience, the local community of the river requires that the tourism company should do their boat tour business on a contiguous sequence of the river segments, i.e, if the company chooses segment i as the starting segment and segment j as the ending segment, all the segments in between should also be covered by the tour service, no matter whether the company will earn or lose money. The company's goal is to determine the starting segment and ending segment of boat tours along the river, such that their total profit can be maximized. Design an efficient algorithm to achieve this goal, and analyze its run time (Note that brute-force algorithm achieves $\Theta(n^2)$, so your algorithm must do better.)

4) 16 pts

Consider the following matching problem. There are m students s_1, s_2, \dots, s_m and a set of n companies $C = \{c_1, c_2, \dots, c_n\}$. Each student can work for only one company, whereas company c_j can hire up to b_j students. Student s_i has a preferred set of companies $\Lambda_i \subseteq C$ at which he/she is willing to work. Your task is to find an assignment of students to companies such that all of the above constraints are satisfied and each student is assigned. Formulate this as a network flow problem and describe any subsequent steps necessary to arrive at the solution. Prove correctness.

5) 16 pts

Consider a directed, weighted graph G where all edge weights are positive. You have one Star, which lets you change the weight of any one edge to zero. In other words, you may change the weight of any one edge to zero. Give an efficient algorithm using Dijkstra's algorithm to find a lowest-cost path between two vertices s and t , given that you may set one edge weight to zero. Note: you will receive 10 pts if your algorithm is efficient. You will receive full points (16 pts) if your algorithm has the same run time complexity as Dijkstra's algorithm.

6) 16 pts

You are given n rods; they are of length l_1, l_2, \dots, l_n , respectively. Our goal is to connect all the rods and form a single rod. The length after connecting two rods and the cost of connecting them are both equal to the sum of their lengths. Give an algorithm to minimize the cost of connecting them to form a single rod. State the complexity of your algorithm and prove that your algorithm is optimal.

Additional Space

CS570
Analysis of Algorithms
Spring 2015
Exam III

Name: _____

Student ID: _____

Email Address: _____

_____ **Check if DEN Student**

	Maximum	Received
Problem 1	20	
Problem 2	16	
Problem 3	16	
Problem 4	16	
Problem 5	16	
Problem 6	16	
Total	100	

Instructions:

1. This is a 2-hr exam. Closed book and notes
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.

1) 20 pts

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

[**TRUE**]

If $\text{SAT} \leq_P A$, then A is NP-hard.

[**FALSE**]

If a problem X can be reduced to a known NP-hard problem, then X must be NP-hard.

[**TRUE**]

If P equals NP, then NP equals NP-complete.

[**FALSE**]

Let X be a decision problem. If we prove that X is in the class NP and give a poly-time reduction from X to Hamiltonian Cycle, we can conclude that X is NP-complete.

[**TRUE**]

The recurrence $T(n) = 2T(n/2) + 3n$, has solution $T(n) = \theta(n \log(n^2))$.

[**FALSE**]

On a connected, directed graph with only positive edge weights, Bellman-Ford runs asymptotically as fast as Dijkstra.

[**TRUE**]

Linear programming is at least as hard as the Max Flow problem in a flow network.

[**TRUE**]

If you are given a maximum s-t flow in a graph then you can find a minimum s-t cut in time $O(m)$ where m is the number of the edges in the graph.

[**TRUE**]

Fibonacci heaps can be used to make Dijkstra's algorithm run in $O(|E| + |V| \log|V|)$ time on a graph $G=(V,E)$

[**FALSE**]

A graph with non-unique edge weights will have at least two minimum spanning trees

2) 16 pts

Given a graph $G=(V, E)$ with an even number of vertices as the input, the HALF-IS problem is to decide if G has an independent set of size $|V|/2$. Prove that HALF-IS is in NP-Complete.

Solution:

Given a graph $G(V, E)$ and a certifier $S \subset V$, $|S| = |V|/2$, we can verify if no two nodes are adjacent in polynomial time ($O(|S|^2) = O(|V|^2)$). Therefore $\text{HALF-IS} \in NP$.

We prove the NP-Hardness using a reduction of the NP-complete problem Independent set problem (IS) to HALF-IS . Consider an instance of IS , which asks for an independent set $A \subset V$, $|A| = k$, for a graph $G(V, E)$, such that no two pair of vertices in A are adjacent to each other.

- (i) If $k = \frac{|V|}{2}$, IS reduces to HALF-IS.
- (ii) If $k < \frac{|V|}{2}$, then add m new nodes such that $k + m = (|V| + m)/2$, i.e., $m = |V| - 2k$. Note that the modified set of nodes V' has even number of nodes. Since the additional nodes are all disconnected from each other, they form a subset of independent set. Therefore, the new graph $G'(V', E')$ where $E' = E$ has an independent-set of size $\frac{|V'|}{2}$ if and only if $G(V, E)$ has an independent set of size k .
- (iii) If $k > \frac{|V|}{2}$, then again add $m = |V| - 2k$ new nodes to form the modified set of nodes V' . Connect these new nodes to all the other $|V| + m - 1$ nodes. Since these m new nodes are connected to every other node of them should belong to an independent set. . Therefore, the new graph $G'(V', E')$ has an independent-set of size $\frac{|V'|}{2}$ if and only if $G(V, E)$ has an independent set of size k .

Hence, any instance of IS $(G(V, E), k)$, can be reduced to an instance of HALF-IS $(G'(V', E'))$.

$$IS \leq_P HALF-IS.$$

$\text{HALF-IS} \in NP$ and $\text{HALF-IS} \in NP\text{-Hard} \Rightarrow \text{HALF-IS} \in NP\text{-complete}$.

3) 16 pts

A variant on the decision version of the subset sum problem is as follows: Given a set of n integer numbers $A = \{a_1, a_2, \dots, a_n\}$ and a target number t . Determine if there is a subset of numbers in A whose product is precisely t . That is, the output is *yes* or *no*. Describe an algorithm (and provide pseudo-code) to solve this problem, and analyze its complexity.

Solution:

Solution: *Use dynamic programming:*

Let $S[i,j]$ shows if there exists a subset of $\{a_1, a_2, \dots, a_i\}$ that add up to j , where $0 \leq j \leq t$.

$S[i,j]$ is true or false.

Initialize: $S[0,0] = \text{true}$; $S[0,j] = \text{false}$ if $j \neq 0$

Recurrence formula:

$S[i,j] = S[i-1,j] \text{ OR } S[i-1, j-a_i]$

Output is $S[n,t]$

Complexity is $O(tn)$

4) 16 pts

We've been put in charge of a phone hotline. We need to make sure that it's staffed by at least one volunteer at all times. Suppose we need to design a schedule that makes sure the hotline is staffed in the time interval $[0, h]$. Each volunteer i gives us an interval $[s_i, f_i]$ during which he or she is willing to work. We'd like to design an algorithm which determines the minimum number of volunteers needed to keep the hotline running. Design an efficient greedy algorithm for this problem that runs in time $O(n \log n)$ if there are n student volunteers. Prove that your algorithm is correct. You may assume that any time instance has at least one student who is willing to work for that time.

Solution:

Algorithm: Initially, select the student who can start at or before time 0 and whose finish time is the latest. For each subsequent volunteer, select the one whose start time is no later than the finish time of the last selected volunteer and whose finish time is the latest. Keep selecting volunteers sequentially in this way until the interval $[0, h]$ is covered.

The running time is $O(n \log n)$:

First, sort the start time of all the volunteers takes time $O(n \log n)$; then, searching and selecting the valid successive volunteers with the latest finish time takes $O(n)$ time in total (each volunteer is checked at most once).

Proof:

Let g_1, g_2, \dots, g_m be the sequence of volunteers we selected according to the greedy algorithm; let p_1, p_2, \dots, p_k be the sequence of selected volunteers of an optimal solution.

Clearly, for any feasible solution, we must have someone who can starts before or at time 0. So in the above two solutions: g_1 and p_1 must start at or before time 0.

According to our algorithm, we have $f_{g1} \geq f_{p1}$. By replacing p_1 by g_1 in the optimal solution, we get another solution: g_1, p_2, \dots, p_k , which uses k volunteers and cover the interval $[0, h]$ and therefore is another optimal solution.

Induction hypothesis: assume that our greedy solution is the same as an optimal solution up to the $r-1$ th selected volunteer, i.e., $g_1, \dots, g_{r-1}, p_r, \dots, p_k$ is an optimal solution. With the same argument: $f_{g_r} \geq f_{p_r}$ by replacing p_r by g_r in the optimal solution, we get another solution $g_1, \dots, g_r, p_{r+1}, \dots, p_k$, which is also optimal.

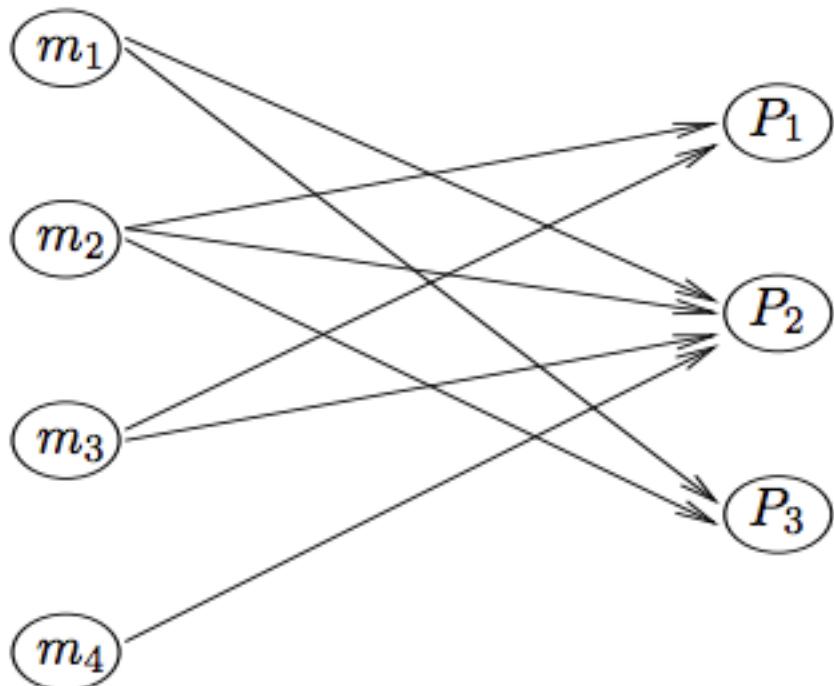
By induction: it follows that g_1, g_2, \dots, g_m is an optimal solution and $m=k$.

5) 16 pts

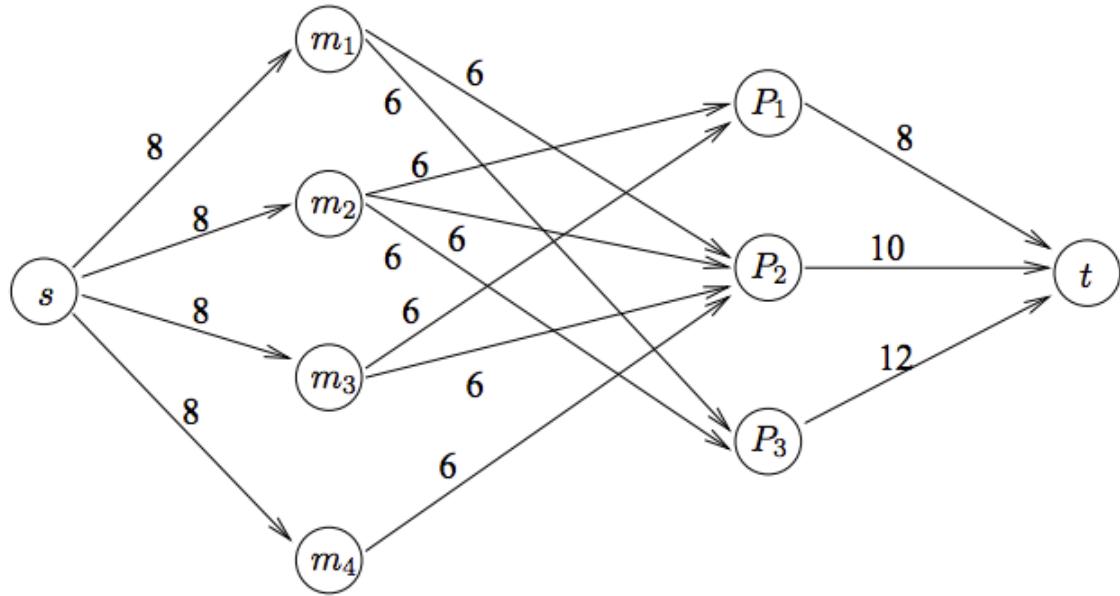
A software house has to handle 3 projects, P_1, P_2, P_3 , over the next 4 months. P_1 can only begin after month 1, and must be completed within month 3. P_2 and P_3 can begin at month 1, and must be completed, respectively, within month 4 and 2. The projects require, respectively, 8, 10, and 12 man-months. For each month, 8 engineers are available. Due to the internal structure of the company, at most 6 engineers can be working, at the same time, on the same project. Determine whether it is possible to complete the projects within the time constraints. Describe how to reduce this problem to the problem of finding a maximum flow in a flow network and justify your reduction.

Solution

We build a product network where months and projects are represented by, respectively, month-nodes m_1, m_2, m_3, m_4 and project-nodes P_1, P_2, P_3 . Each (i, j) edge denotes the possibility of allocating man-hours of month i to project j . For instance, since project P_1 can only begin after month 1 and must be completed before month 3, only arcs outgoing from m_2, m_3 are incident in P_1 .



We add to super nodes s, t , denoting the source and sink of the flow that represents the allocation of men-hours.



All edges outgoing from s have capacity 8, equivalent to the number of available engineers per month. All edges connecting month-nodes to project-nodes have capacity 6, as no more than 6 engineers can work on the same project in the same month. All edges incident in t have capacity equivalent to the number of man-months needed to complete the project. Since all capacities are integer, the maximum flow will be integer as well. To check whether all projects can be completed within the time limits, it suffices to check whether the network admits a feasible flow of value $8 + 10 + 12 = 30$.

6) 16 pts

There are n people and n jobs. You are given a cost matrix, C, where C[i][j] represents the cost of assigning person i to do job j. You want to assign all the jobs to people and also only one job to a person. You also need to minimize the total cost of your assignment. Can this problem be formulated as a linear program? If yes, give the linear programming formulation. If no, describe why it cannot be formulated as an LP and show how it can be reduced to an integer program.

Solution:

We need a 0,1 decision variable to solve the problem and therefore we need to formulate this as an integer program. Below is a formulation of integer program.

Let $x_{ij} = 1$, if job j is assigned to worker i.
= 0, if job j is not assigned to worker i.

Objective function: Minimize

$$\sum_{i=1}^m \sum_{j=1}^n c_{ij} x_{ij}$$

Constraints:

$$\sum_{i=1}^m x_{ij} = 1, \text{ for } j = 1, 2, \dots, n$$

$$\sum_{j=1}^n x_{ij} = 1, \text{ for } i = 1, 2, \dots, m$$

$$x_{ij} = 0 \text{ or } 1$$

Additional Space

Additional Space

CS570 Spring 2018: Analysis of Algorithms Exam III

	Points		Points
Problem 1	20	Problem 5	15
Problem 2	15	Problem 6	10
Problem 3	15	Problem 7	10
Problem 4	15		
Total: 100 Points			

Instructions:

1. This is a 2-hr exam. Closed book and notes
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.

1) 20 pts

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

[TRUE]

To prove that a problem X is NP-hard, it is sufficient to prove that SAT is polynomial time reducible to X.

[FALSE]

If a problem Y is polynomial time reducible to X, then a problem X is polynomial time reducible to Y.

[TRUE]

Every problem in NP can be solved in polynomial time by a nondeterministic Turing machine.

[TRUE]

Suppose that a divide and conquer algorithm reduces an instance of size n into 4 instances of size $n/5$ and spends $\Theta(n)$ time in the conquer steps. The algorithm runs in $\Theta(n)$ time.

[FALSE]

A linear program with all integer coefficients and constants must have an integer optimum solution.

[FALSE]

Let M be a spanning tree of a weighted graph $G=(V, E)$. The path in M between any two vertices must be a shortest path in G.

[TRUE]

A linear program can have an infinite number of optimal solutions.

[TRUE]

Suppose that a Las Vegas algorithm has expected running time $\Theta(n)$ on inputs of size n . Then there may still be an input on which it runs in time $\Omega(n^2)$.

[FALSE]

The total amortized cost of a sequence of n operations gives a lower bound on the total actual cost of the sequence.

[FALSE]

The maximum flow problem can be efficiently solved by dynamic programming.

2) 15 pts

Consider a uniform hash function h that evenly distributes n integer keys $\{0, 1, 2, \dots, n-1\}$ among m buckets, where $m < n$. Several keys may be mapped into the same bucket. The uniform distribution formally means that $\Pr(h(x)=h(y))=1/m$ for any $x, y \in \{0, 1, 2, \dots, n-1\}$. What is the expected number of total collisions? Namely how many distinct keys x and y do we have such that $h(x) = h(y)$?

Solution:

Let the indicator variable X_{ij} be 1 if $h(k_i) = h(k_j)$ and 0 otherwise for all $i \neq j$. Then let X be a random variable representing the total number of collisions. It can be calculated as the sum of all the X_{ij} 's: $X = \sum_{i < j} X_{ij}$

Now we take the expectation and use linearity of expectation to get:

$$E[X] = E\left[\sum_{i < j} X_{ij}\right] = \sum_{i < j} E[X_{ij}] = \sum_{i < j} P(h(k_i) = h(k_j)) = \sum_{i < j} \frac{1}{m} = \frac{n(n-1)}{2m}$$

Rubrics:

Describe choosing 2 keys from n keys: 10 pts.

Describe $\sum_{i < j} P(h(k_i) = h(k_j))$: 10 pts.

Common mistakes:

1. n/m : The mistake is that it thinks the expected number of collisions for each key is $1/m$. 7pts.
2. $2n/m$: Similar to 1. 7pts.
3. $(n-1)/m$: Similar to 1. 7pts.
4. $(m+1)/2$: Basically, no clue. 3 pts.
5. $n(n-1)/m$: Duplicates. 13 pts.
6. n^2/m : duplicated. 13 pts
7. $(1-(1-1/m)^{(n-1)}) * n$: wrong approach. 5 pts.
8. $n-m$: wrong approach. 3pts.
9. $\frac{1}{m} \sum_{i=0}^{n-1} i$: 15 pts
10. $n(n-1)/2$: The mistake is that it does not consider collision probability. 10 pts.
11. $N(n+1)/2m$: minor mistake. 13 pts.

3) 15 pts.

There are 4 production plants for making cars. Each plant works a little differently in terms of labor needed, materials, and pollution produced per car:

	Labor	Materials	Pollution
Plant 1	2	3	15
Plant 2	3	4	10
Plant 3	4	5	9
Plant 4	5	6	7

The goal is to maximize the number of cars produced under the following constraints:

- There are at most 3300 hours of labor
- There are at most 4000 units of material available.
- The level of pollution should not exceed 12000 units.
- Plant 3 must produce at least 400 cars.

Formulate a linear programming problem, using minimal number of variables, to solve the above task of maximizing the number of cars.

Solution:

We need four variables, to formulate the LP problem: x_1, x_2, x_3, x_4 , where x_i denotes the number of cars at plant-i.

Maximize $x_1 + x_2 + x_3 + x_4$

s.t.

$$x_i \geq 0 \text{ for all } i$$

$$x_3 \geq 400$$

$$2x_1 + 3x_2 + 4x_3 + 5x_4 \leq 3300$$

$$3x_1 + 4x_2 + 5x_3 + 6x_4 \leq 4000$$

$$15x_1 + 10x_2 + 9x_3 + 7x_4 \leq 12000$$

Rubric:

- Identifying 4 variables (2 pts)
- Correct objective function (3 pts)
- Each condition (2 pts)

4) 15 pts.

Given an undirected connected graph $G = (V, E)$ in which a certain number of tokens $t(v) \geq 1$ placed on each vertex v . You will now play the following game. You pick a vertex u that contains at least two tokens, remove two tokens from u and add one token to any one of adjacent vertices. The objective of the game is to perform a sequence of moves such that you are left with exactly one token in the whole graph. You are not allowed to pick a vertex with 0 or 1 token. Prove that the problem of finding such a sequence of moves is NP-complete by reduction from Hamiltonian Path.

Solution:

Construction: given a HP in G , we construct G' as follows. Traverse a HP in G and placed 2 tokens on the starting vertex and one token on each other vertex in the path.

Claim: G has a HP iff G' has a winning sequence.

->) by construction before the last move we will end up with a single vertex having two tokens on it. Making the last move, we will have exactly one token on the board.

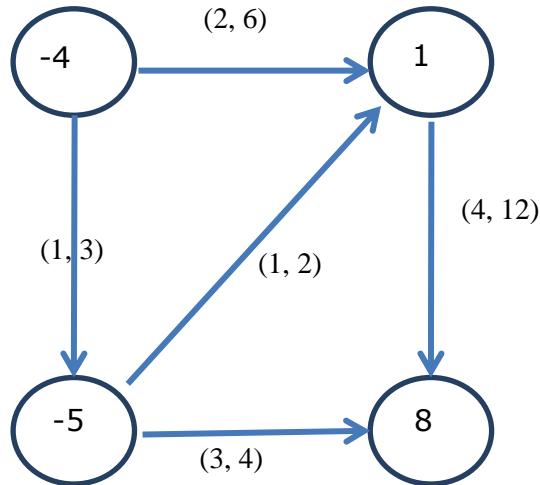
<-) since there is only one vertex with 2 tokens, we will start right there playing the game. Each next move is forced. When we finish the game, we get a sequence moves which represents a HP.

