Dynamic Programming

Liu Ziwei

Reference: Computer Algorithms: Introduction to Design and Analysis, 3rd Ed, by Sara Basse and Allen Van Gelder. Setions *10.1, 10.2 & 10.3*

Sudoku

	C1	C2	C3	C4	C5	C6	C7	C8	C9
R1	9	6					7		8
R2	8					4	3		
R3	1			5					
R4							1	7	6
R5	2				9	3			5
R6	7		8						
R7			7		3	2		4	
R8	3	8	2	1		5	6		
R9		4	1			9	5	2	

Outline

1. Concepts of dynamic programming
2. Longest common subsequence
3. Chain matrix multiplication
4. 0/1 Knapsack problem

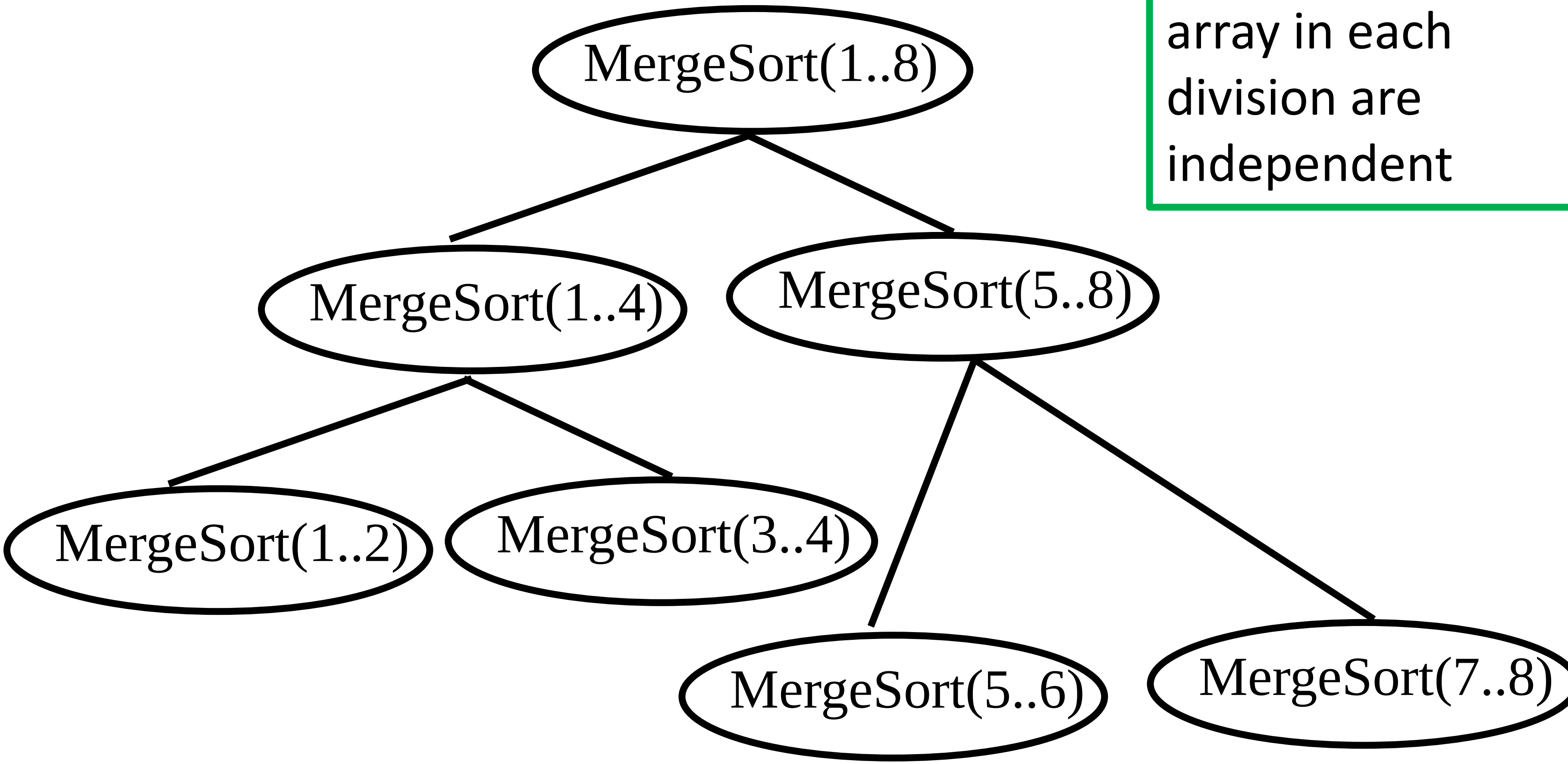
Concepts of Dynamic Programming

What is Dynamic Programming?

- It is a problem solving paradigm
- To a certain extent, it is similar to divide-and-conquer
- What do we do in divide-and-conquer?
  - Divide a problem into *independent* subproblems
  - Solve each subproblem recursively
  - Combine the solutions to subproblems into a solution for the given problem
  - Example: MergeSort

Example: MergeSort(8)

The 2 half sections of the array in each division are independent



Overlapping Subproblems

What is Dynamic Programming?

- Dynamic programming:
  - Divide a problem into ***overlapping*** subproblems
  - Solve each subproblem recursively
  - Combine the solutions to subproblems into a solution for the given problem
  - ***Do not compute the answer to the same subproblem more than once***
  - Example: computing Fibonacci numbers

Fibonacci seqence

The Fibonacci sequence is defined recursively as:

$$F_n = F_{n-1} + F_{n-2} \quad \text{for } n \geq 2$$

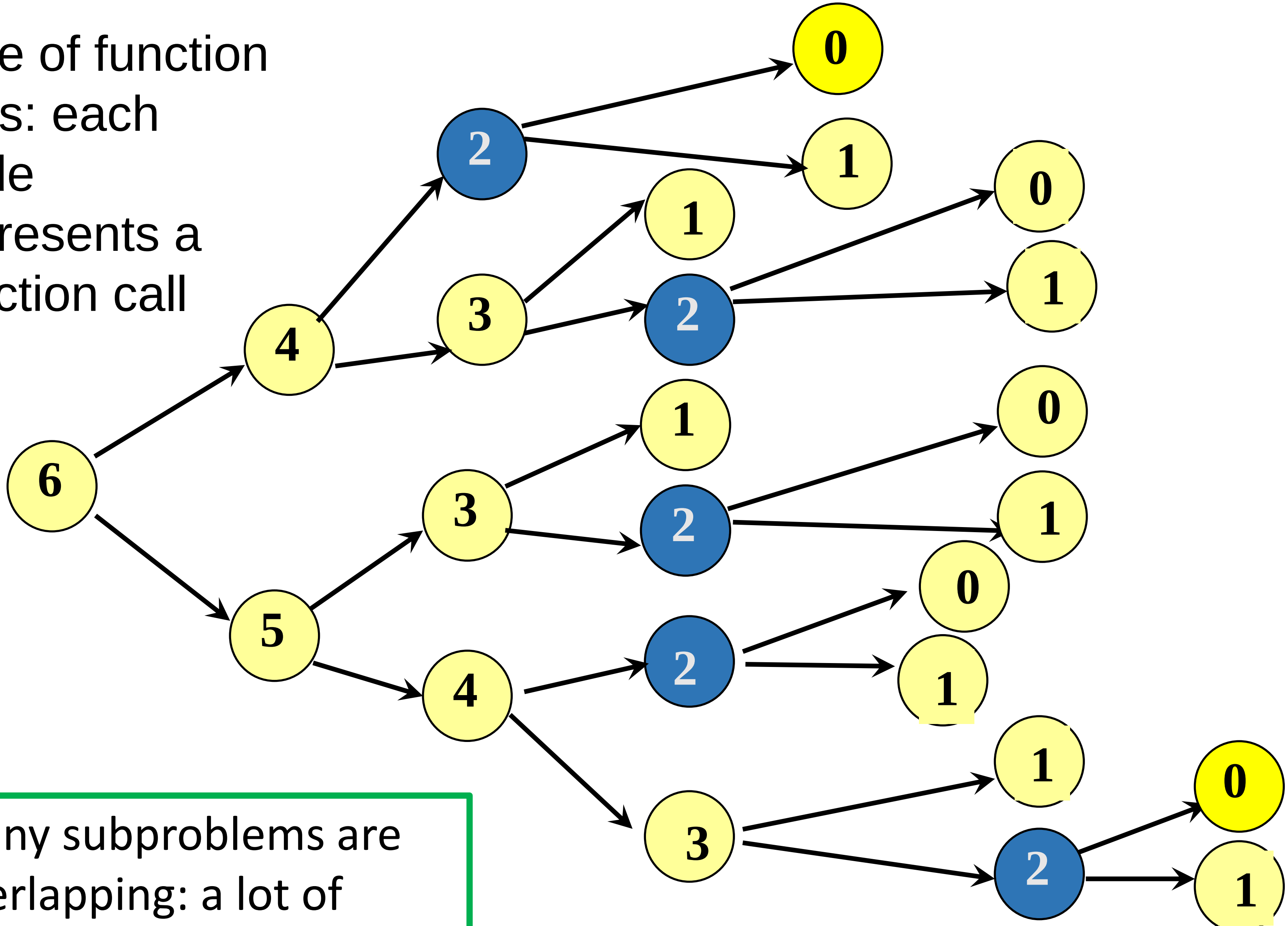
$$F_0 = 0, F_1 = 1$$

This series occurs frequently in algorithm analysis.

