Algorithm Design

- Divide-and-Conquer
  – Break Problem into Independent Subproblems
- Dynamic Programming
  – Break problem into Overlapping Subproblems.

Dynamic Programming

- Subproblems Share Subproblems
- Dynamic Programming Algorithm:
  - solves each subproblem once
  - Save subproblem results in table for re-use.

Designing Dynamic Programming Algorithm

1. Characterize the structure of an optimal solution.
2. Recursively define the value of an optimal solution.
3. Compute the value of an optimal solution, typically in a bottom-up fashion.
4. Construct an optimal solution from computed information.
   – Step 4 not needed if only computing optimum value.

Rod-Cutting Problem

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

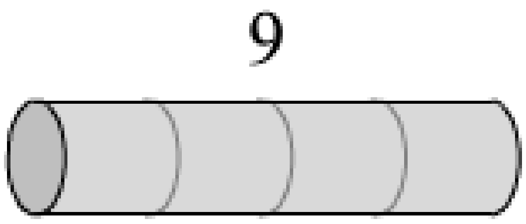
- Rod Prices

Rod-Cutting Problem

- Rod Prices

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

- Rod Length = 4



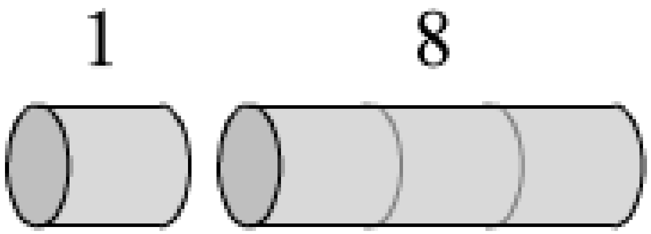
(a)

Rod-Cutting Problem

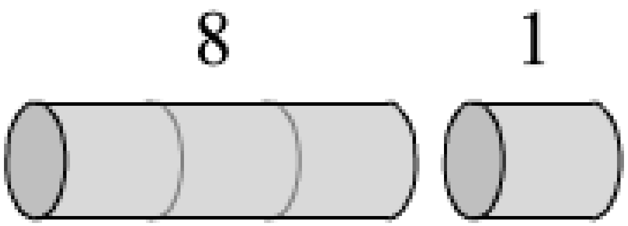
- Rod Prices

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

- Rod Length = 4



(b)



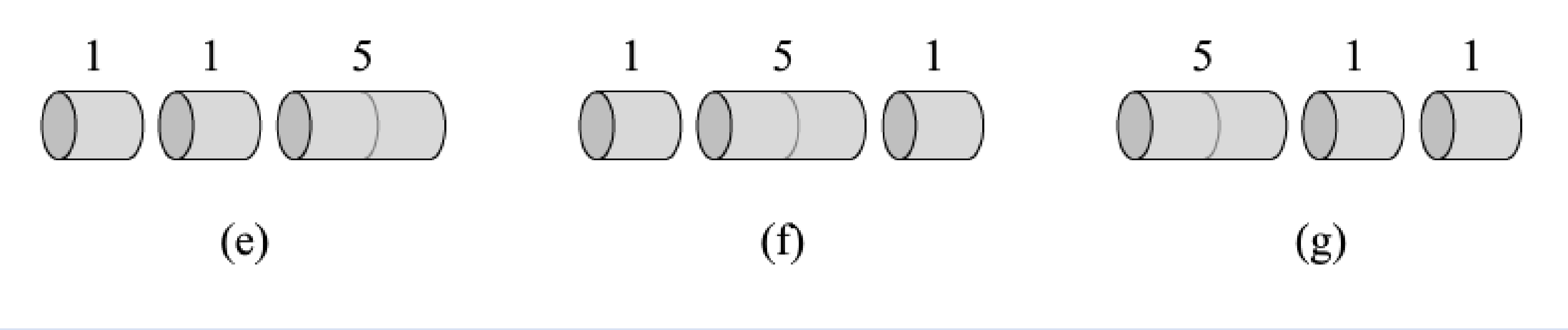
(d)

Rod-Cutting Problem

- Rod Prices

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

- Rod Length = 4

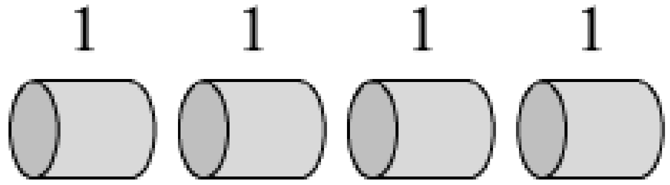


Rod-Cutting Problem

- Rod Prices

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

- Rod Length = 4



(h)

