Dynamic Programming

Hsin-Hung Lin (林信宏)

hlin@iis.sinica.edu.tw

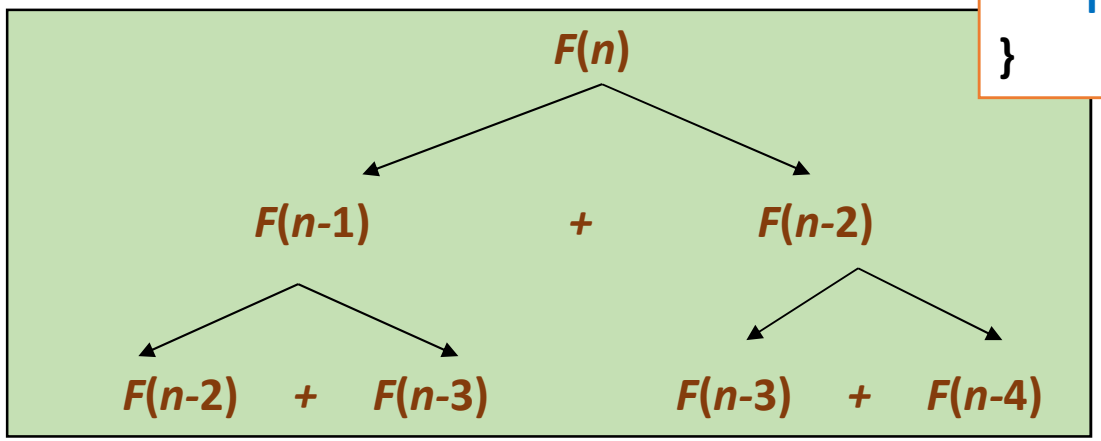
2018/12/10

Based on slides from:
Shi-Chun Tsai
Roger Crawfis

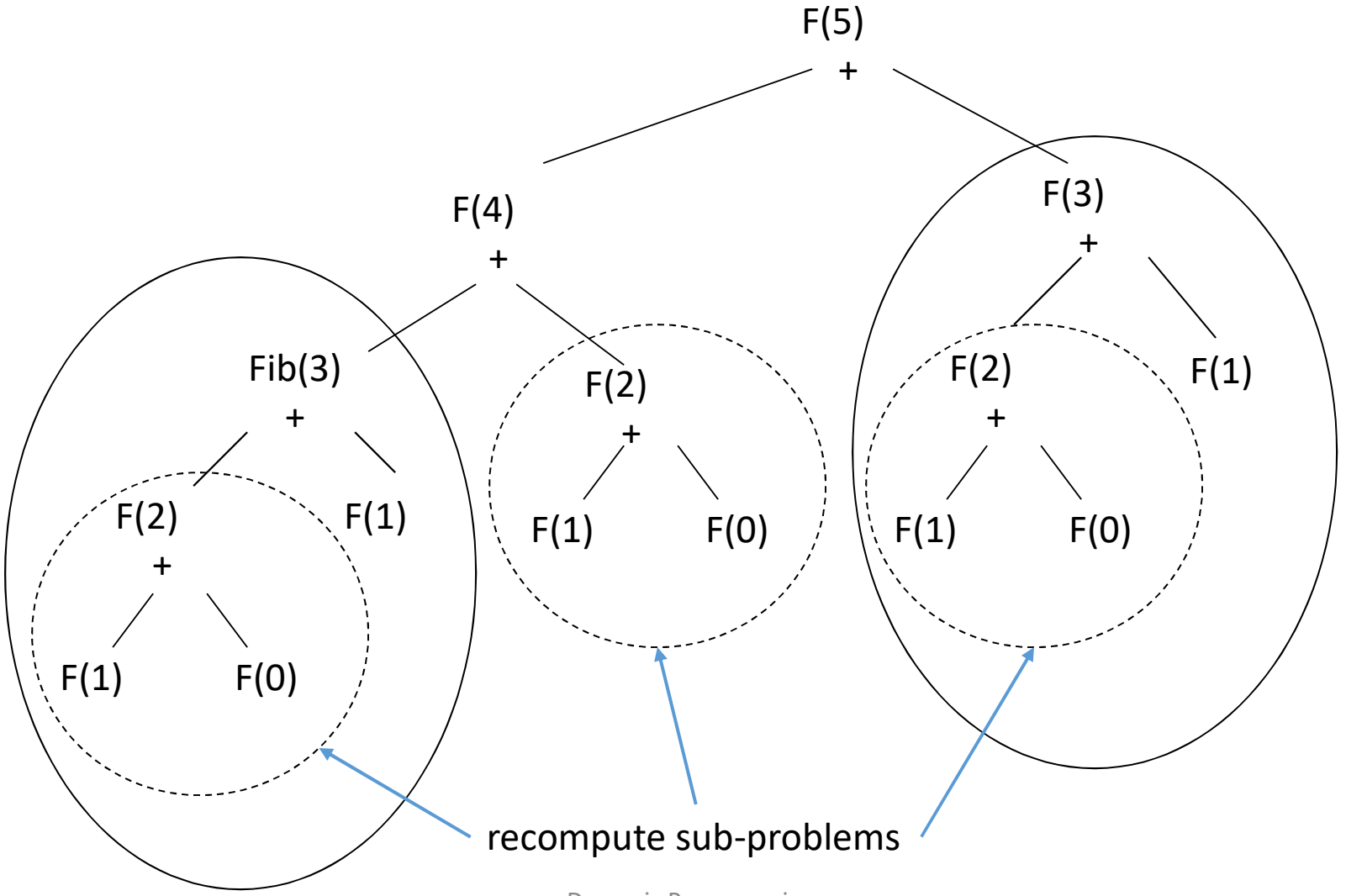
Fibonacci Numbers

- Computing the n-th Fibonacci number recursively:
  - F(n) = F(n-1) + F(n-2)
  - F(0) = 0
  - F(1) = 1
  - Top-down approach
  - Not efficient! ($O(2^n)$)

```
int F(int n)
{
   if (n <= 1)
      return 1;
   else
      return F(n - 1) + F(n - 2);
}
```



Fibonacci Numbers



Fibonacci Numbers

- Computing the n-th Fibonacci number using a bottom-up approach:
  - F(0) = 0
  - F(1) = 1
  - F(2) = 1+0 = 1
  - …
  - F(n-2) =
  - F(n-1) =
  - F(n) = F(n-1) + F(n-2)

0	**1**	**1**	**. . .**	***F*(*n*-2)**	***F*(*n*-1)**	***F*(*n*)**

- Efficiency:
  - Time – O(n)
  - Space – O(n)

Dynamic Programming

- Dynamic Programming is an algorithm design technique for *optimization problems:* often minimizing or maximizing.
- Like divide and conquer, DP solves problems by combining solutions to sub-problems.
- Unlike divide and conquer, sub-problems are not independent.
  - Sub-problems may share sub-sub-problems

Dynamic Programming

- The term Dynamic Programming comes from Control Theory, not computer science. Programming refers to the use of tables (arrays) to construct a solution.
- In dynamic programming we usually reduce time by increasing the amount of space
- We solve the problem by solving sub-problems of increasing size and saving each optimal solution in a table (usually).
- The table is then used for finding the optimal solution to larger problems.
- Time is saved since each sub-problem is solved only once.

