DYNAMIC PROGRAMMING

Prof. Zheng Zhang

Harbin Institute of Technology, Shenzhen

FIBONACCI NUMBERS

第1月：1只新生母兔，未成熟，1只

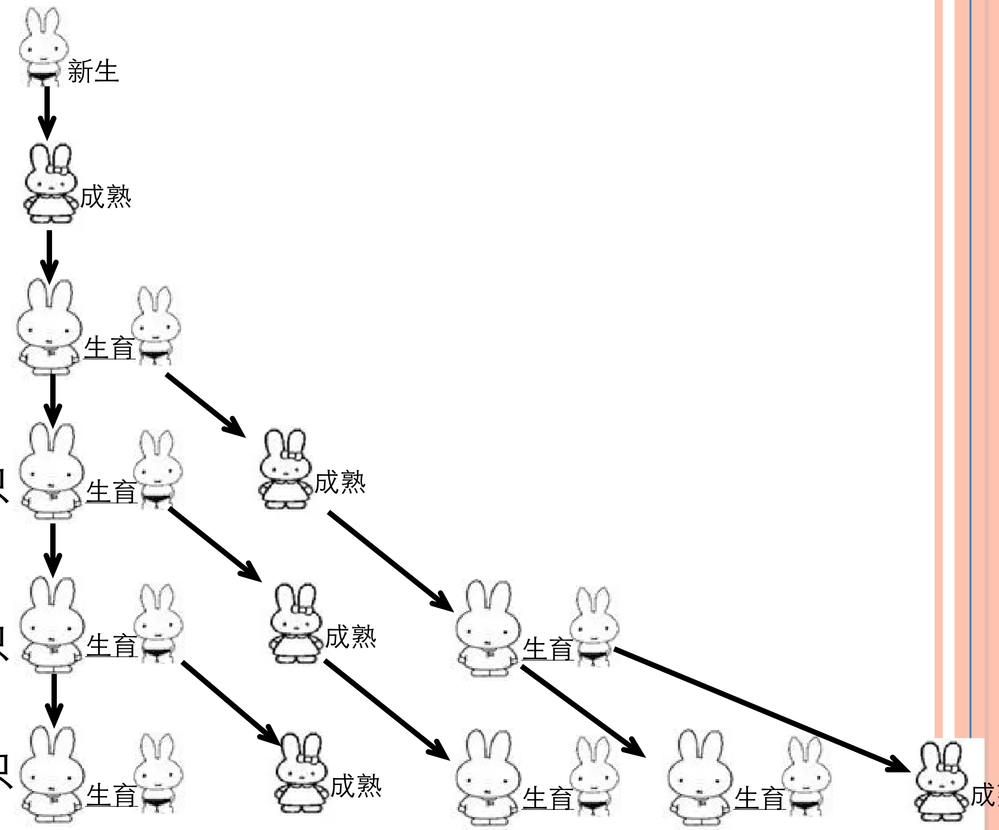
第2月：无母兔生育，1只母兔成熟，1只

第3月：1只母兔生育，无新母兔成熟，共2只

第4月：1只母兔生育，另1只母兔成熟，共3只

第5月：2只母兔生育，另1只母兔成熟，共5只

第6月：3只母兔生育，另2只母兔成熟，共8只



Months and Numbers：1, 1, 2, 3, 5, 8, 13, 21, 34, ...

$$F(n)=\begin{cases}1 & ,n=1\\ 1 & ,n=2\\ F(n-1)+F(n-2) & ,n>2\end{cases}$$

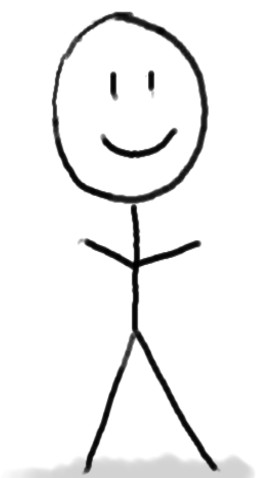
TOP-DOWN

$$F(n) = \begin{cases} 1 & , n = 1 \\ 1 & , n = 2 \\ F(n-1) + F(n-2) & , n > 2 \end{cases}$$

- How to design algorithms and write procedure to solve F(n)?

```
public int fib(int n) {
     if (n < 1) {
         return -1;
     }
     if (n == 1 || n == 2) {
         return 1;
     }
     return fib(n - 1) + fib(n - 2);
 }
```

Complexity: **$O(2^n)$**

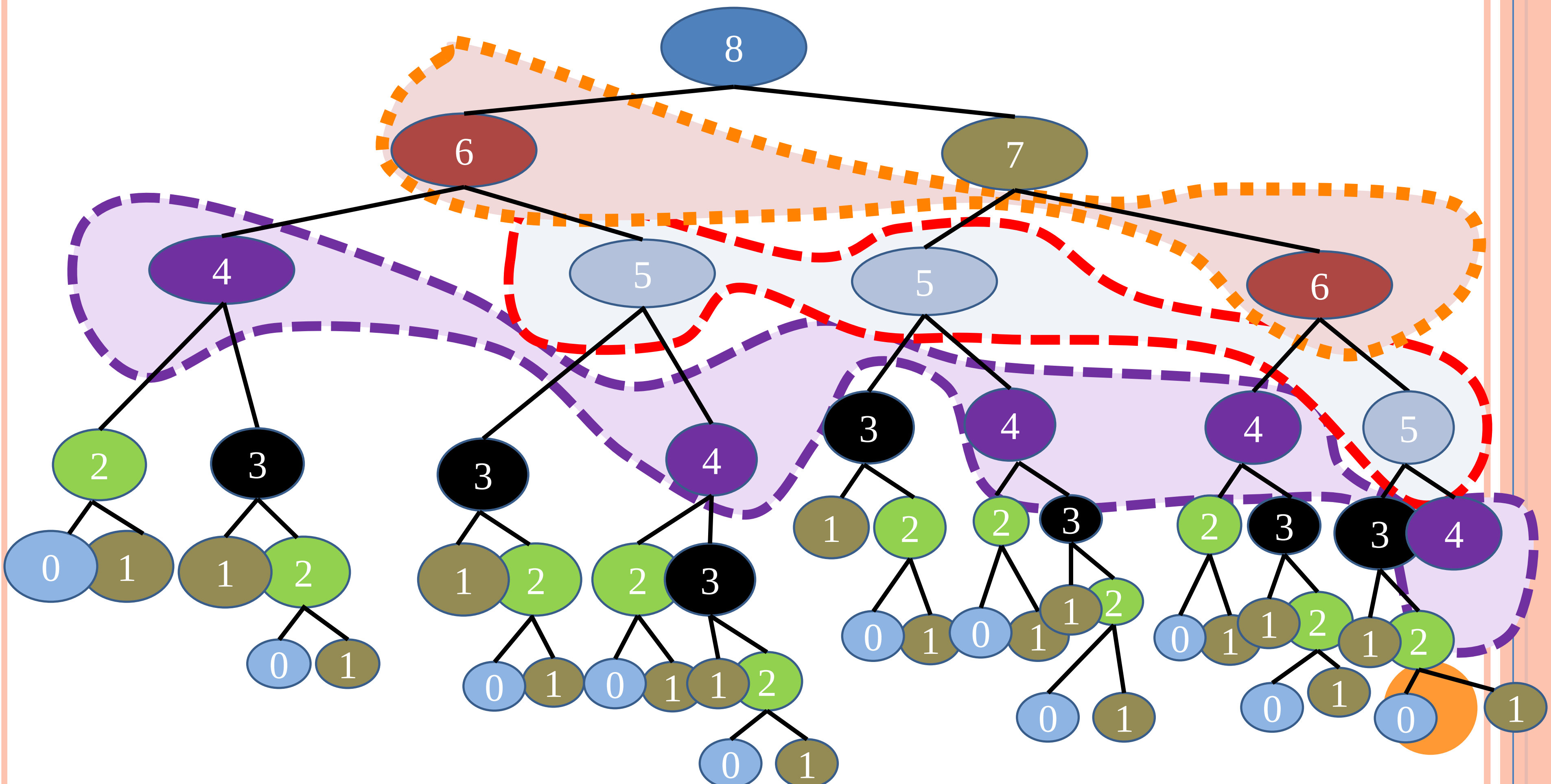


Can we do better ?

Fibonacci Numbers

Complexity: **O(2^n)**

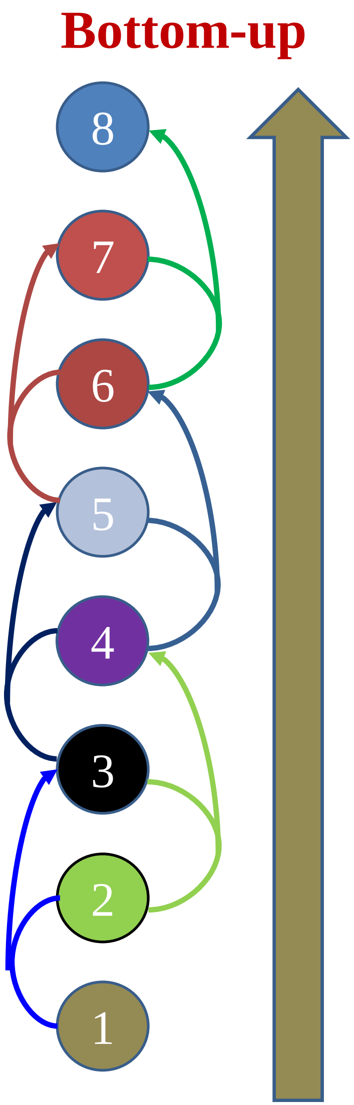
There is a lot of repeated counting. How can it be avoided?



