# Introduction to Algorithms

## CS 430

## Lecture L17

ILLINOIS INSTITUTE OF TECHNOLOGY

By Lan Yao

# Outlines

- DP- Rod Cutting
- DP-LCS

# Dynamic Programming

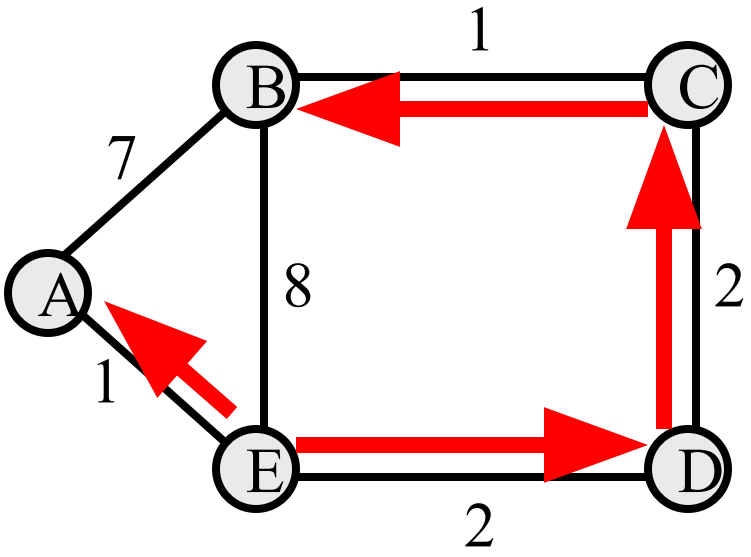
**What is Dynamic Programming?**
**Maximum/minimum/ ith smallest element---Order Statistics**

ex. Given a triangle, find the minimum-sum path from top to bottom. Each step you may move to adjacent numbers on the row below.

```
[2],
[3,4],
[6,5,7],
[4,1,8,3]
```

# eg. D-V Routing

- Solve the minimum cost of all possible routes *D* (destination, neighbor) from E, as a source of a packet, via its neighbors A, B and D
- Items on E in the routing table are extracted from *T*




*T* (des,nei)

| From E | Via A | Via B | Via D |
|---|---|---|---|
| To A | 1 | 15 | 5 |
| To B | 8 | 8 | 5 |
| To C | 6 | 9 | 4 |
| To D | 4 | 11 | 2 |

Routing table

| Des | D,V |
|---|---|
| A | 1,A |
| B | 5,D |
| C | 4,D |
| D | 2,D |

- Dynamic Programming (DP) is an algorithmic technique for solving an optimization problem by breaking it down into simpler subproblems and utilizing the fact that the optimal solution to the overall problem depends upon the optimal solution to its subproblems.