Dynamic Programming

- The best way to get a feel for this is through some more examples.
  - Rod-Cutting Problem
  - Matrix Chaining Multiplication
  - Longest Common Subsequence
  - Optimal Binary Search

Rod cutting problem

Given a rod of length n inches and a table of prices p[i] for i = 1, 2, … , n, determine the maximum revenue r[n] obtained by cutting up the rod and selling the pieces.

P[i] for example：

i	1	2	3	4	5	6	7	8	9	10
p[i]	1	5	8	9	10	17	17	20	24	30

given n=7,
7=3+4, revenue = 8 + 9 = 17
7=1+6, revenue = 1+17 = 18.
7=2+2+3, revenue = 5+5+8 = 18.

Rod cutting problem

If an optimal solution cuts the rod into k pieces:

$n=i_1+i_2+\ldots+i_k$ ---- sum of each pieces' lengths

$r[n]=p[i_1]+\ldots+p[i_k]$ ---- corresponding revenue

$r[n] = \max_{i=1..n} \{ p[i]+r[n-i]\}$,

$r[0]=0$.

```
CUT-ROD(p, n)
1.  if n==0 return 0;
2.  q=-99999999;
3.  for  i=1 to n
4.      q= max(q, p[i]+CUT-ROD(p, n - i));
5.   return q;
```

Rod cutting problem

```
Memoized-CUT-ROD(p, n)
1.  let r[0,..n] be a new array
2.  for  i=0 to n
3.      r[i]= -9999999;
4.  return Memoized-CUT-ROD-AUX(p, n, r)
```

Time $=O(n^2)$, why?

```
Memoized-CUT-ROD-AUX(p, n, r)
1.  if  r[n]>=0   return r[n] ;
2.  if  n==0    q=0;
3.  else q=-9999999;
4.       for  i=0 to n
5.            q = max(q, p[i]+Memoized-CUT-ROD-AUX(p, n-i, r));
6.  r[n] = q;
7.   return q;
```

```
BOTTOM-UP-CUT-ROD(p, n)
1.   let r[0,..n] be a new array
2.   r[0]=0;
3.   for  j=1 to n
4.        q=-9999999;
5.        for i= 1 to j
6.             q=max(q,  p[i] + r[j-i]);
7.        r[j]=q;
8.   return r[n];
```

```
EXTENDED-BOTTOM-UP-CUT-ROD(p, n)
1.   let r[0,..n] and s[0..n] be a new arrays
2.   r[0]=0;
3.   for  j=1 to n
4.        q=-9999999;
5.        for i= 1 to j
6.             if q < p[i] + r[j-i])
7.                  q = p[i] + r[j-i]);
8.                  s[j]=i;
9.        r[j]=q;
10.  return r and s;
```

```
PRINT-CUT-ROD-SOLUTION(p, n)
1.   (r, s) = EXTENDED-BOTTOM-UP-CUT-ROD(p, n);
2.   while n > 0
3.        print s[n];      n=n - s[n];
```

i	1	2	3	4	5	6	7	8	9	10
p[i]	1	5	8	9	10	17	17	20	24	30
r[i]	1	5	8	10	13	17	18	22	25	30
s[i]	1	2	3	2	2	6	1	2	3	10

Matrix Chain Multiplication

Given a chain $A_1, A_2, \cdots, A_n$ of matrices where A_i, $1 \le i \le n$, has dimension $p_{i-1} \times p_i$, fully parenthesize (i.e., find a way to evaluate) the product $A_1 A_2 \cdots A_n$ such that the number of scalar multiplications is minimum.

Ex : $A_1 \times A_2 \times A_3 \times A_4$

p_i **:13 5 89 3 34**

There are five ways to parenthesize

$(A_1(A_2(A_3A_4)))$, $(A_1((A_2A_3)A_4))$, $((A_1A_2)(A_3A_4))$,

$((A_1(A_2A_3))A_4)$, $(((A_1A_2)A_3)A_4)$.

串列矩陣相乘(例)

$(A_1(A_2(A_3A_4))) \rightarrow A_1 \times (A_2A_3A_4) \rightarrow A_2 \times (A_3A_4) \rightarrow A_3 \times A_4$

