Algorithm Design Techniques

Classification

- Implementation Method
- Design Method
- Other Classifications

Implementation Method

- Recursion or Iteration
- Procedural or Declarative
  - For example: C, PHP/ SQL
- Serial or parallel or Distributed
- Deterministic or Non-Deterministic
- Exact or Approximate

Design Method

- Greedy Method
- Divide and Conquer
- Dynamic Programming
- Linear Programming
  - A method to allocate scarce resources to competing activities in an optimal manner when the problem can be expressed using a linear objective function and linear inequality constraints.
- Reduction

Other Classifications

- Research Area
- Complexity
- Randomized Algorithm

Greedy Algorithm

- Decision is made that is good at that point, without bothering about future.
- i.e local best is chosen.
- Two basic properties:
  - Greedy choice property
  - Optimal substructure

Greedy Algorithm

- Advantages
  - Straightforward
  - Easy to understand
  - Easy to code
- Disadvantges
  - No guarantee that making locally optimal improvement will give globally optimal solution.

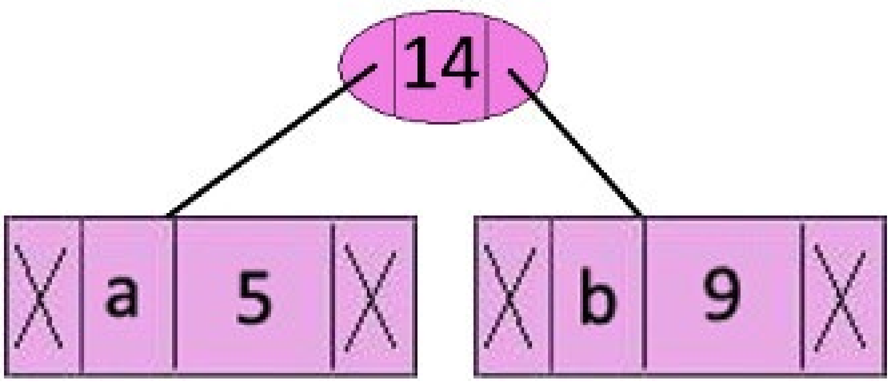
Greedy Algorithm

- Applications
  - Selection sort, topological sort
  - Priority queue
  - Huffman coding
  - Prim’s and kruskal’s algorithm
  - Job scheduling
  - Shortest path in weighted graph[Dijkstra’s]

Huffman code

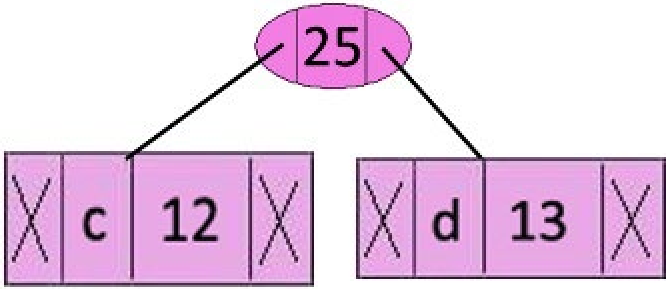
Character	Frequency
a	5
b	9
c	12
d	13
e	16
f	45