# Rod Cutting

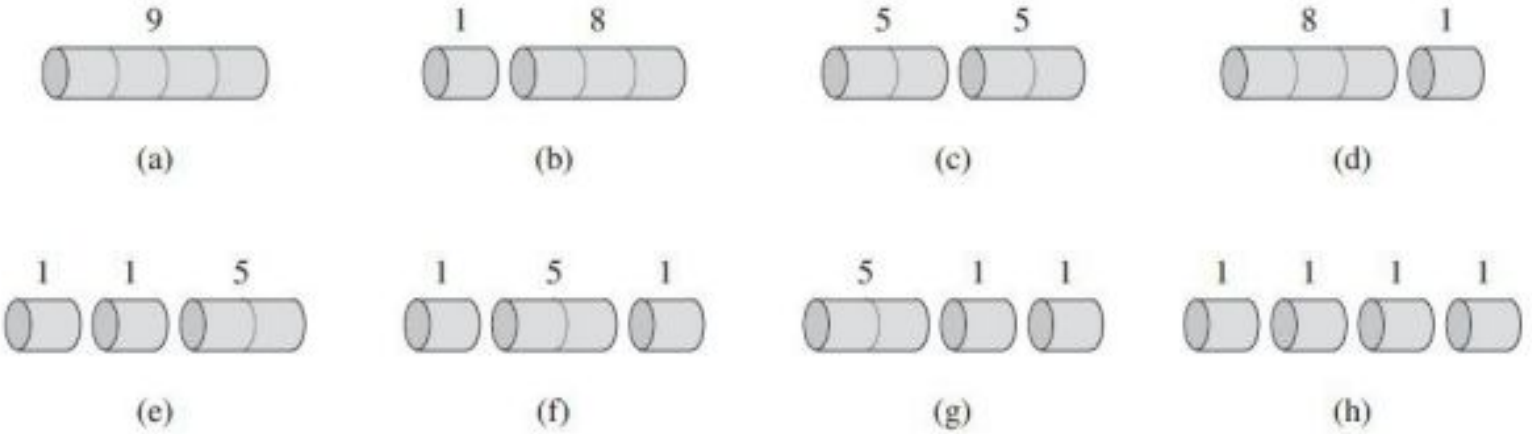
ex: optimal revenue of Rod Cutting

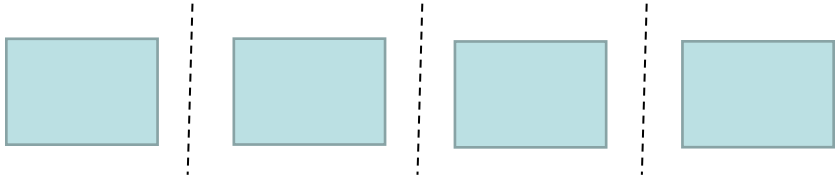
- The length of rod is 4. Cut it into some segments such that the revenue is optimized

| length $i$ | 1 | 2 | 3 | 4 |
|---|---|---|---|---|
| price $p_i$ | 1 | 5 | 8 | 9 |

- The number of potential cuttings=?

# Rod Cutting: n=4, 8 Possible Ways



cut/ no cut cut/ no cut cut/ no cut
1 / 0 1 / 0 1 / 0

The number of potential cuttings
$=2^{n-1}$

- The dynamic programming like the divide-conquer method solves problems by combining the solutions to such problems. We usually apply dynamic programming to optimization problems.
- Such problems can have many possible solutions. Each solution has a value. And we wish to find a solution with the optimal (minimum / maximum) value. We call such solution optimal solution. (example on an quadratic function)

- example for optimal solution: quadratic function
  - $y=ax^2+bx+c$. $y'=2ax+b$
  - to solve the max/min of y, $2ax+b=0 \Rightarrow x=-b/2a$

    bring it in, then $y=(-b^2+4ac)/4a$.
  - when $a>0$, it is the min; when $a<0$, it is the max.

# • Generalize our Rod Cutting example as:

- Given a rod of length n inches and a table of prices pi for i=1,2,...,n, determine the maximal revenue $r_n$ obtainable by cutting up the rod and selling the pieces. If the price $p_n$ for a rod of length n is large enough, an optimal solution may require no cutting at all.
- If an optimal solution cuts the rod into k pieces, for some 1=<k<=n, then an optimal decomposition, $n=i_1+i_2+...+i_k$, of the rod with pieces of length $i_1,i_2,...,i_k$ provides maximal corresponding revenue, $r_n=p_i+p_2+...+p_k$.
- Based on the following table, we have the optimal revenues as:

| length $i$ | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|---|---|---|---|---|---|---|---|---|---|---|
| price $p_i$ | 1 | 5 | 8 | 9 | 10 | 17 | 17 | 20 | 24 | 30 |

$r_1=1$

$r_2=5$

$r_3=8$

$r_4=p_2+p_2=10$

... ...

$r_7=p_1+p_6=p_2+p_2+p_3=18$ $r_5=13$ $(p_2+p_3)$, $r_6=17$ $(p_6)$

... ...

4
4--9
3+1---8+1=9
2+2---5+5=10
1+3---1+8=9
n

$r_n=\max\{p_n, r_1+r_{n-1}, r_2+r_{n-2}, ..., r_{n-1}+r_1\} \Leftrightarrow r_n = \max_{1\le i\le n}(p_i + r_{n-i})$