cost = 13*5*34 + 5*89*34 + 89*3*34

= 2210 + 15130 + 9078

= 26418

$A_1 \times A_2 \times A_3 \times A_4$

13 5 89 3 34

$(A_1(A_2(A_3A_4)))$, costs = 26418

$(A_1((A_2A_3)A_4))$, costs = 4055

$((A_1A_2)(A_3A_4))$, costs = 54201

$((A_1(A_2A_3))A_4)$, costs = 2856

$(((A_1A_2)A_3)A_4)$, costs = 10582

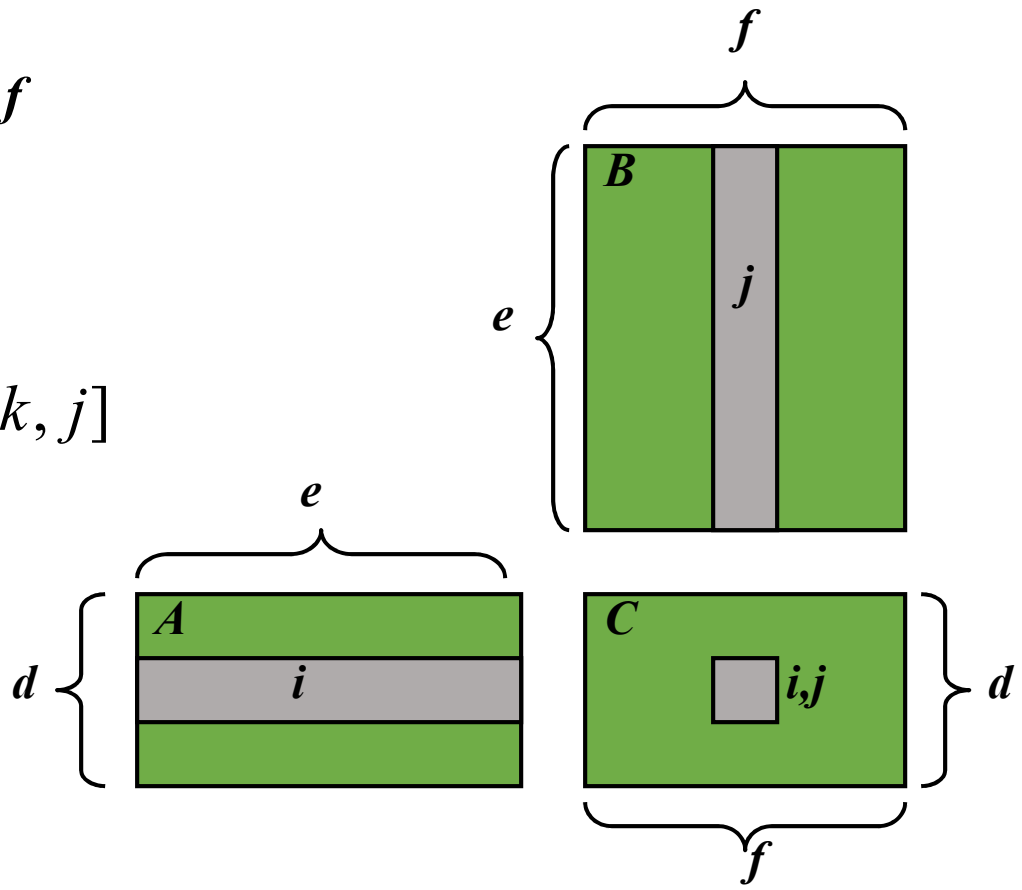
Enumeration Approach

- **Matrix Chain-Product Alg.:**
  - Try all possible ways to parenthesize $A=A_0*A_1*\ldots*A_{n-1}$
  - Calculate number of ops for each one
  - Pick the one that is best
- Running time:
  - The number of parenthesizations is equal to the number of binary trees with n nodes
  - This is **exponential**!
  - It is called the Catalan number, and it is almost 4^n.
  - This is a terrible algorithm!

Matrix Chain Multiplication

- Review: Matrix Multiplication.
  - $C = A*B$
  - A is $d \times e$ and B is $e \times f$
  - $O(d \cdot e \cdot f)$ time

$$C[i,j] = \sum_{k=0}^{e-1} A[i,k] * B[k,j]$$



Matrix Chain Multiplication

- **Matrix Chain Multiplication:**
  - Compute $A=A_0*A_1*\ldots*A_{n-1}$
  - A_i is $d_i \times d_{i+1}$
  - Problem: How to parenthesize?
- Example
  - B is 3×100
  - C is 100×5
  - D is 5×5
  - (B*C)*D takes 1500 + 75 = 1575 ops
  - B*(C*D) takes 1500 + 2500 = 4000 ops

Catalan Number

For any n, # ways to fully parenthesize the product of a chain of n+1 matrices

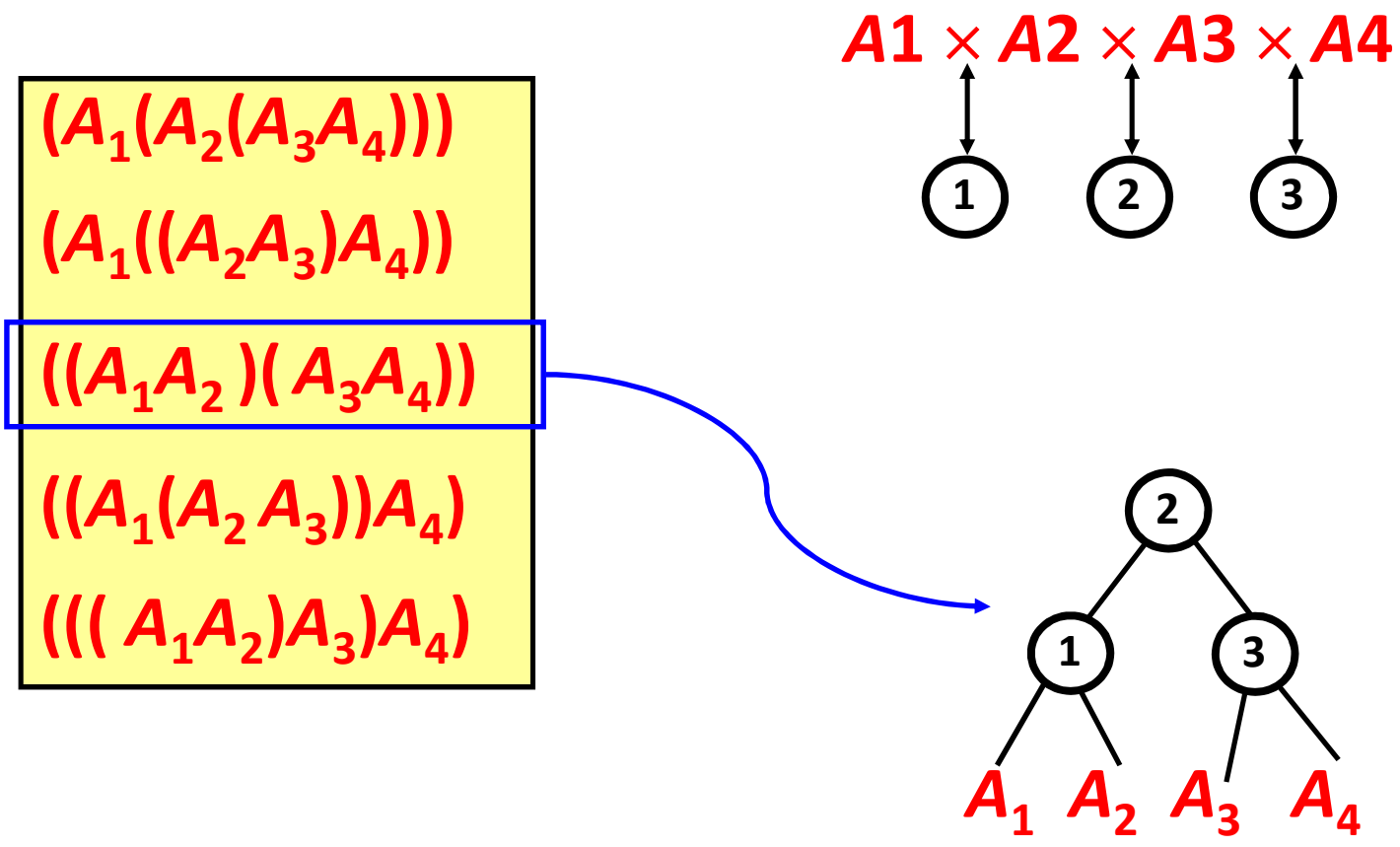
= # binary trees with n nodes.

