

# SUFFIX TREES

# Exact String matching: a brief history

- Naive algorithm
- Knuth-Morris-Pratt 1977.
- Boyer-Moore 1977.
- Suffix Trees: Linear time.
  - Weiner 1973 - Right to left
  - McCreight 1978 - Left to right.
  - Ukkonen 1995 - Online, Left to right.

# Suffix Trees Uses

- Suitable for DNA searching because:
  - small alphabet (4 chars),
  - size of  $T = m$  is huge (billion)
  - many patterns to search on  $T$ .
  - Linear time algorithms (large constants).
- Music database

# Finding repeats in DNA

- human chromosome 3
  - the first 48 999 930 bases
  - 31 min cpu time (8 processors, 4 GB)
- 
- Human genome:  $3 \times 10^9$  bases
  - Tree(HumanGenome) feasible

# DNA string

tgagacggagtctcgctctgtcgcccaggctg  
gagtgcagtggcggatctcggtcactgca  
agctccgcctcccggttcacgccatttcctg  
cctcagcctcccaagtagtagctggactacagg  
cgccccccactacgccccggctaattttgtatt  
tttagtagagacggggttcaccgttagccg  
ggatggtctcgatctcctgacctcgatccgc  
ccgcctcggcctcccaaagtgctggattaca  
ggcgt....

# Example: Longest repeat in a DNA?

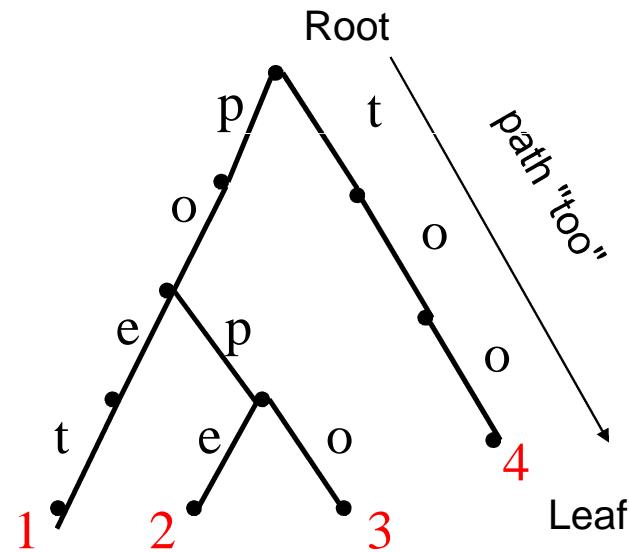
**Occurrences at: 28395980, 28401554r      Length: 2559**

# Suffix Trees

- Specialized form of keyword trees
- New ideas
  - preprocess text  $T$ , not pattern  $P$ 
    - $O(m)$  preprocess time
    - $O(n+k)$  search time
      - $k$  is number of occurrences of  $P$  in  $T$
  - edges can have string labels

# Keyword Tree example

- P = {poet, pope, popo, too}

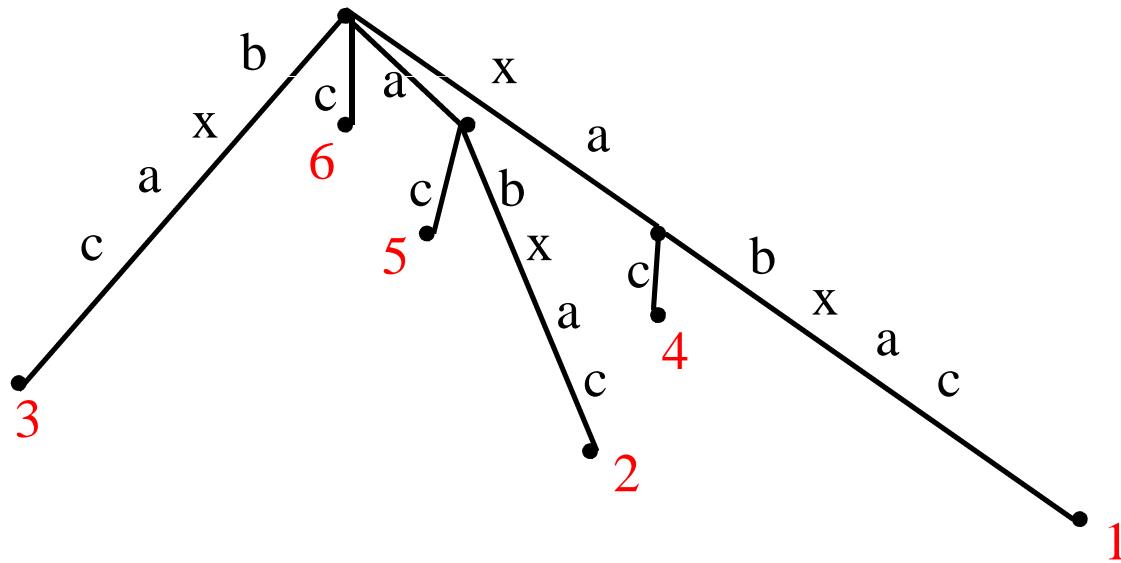


# Suffix Tree

- Take any  $m$  character string  $S$  like "xabxac"
- Set of keywords is the set of **suffixes** of  $S$ 
  - {xabxac, abxac, bxac, xac, ac, c}
- Suffix tree  $T$  is essentially the **keyword tree** for this set of suffixes of  $S$ ,
- Changes:
  - Assumption: no suffix is a prefix of another suffix (can be a substring, but not a prefix)
    - Assure this by adding a character \$ to end of  $S$
  - Internal nodes except root must have at least 2 children
  - edges can be labeled with strings

# Example: Suffix tree for "xabcxac"

- {xabxac, abxac, bxac, xac, ac, c}



# Notation

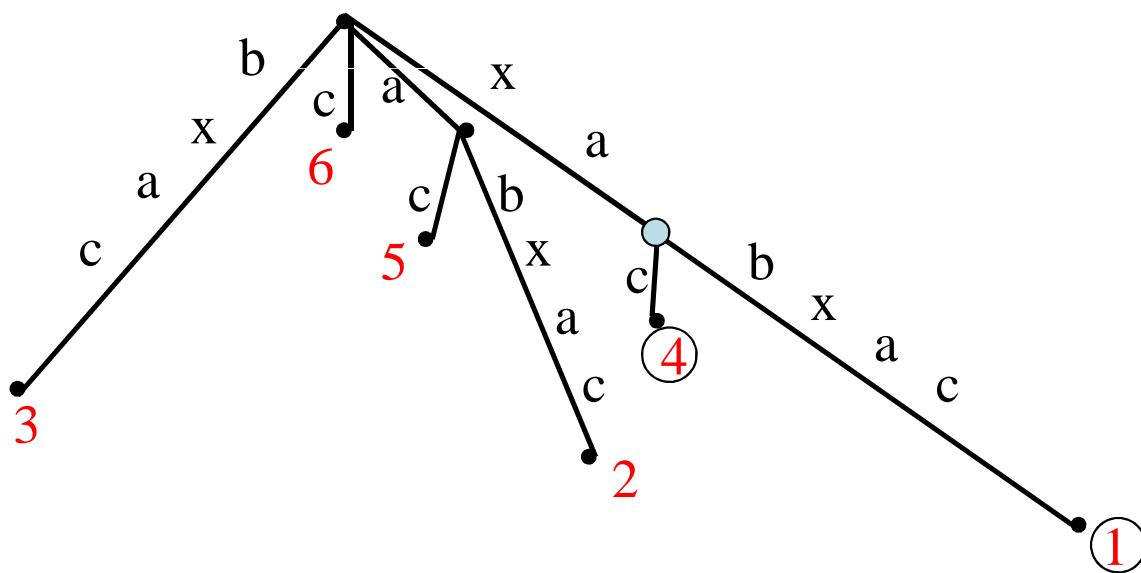
- Label of a path from  $r$  (root) to a node  $v$  is the concatenation of labels on edges from  $r$  to  $v$
- label of a node  $v$  is  $L(v)$ , path label from  $r$  to  $v$
- string-depth of  $v$  is number of characters in  $v$ 's label  $L(v)$
- Comment: In constructing suffix trees, we will need to be able to “split” edges “in the middle”

# Suffix trees for exact matching

- Build suffix tree  $T$  for text.
- Match pattern  $P$  against tree starting at root until
  - Case 1,  $P$  is completely matched
    - Every leaf below this match point is the starting location of  $P$  in  $T$
  - Case 2: No match is possible
    - $P$  does not occur in  $T$

# Example: suffix tree match( $P$ , $T$ )

- $T = xabxac$   
    suffixes of  $T = \{xabxac, abxac, bxac, xac, ac, c\}$
  - Pattern  $P_1 : xa \dots$  matches at 1, 4.
  - Pattern  $P_2 : xb \dots$  no matches.



# Running Time Analysis

- Build suffix tree:
  - Ukkonen's Linear time algorithm:  $O(m)$
  - This is preprocessing of data.
- Search time:
  - $O(n+k)$  where  $k$  is the number of occurrences of  $P$  in  $T$
  - $O(n)$  to find match point if it exists
  - $O(k)$  to find all leaves below match point

# Building trees: $O(m^2)$ algorithm

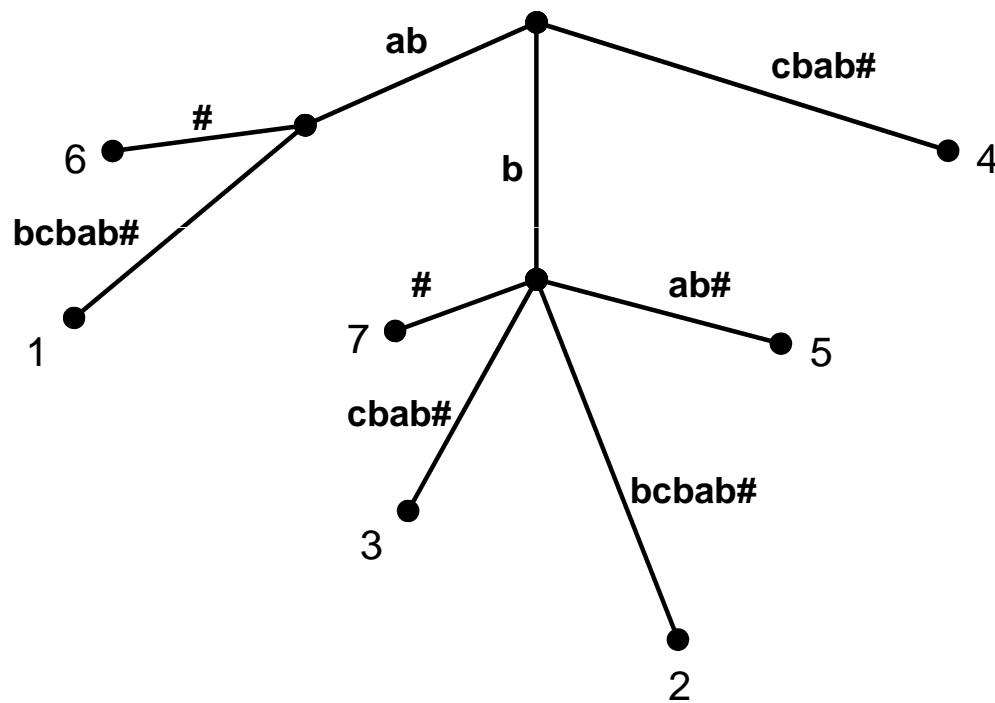
- Initialize
  - One edge for the entire string  $S[1..m]\$$
- For  $i = 2$  to  $m$ 
  - Add suffix  $S[i..m]$  to suffix tree
    - Find match point for string  $S[i..m]$  in current tree
    - If in “middle” of edge, create new node  $w$
    - Add remainder of  $S[i..m]$  as edge label to suffix  $i$  leaf
- Running Time
  - $O(m-i)$  time to add suffix  $S[i..m]$

# Exercises: build suffix trees

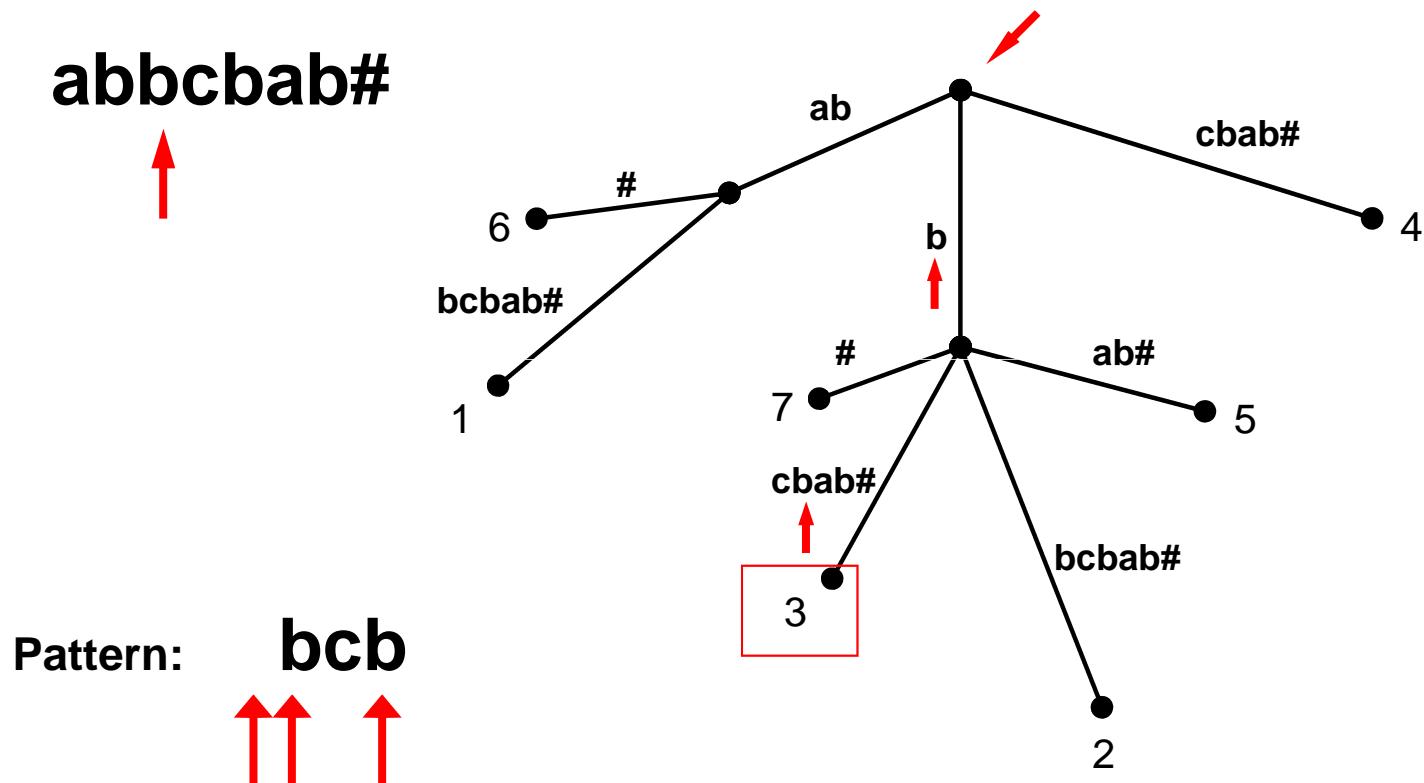
1. Build a suffix tree for: "abbcbab#"
2. Search for "bcb" in the tree.

# Suffix Tree - Solution

**abcbab#**

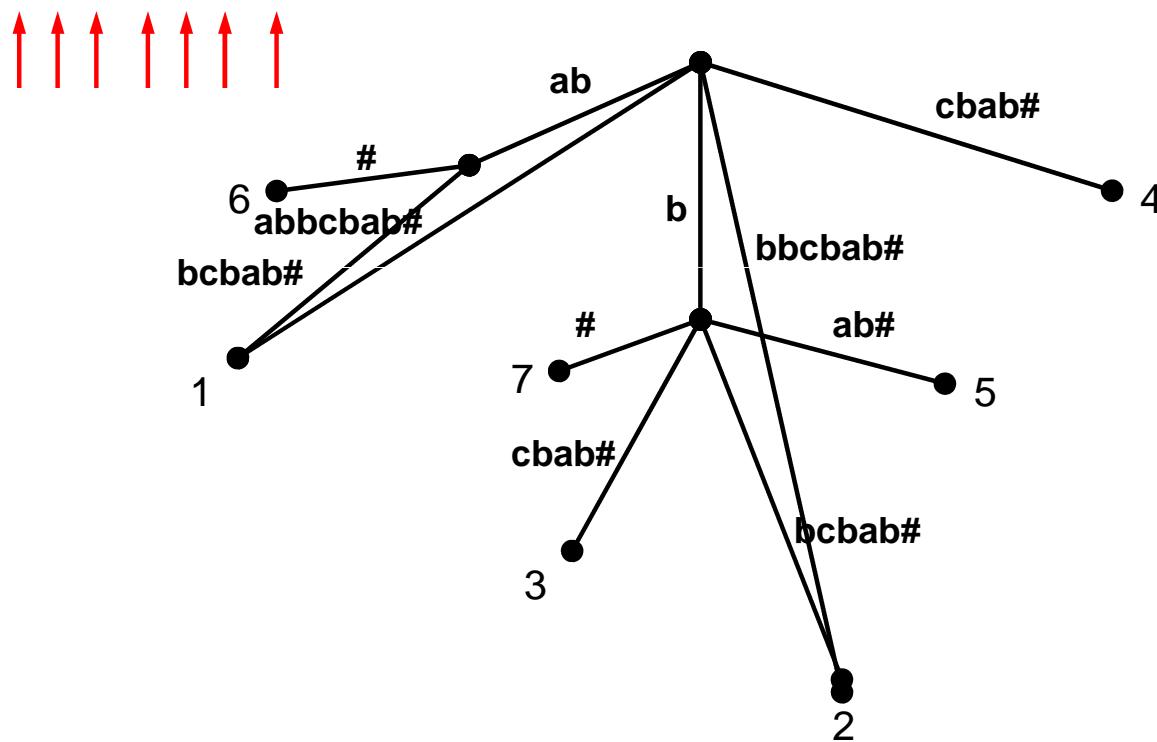


# Suffix Trees – searching a pattern



# Suffix Trees – naive construction

**abbcbab#**

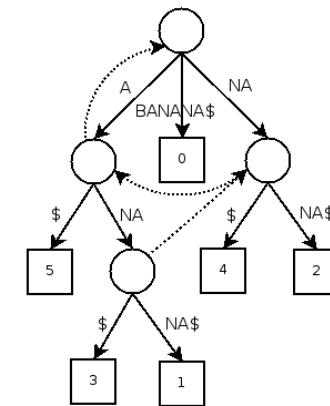


# Drawbacks

- Suffix trees consume a lot of space
- It is  $O(n)$
- The constant is quite big
- Notice that if we indeed want to traverse an edge in  $O(1)$  time then we need an array of pointers of size  $|\Sigma|$  in each node (useful when alphabet is small, e.g. 4 chars in DNA and RNA).

# Ukkonen Algorithm

## Suffix trees



# Ukkonen Algorithm

Ukkonen algorithm [1995] is the fastest and well performing algorithm for building a suffix tree in linear time.

The basic idea is constructing iteratively the **implicit suffix trees** for  $S[1..i]$  in the following way:

**Construct tree<sub>1</sub>**

for  $i = 1$  to  $m-1$  // do **phase  $i+1$**

    for  $j = 1$  to  $i+1$  // do **extension  $j$**

1. find the end of the path from the root with  
label  $S[j...i]$  in the current tree.

2. Extend the path adding character  $S(i+1)$ ,  
so that  $S[j...i+1]$  is in the tree.

# Ukkonen: 3 extension rules

The extension will follow one of the next three rules, being  $b = S[j..i]$ :

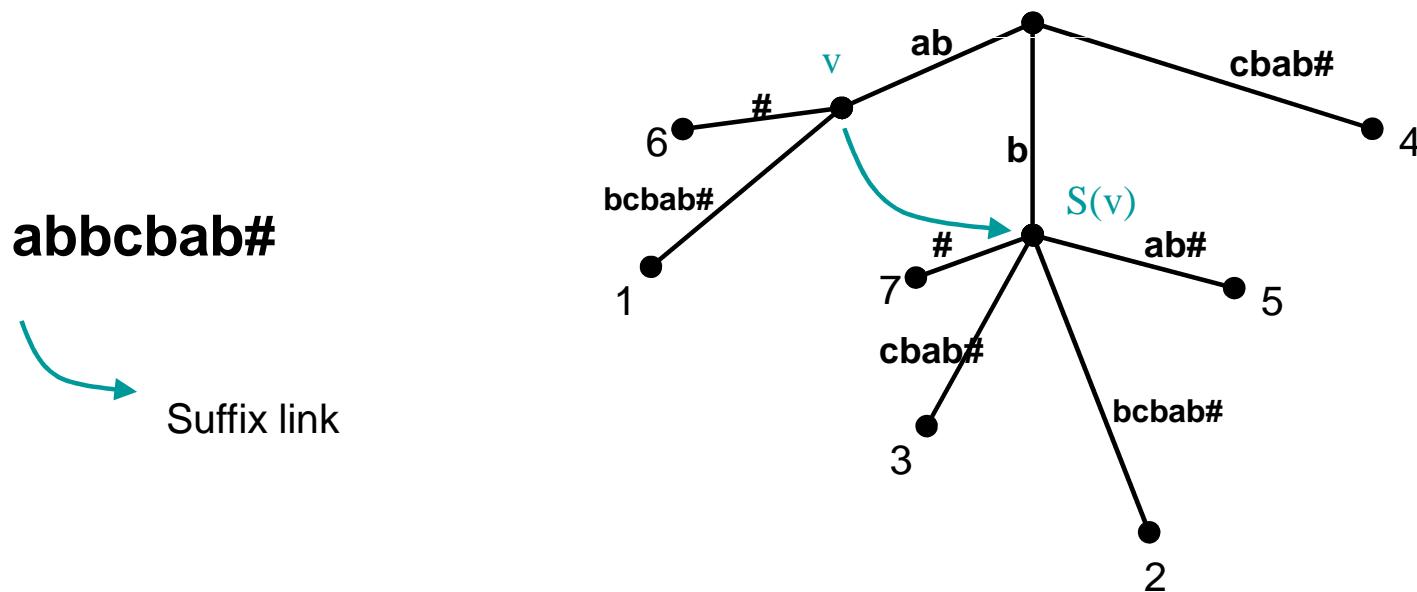
1.  $b$  ends at a leaf. Add  $S(i+1)$  at the end of the label of the path to the leaf
2. There's one path continuing from the end of  $b$ , but none starting with  $S(i+1)$ . Add a node at the end of  $b$  and a path starting from the new node with label  $S(i+1)$ , terminating in a leaf with number  $j$ .
3. There's one path from the end of  $b$  starting with  $S(i+1)$ . In this case do nothing.

# Ukkonen Algorithm - II

The main idea to speed up the construction of the tree is the concept of **suffix link**.

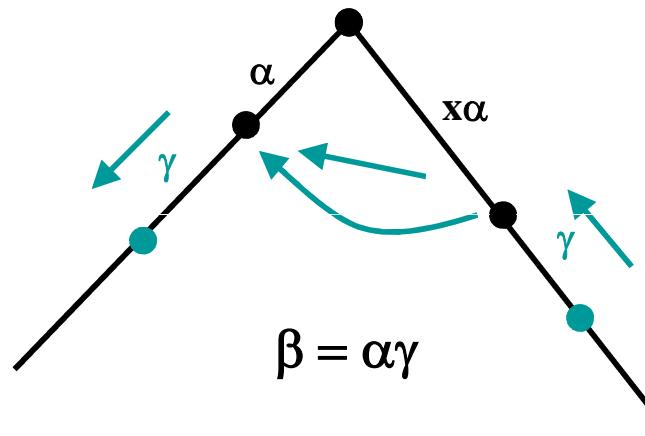
Suffix links are pointers from a node  $v$  with path label  $xa$  to a node  $s(v)$  with path label  $a$  ( $a$  is a string and  $x$  a character).

The interesting feature of suffix trees is that every internal node, except the root, **has** a suffix link towards another node.



# Suffix Trees – Ukkonen Algorithm - III

With suffix links, we can speed up the construction of the ST



In addition, every node can be crossed in constant time, just keeping track of the label's length of every single edge. This can be done because no two edges exiting from a node can start with the same character, hence a single comparison is needed to decide which path must be taken.

Anyway, using suffix links, complexity is still quadratic.

# Ukkonen – speed up from $n^2$ to $n$

Storing the path labels explicitly will cost a quadratic space. Anyway, each edge need only constant space, i.e. two pointers, one to the beginning and one to the end of the substring it has as label.

To complete the speed up of the algorithm, we need the following observations:

Once a leaf is created, it will remain forever a leaf.

Once a phase rule 3 is used, all successive extensions make use of it, hence we can ignore them.

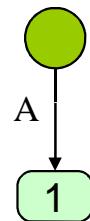
If in phase  $i$  the rule 1 and 2 are applied in the first  $j_i$  moves, in phase  $i+1$  the first  $j_i$  extensions can be made in constant time, simply adding the character  $S(i+2)$  at the end of the paths to the first  $j_i$  leaves (we will use a global variable  $e$  to do this). Hence the extensions will be computed explicitly from  $j_i+1$ , reducing their global number to  $2m$ .

Ukkonen: construct  
suffix tree in  
linear time

# Ukkonen's linear time construction

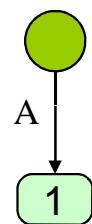
ACTAATC

A



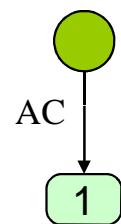
ACTAATC

AC



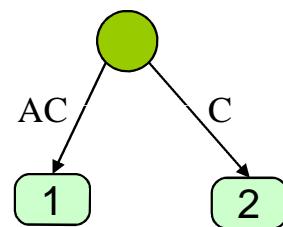
ACTAATC

AC



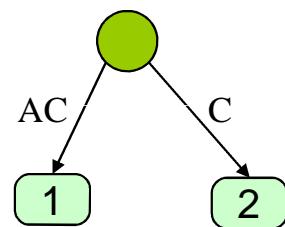
ACTAATC

AC



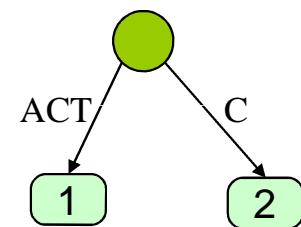
ACTAATC

ACT



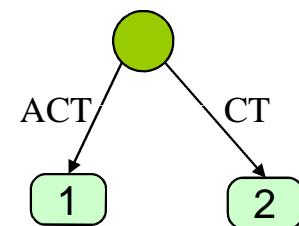
ACTAATC

ACT



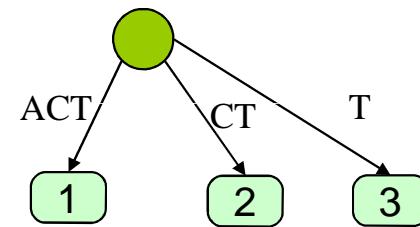
ACTAATC

ACT



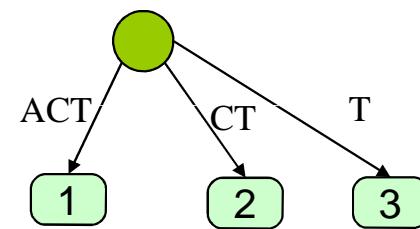
ACTAATC

ACT



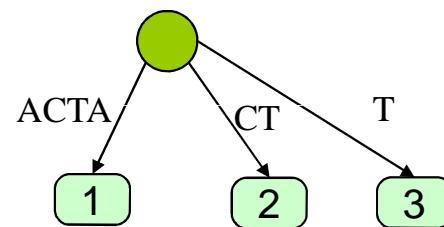
ACTAATC

ACTA



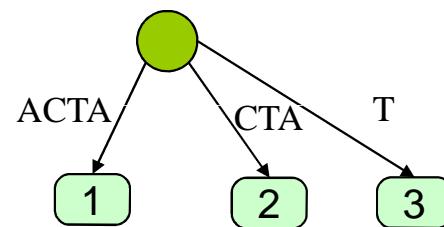
ACTAATC

ACTA



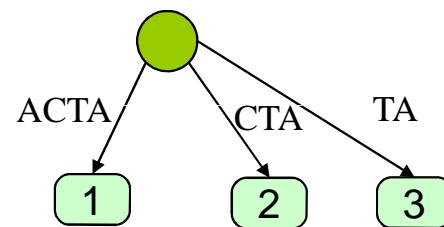
ACTAATC

ACTA



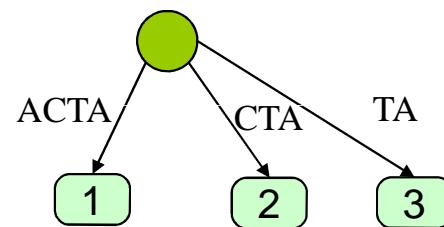
ACTAATC

ACTA



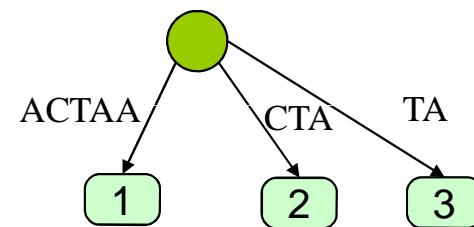
ACTAATC

ACTAA



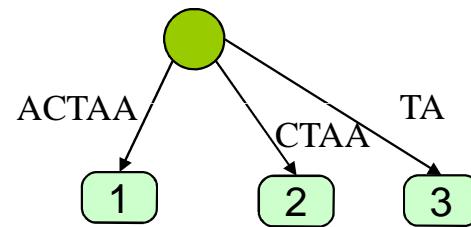
ACTAATC

ACTAA



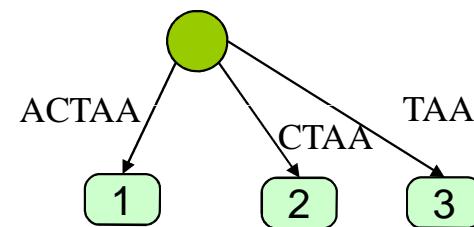
ACTAATC

ACTAA



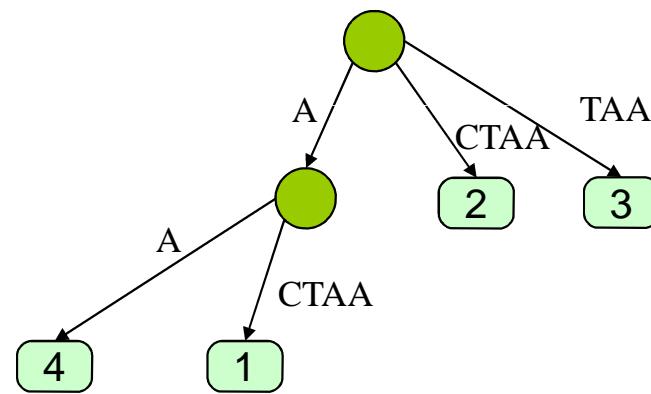
ACTAATC

ACTAA



ACTAATC

ACTAA



# Phases & extensions

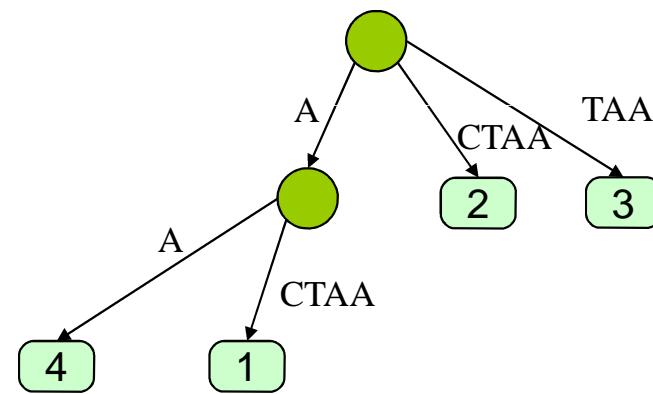
- Phase **i** is when we add character **i**



- In phase **i** we have **i extensions** of suffixes

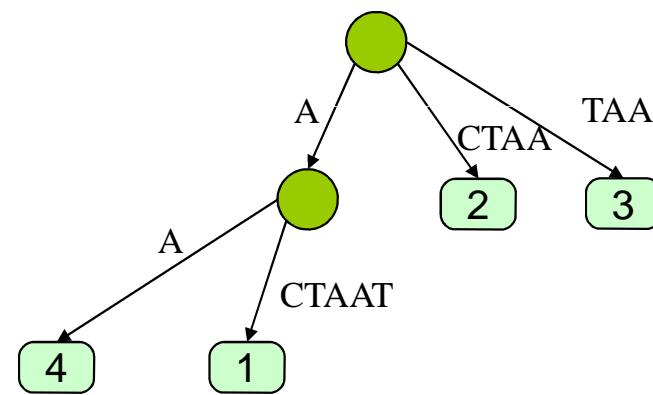
ACTAATC

ACTAAT



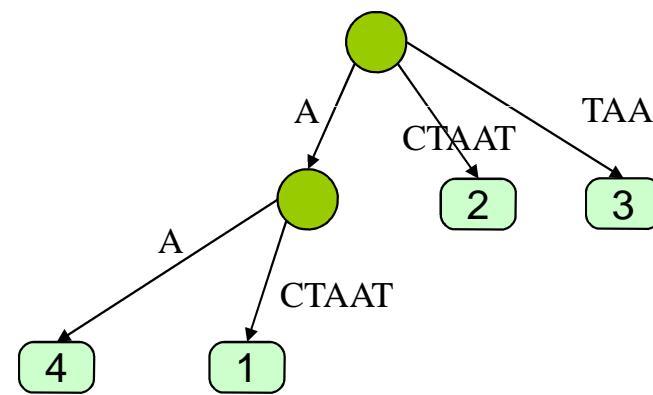
ACTAATC

ACTAAT



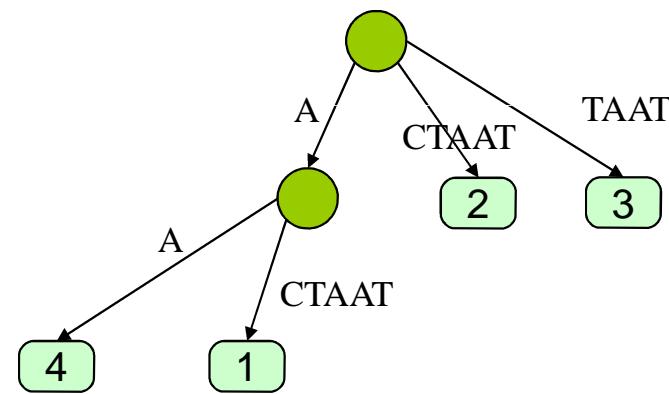
ACTAATC

ACTAAT



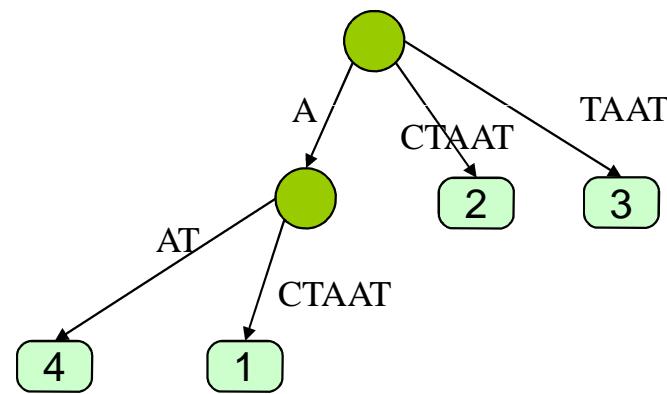
ACTAATC

ACTAAT



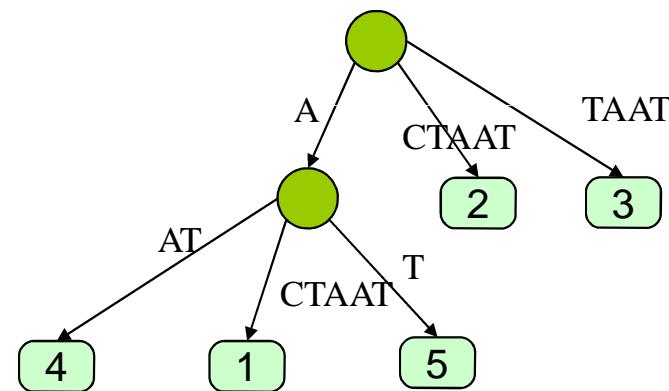
ACTAATC

ACTAAT



ACTAATC

ACTAAT

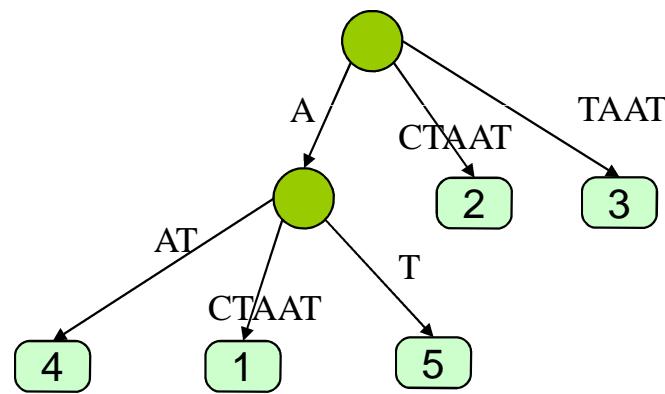


# Extension rules

- Rule 1: The suffix ends at a leaf, you add a character on the edge entering the leaf
- Rule 2: The suffix ended internally and the extended suffix does not exist, you add a leaf and possibly an internal node
- Rule 3: The suffix exists and the extended suffix exists, you do nothing