Formal Definition

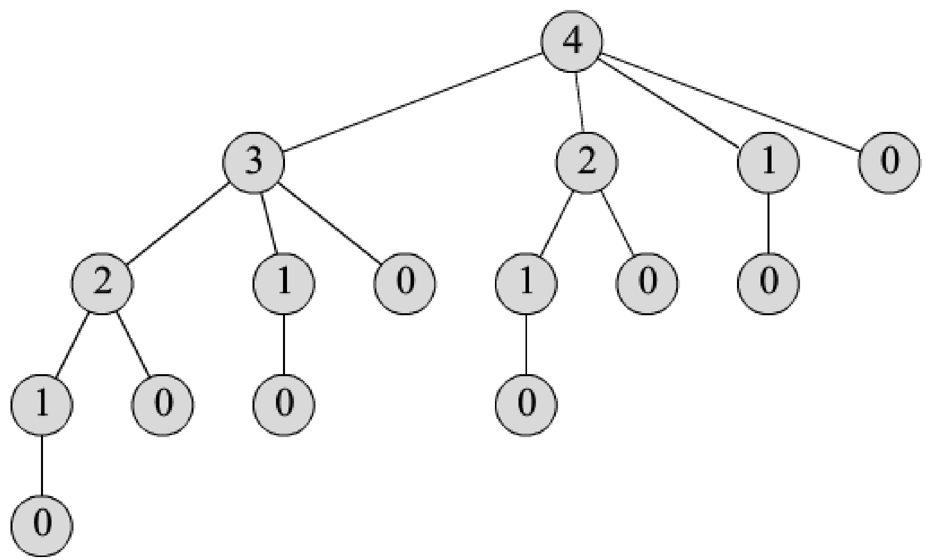
$$r_n = \max\left(p_n, r_1 + r_{n-1}, r_2 + r_{n-2}, \ldots, r_{n-1} + r_1\right) .$$

$$r_n = \max_{1 \le i \le n} \left(p_i + r_{n-i}\right) .$$

Recursive Top-Down

```
CUT-ROD(p, n)
if n == 0
    return 0
q = −∞
for i = 1 to n
    q = max(q, p[i] + CUT-ROD(p, n − i))
return q
```

```
CUT-ROD(p, n)
if n == 0
    return 0
q = −∞
for i = 1 to n
    q = max(q, p[i] + CUT-ROD(p, n − i))
return q
```

Cut-Rod Improvement

- Time-Memory Tradeoff
  - Remember subproblem results

```
CUT-ROD(p, n)
if n == 0
    return 0
q = −∞
for i = 1 to n
    q = max(q, p[i] + CUT-ROD(p, n − i))
return q
```

Cut-Rod Improvement

- Top-Down w/ Memoization
  - Remember subproblem results

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

[2]This is not a misspelling. The word really is *memoization*, not *memorization*. *Memoization* comes from *memo*, since the technique consists of recording a value so that we can look it up later.

Cut-Rod Improvement

- Top-Down w/ Memoization
  - Remember subproblem results

```
MEMOIZED-CUT-ROD-AUX(p, n, r)
if r[n] ≥ 0
    return r[n]
if n == 0
    q = 0
else q = −∞
    for i = 1 to n
        q = max(q, p[i] + MEMOIZED-CUT-ROD-AUX(p, n − i, r))
r[n] = q
return q
```

Cut-Rod Improvement

- Bottom-Up w/ Solution

EXTENDED-BOTTOM-UP-CUT-ROD(p, n)

```
let r[0..n] and s[0..n] be new arrays
r[0] = 0
for j = 1 to n
    q = −∞
    for i = 1 to j
        if q < p[i] + r[j − i]
            q = p[i] + r[j − i]
            s[j] = i
    r[j] = q
return r and s
```

MEMOIZ

```
if r[n]
    r
if n =
    q
else q
    f
                              − i, r))
r[n] =
return q
```

Cut-Rod Improvement

- Bottom-Up w/ Solution

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

In our rod-cutting example, the call EXTENDED-BOTTOM-UP-CUT-ROD$(p, 10)$ would return the following arrays:

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

Sequence alignment

- Compare two strings to see if they are similar
  - We need to define similarity
  - Very useful in many applications
  - Comparing DNA sequences, articles, source code, etc.

  - Example: Longest Common Subsequence problem (LCS)

Common subsequence

- A subsequence of a string is the string with zero or more chars left out
- A common subsequence of two strings:
  - A subsequence of both strings
  - Ex: x = {A B C B D A B }, y = {B D C A B A}
  - {B C} and {A A} are both common subsequences of x and y

Longest Common Subsequence

- Given two sequences $x[1 \,.\,.\, m]$ and $y[1 \,.\,.\, n]$, find a longest subsequence common to them both.

"a" *not* "the"

x:	A	B	C	B	D	A	B
y:	B	D	C	A	B	A	

BCBA = LCS(x, y)

functional notation, but not a function

Brute-force LCS algorithm

Check every subsequence of $x[1 \,.\,.\, m]$ to see if it is also a subsequence of $y[1 \,.\,.\, n]$.