Bottom-Up

$$F(n)=\begin{cases}1 & ,n=1\\ 1 & ,n=2\\ F(n-1)+F(n-2), & n>2\end{cases}$$

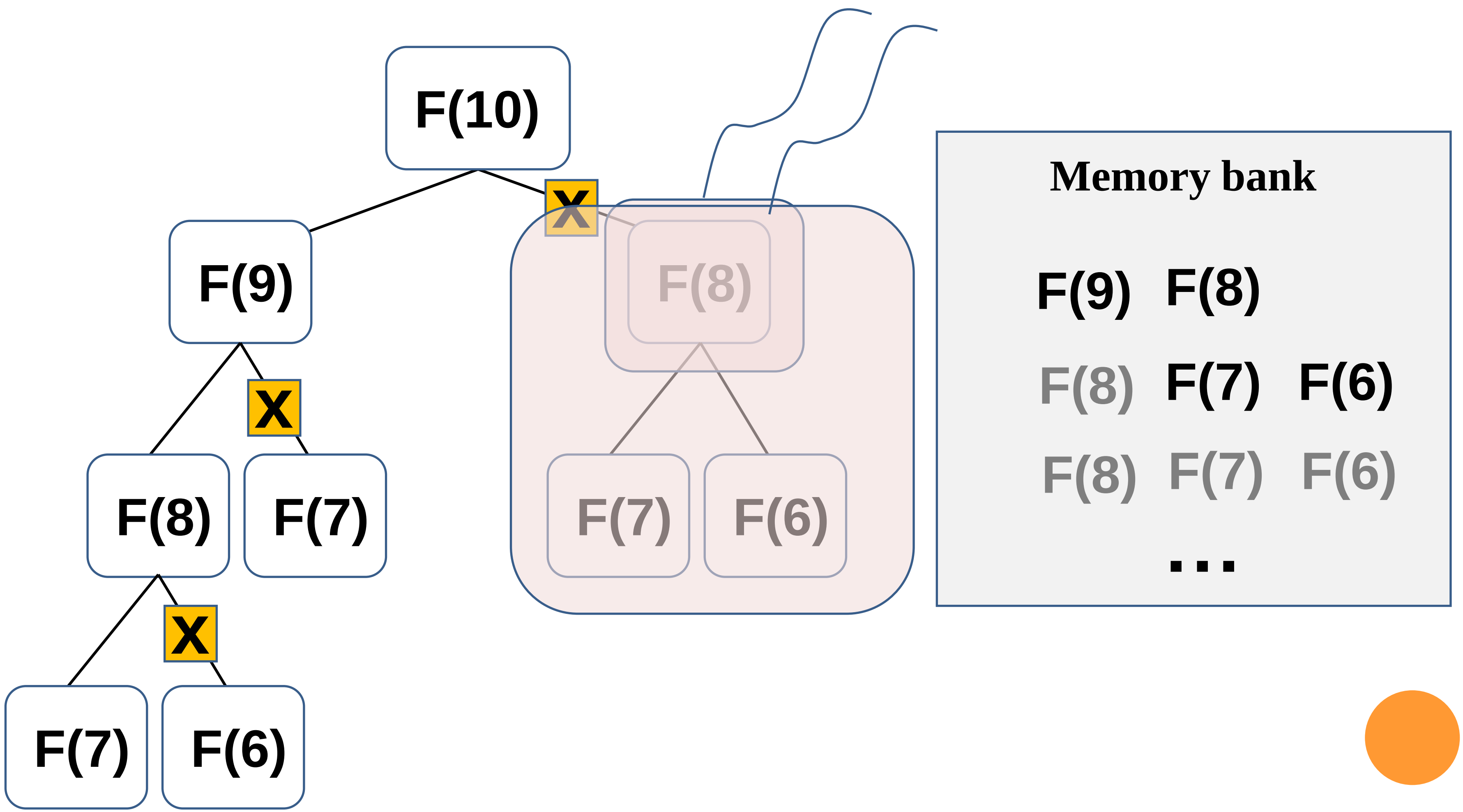
- First compute the smallest problem
  - Record the values of F[0] and F[1].
- Then compute the bigger ones
  - Record the value of F[2].
- ……
- Then compute the bigger ones
  - Record the value of F[n-1].
- Compute the final problem
  - Get F[n].



Complexity: **O(n)**

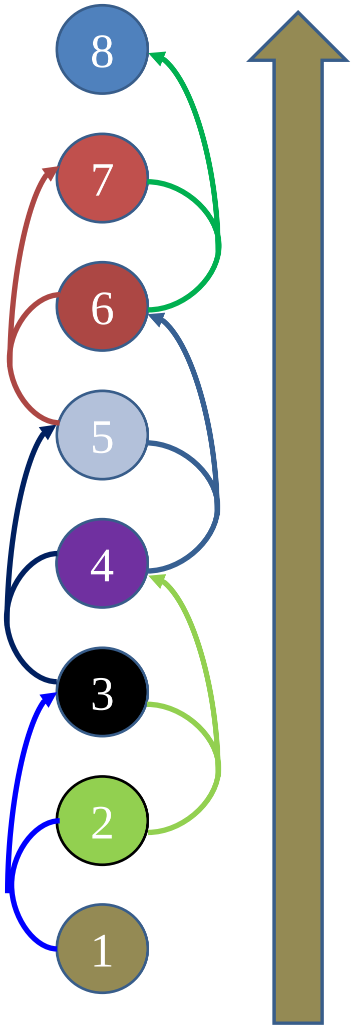
Bottom-Up

$$F(n) = \begin{cases} 1 & , n = 1 \\ 1 & , n = 2 \\ F(n-1) + F(n-2), & n > 2 \end{cases}$$



Bottom-Up

Bottom-up



$$F(n) = \begin{cases} 1 & , n = 1 \\ 1 & , n = 2 \\ F(n-1) + F(n-2), & n > 2 \end{cases}$$

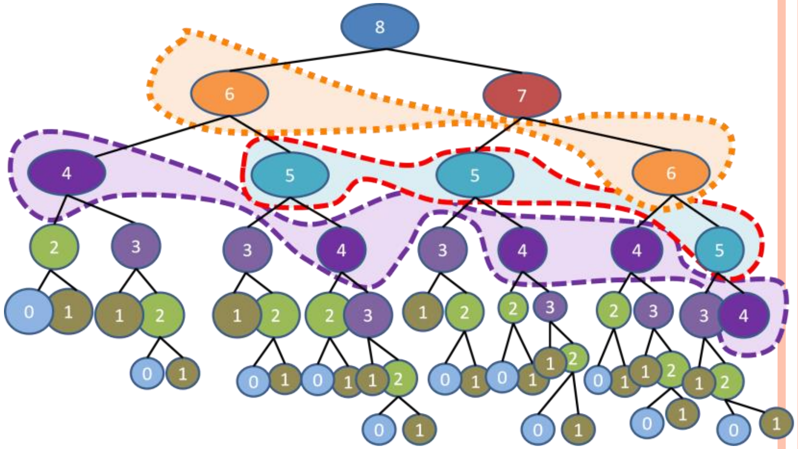
Bottom-up

```
public int fib(int n)
    {
        fib_n_1 = 1, fib_n_2= 1, fib_n=0;
        if (n < 1)
        {
            return -1;
        }
        if (n == 1 || n == 2)
        {
            return 1;
        }
        for (i = 3; i <= n; ++i)
        {
                fib_n = fib_n_1 + fib_n_2;
                fib_n_2 = fib_n_1;
                fib_n_1 = fib_n;
        }
        return fib_n ;
    }
```

Avoid repeat computation and lower complexity O(n) .

FIBONACCI NUMBERS

- Divide-and-Conquer（**无记忆递归**）
  - Independent subproblems
  - double counting common subproblems → inefficient
  - Top-down design algorithm

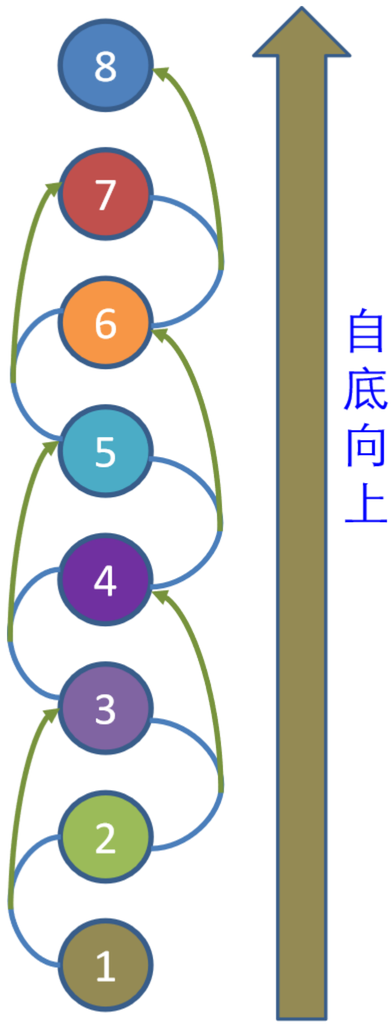


“那些遗忘过去的人注定要重蹈覆辙。”

——乔治•桑塔亚纳

FIBONACCI NUMBERS

- **Dynamic Programming(有记忆的迭代)**
  - Divide into a series of subproblems
  - Solve each subproblem only once, save its results in a table, and **directly access it when you use it later**, without repeating calculations, saving calculation time
  - Bottom-up
- **Applications**
  - Optimization & reused sub-problems



"全局谋划一域、以一域服务全局"

——习近平

OUTLINE

- Introduction to Dynamic Programming
- Famous Examples
  - Matrix-chain Multiplication
  - Longest Common Subsequence
  - Triangle Decomposition of Convex Polygon
  - The Optimal Binary Search Trees

Introduction to Dynamic Programming

Why?

