MATRIX CHAIN MULTIPLICATION AND MATRIX POLYNOMIALS.

DYNAMIC PROGAMMING

- **Dynamic programming :**

  Dynamic programming is a tabular method, which solves the problem by combining the solutions to sub problems.

- It is different from divide and conquer rule as , divide and conquer solve the sub problem recursively and then combine their solutions to solve the original problem, where as in dynamic programming , the sub problems are not independent, that is sub problems share sub problems.

- dynamic programming solves every sub problem once and then saves its answer in a table , there by avoiding , recomputating the sub problem solution every time , thereby saving time , resources and is efficient when compared to divide and conquer method.
- Dynamic programming has many possible solutions and is typically applied to optimization problems , where each solution has a value and where we need to find the best or worst ie(optimal maximum or minimum) value.

The different steps in in Dynamic programming are

- Characterize the structure of an optimal solution.
- Recursively define the value of an optimal solution.
- Compute the value of an optimal solution in a bottom-up fashion.
- Construct an optimal solution from computed information.

MATRIX CHAIN MULTIPLICATION

- **Matrix chain multiplication** is an optimization problem that can be solved using dynamic programming.
- Matrix multiplication is associative , but not commutative.
- Even though the answer is same for different for different ways , we need to find an optimal way through dynamic programming.

- MATRIX-MULTIPLY(*A, B*)

```
if columns[A] != rows[B]
    then error "incompatible dimensions"
    else for i ← 1 to rows[A]
        do for j ← 1 to columns[B]
            do C[i, j ] ← 0
                for k ← 1 to columns[A]
                    do C[i, j ] ← C[i, j ] + A[i, k] · B[k, j ]
return C
```

Different steps involving in matrix chain multiplication are:

- Counting number of Parenthesizations
- Finding the Structure of an optimal parenthesization
- Finding the Recursive Solution
- Computing the optimal costs
- Constructing an optimal solution

Counting number of Parenthesizations

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

Finding the Structure of an optimal parenthesization

- In dynamic programming we need to find the optimal sub structure and then use it to construct an optimal solution to the problem from optimal solutions to subproblems.

Finding the Recursive Solution

- The recursive definition for minimum cost of parenthesizing the product $A_iA_{i+1}.....A_j$ is

$$m[i, j] = \begin{cases} 0 & if\ \ i = j \\ \min\{\ m[i,k] +\ m[k+1,j] +\ pi - 1pkpj\} if\ i < j \end{cases}$$

m[i,j] give the costs of optimal solutions to sub problems.

Computing the optimal costs

MATRIX-CHAIN-ORDER(*p*)

```
n ← length[p] − 1
for i ← 1 to n
    do m[i, i ] ← 0
for l ← 2 to n        ( l is the chain length.)
    do for i ← 1 to n − l + 1
        do j ←i + l − 1
            m[i, j ]←∞
            for k ←i to j − 1
                do q ← m[i, k] + m[k + 1, j ] + p_{i−1} p_k p_j
                    if q < m[i, j ]
                        then m[i, j ]← q
                            s[i, j ] ← k
return m and s
```