Rubrics:

- Didn't prove it is in NP: -5
- Didn't prove it is NP-hard: -10
- Assigned two tokens on one random vertex instead of the starting vertex of Hamiltonian path: -2 (Since it is a reduction from Hamiltonian path, not from Hamiltonian cycle, 2 tokens should be assigned on the starting vertex)
- Assigned wrong number of tokens on one vertex: -3
- Assigned wrong number of tokens on two vertices: -6
- Assigned wrong number of tokens on three or more vertices (considered as not a valid reduction from Hamiltonian path): -7
- Not a valid reduction from Hamiltonian path: -7

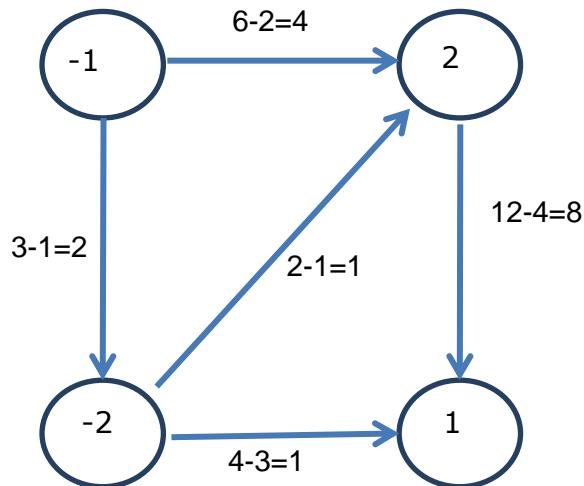
5) 15 pts.

In the network 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.



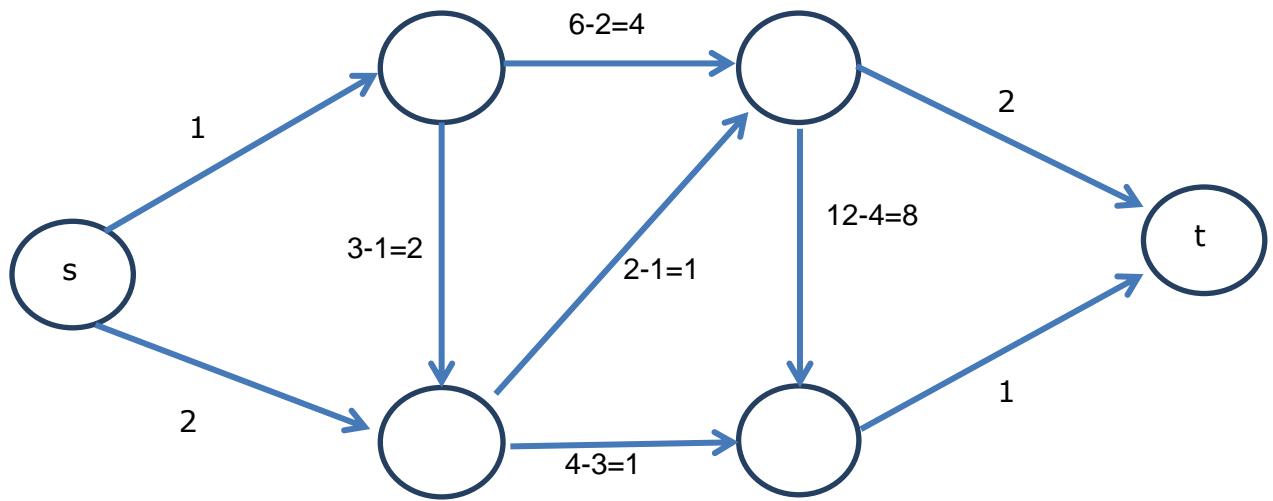
- a) Turn the circulation with lower bounds problem into a circulation problem without lower bounds (6 pts)

- **Each wrong number for nodes: -1**
- **Each wrong number for edges: -0.5**



b) Turn the circulation with demands problem into the max-flow problem (5 pts)

- **s and t nodes on right places: each has 0.5 point**
- **directions of edges from s and t: each direction 0.5 point**
- **values of edges from s and t: each value 0.5 point**

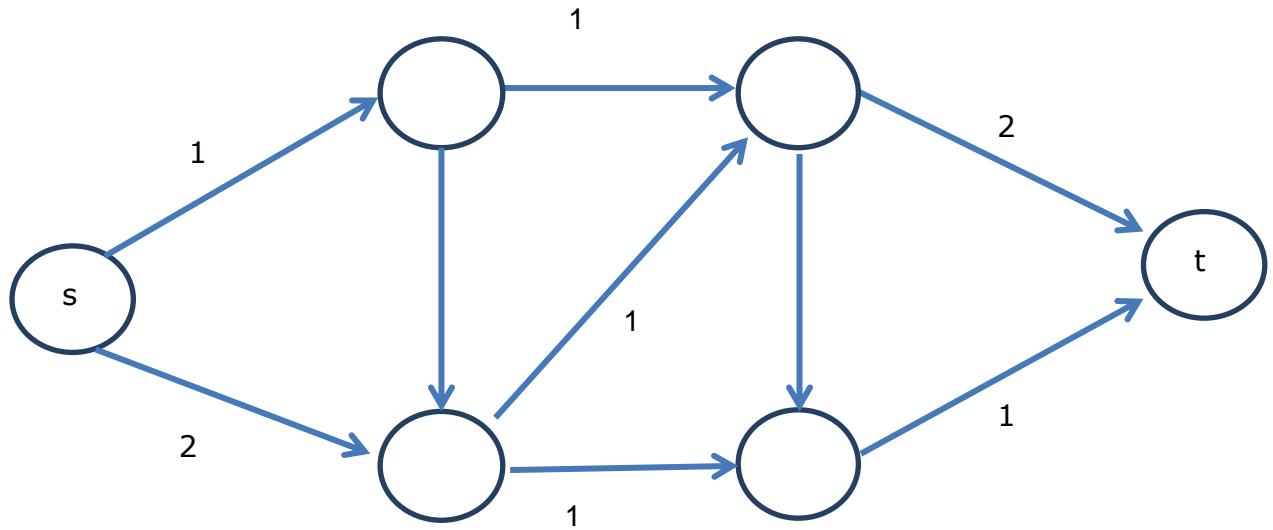


c) Does a feasible circulation exist? Explain your answer. (4 pts)

Solution.

Yes. There is a s-t flow $|f|=2+1=1+2=3$. It follows, there is a feasible circulation.

- **Explanation: 2 points.**
- **Correct Graph/flow: 2 points.**



6) 10 pts.

We wish to determine the most efficient way to deliver power to a network of cities.

Initially, we have n cities that are all connected. It costs $c_i \geq 0$ to open up a power plant at city i . It costs $r_{ij} \geq 0$ to build a cable between cities i and j . A city is said to have power if either it has a power plant, or it is connected by a series of cables to some other city with a power plant (in other words, the cables act as undirected edges). Devise an efficient algorithm for finding the *minimum cost* to power all the cities.

Solution

Consider a graph with the cities at the vertices and with weight r_{ij} on the (undirected) edge between city i and city j . Add a new vertex s and add an edge with weight c_i from s to the i th city, for each i . The minimum spanning tree on this graph gives the best solution (power stations need to be opened at those cities i for which (s,i) is part of the MST.)

Common Mistakes:

- Dynamic Programming: There does not exist a predefined order in which the nodes/edges will be processed and because of this you cannot define the problem based on smaller sub-problems and hence, none of the solutions which were based on Dynamic Programming were correct.
- Max-Flow/Min Cut: It was a surprise to see many ran a Max Flow algorithm to minimize the overall cost of edges. The minimum cut in a network has nothing to do with minimizing flow!! Furthermore, in this problem we want connectivity in the graph and edges on the min cut don't guarantee any level of connectivity.
- ILP: I can't recall anybody correctly formalizing the problem as in ILP. However, even a correct formulation is going to be NP-Hard and far from efficient. A maximum of 3 points was given to a CORRECT ILP solution.
- Dijkstra's algorithm is for finding the shortest path between a single point and every other node in the graph. The resulting Tree from running Dijkstra, is not necessarily an MST so it's not the correct approach to solve this problem.

7) 10 pts.

We want to break up the graph $G = (V, E)$ into two disjoint sets of vertices $V=A \cap B$, such that the number of edges between two partitions is as large as possible. This problem is called a max-cut problem. Consider the following algorithm:

- Start with an arbitrary cut C .
- While there exists a vertex v such that moving v from A to B increases the number of edges crossing cut C , move v to B and update C .

Prove that the algorithm is a 2-approximation.

Hint: after algorithm termination the number of edges in each of two partitions A and B cannot exceed the number of edges on the cut C .

Solution

Let $w(u,v) = 1$, there exists an edge between u and v , and 0 otherwise. Let

$$W = \sum_{(u,v) \in E} w(u,v)$$

By construction, for any node $u \in A$:

$$\sum_{v \in A} w(u, v) \leq \sum_{v \in B} w(u, v)$$

Here, the left hand side is all edges in partition A . The RHS – the crossing edges. If we sum up the above inequality for all $u \in A$, the LHS will contain the twice number of edges in A .

$$2 \sum_{(u,v) \in A} w(u, v) \leq \sum_{u \in A, v \in B} w(u, v) = C$$

We do the same for any node in B :

$$2 \sum_{(u,v) \in B} w(u, v) \leq \sum_{u \in A, v \in B} w(u, v) = C$$

Add them up

$$\sum_{(u,v) \in A} w(u, v) + \sum_{(u,v) \in B} w(u, v) \leq C$$

Next we add edges on the cut C to both sides.

$$\sum_{(u,v) \in A} w(u, v) + \sum_{(u,v) \in B} w(u, v) + \sum_{u \in A, v \in B} w(u, v) = W \leq 2C$$

Clearly the optimal solution $\text{OPT} \leq W$, thus $\text{OPT} \leq W \leq 2C$. It follows, $\text{OPT}/C \leq 2$.

A: # of edges within partition A
B: # of edges within partition B
C: # of edges between partition A and B

Correct direction

inCorrect direction

Prove that $A + B \leq C$

Prove that $A \leq C$ and $B \leq C$
Correct but useless, at most 1 points

Stage 1. 6 points

Prove that $\text{OPT} \leq 2*C$

Prove that $\text{OPT} \leq 3*C$
At most 2 points, if relations between A, B, C, E are used in reasonable way.

Stage 2. 4 points

- The solution can be divided into two parts. **Part 1** is worth 6 points, and **part 2** is worth 4 points.
 - Proving the hint, i.e. "numbers of edges in partition A and partition B cannot exceed the number of edges on the cut C". It is most important and challenging part of this problem.
 - Prove the algorithm is a 1/2-approximation.
- About part 1, here are two acceptable solutions and some solutions not acceptable:
 - If your solution is same to the official solution, using inequalities to rigorously prove the hint, it is perfect, **6 points**.
 - Here is a weaker version. A the end of algorithm, for any node u in the graph, without loss of generality assume it is in part A, the number of edges connecting it to nodes in part B (i.e. edges in the cut) is greater than or equal to the number of edges connecting it to other nodes in part A. So, the total number of edges in the cut is greater than or equal to the total number of edges within each part. This solution did not consider the important fact that both inter-partition and intra-partition edges are counted twice. **3 ~5 points according to the quality of statements.**
 - One incorrect proof: some answers declare that C is increasing and $A + B$ is decreasing when calling steps 2 of the algorithm. This is a correct but useless statement. **0 points**
 - YOU MUST PROVE THE HINT. SIMPLY STATING OR REPHRASING IT GET 0 POINTS IN THIS STEP.** Here is one example that cannot get credits: "According to the algorithm, we move one vertex from A to B if it can increase the number of edges in cut C. Then the number of edges in C is more than edges in partition A and partition B." **0 points**
 - "If move v from A to B can increase the number of edges cross cut C, it means v connect to more vertex in A than in B". It is rephrasing the algorithm, and failed to get the key point. If the above statement is changed to "If move v from A to B can increase the number of edges cross cut C, it means v connect to more vertex in different partition than in the same partition", it will get at least **3 points**.
- About part 2, the correct answer should use relation between OPT, E, A, B, C. Missing key inequalities like $\text{OPT} \leq E$, will lead to loss of grades.
- If the answer is incorrect, you may get some points from some steps that make sense. If the answer is on the incorrect direction in the above diagram:
 - Many students attempted to PROVE the $\text{edges_in_partition_A} \leq C$, and $\text{edges_in_partition_B} \leq C$, but not proving $\text{edges_in_partition_A} + \text{edges_in_partition_B} \leq C$. This is trivial according to step 2 of algorithm, and is not useful for proving the final statement. This will be treated as "correct but useless/irrelevant statements". **AT MOST 1 point.**
 - From above inequalities, $\text{OPT} \leq 3*C$ is a correct conclusion. It is different from the question, but since the logics in this stage is same, you can get at most **2 points out of 4**.

CS570
Analysis of Algorithms
Fall 2014
Exam III

Name: _____
Student ID: _____
Email: _____

Wednesday Evening Section

	Maximum	Received
Problem 1	20	
Problem 2	16	
Problem 3	16	
Problem 4	16	
Problem 5	16	
Problem 6	16	
Total	100	

Instructions:

1. This is a 2-hr exam. Closed book and notes
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.

1) 20 pts

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

[/FALSE]

All the NP-hard problems are in NP.

[/FALSE]

Given a weighted graph and two nodes, it is possible to list all shortest paths between these two nodes in polynomial time.

[TRUE/]

In the memory efficient implementation of Bellman-Ford, the number of iterations it takes to converge can vary depending on the order of nodes updated within an iteration

[/FALSE]

There is a feasible circulation with demands $\{d_v\}$ if $\sum_v d_v = 0$.

[/FALSE]

Not every decision problem in P has a polynomial time certifier.

[TRUE/]

If a problem can be reduced to linear programming in polynomial time then that problem is in P.

[/FALSE]

If we can prove that $P \neq NP$, then a problem $A \in P$ does not belong to NP.

[/FALSE]

If all capacities in a flow network are integers, then every maximum flow in the network is such that flow value on each edge is an integer.

[/FALSE]

In a dynamic programming formulation, the sub-problems must be mutually independent.

[~~TRUE~~/]

In the final residual graph constructed during the execution of the Ford–Fulkerson Algorithm, there's no path from sink to source.

2) 16 pts

In the Bipartite Directed Hamiltonian Cycle problem, we are given a bipartite directed graph $G = (V; E)$ and asked whether there is a simple cycle which visits every node exactly once. Note that this problem might potentially be easier than Directed Hamiltonian Cycle because it assumes a bipartite graph. Prove that Bipartite Directed Hamiltonian Cycle is in fact still NP-Complete.

Solution:

- i) This problem is NP. Given a candidate path, we just check the nodes along the given path one by one to see if every node in the network is visited exactly once and if each consecutive node pair along the path are joined by an edge.
- ii) To prove the problem is NP-complete, we reduce Directed Hamiltonian Cycle (DHC) problem to Bipartite Directed Hamiltonian Cycle (BDHC) problem.

Given an arbitrary directed graph G , we split each vertex v in G into two vertex v_{in} and v_{out} . Here v_{in} connects all the incoming edges to v in G ; v_{out} connects all the outgoing edges from v in G . Moreover, we connect one directed edge from v_{in} to v_{out} . After doing these operations for each node in G , we form a new graph G' .

Here G' is bipartite graph, because we can color each v_{in} “blue” and each v_{out} “red” without any coloring conflict.

If there is DHC in G , then there is a BDHC in G' . We can replace each node v on the DHC in G into consecutive nodes v_{in} and v_{out} , and (v_{in}, v_{out}) is an edge in G' . Then the new path is a BDHC in G' .

On the other hand, if there is BDHC in G' , then there is a DHC in G . Note that if v_{in} is on the BDHC, v_{out} must be the successive node on the BDHC. Then we can merge each node pair (v_{in}, v_{out}) on the BDHC in G' into node v and form a DHC in G .

In sum, G has DHC if and only if G' has a BDHC. This completes the reduction, and we confirm that the given problem is NP-complete

3) 16 pts

A tourism company is providing boat tours on a river with n consecutive segments. According to previous experience, the profit they can make by providing boat tours on segment i is known as a_i . Here a_i could be positive (they earn money), negative (they lose money), or zero. Because of the administration convenience, the local community of the river requires that the tourism company should do their boat tour business on a contiguous sequence of the river segments, i.e, if the company chooses segment i as the starting segment and segment j as the ending segment, all the segments in between should also be covered by the tour service, no matter whether the company will earn or lose money. The company's goal is to determine the starting segment and ending segment of boat tours along the river, such that their total profit can be maximized. Design an efficient algorithm to achieve this goal, and analyze its run time (Note that brute-force algorithm achieves $\Theta(n^2)$, so your algorithm must do better.)

Solution1:

Using dynamic programming:

Define $\text{OPT}(i)$ as the maximum total profit the company can get when running the boat tours on a contiguous sequence of segments starting at segment i .

Base case: $\text{OPT}(n) = a_n$.

Recursive relation: $\text{OPT}(i) = \max\{\text{OPT}(i+1)+a_i, a_i\}$, for $1 \leq i < n$

Algorithm:

For $i = n, \dots, 1$
 Compute $\text{OPT}(i)$.
End

Maximum profit = $\max_{i=1}^n \{\text{OPT}(i)\}$. Denote i^* as the starting index which achieves the maximum profit, then i^* is the optimal starting segment

For $i = i^*, \dots, n$
 If ($\text{OPT}(i) = a_i$)
 Set $j^* = i$;
 Break;
 End
End
The value j^* is the index of the optimal ending segment.

Complexity: Computing all the OPT values takes time $O(n)$, comparing all OPT values and find the maximum profit and the corresponding starting segment takes time $O(n)$. Finding the optimal ending segment takes time $O(n)$. In sum, the algorithm takes time $O(n)$

Remark: in this solution, we implicitly assume that the company must provide the boat tour, while providing no boat tour is not a choice. If you consider providing no boat tour also as a choice, it is also treated as a correct solution, however, the step of computing the maximum profit above solution should be modified as follows:

$$\text{Maximum profit} = \max\{0, \max_{\{i\}} \{\text{OPT}(i)\}\}.$$

Correspondingly, if 0 is the best result you can get, you need to claim that providing no boat tour is the best solution, and there is no need to confirm the starting segment or ending segment.

Solution 2 (outline):

Using divide and conquer.

- i) Divide operation: Divide the profit sequences of the n consecutive segments, denoted as $A[1,n]$, as two parts:
 $a_1, \dots, a_{n/2}$ and $a_{n/2+1}, \dots, a_n$, denoted as $A[1,n/2]$ and $A[n/2+1, n]$ respectively.
- ii) Merge operation:
 Suppose you successfully find the maximum profit in $A[1,n/2]$ and $A[n/2+1, n]$, denoted as $P_{left}(1,n)$, $P_{right}(1,n)$, and the corresponding contiguous sequence of segments, denoted as $B_{left}(1,n)$ and $B_{right}(1,n)$
 Then the optimal contiguous segment can be in one of the three cases:
 - a) $B_{left}(1,n)$
 - b) $B_{right}(1,n)$
 - c) An optimal contiguous segment sequence crossing $A[1,n/2]$ and $A[n/2+1, n]$, denoted as $B_{cross}(1,n)$.

For $B_{cross}(1,n)$, denote the corresponding profit as $P_{cross}(1,n)$. The method to confirm $B_{cross}(1,n)$ and $P_{cross}(1,n)$ is as follows:

- Starting from sequence $[a_{n/2}, a_{n/2+1}]$, compute the summation $S_{left}(1) = a_{n/2} + a_{n/2+1}$ as one candidate solution for $P_{cross}(1,n)$.
- Next, including $a_{n/2-1}$ into the above sequence as $[a_{n/2-1}, a_{n/2}, a_{n/2+1}]$, compute the summation of the three elements in the sequence as the second candidate solution: $S_{left}(2) = S_{left}(1) + a_{n/2-1}$.

- Keep including the element one by one to the left until a_1 is included, during each step, compute and record the summation values.
- Find $S_{\text{opt_left}} = \max_i \{S_{\text{left}}(i)\}$, record the corresponding index of the included element that achieves the maximum, say i_{cross^*} . Then i_{cross^*} is the optimal starting index for $B_{\text{cross}}(1,n)$;
- Starting from $[a_{i_{\text{cross}^*}}, \dots, a_{n/2+1}]$ includes the element to the right one by one until a_n is included, during each step, compute the summation values in the same way as in the left part, denote the value as $S_{\text{right}}(i)$.
- Find $P_{\text{cross}}(1,n) = \max_i \{S_{\text{right}}(i)\}$, record the corresponding index of the included element that achieves the maximum, say j_{cross^*} . Then j_{cross^*} is the optimal ending index for $B_{\text{cross}}(1,n)$;

Then

Maximum Profit = $\max \{P_{\text{left}}(1,n), P_{\text{right}}(1,n), P_{\text{cross}}(1,n)\}$, and the corresponding optimal starting index and ending index are the ones that achieve the maximum.

- Before doing step ii) for $A[1,n]$, recursively run the above procedure in each array $A[1,n/2]$ and $A[n/2+1, n]$ and so on.
- Termination condition: for $A[i,j]$, if $i=j$, return a_i as the maximum profit, return i as both the optimal starting and ending index in $A[i,j]$.

Complexity:

The Divide operation takes time $O(1)$.

The Merge operation takes time $O(n)$.

Define $T(n)$ as the running time,

$$T(n) = 2*T(n/2) + O(n)$$

The complexity is $O(n \log(n))$.

Remark: similar as in solution 1, considering no bout tour as a choice is also treated as a correct solution.

4) 16 pts

Consider the following matching problem. There are m students s_1, s_2, \dots, s_m and a set of n companies $C = \{c_1, c_2, \dots, c_n\}$. Each student can work for only one company, whereas company c_j can hire up to b_j students. Student s_i has a preferred set of companies $\Lambda_i \subseteq C$ at which he/she is willing to work. Your task is to find an assignment of students to companies such that all of the above constraints are satisfied and each student is assigned. Formulate this as a network flow problem and describe any subsequent steps necessary to arrive at the solution. Prove correctness.

