**CPSC 6109:** [**Advanced**](https://colstate.view.usg.edu/d2l/lp/ouHome/home.d2l?ou=1218642) **Algorithms**

**Spring 2018**

**Assignment #3**

**Student: Lu Lin**

**Date: Feb.20, 2018**

**Due: 11:59 PM Tuesday, Feb. 20**

Do the following exercises/problems. Each problem is worth 50 points with a total of 100 points.

1. Find an optimal parenthesization of a matrix-chain product whose sequence of dimensions is <2, 10, 5, 12, 2, 20, 5, 4>. That is, the dimensions of each matrix in the sequence < A1, A2, A3, A4, A5, A6, A7> are listed below:

A1 = 2 × 10

A2 = 10 × 5

A3 = 5 × 12

A4 = 12 × 2

A5 = 2 × 20

A6 = 20 × 5

A7 = 5 × 4.

Solution:

First of all, all the sequence of dimensions mean that we have 7 matrices with these specified dimensions:

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Matrix** | **A1** | | **A2** | | **A3** | | **A4** | | **A5** | | **A6** | | **A7** | |
| **p-value** | p0=2 | p1=10 | p1=10 | p2=5 | p2=5 | p3=12 | p3=12 | p4=2 | p4=2 | p5=20 | p5=20 | p6=5 | p6=5 | p7=4 |
| **dimension** | 2 × 10 | | 10 × 5 | | 5 × 12 | | 12 × 2 | | 2 × 20 | | 20 × 5 | | 5 × 4 | |

Secondly, we need a 7-by-7 table/array that is m. Initially, we set m[i, i] = 0 for every 1 ≤ i ≤ 7. As the definition m[i, j] only for i ≤ j, only the portion of the table m above the main diagonal is used. The m table uses only the main diagonal and upper triangle, and the later s table will only use the upper triangle (as Figure 15.5 shows). Now consider all chain length = 2 situation. In this situation there is only one possible split, thus there is no choice except one. We would have:

m[1, 2] = p0 \* p1 \* p2 = 2\*10\*5 = 100; likewise m[2, 3] = p1 \* p2 \* p3 = 600; m[3, 4] = p2 \* p3 \* p4 = 120; m[4, 5] = p3 \* p4 \* p5 = 480; m[5, 6] = p4 \* p5 \* p6 = 200; m[6, 7] = p5 \* p6 \* p7 = 400. And because of its one possible split we have s[1, 2] = 1, s[2, 3] = 2, s[3, 4] = 3, s[4, 5] = 4, s[5, 6] = 5, and s[6, 7] = 6.

Based on recurrence 15.7 we would compute the optimal costs. Consider chain length = 3 situation, we must fill in cells m[1, 3], m[2, 4] m[3, 5], m[4, 6] and m[5, 7]. For m[1, 3], we have two choices: k = 1 or k = 2. Calculations are as following:

k = 1: m[1, 1] + m[2, 3] + p0 \* p1 \* p3 = 0 + 600 + 240 = 840

k = 2: m[1, 2] + m[3, 3] + p0 \* p2 \* p3 = 100 + 0 + 120 = 220

Hence we would set m[1, 3] = 220. According to textbook s[i, j] records a value of k such that an optimal parenthesization splits the product between Ak and Ak+1. Hence s[1, 3] = 2.

Similarly for m[2, 4] the calculations are as following:

k = 2: m[2, 2] + m[3, 4] + p1 \* p2 \* p4 = 0 + 120 + 100 = 220

k = 3: m[2, 3] + m[4, 4] + p1 \* p3 \* p4 = 600 + 0 + 240 = 840

Hence m[2, 4] = 220 and s[2, 4] = 2.

For m[3, 5] the calculations are as following:

k = 3: m[3, 3] + m[4, 5] + p2 \* p3 \* p5 = 0 + 480 + 1200 = 1680

k = 4: m[3, 4] + m[5, 5] + p2 \* p4 \* p5 = 120 + 0 + 200 = 320

Hence m[3, 5] = 320 and s[3, 5] = 4.

For m[4, 6] the calculations are as following:

k = 4: m[4, 4] + m[5, 6] + p3 \* p4 \* p6 = 0 + 200 + 120 = 320

k = 5: m[4, 5] + m[6, 6] + p3 \* p5 \* p6 = 480 + 0 + 1200 = 1680

Hence m[4, 6] = 320 and s[4, 6] = 4.

For m[5, 7] the calculations are as following:

k = 5: m[5, 5] + m[6, 7] + p4 \* p5 \* p7 = 0 + 400 + 160 = 560

k = 6: m[5, 6] + m[7, 7] + p4 \* p6 \* p7 = 200 + 0 + 40 = 240

Hence m[5, 7] = 240 and s[5, 7] = 6.