Analysis

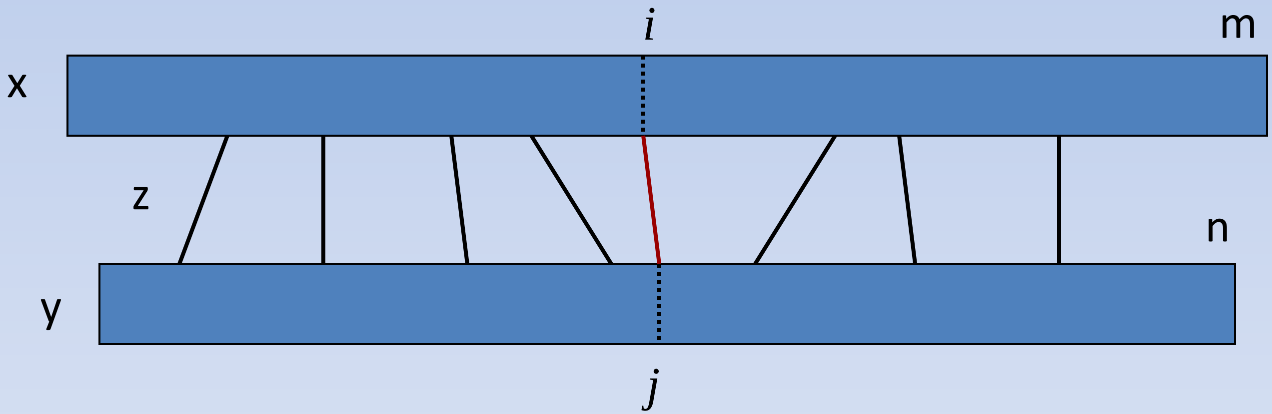
- 2^m subsequences of x (each bit-vector of length m determines a distinct subsequence of x).
- Hence, the runtime would be exponential !

Towards a better algorithm: a DP strategy

- Key: optimal substructure and overlapping sub-problems
- First we'll find the length of LCS. Later we'll modify the algorithm to find LCS itself.

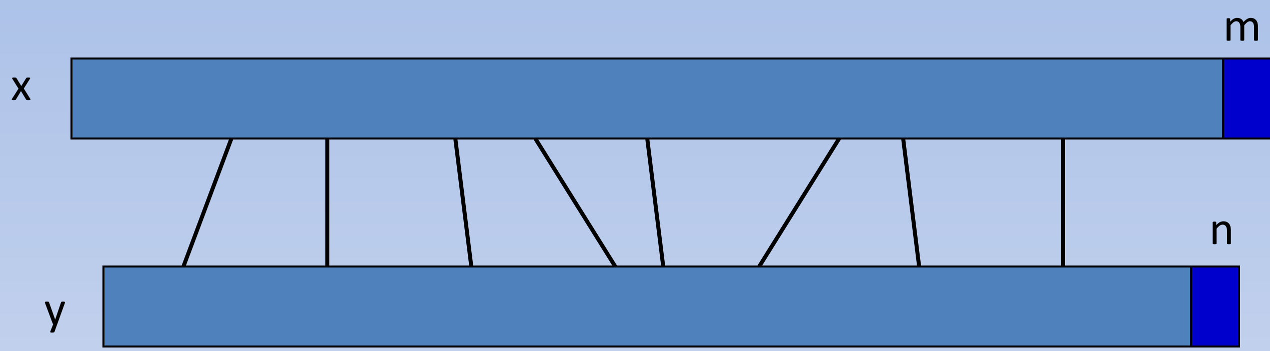
Optimal substructure

- Notice that the LCS problem has *optimal substructure*: parts of the final solution are solutions of subproblems.
  - If z = LCS(x, y), then any prefix of z is an LCS of a prefix of x and a prefix of y.



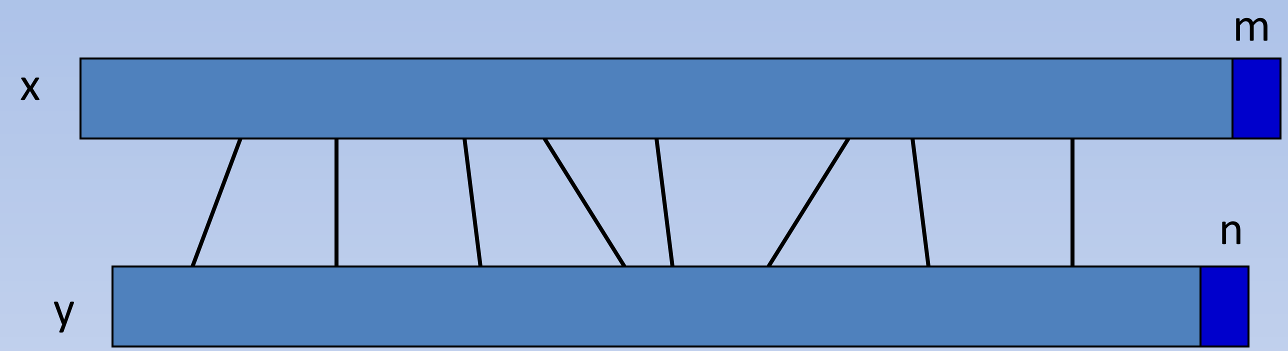
- Subproblems: "find LCS of pairs of *prefixes* of x and y"

