

# Dynamic Programming

JinJu Lee & Beatrice Seifert

CSE 5311 Fall 2005

Week 10 (Nov 1 & 3)

# Contents

- Edmond-Karp-Algorithm
- Bipartite Graph Matching Problem
- Dynamic Programming Paradigm
  - Fibonacci Sequence
  - Rod Cutting
  - Longest Common Subsequence Problem
  - String Matching Algorithms
    - Substring Matching
    - Knuth-Morris-Pratt Algorithm

# Edmond-Karp algorithm

- Improved version of Ford-Fulkerson algorithm
- Uses BFS to find the augmenting path
- The augmenting path is a shortest path from  $s$  to  $t$  in the residual network

# Edmond-Karp algorithm

Do FORD-FULKERSON, but compute augmenting paths by BFS of  $G_f$ . Augmenting paths are shortest paths  $s \rightsquigarrow t$  in  $G_f$ , with all edge weights = 1.

Edmonds-Karp runs in  $O(VE^2)$  time.

We prove this through the following theorem

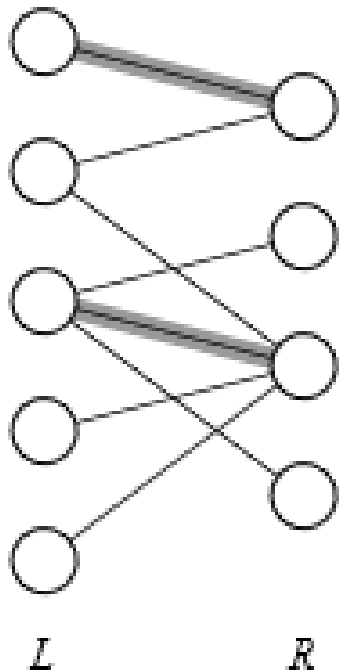
## *Theorem*

Edmonds-Karp performs  $O(VE)$  augmentations.

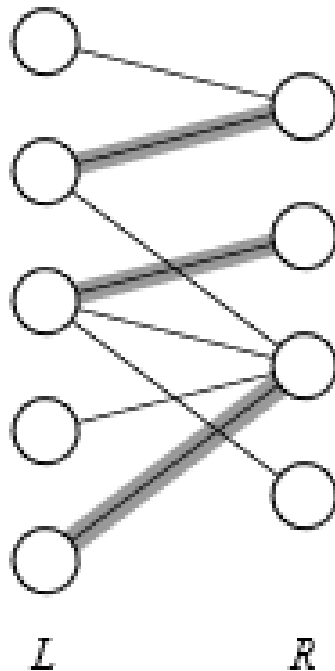
# Max Bipartite Matching Problem

Example of a problem that can be solved by turning it into a flow problem.

$G = (V, E)$  (undirected) is *bipartite* if we can partition  $V = L \cup R$  such that all edges in  $E$  go between  $L$  and  $R$ .



matching



maximum matching

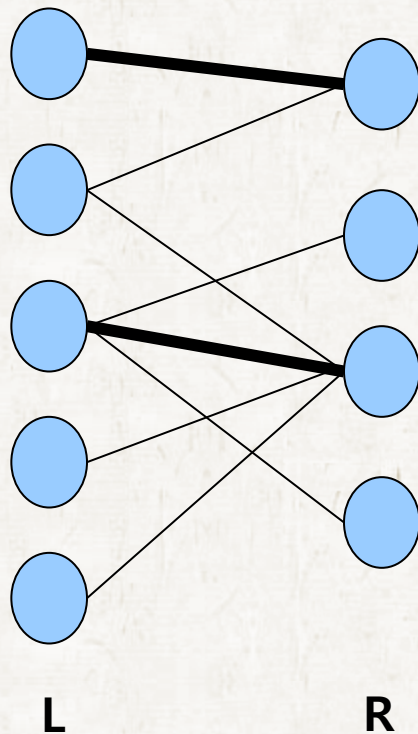


# Max Bipartite Matching Problem

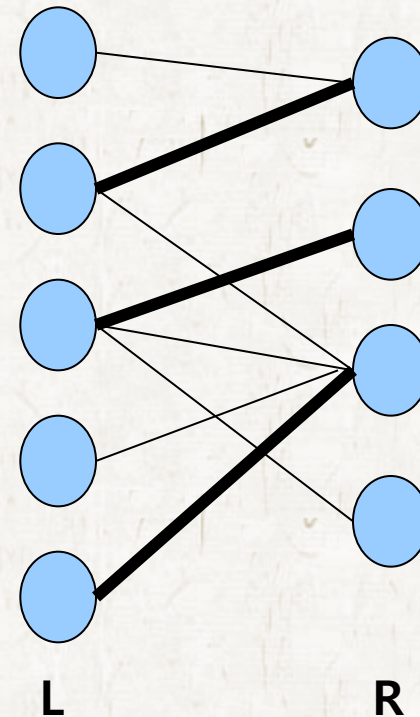
- **Matching:** Given an undirected graph  $G = (V, E)$ , a subset of edges  $M \subseteq E$  such that for all vertices  $v \in V$ , at most one edge of  $M$  is incident on  $v$ .
- Vertex  $v$  is matched if an edge of  $M$  is incident on it; otherwise unmatched
- **Maximum Matching:** a matching of maximum cardinality, that is, a matching  $M$  such that for any matching  $M'$ , we have  $|M| \geq |M'|$

# Max Bipartite Matching Problem

Given a bipartite graph (with the partition), find a maximum matching



(a) Matching with Cardinality 2



(b) A maximum matching with cardinality 3.

# Finding a Max Bipartite Matching

- Use Ford-Fulkerson (or Edmund-Karp) method
- Original problem: undirected bipartite graph
- Solution
  - construct a flow network (add source  $s$  and destination  $t$ )
  - put arrows from  $L$  to  $R$
  - assign unit capacity to each edge
- Running Time:  $O(VE)$



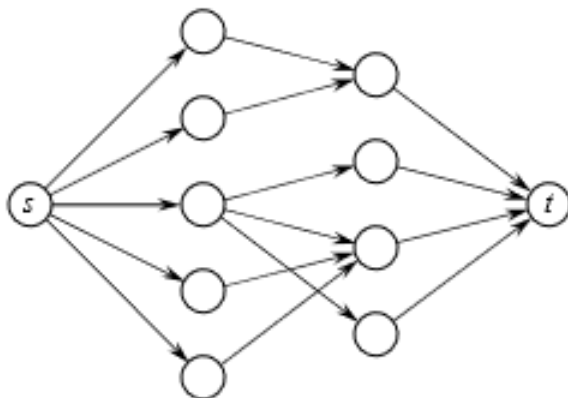
# An Application of Max Bipartite Matching

Matching planes to routes.