Construction of flow network: Represent each student and each company as a separate node. Add one source node s and a sink node t for a total of $m + n + 2$ nodes. Add the following directed edges with capacities:

- i. $s \rightarrow s_i$ with capacity 1 unit, for each $1 \leq i \leq m$,
- ii. For each $1 \leq i \leq m$, $s_i \rightarrow c$ with capacity 1 unit for all nodes $c \in \Lambda_i$.
- iii. $c_j \rightarrow t$ with capacity b_j units, for each $1 \leq j \leq n$.

Constructing the solution: Run any max-flow algorithm on the above network to get the realization of edge flows under a max flow configuration (lets call it O for brevity). If an edge $s_i \rightarrow c_j$ shows a non-zero flow in this configuration, assign student s_i to company c_j . Repeat this process for each edge between the student nodes and the company nodes to get the final assignment.

Proof of correctness: We have to show that the solution so obtained satisfies all constraints of the problem.

- i. Since all outgoing edges from s_i are to the node set Λ_i , it is impossible for s_i to have a flow to a node outside Λ_i in configuration O .
- ii. Since the only outgoing edge from c_j is of capacity b_j , configuration O cannot have more than b_j incoming edges of non-zero flow to node c_j . Thus, not more than b_j students can get assigned to company c_j .
- iii. As s_i has a single incoming edge and multiple outgoing edges of capacity 1, configuration O cannot have more than one outgoing edge from s_i with non-zero flow. Hence, s_i can get assigned to at most one company.

We have proved that our mapping from configuration O to an assignment does not violate any of the constraints except possibly that all students might not be assigned. Note that this may happen in practice if it is impossible to assign all students while satisfying all of the given constraints. Thus, what we need to show is that if there exists a feasible assignment then configuration O will have each $s \rightarrow s_i$ edge carry a non-zero flow. Given a feasible assignment $\{(s_i, c_{\sigma(i)}), 1 \leq i \leq m\}$, by construction of the flow network, it is possible to set each edge $s_i \rightarrow c_{\sigma(i)}$ to carry 1 unit of flow. By feasibility of the assignment, company c_j gets no more than b_j incoming edges with non-zero flow, so the outgoing edge from c_j has enough capacity to carry away all incident flow on node c_j . Finally, since each s_i has an outgoing flow of 1 unit in

this assignment, all $s \rightarrow s_i$ edges can be set to have 1 unit of flow, completing the proof.

5) 16 pts

Consider a directed, weighted graph G where all edge weights are positive. You have one Star, which lets you change the weight of any one edge to zero. In other words, you may change the weight of any one edge to zero. Give an efficient algorithm using Dijkstra's algorithm to find a lowest-cost path between two vertices s and t , given that you may set one edge weight to zero. Note: you will receive 10 pts if your algorithm is efficient. You will receive full points (16 pts) if your algorithm has the same run time complexity as Dijkstra's algorithm.

Solution:

Use Dijkstra's algorithm to find the shortest paths from s to all other vertices. Reverse all edges and use Dijkstra to find the shortest paths from all vertices to t . Denote the shortest path from u to v by $u \rightsquigarrow v$, and its length by $\delta(u, v)$.

Now, try setting each edge to zero. For each edge $(u, v) \in E$, consider the path $s \rightsquigarrow u \rightarrow v \rightsquigarrow t$. If we set $w(u, v)$ to zero, the path length is $\delta(s, u) + \delta(v, t)$. Find the edge for which this length is minimized and set it to zero; the corresponding path $s \rightsquigarrow u \rightarrow v \rightsquigarrow t$ is the desired path. The algorithm requires two invocations of Dijkstra, and an additional $\Theta(E)$ time to iterate through the edges and find the optimal edge to take for free. Thus the total running time is the same as that of Dijkstra:

$O(E + V \lg V)$: based on using Fibonacci heap

$O(|V| + |E|) \log |V|$): based on using binary heap

6) 16 pts

You are given n rods; they are of length l_1, l_2, \dots, l_n , respectively. Our goal is to connect all the rods and form a single rod. The length after connecting two rods and the cost of connecting them are both equal to the sum of their lengths. Give an algorithm to minimize the cost of connecting them to form a single rod. State the complexity of your algorithm and prove that your algorithm is optimal.

1st interpretation: At each step we connect a single rod to the set of rods that are already connected together.

Of course, the solution is to sort rods by length and connect them in the order of increasing length. To prove this is optimal, you can assume that there is an optimal solution and compare our solution to the optimal solution: At each step our solution stays ahead of (or at least does no worse than) the optimal solution.

Sort Takes $O(n \log n)$

We can use mathematical induction to prove that we always stay ahead of the optimal solution.

Claim: at each step the cost of connecting rods in our solution is less than or equal to that of the other solution and the length of the rod after this connection is smaller than or equal in size to that of the other solution.

Base case: first two shortest rods – obviously this is the opt solution

Assuming that the cost of connecting k rods in our solution is less than or equal to that of another (optimal) solution, we can show that the cost of connecting the next ($k+1^{\text{st}}$) rod will be less than or equal to the cost of connecting the $k+1^{\text{st}}$ rod in the other solution:

Cost of the last connection in our solution = total length of all rods 1 to $k+1$ (ordered by length)

Cost of the last connection in the other solution = total length of all rods 1 to $k+1$ (not necessarily ordered by length)

It is obvious that the cost of our last connection is smaller or equal to the cost of the other connection because the only way to get the minimum total length of $k+1$ rods is to pick the shortest $k+1$ rods.

2nd interpretation: At each step we can connect any rod or set of already connected rods to another rod or set of already connected rods.

The solution is to keep connecting the smallest two rods or sets of already connected rods.

Implementation: Place all rods in a min heap with their key values representing their length. At each step extract two elements from the set and insert the combined rod back into the min heap.

This takes a total of $O(n \log n)$ time

Fact 1: the cost contribution of a rod i to the total assembly cost is $\text{Length}(i) * \text{Level}(i)$, where $\text{Level}(i)$ is the level at which the rod is first assembled with other rods (level 1=root level, level 2= level below root, etc.).

Proof: By observation of cost of assembly tree

Fact2: There is an optimal assembly tree of rods in which the two smallest rods are leaf nodes at the lowest level of the tree and the children of the same parent.

Proof: let's say the two smallest rods are not leaf nodes at the lowest level of the tree, using fact 1, we can swap these rods with two rods at the lowest level of the tree and thereby reducing the total cost of the assembly tree. If the two rods are leaf nodes at the lowest level but not children of the same parent we can swap two rods to make these rods children of the same parent without changing the total cost of the tree (again based on Fact 1).

Assume that there is an optimal assembly tree T^* and our solution produces tree T . We will show that our tree T is also optimal.

To do this, we apply fact 2 to T^* and move the smallest rods to the lowest level of the tree as children of the same parent—without increasing the cost of T^* . We then eliminate the two smallest rods at the bottom of the two trees(since these are the first two rods that are assembled in our algorithm) and assume that there is a new combined rod in place of their parent node. We will get two new trees $T^{*''}$ and T' , where

$$\begin{aligned}\text{Total assembly cost of } T^* &= \text{Total assembly cost of } T^{*''} + \text{total length of the two smallest rods} \\ \text{Total assembly cost of } T &= \text{Total assembly cost of } T' + \text{total length of the two smallest rods}\end{aligned}$$

Since the costs of the two new trees are reduced by the same amount we can now compare the cost of our new tree T' with the cost of the new optimal tree $T^{*''}$. And show that assembly tree T' is also optimal (as compared to the optimal tree $T^{*''}$).

To do this, we apply fact 2 recursively and place the two smallest rods in $T^{*''}$ at the lowest level of the tree and under the same parent node without increasing the cost of $T^{*''}$. These are the next two rods that are combined in our algorithm. We then eliminate these two rods in both trees T' and $T^{*''}$, etc.

By repeating the same steps (applying fact 2 and eliminating the two smallest rods from the optimal assembly tree and our tree) recursively, we will find an optimal solution that follows the same exact assembly sequence that is found in our algorithm. Therefore we can state that our algorithm also produces an optimal assembly tree.

Additional Space

CS570
Analysis of Algorithms
Fall 2014
Exam III

Name: _____
Student ID: _____
Email: _____

Wednesday Evening Section

	Maximum	Received
Problem 1	20	
Problem 2	16	
Problem 3	16	
Problem 4	16	
Problem 5	16	
Problem 6	16	
Total	100	

Instructions:

1. This is a 2-hr exam. Closed book and notes
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.

1) 20 pts

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

[/FALSE]

All the NP-hard problems are in NP.

[/FALSE]

Given a weighted graph and two nodes, it is possible to list all shortest paths between these two nodes in polynomial time.

[TRUE/]

In the memory efficient implementation of Bellman-Ford, the number of iterations it takes to converge can vary depending on the order of nodes updated within an iteration

[/FALSE]

There is a feasible circulation with demands $\{d_v\}$ if $\sum_v d_v = 0$.

[/FALSE]

Not every decision problem in P has a polynomial time certifier.

[TRUE/]

If a problem can be reduced to linear programming in polynomial time then that problem is in P.

[/FALSE]

If we can prove that $P \neq NP$, then a problem $A \in P$ does not belong to NP.

[/FALSE]

If all capacities in a flow network are integers, then every maximum flow in the network is such that flow value on each edge is an integer.

[/FALSE]

In a dynamic programming formulation, the sub-problems must be mutually independent.

[~~TRUE/~~]

In the final residual graph constructed during the execution of the Ford–Fulkerson Algorithm, there's no path from sink to source.

2) 16 pts

In the Bipartite Directed Hamiltonian Cycle problem, we are given a bipartite directed graph $G = (V; E)$ and asked whether there is a simple cycle which visits every node exactly once. Note that this problem might potentially be easier than Directed Hamiltonian Cycle because it assumes a bipartite graph. Prove that Bipartite Directed Hamiltonian Cycle is in fact still NP-Complete.

Solution:

- i) This problem is NP. Given a candidate path, we just check the nodes along the given path one by one to see if every node in the network is visited exactly once and if each consecutive node pair along the path are joined by an edge.
- ii) To prove the problem is NP-complete, we reduce Directed Hamiltonian Cycle (DHC) problem to Bipartite Directed Hamiltonian Cycle (BDHC) problem.

Given an arbitrary directed graph G , we split each vertex v in G into two vertex v_{in} and v_{out} . Here v_{in} connects all the incoming edges to v in G ; v_{out} connects all the outgoing edges from v in G . Moreover, we connect one directed edge from v_{in} to v_{out} . After doing these operations for each node in G , we form a new graph G' .

Here G' is bipartite graph, because we can color each v_{in} “blue” and each v_{out} “red” without any coloring conflict.

If there is DHC in G , then there is a BDHC in G' . We can replace each node v on the DHC in G into consecutive nodes v_{in} and v_{out} , and (v_{in}, v_{out}) is an edge in G' . Then the new path is a BDHC in G' .

On the other hand, if there is BDHC in G' , then there is a DHC in G . Note that if v_{in} is on the BDHC, v_{out} must be the successive node on the BDHC. Then we can merge each node pair (v_{in}, v_{out}) on the BDHC in G' into node v and form a DHC in G .

In sum, G has DHC if and only if G' has a BDHC. This completes the reduction, and we confirm that the given problem is NP-complete

3) 16 pts

A tourism company is providing boat tours on a river with n consecutive segments. According to previous experience, the profit they can make by providing boat tours on segment i is known as a_i . Here a_i could be positive (they earn money), negative (they lose money), or zero. Because of the administration convenience, the local community of the river requires that the tourism company should do their boat tour business on a contiguous sequence of the river segments, i.e, if the company chooses segment i as the starting segment and segment j as the ending segment, all the segments in between should also be covered by the tour service, no matter whether the company will earn or lose money. The company's goal is to determine the starting segment and ending segment of boat tours along the river, such that their total profit can be maximized. Design an efficient algorithm to achieve this goal, and analyze its run time (Note that brute-force algorithm achieves $\Theta(n^2)$, so your algorithm must do better.)

Solution1:

Using dynamic programming:

Define $\text{OPT}(i)$ as the maximum total profit the company can get when running the boat tours on a contiguous sequence of segments starting at segment i .

Base case: $\text{OPT}(n) = a_n$.

Recursive relation: $\text{OPT}(i) = \max\{\text{OPT}(i+1)+a_i, a_i\}$, for $1 \leq i < n$

Algorithm:

For $i = n, \dots, 1$
 Compute $\text{OPT}(i)$.
End

Maximum profit = $\max_{i=1}^n \{\text{OPT}(i)\}$. Denote i^* as the starting index which achieves the maximum profit, then i^* is the optimal starting segment

For $i = i^*, \dots, n$
 If ($\text{OPT}(i) = a_i$)
 Set $j^* = i$;
 Break;
 End
End
The value j^* is the index of the optimal ending segment.

Complexity: Computing all the OPT values takes time $O(n)$, comparing all OPT values and find the maximum profit and the corresponding starting segment takes time $O(n)$. Finding the optimal ending segment takes time $O(n)$. In sum, the algorithm takes time $O(n)$

Remark: in this solution, we implicitly assume that the company must provide the boat tour, while providing no boat tour is not a choice. If you consider providing no boat tour also as a choice, it is also treated as a correct solution, however, the step of computing the maximum profit above solution should be modified as follows:

$$\text{Maximum profit} = \max\{0, \max_{\{i\}} \{\text{OPT}(i)\}\}.$$

Correspondingly, if 0 is the best result you can get, you need to claim that providing no boat tour is the best solution, and there is no need to confirm the starting segment or ending segment.

Solution 2 (outline):

Using divide and conquer.

- i) Divide operation: Divide the profit sequences of the n consecutive segments, denoted as $A[1,n]$, as two parts:
 $a_1, \dots, a_{n/2}$ and $a_{n/2+1}, \dots, a_n$, denoted as $A[1,n/2]$ and $A[n/2+1, n]$ respectively.
- ii) Merge operation:
 Suppose you successfully find the maximum profit in $A[1,n/2]$ and $A[n/2+1, n]$, denoted as $P_{left}(1,n)$, $P_{right}(1,n)$, and the corresponding contiguous sequence of segments, denoted as $B_{left}(1,n)$ and $B_{right}(1,n)$
 Then the optimal contiguous segment can be in one of the three cases:
 - a) $B_{left}(1,n)$
 - b) $B_{right}(1,n)$
 - c) An optimal contiguous segment sequence crossing $A[1,n/2]$ and $A[n/2+1, n]$, denoted as $B_{cross}(1,n)$.

For $B_{cross}(1,n)$, denote the corresponding profit as $P_{cross}(1,n)$. The method to confirm $B_{cross}(1,n)$ and $P_{cross}(1,n)$ is as follows:

- Starting from sequence $[a_{n/2}, a_{n/2+1}]$, compute the summation $S_{left}(1) = a_{n/2} + a_{n/2+1}$ as one candidate solution for $P_{cross}(1,n)$.
- Next, including $a_{n/2-1}$ into the above sequence as $[a_{n/2-1}, a_{n/2}, a_{n/2+1}]$, compute the summation of the three elements in the sequence as the second candidate solution: $S_{left}(2) = S_{left}(1) + a_{n/2-1}$.

- Keep including the element one by one to the left until a_1 is included, during each step, compute and record the summation values.
- Find $S_{\text{opt_left}} = \max_i \{S_{\text{left}}(i)\}$, record the corresponding index of the included element that achieves the maximum, say i_{cross^*} . Then i_{cross^*} is the optimal starting index for $B_{\text{cross}}(1,n)$;
- Starting from $[a_{i_{\text{cross}^*}}, \dots, a_{n/2+1}]$ includes the element to the right one by one until a_n is included, during each step, compute the summation values in the same way as in the left part, denote the value as $S_{\text{right}}(i)$.
- Find $P_{\text{cross}}(1,n) = \max_i \{S_{\text{right}}(i)\}$, record the corresponding index of the included element that achieves the maximum, say j_{cross^*} . Then j_{cross^*} is the optimal ending index for $B_{\text{cross}}(1,n)$;

Then

Maximum Profit = $\max \{P_{\text{left}}(1,n), P_{\text{right}}(1,n), P_{\text{cross}}(1,n)\}$, and the corresponding optimal starting index and ending index are the ones that achieve the maximum.

- Before doing step ii) for $A[1,n]$, recursively run the above procedure in each array $A[1,n/2]$ and $A[n/2+1, n]$ and so on.
- Termination condition: for $A[i,j]$, if $i=j$, return a_i as the maximum profit, return i as both the optimal starting and ending index in $A[i,j]$.

Complexity:

The Divide operation takes time $O(1)$.

The Merge operation takes time $O(n)$.

Define $T(n)$ as the running time,

$$T(n) = 2*T(n/2) + O(n)$$

The complexity is $O(n \log(n))$.

Remark: similar as in solution 1, considering no bout tour as a choice is also treated as a correct solution.

4) 16 pts

Consider the following matching problem. There are m students s_1, s_2, \dots, s_m and a set of n companies $C = \{c_1, c_2, \dots, c_n\}$. Each student can work for only one company, whereas company c_j can hire up to b_j students. Student s_i has a preferred set of companies $\Lambda_i \subseteq C$ at which he/she is willing to work. Your task is to find an assignment of students to companies such that all of the above constraints are satisfied and each student is assigned. Formulate this as a network flow problem and describe any subsequent steps necessary to arrive at the solution. Prove correctness.

Construction of flow network: Represent each student and each company as a separate node. Add one source node s and a sink node t for a total of $m + n + 2$ nodes. Add the following directed edges with capacities:

- i. $s \rightarrow s_i$ with capacity 1 unit, for each $1 \leq i \leq m$,
- ii. For each $1 \leq i \leq m$, $s_i \rightarrow c$ with capacity 1 unit for all nodes $c \in \Lambda_i$.
- iii. $c_j \rightarrow t$ with capacity b_j units, for each $1 \leq j \leq n$.

Constructing the solution: Run any max-flow algorithm on the above network to get the realization of edge flows under a max flow configuration (lets call it O for brevity). If an edge $s_i \rightarrow c_j$ shows a non-zero flow in this configuration, assign student s_i to company c_j . Repeat this process for each edge between the student nodes and the company nodes to get the final assignment.

Proof of correctness: We have to show that the solution so obtained satisfies all constraints of the problem.

- i. Since all outgoing edges from s_i are to the node set Λ_i , it is impossible for s_i to have a flow to a node outside Λ_i in configuration O .
- ii. Since the only outgoing edge from c_j is of capacity b_j , configuration O cannot have more than b_j incoming edges of non-zero flow to node c_j . Thus, not more than b_j students can get assigned to company c_j .
- iii. As s_i has a single incoming edge and multiple outgoing edges of capacity 1, configuration O cannot have more than one outgoing edge from s_i with non-zero flow. Hence, s_i can get assigned to at most one company.

We have proved that our mapping from configuration O to an assignment does not violate any of the constraints except possibly that all students might not be assigned. Note that this may happen in practice if it is impossible to assign all students while satisfying all of the given constraints. Thus, what we need to show is that if there exists a feasible assignment then configuration O will have each $s \rightarrow s_i$ edge carry a non-zero flow. Given a feasible assignment $\{(s_i, c_{\sigma(i)}), 1 \leq i \leq m\}$, by construction of the flow network, it is possible to set each edge $s_i \rightarrow c_{\sigma(i)}$ to carry 1 unit of flow. By feasibility of the assignment, company c_j gets no more than b_j incoming edges with non-zero flow, so the outgoing edge from c_j has enough capacity to carry away all incident flow on node c_j . Finally, since each s_i has an outgoing flow of 1 unit in

this assignment, all $s \rightarrow s_i$ edges can be set to have 1 unit of flow, completing the proof.

5) 16 pts

Consider a directed, weighted graph G where all edge weights are positive. You have one Star, which lets you change the weight of any one edge to zero. In other words, you may change the weight of any one edge to zero. Give an efficient algorithm using Dijkstra's algorithm to find a lowest-cost path between two vertices s and t , given that you may set one edge weight to zero. Note: you will receive 10 pts if your algorithm is efficient. You will receive full points (16 pts) if your algorithm has the same run time complexity as Dijkstra's algorithm.

Solution:

Use Dijkstra's algorithm to find the shortest paths from s to all other vertices. Reverse all edges and use Dijkstra to find the shortest paths from all vertices to t . Denote the shortest path from u to v by $u \rightsquigarrow v$, and its length by $\delta(u, v)$.

Now, try setting each edge to zero. For each edge $(u, v) \in E$, consider the path $s \rightsquigarrow u \rightarrow v \rightsquigarrow t$. If we set $w(u, v)$ to zero, the path length is $\delta(s, u) + \delta(v, t)$. Find the edge for which this length is minimized and set it to zero; the corresponding path $s \rightsquigarrow u \rightarrow v \rightsquigarrow t$ is the desired path. The algorithm requires two invocations of Dijkstra, and an additional $\Theta(E)$ time to iterate through the edges and find the optimal edge to take for free. Thus the total running time is the same as that of Dijkstra:

$O(E + V \lg V)$: based on using Fibonacci heap

$O(|V| + |E|) \log |V|$): based on using binary heap

6) 16 pts

You are given n rods; they are of length l_1, l_2, \dots, l_n , respectively. Our goal is to connect all the rods and form a single rod. The length after connecting two rods and the cost of connecting them are both equal to the sum of their lengths. Give an algorithm to minimize the cost of connecting them to form a single rod. State the complexity of your algorithm and prove that your algorithm is optimal.

1st interpretation: At each step we connect a single rod to the set of rods that are already connected together.

Of course, the solution is to sort rods by length and connect them in the order of increasing length. To prove this is optimal, you can assume that there is an optimal solution and compare our solution to the optimal solution: At each step our solution stays ahead of (or at least does no worse than) the optimal solution.

Sort Takes $O(n \log n)$

We can use mathematical induction to prove that we always stay ahead of the optimal solution.

Claim: at each step the cost of connecting rods in our solution is less than or equal to that of the other solution and the length of the rod after this connection is smaller than or equal in size to that of the other solution.