<u>Divide-and-conquer: Recursive Fibonacci function</u>

```
int fib(n)
  {      if (n == 0 || n == 1)  return n;
         else return fib(n - 1) + fib(n - 2);
  }
```

Tree of function calls: each node represents a function call



Many subproblems are overlapping: a lot of recomputation

- Example of repetition is given in the shaded nodes
- Notice that this is a full binary tree up to depth 3 (i.e. n/2)
- The deepest level is 5 (i.e. n-1)
- The number of recursive calls R is such that

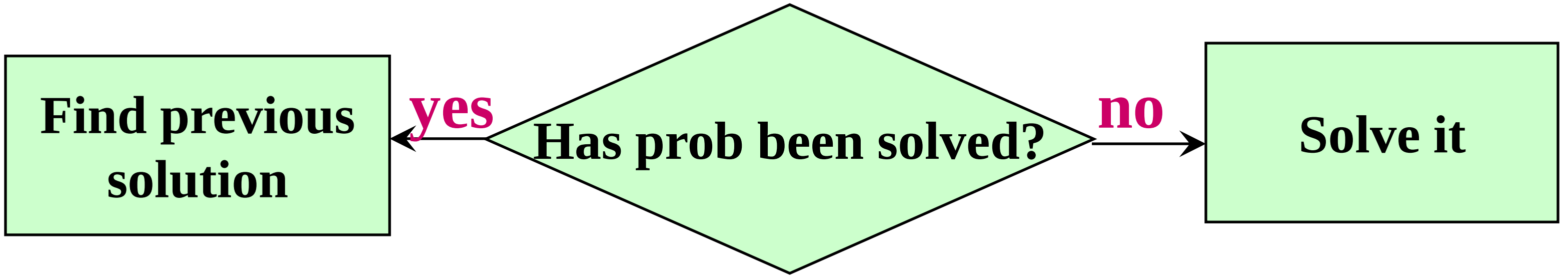
$$2^4 - 1 < R < 2^6 - 1$$

In general,

$$2^{\frac{n}{2}+1} - 1 < R < 2^n - 1$$

So this is an exponential time algorithm: $O(2^n)$

- The main feature of dynamic programming is that it replaces an exponential-time computation by a polynomial-time computation
- It is done by this: Memorize the solutions and do not recompute
- Recall that a DFS on a graph only explores edges to undiscovered vertices, and it checks the other edges.
- This strategy may be applied to only solve unsolved subproblems, and checks and retrieves solutions to the solved subproblems



Dynamic Programming
(Top Down)

Dynamic programming (Top Down)

1. Formulate the problem P in terms of smaller versions of the problem (recursively), say, Q_1, Q_2, …
2. Turn this formulation into a recursive function to solve problem P
3. Use a dictionary to store solutions to subproblems
4. In the recursive function to solve P
   - ❖ Before any recursive call, say on subproblem Q_i, check the dictionary to see if a solution for Q_i has been stored
     - If no solution has been stored, make the recursive call
     - Otherwise, retrieve the stored solution
   - ❖ Just before returning the solution for P, store the solution in the dictionary - memorization

The top-down approach

A dynamic programming version of fib(n)

```
int fibDP(n)
{       int f1, f2;
        if (n == 0 || n == 1)  {
            store(Soln, n, n);
            return n;    }
        else {
            if (not member(Soln, n - 1))
                    f1 = fibDP(n - 1);
            else f1 = retrieve(Soln, n - 1);
```

`Store`, **`member`**, **`retrieve`** are all methods of the Dictionary

```
        if (not member(Soln, n - 2))
                f2 = fibDP(n - 2);
        else f2 = retrieve(Soln, n - 2);

        f1 += f2;
        store(Soln, n, f1);
        return f1;      }
}
```

- Before calling fibDP, the dictionary Soln has to be initialized. E.g.

	0	1	2	3	4	5	6
Soln	-1	-1	-1	-1	-1	-1	-1

- member(Dictionary, j):

  return Dictionary[j] <> -1

- store(Dictionary, j, s):

  Dictionary[j] = s

- retrieve(Dictionary, j):

  return Dictionary[j]

```
int fibDP(n)
{  int f1, f2;
   if (n == 0 || n == 1)  {
        store(Soln, n, n);
        return n;    }
   else {
        if (not member(Soln, n - 1))
             f1 = fibDP(n - 1);
        else f1 = retrieve(Soln, n - 1);
        if (not member(Soln, n - 2))
             f2 = fibDP(n - 2);
        else f2 = retrieve(Soln, n - 2);

        f1 += f2;
        store(Soln, n, f1);
        return f1;      }
}
```

Complexity: O(n)

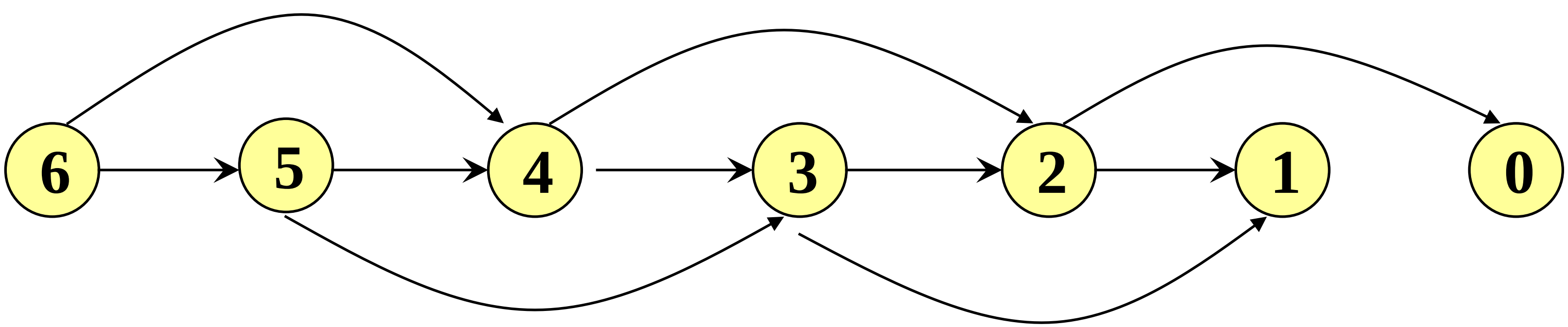
The total computational time in each function call, excluding that of the calls to fibDP() on subproblems is bounded by a constant. The total computational cost is thus proportional to the number of calls to fibDP() when solving fibDP(n) – n+1 times. So O(n).

Dynamic Programming (Bottom Up)

Dynamic programming (Bottom Up)

Subproblem graphs

- For a recursive algorithm A, the subproblem graph for A is a directed graph whose vertices are the instances for this problem. The directed edges (I, J) for all pairs that indicate: if A is invoked on problem I, it makes a recursive call directly on instance J.
- E.g. the subproblem graph for fib(6):

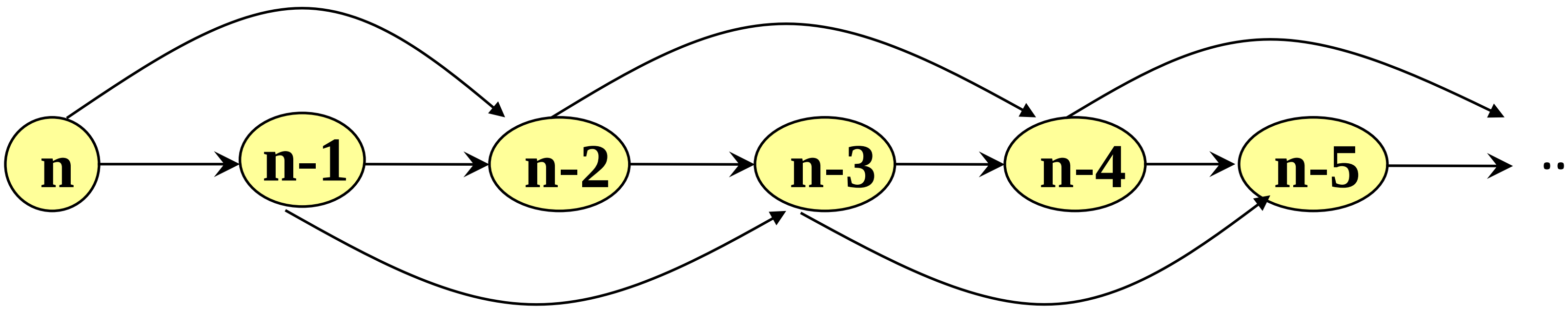


Dynamic programming (Bottom Up)

1. Formulate the problem P in terms of smaller versions of the problem (recursively), say, Q_1, Q_2, …
2. Turn this formulation into a recursive function to solve problem P
3. Draw the subproblem graph and find the dependencies among subproblems
4. Use a dictionary to store solutions to subproblems
5. In the iterative function to solve P

- compute the solutions of subproblems of a problem first
- The solution to P is computed based on the solutions to its subproblems and is stored into the dictionary

The subproblem graph of fib(n)



- Observation 1: Since we have a sequence of subproblems for fib(n), we will use an one-dimensional array to memorize the solutions of subproblems. E.g. fib(6)

	0	1	2	3	4	5	6
Soln							

- Observation 2: Seeing the dependencies among the solutions – fib(n) needs the solutions of fib(n-1) and fib(n-2), we can compute the elements in this array in a correct order.

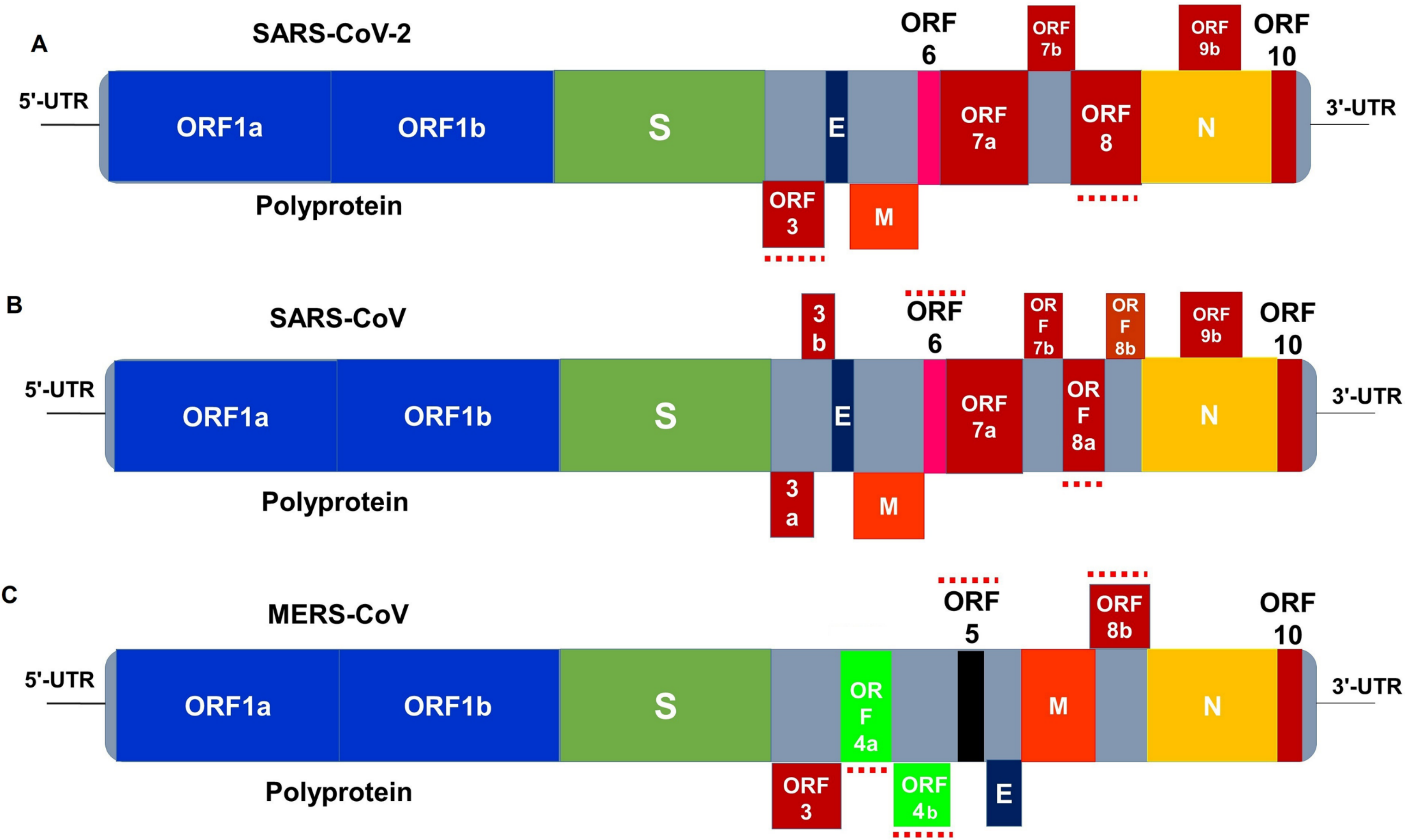
```
soln[0] = 0;
soln[1] = 1;
for j = 2 to n
    soln[j] = soln[j-1] + soln[j-2];
```

Complexity: O(n)

	0	1	2	3	4	5	6
Soln	0	1	1	2	3	5	8

Longest Common Subsequence

Comparative Genomics



Longest Common Subsequence

- Given a sequence $s = <s_1, s_2, \ldots, s_n>$, a **subsequence** is any sequence $<s_{i_1}, s_{i_2}, \ldots, s_{i_m}>$, with i_j strictly increasing.

