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, nlog(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->B)=100, (A->C)=51, (B->D)=1, (C->D)=51

Shortest path from A to D is A->B->D Path length=101
After squaring this path length become 100^2+1^2=10001
However, A->C->D has path length 51^2+51^2=5202<10001
Thus A->C->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)<=P(n) for n=1, 2, 3, ….We prove this by induction

It is obvious that G(1)<=P(1)

Given G(n-1)<=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)<=P(n-1)<=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)>=P(n-1)>=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>=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]<j, 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]>=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 ware 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)

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 $\times$ 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

Date: Oct 17, 2005.

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 $i = 1, 2, ..., n$, 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 $O(n \log g)$. 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 $S = \{O_1, ..., O_m\}$ is the objects the optimal solution with weight $w_1, ..., w_m$. 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 $\sum_{k=1}^{m} w_k/W_kV_k + min\{w_i', w_j\}/W_iV_i - min\{w_i', w_j\}/W_jV_j$. $w_i' > 0, w_j > 0$ 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 **Ω**(f(n)), then T(n) is **Θ**(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(lgn) 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 G1 and G2 may get partial credit up to 6 marks. Because that is a mistake, nodes in G1 can not exist in G2 and vice versa.

- Any one says it is false and give correct disprove (counter example) get 10 out of 10.

5) 10 pts

Suppose that you divided a graph G = (V,E) into two sub graphs G1 = (V1,E1) and G2 = (V2, E2). And, we can find M1 which is a MST of G1 and M2 which is MST of G2. Then, M1 U M2 U {minimum weight edge among those connecting two graph G1 and G2} 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 V1={A,B} with edge AB, V2={C,D} with edge CD. Then the weight of M1 U M2 U 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 G1 and G2 may get partial credit up to 6 marks. Because that is a mistake, nodes in G1 can not exist in G2 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 $p_1, p_2, \ldots, p_n$, and n working-clothes with height sizes of $s_1, s_2, \ldots, s_n$. The problem is to find best matching strategy such that we minimize the following average differences.

$$\frac{1}{n}\Sigma\left|p_i - s_i\right|$$

Present an algorithm to solve this problem along with its proof of correctness.

Solution: Sort the height of workers in increasing order $p_1 \le p_2 \le \ldots \le p_n$

Sort the height size of clothes in increasing order $s_1 \le s_2 \le \ldots \le s_n$

Match h_i with s_i is the best matching strategy such that we minimize the following average differences.

$$\frac{1}{n}\Sigma\left|p_i - s_i\right|$$

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 p1, . . . , pn, s1, . . . , sn for which it does not

produce an optimal solution. Let the optimal solution be T = {(p1, s_(1)), . . . , (pn, s_(n))}, and let the output of the greedy algorithm be G = {(p1, s1), . . . , (pn, sn)}. Beginning with p1, compare T and G. Let pi be the first person who is assigned different cloth in G than in T. Let sj be the pair of cloth assigned to pi in T. Create solution T' by switching the cloth assignments of pi and pj . By the definition of the greedy algorithm, si is equal or greater than sj . 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(nlog n) and hence the running time is O(nlog 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 $x_1, x_2, \ldots, x_n$, 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 $x_1, x_2, \ldots, x_n$.

Solution: We solve this question by dynamic programming. Denote OPT(i) as the possibility of paying value i using coins of denominations $x_1, x_2, \ldots, x_n$. Then OPT(i) is true if and only if we can paying $i-x_1$, or $i-x_2$, …or $i-x_n$ using coins of denominations $x_1, x_2, \ldots, x_n$. In other words,
$OPT(i)= OPT(i- x_1) \vee OPT(i- x_2) \vee \ldots OPT(i- x_n)$
Following the recursive relation with initial value $OPT(x_1)=\ldots =OPT(x_n)$=true we compute the value of OPT(v). If OPT(v) is true, then we can change for *v* using coins of denominations $x_1, x_2, \ldots, x_n$, otherwise we can not.

Clearly to compute OPT(v) we need an array of size v, and each time when we compute 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;
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(nlgn)

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 c$n^{1/2}$ so again, at this level we spend cn time
Same goes with every other level. There are lgn levels in the tree because at every we take the square root of n. So the total run time is $\theta(nlgn)$

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

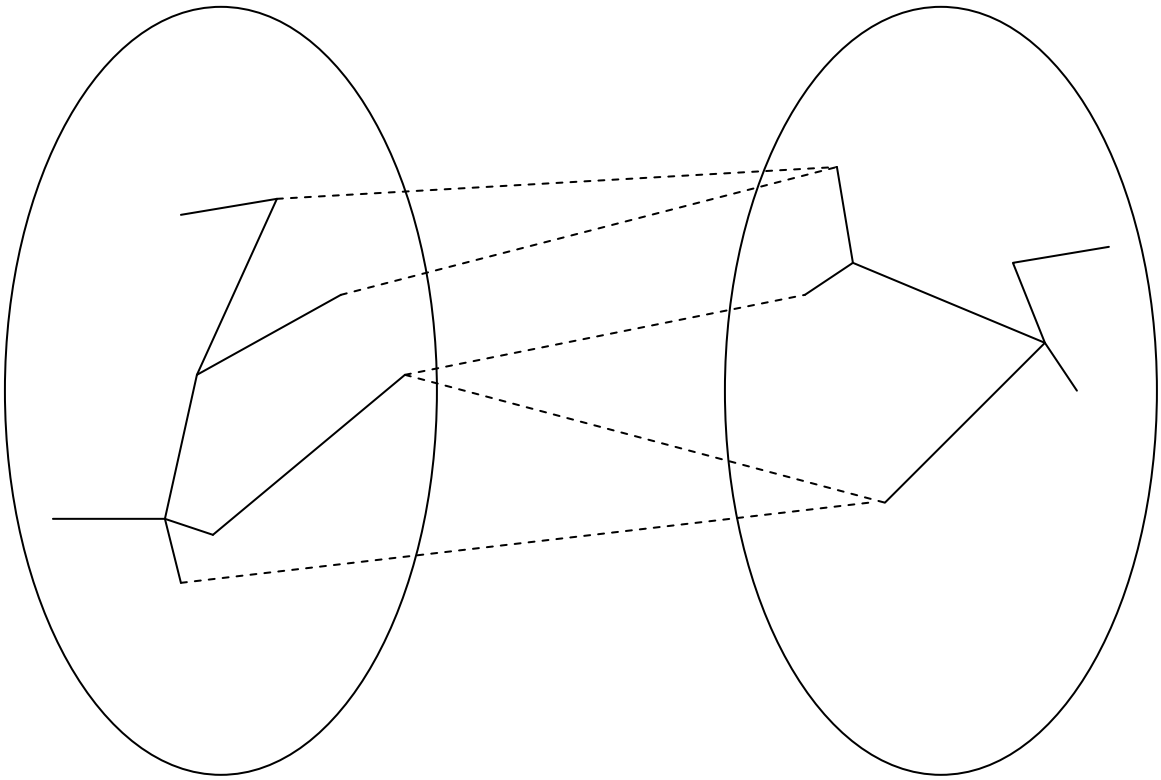
(b) Give a greedy algorithm and use an interchange argument to prove that it works for a single server as long as no new customers arrive.

(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 G1 = (V1, E1) and G2 = (V2, E2). Also assume that we have T1 which is a MST of G1 and T2 which is MST of G2. Now consider a new graph G = (V, E) such that V = V1 ∪ V2 and E = E1 ∪ E2 ∪ E3 where E3 is a new set of edges that all cross the cut (V1, V2). Following is an example of what G might look like.



The dashed edges are E3, the solid edges in G1 (on the left) are T1, and the solid edges in G2 (on the right) are T2.

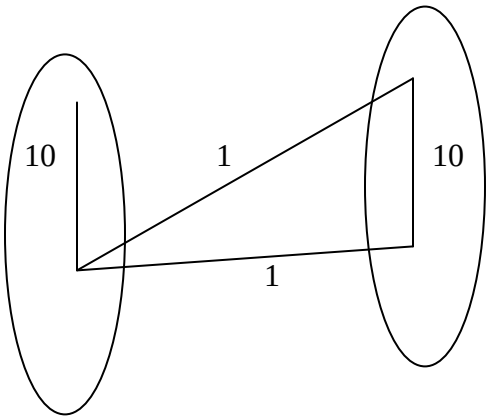
Now assume we want to fine a MST of the new graph. Consider the following algorithm:

Maybe-MST(T1, T2, E3)
e_{min} = a minimum weight edge in E3
T = T1 U T2 U { e_{min}}
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.
$M = Max\ (x_i + x_{i+1});\ i=1\ to\ n-1$

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

x

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.

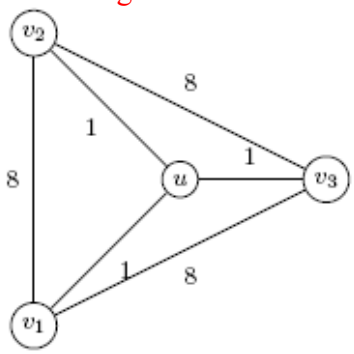
[TRUE]
A BFS tree is a spanning tree

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 $V1$ and $V2$ such that $|V1|$ and $|V2|$ differ by at most 1. Let $E1$ be the set of edges that are incident only on vertices in $V1$, and let $E2$ be the set of edges that are incident only on vertices in $V2$. Recursively solve a minimum spanning tree problem on each of the two subgraphs $G1 = (V1, E1)$ and $G2 = (V2, E2)$. Finally, select the minimum-weight edge in E that crosses the cut $(V1, V2)$, 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 ith element of set A, and let b_i be the ith 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 ai^{bp} and aj^{bq}. We want to show that it is no worse to include these terms in the payoff than to include ai^{bq} and aj^{bp} , that is, $ai^{bp}\ aj^{bq} \geq ai^{bq}\ aj^{bp}$.

Since A and B are sorted into monotonically decreasing order and $i < j$, $p < q$, we have $ai \geq aj$ and $bp \geq bq$. Since ai and aj are positive and $bp - bq$ is nonnegative, we have $ai^{bp-bq} \geq aj^{bp-bq}$.

Multiplying both sides by $ai^{bq}\ aj^{bq}$ yields $ai^{bp}\ aj^{bq} \geq ai^{bq}\ aj^{bp}$.

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(nlogn).

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 ≥ 1, and ε > 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) $\log(\log n)=O(\log n)$ => $k\log(\log n)=O(\varepsilon\log n)$ => $\log(\log^k n)=O(\log n^\varepsilon)$ => $\log^k n=O(n^\varepsilon)$

(2) $-1<=\sin n<=1$, 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]; ... ;A[n] is unimodal. In other words for some unknown index p between 1 and n, we have
A[1] < A[2] < ... < A[p] and A[p] > A[p + 1] > ... > 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 FindPeak() be the function of finding the peak in A . Then 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 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 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 mid=3. We find A[mid-1]<A[mid]<A[mid+1]. A common but wrong practice here is to continue search only in the A[1: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*<*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' = $a_1a_2\ldots a_{i-1}a_{i+1}\ldots a_{n-1}$
Without loss of generality, suppose we delete S[j] rather than S[i], the we have
$S_1 = a_1a_2\ldots a_{j-1}a_{j+1}\ldots a_{n-1}$ (j!=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!=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, p1, p2, …, p_{k+1}, in S.
If p'=pi (1<=i<=k+1), then N'=N; if p'<p1 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<=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.

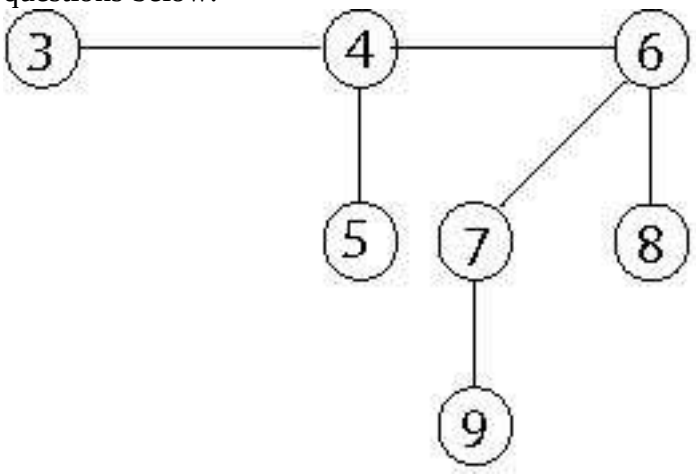
[TRUE]
The complexity of the recursion given by $T(n) = 4T(n/2) + cn^2$, for some positive constant c, is $O(n^2 \log n)$.

[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.

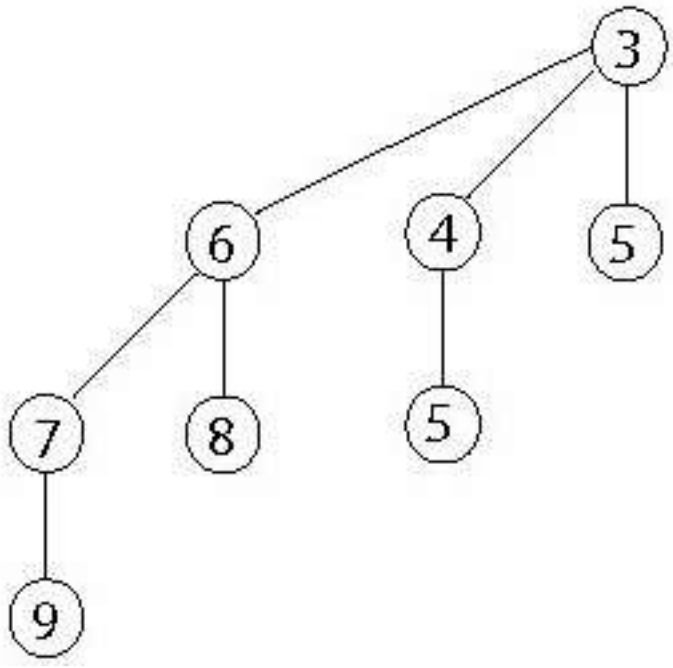
[FALSE]
Any divide and conquer algorithm will run in best case $\Omega(n \log n)$ time because the height of the recursion tree is at least $\log n$.