Recursive thinking



- Case 1: $x[m]=y[n]$. There is **an** optimal LCS that matches $x[m]$ with $y[n]$. → Find out LCS $(x[1..m\text{-}1], y[1..n\text{-}1])$
- Case 2: $x[m]\neq y[n]$. At most one of them is in LCS
  - Case 2.1: $x[m]$ *not in LCS* → Find out LCS $(x[1..m\text{-}1], y[1..n])$
  - Case 2.2: $y[n]$ *not in LCS* → Find out LCS $(x[1..m], y[1..n\text{-}1])$

Recursive thinking



- Case 1: $x[m]=y[n]$
  - $LCS(x, y) = LCS(x[1..m\text{-}1], y[1..n\text{-}1]) \,||\, x[m]$

Reduce both sequences by 1 char

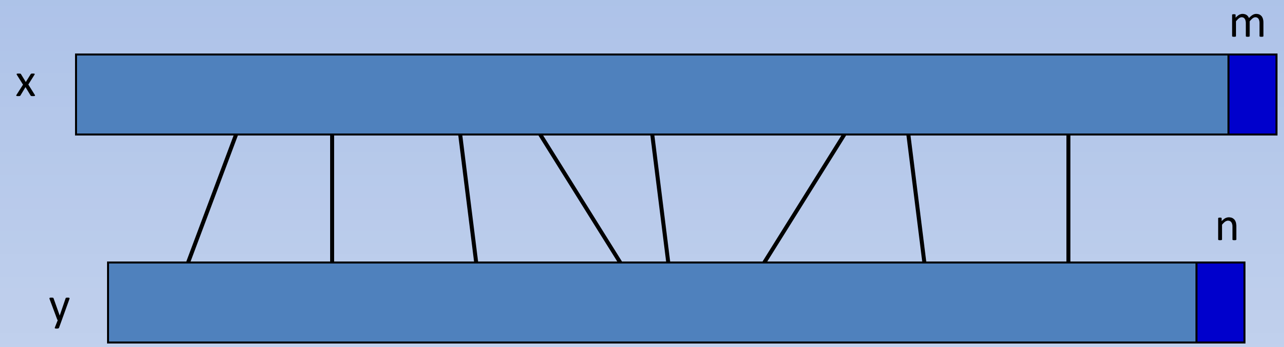
concatenate

- Case 2: $x[m] \neq y[n]$
  - $LCS(x, y) = LCS(x[1..m\text{-}1], y[1..n])$ or

    $LCS(x[1..m], y[1..n\text{-}1])$, whichever is longer

Reduce either sequence by 1 char

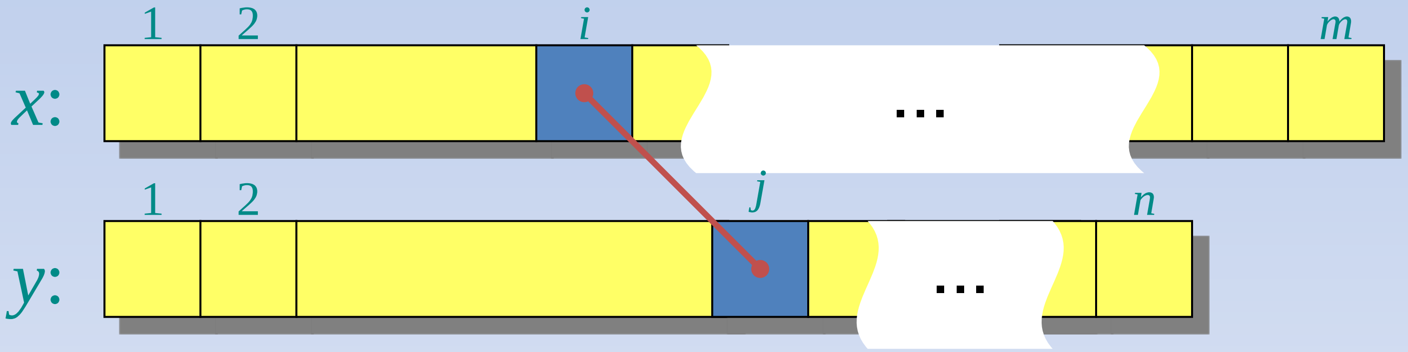
Finding length of LCS



- Let $c[i, j]$ be the length of LCS($x[1..i]$, $y[1..j]$)
  => $c[m, n]$ is the length of LCS(x, y)
- If $x[m] = y[n]$
  $c[m, n] = c[m\text{-}1, n\text{-}1] + 1$
- If $x[m] \mathrel{!}= y[n]$
  $c[m, n] = max \{ c[m\text{-}1, n], c[m, n\text{-}1] \}$