What?

How?

The problem of Divide-and-Conquer

- Treat the subproblems independently.
- If they are dependent, we will calculate redundantly.
- Leads low efficiency.

Optimization Problem

- Given a group of constrains and the cost function, find a solution with *the* optimal value (min or max) in the solution space.
- Lots of the optimization can be divided into subproblems, which are *dependent*, so the solution of the subproblem can be *reused*.

Introduction to Dynamic Programming

Why?

What?

How?

Those who cannot remember the past are doomed to repeat it.

那些遗忘过去的人注定要重蹈覆辙。

-----George Santayana,
The life of Reason,
Book I: Introduction and Reason in Common Sense (1905)

Introduction to Dynamic Programming

Why?

What?

How?

Dynamic Programming

- Divide into subproblem
- Solve each subproblem only once, store the solutions in a list, and access it when the solution is reused
- Bottom-up

“全局谋划一域、以一域服务全局”

——习近平

INTRODUCTION TO DYNAMIC PROGRAMMING

Why?

What?

How?

- **动态规划是一种算法设计的范式（paradigm）或思想，不是解决某一具体问题的具体算法**
- **理查德·贝尔曼（Richard Bellman）在1950年代首次提出了这个名字，当时他正为美国兰德公司工作，主要为美国空军和政府项目服务。**

"It's impossible to use the word, dynamic, in the pejorative sense…I thought dynamic programming was a good name. It was something not even a Congressman could object to." *——理查德·贝尔曼*

- **理查德·贝尔曼（1920年8月26日 - 1984年3月19日），美国应用数学家，美国国家科学院院士。**

INTRODUCTION TO DYNAMIC PROGRAMMING

Why?

What?

How?

Elements of dynamic programming

- Optimal substructure (correctness)
  - A problem *exhibits optimal substructure* if an optimal solution to the problem is contained within its optimal solutions to subproblems.
- Overlapping subproblems (necessity)
  - When a recursive algorithm <u>revisits the same problem</u> over and over again, we say that the optimization problem has overlapping subproblems.

OPTIMAL SUBSTRUCTURE

Why?

What?

How?

- 1) Show that a solution to the problem consists of making a choice.
- 2) Suppose the choices are known.
- 3) Determine which subproblems ensue.
- 4) Cut-and-paste.

INTRODUCTION TO DYNAMIC PROGRAMMING

Why?

What?

How?

The step of dynamic programming:

- 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

PROBLEM DEFINITION

Inputs: Matrix chain $< A_1, A_2, \ldots, A_n >$

Outputs: $A_1 A_2 \ldots A_n$

If ***A*** is a $\boldsymbol{p} \times \boldsymbol{q}$ matrix, and ***B*** a $\boldsymbol{q} \times \boldsymbol{r}$ matrix, then normally we spend ***O*****(p*q*r)** times to compute ***AB*****.**

MOTIVATION

Matrix multiplication fulfill multiplication associativity.

Example:

$$(A_1 A_2 A_3 A_4)$$
$$= (A_1 (A_2 (A_3 A_4)))$$
$$= ((A_1 A_2) (A_3 A_4))$$
$$....$$
$$= (((A_1 A_2) A_3) A_4)$$

$$A_1 \times A_2 \times A_3 \times A_4 = A_1 \times (A_2 \times (A_3 \times A_4))$$
$$= (A_1 \times A_2) \times (A_3 \times A_4)$$
$$....$$
$$= ((A_1 \times A_2) \times A_3) \times A_4$$

Multiplication Order

The complexity depends on the order

- Suppose A_1 a 10×100 matrix, A_2 a 100×5 matrix, A_3 a 5×50 matrix
- $T((A_1 A_2) A_3) = 10 \times 100 \times 5 + 10 \times 5 \times 50 = 7500$
- $T(A_1 (A_2 A_3)) = 100 \times 5 \times 50 + 10 \times 100 \times 50 = 750000$

Solution Space

- Denote the number of different solutions as ***P(n),*** the recursive function of ***P(n)***

$$P(n)=\begin{cases}1 & n=1\\ \sum_{k=1}^{n-1}P(k)P(n-k) & n>1\end{cases}$$

- *P(n) is Catalan number*

$$P(n)=C(n-1)=\frac{1}{n}\binom{2n-2}{n-1}=\Omega(\frac{4^n}{n^{3/2}})$$

$P(n)$ is so big to solute the problem through enumeration methods.

Enumerating won't work

Step 1: Structure of Optimal Solution

- Denote the multiplication $A_i A_{i+1} \ldots A_j$ as $A\,[i{:}\,j]$
- Suppose we cut at k, and get a optimal order

$$(A_1 A_2 \ldots A_n) = ((A_1 \ldots A_k)(A_{k+1} \ldots A_n))$$

- $A\,[1{:}n] = A\,[1{:}k]\;\; A\,[k+1{:}n]$
- Compatible of matrix multiplication
- We can prove $A\,[1{:}k]$ and $A\,[k+1{:}n]$ are optimal order too

Step 1: Structure of Optimal Solution

Multiplication Order

The complexity depends on the order

- Suppose A_1 a 10×100 matrix, A_2 a 100×5 matrix, A_3 a 5×50 matrix
- $T((A_1 A_2) A_3) = 10 \times 100 \times 5 + 10 \times 5 \times 50 = 7500$
- $T(A_1 (A_2 A_3)) = 100 \times 5 \times 50 + 10 \times 100 \times 50 = 750000$

Cost = Cost of $\boldsymbol{A_{i\ldots k}}$ + Cost of $\boldsymbol{A_{k+1\ldots j}}$ + Cost of $\boldsymbol{A_{i\ldots k}} * \boldsymbol{A_{k+1\ldots j}}$

$\boldsymbol{p_{i-1} * p_k}$ $\boldsymbol{p_k * p_j}$

Step 1: Structure of Optimal Solution

Subproblem Overlapping

- $A_1 \times A_2 \times A_3 \times A_4$
  - $(A_1) \times (A_2 \times A_3 \times A_4)$
    - $(A_2 \times A_3)$
    - $(A_3 \times A_4)$
  - $(A_1 \times A_2) \times (A_3 \times A_4)$
    - $(A_1 \times A_2)$
    - $(A_3 \times A_4)$
  - $(A_1 \times A_2 \times A_3) \times (A_4)$
    - $(A_1 \times A_2)$
    - $(A_2 \times A_3)$

STEP 2: RECURSION

Denotation

***m*[*i, j*] *----the minimal cost(times) to calculate A* [*i:j*]**

***m*[*1, n*] *----the minimal cost(times) to calculate A* [*1:n*]**

Recursion cost

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

Step 3: Bottom-Up

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

m[1,1]	***m[1,2]***	***m[1,3]***	***m[1,4]***	***m[1,5]***
	m[2,2]	***m[2,3]***	***m[2,4]***	***m[2,5]***
		m[3,3]	***m[3,4]***	***m[3,5]***
			m[4,4]	***m[4,5]***
				m[5,5]

$$m[2, 4] = min \begin{cases} m[2, 2] + m[3, 4] + p_1 p_2 p_4 \\ m[2,3] + m[4, 4] + p_1 p_3 p_4 \end{cases}$$