Huffman code



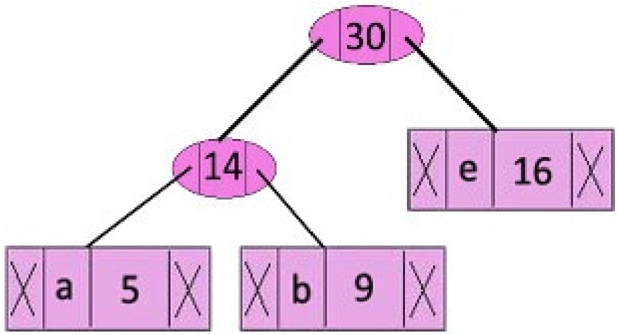
Huffman code

Character	Frequency
c	12
d	13
Internal node	14
e	16
f	45



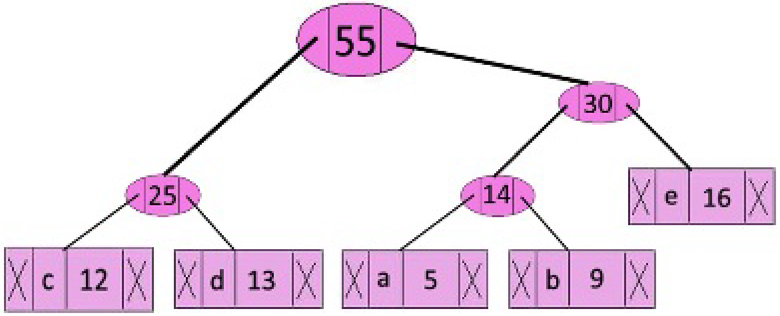
Huffman code

Character	Frequency
Internal node	14
e	16
Internal node	25
f	45



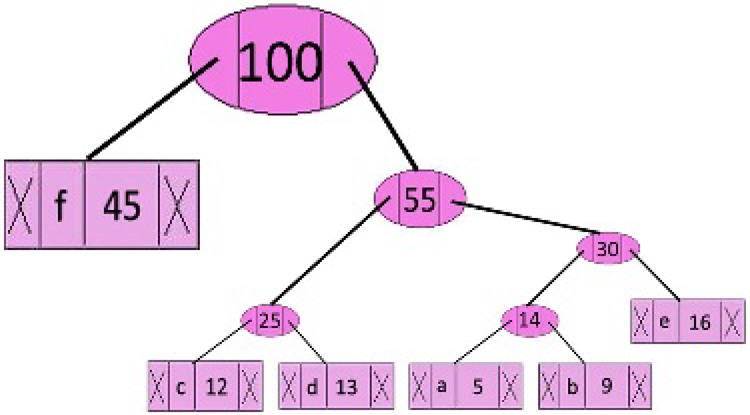
Huffman code

Character	Frequency
Internal node	25
Internal node	30
f	45

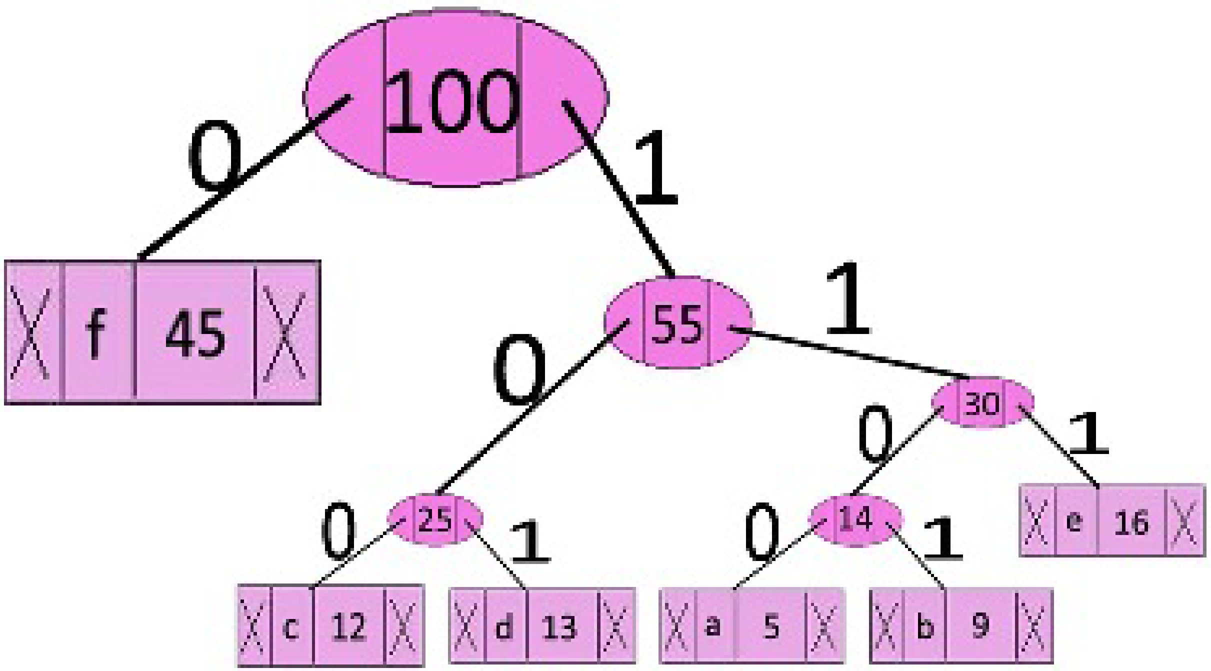


Huffman code

Character	Frequency
f	45
Internal node	55



Huffman code



Character	Code-word	Frequency
f	0	5
c	100	9
d	101	12
a	1100	13
b	1101	16
e	111	45

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.