Generalize: recursive formulation

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

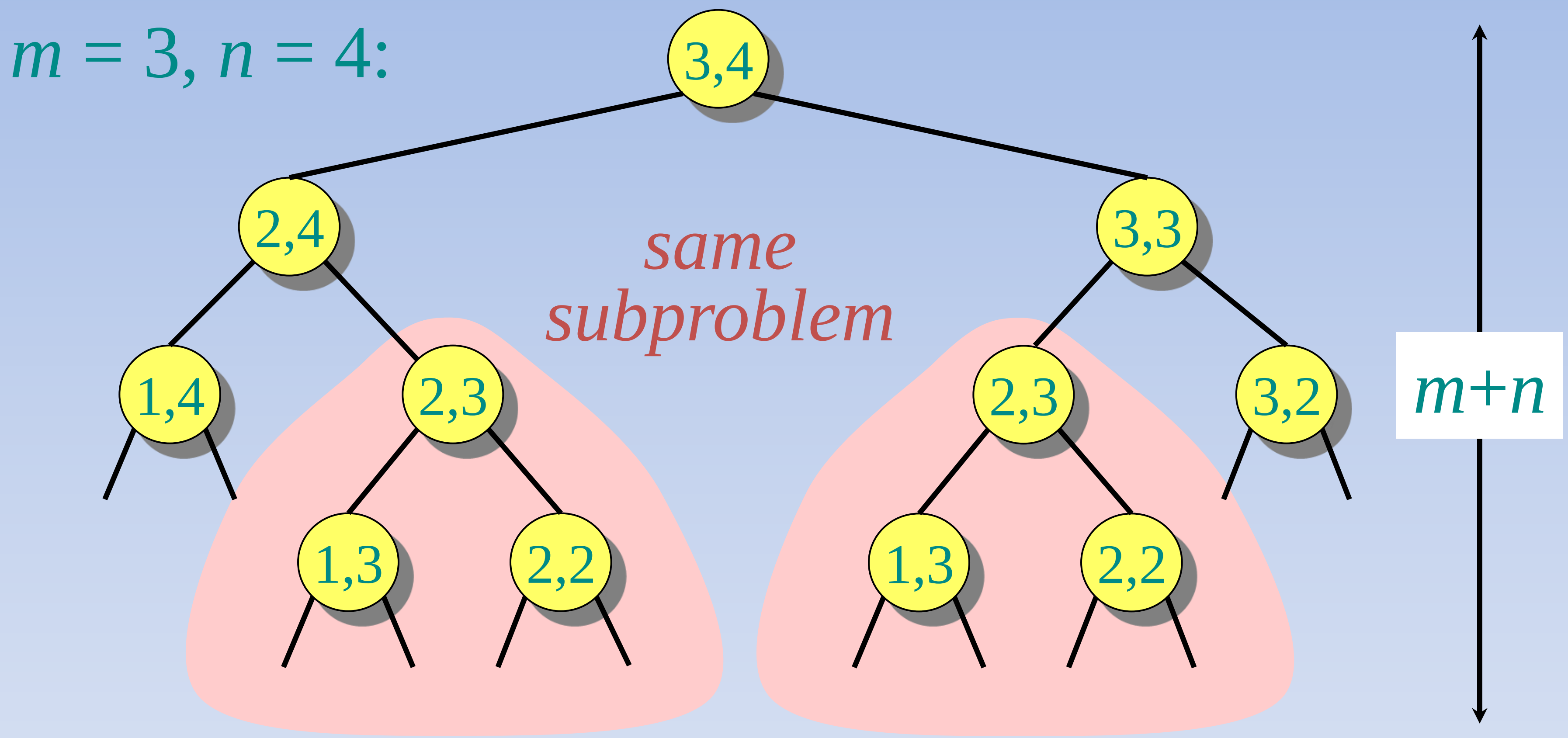


Recursive algorithm for LCS

LCS(x, y, i, j)
  if $x[i] = y[j]$
    then $c[i, j] \leftarrow$ LCS$(x, y, i-1, j-1) + 1$
    else $c[i, j] \leftarrow \max\{$ LCS$(x, y, i-1, j)$,
          LCS$(x, y, i, j-1)\}$

Worst-case: $x[i] \neq y[j]$, in which case the algorithm evaluates two subproblems, each with only one parameter decremented.