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?



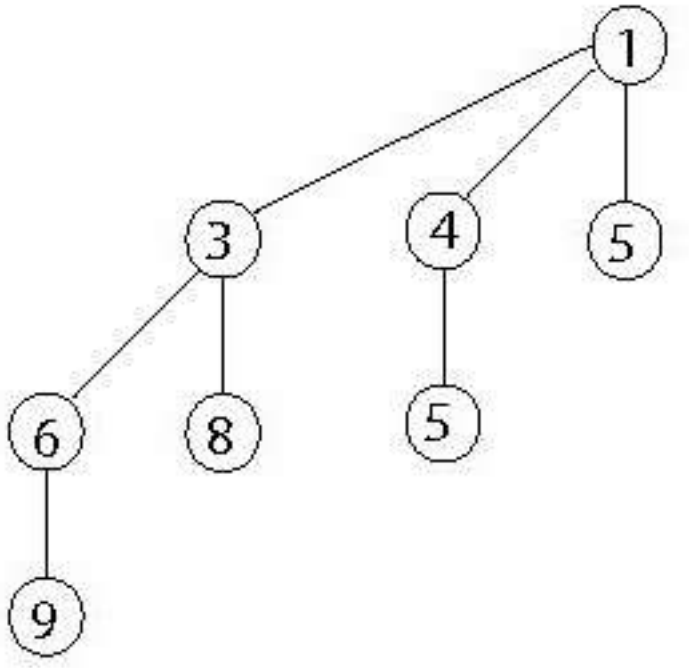
This is also a Fibonacci heap, but not binary heap.

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 $1 \le i < n$ 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 $A[i + 1..n]$.
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 $a \le b$".

When the loop completes, $a \ge b$ (since the loop condition failed) and $a \le b$ (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 $a \le b$ is also true. At the start of the loop $a < b$. Therefore, $a \le \lfloor (a + b)/2 \rfloor < b$. This means that $a \le mid < b$ such that after line 4 or line 5 in which a and b get updated $a \le b$ 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 $V[max], V[max + 1], \ldots, V[n - 1], V[n], V[1]$ 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 theorm, $a=2$, $b=5$, $f(n)=c\ n$, $degree(f(n))=1$
Since $log_2\ 5 > 1$, $T(n) = O(n^{log_a\ b}) = O(n^{log_2\ 5})$

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 theorm, $a=3$, $b=9$, $f(n)= c\ n^2$, $degree(f(n))=2$
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)};
cost((s, a))=9; cost((a, t))=16; 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:
cost((s, a))=3; cost((a, t))=4; 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.

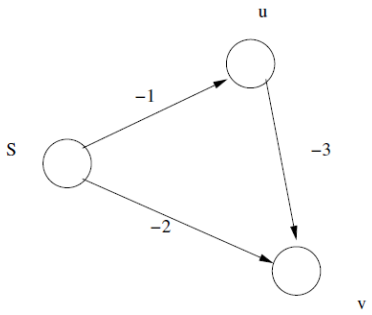
1) 20 pts
Mark the following statements as **TRUE** or **FALSE**. No need to provide any justification.

[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 $O(logn)$.

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 neceassarily 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 $M = \{m_1, m_2, ..., m_N\}$ and the set of women is $W = \{w_1, w_2, ..., w_N\}$. Consider their preference lists to have the following properties:

$$\forall w_i \in W: w_i \text{ prefers } m_i \text{ over } m_j, \forall\, j > i$$
$$\forall m_i \in M: m_i \text{ prefers } w_i \text{ over } w_j, \forall\, j > i$$

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 $i = 1, 2, \ldots, N$ 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, $a_1, a_2, \ldots, a_n$. Initial value of each container is $v_1, v_2, \ldots, v_n$ and depreciation factor of each container is $d_1, d_2, \ldots, d_n$. 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 logn), 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.
$E \leq V(V\text{-}1)/2 = O(V^2)$.

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->x->v with a total weight of 6. However, if we now add 2 to every edge weight, the path u->x->v will have a total weight of 10 while the path u->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

$(p_j + \min(p_k; p_l)) - (p_k + p_l) = p_j - \max(p_k , p_l) >= 0$

since p_j 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 $\{1, \ldots, n\}$ 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 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 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 $i = 1, 2, \ldots, N$. 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 $d*(i-1) + 2$, $d*(i-1) + 3$, ... $d*(i-1) + d + 1$, 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,

$$n = 1 + d + d^2 + d^3 + \ldots + d^h$$

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 $f(n)= 5n^3 2^n + 6n^2 3^n$ is $O(n^3 2^n)$.

[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, Fibbonacci implementation is the best one among Binary, Binomial, and Fibbonacci.

[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 < 2lg(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(mlog n)

4) 20 pts
Suppose you are given two set *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 $\sum^{n}_{i=1} b_i \log a_i$. 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 $\{a_1,\ldots,a_n\}$ and $\{b_1,\ldots b_n\}$ 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:

$$a_i > a_j, b_i < b_j \Leftrightarrow a_i^{b_j - b_i} > a_j^{b_j - b_i} \Leftrightarrow a_i^{b_j} a_j^{b_i} > a_j^{b_j} a_i^{b_i} \Leftrightarrow \log(a_i^{b_j} a_j^{b_i}) > \log(a_j^{b_j} a_i^{b_i})$$
$$\Leftrightarrow b_j \log a_i + b_i \log a_j > b_j \log a_j + b_i \log a_i$$

which contradicts that $\{a_1,\ldots,a_n\}$ and $\{b_1,\ldots b_n\}$ is an optimal solution

Running time: Sorting takes O(nlog n). Hence the running time of our algorithm is O(nlog 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→t* consist of two edges, each with cost 1, and there is also an edge *e=(s,t)* in G with 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 $P_1,\ldots,P_k$ 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],….,s[n]. For the sub string s[1],….,s[m], we construct the table P as follows: if s[1],….,s[m] is composed of a sequence of valid words, then P[m]=true; otherwise P[m]=false. The recursive relation is as follows:

$$P[0] = true$$

$$P[m] = \vee_{0 \le i \le m-1} (P[i] \wedge dictionary \ \ (s[i+1],..., \ s[m]))$$

If P[m] is true, we denote q[m]=j where j is such that $P[j] \wedge dictionary(s[j+1],..., s[m])$ 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]],…,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 $m \le n$. 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 $quality(Y_{i+1}, k) = 0$ if $dictionary(Y_{i+1}k) = false$ and $quality(Y_{i+1}, k) = 1$ if $dictionary(Y_{i+1}k) = true$.

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 $A_i[1,\ldots.n]$. 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(nlog 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 v1 and v2 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) = Θ(n log n).
False. By the Master Theorem T(n) = Θ(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).

$x^{\sqrt{\log n}}$ x^n n^{10} $n(\log n)^4$ $n^{\log n}$ x^{x^n} x^{n^x}

Solution:

$x^{\sqrt{\log n}}$ $n(\log n)^4$ n^{10} $n^{\log n}$ x^n x^{n^x} 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 usedare 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× (10 cents) + 4× (1 cents). This is 5 coins. However, we can see that 2× (7 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

$w'(u, v) = w(u, v)$, if $(u, v) \neq (x, y)$

$w'(u, v) = w(u, v) - k$, if $(u, v) = (x, y)$

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)) Î 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) Ï 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 b1, b2, . . . , bn; and n girls g1, g2, . . . , gn. Without loss of generality, assume that all the boys prefer g1 over g2, g2 over g3, and so on. Likewise all the girls most prefer b1 and least prefer bn. It is easy to verify that the pairing (b1, g1), (b2, g2), . . . , (bn, gn) is a stable matching. Suppose that there is another stable matching, including couple (bi, gj) for i != j. If there are multiple such pairs, let i be the smallest such value. Therefore j > i, because g1 to gi－1 are engaged to b1 through bi－1 respectively: so j cannot be less than i. By the definition of the problem, everyone is engaged, including gi, and thus for some k > i, our alternate pairing includes the couple (bk, gi). Note that bi is a more popular boy than bk, since we chose i to be the smallest possible value. Therefore gi prefers bi over her fiance. And because i < j, we know bi prefers gi over his fiancee. Thus bi and gi constitute a instable 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 reursively and return the average
Combine: Compute the average for the orginal 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 is 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 Fiboannci 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 (Omega) 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 $s_1 \leq s_2 \leq \cdots \leq s_n$.

Define $f_1, f_2, \cdots, f_n$ 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 $\mathrm{O(m + n)}$

(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 $\left\lceil \frac{n}{2} \right\rceil$ smaller elements to build a max heap. We use the remaining $\left\lfloor \frac{n}{2} \right\rfloor$ 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 $x < median$ we insert *x* into max heap. Otherwise we insert *x* into min heap. If $size(maxHeap) > size(minHeap) + 1$, then we ExtractMax() on max heap and insert the extracted value into the min heap. Also if $size(minHeap) > size(maxHeap)$ 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 $size(maxHeap) < size(minHeap)$ 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 key(w) ≤ key(v).
2. the element w at node i and its right sibling w' satisfies key(w) ≤ key(w').

Min-heap property: for every element v at a node i and the element w at i's parent satisfies key(w) ≤ 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 ith 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 >= 1 and $p_{k+2} >> p_{k+3} > p_{k+4} > p_{k+1} > p_{k+2}$. 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, nlog(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) => there exists at least one pair of vertices (u,v) in G with more than one shortest path.

G is a bipartite graph => it can not contain an odd cycle => all the cycle(s) have even number of nodes=> 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 log2 (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 log10 (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(nlogn)

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(logn)

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^logb(a))*(logn)^k) with k >= 0, then T(n) = theta((n^logb(a))*(logn)^(k+1))

theta(n(logn)^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 *v1 … vn* and weights *w1…wn* 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 you 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 ***exceed 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…j and the optimal box includes items i…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…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:

$M = Max\ (x_i + x_{i+1} - y_i);\ i=1 \text{ to } n-1$

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(the last element of S1) + X(the first element of S2) – Y(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 thourgh 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 2 {1, . . . , 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 form s to t. This can be done in both direct and undriected 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 $A_1 \ldots A_n$, 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 x 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:

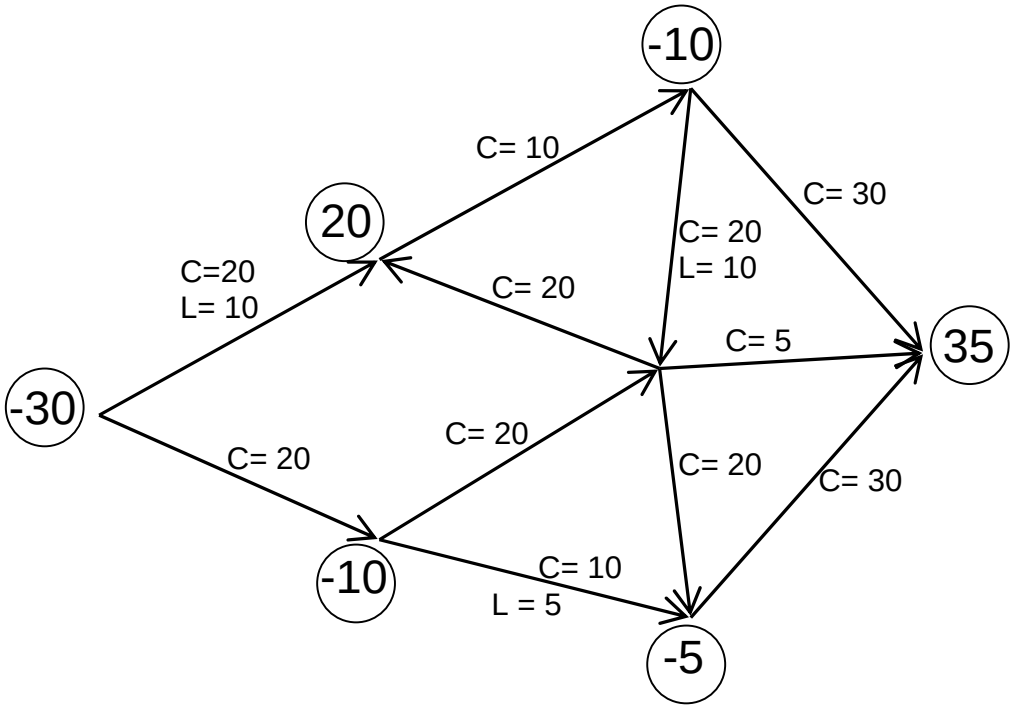
(a) Patient with type AB can receive types O, A, B, AB (universal recipient)
(b) Patient with type A can receive types O, A
(c) Patient with type B can receive types O, B
(d) Patient with type O can receive type O

Give a network flow algorithm to find the assignment such that the maximal number of patients receive blood, and prove its correctness.