$k = 2$ or 3

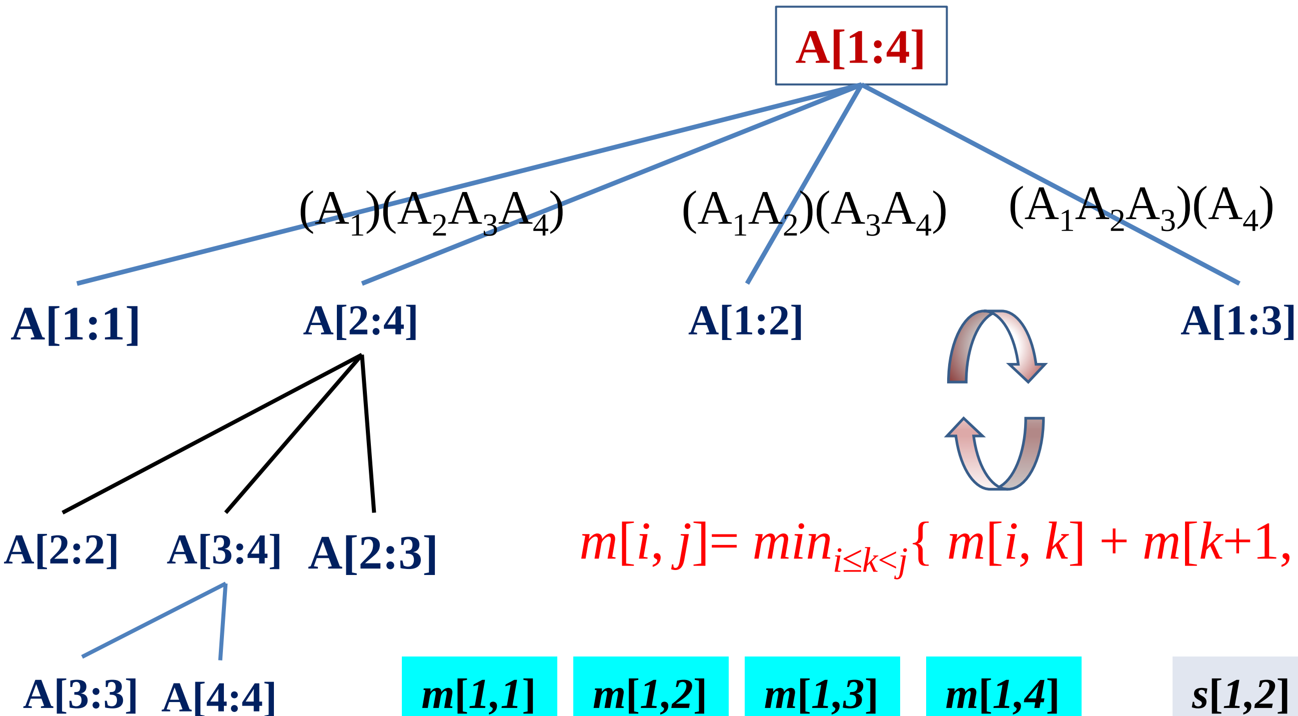
Step 3: Bottom-Up

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

m[1,1]	*m[1,2]*	*m[1,3]*	*m[1,4]*	*m[1,5]*
	m[2,2]	*m[2,3]*	*m[2,4]*	*m[2,5]*
		m[3,3]	*m[3,4]*	*m[3,5]*
			m[4,4]	*m[4,5]*
				m[5,5]

$$m[2, 4] = \min \begin{cases} m[2, 2] + m[3, 4] + p_1\, p_2\, p_4 \\ m[2,3] + m[4, 4] + p_1\, p_3\, p_4 \end{cases}$$

STEP 3: BOTTOM-UP



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

***m*[1,1]**	***m*[1,2]**	***m*[1,3]**	***m*[1,4]**
	***m*[2,2]**	***m*[2,3]**	***m*[2,4]**
		***m*[3,3]**	***m*[3,4]**
			***m*[4,4]**

***s*[1,2]**	***s*[1,3]**	***s*[1,4]**
	***s*[2,3]**	***s*[2,4]**
		***s*[3,4]**

Step 3: Bottom-Up

```
public static void matrixChain(int [] p, int [][] m, int [][] s)
 {
   int n=p.length-1;
   for (int i = 1; i <= n; i++) m[i][i] = 0; /*边界*/
   for (int r = 2; r <= n; r++)  /* 计算第r对角线 */
     for (int i = 1; i <= n–r +1; i++) {
       int j = i+r-1;
       m[i][j] = ∞;
       for (int k = i; k < j; k++) /* 计算m[i,j] */ {
         int t = m[i][k] + m[k+1][j] + p[i-1]*p[k]*p[j];
                 /* q ← m[i, k]+m[k+1, j]+p_{i-1}p_kp_j */
         if (t < m[i][j]) {
          m[i][j] = t;
          s[i][j] = k;}
         }
       }
 }
```

m[1,1]	*m[1,2]*	*m[1,3]*	*m[1,4]*
	m[2,2]	*m[2,3]*	*m[2,4]*
		m[3,3]	*m[3,4]*
			m[4,4]

S[i,j]记录$A_iA_{i+1}\ldots A_j$的最优划分处

Complexity:

It is based on the ternary recurrence of ***r, i, k***. So the time complexity is $O(n^3)$ while the space complexity is $O(n^2)$.

Step 4: Constructing

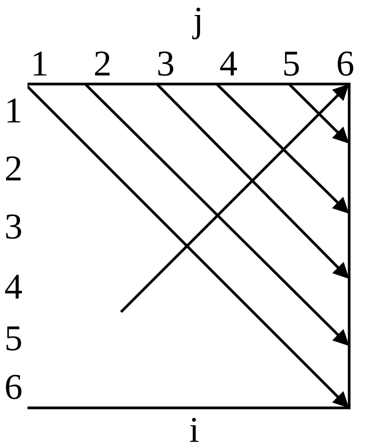
Print-Optimal-Parens(s, i, j)

1. IF j=i THEN Print "A_i";
2. ELSE Print "("
4. Print-Optimal-Parens(s, i, $s[i, j]$)
5. Print-Optimal-Parens(s, $s[i, j]+1$, j)
6. Print ")"

$s[1,2]$	$s[1,3]$	$s[1,4]$
	$s[2,3]$	$s[2,4]$
		$s[3,4]$

EXAMPLE

A1	A2	A3	A4	A5	A6
30×35	35×15	15×5	5×10	10×20	20×25



(a) ***order***

j

i	1	2	3	4	5	6
1	0	15750	7875	9375	11875	15125
2		0	2625	4375	7125	0500
3			0	750	2500	5375
4				0	1000	3500
5					0	5000
6						0

(b) ***m[i][j]***

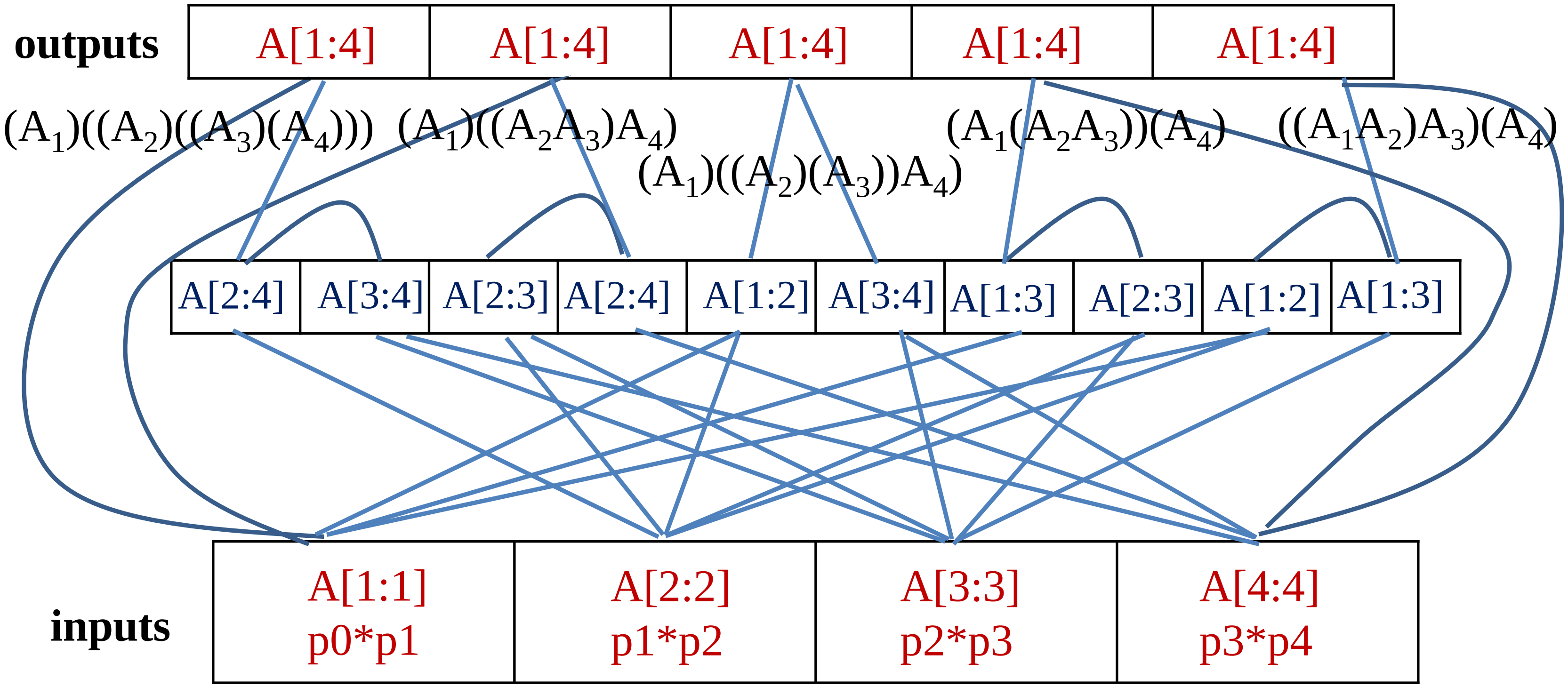
j

i	1	2	3	4	5	6
1	0	1	1	3	3	3
2		0	2	3	3	3
3			0	3	3	3
4				0	4	5
5					0	5
6						0

(c) ***s[i][j]***

$$m[2][5] = \min\begin{cases} m[2][2] + m[3][5] + p_1 p_2 p_5 = 0 + 2500 + 35\times15\times20 = 13000 \\ m[2][3] + m[4][5] + p_1 p_3 p_5 = 2625 + 1000 + 35\times5\times20 = 7125 \\ m[2][4] + m[5][5] + p_1 p_4 p_5 = 4375 + 0 + 35\times10\times20 = 11375 \end{cases}$$