Properties of DP

- Optimal substructure
- Overlapping sub problems

Fibonacci series

- Fib(n)=0, for n=0

  = 1, for n=1

  =fib(n-1) +fib(n-2), for n>1

  fib(5)

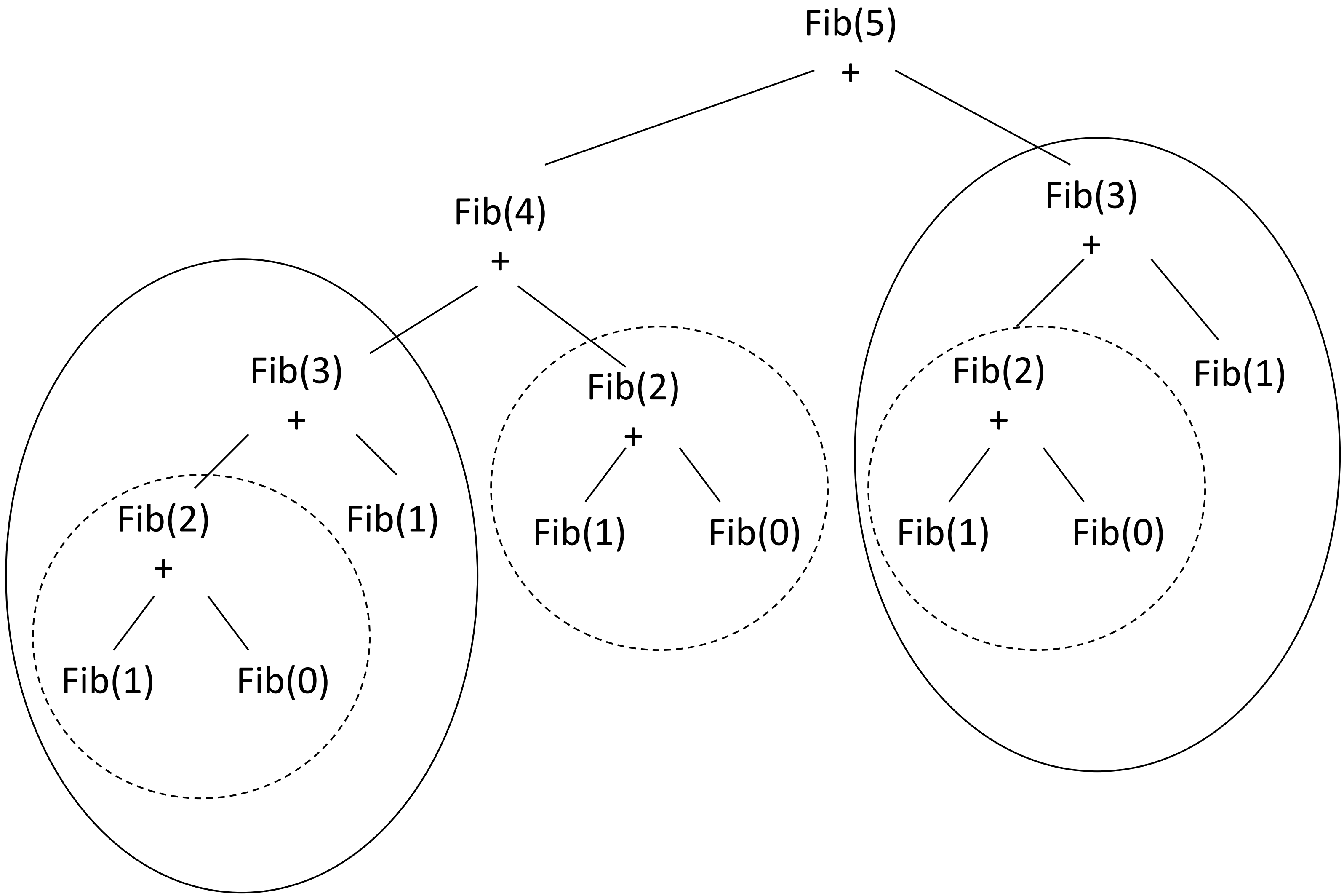
  fib(4)+fib(3)