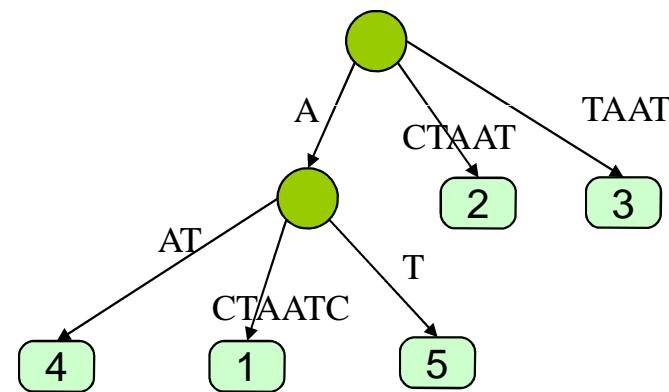
ACTAATC

ACTAATC



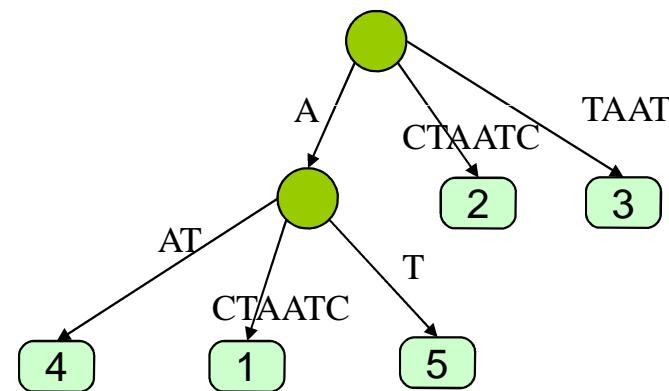
ACTAATC

ACTAATC



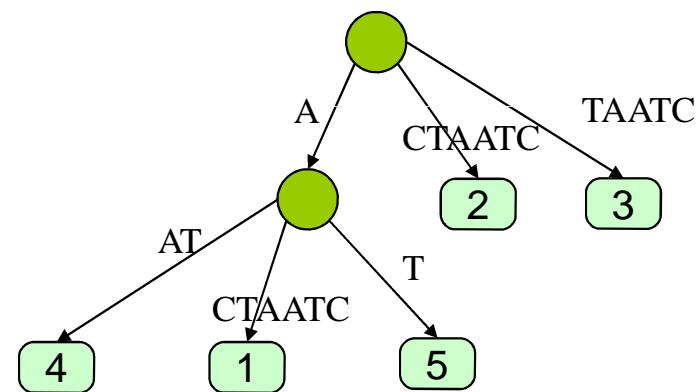
ACTAATC

ACTAATC



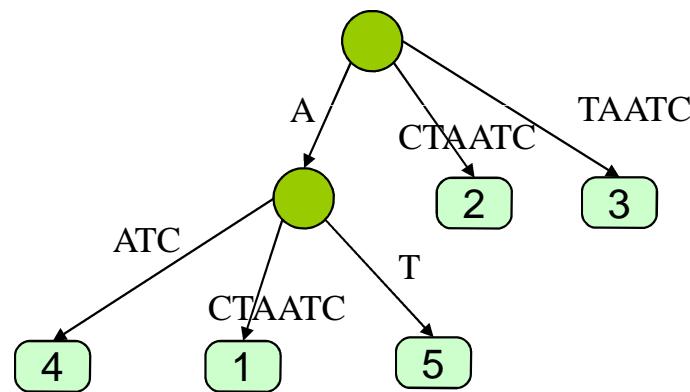
ACTAATC

ACTAATC



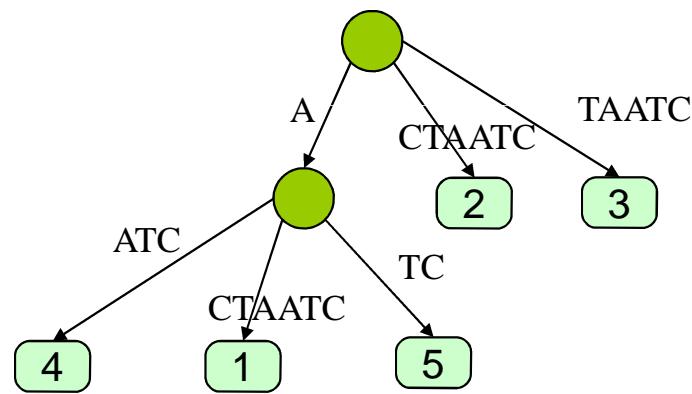
ACTAATC

ACTAATC



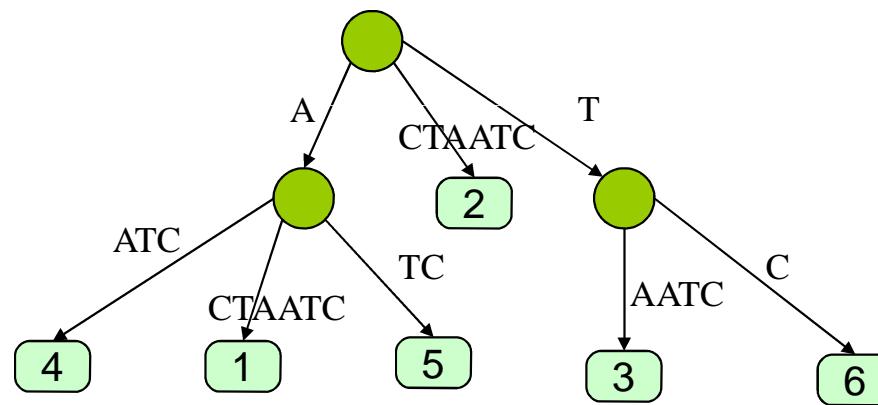
ACTAATC

ACTAATC

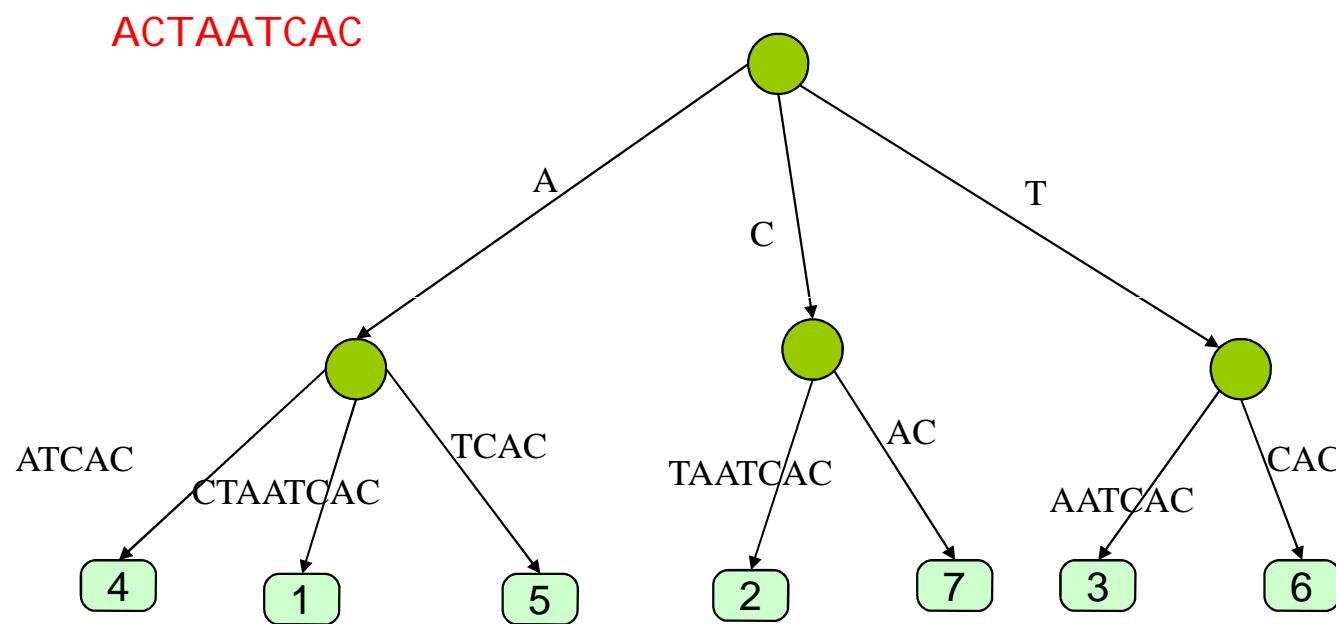


ACTAATC

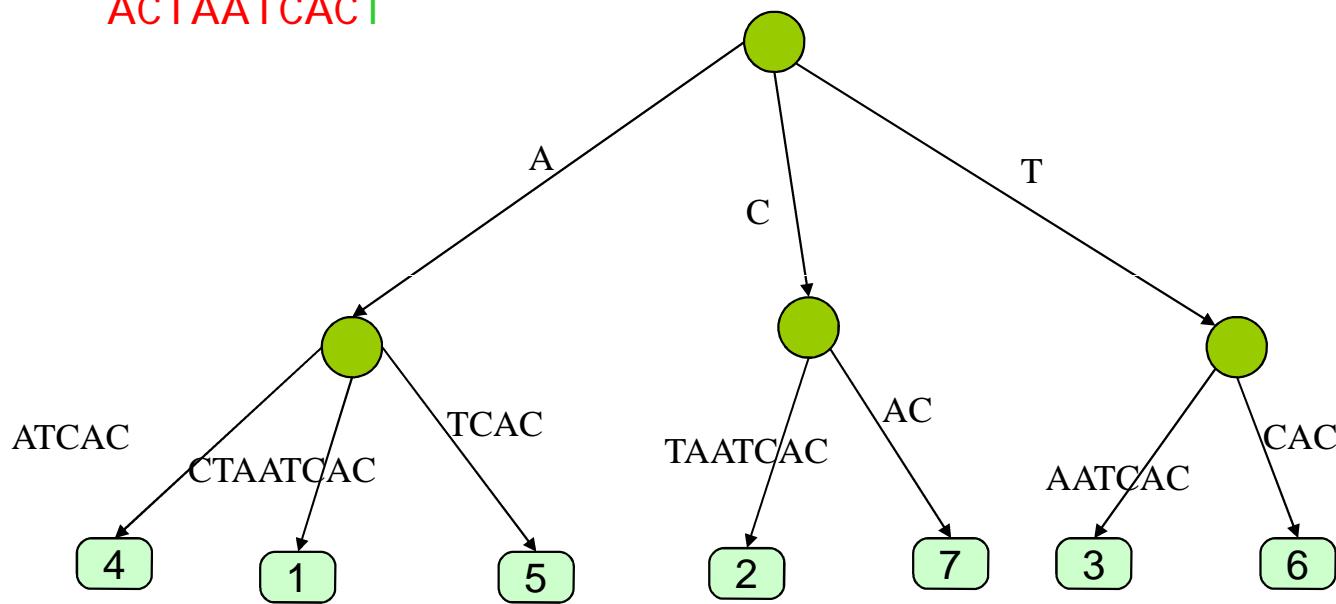
ACTAATC

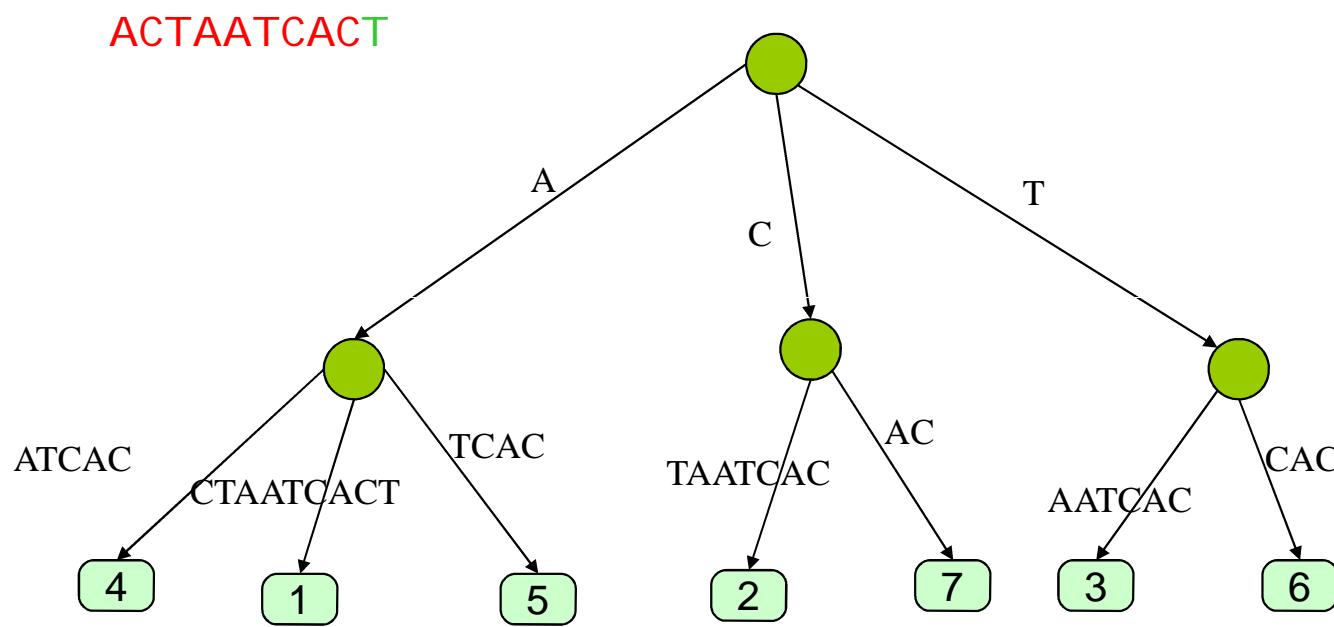


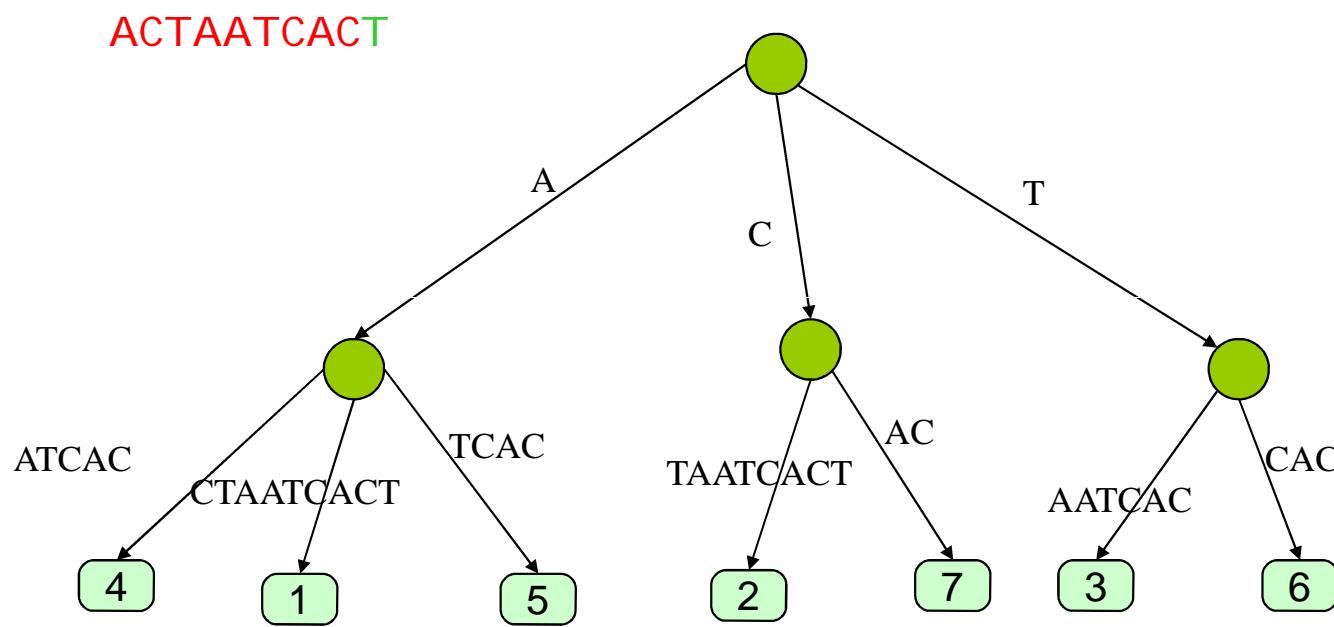
# Skip forward..

