CS570
Analysis of Algorithms
Summer 2013
Exam II

Name: _____

Student ID: _____

____On Campus ____DEN

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

2 hr exam
Close book and notes

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

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

[TRUE/FALSE]
If a network with a source s and sink t has a unique max s-t flow, then it has a unique min s-t cut.

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

[TRUE/FALSE]
Fractional knapsack problem can be solved in polynomial time.

[TRUE/FALSE]
Subset-sum problem can be solved in polynomial time

[TRUE/FALSE]
0-1 knapsack problem can be solved in polynomial time

[TRUE/FALSE]
Max flow in a flow networks can be found in polynomial time

[TRUE/FALSE]
Ford-Fulkerson algorithm runs in polynomial time

[TRUE/FALSE]
One can determine if a flow is valid or not in linear time with respect to the number of edges and nodes in the graph.

[TRUE/FALSE]
Given the value of flow v(f) for a flow network, one can determine if this value is the maximum value of flow or not in linear time.

[TRUE/FALSE]
A dynamic programming algorithm always uses some type of recurrence relation.

1. False. For example consider \{(s,x),(x,t)\} where all the edges have the same capacity.

2. True. Since the source can have at most $n-1$ adjacent vertices, the capacity of the min cut is at most $n-1$. Further the number of edges is at most $n(n-1)$.

3. True. The greedy strategy of picking as much of the item with the highest value to weight ratio as possible is optimal.

4. False: The running time is in the worst case exponential in the number of bits required to write down the problem instance.

5. False: The running time is in the worst case exponential in the number of bits required to write down the problem instance.

6. True: For example the variation of Ford-Fulkerson which at each step uses BFS to compute augmenting paths runs in time polynomial in the number of vertices irrespective of the capacities.

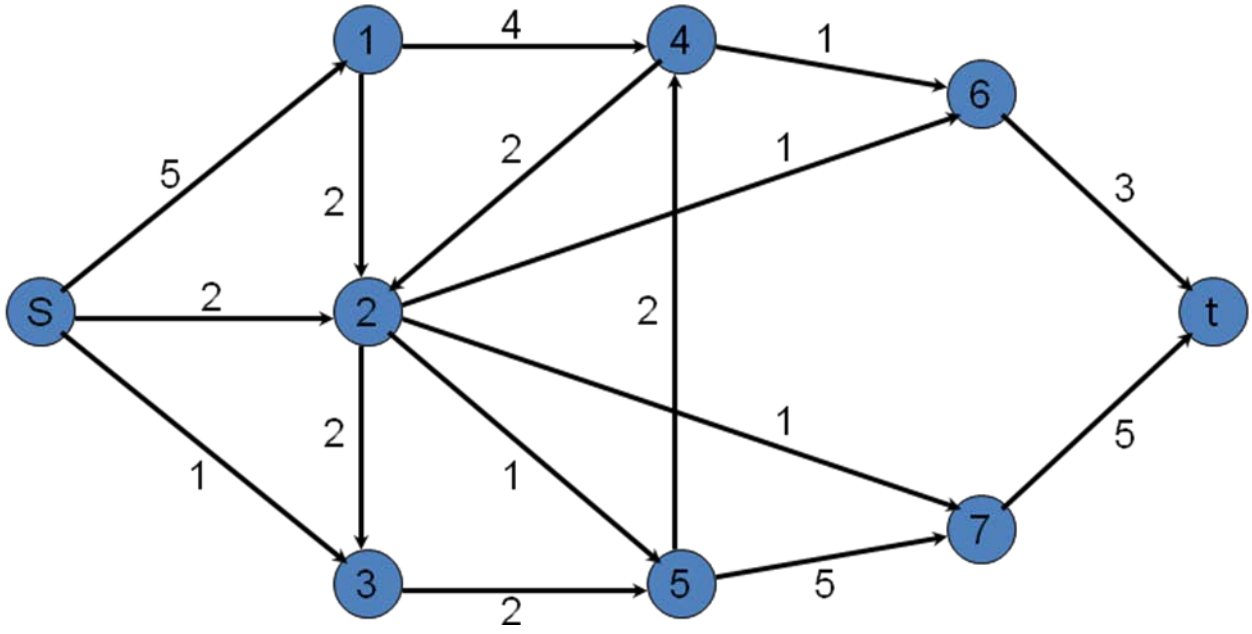
8. True: Each edge is checked once for capacity constraints and at most twice for conservation constraints.

