# More dynamic programming

Dynamic programming solutions rely on the optimal substructure property. Usually the recursive solutions to these problems take exponential time with many redundant calculations because the subproblems are not independent.

One more dynamic programming example from Chapter 15 that we will cover:

- Matrix-Chain Product

1

## Matrix-Chain Product

If A is an $m \times n$ matrix and B is an $n \times p$ matrix, then

$A \cdot B = C$ is an $m \times p$ matrix

and the time needed to compute C is $O(mnp)$.

- there are $mp$ elements of C
- each element of C requires $n$ scalar multiplications and $n-1$ scalar additions

***Matrix-Chain Multiplication Problem***:

Given matrices $A_1, A_2, A_3, \ldots, A_n$, where the dimension of $A_i$ is $p_{i-1} \times p_i$, determine the *minimum* number of multiplications needed to compute the product $A_1 \cdot A_2 \cdot \ldots \cdot A_n$. This involves finding the optimal way to *parenthesize* the matrices.

For more than 2 matrices, there exists more than one order of multi-plication.

## Matrix-Chain Product

The running time of a brute-force solution (exhaustively checking all ways to parenthesize 2 matrices) is:

T(n) = 1 if n=1,

$$= \sum_{k=1}^{n-1} P(k)P(n-k) = \Omega(2^n) \quad \text{if } n \geq 2$$

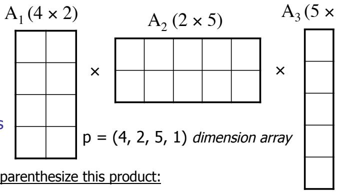
Here, P(k) is the way to parenthesize first k matrices and P(n – k) is the way to parenthesize the rest.

Hopefully, we can do better using Dynamic Programming.

## Matrix-Chain Product Example

$A_1$ (4 × 2) $A_2$ (2 × 5) $A_3$ (5 × 1)

The order of mult can make a difference in # of steps



p = (4, 2, 5, 1) *dimension array*

How do we find the minimum over all possibilities?

Two ways to parenthesize this product:

$(A_1 \cdot A_2) \cdot A_3$
$M_1 = A_1 \cdot A_2$: requires $4 \cdot 2 \cdot 5 = 40$ multiplications, $M_1$ is $4 \times 5$ matrix
$M_2 = M_1 \cdot A_3$: requires $4 \cdot 5 \cdot 1 = 20$ multiplications, $M_2$ is $4 \times 1$ matrix
–> total multiplications = 40 + 20 = 60

$A_1 \cdot (A_2 \cdot A_3)$
$M_1 = A_2 \cdot A_3$: requires $2 \cdot 5 \cdot 1 = 10$ multiplications, $M_1$ is $2 \times 1$ matrix
$M_2 = A_1 \cdot M_1$: requires $4 \cdot 2 \cdot 1 = 8$ multiplications, $M_2$ is $4 \times 1$ matrix
–> total multiplications = 10 + 8 = 18

## Matrix-Chain Product

The optimal substructure of this problem can be given with the following argument:

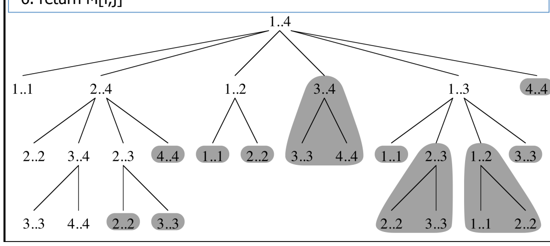
Suppose an optimal way to parenthesize $A_i A_{i+1} \ldots A_j$ splits the product between $A_k$ and $A_{k+1}$. Then the way the prefix subchain $A_i A_{i+1} \ldots A_k$ is parenthesized must be optimal. Why?

If there were a less costly way to parenthesize $A_i A_{i+1} \ldots A_k$, substituting that solution as the way to parenthesize $A_i A_{i+1} \ldots A_j$ gives a solution with lower cost, contradicting the assumption that the way the original group of matrices was parenthesized was optimal.

Therefore, the structure of the subproblem solutions must be optimal.