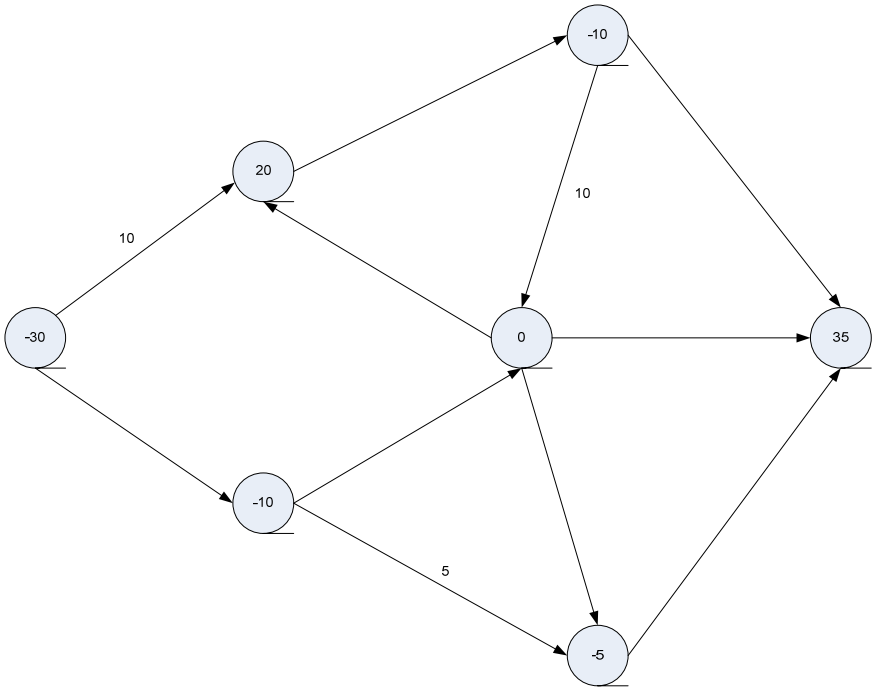
Given a set of n units of donor blood, and m patients in need of blood, each with some given blood type. We construct a network as follows. There is a source node, and a sink node, n donor nodes di, and m patient nodes pj. We connect the source to every donor node di with a capacity of 1. We connect each donor node di to every patient node pj which has blood type compatible with the donor's blood type, with a capacity of 1. We connect each patient node pj 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 di to pj in the maximum flow. The total flow into each donor node di is bounded by 1, and the total flow out of each patient node pj 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.

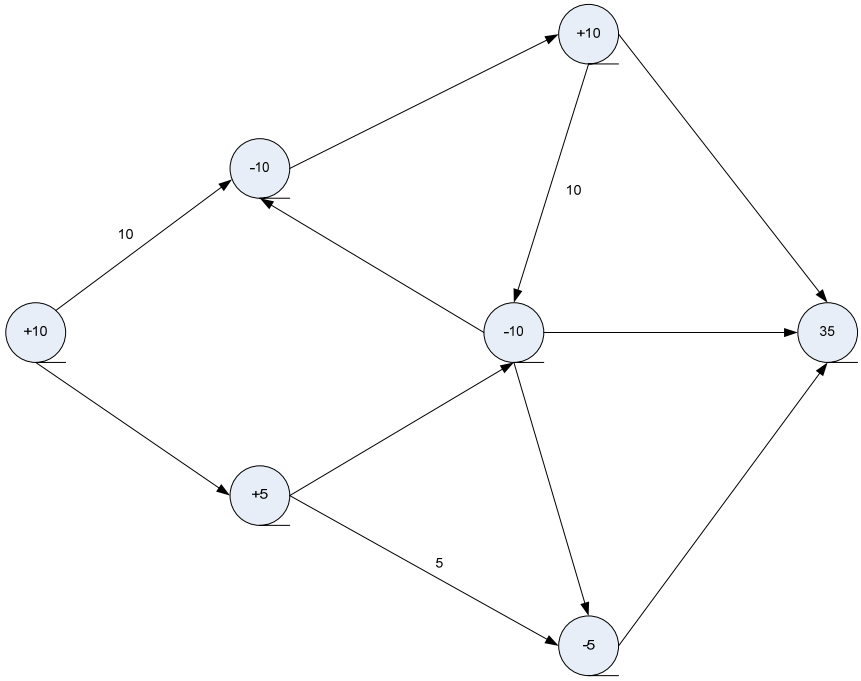
5) 20 pts
Given the graph below with demands/supplies as indicated below and edge capacities and possible lower bounds on flow marked up on each edge, find a feasible circulation. Show all your work



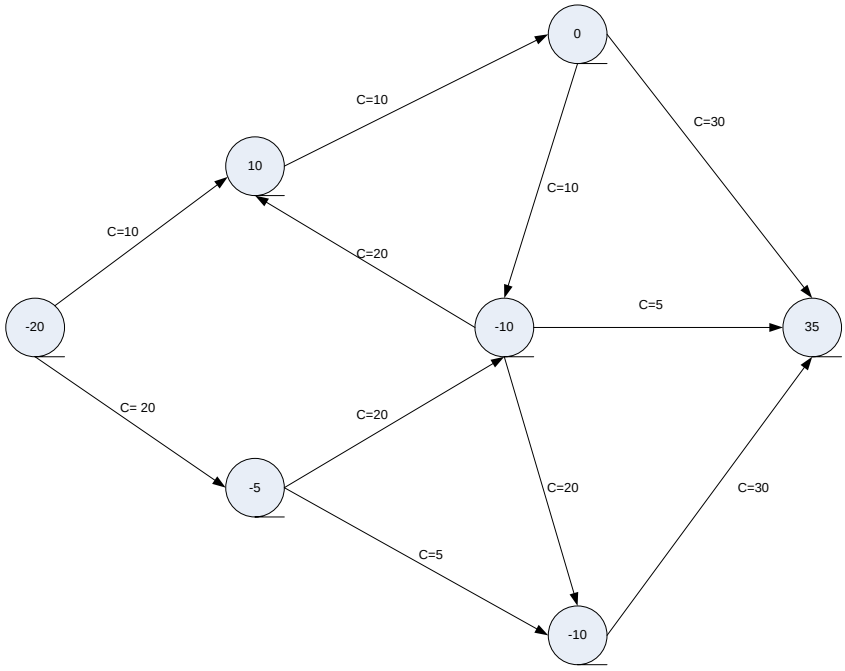
Solution to Q5
f0



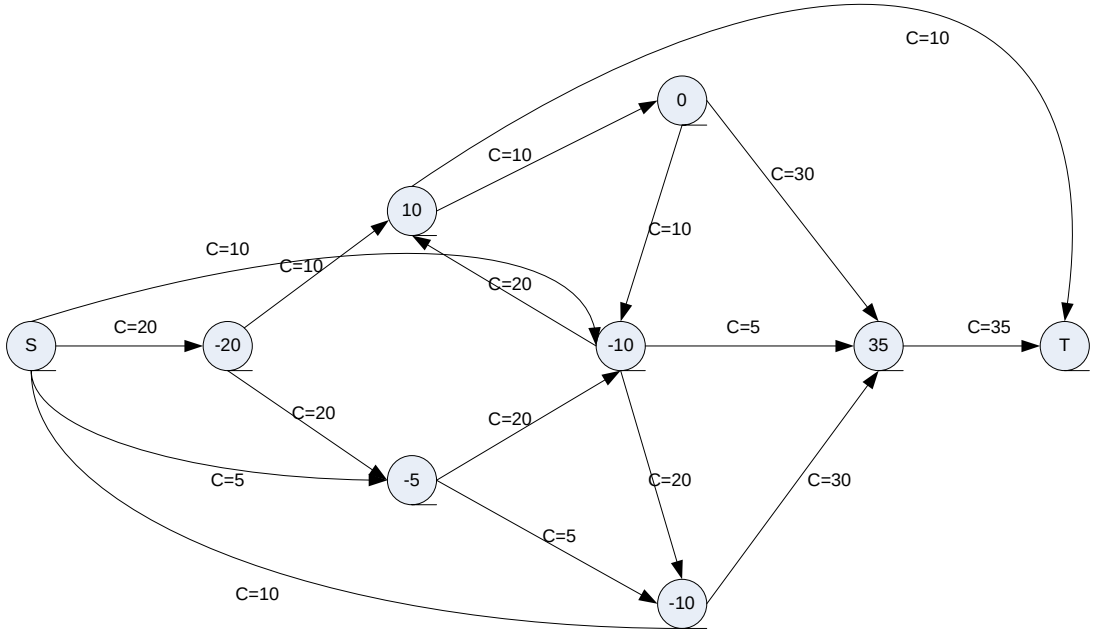
Lv



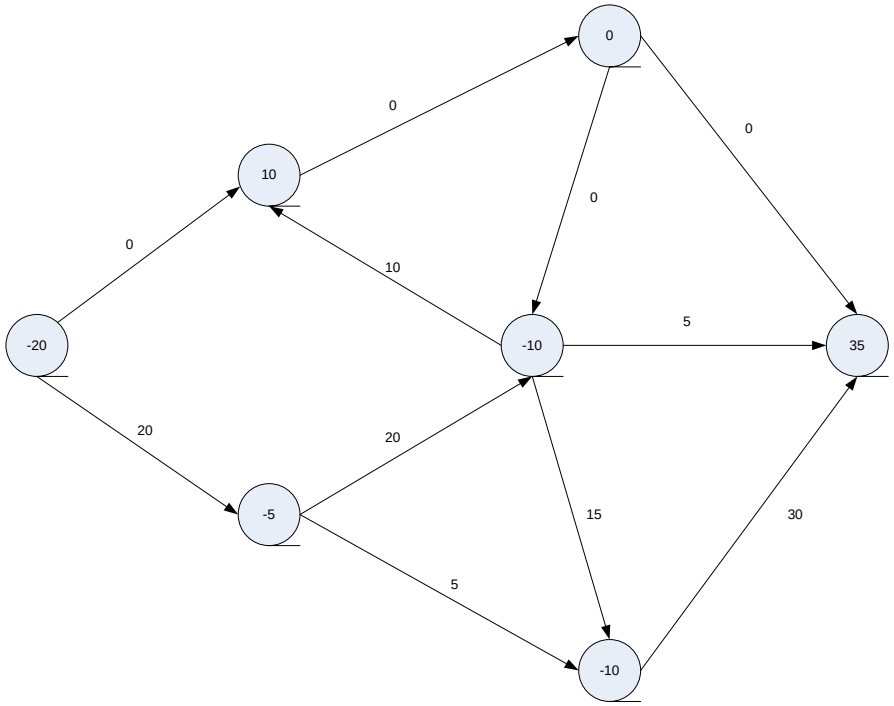
G’



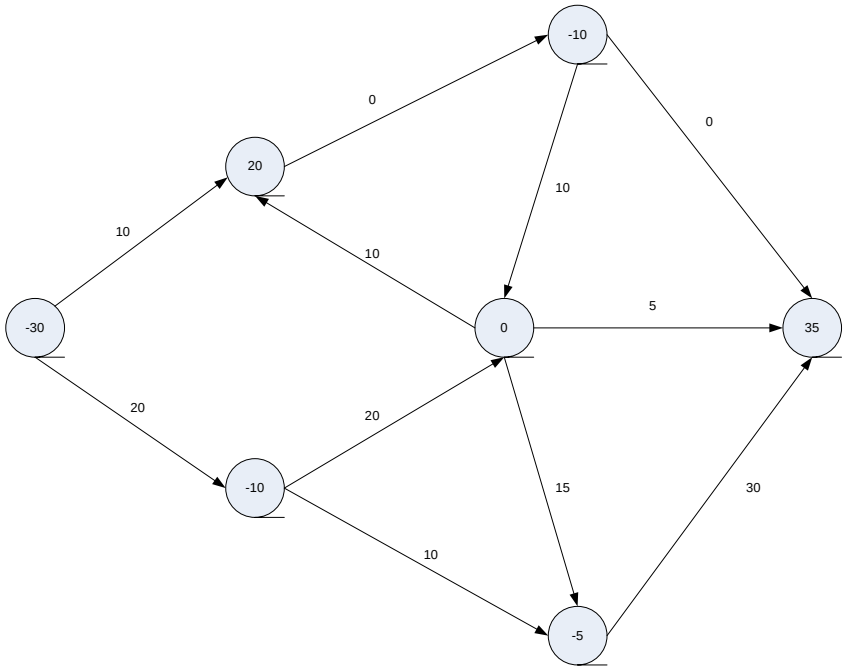
Convert G’ to a st network



f1



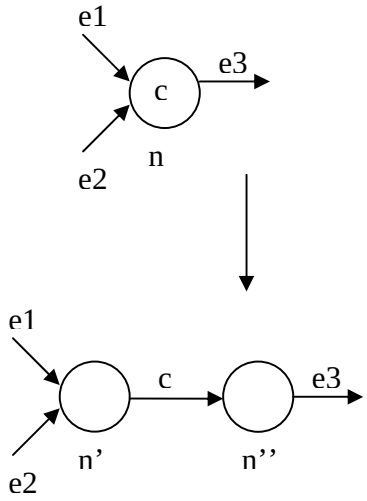
Circulation = f0+f1



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 $k_1,\ldots,k_n$ corresponding to the knights, m vertices $c_1,\ldots,c_m$ 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 $1 \le i \le n$. 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.

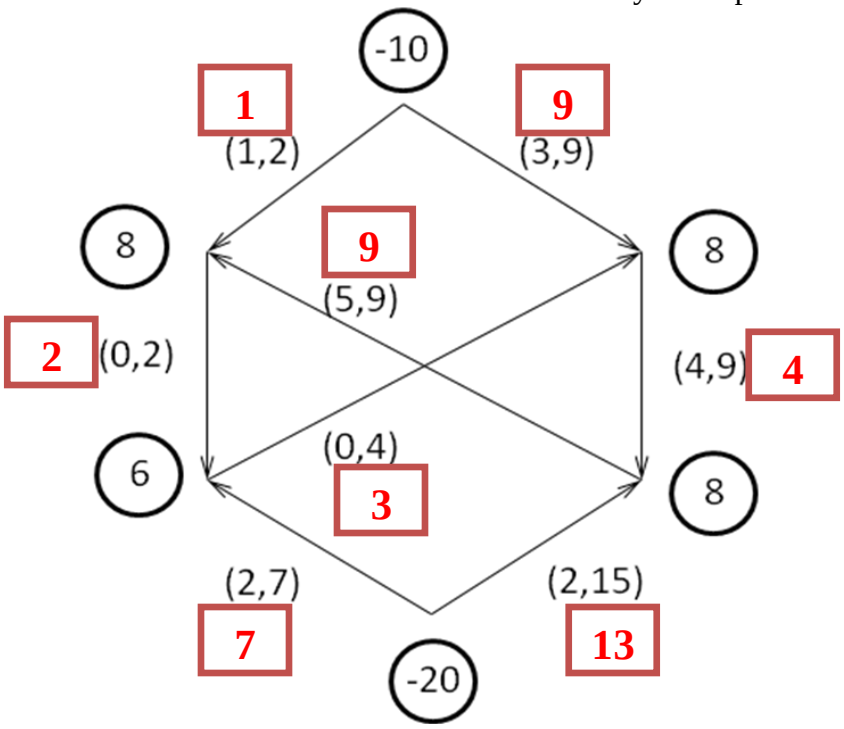
$$OPT[i] = \begin{cases} 0 & if \quad i = n \\ \min_{i \le j \le n} \left(C[i,j] + OPT[j] \right) & otherwise \end{cases}$$

Proof: The car must be rented at the starting city i and then dropped off at another city among i+1,…,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,…,n is solved, we have the optimal substructure property.