Base case: first two shortest rods – obviously this is the opt solution

Assuming that the cost of connecting k rods in our solution is less than or equal to that of another (optimal) solution, we can show that the cost of connecting the next ($k+1^{\text{st}}$) rod will be less than or equal to the cost of connecting the $k+1^{\text{st}}$ rod in the other solution:

Cost of the last connection in our solution = total length of all rods 1 to $k+1$ (ordered by length)

Cost of the last connection in the other solution = total length of all rods 1 to $k+1$ (not necessarily ordered by length)

It is obvious that the cost of our last connection is smaller or equal to the cost of the other connection because the only way to get the minimum total length of $k+1$ rods is to pick the shortest $k+1$ rods.

2nd interpretation: At each step we can connect any rod or set of already connected rods to another rod or set of already connected rods.

The solution is to keep connecting the smallest two rods or sets of already connected rods.

Implementation: Place all rods in a min heap with their key values representing their length. At each step extract two elements from the set and insert the combined rod back into the min heap.

This takes a total of $O(n \log n)$ time

Fact 1: the cost contribution of a rod i to the total assembly cost is $\text{Length}(i) * \text{Level}(i)$, where $\text{Level}(i)$ is the level at which the rod is first assembled with other rods (level 1=root level, level 2= level below root, etc.).

Proof: By observation of cost of assembly tree

Fact2: There is an optimal assembly tree of rods in which the two smallest rods are leaf nodes at the lowest level of the tree and the children of the same parent.

Proof: let's say the two smallest rods are not leaf nodes at the lowest level of the tree, using fact 1, we can swap these rods with two rods at the lowest level of the tree and thereby reducing the total cost of the assembly tree. If the two rods are leaf nodes at the lowest level but not children of the same parent we can swap two rods to make these rods children of the same parent without changing the total cost of the tree (again based on Fact 1).

Assume that there is an optimal assembly tree T^* and our solution produces tree T . We will show that our tree T is also optimal.

To do this, we apply fact 2 to T^* and move the smallest rods to the lowest level of the tree as children of the same parent—without increasing the cost of T^* . We then eliminate the two smallest rods at the bottom of the two trees(since these are the first two rods that are assembled in our algorithm) and assume that there is a new combined rod in place of their parent node. We will get two new trees T^{**} and T' , where

$$\begin{aligned}\text{Total assembly cost of } T^* &= \text{Total assembly cost of } T^{**} + \text{total length of the two smallest rods} \\ \text{Total assembly cost of } T &= \text{Total assembly cost of } T' + \text{total length of the two smallest rods}\end{aligned}$$

Since the costs of the two new trees are reduced by the same amount we can now compare the cost of our new tree T' with the cost of the new optimal tree T^{**} . And show that assembly tree T' is also optimal (as compared to the optimal tree T^{**}).

To do this, we apply fact 2 recursively and place the two smallest rods in T^{**} at the lowest level of the tree and under the same parent node without increasing the cost of T^{**} . These are the next two rods that are combined in our algorithm. We then eliminate these two rods in both trees T' and T^{**} , etc.

By repeating the same steps (applying fact 2 and eliminating the two smallest rods from the optimal assembly tree and our tree) recursively, we will find an optimal solution that follows the same exact assembly sequence that is found in our algorithm. Therefore we can state that our algorithm also produces an optimal assembly tree.

Additional Space

CS570 Fall 2018: Analysis of Algorithms Exam I

	Points		Points
Problem 1	20	Problem 5	20
Problem 2	20	Problem 6	8
Problem 3	16		
Problem 4	16		
	Total	100	

Instructions:

1. This is a 2-hr exam. Closed book and notes
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.

1) 20 pts

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

[**TRUE/FALSE**]

Dynamic programming only works on problems with non-overlapping subproblems.

[**TRUE/FALSE**]

If a flow in a network has a cycle, this flow is not a valid flow.

[**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.

[**TRUE/FALSE**]

A dynamic programming solution always explores the entire search space for all possible solutions.

[**TRUE/FALSE**]

Decreasing the capacity of an edge that belongs to a min cut in a flow network always results in decreasing the maximum flow.

[**TRUE/FALSE**]

Suppose f is a flow of value 100 from s to t in a flow network G . The capacity of the

minimum $s - t$ cut in G is equal to 100.

[**TRUE/FALSE**]

One can **efficiently** find the maximum number of edge disjoint paths from s to t in a directed graph by reducing the problem to max flow and solving it using the Ford-Fulkerson algorithm.

[**TRUE/FALSE**]

If all edges in a graph have capacity 1, then Ford-Fulkerson runs in linear time.

[**TRUE/FALSE**]

Given a flow network where all the edge capacities are even integers, the algorithm will require at most $C/2$ iterations, where C is the total capacity leaving the source s.

[**TRUE/FALSE**]

By combining divide and conquer with dynamic programming we were able to reduce the space requirements for our sequence alignment solution at the cost of increasing the computational complexity of our solution.

2) 20 pts.

Suppose that we have a set of students S_1, \dots, S_s , a set of projects P_1, \dots, P_p , a set of teachers T_1, \dots, T_t . A student is interested in a subset of projects. Each project p_i has an upper bound $K(P_i)$ on the number of the students that can work on it. A teacher T_i is willing to be the leader of a subset of the projects. Furthermore, he/she has an upper bound $H(T_i)$ on the number of students he/she is willing to supervise in total. We assume that no two teachers are willing to supervise the same project. The decision problem here is whether there is really a feasible assignment without violating any of the constraints in K and in H .

a) Solve the decision problem using network flow techniques. (15 pts)

The source node is connected to each student with the capacity of 1. Each student is connected to his/her desired projects with a capacity of 1.

Each project is connected to teachers who are willing to supervise it with a capacity of $K(P_i)$. Each teacher is connected to the sink node with the capacity of $H(T_i)$. If there is a max flow that is equal to the number of students, there is a solution to the problem. You can also convert the order of nodes and edges (source->teachers->projects->students->sink) as long as the edges are correct. You can have 2 sets of nodes for projects (inputs and outputs) as long as the output nodes do not constrain the flow.

b) Prove the correctness of your algorithm. (5 pts)

We need to show that

A – If there is a feasible assignment of students and teachers to projects, we can find a max flow of value s (# of students) in G

B – If we find a max flow of value s in G , we can find a feasible assignment of students and teachers to projects.

Rubric:

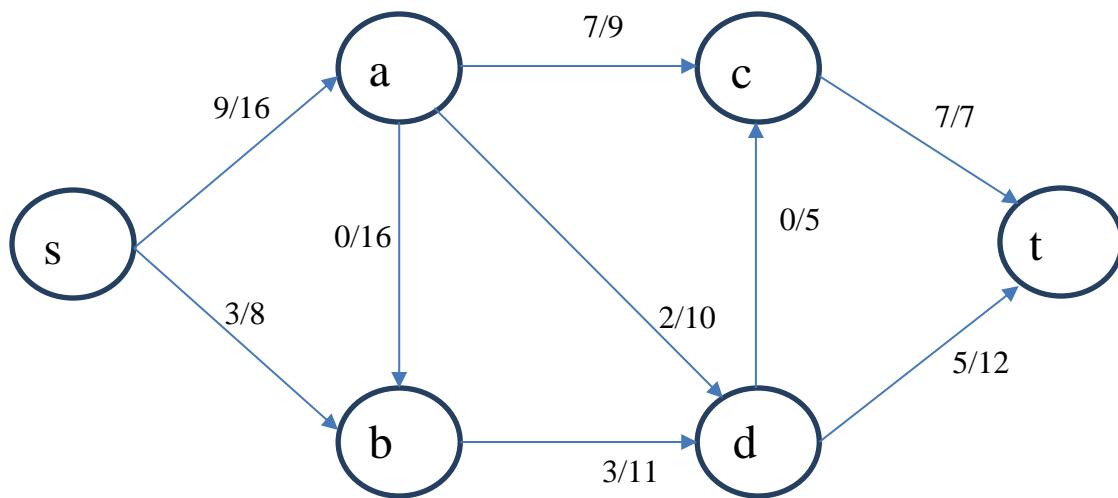
- 1) incorrect/missing nodes (incorrect nodes * -3)
- 2) incorrect/missing edges (incorrect edges * -2)
- 3) extra demand values (extra demands * -1)
- 4) not mentioning that max flow = number of students (-3)
- 5) if the proof is only provided for one side (-2)
- 6) if proof is not sufficient (-1)

3) 16 pts.

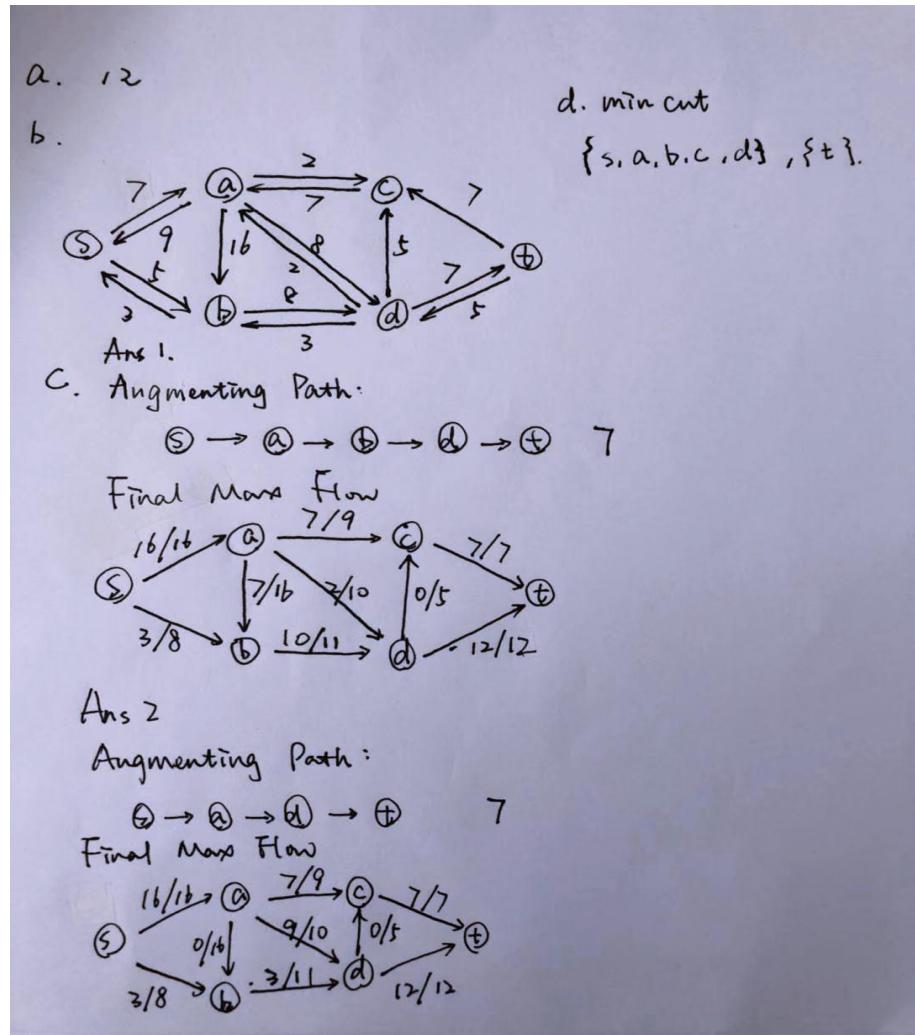
Consider the below flow-network, for which an s-t flow has been computed. The numbers x/y on each edge shows that the capacity of the edge is equal to y , and the flow sent on the edge is equal to x .

- What is the current value of flow? (2 pts)
- Draw the residual graph for the corresponding flow-network. (6 pts)
- Calculate the maximum flow in the graph using the Ford-Fulkerson algorithm. You need to show the augmenting path at each step and final max flow. (6 pts)
- Show the min cut found by the max flow you found in part c. (2 pts)

Note: extra space provided for this problem on next page



Solution:



Rubric:

- a) Incorrect value -> 0 point
- b) Every incorrect/missing edge or capacity -> -1 point
- c) Every possible flow which has a capacity of 19 is correct.
Answers without correct value(other than) 19 will get 0 point.
Answers with correct value/final max flow/final residual graph without augmenting path/detail steps will get 3 points
Answers with correct value/final max flow/final residual graph with augmenting path&detail steps will get 6 points
Answers with a correct augmenting path but incomplete max flow will get 1 point.
- d) Incorrect set -> 0 point

4) 16 pts

A symmetric sequence or palindrome is a sequence of characters that is equal to its own reversal; e.g. the phrase “a man a plan a canal panama” forms a palindrome (ignoring the spaces between words).

a) Describe how to use the **sequence alignment algorithm** (described in class) to find a longest symmetric subsequence (longest embedded palindrome) of a given sequence. (14 pts)

Example: CTGACCTAC ‘s longest embedded palindromes are CTCCTC and CACCAC

Solution: given the sequence S , reverse the string to form S^r . Then find the longest common substring between S and S^r by setting all mismatch costs to ∞ and $\delta=1$ (or anything greater than zero) .

b) What is the run time complexity of your solution? (2 pts)

$O(n^2)$

Rubric 4a)

- 10 points for reversing the string and using the sequence alignment algorithm
 - If the recurrence is given without explicitly mentioning the sequence alignment algorithm is fine.
 - Recurrence is discussed in class
- 2 points for allocating the correct mismatch value.
- 2 points for allocating the correct gap score.
- Any other answer which does not use sequence alignment algorithm is wrong. **The question explicitly asks to use the sequence alignment algorithm.** (0 to 2 points)
 - 2 points for correctly mentioning any other algorithm to get palindrome within a string
- Wrong answers
 - Dividing the string into two and reversing the second half.
 - Will not work, as the entire palindrome can be located within the first or the second half
 - Aligning the original string with the copy of the same string
 - This will return the original string and not the palindrome
 - Any other algorithm which finds the palindrome in a string

Rubric 4b)

2 points

0 points for any other answer

5) 20 pts

Let's say you have a dice with m sides numbered from 1 to m , and you are throwing it n times. If the side with number i is facing up you will receive i points. Consider the variable Z which is the summation of your points when you throw the dice n times. We want to find the number of ways we can end up with the summation Z after n throws. We want to do this using dynamic programming.

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

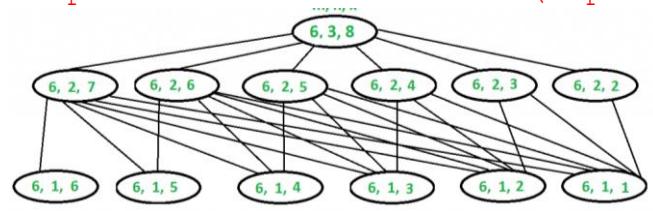
$\text{OPT}(n, Z) = \text{number of ways to get the sum } Z \text{ after } n \text{ rolls of dice.}$

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

$\text{OPT}(n, Z) = \sum_{i=1 \text{ to } m} \text{OPT}(n-1, Z-i)$

Example: Given, $m=6$, Calculate: $\text{OPT}(3, 8)$

Subproblems are as follows (represented as (m, n, Z)):



Rubric:

-6 if not summing over m

-4 if summing over m but OPT is somewhat incorrect.

c) Using the recurrence formula in part b, write pseudocode to compute the number of ways to obtain the value SUM . (6 pts)

Make sure you have initial values properly assigned. (2 pts)

```

OPT(1, i) = 1 for i=1 to MIN(m, SUM)
OPT(1, i) = 0, for i=MIN(m, SUM) to MAX(m, SUM)
For i = 2 to n
    For j = 1 to SUM
        OPT(i, j) = 0
        For k = 1 to MIN (m, i)
            OPT(i, j) = OPT(i, j) + OPT(i-1, j-k)
        Endfor
    Endfor
Endfor
  
```

Return OPT(n, SUM)

Example, Given, m=6, Calculate: OPT(3, 8)
OPT Table looks as follows:

0	0	0	0	0	0	0	0	0
0	1	1	1	1	1	1	0	0
0	0	1	2	3	4	5	6	5
0	0	0	1	3	6	10	15	21

Alternatively, you can initialize $\text{OPT}(0,0) = 1$ and start the iteration of i from 1.

Rubric:

- 1 for each initialization missed.
- 6 for any incorrect answer. (e.g incorrect recurrence relation, incorrect loops, etc)
i.e. no partial grading for any pseudocode producing incorrect answer.

- d) Compute the runtime of the algorithm described in part c and state whether your solution runs in polynomial time or not (2 pts)

Time Complexity: $O(m * n * Z)$ where m is number of sides, n is the number of times we roll the dice, and Z is given sum. This is pseudo polynomial since the complexity depends on the numerical value of input terms.

Rubric:

- Not graded if part c is incorrect.
- 2 if stated as polynomial time.
- 2 if incorrect complexity equation.

6) 8 pts

- a) Given a flow network $G=(V, E)$ with integer edge capacities, a max flow f in G , and a specific edge e in E , design a linear time algorithm that determines whether or not e belongs to a min cut. (6 pts)

Solution:

Reduce capacity of e by 1 unit. - O(1)

Find an s - t path containing e and subtract 1 unit of s - t flow on that path. For this, considering e 's adjacent nodes to be (u, v) , you can run BFS from source to find an s - u path and BFS from v to find an v - t path which gives you an s - u - v - t path crossing e . - $O(|V|+|E|)$

Then construct the new residual graph G_f - $O(|E|)$

If there is an augmenting s - t path in G_f then e does not belong to a min cut, otherwise it does.

- b) Describe why your algorithm runs in linear time.

You can find this using BFS from source(only one iteration of the FF algorithm) - $O(|V| + |E|)$

All the above steps are linear in the size of input, therefore the entire algorithm is linear.

Rubric:

If providing the complete algorithm: full points

- If not decreasing the s - t flow by 1: -1 points
- If running the entire Ford-Fulkerson algorithm to find the flow: -2 points
- If providing the overall idea without description of implementation steps: -3 or -4 points depending on your description

If the provided algorithm is not scalable to graphs having multiple(more than two) min-cuts(for example using BFS to find reachable/unreachable nodes from source/sink): -4 points

- Incomplete or incorrect description of the above algorithm (-1 or -2 additional points depending on your description)

If removing the edge and finding max-flow: -3 or -4 points depending on your description (since this algorithm doesn't run in linear time)

If only considering saturated edges to check for min-cut edges: -7 points

Adding the capacity of e and checking the flow: no points (this approach is not checking for min-cut)

Checking all possible cuts in the graph: no points (this takes exponential time)

Other algorithms: no points

	Points		Points
Problem 1	20	Problem 5	20
Problem 2	6	Problem 6	14
Problem 3	20		
Problem 4	20		
	Total	100	

Instructions:

1. This is a 2-hr exam. Closed book and notes
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.

1) 20 pts

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

[**TRUE/FALSE**]

The Ford-Fulkerson algorithm can be used to find the maximum flow through a graph with cycles.

[**TRUE/FALSE**]

In a flow network where all edge capacities are unique, the min cut will be unique.

[**TRUE/FALSE**]

A Dynamic Programming algorithm with n^3 unique sub-problems will run in $O(n^4)$ time.

[**TRUE/FALSE**]

It is possible for a dynamic programming algorithm to have exponential running time.

[**TRUE/FALSE**]

Let G be a weighted directed graph with exactly one source s and exactly one sink t . Let (A, B) be a maximum cut in G , that is, A and B are disjoint sets whose union is V , $s \in A$, $t \in B$, and the sum of the weights of all edges from A to B is the maximum for any two such sets. Now let H be the weighted directed graph obtained by adding 1 to the weight of each edge in G . Then (A, B) must still be a maximum cut in H .

[**TRUE/FALSE**]

The recurrence $OPT(j) = \max_{1 \leq i \leq j} \{A[i] - B[j-i] + OPT(j-i)\}$ where A and B are fixed Input arrays, will lead to an $O(n)$ dynamic programming algorithm.

[**TRUE/FALSE**]

In a graph with more than one minimum cut, if we take any edge e that is part of the (A, B) minimum cut and increase the capacity of that edge by any integer $k \geq 1$, then (A, B) will no longer be a minimum cut.

[**TRUE/FALSE**]

The Ford-Fulkerson algorithm (without scaling) can be used to efficiently solve the maximum matching problem in a bipartite graph.

[**TRUE/FALSE**]

The space efficient version of the sequence alignment algorithm (which uses a combination of divide and conquer and dynamic programming) has the same running time complexity as the basic dynamic programming algorithm for this problem.

[**TRUE/FALSE**]

Suppose, increasing the capacity of edge e by 1, increases value of max flow f by 1. Then it must be that $f(e) = \text{capacity}(e)$.

2) 6 pts

Let $G = (V, E)$ be a flow network with source s , sink t , and integer capacities.

Suppose that we are given a maximum flow f in G . Now, we increase the capacity of a single edge e by 1.

a) If $f(e) < \text{Capacity}(e)$, can we say that f will still be a maximum flow in G ? (2pts)

Yes.

Remarks: this part is not graded since there was confusion about whether the capacity was before the flow or after. So “No” is also accepted as correct answer.

b) If $f(e) = \text{Capacity}(e)$, can we say that f will no longer be a maximum flow in G ? (2 pts)

No.

c) Give an $O(V + E)$ -time algorithm to find a new maximum flow in G . (2 pts)

Find new G_f and see if there is a path from s to t . If there is, do one more step of augmentation.

Rubric: All-or-nothing. No partial credit.

3) 20 pts

We are given a string $S = s_1s_2s_3\dots s_n$, and we want to insert some characters such that the resulting string is a palindrome. Give an efficient dynamic programming algorithm to determine the minimum number of insertions.

Definition: A palindrome is a string that reads the same forward and backward such as ACCBCCA.

Example: Input: ACB Output: 2 (ACBCA)

a) Define (in plain English) subproblems to be solved. (5 pts)

$\text{opt}(i,j) = \text{minimum number of insertions to make substring } s[i:j] \text{ a palindrome.}$

grading rubric:

3pts if using $\text{opt}(i)$ to represent the number of insertions to make substring $s[0:i]$ a palindrome

reverse the string is ok and is graded in a similar scheme.

