CSCI 5421 - Homework 5

Email	neis@umn.edu
Name:	Kelsey Neis

15.2-2

```
Matrix-Chain-Multiply(Z, s, i, j):
1 if A.length == 1 //if there is only one matrix, return it
2   return A
3 if i == (j-1) //if i and j are adjacent, perform multiplication
4   A_i = Matrix-Multiply(A_i, A_j)
5   remove A_j from A //keep only the product of A_i and A_j
6   return
7 Matrix-Chain-Multiply(A, s, i, s[i, j])
8 Matrix-Chain-Multiply(A, s, i+1, j - s[i, j] + i)
```

The right-hand side i and j values are adjusted assuming that once the left-hand side (line 7) is finished recursing, it will be reduced down to one matrix. So, the right hand recursive call starts at i + 1 and ends at j - (s[i, j] - i).

15.2-3

$$P(n) = \begin{cases} 1 & \text{if } n = 1\,, \\ \sum_{k=1}^{n-1} P(k)P(n-k) & \text{if } n \geq 2\,. \end{cases} \tag{15.6}$$

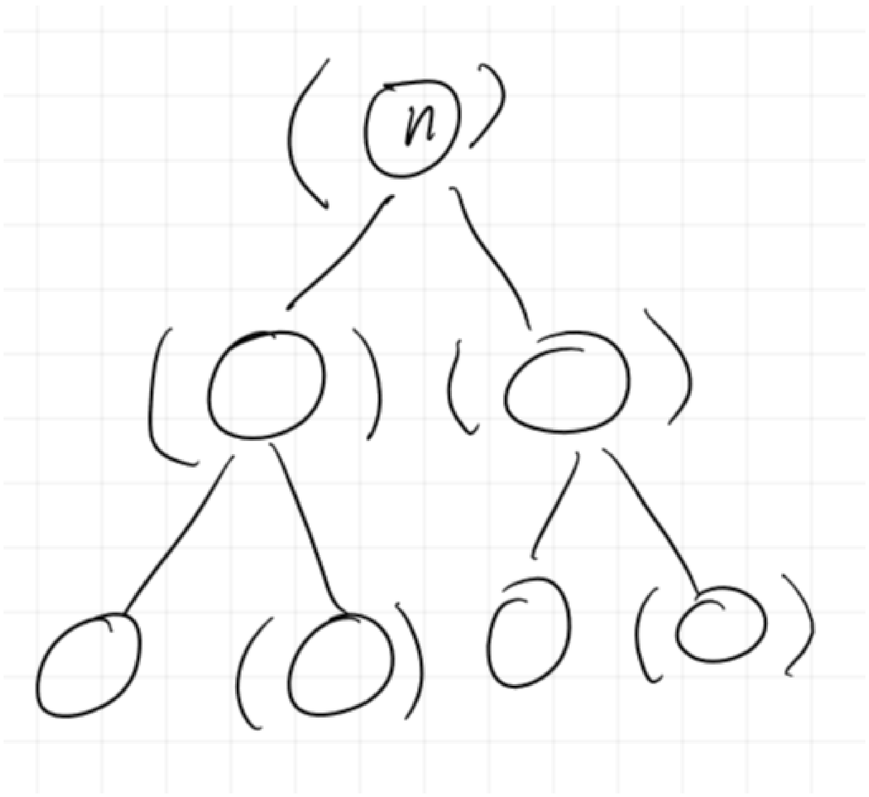
$P(n)$ is a convolution of all subproblem possibilities. Therefore, it has the structure of a decision tree. Unlike with the decision tree, though, all nodes in the tree need to be considered in order to determine the optimal solution. Since the problem size is n, the decision tree will have 2^n nodes. So, the solution to the recurrence can be no less than $\Omega(2^n)$.

15.2-4

A dependency graph for matrix chain multiplication of n matrices would have $n+1$ vertices. Each parent vertex would have two child vertices - one for the left subproblem, one for the right subproblem. At the bottom, a vertex will have two children only if the parent is greater than size 2, otherwise it will have one child, the base case. The sizes would be k for left and $j-i+1$ for a matrix chain $A_{i \ldots j}$. The edges would go from each parent to its immediate children, so there would be n edges.

15.2-6

Given a subproblem graph like the one below, which has $n+1$ vertices, parentheses will be needed for all non-leaf vertices and for the floor of half of the leaves, because if the leaves have a sibling, then it means only one of them is more than one matrix, and if they don't have a sibling, it is the base case of one matrix and does not need parentheses.



15.4-2

```
Reconstruct-LCS(c, X, Y, i, j):
  LCS = []
  if i == 0 or j == 0:
    return
  if X_i == Y_j:
    Reconstruct-LCS(c, X, Y, i-1, j-1)
    print X_i
  elseif c[i-1, j] == c[i, j]:
    Reconstruct-LCS(c, X, Y, i-1, j)
  else
    Reconstruct-LCS(c, X, Y, i, j-1)
```

Runtime: It will be m+n because either i or j or both are decremented at each recursive call. So, even in the worst case, where X_i and Y_j are never equal, the code will go through m + n cells of the table.

15.4-3

```
Memoized-LCS-Length(X, Y, c)
1 n = X.length
2 m = Y.length
3 let c be a new mxn table
4 for i = 1 to m:
5   for j = 1 to n:
6     c[i, j] = -infinity
7 return Lookup-LCS(X, Y, c, m, n)

Lookup-LCS(X, Y, c, i, j)
1 if c[i, j] > -infinity:
2   return c[i, j]
3 if i == 0 or j == 0
4   return 0
5 if X_i == Y_j:
6   c[i, j] = c[i - 1, j - 1] + 1
7 else
8   c[i, j] = MAX(Lookup-LCS(X, Y, c, i-1, j) , Lookup-LCS(X, Y, c, i, j-1))
9 return c[i, j]
```

Runtime is $O(n)$ because Memoized-LCS-Length requires time mn to populate the table in lines 4-6 and Lookup-LCS traverses the table without revisiting any cells.

15.4-5

1. Build an $n + 1$ by $n + 1$ table, populating the bottom half triangle with 0 and a bugger row and column with 0 as well:

Initialize table

0	O	x1	x2	x3	x4
0	0	0	0	0	0
y1	0	0			
y2	0	0	0		
y3	0	0	0	0	
y4	0	0	0	0	0

Here, x and y are both the same set, duplicated to allow comparisons

2. Fill in the rest of the table as follows:

- if X_i > Y_j, c[i, j] = c[i-1, j-1] + 1
- else if c[i-1, j] > c[i, j-1], c[i, j] = [i-1, j]
- else: c[i, j] = c[i, j-1]

(Note: this is similar to how the table is filled for LCS)

- Keep track of the highest number stored in the table

3. Starting from the lower right corner of the table, work your way back diagonally until the highest number is reached (which was stored in step 2). This will be the last number in the optimal solution. Add it to the array.

    - keep going with this process, adding to the array every number that is both the highest number among those with its same table value and does not exceed the last value in the solution array.

Runtime: At most the algorithm will iterate through each nxn grid cell once, never revisiting a cell, since we're always going in a decreasing i or j or both direction. In fact, it will not visit all cells because it cannot traverse backwards to cover all cells. The rest of the operations are constant time, giving a runtime of $O(n^2)$.