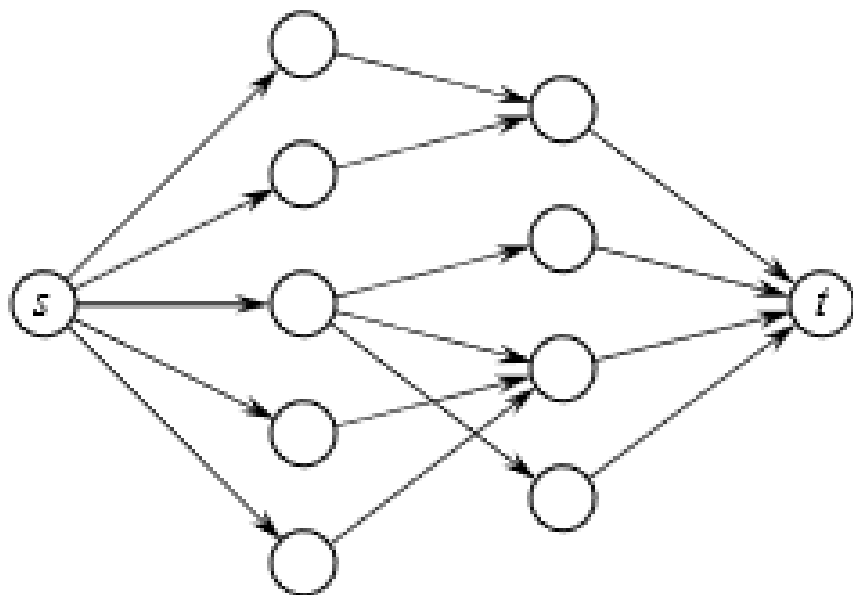
- $L$  = set of planes.
- $R$  = set of routes.
- $(u, v) \in E$  if plane  $u$  can fly route  $v$ .
- Want maximum # of routes to be served by planes.

Given  $G$ , define flow network  $G' = (V', E')$ .

- $V' = V \cup \{s, t\}$ .
- $E' = \{(s, u) : u \in L\} \cup \{(u, v) : (u, v) \in E\} \cup \{(v, t) : v \in R\}$ .
- $c(u, v) = 1$  for all  $(u, v) \in E'$ .



# An Application of Max Bipartite Matching



Each vertex in  $V$  has  $\geq 1$  incident edge  $\Rightarrow |E| \geq |V|/2$ .

Therefore,  $|E| \leq |E'| = |E| + |V| \leq 3|E|$ .

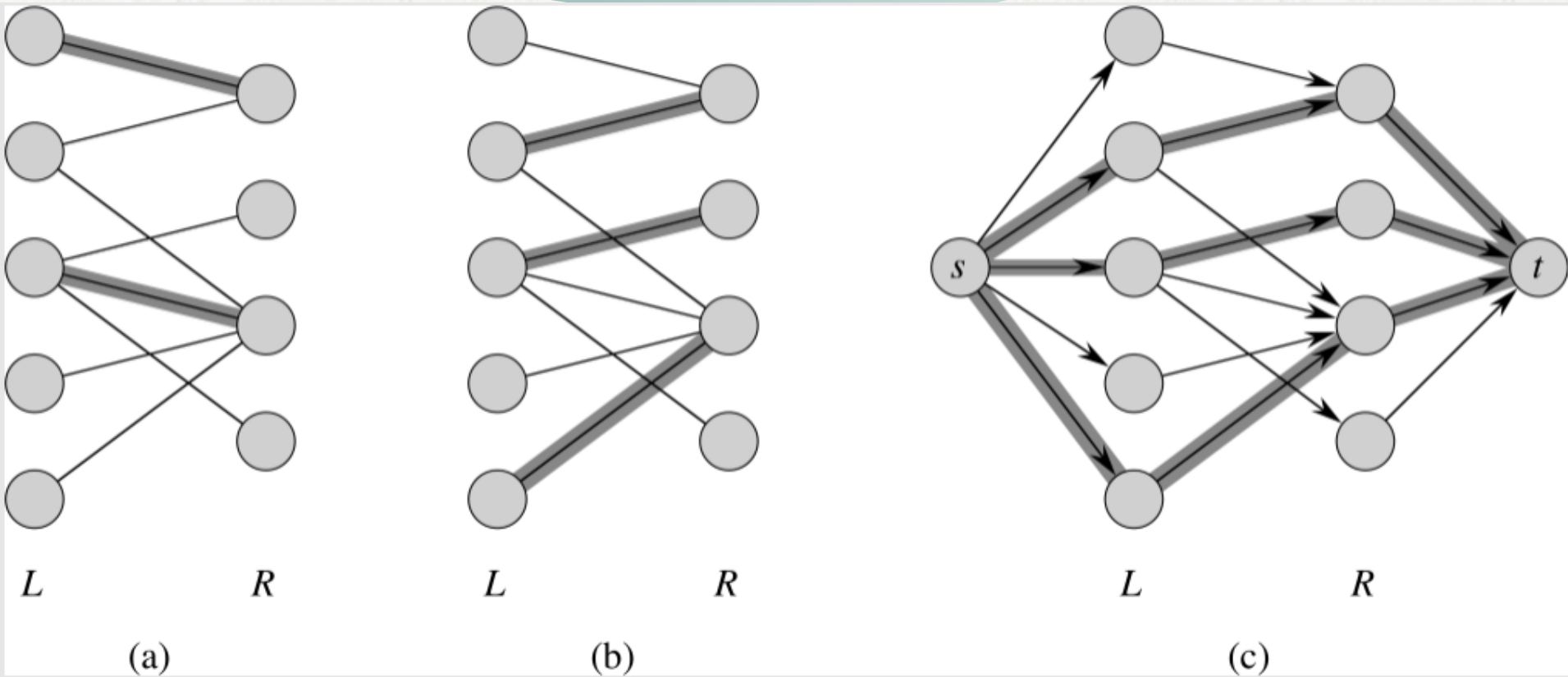
Therefore,  $|E'| = \Theta(E)$ .

Find a max flow in  $G'$ . Book shows that it will have integer values for all  $(u, v)$ .

Use edges that carry flow of 1 in matching.

This method provides us with maximum matching

# Finding a Max Bipartite Matching



- (a) A matching of cardinality 2
- (b) A maximum matching with cardinality 3
- (c) The corresponding flow network  $G'$  with a maximum flow shown. Each edge has unit capacity. Shaded edges have a flow of 1 and all other edges carry no flow. The shaded edges from  $L$  to  $R$  correspond to those in the maximum matching from (b)

# What is dynamic programming?

Not a specific algorithm but a technique (like divide-and-conquer)

It is a method to reduce the runtime of algorithms by :

- Breaking the problem into smaller sub-problems
- Solving these sub-problems optimally (greedy algorithms)
- Using these optimal solutions to the sub-problems to construct an optimal solution for the original problem

# Example: Fibonacci Sequence

**FIBONACCI**( $n$ )

  let  $fib[0 \dots n]$  be a new array

$fib[0] = fib[1] = 1$

**for**  $i = 2$  **to**  $n$

$fib[i] = fib[i - 1] + fib[i - 2]$

**return**  $fib[n]$

FIBONACCI directly implements the recurrence relation of the Fibonacci sequence. Each number in the sequence is the sum of the two previous numbers in the sequence. The running time is clearly  $O(n)$ .