= # permutations generated from 1 2 … n through a stack.

= # n pairs of fully matched parentheses.

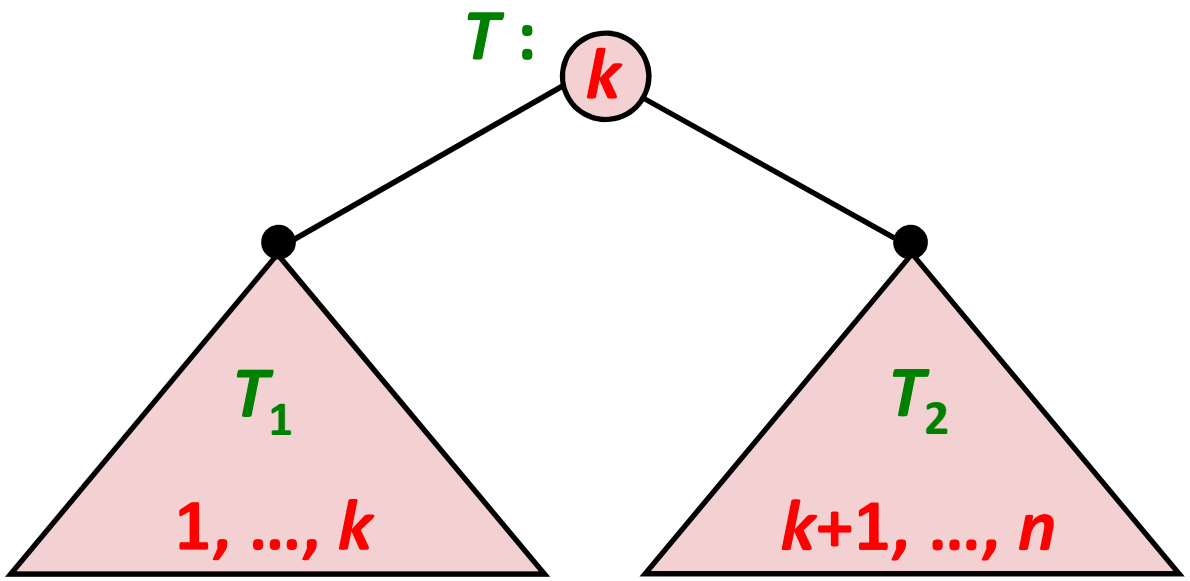
= n-th Catalan Number = $C(2n, n)/(n+1) = \Omega(4^n/n^{3/2})$

Binary Tree View



Define Sub-problem

- **If T is an optimal solution for $A_1, A_2, \ldots, A_n$**



- **then, T_1 (resp. T_2) is an optimal solution for $A_1, A_2, \ldots, A_k$ (resp. $A_{k+1}, A_{k+2}, \ldots, A_n$).**

Define Sub-problem

- **Let $m[i, j]$ be the minmum number of scalar multiplications needed to compute the product $A_i \ldots A_j$, for $1 \le i \le j \le n$.**
- **If the optimal solution splits the product $A_i \ldots A_j = (A_i \ldots A_k) \times (A_{k+1} \ldots A_j)$, for some k, $i \le k < j$, then $m[i, j] = m[i, k] + m[k+1, j] + p_{i-1} p_k p_j$. Hence, we have :**

$$m[i, j] = \min_{i \le k < j} \{m[i, k] + m[k+1, j] + p_{i-1} p_k p_j\}$$
$$= 0 \text{ if } i = j$$

$<P_0, P_1, \ldots, P_n>$

$A_{P_0 \times P_1} A_{P_1 \times P_2} \ldots A_{P_{n-1} \times P_n}$

```
MATRIX-CHAIN-ORDER(P)
1.  n = p.length -1;
2.  let m[1..n, 1..n] and s[1..n-1, 2..n] be new tables;
3.  for i= 1 to n      m[i, i]=0;
4.  for ℓ= 2 to n     // the chain length
5.        { for i = 1 to n – ℓ + 1
6.                { j = i + ℓ - 1;
7.                  m[i, j]= ∞;
8.                  for k= i to j-1
9.                        { q = m[i, k] + m[k+1, j]+ P_{i-1}P_kP_j
10.                         if q<m[i, j]
11.                               { m[i, j] = q ;  s[i, j] = k ;}
12.                       } } }
13. return m and s
```

Time = $O(n^3)$

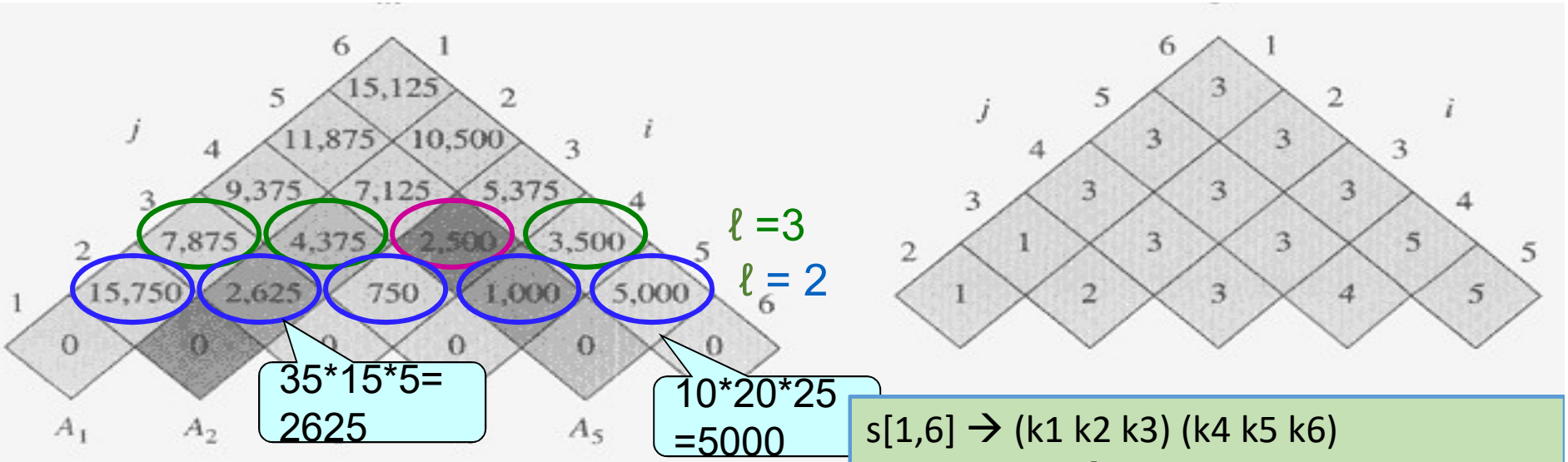


Figure 15.3 The m and s tables computed by MATRIX-CHAIN-ORDER for $n = 6$ and the following matrix dimensions:

