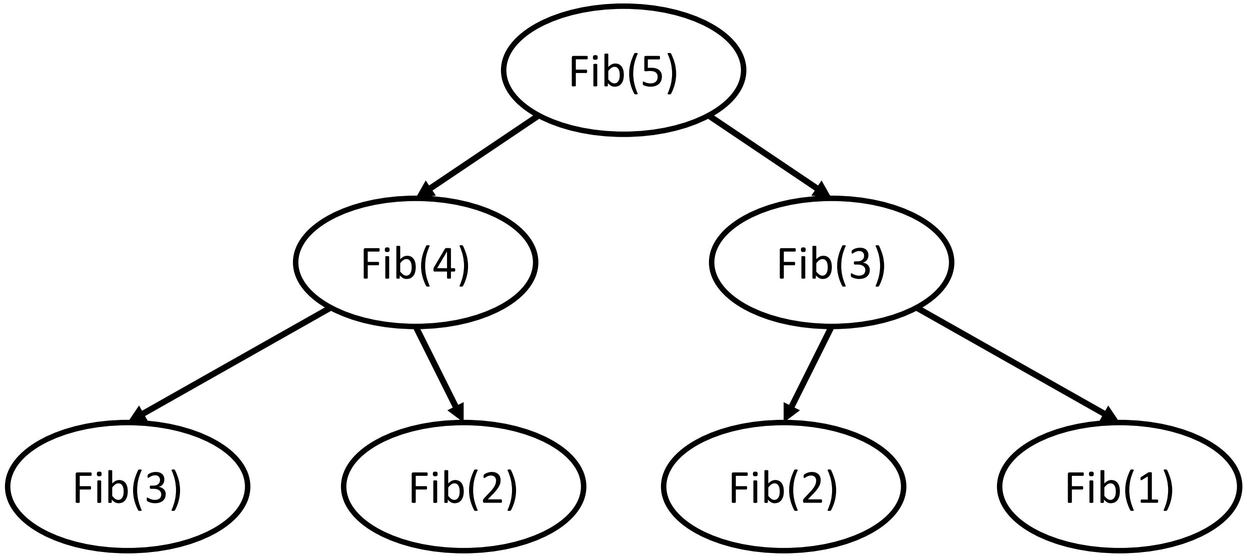
Dynamic Programming

Divide & Conquer vs Dynamic Programming

- Both solve problems by combining the solutions to subproblems.
- Divide & conquer algorithms partition the problem into disjoint subproblems, solve them recursively and then combine their solutions to solve the original problem.
- Dynamic programming applies when the subproblems overlap - i.e. when subproblems share subsubproblems.
- Divide & conquer algorithm does more work than necessary, repeatedly solving the common subsubproblems.
- Dynamic programming algorithm solves each subsubproblem just once and then saves its answer in a table avoiding recomputation.

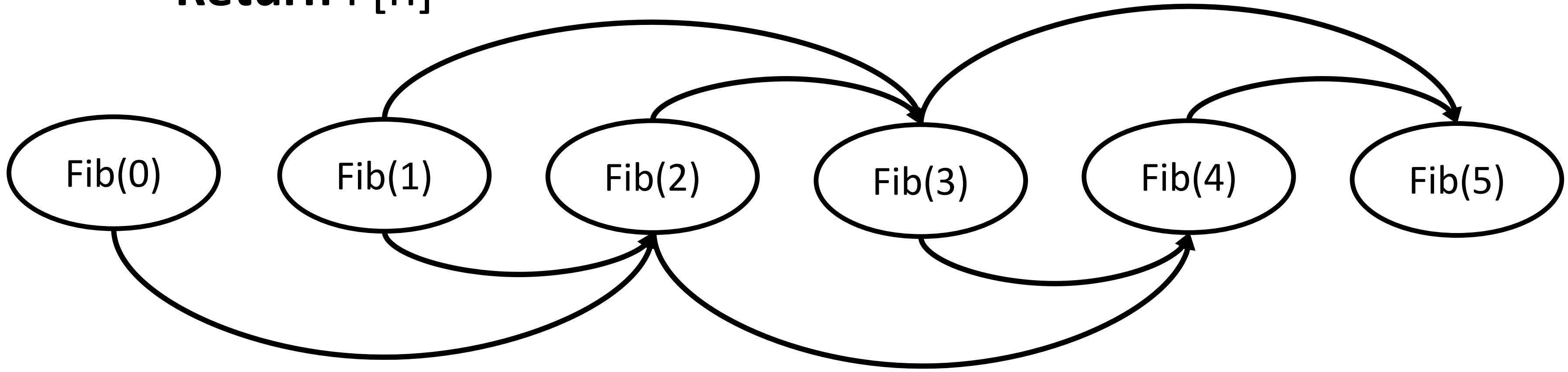
Divide & Conquer vs Dynamic Programming (Example)

```
Fibonacci(n)
        if(n = 0) return 0
        if(n = 1) return 1
        return Fibonacci(n-1)+Fibonacci(n-2)
```



Divide & Conquer vs Dynamic Programming (Example)

```
DynamicFibonacci(n)
    F[0] ← 0
    F[1] ← 1
    for i = 2 to n
        F[i] ← F[i-1] + F[i-2]
    Return F[n]
```