## Matrix-Chain Product – Recursive Solution

```
RMP(p, i, j)        // p is array of dimensions, initially i = 1, j = # of matrices
1. if i = j  return 0 // nothing need be done with a single matrix
2. M[i,j] = ∞
3. for k = i to j-1
4.      q = RMP(p, i, k) + RMP(p, k+1, j) + p_{i-1}p_kp_j
5.      if q < M[i,j] then M[i,j] = q
6. return M[i,j]
```



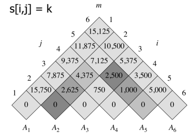
## Matrix-Chain Product – Bottom-up solution

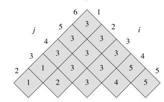
```
Matrix-Chain-Order (p)  // p is array of dimensions
  n = p.length – 1
  let M[1…n, 1…n] and s[1…n-1,2…n] be new tables
  for i = 1 to n
      M[i,i] = 0                 /** fill in main diagonal of M with 0s **/
  for d = 2 to n                 /** d is chain length**/
      for i = 1 to n – d + 1
          j = i + d - 1
          M[ i, j ] = ∞
          for k = i to j-1
              /** q is cost in scalar multiplications **/
              q = M[ i, k]+ M[k+1,j] + p_{i-1}p_k p_j)
              if q < M[ i, j ]
                  M[ i, j ] = q
                  s[ i, j ] = k
  return M and s
```

## Matrix-Chain Product – Bottom-up solution

```
Matrix-Chain-Order (p)          /** p is array of dimensions **/
  n = p.length – 1
  let M[1…n, 1…n] and s[1…n-1,2…n] be new tables
  for i = 1 to n
      M[i,i] = 0                /** fill in main diagonal with 0s **/
  for d = 2 to n
      for i = 1 to n – d + 1
          j = i + d - 1
          M[i,j] = ∞
          for k = i to j-1
              q = M[i,k]+ M[k+1,j] + p_{i-1}p_k p_j)
              if q < M[i,j]
                  M[i,j] = q
                  s[i,j] = k
  return M and s
```

p = [30, 35, 15, 5, 10, 20, 25]

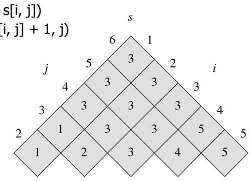



## Matrix-Chain Product – Bottom-up solution

Input: s array, dimensions of M matrix
Output: side-effect printing of optimal parenthesization

```
Print-Optimal-Parens(s, i, j)  // initially, i = 1 and j = n
  if i == j
      print "A"_i
  else
      print "("
      Print-Optimal-Parens(s, i, s[i, j])
      Print-Optimal-Parens(s, s[i, j] + 1, j)
      print ")"
```




"((A1 (A2 A3))((A4 A5) A6))"

## Matrix-Chain Product – Bottom-Up Solution

Complexity:
- $O(n^3)$ time because of the nested for loops with each of d, i, and k taking on at most n-1 values.
- $O(n^2)$ space for two n x n matrices M and s

## Longest Common Subsequence Problem (§ 15.4)

***Problem:*** Given X = < $x_1$, $x_2$, ..., $x_m$ > and Y = <$y_1$, $y_2$,..., $y_n$), find a longest common subsequence (LCS) of X and Y.

Example:

X = 〈 A, B, C, B, D, A, B 〉

Y = 〈 B, D, C, A, B, A 〉

$LCS_{XY}$ = 〈 B, C, B, A 〉 or $LCS_{XY}$ = 〈 B, D, A, B 〉)

Brute-Force solution:
1. Enumerate all subsequences of X and check to see if they appear in the correct order in Y (chars in sequences are not necessarily consecutive).
2. Each subsequence of X corresponds to a subset of the indices {1,2,...,m} of the elements of X, so there are $2^m$ subsequences of X to be enumerated.
3. Clearly, this is not a good approach...time to try dynamic programming!