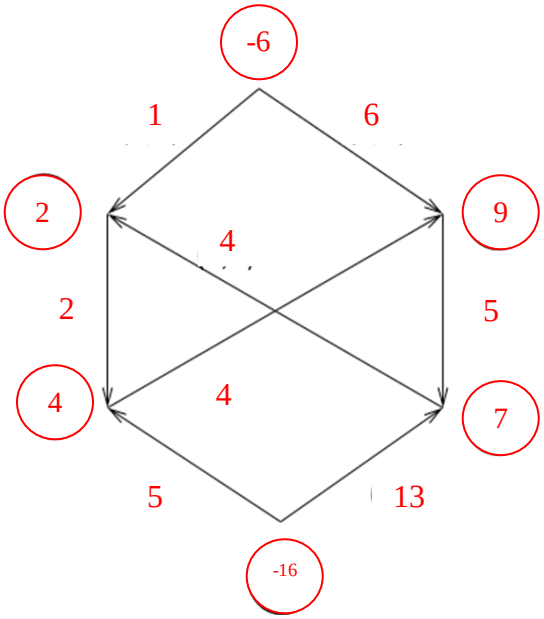
For the time complexity, there are n subproblems to be solved each of which takes linear time, O(n). These subproblems can be computed in the order OPT[n], OPT[n-1],…,OPT[1], in a linear fashion. Hence the overall time complexity is $O(n^2)$.

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/]

$T(n) = 9T(\frac{n}{5}) + n\log n$ 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($\frac{n}{b}$) + 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] < …. < A[j] 1 < j < n
- A[j] > A[j+1] > …. > A [k] j < k < n
- A[k] < A[k+1] < …. < A[n]
- A[n] < A[1]

Your task is to design a divide-and-conquer algorithm to find the maximum element of the array. You 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 and 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

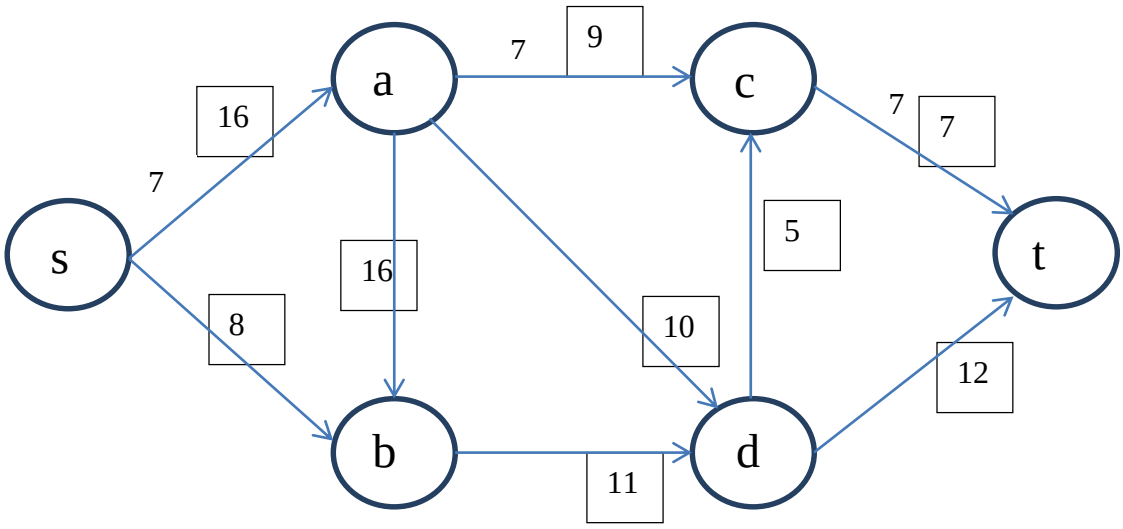
There are a series of part-time jobs lined up **one after the other**, $J_1, J_2, \ldots, J_n$. For any i^{th} part-time job, you are getting paid M_i amount of money. Also for the i^{th} part-time job, you are also given N_i, which is the number of **immediately** following part-time jobs that you cannot 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 $1 <= i <= n$, let S_i denote a solution for the set of jobs $\{J_i,....,J_n\}$, S_i maximizes the amount of money, and let 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 $\{J_{i+N_i+1},...,J_n\}$. Since S_i is optimal for the set $\{J_i,..,J_n\}$, S_i restricted to the subset $\{J_{i+N_i+1},...,J_n\}$ has to be optimal. Thus in this case, opt(i) = M_i + 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 $\{J_{i+1},...,J_n\}$. Since S_i is optimal for the set $\{J_i,..,J_n\}$, S_i restricted to the subset $\{J_{i+1}...,J_n\}$ has to be optimal. Thus in this case, opt(i) = opt$(i + 1)$.
We thus have a recurrence for all i, opt(i)= max(opt$(i + 1)$, opt$(i + N_i + 1)$+M_i).
(We understand that opt(index > n) = 0).
The boundary condition is that opt(n)=M_n and we start solving recurrence starting with opt(n), opt(n-1) and so on until opt(1). Complexity of this solution is O(n).

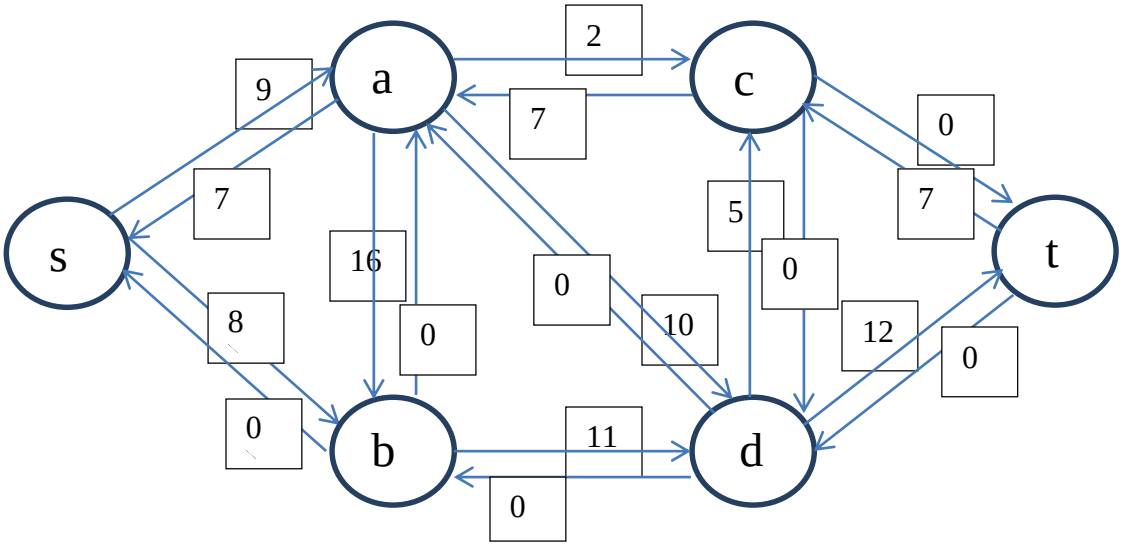
4) 16 pts

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.

a. What is the current value of flow? (2 pts)
b. Show the residual graph for the corresponding flow-network. (2 pts)
c. 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/



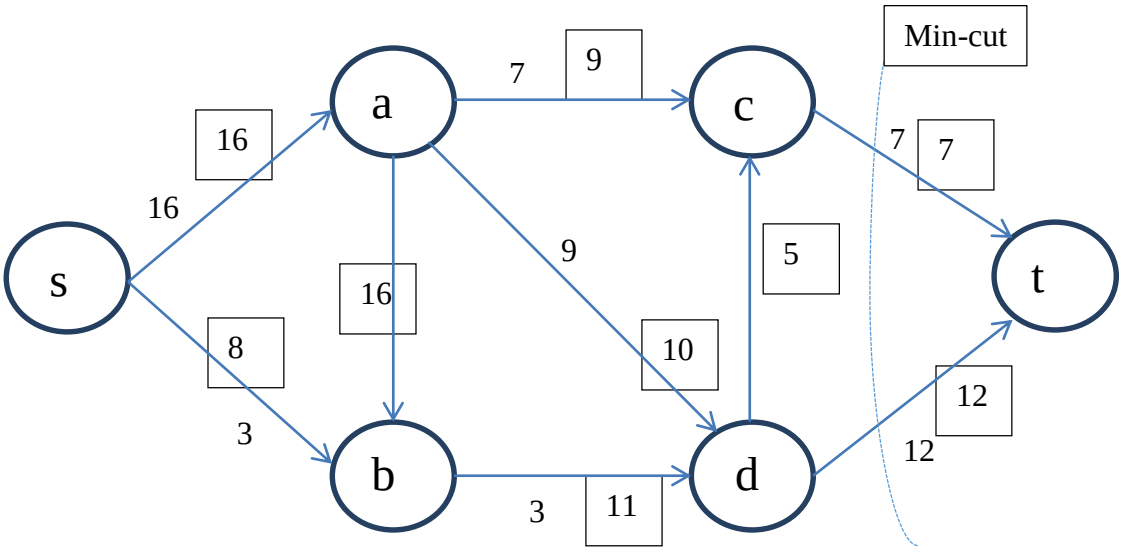
a) Current value of the flow is 7.
b) Residual graph is shown below



Additional Space for Problem 4

c) Augmenting paths for one of the maximum flow are
   a. S->a->d->t, increment in flow along these edges is 9.
   b. 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)>=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 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 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<= i<=n-1, 0<= j<=n-1.

opt[i, j]= c(i, j)+ max(opt[i-1,j]-2, opt[i, j-1]-1, opt[i-1, j-1])

If we assume that borrowing, i.e., negative number of golds are fine the initial conditions are

opt(i,-1)=opt(j,-1)=-inf , opt(0,0)=c(0,0)

if the assumption is that borrowing is not allowed the initial conditions are

opt(i,-1)=opt(j,-1)=-inf , opt(0,0)=c(0,0)

$$opt(1,0) = \begin{cases} -inf, & c(0,0) - 2 < 0 \\ c(0,0) + c(1,0) - 2, & c(0,0) - 2 \geq 0 \end{cases}$$

$$opt(0,1) = \begin{cases} -inf, & c(0,0) - 1 < 0 \\ c(0,0) + c(0,1) - 1, & c(0,0) - 1 \geq 0 \end{cases}$$

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,…, 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,…, 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 $t_1 < t_2 < \cdots < t_n$. 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 $i, i+1, \ldots, n$ 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 if 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 $t_1 < t_2 < \cdots < t_n$. 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 $t_1 < t_2 < \cdots < t_n$. 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 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 $T(n) = aT\left(\frac{n}{b}\right) + \Theta\left(n^{\log_b a} \log^k n\right)$ with $k \geq 0$, we have $T(n) = \Theta\left(n^{\log_b a} \log^{k+1} n\right)$.

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 easy 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 $T(2^k)=2(2T(2^{k/4})+k/2)+k=2^2\ T(2^{k/4})+2k=2^3\ T(2^{k/8})+3k=\ldots.=2^mT(1)+mk$
$=2^mT(1)+m2^m$. 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, . . . , 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= Compute-Even-Integers(l,m)+ 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 node are the families f_i(i=1,…,N) and the tables t_j(j=1,…,M). We add edge e_{ij}=(f_i, t_j) in the graph for i=1,…,N and j=1,…,M and set e_{ij}=1. Then we add a course s and sink in G. For each family f_i, we add e=(s, f_i) in the graph and set c_e=a(i). For each stable 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)+…+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.

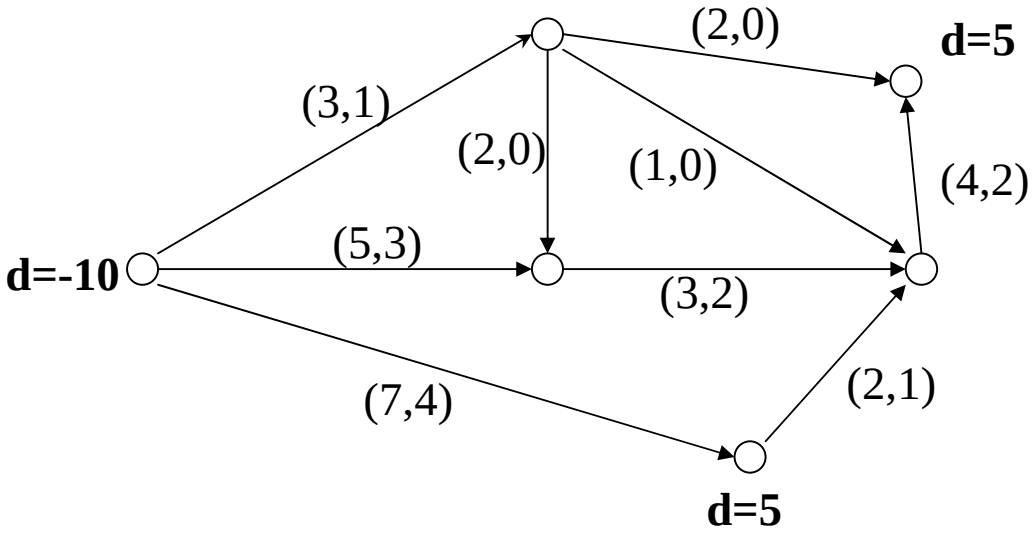
To prove the correctness of the algorithm, we prove that no two members of the same family use the same table if and only if the value of the maximum value of an s-t flow in G is a(1)+…+a(N):

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)+…+a(N).