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

m[3,5] = min
- **m[3,4]+m[5,5] + 15*10*20**
  =750 + 0 + 3000 = 3750
- **m[3,3]+m[4,5] + 15*5*20**
  =0 + 1000 + 1500 = 2500

The tables are rotated so that the main diagonal runs horizontally. Only the main diagonal and upper triangle are used in the m table, and only the upper triangle is used in the s table. 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 9 when computing

$$m[2,5] = \min \begin{cases} m[2,2] + m[3,5] + p_1 p_2 p_5 = 0 + 2500 + 35 \cdot 15 \cdot 20 = 13000\,, \\ m[2,3] + m[4,5] + p_1 p_3 p_5 = 2625 + 1000 + 35 \cdot 5 \cdot 20 = 7125\,, \\ m[2,4] + m[5,5] + p_1 p_4 p_5 = 4375 + 0 + 35 \cdot 10 \cdot 20 = 11375 \end{cases}$$
$$= 7125\,.$$

Example

- **Consider an example with sequence of dimensions <5,2,3,4,6,7,8>**

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

	1	2	3	4	5	6
1	0	30	64	132	226	348
2		0	24	72	156	268
3			0	72	198	366
4				0	168	392
5					0	336
6						0

m[2,5] = min(
m[2,2]+m[3,5] + 2*3*7, (0+198+42)
m[2,3]+m[4,5] + 2*4*7, (24+168+56)
m[2,4]+m[5,5] + 2*6*7, (72+0+84)
)

- Constructing an optimal solution
  - Each entry s[i, j]=k records that the optimal parenthesization of $A_iA_{i+1}...A_j$ splits the product between A_k and A_{k+1}
  - $A_{i..j}$ ➔ $(A_{i..s[i..j]})(A_{s[i..j]+1..j})$

```
PRINT-OPTIMAL-PARENS(s, i, j)
1  if i = j
2      then print "A"_i
3      else print "("
4           PRINT-OPTIMAL-PARENS(s, i, s[i, j])
5           PRINT-OPTIMAL-PARENS(s, s[i, j] + 1, j)
6           print ")"
```

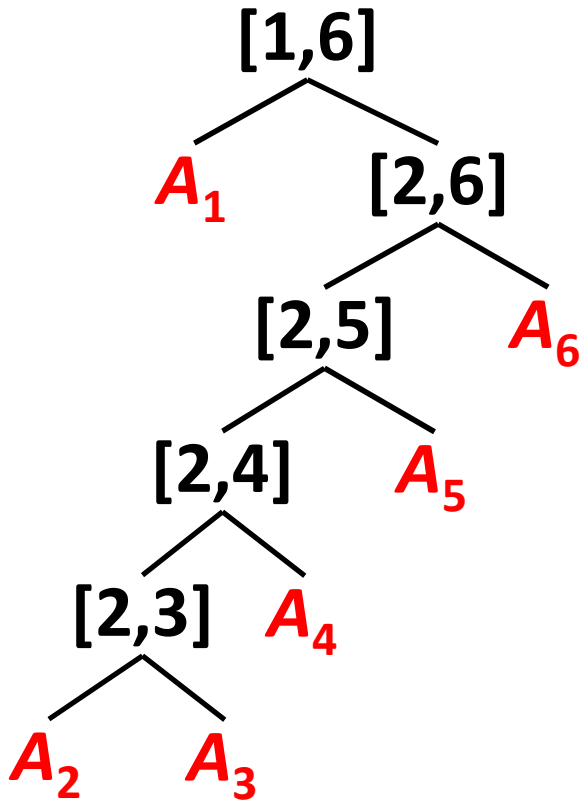
Matrix Chain Multiplication: Example

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

$s[i, j]$ = a value of k that gives the minimum

s	1	2	3	4	5	6
1		1	1	1	1	1
2			2	3	4	5
3				3	4	5
4					4	5
5						5

$A_1((((A_2A_3)A_4)A_5)A_6)$



Complexity

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

- **To fill the entry $m[i, j]$, it needs $\Theta(j-i)$ operations. Hence the execution time of the algorithm is**

$$\sum_{i=1}^{n}\sum_{j=i}^{n}(j-i) = \sum_{j=1}^{n}\sum_{i=1}^{j}(j-i) = \sum_{j=1}^{n}\left[j^2 - \frac{j(j+1)}{2}\right]$$
$$= \sum_{j=1}^{n}\Theta(j^2) = \Theta(n^3)$$

Time: $\Theta(n^3)$

Space: $\Theta(n^2)$

```
MEMOIZED-MATRIX-CHAIN(P)                    <P0, P1,…, Pn>
1.  n = p.length -1;
2.  let m[1..n, 1..n] be a new table;
3.  for i= 1 to n
4.      for j= i to n
5.              m[i, j]= ∞;
6.  return LOOKUP-CHAIN(m, p, 1, n)
```

```
Lookup-Chain(m, P, i, j)
1.    if m[i, j] < ∞  return m[i, j];
2.    if  i==j   m[i, j] = 0
3.    else for k=i  to  j-1
4.       { q= Lookup-Chain(m, P, i, k)
                + Lookup-Chain(m, P, k+1, j) + P_{i-1}P_kP_j ;
5.         if  q < m[i, j]  m[i, j] = q;}
6.     return m[i, j] ; }        time: O(n^3)  space: Θ(n^2)
```

Matrix Chain Multiplication: Quiz!

- **Consider an example with sequence of dimensions <2,3,5,2,4,1>**

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

m[i,j]	1	2	3	4	5
1	0				
2	-	0			
3	-	-	0		
4	-	-	-	0	
5	-	-	-	-	0

s[i,j]	1	2	3	4	5
1	-				
2	-	-			
3	-	-	-		
4	-	-	-	-	

Answer: parenthesize A_1 A_2 A_3 A_4 A_5

Matrix Chain Multiplication: Quiz!