Recursion tree

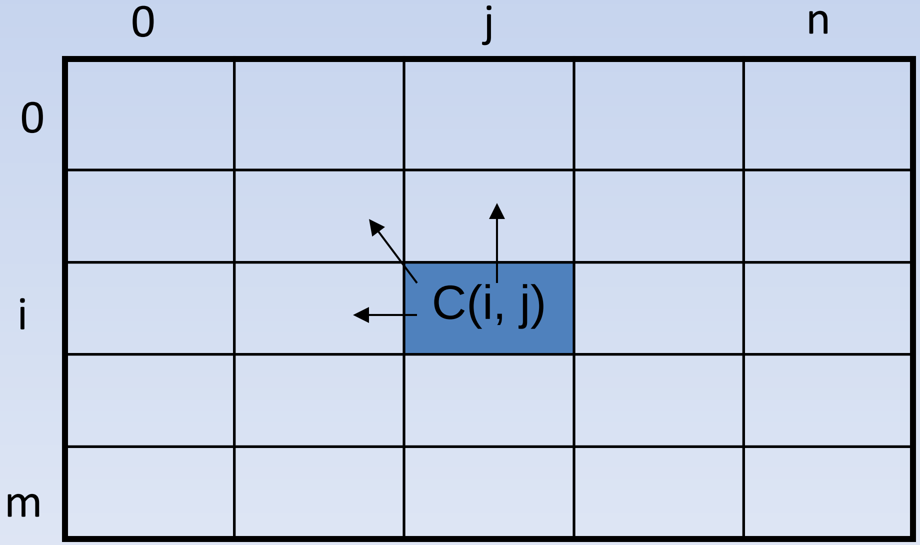


Height $= m + n \Rightarrow$ work potentially exponential, but we're solving subproblems already solved!

DP Algorithm

- Key: find out the correct order to solve the sub-problems
- Total number of sub-problems: *m * n*

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



DP Algorithm

```
LCS-Length(X, Y)
1. m = length(X)  // get the # of symbols in X
2. n  = length(Y) // get the # of symbols in Y
3. for i = 1 to m  c[i,0] = 0   // special case: Y[0]
4. for j = 1 to n   c[0,j] = 0   // special case: X[0]
5. for i = 1 to m                       // for all X[i]
6.      for j = 1 to n                      // for all Y[j]
7.           if ( X[i] == Y[j])
8.                c[i,j] = c[i-1,j-1] + 1
9.           else c[i,j] = max( c[i-1,j], c[i,j-1] )
10. return c
```

LCS Example

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

- X = ABCB
- Y = BDCAB

What is the LCS 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		Y[j]	**B**	**D**	**C**	**A**	**B**
0	X[i]						
1	**A**						
2	**B**						
3	**C**						
4	**B**						

X = ABCB; m = |X| = 4
Y = BDCAB; n = |Y| = 5
Allocate array c[5,6]

LCS Example (1)

ABCB
BDCAB

i		j	0	1	2	3	4	5
			Y[j]	**B**	**D**	**C**	**A**	**B**
0	X[i]		**0**	**0**	**0**	**0**	**0**	**0**
1	**A**		**0**					
2	**B**		**0**					
3	**C**		**0**					
4	**B**		**0**					

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

LCS Example (2)

ABCB
BDCAB

		j	0	1	2	3	4	5
i			Y[j]	**B**	**D**	**C**	**A**	**B**
0	X[i]		**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		Y[j]	**B**	**D**	**C**	**A**	**B**
0	X[i]	**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

i		j	0	1	2	3	4	5
			Y[j]	**B**	**D**	**C**	**A**	**B**
0	X[i]		**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		Y[j]	**B**	**D**	**C**	**A**	**B**
0	X[i]	**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
			Y[j]	**B**	**D**	**C**	**A**	**B**
0	X[i]		**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		Y[j]	**B**	**D**	**C**	**A**	**B**
0	X[i]	**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		Y[j]	**B**	**D**	**C**	**A**	**B**
0	X[i]	**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 (9)

ABCB
BDCAB

		j	0	1	2	3	4	5
i			Y[j]	**B**	**D**	**C**	**A**	**B**
0	X[i]		**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 (10)

ABCB
BDCAB

		j	0	1	2	3	4	5
i			Y[j]	**B**	**D**	**C**	**A**	**B**
0	X[i]		**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 (11)

ABCB
BDCAB

		j	0	1	2	3	4	5
i			Y[j]	**B**	**D**	**C**	**A**	**B**
0	X[i]		**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 (12)

ABCB
BDCAB

		j	0	1	2	3	4	5
i			Y[j]	**B**	**D**	**C**	**A**	**B**
0	X[i]		**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 (13)

ABCB
BDCAB

		j	0	1	2	3	4	5
i			Y[j]	**B**	**D**	**C**	**A**	**B**
0	X[i]		**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 (14)

ABCB
BDCAB

i		j	0	1	2	3	4	5
			Y[j]	**B**	**D**	**C**	**A**	**B**
0	X[i]		**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])

LCS Algorithm Running Time

- LCS algorithm calculates the values of each entry of the array c[m,n]
- So what is the running time?

O(m*n)

since each c[i,j] is calculated in constant time, and there are m*n elements in the array

How to find actual LCS

- The algorithm just found the *length* of LCS, but not LCS itself.
- How to find the actual LCS?
- For each c[i,j] we know how it was acquired:

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

- A match happens only when the first equation is taken
- So we can start from *c[m,n]* and go backwards, remember *x[i]* whenever *c[i,j] = c[i-1, j-1]+1*.