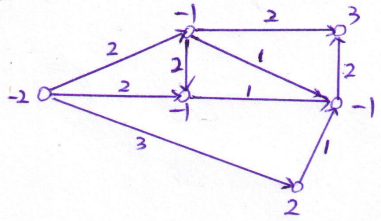
For the converse direction, assume that there is and s-t flow of value a(1)+…+a(N). The construction given in the algorithm makes sure that the there is at most one member in family j sitting in the table j as the capacity of the flow goes from f_i to c_j is 1. So we only need to make sure that all families are seated. Note that {s}, V-{s} is an s-t cut with value a(1)+…+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 construcing the graph is O(MN). The number of edges in the graph is MN+M+N, and v(f)= a(1)+…+a(N). Let T= a(1)+…+a(N), then the running time applying Ford-Fulkerson algorithm is O(MNT).

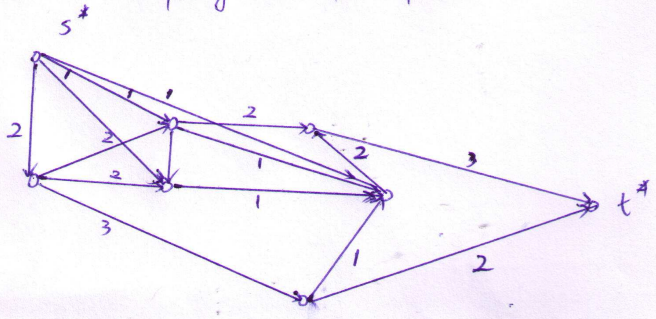
5) 20 pts
Determine if there is a feasible circulation in the below network. If so, determine the circulation, if not show where the bottlenecks are. The numbers in parentheses are (*lowerbound, upperbound*) on flow.



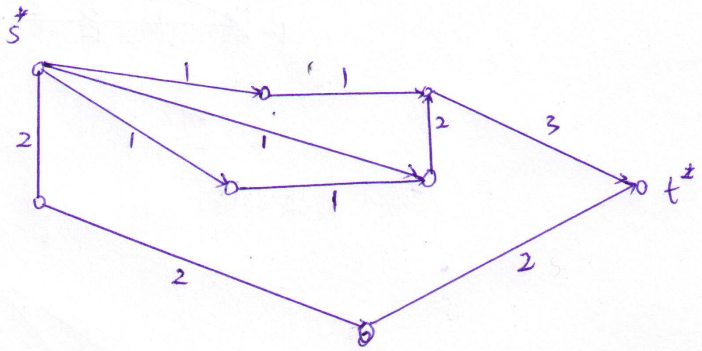
Step 1:



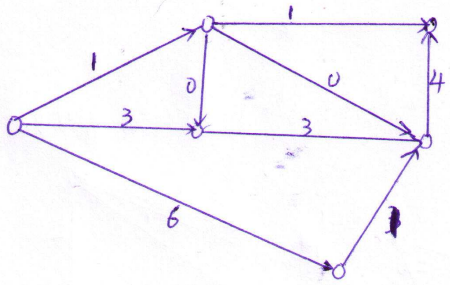
step 2: the ~~the~~ corresponding max-flow problem



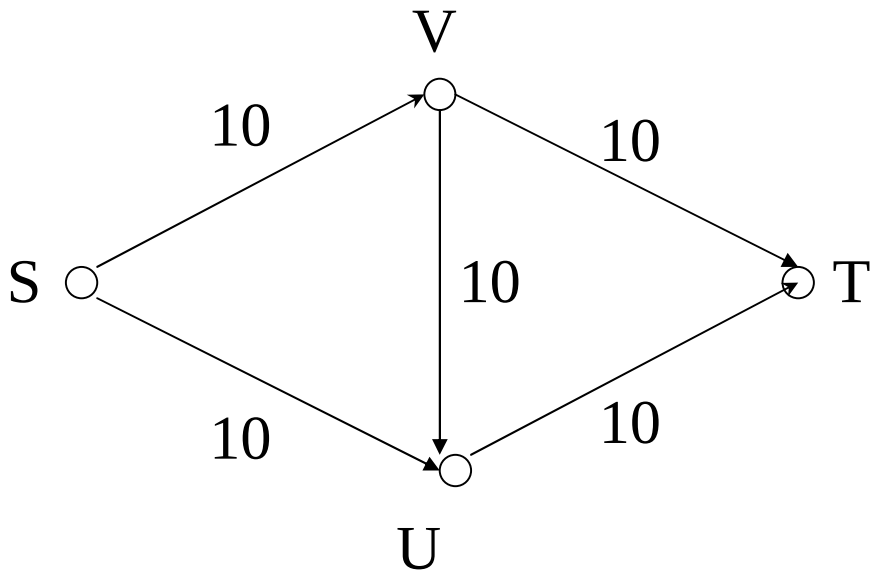
Step 3. Solving the max-flow problem and the result is as follows:



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

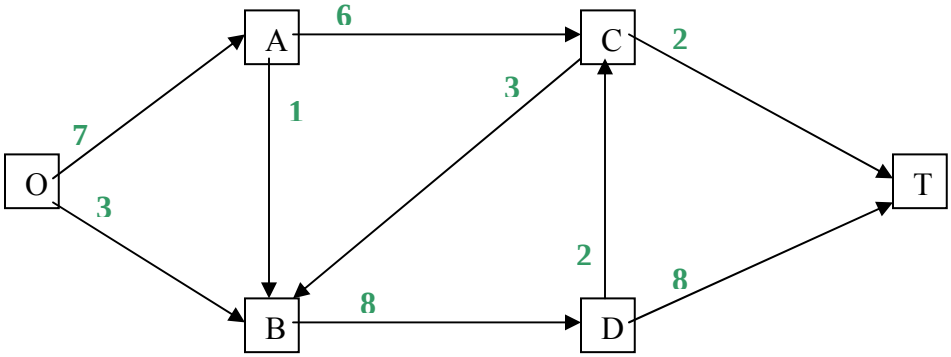
Present a divide-and-conquer algorithm that determines the minimum difference between any two elements of a sorted array of real numbers.

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.

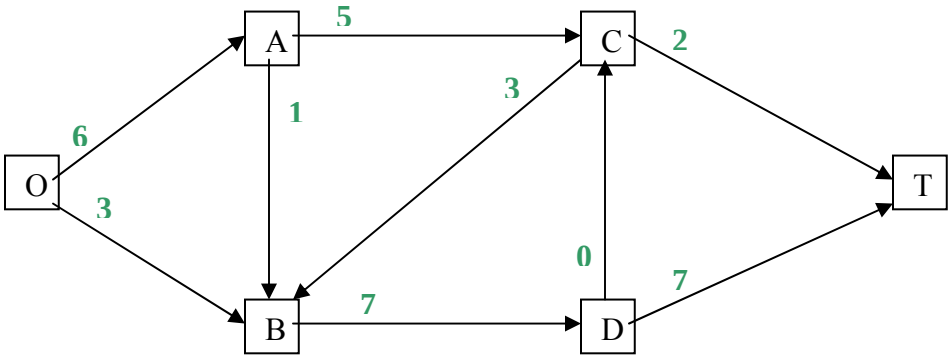
3) 15 pts
You are given the following directed network with source O and sink T.



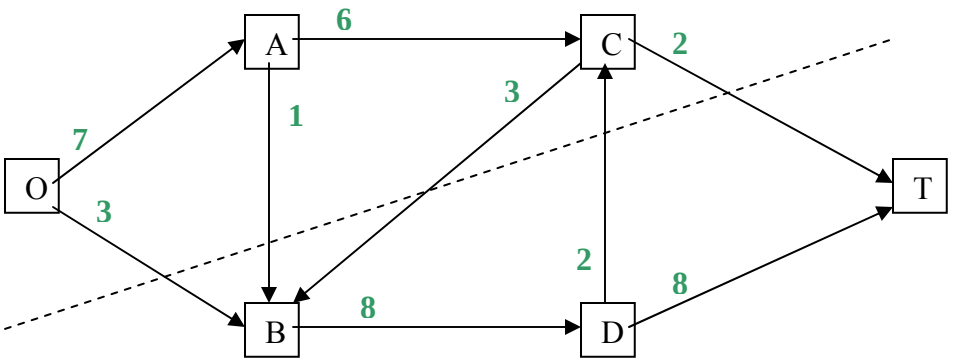
a) Find a maximum flow from O to T in the network.

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:

$$T(n) = 2T(\frac{n}{2}) + n\log n = 2(2T(\frac{n}{4}) + \frac{n}{2}\log n - \frac{n}{2}\log 2) + n\log n$$

$$= 4T(\frac{n}{4}) + 2n\log n - n\log 2 = \ldots = 2^k T(\frac{n}{2^k}) + kn\log n - \frac{k(k-1)}{2} n\log 2$$

$$= \ldots =_{(k=\log n)} nT(1) + \Theta(n\log^2 n) = \Theta(n\log^2 n)$$

b) $T(n) = 2T(n/2) + \log n$

Similar to a), the result is $T(n) = \Theta(n)$.

c) $T(n) = 2T(n-1) - T(n-2)$ for $n \geq 2$; $T(0) = 3$; $T(1) = 3$

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 a 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<=|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 index[].

Running time is $O(k|E|\cdot T(\max\_flow))$, 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 folloing i steps, not vice versa.

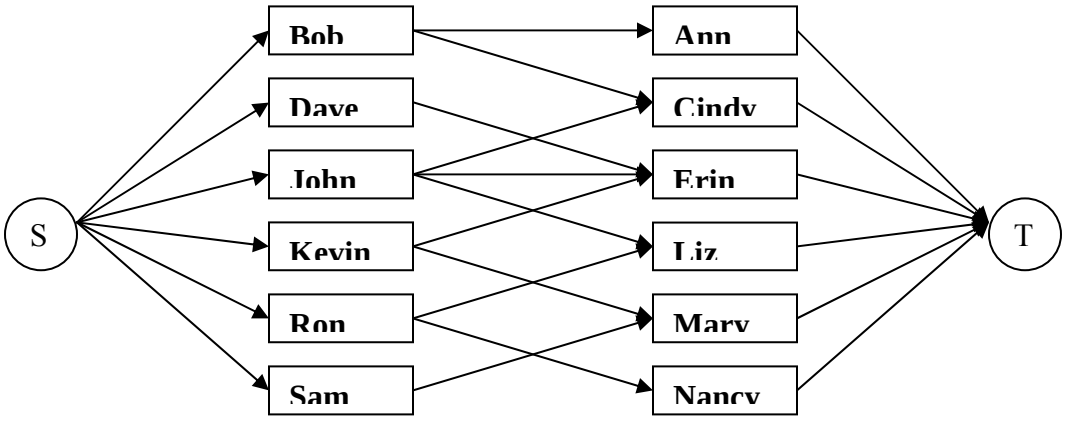
6) 15 pts

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 $<e_1, \ldots, e_k>$ carries positive flow iff $f(e_1) > 0, \ldots, f(e_k) > 0$.)

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

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

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

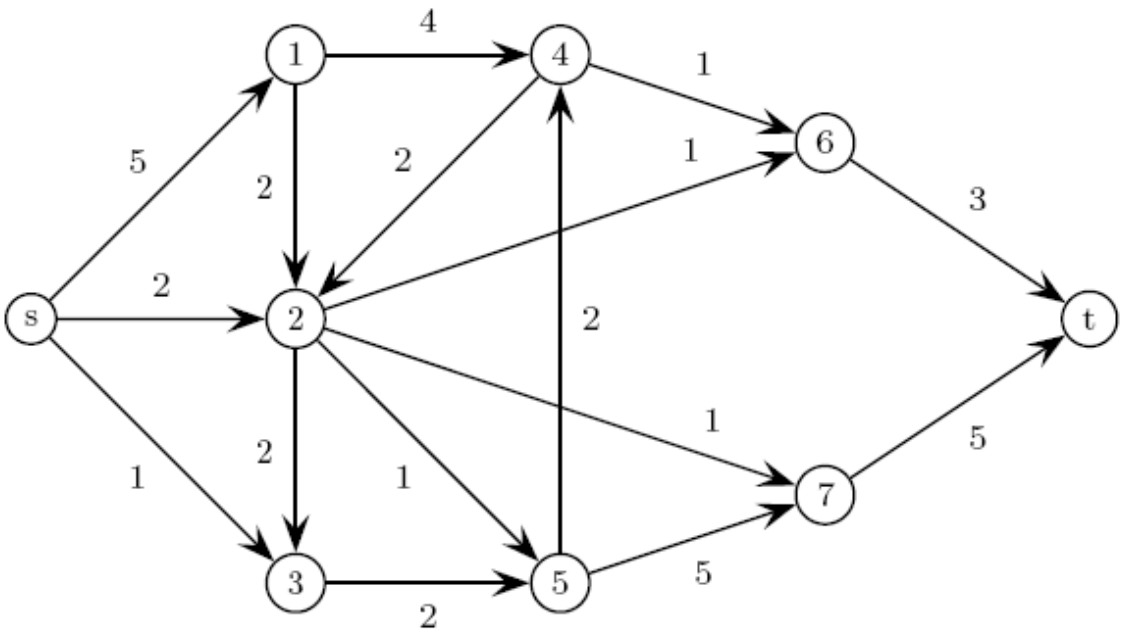
[TRUE/FALSE] FALSE
Any Dynamic Programming algorithm with n unique subproblems will run in $O(n)$ time.