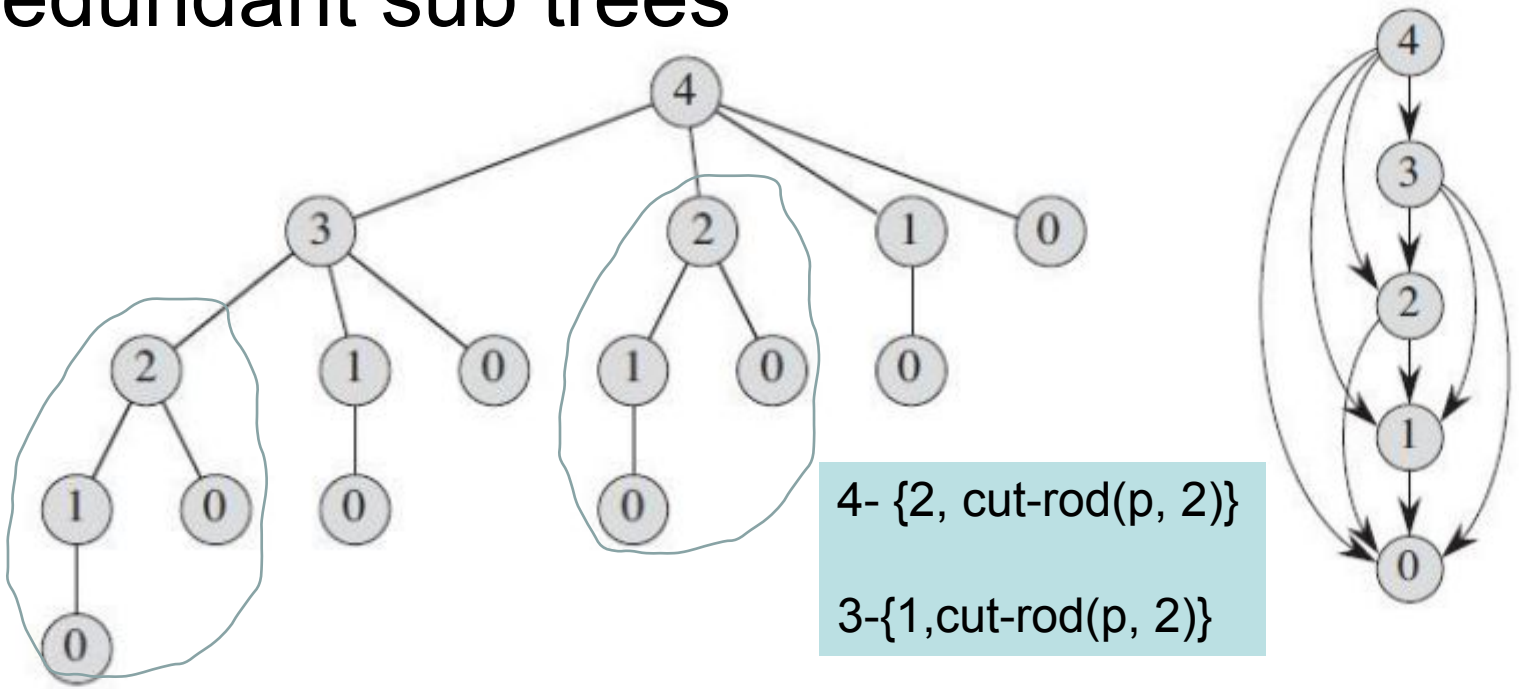
- We follow a sequence of four steps:
  - Characterize the structure of an optimal solution;
  - Recursively define the value of an optimal solution;
  - Compute the value of an optimal solution;
  - Construct an optimal solution from computed information. (example on optimal package delay in a network)

- Decomposition of the problem: first piece of length i and then reminder of length n-i.
- Algorithm of rod cutting:

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

- It returns the optimal revenue when the length is n and the price table is p.

- Complexity analysis on recursion tree: redundant sub trees




- Assume that T(0)=1 (The initial 1 is for the call of the root), then based on the recursion tree: $T(n) = 1 + \sum_{j=0}^{n-1} T(j)$

- The proof of this complexity: the closed form for this recursion tree is $2^n$.
- Cut-Rod (p, n) is not efficient because of redundant sub trees.
  - T(0)=1
  - T(1)=1+1=2
  - T(2)=1+T(0)+T(1)=4
  - T(3)=1+T(0)+T(1)+T(2)=8
  - $T(n)=2^n$

- The proof of this complexity: the closed form for this recursion tree is $2^n$.

Proof: (inductive proof)

Suppose that $T(n) = 1 + \sum_{j=0}^{n-1} T(j) = 2^n$, then

$$T(n+1) = 1 + \sum_{j=0}^{n} T(j) = 1 + \sum_{j=0}^{n-1} T(j) + T(n)$$

$$= 1 + \sum_{j=0}^{n-1} T(j) + 1 + \sum_{j=0}^{n-1} T(j) = 2 \times [1 + \sum_{j=0}^{n-1} T(j)]$$

$$= 2 \times 2^n = 2^{n+1}$$

QED.