Now consider chain length = 4 situation, we must fill in cells m[1, 4], m[2, 5], m[3, 6] and m[4, 7].

For m[1, 4] the calculations are as following:

k = 1: m[1, 1] + m[2, 4] + p0 \* p1 \* p4 = 0 + 220 + 40 = 260

k = 2: m[1, 2] + m[3, 4] + p0 \* p2 \* p4 = 100 + 120 + 20 = 240

k = 3: m[1, 3] + m[4, 4] + p0 \* p3 \* p4 = 220 + 0 + 48 = 268

Hence m[1, 4] = 240 and s[1, 4] = 2.

For m[2, 5] the calculations are as following:

k = 2: m[2, 2] + m[3, 5] + p1 \* p2 \* p5 = 0 + 320 + 1000 = 1320

k = 3: m[2, 3] + m[4, 5] + p1 \* p3 \* p5 = 600 + 480 + 2400 = 3480

k = 4: m[2, 4] + m[5, 5] + p1 \* p4 \* p5 = 220 + 0 + 400 = 620

Hence m[2, 5] = 620 and s[2, 5] = 4.

For m[3, 6] the calculations are as following:

k = 3: m[3, 3] + m[4, 6] + p2 \* p3 \* p6 = 0 + 320 + 300 = 620

k = 4: m[3, 4] + m[5, 6] + p2 \* p4 \* p6 = 120 + 200 + 50 = 370

k = 5: m[3, 5] + m[6, 6] + p2 \* p5 \* p6 = 320 + 0 + 500 = 820

Hence m[3, 6] = 370 and s[3, 6] = 4.

For m[4, 7] the calculations are as following:

k = 4: m[4, 4] + m[5, 7] + p3 \* p4 \* p7 = 0 + 240 + 96 = 336

k = 5: m[4, 5] + m[6, 7] + p3 \* p5 \* p7 = 480 + 400 + 960 = 1840

k = 6: m[4, 6] + m[7, 7] + p3 \* p6 \* p7 = 320 + 0 + 240 = 560

Hence m[4, 7] = 336 and s[4, 7] = 4.

Now consider chain length = 5 situation, we must fill in cells m[1, 5], m[2, 6] and m[3, 7].

For m[1, 5] the calculations are as following:

k = 1: m[1, 1] + m[2, 5] + p0 \* p1 \* p5 = 0 + 620 + 400 = 1020

k = 2: m[1, 2] + m[3, 5] + p0 \* p2 \* p5 = 100 + 320 + 200 = 620

k = 3: m[1, 3] + m[4, 5] + p0 \* p3 \* p5 = 220 + 480 + 480 = 1180

k = 4: m[1, 4] + m[5, 5] + p0 \* p4 \* p5 = 240 + 0 + 80 = 320

Hence m[1, 5] = 320 and s[1, 5] = 4.

For m[2, 6] the calculations are as following:

k = 2: m[2, 2] + m[3, 6] + p1 \* p2 \* p6 = 0 + 370 + 250 = 620

k = 3: m[2, 3] + m[4, 6] + p1 \* p3 \* p6 = 600 + 320 + 600 = 1520

k = 4: m[2, 4] + m[5, 6] + p1 \* p4 \* p6 = 220 + 200 + 100 = 520

k = 5: m[2, 5] + m[6, 6] + p1 \* p5 \* p6 = 620 + 0 + 1000 = 1620

Hence m[2, 6] = 520 and s[2, 6] = 4.

For m[3, 7] the calculations are as following:

k = 3: m[3, 3] + m[4, 7] + p2 \* p3 \* p7 = 0 + 336 + 240 = 576

k = 4: m[3, 4] + m[5, 7] + p2 \* p4 \* p7 = 120 + 240 + 40 = 400

k = 5: m[3, 5] + m[6, 7] + p2 \* p5 \* p7 = 320 + 400 + 400 = 1120

k = 6: m[3, 6] + m[7, 7] + p2 \* p6 \* p7 = 370 + 0 + 100 = 470

Hence m[3, 7] = 400 and s[3, 7] = 4.

Now consider chain length = 6 situation, we must fill in cells m[1, 6] and m[2, 7].

For m[1, 6] the calculations are as following:

k = 1: m[1, 1] + m[2, 6] + p0 \* p1 \* p6 = 0 + 520 + 100 = 620

k = 2: m[1, 2] + m[3, 6] + p0 \* p2 \* p6 = 100 + 370 + 50 = 520

k = 3: m[1, 3] + m[4, 6] + p0 \* p3 \* p6 = 220 + 320 + 120 = 660

k = 4: m[1, 4] + m[5, 6] + p0 \* p4 \* p6 = 240 + 200 + 20 = 460

k = 5: m[1, 5] + m[6, 6] + p0 \* p5 \* p6 = 320 + 0 + 200 = 520

Hence m[1, 6] = 460 and s[1, 6] = 4.