  Example: s = *ACTTGCG*

  ACT, AG, ATTC, T, ACTTGC are all subsequences.
  TTA, AGGC are not subsequences.

- Given two sequences $x = <x_1, x_2, \ldots, x_n>$, $y = <y_1, y_2, \ldots, y_m>$, a **common subsequence** is a subsequence of both x and y.

- A **longest common subsequence** (LCS) is a common subsequence of maximum length

  Example: x = *AAACCGTGAGTTATTCGTTCTAGAA*

  y = *CACCCCTAAGGTACCTTTGGTTC*

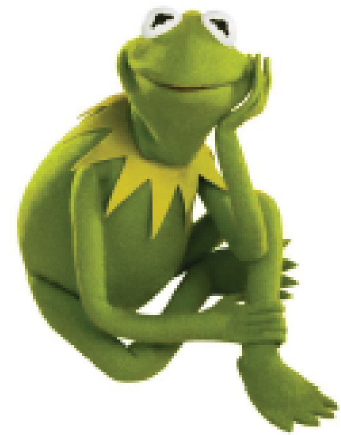
  Common subsequences: *ACGG*, *CAGTTTC*

  LCS = ***ACCTAGTACTTTG*** (LCS may not be unique)

- LCS has many applications including document analysis and computational biology – the similarity between two sequences is measured by the length of LCS

- How similar are these two species?

DNA:
AGCCCTAAGGGCTACCTAGCTT



DNA:
GACAGCCTACAAGCGTTAGCTTG

<u>Problem definition</u>: Given two sequences $x = \langle x_1, x_2, \ldots, x_n \rangle$, $y = \langle y_1, y_2, \ldots, y_m \rangle$, compute LCS(n, m) that gives the length of the longest common subsequence

Principle of Optimality

- A trivial algorithm: find all subsequences of x (there are up to 2^n of them) and check whether they are subsequences of y.

- What is the complexity of this trivial algorithm?

- This is an optimization problem

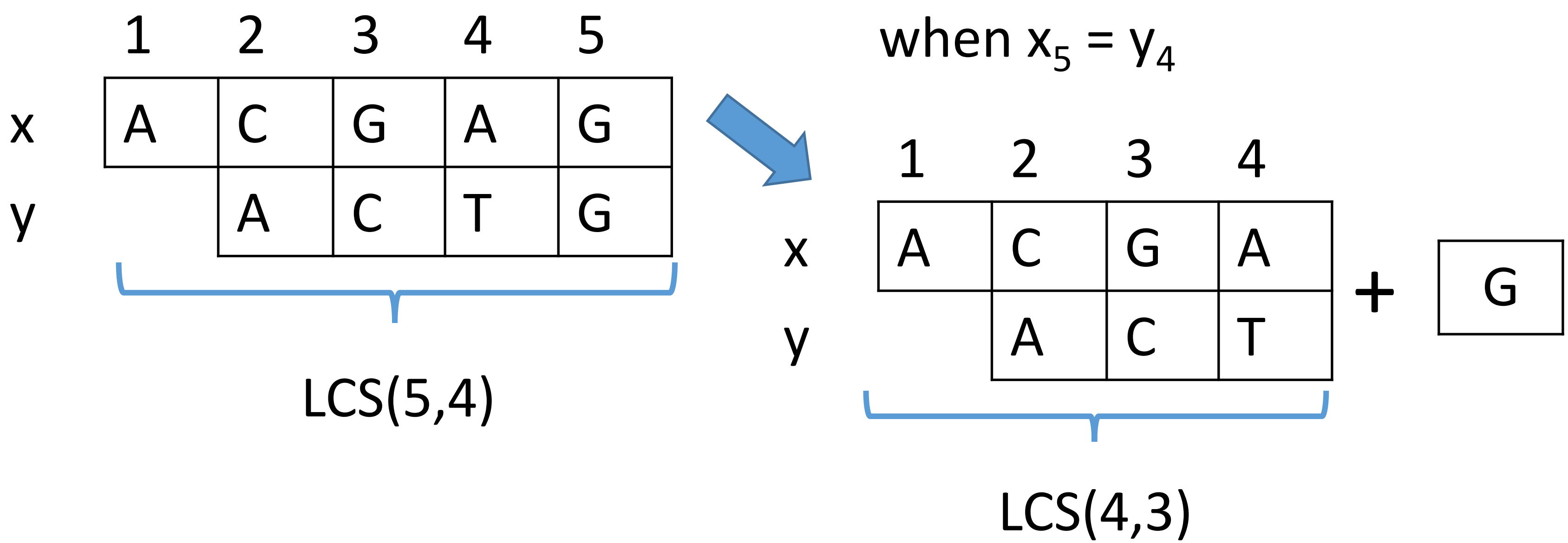
- Dynamic programming is a powerful tool to solve optimization problems that satisfy the ***Principle of Optimality***

- A problem is said to satisfy the principle of optimality if the subsolutions of an optimal solution of the problem are themselves optimal solutions for their subproblems.

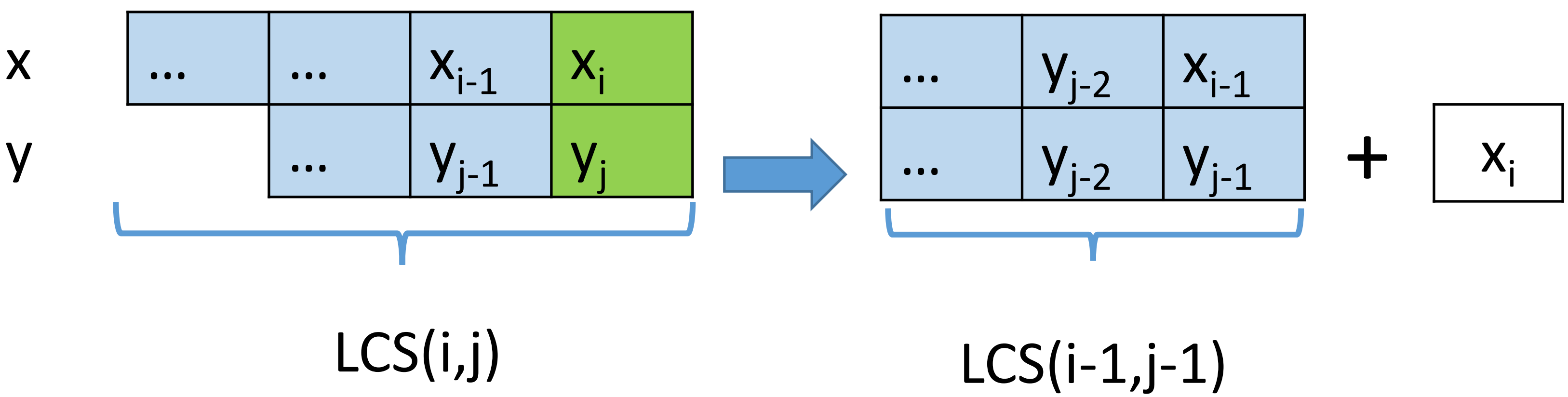
- Does the LCS problem satisfy this principle?

Step 1: Formulate the problem P in terms of smaller versions of the problem

- Consider two sequences x = $<x_1, x_2, \ldots, x_i>$, y = $<y_1, y_2, \ldots, y_j>$. Take them as character strings. E.g.



- If $x_i = y_j$, then this character is the last character in the longest common subsequence. The longest common subsequence is the longest common subsequence of $<x_1, x_2, \ldots, x_{i-1}>$, and $<y_1, y_2, \ldots, y_{j-1}>$ followed by x_i.

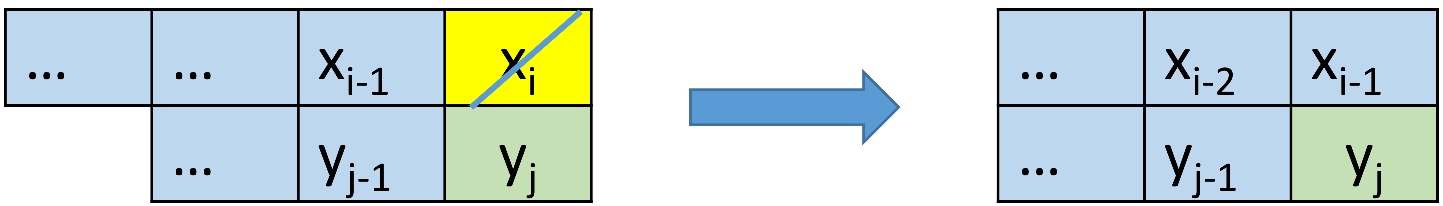


LCS(i-1,j-1) is a subsolution of LCS(i,j) when $x_i = y_j$.
LCS(i-1,j-1) is an optimal solution.
Otherwise LCS(i,j) cannot be an optimal solution

- If $x_i \neq y_j$, then either x_i is not in the LCS or y_j is not in the LCS (or both of them are not in the LCS).

  If x_i is not in the LCS, we just need to find the longest common subsequence of $<x_1, x_2, \ldots, x_{i-1}>$ and $<y_1, y_2, \ldots, y_j>$.