# Example: Fibonacci Sequence

We build an array, with the first two cells containing a 1 each (Fib(0) and Fib(1)).

To get the next cell (Fib(2)) we sum the content of the first two cells and store the result in the third cell. For the fourth cell we sum the content of the second and the third cell, and so on.

This way every number is only calculated once.

The runtime of this algorithm is linear:  $T(n) = O(n)$ , and so are the space requirements, as they remain constant.

# Example: Rod Cutting

How to cut steel rods into pieces in order to maximize the revenue you can get? Each cut is free. Rod lengths are always an integral number of inches.

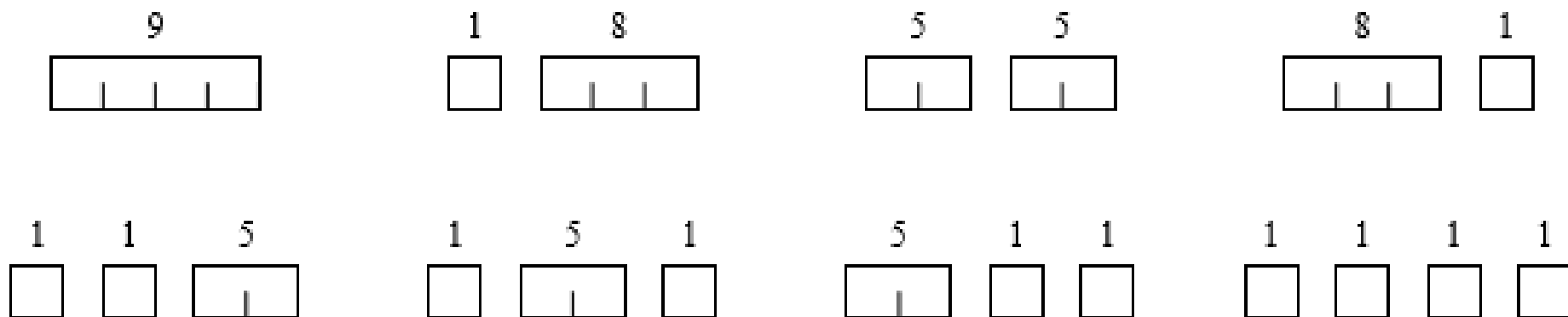
**Input:** A length  $n$  and table of prices  $p_i$ , for  $i = 1, 2, \dots, n$ .

**Output:** The maximum revenue obtainable for rods whose lengths sum to  $n$ , computed as the sum of the prices for the individual rods.

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

Can cut up a rod in  $2^{n-1}$  different ways, because can choose to cut or not cut after each of the first  $n - 1$  inches.

Here are all 8 ways to cut a rod of length 4, with the costs from the example:



The best way is to cut it into two 2-inch pieces, getting a revenue of  $p_2 + p_2 = 5 + 5 = 10$ .

# Example: Rod Cutting

Let  $r_i$  be the maximum revenue for a rod of length  $i$ . Can express a solution as a sum of individual rod lengths.

Can determine optimal revenues  $r_i$  for the example, by inspection:

$i$	$r_i$	optimal solution
1	1	1 (no cuts)
2	5	2 (no cuts)
3	8	3 (no cuts)
4	10	2 + 2
5	13	2 + 3
6	17	6 (no cuts)
7	18	1 + 6 or 2 + 2 + 3
8	22	2 + 6

Can determine optimal revenue  $r_n$  by taking the maximum of

- $p_n$ : the price we get by not making a cut,
- $r_1 + r_{n-1}$ : the maximum revenue from a rod of 1 inch and a rod of  $n - 1$  inches,
- $r_2 + r_{n-2}$ : the maximum revenue from a rod of 2 inches and a rod of  $n - 2$  inches, ...
- $r_{n-1} + r_1$ .

That is,

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

# Rod Cutting: Recursive Top-down

Direct implementation of the simpler equation for  $r_n$ .

The call CUT-ROD( $p, n$ ) returns the optimal revenue  $r_n$ :

CUT-ROD( $p, n$ )

**if**  $n == 0$

**return** 0

$q = -\infty$

**for**  $i = 1$  **to**  $n$

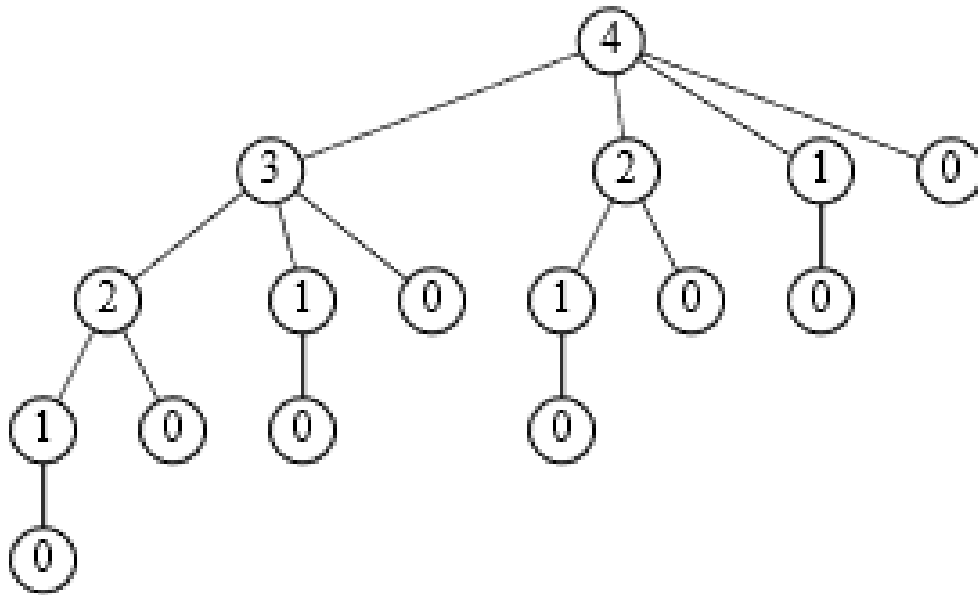
$q = \max(q, p[i] + \text{CUT-ROD}(p, n - i))$

**return**  $q$

This procedure works, but it is terribly *inefficient*. If you code it up and run it, it could take more than an hour for  $n = 40$ . Running time almost doubles each time  $n$  increases by 1.