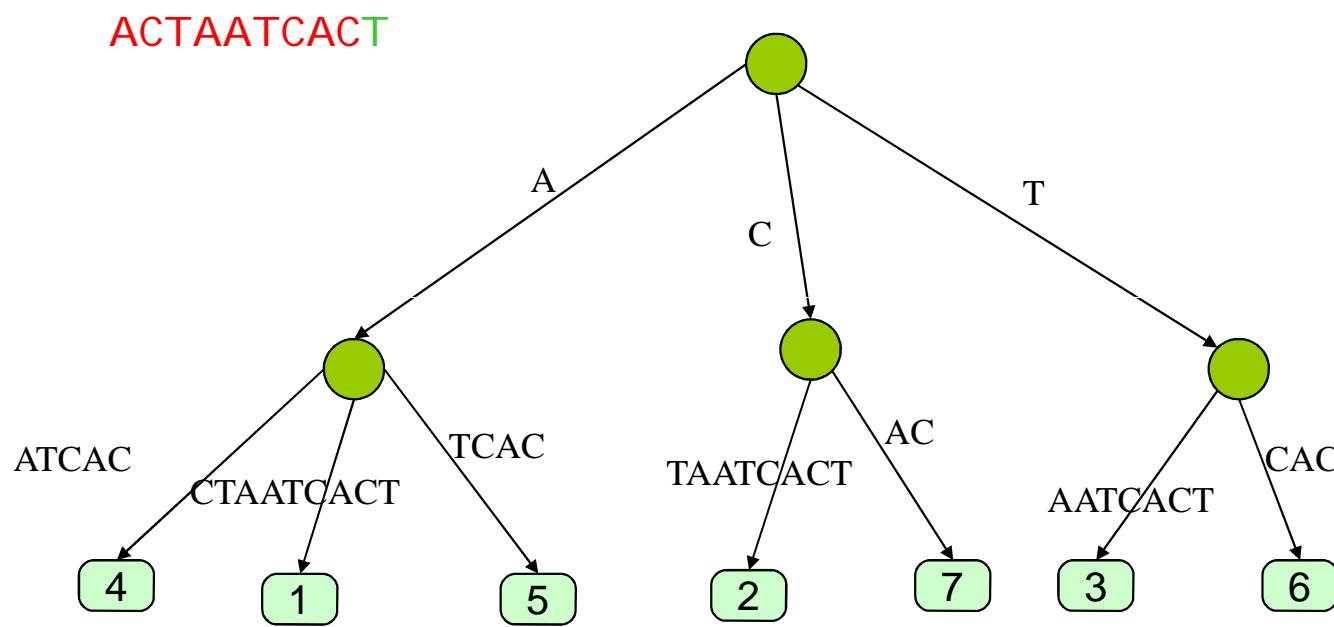


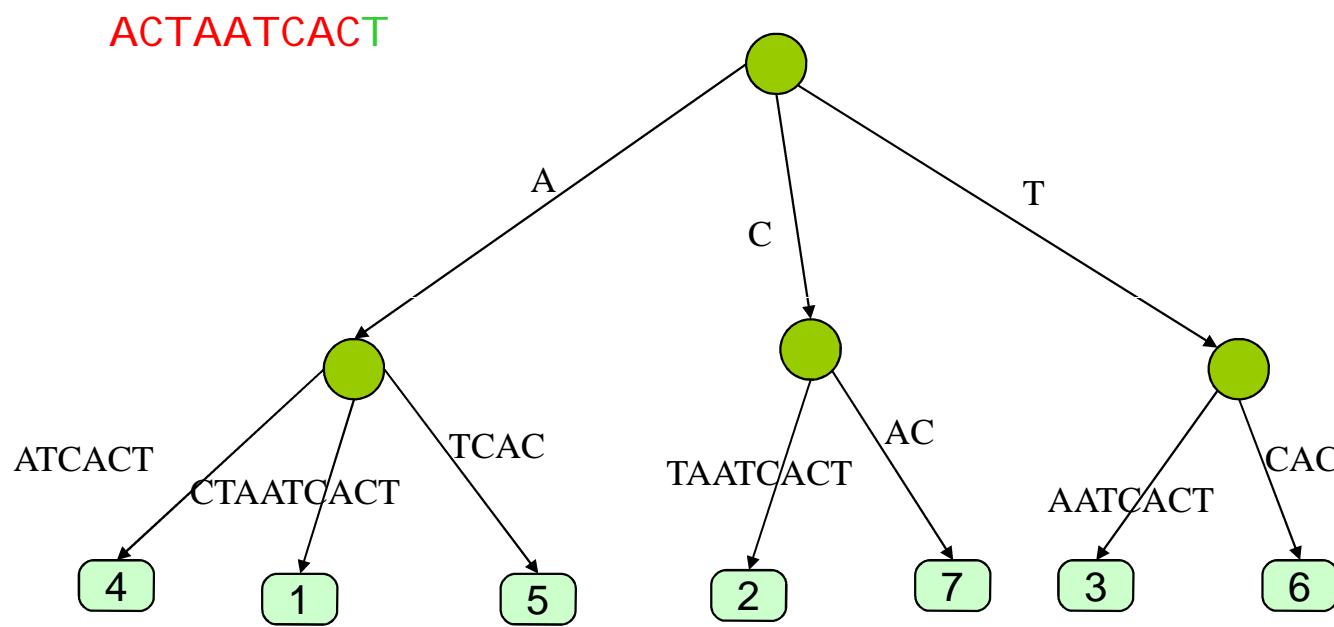
ACTAATCACT

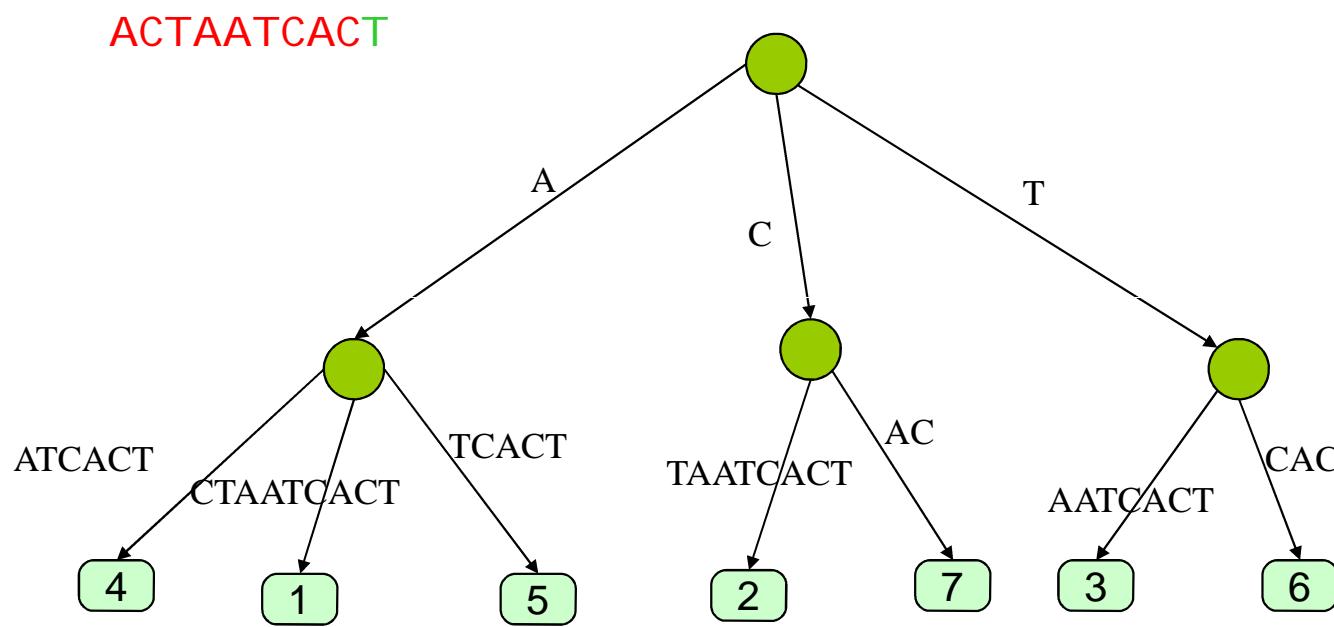


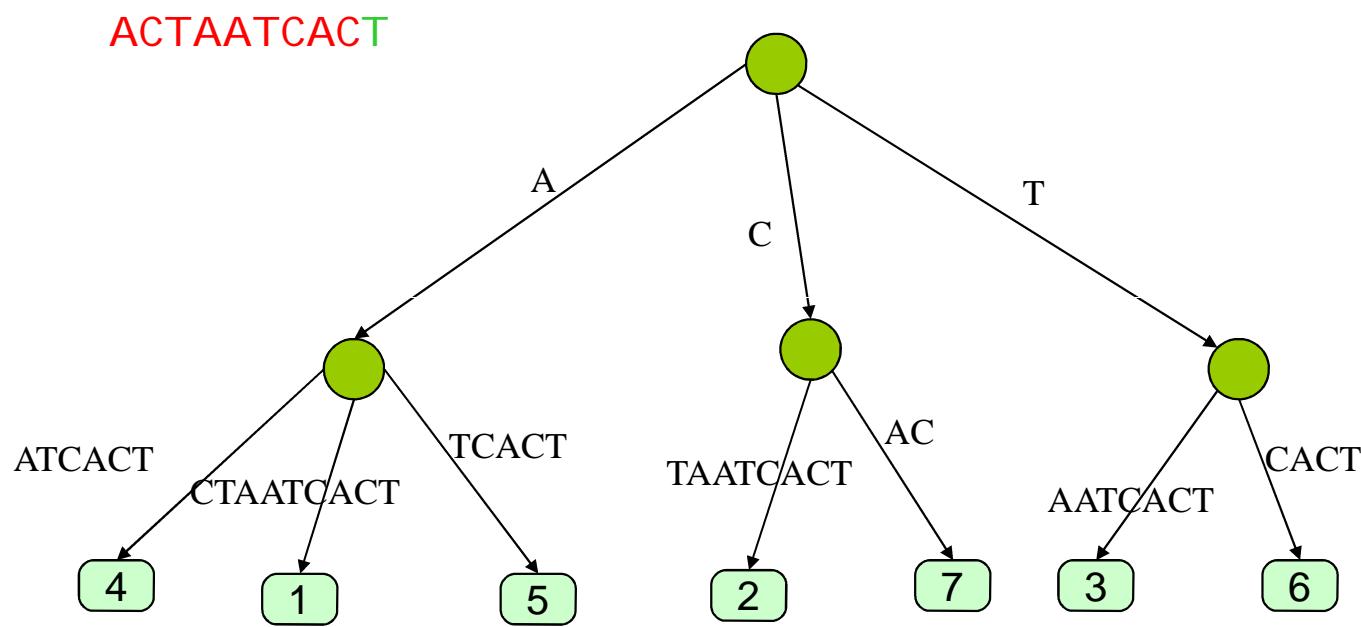


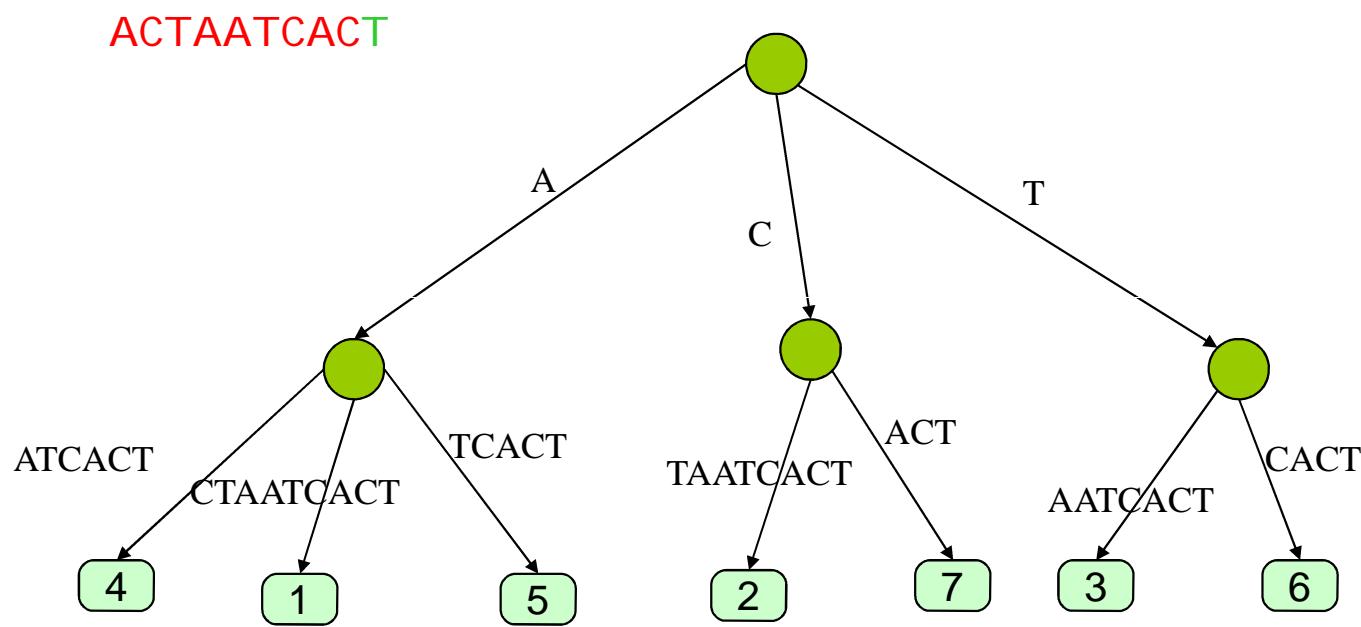


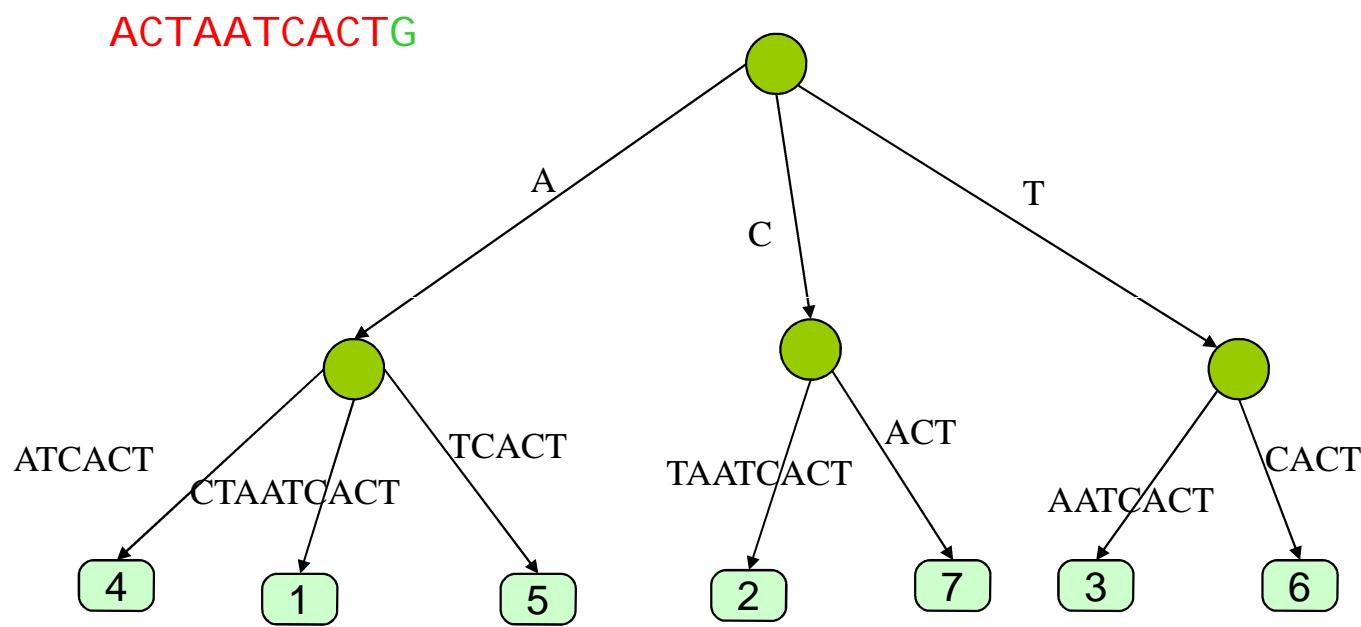


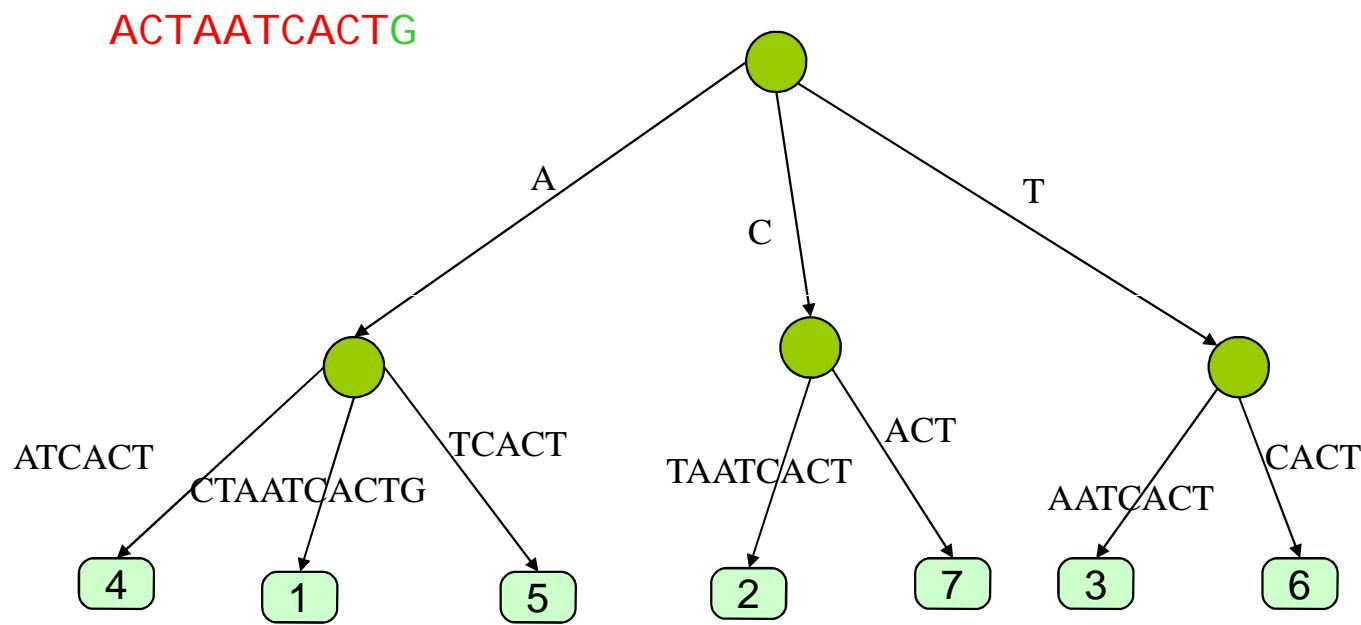


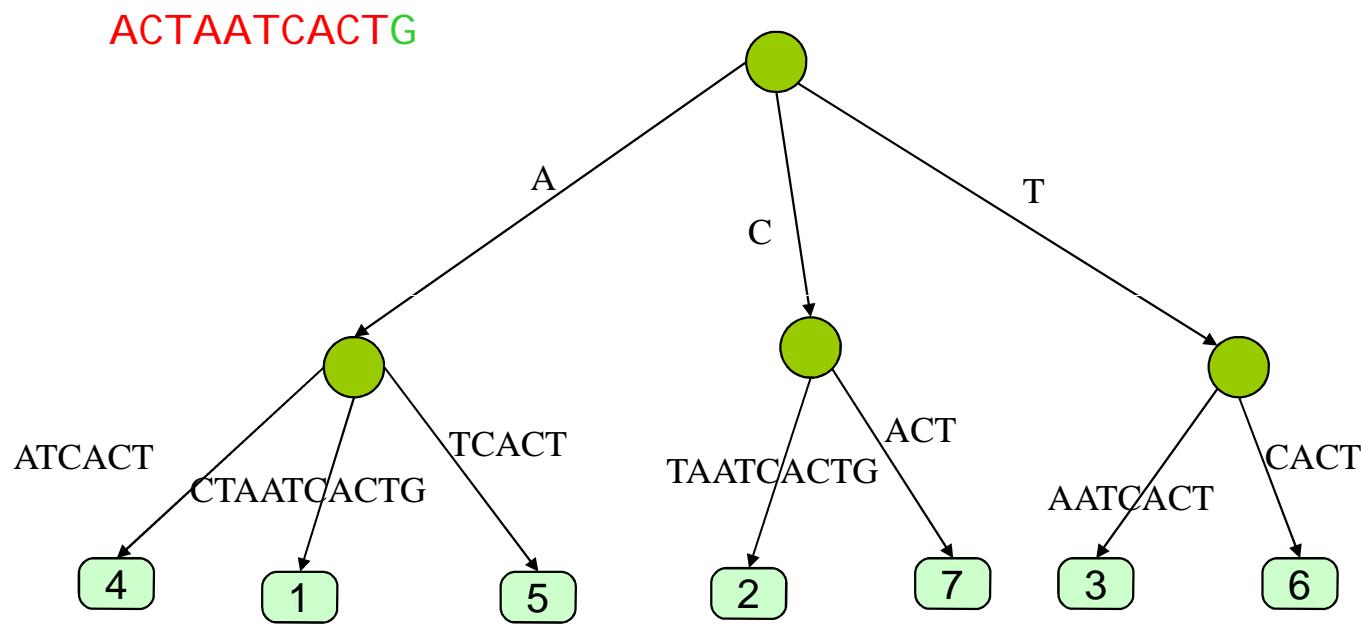


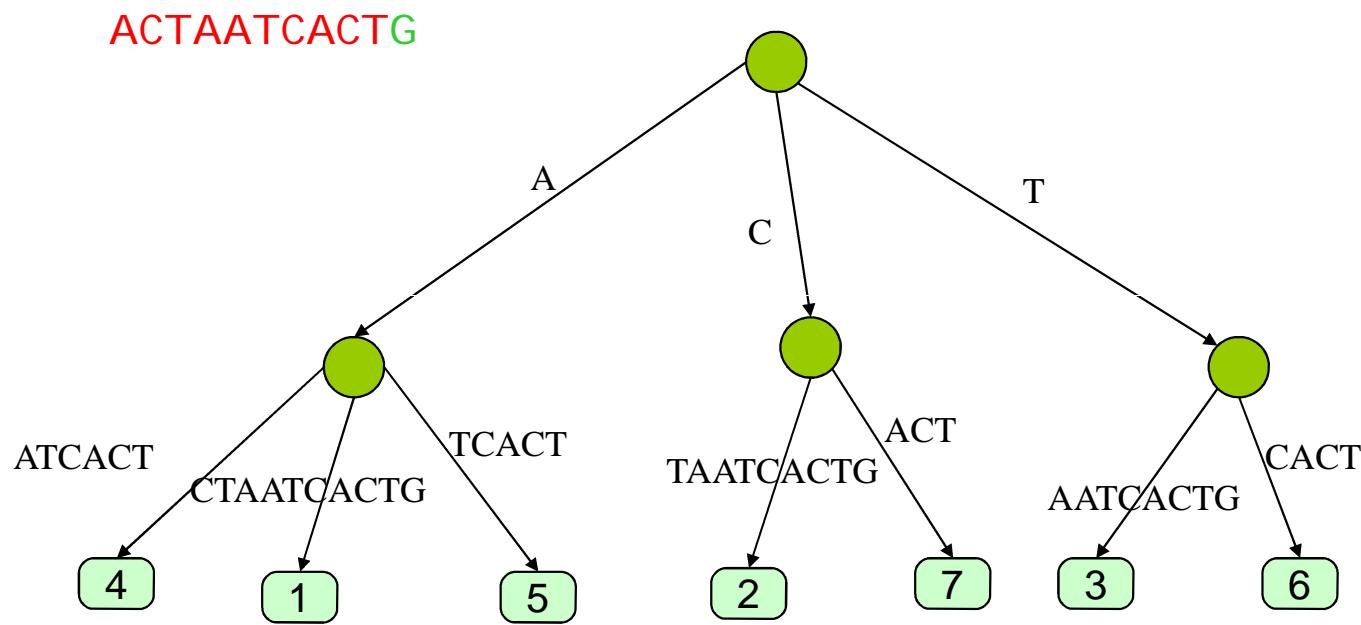


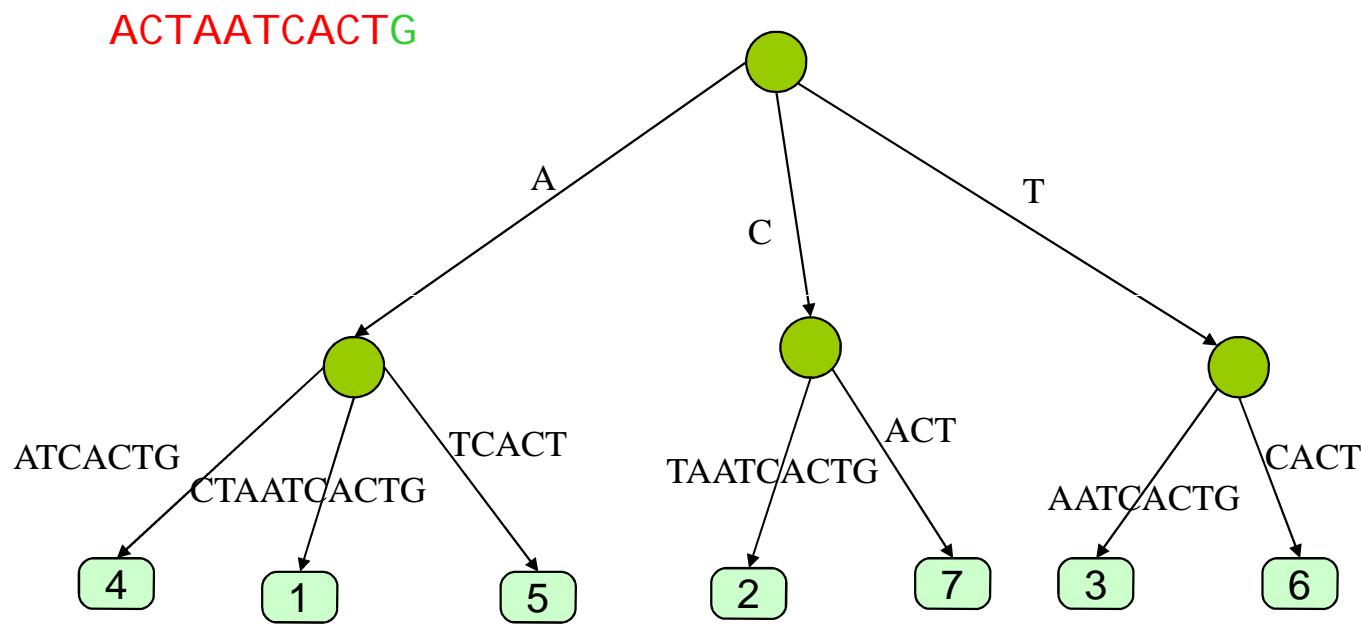


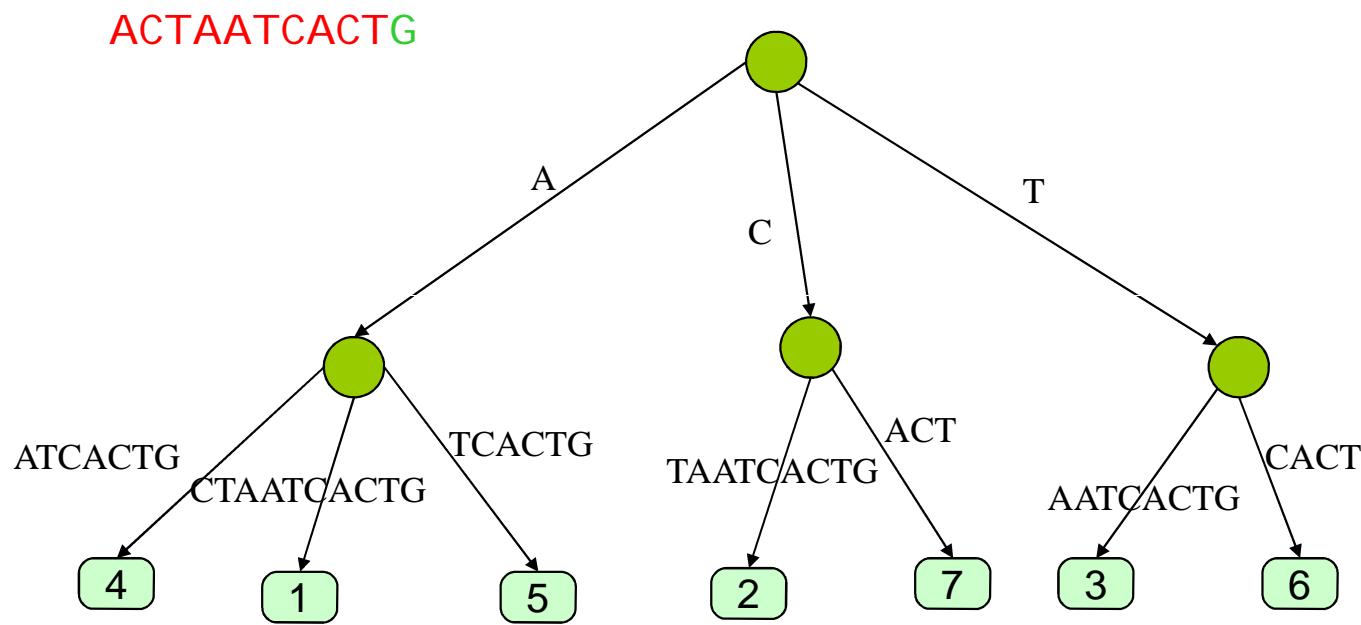


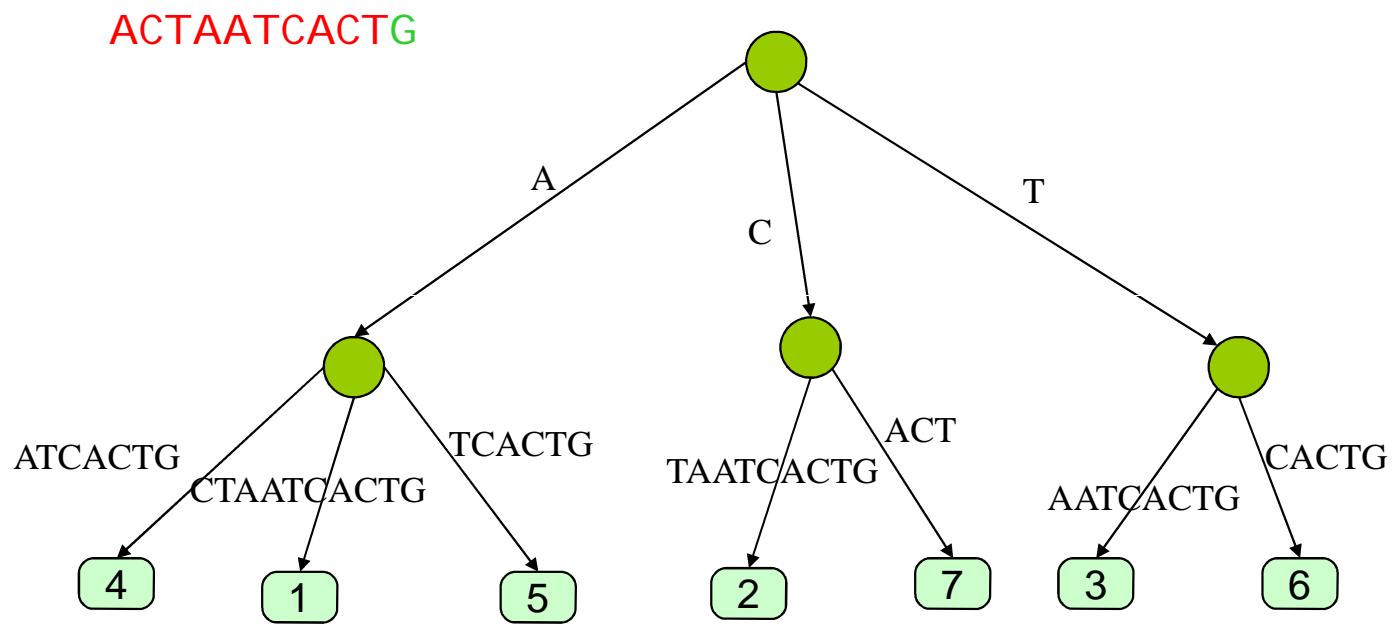


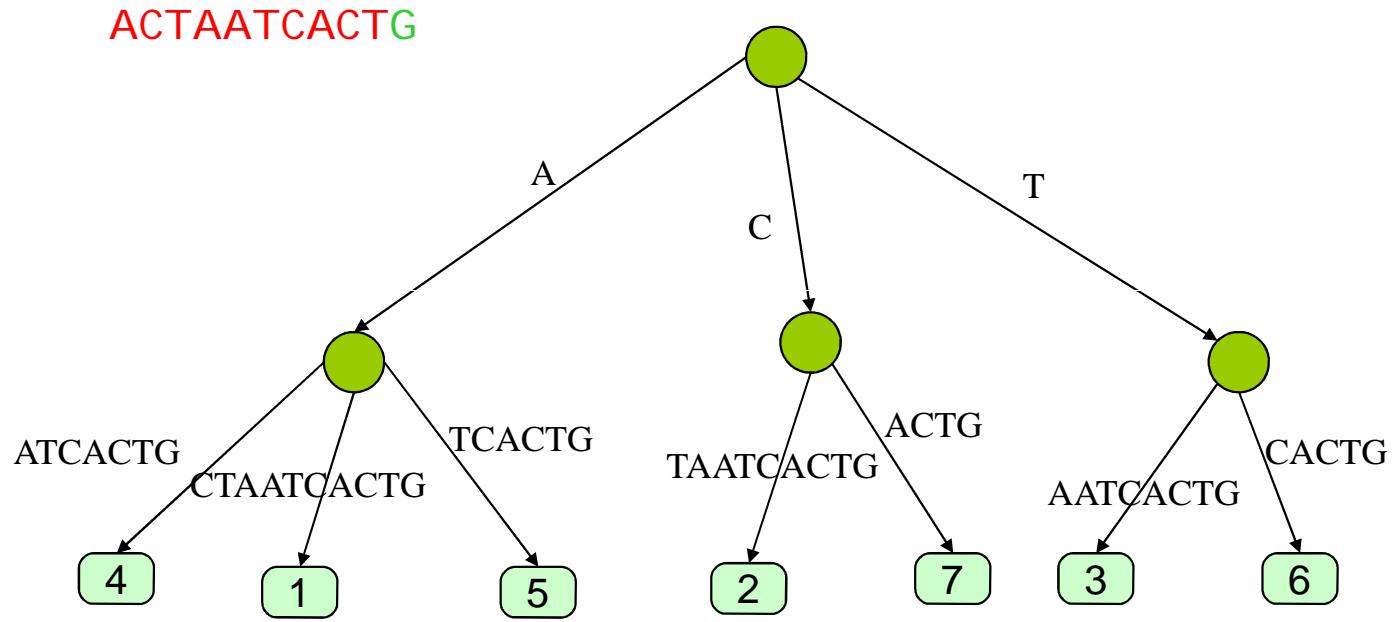


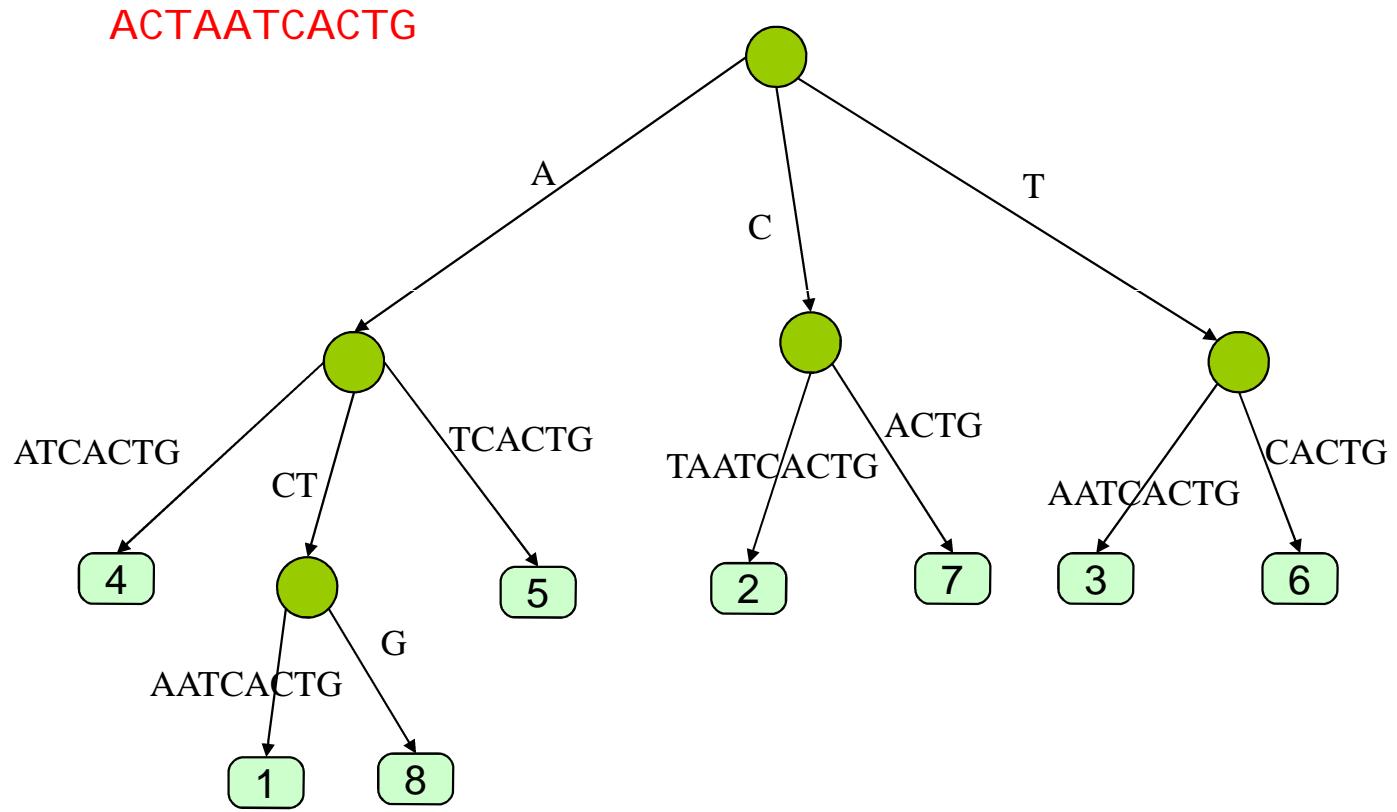


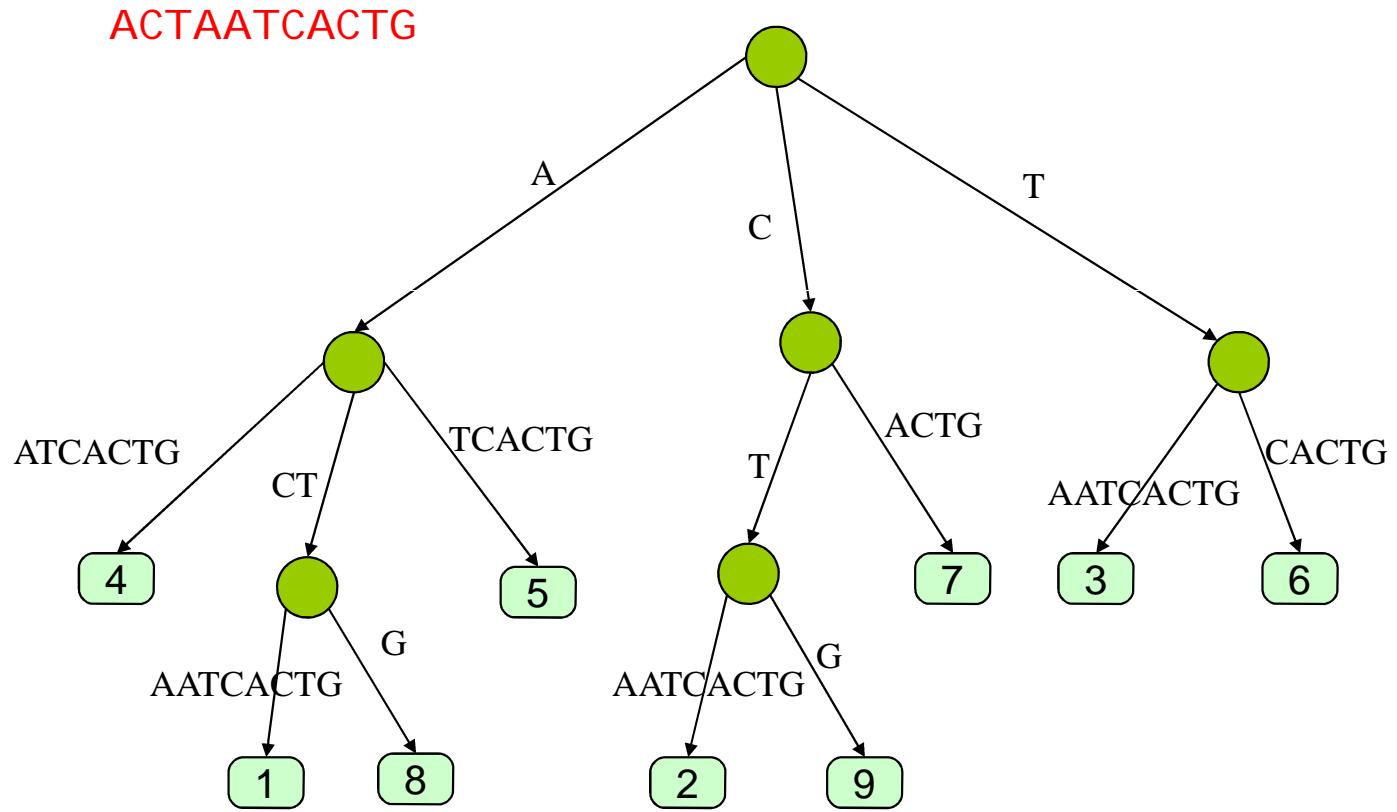


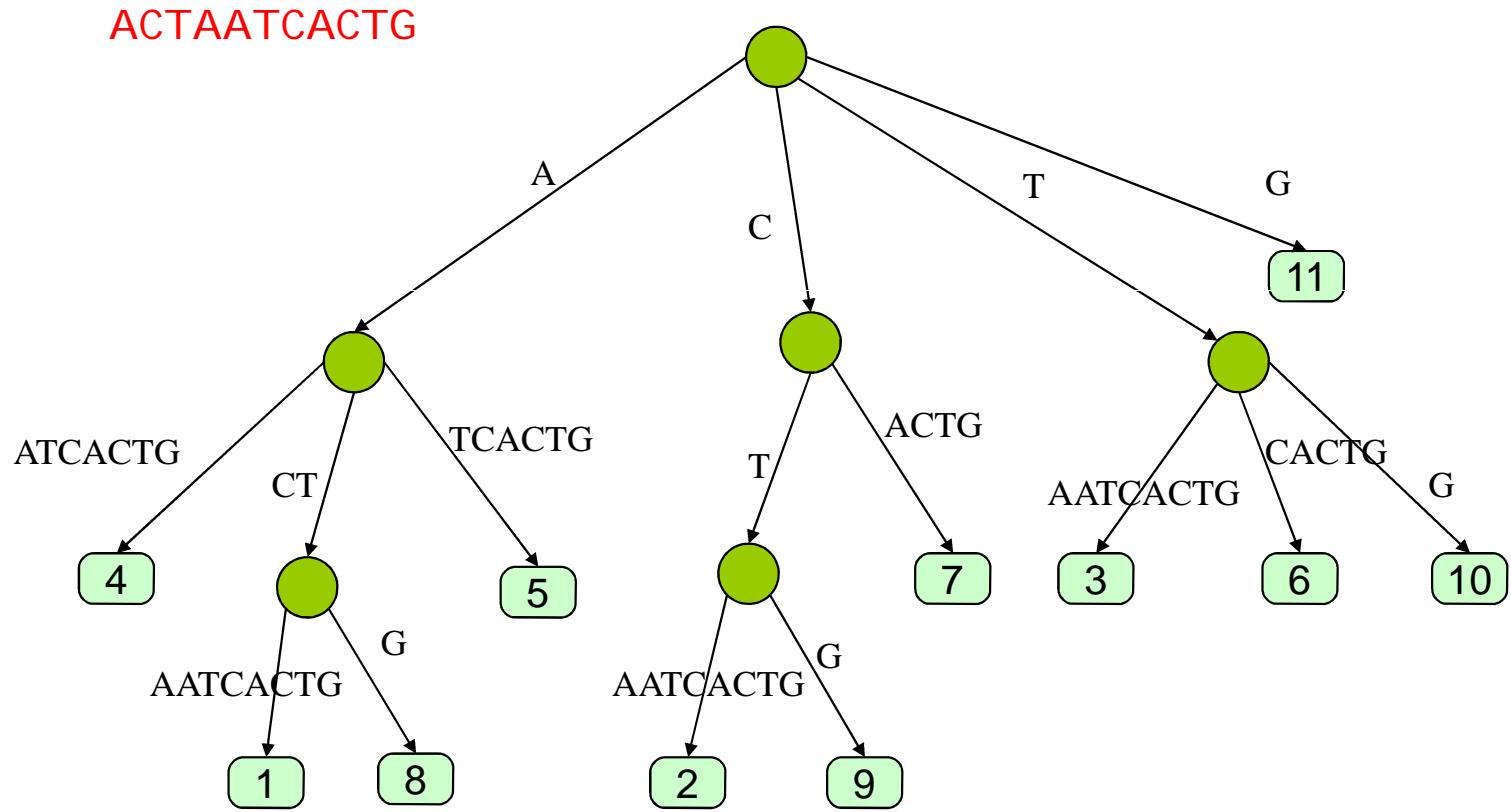




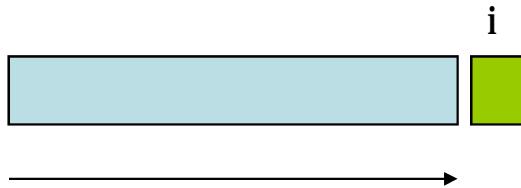




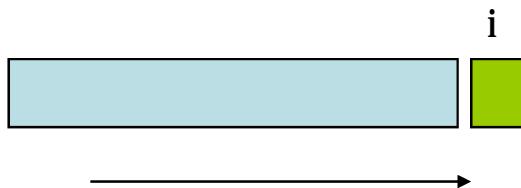




# Observations



At the first extension we must end at a leaf because no longer suffix exists ([rule 1](#))



At the second extension we still most likely to end at a leaf.

We will not end at a leaf only if the second suffix is a prefix of the first



Say at some extension we do not end at a leaf

Then this suffix is a prefix of some other suffix (suffixes)

We will not end at a leaf in subsequent extensions



Is there a way to continue using  $i^{\text{th}}$  character ?

(Is it a prefix of a suffix where the next character is the  $i^{\text{th}}$  character ?)