9. True. If the flow is maximal, then for every min cut, the flow across it equals its capacity and thus every edge crossing it should be saturated.

10. True

2) 20 pts

In the familiar graph below, find the maximum s-t flow. The numbers on the edges are the capacities. Show all your steps.



Additional Space

2 One way to solve the problem is using the Ford-Fulkerson algorithm. If you choose to do so, then you have explicitly write down each step. (That is, the choice of augmenting path (and the flow sent on it) at each step and resulting residual graph). If you computed the max flow by other means, you have to prove that your flow is indeed a max flow (perhaps by demonstrating a cut whose capacity equals the value of your valid flow.)

One max flow is : f((s,1)) = 3, f((s,2)) = 2, f((s,3)) = 1, f((1,4)) =1, f((1,2)) =2,f((2,3)) =1,f((2,5)) =1,f((2,6)) =1,f((2,7)) =1,f((3,5)) =2,f((4,2)) =0,f((4,6)) =1, f((5,4)) =0,f((5,7)) =3,f((7,t)) =4,f((6,t)) =2.

There are multiple flows that are maximal (each of value 6). So your answer may be different and yet correct. There is however a unique min-cut ({s,1,2,3,4},{5,6,7,t}).

3) 20 pts

There are n cities {c1, c2,…, cn}. A city either has a bank or it does not have a bank and you know which cities have banks. Between every ordered pair (ci, cj) of cities, there is a one way road ri,j that goes from ci to cj . Assume that the city c1 does not have a bank and a bank robber has escaped from the prison and is now in c1. You are the police chief and you plan to install inspection stations on certain roads to prevent the bank robber from getting into cities with banks. If the bank robber attempts to travel on a road that has an inspection station, he will be caught. To install an inspection station on the road ri,j , it costs ci,j dollars. Design an algorithm to determine the set of roads on which to install the inspection stations that minimizes the total installation cost under the constraint that the bank robber cannot get to a city with a bank without being caught on one of the inspection stations. Prove that your algorithm is correct and analyze its running time.

We build a flow network with the cities as the vertices as follows. Set c_1 as the source and introduce a sink t. From c_i to c_j, add an edge of capacity r_{i,j}. From every city with a bank to t, add an edge of "infinite" capacity. Here, by "infinite" we mean something large enough that these edges won’t be in a min-cut: thus "infinite" could be the one plus the cost of placing inspections on all roads. There is now a one to one correspondence between valid placements of inspection stations on roads and c_1-t cuts. Thus the valid placement that minimizes the total cost is one that places the inspection stations on roads corresponding to a min c_1-t cut.

4) 20 pts

Solve the following knapsack problem, i.e. the combination of objects that give us the highest value. Knapsack capacity = 8

Item i	Value vi	Weight wi
1	15	1
2	10	5
3	9	3
4	5	4

a) Show the recurrence relation

Initial Settings: Set

$V[0, w] = 0$ for $0 \leq w \leq W$, no item

$V[i, w] = -\infty$ for $w < 0$, illegal

Recursive Step: Use

$$V[i, w] = \max(V[i-1, w], v_i + V[i-1, w-w_i])$$

for $1 \leq i \leq n$, $0 \leq w \leq W$.

Correctness of the Method for Computing $V[i, w]$

Lemma: For $1 \leq i \leq n$, $0 \leq w \leq W$,

$$V[i, w] = \max(V[i-1, w], v_i + V[i-1, w-w_i]).$$

Proof: To compute $V[i, w]$ we note that we have only two choices for file i:

Leave file i: The best we can do with files $\{1, 2, \ldots, i-1\}$ and storage limit w is $V[i-1, w]$.

Take file i (only possible if $w_i \leq w$): Then we gain v_i of computing time, but have spent w_i bytes of our storage. The best we can do with remaining files $\{1, 2, \ldots, i-1\}$ and storage $(w - w_i)$ is $V[i-1, w-w_i]$.
Totally, we get $v_i + V[i-1, w-w_i]$.

Note that if $w_i > w$, then $v_i + V[i-1, w-w_i] = -\infty$ so the lemma is correct in any case.