Dynamic Programming

Matrix Chain Multiplication

Matrix Chain Multiplication

- A sequence (chain) $\langle A_1, A_2, \ldots, A_n \rangle$ of n matrices to be multiplied and we wish to compute the product

$$A_1A_2......A_n$$

- 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.

$$(A_1 (A_2 (A_3 A_4)))$$
$$(A_1 ((A_2 A_3) A_4))$$
$$((A_1 A_2) (A_3 A_4))$$
$$((A_1 (A_2 A_3)) A_4)$$
$$(((A_1 A_2) A_3) A_4)$$

Matrix Chain Multiplication

```
MATRIX-MULTIPLY(A, B)
    if A.columns ≠ B.rows
        error "incompatible dimensions"
    else let C be a new A.rows X B.columns matrix
        for i = 1 to A.rows
            for j = 1 to B.columns
                $c_{ij} = 0$
                for k = 1 to A.columns
                    $c_{ij} = c_{ij} + a_{ik} * b_{kj}$
    return C
```

Matrix Chain Multiplication

Consider the problem of a chain $\langle A_1, A_2, A_3 \rangle$ of three matrices.

The dimensions of the matrices are 10 X 100, 100 X 5 and 5 X 50 respectively.

According to the parenthesization $((A_1A_2)A_3)$,

10 . 100 . 5 = 5000 scalar multiplications to compute the 10 X 5 matrix product (A_1A_2), plus another 10 . 5 . 50 = 2500 scalar multiplications to multiply this matrix by A_3, for a total of 7500 scalar multiplications.

according to the parenthesization $(A_1(A_2A_3))$,

100 . 5 . 50 = 25,000 scalar multiplications to compute the 100 X 50 matrix product (A_2A_3), plus another 10 . 100 . 50 = 50,000 scalar multiplications to multiply A_1 by this matrix, for a total of 75,000 scalar multiplications.

Matrix Chain Multiplication

Problem Statement: Given a chain $\langle A_1, A_2, \ldots, A_n \rangle$ of n matrices, where for i = 1,2, … ,n, matrix A_i has dimension $p_{i-1} \times p_i$, fully parenthesize the product $A_1A_2 \ldots A_n$ in a way that minimizes the number of scalar multiplications.

Counting the number of parenthesizations:

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

Structure of an Optimal Solution

Suppose we have to parenthesize the product $\boldsymbol{A_i A_{i+1} \ldots A_j}$.

- we must split the product between $\mathbf{A_k}$ and $\mathbf{A_{k+1}}$ for some integer **k** in the range $\mathbf{i \le k < j}$
- we first compute the matrices $\mathbf{A_{i \ldots k}}$ and $\mathbf{A_{k+1 \ldots j}}$ and then multiply them together to produce the final product $\mathbf{A_{i \ldots j}}$

The cost of parenthesizing this way is:

the cost of computing the matrix $\mathbf{A_{i \ldots k}}$

+ the cost of computing $\mathbf{A_{k+1 \ldots j}}$

+ the cost of multiplying them together.

A Recursive Solution

For the matrix-chain multiplication problem, we pick as our subproblems the problems of determining the minimum cost of parenthesizing $\mathbf{A_i A_{i+1} \ldots A_j}$ for $\mathbf{1 \le i \le j \le n}$.

Let ***m[i, j]*** be the minimum number of scalar multiplications needed to compute the matrix $\mathbf{A_{i \ldots j}}$.

For the full problem, the lowest cost way to compute $\mathbf{A_{1 \ldots n}}$ would thus be ***m[1, n]***.

A Recursive Solution

We can define ***m[i, j]*** recursively as follows:
For i=j, there is only one matrix $\mathbf{A}_{i..i} = \mathbf{A}_i$, no scalar multiplications are necessary.

Thus ***m[i,i]*** = 0

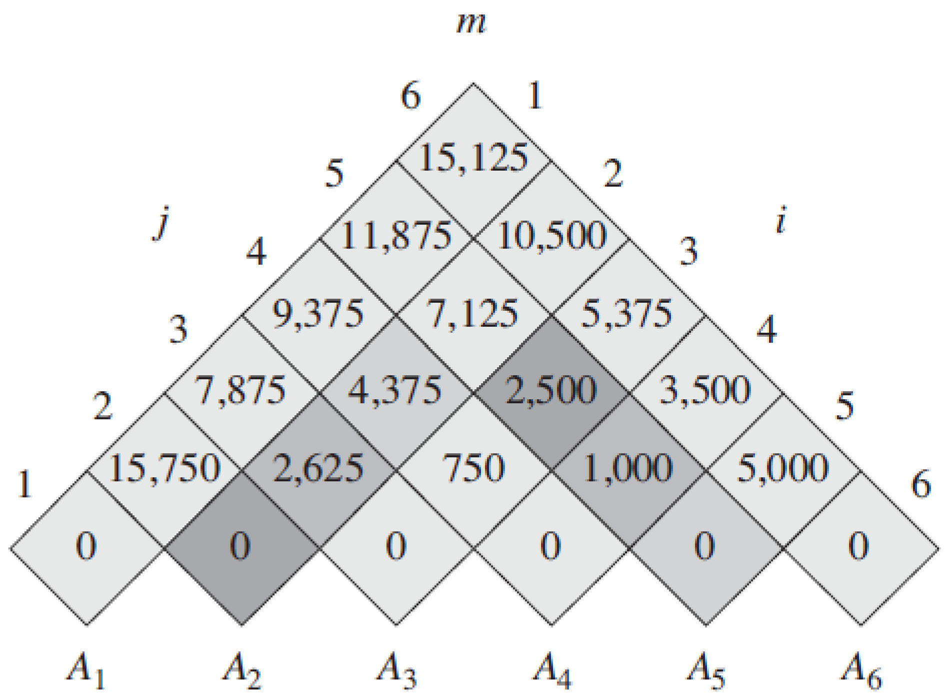
$$m[i,j] = m[i,k] + m[k+1,j] + p_{i-1}p_k p_j$$