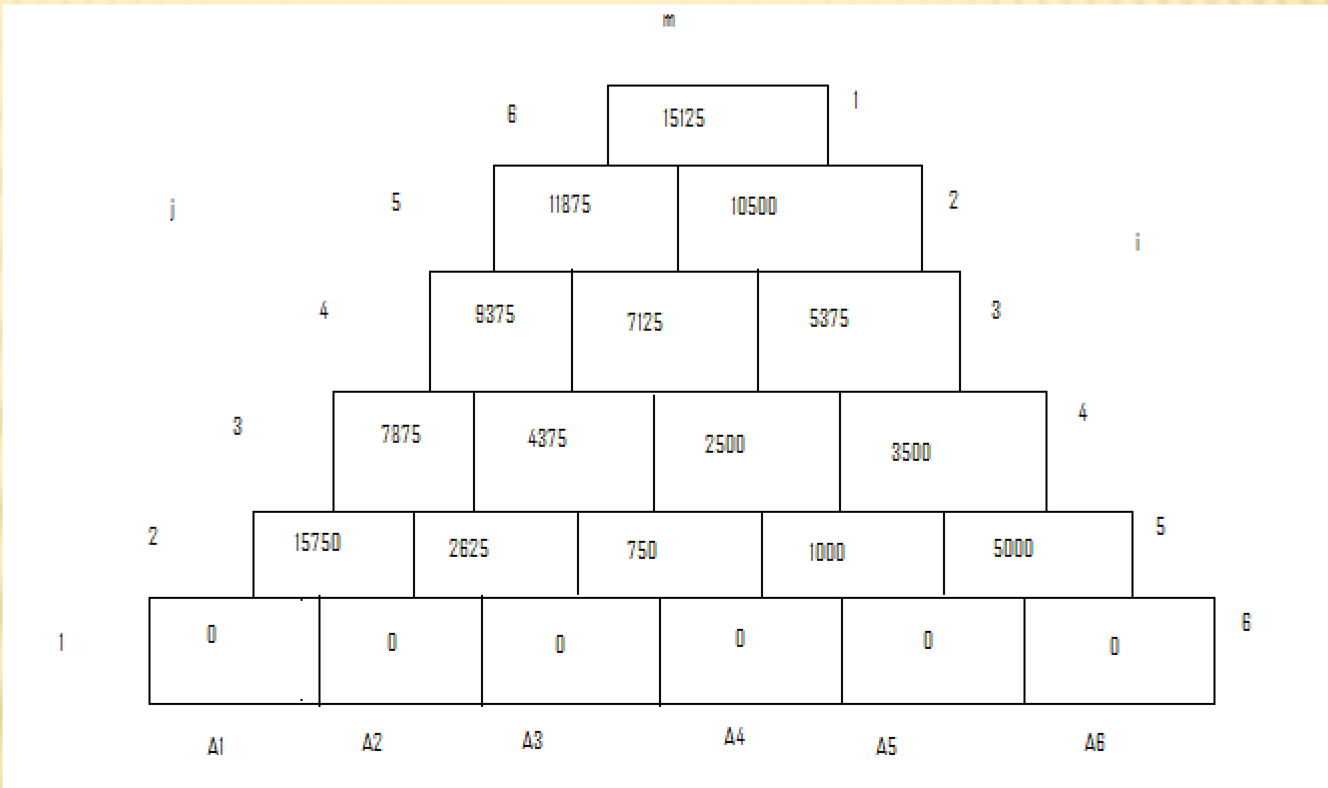
Constructing an optimal solution

- PRINT-OPTIMAL-PARENS(s, i, j)
- **if** $i = j$
-     **then** print "A"i
-     **else** print "("
-         PRINT-OPTIMAL-PARENS$(s, i, s[i, j])$
-         PRINT-OPTIMAL-PARENS$(s, s[i, j] + 1, j)$
-         print ")"

Example of matrix chain multiplication

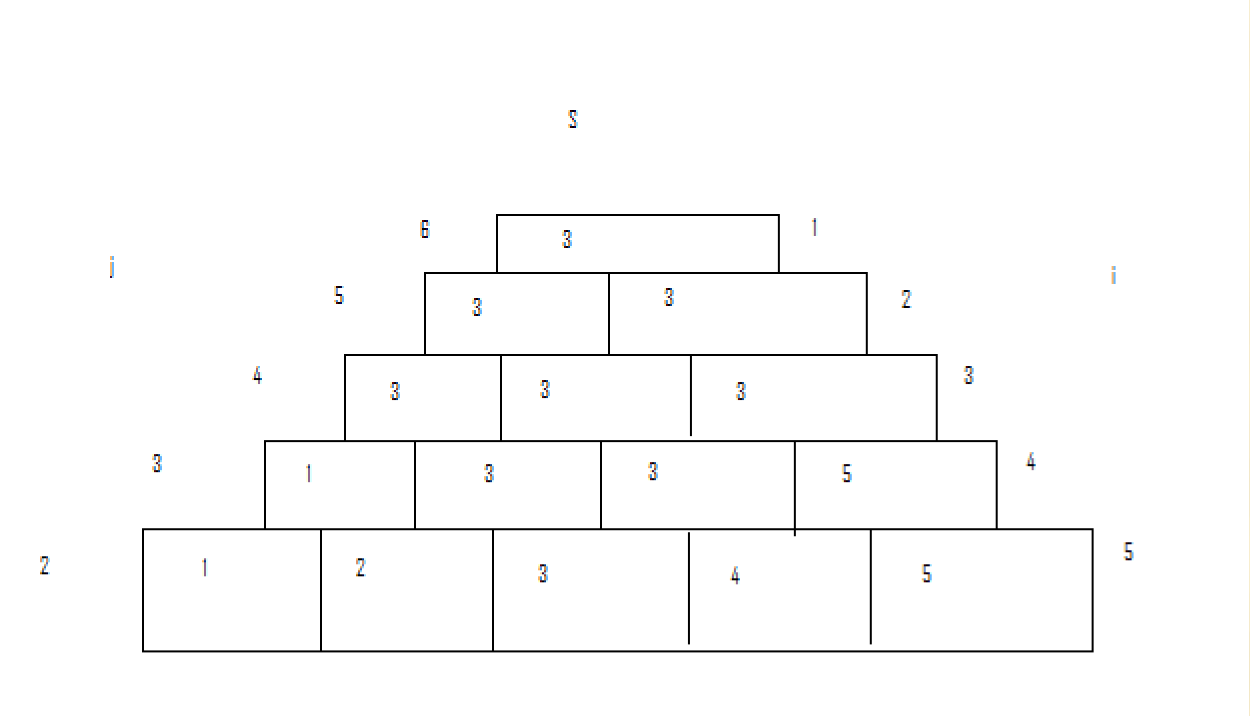
- matrix dimension
- $A1$ 30×35
- $A2$ 35×15
- $A3$ 15×5
- $A4$ 5×10
- $A5$ 10×20
- $A6$ 20×25