0pts everything else.

b) Write the recurrence relation for subproblems. (7 pts)

if $s[i] == s[j]: \text{opt}(i,j) = \text{opt}(i+1, j-1)$
else: $\text{opt}(i,j) = \min(\text{opt}(i+1,j), \text{opt}(i, j-1)) + 1$

grading rubric:

$s[i] == s[j]: 2\text{pts}$

$\text{opt}(i+1, j): 2\text{pts}$

$\text{opt}(i, j-1): 2\text{pts}$

+1: 7pts

reverse the string and compare are ok and is graded in a similar scheme. use sigma (or other constants) to represent mismatch penalty is ok.

c) Using the recurrence formula in part b, write pseudocode to compute the minimum number of insertions. (5 pts)

Make sure you have initial values properly assigned. (3 pts)

for i=1 to n:

 for j=1 to n:

$\text{opt}(i,j) = 0$

```

for j=1 to n:
    for i = j-1 to 1:
        if si == sj:
            if i < j-1:
                opt(i,j) =opt(i+1, j-1)
            else:
                opt(i,j) = 0
        else:
            opt(i,j) = min(opt(i+1,j), opt(i,j-1))+1
return opt(1,n)

```

grading rubrics:

initialization: 3pts (need to initialize to 0).

recursion: if $si == sj$: $opt(i,j) = opt(i+1, j-1)$ 2pts; $opt(i,j) = \min(opt(i+1,j), opt(i,j-1))+1$ 3pts.

4) 20 pts

The Laver cup is a tennis tournament that takes place every year between Team Europe and Team US, each consisting of k players. Each player participating has a world ranking. A match between players of ranking r_i and r_j is fair if $|r_i - r_j| \leq d$. The tournament must consist of n matches, each between a player from Team EU and a player from Team US. Describe an efficient algorithm that, given the rankings of all the players, determines if it is possible to schedule n fair matches between the two teams such that each player plays at least one match but no more than m matches, and no two players play each other more than once.

A4.

- a) bipartite graph with k players of team US on left and k players of team EU on right.
- b) connect with an edge each US player (us_i) to the corresponding EU player (eu_j) whose skill level is within d . Each such edge has weight one to ensure at most one match between any two players in opposing teams.
- c) two nodes source and sink. Where edges (s, us_i) have a lower bound of 1 and capacity m . Likewise edges (eu_j, t) have lower bound 1 and capacity m .
- d) Add a demand of n at t and supply of n at s .

To solve: Convert to the problem of circulation with demand.

Claim: a matching exists if there exists a feasible circulation in the network.

Grading Rubric:

1. $S \rightarrow US \rightarrow EU \rightarrow T$ (or $S \rightarrow EU \rightarrow US \rightarrow T$) structure

2. Demand of $-n$ for S and n for T
3. Connecting players with the restriction of fair matching
4. For $S \rightarrow US$ and $EU \rightarrow T$ edges, capacity= m and lower bound=1; for $US \rightarrow EU$ edges, capacity=1 and no lower bound
5. Final claim indicating a maxflow (available flow) of size n . Note that $\text{maxflow} \geq n$ is incorrect, because when $\text{maxflow} > n$, a flow of n is not necessarily possible in the network (e.g. when $k > n$, we get $\text{maxflow} \geq k > n$, but a flow of size n is impossible).

4 points each for each correct use of above. No partial credit!

Algorithms other than network flow are considered incorrect/inefficient, with 0 point given.

20 pts

- 5) Your company, Stuckbars Coffee and Toffee is planning to open several stores on Main Street. Main Street has n blocks $1, 2, \dots, n$ from East to West, and for each block i you know the revenue $r_i > 0$ you expect from a store on this block. But if you open a store on block i , you cannot open another store in block i , or the two blocks to its East or the two blocks to its West. You have designed a dynamic programming algorithm for finding the store locations that maximizes total revenue, but you spilled coffee on it and now you have to reconstruct it. Here are the steps you need to follow.

All solution more complex than the linear solution are considered incorrect.
The incorrect part will lead to score reduction in every section. If the solution uses other methods, a student must show it has a complexity of $O(n)$.
Otherwise, it is very likely to get a lower score if request to be regraded.

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

Choice of subproblem: $R[i]$ = the maximum total revenue if stores can only be opened on blocks $1, \dots, i$

Rubric:

If state $opt(i)$ means max revenue for store 1 to i , or i to n . (full credits)

If state placement on a store on i th block but not mention 1 to i or i to n . (-1 or -2)

Other complicated sub problem or ambiguous statement (-2 or -3)

If not a subproblem: -4

- e) Write the recurrence relation for subproblems. (6 pts)

Recurrence relation: $R[i] = \max \{ R[i - 1], r_i + R[i - 3] \}$

Rubric:

If $\max(r_i + opt(i-3), opt(i-1))$ (full credits)

Write $\max(opt(i-1), opt(i-2), opt(i-3) + r_i)$ is ok, since it doesn't increase complexity

If writes $i-3$ as $i-2$: (-1 or -2)

Other incorrect answer (-3 to -5)

Empty (-6)

- f) Using the recurrence formula in part b, write pseudocode to compute the maximum revenue possible. (4 pts)

Make sure you have initial values properly assigned. (2 pts)

$R[1] = r_1$ $R[2] = \max(r_1, r_2)$ $R[3] = \max(r_1, r_2, r_3)$

For $i = 4, \dots, n;$

$R[i] = \max \{ R[i - 1], r_i + R[i - 3] \}$

Initializing n, n-1 and n-2, or R[0], R[1], R[2], or R[<0], R[1] are all correct

If index is incorrect (-1 or -2)

If other incorrect methods which have higher complexity (-3 or -4)

Initialization as R[3] = r3 R[2]=r2 (-1 or -2)

- g) Using the results in part c, write pseudocode to determine the store locations.
(2 pts)

Recover solution: $prev[i] = \begin{cases} i - 3, & \text{if } R[i] = r_i + R[i - 3] \\ i - 1, & \text{otherwise} \end{cases}$

Rubric:

Only mention Trace back (-2 or -1)

Wrong in bound (-1)

Not using result in c or empty (-2)

Include i-1 or i-3 in the location (-1 or -2) consider a case: r1=1, r2=1, r3=1000, r4=1, r5=1, r6=1, r7=1, r8=1000, r9=1, r10=1

we should only select r3 and r8.

- h) Compute the runtime of the algorithm described in part c and state whether your solution runs in polynomial time or not (2 pts)

Running time: O(n), since each subproblem takes constant time to compute based on smaller subproblems, and there are n subproblems to compute.

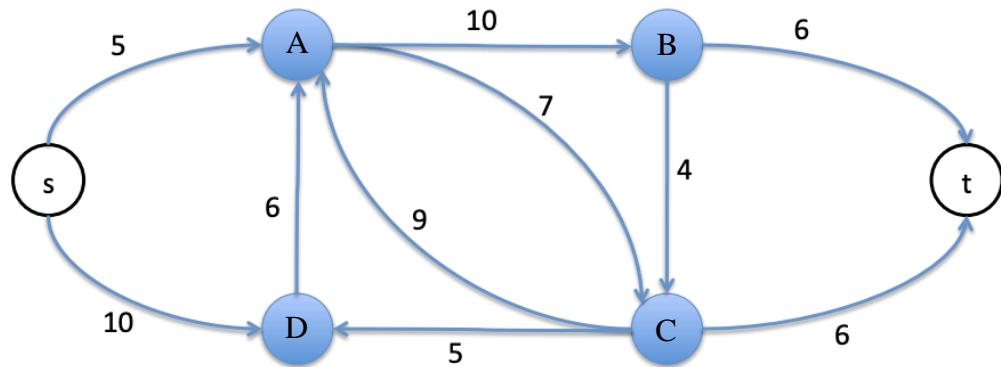
Rubric:

If not O(n) (-1)

If not strong polynomial (-1)

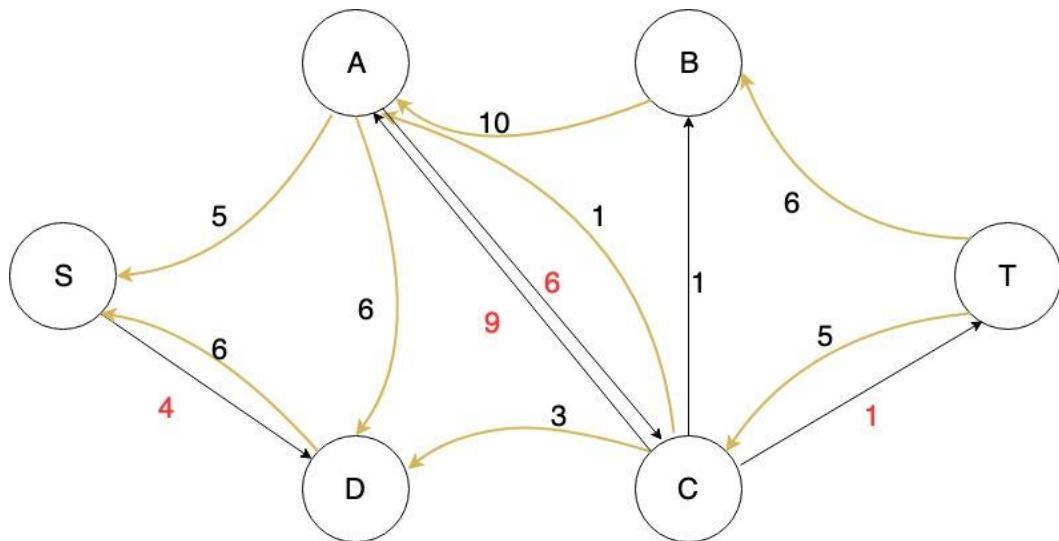
10 pts

- 6) For the following flow network, with edge capacities as shown,

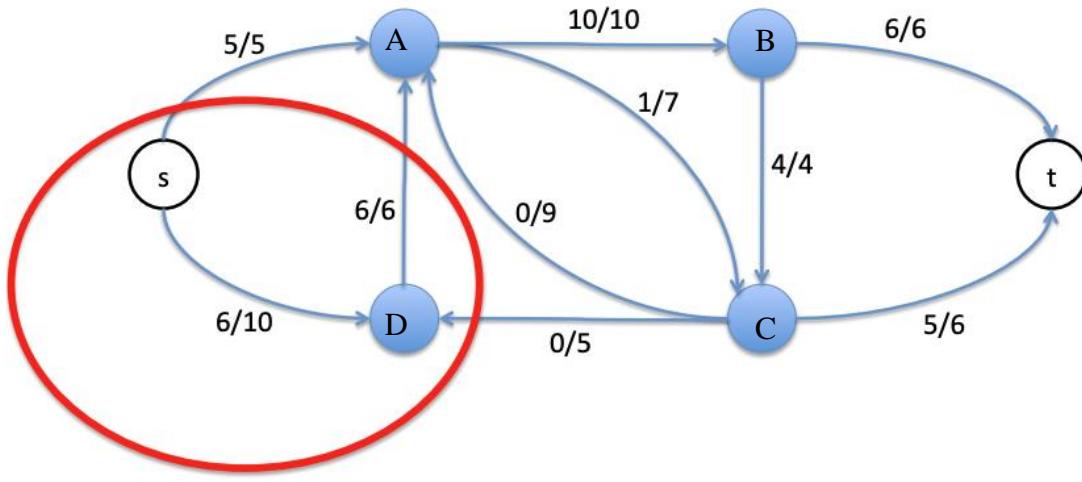


- a) Find a minimum s-t cut in the network and indicate what its capacity is.
(Clearly show which nodes are in which set)

Solution



This is the final residual graph, yellow ones are back edge, red values are residual capacity.



The cut has cost 11, and we can show that it is the minimum cut by demonstrating a flow with value 11—see above for the flow assignment.

Two sets: set1{D},set2{A, C, B}

Since the maximum-flow is equal to the minimum cut, this proves that 11 is the minimum cut.

- b) Describe how you would determine if this is the only minimum cut in G.

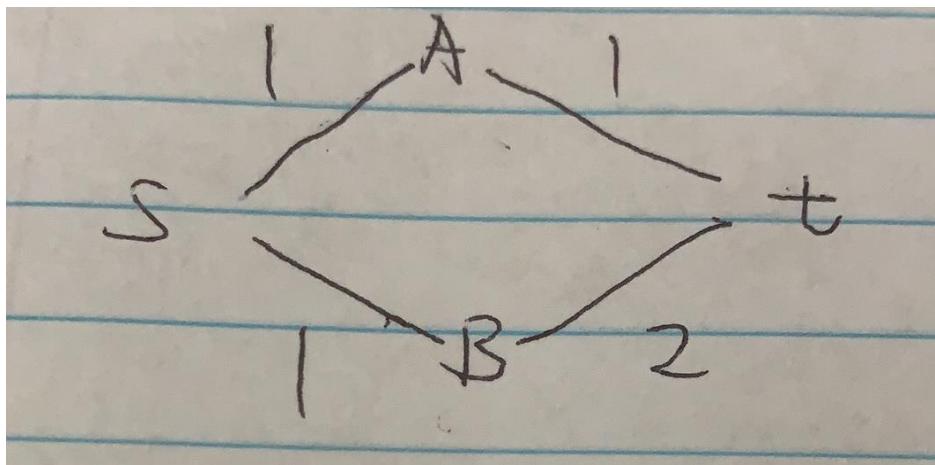
Solution1. BFS from source to output the min-cut. Then BFS from sink to output another min-cut. If same then this is the only mincut, else multiple min-cuts exist.

Solution2. For every edge e in (set1, set2), increase the capacity of that edge and compute the value of the max-flow again. If the value does not increase, then it means that there is another min-cut that there was another min-cut in the original graph that did not include this edge, and hence the min-cut was not unique. If for all the edges in (set1, set2) the value of the max-flow increases, then the min-cut is unique.

Common mistakes:

1. Increase value on only one of the edges in min-cut. Or increase all of them at the same time instead of one by one.

Counterexample:



2. Try to find augment path in other orders to find different min-cut.
 Counterexample: s-1-A-2-B-1-t

Rubric:

a.

Steps for 5 points.

Min-cut for 3 points.

Capacity for 2 points.

b.

Partial score is given only on common mistake 1.

CS570
Analysis of Algorithms
Fall 2014
Exam II

Name: _____
Student ID: _____
Email: _____

Wednesday Evening Section

	Maximum	Received
Problem 1	20	
Problem 2	16	
Problem 3	16	
Problem 4	20	
Problem 5	16	
Problem 6	12	
Total	100	

Instructions:

1. This is a 2-hr exam. Closed book and notes
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.

1) 20 pts

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

[/FALSE]

If an iteration of the Ford-Fulkerson algorithm on a network places flow 1 through an edge (u, v) , then in every later iteration, the flow through (u, v) is at least 1.

[TRUE/]

For the recursion $T(n) = 4T(n/3) + n$, the size of each subproblem at depth k of the recursion tree is $n/3^{k-1}$.

[/FALSE]

For any flow network G and any maximum flow on G , there is always an edge e such that increasing the capacity of e increases the maximum flow of the network.

[/FALSE]

The asymptotic bound for the recurrence $T(n) = 3T(n/9) + n$ is given by $\Theta(n^{1/2} \log n)$.

[/FALSE]

Any Dynamic Programming algorithm with n subproblems will run in $O(n)$ time.

[/FALSE]

A pseudo-polynomial time algorithm is always slower than a polynomial time algorithm.

[TRUE/]

The sequence alignment algorithm can be used to find the longest common subsequence between two given sequences.

[/FALSE]

If a dynamic programming solution is set up correctly, i.e. the recurrence equation is correct and each unique sub-problem is solved only once (memoization), then the resulting algorithm will always find the optimal solution in polynomial time.

[TRUE/]

For a divide and conquer algorithm, it is possible that the divide step takes longer to do than the combine step.

[TRUE/]

Maximum value of an $s - t$ flow could be less than the capacity of a given $s - t$ cut in a flow network.

2) 16 pts

Recall the Bellman-Ford algorithm described in class where we computed the shortest distance from all points in the graph to t. And recall that we were able to find all shortest distance to t with only $O(n)$ memory.

How would you extend the algorithm to compute both the shortest distance and to find the actual shortest paths from all points to t with only $O(n)$ memory?

We need an array of size n to hold a pointer to the neighbor that gives us the shortest distance to t. Initially all pointers are set to Null. Whenever a node's distance to t is reduced, we update the pointer for that node and point it to the node that is giving us a lower distance to t. Once all shortest distances are computed, to find a path from any node v to t, one can simply follow these pointers to reach t on the shortest path.

3) 16 pts

During their studies, 7 friends (Alice, Bob, Carl, Dan, Emily, Frank, and Geoffrey) live together in a house. They agree that each of them has to cook dinner on exactly one day of the week. However, assigning the days turns out to be a bit tricky because each of the 7 students is unavailable on some of the days. Specifically, they are unavailable on the following days (1 = Monday, 2 = Tuesday, ..., 7 = Sunday):

- Alice: 2, 3, 4, 5
- Bob: 1, 2, 4, 7
- Carl: 3, 4, 6, 7
- Dan: 1, 2, 3, 5, 6
- Emily: 1, 3, 4, 5, 7
- Frank: 1, 2, 3, 5, 6
- Geoffrey: 1, 2, 5, 6

Transform the above problem into a maximum flow problem and draw the resulting flow network. If a solution exists, the flow network should indicate who will cook on each day; otherwise it must show that a feasible solution does not exist

Solution:

I will use the initials of each person's name to refer to them in this solution.

Construct a graph $G = (V, E)$. V consists of 1 node for each person (let us denote this set by $P = \{A, B, C, D, E, F, G\}$), 1 node for each day of the week (let's call this set $D = \{1, 2, 3, 4, 5, 6, 7\}$), a source node s , and a sink node t . Connect s to each node p in P by a directed (s, p) edge of unit capacity. Similarly, connect each node d in D to t by a directed (d, t) edge of unit capacity. Connect each node p in P by a directed edge of unit capacity to those nodes in D when p is **available** to cook. This completes our construction of the flow network. I am omitting the actual drawing of G here.

Finding a max-flow of value 7 in G translates to finding a feasible solution to the allocation of cooking days problem. Since there can be at most unit flow coming into any node p in P , a maximum of unit flow can leave it. Similarly, at most a flow of value 1 can flow into any node d in D because a maximum of unit flow can leave it. Thus, a max-flow of value 7 means that there exists a flow-carrying s - p - d - t path for each p and d . Any (p, d) edge with unit flow indicates that person p will cook on day d .

The following lists one possible max-flow of value 7 in G :

Send unit flow on each (s, p) edge and each (d, t) edge. Also send unit flow on the following (p, d) edges: $(A, 6)$, $(B, 5)$, $(C, 1)$, $(D, 4)$, $(E, 2)$, $(F, 7)$, $(G, 3)$

4) 20 pts

Suppose that there are n asteroids that are headed for earth. Asteroid i will hit the earth in time t_i and cause damage d_i unless it is shattered before hitting the earth, by a laser beam of energy e_i . Engineers at NASA have designed a powerful laser weapon for this purpose. However, the laser weapon needs to charge for a duration ce before firing a beam of energy e . Can you design a dynamic programming based pseudo-polynomial time algorithm to decide on a firing schedule for the laser beam to minimize the damage to earth? Assume that the laser is initially uncharged and the quantities c, t_i, d_i, e_i are all positive integers. Analyze the running time of your algorithm. You should include a brief description/derivation of your recurrence relation. Description of recurrence relation = 8pts, Algorithm = 6pts, Run Time = 6pts

Solution 1 (Assuming that the laser retains energy between firing beams): Sort all asteroids by the time t_i . Label the asteroids from 1 to n and without loss of generality assume $t_1 < t_2 < \dots < t_n$. Also assume that if asteroid i is destroyed then it is done exactly at time t_i (if the laser continuously accumulates energy then the destruction order of the asteroids does not change even if i^{th} asteroid is shot down before time t_i).

Define $OPT(i, T)$ as the minimum possible damage caused to earth due to asteroids $i, i + 1, \dots, n$ if $\frac{T}{c}$ energy is left in the laser just before time t_i . We want the solution corresponding to $OPT(1, t_1)$. If $T \geq ce_i$ and the i^{th} asteroid is destroyed then $OPT(i, T) = OPT(i + 1, T - ce_i + t_{i+1} - t_i)$, otherwise $OPT(i, T) = d_i + OPT(i + 1, T + t_{i+1} - t_i)$. Hence,

$$OPT(i, T) = \begin{cases} d_i + OPT(i + 1, T + t_{i+1} - t_i), & T < ce_i \\ \min\{d_i + OPT(i + 1, T + t_{i+1} - t_i), OPT(i + 1, T - ce_i + t_{i+1} - t_i)\}, & T \geq ce_i \end{cases}$$

Boundary condition: $OPT(n, T) = 0$ if $T \geq ce_n$ and $OPT(n, T) = d_n$ if $T < ce_n$, since if there is enough energy left to destroy the last asteroid then it is always beneficial to do so. Furthermore, $T \leq t_n$ since a maximum of $\frac{t_n}{c}$ energy is accumulated by the laser before the last asteroid hits the earth and $T \geq 0$ since the left over energy in the laser is always non-negative.

Algorithm:

- i. Initialize $OPT(n, T)$ according to the boundary condition above for $0 \leq T \leq t_n$.
- ii. For each $n - 1 \geq i \geq 1$ and $0 \leq T \leq t_n$ populate $OPT(i, T)$ according to the recurrence defined above.
- iii. Trace forward through the two dimensional OPT array starting at $(1, t_1)$ to determine the firing sequence. For $1 \leq i \leq n - 1$, destroy the i^{th} asteroid with $\frac{T}{c}$ energy left if and only if $OPT(i, T) = OPT(i + 1, T - ce_i + t_{i+1} - t_i)$. Destroy the n^{th} asteroid only if $T \geq ce_n$.

