Michael Kaiser
Assignment 1
Comp 510
Fall 2012

(2nd edition: 15.2-1): Matrix Chain Multiplication.
Find an optimal parenthesization of a matrix-chain product whose sequence of dimensions is: (5, 10, 3, 12, 5, 50, 6).

From the book, we have the algorithm MATRIX-CHAIN-ORDER(p), which will be used to solve this problem.

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

The corresponding matricies are:
A1 – 5x10
A2 – 10x3
A3 – 3x12
A4 – 12x5
A5 – 5x50
A6 – 50x6

From the algorithm, we have, for all x, m[x,x] = 0.

m[1,2] = m[1,1] + m[2,2] + p0 * p1 * p2
m[1,2] = 0 + 150
m[1,2] = 150

m[3,4] = m[3,3] + m[4,4] + p2 * p3 * p4
m[3,4] = 0 + 180
m[3,4] = 180

m[4,5] = m[4,4] + m[5,5] + p3 * p4 * p5
m[4,5] = 0 + 3000
m[4,5] = 3000

m[5,6] = m[5,5] + m[6,6] + p4 * p5 * p6
m[5,6] = 0 + 1500
m[5,6] = 1500

m[1,3] = min of { m[1,1] + m[2,3] + p0 * p1 * p3 = 750 }
{ m[1,2] + m[3,3] + p0 * p2 * p3 = 330 }

m[2,4] = min of **{ m[2,2] + m[3,4] + p1 * p2 * p4 = 330}**
{ m[2,3] + m[4,4] + p1 * p3 * p4 = 960}

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

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

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

m[2,5] = min of **{ m[2,2] + m[3,5] + p1 * p2 * p5 = 2430 }**
{ m[2,3] + m[4,5] + p1 * p3 * p5 = 9360 }
{ m[2,4] + m[5,5] + p1 * p4 * p5 = 2830 }

m[3,6] = min of { m[3,3] + m[4,6] + p2 * p3 * p6 = 2076 }
{ m[3,4] + m[5,6] + p2 * p4 * p6 = 1770 }
{ m[3,5] + m[6,6] + p2 * p5 * p6 = 1830 }

m[1,5] = { m[1,1] + m[2,5] + p0 * p1 * p5 = 4930 }
{ m[1,2] + m[3,5] + p0 * p2 * p5 = 1830 }
{ m[1,3] + m[1,4] + p0 * p3 * p5 = 6330 }
{ m[1,4] + m[1,5] + p0 * p4 * p5 = 1655 }

m[2,6] = **{ m[2,2] + m[3,6] + p1 * p2 * p6 = 1950 }**
{ m[2,3] + m[4,6] + p1 * p3 * p6 = 2940 }
{ m[2,4] + m[5,6] + p1 * p4 * p6 = 2130 }
{ m[2,5] + m[6,6] + p1 * p5 * p6 = 5430 }

m[1,6] = { m[1,1] + m[2,6] + p0 * p1 * p6 = 2250 }
{ m[1,2] + m[3,6] + p0 * p2 * p6 = 2010 }
{ m[1,3] + m[4,6] + p0 * p3 * p6 = 2550 }
{ m[1,4] + m[5,6] + p0 * p4 * p6 = 2055 }
{ m[1,5] + m[6,6] + p0 * p5 * p6 = 3155 }

M	1	2	3	4	5	6
6	2010	1950	1770	1840	1500	0
5	1655	2430	930	3000	0	
4	405	330	180	0		
3	330	360				
2	150					
1	0					

And using this, we construct the S table:

S	1	2	3	4	5
6	2	2	4	4	5
5	4	2	4	4	
4	2	2	3		
3	2	2			
2	1				

The minimum cost is therefore 2010 and the optimal parenthesization is:
((A1 * A2) * (A3 * A4) * (A5 * A6))

(2nd edition: 15.2-2) : Matrix Chain Multiplication.

Give a recursive algorithm MATRIX-CHAIN-MULTIPLY(A,s,i,j) that actually performs the optimal matrix-chain multiplication, given the sequence of matrices (A1, A2, ..., An), the s table computed by MATRIX-CHAIN-ORDER, and the indices i and j. (The initial call would be MATRIX-CHAIN-MULTIPLY(A, s, 1, n).

The Algorithm should look at follows:

```
MATRIX-CHAIN-MULTIPLY(A, s, i, j)
{
        if ( i >= j)
        {
                return A[i];
        }
        else
        {
                return MATRIX-MULTIPLY(MATRIX-CHAIN-MULTIPLY(A, s, i, s[i][j]),
                          MATRIX-CHAIN-MULTIPLY(A, s, s[i][j] + 1, j);
        }
}
```

(2nd edition: 15-1): The Bitonic Euclidean Traveling-Salesman Problem.

