Design and Analysis of Algorithm

Mahesh Shirole,

VJTI, Mumbai-19.

Dynamic Programming (DP)

- DP is a powerful algorithm design technique to solve ***combinatorial optimization problems***
- Optimization problems can have ***many possible solutions***
- Each solution has a value, and we wish to find a solution with the optimal (***minimum or maximum***) value
- We call such a solution ***an optimal solution to the problem***, as opposed to the optimum solution, since there may be several solutions that achieve the optimal value
- When developing a dynamic-programming algorithm, we follow a sequence of four steps:
    - Characterize the structure of an optimal solution
    - Recursively define the value of an optimal solution
    - Compute the value of an optimal solution, typically in a bottom-up fashion
    - Construct an optimal solution from computed information

Dynamic Programming (DP)

Richard Bellman first used the term ***dynamic programming*** to describe a type of problem in which the most efficient solution depended on choices that may change with time instead of being predetermined

- Dynamic programming is similar to divide-and-conquer in that an instance of a problem is divided into smaller instances
- Divide-and-conquer algorithms ***partition the problem into disjoint sub-problems***, solve the sub-problems recursively, and then combine their solutions to solve the original problem
- In contrast, dynamic programming applies when the ***sub-problems overlap***—that is, when sub-problems share sub-sub-problems
- A dynamic-programming algorithm ***solves each sub-sub-problem just once*** and then ***saves its answer in a table***, thereby avoiding the work of re-computing the answer every time it solves each sub-sub-problem

Dynamic Programming (DP)

- There are usually two equivalent ways to implement a dynamic-programming approach
  - **Top-down with memoization**: In this approach, we write the procedure recursively in a natural manner, but modified to save the result of each sub-problem (usually in an array or hash table). The procedure now first checks to see whether it has previously solved this sub-problem. If so, it returns the saved value, saving further computation at this level; if not, the procedure computes the value in the usual manner
  - **Bottom-up method**: In this approach, we sort the sub-problems by size; and solve them in size order, smallest first. We solve each sub-problem only once, and when we first see it, we have already solved all of its prerequisite sub-problems
- These two approaches yield algorithms with the **same asymptotic running time**
- The bottom-up approach often has *much better constant factors*, since it has less overhead for procedure calls

Fibonacci sequence

- Fibonacci sequence: f1 =1, f2 =1, f3 =2, f4 =3, f5 =5, f6 =8, f7 =13,... .
- Each number in the sequence 1, 2, 3, 5, 8, 13,... is the sum of the two preceding numbers

$$f(n) = \begin{cases} 1 & \text{if } n = 1 \text{ or } n = 2, \\ f(n-1) + f(n-2) & \text{if } n \geq 3. \end{cases}$$

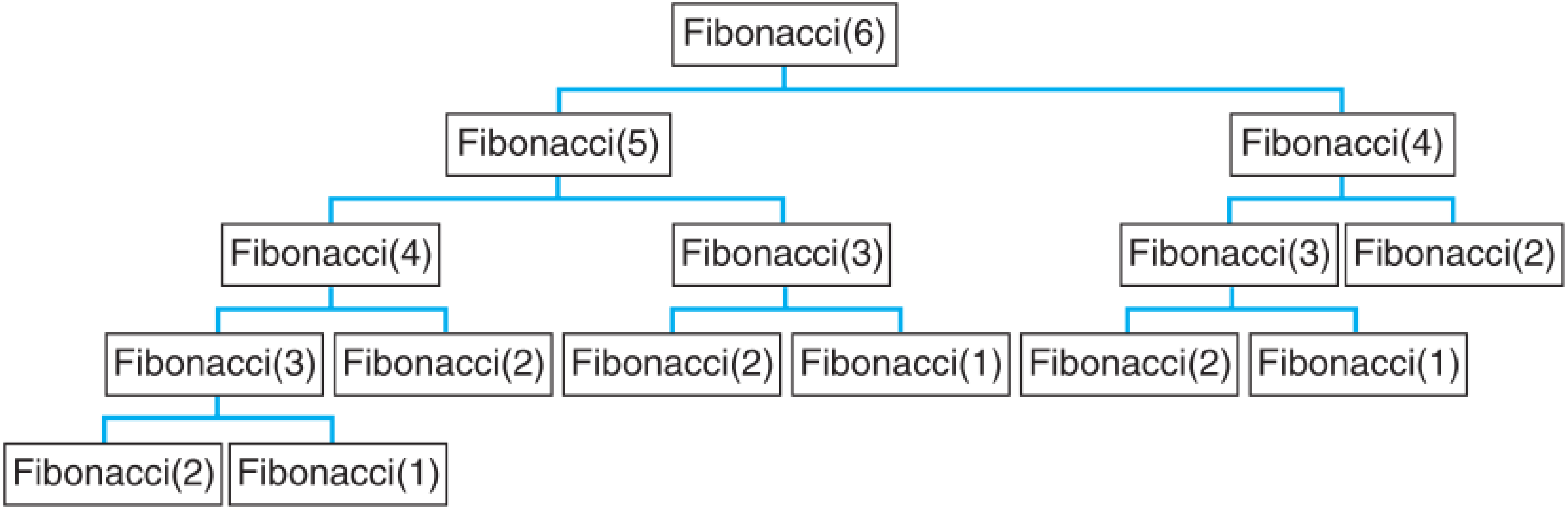
1. **procedure** $f(n)$
2. **if** $(n = 1)$ **or** $(n = 2)$ **then return** 1
3. **else return** $f(n-1) + f(n-2)$

Is this algorithm is efficient?