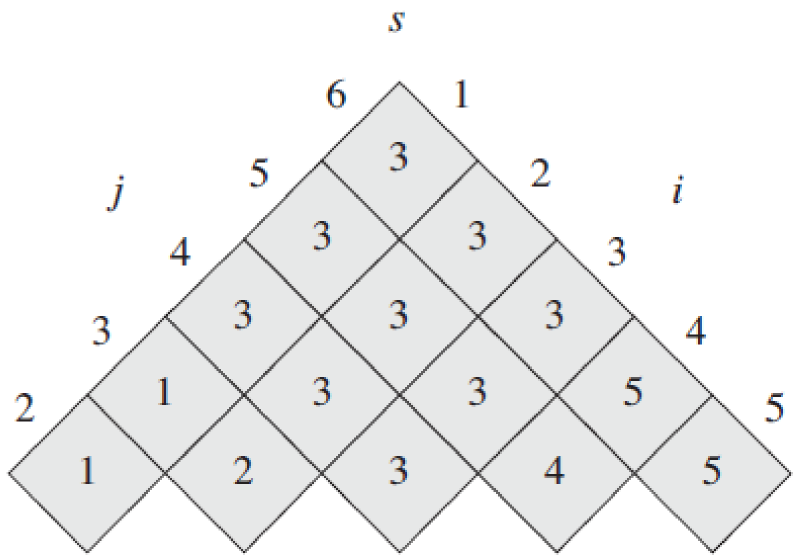
$$m[i,j] = \begin{cases} 0 & \text{if } i = j \\ \min\limits_{i \le k < j} \{m[i,k] + m[k+1,j] + p_{i-1}p_k p_j\} & \text{if } i < j \end{cases}$$

We define ***s[i, j]*** to be a value of k at which we split the product $\mathbf{A}_i\mathbf{A}_{i+1}\ldots\mathbf{A}_j$.

Computing the Optimal Costs

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





Computing the Optimal Costs

For example we compute ***m[2, 5]***

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

The minimum number of scalar multiplications to multiply the 6 matrices is m[1, 6] = 15,125

Pseudocode

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** $n - l + 1$
        $j = i + l - 1$
        $m[i, j] = \infty$
        **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 $s$
```

Constructing an Optimal Solution

The following recursive procedure prints an optimal parenthesization of $\langle A_i, Ai_{+1},\ldots, Aj\rangle$, given the **s** table computed by MATRIX-CHAIN-ORDER and the indices **i** and **j**

```
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 ")"
```

PRINT-OPTIMAL-PARENS(s, 1, 6) prints $((A_1(A_2A_3))((A_4A_5)A_6))$

Dynamic Programming

Longest Common Subsequence

Longest Common Subsequence

- Biological applications often need to compare the DNA of two (or more) different organisms.
- 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.

S1 = ACCGGTCGAGTGCGCGGAAGCCGGCCGAA

S2 = GTCGTTCGGAATGCCGTTGCTCTGTAAA

Here we will not find whether one is substring of another.

Rather we will determine what is the least possible number of changes needed to turn one into the other.

LCS - Example

X = <A, B, C, B, D, A, B>

Y = <B, D, C, A, B, A>

The sequence **<B, C, A>** is a common subsequence of both X and Y. But it is not a longest common subsequence.

The sequence **<B, C, B, A>** which is also common to both X and Y.

Since X and Y have no common subsequence of length 5 or greater, **<B, C, B, A>** is a LCS.

Optimal Substructure of an LCS

Theorem 15.1:

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

A Recursive Solution

- To find an LCS of $X = \langle x_1, x_2, \ldots, x_m \rangle$ and $Y = \langle y_1, y_2, \ldots, y_n \rangle$. If $x_m = y_n$, we must find an LCS of X_{m-1} and Y_n. Appending $x_m = y_n$ to this LCS yields an LCS of X and Y.

- If $x_m \neq y_n$, then we must solve two subproblems:
  - finding an LCS of X_{m-1} and Y
  - finding an LCS of X and Y_{n-1}

- Let us define ***c[i, j]*** to be the length of an LCS of the sequences X_i and Y_j. If either i = 0 or j = 0, one of the sequences has length 0, and so the LCS has length 0.

A Recursive Solution

- The optimal substructure of the LCS problem gives the recursive formula:

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

Pseudocode