Complexity: Step (i) initializes t_n values each taking constant time. Step (ii) computes $(n - 1)t_n$ values, each taking one invocation of the recurrence and hence is done in $O(nt_n)$ time. Trace back takes $O(n)$ time since the decision for each i takes constant time. Initial sorting takes $O(n \log n)$ time. Thus overall complexity is $O(nt_n + n \log n)$.

Solution 2 (Assuming that the laser does not retain energy left over after firing and if asteroid i is destroyed then it is done exactly at time t_i): Sort all asteroids by the time t_i . Label the asteroids from 1 to n and without loss of generality assume $t_1 < t_2 < \dots < t_n$. In contrast to Solution 1, the destroying sequence will change if we are free to destroy asteroid i before time t_i (this case is not solved here).

Define $OPT(i)$ to be the minimum possible damage caused to earth due to the first i asteroids. We want the solution corresponding to $OPT(n)$. If the i^{th} asteroid is not destroyed either by choice or because $t_i < ce_i$ then $OPT(i) = d_i + OPT(i - 1)$. On the other hand, if the i^{th} asteroid is destroyed ($t_i \geq ce_i$ is necessary) then none of the asteroids arriving between times $t_i - ce_i$ and t_i (both exclusive) can be destroyed. Letting $p[i]$ denote the largest positive integer such that $t_{p[i]} \leq t_i - ce_i$ (and $p[i] = 0$ if no such integer exists), we have $OPT(i) = OPT(p[i]) + \sum_{j=p[i]+1}^{i-1} d_j$ if $p[i] \leq i - 2$ and $OPT(i) = OPT(i - 1)$ if $p[i] = i - 1$. Hence for $i \geq 1$,

$$OPT(i) = \begin{cases} OPT(i - 1), & t_i \geq ce_i \text{ and } p[i] = i - 1 \\ d_i + OPT(i - 1), & t_i < ce_i \\ \min \left\{ d_i + OPT(i - 1), OPT(p[i]) + \sum_{j=p[i]+1}^{i-1} d_j \right\}, & t_i \geq ce_i \text{ and } p[i] \leq i - 2 \end{cases}$$

Boundary condition: $OPT(0) = 0$ since no asteroids means no damage.

Algorithm:

- i. Form the array p element-wise. This is done as follows.
 - a. Set $p[1] = 0$.
 - b. For $2 \leq i \leq n$, binary search for $t_i - ce_i$ in the sorted array $t_1 < t_2 < \dots < t_n$. If $t_i - ce_i < t_1$ then set $p[i] = 0$ else record $p[i]$ as the index such that $t_{p[i]} \leq t_i - ce_i < t_{p[i]+1}$.
- ii. Form array D to store cumulative sum of damages. Set $D[0] = 0$ and $D[j] = D[j - 1] + d_j$ for $1 \leq j \leq n$.
- iii. Set $OPT(0) = 0$ and for $1 \leq i \leq n$ populate $OPT(i)$ according to the recurrence defined above, computing $\sum_{j=p[i]+1}^{i-1} d_j$ as $D[i - 1] - D[p[i]]$.
- iv. Trace back through the one dimensional OPT array starting at $OPT(n)$ to determine the firing sequence. Destroy the first asteroid if and only if $OPT(1) = 0$. For $2 \leq i \leq n$, destroy the i^{th} asteroid if and only if either $OPT(i) = OPT(i - 1)$ with $p[i] = i - 1$ or $OPT(i) = OPT(p[i]) + D[i - 1] - D[p[i]]$ with $p[i] \leq i - 2$.

Complexity: initial sorting takes $O(n \log n)$. Construction of array p takes $O(\log n)$ time for each index and hence a total of $O(n \log n)$ time. Forming array D is done in $O(n)$ time. Using array D , $OPT(i)$ can be populated in constant time for each $1 \leq i \leq n$ and hence step (iii) takes $O(n)$ time. Trace back takes $O(n)$ time since the decision for each i takes constant time. Therefore, overall complexity is $O(n \log n)$.

5) 16 pts

Consider a two-dimensional array $A[1:n, 1:n]$ of integers. In the array each row is sorted in ascending order and each column is also sorted in ascending order. Our goal is to determine if a given value x exists in the array.

- a. One way to do this is to call binary search on each row (alternately, on each column). What is the running time of this approach? [2 pts]
 - b. Design another divide-and-conquer algorithm to solve this problem, and state the runtime of your algorithm. Your algorithm should take strictly less than $O(n^2)$ time to run, and should make use of the fact that each row and each column is in sorted order (i.e., don't just call binary search on each row or column). State the run-time complexity of your solution.
- a) $O(n \log n)$.
- b) Look at the middle element of the full matrix. Based on this, you can either eliminate $A[1.. \frac{n}{2}, 1.. \frac{n}{2}]$ or $A[\frac{n}{2}..n, \frac{n}{2}..n]$. If x is less than middle element then you can eliminate $A[\frac{n}{2}..n, \frac{n}{2}..n]$. If x is greater than middle element then you can eliminate $A[1.. \frac{n}{2}, 1.. \frac{n}{2}]$. You can then recursively search in the remaining three $\frac{n}{2} \times \frac{n}{2}$ matrices. The total runtime is $T(n) = 3T(\frac{n}{2}) + O(1)$, $T(n) = O(n^{\log_2 3})$.

6) 12 pts

Consider a divide-and-conquer algorithm that splits the problem of size n into 4 sub-problems of size $n/2$. Assume that the divide step takes $O(n^2)$ to run and the combine step takes $O(n^2 \log n)$ to run on problem of size n . Use any method that you know of to come up with an upper bound (as tight as possible) on the cost of this algorithm.

Solution: Use the generalized case 2 of Master's Theorem. For $T(n) = aT\left(\frac{n}{b}\right) + \Theta(n^{\log_b a} \log^k n)$ with $k \geq 0$, we have $T(n) = \Theta(n^{\log_b a} \log^{k+1} n)$.

The divide and combine steps together take $O(n^2 \log n)$ time and the worst case is that they actually take $\Theta(n^2 \log n)$ time. Hence the recurrence for the given algorithm is $T(n) = 4T\left(\frac{n}{2}\right) + \Theta(n^2 \log n)$ in the worst case. Comparing with the generalized case, $a = 4, b = 2, k = 1$ and so $T(n) = \Theta(n^2 \log^2 n)$. Since this expression for $T(n)$ is the worst case running time, an upper bound on the running time is $O(n^2 \log^2 n)$.

Additional Space

CS570
Analysis of Algorithms
Spring 2015
Exam II

Name: _____

Student ID: _____

Email Address: _____

_____ Check if DEN Student

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

Instructions:

1. This is a 2-hr exam. Closed book and notes
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.

1) 20 pts

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

[**FALSE**]

A flow network with unique edge capacities has a unique min cut.

[**TRUE**/]

If a problem can be solved by dynamic programming, then it can always be solved by exhaustive search (Brute Force).

[**TRUE**/]

A divide and conquer algorithm acting on an input size of n can have a lower bound less than $\Theta(n \log n)$.

[**FALSE**]

If a flow in a network has a cycle, this flow is not a valid flow.

[**FALSE**]

In the divide and conquer algorithm to compute the closest pair among a given set of points on the plane, if the sorted order of the points on both X and Y axis are given as an added input, then the running time of the algorithm improves to $O(n)$.

[**TRUE**/]

In a flow network, an edge that goes straight from s to t is always saturated when maximum $s - t$ flow is reached.

[**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.

[**TRUE**/]

If f is a max $s-t$ flow of a flow network G with source s and sink t , then the capacity of the min $s-t$ cut in the residual graph G_f is 0.

[**FALSE**]

In a dynamic programming solution, the space requirement is always at least as big as the number of unique sub problems.

[**FALSE**]

Decreasing the capacity of an edge that belongs to a min cut in a flow network may not result in decreasing the maximum flow.

2) 20 pts

A city is located at one node x in an undirected network $G = (V, E)$ of channels. There is a big river beside the network. In the rainy season, the flood from the river flows into the network through a set of nodes Y . Assume that the flood can only flow along the edges of the network. Let c_{uv} (integer value) represent the minimum effort (counted in certain effort unit) of building a dam to stop the flood flowing through edge (u, v) . The goal is to determine the minimum total effort of building dams to prevent the flood from reaching the city. Give a pseudo-polynomial time algorithm to solve this problem. Justify your algorithm.

Algorithm:

1. Add node s , and add edges from s to each node in Y .
2. Set the capacity of each of these new edges as infinity.
3. Set city node x as the sink node. Then the flood network G becomes a larger network G' with source s and sink x .
4. Find the min s - x cut on G' by running a polynomial time max-flow algorithm, where you can treat the cost of building a dam on each edge to be the capacity of this edge.
5. Build a dam on each edge in the min-cut's cut-set.

Complexity:

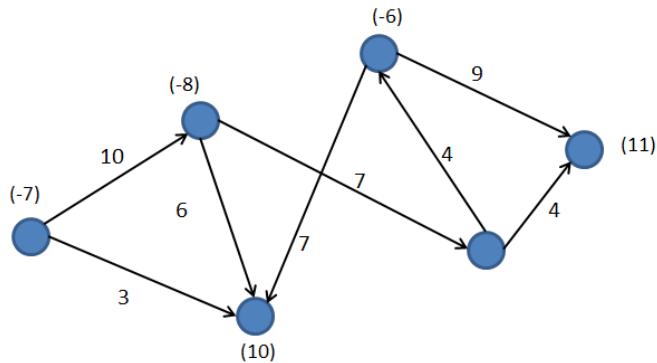
Since the construction of the new edges takes $O(|Y|)$ times, together with a polynomial time max-flow algorithm to find the min-cut, the entire algorithm takes polynomial time.

Justification:

We're simply trying to separate x from Y by choosing edges with the minimum cost, which is a min-cut problem. But min-cut only works when Y is a single node. We fix this by creating a super source s directly connected to Y . But we want to make sure the min-cut's cut-set doesn't include any edge connecting s , because these edges do not belong to G . This is why we put the capacities arbitrarily large on these edges. Min-cut would then find the min-cost way to separate x from Y .

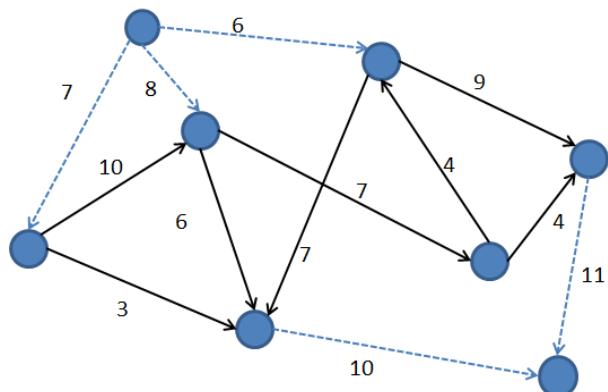
3) 20 pts

The following graph G is an instance of a circulation problem with demands. The edge weights represent capacities and the node weights (in parentheses) represent demands. A negative demand implies source.



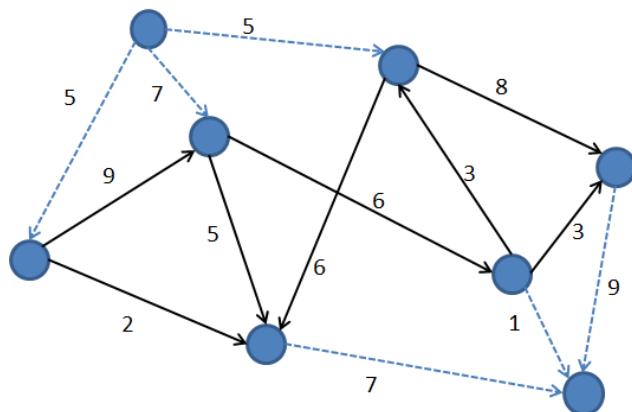
(i) Transform this graph into an instance of max-flow problem.

Solution:



(ii) Now, assume that each edge of G has a constraint of lower bound of 1 unit, i.e., one unit must flow along all edges. Find the new instance of max-flow problem that includes the lower bound constraint.

Solution:



4) 20 pts

There is a series of activities lined up one after the other, J_1, J_2, \dots, J_n . The i^{th} activity takes T_i units of time, and you are given M_i amount of money for it. Also for the i^{th} activity, you are given N_i , which is the number of immediately following activities that you cannot take if you perform that i^{th} activity. Give a dynamic programming solution to maximize the amount of money one can make in T units of time. Note that an activity has to be completed in order to make any money on it. State the runtime of your algorithm.

Solution:

Recurrence formula:

If $T_i > t$ then $\text{opt}(i, t) = \text{opt}(i+1, t)$,

Otherwise, $\text{opt}(i, t) = \max(\text{opt}(i+1, t), \text{opt}(i + N_i + 1, t - T_i) + M_i)$

Boundary conditions: $\text{opt}(i, t) = 0$ for $i > n$ and all t ,

$\text{opt}(i, t) = 0$ for $t < 1$ and all i

To find the value of the optimal solution:

for $i=n$ to 1 by -1

 for $t=1$ to T

 If $T_i > t$ then $\text{opt}(i, t) = \text{opt}(i+1, t)$,

 Otherwise, $\text{opt}(i, t) = \max(\text{opt}(i+1, t), \text{opt}(i + N_i + 1, t - T_i) + M_i)$

 end for

end for

$\text{opt}(1, T)$ will hold the value of the optimal solution (maximum money that can be made). Complexity is $O(nT)$

Given all the values in the two dimensional $\text{opt}()$ array, the optimal set of activities can be found in $O(n)$

Overall complexity of the algorithm is $O(nT)$ which is pseudopolynomial.

5) 20 pts

A polygon is called convex if all of its internal angles are less than 180° and none of the edges cross each other. We represent a convex polygon as an array V with n elements, where each element represents a vertex of the polygon in the form of a coordinate pair (x, y) . We are told that $V[1]$ is the vertex with the least x coordinate and that the vertices $V[1], V[2], \dots, V[n]$ are ordered counter-clockwise. Assuming that the x coordinates (and the y coordinates) of the vertices are all distinct, do the following.

Give a divide and conquer algorithm to find the vertex with the largest x coordinate in $O(\log n)$ time.

Solution:

Since $V[1]$ is known to be the vertex with the minimum x -coordinate (leftmost point), moving counter-clockwise to complete a cycle must first increase the x -coordinates and then after reaching a maximum (rightmost point), should decrease the x -coordinate back to that of $V[1]$. To see this more formally, we claim that there cannot be two distinct local maxima in the x -axis projection of the counter-clockwise tour. For the sake of contradiction, assume otherwise. Then there exists a vertical line that would intersect the boundary of the polygon at more than two points. This is impossible by convexity of the polygon since the line segments between the intersection points must lie completely in the interior of the polygon, thus contradicting our assumption. Thus, $V_x[1 : n]$ is a unimodal array, and the first part of this question is synonymous with detecting the location of the maximum element in this array.

Consider the following algorithm:

- (a) If $n = 1$, return $V_x[1]$.
- (b) If $n = 2$, return $\max\{V_x[1], V_x[2]\}$.
- (c) $k = \lceil \frac{n}{2} \rceil$.
- (d) If $V_x[k] > V_x[k - 1]$ and $V_x[k] > V_x[k + 1]$, then return $V_x[k]$.
- (e) If $V_x[k] < V_x[k - 1]$ then call the algorithm recursively on $V_x[1 : k - 1]$, else call the algorithm recursively on $V_x[k + 1 : n]$.

Complexity: If $T(n)$ is the running time on an input of size n , then beside a constant number of comparisons, the algorithm is called recursively on at most one of $V_x[1 : k - 1]$ (size = $k - 1 = \lceil \frac{n}{2} \rceil - 1 \leq \lfloor \frac{n}{2} \rfloor$) or $V_x[k + 1 : n]$ (size = $n - k = n - \lceil \frac{n}{2} \rceil = \lfloor \frac{n}{2} \rfloor$). Therefore, $T(n) \leq T(\lfloor \frac{n}{2} \rfloor) + \theta(1)$.

Assuming n to be a power of 2, this recurrence simplifies to $T(n) \leq T(n/2) + \theta(1)$ and invoking Master's Theorem gives $T(n) = O(\log n)$.

Additional Space

RCS570 Fall 2017: Analysis of Algorithms Exam II

	Points
Problem 1	20
Problem 2	20
Problem 3	15
Problem 4	15
Problem 5	15
Problem 6	15
Total	100

Instructions:

1. This is a 2-hr exam. Closed book and notes
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.

1) 20 pts

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

[**TRUE/FALSE**]

It is possible for a dynamic programming algorithm to have an exponential running time.

[**TRUE/FALSE**]

In a connected, directed graph with positive edge weights, the Bellman-Ford algorithm runs asymptotically faster than the Dijkstra algorithm.

[**TRUE /FALSE**]

There exist some problems that can be solved by dynamic programming, but cannot be solved by greedy algorithm.

[**TRUE /FALSE**]

The Floyd-Warshall algorithm is asymptotically faster than running the Bellman-Ford algorithm from each vertex.

[**TRUE /FALSE**]

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

[**TRUE /FALSE**]

The Ford-Fulkerson algorithm solves the maximum bipartite matching problem in polynomial time.

[**TRUE /FALSE**]

Given a solution to a max-flow problem, that includes the final residual graph G_f . We can verify in a *linear* time that the solution does indeed give a maximum flow.

[**TRUE /FALSE**]

In a flow network, a flow value is upper-bounded by a cut capacity.

[**TRUE/FALSE**]

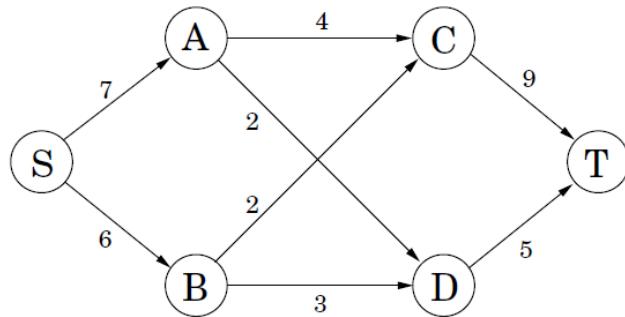
In a flow network, a min-cut is always unique.

[**TRUE/FALSE**]

A maximum flow in an integer capacity graph must have an integer flow on each edge.

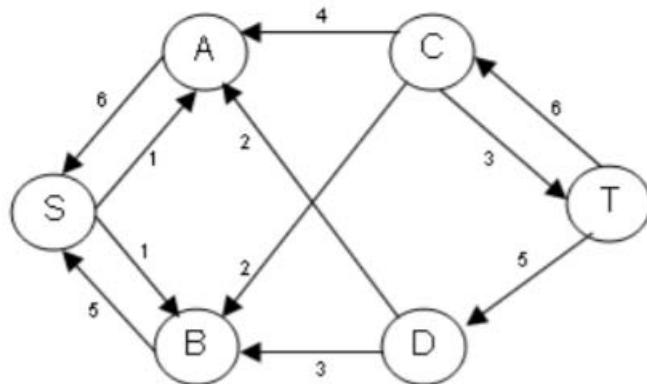
2) 20 pts.

You are given the following graph G . Each edge is labeled with the capacity of that edge.



- a) Find a max-flow in G using the Ford-Fulkerson algorithm. Draw the residual graph G_f corresponding to the max flow. You do not need to show all intermediate steps. (10 pts)

solution



Grading Rubics:

Residual graph: 10 points in total

For each edge in the residual graph, any wrong number marked, wrong direction, or missing will result in losing 1 point.

The total points you lose is equal to the number of edges on which you make any mistake shown above.

b) Find the max-flow value and a min-cut. (6 pts)

Solution: $f = 11$, cut: ($\{S, A, B\}$, $\{C, D, T\}$)

Grading Rubics:

Max-flow value and min-cut: 6 points in total

Max-flow: 2 points.

- If you give the wrong value, you lose the 2 points.

Min-cut: 4 points

- If your solution forms a cut but not a min-cut for the graph, you lose 3 points
- If your solution does not even form a cut, you lose all the 4 points

c) Prove or disprove that increasing the capacity of an edge that belongs to a min cut will always result in increasing the maximum flow. (4 pts)

Solution: increasing (B,D) by one won't increase the max-flow

Grading Rubics:

Prove or Disprove: 4 points in total

- If you judge it "True", but give a structural complete "proof". You get at most 1 point
- If you judge it "False", you get 2 points.
- If your counter example is correct, you get the rest 2 points.

Popular mistake: a number of students try to disprove it by showing that if the min-cut in the original graph is non-unique, then it is possible to find an edge in one min-cut set, such that increasing the capacity of this does not result in max-flow increase.

But they did not do the following thing:

The existence of the network with multiple min-cuts needs to be proved, though it seems to be obvious. The most straightforward way to prove the existence is to give an example-network that has multiple min-cuts. Then it turns out to be giving a counter example for the original question statement.

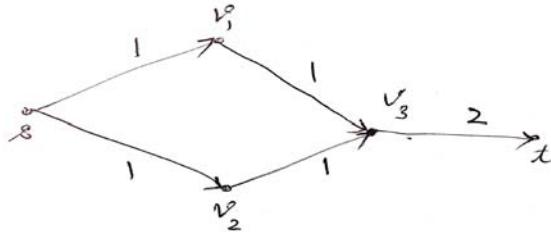
3) 15 pts.

Given a flow network with the source s and the sink t , and positive integer edge capacities c . Let (S, T) be a minimum cut. Prove or disprove the following statement:
If we increase the capacity of every edge by 1, then (S, T) still be a minimum cut.

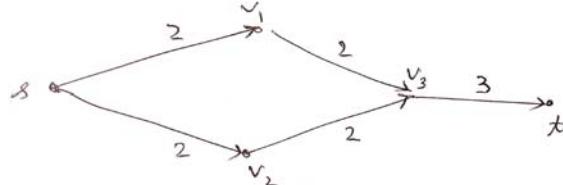
False. Create a counterexample.