2	2
2	3

For example, here
c[i,j] = c[i-1,j-1] +1 = 2+1=3

Finding LCS

	j	0	1	2	3	4	5
i		Y[j]	**B**	**D**	**C**	**A**	**B**
0	X[i]	**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**

Time for trace back: $O(m+n)$.

Finding LCS (2)

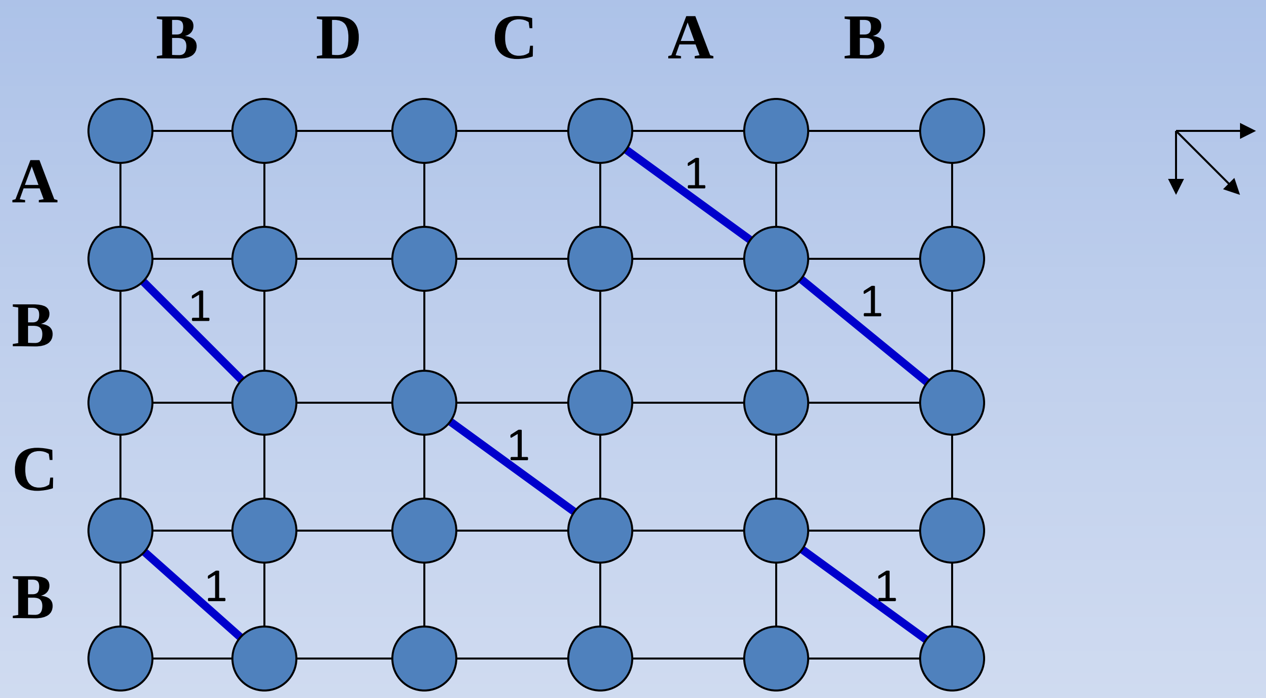
i	X[i] \ Y[j]	j = 0	1: **B**	2: **D**	3: **C**	4: **A**	5: **B**
0		**0**	**0**	**0**	**0**	**0**	**0**
1	**A**	**0**	**0**	**0**	**0**	**1**	**1**
2	**B**	**0**	**1** ←	**1**	**1**	**1**	**2**
3	**C**	**0**	**1**	**1**	**2** ←	**2**	**2**
4	**B**	**0**	**1**	**1**	**2**	**2**	**3**