- **Consider an example with sequence of dimensions <2,3,5,2,4,1>**

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

m[i,j]	1	2	3	4	5
1	0	30	42	58	39
2	-	0	30	54	33
3	-	-	0	40	18
4	-	-	-	0	8
5	-	-	-	-	0

s[i,j]	1	2	3	4	5
1	-	1	1	3	1
2	-	-	2	3	2
3	-	-	-	3	3
4	-	-	-	-	4

Answer: parenthesize $(A_1)\ ((A_2)\ ((A_3)((A_4)(A_5))))$

Dynamic Programming - Steps

1. **Characterize the structure of an optimal solution.**
2. **Derive a recursive formula for computing the values of optimal solutions.**
3. **Compute the value of an optimal solution in a bottom-up fashion (top-down is also applicable).**
4. **Construct an optimal solution in a top-down fashion.**

Elements of Dynamic Programming

- **Optimal substructure (a problem exhibits *optimal substructure* if an optimal solution to the problem contains within it optimal solutions to subproblems)**
- **Overlapping subproblems**
- **Reconstructing an optimal solution**
- **Memoization**

Longest Common Subsequence (LCS)

- A subsequence of a sequence/string *S* is obtained by deleting zero or more symbols from *S*. For example, the following are some subsequences of "president": pred, sdn, predent. In other words, the letters of a subsequence of S appear in order in *S*, but they are not required to be consecutive.

- The longest common subsequence problem is to find a maximum length common subsequence between two sequences.

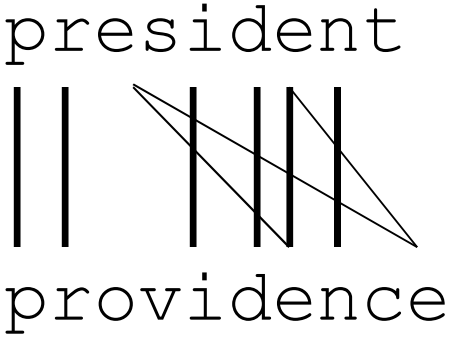
Longest Common Subsequence (LCS)

For instance,

Sequence 1: president

Sequence 2: providence

Its LCS is priden.



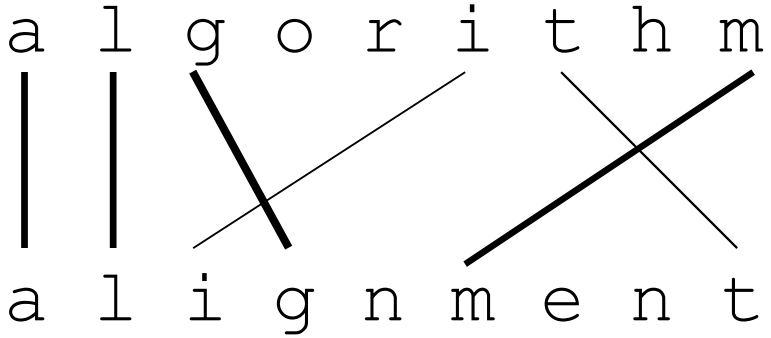
Longest Common Subsequence (LCS)

Another example:

Sequence 1: algorithm

Sequence 2: alignment

One of its LCS is algm.



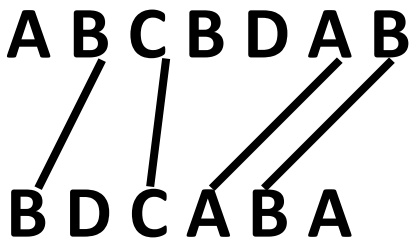
Longest Common Subsequence

Given two sequences $X = \langle x_1, x_2, \ldots, x_m \rangle$ and $Y = \langle y_1, y_2, \ldots, y_n \rangle$ find a maximum-length common subsequence of X and Y.

ex1：Input: ABCBDAB BDCABA

C.S.'s: AB, ABA, BCB, BCAB, BCBA …

Longest: BCAB, BCBA, … Length = 4



ex2 :

vintner

writers

Naive Algorithm

- For every subsequence of X, check whether it's a subsequence of Y .
- Time: $O(n2^m)$.
  - 2^m subsequences of X to check.
  - Each subsequence takes $\Theta(n)$ time to check: scan Y for first letter, for second, and so on.

Step 1: Characterize

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

1. If $x_m = y_n$, then $z_k = x_m = y_n$ and $\langle z_1, z_2, \ldots, z_{k-1} \rangle$ is a LCS of $\langle x_1, x_2, \ldots, x_{m-1} \rangle$ and $\langle y_1, y_2, \ldots, y_{n-1} \rangle$.
2. If $z_k \neq x_m$, then Z is a LCS of $\langle x_1, x_2, \ldots, x_{m-1} \rangle$ and Y.
3. If $z_k \neq y_n$, then Z is a LCS of X and $\langle y_1, y_2, \ldots, y_{n-1} \rangle$.

Step 2: A recursive solution

- **Let $C[i, j]$ be the length of an LCS of the prefixes $X_i = \langle x_1, x_2, \ldots, x_i \rangle$ and $Y_j = \langle y_1, y_2, \ldots, y_j \rangle$, for $1 \le i \le m$ and $1 \le j \le n$. We have :**

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

Step 3: Computing the length of an LCS

```
LCS-Length(X,Y)
1.  m = X.length;
2.  n  = Y.length;
3.  let b[1..m, 1..n] and c[0..m, 0..n] be new tables;
4.  for i= 1 to m do c[i, 0] = 0;
5.  for j= 1 to n do c[0, j] = 0;
6.  for i= 1 to m do
7.       for j= 1 to n do
8.            { if x_i == y_j
9.                 { c[i,j] = c[i-1,j-1]+1;   b[i,j] = "↖"; }
10.           else if c[i-1, j] ≥ c[i, j-1]
11.                { c[i, j] = c[i-1, j];   b[i, j] = "↑"; }
12.           else  { c[i, j] = c[i, j-1];   b[i, j] = "←"; }
13.                }
14. return b, c
```

Step 4: Constructing an LCS

```
Print-LCS(b, X, i, j)
1.  if i==0 or j==0  return;
2.  if b[i,j] == "↖ ";
3.       { Print-LCS(b, X, i-1, j-1);
4.           print x_i;
5.           }
6.  else if b[i,j] == "↑"
7.        Print-LCS(b, X, i-1, j);
8.  else Print-LCS(b, X, i, j-1);
```

Print-LCS(b, X, X.length, Y.length)