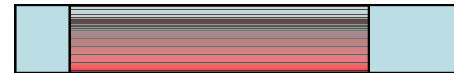
Rule 3



Rule 2



Rule 3



Rule 2

If we apply rule 3 then in all subsequent extensions we will apply rule 3

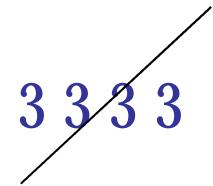
Otherwise we keep applying rule 2 until in some subsequent extensions we will apply rule 3



Rule 3

In terms of the rules that we apply  
a phase looks like:

1 1 1 1 1 1 2 2 2 2 3 3 3 3



We have nothing to do when applying rule 3, so once  
rule 3 happens we can stop

We don't really do anything significant when we apply  
rule 1 (the structure of the tree does not change)

# Representation

- We do not really store a substring with each edge, but rather pointers into the starting position and ending position of the substring in the text
- With this representation we do not really have to do anything when rule 1 applies

# How do phases relate to each other

1 1 1 1 1 1 1 2 2 2 2 3 3 3 3



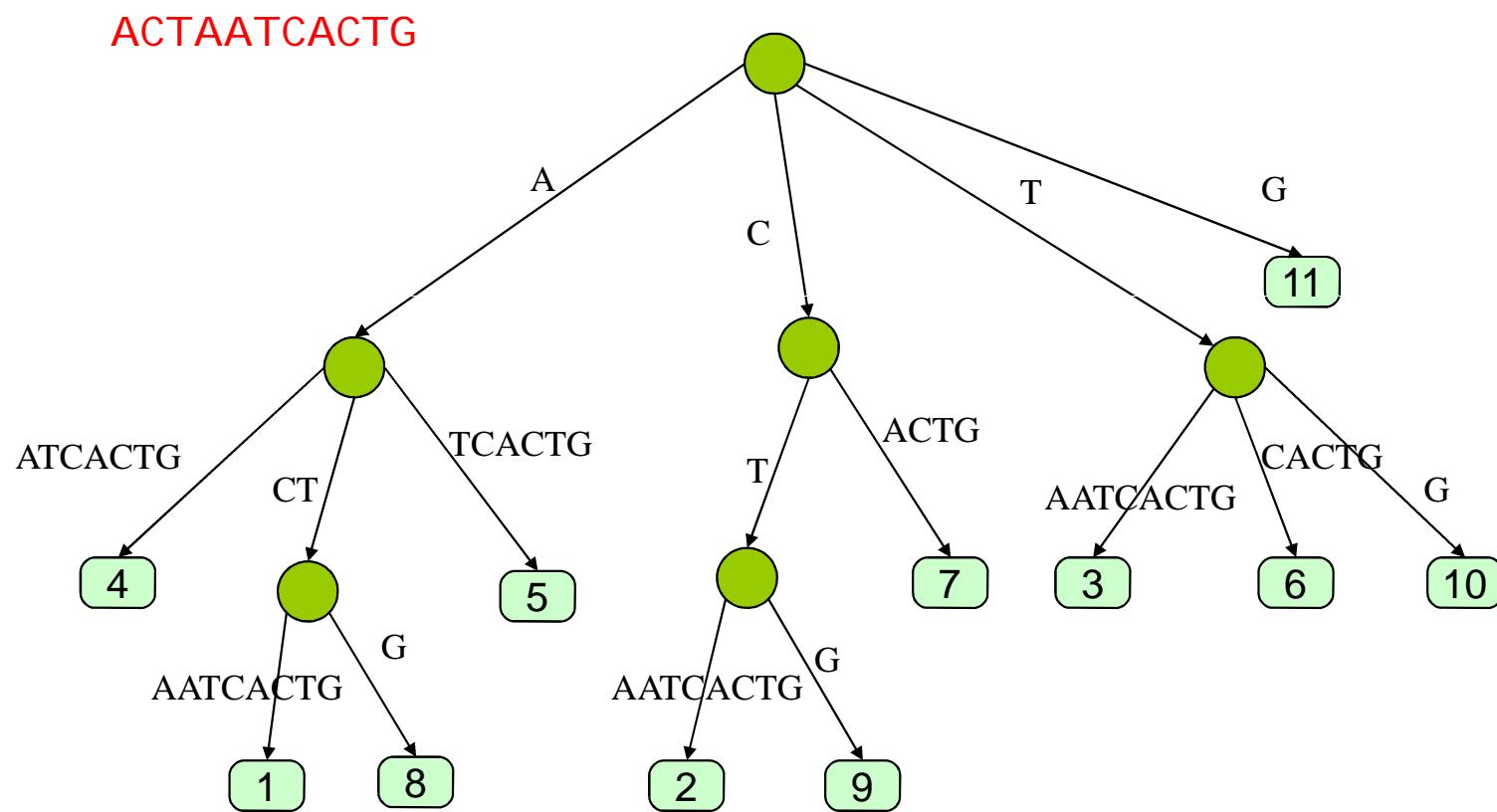
The next phase we must have:

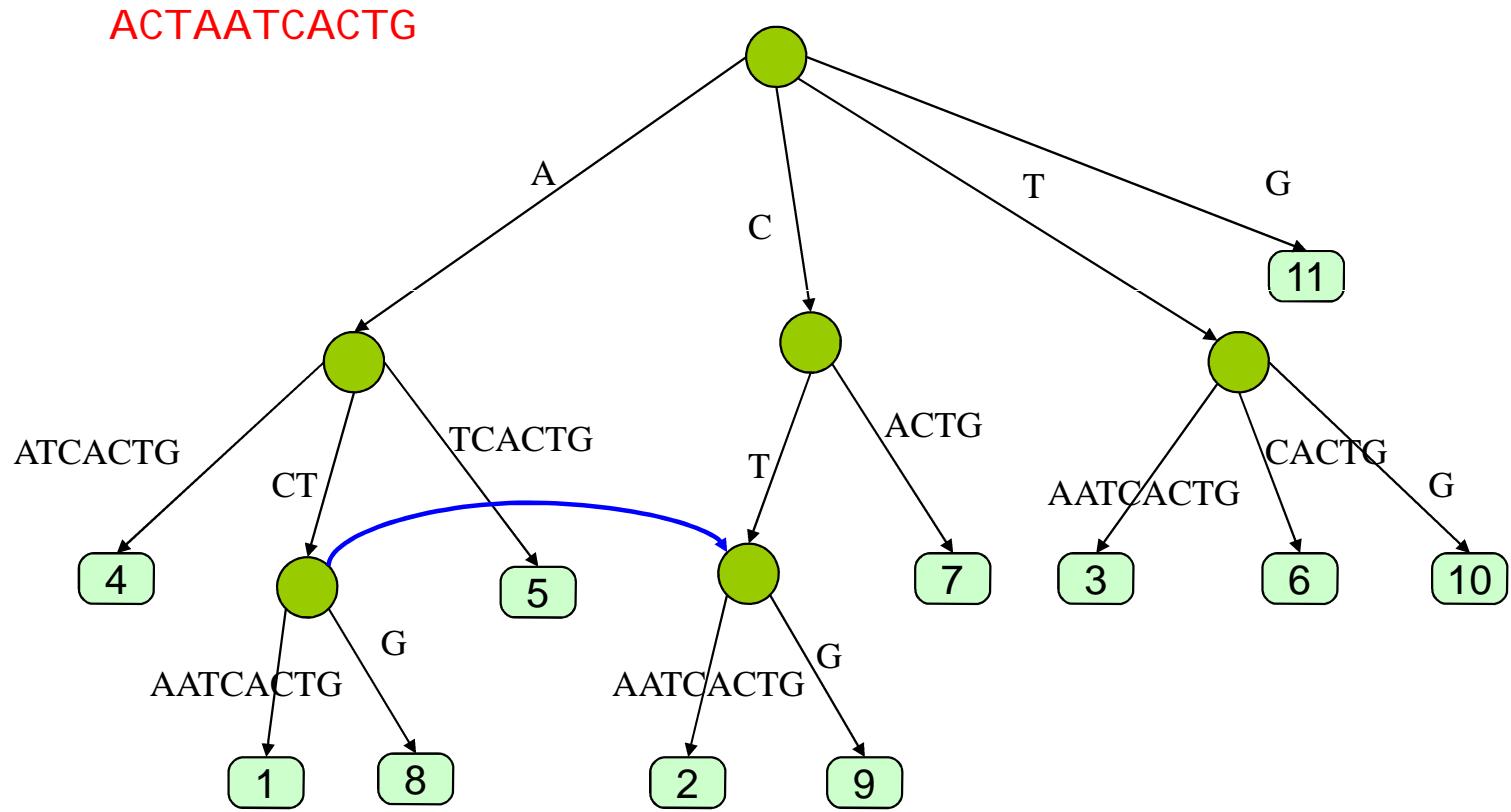
1 1 1 1 1 1 1 1 1 1 2/3

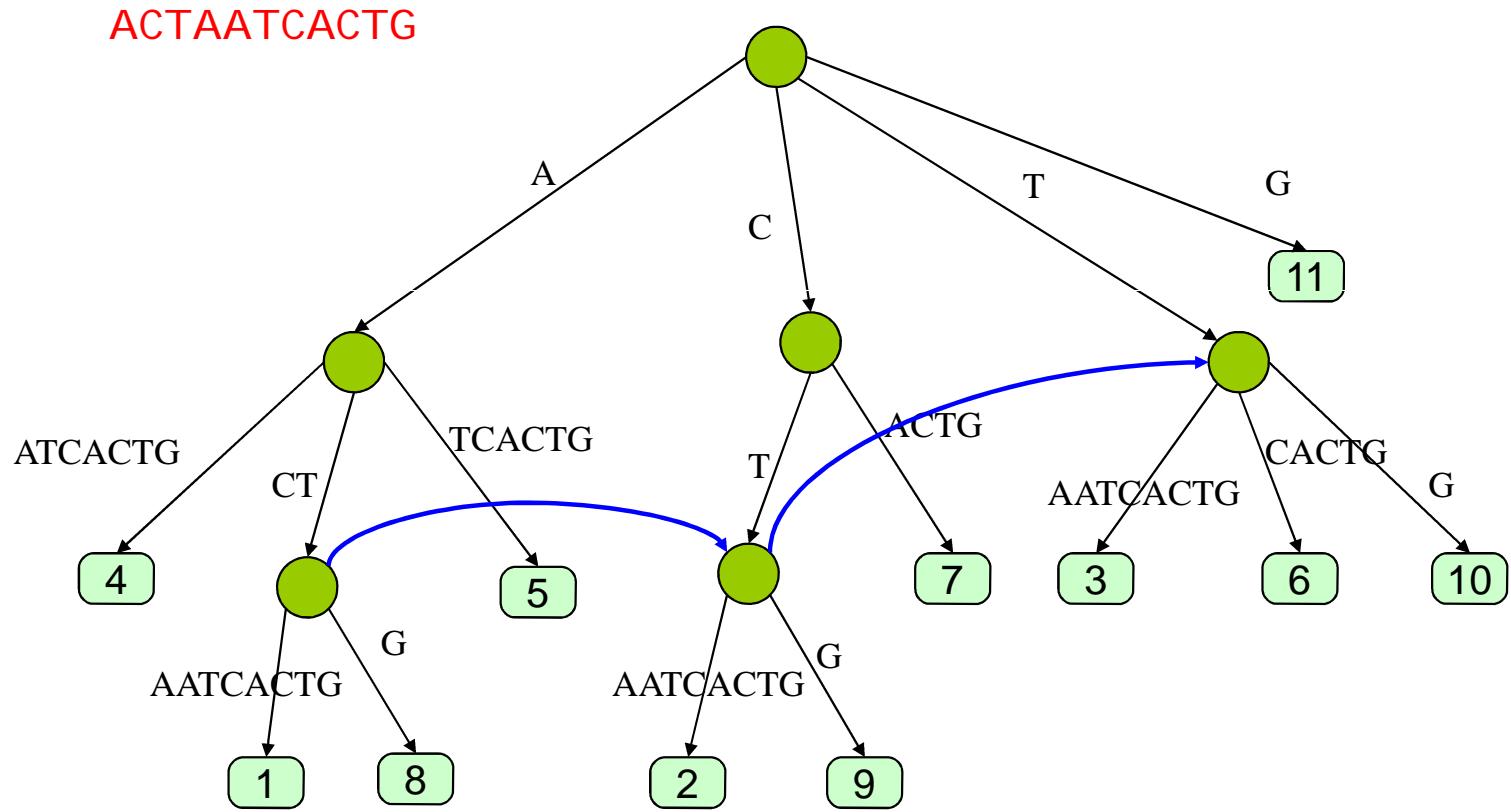


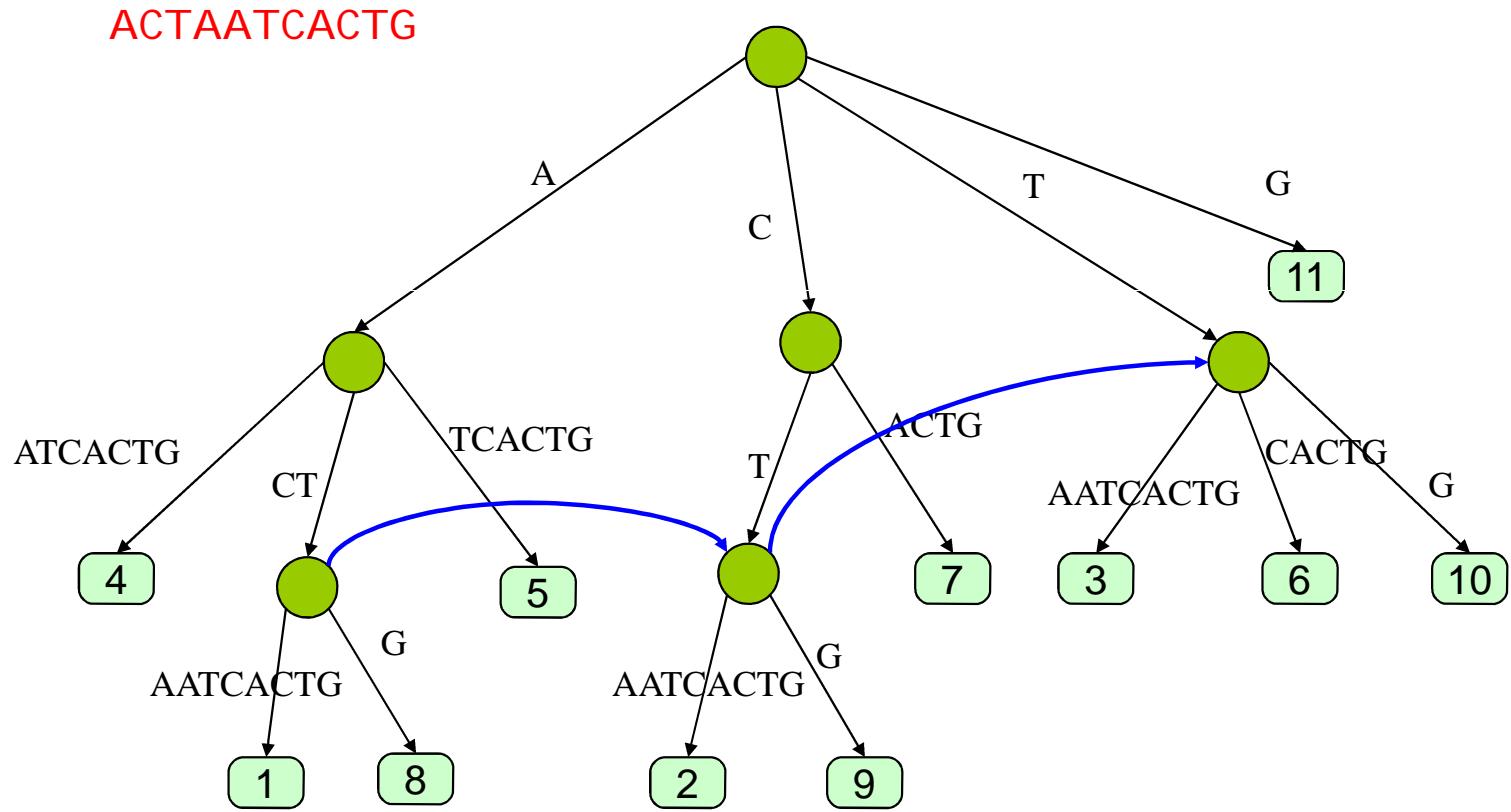
So we start the phase with the extension that was the first where we applied rule 3 in the previous phase

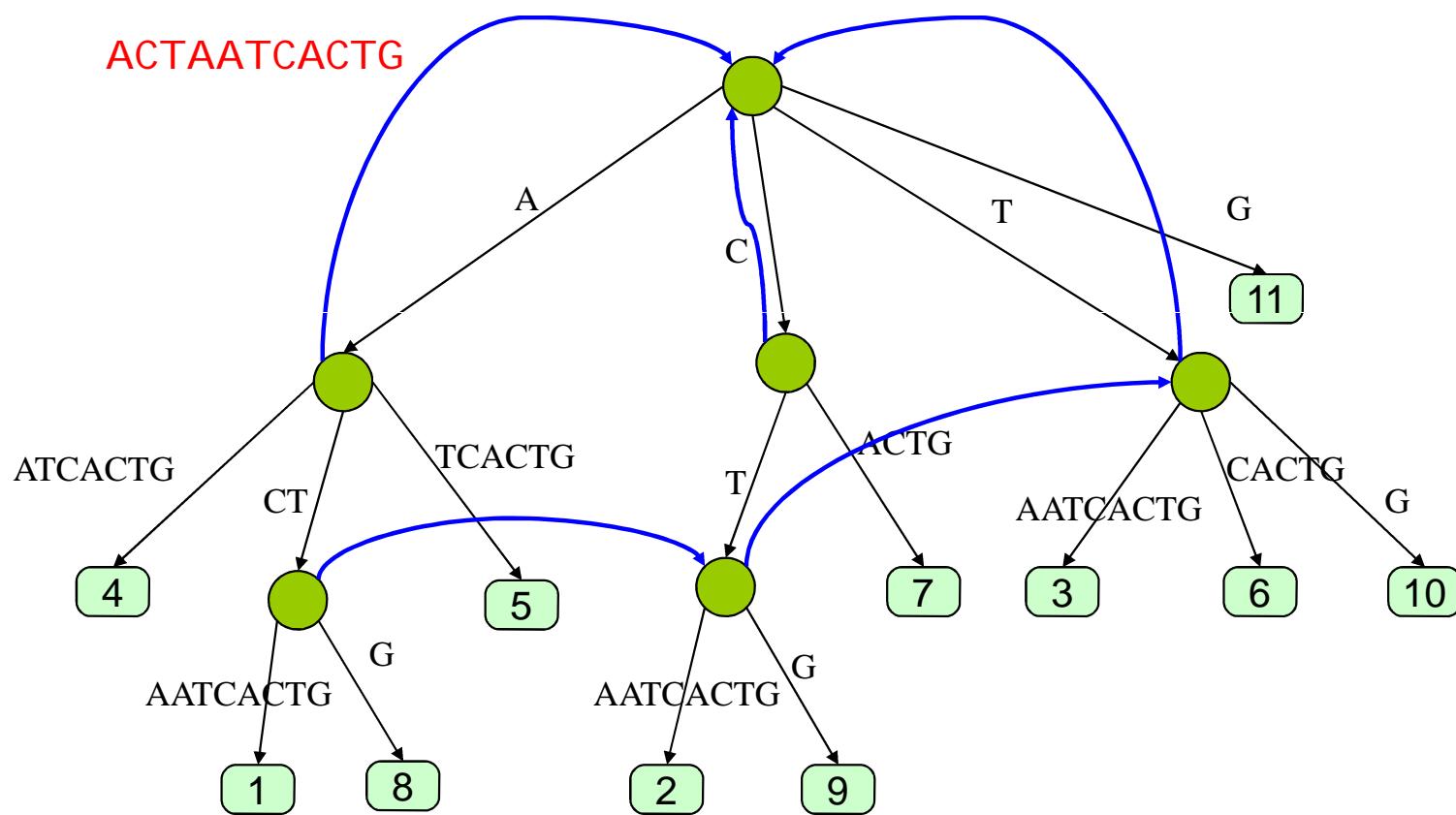
# Suffix Links





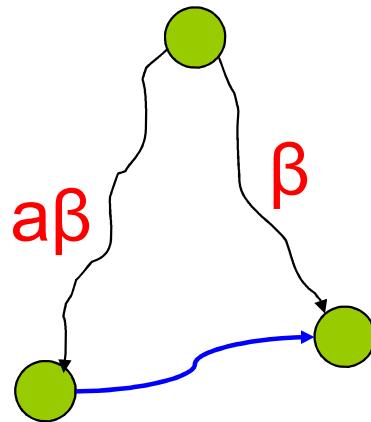






# Suffix Links

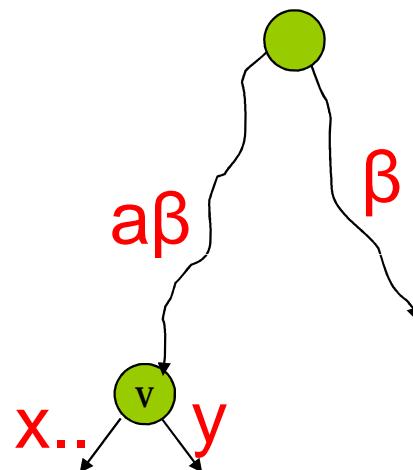
- From an internal node that corresponds to the string  $a\beta$  to the internal node that corresponds to  $\beta$  (if there is such node)



- Is there such a node ?

Suppose we create  $v$  applying rule 2. Then there was a suffix  $a\beta x\dots$  and now we add  $a\beta y$

So there was a suffix  $\beta x\dots$



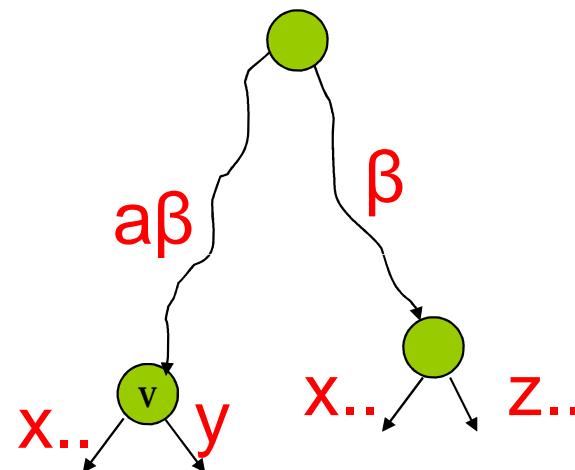
- Is there such a node ?

Suppose we create  $v$  applying rule 2. Then there was a suffix  $a\beta x\dots$  and now we add  $a\beta y$

So there was a suffix  $\beta x\dots$

If there was also a suffix  $\beta z\dots$

Then a node corresponding to  $\beta$  is there



- Is there such a node ?

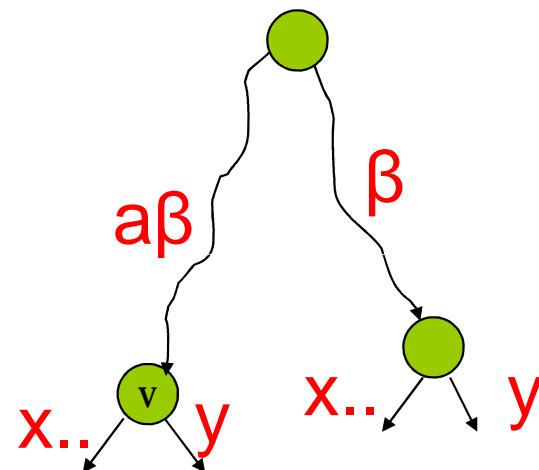
Suppose we create  $v$  applying rule 2. Then there was a suffix  $a\beta x\dots$  and now we add  $a\beta y$

So there was a suffix  $\beta x\dots$

If there was also a suffix  $\beta z\dots$

Then a node corresponding to  $\beta$  is there

Otherwise it will be created in the next extension when we add  $\beta y$



**I nv:** All suffix links are there except (possibly) of the last internal node added

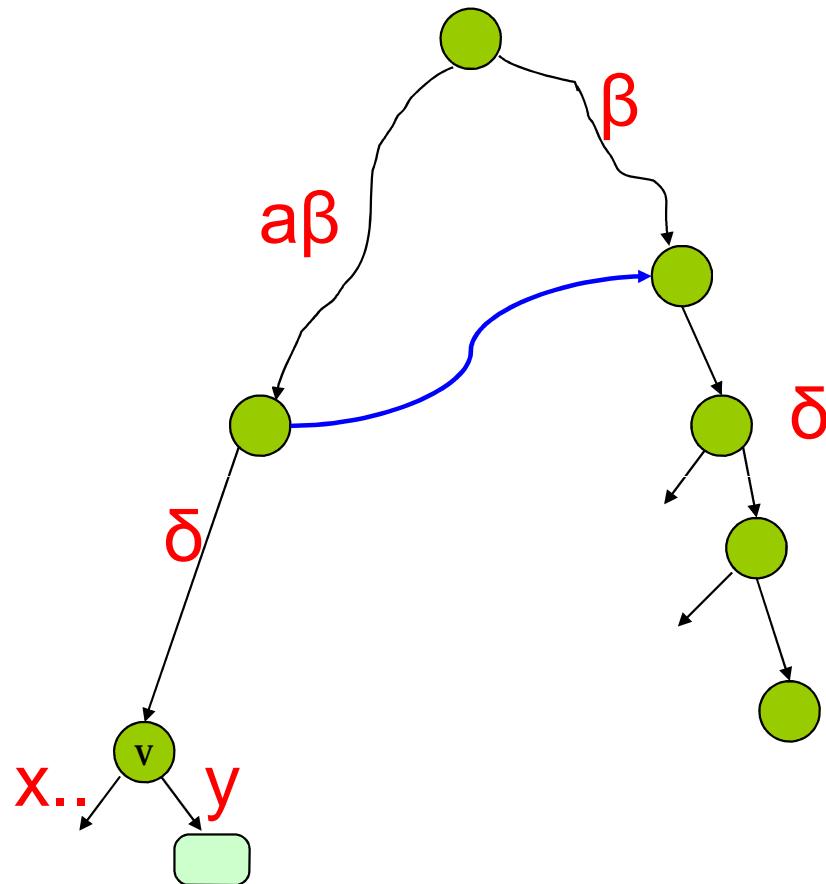
You are at the (internal) node corresponding to the last extension



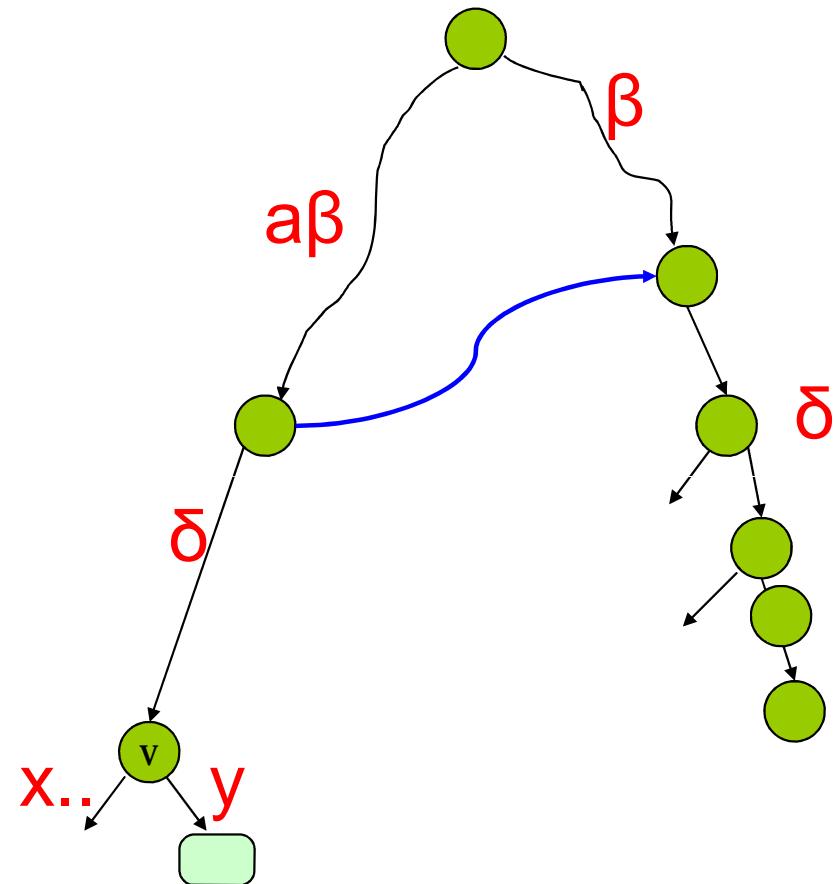
Remember: we apply rule 2

You start a phase at the last internal node of the first extension in which you applied rule 3 in the previous iteration

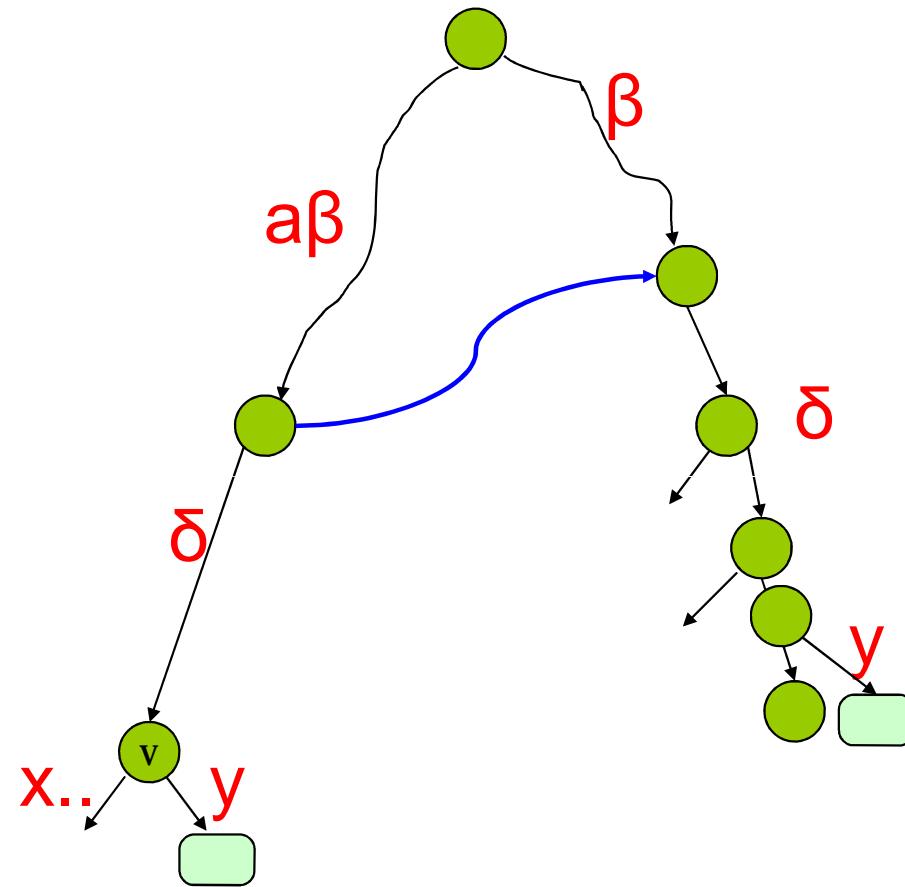
- 1) Go up one node (if needed) to find a suffix link
- 2) Traverse the suffix link
- 3) If you went up in step 1 along an edge that was labeled  $\delta$  then go down consuming a string  $\delta$



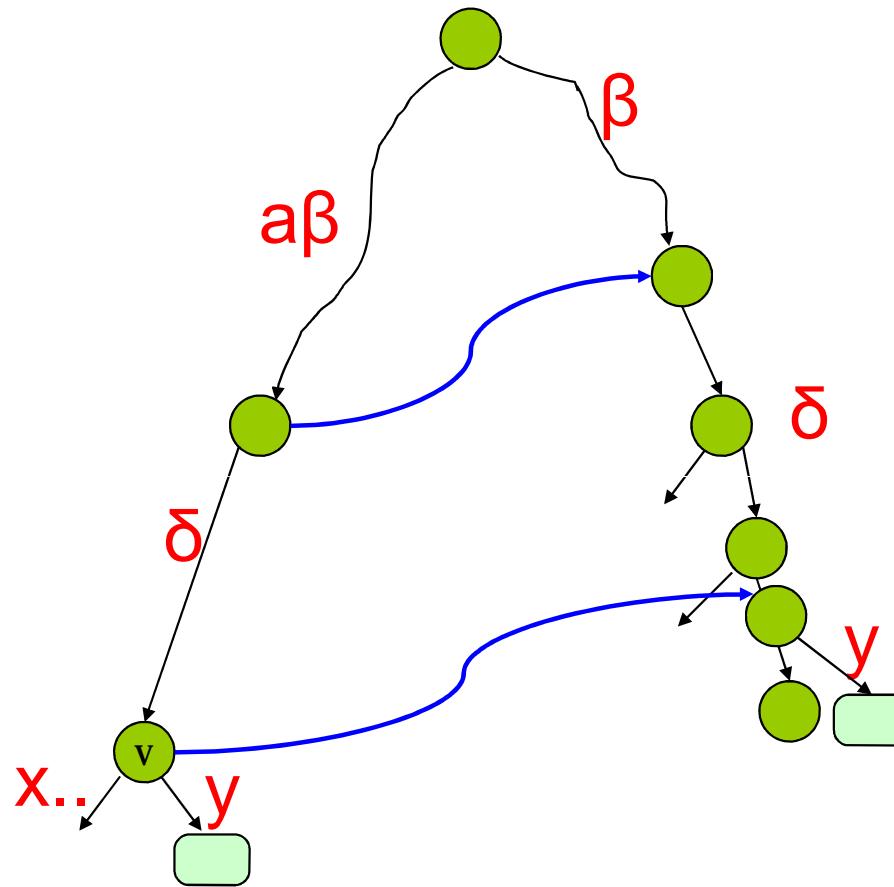
Create the new internal node if necessary



Create the new internal node if necessary



Create the new internal node if necessary, add the suffix



Create the new internal node if necessary, add the suffix and install a suffix link if necessary

# Analysis

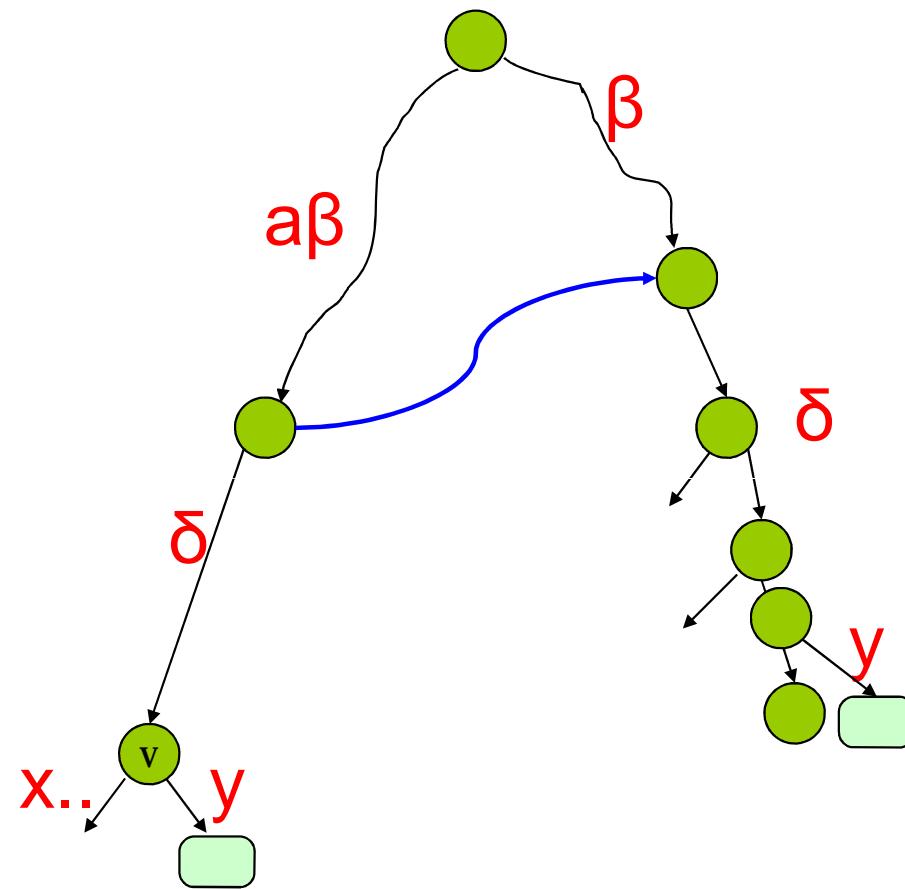
Handling all extensions of rule 1 and all extensions of rule 3 per phase take  $O(1)$  time  $\Rightarrow O(n)$  total

How many times do we carry out rule 2 in all phases ?

$O(n)$

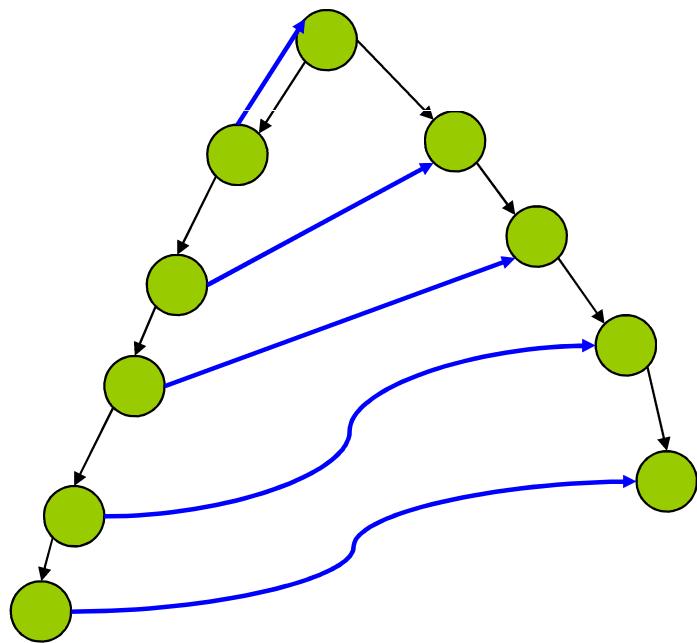
Does each application of rule 2 takes constant time ?

No ! (going up and traversing the suffix link takes constant time, but then we go down possibly on many edges..)



So why is it a linear time algorithm ?

How much can the depth change when we traverse a suffix link ?