Fibonacci sequence

$$\begin{aligned} f(n) &= f(n-1) + f(n-2) \\ &= 2f(n-2) + f(n-3) \\ &= 3f(n-3) + 2f(n-4) \\ &= 5f(n-4) + 3f(n-5). \end{aligned}$$

- There are many duplicate recursive calls to the procedure



Fibonacci sequence

Top-down Approach (memoization)

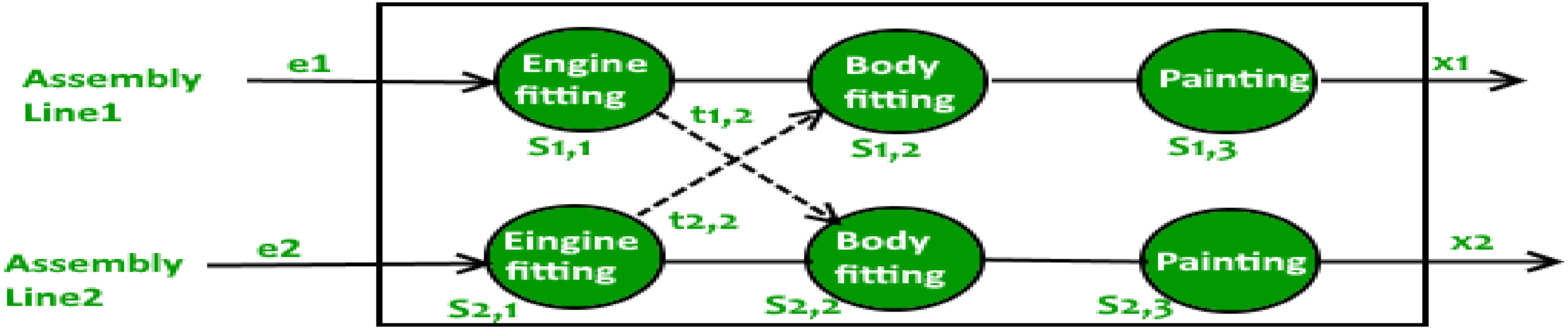
```
procedure f(n)
if n in Memo return Memo[n]
  Memo[1]=Memo[2]=1
  if (n =1) or (n =2) then return 1
  else
    sub-prob-sol= f(n – 1) + f(n – 2)
    Memo[n]=sub-prob-sol
 return sub-prob-sol
```

Bottom-up Approach (iterative)

```
procedure f(n)
if n<=2
   Table[1]=Table[2]=1 return 1
for (i=3; i=<n; i++)
  Table[i]=Table[i-1] + Table[i-2]
return Table[n]
```

Assembly Line Scheduling in Manufacturing Sector

- **Problem Statement**: A manufacturing company has two assembly lines, each with ***n*** stations. A station is denoted by $S_{i,j}$ where ***i*** denotes the assembly line the station is on and ***j*** denotes the number of the station. The time taken per station is denoted by $a_{i,j}$. Each station is dedicated to do some sort of work in the manufacturing process. So, a chassis must pass through each of the ***n*** stations in order before exiting the company. The parallel stations of the two assembly lines perform the same task. After it passes through station $S_{i,j}$, it will continue to station $S_{i,j+1}$ unless it decides to transfer to the other line. Continuing on the same line incurs no extra cost, but transferring from line ***i*** at station $j - 1$ to station ***j*** on the other line takes time $t_{i,j}$. Each assembly line takes an entry time e_i and exit time x_i . Give an algorithm for computing the minimum time from start to exit.
- **Objective**: To find the optimal scheduling i.e., the fastest way from start to exit.
- **Note**: let $f_i\,[j]$ denotes the fastest way from start to station $S_{i,j}$.



Optimal Substructure

- An optimal solution to a problem is determined using optimal solutions to subproblems (in turn, sub subproblems and so on). The immediate question is, how to break the problem in to smaller sub problems ?
- The answer is: if we know the minimum time taken by the chassis to leave station $S_{i,j-1}$, then the minimum time taken to leave station $S_{i,j}$ can be calculated quickly by combining $a_{i,j}$ and $t_{i,j}$

Final Solution: $f^{OPT} = \min\{f_1[n] + x_1, f_2[n] + x_2\}$.

Base Cases: $f_1[1] = e_1 + a_{1,1}$ and $f_2[1] = e_2 + a_{2,1}$.

