CSCI 532 – Algorithm Design
Assignment 6

Name: Badarika Namineni **CWID:** 50233279

Question 1: Find an optimal parenthesization of a matrix-chain product whose sequence of dimensions is (5, 10, 3, 12, 5, 50, 6)

Solution:

We have p0 = 5, p1=10, p2=3, p3=12, p4=5, p5=50, p6=6

Using the matrix-chain,
A1 = 5 x 10
A2 = 10 x 3
A3 = 3 x 12
A4 = 12 x 5
A5 = 5 x 50
A6 = 50 x 6

If i = j, then m [i, j] = 0
If I <j, then min {m [i, k] + m [k+1, j] + pi –1*pk*pj}

M [1,1] = m [2,2] = m [3,3] = m [4,4] = m [5,5] = m [6,6] = 0

m [1,2] = p0xp1xp2 = 5x10x3 = 150
m [2,3] = p1xp2xp3 = 10x3x12 = 360
m [3,4] = p2xp3xp4 = 3x12x5 = 180
m [4,5] = p3xp4xp5 = 12x5x50 = 3000
m [5,6] = p4xp5xp6 = 5x50x6 = 1500

m [1,3] = min {m [1,1] + m [2,3] + p0*p1*p3 = 960}
{m [1,2] + m [3,3] + p0*p2*p3 = 330}

m [1,4] = min {m [1,1] + m [2,4] + p0*p4*p1 = 580}
{m [1,2] + m [3,4] + p0*p2*p4 = 405}
{m [1,3] + m [4,4] + p0*p3*p4 = 630}

m [2,4] = min **{m [2,2] + m [3,4] + p1*p2*p4 = 330**
{m [2,3] + m [4,4] + p2*p3*p4 = 540}

m [1,5] = min {m [1,1] + m [2,5] + p0*p1*p5 = 4930}
{m [1,4] + m [5,5] + p0*p4*p5 = 1655}

m [2,5] = min **{m [2,2] + m [3,5] + p1*p2*p5 = 2430}**
{m [2,3] + m [4,5] + p1*p3*p5 = 9360}

m [3,5] = min {m [3,3] + m [4,5] + p2*p3*p5 = 4800}
{m [3,4] + m [5,5] + p2*p4*p5 = 930}

m [1,6] = min {m [1,1] + m [2,6] + p0*p1*p6 = 2250}
{m [1,2] + m [3,6] + p0*p2*p6 = 2010}

m [2,6] = min **{m [2,2] + m [3,6] + p1*p2*p6 = 1950}**
{m [2,4] + m [5,6] + p1*p4*p6 = 2130}

m [3,6] = min {m [3,3] + m [4,6] + p2*p3*p6 = 2076}
{m [3,4] + m [5,6] + p2*p4*p6 = 1770}

m [4,6] = min **{m [4,4] + m [5,6] + p3*p4*p6 = 1860}**
{m [4,5] + m [6,6] + p3*p5*p6 = 6600}

m	1	2	3	4	5	6
1	0	150	330	405	1655	2010
2		0	360	330	2430	1950
3			0	180	930	1770
4				0	3000	1860
5					0	1500
6						0

From this table, we create S table

m	1	2	3	4	5	6
1	0	1	2	2	4	2
2		0	2	2	2	2
3			0	3	4	4
4				0	4	4
5					0	5
6						0

So, the final multiplication sequence is **(A1*A2) (A3*A4) (A5*A6)**

Question 2: Describe the subproblem graph for matrix-chain multiplication with an input chain of length n. How many vertices does it have? How many edges does it have, and which edges are they?

Solution:

The subproblem graph for matrix-chain multiplication has a vertex of each pair (i, j) such that 1<=i<=j<=n. There are $\frac{n(n+1)}{2}$vertices and the number of edges is $\sum_{i=1}^{n}(n-i)(n-i+1)/2$
Let's substitute x =(n-i)

$\sum_{i=1}^{n} r(r+1)/2$
=1/2 ((n-1)n(2n-1)/6 + (n-1)n/2)

= (n-1) n (n+1)/6

Thus, there are $\Theta(n^2)$ vertices and $\Theta(n^3)$ edges

Question 3: Which is a more efficient way to determine the optimal number of multiplications in a matrix-chain multiplication problem: enumerating all the ways of parenthesizing the product and computing the number of multiplications for each, or running RECURSIVE-MATRIX-CHAIN? Justify your answer.

Solution:

Running RECURSIVE-MATRIX-CHAIN is more efficient way to determine the optimal number of multiplications than enumerating all the ways of parenthesizing the product.