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

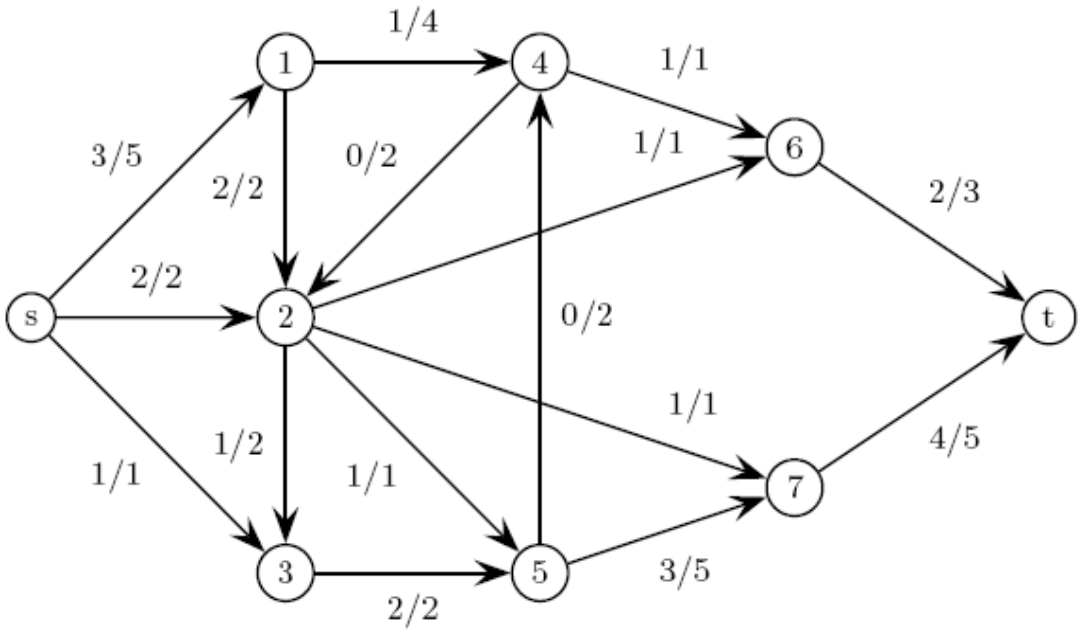
[TRUE/FALSE] FALSE
In dynamic programming you must calculate the optimal value of a subproblem twice, once during the bottom up pass and once during the top down pass.

2) 20 pts

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



Solution:



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

Solution:

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

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

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

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

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

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

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

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

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

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

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

Solution:

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

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

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

5) 20 pts

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

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

Solution

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

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

This gets us to the following recurrence

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

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

The iterative program will look as follows

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

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

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

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

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 $\boldsymbol{s}$ to $\boldsymbol{t}$.)
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.

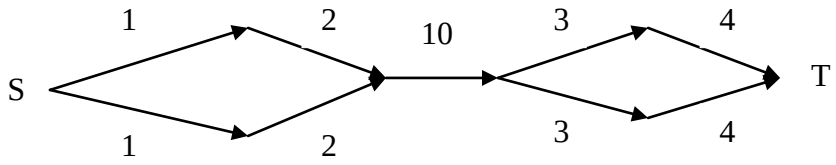
Given a directed graph, if deleting edge *e* reduces the original maximum flow more than deleting any other edge does, then edge *e* must be part of a minimum s-t cut in the original graph.

Solution:

False.

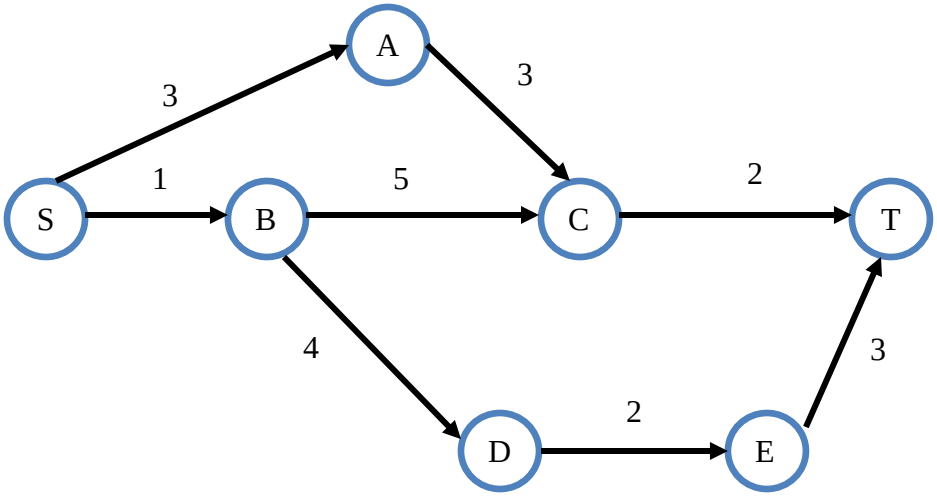
Counter-example:

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.

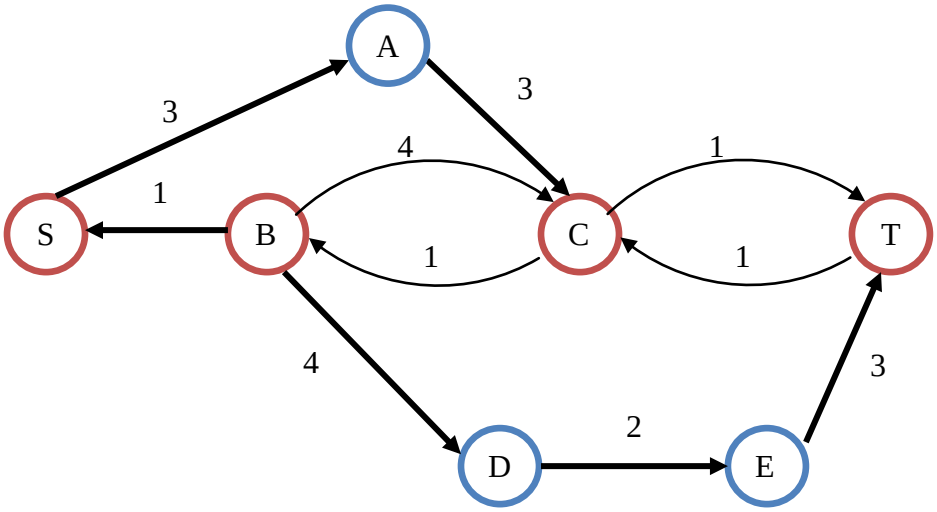
3) 20 pts



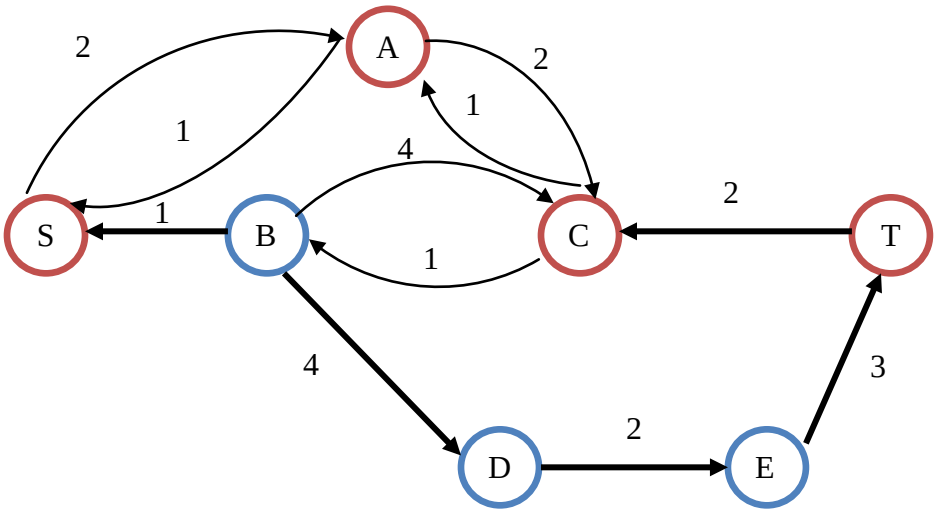
a- Show all steps involved in finding maximum flow from S to T in above flow network using the Ford-Fulkerson algorithm.
b- What is the value of the maximum flow?
c- Identify the minimum cut.

a.

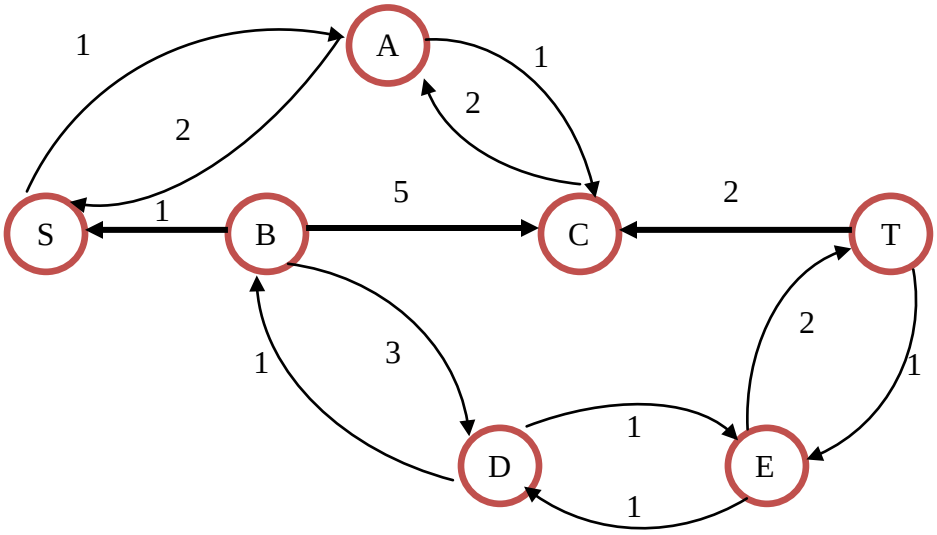
1. Augment path: [S, B, C, T], bottleneck is 1, residual graph:



2. Augment path: [S, A, C, T], bottleneck is 1, residual graph:



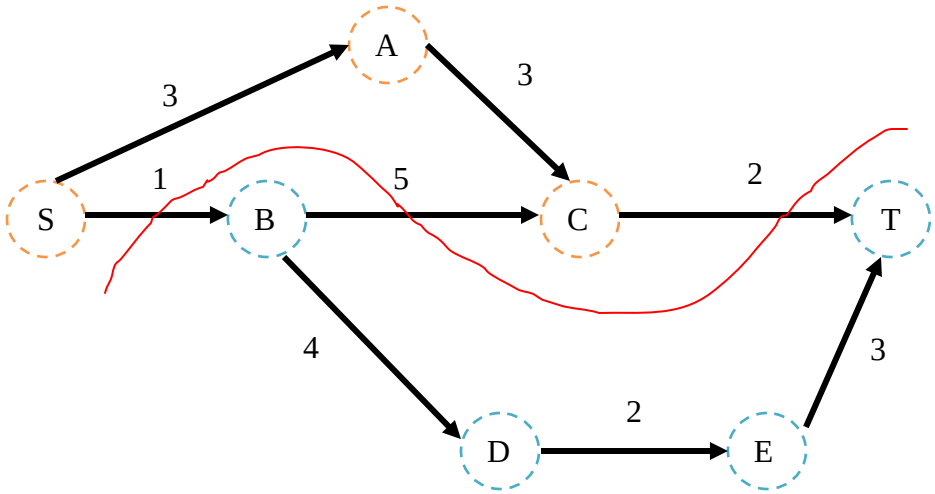
3. Augment path: [S, A, C, B, D, E, T], bottleneck is 1, residual graph:



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, s1 and s2, is defined as the minimum number of point mutations required to change s1 into s2, 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, $d\ (s1, s2)$, of two strings s1 and s2:

1. $d\ (', ') = 0$ (for empty strings)
2. $d\ (s, ') = d\ (', s) = |s|$ (length of s)
3. $d\ (s1 + ch1, s2 + ch2) = \boldsymbol{min}(d(s1, s2) + \begin{cases} 0, if\ ch1 = ch2 \\ 1, else \end{cases}, d(s1 + ch1, s2) + 1, d(s1, s2 + ch2) + 1\ )$

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, $ch1$ and $ch2$ respectively. Somehow, $ch1$ and $ch2$ have to be explained in an edit of $s1 + ch1$ into $s2 + ch2$.

- If $ch1$ equals $ch2$, they can be matched for no penalty, which is 0, and the overall edit distance is $d(s1, s2)$.
- If $ch1$ differs from $ch2$, then $ch1$ could be changed into $ch2$, costing 1, giving an overall cost $d(s1, s2) + 1$.
- Another possibility is to delete $ch1$ and edit $s1$ into $s2 + ch2$, $d(s1, s2 + ch2) + 1$.
- The last possibility is to edit $s1 + ch1$ into $s2$ and then insert $ch2$, $d(s1 + ch1, s2) + 1$.

There are no other alternatives. We take the least expensive, ***min***, of these alternatives.