  (fib(3)+fib(2))+(fib(2)+fib(1))

  ((fib(2)+fib(1))+(fib(1)+fib(0)))+((fib(1)+fib(0))+fib(1))

  (((fib(1)+fib(0))+fib(1))+(fib(1)+fib(0)))+((fib(1)+fib(0))+fib(1))

Fibonacci Numbers



Approaches of DP

- Two approaches:
  - Top-down (Memoization)
    - Problem is broken in subproblems
    - Each subproblem is solved and remembered
  - Bottom-up (Tabulation)
    - Evaluate function starting with smallest possible input argument value.
    - Increase each input argument value slowly
    - Store all computed values in a table.

- DP=Overlapping subproblems+Memoization/Tabulation

Memoization solution

```
fibTable={1:0,2:1}
def fibo(n):
    if n<=2:
        return 1
    if n in fibTable:
        return fibTable[n]
    else:
        fibTable[n]=fibo(n-1)+fibo(n-2)
        return fibTable[n]
```

Tabulation solution

```
def fibo(n):
    fibTable=[0,1]
    for i in range(2,n+1):
        fibTable.append(fibTable[i-1]+fibTable[i-2])
    return fibTable[n]
```

Longest Common Subsequence (LCS)

Application: comparison of two DNA strings

Ex: X= {A B C B D A B }, Y= {B D C A B A}

Longest Common Subsequence:

X = A **B** **C** **B** D **A** B

Y = **B** D **C** A **B** **A**

Brute force algorithm would compare each subsequence of X with the symbols in Y

Subsequence need not be consecutive, but must be in order.

LCS Example

We’ll see how LCS algorithm works on the following example:

- X = ABCB
- Y = BDCAB

What is the Longest Common Subsequence of X and Y?

LCS(X, Y) = BCB

X = A **B** **C** **B**

Y = **B** D **C** A **B**

LCS Example (0)

ABCB
BDCAB

j		0	1	2	3	4	5
i		Yj	**B**	**D**	**C**	**A**	**B**
0	Xi						
1	**A**						
2	**B**						
3	**C**						
4	**B**						

X = ABCB; m = |X| = 4
Y = BDCAB; n = |Y| = 5
Allocate array c[5,4]

LCS Example (1)

ABCB
BDCAB

	j	0	1	2	3	4	5
i		Yj	**B**	**D**	**C**	**A**	**B**
0	Xi	**0**	**0**	**0**	**0**	**0**	**0**
1	**A**	**0**					
2	**B**	**0**					
3	**C**	**0**					
4	**B**	**0**					

for $i = 1$ to m $\quad c[i,0] = 0$
for $j = 1$ to n $\quad c[0,j] = 0$

LCS Example (2)

ABCB
BDCAB

		j	0	1	2	3	4	5
i			Yj	B	D	C	A	B
0	Xi		0	0	0	0	0	0
1	A		0	0				
2	B		0					
3	C		0					
4	B		0					

if ($X_i == Y_j$)
$c[i,j] = c[i-1,j-1] + 1$
else $c[i,j] = \max(c[i-1,j], c[i,j-1])$

LCS Example (3)

ABCB
BDCAB

	j	0	1	2	3	4	5
i		Yj	**B**	**D**	**C**	**A**	**B**
0	Xi	**0**	**0**	**0**	**0**	**0**	**0**
1	**A**	**0**	**0**	**0**	**0**		
2	**B**	**0**					
3	**C**	**0**					
4	**B**	**0**					

if (X_i == Y_j)
$c[i,j] = c[i-1,j-1] + 1$
else $c[i,j] = \max(c[i-1,j], c[i,j-1])$