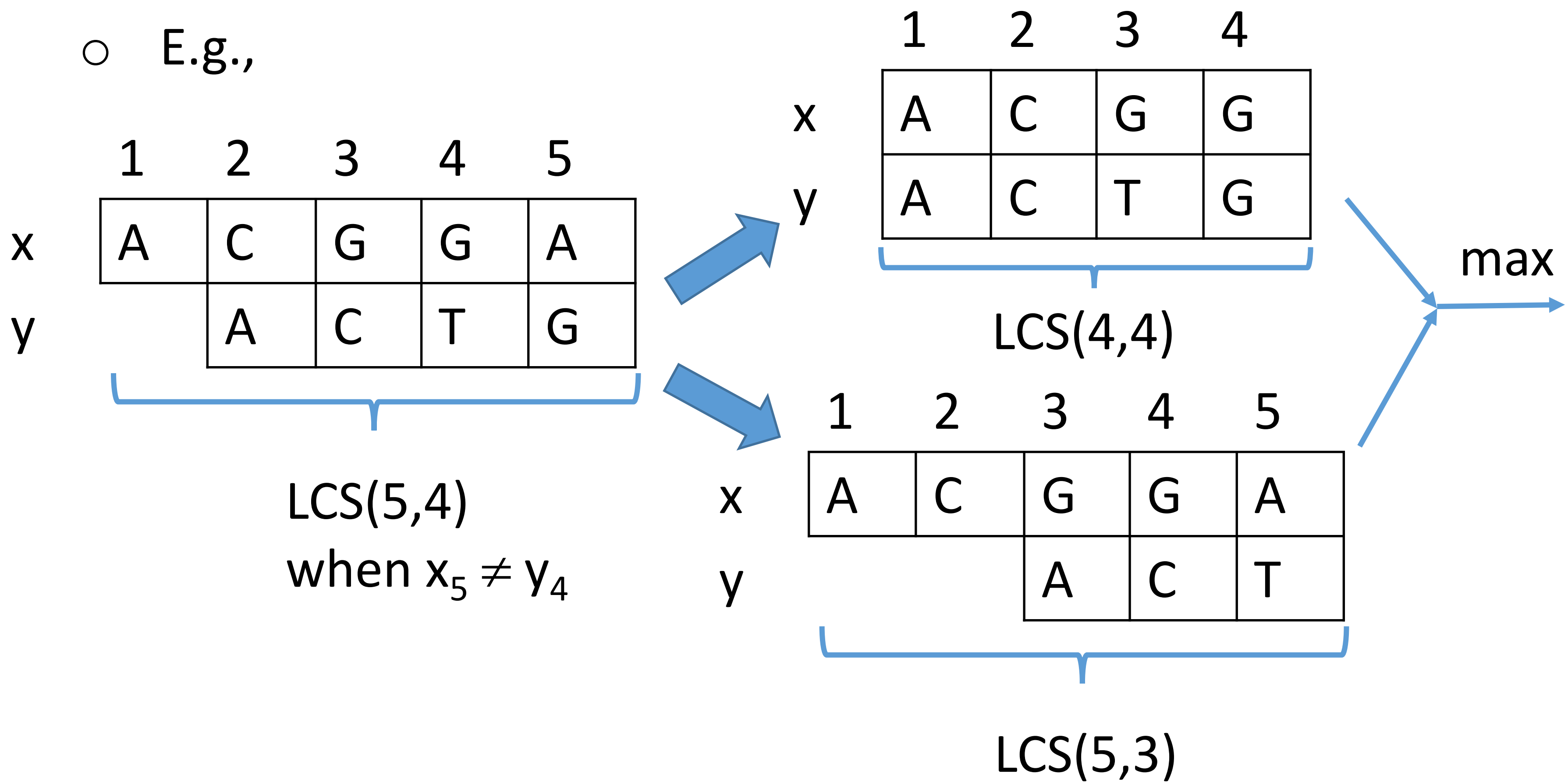
LCS(i,j) LCS(i-1,j)



  If y_j is not in the LCS, we just need to find the longest common subsequence of $<x_1, x_2, \ldots, x_i>$ and $<y_1, y_2, \ldots, y_{j-1}>$.

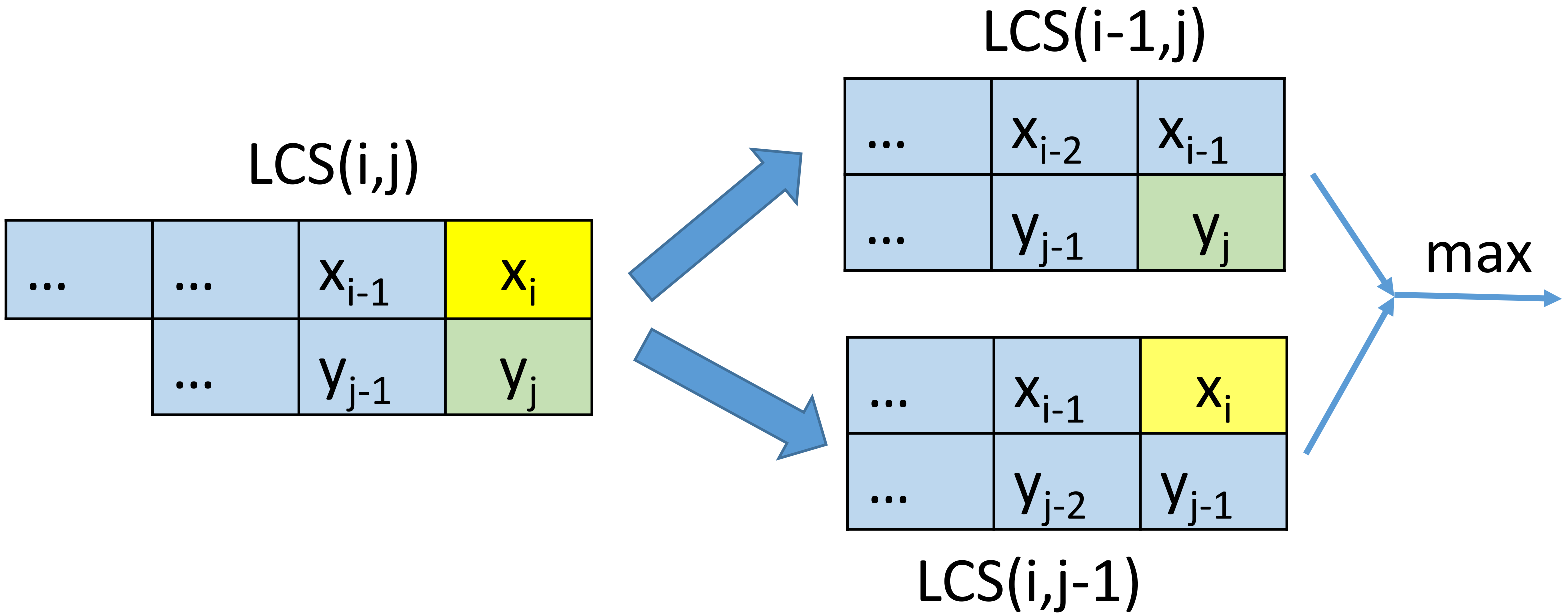
LCS(i,j) LCS(i,j-1)

- E.g.,



- The dynamic programming selection rule: **when given a number of possibilities, compute all and take the best**.

Therefore, when $x_i \neq y_j$,



- LCS(i-1,j-1), LCS(i-1,j) and LCS(i,j-1) are the optimal solutions for the respective subproblems. Otherwise LCS(i,j) cannot be optimal – principle of optimality.

Recursive Function

Step 2: Turn this formulation into a recursive function to solve the longest common subsequence problem:

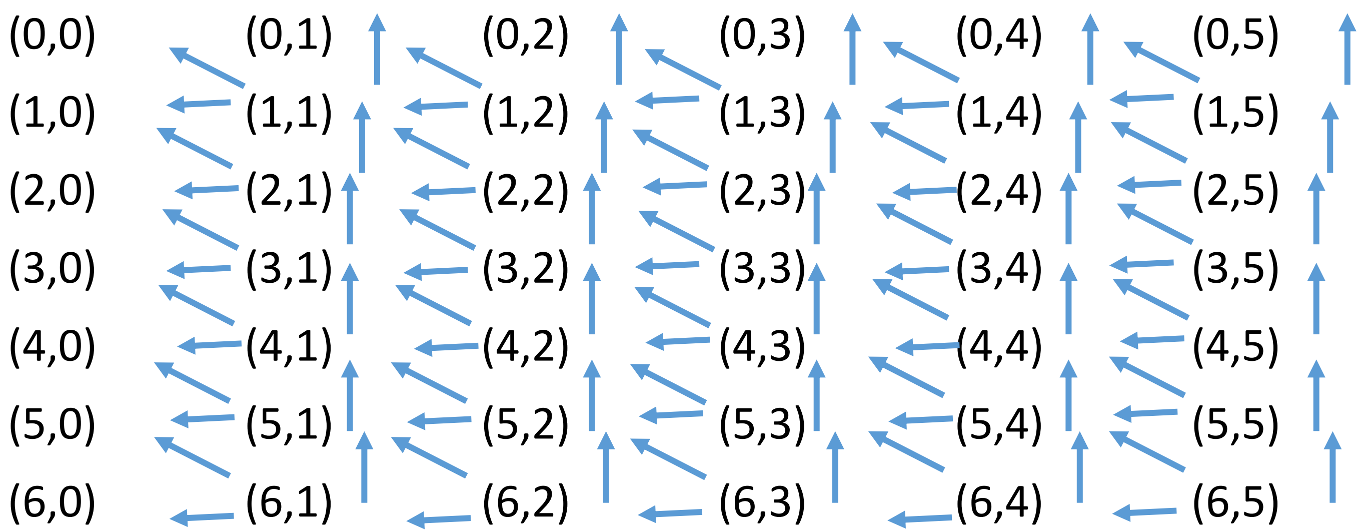
$$\begin{aligned}
LCS(i,j) &= 0 && \text{if } i=0 \text{ or } j=0 \\
LCS(i,j) &= LCS(i-1,j-1) + 1 && \text{if } i,j > 0,\ x_i = y_j \\
LCS(i,j) &= \max(LCS(i-1,j),\ LCS(i,j-1)) && \text{if } i,j > 0,\ x_i \neq y_j
\end{aligned}$$

- The top down approach using a recursive function will be very inefficient.

Longest Common Subsequence (Bottom-Up Approach)

Step 3 (bottom up approach): Draw the subproblem graph and find the dependencies among subproblems

E.g, the subproblem graph of LCS(6,5)



Step 4: the dictionary is a n+1 by m+1 array.

Initialise row 0, column 0.

Compute from row 1 to row n, column 1 to column m within each row.