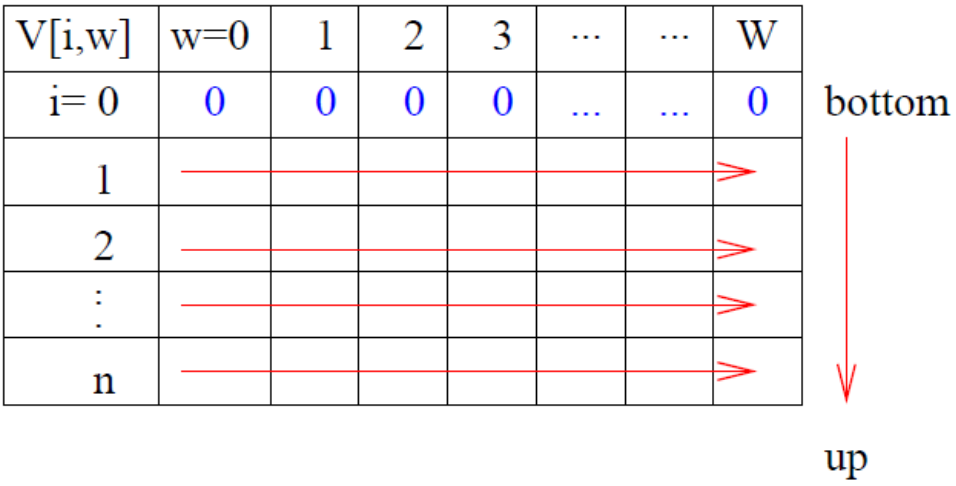
b) Numerically solve the problem.

Bottom: $V[0, w] = 0$ for all $0 \leq w \leq W$.

Bottom-up computation: Computing the table using

$$V[i, w] = \max(V[i-1, w], v_i + V[i-1, w-w_i])$$

row by row.



Example of the Bottom-up computation

Let $W = 8$ and

i	1	2	3	4
v_i	15	10	9	5
w_i	1	5	3	4

$V[i, w]$	0	1	2	3	4	5	6	7	8
$i = 0$	0	0	0	0	0	0	0	0	0
1	0	15	15	15	15	15	15	15	15
2	0	15	15	15	15	15	25	25	25
3	0	15	15	15	24	24	25	25	25
4	0	15	15	15	24	24	25	25	29

5) 20 pts

You are given integers $p_0, p_1, \ldots, p_n$ and matrices $A_1, A_2, \ldots, A_n$ where matrix A_i has dimension $p_{i-1} * p_i$

(a) Let m(i, j) denote the minimum number of scalar multiplications needed to evaluate the matrix product Ai Ai+1 … Aj. Write down a recurrence relation to compute m(i, j) for all $1 \leq i \leq j \leq n$ that runs in O(n^3) time.

This problem, like other dynamic programming problems involves determining a structure (in this case, a parenthesization). We want to break the problem into sub problems, whose solutions can be combined to solve the global problem. As is common to any DP solution, we need to find some way to break the problem into smaller sub problems, and we need to determine a recursive formulation, which represents the optimum solution to each problem in terms of solutions to the sub problems. Let us think of how we can do this.

Since matrices cannot be reordered, it makes sense to think about sequences of matrices. Let Ai..j denote the result of multiplying matrices i through j. It is easy to see that Ai..j is a pi−1 ×pj matrix. (Think about this for a second to be sure you see why.) Now, in order to determine how to perform this multiplication optimally, we need to make many decisions. What we want to do is to break the problem into problems of a similar structure.

In parenthesizing the expression, we can consider the highest level of parenthesization. At this level we are simply multiplying two matrices together. That is, for any k, $1 \leq k \leq n-1$,

$$A_{1..n} = A_{1..k} \cdot A_{k+1..n}.$$

Thus the problem of determining the optimal sequence of multiplications is broken up into two questions: how do we decide where to split the chain (what is k?) and how do we parenthesize the subchains $A_{1..k}$ and $A_{k+1..n}$?

The subchain problems can be solved recursively, by applying the same scheme.