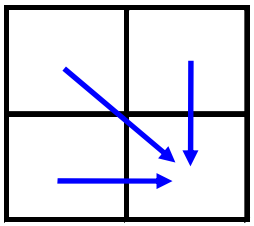
Longest Common Subsequence

$C[i, j] = 0$ if $i = 0$, or $j = 0$

$= C[i-1, j-1] + 1$ if $i, j > 0$ and $x_i = y_j$

$= \max(C[i, j-1], C[i-1, j])$ if $i, j > 0$ and $x_i \neq y_j$

		A	B	C	B	D	A	B
	0	0	0	0	0	0	0	0
B	0	0	1	1	1	1	1	1
D	0	0	1	1	1	2	2	2
C	0	0	1	2	2	2	2	2
A	0	1	1	2	2	2	3	3
B	0	1	2	2	3	3	3	4
A	0	1	2	2	3	3	4	4



Time: O(*mn*)

Space: O(*mn*)

LCS: BCBA

Longest Common Subsequence: Quiz!

$$C[i,j] = 0 \quad \text{if } i = 0, \text{ or } \quad j = 0$$

$$= C[i-1, j-1] + 1 \text{ if } i, j > 0 \text{ and } x_i = y_j$$

$$= \max(C[i, j-1], C[i-1, j]) \text{ if } i, j > 0 \text{ and } x_i \neq y_j$$

String1: ABCDH
String2: AEDHR

LCS: ADH

c[i,j]	A	B	C	D	H
A					
E					
D					
H					
R					

b[i,j]	A	B	C	D	H
A					
E					
D					
H					
R					

Longest Common Subsequence: Quiz!

$$C[i,j] = 0 \quad \text{if } i = 0, \text{ or } \quad j = 0$$

$$= C[i-1, j-1] + 1 \text{ if } i, j > 0 \text{ and } x_i = y_j$$

$$= \max(C[i, j-1], C[i-1, j]) \text{ if } i, j > 0 \text{ and } x_i \neq y_j$$

String1: ABCDH
String2: AEDHR

LCS: ADH

c[i,j]	A	B	C	D	H
A	1	1	1	1	1
E	1	1	1	1	1
D	1	1	1	2	2
H	1	1	1	2	3
R	1	1	1	2	3

b[i,j]	A	B	C	D	H
A	↖	←	←	←	←
E	↑	↑	↑	↑	↑
D	↑	↑	↑	↖	←
H	↑	↑	↑	↑	↖
R	↑	↑	↑	↑	↑

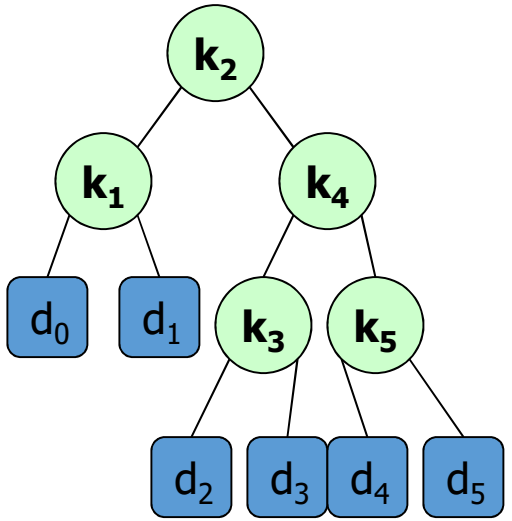
Optimal binary search trees:

- $k=(k_1,k_2,\ldots,k_n)$ of n distinct keys in sorted order ($k_1<k_2<\ldots< k_n$)
- Each key k_i has a probability p_i that a search will be for k_i
- Some searches may fail, so we also have n+1 dummy keys: $d_0,d_1,\ldots,d_n$, where $d_0<k_1$, $k_n<d_n$ and $k_i<d_i<k_{i+1}$ for $i=1,\ldots,n-1$.
- For each dummy key d_i, we have a probability q_i that a search will correspond to d_i .

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

Optimal binary search trees: Example

i	0	1	2	3	4	5
p_i		0.15	0.1	0.05	0.1	0.2
q_i	0.05	0.1	0.05	0.05	0.05	0.1



Expected search cost: 2.8

$$\text{E[Search cost in T]}$$

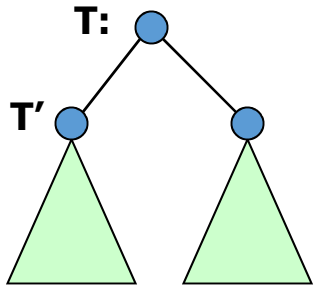
$$= \sum_{i=1}^{n} (\text{depth}_T(k_i) + 1) \cdot p_i + \sum_{i=0}^{n} (\text{depth}_T(d_i) + 1) \cdot q_i$$

$$= 1 + \sum_{i=1}^{n} \text{depth}_T(k_i) \cdot p_i + \sum_{i=0}^{n} \text{depth}_T(d_i) \cdot q_i$$

- Goal: **Given a set of probabilities, want to construct a binary search tree whose expected search cost is smallest .**

  - Step 1: The structure of an optimal BST.

    Consider any subtree of a BST, which must contain contiguous keys $k_i,\ldots,k_j$ for some $1 \le i \le j \le n$ and must also have as its leaves the dummy keys $d_{i-1},\ldots,d_j$.

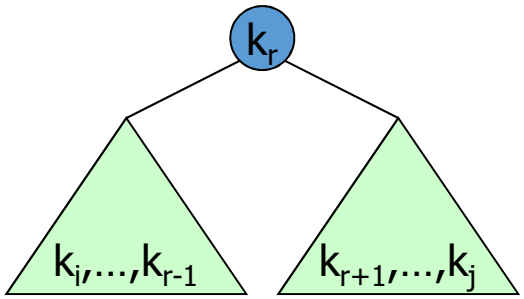


    Optimal-BST has the property of optimal substructure.

Step 2: A recursive solution

- Find an optimal BST for the keys $k_i,\ldots,k_j$, where $j \geq i-1$, $i \geq 1$, $j \leq n$. (When $j=i-1$, there is no actual key, but d_{i-1})
- Define $e[i,j]$ as the expected cost of searching an optimal BST containing $k_i,\ldots,k_j$. Want to find $e[1,n]$.

  For dummy key d_{i-1}, $e[i,i-1]=q_{i-1}$.
- For $j \geq i$, need to select a root k_r among $k_i,\ldots,k_j$.