LCS Example (4)

ABCB
BDCAB

	j	0	1	2	3	4	5
i		Yj	B	D	C	A	B
0	Xi	0	0	0	0	0	0
1	A	0	0	0	0	1	
2	B	0					
3	C	0					
4	B	0					

if ($X_i == Y_j$)

$c[i,j] = c[i-1,j-1] + 1$

else $c[i,j] = \max(c[i-1,j], c[i,j-1])$

LCS Example (5)

ABCB
BDCAB

	j	0	1	2	3	4	5
i		Yj	**B**	**D**	**C**	**A**	**B**
0	Xi	**0**	**0**	**0**	**0**	**0**	**0**
1	**A**	**0**	**0**	**0**	**0**	**1**	→ **1**
2	**B**	**0**					
3	**C**	**0**					
4	**B**	**0**					

if ($X_i == Y_j$)
$c[i,j] = c[i-1,j-1] + 1$
else $c[i,j] = \max(c[i-1,j], c[i,j-1])$

LCS Example (6)

ABCB
BDCAB

i		j	0	1	2	3	4	5
		Yj		**B**	**D**	**C**	**A**	**B**
0	Xi		**0**	**0**	**0**	**0**	**0**	**0**
1	**A**		**0**	**0**	**0**	**0**	**1**	**1**
2	**B**		**0**	**1**				
3	**C**		**0**					
4	**B**		**0**					

if ($X_i == Y_j$)
$c[i,j] = c[i-1,j-1] + 1$
else $c[i,j] = \max(c[i-1,j], c[i,j-1])$

LCS Example (7)

ABCB
BDCAB

	j	0	1	2	3	4	5
i		Yj	B	D	C	A	B
0	Xi	**0**	**0**	**0**	**0**	**0**	**0**
1	A	**0**	**0**	**0**	**0**	**1**	**1**
2	B	**0**	**1**	**1**	**1**	**1**	
3	C	**0**					
4	B	**0**					

if ($X_i == Y_j$)
$c[i,j] = c[i-1,j-1] + 1$
else $c[i,j] = \max(c[i-1,j], c[i,j-1])$

LCS Example (8)

ABCB
BDCAB

	j	0	1	2	3	4	5
i		Yj	**B**	**D**	**C**	**A**	**B**
0	Xi	**0**	**0**	**0**	**0**	**0**	**0**
1	**A**	**0**	**0**	**0**	**0**	**1**	**1**
2	**B**	**0**	**1**	**1**	**1**	**1**	**2**
3	**C**	**0**					
4	**B**	**0**					

if (X_i == Y_j)
    c[i,j] = c[i-1,j-1] + 1
else c[i,j] = max(c[i-1,j], c[i,j-1])

LCS Example (10)

ABCB
BDCAB

		j	0	1	2	3	4	5
i		Yj		**B**	**D**	**C**	**A**	**B**
0	Xi		**0**	**0**	**0**	**0**	**0**	**0**
1	**A**		**0**	**0**	**0**	**0**	**1**	**1**
2	**B**		**0**	**1**	**1**	**1**	**1**	**2**
3	**C**		**0**	**1**	**1**			
4	**B**		**0**					

if ($X_i == Y_j$)
$c[i,j] = c[i-1,j-1] + 1$
else $c[i,j] = \max(c[i-1,j], c[i,j-1])$

LCS Example (11)

ABCB
BDCAB

	j	0	1	2	3	4	5
i		Yj	**B**	**D**	**C**	**A**	**B**
0	Xi	**0**	**0**	**0**	**0**	**0**	**0**
1	**A**	**0**	**0**	**0**	**0**	**1**	**1**
2	**B**	**0**	**1**	**1**	**1**	**1**	**2**
3	**C**	**0**	**1**	**1**	**2**		
4	**B**	**0**					

if ($X_i == Y_j$)
$c[i,j] = c[i-1,j-1] + 1$
else $c[i,j] = \max(c[i-1,j], c[i,j-1])$

LCS Example (12)