An instance of a counter-example:

Initially, a (S, T) min cut is $S : \{s, v_1, v_2\}$ and $T : \{v_3, t\}$



After increasing capacity of every edge by 1, (S, T) is no longer a min-cut. We now have $S' : \{s, v_1, v_2, v_3\}$ and $T' : \{t\}$



Grading rubric:

If you try to prove the statement: -15

For an incorrect counter-example: -9

No credit for simply stating true or false.

4) 15 pts.

Given an unlimited supply of coins of denominations d_1, d_2, \dots, d_n , we wish to make change for an amount C . This might not be always possible. Your goal is to verify if it is possible to make such change. Design an algorithm by *reduction* to the knapsack problem.

- a) Describe reduction. What is the knapsack capacity, values and weights of the items? (10 pts.)

Capacity is C . (2 points)
values = weights = d_k . (4points)

- You need to recognize that the problem should be modeled based on the unbounded knapsack problem with description of the reduction: (5 points)
- Explanation of the verification criteria (4 points)
 $\text{Opt}(j)$: the maximum change equal or less than j that can be achieved with d_1, d_2, \dots, d_n
 $\text{Opt}(0)=0$
 $\text{Opt}(j) = \max[\text{opt}(j-d_i)+d_i] \text{ for } d_i \leq j$
If we obtain $\text{Opt}(C) = C$, it means the change is possible.

- b) Compute the runtime of your verification algorithm. (5 pts)

$O(nC)$ (3 points), Explanation (2 points).

5) 15 pts.

You are considering opening a series of electrical vehicle charging stations along Pacific Coast Highway (PCH). There are n possible locations along the highway, and the distance from the start to location k is $d_k \geq 0$, where $k = 1, 2, \dots, n$. You may assume that $d_i < d_k$ for $i < k$. There are two important constraints:

- 1) at each location k you can open only one charging station with the expected profit p_k , $k = 1, 2, \dots, n$.
- 2) you must open at least one charging station along the whole highway.
- 3) any two stations should be at least M miles apart.

Give a DP algorithm to find the maximum expected total profit subject to the given constraints.

- a) Define (in plain English) subproblems to be solved. (3 pts)

Let $\text{OPT}(k)$ be the maximum expected profit which you can obtain from locations $1, 2, \dots, k$.

Rubrics

Any other definition is okay as long as it will recursively solve subproblems

- b) Write the recurrence relation for subproblems. (6 pts)

$$\text{OPT}(k) = \max \{\text{OPT}(k - 1), p_k + \text{OPT}(f(k))\}$$

where $f(k)$ finds the largest index j such that $d_j \leq d_k - M$, when such j doesn't exist, $f(k)=0$

Base cases:

$$\text{OPT}(1) = p_1$$

$$\text{OPT}(0) = 0$$

Rubrics:

Error in recursion -2pts, if multiple errors, deduction adds up

No base cases: -2pts

Missing base cases: -1pts

Using variables without definition or explanation: -2pts

Overloading variables (re-defining n , p , k , or M): -2pts

- c) Compute the runtime of the above DP algorithm in terms of n . (3 pts)

algorithm solves n subproblems; each subproblem requires finding an index $f(k)$ which can be done in time $O(\log n)$ by binary search.
Hence, the running time is $O(n \log n)$.

Rubric:

$O(n^2)$ is regarded as okay

$O(d_n n)$ pseudo-polynomial is also okay if the recursion goes over all d_n values

$O(n)$ is also okay

$O(n^3), O(nM), O(kn)$: no credit

- d) How can you optimize the algorithm to have $O(n)$ runtime? (3 pts)

Preprocess distances $r_i = d_i - M$. Merge d-list with r-list.

Rubric

Claiming “optimal solution is already found in c ” only gets credit when explanation about pre-process is described in either part b or c.

Without proper explanation (e.g. assume we have, or we can do): no credit

Keeping an array of max profits: no credit, finding the index that is closest to the installed station with M distance away is the bottleneck, which requires pre-processing.

6) 15 pts.

A group of traders are leaving Switzerland, and need to convert their Francs into various international currencies. There are n traders t_1, t_2, \dots, t_n and m currencies c_1, c_2, \dots, c_m . Trader t_k has F_k Francs to convert. For each currency c_j , the bank can convert at most B_j Francs to c_j . Trader t_k is willing to trade as much as S_{kj} of his Francs for currency c_j . (For example, a trader with 1000 Francs might be willing to convert up to 200 of his Francs for USD, up to 500 of his Francs for Japanese's Yen, and up to 200 of his Francs for Euros). Assuming that all traders give their requests to the bank at the same time, describe an algorithm that the bank can use to satisfy the requests (if it can).

a) Describe how to construct a flow network to solve this problem, including the description of nodes, edges, edge directions and their capacities. (8 pts)

Bipartite graph: one partition traders t_1, t_2, \dots, t_n . Other, available currency, c_1, c_2, \dots, c_m .

Connect t_k to c_j with the capacity S_{kj}

Connect source to traders with the capacity F_k .

Connect available currency c_j to the sink with the capacity B_j .

Rubrics:

- Didn't include supersource (-1 point)
- Didn't include traders nodes (-1 point)
- Didn't include currencies nodes (-1 point)
- Didn't include supersink (-1 point)
- Didn't include edge direction (-1 point)
- Assigned no/wrong capacity on edges between source & traders (-1 point)
- Assigned no/wrong capacity on edges between traders & currencies (-1 point)
- Assigned no/wrong capacity on edges between currencies & sink (-1 point)

b) Describe on what condition the bank can satisfy all requests. (4 pts)

If there is a flow f in the network with $|f| = F_1 + \dots + F_n$, then all traders are able to convert their currencies.

Rubrics:

- Wrong condition (-4 points): Unless you explicitly included $|f| = F_1 + \dots + F_n$, no partial points were given for this subproblem.
- In addition to correct answer, added additional condition which is wrong (-2 points)
- No partial points were given for conditions that satisfy only some of the requests, not all requests.
- Note that $\sum S_{kj}$ and $\sum B_j$ can be larger than $\sum F_k$ in some cases where bank satisfy all requests.
- Note that $\sum S_{kj}$ can be larger than $\sum B_j$ in some cases where bank satisfy all requests.
- It is possible that $\sum S_{kj}$ or $\sum B_j$ is smaller than $\sum F_k$. In these cases, bank can never satisfy all requests because max flow will be smaller than $\sum F_k$.

- c) Assume that you execute the Ford-Fulkerson algorithm on the flow network graph in part a), what would be the runtime of your algorithm? (3 pts)

$O(n m |f|)$

Rubrics:

- Flow($|f|$) was not included (-1 point)
- Wrong description (-1 point)
- Computation is incorrect (-1 point)
- Notation error (-1 point)
- Missing big O notation (-1 point)
- Used big Theta notation instead of big O notation (-1 point)
- Wrong runtime complexity (-3 points)
- No runtime complexity is given (-3 points)

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, $E \leq 3V - 6$. 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 $O(V \log V + E \log V) = O(V \log V)$. Therefore, Prim's algorithm using array will run slower than Prim's algorithm using binary heap.

16. If $f(n) = \Omega(n \log n)$ 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 $T(n) = 8 T(n/4) + O(n^{1.5} \log n)$ 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 combining cost of $\Theta(n \log n)$

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 k^2 when $i = 0 \bmod k$ (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 B_k , 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:

$\text{head} \rightarrow a_1 \rightarrow a_2 \rightarrow \dots \rightarrow a_n$

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:

$$2*2 + 2*4 + 2*6 + \dots + 2*(n-2) + 2*n + n(\text{cost of swap}) = 4*(1+2+3+\dots+n/2) + n = n(n+2)/2 + n = O(n^2)$$

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: -6 pt
 - 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 $\text{node_value}(n) < \text{node_value}(x) * \text{weight}(x, n)$:

$\text{node_value}(n) = \text{node_value}(x) * \text{weight}(x, n)$

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, \dots, 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 $1 \leq i \leq n$ and sort the questions in descending order of $\frac{p_i}{t_i}$. Let the sorted order of questions denoted by $s(1), s(2), \dots, s(n)$. Answer questions in this order until $s(j)$ such that $\sum_{k=1}^j t_{s(k)} \leq T$ 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 p_i/t_i ratios: 1 point
- considering at least one of calculating p_i/t_i 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 $P(1), P(2), \dots, P(l)$ be true for some arbitrary l , i.e., the set of questions

$\{s(1), s(2), \dots, s(l)\}$ are part of an optimal solution O . It is clear that O should also be optimal with respect to the remaining questions $\{1, 2, \dots, n\} \setminus \{s(1), s(2), \dots, s(l)\}$ 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 \rightarrow 2 \rightarrow 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 we 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
 - 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:
- 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 $O(n^2)$

$$D[1] = 0$$

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

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 .

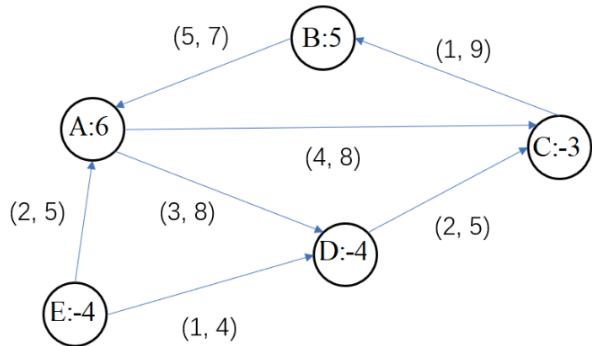
(5 points)

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

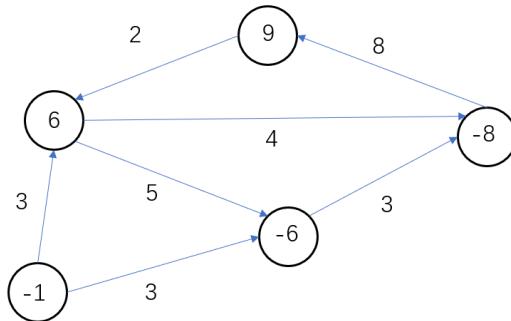
Q18 In the network below, the demand values are shown on vertices (supply values are negative). Lower bounds on flow and edge capacities are shown as (lower bound, capacity) for each edge. Determine if there is a feasible circulation in this graph. Please complete the following steps.



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

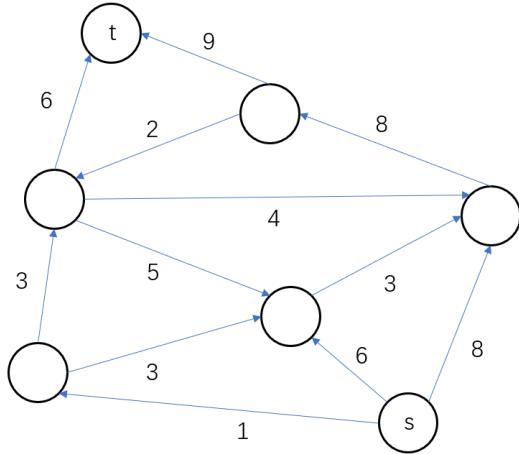
A: 6, B: 9, C: -8, D: -6, E: -1. (5 points)

(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 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 adjacent, 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:

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

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

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

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 $f(n) = 5n^24^n + 6n^43^n$ is $O(n^24^n)$.
- 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 $\Theta(n^3)$ and algorithm B has a worst case running time of $\Theta(n^2 \log n)$. 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 result 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)

$$n = 2^k - 1 \quad (2\text{pts for this answer})$$

- 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 $n \geq 1$, 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

Call Locally tallest (root)

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)

$$F(n) = \Theta(1)$$

$$T(n) = T(n/2) + f(n)$$

$$n^{\log \text{base } b \text{ of } a} = n^{\log \text{base 2 of } 1} = n^0 = \Theta(1)$$

$$\text{Case 2} \rightarrow T(n) = \Theta(\log n) = \Theta(k)$$

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, \dots, 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)

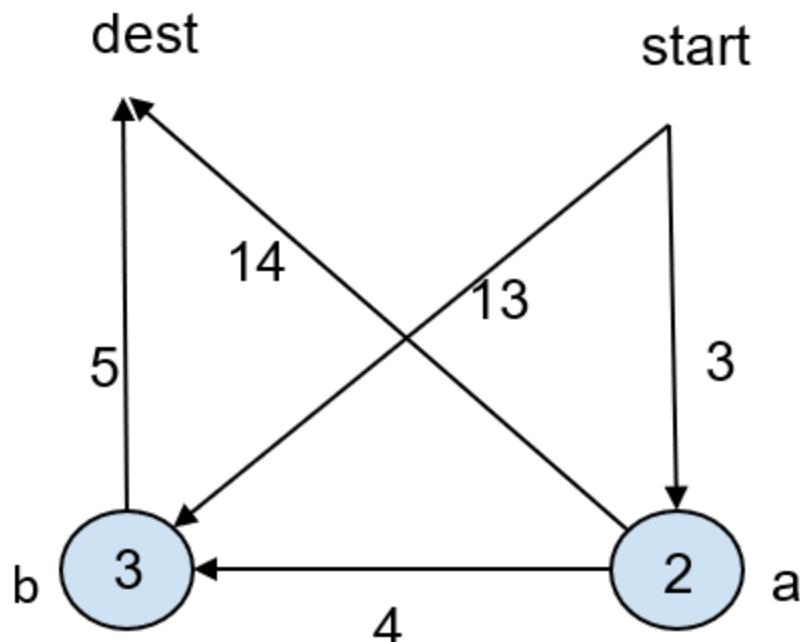
Only requires a Sort which takes $\Theta(n \log n)$

- 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* \rightarrow *a* \rightarrow *b* \rightarrow *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 - $O(m \log n)$ (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

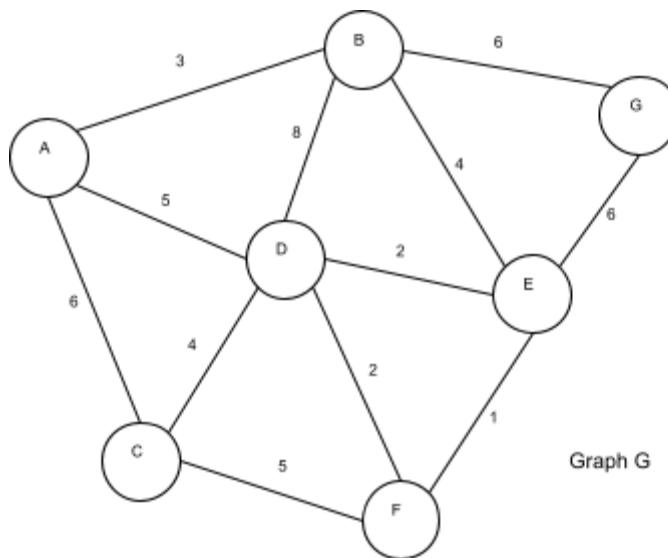
valid proof (3pts)

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.

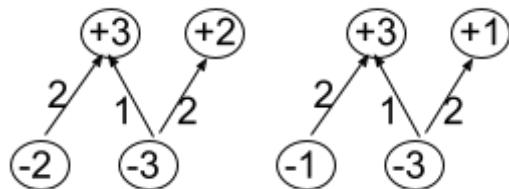
True/False Questions

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 $\text{Opt}(W, n+1) = \text{Opt}(W, n) + \text{Opt}(W - W_{n+1}, n)$ with problems on the right already computed.

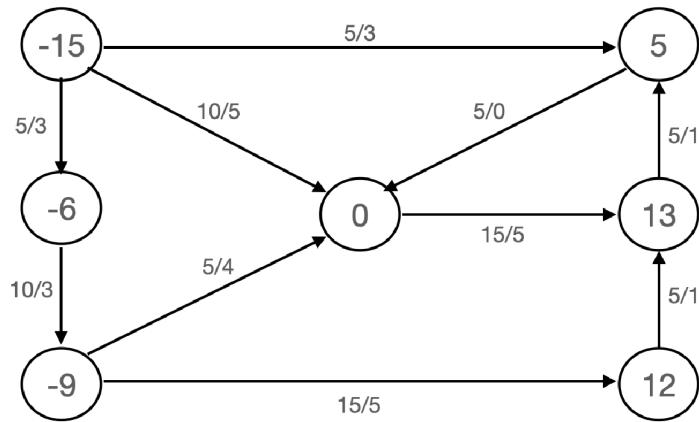
Dynamic Programming (4 pts)

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 $\text{Opt}(w,i)$ for all w , for a given i , we need all the values $\text{Opt}(w, i-1)$ 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.

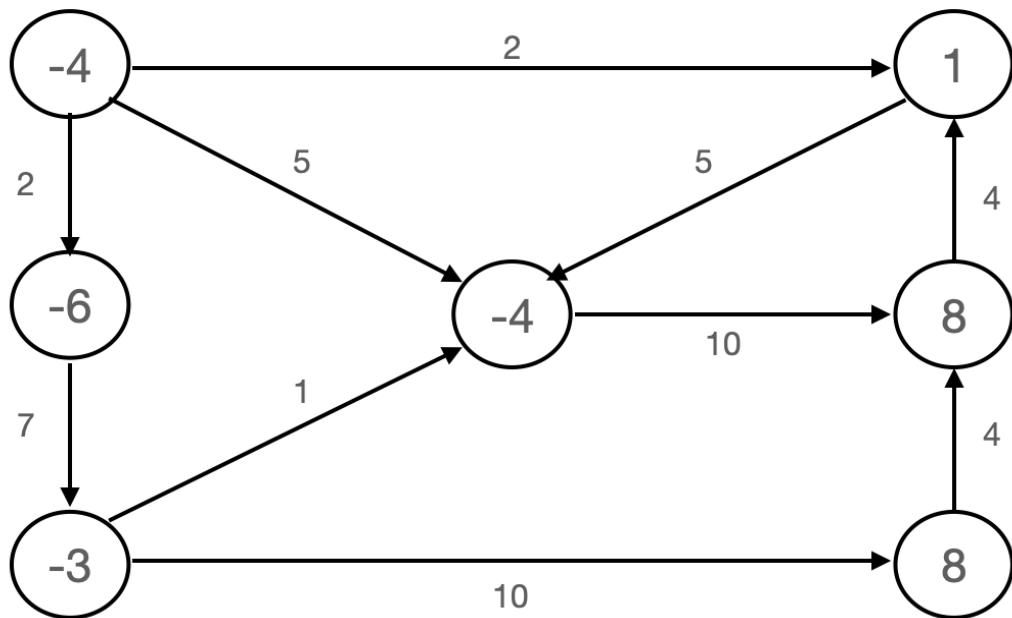


- a. Reduce the Feasible Circulation with Lower Bounds problem to a Feasible Circulation problem without lower bounds. (8 pts)

Draw G'

If demand is correct, get 4 pts

If flow on edges are correct, get 4 pts



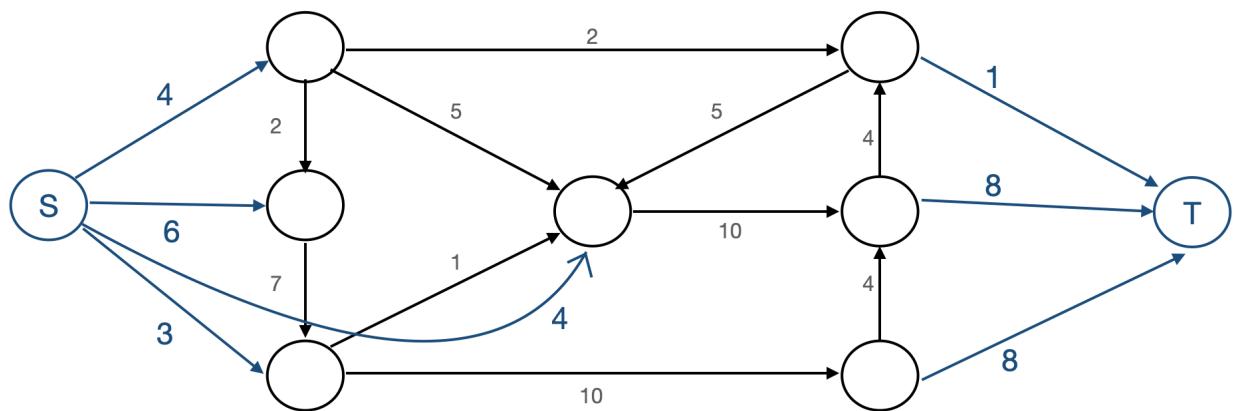
b. Reduce the Feasible Circulation problem obtained in part a to a Maximum Flow problem in a Flow Network.(8 pts)

Add S and T (2 pts)

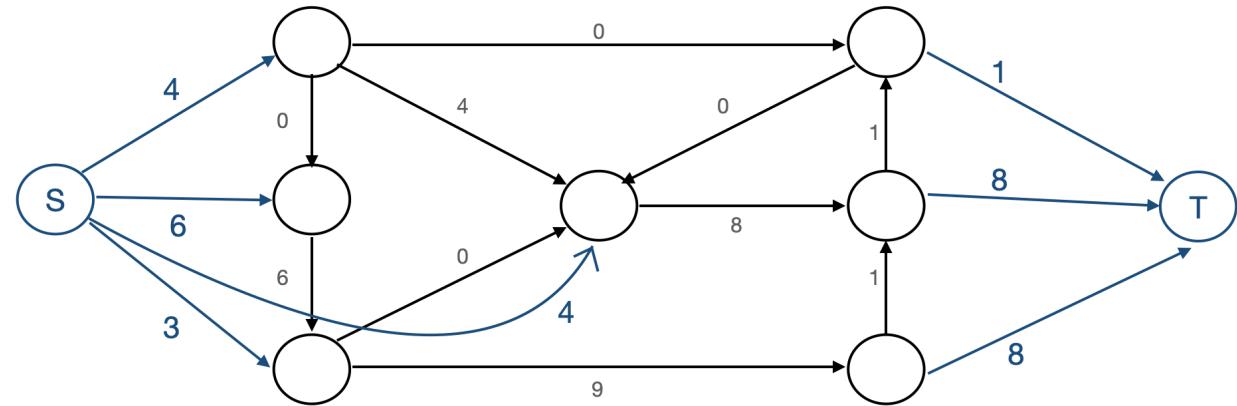
Connect S with correct nodes (2 pts)

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)

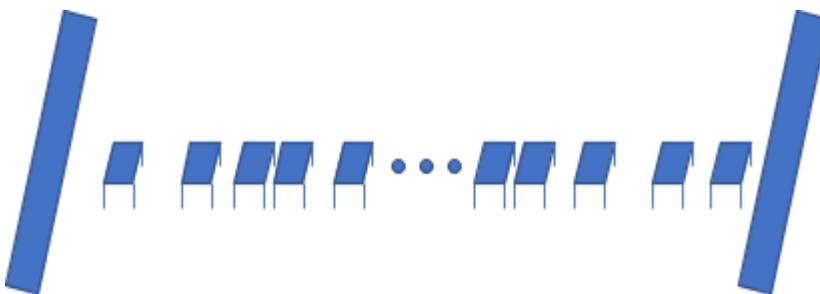


This is the max flow in Flow network. (2 pts)

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 $\{A, \dots, N\}$)
2. For every platform node $\{A, \dots, N\}$ create a dummy node $\{A', B', \dots, N'\}$ respectively. Connect all the platform nodes $\{A, \dots, N\}$ to the corresponding dummy nodes $\{A', \dots, N'\}$ so that the edge $A \rightarrow A'$ 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 $\{A, B, \dots, N\}$.
4. Connect the dummy nodes $\{A', B', \dots, N'\}$ 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 $\{A', B', \dots, N'\}$

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 $p \rightarrow p'$. 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 $p \rightarrow p'$ 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 i^{th} 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 i^{th} day" instead of **until** the i^{th} 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)

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

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
    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. $[w_1, w_2, \dots, w_n]$, 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)

$\text{OPT}(w,k) = \min$ 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 $w \geq w_k$

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

Else

$\text{OPT}(w,k) = \text{OPT}(w, k-1)$

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)

$\text{OPT}(w,0) = \infty$ for all $w > 0$

$\text{OPT}(0,0) = 0$

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

$\text{OPT}(W,n)$

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

$OPT(w,0) = \infty$ 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)

$\text{OPT}(w) = \min$ packages needed to make up a capacity of exactly w , (by considering all packages)

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

$$\text{OPT}(w) = 1 + \min_{\{j=1 \text{ to } n, w \geq w_j\}} \text{OPT}(w - w_j)$$

(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)
 $\text{OPT}(w) = \min\{\text{OPT}(w), 1 + \text{OPT}(w - w_j)\}$
3. Return ans (mentioned below for clarity)

Make sure you specify

- base cases and their values (2 pts)

$$\text{OPT}(0) = 0$$

$\text{OPT}(w) = \infty$ 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. $\text{opt}(n)$, or $\text{opt}(0,n)$, etc.) (1 pt)

$$\text{OPT}(W)$$

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.