- For a subtree with keys $k_i,\ldots,k_j$, denote w(i,j) as the sum of probabilities of the subtree for $k_i,\ldots,k_j$ whose root is k_r :

$$w(i,j) = \sum_{\ell=i}^{j} p_\ell + \sum_{\ell=i-1}^{j} q_\ell = w(i,r-1) + p_r + w(r+1,j)$$

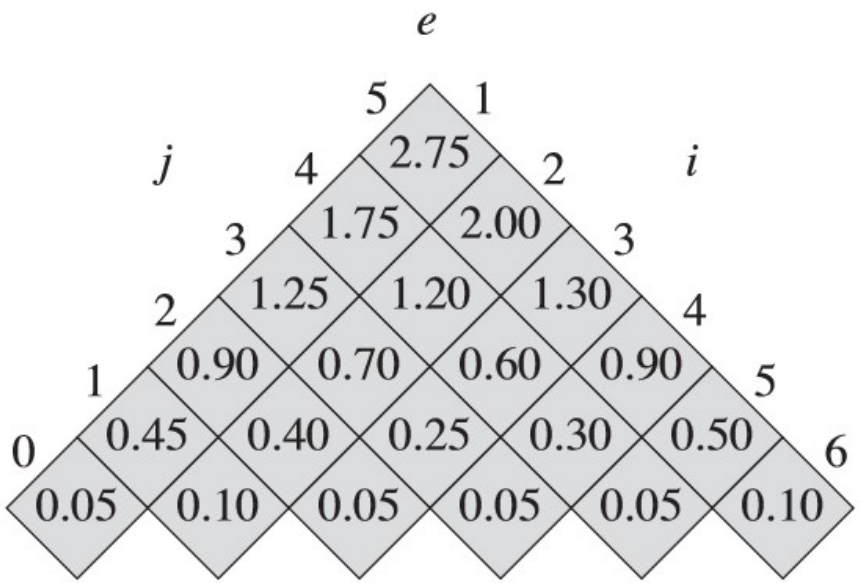
- $e[i,j]=p_r+(e[i,r-1]+w(i,r-1))+(e[r+1,j]+w(r+1,j))$
  $=e[i,r-1]+e[r+1,j]+w(i,j)$
- Thus

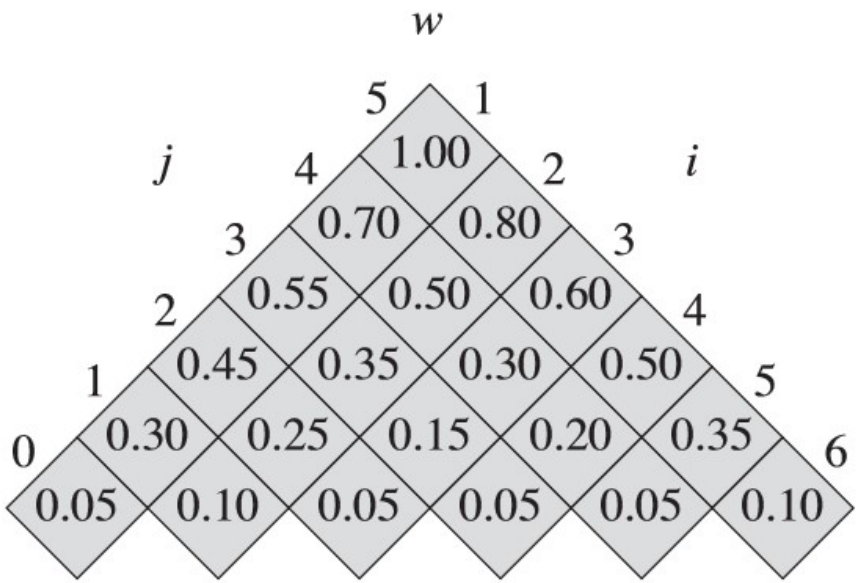
$$e[i,j]=\begin{cases} q_{i-1} & \text{if } j=i-1 \\ \min\limits_{i\le r\le j}\{e[i,r-1]+e[r+1,j]+w(i,j)\} & \text{if } i\le j \end{cases}$$

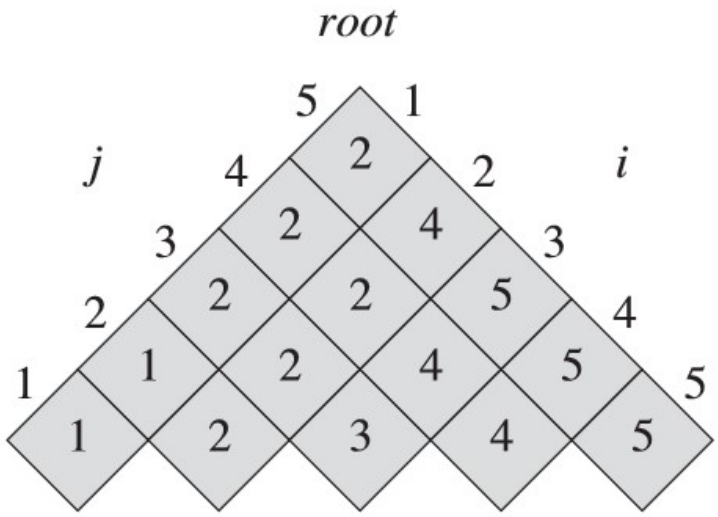
- Step 3: Computing the expected search cost of an optimal BST
- Use a table w[1..n+1,0..n] for w(i,j)'s .
  $w[i,i-1]=q_{i-1}$
  $w[i,j]=w[i,j-1]+p_j+q_j$

```
OPTIMAL-BST(p,q,n)
1. let e[1..n+1, 0..n], w[1..n+1, 0..n] and root[1..n, 1..n] be
   new tables;
2. for i=1 to n+1
3.       e[i,i-1]=q_{i-1} ;
4.       w[i,i-1]=q_{i-1} ;
5. for ℓ=1 to n     // key span
6.       for i=1 to n-ℓ+1
7.             { j=i+ℓ-1;
8.               e[i,j]=∞;
9.               w[i,j]=w[i,j-1]+p_j+q_j ;
10.               for r = i to j
11.                  { t=e[i, r-1]+e[r+1, j] + w[i, j];
12.                    if t < e[i,j]
13.                       e[i,j]=t;
14.                       root[i,j] = r;} }
15.return e and root
```

$\theta(n^3)$


e
j
i
5 1
2.75
4 2
1.75 2.00
3 3
1.25 1.20 1.30
2 4
0.90 0.70 0.60 0.90
1 5
0.45 0.40 0.25 0.30 0.50
0 6
0.05 0.10 0.05 0.05 0.05 0.10


w
j
i
5 1
1.00
4 2
0.70 0.80
3 3
0.55 0.50 0.60
2 4
0.45 0.35 0.30 0.50
1 5
0.30 0.25 0.15 0.20 0.35
0 6
0.05 0.10 0.05 0.05 0.05 0.10


root
j
i
5 1
2
4 2
2 4
3 3
2 2 5
2 4
1 2 4 5
1 5
1 2 3 4 5