Step 5

```
Int LCS(n, m)
{
   for i = 0 to n   c[i][0] = 0;
   for j = 1 to m    c[0][j] = 0;
   for i = 1 to n
        for j = 1 to m
              if x[i] == y[j]
                    c[i][j] = c[i-1][j-1] + 1;
              else if c[i-1][j] >= c[i][j-1]
                    c[i][j] = c[i-1][j];
              else c[i][j] = c[i][j-1];
   return c[n][m];
}
```

```
Int LCS(n, m)
{
   for i = 0 to n   c[i][0] = 0;
   for j = 1 to m    c[0][j] = 0;
   for i = 1 to n
        for j = 1 to m
             if x[i] == y[j]
                  c[i][j] = c[i-1][j-1] + 1;
             else if c[i-1][j] >= c[i][j-1]
                  c[i][j] = c[i-1][j];
             else c[i][j] = c[i][j-1];
   return c[n][m];
}
```

Space Complexity:
(n+1)x(m+1) array
O(nm)

Total no. of iterations: nm

Bounded by a constant time

Time Complexity:
O(nm)

Example 1

	1	2	3	4	5
x	A	C	G	G	A
y	A	C	T	G	

for i = 0 to n   c[i][0] = 0;
for j = 1 to m   c[0][j] = 0;

		A	**C**	**T**	**G**
	0	**0**	**0**	**0**	**0**
A	**0**				
C	**0**				
G	**0**				
G	**0**				
A	**0**				

Example 1

	1	2	3	4	5
x	A	C	G	G	A
y	A	C	T	G	

		A	C	T	G
	0	0	0	0	0
A	0	1	1	1	1
C	0	1	2	2	2
G	0	1	2	2	3
G	0	1	2	2	3
A	0	1	2	2	**3**

```
for i = 1 to n
   for j = 1 to m
      if x[i] == y[j]
         c[i][j] = c[i-1][j-1] + 1;
      else if c[i-1][j] >= c[i][j-1]
         c[i][j] = c[i-1][j];
      else c[i][j] = c[i][j-1];
```

LCM(5,4) = 3

Longest Common Subsequence (Find the Subsequence)

- To find the longest common subsequence, a hint array is used to indicate for LCS(i,j) where the optimal subsolution is from : LCS(i-1, j-1), LCS(i-1, j) or LCS(i, j-1).
    - First column of the hint array will be filled with ‘|’.
    - First row of the hint array will be filled ‘—’.
    - For the remaining cells h[i][j],
        - If LCS(i,j) = LCS(i-1, j-1)+1, h[i][j] = ‘\’
        - If LCS(i,j) = LCS(i-1, j), h[i][j] = ‘|’
        - If LCS(i,j) = LCS(i, j-1), h[i][j] = ‘—’

```
Int LCS(n, m)     // with hints to find the sequence
{
   for i = 0 to n   {  c[i][0] = 0;  h[i][0] = '|';   }
   for j = 1 to m  {  c[0][j] = 0;  h[0][j] = '—';  }
   for i = 1 to n
        for j = 1 to m
             if x[i] == y[j]
                  { c[i][j] = c[i-1][j-1] + 1; h[i][j] = '\';  }
             else if c[i-1][j] >= c[i][j-1]
                  { c[i][j] = c[i-1][j]; h[i][j] = '|';  }
             else {  c[i][j] = c[i][j-1]; h[i][j] = '—';  }
   return c[n][m];
}
```

Time Complexity:
O(nm)

- To obtain the longest common subsequence computed, we start from h[n][m].
- For each element of the hint array, h[i][j],
  - If h[i][j] = '\', it means $x_i = y_j$ and this character is the last character of the longest common subsequence of $x_1 .. x_i$ and $y_1 .. y_j$. This character is preceded by the longest common subsequence of $x_1 .. x_{i-1}$ and $y_1 .. y_{j-1}$.
  - If h[i][j] = '|', it means the longest common subsequence of $x_1 .. x_i$ and $y_1 .. y_j$ is the longest common subsequence of $x_1 .. x_{i-1}$ and $y_1 .. y_j$.
  - If h[i][j] = '—', it means the longest common subsequence of $x_1 .. x_i$ and $y_1 .. y_j$ is the longest common subsequence of $x_1 .. x_i$ and $y_1 .. y_{j-1}$.
- After reaching the 1st row/column of the hint array, end

Example 1

	A	C	T	G
	0	**0**	**0**	**0**	**0**
A	**0**	**1**	**1**	**1**	**1**
C	**0**	**1**	**2**	**2**	**2**
G	**0**	**1**	**2**	**2**	**3**
G	**0**	**1**	**2**	**2**	**3**
A	**0**	**1**	**2**	**2**	**3**

<table>
<tr><th></th><th></th><th>A</th><th>C</th><th>T</th><th>G</th></tr>
<tr><td></td><td>—</td><td>—</td><td>—</td><td>—</td><td>—</td></tr>
<tr><td>A</td><td>|</td><td>\</td><td>—</td><td>—</td><td>—</td></tr>
<tr><td>C</td><td>|</td><td>|</td><td>\</td><td>—</td><td>—</td></tr>
<tr><td>G</td><td>|</td><td>|</td><td>|</td><td>|</td><td>\</td></tr>
<tr><td>G</td><td>|</td><td>|</td><td>|</td><td>|</td><td>\</td></tr>
<tr><td>A</td><td>|</td><td>\</td><td>|</td><td>|</td><td>|</td></tr>
</table>

	1	2	3	4	5
x	A	C	G	G	A
y	A	C	T	G	

h(5,4) = ‘|’
h(4,4) = ‘\’ → **G**
h(3,3) = ‘|’
h(2,3) = ‘—’
h(2,2) = ‘\’ → **C**
h(1,1) = ‘\’ → **A**
end

The sub sequence: **A C G**

Chain Matrix Multiplication

Chain Matrix Multiplication

- The Order problem

$$\begin{bmatrix} \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot \end{bmatrix} \begin{bmatrix} \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \end{bmatrix}$$

2 X 3 X 4

Consider A_1 x A_2 x A_3 x A_4

30x1 1x40 40x10 10x25

Many possibilities. For examples,

$((A_1A_2)A_3)A_4 \Rightarrow$ 30x1x40 + 30x40x10 + 30x10x25 = 20,700 multiplications

$A_1(A_2(A_3A_4)) \Rightarrow$ 40x10x25 + 1x40x25+ 30x1x25 = 11,750 multiplications

$(A_1A_2)(A_3A_4) \Rightarrow$ 30x1x40 + 40x10x25 + 30x40x25 = 41,200 multiplications

$A_1((A_2A_3)A_4) \Rightarrow$ 1x40x10 + 1x10x25 + 30x1x25 = 1,400 multiplications

Problem definition: given matrices A_1, A_2, ….A_n where dimensions of A_i are d_{i-1} x d_i (for $1 \leq i \leq n$), what order should the matrix multiplications be computed in order to incur minimum cost? Cost is the number of multiplications.

d_0 d_1 d_2 d_3 … d_{n-1} d_n

- There are (n-1)! ways for n matrices
- Matrix multiplication is associative: (AB)C = A(BC). So different ways give the same result
- This is an optimization problem

Step 1: formulate the matrix multiplication cost problem in terms of smaller versions of the same problem

Consider a sequence of 6 matrices:

$A_1 \quad x \quad A_2 \quad x \quad A_3 \quad x \quad A_4 \quad x \quad A_5 \quad x \quad A_6$ matrices

$$\underbrace{d_0xd_1 \quad d_1xd_2 \quad d_2xd_3}_{B_1} \quad \underbrace{d_3xd_4 \quad d_4xd_5 \quad d_5xd_6}_{B_2}$$ dimensions

Suppose the <u>last</u> matrix multiplication were at A_3; then

1) We need to multiply $A_1 \ x \ A_2 \ x \ A_3$ to create B_1, a d_0 x d_3 matrix
2) We need to multiply $A_4 \ x \ A_5 \ x \ A_6$ to create B_2, a d_3 x d_6 matrix

Cost would be the cost of (1)+(2)+ cost of(B_1 x B_2)

$A_1 \quad x \quad A_2 \quad x \quad A_3 \quad x \quad A_4 \quad x \quad A_5 \quad x \quad A_6$ matrices

$d_0 \times d_1 \quad d_1 \times d_2 \quad d_2 \times d_3 \quad d_3 \times d_4 \quad d_4 \times d_5 \quad d_5 \times d_6$ dimensions

The last multiplication may be at each of the 5 matrices.

$$\text{Cost}((A_1A_2A_3A_4\,A_5)(A_6)) = \text{Cost}(A_1A_2A_3A_4A_5) + \text{Cost}(A_6) + d_0 \times d_5 \times d_6$$

$$\text{Cost}((A_1A_2A_3A_4)(A_5A_6)) = \text{Cost}(A_1A_2A_3A_4) + \text{Cost}(A_5\,A_6) + d_0 \times d_4 \times d_6$$

$$\text{Cost}((A_1A_2A_3)(A_4A_5A_6)) = \text{Cost}(A_1A_2A_3) + \text{Cost}(A_4A_5A_6) + d_0 \times d_3 \times d_6$$

$$\text{Cost}((A_1A_2)(A_3A_4A_5A_6)) = \text{Cost}(A_1A_2) + \text{Cost}(A_3A_4A_5A_6) + d_0 \times d_2 \times d_6$$

$$\text{Cost}((A_1)(A_2A_3A_4A_5A_6)) = \text{Cost}(A_1) + \text{Cost}(A_2A_3A_4A_5A_6) + d_0 \times d_1 \times d_6$$

$A_1 \quad x \quad A_2 \quad x \quad A_3 \quad x \quad A_4 \quad x \quad A_5 \quad x \quad A_6$ matrices

$d_0 \times d_1 \quad d_1 \times d_2 \quad d_2 \times d_3 \quad d_3 \times d_4 \quad d_4 \times d_5 \quad d_5 \times d_6$ dimensions

The dynamic programming selection rule: **when given a number of possibilities, compute all and take the best**.

The optimal cost of multiplying the 6 matrices:

OptCost($A_1A_2A_3A_4A_5A_6$) = **Min**(

OptCost($A_1A_2A_3A_4A_5$) + OptCost(A_6) + $d_0 \times d_5 \times d_6$,

OptCost($A_1A_2A_3A_4$) + OptCost(A_5A_6) + $d_0 \times d_4 \times d_6$,

OptCost($A_1A_2A_3$) + OptCost($A_4A_5A_6$) + $d_0 \times d_3 \times d_6$,

OptCost(A_1A_2) + OptCost($A_3A_4A_5A_6$) + $d_0 \times d_2 \times d_6$,

OptCost(A_1) + OptCost($A_2A_3A_4A_5A_6$) + $d_0 \times d_1 \times d_6$)

OptCost(A) = 0

Recursive Function

Step 2: Turn this formulation into a recursive function to solve the chain matrix multiplication problem.

Suppose we use array **d** to store the dimensions of the matrices.

d_0	d_1	d_2	...		

Let OptCost(i,j) be the optimal cost of multiplying matrices with dimensions $d_i \times d_{i+1}$, $d_{i+1} \times d_{i+2}$, …, $d_{j-1} \times d_j$.