- Dynamic Programming with additional memory to save computation time.
- Two equivalent ways to reduce the repeated computation: top-down and bottom-up.

- Top-down
  - We write the procedure recursively in a natural manner, but modified to save the results of sub problem.

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

```
MEMOIZED-CUT-ROD-AUX(p, n, r)
1  if r[n] ≥ 0
2      return r[n]
3  if n == 0
4      q = 0
5  else q = -∞
6      for i = 1 to n
7          q = max(q, p[i] + MEMOIZED-CUT-ROD-AUX(p, n - i, r))
8  r[n] = q
9  return q
```

- Bottom-up
  - When solving a particular sub problem, we have already solved all of the smaller sub problems its solutions depend upon, and we have saved their solutions.

```
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 = -∞
5      for i = 1 to j
6          q = max(q, p[i] + r[j - i])
7      r[j] = q
8  return r[n]
```

- Cutting strategy for optimal revenue
  - we should save the corresponding cutting to the revenue

| $i$ | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| $r[i]$ | 0 | 1 | 5 | 8 | 10 | 13 | 17 | 18 | 22 | 25 | 30 |
| $s[i]$ | 0 | 1 | 2 | 3 | 2 | 2 | 6 | 1 | 2 | 3 | 10 |

The length of the first piece

```
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 = -∞
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 new arrays
2  r[0] = 0
3  for j = 1 to n
4      q = -∞
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 the cutting strategy

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

# LCS

- Application background
  - A strand of DNA consists of a string of molecules called bases, where the possible bases are Adenine (A), Cytosine (C), Guanine (G), Thymine (T). the DNA of one organism may be : S1=ACCGTGACC....; and DNA in another organism S2 may be GTCGAGTT.....