It can decrease by  
at most 1

# Punch line

Each time we go up or traverse a suffix link the depth decreases by at most 1

When starting the depth is 0, final depth is at most  $n$

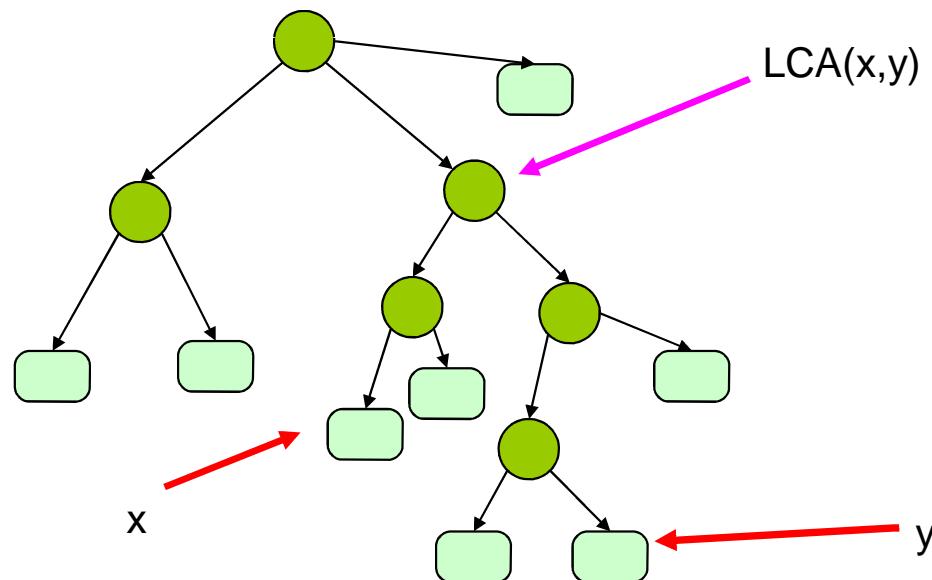
So during all applications of rule 2 together we cannot go down more than  $3n$  times

THM: The running time of Ukkonen's algorithm is  $O(n)$

# Suffix tree Applications

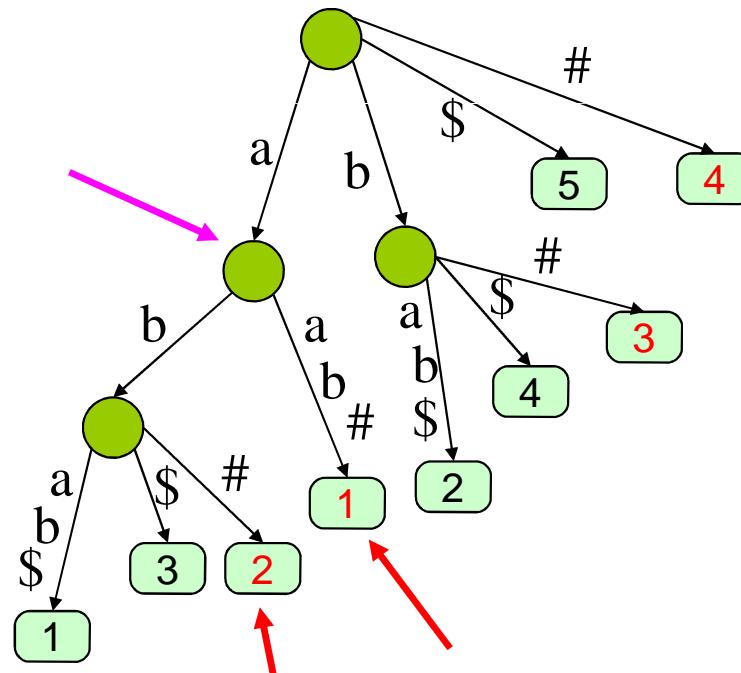
# Lowest common ancestors (LCA)

A lot more can be gained from the suffix tree if we preprocess it so that we can answer LCA queries on it



# Why LCA?

The LCA of two leaves represents the longest common prefix (LCP) of these 2 suffixes

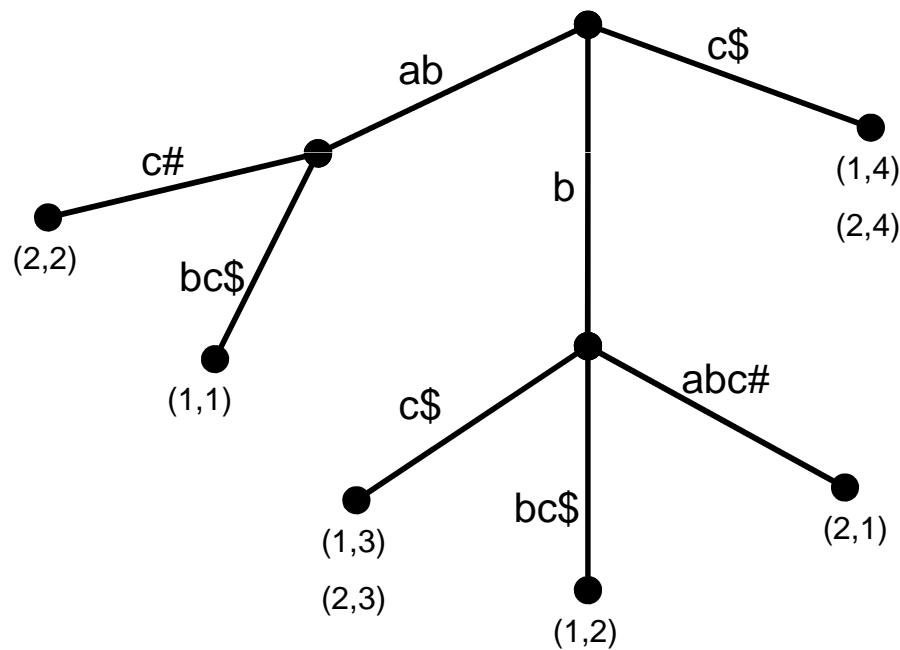


# Generalized Suffix Trees (GST)

A GST is simply a ST for a set of strings, each one ending with a different marker. The leafs have two numbers, one identifying the string and the other identifying the position inside the string.

$S_1 = \mathbf{abbc\$}$

$S_2 = \mathbf{babc\#}$



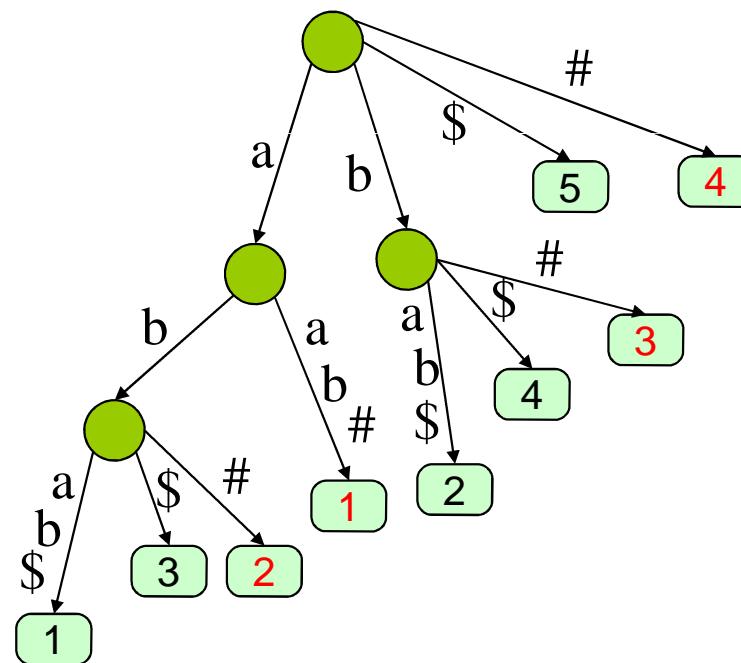
# Exercise: generalized suffix tree

1. Draw the GST for  $s_1=abab$ ,  $s_2=aab$

# GST Solution

The generalized suffix tree of  $s_1=abab$  and  $s_2=aab$  is:

{  
  \$           #  
  b\$          b#  
  ab\$         ab#  
  bab\$        aab#  
  abab\$          
}



# ST applications

- Searching for substrings
- LCS
- Hamming distance
- Longest palindrome

# Longest common substring

Let  $S_1$  and  $S_2$  be two string over the same alphabet. The **Longest Common Substring** problem is to find the longest substring of  $S_1$ , that is also a substring of  $S_2$ .

Knuth in 1970 conjectured that this problem was  $\Theta(n^2)$

Building a generalized suffix tree for  $S_1$  and  $S_2$ , to solve the problem one has to identify the nodes which belong to both suffix trees of  $S_1$  and  $S_2$  and choose the one with greatest **string depth** (length of the path label from the root to itself). All these operations cost  $O(n)$ .

# Longest Common Extension

$\text{LCS}(T,P)$  is solved in linearly using suffix trees.

Find the longest substring of  $T[i..]$  that matches a substring of  $P[j..]$ , for all  $(i, j)$ .

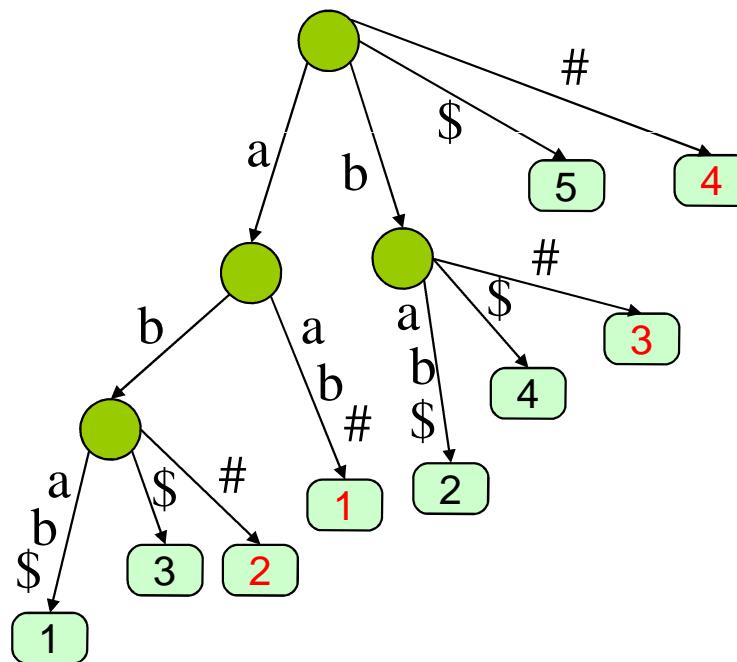
It can be solved in  $O(n+m)$  time. Build a generalized suffix tree for  $T$  and  $P$ .

For every leaf  $i$  of  $T$  and  $j$  of  $P$ , find their LCA ([lowest common ancestor](#)) in the tree (it can be done in constant time after preprocessing the tree).

# Exercise: Find LCS of $s_1$ and $s_2$

using GST of  $s_1=abab$  and  $s_2=aab$  is:

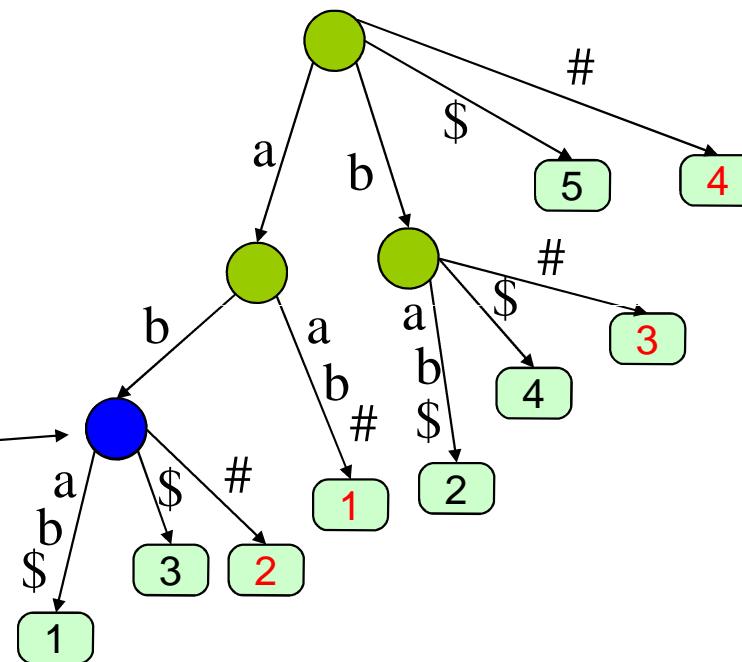
{  
  \$           #  
  b\$          b#  
  ab\$         ab#  
  bab\$        aab#  
  abab\$          
}



## LCS solution

Every node with a leaf descendant from string  $s_1$  and a leaf descendant from string  $s_2$  represents a **maximal common substring** and vice versa.

Find such node with largest “string depth”.



# Finding maximal palindromes

Palindromes e.g: Dad, Radar, Malayalam

Q. To find all maximal palindromes in a string  $s$

Let  $s = cba|aba$

The maximal palindrome with center between  $i-1$  and  $i$  is the LCP of the suffix at position  $i$  of  $s$  and the suffix at position  $m-i+1$  of  $\text{reverse}(s)$ .

# Maximal palindromes algorithm

Algorithm:

- `r = reverse(s).`
- `m = len(s);`
- `build GST(s,r)`
- `for i=1..m`
- `find LCA(s[i...],r[m-i+1...])`

Complexity:  $O(n)$  time to identify all palindromes

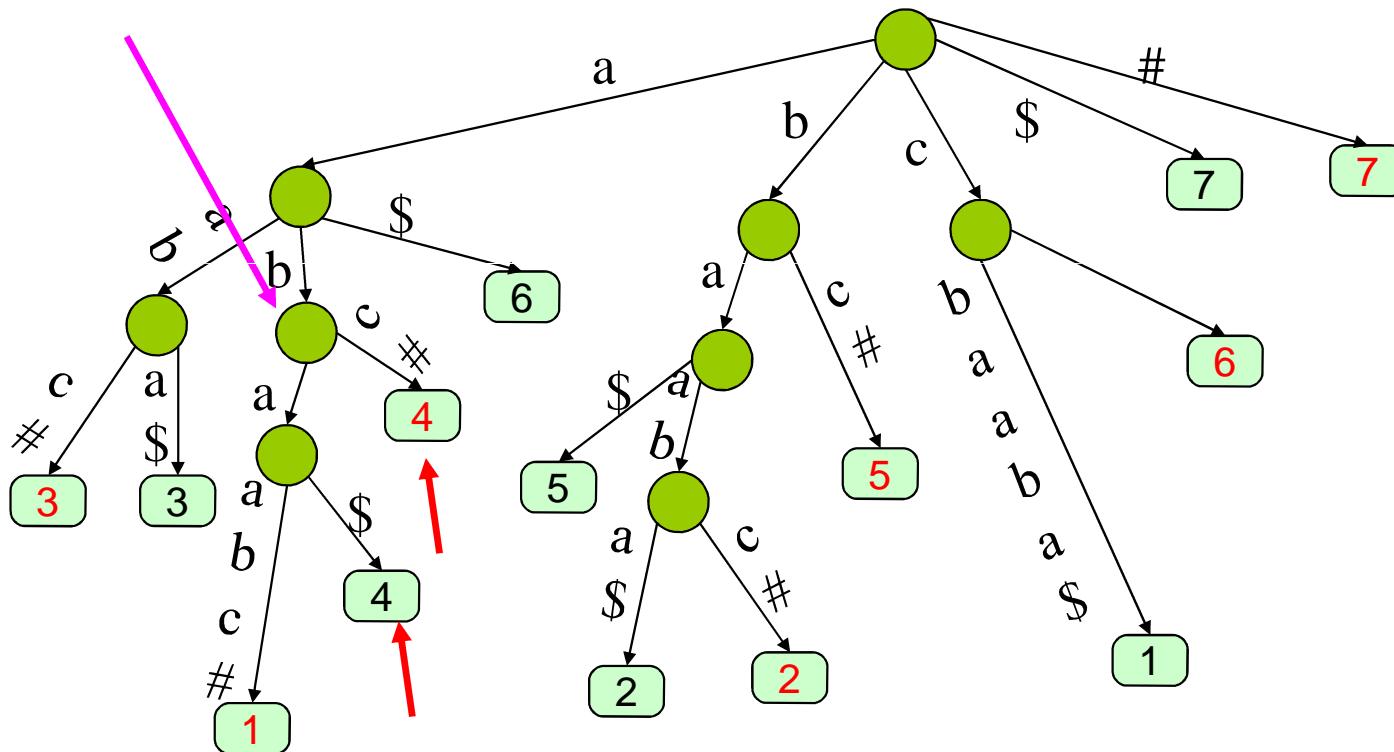
# Exercise

Find the palindrome in cbaaba

1. Build GST( $s=cbaaba\$, r=abaabc\#$ )
2. Find the LCP

# Palindrome example

Let  $s = \text{cbaaba\$}$  then  $\text{reverse}(s) = \text{abaabc\#}$



# Hamming and Edit Distances

**Hamming Distance:** two strings of the same length are aligned and the distance is the number of mismatches between them.

**ab**cdbaabb**c**  
↑↑↑↑↑  
**a**bdcbbbaac

$H = 6$

**Edit Distance:** it is the minimum number of insertions, deletions and substitutions needed to transform a string into another.

**ab**cdbaabb**c**

**ab**cdbaabb**c**

$E = 3$

**c**bcdbaabc

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

# The k-mismatches problem

We have a text  $T$  and a pattern  $P$ , and we want to find occurrences of  $P$  in  $T$ , allowing a maximum of  $k$  mismatches, i.e. we want to find all the substring  $T'$  of  $T$  such that  $H(P, T') \leq k$ .

We can use suffix trees, but they do not perform well anymore: the algorithm scans all the paths to leafs, keeping track of errors, and abandons the path if this number becomes greater than  $k$ .

The algorithm is made faster by using the LCS. For every suffix of  $T$ , the pieces of agreement between the suffix and  $P$  are matched together until  $P$  is exhausted or the errors overcome  $k$ . Every piece is found in constant time. The complexity of the resulting algorithm is  $O(k|T|)$ .

aaac|caabaaaaa....  
|||a|b|aab

An occurrence is found  
in position 2 of  $T$ , with  
one error.

# Inexact Matching

In biology, inexact matching is very important:

- Similarity in DNA sequences implies often that they have the same biological function (viceversa is not true);
- Mutations and error transcription make exact comparison not very useful.

There are a lot of algorithms that deal with inexact matching (with respect to edit distance), and they are mainly based on dynamic programming or on automata. Suffix trees are used as a secondary tools in some of them, because their structure is inadapt to deal with insertions and deletions, and even with substitutions.

The main efforts are spend in speeding up the average behaviour of algorithms, and this is justified because of the fact that random sequences often fall in these cases (and DNA sequences have an high degree of randomness).

# Suffix Arrays

# Suffix array

- We loose some of the functionality but we save space.

Let  $s = abab$

Sort the suffixes lexicographically:

$ab, abab, b, bab$

The suffix array gives the indices of the suffixes in sorted order

3	1	4	2
---	---	---	---

# How do we build it ?

- Build a suffix tree
- Traverse the tree in DFS, lexicographically picking edges outgoing from each node and fill the suffix array.
- $O(n)$  time

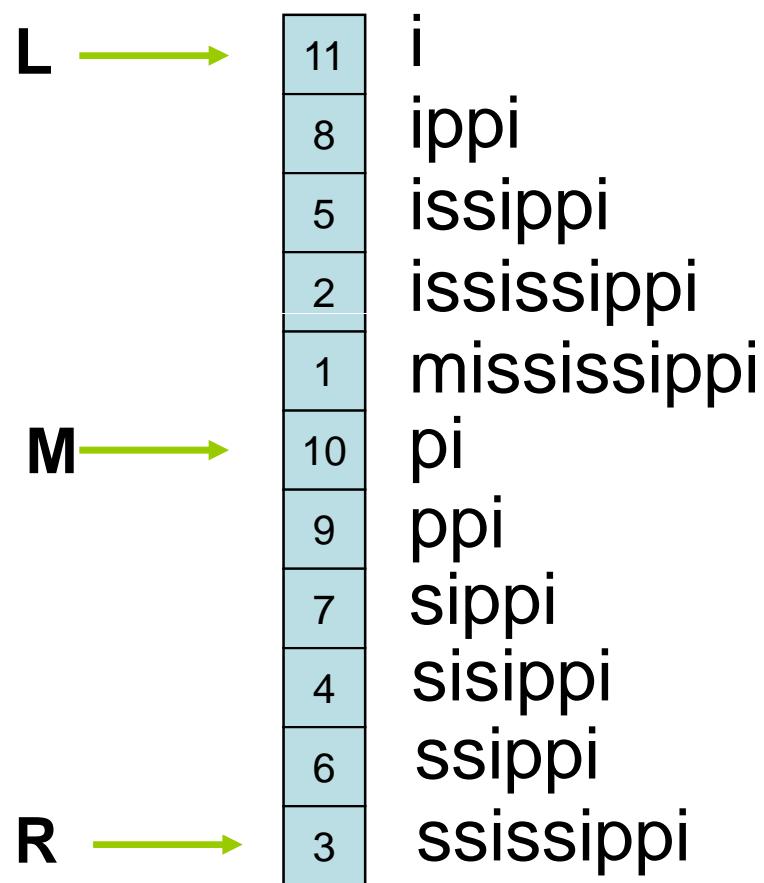
## How do we search for a pattern ?

- If  $P$  occurs in  $T$  then all its occurrences are consecutive in the suffix array.
- Do a binary search on the suffix array
- Takes  $O(m \log n)$  time

# Example

Let  $S = \text{mississippi}$

Let  $P = \text{issa}$

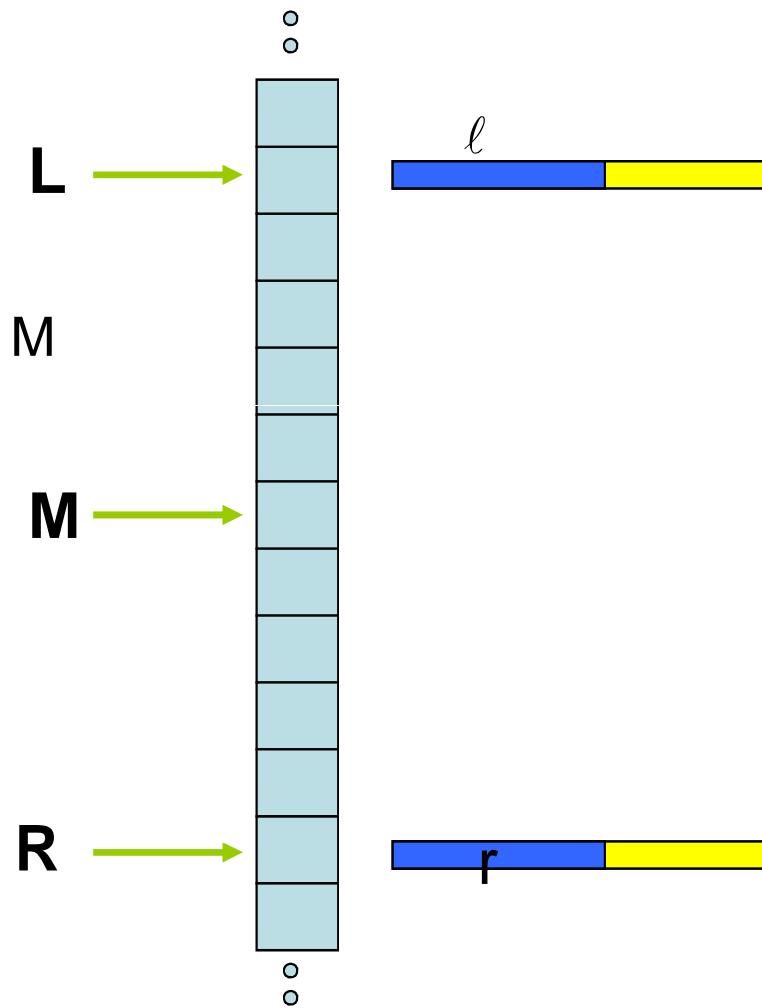


# How do we accelerate the search ?

Maintain  $\ell = \text{LCP}(P, L)$

Maintain  $r = \text{LCP}(P, R)$

If  $\ell = r$  then start comparing M  
to P at  $\ell + 1$



# How do we accelerate the search ?

If  $\ell > r$  then

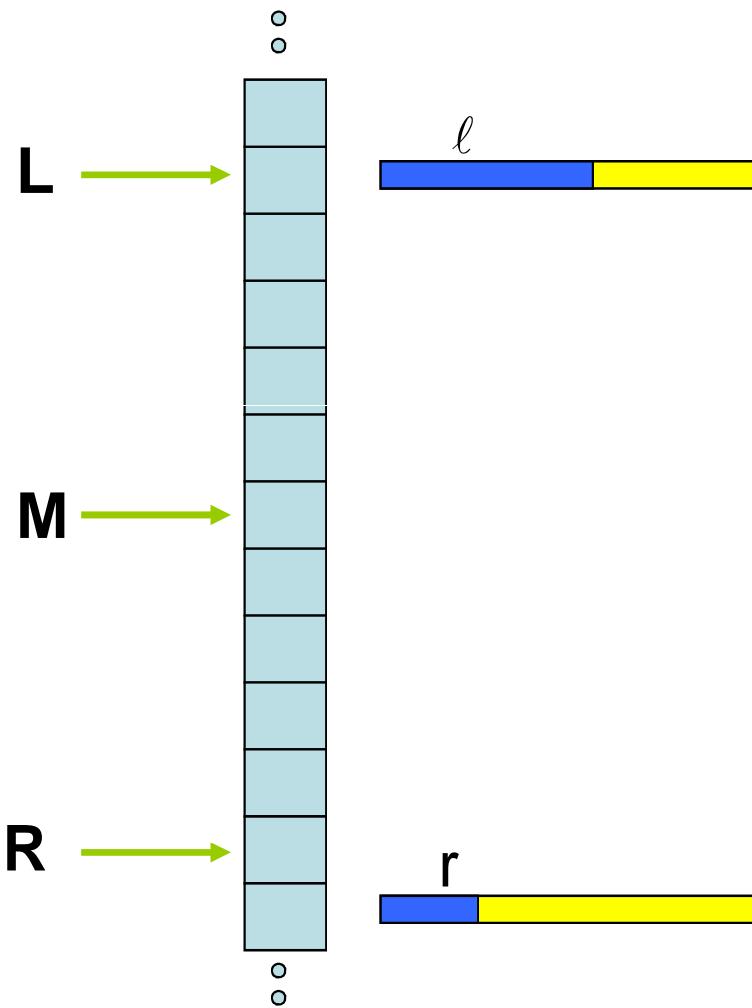
Suppose we know  $LCP(L, M)$

If  $LCP(L, M) < \ell$  we go left

If  $LCP(L, M) > \ell$  we go right

If  $LCP(L, M) = \ell$  we start

comparing at  $\ell + 1$



# Analysis of the acceleration

If we do more than a single comparison in an iteration then  $\max(\ell, r)$  grows by 1 for each comparison:  $O(\log n + m)$  time

# Finite Automata

- A *finite automaton*  $M$  is a 5-tuple  $(Q, q_0, A, \Sigma, \delta)$ , where
  - $Q$  is a finite set of *states*
  - $q_0 \in Q$  is the *start state*
  - $A \subseteq Q$  is a set of *accepting states*
  - $\Sigma$  is a finite *input alphabet*
  - $\delta$  is the *transition function* that gives the next state for a given current state and input

# How a Finite Automaton Works

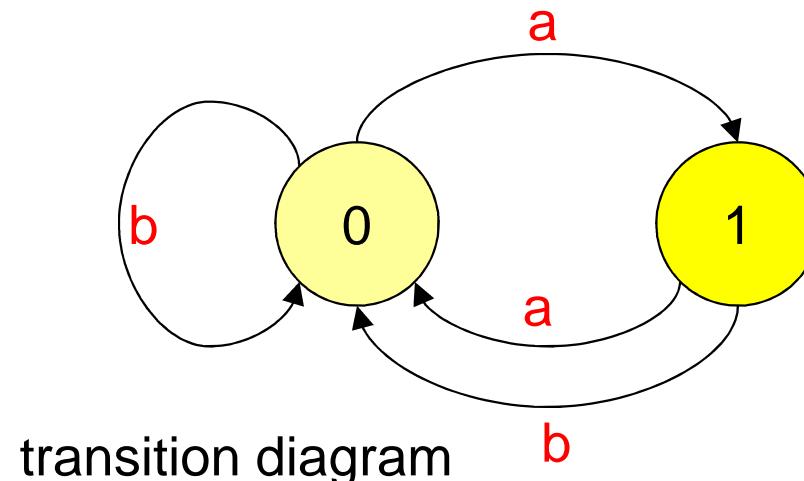
- The finite automaton  $M$  begins in state  $q_0$
- Reads characters from  $\Sigma$  one at a time
- If  $M$  is in state  $q$  and reads input character  $a$ ,  $M$  moves to state  $\delta(q,a)$
- If its current state  $q$  is in  $A$ ,  $M$  is said to have *accepted* the string read so far
- An input string that is not accepted is said to be *rejected*

# Example

- $Q = \{0,1\}$ ,  $q_0 = 0$ ,  $A=\{1\}$ ,  $\Sigma = \{a, b\}$
- $\delta(q,a)$  shown in the transition table/diagram
- This accepts strings that end in an odd number of a's; e.g., abbaaa is accepted, aa is rejected

state	input	
	a	b
0	1	0
1	0	0

transition table



# Finite automata string matcher

- Construct deterministic FA for the pattern P before starting match with T.
- Processing time takes  $\Theta(n)$ .
- T matches P, if DFA(P) accepts T

# String-Matching Automata

- Given the pattern  $P[1..m]$ , build a finite automaton  $M$ 
  - The state set is  $Q=\{0, 1, 2, \dots, m\}$
  - The start state is 0
  - The only accepting state is  $m$
- Time to build  $M$  can be large if  $\Sigma$  is large

# String-Matching Automata

...contd

- Scan the text string  $T[1..n]$  to find all occurrences of the pattern  $P[1..m]$
- String matching is efficient:  $\Theta(n)$ 
  - Each character is examined exactly once
  - Constant time for each character
- But ...time to compute  $\delta$  is  $O(m |\Sigma|)$ 
  - $\delta$  Has  $O(m |\Sigma|)$  entries

# DFA matcher

**Input:** Text string  $T[1..n]$ ,  $\delta$  and  $m$

**Result:** All valid shifts displayed

**FINITE-AUTOMATON-MATCHER** ( $T, m, \delta$ )

$n \leftarrow \text{length}[T]$

$q \leftarrow 0$

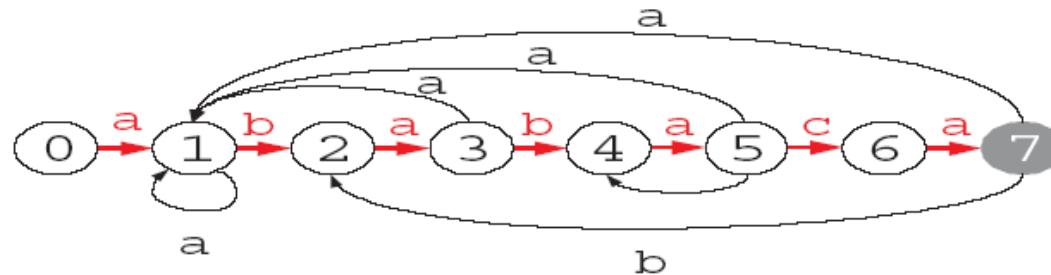
**for**  $i \leftarrow 1$  **to**  $n$

$q \leftarrow \delta(q, T[i])$

**if**  $q = m$

print “pattern occurs with shift”  $i-m$

# Example



state	a	b	c	P
0	1	0	0	a
1	1	2	0	b
2	3	0	0	a
3	1	4	0	b
4	5	0	0	a
5	1	4	6	c
6	7	0	0	a
7	1	2	0	

i	—	1	2	3	4	5	6	7	8	9	10	11
$T[i]$	—	a	b	a	b	a	b	a	c	a	b	a
$\phi(T_i)$	0	1	2	3	4	5	4	5	6	7	2	3

# The KMP Algorithm

- The Knuth-Morris-Pratt (KMP) algorithm looks for the pattern in the text in a *left-to-right* order (like the brute force algorithm).
- It differs from the brute-force algorithm by keeping track of information gained from previous comparisons.
- But it shifts the pattern more intelligently than the brute force algorithm.

# Knuth-Morris-Pratt (KMP) Method

- Avoids computing  $\delta$  (transition function) of DFA matcher
- Instead computes a *prefix function*  $\pi$  in  $O(m)$  time.  $\pi$  has only  $m$  entries
- Prefix function stores info about how the pattern matches against shifts of itself (to avoid testing shifts that will never match).

# How much to skip?

- If a mismatch occurs between the text and pattern  $P$  at  $P[j]$ , what is the *most* we can shift the pattern to avoid wasteful comparisons?
- *Answer*: the largest prefix of  $P[0 .. j-1]$  that is a suffix of  $P[1 .. j-1]$

# KMP failure function

- A **failure function (f)** is computed that indicates how much of the last comparison can be reused if it fails.
- $f$  is defined to be the longest prefix of the pattern  $P[0,..,j]$  that is also a suffix of  $P[1,..,j]$   
**Note:** **not** a suffix of  $P[0,..,j]$

# KMP Advantages

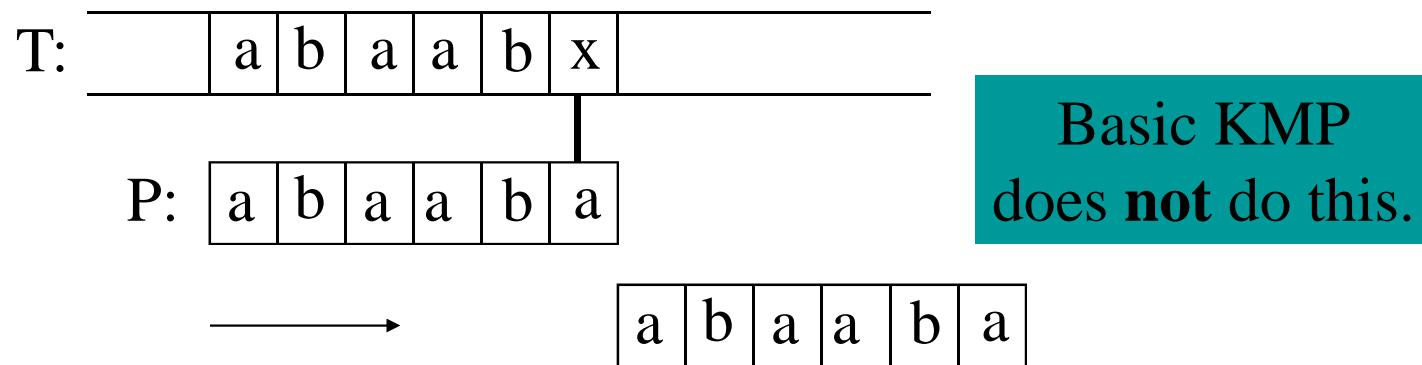
- KMP runs in optimal time:  $O(m+n)$ 
  - fast
- The algorithm never needs to move backwards in the input text,  $T$ 
  - this makes the algorithm good for processing very large files that are read in from external devices or through a network stream

# KMP Disadvantages

- KMP doesn't work so well as the size of the alphabet increases
  - more chance of a mismatch (more possible mismatches)
  - mismatches tend to occur early in the pattern, but KMP is faster when the mismatches occur later

# KMP Extensions

- The basic algorithm doesn't take into account the letter in the text that caused the mismatch.



# KMP failure function

- KMP failure function, for  $P = ababac$

j	0	1	2	3	4	5
$P[j]$	a	b	a	b	a	c
$f(j)$	0	0	1	2	3	0

- This shows how much of the beginning of the string matches up to the portion immediately preceding a failed comparison.
  - if the comparison fails at (4), we know the a,b in positions 2,3 is identical to positions 0,1

## Knuth-Morris-Pratt Algorithm

- Achieves  $\Theta(n + m)$  by avoiding precomputation of  $\delta$ .
- Instead, we precompute  $\pi[1..m]$  in  $O(m)$  time.
- As  $T$  is scanned,  $\pi[1..m]$  is used to deduce information given by  $\delta$  in FA algorithm.

# The KMP Algorithm (contd.)

- Time Complexity Analysis
- define  $k = i - j$
- In every iteration through the while loop, one of three things happens.
  - 1) if  $T[i] = P[j]$ , then  $i$  increases by 1, as does  $j$   $k$  remains the same.
  - 2) if  $T[i] \neq P[j]$  and  $j > 0$ , then  $i$  does not change and  $k$  increases by at least 1, since  $k$  changes from  $i - j$  to  $i - f(j-1)$
  - 3) if  $T[i] \neq P[j]$  and  $j = 0$ , then  $i$  increases by 1 and  $k$  increases by 1 since  $j$  remains the same.

# The KMP Algorithm (contd.)

- Thus, each time through the loop, either  $i$  or  $k$  increases by at least 1, so the greatest possible number of loops is  $2n$
- This of course assumes that  $f$  has already been computed.
- However,  $f$  is computed in much the same manner as KMPMatch so the time complexity argument is analogous. KMPFailureFunction is  $O(m)$
- Total Time Complexity:  $O(n + m)$

# KMP Matcher

Algorithm KMPMatch( $T, P$ )

**Input:** Strings  $T$  (text) with  $n$  characters and  $P$  (pattern) with  $m$  characters.

**Output:** Starting index of the first substring of  $T$  matching  $P$ , or an indication that  $P$  is not

a  
substring of  $T$ .