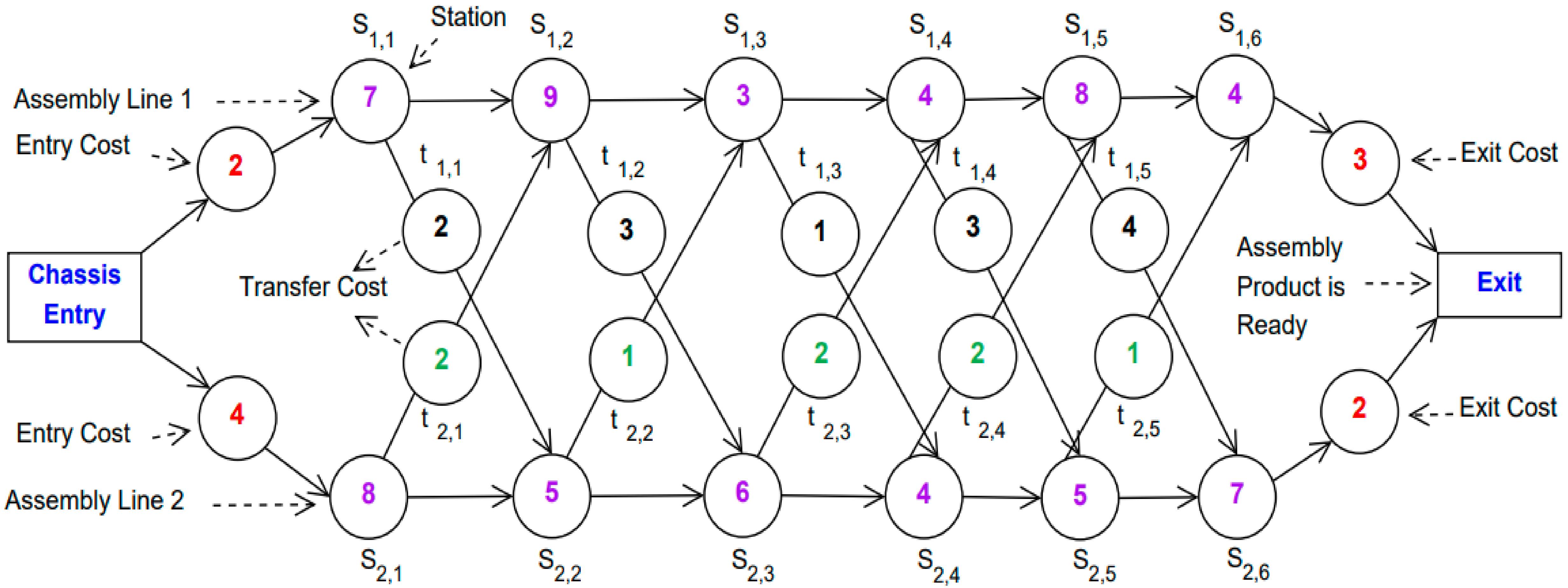
Recursive Solution

- The chassis at station $S_{1,j}$ can come either from station $S_{1,j-1}$ or station $S_{2,j-1}$ (Since, the tasks done by $S_{1,j}$ and $S_{2,j}$ are same). But if the chassis comes from $S_{2,j-1}$, it additionally incurs the transfer cost to change the assembly line. Thus, the recursion to reach the station j in assembly line i are as follows:

$$f_1[j] = \begin{cases} e_1 + a_{1,1} & \text{if } j = 1 \\ \min\{f_1[j-1] + a_{1,j},\, f_2[j-1] + t_{2,j-1} + a_{1,j}\} & \text{if } j \geq 2 \end{cases}$$

$$f_2[j] = \begin{cases} e_2 + a_{2,1} & \text{if } j = 1 \\ \min\{f_2[j-1] + a_{2,j},\, f_1[j-1] + t_{1,j-1} + a_{2,j}\} & \text{if } j \geq 2 \end{cases}$$

Assembly Line Scheduling Problem with Six Stations



The algorithm: **Fastest-way**(a, t, e, x, n) Source: CLRS

---

where a denotes the assembly costs, t denotes the transfer costs, e denotes the entry costs, x denotes the exit costs and n denotes the number of assembly stages.

$f_1[1] = e_1 + a_{1,1}$
$f_2[1] = e_2 + a_{2,1}$
    for $j = 2$ to n
      if $((f_1[j-1] + a_{1,j}) \leq (f_2[j-1] + t_{2,j-1} + a_{1,j}))$ then
        $f_1[j] = f_1[j-1] + a_{1,j}$ and $l_1[j] = 1$ /* l_p denotes the line p */
      else
        $f_1[j] = f_2[j-1] + t_{2,j-1} + a_{1,j}$ and $l_1[j] = 2$
      if $((f_2[j-1] + a_{2,j}) \leq (f_1[j-1] + t_{1,j-1} + a_{2,j}))$ then
        $f_2[j] = f_2[j-1] + a_{2,j}$ and $l_2[j] = 2$
      else
        $f_2[j] = f_1[j-1] + t_{1,j-1} + a_{2,j}$ and $l_2[j] = 1$
    end for
    if $(f_1[n] + x_1 \leq f_2[n] + x_2)$ then
      $f^{OPT} = f_1[n] + x_1$ and $l^{OPT} = 1$
    else
      $f^{OPT} = f_2[n] + x_2$ and $l^{OPT} = 2$

Trace of the algorithm

Iteration 1: $j = 1$ **and** $j = 2$
$f_1[1] = e_1 + a_{1,1} = 2 + 7 = 9$ and start $= S_{1,1}$
$f_2[1] = e_2 + a_{2,1} = 4 + 8 = 12$ and start $= S_{2,1}$
$f_1[2] = \min\{f_1[1] + a_{1,2}, f_2[1] + t_{2,1} + a_{1,2}\} = \min\{9 + 9, 12 + 2 + 9\} = 18,\ l_1[2] = 1$
$f_2[2] = \min\{f_2[1] + a_{2,2}, f_1[1] + t_{1,1} + a_{2,2}\} = \min\{12 + 5, 9 + 2 + 5\} = 16,\ l_2[2] = 1$

Iteration 2: $j = 3$

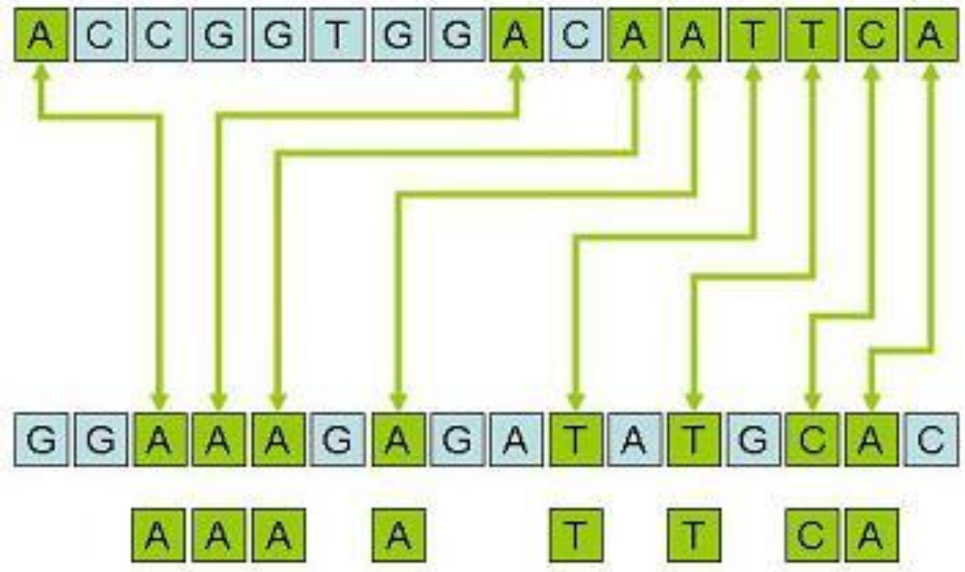
$f_1[3] = \min\{f_1[2] + a_{1,3}, f_2[2] + t_{2,2} + a_{1,3}\} = \min\{18 + 3, 16 + 1 + 3\} = 20,\ l_1[3] = 2$
$f_2[3] = \min\{f_2[2] + a_{2,3}, f_1[2] + t_{1,2} + a_{2,3}\} = \min\{16 + 6, 18 + 3 + 6\} = 22,\ l_2[3] = 2$