# Rod Cutting: Recursive Top-down

**Why so inefficient?:** CUT-ROD calls itself repeatedly, even on subproblems it has already solved. Here's a tree of recursive calls for  $n = 4$ . Inside each node is the value of  $n$  for the call represented by the node:



Lots of repeated subproblems. Solve the subproblem for size 2 twice, for size 1 four times, and for size 0 eight times.



# Rod Cutting: Recursive Top-down

*Exponential growth:* Let  $T(n)$  equal the number of calls to CUT-ROD with second parameter equal to  $n$ . Then

$$T(n) = \begin{cases} 1 & \text{if } n = 0, \\ 1 + \sum_{j=0}^{n-1} T(j) & \text{if } n \geq 1. \end{cases}$$

$$\begin{aligned} T(n) &= 1 + \sum_{j=0}^{n-1} 2^j \\ &= 1 + (2^n - 1) \\ &= 2^n. \end{aligned}$$

# Rod Cutting

## Dynamic-programming solution

Instead of solving the same subproblems repeatedly, arrange to solve each subproblem just once.

Save the solution to a subproblem in a table, and refer back to the table whenever we revisit the subproblem.

“Store, don’t recompute”  $\Rightarrow$  time-memory trade-off.

Can turn an exponential-time solution into a polynomial-time solution.

Two basic approaches: top-down with memoization, and bottom-up.

### *Running time*

Both the top-down and bottom-up versions run in  $\Theta(n^2)$  time.

# Rod Cutting: Top-down

Solve recursively, but store each result in a table.

To find the solution to a subproblem, first look in the table. If the answer is there, use it. Otherwise, compute the solution to the subproblem and then store the solution in the table for future use.

**MEMOIZED-CUT-ROD**( $p, n$ )

  let  $r[0 \dots n]$  be a new array

**for**  $i = 0$  **to**  $n$

$r[i] = -\infty$

**return** **MEMOIZED-CUT-ROD-AUX**( $p, n, r$ )

**MEMOIZED-CUT-ROD-AUX**( $p, n, r$ )

**if**  $r[n] \geq 0$

**return**  $r[n]$

**if**  $n == 0$

$q = 0$

**else**  $q = -\infty$

**for**  $i = 1$  **to**  $n$

$q = \max(q, p[i] + \text{MEMOIZED-CUT-ROD-AUX}(p, n - i, r))$

$r[n] = q$

**return**  $q$

# Rod Cutting: Bottom Up

Sort the subproblems by size and solve the smaller ones first. That way, when solving a subproblem, have already solved the smaller subproblems we need.

**BOTTOM-UP-CUT-ROD**( $p, n$ )

  let  $r[0 \dots n]$  be a new array

$r[0] = 0$

**for**  $j = 1$  **to**  $n$

$q = -\infty$

**for**  $i = 1$  **to**  $j$

$q = \max(q, p[i] + r[j - i])$

$r[j] = q$

**return**  $r[n]$

# Longest Common Subsequence Problem

What is the difference between a substring and a subsequence?

A substring is a contiguous “mini string” within a string.

A subsequence is a number of characters in the same order as within the original string, but it may be non-contiguous.

String                      A: a b b a c d e    (1-2-3-4-5-6-7)

Substring                B: a b b a                (1-2-3-4)

Subsequence            C: a a c e                (1-4-5-7)



# LCS

Finding the longest common subsequence (LCS) is important for applications like spell checking or finding alignments in DNA strings in the field of bioinformatics (e.g. BLAST-basic local alignment search tool).

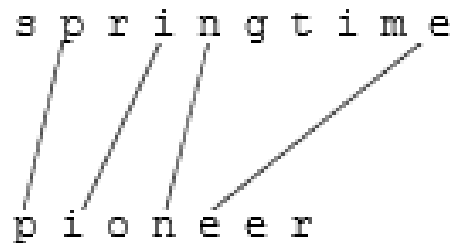
How does it work?

There are several methods, e.g. the “brute force policy” or the LCSS which is a dynamic programming method.

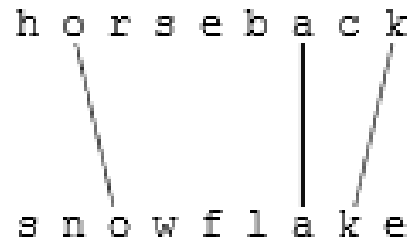
# LCS

**Problem:** Given 2 sequences,  $X = \langle x_1, \dots, x_m \rangle$  and  $Y = \langle y_1, \dots, y_n \rangle$ . Find a subsequence common to both whose length is longest. A subsequence doesn't have to be consecutive, but it has to be in order.

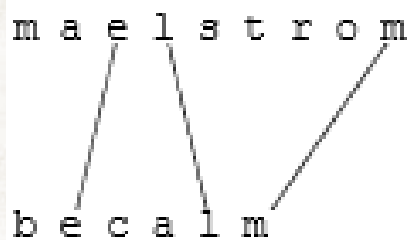
s p r i n g t i m e  
p i o n e e r



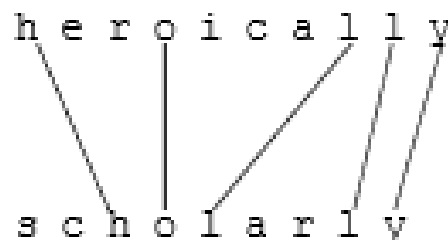
h o r s e b a c k  
s n o w f l a k e



m a e l s t r o m  
b e c a l m



h e r o i c a l l y  
s c h o l a r l y



Brute-force algorithm:

For every subsequence of  $X$ , check whether it's a subsequence of  $Y$ .

Time:  $\Theta(n2^m)$ .

- $2^m$  subsequences of  $X$  to check.
- Each subsequence takes  $\Theta(n)$  time to check: scan  $Y$  for first letter, from there scan for second, and so on.

# LCS - Solution

- ❑ Efficient solution of the LCS problem using dynamic programming
- ❑ The algorithm uses four steps:
  1. Analyze LCS properties (Optimal Substructure)
  2. Devise a recursive solution
  3. Compute the length of LCS
  4. Constructing LCS

But before going any further we specify a new notation

$$\begin{aligned} X_i &= \text{prefix } \langle x_1, \dots, x_i \rangle \\ Y_i &= \text{prefix } \langle y_1, \dots, y_i \rangle \end{aligned}$$

# 1. Optimal Substructure

## *Theorem*

Let  $Z = \langle z_1, \dots, z_k \rangle$  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  $z_k \neq x_m \Rightarrow Z$  is an LCS of  $X_{m-1}$  and  $Y$ .
3. If  $x_m \neq y_n$ , then  $z_k \neq y_n \Rightarrow Z$  is an LCS of  $X$  and  $Y_{n-1}$ .