Multiplication Order



$A\ [i{:}\ j] = A_i \ldots A_j$

P(k) = # ways to

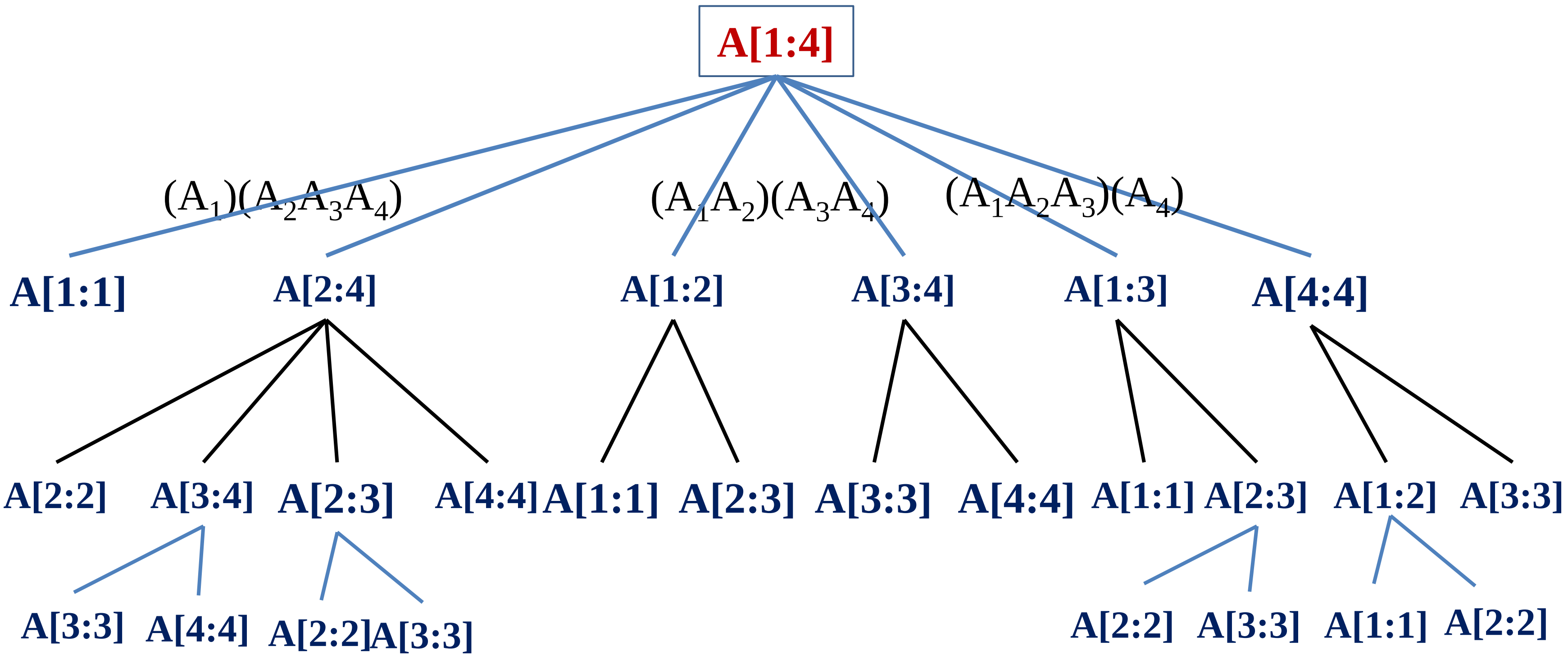
parenthesize *k* matrices

$(A_1 \ldots A_k)(A_k \ldots A_n) = A\ [i{:}\ j]*A[k+1{:}n],\ 1 \leq k < n$

P(1)=1; P(n) =sum $_{1\leq k<n}$ P(k)*P(n-k), n>1

P(n)=C(n-1), where $C(n)=\Omega(4^n/n^{3/2})$

MULTIPLICATION ORDER



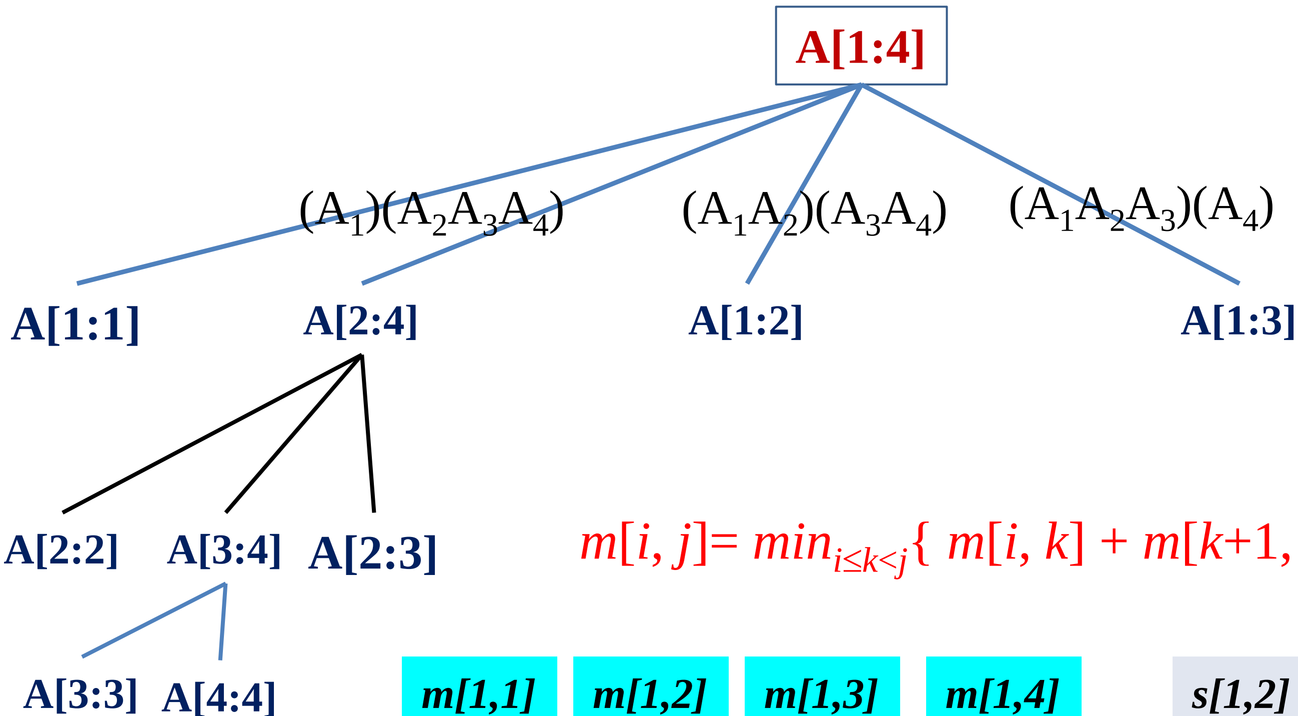
$(A_1 \ldots A_k)(A_k \ldots A_n) = A\,[i{:}\ j] * A[k+1{:}n],\ 1 \le k < n$

$f(1)=1;\ f(n) = \min_{1 \le k < n} \{f(k)+f(n-k)+p_0 p_k p_n),\ n>1$

$T(1)=O(1),\ T(n)=\text{sum}_{1 \le k < n}\,[T(k)+T(n-k)+O(1)]+O(n) = \Omega(2^n)$

$\mathbf{\Omega(2^n)}$

MULTIPLICATION ORDER



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

m[1,1]	*m[1,2]*	*m[1,3]*	*m[1,4]*
	m[2,2]	*m[2,3]*	*m[2,4]*
		m[3,3]	*m[3,4]*
			m[4,4]

s[1,2]	*s[1,3]*	*s[1,4]*
	s[2,3]	*s[2,4]*
		s[3,4]

O(n³) → $O(n^3)$ $O(n^2)$

LONGEST COMMON SEQUENCE

Longest Common Subsequence (LCS)

- Definition
- Characterizing LCS
- Recursive solution
- Bottom-up to computing the length of LCS
- Construct Optimal solution

DEFINITION

Subsequence

Definition: A sequence $\boldsymbol{Z}$=<z_1, z_2,..., z_k> is a subsequence of $\boldsymbol{X}$=<x_1, x_2,..., x_m> if there exists a strictly increasing sequence < i_1, i_2,..., i_k> of indices of $\boldsymbol{X}$ such that for all $j=1,2,\ldots,k$, we have $x_{ij} = z_j$.

Example: **Z**=<A,B,C,D> is a subsequence of **X**=<A,C,B,C,A,D> with corresponding index sequence <1,3,4,6>.

Common subsequence

We say that a sequence $\boldsymbol{Z}$ is a common subsequence of $\boldsymbol{X}$ and $\boldsymbol{Y}$ if $\boldsymbol{Z}$ is a subsequence of both $\boldsymbol{X}$ and $\boldsymbol{Y}$.

The longest-common-subsequence problem (LCS)

Input: $\boldsymbol{X}$ =<x_1, x_2,..., x_m>, $\boldsymbol{Y}$ =<y_1, y_2,..., y_n>.

Output: The common subsequence $\boldsymbol{Z}$ that ***max*(|Z|).**

Example

DNA:

AGCCCTAAGGGCTACCTAGCTT

DNA:

GACAGCCTACAAGCGTTAGCTTG

How similar the two species are?