- We can measure the similarity of strands S1 and S2 by finding a third strand S3 in which the bases appear in each of S1 and S2. These bases must appear in the same order, but not necessarily consecutively.

- The longer the strand S3 we can find, the most similar S1 and S2 are.

- Definitions--subsequence and common subsequence

Given a sequence $X=(x_1, x_2, \ldots, x_n)$ and $Z=(z_1, z_2, \ldots, z_k)$ is a subsequence of X if there exists a strictly increasing sequence $(i_1,i_2,\ldots,i_k)$ of indices of X, such that for all $j=1,2,\ldots,k$, we have $x_{ij}=z_j$.

eg. X=(A,B,C,B,D,A,B). Z=(B,C,D,B) is subsequence of X with the corresponding index sequence (2,3,5,7).

Given two sequences X and Y, we say that a sequence Z is a **common subsequence** of X and Y if Z is a subsequence of both of X and Y.

eg. X=(A,B,C,B,D,A,B)

Y=(B,D,C,A,B,A)

Z=(B,C,A)

Longest common subsequence (LCS)

eg. X=(A,B,C,B,D,A,B)

Y=(B,D,C,A,B,A)

Z=(B,C,B,A) or (B,D,A,B)

# • Four steps of DP to solve LCS

- The LCS problem

We are given two sequences $X=(x_1,x_2,\ldots,x_n)$ and $Y=(y_1,y_2,\ldots,y_m)$, we wish to find the maximal length common subsequence of X and Y.

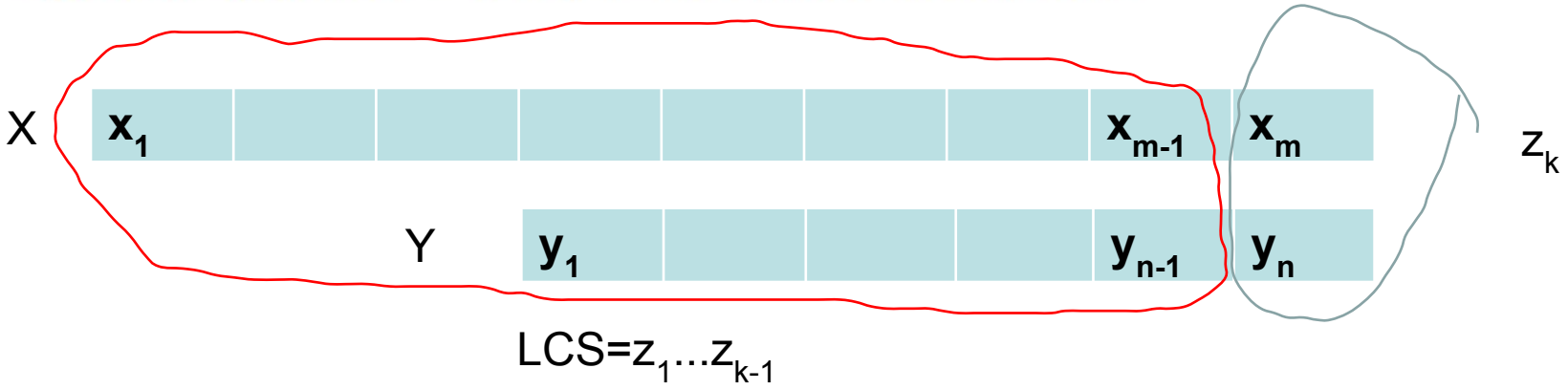
Consider each subsequence of X corresponding to a subset of the indices {1,2,…,n}, there are $2^n$ subsequences.

- Optimal substructure of LCS

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

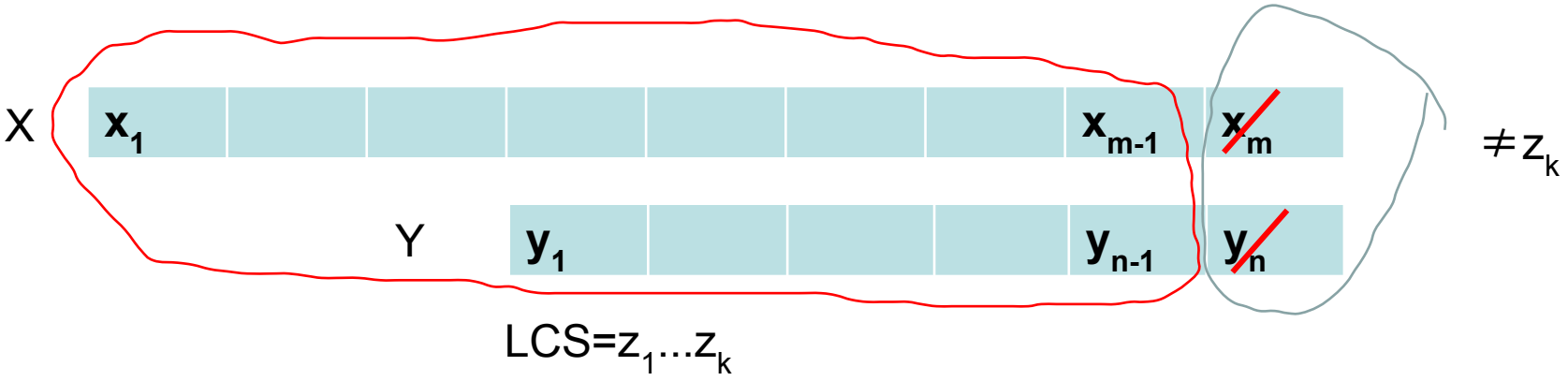
(1) If $x_m=y_n$ then, $z_k = x_m=y_n$ and $Z_{k-1}$ is an LCS of $X_{m-1}$ and $Y_{n-1}$.

(2) If $x_m \neq y_n$, then $z_k \neq x_m$ implies that Z is an LCS of of $X_{m-1}$ and Y.



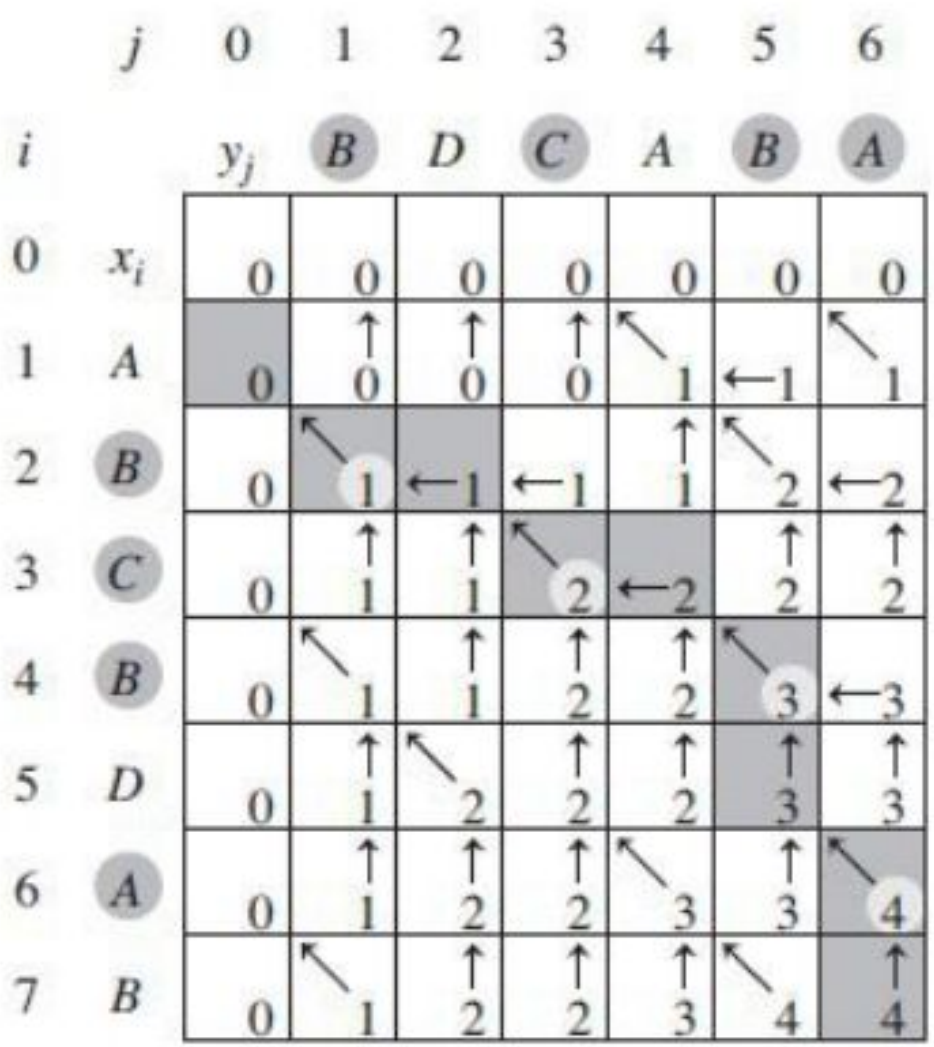