OptCost(i, j) = 0 if j-i=1

OptCost(i, j)

$$= \min_{i+1 \le k \le j-1} (\text{OptCost}(i,k) + \text{OptCost}(k, j) + d_i \times d_k \times d_j)$$

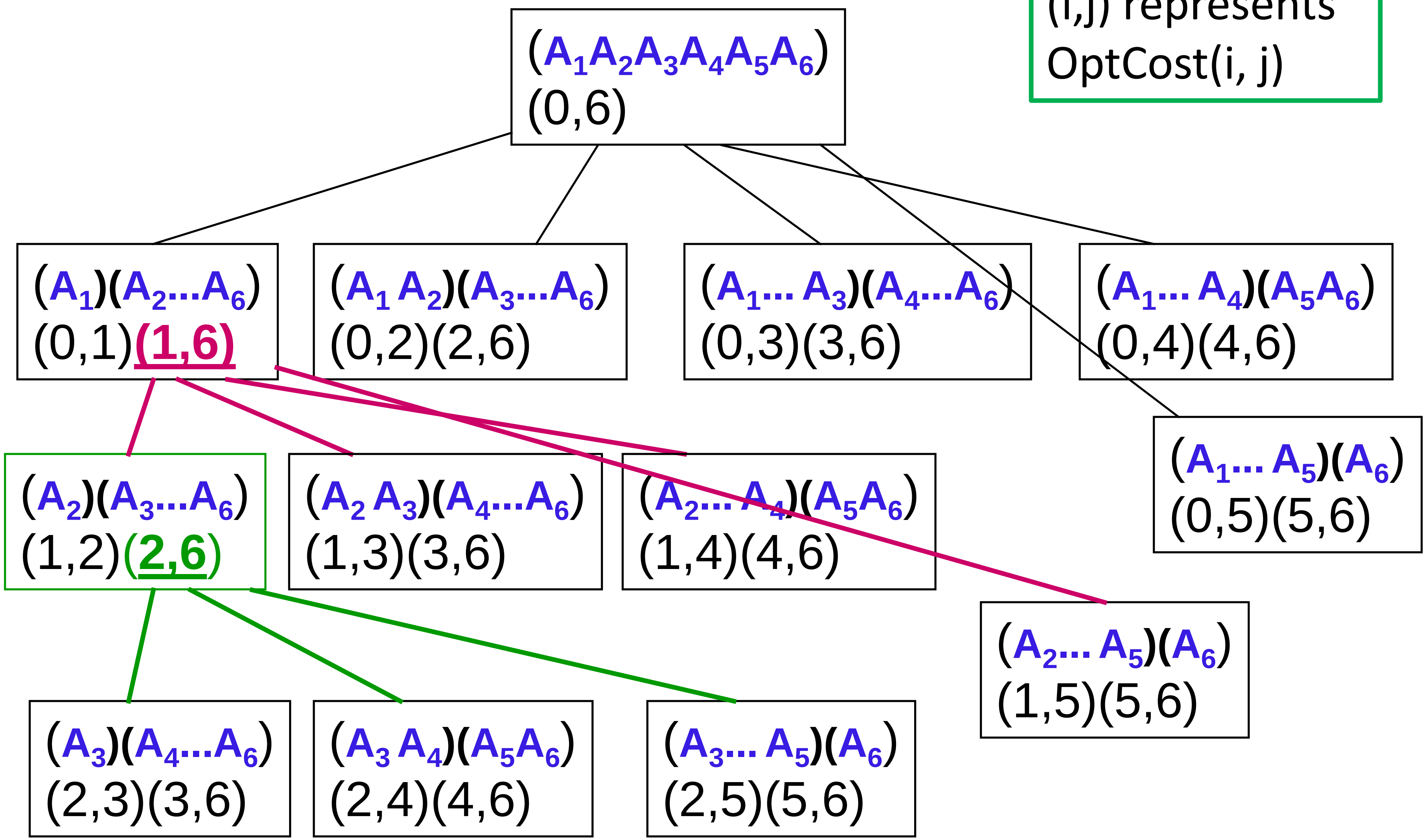
if j-i>1

The optimal cost of multiplying n matrices is OptCost(0, n).

- The chain matrix multiplication problem satisfies the principle of optimality
    - OptCost(i, k) and OptCost(k, j) for k = i+1, …, j-1 are the subsolutions of OptCost(i, j)
    - They are the optimal solutions for the subproblems
    - Proof by contradiction

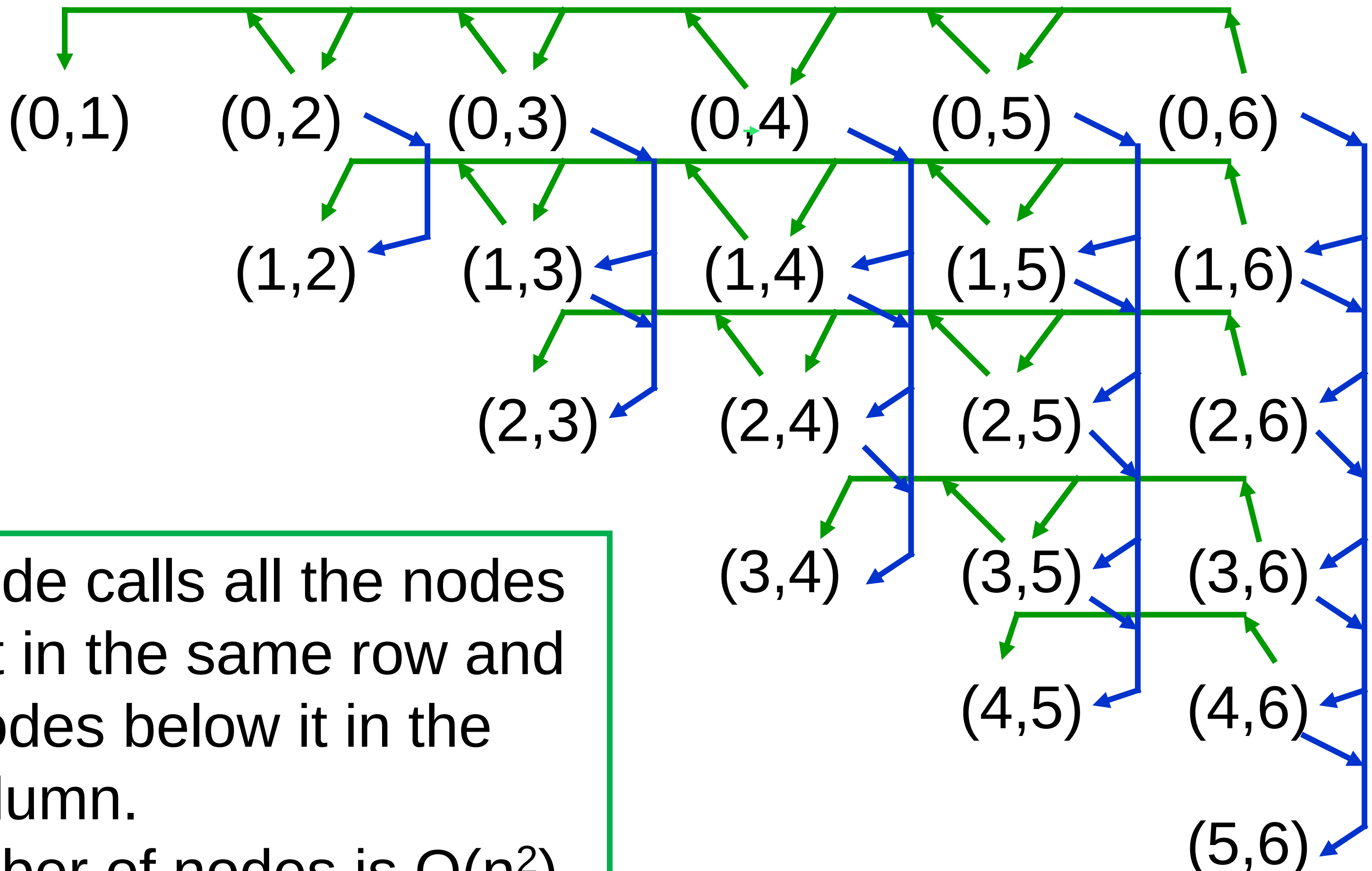
Example: $A_1 \times A_2 \times A_3 \times A_4 \times A_5 \times A_6$

(i,j) represents OptCost(i, j)



Chain Matrix Multiplication (Bottom-Up Approach)

Step 3: Draw the subproblem graph and find the dependencies among subproblems



Every node calls all the nodes on its left in the same row and all the nodes below it in the same column.
The number of nodes is $O(n^2)$.

```
int matrixOrder(int [] d, int n)
{   for i = 0 to n-1      cost[i][i+1] = 0;
    for l = 2 to n
        for i = 0 to n-l
            j = i + l;
            cost[i][j] = ∞;
            for k = i+1 to j-1
                c = cost[i][k] + cost[k][j] + d[i]*d[k]*d[j];
                if (c < cost[i][j])
                    cost[i][j] = c;  last[i][j] = k;
    }
```

$$\min_{i+1 \le k \le j-1} (\text{OptCost}(i,k) + \text{OptCost}(k,j) + d_i \times d_k \times d_j)$$

```
int matrixOrder(int [] d, int n)
{   for i = 0 to n-1        cost[i][i+1] = 0;
   for l = 2 to n
      for i = 0 to n-l
         j = i + l;
         cost[i][j] = ∞;
         for k = i+1 to j-1
            c = cost[i][k] + cost[k][j] + d[i]*d[k]*d[j];
            if (c < cost[i][j])
               cost[i][j] = c;  last[i][j] = k;
   }
```

Repeated $O(n^2)$ times

Repeated $O(n^3)$ times

Complexity of computing the optimal order: $O(n^3)$

Example

A_1 x A_2 x A_3 x A_4

30x1 1x40 40x10 10x25

Array **d**

30	1	40	10	25
0	**1**	**2**	**3**	**4**

Call to matrixOrder(d, 4)

d	30	1	40	10	25
	0	1	2	3	4

cost

	0	**1**	**2**	**3**	**4**
0		0			
1			0		
2				0	
3					0
4					

for i = 0 to n-1

cost[i][i+1] = 0;

d

30	1	40	10	25
0	1	2	3	4

cost

	0	1	2	3	4
0		0	1200		
1			0		
2				0	
3					0
4					

	0	1	2	3	4
0			1		
1					
2					
3					
4					

last

```
for l = 2 to n
    for i = 0 to n-l
        j = i + l;
        cost[i][j] = ∞;
        for k = i+1 to j-1
            c = cost[i][k] + cost[k][j]
                + d[i]*d[k]*d[j];
            if (c < cost[i][j])
                cost[i][j] = c;
                last[i][j] = k;
```

k=1:
Cost[0][1]+
Cost[1][2] + 1200

l=2, i=0, j=2, k=1

d

30	1	40	10	25
0	1	2	3	4

cost

	0	1	2	3	4
0		0	1200		
1			0	400	
2				0	
3					0
4					

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

last

```
for l = 2 to n
    for i = 0 to n-l
        j = i + l;
        cost[i][j] = ∞;
        for k = i+1 to j-1
            c = cost[i][k] + cost[k][j]
                + d[i]*d[k]*d[j];
            if (c < cost[i][j])
                cost[i][j] = c;
                last[i][j] = k;
```

k=2:
Cost[1][2]+
Cost[2][3] + 400

l=2, i=1, j=3, k=2

d	30	1	40	10	25
	0	1	2	3	4

cost

	0	**1**	**2**	**3**	**4**
0		0	1200		
1			0	400	
2				0	10000
3					0
4					

	0	**1**	**2**	**3**	**4**
0			1		
1				2	
2					3
3					
4					

last

```
for l = 2 to n
    for i = 0 to n-l
        j = i + l;
        cost[i][j] = ∞;
        for k = i+1 to j-1
            c = cost[i][k] + cost[k][j]
                + d[i]*d[k]*d[j];
            if (c < cost[i][j])
                cost[i][j] = c;
                last[i][j] = k;
```