Iteration 3: $j = 4$
$f_1[4] = \min\{f_1[3] + a_{1,4}, f_2[3] + t_{2,3} + a_{1,4}\} = \min\{20 + 4, 22 + 2 + 4\} = 24,\ l_1[4] = 1$
$f_2[4] = \min\{f_2[3] + a_{2,4}, f_1[3] + t_{1,3} + a_{2,4}\} = \min\{22 + 4, 20 + 1 + 4\} = 25,\ l_2[4] = 1$

Iteration 4: $j = 5$
$f_1[5] = \min\{f_1[4] + a_{1,5}, f_2[4] + t_{2,4} + a_{1,5}\} = \min\{24 + 8, 25 + 2 + 8\} = 32,\ l_1[5] = 1$
$f_2[5] = \min\{f_2[4] + a_{2,5}, f_1[4] + t_{1,4} + a_{2,5}\} = \min\{25 + 5, 24 + 3 + 5\} = 30,\ l_2[5] = 2$

Iteration 5: $j = 6$
$f_1[6] = \min\{f_1[5] + a_{1,6}, f_2[5] + t_{2,5} + a_{1,6}\} = \min\{32 + 4, 30 + 1 + 4\} = 35,\ l_1[6] = 2$
$f_2[6] = \min\{f_2[5] + a_{2,6}, f_1[5] + t_{1,5} + a_{2,6}\} = \min\{30 + 7, 32 + 4 + 7\} = 37,\ l_2[6] = 2$
Optimum Schedule: $\min\{f_1[6] + x_1, f_2[6] + x_2\} = \{35 + 3, 37 + 2\} = 38$; $l^{OPT} = 1$.

Longest Common Subsequence



- Biological applications often need to compare the DNA of two (or more) different organisms
- A strand of DNA consists of a string of molecules called ***bases***, where the possible bases are adenine (A), guanine (G), cytosine (C), and thymine (T)
- We express a strand of DNA as a string over the finite set {A, C, G, T}
- One reason to compare two strands of DNA is to determine how "***similar***" the two strands are, as some measure of how closely related the two organisms are

Longest Common Subsequence

- For example, the DNA of one organism $\boldsymbol{S_1}$ and another organism $\boldsymbol{S_2}$
  - $\boldsymbol{S_1}$ = ACCGGTCGAGTGCGCGGAAGCCGGCCGAA
  - $\boldsymbol{S_2}$ = GTCGTTCGGAATGCCGTTGCTCTGTAAA
- We can define similarity in many different ways
  1. Two DNA strands are similar if one is a **substring** of the other
  2. Two strands are similar if the *number of changes needed to turn one into the other* is small
  3. To measure the similarity of strands $\boldsymbol{S_1}$ and $\boldsymbol{S_2}$ is by finding a third strand $\boldsymbol{S_3}$ in which the bases in $\boldsymbol{S_3}$ appear in each of $\boldsymbol{S_1}$ and $\boldsymbol{S_2}$. These ***bases must appear in the same order***, but not ***necessarily consecutively***

- In our example, the longest strand
  - $\boldsymbol{S_3}$ = GTCGTCGGAAGCCGGCCGAA

Longest Common Subsequence

- Formally, given a sequence $X = <x_1, x_2, ..., x_m>$, another sequence $Z = <z_1, z_2 ... , z_k>$ is a subsequence of X if there exists a strictly increasing sequence $<i_1, i_2, ..., i_k>$ of indices of X such that for all j =1, 2, ... , k, we have $xi_j = z_j$
- For example Z= <B, C, D, B> is a subsequence of X =<A, B, C, B, D, A, B> with corresponding index sequence <2, 3, 5, 7>
- Given two sequences X and Y, we say that a sequence Z is a ***common subsequence*** of X and Y, if Z is a subsequence of both X and Y
- For example, if X=<A, B, C, B, D, A, B> and Y=<B, D, C, A, B, A>, the sequence <B, C, A> is a common subsequence of both X and Y
- The sequence <B, C, A> is not a longest common subsequence (LCS) of X and Y, however, since it has length 3
- The sequence <B, C, B, A>, which is also common to both X and Y, has length 4
- The sequence <B, C, B, A> is an LCS of X and Y, as is the sequence <B, D, A, B>, since X and Y have no common subsequence of length 5 or greater

Longest Common Subsequence

- In the longest-common-subsequence problem, we are given two sequences $X = \langle x_1, x_2, \ldots, x_m \rangle$ and $Y = \langle y_1, y_2, \ldots, y_n \rangle$ and wish to find a maximum length common subsequence of X and Y
- In a brute-force approach to solving the LCS problem, we would enumerate all subsequences of X and check each subsequence to see whether it is also a subsequence of Y, keeping track of the longest subsequence we find
- Each subsequence of X corresponds to a subset of the indices {1, 2, ..., m} of X
- Because X has **2^m** subsequences, this approach requires ***exponential time***, making it impractical for long sequences

Step 1: Characterizing a Longest Common Subsequence (LCS)

- The LCS problem has an optimal-substructure property
- As we shall see, the natural classes of subproblems correspond to pairs of "prefixes" of the two input sequences
- To be precise, given a sequence ***$X = <x_1, x_2, \ldots, x_m>$***,we define the *$i^{th}$* prefix of X, for i = 0, 1, . . ., m, as ***$X_i = <x_1, x_2, \ldots, x_i>$***
- For example, if X = <A, B, C, B, D, A, B>, then X_4 = <A, B, C, B> and X_0 is the empty sequence

Theorem 15.1 (Optimal substructure of an LCS)

Let $X = \langle x_1, x_2, \ldots, x_m \rangle$ and $Y = \langle y_1, y_2, \ldots, y_n \rangle$ be sequences, and let $Z = \langle z_1, z_2, \ldots, z_k \rangle$ be any LCS of X and Y.

1. If $x_m = y_n$, then $z_k = x_m = y_n$ and Z_{k-1} is an LCS of X_{m-1} and Y_{n-1}.
2. If $x_m \neq y_n$, then $z_k \neq x_m$ implies that Z is an LCS of X_{m-1} and Y.
3. If $x_m \neq y_n$, then $z_k \neq y_n$ implies that Z is an LCS of X and Y_{n-1}.