(3) If $x_m \neq y_n$, then $z_k \neq y_n$ implies that Z is an LCS of of X and $Y_{n-1}$.

- Conclusion : an LCS of two sequences contains within it an LCS of prefixes of the two sequences.
  - If $x_m=y_n$, we must find an LCS of $X_{m-1}$ and $Y_{n-1}$. Appending $x_m=y_n$ to this LCS yielding an LCS of X an Y.
  - If $x_m \neq y_n$, we must solve two sub problems:
    - Finding an LCS of $X_{m-1}$ and Y;
    - Finding an LCS of X and $Y_{n-1}$;
    - Whichever of these two LCSs is longer is an LCS of X and Y.



- Algorithm of finding the length LCS

  - c(i,j) is the length of an LCS of two sequences $X_i$ and $Y_j$;
  - $$c[i,j] = \begin{cases} 0, \text{ if } i=0 \text{ or } j=0; \\ c[i-1,j-1]+1, \text{ if } i,j>0 \text{ and } x_i=y_j; \\ \\ \text{Max}(c[i,j-1], c[i-1,j]) \text{ if } x_i \neq y_j. \end{cases}$$


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

- b(i,j) points to the table entry to corresponding to the optimal subproblem solution chosen when computing c[i,j].

```
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
 5      c[i, 0] = 0
 6  for j = 0 to n
 7      c[0, j] = 0
 8  for i = 1 to m
 9      for j = 1 to n
10          if x_i == y_j
11              c[i, j] = c[i − 1, j − 1] + 1
12              b[i, j] = "↖"
13          elseif c[i − 1, j] ≥ c[i, j − 1]
14              c[i, j] = c[i − 1, j]
15              b[i, j] = "↑"
16          else c[i, j] = c[i, j − 1]
17              b[i, j] = "←"
18  return c and b
```

- Algorithm to print the LCS

## Printing the Resulting LCS

```
PRINT-LCS(b, X, i, j)
1  if i == 0 or j == 0
2       return
3  if b[i, j] == "↖"
4       PRINT-LCS(b, X, i − 1, j − 1)
5       print x_i
6  elseif b[i, j] == "↑"
7       PRINT-LCS(b, X, i − 1, j)
8  else PRINT-LCS(b, X, i, j − 1)
```