For m[2, 7] the calculations are as following:

k = 2: m[2, 2] + m[3, 7] + p1 \* p2 \* p7 = 0 + 400 + 200 = 600

k = 3: m[2, 3] + m[4, 7] + p1 \* p3 \* p7 = 600 + 336 + 480 = 1416

k = 4: m[2, 4] + m[5, 7] + p1 \* p4 \* p7 = 220 + 240 + 80 = 540

k = 5: m[2, 5] + m[6, 7] + p1 \* p5 \* p7 = 620 + 400 + 800 = 1820

k = 6: m[2, 6] + m[7, 7] + p1 \* p6 \* p7 = 520 + 0 + 200 = 720

Hence m[2, 7] = 540 and s[2, 7] = 4.

Finally, we must fill in cells m[1, 7]. For m[1, 7] the calculations are as following:

k = 1: m[1, 1] + m[2, 7] + p0 \* p1 \* p7 = 0 + 540 + 80 = 620

k = 2: m[1, 2] + m[3, 7] + p0 \* p2 \* p7 = 100 + 400 + 40 = 540

k = 3: m[1, 3] + m[4, 7] + p0 \* p3 \* p7 = 220 + 336 + 96 = 652

k = 4: m[1, 4] + m[5, 7] + p0 \* p4 \* p7 = 240 + 240 + 16 = 496

k = 5: m[1, 5] + m[6, 7] + p0 \* p5 \* p7 = 320 + 400 + 160 = 880

k = 6: m[1, 6] + m[7, 7] + p0 \* p6 \* p7 = 460 + 0 + 40 = 500

Hence m[1, 7] = 496 and s[1, 7] = 4.

Now we have both of m and s table as shown in Figure 1 and 2:

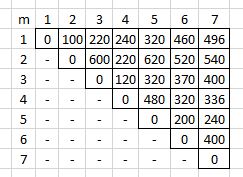


Figure 1: m-table

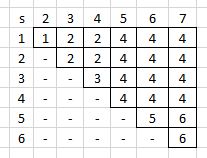


Figure 2: s-table

According to textbook an optimal parenthesization of this < A1, A2, A3, A4, A5, A6, A7> is (((A1A2)(A3A4))((A5A6)A7))

1. Problem ***15-6*** on page 408. Professor Steward is consulting for the president of a corporation that is planning a company party. The company has a hierarchical structure; that is, the supervisor relation forms a tree rooted at the president. The personnel office has ranked each employee with a conviviality rating, which is a real number. In order to make the party fun for all attendees, the president does not want both an employee and his or her immediate supervisor to attend. Professor Steward is given the tree that describes the structure of the corporation, using the left-child, right-sibling representation described in Section 10.4. Each node of the tree holds, in addition to the pointers, the name of an employee and that employee’s conviviality ranking. Describe an algorithm to make up a guest list that maximizes the sum of the conviviality ratings of the guests. Analyze the running time of your algorithms.

You must prove that this problem exhibits ***optimal substructure*** so that this problem can be solved by using the Dynamic Programming method, and thenderive a recursive solution. Implementation and pseudo code of the Dynamic Programming algorithm is NOT required.

Solution:

According to given problem, the company hierarchy would make a tree with left-child right-sibling structure; which makes these sets S1, S2, S3,…,Sn where the set Si contains immediate subordinates of the ith employee for people p1, p2, p3,…,pn in this company (the tree top root, the president, is p1). Immediate supervisor and his/her subordinates will not attend together means for every employee(tree node) he/she has two choices: to go or not go; if the choice is to go then the choice of its child nodes is not go, vice versa if a node’s choice is not go then the choice of its child nodes is to go or not go. For every node we could add attributes for conviviality value (a real number) in situations of to go or not go. We’d calculate every tree node’s(employee’s) maximum total conviviality value by taking its children and grandchildren into consideration. The sum of conviviality of a guest list is the sum of conviviality of people who attend the party. Accordingly I define a function *C(i)* that calculate every node’s conviviality value and three quantities representing to go or not go as following:

*[i]: the conviviality value of employee pi*

*conviviality\_go[i]: the max conviviality of employee pi and his/her team members if he/she goes to party*

*conviviality\_notgo[i]: the max conviviality of employee pi and his/her team members if he/she does not go to party*

Function *C(i)* is defined as max(*conviviality\_go[i], conviviality\_notgo[i]*). Here

*conviviality\_go[i]* = )

*conviviality\_notgo[i]* =

Hence *C(i)* = max

The above recurrence my recursive solution for provided problem. When number i has no j, which means the corresponding tree node i is a leaf (have no children), the function *C(i) = .*

The *conviviality\_go[i]* means the current node i (employee pi) is selected for party and thus pi’s immediate subordinates (children) are not be able to be chosen to the party, but immediate subordinates’ immediate subordinates (grandchildren) are. The *conviviality\_notgo[i]* means the current node is not chosen for the party but its children are able to be chosen to go or not to go. The algorithm is based on the recurrence relationship as stated above.