COST MATRIX



m
6
15125
1
j
5
11875
10500
2
i
4
9375
7125
5375
3
3
7875
4375
2500
3500
4
2
15750
2625
750
1000
5000
5
1
0
0
0
0
0
0
6
A1
A2
A3
A4
A5
A6

BEST MATRIX



MATRIX POLYNOMIAL

- A polynomial with matrix coefficients is called matrix polynomial. An nth order matrix polynomial is a variable t is given by

$P(A) = b_0 + b_1 A + b_2 A^2 + \ldots\ldots + b_n A^n$

Where $b_i \, \varepsilon \, R$, $A \, \varepsilon \, R^{nxn}$

- One of the method to evaluate matrix polynomial is using horner's rule

Where $S_0 = b_q I$

For $k = 1,2, \ldots.., q$

$\llcorner$ $S_k = AS_{k-1} + b_{q-k} I$, where $S_q = p(A)$

- This algorithm require $2n^2$ storage and $(q-1)n^3$ multiplications.
- On applying horner's rule to $p(A)$ which is polynomial in A^s , where $s = \sqrt{q}$, if $q = 6$
- $P(A) = (b_9I)(A^3)^3 + (b_8A^2 + b_7A + b_6I)(A^3)^2 + (b_5A^2 + b_4A + b_3I)(A^3) + (b_2A^2 + b_1A + b_0I)$

- Once A^2 and A^3 are computed , then only $2n^3$ multiplications are only required to evaluate p(A) as evidenced by the following “Horners regrouping”
- $P(A) = A^3[A^3[b_9A^3 + (b_8A^2 + b_7A + b_6I)] + (b_5A^2 + b_4A + b_3I)] + (b_2A^2 + b_1A + b_0I)$

- In general, if s is any integer satisfying $1 \leq s \leq q$ and $r = q/s$
- Then P(A) =
- Where $B_k = b_{sk+s-1} A^{s-1} + \ldots. + b_{sk+1}A + b_{sk}I$ $(k = 0,\ldots.,r-1)$
- And $B_r = b_q A^{q-sr} + \ldots. + b_{sr+1}A + b_{sr}I$

- Applying horner's rule to P(A) we get
- $F_0 = B_r$ for $k = 1,\ldots\ldots,r$
- $F_k = (A^s)F_{k-1} + B_{r-k}$ where $F_r = P(A)$

- algorithm 1:
- $Y_k = A^k \ (k = 0,1,\ldots,s)$
- $F_0 = b_q Y_{q-sr} + b_{q-1} Y_{q-sr-1} + \ldots + b_{sr+1} Y_1 + b_{sr} Y_0$
- For $k = 1,\ldots,r$
- $\quad F_k = Y_s F_{k-1} + b_{s(r-k)+s-1} Y_{s-1} + \ldots + b_{s(r-k)} Y_0$
  where $F_r = p(A)$

- Let e_j denotes jth column of identity matrix then we have

$P(A)e_j = (A^s)^k (B_k e_r)$

- Which can be computed as follows

$.f_0^{(j)} = B_r e_j$ for k =1 ....r

$f_k^{(j)} = (A^s)f^{(j)}_{k-1} + B_{-k}\, e_j$ where $f_r^{(j)} = p(A)e_j$, the jth column of P(A).

- $B_k e_j = b_{sk+s-1} (A_{s-1} e_j) + ... + b_{sk+1} (Ae_j) + b_{sk} e_j$ (k=0.....j)
- $B_r e_j = b_q\, A_{q-sr} e_j + b_{sr+1} Ae_j + b_{sr} e_j$

- **Algorithm 2:**
- $Y = A^s$
- For $j = 1,\ldots,n$
- $\llcorner\{$
- $Y_0^{(j)} = e_j$
- For $k = 1,\ldots,s-1$
- $\llcorner Y_k^{(j)} = A\,Y_{k-1}^{(j)}$
- $.f_0^{(j)} = b_q Y_{q-sr}^{(j)} + \ldots + b_{sr+1} Y_1^{(j)} + b_{sr} Y_0^{(j)}$
- For $k = 1,\ldots,r$
- $\llcorner f_k^{(j)} = Yf_{k-1}^{(j)} + b_{s(r-k)+s-1} Y_{s-1}^{(j)} + \ldots + b_{s(r-k)} Y_0^{(j)} \quad \}$

- Since arrays are required only for A , As and P(A) Algorithm 2 requires only $3n^2$ storage.
- And for s>1 , it require $w_2(s)n^3$ multiplications
- $w_2(s)$ = 2(s-1) + [q / s] -2 if s divides q

  2(s-1) + [q / s] -1 otherwise

References:

- Introduction to algorithms , 2nd edition ,Thomas H Cormen, Charles E Leiserson, Ronald L Rivest ,Clifford Stein.
- Algorithms , S Dasgupta , C. H. Papadimitriou, and U. V. Vazirani
- A note on the Evaluation of Matrix Polynomials by Charles Van Loan
- On the number of nonscalar multiplications necessary to evaluate polynomials., Mike Paterson and Larry J. Stockmeyer
- http://docs.linux.cz/programming/algorithms/Algorithms-Morris/mat_chain.html