```
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] = $ "$\nwarrow$"
          elseif $c[i-1, j] \geq c[i, j-1]$
              $c[i, j] = c[i-1, j]$
              $b[i, j] = $ "$\uparrow$"
          else $c[i, j] = c[i, j-1]$
              $b[i, j] = $ "$\leftarrow$"
  return $c$ and $b$
```

Computing LCS

	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						
2	B	0						
3	C	0						
4	B	0						
5	D	0						
6	A	0						
7	B	0						

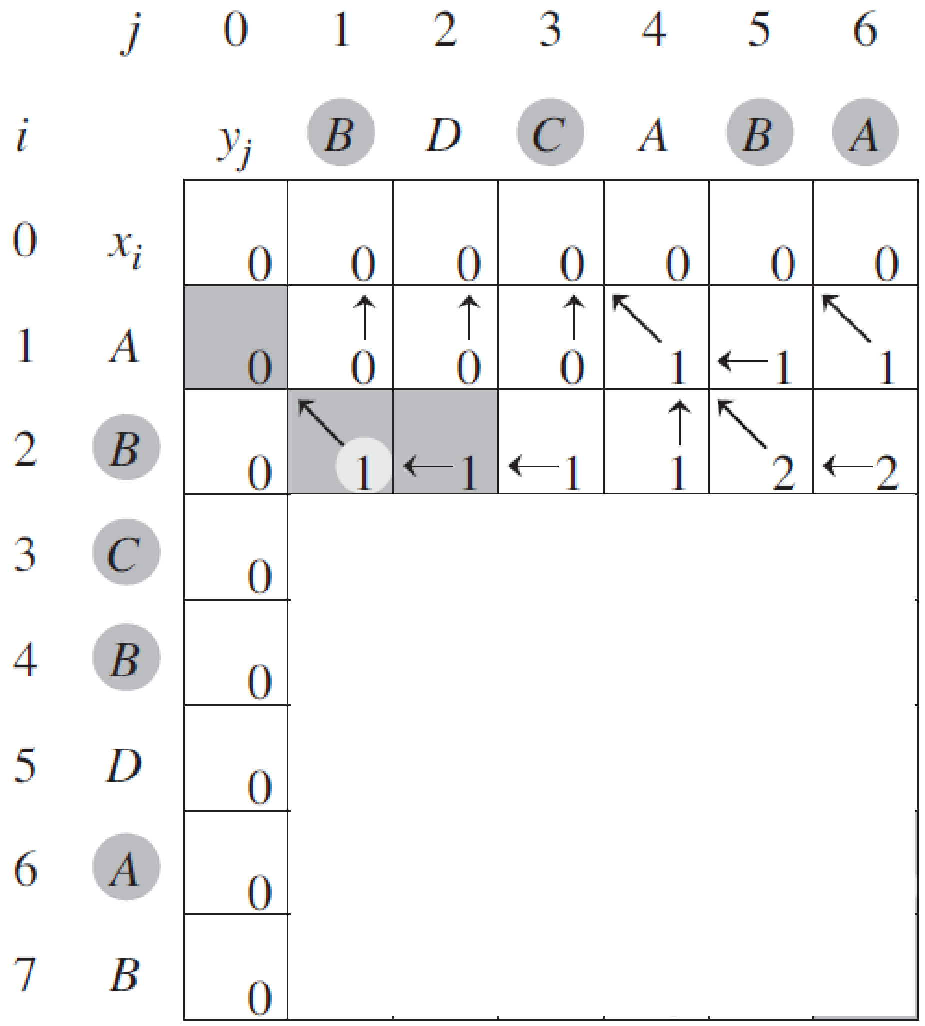
```
**if** $x_i == y_j$
    $c[i, j] = c[i-1, j-1] + 1$
    $b[i, j] =$ "$\nwarrow$"
**elseif** $c[i-1, j] \geq c[i, j-1]$
    $c[i, j] = c[i-1, j]$
    $b[i, j] =$ "$\uparrow$"
**else** $c[i, j] = c[i, j-1]$
    $b[i, j] =$ "$\leftarrow$"
```

Computing LCS

	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	$\uparrow$ 0	$\uparrow$ 0	$\uparrow$ 0	$\nwarrow$ 1	$\leftarrow$ 1	$\nwarrow$ 1
2	B	0						
3	C	0						
4	B	0						
5	D	0						
6	A	0						
7	B	0						