## *Proof*

1. First show that  $z_k = x_m = y_n$ . Suppose not. Then make a subsequence  $Z' = \langle z_1, \dots, z_k, x_m \rangle$ . It's a common subsequence of  $X$  and  $Y$  and has length  $k + 1 \Rightarrow Z'$  is a longer common subsequence than  $Z \Rightarrow$  contradicts  $Z$  being an LCS.

Now show  $Z_{k-1}$  is an LCS of  $X_{m-1}$  and  $Y_{n-1}$ . Clearly, it's a common subsequence. Now suppose there exists a common subsequence  $W$  of  $X_{m-1}$  and  $Y_{n-1}$  that's longer than  $Z_{k-1} \Rightarrow$  length of  $W \geq k$ . Make subsequence  $W'$  by appending  $x_m$  to  $W$ .  $W'$  is common subsequence of  $X$  and  $Y$ , has length  $\geq k + 1 \Rightarrow$  contradicts  $Z$  being an LCS.

2. If  $z_k \neq x_m$ , then  $Z$  is a common subsequence of  $X_{m-1}$  and  $Y$ . Suppose there exists a subsequence  $W$  of  $X_{m-1}$  and  $Y$  with length  $> k$ . Then  $W$  is a common subsequence of  $X$  and  $Y \Rightarrow$  contradicts  $Z$  being an LCS.
3. Symmetric to 2.

■ (theorem)

# 1. Optimal Substructure

- ❑ LCS of two sequences contains within it an LCS of prefixes of the two sequences
- ❑ Thus, the LCS problem has an optimal-substructure property
- ❑ Theorem above shows that for  $X = \langle x_1, x_2, \dots, x_m \rangle$  and  $Y = \langle y_1, y_2, \dots, y_n \rangle$ 
  - If  $x_m = y_n$  we can find a LCS of  $X_{m-1}$  and  $Y_{n-1}$ . Appending  $x_m = y_n$  to this LCS yields the LCS of  $X$  and  $Y$
  - If  $x_m \neq y_n$ , then we must solve two subproblems:
    - Find LCS for  $X$  and  $Y_{n-1}$
    - Find LCS for  $X_{m-1}$  and  $Y$Whichever LCS is longer is the LCS of  $X$  and  $Y$
- This provides the basis for a dynamic programming based recursive solution



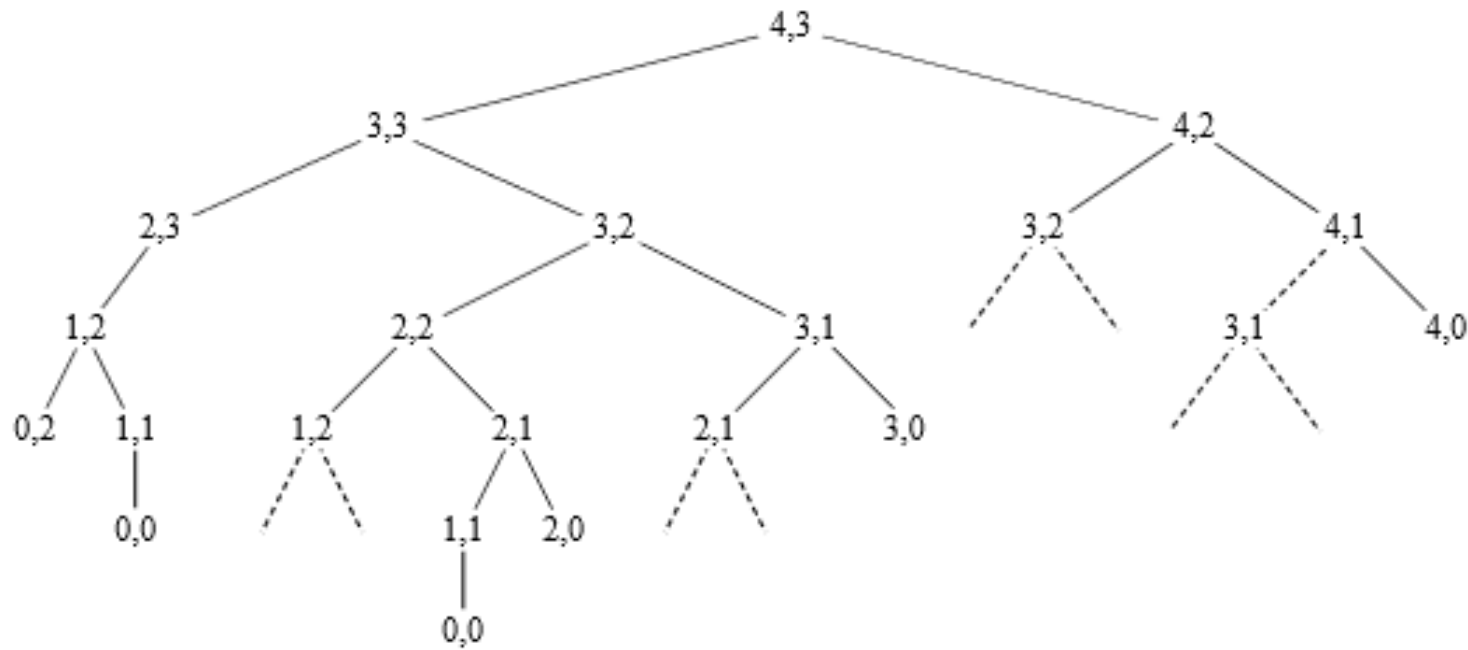
## 2. Recursive Solution

Define  $c[i, j]$  = length of LCS of  $X_i$  and  $Y_j$ . We want  $c[m, n]$ .

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

Again, we could write a recursive algorithm based on this formulation.

Try with  $X = \langle a, t, o, m \rangle$  and  $Y = \langle a, n, t \rangle$ . Numbers in nodes are values of  $i, j$  in each recursive call. Dashed lines indicate subproblems already computed.



- Lots of repeated subproblems.
- Instead of recomputing, store in a table.

### 3. Computing LCS Length

LCS-LENGTH( $X, Y, m, n$ )

let  $b[1..m, 1..n]$  and  $c[0..m, 0..n]$  be new tables

for  $i = 1$  to  $m$

$c[i, 0] = 0$

for  $j = 0$  to  $n$

$c[0, j] = 0$

for  $i = 1$  to  $m$

    for  $j = 1$  to  $n$

        if  $x_i == y_j$

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

$b[i, j] = \nwarrow$

        else if  $c[i - 1, j] \geq c[i, j - 1]$

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

$b[i, j] = \uparrow$

        else  $c[i, j] = c[i, j - 1]$

$b[i, j] = \leftarrow$

return  $c$  and  $b$

# 4. Constructing LCS

PRINT-LCS( $b, X, i, j$ )

**if**  $i == 0$  or  $j == 0$

**return**

**if**  $b[i, j] == \nwarrow$

    PRINT-LCS( $b, X, i - 1, j - 1$ )

    print  $x_i$

**elseif**  $b[i, j] == \uparrow$

    PRINT-LCS( $b, X, i - 1, j$ )

**else** PRINT-LCS( $b, X, i, j - 1$ )

- 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  $b[i, j] = \nwarrow$ , we have extended LCS by one character. So longest common subsequence = entries with  $\nwarrow$  in them.