## Recursive Solution to LCS Problem

The recursive LCS Formulation
- Let C[i,j] = length of the LCS of $X_i$ and $Y_j$, where $X_i = \langle x_1, x_2,\ldots, x_i \rangle$ and $Y_j = \langle y_1, y_2,\ldots, y_j \rangle$
- Our goal: C[m,n] (consider entire X and Y)
- Basis: C[0,j] = 0 and C[i,0] = 0
- C[i,j] is calculated as shown below (two cases):

Case 1: $x_i = y_j$ (i, j > 0)
In this case, we can increase the size of the LCS of $X_{i-1}$ and $Y_{j-1}$ by one by appending $x_i = y_j$ to the LCS of $X_{i-1}$ and $Y_{j-1}$, i.e.,
C[i, j] = C[i-1, j-1] + 1

Case 2: $x_i \neq y_j$ (i, j > 0)
In this case, we take the LCS to be the longer of the LCS of $X_{i-1}$ and $Y_j$, and the LCS of $X_i$ and $Y_{j-1}$, i.e.,
C[i, j] = max(C[i, j-1], C[i-1, j])

## Bottom-Up DP Solution to LCS Problem

To compute C[i, j], we need the solutions to:
C[i-1, j-1] (when $x_i = y_j$)
C[i-1, j] and C[i, j-1] (when $x_i \neq y_j$ )

We need an m by n matrix to store results.

**LCS(X, Y)**
**1.** m = length[X]
**2.** n = length[Y]
**3.** **for** i = 0 **to** m  C[i, 0] = 0  // 0 in first col of each row
**4.** **for** j = 0 **to** n  C[0, j] = 0 // 0 in first row of each col
**5.** **for** i = 1 **to** m **do**
**6.** **for** j = 1 **to** n **do**  // process row by row
**7.** **if** $x_i$ == $y_j$  C[i, j] = C[i-1, j-1] + 1
**8.** **else**  C[i, j] = max (C[i, j-1], C[i-1, j])
**9.** **return** C[m, n]

## Bottom-Up LCS DP

Running time = O(mn) (constant time for each entry in C)

This algorithm finds the value of the LCS, but how can we keep track of the characters in the LCS?

We need to keep track of which neighboring table entry gave the optimal solution to a sub-problem (break ties arbitrarily).
if $x_i = y_j$ the answer came from the upper left (diagonal),
if $x_i \neq y_j$ the answer came from above or to the left, otherwise, whichever value is larger (if equal, default to above).

## Bottom-Up DP Solution to LCS Problem

As usual, we need a procedure to interpret the results of our DP solution.

**Idea**: Save a pointer to find the path representing the longest common subsequence. Use a 2-dimensional array B to store the pointers (initially this array will be all NIL).

**LCS(X, Y)**
**1.** m = length[X]
**2.** n = length[Y]
**3.** **for** i = 0 **to** m  C[i, 0] = 0
**4.** **for** j = 0 **to** n  C[0, j] = 0
**5.** **for** i = 1 **to** m
**6.** **for** j = 1 **to** n
**7.** **if** $x_i$ == $y_j$
**8.** C[i, j] = C[i-1, j-1] + 1
**9.** B[i, j] = Up&Left
**10.** **else** **if** C[i - 1, j] >= C[i, j – 1]
**11.** C[i, j] = C[i - 1, j]
**12.** B[i, j] = Up
**13.** **else** C[i , j] = C[i, j - 1]
**14.** B[i, j] = Left

## Bottom-Up LCS DP

| i ⇓ \ j ⇒ | 0 | 1 b | 2 a | 3 b |
|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 |
| 1 a | 0 | ↑ 0 | ↖ 1 | ← 1 |
| 2 b | 0 | ↖ 1 | ↑ 1 | ↖ 2 |
| 3 b | 0 | ↖ 1 | ↑ 1 | ↖ 2 |
| 4 a | 0 | ↑ 1 | ↖ 2 | ↑ 2 |

X = abba
Y = bab

## Constructing the LCS

**Print-LCS (B, X, i, j)**
1. **if** i == 0 or j == 0 **then** return
2. **if** B[i,j] == Up&Left
3. Print-LCS(B,X,i-1,j-1)
4. print $x_i$
5. **else if** B[i,j] = Up
6. Print-LCS(B,X,i-1,j)
7. **else** Print-LCS(B,X,i,j-1)

Initial call is Print-LCS(B,X,len(X),len(Y)), where B is the arrow table

## Complexity of LCS Algorithm

The running time of the LCS algorithm is O(mn), since each table entry takes O(1) time to compute.

The running time of the Print-LCS algorithm is O(m + n), since one of m or n is decremented in each stage of the recursion.