Step 2: A Recursive Solution

- Theorem 15.1 implies that we should examine either one or two subproblems when finding an LCS of $X =<x_1,x_2, ..., x_m>$ and $Y = <y_1, y_2, ... , y_n>$
- If $x_m = y_n$, we must find an LCS of X_{m-1} and Y_{n-1}
  - Appending $x_m = y_n$ to this LCS yields an LCS of X and Y
- If $x_m != y_n$, then we must solve two sub-problems:
  - finding an LCS of X_{m-1} and Y and
  - finding an LCS of X and Y_{n-1}
  - Whichever of these two LCSs is longer is an LCS of X and Y
- There is overlapping sub-problems property in the LCS problem

Step 2: A Recursive Solution

- Let us define c[i, j] to be the length of an LCS of the sequences $\mathbf{X}_i$ and $\mathbf{Y}_j$
- If either i = 0 or j = 0, one of the sequences has length 0, and so the LCS has length 0
- The optimal substructure of the LCS problem gives the recursive formula

$$c[i,j] = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0 , \\ c[i-1, j-1] + 1 & \text{if } i, j > 0 \text{ and } x_i = y_j , \\ \max(c[i, j-1], c[i-1, j]) & \text{if } i, j > 0 \text{ and } x_i \neq y_j . \end{cases}$$

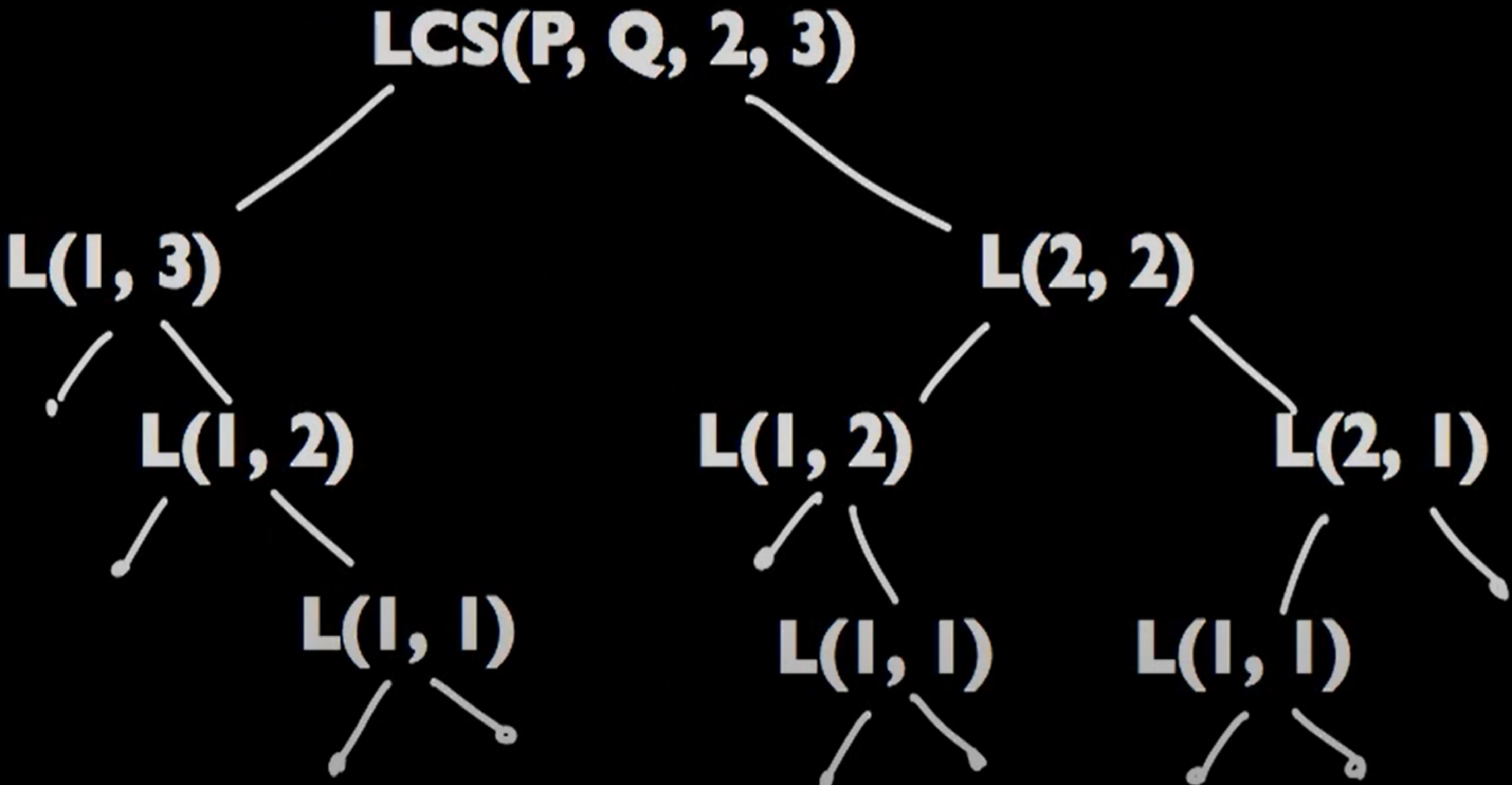
Recursive Solution (Code)

```
def LCS(P, Q, n, m)
    if n == 0 or m == 0: // base case
        result = 0
    else if P[n-1] == Q[m-1]:
        result = 1 + LCS(P, Q, n-1, m-1)
    else if P[n-1] != Q[n-1]: // just for clarity
        tmp1 = LCS(P, Q, n-1, m)
        tmp2 = LCS(P, Q, n, m-1)
        result = max{ tmp1, tmp2 }
    return result
```

Analysis of Recursive Solution

P = “AA”

Q = “BBB”