```
**if** $x_i == y_j$
    $c[i, j] = c[i-1, j-1] + 1$
    $b[i, j] = $ "$\nwarrow$"
**elseif** $c[i-1, j] \geq c[i, j-1]$
    $c[i, j] = c[i-1, j]$
    $b[i, j] = $ "$\uparrow$"
**else** $c[i, j] = c[i, j-1]$
    $b[i, j] = $ "$\leftarrow$"
```

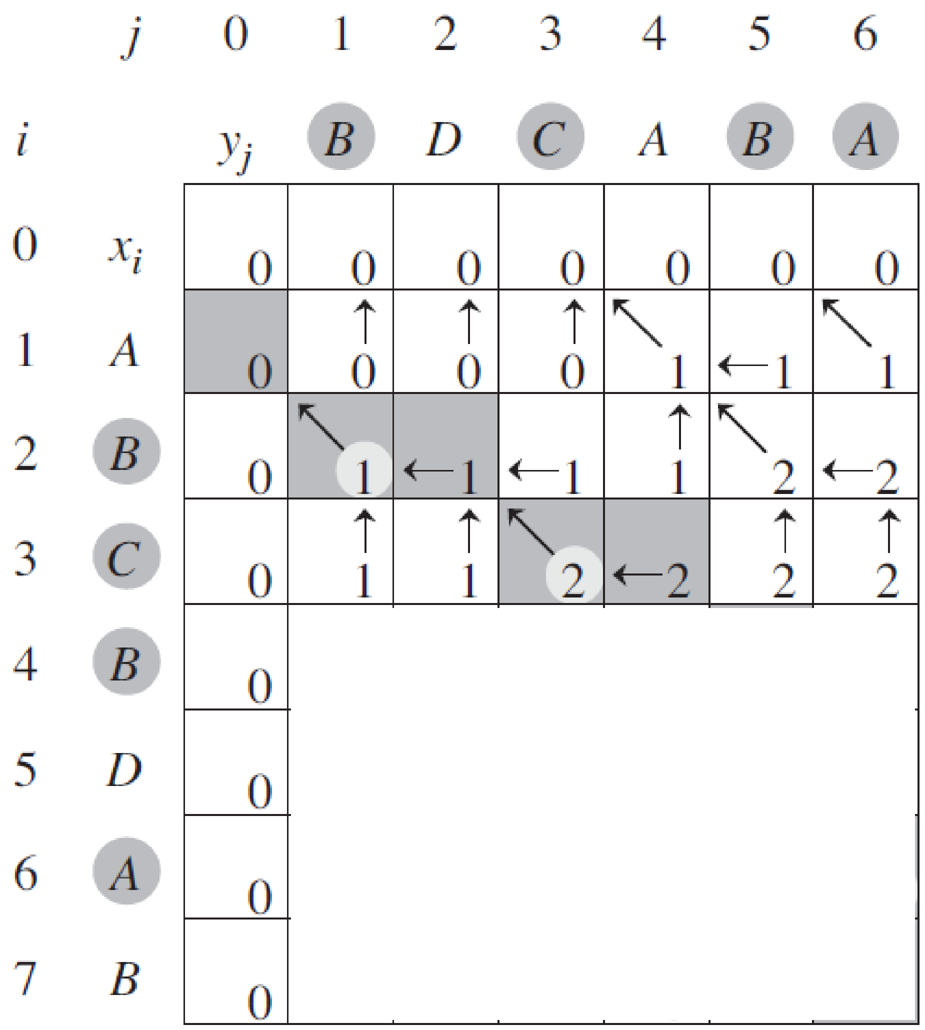
Computing LCS



	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	$\uparrow$ 0	$\uparrow$ 0	$\uparrow$ 0	$\nwarrow$ 1	$\leftarrow$ 1	$\nwarrow$ 1
2	B	0	$\nwarrow$ 1	$\leftarrow$ 1	$\leftarrow$ 1	$\uparrow$ 1	$\nwarrow$ 2	$\leftarrow$ 2
3	C	0						
4	B	0						
5	D	0						
6	A	0						
7	B	0						

```
if $x_i == y_j$
    $c[i, j] = c[i-1, j-1] + 1$
    $b[i, j] = $ "$\nwarrow$"
elseif $c[i-1, j] \geq c[i, j-1]$
    $c[i, j] = c[i-1, j]$
    $b[i, j] = $ "$\uparrow$"
else $c[i, j] = c[i, j-1]$
    $b[i, j] = $ "$\leftarrow$"
```

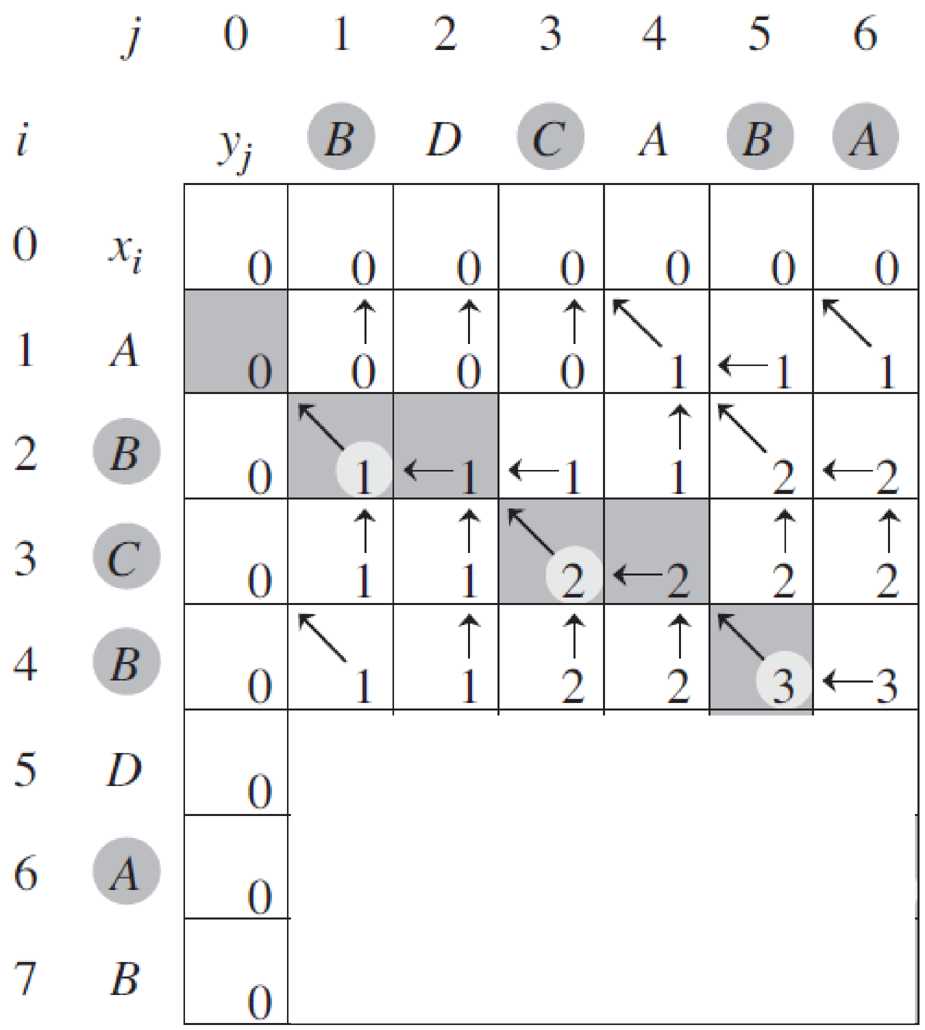
Computing LCS



	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						
5	D	0						
6	A	0						
7	B	0						