EXAMPLE

DNA:

AGCCCTAAGGGCTACCTAGCTT

DNA:

GACAGCCTACAAGCGTTAGCTTG

LCS: AGC CTAA GCT TAGCTT

Example

springtime **horseback**

pioneer **snowflake**

maelstrom **heroically**

becalm **scholarly**

Brute-force algorithm:
For every subsequence of X, check whether it's a subsequence of Y.
Time: $\Theta(n2^m)$

Characterizing LCS

The *i*-th ***prefix*** of $\boldsymbol{X}$-----$\boldsymbol{X}_{\text{i}}$

Given $\boldsymbol{X} = \langle x_1, x_2, \ldots, x_m \rangle$, the *i*-th prefix of $\boldsymbol{X}$ is $\boldsymbol{X}_i = \langle x_1, x_2, \ldots, x_i \rangle$.

Example: $\boldsymbol{X} = (A, B, D, C, A)$; $\boldsymbol{X}_1=(A)$, $\boldsymbol{X}_2=(A, B)$, $\boldsymbol{X}_3=(A, B, D)$

<u>Optimal substructure</u> of an LCS

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

$$
\begin{cases}
\text{If } x_m = y_n\text{, then } z_k=x_{\text{m}}=y_{\text{n}} \text{ and } Z_{k-1} \text{ is an LCS of } X_{m-1} \text{ and } Y_{n-1}\ ; \\
\text{If } x_m \neq y_n\text{, then } z_k \neq x_m \text{ implies that } Z \text{ is an LCS of } X_{m-1} \text{ and } Y; \\
\text{If } x_m \neq y_n\text{, then } z_k \neq y_n \text{ implies that } Z \text{ is an LCS of } X \text{ and } Y_{n-1}\ ;
\end{cases}
$$

Recursion Solution

Define $c[i, j]$ as the length of an LCS of the sequences $\boldsymbol{X_i}$ and $\boldsymbol{Y_j}$

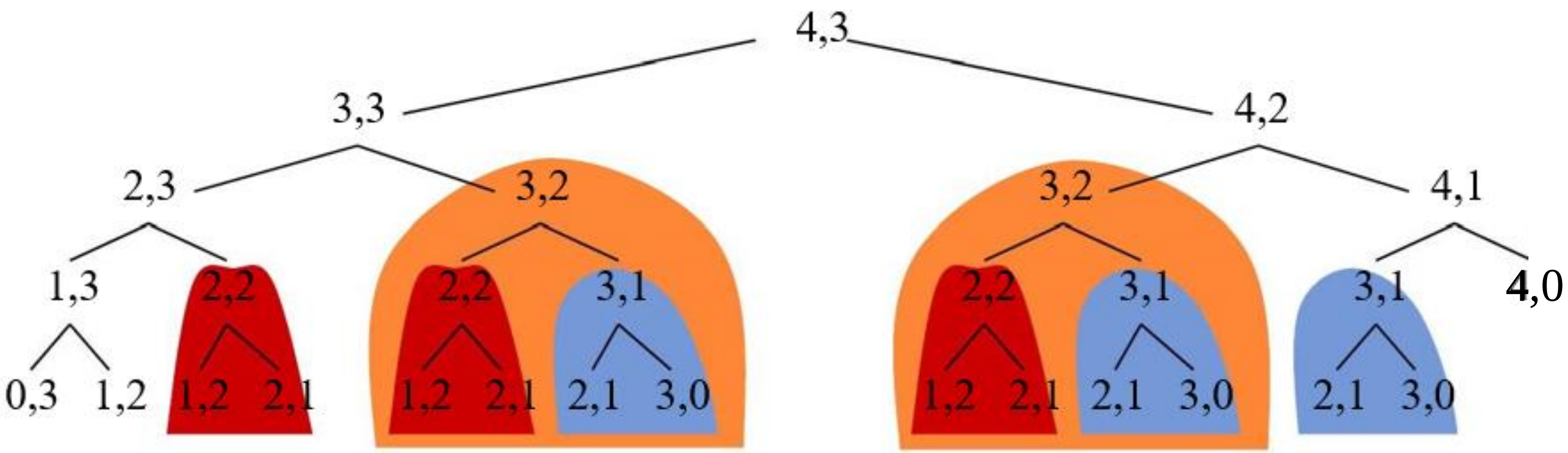
The recursive formula is

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

Recursion Solution

We could write a recursive algorithm based on this formulation.

Try with "bozo", "bat".

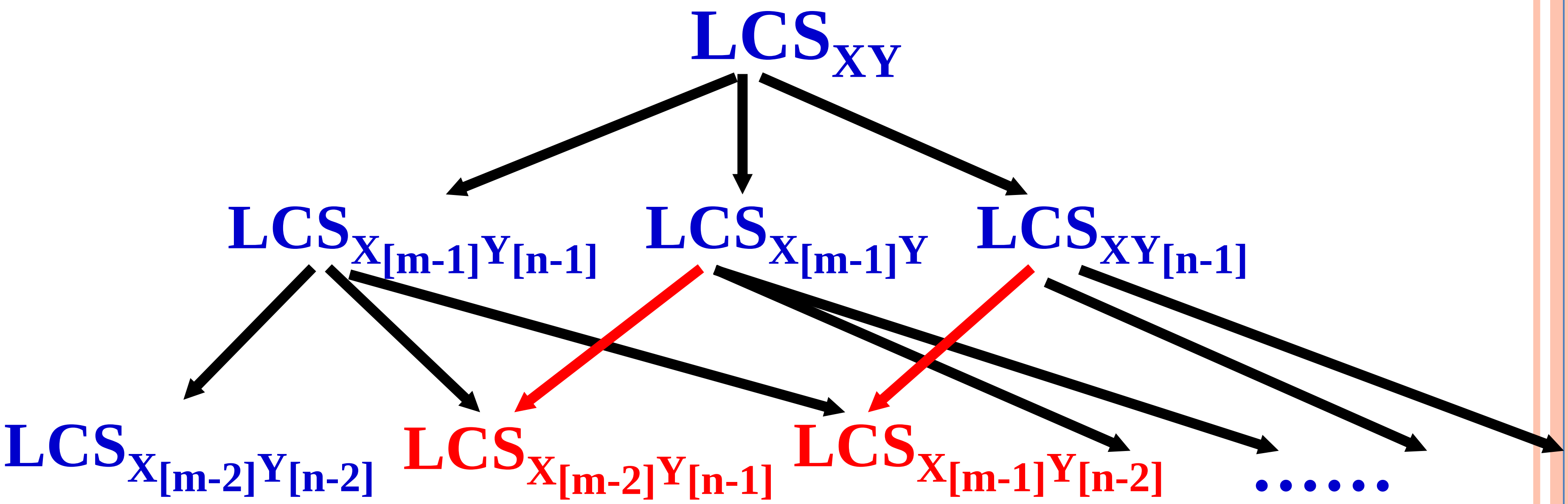


$$c[i, j] = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0 \\ c[i\text{-}1, j\text{-}1] + 1 & \text{if } i, j > 0 \text{ and } x_i = y_j \\ Max(\, c[i, j\text{-}1], c[i\text{-}1, j]\,) & \text{if } i, j > 0 \text{ and } x_i \neq y_j \end{cases}$$

Lots of repeated subproblems.

Instead of recomputing, store in a table.

OVERLAPPING PROBLEM



RECURSION SOLUTION

C[i-1, j-1]	C[i-1,j]
C[i, j-1]	C[i, j]

C[0,0]	C[0,1]	C[0,2]	C[0,3]	C[0,4]
C[1,0]	C[1,1]	C[1,2]	C[1,3]	C[1,4]
C[2,0]	C[2,1]	C[2,2]	C[2,3]	C[2,4]
C[3,0]	C[3,1]	C[3,2]	C[3,3]	C[3,4]

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

Computing The Length of An LCS

Input: Sequences ***X*** and ***Y***

Output: The length of LCS: **C**; Optimal solution information: **B**

LCS-length(*X*, *Y*)

Procedures: **LCS-length(*X*, *Y*)**

```
m←length(X); n←length(Y);
FOR i←1 TO m DO C[i,0]←0;
FOR j←1 TO n DO C[0, j]←0;
FOR i←1 TO m DO
    FOR j←1 TO n DO
        IF x_i = y_j THEN
            C[i,j] ← C[i-1,j-1]+1; B[i,j] ←"↖";
        ELSE IF C[i-1,j] ≥ C[i,j-1] THEN
            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
```

CONSTRUCTING AN LCS

- Initial call is

  PRINT-LCS (*B, X, m, n*)

- *B[i, j]* points to table entry whose subproblem we used in solving LCS of X_i and Y_j .
- When *b*[*i* , *j*] = ↖ , we have extended LCS by one character. So longest common subsequence = entries With ↖ in them.

```
Print-LCS(B, X, i, j)
    1. IF  i=0  OR  j=0  THEN  Return;
    2. IF  B[i, j] = "↖"
           THEN  Print-LCS(B, X, i-1, j-1)
                 Print x_i
    3. ELSE IF   B[i, j]="↑"
           THEN   Print-LCS(B, X, i-1, j)
    4. ELSE   Print-LCS(B, X, i, j-1)
```