Memorize Intermediate Results

```
// Initialize arr[n][m] to undefined
def LCS(P, Q, n, m)
    if arr[n][m] != undefined: return arr[n][m]
    if n == 0 or m == 0:
        result = 0
    else if P[n-1] == Q[m-1]:
        result =  1 + LCS(P, Q, n-1, m-1)
    else if P[n-1] != Q[n-1]: // just for clarity
        tmp1 = LCS(P, Q, n-1, m)
        tmp2 = LCS(P, Q, n, m-1)
        result = max{ tmp1, tmp2 }
    arr[n][m] = result
    return result
```

Step 3: Computing the length of an LCS

- Dynamic programming to compute the solutions bottom up
- Procedure LCS-LENGTH takes two sequences X =<x_1,x_2, ..., x_m> and Y = <y_1, y_2, ... , y_n> as inputs
- It stores the c[i, j] values in a table c[0 .. m, 0 .. n], and it computes the entries in **row-major** order
- The procedure also maintains the table b[0 .. m, 0 .. n] to help us construct an **optimal solution**

```
LCS-LENGTH(X, Y)
m = X.length
n = Y.length
let b[1..m, 1..n] and c[0..m, 0..n] be new tables
for i = 1 to m
    c[i, 0] = 0
for j = 0 to n
    c[0, j] = 0
for i = 1 to m
    for j = 1 to n
        if x_i == y_j
            c[i, j] = c[i − 1, j − 1] + 1
            b[i, j] = "↖"
        elseif c[i − 1, j] ≥ c[i, j − 1]
            c[i, j] = c[i − 1, j]
            b[i, j] = "↑"
        else c[i, j] = c[i, j − 1]
            b[i, j] = "←"
return c and b
```

Step 3: Computing the length of an LCS

- Figure shows the tables produced by LCS-LENGTH on the sequences X =<A, B, C, B, D, A, B> and Y =<B, D, C, A, B, A>
- To quickly construct an LCS simply begin at b[m, n] and trace through the table by following the arrows.
- Whenever we encounter a "↖"in entry b[i, j], it implies that $\mathbf{x_i} = \mathbf{y_j}$ is an element of the LCS
- With this method, we encounter the elements of this LCS in reverse order
- The running time of the procedure is θ(mn), since each table entry takes θ(1) time to compute

	j	0	1	2	3	4	5	6
i		y_j	B	D	C	A	B	A
0	x_i	0	0	0	0	0	0	0
1	A	0	↑ 0	↑ 0	↑ 0	↖ 1	← 1	↖ 1
2	B	0	↖ 1	← 1	← 1	↑ 1	↖ 2	← 2
3	C	0	↑ 1	↑ 1	↖ 2	← 2	↑ 2	↑ 2
4	B	0	↖ 1	↑ 1	↑ 2	↑ 2	↖ 3	← 3
5	D	0	↑ 1	↖ 2	↑ 2	↑ 2	↑ 3	↑ 3
6	A	0	↑ 1	↑ 2	↑ 2	↖ 3	↑ 3	↖ 4
7	B	0	↖ 1	↑ 2	↑ 2	↑ 3	↖ 4	↑ 4

Step 4: Constructing an LCS

- The following recursive procedure prints out an LCS of X and Y in the proper, forward order.
- The initial call is PRINT-LCS(b,X, X.length, Y.length)

```
PRINT-LCS(b, X, i, j)
if i == 0 or j == 0
    return
if b[i, j] == "↖"
    PRINT-LCS(b, X, i − 1, j − 1)
    print x_i
elseif b[i, j] == "↑"
    PRINT-LCS(b, X, i − 1, j)
else PRINT-LCS(b, X, i, j − 1)
```

i \ j		0	1	2	3	4	5	6
	y_j		B	D	C	A	B	A
0	x_i	0	0	0	0	0	0	0
1	A	0	↑ 0	↑ 0	↑ 0	↖ 1	← 1	↖ 1
2	B	0	↖ 1	← 1	← 1	↑ 1	↖ 2	← 2
3	C	0	↑ 1	↑ 1	↖ 2	← 2	↑ 2	↑ 2
4	B	0	↖ 1	↑ 1	↑ 2	↑ 2	↖ 3	← 3
5	D	0	↑ 1	↖ 2	↑ 2	↑ 2	↑ 3	↑ 3
6	A	0	↑ 1	↑ 2	↑ 2	↖ 3	↑ 3	↖ 4
7	B	0	↖ 1	↑ 2	↑ 2	↑ 3	↖ 4	↑ 4

Matrix Chain Multiplication

- Suppose we want to compute the product M1M2M3 of three matrices M1,M2,and M3 of dimensions 2×10, 10×2, and 2×10, respectively, using the standard method of matrix multiplication
- If we multiply M1 by the result of multiplying M2 and M3,then the number of scalar multiplications becomes $10 \times 2 \times 10 + 2 \times 10 \times 10 = 400$
- If we multiply M1 and M2 and then multiply the result by M3, the number of scalar multiplications will be $2 \times 10 \times 2 + 2 \times 2 \times 10 = 80$
- Thus, carrying out the multiplication M1(M2M3) costs five times the multiplication (M1M2)M3
- In general, the cost of multiplying a chain of n matrices M1M2 ...Mn depends on the order in which the $n-1$ multiplications are carried out

Matrix Chain Multiplication

- An order which minimizes the number of scalar multiplications can be found in many ways
- Consider, for example, the brute-force method that tries to compute the number of scalar multiplications of every possible order
- For instance, if we have four matrices M1,M2,M3,and M4, the algorithm will try all the following five orderings:
  - (M1(M2(M3M4)))
  - (M1((M2M3)M4))
  - ((M1M2)(M3M4))
  - ((M1M2)M3)M4))
  - ((M1(M2M3))M4)

Matrix-chain multiplication

- We are given a sequence (chain) <A1, A2, . . . , An> of ***n*** matrices to be multiplied, and we wish to compute the product

  (. . .((A1*A2)*A3) . . .) *An)

- Matrix multiplication is *associative*, and so all parenthesizations yield the same product
- A product of matrices is fully parenthesized, if it is either a single matrix or the product of two fully parenthesized matrix products, surrounded by parentheses
- For example, if the chain of matrices is <A1, A2, A3, A4>, then we can fully parenthesize the product A1A2A3A4 in five distinct ways
- How we parenthesize a chain of matrices can have a dramatic impact on the cost of evaluating the product