In dynamic programming, if one wants optimal solution of a problem then it is required to find the optimal solution of a sub-problem of same form. Obviously from the definition *C(*1*)* is what we want to calculate with (to choose the max value of the situations of either president went to party or not), the maximum conviviality value of the guest list. The base case is, consider the tree with one single node, the function *C(i)* would be this node’s conviviality value. As we can see from the above recurrence that *C(i)* is depends on i’s immediate subordinate j (), and we’d conduct calculations in a bottom-up way in the company hierarchy tree from n . In conclusion, this provided problem exhibit optimal substructure. Here is the recursive solution, and this recursive solution is thus the proof of optimality.

Function (n)

let c\_go[0..n] be a new array

let c\_not\_go[0..n] be a new array

for i = 0 to n

c\_go[i] = -1

c\_not\_go[i] = -1

return max(Helper(i, true, c\_go, c\_not\_go), Helper(i, false, c\_go, c\_not\_go))

Helper (i, go, c\_go, c\_not\_go)

let Si be the set of children of node i

let c[i] be the conviviality of node i itself

if Si is empty

if go

return c[i]

else

return 0

if go

if c\_go[i] >= 0

return c\_go[i]

max\_i = c[i]

for j in Si

max\_i += Helper (j, false, c\_go, c\_not\_go)

else

if c\_not\_go[i] >= 0

return c\_not\_go[i]

max\_i = 0

for j in Si

max\_i += max(Helper (j, true, c\_go, c\_not\_go), Helper (j, false, c\_go, c\_not\_go))

return max\_i

The provided problem can be solved in linear time with the help of dynamic programming. The recursive procedure is top-down with memorization. For each node i, c\_go[i] and c\_not\_go[i] are only computed once. Considering n nodes, the total time complex is T(n) = n\*O(2) = O(n).

|  |  |
| --- | --- |
| Submission Feedback |  |
| 1)   2) error in the proof for optimal substructure -8 | |

**CPSC 6109:** [**Advanced**](https://colstate.view.usg.edu/d2l/lp/ouHome/home.d2l?ou=1218642) **Algorithms**

**Spring 2018**

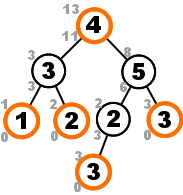
**Assignment #3 – Problem 2**

**Soluitons**

(2) Problem ***15-6*** on page 408.

**Solution:**

Here is an example to help you understand the problem – a tree with each node associated with a number (i.e., a conviviality rating):



In this example, the optimal solution contains all the tree nodes colored orange. The total value of all the nodes in the optimal solution is:

4 + 1 + 2 + 3 + 3 = 13.

**Proof for optimal substructure**:

Let T denote the original tree, **root(T)** the root node of the T, and **V(T)** the set of all the nodes of T. Given an optimal solution S to the original problem T. Then S is a subset of V(T) and any two nodes in S is NOT adjacent in T. Now remove an edge connecting to root(T) from T. This removal divides T into two subtrees T’ and T’’. The set S is also divided into two smaller subsets S’ and S’’ such that all nodes of S’ belong to T’ and all nodes of S’’ belong to T’’. It is easy to see that either root(T’) is in S or root(T’’) is in S. Without loss of generality, assume that root(T’) is in S. Consider the sub-problem T’ whose root node is in S. We only need to prove that S’ must be an optimal solution of T’. This can be easily verified by using the cut-and-paste method.

**The dynamic programming algorithm to solve this problem**:

We make an array C[u] indexed by vertex u ϵ V(T) which is the optimal conviviality ranking of a person list obtained from the subtree rooted at the vertex u. We also make an array G such that G[v] is the person list we would use when vertex v is at the root. To solve the problem, we need to find the person list stored at G[root(T)] and compute the total value C[root(T)].

First solve the problem at each leaf node L (the base case):

G[L] = {L} if the node L is in the solution, otherwise G[L] is empty

C[L] = L.conviviality-rank if the node L is in the solution, otherwise C[L] = 0

Iteratively, we solve the subproblems located at parents of nodes at which the subproblems have been solved. In general, for any node x, the recursive equation to compute C[x] is given below:



The total running time of the algorithm is O(n2), where n is the number of vertices in T, because we solve n subproblems, each in constant time, but the tree traversals required to find the appropriate next node to solve could take linear time.