Examination of the relations reveals that $d(s1, s2)$ depends only on $d(s1', s2')$ where $s1'$ is shorter than $s1$, or $s2'$ is shorter than $s2$, 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 $(i,j) \in \{(n1-1,n2),(n1,n2-1),(n1-1,n2-1)\}$ 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 $(i, j) = (n1 - 1, n2 - 1)$ then
Print "$change\ s1's\ n1^{th}\ letter\ into\ s2's\ n2^{th}\ letter$"
else if $(i, j) = (n1, n2 - 1)$ then
Print "$insert\ s2's\ n2^{th}\ letter\ after\ s1's\ n1^{th}\ position$"
else
Print "$delete\ s1's\ n1^{th}\ letter$"

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, 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 num[m][n].
We have

$$num[i][j] = \begin{cases} 0 & \text{i=0 or j=0 or a[i][j]=0} \\ a[1][1] & \text{i=1 and j=1} \\ num[i-1][j] + num[i][j-1] & other \end{cases}$$

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, 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 num[m][n] at last. This is an O(mn) solution since we use O(1) time to calculate each 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

Ture [TRUE/FALSE]
The Ford Fulkerson algorithm with real valued capacities can run forever

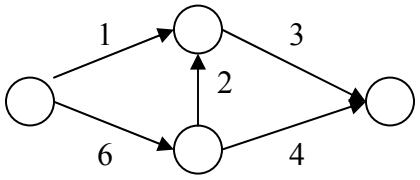
Ture [TRUE/FALSE]
If we have a 0-1 valued s-t flow in a graph of value f, then we have f edge disjoint s-t paths in the graph

Ture [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 Ci

a. If the capacities on each edge are unique then there is a unique min cut

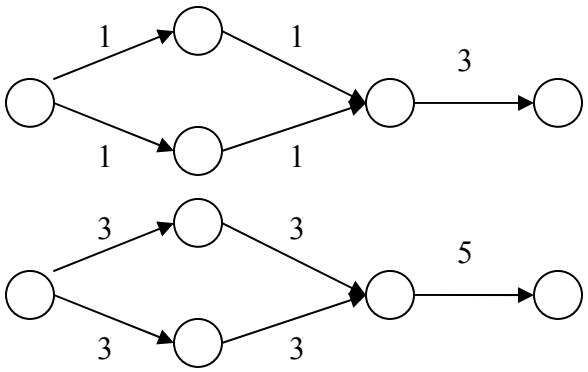
Disprove :



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.)



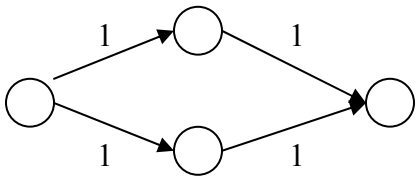
Before add 2 to every edge

After add 2 to every edge

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



Max flow: 2

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 the first two array takes O(n), … the merge of the last array takes O(kn), totally there are k merge sorts. Thus, it takes $O(k^2n)$ 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 element.
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(logk), and take O(kn) operations, thus, the running time is O(kn*logk)

4) 10 pts
Derive a recurrence equation that describes the following code. Then, solve the recurrence equation.

```
COMPOSE (n)
    1. for i ← 1 to n
    2.         do for j ← 1 to sqrt(n)
    3.                 do print(i, j, n)
    4. if n > 0
    5.         then for i ← 1 to 5
    6.                 COMPOSE (⌊n/2⌋)
```

$T(n) = 5\ T(n/2) + O(n^{3/2})$

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 T1+T2+T3 > 12 or T1+T2+T3 > H1 + H2 + H3, 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 $T(n) \leq cn$
$\Rightarrow T(n) \leq 2 * 2c/5n + an \leq cn$ 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, $T(n) = \Theta\left(n^{\log_2 2}\right) = \Theta(n)$

b. $(n \log n)^4 = n^4 \log^4 n = \Theta\left(n^{\log_2 16} \log^4 n\right)$,
$\Rightarrow$ case 2 of master method, $T(n) = \Theta\left(n^{\log_2 16} \log^{(4+1)} n\right) = \Theta(n^4 \log^5 n)$

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]<j, 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]>=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 ith 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_i'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 $i = 1 \ldots n$

  Compute D_i

end for

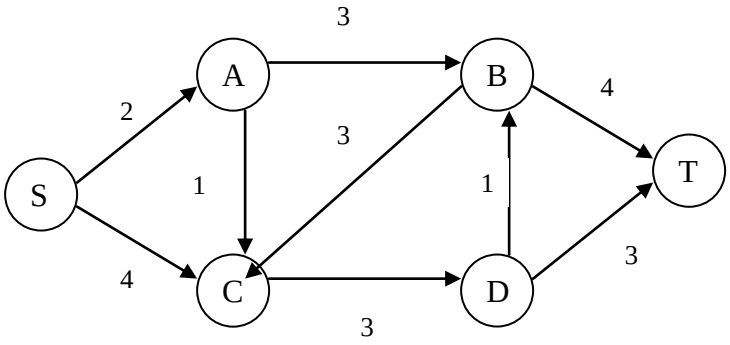
return the largest number among D_1 *to* D_n.

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.

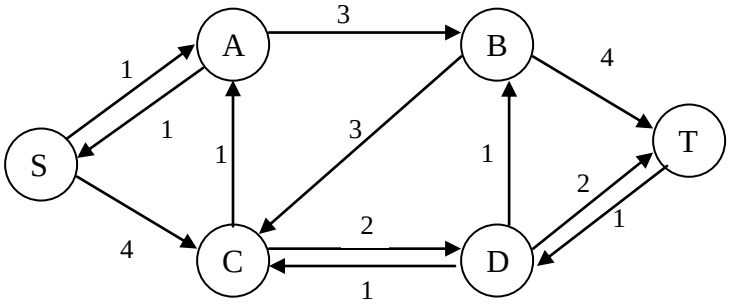


a) 10 pts

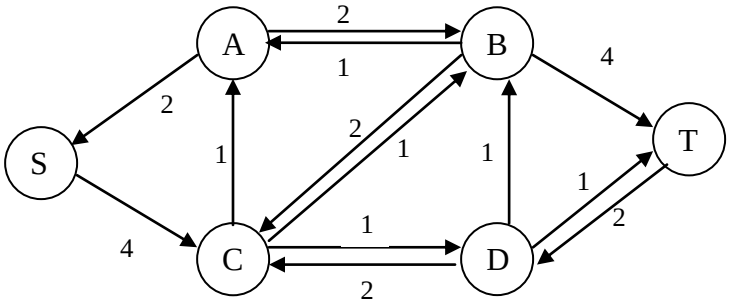
Draw the residual network after we have updated the flow using these two augmenting paths(in the order given)

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:

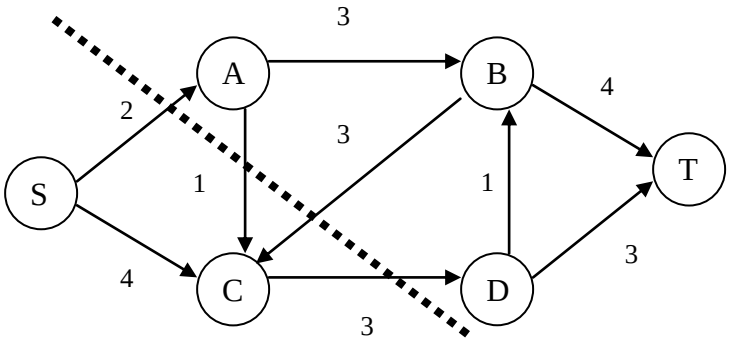
S-C-B-T
S-C-D-T
S-C-A-B-T
S-C-D-B-T

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

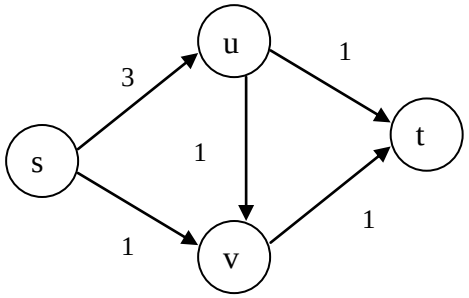
Decide whether you think the following statements are true or false. If true, give a short explanation. If false, give a counterexample.

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.

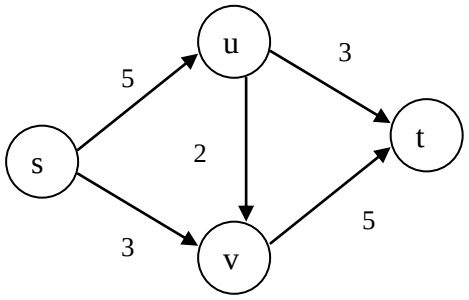


Clearly, the maximum s-t flow in the above graph is 2.
The edge (s, u) does not have $f(e) = c_e$

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≥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 $d_1, d_2, \ldots, d_i$.
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:
   $$OPT[T] = \min_{1 \le i \le n} \{OPT[T - X_i] + 1\}$$
   The boundary values will be as follows:
   $$OPT[0] = 0$$
   $$OPT[v] = \infty \; if \; v < 0$$
   We fill in the *OPT[]* array in the following order:

```
for (int i = 1; i <= v; i++) {
   int min = infinity;
   for (int x : {X_1, X_2, ..., X_n}) {
      if (OPT[i – x] + 1 > min) min = OPT[i – x] + 1;
   }
   OPT[i] = min;
}
```

   At the end if $OPT[v] \le k$ 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:
   $$OPT[k, v] = \min\left\{OPT[k-1, v], \min_{1 \le i \le n} \{OPT[k-1, v - X_i] + 1\}\right\}$$
   Here the boundary values will be:
   $$OPT[k, 0] = 0$$
   $$OPT[0, v] = \infty \; if \; v \neq 0$$
   $$OPT[k, v] = \infty \; if \; v < 0$$
   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 : {X_1, X_2, ..., X_n}) {
        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:

$$OPT[k, 0] = true$$
$$OPT[0, v] = false$$
$$OPT[k, v] = \ false \ if \ v < 0$$

We should fill the $OPT[]$ array in the following order:

```
for (int i = 1; i <= v; i++) {
   for (int j = 1; j <= k; j++) {
      OPT[i, j] = OPT[j-1, i];
      for (int x : {X_1, X_2, ..., X_n}) {
         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]>=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]>=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]>=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]>=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...n] with n>3, A[1]>=A[2] and A[n]>=A[n-1]. As the induction hypothesis, assume that all arrays B[1....k] with 2<k<n, B[1]>=B[2] and B[k]>=B[k-1] have a local minimum. Let j=floor(n/2). If A[j]<=A[j+1], then A[1....j+1] has a local minimum (by the induction hypothesis) and hence A[1....n] has a local minimum. Else (that is, A[j]>A[j+1]), then A[j....n] has a local minimum (by the induction hypothesis) and hence A[1....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=floor(n/2). If A[j]<=A[j+1], then search for a local minimum in A[1....j+1]. Else, search for a local minimum in A[j....n]. Let T(n) denote the number of pairwise comparisons performed by our recursive algorithm for finding a local minimum in A[1...n]. Then T(n)<=T(ceil(n/2))+1 which implies that T(n) = O(log(n)).

Q5:

(a) The number of unique ways are shown as follows:

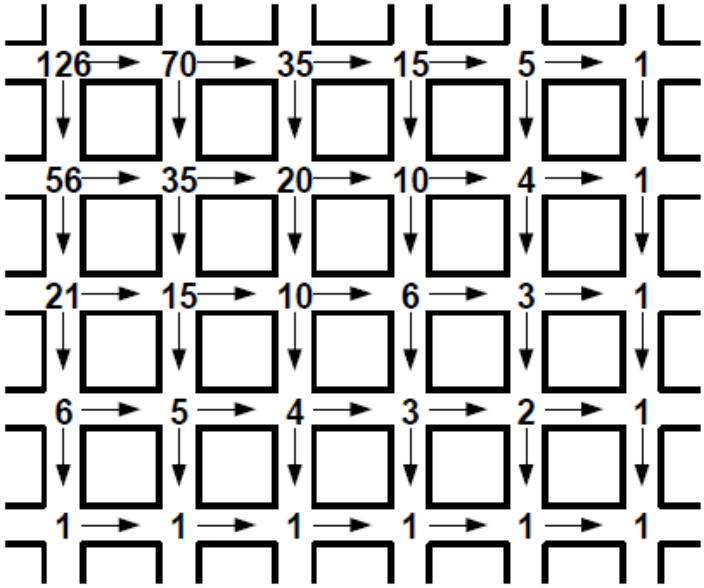


Figure A.

Answer: 126

(b) Define OPT(i,j) as the number of unique ways from intersection (i,j) to E: (5,4). The recursive relation is :
OPT(i,j) = OPT(i+1, j) + OPT(i,j+1) , for i<5, and j<4
Boundary condition: OPT(5,j) = 1; OPT(i,4) = 1

Alternative solution:
If you define 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:

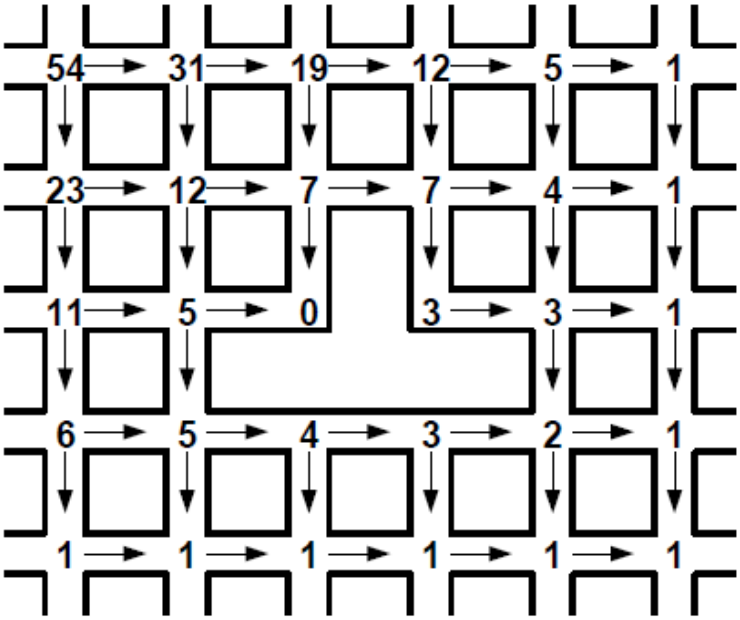


Figure B.

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:

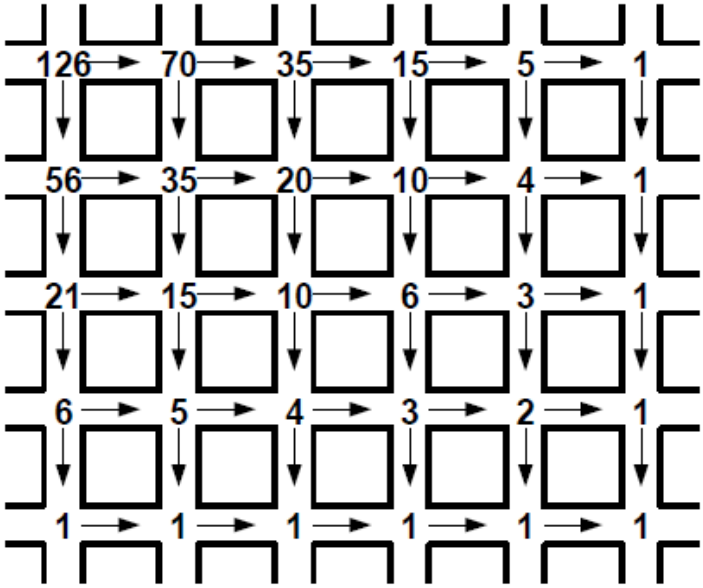


Figure A.

Answer: 126

(b) Define OPT(i,j) as the number of unique ways from intersection (i,j) to E: (5,4). The recursive relation is :
OPT(i,j) = OPT(i+1, j) + OPT(i,j+1) , for i<5, and j<4
Boundary condition: OPT(5,j) = 1; OPT(i,4) = 1

Alternative solution:
If you define 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:

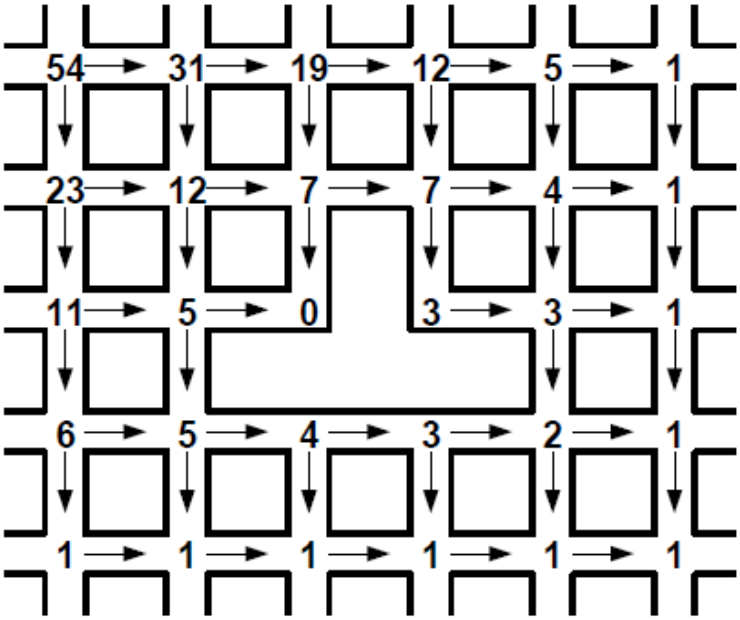


Figure B.

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-setdoesn'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]
L1 can be reduced to L2 in Polynomial time and L2 is in NP, then L1 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 = *{dv}* 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 ← 0
3 Q ← V[G]
4 while Q ≠ 0
5       do u ← EXTRACT-MIN(Q)
6       S ← S ∪ {u}
7       for each vertex v ∈ 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] ← d[u] + w(u, v)
3 π[v]← u
```