ABCB
BDCAB

		j	0	1	2	3	4	5
i			Yj	**B**	**D**	**C**	**A**	**B**
0	Xi		**0**	**0**	**0**	**0**	**0**	**0**
1	**A**		**0**	**0**	**0**	**0**	**1**	**1**
2	**B**		**0**	**1**	**1**	**1**	**1**	**2**
3	**C**		**0**	**1**	**1**	**2**	**2**	**2**
4	**B**		**0**					

if (X_i == Y_j)
$c[i,j] = c[i-1,j-1] + 1$
else $c[i,j] = \max(c[i-1,j], c[i,j-1])$

LCS Example (13)

ABCB
BDCAB

i		j	0	1	2	3	4	5
		Yj		B	D	C	A	B
0	Xi		**0**	**0**	**0**	**0**	**0**	**0**
1	A		**0**	**0**	**0**	**0**	**1**	**1**
2	B		**0**	**1**	**1**	**1**	**1**	**2**
3	C		**0**	**1**	**1**	**2**	**2**	**2**
4	B		**0**	**1**				

if ($X_i == Y_j$)
$c[i,j] = c[i-1,j-1] + 1$
else $c[i,j] = \max(c[i-1,j], c[i,j-1])$

LCS Example (14)

ABCB
BDCAB

	j	0	1	2	3	4	5
i		Yj	**B**	**D**	**C**	**A**	**B**
0	Xi	**0**	**0**	**0**	**0**	**0**	**0**
1	**A**	**0**	**0**	**0**	**0**	**1**	**1**
2	**B**	**0**	**1**	**1**	**1**	**1**	**2**
3	**C**	**0**	**1**	**1**	**2**	**2**	**2**
4	**B**	**0**	**1**	**1**	**2**	**2**	

if ($X_i == Y_j$)
$c[i,j] = c[i-1,j-1] + 1$
else $c[i,j] = \max(c[i-1,j], c[i,j-1])$

LCS Example (15)

ABCB
BDCAB

	j	0	1	2	3	4	5
i		Yj	**B**	**D**	**C**	**A**	**B**
0	Xi	**0**	**0**	**0**	**0**	**0**	**0**
1	**A**	**0**	**0**	**0**	**0**	**1**	**1**
2	**B**	**0**	**1**	**1**	**1**	**1**	**2**
3	**C**	**0**	**1**	**1**	**2**	**2**	**2**
4	**B**	**0**	**1**	**1**	**2**	**2**	3

if ($X_i == Y_j$)
$c[i,j] = c[i-1,j-1] + 1$
else $c[i,j] = \max(c[i-1,j], c[i,j-1])$

How to find actual LCS - continued

- Remember that

$$c[i,j]=\begin{cases} c[i-1,j-1]+1 & \text{if } x[i]=y[j], \\ \max(c[i,j-1],c[i-1,j]) & \text{otherwise} \end{cases}$$

- So we can start from *c[m,n]* and go backwards
- Whenever *c[i,j] = c[i-1, j-1]+1*, remember *x[i]* (because *x[i]* is a part of LCS)
- When i=0 or j=0 (i.e. we reached the beginning), output remembered letters in reverse order

Finding LCS

i		j: 0	1	2	3	4	5
		Yj	**B**	**D**	**C**	**A**	**B**
0	Xi	**0**	**0**	**0**	**0**	**0**	**0**
1	**A**	**0**	**0**	**0**	**0**	**1**	**1**
2	**B**	**0**	**1** ←	**1**	**1**	**1**	**2**
3	**C**	**0**	**1**	**1**	**2** ←	**2**	**2**
4	**B**	**0**	**1**	**1**	**2**	**2**	**3**

Finding LCS (2)

i	j	0	1	2	3	4	5
		Yj	**B**	**D**	**C**	**A**	**B**
0	Xi	**0**	**0**	**0**	**0**	**0**	**0**
1	**A**	**0**	**0**	**0**	**0**	**1**	**1**
2	**B**	**0**	**1**	**1**	**1**	**1**	**2**
3	**C**	**0**	**1**	**1**	**2**	**2**	**2**
4	**B**	**0**	**1**	**1**	**2**	**2**	**3**

LCS (reversed order): **B C B**

LCS (straight order): **B C B**

(this string turned out to be a palindrome)