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

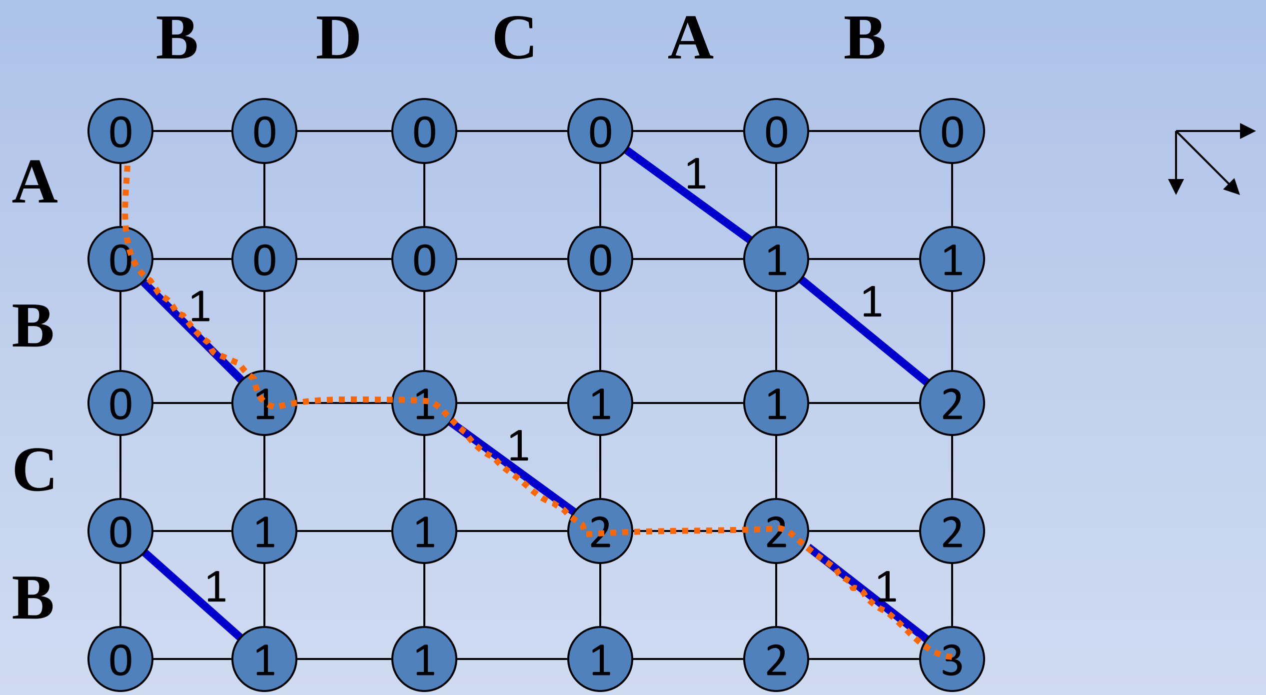
LCS (straight order): **B C B**

(this string turned out to be a palindrome)

LCS as a longest path problem



LCS as a longest path problem



A more general problem

- Aligning two strings, such that

  Match = 1

  Mismatch = 0

  Insertion/deletion = -1

  (or other scores)

- Aligning ABBC with CABC
  - LCS = 3: ABC
  - Best alignment

```
ABBC
CABC
```

Score = 2

```
–ABBC
CAB–C
```

Score = 1

- Let $F(i, j)$ be the best alignment score between $X[1..i]$ and $Y[1..j]$.
- $F(m, n)$ is the best alignment score between X and Y
- Recurrence

$$F(i,j) = \max \begin{cases} F(i\text{-}1, j\text{-}1) + \delta(i, j) & \text{Match/Mismatch} \\ F(i\text{-}1,j) - 1 & \text{Insertion on Y} \\ F(i, j\text{-}1) - 1 & \text{Insertion on X} \end{cases}$$

$\delta(i, j) = 1$ *if* $X[i]=Y[j]$ *and* 0 *otherwise.*

Alignment Example

ABBC
CABC

i	j	0	1	2	3	4
		Y[j]	**C**	**A**	**B**	**C**
0	X[i]					
1	**A**					
2	**B**					
3	**B**					
4	**C**					

X = ABBC; m = |X| = 4
Y = CABC; n = |Y| = 4
Allocate array F[5,5]

Alignment Example

ABBC
CABC

i	j	0	1	2	3	4
		Y[j]	**C**	**A**	**B**	**C**
	X[i]					
0		0	-1	-2	-3	-4
1	**A**	-1	0	0	-1	-2
2	**B**	-2	-1	0	1	0
3	**B**	-3	-2	-1	1	1
4	**C**	-4	-2	-2	0	2

$$F(i, j) = \max \begin{cases} F(i\text{-}1, j\text{-}1) + \delta(i, j) & \text{Match/Mismatch} \\ F(i\text{-}1, j) - 1 & \text{Insertion on Y} \\ F(i, j\text{-}1) - 1 & \text{Insertion on X} \end{cases}$$

$\delta(i, j) = 1$ *if* $X[i]=Y[j]$ *and* 0 *otherwise.*

Alignment Example

ABBC
CABC

i		j: 0	1	2	3	4
	X[i] \ Y[j]		C	A	B	C
0		0	-1	-2	-3	-4
1	A	-1	0	0	-1	-2
2	B	-2	-1	0	1	0
3	B	-3	-2	-1	1	1
4	C	-4	-2	-2	0	2

ABBC
• • | |
CABC

Score = 2

$$F(i, j) = \max \begin{cases} F(i\text{-}1, j\text{-}1) + \delta(i, j) & \text{Match/Mismatch} \\ F(i\text{-}1, j) - 1 & \text{Insertion on Y} \\ F(i, j\text{-}1) - 1 & \text{Insertion on X} \end{cases}$$

$\delta(i, j) = 1$ *if* $X[i]=Y[j]$ *and 0 otherwise.*