# Algorithm

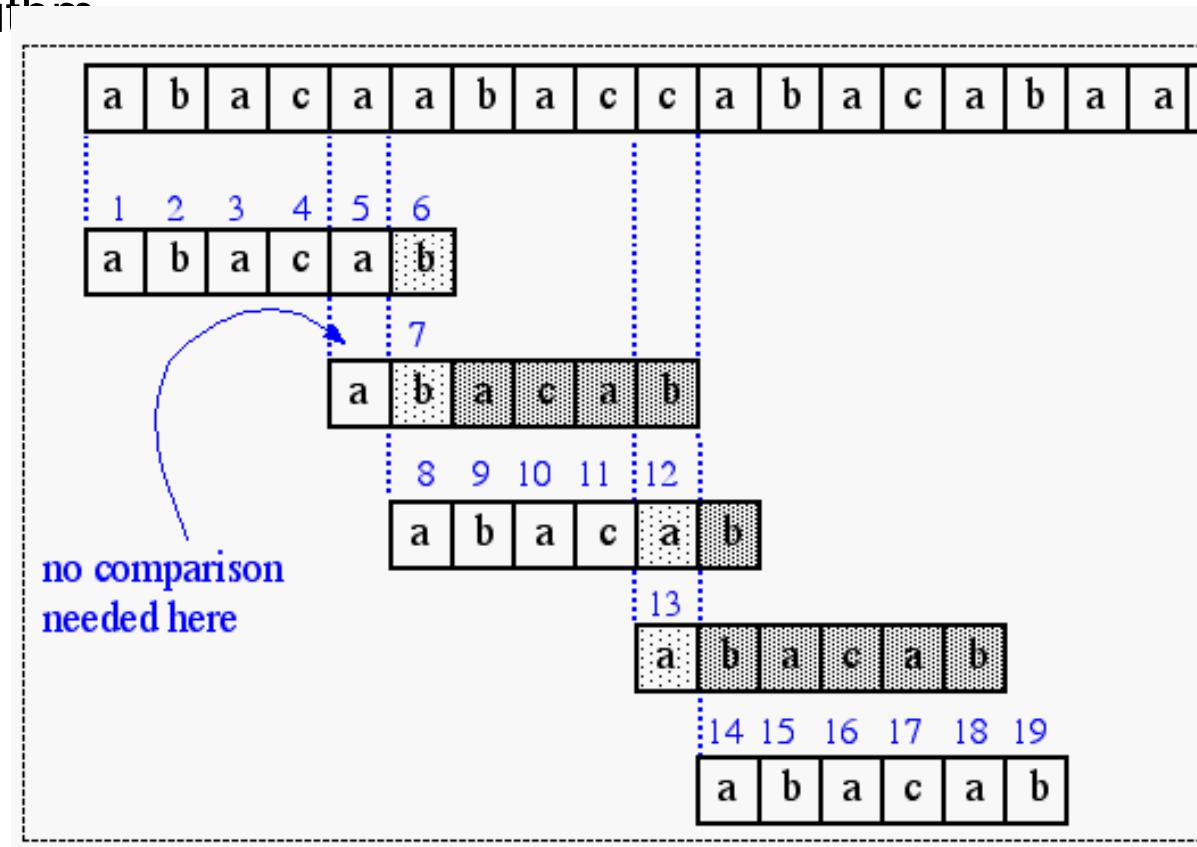
```
f ← KMPFailureFunction(P) {build failure function}
i ← 0
j ← 0
while i < n do
    if P[j] = T[i] then
        if j = m - 1 then
            return i - m - 1 {a match}
        i ← i + 1
        j ← j + 1
    else if j > 0 then {no match, but we have advanced}
        j ← f(j-1) {j indexes just after matching prefix in
P}
    else
        i ← i + 1
return "There is no substring of T matching P"
```

# Create failure function

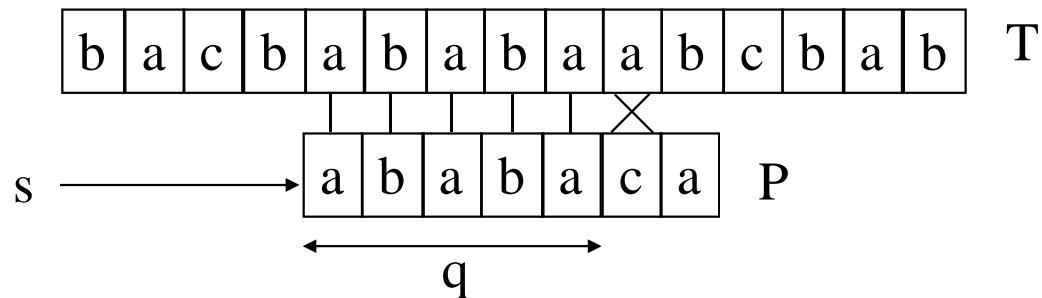
```
f ← KMPFailureFunction(P) {build failure function}
    i ← 0, j ← 0
    while i ≤ m-1 do
        if P[j] = T[i] then
            if j = m - 1 then
                { we have matched j+1 characters}
                f(i) ← j + 1
                i ← i + 1
                j ← j + 1
            else if j > 0 then
                j ← f(j-1) {j indexes just after matching prefix in
P}
            else {there is no match}
                f(i) ← 0
                i ← i + 1
```

# The KMP Algorithm (contd.)

- A graphical representation of the KMP string searching algorithm



# Motivating Example

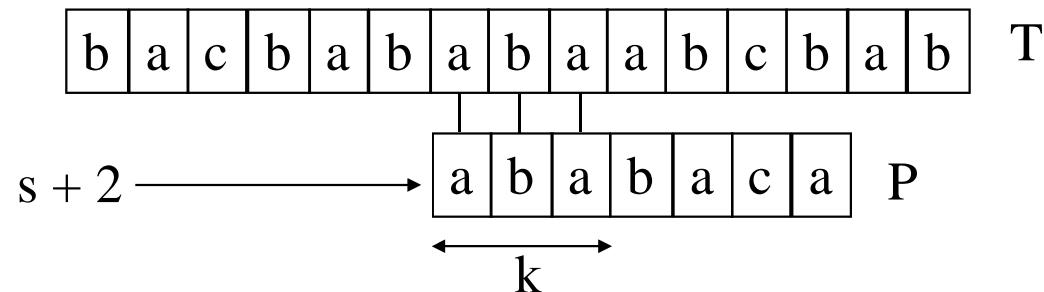


Shift  $s$  is discovered to be invalid because of mismatch of 6<sup>th</sup> character of  $P$ .

By definition of  $P$ , we also know  $s + 1$  is an invalid shift

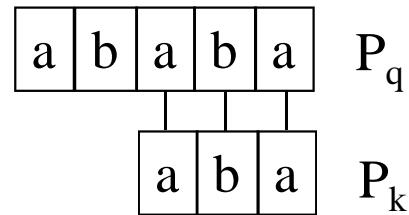
However,  $s + 2$  may be a valid shift.

# Motivating Example



The shift  $s + 2$ . Note that the first 3 characters of  $T$  starting at  $s + 2$  don't have to be checked again -- we already know what they are.

# Motivating Example



The longest prefix of  $P$  that is also a proper suffix of  $P_5$  is  $P_3$ .  
We will define  $\pi[5] = 3$ .

In general, if  $q$  characters have matched successfully at shift  $s$ , the next potentially valid shift is  $s' = s + (q - \pi[q])$ .

# The Prefix Function

$\pi$  is called the **prefix function** for  $P$ .

$\pi: \{1, 2, \dots, m\} \rightarrow \{0, 1, \dots, m-1\}$

$\pi[q] =$  length of the longest prefix  
of  $P$  that is a proper suffix  
of  $P_q$ , i.e.,

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

```
Compute- $\pi(P)$ 
1   $m := \text{length}[P];$ 
2   $\pi[1] := 0;$ 
3   $k := 0;$ 
4  for  $q := 2$  to  $m$  do
5      while  $k > 0$  and  $P[k+1] \neq P[q]$  do
6           $k := \pi[k]$ 
7          od;
8          if  $P[k+1] = P[q]$  then
9               $k := k + 1$ 
10             fi;
11              $\pi[q] := k$ 
12         od;
13     return  $\pi$ 
```

# Example

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

Same as our  
FA example

$$P_7 = a b a b a c a$$

|

$a = P_1$

$$P_4 = a b a b$$

| |

$a b = P_2$

$$P_1 = a$$

|

$\epsilon = P_0$

$$P_6 = a b a b a c$$

|

$\epsilon = P_0$

$$P_3 = a b a$$

|

$a = P_1$

$$P_5 = a b a b a$$

| | |

$a b a = P_3$

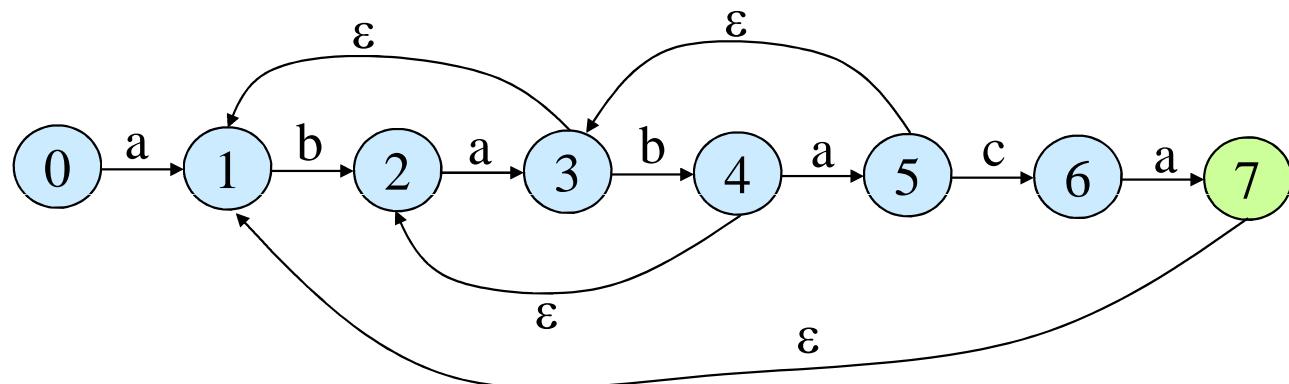
$$P_2 = a b$$

|

$\epsilon = P_0$

## Another Explanation ...

Essentially KMP is computing a **FA with epsilon moves**. The “spine” of the FA is implicit and doesn’t have to be computed -- it’s just the pattern P.  $\pi$  gives the  $\epsilon$  transitions. There are  $O(m)$  such transitions.



Recall from Comp 455 that a FA with epsilon moves is conceptually able to be in several states at the same time (“in parallel”). That’s what’s happening here -- we’re exploring pieces of the pattern “in parallel”.

## Another Example

i	1	2	3	4	5	6	7	8	9	10
P[i]	a	b	b	a	b	a	a	b	b	a
$\pi[i]$	0	0	0	1	2	1	1	2	3	4

$$P_{10} = a \ b \ b \ a \ b \ a \ a \ b \ b \ a$$

$$\begin{array}{|c|c|c|c|} \hline & | & | & | \\ \hline \end{array}$$

$$a \ b \ b \ a = P_4$$

$$P_7 = a \ b \ b \ a \ b \ a \ a$$

$$\begin{array}{|c|} \hline & | \\ \hline \end{array}$$

$$a = P_1$$

$$P_4 = a \ b \ b \ a$$

$$\begin{array}{|c|} \hline & | \\ \hline \end{array}$$

$$a = P_1$$

$$P_1 = a$$

$$\begin{array}{|c|} \hline & | \\ \hline \end{array}$$

$$\epsilon = P_0$$

$$P_9 = a \ b \ b \ a \ b \ a \ a \ b \ b$$

$$\begin{array}{|c|c|c|} \hline & | & | \\ \hline \end{array}$$

$$a \ b \ b = P_3$$

$$P_6 = a \ b \ b \ a \ b \ a$$

$$\begin{array}{|c|} \hline & | \\ \hline \end{array}$$

$$a = P_1$$

$$P_3 = a \ b \ b$$

$$\begin{array}{|c|} \hline & | \\ \hline \end{array}$$

$$\epsilon = P_0$$

$$P_8 = a \ b \ b \ a \ b \ a \ a \ b$$

$$\begin{array}{|c|c|} \hline & | \\ \hline \end{array}$$

$$a \ b = P_2$$

$$P_5 = a \ b \ b \ a \ b$$

$$\begin{array}{|c|c|} \hline & | \\ \hline \end{array}$$

$$a \ b = P_2$$

$$P_2 = a \ b$$

$$\begin{array}{|c|} \hline & | \\ \hline \end{array}$$

$$\epsilon = P_0$$

# Time Complexity

Amortized Analysis --

$\Phi_0$   
↓ loop q = 2 (1<sup>st</sup> iteration)

$\Phi_1$   
↓ loop q = 3 (2<sup>nd</sup> iteration)

$\Phi_2$   
↓ loop q = 4 (3<sup>rd</sup> iteration)

⋮

↓ loop q = m ((m - 1)<sup>st</sup> iteration)

$\Phi_{m-1}$

$\Phi$  = potential function = value of k

Amortized cost:  $\hat{c}_i = c_i + \Phi_i - \Phi_{i-1}$

↑  
iteration      ↗ actual loop cost

## Time Complexity (Continued)

Total amortized cost:  $\sum_{i=1}^{m-1} \hat{c}_i$

$$= \sum_{i=1}^{m-1} (c_i + \Phi_i - \Phi_{i-1})$$

$$= \sum_{i=1}^{m-1} c_i + \Phi_{m-1} - \Phi_0$$

If  $\Phi_{m-1} \geq \Phi_0$ , then amortized cost upper bounds real cost.

We have  $\Phi_0 = 0$  (initial value of k)

$\Phi_{m-1} \geq 0$  (final value of k).

We show  $\hat{c}_i = O(1)$ .

## Time Complexity (Continued)

The value of  $\hat{c}_i$  obviously depends on how many times statement 6 is executed.

Note that  $k > \pi[k]$ . Thus, each execution of statement 6 decreases  $k$  by at least 1.

So, suppose that statements 5..6 iterate several times, decreasing the value of  $k$ .

**We have:** number of iterations  $\leq k_{\text{old}} - k_{\text{new}}$ . Thus,

$$\hat{c}_i \leq O(1) + 2(k_{\text{old}} - k_{\text{new}}) + \Phi_i - \Phi_{i-1}$$

↑                              ↑                      ↑  
for statements                    =  $k_{\text{new}}$     =  $k_{\text{old}}$   
other than 5 & 6

Hence,  $\hat{c}_i = O(1)$ . Total cost is therefore  $O(m)$ .

# Rest of the Algorithm

```
KMP(T, P)
```

```
    n := length[T];
    m := length[P];
     $\pi$  := Compute- $\pi$ (P);
    q := 0;
    for i := 1 to n do
        while q > 0 and P[q+1]  $\neq$  T[i] do
            q :=  $\pi$ [q]
        od;
        if P[q+1] = T[i] then
            q := q + 1
        fi;
        if q = m then
            print "pattern occurs with shift i – m";
            q :=  $\pi$ [q]
        fi
    od
```

Time complexity of loop is  $O(n)$  (similar to the analysis of Compute- $\pi$ ).

Total time is  $O(m + n)$ .

# Example

$P = a b a b c$

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

$T = \begin{matrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 \\ a & b & b & a & b & a & b & a & b & c \end{matrix}$

**Start of 1<sup>st</sup> loop:**  $q = 0, i = 1$  [a]

**2<sup>nd</sup> loop:**  $q = 1, i = 2$  [b]

**3<sup>rd</sup> loop:**  $q = 2, i = 3$  [b] ↗ mismatch

**4<sup>th</sup> loop:**  $q = 0, i = 4$  [a] ↗ detected

**5<sup>th</sup> loop:**  $q = 1, i = 5$  [b]

**6<sup>th</sup> loop:**  $q = 2, i = 6$  [a]

**7<sup>th</sup> loop:**  $q = 3, i = 7$  [b]

**8<sup>th</sup> loop:**  $q = 4, i = 8$  [a] ↗ mismatch

**9<sup>th</sup> loop:**  $q = 3, i = 9$  [b] ↗ detected

**10<sup>th</sup> loop:**  $q = 4, i = 10$  [c] ↗ match

**Termination:**  $q = 5$  ↗ detected

Please see the book for formal correctness proofs.  
(They're *very* tedious.)

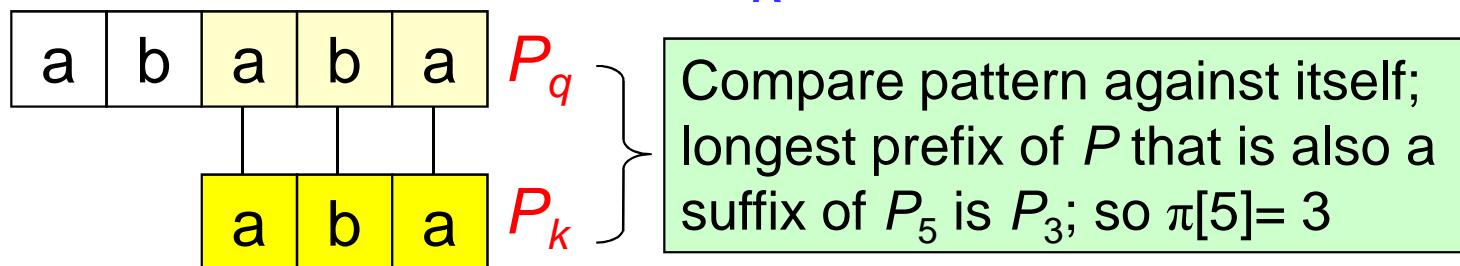
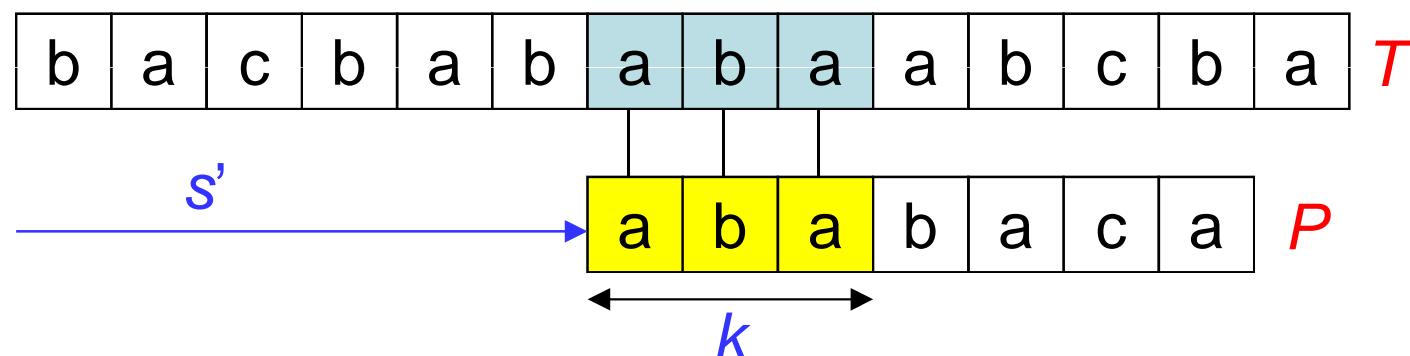
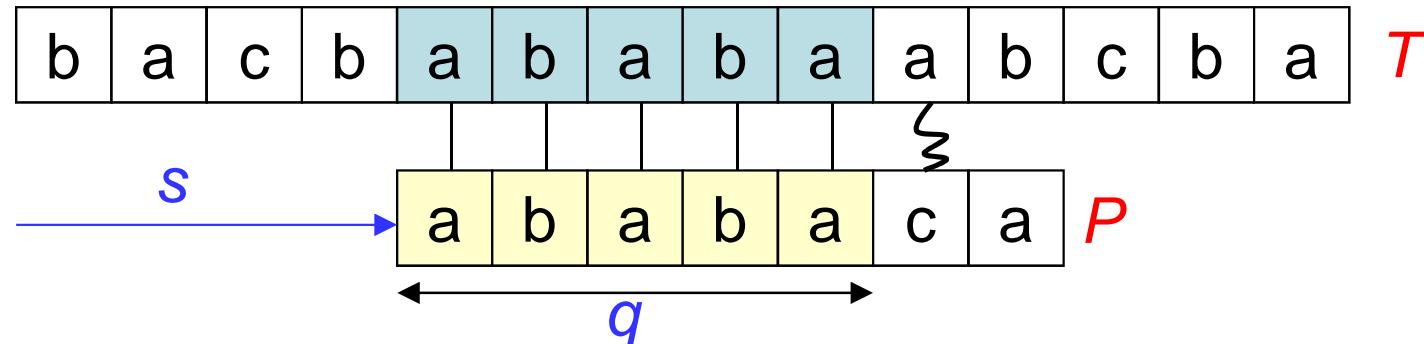
# Terminology/Notations

- String  $w$  is a *prefix* of string  $x$ , if  $x=wy$  for some string  $y$  (e.g., “srilan” of “srilanka”)
- String  $w$  is a *suffix* of string  $x$ , if  $x=yw$  for some string  $y$  (e.g., “anka” of “srilanka”)
- The  $k$ -character prefix of the pattern  $P[1..m]$  denoted by  $P_k$ 
  - E.g.,  $P_0 = \epsilon$ ,  $P_m = P = P[1..m]$

# Prefix Function for a Pattern

- Given that pattern prefix  $P[1..q]$  matches text characters  $T[(s+1)..(s+q)]$ , what is the least shift  $s' > s$  such that
$$P[1..k] = T[(s'+1)..(s'+k)] \text{ where } s'+k=s+q?$$
- At the new shift  $s'$ , no need to compare the first  $k$  characters of  $P$  with corresponding characters of  $T$ 
  - Since we know that they match

## Prefix Function: Example 1

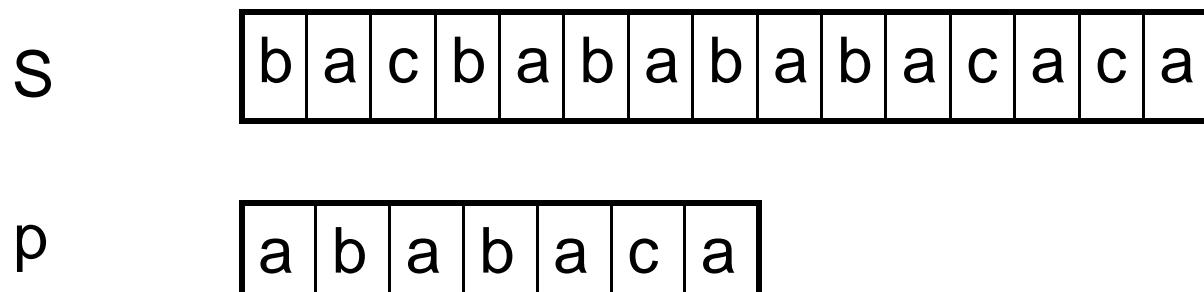


## Prefix Function: Example 2

$i$	1	2	3	4	5	6	7	8	9	10
$P[i]$	a	b	a	b	a	b	a	b	c	a
$\pi[i]$	0	0	1	2	3	4	5	6	0	1

$$\pi[q] = \max \{ k \mid k < q \text{ and } P_k \text{ is a suffix of } P_q \}$$

**Illustration:** given a String 'S' and pattern 'p' as follows:



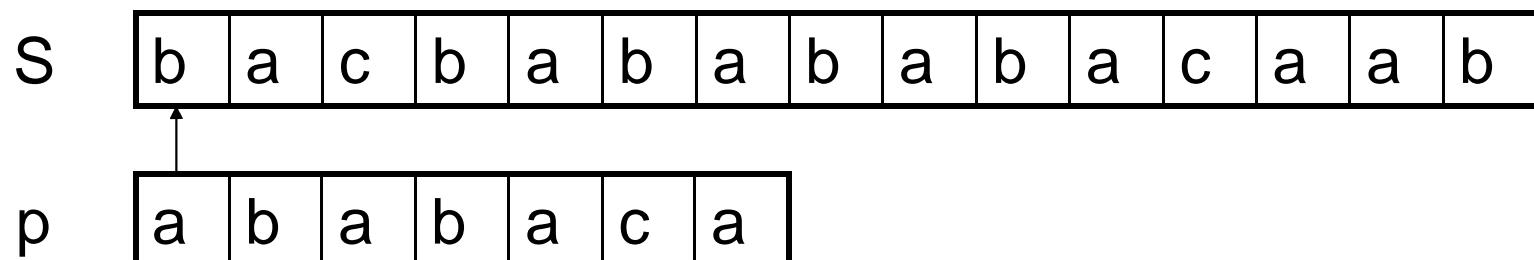
Let us execute the KMP algorithm to find whether 'p' occurs in 'S'.

*For 'p' the prefix function,  $\Pi$  was computed previously and is as follows:*

q	1	2	3	4	5	6	7
p	a	b	a	b	a	c	a
$\Pi$	0	0	1	2	3	0	1

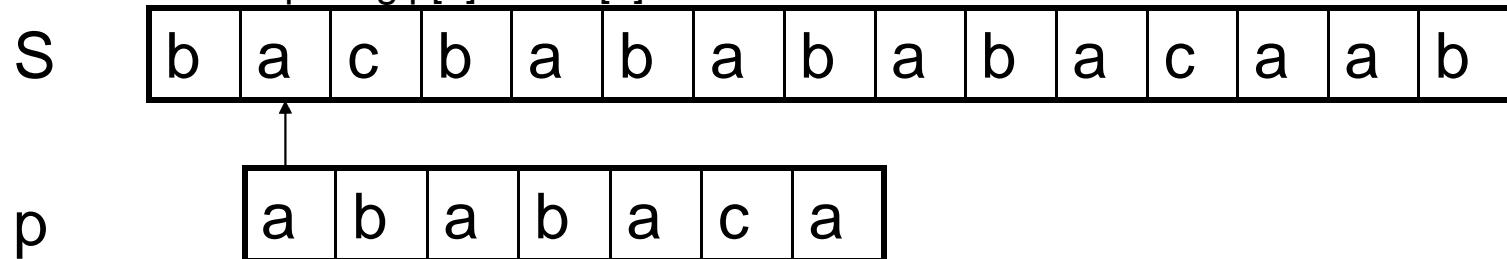
Initially:  $n = \text{size of } S = 15$ ;  
 $m = \text{size of } p = 7$

Step 1:  $i = 1, q = 0$   
comparing  $p[1]$  with  $S[1]$



$P[1]$  does not match with  $S[1]$ . ‘p’ will be shifted one position to the right.

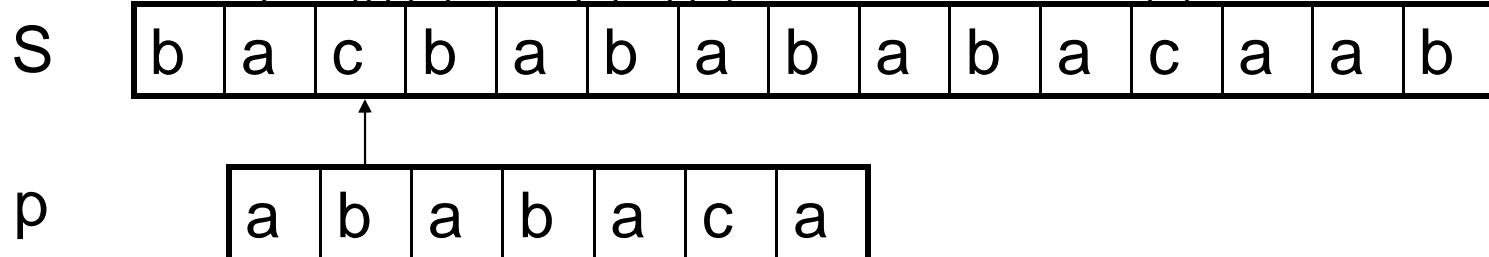
Step 2:  $i = 2, q = 0$   
comparing  $p[1]$  with  $S[2]$



$P[1]$  matches  $S[2]$ . Since there is a match, p is not shifted.

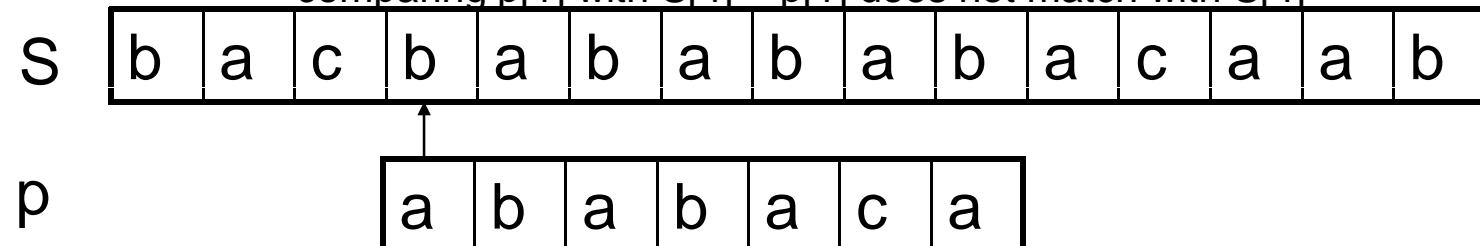
Step 3:  $i = 3, q = 1$

Comparing  $p[2]$  with  $S[3]$   $p[2]$  does not match with  $S[3]$

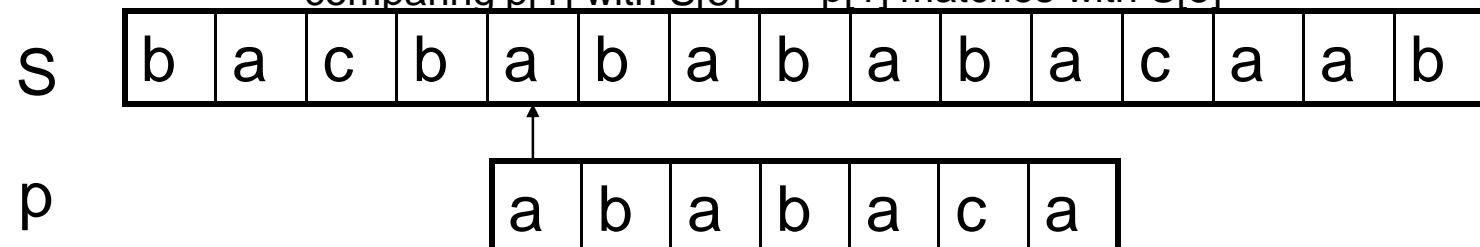


Backtracking on p, comparing  $p[1]$  and  $S[3]$

Step 4:  $i = 4, q = 0$  Comparing  $p[1]$  with  $S[4]$   $p[1]$  does not match with  $S[4]$

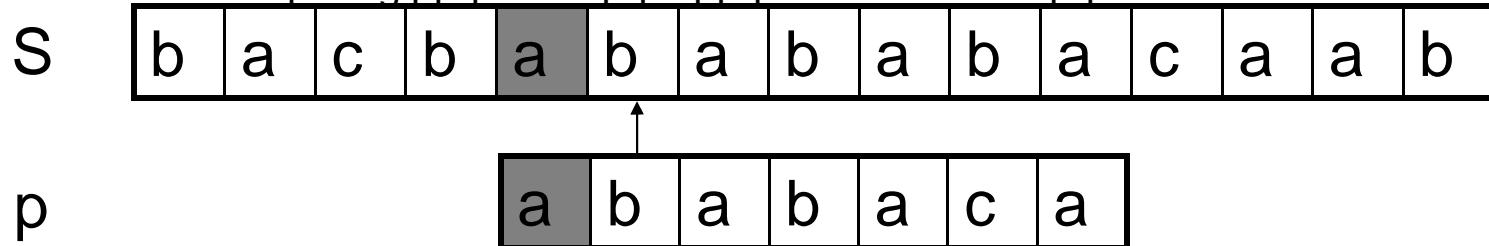


Step 5:  $i = 5, q = 0$  Comparing  $p[1]$  with  $S[5]$   $p[1]$  matches with  $S[5]$



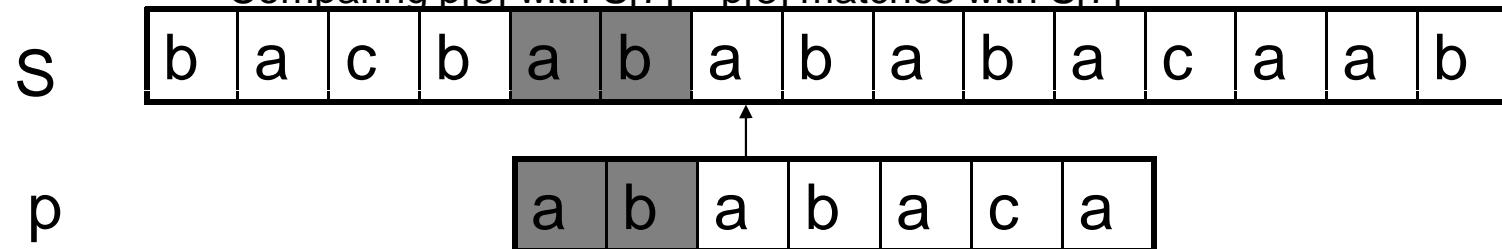
Step 6:  $i = 6, q = 1$

Comparing  $p[2]$  with  $S[6]$   $p[2]$  matches with  $S[6]$



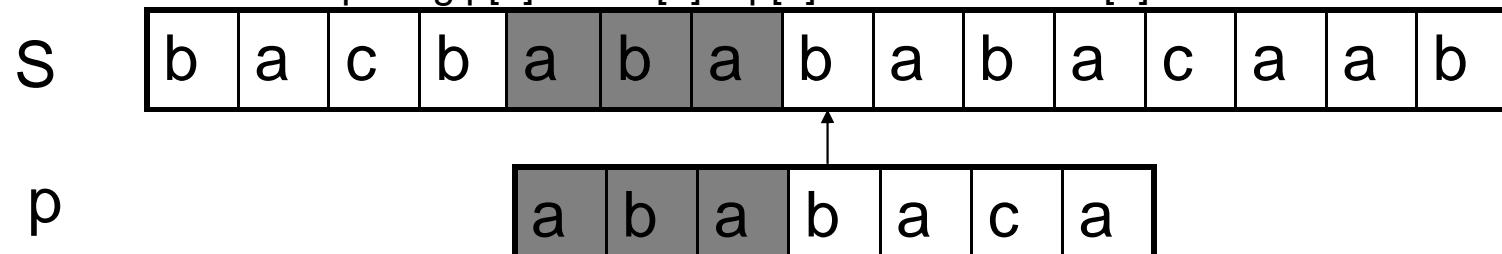
Step 7:  $i = 7, q = 2$

Comparing  $p[3]$  with  $S[7]$   $p[3]$  matches with  $S[7]$

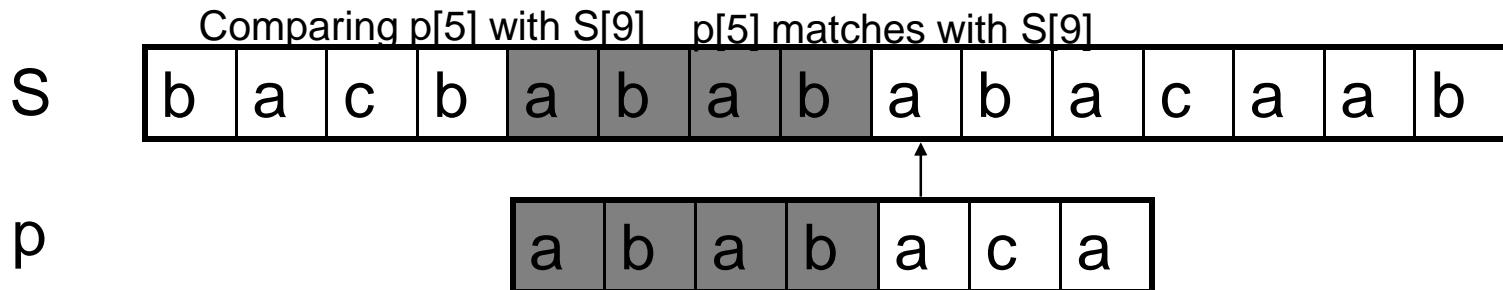


Step 8:  $i = 8, q = 3$

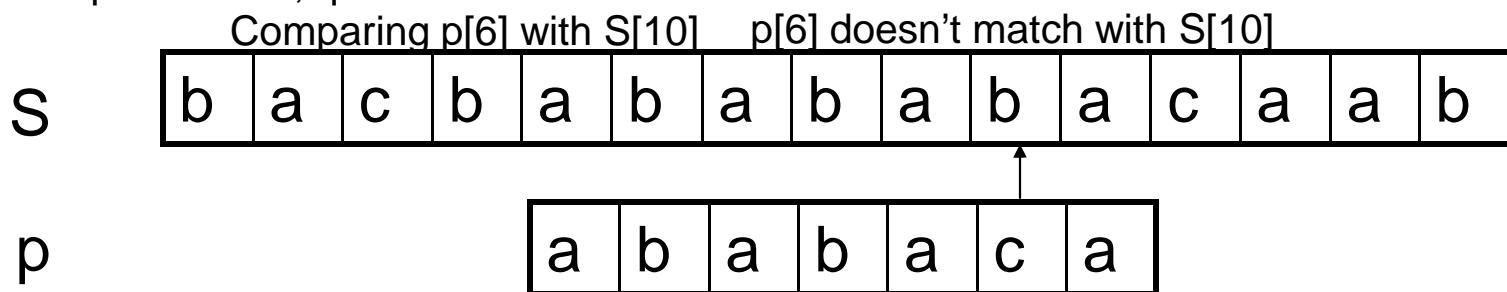
Comparing  $p[4]$  with  $S[8]$   $p[4]$  matches with  $S[8]$