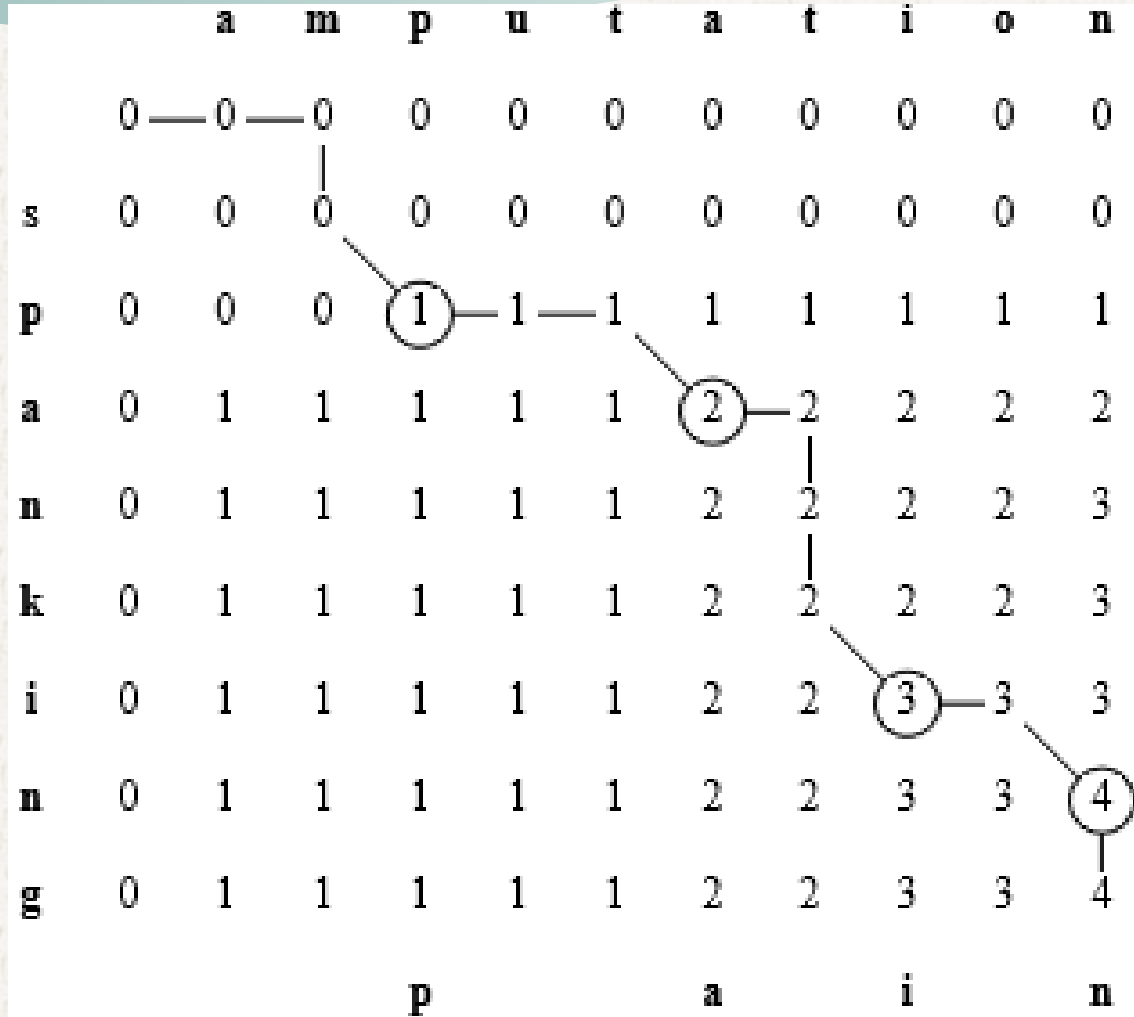
# Demonstration

- The  $c$  and  $b$  tables computed by LCS-LENGTH on the sequences  $X = \langle A, B, C, B, D, A, B \rangle$  and  $Y = \langle B, D, C, A, B, A \rangle$  are displayed
- The number represents length  $c[i, j]$  and arrows represent  $b[i, j]$ , that point to the subproblem used
- Length of the LCS is in the lower right cell,  $c[7, 6]$
- LCS is  $\langle B, C, BA \rangle$  for  $X$  and  $Y$
- To construct LCS, follow  $b[i, j]$  arrows from lower right corner. Each " $\nwarrow$ " corresponds to an LCS entry

		$j$	0	1	2	3	4	5	6
$i$	$x_i$	$y_j$		B	D	C	A	B	A
0			0	0	0	0	0	0	0
1	A		0	$\uparrow$	$\uparrow$	$\uparrow$	$\nwarrow$ 1	$\leftarrow$ 1	$\nwarrow$ 1
2	B		0	$\nwarrow$ 1	$\nwarrow$ 1	$\nwarrow$ 1	$\uparrow$ 1	$\nwarrow$ 2	$\leftarrow$ 2
3	C		0	$\uparrow$ 1	$\uparrow$ 1	$\nwarrow$ 2	$\nwarrow$ 2	$\uparrow$ 2	$\uparrow$ 2
4	B		0	$\nwarrow$ 1	$\uparrow$ 1	$\uparrow$ 2	$\uparrow$ 2	$\nwarrow$ 3	$\leftarrow$ 3
5	D		0	$\uparrow$ 1	$\nwarrow$ 2	$\uparrow$ 2	$\uparrow$ 2	$\nwarrow$ 3	$\uparrow$ 3
6	A		0	$\uparrow$ 1	$\uparrow$ 2	$\uparrow$ 2	$\nwarrow$ 3	$\uparrow$ 3	$\nwarrow$ 4
7	B		0	$\nwarrow$ 1	$\uparrow$ 2	$\uparrow$ 2	$\uparrow$ 3	$\nwarrow$ 4	$\uparrow$ 4

# Demonstration II

- What do spanking and amputation have in common?
- Answer: pain (LCS)





# LCS - Analysis

- For LCS-LENGTH the running time is  $\Theta(mn)$ , since each table entry takes  $\Theta(1)$  time
- The PRINT-LCS procedure takes time  $O(m + n)$ , since it decrements at least one of  $i$  or  $j$  in each recursive call
- To improve we can eliminate  $b$  table altogether.  $c[i, j]$  can be constructed from  $c[i - 1, j - 1]$ ,  $c[i, j - 1]$ , or  $c[i - 1, j]$ , without creating  $b[i, j]$
- This new algorithm can construct LCS in  $O(m + n)$ , like PRINT-LCS
- Exercise 15.4-2 asks for creating the new algorithm/pseudocode