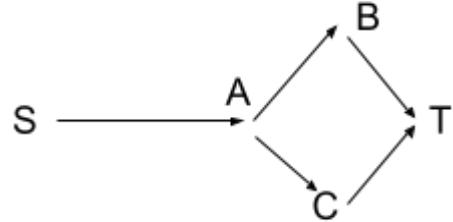
1) Mark the following statements as **TRUE** or **FALSE**. No need to provide justification.

[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.

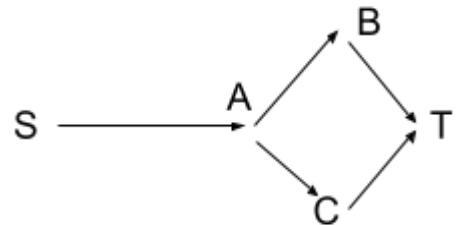


[TRUE/FALSE] (2 pts)

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.

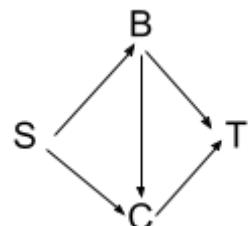
Note: an edge on the min cut (A, B) refers to an edge that carries flow out of A (where the source S is) and into B (where the sink T is).

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.

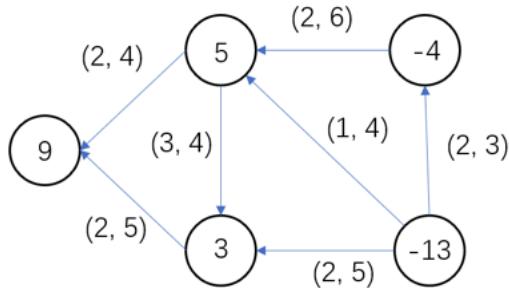
[TRUE/FALSE] (4 pts)

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.

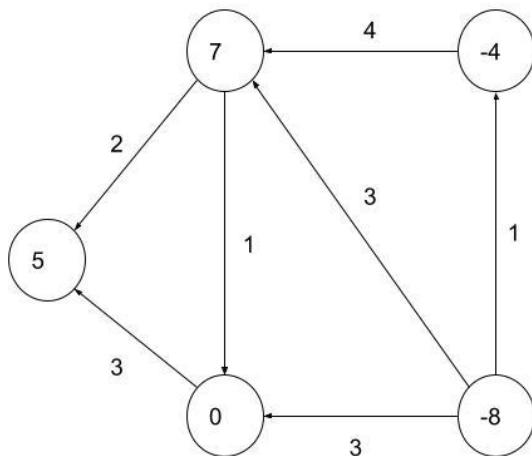


- a. Reduce the Feasible Circulation with Lower Bounds problem to a Feasible Circulation problem without lower bounds. (8 pts)

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:



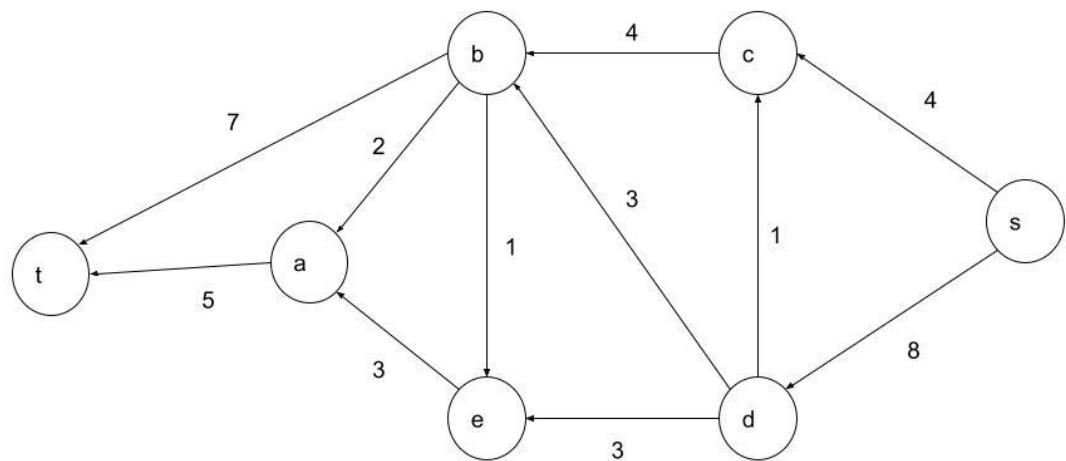
Rubrics:

- (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)

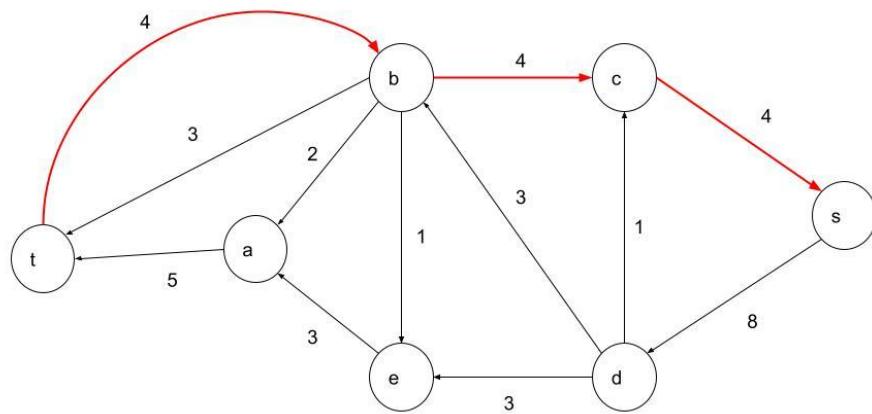
c. Using the solution to the resulting Max Flow problem explain whether there is a Feasible Circulation in G. (4 pts)

Solution:

Candidate Solution 1:

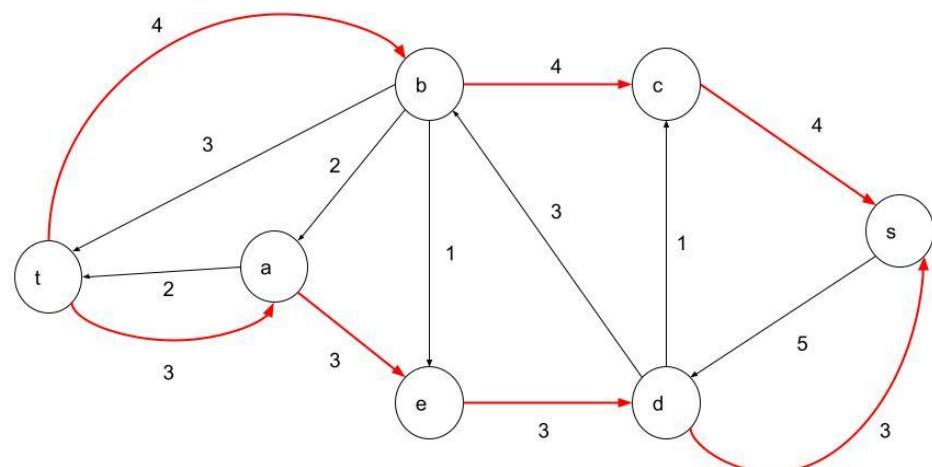
First Augmenting Path: s -> c -> b -> t with flow = 4

Residual Graph 1:



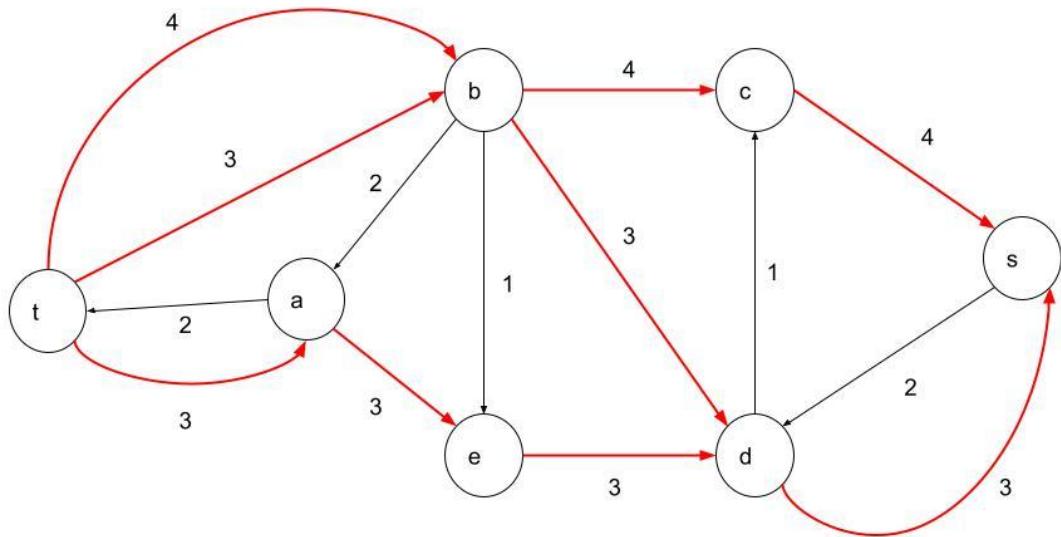
Second Augmenting Path: s -> d -> e -> a -> t with flow = 3

Residual Graph 2:



Third Augmenting path: $s \rightarrow d \rightarrow b \rightarrow t$ with flow 3

Residual Graph 3:



Candidate Solution 2:

First Augmenting Path: $s \rightarrow c \rightarrow b \rightarrow t$ with flow = 4

Second Augmenting Path: $s \rightarrow d \rightarrow b \rightarrow a \rightarrow t$ with flow = 2

Third Augmenting Path: $s \rightarrow d \rightarrow b \rightarrow t$ with flow = 1

Fourth Augmenting Path: $s \rightarrow d \rightarrow e \rightarrow a \rightarrow t$ with flow = 3

Candidate Solution 3:

First Augmenting Path: $s \rightarrow d \rightarrow e \rightarrow a \rightarrow t$ with flow = 3

Second Augmenting Path: $s \rightarrow d \rightarrow b \rightarrow a \rightarrow t$ with flow = 2

Third Augmenting Path: $s \rightarrow d \rightarrow c \rightarrow b \rightarrow t$ with flow = 1

Fourth Augmenting Path: $s \rightarrow c \rightarrow b \rightarrow t$ with flow = 3

Fifth Augmenting Path: $s \rightarrow d \rightarrow b \rightarrow t$ 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:

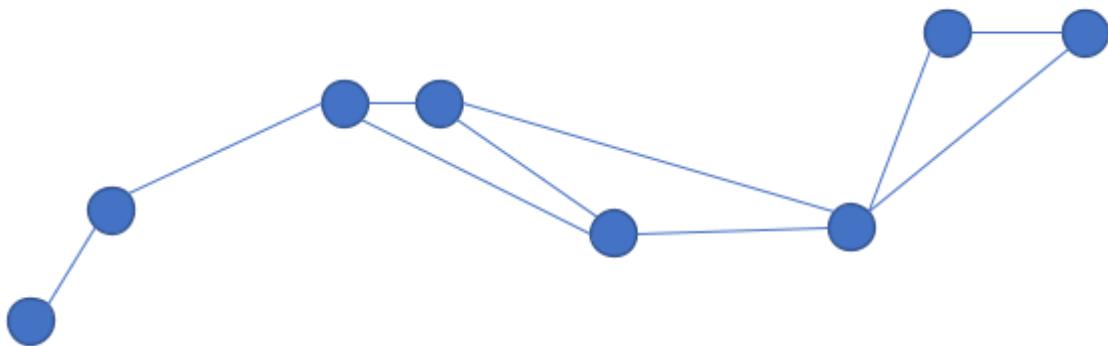
2 pts: Finding correct Max-Flow and presenting their appropriate residual graphs according to the sequence of augmenting paths that the student has chosen

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 S_i students assigned to each room i . The copy center made some big mistakes and instead of delivering S_i papers they delivered R_i papers to each room i , where R_i was higher than S_i 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 S_i (= papers needed in total) OR Summation of R_i (= papers available in total) OR $\sum_{\{i \text{ where } S_i - R_i > 0\}} (S_i - R_i)$ (= sum of shortages of papers) OR $\sum_{\{i \text{ where } R_i - S_i > 0\}} (R_i - S_i)$ (= 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 $S \rightarrow L_i$ 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 $L_i \rightarrow T$ 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 $\sum R_i$ at T and $-\sum R_i$ at S. Alternatively (instead of adding the demands at S,T), adding a T \rightarrow 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 $\sum R_i$.

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 $\sum R_i$

Solution 3: Relative to solution 2 , we can get rid of the lower bounds on edges $L_i \rightarrow T$. In this case, we have (In addition to the common steps laid out in the beginning) a demand of $S_i - R_i$ at location L_i and sink T with a demand (sum of R_i - sum of S_i). We have edges $L_i \rightarrow T$ of capacity $R_i - S_i$ wherever it is positive (but okay to add edges from all the $L_i \rightarrow T$ 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 R_i - sum of S_i). This network must have a max flow of $\sum_{\{i \text{ where } R_i - S_i > 0\}} (R_i - S_i)$.

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 $\sum_{\{i \text{ where } S_i - R_i > 0\}} (S_i - R_i)$ (note this crucial difference with respect to max flow value of Solution 4).

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

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 $S \rightarrow L_i$ nodes with capacity R_i . Create a sink T, and connect $L'_i \rightarrow T$ with capacity S_i . Each L_i and its copy L'_i should have edges both ways. But for $i \neq j$, $L_i \rightarrow L'_j$ 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 (S_i)

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 $F(L_i \rightarrow L_j) - F(L_j - L_i)$ 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 your 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,
- 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

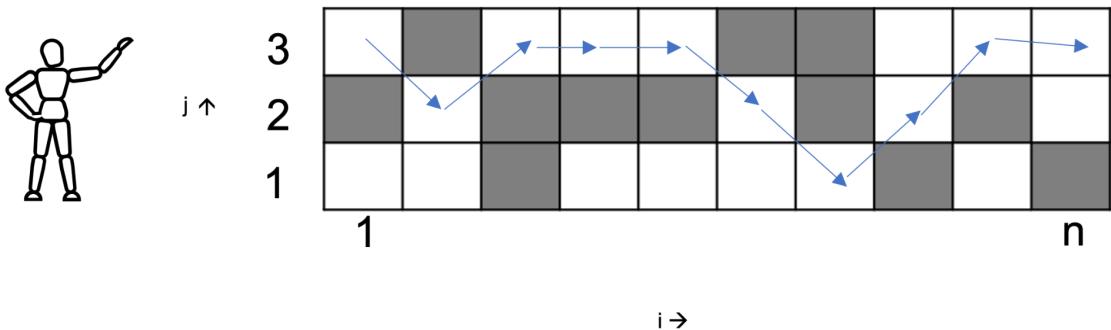
for this error will be reduced from 3.5 to 2.5 IF the claim/final answer is correctly given in terms of the max flow value of the resultant network).

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 $\text{BadTile}(3,n)$ 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)

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

OR

$\text{OPT}(j, i)$ = 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:

$$\begin{aligned} \text{OPT}(j, i) &= 0 && \text{if } \text{BadTile}(j, i) = 1; \\ &= \text{OPT}(j, i-1) + \text{OPT}(j+1, i-1) && \text{if } j = 1; \\ &= \text{OPT}(j, i-1) + \text{OPT}(j+1, i-1) + \text{OPT}(j-1, i-1) && \text{if } j = 2; \\ &= \text{OPT}(j, i-1) + \text{OPT}(j-1, i-1) && \text{if } j = 3; \end{aligned}$$

Variant 2 :

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

Variant 3 :

$$\begin{aligned} \text{OPT1}[i] &= 0 && \text{if BadTile} = 1 \\ &= 1 && \text{if}(i == 1) \\ &= \text{opt2}[i-1] + \text{opt1}[i-1] && \text{otherwise} \end{aligned}$$

$$\begin{aligned} \text{OPT2}[i] &= 0 && \text{if BadTile} = 1 \\ &= 1 && \text{if}(i == 1) \\ &= \text{opt2}[i-1] + \text{opt1}[i-1] + \text{opt3}[i-1] && \text{otherwise} \end{aligned}$$

$$\begin{aligned} \text{OPT3}[i] &= 0 && \text{if BadTile} = 1 \\ &= 1 && \text{if}(i == 1) \\ &= \text{opt2}[i-1] + \text{opt3}[i-1] && \text{otherwise} \end{aligned}$$

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. $\text{opt}(n)$, or $\text{opt}(0,n)$, etc.) (1 pt)

For Variant 1:

Given BadTile (3,n)

Initialize OPT(3,n) with 0 for each element

$\text{OPT}(1,1) = 1$ if BadTile (1,1) equals to 0, else 0

$\text{OPT}(2,1) = 1$ if BadTile (2,1) equals to 0, else 0

$\text{OPT}(3,1) = 1$ 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:

$\text{OPT}(j, i) = 0$

 if j equals to 1:

$\text{OPT}(j, i) = \text{OPT}(j, i-1) + \text{OPT}(j+1, i-1)$

 if j equals to 2:

$\text{OPT}(j, i) = \text{OPT}(j, i-1) + \text{OPT}(j+1, i-1) + \text{OPT}(j-1, i-1)$

 if j equals to 3:

$\text{OPT}(j, i) = \text{OPT}(j, i-1) + \text{OPT}(j-1, i-1)$

return $\text{OPT}(1, n) + \text{OPT}(2, n) + \text{OPT}(3, n)$

For Variant 2:

Initialize OPT(5,n)

$\text{OPT}(0,i) = 0$ //outside the grid

$\text{OPT}(4,i) = 0$ // outside the grid

For i in 1 to n:

 For j =1 to 3:

$$\text{OPT}(j, i) = \text{OPT}(j,i-1) + \text{OPT}(j+1,i-1) + \text{OPT}(j-1,i-1)$$

return $\text{OPT}(1, n)+\text{OPT}(2, n)+\text{OPT}(3, n)$

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 $\text{OPT}(1, n)+\text{OPT}(2, n)+\text{OPT}(3, n)$

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. $[w_1, w_2, \dots, w_n]$, 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 out of a maximum 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 online version of the exam, please refer to it for solutions and rubrics.

Solution 1:

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

$\text{OPT}(w,k) = \min$ 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)

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

If $w \geq w_k$

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

Else

$\text{OPT}(w,k) = \text{OPT}(w, k-1)$

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

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

- 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)

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

$\text{OPT}(0,0) = 0$

$\text{OPT}(w,k) = \text{infinity}$ 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. $\text{opt}(n)$, or $\text{opt}(0,n)$, etc.) (1 pt)

$\text{OPT}[W, n]$ in the first definition

$\text{OPT}[W, 1]$ 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)

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

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 $w \geq w_k$

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

Else

$\text{OPT}(w,k) = \text{infinity}$

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

6. Return ans (mentioned below for clarity)

Make sure you specify

- base cases and their values (2 pts)

$\text{OPT}(w,0) = \text{inf}$ for all $w > 0$

$\text{OPT}(0,0) = 0$

$\text{OPT}(w,k) = \infty$ 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. $\text{opt}(n)$, or $\text{opt}(0,n)$, 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