Step 9:  $i = 9, q = 4$

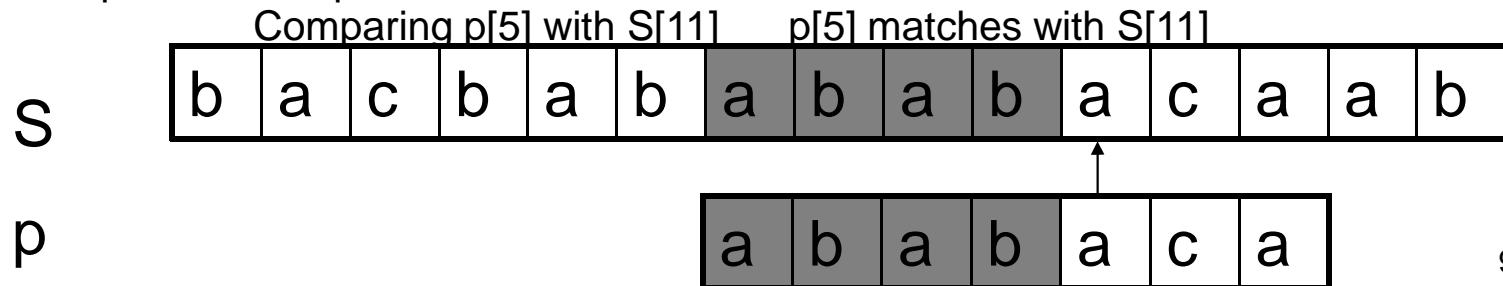


Step 10:  $i = 10, q = 5$

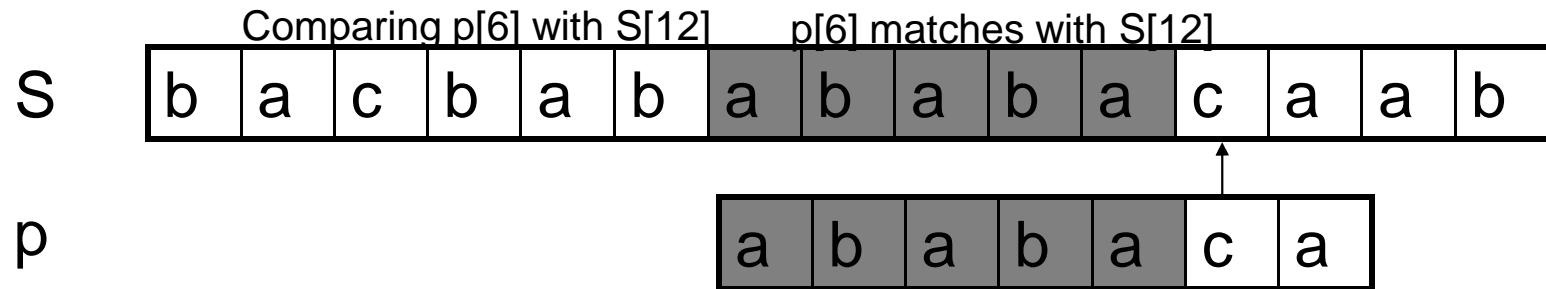


Backtracking on  $p$ , comparing  $p[4]$  with  $S[10]$  because after mismatch  $q = \Pi[5] = 3$

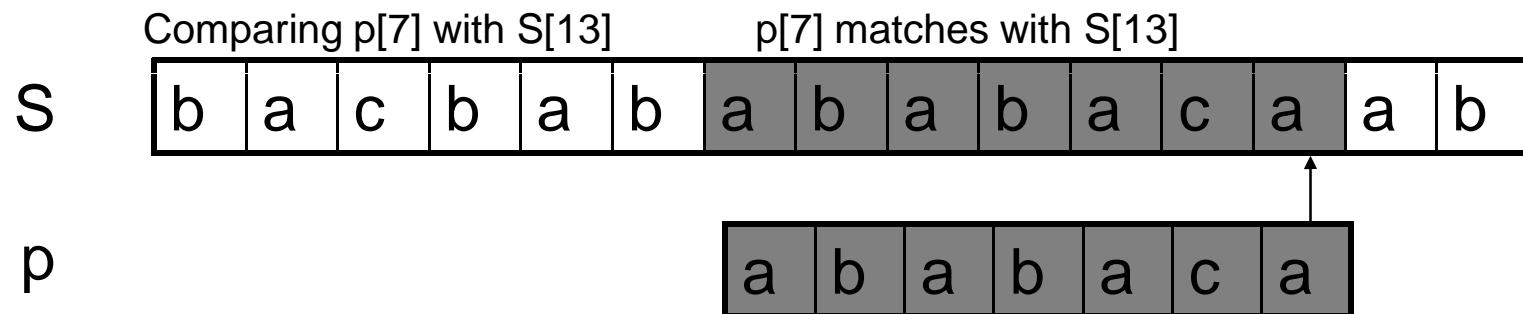
Step 11:  $i = 11, q = 4$



Step 12:  $i = 12, q = 5$

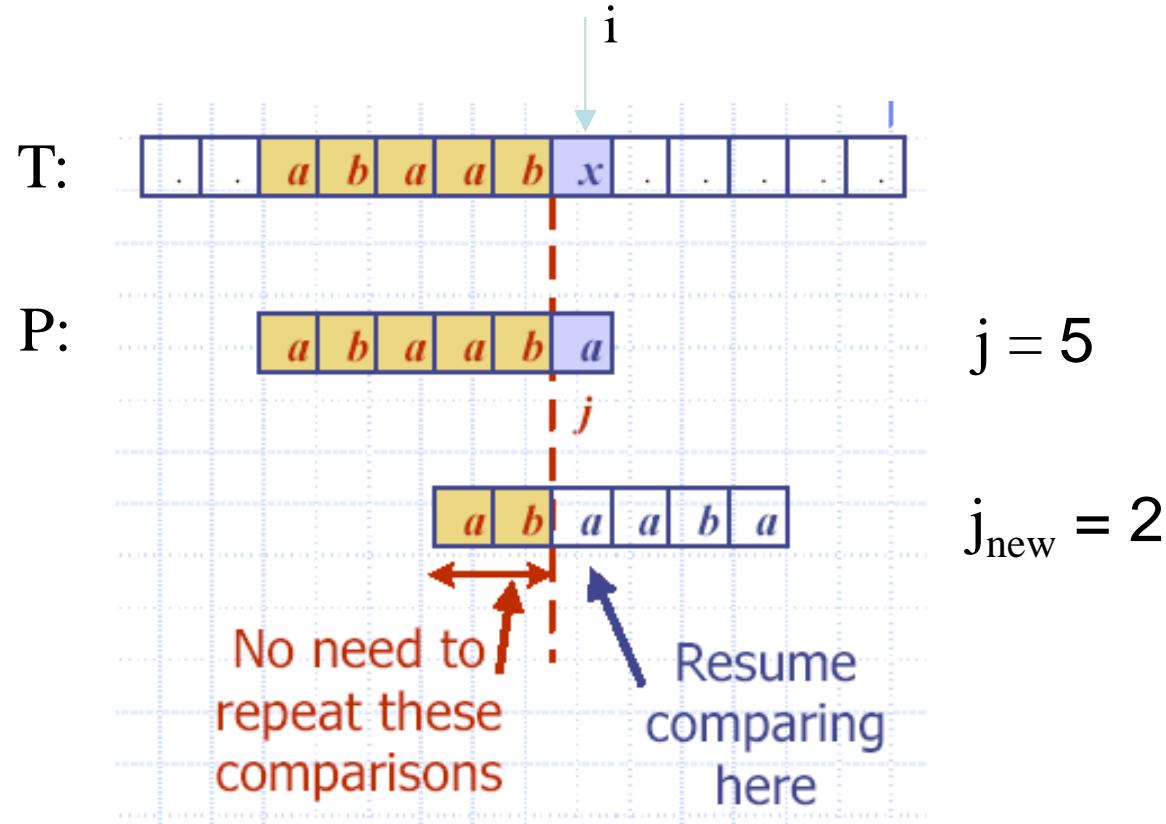


Step 13:  $i = 13, q = 6$



Pattern 'p' has been found to completely occur in string 'S'. The total number of shifts that took place for the match to be found are:  $i - m = 13 - 7 = 6$  shifts.

# KMP Example



# Why

j == 5

- Find largest prefix (start) of:  
"a b a a b"                                   ( P[0..j-1] )

which is suffix (end) of:

"b a a b"                                   ( p[1 .. j-1] )

- Answer: "a b"
- Set j = 2 // the new j value

# KMP Failure Function

- KMP preprocesses the pattern to find matches of prefixes of the pattern with the pattern itself.
- $j$  = mismatch position in  $P[]$
- $k$  = position before the mismatch ( $k = j - 1$ ).
- The *failure function*  $F(k)$  is defined as the size of the largest prefix of  $P[0..k]$  that is also a suffix of  $P[1..k]$ .

# Failure Function Example

( $k == j-1$ )

- P: "abaaba"  
j: 012345

	0	1	2	3	4
	0	0	1	1	2

$F(k)$  is the size of  
the largest prefix.

- In code,  $F()$  is represented by an array,  
like the table.

# Why is $F(4) == 2$

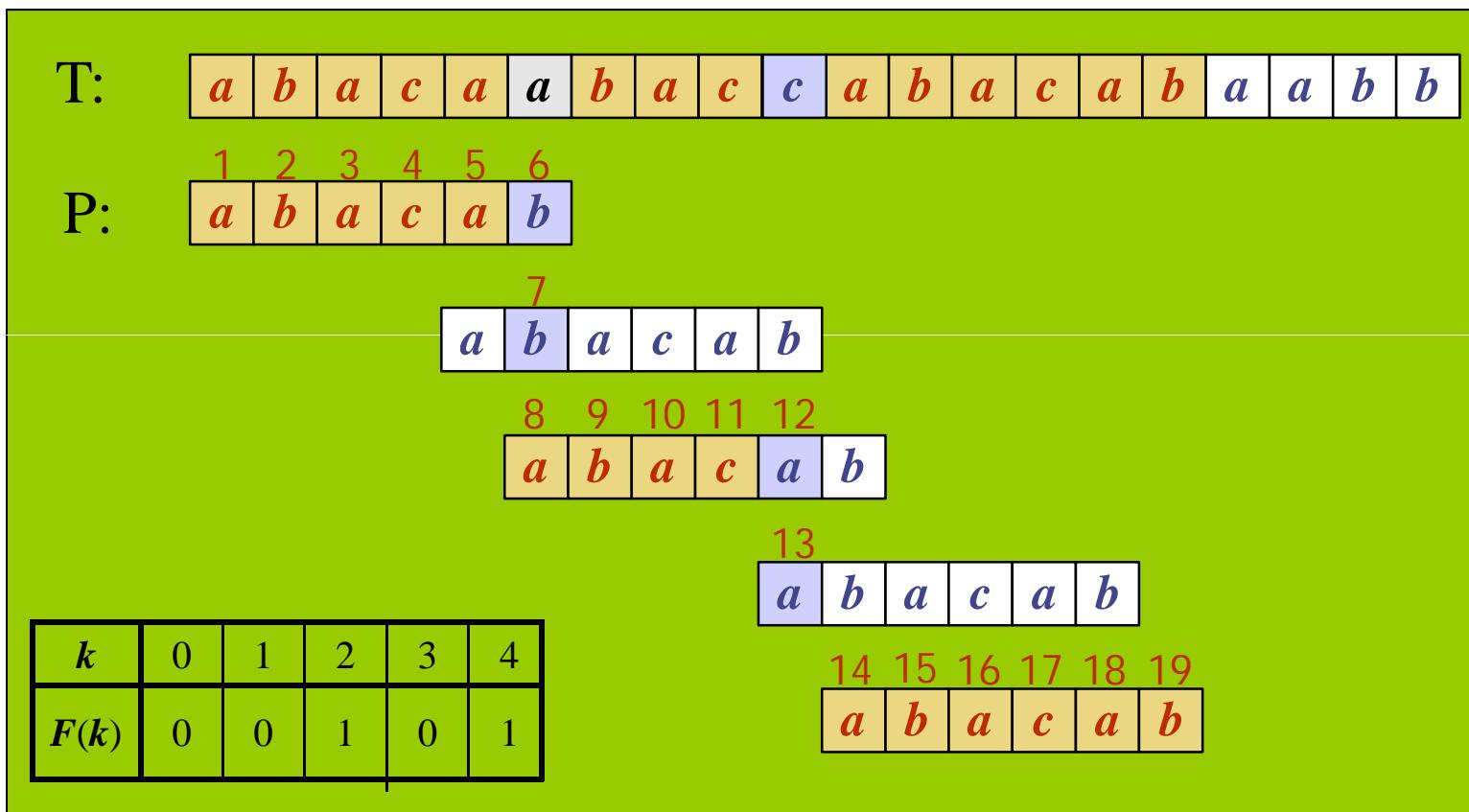
P: "abaaba"

- $F(4)$  means
  - find the size of the largest prefix of  $P[0..4]$  that is also a suffix of  $P[1..4]$
  - = find the size largest prefix of "abaab" that is also a suffix of "baab"
  - = find the size of "ab"
  - = 2

# Using the Failure Function

- Knuth-Morris-Pratt's algorithm modifies the brute-force algorithm.
  - if a mismatch occurs at  $P[j]$  (i.e.  $P[j] \neq T[i]$ ), then
    - $k = j - 1;$
    - $j = F(k);$  // obtain the new  $j$

# Example



Why is  $F(4) == 1$   
P: "abacab"

- $F(4)$  means
  - find the size of the largest prefix of  $P[0..4]$  that is also a suffix of  $P[1..4]$
  - = find the size largest prefix of "abaca" that is also a suffix of "baca"
  - = find the size of "a"
  - = 1

# **Boyer and Moore String matching Algorithm**

A fast string searching algorithm. *Communications of the ACM.*  
Vol. 20 p.p. 762-772, 1977.

**BOYER, R.S. and MOORE, J.S.**

# Boyer and Moore Algorithm

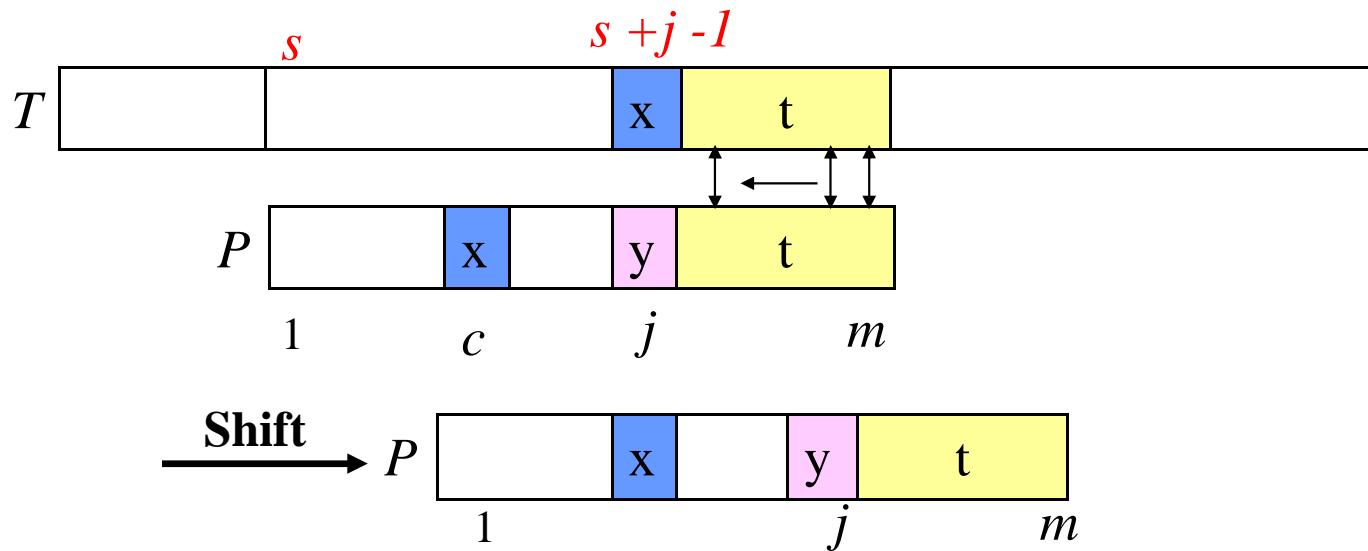
- The algorithm compares the pattern  $P$  with the substring of sequence  $T$  within a sliding window in the **right-to-left order**.
- The **bad character rule** and **good suffix rule** are used to determine the movement of sliding window.
- Very fast, used in Python library.

# Bad Character Rule

Suppose that  $P_1$  is aligned to  $T_s$  now, and we perform a pair-wise comparing between text  $T$  and pattern  $P$  from right to left.

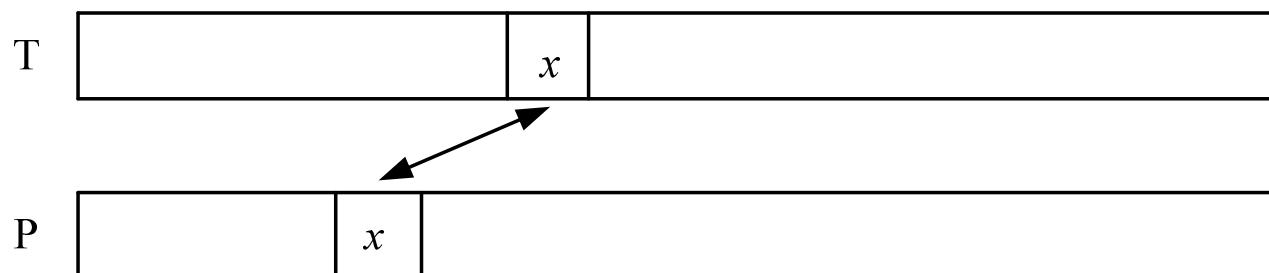
Assume that the first mismatch occurs when comparing  $T_{s+j-1}$  with  $P_j$ .

Since  $T_{s+j-1} \neq P_j$ , we move the pattern  $P$  to the right such that the largest position  $c$  in the left of  $P_j$  is equal to  $T_{s+j-1}$ . We can shift the pattern at least  $(j-c)$  positions right.



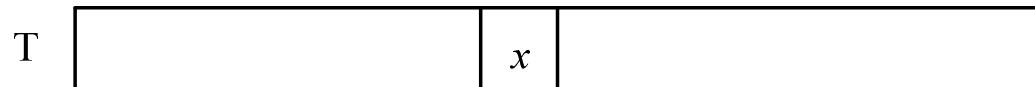
## Rule 2-1: Character Matching Rule (A Special Version of Rule 2)

- Bad character rule uses Rule 2-1 (Character Matching Rule).
- For any character  $x$  in  $T$ , find the nearest  $x$  in  $P$  which is to the left of  $x$  in  $T$ .

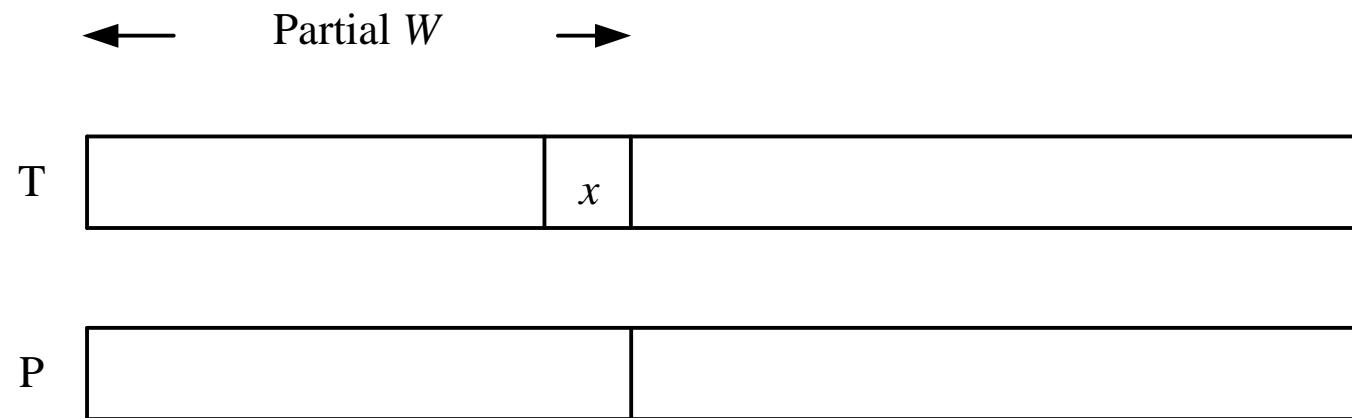


# Implication of Rule 2-1

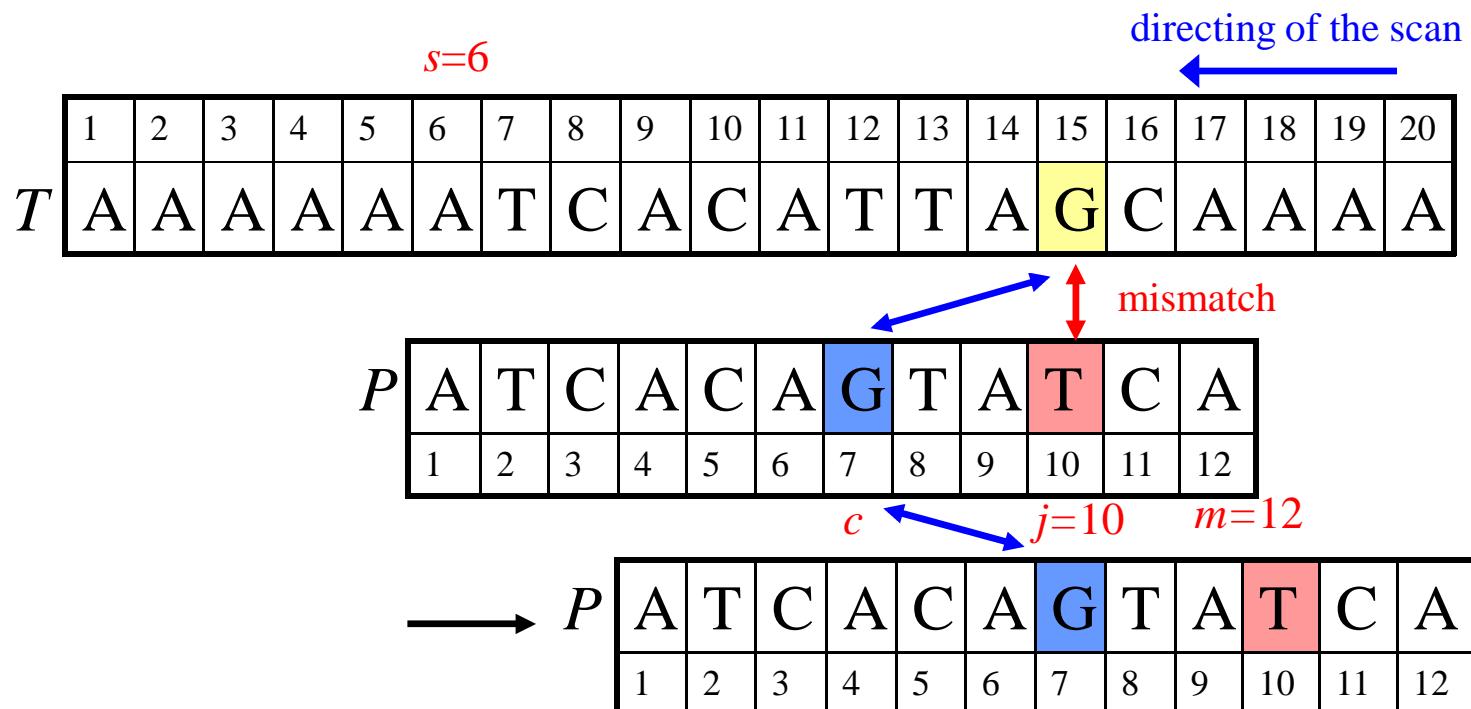
- Case 1. If there is a  $x$  in  $P$  to the left of  $T$ , move  $P$  so that the two  $x$ 's match.



- Case 2: If no such a  $x$  exists in  $P$ , consider the partial window defined by  $x$  in  $T$  and the string to the left of it.

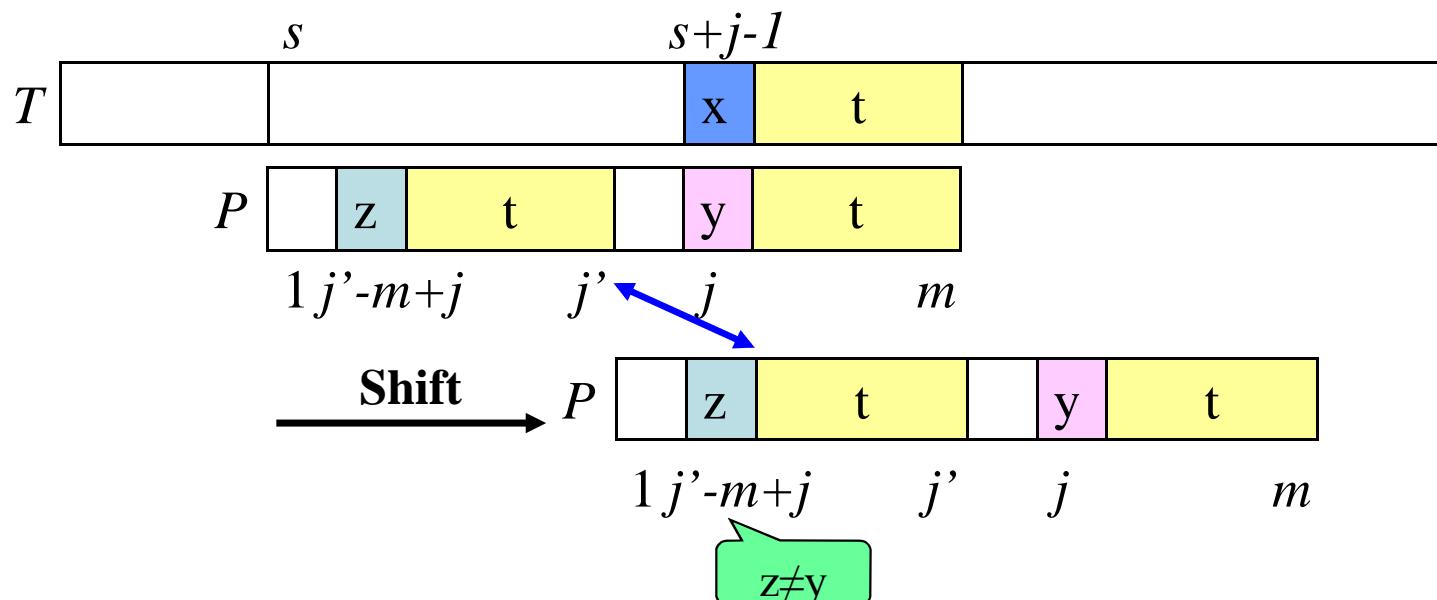


- Ex: Suppose that  $P_1$  is aligned to  $T_6$  now. We compare pairwise between  $T$  and  $P$  from right to left. Since  $T_{16,17} = P_{11,12} = \text{“CA”}$  and  $T_{15} = \text{“G”} \neq P_{10} = \text{“T”}$ . Therefore, we find the rightmost position  $c=7$  in the left of  $P_{10}$  in  $P$  such that  $P_c$  is equal to “G” and we can move the window at least  $(10-7=3)$  positions.



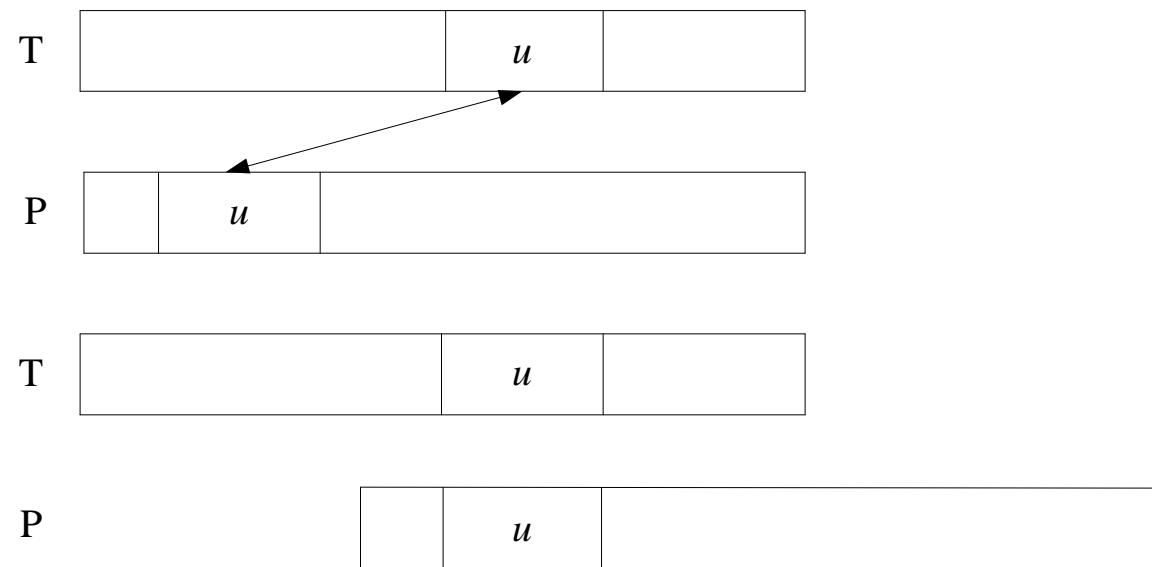
# Good Suffix Rule 1

- If a mismatch occurs in  $T_{s+j-1}$ , we match  $T_{s+j-1}$  with  $P_{j'-m+j}$ , where  $j'$  ( $m-j+1 \leq j' < m$ ) is the **largest position** such that
  - (1)  $P_{j+1,m}$  is a suffix of  $P_{1,j}$
  - (2)  $P_{j'-(m-j)} \neq P_j$
- We can move the window at least  $(m-j')$  position(s).

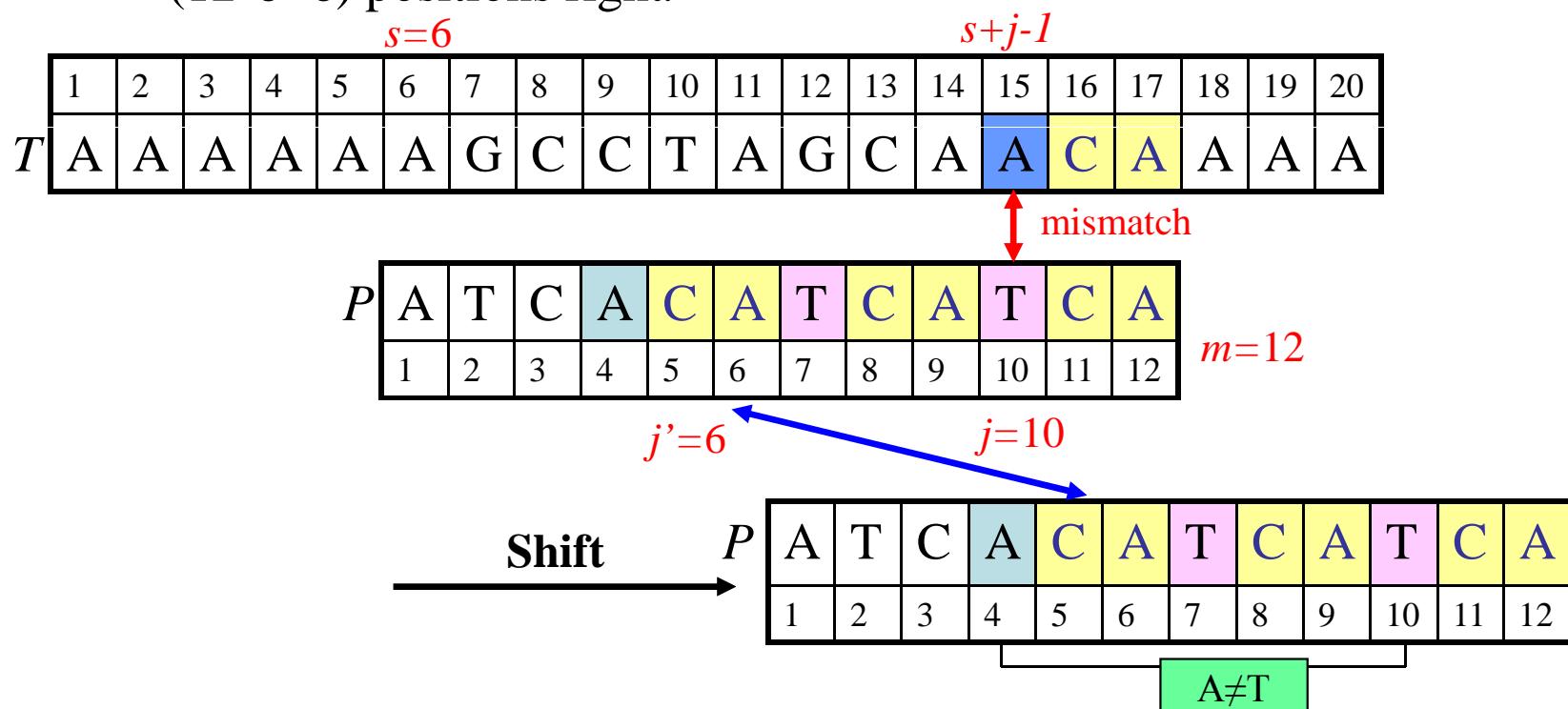


## Rule 2: The Substring Matching Rule

- For any substring  $u$  in  $T$ , find a nearest  $u$  in  $P$  which is to the left of it. If such a  $u$  in  $P$  exists, move  $P$ ; otherwise, we may define a new partial window.