# String Matching

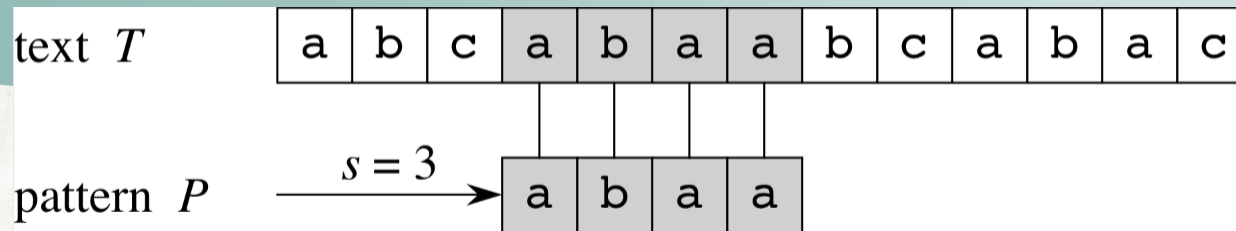
- Who has never used Ctrl-F?
- Text-editing programs frequently need to find all occurrences of a pattern in the text
- Typically, the text is a document being edited, and the pattern searched for a particular word from the user
- Searching for patterns in DNA is another application for string matching
- Efficient algorithms can speed up our searches

# String Matching: Formal Definition

**Input:** The text is an array  $T[1..n]$ , and the pattern is an array  $P[1..m]$ , where  $m \leq n$ . The elements of  $P$  and  $T$  are drawn from a finite *alphabet*  $\Sigma$ . Examples:  $\Sigma = \{0, 1\}$ ,  $\Sigma$  is the ASCII characters, or  $\Sigma = \{A, C, G, T\}$  for DNA matching. We'll call the elements of  $P$  and  $T$  *characters*.

**Output:** All amounts that we have to shift  $P$  to match it with characters of  $T$ . Say that  $P$  *occurs with shift  $s$  in  $T$*  if  $0 \leq s \leq n - m$  and  $T[s + 1..s + m] = P[1..m]$ . (Need  $s \leq n - m$  so that the pattern doesn't run off the end of the text.) If  $P$  occurs with shift  $s$  in  $T$ , then  $s$  is a *valid shift*, otherwise,  $s$  is an *invalid shift*. We want to find all valid shifts.

# String Matching: Examples



- We want to find all occurrences of the pattern  $P = abaa$  in the text  $T = abcabaabcabac$
- The pattern only occurs once at  $s = 3$ , which is called a valid shift

$T = \text{GTAACAGTAAACG}, P = \text{AAC}.$

- $P$  occurs in  $T$  with shifts  $s = 2$  &  $9$

How long does it take to check whether  $P$  occurs in  $T$  with a given shift amount? Need to check each character of  $P$  against the corresponding position in  $T$ , taking  $\Theta(m)$  time in the worst case. (Assumes that checking each character takes constant time.) Can stop once we find a mismatch, so matching time is  $O(m)$  in any case.

Some string-matching algorithms require preprocessing, which we'll account for separately from the matching time.

# String Matching Algorithms

Algorithm	Preprocessing time	Matching time
Naive	0	$O((n - m + 1)m)$
Rabin-Karp	$\Theta(m)$	$O((n - m + 1)m)$
Finite automaton	$O(m  \Sigma )$	$\Theta(n)$
Knuth-Morris-Pratt	$\Theta(m)$	$\Theta(n)$

Some string-matching algorithms and their processing times and matching times

# Naive String-Matching Algorithm

Just try each shift.

NAIVE-STRING-MATCHER( $T, P, n, m$ )

```
for  $s = 0$  to  $n - m$ 
    if  $P[1..m] == T[s + 1..s + m]$ 
        print "Pattern occurs with shift"  $s$ 
```

**Time:** Tries  $n - m + 1$  shift amounts, each taking  $O(m)$  time, so  $O((n - m + 1)m)$ . This bound is tight in the worst case, such as when the text is  $n$  As and the pattern is  $m$  As. No preprocessing needed.

This algorithm is not efficient. It throws away valuable information. Example: For  $T$  and  $P$  above, when we look at shift amount  $s = 2$ , we see all the characters in  $T[3..5] = AAC$ . But at the next shift, for  $s = 3$ , we look at  $T[4]$  and  $T[5]$  again. Since we've already seen these characters, we'd like to avoid having to look at them again.



# Knuth-Morris-Pratt-Algorithm

- The Knuth-Morris-Pratt Algorithm (KMP) is an optimization of the substring matching algorithm. It lets us bypass previously matched characters. It runs in  $T(n) = O(m + n) \approx O(n)$  time, i.e., linear
- We have a text  $T$  of length  $n$ , a pattern  $P$  of length  $m$  and a prefix array/function  $\pi$  of the same length as the pattern, i.e.  $m$
- The prefix array tells us how far we have to move pattern  $P$  along text  $T$  to the right at each step
- We have two pointers, one moves along  $T$  and the other along  $P$