We have the algorithm as shown above.

We show that in each iteration of Dijkstra's algorithm, d[u]=δ(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 δ(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)= δ(s,u) for all $u \in N^-(s)$ and by the upper bound property d[u]= δ(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.

Item type	Weight	Volume	Number	Cost if not carried
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…K. Give an efficient algorithm to partition these integers into two subsets such that the difference |S1-S2| is minimized, where S1 and S2 denote the sums of the elements in each of the two subsets.

Proposed Solution :

```
a[1]...a[m] = the set of integers
opt[i,k] = true if and only if it is possible to sum to k using elements 1...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 Є V , a sink t Є 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*size(W) times

16 pts
Suppose that we have n files F_1, F_2, ..., F_n of lengths l_1, l_2, ... , l_n respectively, and want to concatenate them to produce a single file $F = F_1 \circ F_2 \circ ... \circ F_n$ of length $l_1 + l_2 + ... + l_n$. 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, ..., F_n and a cost function $c(.,.)$, 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. 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 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 assignmentg}.

For instance, the formula ψ: (x v y v z) ^ (x′ v y′ ∨ z′) ^ (x′ ∨ y′ ∨ 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 $\psi(x_1, \ldots, x_n)$:
1. Introduce a new variable y.
2. Output formula $\psi'(x_1, \ldots, x_n, y) = \psi(x_1, \ldots, x_n)$ ^ (y | not y).
If $\psi(x_1, \ldots, x_n)$ belongs to SAT, then ψ' has at least 1 satisfying assignment, and therefore $\psi'(x_1, \ldots, x_n, y)$ has at least 2 satisfying assignments as we can satisfy the new clause (y | not y) by assigning either
y = 1 or y = 0 to the new variable y, so $\psi'(x_1, \ldots, x_n, y)$ belongs to Double-SAT. On the other hand, if $\psi(x_1, \ldots, x_n)$ does not belong to SAT, then clearly $\psi'(x_1, \ldots, x_n, y) = \psi(x_1, \ldots, x_n)$ ^ (y | not y) has no satisfying assignment either, so $\psi'(x_1, \ldots, x_n, y)$ 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 $w : E \to \{0, 1, \ldots, W\}$ 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 $w : E \to \{1, \ldots W - 1\}$. To solve this efficiently, implement the priority queue by an array A of length $WV + 1$. Any node with shortests 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 minimun 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),…,g(n), and women h(1),…,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, $MAX_i\ |g(i)-h(w_i)|$.

Give an algorithm that runs in O(n logn) 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 p0, p1, …, pn and matrices A1, A2, …, An where matrix Ai has dimension (pi-1) * pi

(a) Let m(i, j) denote the minimum number of scalar multiplications needed to evaluate the matrix product Ai Ai+1 … Aj. Write down a recursive algorithm to compute m(i, j), 1 <= i <= j <= 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  1 <= i <= j <= 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) +Pi-1*Pk*Pk}

   return m[i, j]
}
```

(b) Use this algorithm to compute m(1,4) for p0=2, p1=5, p2=3, p3=6, p4=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

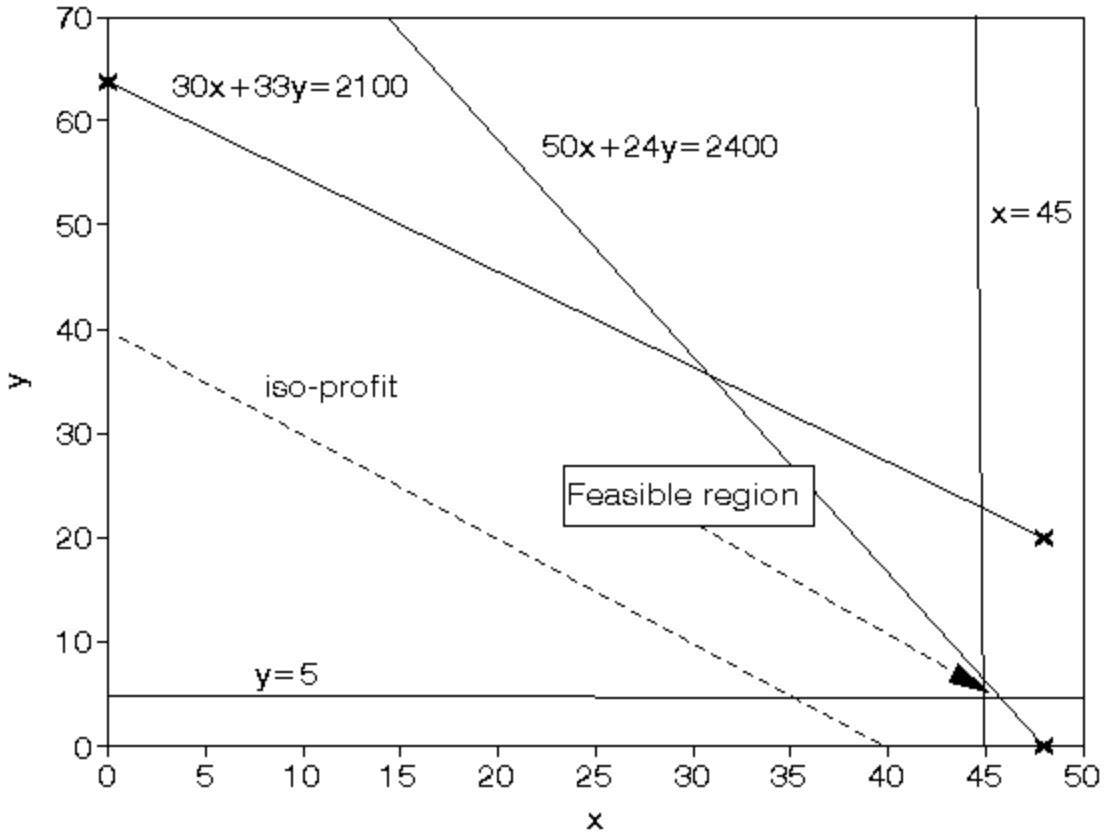
then the constraints are:

- 50x + 24y <= 40(60) machine A time
- 30x + 33y <= 35(60) machine B time
- x >= 75 - 30
- i.e. x >= 45 so production of X >= demand (75) - initial stock (30), which ensures we meet demand
- y >= 95 - 90
- i.e. y >= 5 so production of Y >= 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: $(1,1)$—$(1,2)$—$(1,3)$--…--$(1, n-1)$—$(1, n)$—$(2, n)$—$(2, n-1)$--…--$(2,2)$—$(2,1)$—$(3,1)$--...—(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 x1; denote the number of edges we find in step ii) as x2.

Then we have x1 * 2 + x2 = |V|.

Consider an arbitrary edge cover set Y. Suppose Y contains y1 edges, each of which is counted as the one covering 2 nodes. Suppose Y contains y2 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 y1 * 2 +y2 = |V|.

Here x1 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 x1 >= y1.

Then we have:
x1+x2 = |V| - x1 = y1* 2 +y2 –x1 = y1 + y2 - (x1-y1) <= y1 + y2.

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 SQD(n) of n is defined to be the maximum number of perfect squares you could observe among all such sequences. For example, SQD(32492) = 3 via the sequence

32492 → **3249** → **324** → 24 → **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 → **3249** → 249 → **49** → **9**

also gives you 3 perfect squares, viz. 3249, 49, and 9.

Describe an efficient algorithm to compute the square-depth SQD(n), of a given number n, written as a d-digit decimal number $a_1 a_2 \ldots a_d$. 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 nij = ai . . . aj. That is, nij 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 nij .

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 nij itself is square. We can express this as a recurrence:

D[i, j] = max(D[i + 1, j], D[i, j − 1]) + IS-SQUARE(nij)

The base cases are D[i, i] = IS-SQUARE(ai), for all $1 \leq i \leq d$. The solution to the general problem is 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 >= 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)

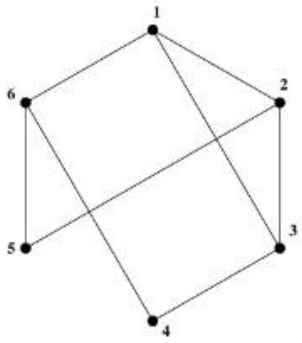
check that there are edges between every two adjacent nodes on the path in O(n)

check that the length of the path is at least k in O(n)

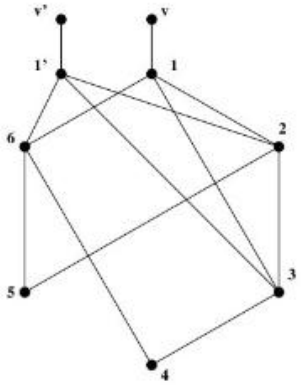
b) Two possible solutions:

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 of 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 ≠ NP, then a problem A ∈ 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 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: OPT(n) = a_n.

Recursive relation: OPT(i) = max{OPT(i+1)+a_i, a_i}, for 1<= i < n

Algorithm:

For i = n, …., 1
    Compute OPT(i).
End

Maximum profit = max_{i} {OPT(i)}. Denote i* as the starting index which achieves the maximum profit, then i* is the optimal starting segment

For i = i*,……, n
    If (OPT(i) = = a_i)
        Set j* = i;
        Break;
    End
End
The value j* is the index of the optimal ending segment.

Complextiy: 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 modify as follows:

Maximum profit = max{0, max_{i} {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 part:
a_1,…..a_{n/2} and a_{n/2+1},…..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 segements, 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_opt_left = max_{i} {S_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_cross(1,n);
- Starting from [a_{i_cross*},……, 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_right(i).
- Find P_cross (1,n) = max_{i} {S_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_cross(1,n);

Then
Maximum Profit = max{P_left(1,n), P_right(1,n), P_cross(1,n)}, and the corresponding optimal starting index and ending index are the ones that achieve the maximum.

iii) 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.

iv) 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(nlog(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 $s_1, s_2, \ldots, s_m$ and a set of n companies $C = \{c_1, c_2, \ldots, c_n\}$. 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. $s \to s_i$ with capacity 1 unit, for each $1 \leq i \leq m$,
ii. For each $1 \leq i \leq m$, $s_i \to c$ with capacity 1 unit for all nodes $c \in \Lambda_i$.
iii. $c_j \to t$ 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 $s_i \to c_j$ 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 $s \to s_i$ 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 $s_i \to c_{\sigma(i)}$ 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+1st) rod will be less than or equal to the cost of connecting the k+1st 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 Length(i) * Level(i), where 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
Total assembly cost of T* = Total assembly cost of T*' + total length of the two smallest rods
Total assembly cost of T = Total assembly cost of T' + total length of the two smallest rods

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