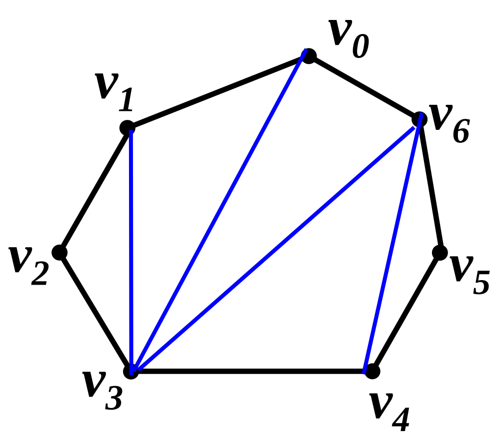
DEMONSTRATION

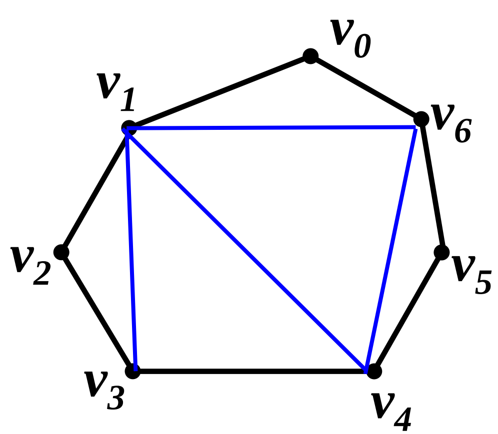
	j	0	1	2	3	4	5	6	7	8	9	10
i		y_j	a	m	p	u	t	a	t	i	o	n
0	x_i		0	0	0	0	0	0	0	0	0	0
1	s	0	0↑	0↑	0↑	0↑	0↑	0↑	0↑	0↑	0↑	0↑
2	p	0	0↑	0↑	1↖	1←	1←	1←	1←	1←	1←	1←
3	a	0	1↖	1←	1↑	1↑	1↑	2↖	2←	2←	2←	2←
4	n	0	1↑	1↑	1↑	1↑	1↑	2↑	2↑	2↑	2↑	3↖
5	k	0	1↑	1↑	1↑	1↑	1↑	2↑	2↑	2↑	2↑	3↑
6	i	0	1↑	1↑	1↑	1↑	1↑	2↑	2↑	3↖	3←	3↑
7	n	0	1↑	1↑	1↑	1↑	1↑	2↑	2↑	3↑	3↑	4↖
8	g	0	1↑	1↑	1↑	1↑	1↑	2↑	2↑	3↑	3↑	4↑

TRIANGLE DECOMPOSITION OF CONVEX POLYGON

Definition

- Convex polygon $P = \{v_0, v_1, ..., v_{n\text{-}1}\}$.
- Triangle decomposition: chords (弦) set T that decompose polygon into disjoint triangle.
- We can improve that in a triangle decomposition of an ***n*** vertices convex polygon, there happened to be n-3 chords and n-2 triangles.





DEFINITION

Weighting function ***w***

For example:

$$w(v_i\, v_j\, v_k) = |v_i\, v_j| + |v_j\, v_k| + |v_i\, v_k|$$

where $|v_i\, v_j|$ is the Euclid distance between v_i and v_j.

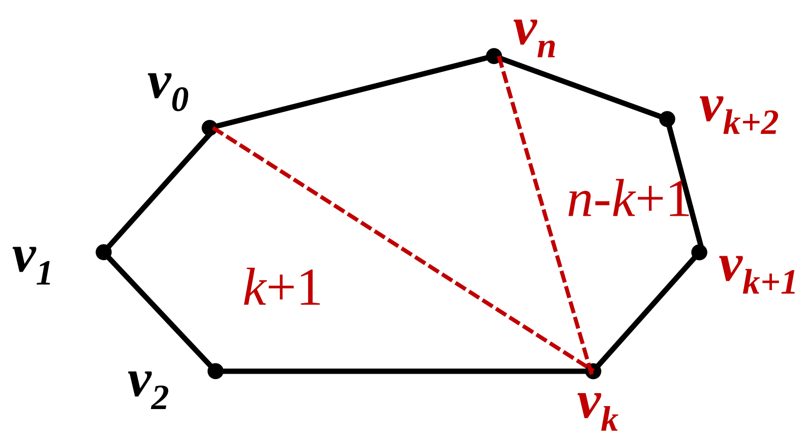
Optimal triangle decomposition

- **Input**: polygon P and weighting function ***w***
- **Output**: triangle decomposition T, to minimize

$$\sum_{S \in S_T} W(s)$$

STRUCTURE OF OPTIMAL SOLUTION

- $P = (v_0, v_1, \ldots, v_n)$ is a $n+1$ vertices polygon
- T_p is an optimal triangle decomposition, v_k is the decomposition point.



- The structure of optimal solution

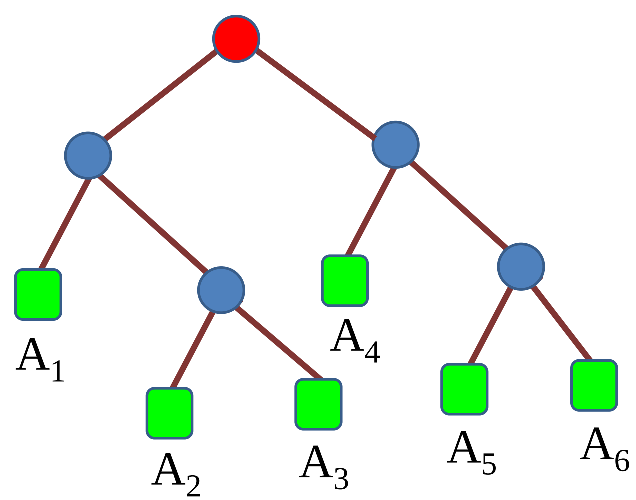
$$T_P = T(v_0, \ldots, v_k) \cup T(v_k, \ldots, v_n) \cup \{v_0v_k, v_kv_n, v_0v_n\}$$

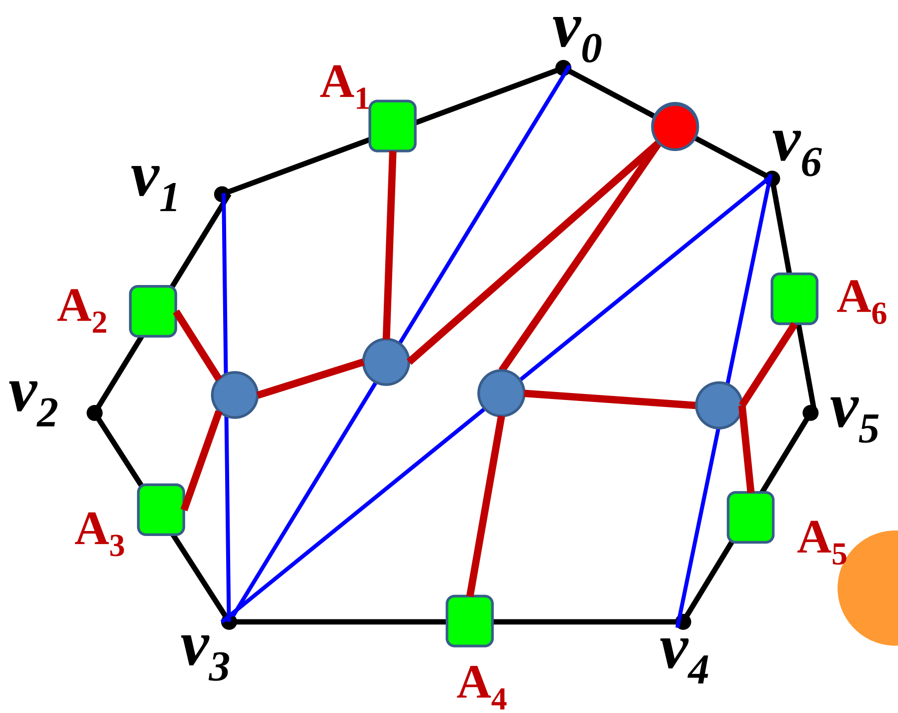
Triangular Decomposition and Matrix-Chain-Order

A matrix chain is corresponding to a binary tree.

For example:

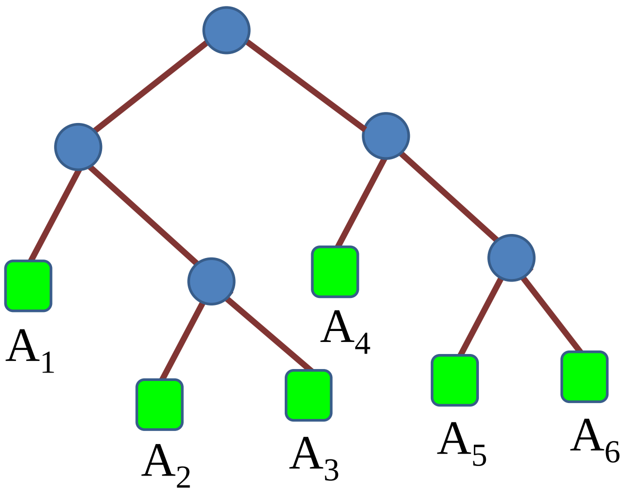
$((A_1(A_2A_3))\ (A_4(A_5A_6)))$ is converted to a binary tree.