k=3:
Cost[2][3]+
Cost[3][4] + 10000

l=2, i=2, j=4, k=3

d					
	30	1	40	10	25
	0	1	2	3	4

cost

	0	1	2	3	4
0		0	1200	700	
1			0	400	
2				0	10000
3					0
4					

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

last

k=1:
Cost[0][1]+
Cost[1][3] + 300
k=2:
Cost[0][2]+
Cost[2][3] + 12000

```
for l = 2 to n
    for i = 0 to n-l
        j = i + l;
        cost[i][j] = ∞;
        for k = i+1 to j-1
            c = cost[i][k] + cost[k][j]
                + d[i]*d[k]*d[j];
            if (c < cost[i][j])
                cost[i][j] = c;
                last[i][j] = k;
```

l=3, i=0, j=3, k=1,2

d	30	1	40	10	25
	0	1	2	3	4

cost

	0	1	2	3	4
0		0	1200	700	
1			0	400	650
2				0	10000
3					0
4					

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

last

k=2:
Cost[1][2]+
Cost[2][4] + 1000
k=3:
Cost[1][3]+
Cost[3][4] + 250

```
for l = 2 to n
    for i = 0 to n-l
        j = i + l;
        cost[i][j] = ∞;
        for k = i+1 to j-1
            c = cost[i][k] + cost[k][j]
                + d[i]*d[k]*d[j];
            if (c < cost[i][j])
                cost[i][j] = c;
                last[i][j] = k;
```

l=3, i=1, j=4, k=2,3

d					
	30	1	40	10	25
	0	1	2	3	4

cost

	0	1	2	3	4
0		0	1200	700	1400
1			0	400	650
2				0	10000
3					0
4					

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

last

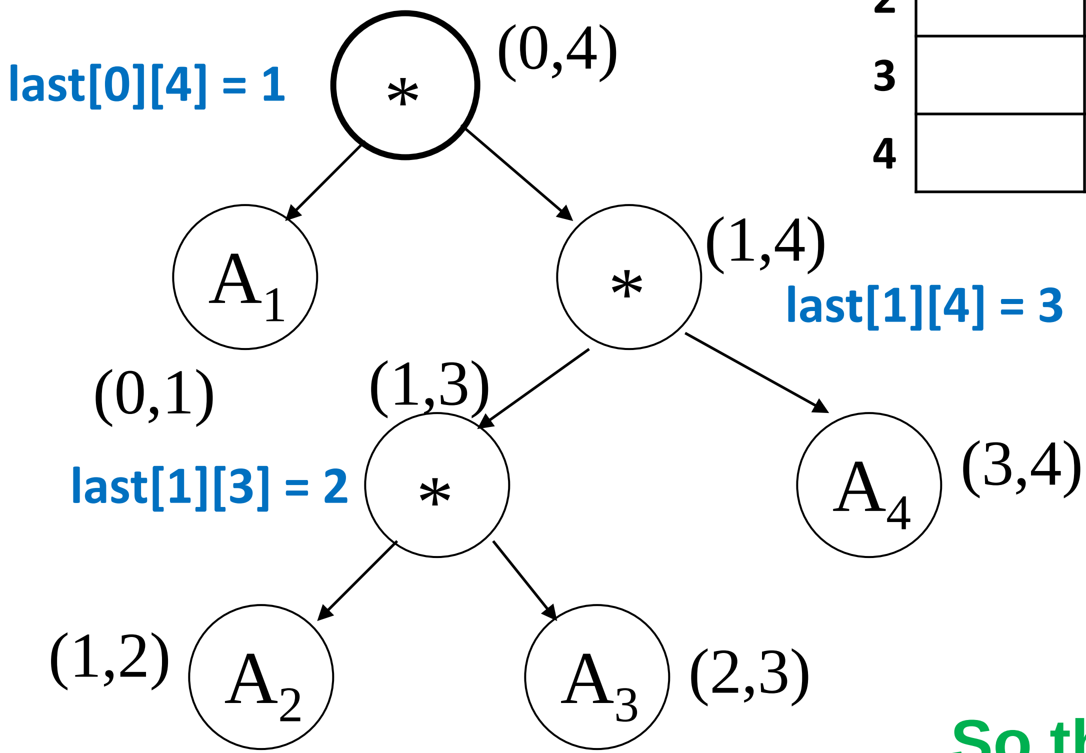
k=1:
Cost[0][1]+
Cost[1][4] + 750
k=2:
Cost[0][2]+
Cost[2][4] + 30000
k=3:
Cost[0][3]+
Cost[3][4] + 7500

```
for l = 2 to n
    for i = 0 to n-l
        j = i + l;
        cost[i][j] = ∞;
        for k = i+1 to j-1
            c = cost[i][k] + cost[k][j]
                + d[i]*d[k]*d[j];
            if (c < cost[i][j])
                cost[i][j] = c;
                last[i][j] = k;
```

l=4, i=0, j=4, k=1,2,3

	0	1	2	3	4
0			1	1	**1**
1				**2**	**3**
2					3
3					
4					

last



So the best sequence is (A1 x ((A2 x A3) x A4))

0/1 Knapsack Problem

0/1 Knapsack problem

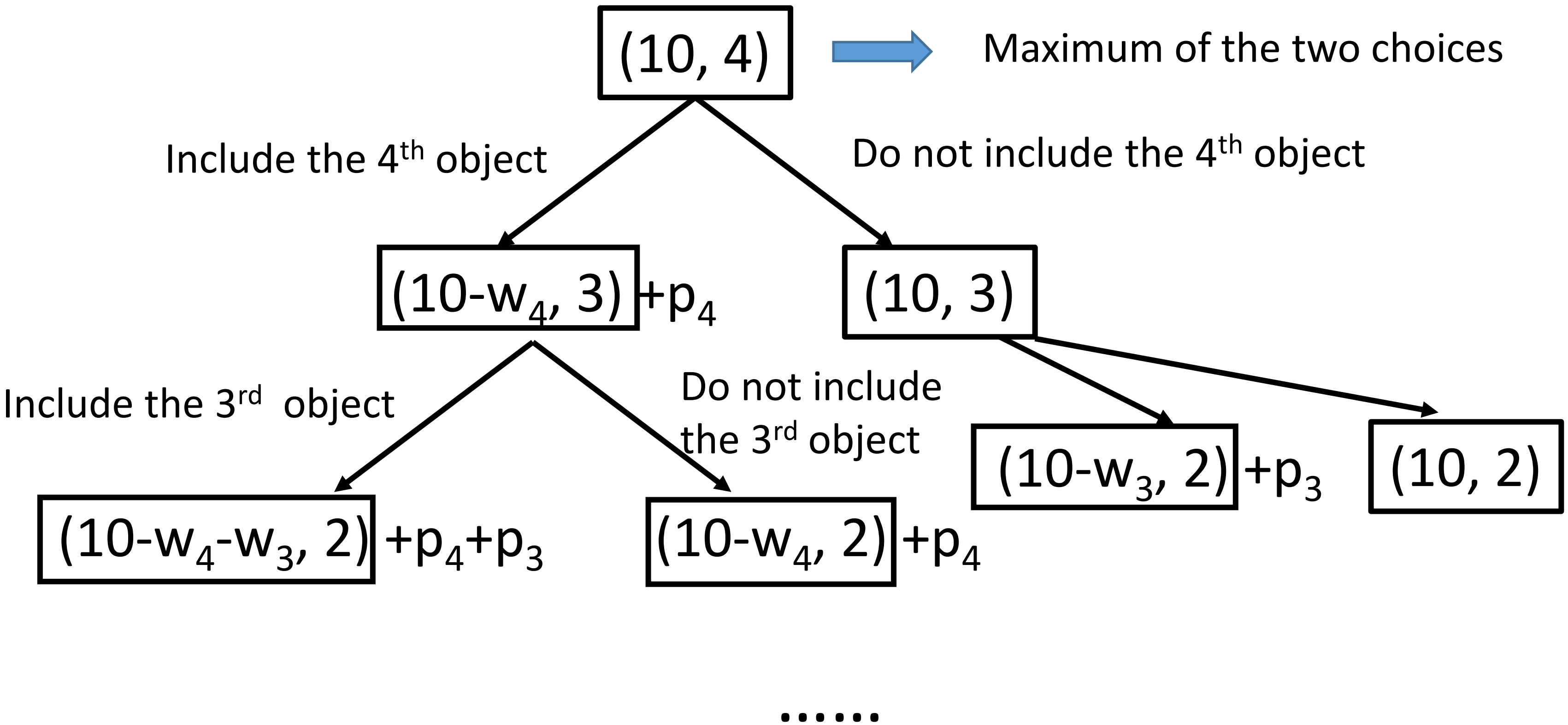
Problem definition: We have a knapsack of capacity weight *C* (a positive integer) and *n* objects with weights w_1, w_2, …w_n and profits p_1, p_2, …p_n (all w_i and all p_i are positive integers), find the largest total profit of any subset of the objects that fits in the knapsack.

- We have to take whole objects.
- There are 2^n subsets of n objects: examining all subsets takes $O(2^n)$ time

E.g. application: ***C*** is amount of money to invest, weights, $\mathbf{w_1}$,… are investment amounts and profit is the expected return on investment.

- Step 1: formulate the 0/1 knapsack problem in terms of smaller versions of the same problem
  - Consider the last object of the *n* objects with weights w_1, w_2, …w_n.
  - If we include it in the knapsack, the available weight capacity in the knapsack will be reduced by w_n. Then our profit will be p_n, plus the maximum we can get from solving the subproblem of n-1 objects and capacity of C- w_n.
  - If we do not include it in the knapsack, our profit will be the maximum we can get from solving the subproblem of n-1 objects and capacity of C.
  - The dynamic programming selection rule: **when given a number of possibilities, compute all and take the best**.

For example, a knapsack of capacity 10 and 4 objects



Recursive Function

<u>Step 2</u>: Turn this formulation into a recursive function to solve the 0/1 knapsack problem

Let P(C, j) be the maximum profit that can be made by selecting a subset of the j objects with knapsack capacity of C.