So, let us think about the problem of determining the best value of k. At this point, you may be tempted to consider some clever ideas. For example, since we want matrices with small dimensions, pick the value of k that minimizes p_k. Although this is not a bad idea, in principle. (After all it might work. It just turns out that it doesn't in this case. This takes a bit of thinking, which you should try.) Instead, as is true in almost all dynamic programming solutions, we will do the dumbest thing of simply considering all possible choices of k,and taking the best of them. Usually trying all possible choices is bad, since it quickly leads to an exponential number of total possibilities. What saves us here is that there are only $O(n^2)$ different sequences of matrices.

(There are $C(n,2) = n(n-1)/2$ ways of choosing i and j to form $A_{i..j}$ to be precise.) Thus, we do not encounter the exponential growth.

Notice that our chain matrix multiplication problem satisfies the principle of optimality, because once we decide to break the sequence into the product $A_{1..k} \cdot A_{k+1..n}$, we should compute each subsequence optimally. That is, for the global problem to be solved optimally, the sub problems must be solved optimally as well.

Dynamic Programming Formulation: We will store the solutions to the sub problems in a table, and build the table in a bottom-up manner. For $1 \leq i \leq j \leq n$, let $m[i, j]$ denote the minimum number of multiplications needed to compute $A_{i..j}$. The optimum cost can be described by the following recursive formulation.

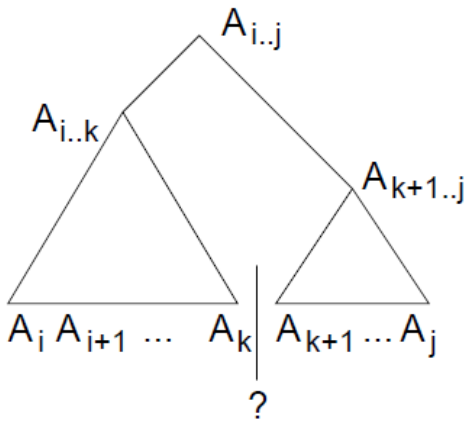
Basis: Observe that if $i = j$ then the sequence contains only one matrix, and so the cost is 0. (There is nothing to multiply.) Thus, $m[i, i] = 0$.

Step: If $i < j$, then we are asking about the product $A_{i..j}$. This can be split by considering each k, $i \leq k < j$, as $A_{i..k}$ times $A_{k+1..j}$.

The optimum times to compute $A_{i..k}$ and $A_{k+1..j}$ are, by definition, $m[i, k]$ and $m[k + 1, j]$, respectively.

We may assume that these values have been computed previously and are already stored in our array. Since $A_{i..k}$ is a $p_{i-1} \times p_k$ matrix, and $A_{k+1..j}$ is a $p_k \times p_j$ matrix, the time to multiply them is $p_{i-1}p_kp_j$. This suggests the following recursive rule for computing $m[i, j]$.

$$
\begin{aligned}
m[i,i] &= 0 \\
m[i,j] &= \min_{i \le k < j} (m[i,k] + m[k+1,j] + p_{i-1} p_k p_j) \qquad \text{for } i < j.
\end{aligned}
$$



It is not hard to convert this rule into a procedure, which is given below. The only tricky part is arranging the order in which to compute the values. In the process of computing m[i, j] we need to access values m[i, k] and m[k+1, j] for k lying between i and j. This suggests that we should organize our computation according to the number of matrices in the subsequence. Let L = j−i+1 denote the length of the subchain being multiplied. The subchains of length 1 (m[i, i]) are trivial to compute. Then we build up by computing the subchains of lengths 2, 3, . . . , n. The final answer is m[1, n]. We need to be a little careful in setting up the loops. If a subchain of length L starts at position i, then j = i + L − 1. Since we want j ≤ n, this means that i + L − 1 ≤ n, or in other words, i ≤ n−L+1. So our loop for i runs from 1 to n−L+1 (in order to keep j in bounds). The code is presented below.
The array s[i, j] is used to extract the actual sequence. The running time of the procedure is $\Theta(n^3)$. The key is that there are three nested loops, and each can iterate at most n times.

Extracting the final Sequence: Extracting the actual multiplication sequence is a fairly easy extension. The basic idea is to leave a split marker indicating what the best split is, that is, the value of k that leads to the minimum value of m[i, j].