Let us consider the treatment of subproblems by the two approaches:

- For each possible place to split the matrix chain, the enumeration approach ends all ways to parenthesize the left half, finds all ways to parenthesize the right half, and looks at all possible combinations of the left half with the right half. The amount of work to look at each combination of left- and right-half subproblem results is thus the product of the number of ways to do the left half and the number of ways to do the right half.
- For each possible place to split the matrix chain, RECURSIVE-MATRIX-CHAIN finds the best way to parenthesize the left half, finds the best way to parenthesize the right half, and combines just those two results. Thus, the amount of work to combine the left- and right-half subproblem results is O (1)

The running time of enumeration is $\Omega(4^n/n^{3/2})$ and the running time of RECURSIVE-MATRIX-CHAIN is $O(n3^{n-1})$

Question 4: Give a dynamic-programming algorithm for the activity-selection problem, based on recurrence (16.2). Have your algorithm compute the sizes c [i, j] as defined above and also produce the maximum-size subset of mutually compatible activities. Assume that the inputs have been sorted as in equation (16.1). Compare the running time of your solution to the running time of GREEDY-ACTIVITY-SELECTOR.

Solution:

If activity k is in Sij , then we must have $i < k < j$, which means that $j - i \geq 2$, but we must also have that $f_i \leq s_k$ and $f_k \leq s_j$. If we start k at $j - 1$ and decrement k, we can stop once k reaches i, but we can also stop once we find that k.

We create two fictitious activities, a_0 with $f_0 = 0$ and a_{n+1} with $s_{n+1} = \infty$. We are interested in a maximum-size set $A_{0,n+1}$ of mutually compatible activities in $S_{0.n+1}$. We'll use tables c [0..n + 1, 0..n + 1] (so that c[i, j] = |Aij |, and act[0..n + 1, 0..n + 1], where act[i, j] is the activity k that we choose to put into Aij .

We fill the tables in according to increasing difference $j - i$, which we denote by l in the pseudocode. Since Sij = ∅ if $j - i < 2$, we initialize c[i, j] = 0 for all i and c[i, i + 1] = 0 for $0 \leq i \leq n$. As in RECURSIVE-ACTIVITY -SELECTOR and GREEDY -ACTIVITY-SELECTOR, the start and finish times are given as arrays s and f , where we assume that the arrays already include the two fictitious activities and that the activities are sorted by monotonically increasing finish time.

GREEDY-ACTIVITY-SELECTOR runs in Θ(n) time and the DYNAMIC ACTIVITY -SELECTOR procedure runs in $O(n^3)$ time.

Question 5: Not just any greedy approach to the activity-selection problem produces a maximum-size set of mutually compatible activities. Give an example to show that the approach of selecting the activity of least duration from among those that are compatible with previously selected activities does not work. Do the same for the approaches of always selecting the compatible activity that overlaps the fewest other remaining activities and always selecting the compatible remaining activity with the earliest start time.

Solution:

Let's consider the optimality of greedily selecting the shortest. Suppose, our activity times are {(1, 9), (8, 11), (10, 20)} then, picking the shortest first, we have to eliminate the other two. Whereas if we picked the other two instead, we would have two tasks.

Let's consider the optimality of greedily selecting the task that conflicts with the fewest remaining activities. Suppose the activity times are {(−1, 1),(2, 5),(0, 3),(0, 3),(0, 3),(4, 7),(6, 9),(8, 11),(8, 11),(8, 11),(10, 12)}. Then, by this greedy strategy, we would first pick (4, 7) since it only has two conflicts. However, doing so would mean that we would not be able to pick the only optimal solution of (−1, 1), (2, 5), (6, 9), (10, 12).

Now, let's consider the optimality of greedily selecting the earliest start times, suppose our activity times are {(1, 10),(2, 3),(4, 5)}. If we pick the earliest start time, we will only have a single activity, (1, 10), whereas the optimal solution would be to pick the two other activities.

Question 6: Give a dynamic-programming solution to the 0-1 knapsack problem that runs in O (n W) time, where n is the number of items and W is the maximum weight of items that the thief can put in his knapsack.

Solution:

Suppose we know that a particular item of weight w is in the solution. Then we must solve the subproblem on n − 1 items with maximum weight W − w. To take a bottom-up approach we must solve the 0-1 knapsack problem for all items and possible weights smaller than W. We'll build n + 1 by W + 1 table of values where the rows are indexed by item and the columns are indexed by total weight. For row i column j, we decide whether or not it would be advantageous to include item i in the knapsack by comparing the total value of a knapsack including items 1 through (i – 1) with max weight j, and the total value of including items 1 through (i – 1) with max weight (j – i) weight and also item i. To solve the problem, we simply examine the n, W entry of the table to determine the maximum value we can achieve. To read off the items we include, start with entry n, W. In general, proceed as follows: if entry i, j equals entry i−1, j, don't include item i, and examine entry i−1, j next. If entry i, j doesn't equal entry i − 1, j, include item i and examine entry i − 1, j – i.