$P(C, 0) = P(0, j) = 0$

$P(C, j) = \max(P(C, j-1), p_j + P(C-w_j, j-1))$

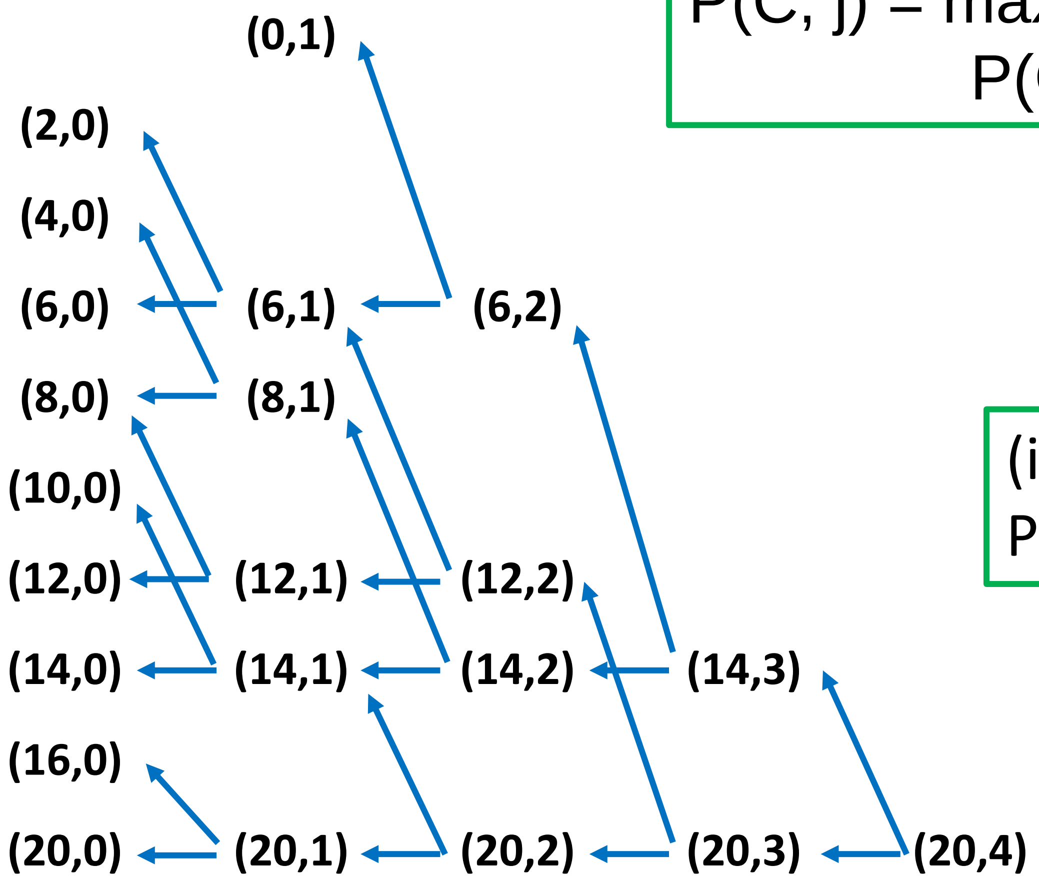
- <u>Step 3</u>: Draw the subproblem graph and find the dependencies among subproblems

For example, C = 20

	1	2	3	4
$\mathbf{w_i}$	4	6	8	6
$\mathbf{p_i}$	7	6	9	5

	1	2	3	4
w_i	4	6	8	6
p_i	7	6	9	5

$$P(C, j) = \max(P(C, j-1), p_j + P(C-w_j, j-1))$$



(i,j) represents P(i, j)

0/1 Knapsack Problem (Bottom-Up Approach)

Step 4 :Dictionary: profit[C+1][n+1]

	0	1	2	…	n
0					
1					
2					
3					
4					
5					
6					
7					
…					
C					

Step 5

```
int knapsack(int [] w, int [] p, int C, int n)
{   for c = 0 to n        profit[0][c] = 0;
    for r = 1 to C        profit[r][0] = 0;
    for r = 1 to C
        for c = 1 to n
            profit[r][c] = profit[r][c-1];
            if (w[c] <= r)
                if (profit[r][c] < profit[r-w[c]][c-1] + p[c])
                    profit[r][c] = profit[r-w[c]][c-1] + p[c];
}
```

Complexity : O(nC)

$P(C, j) = \max(P(C, j-1),\ p_j + P(C-w_j, j-1))$

Example 1:
C = 3

	1	2	3
w_i	1	2	3
p_i	1	4	6

	0	1	2	3
0	0	0	0	0
1	0			
2	0			
3	0			

```
for c = 0 to n
    profit[0][c] = 0;
for r = 1 to C
    profit[r][0] = 0;
```

Example 1:
C = 3

	1	2	3
w_i	1	2	3
p_i	1	4	6

	0	1	2	3
0	0	0	0	0
1	0	1	1	1
2	0	1	4	4
3	0	1	5	**6**

```
for r = 1 to C
    for c = 1 to n
        profit[r][c] = profit[r][c-1];
        if (w[c] <= r)
            if (profit[r][c] <
        profit[r-w[c]][c-1] + p[c])
                profit[r][c] =
                    profit[r-w[c]][c-1]
                    + p[c]
```

- The dynamic programming algorithm has a complexity of O(nC).
- An algorithm is polynomial time if it is a polynomial function of the <u>size</u> of the input.

  E.g. there are n weight numbers and n profit numbers

- An algorithm is pseudo-polynomial time if it is a polynomial function of the <u>value</u> of the input.

  E.g. there is only one number specifying C

- So the dynamic programming algorithm for knapsack problem is pseudo-polynomial.

Thanks!

Q & A

Appendix 1

```
getSequence(n m)        // get the LCS from hint array
{   s = empty stack;     // s stores the characters in LCS
    i = n;
    j = m;
    while (i ≠ 0  and  j ≠ 0)
        if (h[i][j] == '\')
            { s.push(x[i]);   i--;    j--;  }
        else if (h[i][j] == '|')
            i--;
        else    j--;
    pop and output from s;          }
```

Maximum no. of iterations: n+m. Complexity: O(n+m)

Bounded by a constant time

Example 2:

x	C	G	G	T	A	T
y	A	G	T	T	G	C

		A	**G**	**T**	**T**	**G**	**C**
	0	**0**	**0**	**0**	**0**	**0**	**0**
C	**0**	**0**	**0**	**0**	**0**	**0**	**1**
G	**0**	**0**	**1**	**1**	**1**	**1**	**1**
G	**0**	**0**	**1**	**1**	**1**	**2**	**2**
T	**0**	**0**	**1**	**2**	**2**	**2**	**2**
A	**0**	**1**	**1**	**2**	**2**	**2**	**2**
T	**0**	**1**	**1**	**2**	**3**	**3**	**3**

<table>
<tr><th></th><th></th><th>A</th><th>G</th><th>T</th><th>T</th><th>G</th><th>C</th></tr>
<tr><td></td><td>—</td><td>—</td><td>—</td><td>—</td><td>—</td><td>—</td><td>—</td></tr>
<tr><td>C</td><td>|</td><td>|</td><td>|</td><td>|</td><td>|</td><td>|</td><td>\</td></tr>
<tr><td>G</td><td>|</td><td>|</td><td>\</td><td>—</td><td>—</td><td>\</td><td>|</td></tr>
<tr><td>G</td><td>|</td><td>|</td><td>\</td><td>|</td><td>|</td><td>\</td><td>—</td></tr>
<tr><td>T</td><td>|</td><td>|</td><td>|</td><td>\</td><td>\</td><td>|</td><td>|</td></tr>
<tr><td>A</td><td>|</td><td>\</td><td>|</td><td>|</td><td>|</td><td>|</td><td>|</td></tr>
<tr><td>T</td><td>|</td><td>|</td><td>|</td><td>\</td><td>\</td><td>—</td><td>—</td></tr>
</table>

LCS(6,6) = 3

Example 2:

x	C	G	G	T	A	T
y	A	G	T	T	G	C

<table>
<tr><th></th><th></th><th>A</th><th>G</th><th>T</th><th>T</th><th>G</th><th>C</th></tr>
<tr><td></td><td>—</td><td>—</td><td>—</td><td>—</td><td>—</td><td>—</td><td>—</td></tr>
<tr><td>C</td><td>|</td><td>|</td><td>|</td><td>|</td><td>|</td><td>|</td><td>\</td></tr>
<tr><td>G</td><td>|</td><td>|</td><td>\</td><td>—</td><td>—</td><td>\</td><td>|</td></tr>
<tr><td>G</td><td>|</td><td>|</td><td>\</td><td>|</td><td>|</td><td>\</td><td>—</td></tr>
<tr><td>T</td><td>|</td><td>|</td><td>|</td><td>\</td><td>\</td><td>|</td><td>|</td></tr>
<tr><td>A</td><td>|</td><td>\</td><td>|</td><td>|</td><td>|</td><td>|</td><td>|</td></tr>
<tr><td>T</td><td>|</td><td>|</td><td>|</td><td>\</td><td>\</td><td>—</td><td>—</td></tr>
</table>

h(6,6) = '—'
h(6,5) = '—'
h(6,4) = '\' → **T**
h(5,3) = '|'
h(4,3) = '\' → **T**
h(3,2) = '\' → **G**
h(2,1) = '|'
h(1,1) = '|'
h(0,1) = '—'
end

LCS(6,6) = 3

The subsequence: **G T T**

Appendix 2

Step 4: Dictionary: cost[n+1][n+1]

cost

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

Not computed

Step 5

Order to solve the subproblems

cost

	0	**1**	**2**	**3**	**4**	**5**	**6**
0		0					
1			0				
2				0			
3					0		
4						0	
5							0
6							

Not computed

Use another array, last[n+1][n+1] to represent the index of the last multiplication to be done for a subproblem

Find the pattern

cost

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

Row number and column number differ by 2, row goes from 0 to 4

Row number and column number differ by 3, row goes from 0 to 3

Row number and column number differ by 4, row goes from 0 to 2

Find the pattern

cost

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

Row number and column number differ by 5, row goes from 0 to 1

Row number and column number differ by 6, row goes from 0 to 0

Thus, row number and column number differ by 2 to 6, within each difference, row goes from 0 to n minus this difference

Appendix 3

Example 2:
C = 20

	1	2	3	4
w_i	4	6	8	6
p_i	7	6	9	5

	0	1	2	3	4
0	0	0	0	0	0
2	0				
4	0				
6	0				
8	0				
10	0				
12	0				
14	0				
16	0				
20	0				

```
for c = 0 to n
    profit[0][c] = 0;
for r = 1 to C
    profit[r][0] = 0;
```

Not all rows are shown

Example 2:
C = 20

	1	2	3	4
w_i	4	6	8	6
p_i	7	6	9	5

	0	1	2	3	4
0	0	0	0	0	0
1	0	0	0	0	0
2	0	0	0	0	0
3	0	0	0	0	0
4	0	7	7	7	7
5	0	7	7	7	7
6	0	7	7	7	7
8	0	7	7	9	9
9	0	7	7	9	9
10	0	7	13	13	13
11	0	7	13	13	13

```
for r = 1 to C
    for c = 1 to n
        profit[r][c] = profit[r][c-1];
        if (w[c] <= r)
            if (profit[r][c] <
        profit[r-w[c]][c-1] + p[c])
                profit[r][c] =
                    profit[r-w[c]][c-1]
                    + p[c]
```

Example 2:
C = 20

	1	2	3	4
w_i	4	6	8	6
p_i	7	6	9	5

	0	1	2	3	4
10	0	7	13	13	13
11	0	7	13	13	13
12	0	7	13	16	16
13	0	7	13	16	16
14	0	7	13	16	16
15	0	7	13	16	16
16	0	7	13	16	18
17	0	7	13	16	18
18	0	7	13	22	22
19	0	7	13	22	22
20	0	7	13	22	**22**

```
for r = 1 to C
    for c = 1 to n
        profit[r][c] = profit[r][c-1];
        if (w[c] <= r)
            if (profit[r][c] <
        profit[r-w[c]][c-1] + p[c])
                profit[r][c] =
                    profit[r-w[c]][c-1]
                    + p[c]
```