Describe an O(n2)-time algorithm for determining the optimal bitonic tour. You may assume that no two points have the same x-coordinate. Hint: scan left to right, maintaining optimal possibilities for the two parts of the tour.

Sort the points from left to right such that we have points $\{p_1,p_2, ... p_n\}$ in order from left to right.

We can define p_{ij} with i < j such that p_{ij} is the shortest bitonic path from p_1 -> p_i -> p_j, which includes all points from p_1 to p_j. We can then define b[i,j] to be the length of the shortest bitonic path P_{ij}.

The following rules define the dynamic programming solution:

b[1,j] = Sum from i = 1 to j – 1 of the Euclidean Distance of (i, i+1).
b[i,j] = b[i, j-1] + Euclidean Distance of (j, j-1), given that i < j – 1.
b[j-1, j] = min of each i < j - 1(b[i, j – 1] + Euclidean distance of (i,j)
b[n,n] = b[n – 1, n] + Eucldiean distance of (n-1, n).

The running time of this algorithm is $O(n^2)$ as n of the lower diagonal entries require O(n) time to compute, with the remaining entries requiring O(1) time. Thus we have n * O(n) = $O(n^2)$.

(2nd edition: 15-2): Printing Neatly. Give a dynamic programming algorithm to print a paragraph of n words neatly on a printer. Analyze the run time and space requirements for your algorithm.

Several defintions for the algorithm:

extras[i,j] = M – j + i – sum from k = i to j of l_k : this is to be the number of extra spaces at the end of a line containing the words i through j.

	{∞	if extras[i,j] < 0	(words dont fit)	}
lc[i,j] =	{0	if j = n and extras[i,j] >= 0	(last line has no cost)	}
	{$(\text{extras}[i,j])^3$	otherwise	(words fit)	}

c[j] will be the cost of an optimal arrangement of words [1,...j].
c[j] = c[i-1] + lc[i,j]
c[0] = 0 as a base case so that c[1] = lc[1,1].

The rules for c[j] are therefore

c[j] =	{ 0	if (j = 0) }
	{ min of all i, 1 <= i < j of (c[i-1] + lc[i,j])	if (j > 0) }

And lastly, we have p as a parallel table that points to where each c value orginated so that we can linebreak in the correct location. When c[j] is computed, if c[j] is based on c[k – 1], p[j] is set to k.

Thus the algorithm to create the tables for printing is:

```
PRINT-NEATLY(l, n, M)
{
        for i = 1 to n
        {
                extras[i,i] = M – l_i;
                for j = (i + 1) to n
                {
                        extras[i,j] = extras[i, j – 1] – l_j – 1;    //See definition of extras
                }
        }
        for i = 1 to n
        {
                for j = i to n
                {
                        if extras[i,j] < 0          //Words don't fit
                        {
                                lc[i,j] = ∞;
                        }
                        else if (j == n) && (extras[i,j] >= 0)
                        {
                                lc[i,j] = 0;
                        }
                        else
                        {
                                lc[i,j] = (extras[i,j])^3
                        }
```

```
        }
    }
    c[0] = 0;
    for j = 1 to n
    {
        c[j] =  ∞;
        for i = 1 to j
        {
            if (c[i-1] + lc[i,j] < c[j])
            {
                c[j] = c[i-1] + lc[i,j];
                p[j] = i;
            }
        }
    }
    return (c and p)
}
```

Then to print the words, we have the following routine:

```
PRINT-WORDS(p, j)
{
    i = p[j]
    if (i == 1)
    {
        k = 1;
    }
    else
    {
        k = PRINT-WORDS(p, i-1) + 1;
    }
    print(k, i, j);
    return k;
}
```

The algorithm's time and space are $O(n^2)$, with the ability be improved, however I'm leaving it as $O(n^2)$ to follow the instructions. Each of the loops are at most nested once, so the greatest power of n is n^2, giving us $O(n^2)$. Space requirements of are also $O(n^2)$ because extras is extras[n,n], lc is lc[n,n] and c is c[n]. The amount of space is precisely $2n^2 + n$, which is $O(n^2)$.

References

Class Textbook and solution manual (3rd ED)
http://en.wikipedia.org/wiki/Matrix_chain_multiplication
http://en.wikipedia.org/wiki/Bitonic_tour
http://mitpress.mit.edu/algorithms/solutions/chap15-solutions.pdf
http://www.google.com/url?sa=t&rct=j&q=&esrc=s&source=web&cd=5&cad=rja&ved=0CD4QFjAE&url=http%3A%2F%2Fciteseerx.ist.psu.edu%2Fviewdoc%2Fdownload%3Fdoi%3D10.1.1.135.5959%26rep%3Drep1%26type%3Dpdf&ei=qm9NUOCqIueUiQLyyYDADQ&usg=AFQjCNH9IgkwabuBn6bgXpQ1yjaxLIVg8g