Chain Matrix Multiplication

```
Matrix-Chain(array p[1..n]) {
    array s[1..n-1,2..n]
    for i = 1 to n do m[i,i] = 0;              // initialize
    for L = 2 to n do {                        // L = length of subchain
        for i = 1 to n-L+1 do {
            j = i + L - 1;
            m[i,j] = INFINITY;
            for k = i to j-1 do {              // check all splits
                q = m[i, k] + m[k+1, j] + p[i-1]*p[k]*p[j]
                if (q < m[i, j]) {
                    m[i,j] = q;
                    s[i,j] = k;
                }
            }
        }
    }
    return m[1,n] (final cost) and s (splitting markers);
}
```

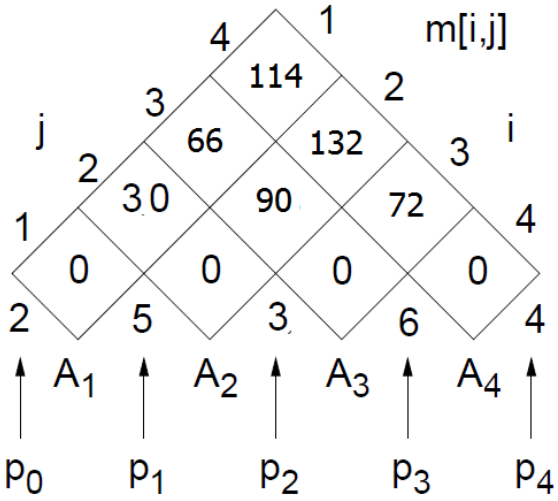
We can maintain a parallel array s[i, j] in which we will store the value of k providing the optimal split. For example, suppose that s[i, j] = k. This tells us that the best way to multiply the subchain Ai..j is to first multiply the subchain Ai..k and then multiply the subchain Ak+1..j , and finally multiply these together. Intuitively, s[i, j] tells us what multiplication to perform last. Note that we only need to store s[i, j] when we have at least two matrices, that is, if j > i.
The actual multiplication algorithm uses the s[i, j] value to determine how to split the current sequence. Assume that the matrices are stored in an array of matrices A[1..n], and that s[i, j] is global to this recursive procedure.
The recursive procedure Mult does this computation and below returns a matrix.

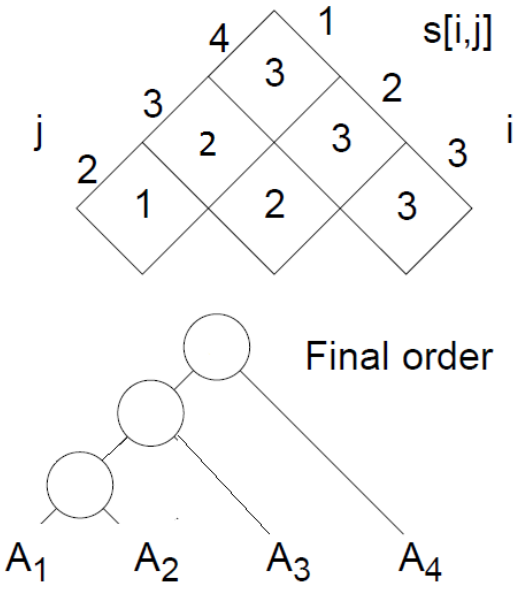
Extracting Optimum Sequence

```
Mult(i, j) {
    if (i == j)                                // basis case
        return A[i];
    else {
        k = s[i,j]
        X = Mult(i, k)                         // X = A[i]...A[k]
        Y = Mult(k+1, j)                       // Y = A[k+1]...A[j]
        return X*Y;                            // multiply matrices X and Y
    }
}
```

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

The initial set of dimensions are <2, 5, 3, 6, 4> meaning that we are multiplying A1 (2 × 5) times A2 (5 × 3) times A3 (3 × 6) times A4 (6 × 4). The optimal sequence is (((A1A2)A3)A4) which results in 114 multiplications.


m[i,j]
j
i
1
2
3
4
114
66
132
30
90
72
0
0
0
0
2
5
3
6
4
A1
A2
A3
A4
p0
p1
p2
p3
p4


s[i,j]
j
i
1
2
3
4
3
2
3
2
3
1
2
3
Final order
A1
A2
A3
A4