```
if $x_i == y_j$
    $c[i, j] = c[i-1, j-1] + 1$
    $b[i, j] =$ "↖"
elseif $c[i-1, j] \geq c[i, j-1]$
    $c[i, j] = c[i-1, j]$
    $b[i, j] =$ "↑"
else $c[i, j] = c[i, j-1]$
    $b[i, j] =$ "←"
```

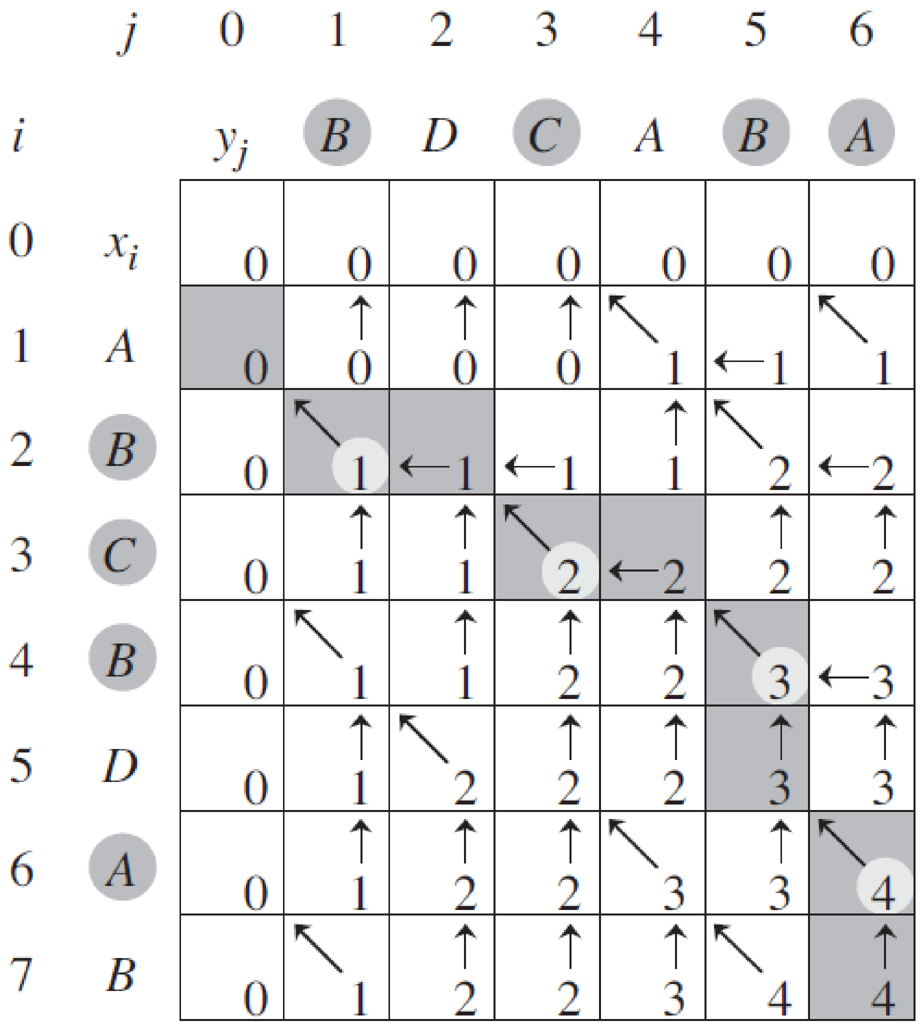
Computing LCS



	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						
6	A	0						
7	B	0						

```
if $x_i == y_j$
    $c[i, j] = c[i-1, j-1] + 1$
    $b[i, j] = $ "↖"
elseif $c[i-1, j] \geq c[i, j-1]$
    $c[i, j] = c[i-1, j]$
    $b[i, j] = $ "↑"
else $c[i, j] = c[i, j-1]$
    $b[i, j] = $ "←"
```

Computing LCS



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

```
if $x_i == y_j$
    $c[i,j] = c[i-1, j-1] + 1$
    $b[i,j] = $ "↖"
elseif $c[i-1, j] \geq c[i, j-1]$
    $c[i,j] = c[i-1, j]$
    $b[i,j] = $ "↑"
else $c[i,j] = c[i, j-1]$
    $b[i,j] = $ "←"
```

Printing LCS