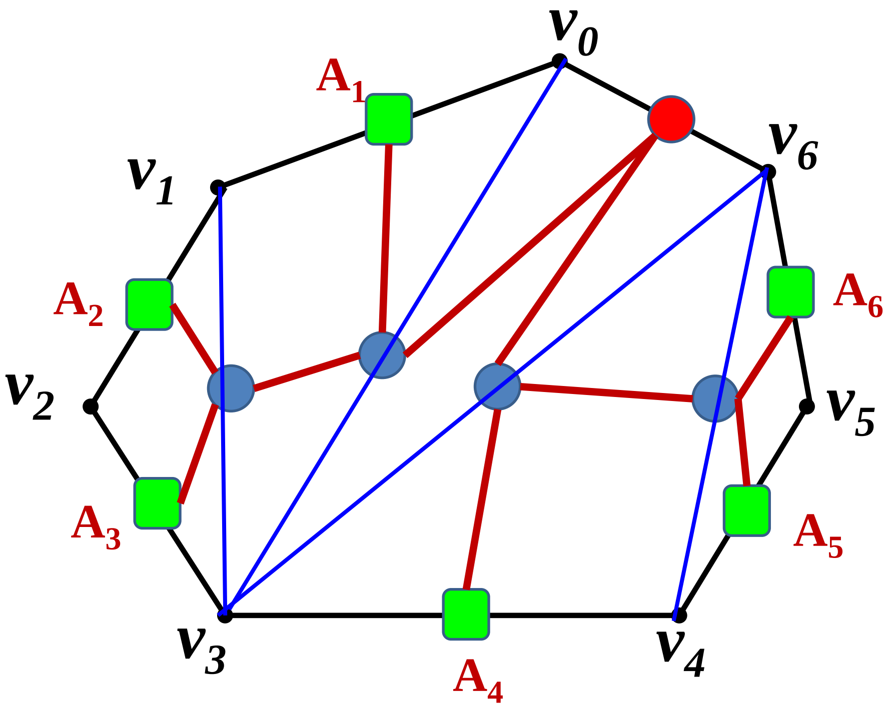




CONSTRUCT OPTIMAL SOLUTION

- It is coordinate with the Matrix-chain-Order.
- By modifying Matrix-chain-order, we can construct optimal triangle decomposition.





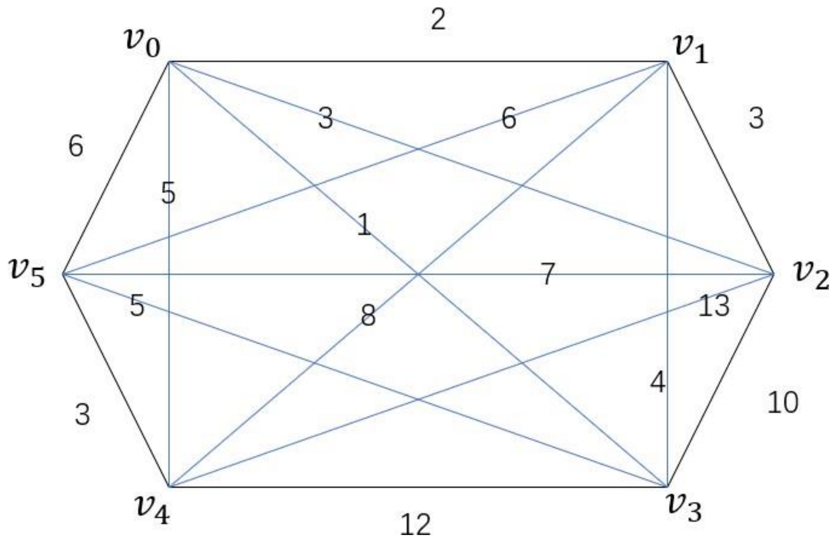
Recursive Solution

$$t[1][n] = \begin{cases} 0 & n = 1 \\ \min\limits_{1 \le k < n}\{t[1][k] + t[k+1][n] + w(v_0 v_k v_n)\} & n > 1 \end{cases}$$

$$t[i][j] = \begin{cases} 0 & i = j \\ \min\limits_{i \le k < j}\{t[i][k] + t[k+1][j] + w(v_{i-1} v_k v_j)\} & i > j \end{cases}$$

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

DEMONSTRATION



g[ ][ ]

	0	1	2	3	4	5
0	0	2	3	1	5	6
1	2	0	3	4	8	6
2	3	3	0	10	13	7
3	1	4	10	0	12	5
4	5	8	13	12	0	3
5	6	6	7	5	3	0

m[ ][ ]

	1	2	3	4	5
1	0				
2		0			
3			0		
4				0	
5					0

s[ ][ ]

	1	2	3	4	5
1	0				
2		0			
3			0		
4				0	
5					0

DEMONSTRATION

Three vertices

$$i=1, j=2: \{v_0, v_1, v_2\}$$
$$k=1: m[1][2]=\min\{m[1][1]+m[2][2]+w(v_0v_1v_2)\}=8$$
$$i=2, j=3: \{v_1, v_2, v_3\}$$
$$k=2: m[2][3]=\min\{m[2][2]+m[3][3]+w(v_1v_2v_3)\}=17$$
$$i=3, j=4: \{v_2, v_3, v_4\}$$
$$k=3: m[3][4]=\min\{m[3][3]+m[4][4]+w(v_2v_3v_4)\}=35$$
$$i=4, j=5: \{v_3, v_4, v_5\}$$
$$k=4: m[4][5]=\min\{m[4][4]+m[5][5]+w(v_3v_4v_5)\}=20$$

m[ ][ ]

	1	2	3	4	5
1	0	8			
2		0	17		
3			0	35	
4				0	20
5					0

s[ ][ ]

	1	2	3	4	5
1	0	1			
2		0	2		
3			0	3	
4				0	4
5					0

DEMONSTRATION

Four vertices

$$i=1, j=3: \{v_0, v_1, v_2, v_3\}$$

$$m[1][3]=\min\begin{cases} k=1, m[1][1]+m[2][3]+w(v_0v_1v_3)=24 \\ k=2, m[1][2]+m[3][3]+w(v_0v_2v_3)=22 \end{cases};$$

$$i=2, j=4: \{v_1, v_2, v_3, v_4\}$$

$$m[2][4]=\min\begin{cases} k=2, m[2][2]+m[3][4]+w(v_1v_2v_4)=59 \\ k=3, m[2][3]+m[4][4]+w(v_1v_3v_4)=41 \end{cases};$$

$$i=3, j=5: \{v_2, v_3, v_4, v_5\}$$

$$m[3][5]=\min\begin{cases} k=3, m[3][3]+m[4][5]+w(v_2v_3v_5)=42 \\ k=4, m[3][4]+m[5][5]+w(v_2v_4v_5)=58 \end{cases};$$

m[ ][ ]

	1	2	3	4	5
1	0	8	22		
2		0	17	41	
3			0	35	42
4				0	20
5					0

s[ ][ ]

	1	2	3	4	5
1	0	1	2		
2		0	2	3	
3			0	3	3
4				0	4
5					0

DEMONSTRATION

Five vertices

$$i=1, j=4: \{v_0, v_1, v_2, v_3, v_4\}$$

$$m[1][4]=\min\begin{cases} k=1, m[1][1]+m[2][4]+w(v_0v_1v_4)=56 \\ k=2, m[1][2]+m[3][4]+w(v_0v_2v_4)=64; \\ k=3, m[1][3]+m[4][4]+w(v_0v_3v_4)=40 \end{cases}$$

$$i=2, j=5: \{v_1, v_2, v_3, v_4, v_5\}$$

$$m[2][5]=\min\begin{cases} k=2, m[2][2]+m[3][5]+w(v_1v_2v_5)=58 \\ k=3, m[2][3]+m[4][5]+w(v_1v_3v_5)=52; \\ k=4, m[2][4]+m[5][5]+w(v_1v_4v_5)=58 \end{cases}$$

m[ ][ ]

	1	2	3	4	5
1	0	8	22	40	
2		0	17	41	52
3			0	35	42
4				0	20
5					0

s[ ][ ]

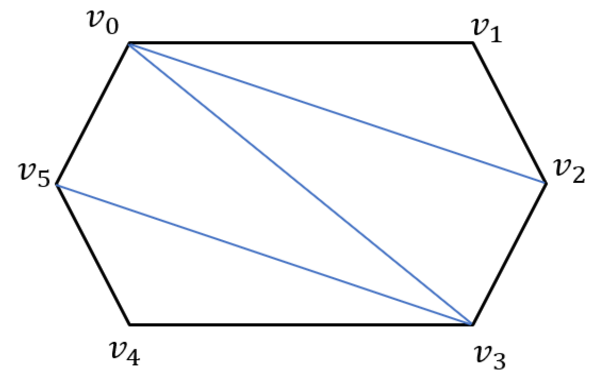
	1	2	3	4	5
1	0	1	2	3	
2		0	2	3	3
3			0	3	3
4				0	4
5					0

DEMONSTRATION

Six vertices

$i=1, j=5: \{v_0, v_1, v_2, v_3, v_4, v_5\}$

$$m[1][4] = \min \begin{cases} k=1, m[1][1]+m[2][5]+w(v_0 v_1 v_5)=66 \\ k=2, m[1][2]+m[3][5]+w(v_0 v_2 v_5)=66 \\ k=3, m[1][3]+m[4][5]+w(v_0 v_3 v_5)=54 \\ k=4, m[1][4]+m[5][5]+w(v_0 v_4 v_5)=54 \end{cases};$$



m[ ][ ]

	1	2	3	4	5
1	0	8	22	40	54
2		0	17	41	52
3			0	35	42
4				0	20
5					0

s[ ][ ]

	1	2	3	4	5
1	0	1	2	3	3
2		0	2	3	3
3			0	3	3
4				0	4
5					0