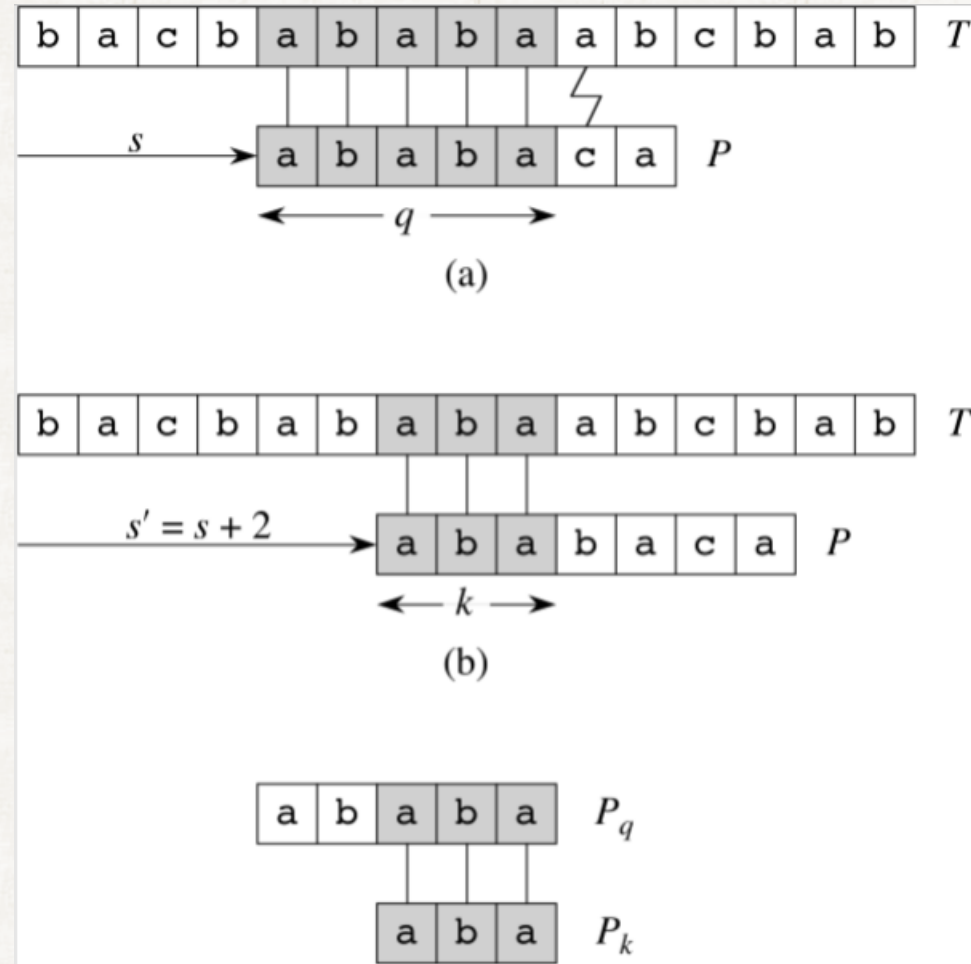
# KMP – Example for $\pi$

- The pattern  $P = ababaca$  aligns with a text  $T$  so that first  $q = 5$  characters match for shift  $s = 4$
- Using this knowledge it can be deduced that  $s + 1$  shift is invalid. However,  $s' = s + 2$  is potentially valid (matching  $k = 3$  characters)
- By comparing the pattern with itself, we can precompute such information via prefix function

$\pi$

Here for  $q = 5$ , largest value of  $k$  is 3. Thus  $\pi[5] = 3$  as given by the formula:

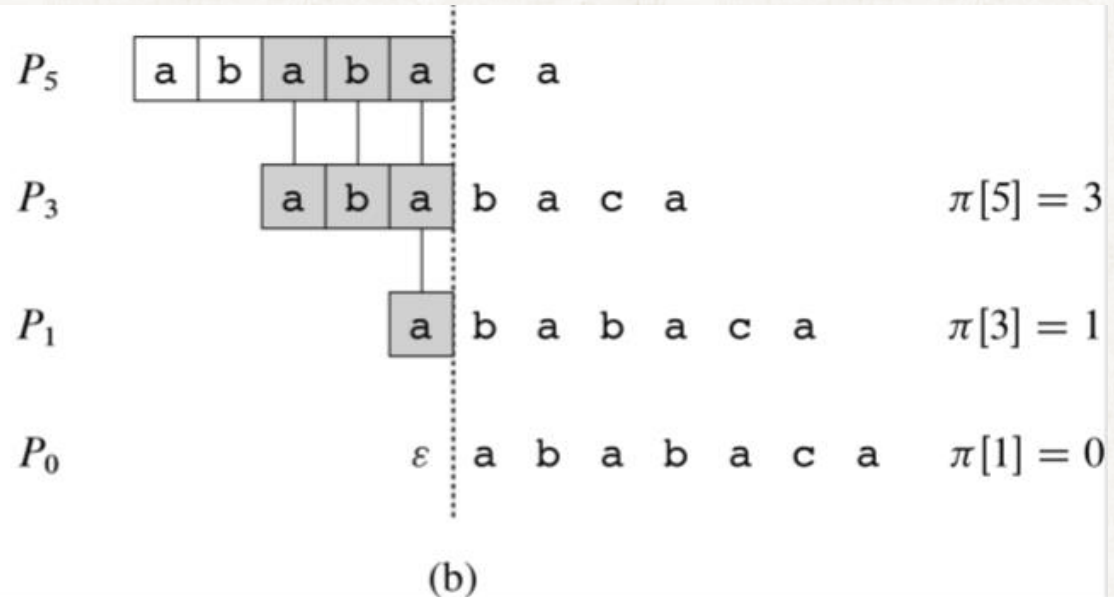
$$\pi[q] = \max\{k: k < q \text{ and } P_k \sqsupseteq P_q\}$$



# KMP – Example for $\pi$ Function

$i$	1	2	3	4	5	6	7
$P[i]$	a	b	a	b	a	c	a
$\pi[i]$	0	0	1	2	3	0	1

(a)



Determination of  $\pi[q]$  for the pattern  $P = ababaca$  and  $q = 5$  as given by the formula:

$$\pi[q] = \max\{k: k < q \text{ and } P_k \sqsupseteq P_q\}$$

# KMP-Example Solved

So we have

Text  $T$       **b a c b a b a b a b a c a c a**    and

Pattern  $P$       **a b a b a c a**    and

Prefix array  $\pi$     **0 0 1 2 3 0 1.**

**b a c b a b a b a b a c a c a**

**a b a b a c a**

**a b a b a c a**

,  $a \neq b$ , so we move  $P$  one over

,  $a = a$ , but  $b \neq c$ , we have one match,  
so we check  $\pi$  at slot one, which is 0, so  
we can skip no characters and move  $P$   
one over

**a b a b a c a**

,  $a \neq c$  so we move  $P$  one over

**a b a b a c a**

,  $a \neq b$  so we move  $P$  one over

**a b a b a c a**

, the first 5 characters match but then  
 $c \neq b$ , so we check  $\pi$  at slot 5 which is 3,  
so we can skip 3 characters and move  $P$   
 $5 - 3 = 2$  over

**a b a b a c a**

, now we have a full match, so we are  
done

# KMP-Pseudocodes

KMP-MATCHER ( $T, P$ )

$n = T.length$

$m = P.length$

$\pi = \text{COMPUTE-PREFIX-FUNCTION}(P)$

$q = 0$

for  $i = 1$  to  $n$

    while  $q > 0$  and  $P[q + 1] \neq T[i]$

$q = \pi[q]$

    if  $P[q + 1] == T[i]$

$q = q + 1$

    if  $q == m$

        print "Pattern occurs with shift"  $i - m$

$q = \pi[q]$

// number of characters matched

// scan the text from left to right

// next character does not match

// next character matches

// is all of  $P$  matched?

// look for the next match



# KMP-Pseudocodes

KMP-PREFIX-FUNCTION( $P$ )

$m = P.length$

let  $\pi[1..m]$  be a new array

$\pi[1] = 0$

$k = 0$

for  $q = 2$  to  $m$

    while  $k > 0$  and  $P[k + 1] \neq P[q]$

$k = \pi[k]$

    if  $P[k + 1] == P[q]$

$k = k + 1$

$\pi[q] = k$

return  $\pi$

# HW

## Exercises

- 26.3-3
- 15.1-1, 15.1-2, 15.1-3, 15.1-5
- 15.4-1, 15.4-2, 5.4-4
- 32.4-1, 32.4-3