```
PRINT-LCS($b, X, i, j$)
if $i == 0$ or $j == 0$
    return
if $b[i, j] ==$ "$\nwarrow$"
    PRINT-LCS($b, X, i - 1, j - 1$)
    print $x_i$
elseif $b[i, j] ==$ "$\uparrow$"
    PRINT-LCS($b, X, i - 1, j$)
else PRINT-LCS($b, X, i, j - 1$)
```

Analysis

- To compute the table of c the time complexity is **Θ**(**mn**) where **m** and **n** are the length of the two sequences.

- To print the LCS the time complexity is **Θ**(**m+n**), because it decrements at least one of **i** and **j** in each recursive call.

Dynamic Programming

Optimal Binary Search Tree

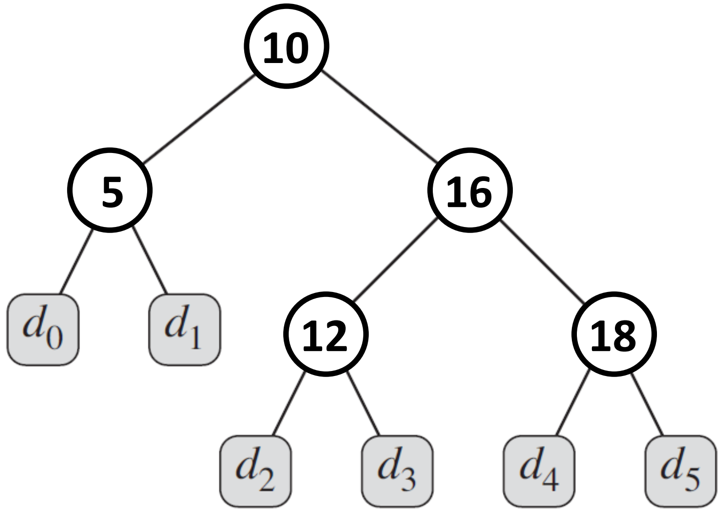
Optimal Binary Search Tree

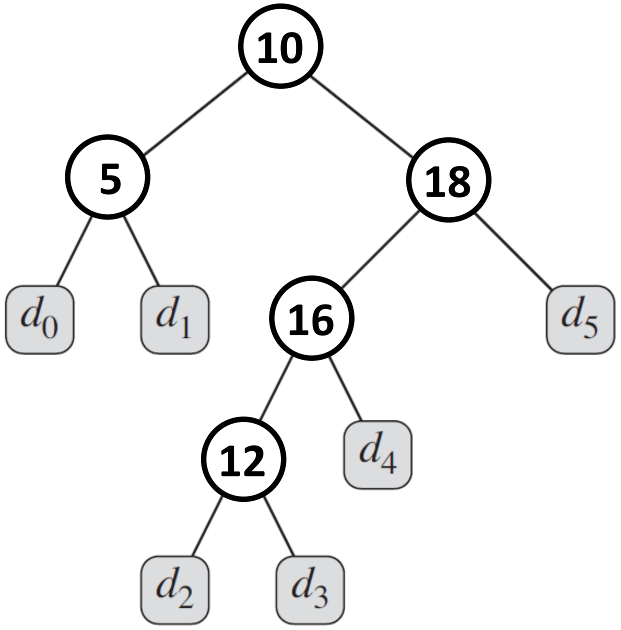
- Let us assume a sequence K = <k_1, k_2, … , k_n> of n distinct keys in sorted order (so that $k_1 < k_2 < \ldots < k_n$).
- For each key **k_i** , we have a probability ***p_i*** that a search will be for **k_i**.
- Some searches may be for values not in K, and so we also have n + 1 “dummy keys” d_0, d_1, d_2, … ,d_n representing values not in K.
- For each dummy key **d_i** , we have a probability ***q_i*** that a search will correspond to **d_i**.

Optimal Binary Search Tree

- The dummy key d_i represents all values between k_i and k_{i+1}.

i	0	1	2	3	4	5
p_i		0.15	0.10	0.05	0.10	0.20
q_i	0.05	0.10	0.05	0.05	0.05	0.10





Optimal Binary Search Tree

- Every search is either successful (finding some key k_i) or unsuccessful (finding some dummy key d_i), and so we have

$$\sum_{i=1}^{n} p_i + \sum_{i=0}^{n} q_i = 1$$

- The expected cost of a search in T is:

$$\mathrm{E}\,[\text{search cost in } T] = \sum_{i=1}^{n} (\text{depth}_T(k_i) + 1) \cdot p_i + \sum_{i=0}^{n} (\text{depth}_T(d_i) + 1) \cdot q_i$$

A Recursive Solution

- Let us consider subproblem for finding an optimal binary search tree containing the keys k_i , … ,k_j, where $i \geq 1$, $j \leq n$, and $j \geq i - 1$.
- When $j = i - 1$, there are no actual keys; we have just the dummy key d_{i-1}.