SQUARE-MATRIX-MULTIPLY

- The attributes rows and columns are the numbers of rows and columns in a matrix
- We can multiply two matrices ***A*** and ***B*** only if they are ***compatible***: the number of columns of ***A*** must equal the number of rows of ***B***
- If ***A*** is a *p X q* matrix and ***B*** is a *q X r* matrix, the resulting matrix ***C*** is a *p X r* matrix
- The time to compute ***C*** is dominated by the number of scalar multiplications in line 8, which is ***pqr***

MATRIX-MULTIPLY(A, B)

1 **if** $A.columns \neq B.rows$
2   **error** "incompatible dimensions"
3 **else let** C be a new $A.rows \times B.columns$ matrix
4   **for** $i = 1$ **to** $A.rows$
5   **for** $j = 1$ **to** $B.columns$
6             $c_{ij} = 0$
7   **for** $k = 1$ **to** $A.columns$
8                 $c_{ij} = c_{ij} + a_{ik} \cdot b_{kj}$
9   **return** C

Matrix-chain multiplication

- The matrix-chain multiplication problem is stated as follows: given a chain <A1, A2, . . . , An> of ***n*** matrices, where for i = 1, 2, . . . , n, matrix Ai has dimension pi - 1 X pi, fully parenthesize the product A1A2 . . . An in a way that ***minimizes*** the number of ***scalar multiplications***
- Note that in the matrix-chain multiplication problem, we are not actually multiplying matrices
- Our goal is only to determine an order for multiplying matrices that has the lowest cost
- Typically, the time invested in determining this optimal order is more than paid for by the time saved later on when actually performing the matrix multiplications

Counting the number of parenthesizations

- Denote the number of alternative parenthesizations of a sequence of ***n*** matrices by ***P(n)***
- When ***n = 1***, we have just one matrix and therefore only one way to fully parenthesize the matrix product
- When ***n >= 2***, a fully parenthesized matrix product is the product of two fully parenthesized matrix subproducts, and the split between the two subproducts may occur between the k^{th} and $(k + 1)^{st}$ matrices for any ***k = 1, 2, . . . , n - 1***
- Thus, we obtain the recurrence

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

The recurrence is the sequence of Catalan numbers, which grows as $\Omega(4^n/n^{3/2})$

Applying dynamic programming

- Use the dynamic-programming method to determine how to optimally parenthesize a matrix chain

1. Characterize the structure of an optimal solution
2. Recursively define the value of an optimal solution
3. Compute the value of an optimal solution
4. Construct an optimal solution from computed information

Step 1: The structure of an optimal parenthesization

- Let us adopt the notation $A_{i..j}$, where $i <= j$, for the matrix that results from evaluating the product $A_iA_{i+1} \ldots A_j$
- If the problem is nontrivial, i.e., $i < j$, then to parenthesize the product $A_iA_{i+1} \ldots A_j$, we must split the product between A_k and A_{k+1} for some integer k in the range $i <= k < j$
- That is, for some value of k, we first compute the matrices $A_{i..k}$ and $A_{k+1..j}$ and then multiply them together to produce the final product $A_{i..j}$
- The cost of parenthesizing this way is the cost of computing the matrix $A_{i..k}$, plus the cost of computing $A_{k+1..j}$, plus the cost of multiplying them together
- Thus, an optimal solution to an instance of the matrix-chain multiplication problem by splitting the problem into two subproblems (optimally parenthesizing $A_iA_{i+1} \ldots A_k$ and $A_kA_{k+1} \ldots A_j$), finding optimal solutions to subproblem instances, and then combining these optimal subproblem solutions

Step 2: A recursive solution

- Define the cost of an optimal solution recursively in terms of the optimal solutions to subproblems
- Let ***m[i, j]*** be the minimum number of scalar multiplications needed to compute the matrix $A_{i..j}$; for the full problem, the lowest cost way to compute $A_{1..n}$ would thus be ***m[i, n]***
- If $i = j$, the problem is trivial
  - the chain consists of just one matrix $A_{i..i} = A_i$, so that no scalar multiplications are necessary to compute the product
  - Thus, ***m[i, j]=0*** for $i = 1, 2, . . . , n$
- If $i < j$, we split the product $A_i A_{i+1} . . . A_j$ between A_k and A_{k+1} for some integer k in the range $i <= k< j$
  - $m[i, j] = m[i, k]+m[k+1, j] + p_{i-1*}p_{k*}p_j$
  - There are only $j - i$ possible values for ***k,*** however, namely $k = i, i +1, . . . , j -1$
  - Since the optimal parenthesization must use one of these values for ***k***, we need only check them all to find the best

$$m[i,j] = \begin{cases} 0 & \text{if } i = j \, , \\ \min\limits_{i \le k < j} \{m[i,k] + m[k+1,j] + p_{i-1} p_k p_j\} & \text{if } i < j \, . \end{cases}$$

Step 3: Computing the optimal costs

- This procedure assumes that matrix ***A_i*** has dimensions ***p_{i-1} X p_i*** for ***$i = 1, 2, . . . , n$***
- Its input is a sequence ***$p = \langle p_0, p_1, \ldots, p_n \rangle$***, where ***$p.length = n + 1$***
- The procedure uses an **auxiliary table** ***m[1 .. n, 1 .. n]*** for storing the ***m[i, j]*** costs and another **auxiliary table** ***s[1 .. n-1, 2 .. n]*** that records which index of ***k*** achieved the optimal cost in computing ***m[i, j]***
- We use the table ***s*** to construct an optimal solution

MATRIX-CHAIN-ORDER(p)

```
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
for l = 2 to n                    // l is the chain length
    for i = 1 to n − l + 1
        j = i + l − 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
```

$O(n^3)$

Step 3: Computing the optimal costs

$d{=}0$ $d{=}1$ $d{=}2$ $d{=}3$ $d{=}4$ $d{=}5$

						i
$C[1,1]$	$C[1,2]$	$C[1,3]$	$C[1,4]$	$C[1,5]$	$C[1,6]$	1
	$C[2,2]$	$C[2,3]$	$C[2,4]$	$C[2,5]$	$C[2,6]$	2
		$C[3,3]$	$C[3,4]$	$C[3,5]$	$C[3,6]$	3
			$C[4,4]$	$C[4,5]$	$C[4,6]$	4
				$C[5,5]$	$C[5,6]$	5
					$C[6,6]$	6

M_1: 5×10, M_2: 10×4, M_3: 4×6, M_4: 6×10, M_5: 10×2.