- Let us define ***e[i, j]*** as the expected cost of searching an optimal binary search tree containing the keys k_i , … ,k_j .
- Ultimately, we wish to compute $e[1, n]$.
- When $j = i – 1$, the expected search cost is $e[i, i – 1] = q_{i-1}$

A Recursive Solution

- When $j \geq i$, we need to select a root k_r from among k_i , … ,k_j .
- Make two optimal binary search trees:
  - Left subtree containing keys k_i , … ,k_{r-1}
  - Right subtree containing keys k_{r+1} , … , k_j .
- If k_r is the root of an optimal subtree containing keys k_i , … , k_j, we have

$$e[i,j] = p_r + (e[i,r-1] + w(i,r-1)) + (e[r+1,j] + w(r+1,j))$$

here, $w(i,j) = w(i,r-1) + p_r + w(r+1,j)$,

- Rewriting, $e[i,j] = e[i,r-1] + e[r+1,j] + w(i,j)$

A Recursive Solution

- We choose the root that gives the lowest expected search cost, giving us our final recursive formulation:

$$e[i, j] = \begin{cases} q_{i-1} & \text{if } j = i - 1 , \\ \min\limits_{i \le r \le j} \{e[i, r-1] + e[r+1, j] + w(i, j)\} & \text{if } i \le j . \end{cases}$$

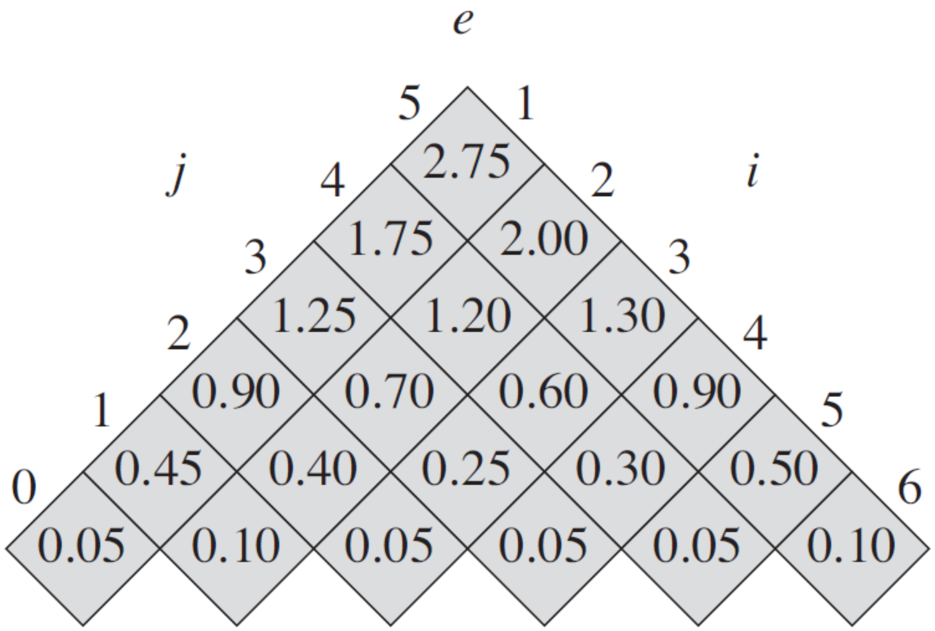
- To help us keep track of the structure of optimal binary search trees, we define ***root[i, j]*** .

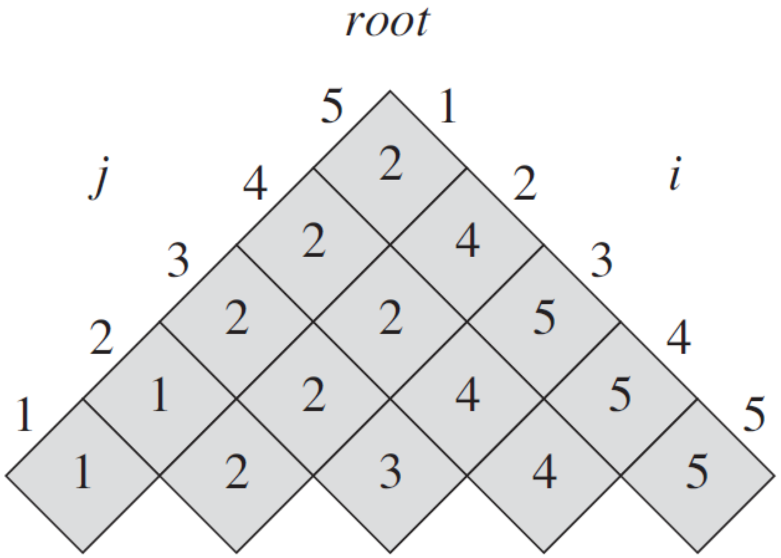
Pseudocode

```
OPTIMAL-BST$(p, q, n)$
let $e[1..n+1, 0..n]$, $w[1..n+1, 0..n]$,
        and $root[1..n, 1..n]$ be new tables
for $i = 1$ to $n + 1$
    $e[i, i-1] = q_{i-1}$
    $w[i, i-1] = q_{i-1}$
for $l = 1$ to $n$
    for $i = 1$ to $n - l + 1$
        $j = i + l - 1$
        $e[i, j] = \infty$
        $w[i, j] = w[i, j-1] + p_j + q_j$
        for $r = i$ to $j$
            $t = e[i, r-1] + e[r+1, j] + w[i, j]$
            if $t < e[i, j]$
                $e[i, j] = t$
                $root[i, j] = r$
return $e$ and $root$
```

Example

i	0	1	2	3	4	5
p_i		0.15	0.10	0.05	0.10	0.20
q_i	0.05	0.10	0.05	0.05	0.05	0.10





Analysis

- **for** loops of the OPTIMAL-BST are nested three deep and each loop index takes on at most n values.
- The OPTIMAL-BST procedure takes $\boldsymbol{\Theta}(\mathbf{n^3})$ time,