$C[1,1] = 0$	$C[1,2] = 200$	$C[1,3] = 320$	$C[1,4] = 620$	$C[1,5] = 348$
	$C[2,2] = 0$	$C[2,3] = 240$	$C[2,4] = 640$	$C[2,5] = 248$
		$C[3,3] = 0$	$C[3,4] = 240$	$C[3,5] = 168$
			$C[4,4] = 0$	$C[4,5] = 120$
				$C[5,5] = 0$

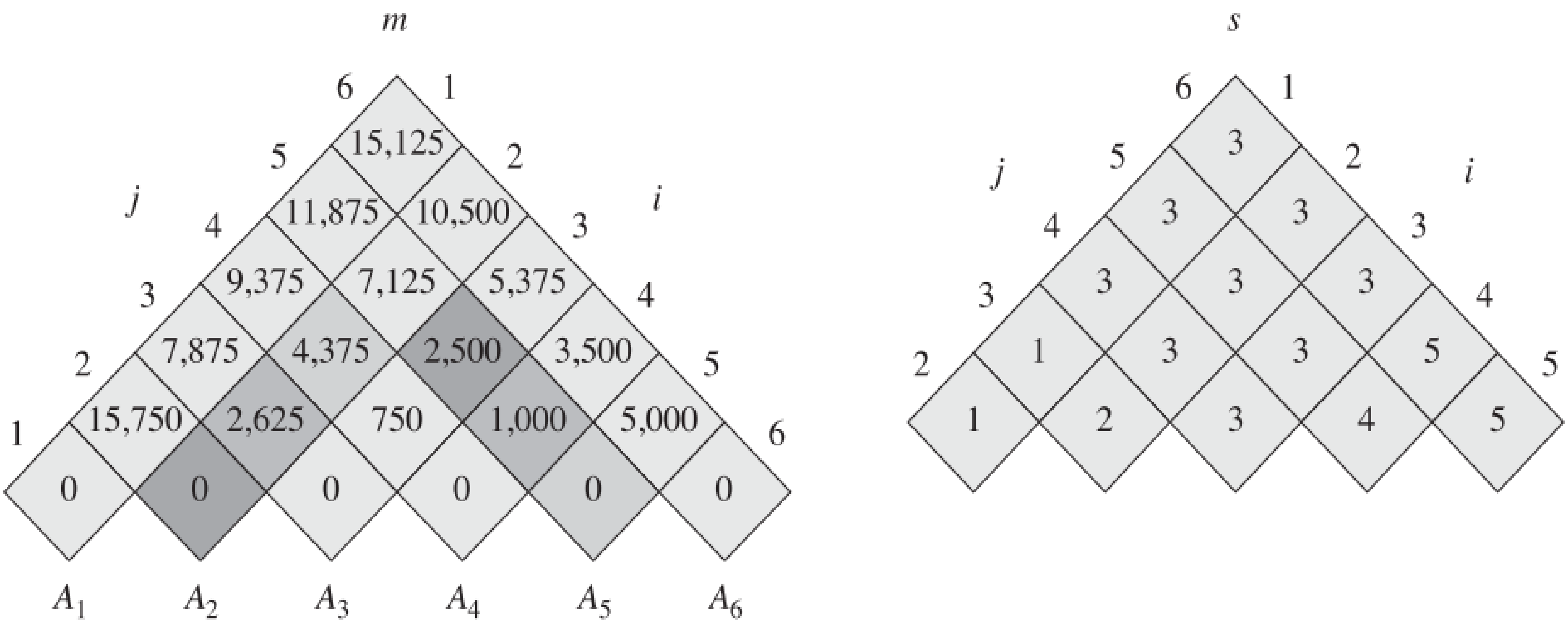


Figure 15.5 The m and s tables computed by MATRIX-CHAIN-ORDER for $n = 6$ and the following matrix dimensions:

matrix	A_1	A_2	A_3	A_4	A_5	A_6
dimension	30×35	35×15	15×5	5×10	10×20	20×25

Step 4: Constructing an optimal solution

- call PRINT-OPTIMAL-PARENS(s,1,n) prints an optimal parenthesization
- Each entry s[i, j] records a value of k such that an optimal parenthesization of $A_i A_{i+1} \dots A_j$ splits the product between A_k and A_{k+1}
- For example , the call PRINT-OPTIMAL-PARENS(s,1,6) prints the parenthesization **((A1(A2A3))((A4A5)A6))**

```
PRINT-OPTIMAL-PARENS(s, i, j)

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 ")"
```

matrix	A_1	A_2	A_3	A_4	A_5	A_6
dimension	30×35	35×15	15×5	5×10	10×20	20×25

The tables are rotated so that the main diagonal runs horizontally. The m table uses only the main diagonal and upper triangle, and the s table uses only the upper triangle. The minimum number of scalar multiplications to multiply the 6 matrices is $m[1,6] = 15{,}125$. Of the darker entries, the pairs that have the same shading are taken together in line 10 when computing

$$
\begin{aligned}
m[2,5] &= \min \begin{cases} m[2,2] + m[3,5] + p_1 p_2 p_5 = 0 + 2500 + 35 \cdot 15 \cdot 20 & = 13{,}000 , \\ m[2,3] + m[4,5] + p_1 p_3 p_5 = 2625 + 1000 + 35 \cdot 5 \cdot 20 & = 7125 , \\ m[2,4] + m[5,5] + p_1 p_4 p_5 = 4375 + 0 + 35 \cdot 10 \cdot 20 & = 11{,}375 \end{cases} \\
&= 7125 .
\end{aligned}
$$