- Ex: Suppose that  $P_1$  is aligned to  $T_6$  now. We compare pairwise between  $P$  and  $T$  from right to left. Since  $T_{16,17} = \text{"CA"} = P_{11,12}$  and  $T_{15} = \text{"A"} \neq P_{10} = \text{"T"}$ . We find the substring "CA" in the left of  $P_{10}$  in  $P$  such that "CA" is the suffix of  $P_{1,6}$  and the left character to this substring "CA" in  $P$  is not equal to  $P_{10} = \text{"T"}$ . Therefore, we can move the window at least  $m-j'$  ( $12-6=6$ ) positions right.

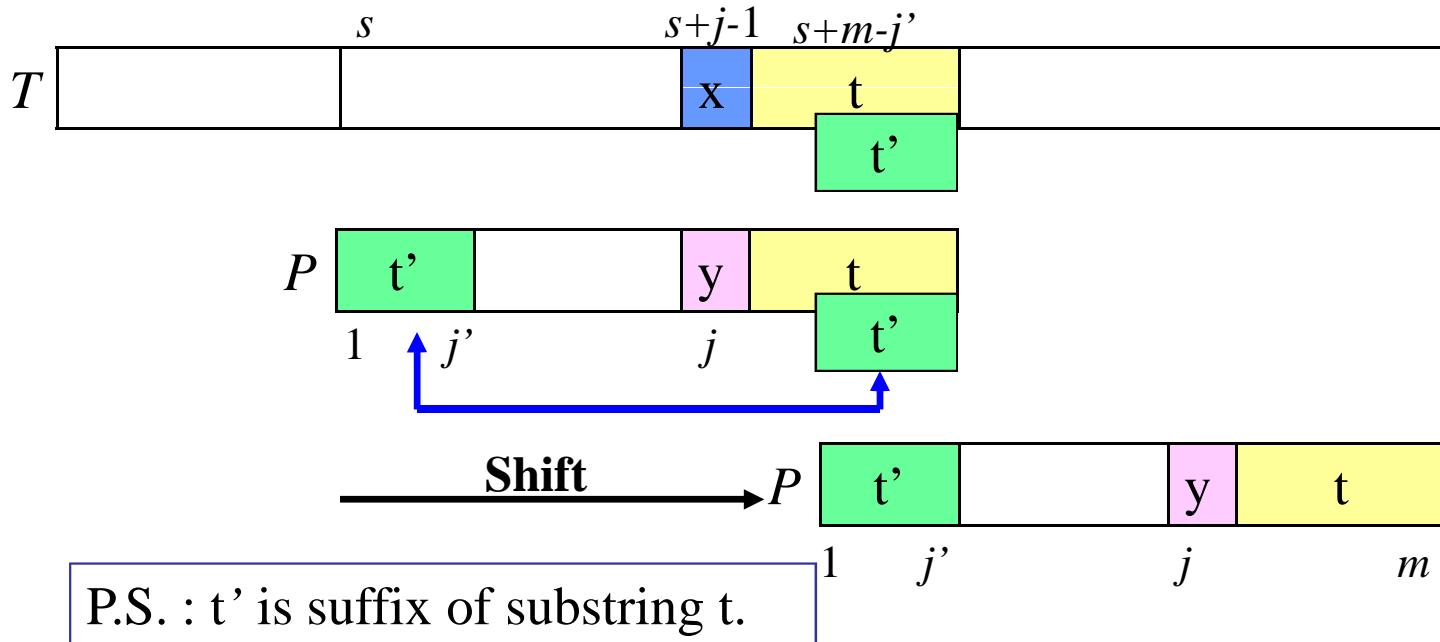


## Good Suffix Rule 2

Good Suffix Rule 2 is used only when Good Suffix Rule 1 can not be used. That is, t does not appear in  $P(1, j)$ . Thus, t is **unique** in P.

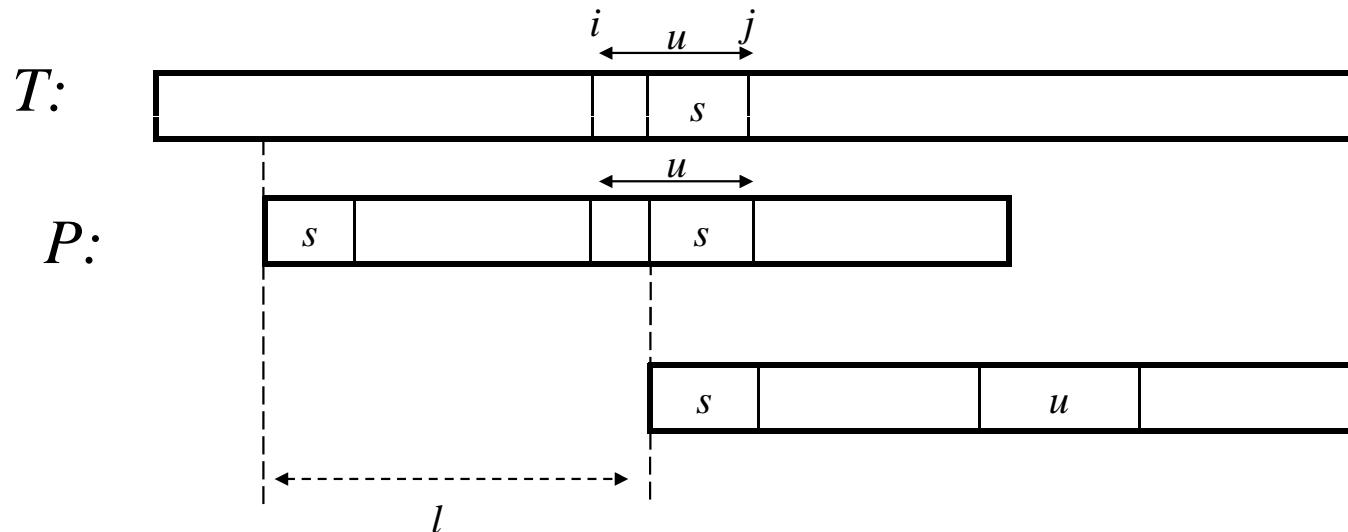
- If a mismatch occurs in  $T_{s+j-1}$ , we match  $T_{s+m-j'}$  with  $P_1$ , where  $j'$  ( $1 \leq j' \leq m-j$ ) is **the largest position** such that

$P_{1,j'}$  is a suffix of  $P_{j+1,m}$ .



# Rule 3-1: Unique Substring Rule

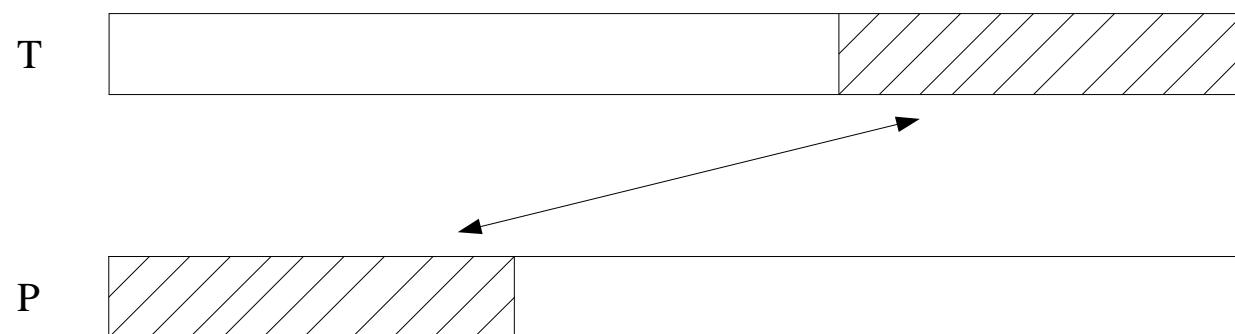
- The substring  $u$  appears in  $P$  exactly once.
- If the substring  $u$  matches with  $T_{i,j}$ , no matter whether a mismatch occurs in some position of  $P$  or not, we can slide the window by  $l$ .



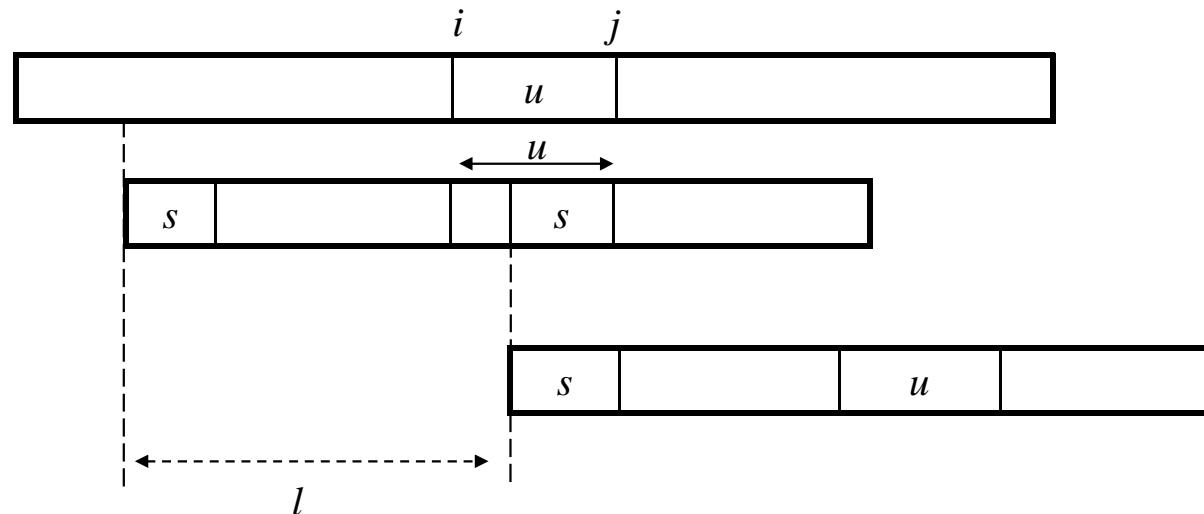
The string  $s$  is the longest prefix of  $P$  which equals to a suffix of  $u$ .

# Rule 1: The Suffix to Prefix Rule

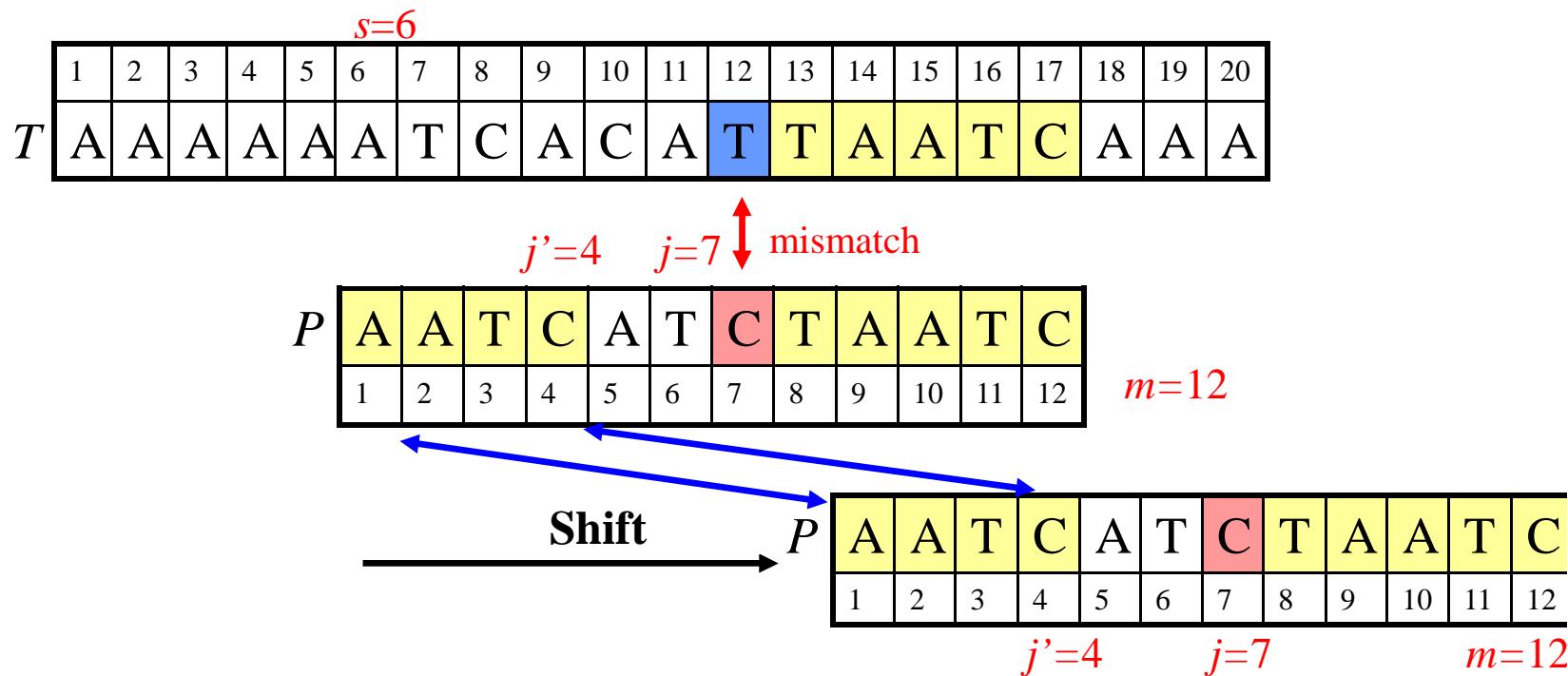
- For a window to have any chance to match a pattern, in some way, there must be a suffix of the window which is equal to a prefix of the pattern.



- Note that the above rule also uses Rule 1.
- It should also be noted that the unique substring is the shorter and the more right-sided the better.
- A short  $u$  guarantees a short (or even empty)  $s$  which is desirable.



- Ex: Suppose that  $P_1$  is aligned to  $T_6$  now. We compare pair-wise between  $P$  and  $T$  from right to left. Since  $T_{12} \neq P_7$  and there is no substring  $P_{8,12}$  in left of  $P_8$  to exactly match  $T_{13,17}$ . We find a longest suffix “AATC” of substring  $T_{13,17}$ , the longest suffix is also prefix of  $P$ . We shift the window such that the last character of prefix substring to match the last character of the suffix substring. Therefore, we can shift at least  $12-4=8$  positions.



- Let  $Bc(a)$  be the rightmost position of  $a$  in  $P$ . The function will be used for applying *bad character rule*.

$j$	1	2	3	4	5	6	7	8	9	10	11	12
$P$	A	T	C	A	C	A	T	C	A	T	C	A
$\Sigma$	A	C	G	T								
$B$	12	11	0	10								

- We can move our pattern right at least  $j - B(T_{s+j-1})$  position by above  $Bc$  function.

$j$	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
$T$	A	G	C	T	A	G	C	C	T	G	C	A	C	G	T	A	C	A
$j$	1	2	3	4	5	6	7	8	9	10	11	12						
$P$	A	T	C	A	C	A	T	C	A	T	C	A						

Move at least  
 $10 - B(G) = 10$  positions

Let  $Gs(j)$  be **the largest number of shifts by *good suffix rule*** when a mismatch occurs for comparing  $P_j$  with some character in  $T$ .

- $gs_1(j)$  be the largest  $k$  such that  $P_{j+1,m}$  is a suffix of  $P_{1,k}$  and  $P_{k-m+j} \neq P_j$ , where  $m-j+1 \leq k < m$ ; 0 if there is no such  $k$ .  
( $gs_1$  is for Good Suffix Rule 1)
- $gs_2(j)$  be the largest  $k$  such that  $P_{1,k}$  is a suffix of  $P_{j+1,m}$ , where  $1 \leq k \leq m-j$ ; 0 if there is no such  $k$ .  
( $gs_2$  is for Good Suffix Rule 2.)
- $Gs(j) = m - \max\{gs_1, gs_2\}$ , if  $j = m$ ,  $Gs(j)=1$ .

$j$	1	2	3	4	5	6	7	8	9	10	11	12
$P$	A	T	C	A	C	A	T	C	A	T	C	A
$gs_1$	0	0	0	0	0	0	9	0	0	6	1	0
$gs_2$	4	4	4	4	4	4	4	4	1	1	1	0
$Gs$	8	8	8	8	8	8	3	8	11	6	11	1

$$gs_1(7)=9$$

$\because P_{8,12}$  is a suffix of  $P_{1,9}$   
and  $P_4 \neq P_7$

$$gs_2(7)=4$$

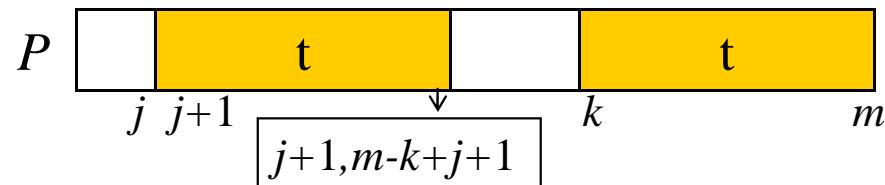
$\because P_{1,4}$  is a suffix of  $P_{8,12}$

How do we obtain  $gs_1$  and  $gs_2$ ?

In the following, we shall show that by constructing the **Suffix Function**, we can kill two birds with one arrow.

# Suffix function $f'$

- For  $1 \leq j \leq m-1$ , let the suffix function  $f'(j)$  for  $P_j$  be the **smallest**  $k$  such that  $P_{k,m} = P_{j+1,m-k+j+1}$ ; ( $j+2 \leq k \leq m$ )
  - If there is no such  $k$ , we set  $f' = m+1$ .
  - If  $j=m$ , we set  $f'(m)=m+2$ .



- Ex:

$j$	1	2	3	4	5	6	7	8	9	10	11	12
$P$	A	T	C	A	C	A	T	C	A	T	C	A
$f'$	10	11	12	8	9	10	11	12	13	13	13	14

- $f'(4)=8$ , it means that  $P_{f'(4),m} = P_{8,12} = P_{5,9} = P_{4+1,4+1+m-f'(4)}$
- Since there is no  $k$  for  $13=j+2 \leq k \leq 12$ , we set  $f'(11)=13$ .

Suppose that the Suffix is obtained. How can we use it to obtain  $gs_1$  and  $gs_2$ ?

$gs_1$  can be obtained by scanning the Suffix function from right to left.

# BM Example

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	
T	G	A	T	C	G	A	T	C	A	A	T	C	A	T	C	A	C	A	T	G	A	T	C	A
<i>P</i>	A	T	C	A	C	A	T	C	A	T	C	A												
	1	2	3	4	5	6	7	8	9	10	11	12												

<i>j</i>	1	2	3	4	5	6	7	8	9	10	11	12
<i>P</i>	A	T	C	A	C	A	T	C	A	T	C	A
<i>f'</i>	10	11	12	8	9	10	11	12	13	13	13	14

## Example

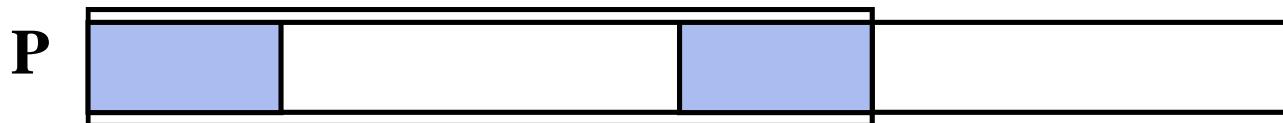
As for Good Suffix Rule 2, it is relatively easier.

$j$	1	2	3	4	5	6	7	8	9	10	11	12
$P$	A	T	C	A	C	A	T	C	A	T	C	A
$f'$	10	11	12	8	9	10	11	12	13	13	13	14

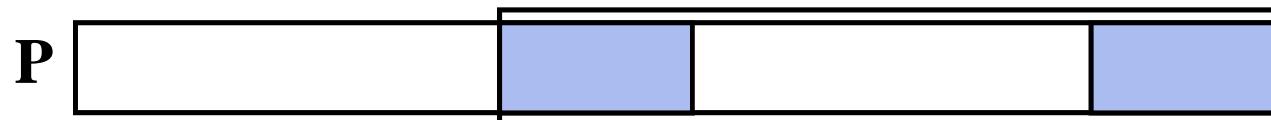
Question: How can we construct the Suffix function?

To explain this, let us go back to the prefix function used in the MP Algorithm.

The following figure illustrates the prefix function in the MP Algorithm.



The following figure illustrates the suffix function of the BM Algorithm.



We now can see that actually the suffix function is the same as the prefix. The only difference is now we consider a suffix. Thus, the recursive formula for the prefix function in MP Algorithm can be slightly modified for the suffix function in BM Algorithm.

- The formula of suffix function  $f'$  as follows :

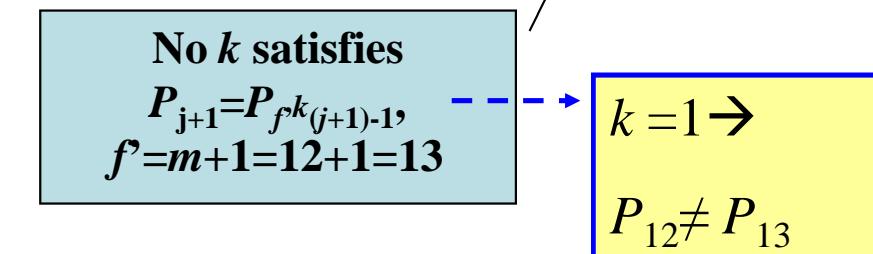
Let  $f'^x(y) = f'(f'^{x-1}(y))$  for  $x > 1$  and  $f'^1(y) = f'(y)$

$$f'(j) = \begin{cases} m+2, & \text{if } j = m \\ f'^k(j+1)-1, & \text{if } 1 \leq j \leq m-1 \text{ and there exists the smallest } k \geq 1 \text{ such that } P_{j+1} = P_{f'^k(j+1)-1}; \\ m+1, & \text{otherwise} \end{cases}$$

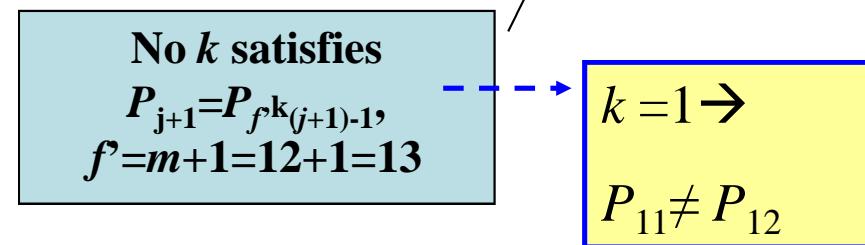
$j$	1	2	3	4	5	6	7	8	9	10	11	12
$P$	A	T	C	A	C	A	T	C	A	T	C	A
$f'$												14

$j=m=12$ ,  
 $f'=m+2=14$

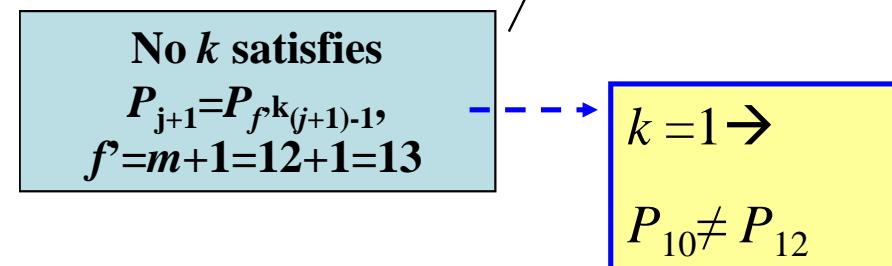
$j$	1	2	3	4	5	6	7	8	9	10	11	12
$P$	A	T	C	A	C	A	T	C	A	T	C	A
$f'$											13	14



<i>j</i>	1	2	3	4	5	6	7	8	9	10	11	12
<i>P</i>	A	T	C	A	C	A	T	C	A	T	C	A
<i>f'</i>										13	13	14



<i>j</i>	1	2	3	4	5	6	7	8	9	10	11	12
<i>P</i>	A	T	C	A	C	A	T	C	A	T	C	A
<i>f'</i>									13	13	13	14



$j$	1	2	3	4	5	6	7	8	9	10	11	12
$P$	A	T	C	A	C	A	T	C	A	T	C	A
$f'$								12	13	13	13	14

$\because P_{j+1} = P_{f'(j+1)-1} \Rightarrow P_9 = P_{12}$ ,  
 $f' = f'(j+1) - 1 = 13 - 1 = 12$

$j$	1	2	3	4	5	6	7	8	9	10	11	12
$P$	A	T	C	A	C	A	T	C	A	T	C	A
$f'$							11	12	13	13	13	14

$\because P_{j+1} = P_{f'(j+1)-1} \Rightarrow P_8 = P_{11}$ ,  
 $f' = f'(j+1) - 1 = 12 - 1 = 11$

$j$	1	2	3	4	5	6	7	8	9	10	11	12
$P$	A	T	C	A	C	A	T	C	A	T	C	A
$f'$					8	9	10	11	12	13	13	13

$\because P_{j+1} = P_{f' \cdot 1(j+1)-1} \Rightarrow P_5 = P_8,$   
 $f' = f'(j+1) - 1 = 9 - 1 = 8$

$j$	1	2	3	4	5	6	7	8	9	10	11	12
$P$	A	T	C	A	C	A	T	C	A	T	C	A
$f'$			12	8	9	10	11	12	13	13	13	14

$\because P_{j+1} = P_{f' \cdot 3(j+1)-1} \Rightarrow P_4 = P_{f' \cdot 3(4)-1} = P_{12},$   
 $f' = f'^{\textcolor{red}{3}}(j+1) - 1 = 13 - 1 = 12$

$j$	1	2	3	4	5	6	7	8	9	10	11	12
$P$	A	T	C	A	C	A	T	C	A	T	C	A
$f'$		11	12	8	9	10	11	12	13	13	13	14

$\because P_{j+1} = P_{f'(j+1)-1} \Rightarrow P_3 = P_{f'(3)-1} = P_{11},$   
 $f' = f'(j+1) - 1 = 12 - 1 = 11$

$j$	1	2	3	4	5	6	7	8	9	10	11	12
$P$	A	T	C	A	C	A	T	C	A	T	C	A
$f'$	10	11	12	8	9	10	11	12	13	13	13	14

$\because P_{j+1} = P_{f'(j+1)-1} \Rightarrow P_2 = P_{f'(2)-1} = P_{10},$   
 $f' = f'(j+1) - 1 = 11 - 1 = 10$

- Let  $G'(j)$ ,  $1 \leq j \leq m$ , to be the largest number of shifts by good suffix rules.
- First, we set  $G'(j)$  to zeros as their initializations.

$j$	1	2	3	4	5	6	7	8	9	10	11	12
$P$	A	T	C	A	C	A	T	C	A	T	C	A
$f'$	10	11	12	8	9	10	11	12	13	13	13	14
$G'$	0	0	0	0	0	0	0	0	0	0	0	0

- **Step1:** We scan from right to left and  $gs_1(j)$  is determined during the scanning, then  $gs_1(j) \geq gs_2(j)$

**Observe:**

If  $P_j = P_4 \neq P_7 = P_{f'(j)-1}$ , we know  $gs_1(f'(j)-1) = m + j - f'(j) + 1 = 9$ .

**If  $t = f'(j)-1 \leq m$  and  $P_j \neq P_t$ ,  $G'(t) = m - gs_1(f'(j)-1) = f'(j) - 1 - j$ .**  
 $f'(k)(x) = f'(k-1)(f'(x) - 1)$ ,  $k \geq 2$

- When  $j=12$ ,  $t=13$ .  $t > m$ .
  - When  $j=11$ ,  $t=12$ . Since  $P_{11} = 'C' \neq 'A' = P_{12}$ ,
- $$\begin{aligned} G'(t) &= m - \max\{gs_1(t), gs_2(t)\} = m - \underline{gs_1(t)} \\ &= \underline{f'(j) - 1 - j} \\ \Rightarrow G'(12) &= 13 - 1 - 11 = 1. \end{aligned}$$

$j$	1	2	3	4	5	6	7	8	9	10	11	12
$P$	A	T	C	A	C	A	T	C	A	T	C	A
$f'$	10	11	12	8	9	10	11	12	13	13	13	14
$G'$	0	0	0	0	0	0	0	0	0	0	0	1

If  $t = f'(j) - 1 \leq m$  and  $P_j \neq P_t$ ,  $G'(t) = f'(j) - 1 - j$ .  
 $f''(k)(x) = f''(k-1)(f'(x) - 1)$ ,  $k \geq 2$

- When  $j=10$ ,  $t=12$ . Since  $P_{10} = 'T' \neq 'A' = P_{12}$ ,  $G'(12) \neq 0$ .
- When  $j=9$ ,  $t=12$ .  $P_9 = 'A' = P_{12}$ .
- When  $j=8$ ,  $t=11$ .  $P_8 = 'C' = P_{11}$ .
- When  $j=7$ ,  $t=10$ .  $P_7 = 'T' = P_{10}$
- When  $j=6$ ,  $t=9$ .  $P_6 = 'A' = P_9$
- When  $j=5$ ,  $t=8$ .  $P_5 = 'C' = P_8$
- When  $j=4$ ,  $t=7$ . Since  $P_4 = 'A' \neq P_7 = 'T'$ ,  $G'(7) = 8 - 1 - 4 = 3$

Besides,  $t = f''(2)(4) - 1 = f(f'(4) - 1) - 1 = 10$ . Since  $P_4 = 'A' \neq P_{10} = 'T'$ ,  $G'(10) = f'(7) - 1 - j = 11 - 1 - 4 = 6$ .

$j$	1	2	3	4	5	6	7	8	9	10	11	12
$P$	A	T	C	A	C	A	T	C	A	T	C	A
$f'$	10	11	12	8	9	10	11	12	13	13	13	14
$G'$	0	0	0	0	0	0	3	0	0	6	0	1

If  $t = f'(j)-1 \leq m$  and  $P_j \neq P_t$ ,  $G'(t) = f'(j) - 1 - j$ .  
 $f'^{(k)}(x) = f'^{(k-1)}(f'(x) - 1)$ ,  $k \geq 2$

- When  $j=3$ ,  $t=11$ .  $P_3 = 'C' = P_{11}$ .
- When  $j=2$ ,  $t=10$ .  $P_2 = 'T' = P_{10}$
- When  $j=1$ ,  $t=9$ .  $P_1 = 'A' = P_9$ .

$j$	1	2	3	4	5	6	7	8	9	10	11	12
$P$	A	T	C	A	C	A	T	C	A	T	C	A
$f'$	10	11	12	8	9	10	11	12	13	13	13	14
$G'$	0	0	0	0	0	0	3	0	6	0	0	1

- By the above discussion, we can obtain the values using the Good Suffix Rule 1 by scanning the pattern from right to left.

- **Step2:** Continuously, we will try to obtain the values using ***Good Suffix Rule 2*** and those values are still zeros now and scan from left to right.

$j$	1	2	3	4	5	6	7	8	9	10	11	12
$P$	A	T	C	A	C	A	T	C	A	T	C	A
$f'$	10	11	12	8	9	10	11	12	13	13	13	14
$G'$	0	0	0	0	0	0	3	0	0	6	0	1

- Let  $k'$  be the **smallest  $k$**  in  $\{1, \dots, m\}$  such that  $P_{f'(k)(1)-1} = P_1$  and  $f'(k)(1)-1 \leq m$ .

**Observe:**

$$\because P_{1,4} = P_{9,12}, \therefore gs_2(j) = m - (f'(1)-1) + 1 = 4, \text{ where } 1 \leq j \leq f'(k')(1)-2.$$

- If  $G'(j)$  is not determined in the first scan and  $1 \leq j \leq f'(k')(1)-2$ , thus, in the second scan, we set  $G'(j) = m - \max\{gs_1(j), gs_2(j)\} = m - gs_2(j) = f'(k')(1) - 2$ . If no such  $k$  exists, set each undetermined value of  $G$  to  $m$  in the second scan.
- $k=1=k'$** , since  $P_{f'(1)-1} = P_9 = "A" = P_1$ , we set  $G'(j) = f'(1)-2$  for  $j=1,2,3,4,5,6,8$ .

$j$	1	2	3	4	5	6	7	8	9	10	11	12
$P$	A	T	C	A	C	A	T	C	A	T	C	A
$f'$	10	11	12	8	9	10	11	12	13	13	13	14
$G'$	8	8	8	8	8	8	3	8	0	6	0	1

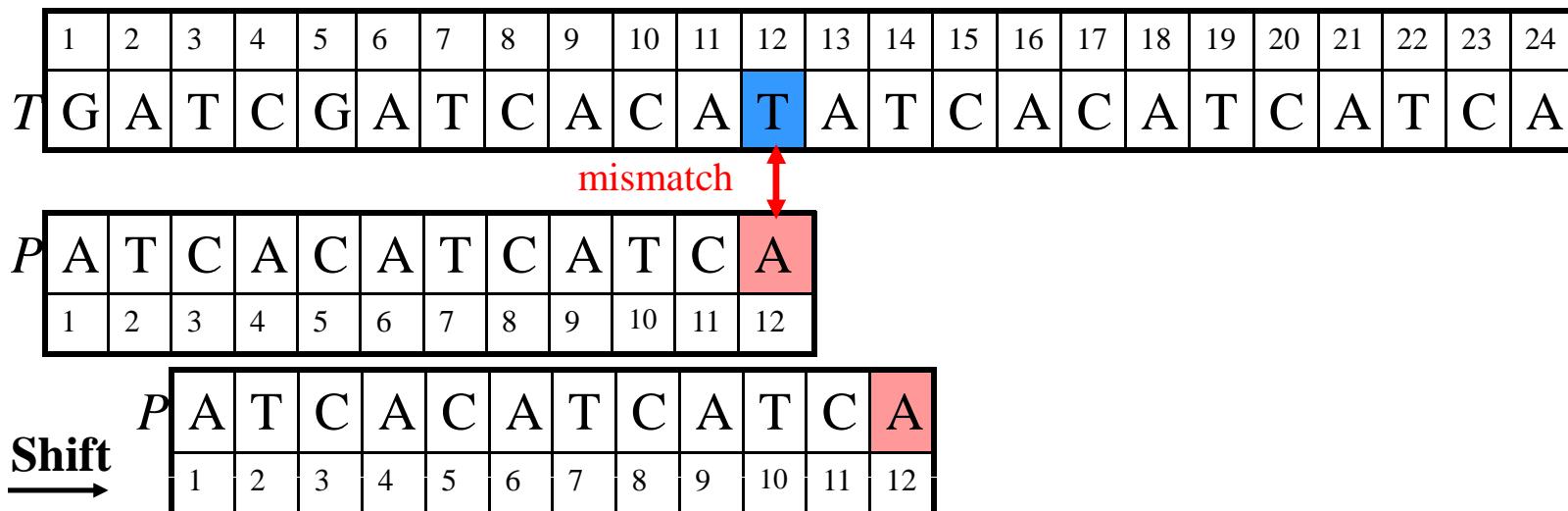
- Let  $z$  be  $f'(k')(1)$ -2. Let  $k''$  be the **largest value  $k$**  such that  $f''^{(k)}(z) - 1 \leq m$ .
- Then we set  $G'(j) = m - gs_2(j) = m - (m - f''^{(i)}(z) - 1) = f''^{(i)}(z) - 1$ , where  $1 \leq i \leq k''$  and  $f''^{(i-1)}(z) < j \leq f''^{(i)}(z) - 1$  and  $f''^{(0)}(z) = z$ .
- For example,  $z=8$  :
  - $\triangleright k=1, f''^{(1)}(8)-1=11 \leq m=12$
  - $\triangleright k=2, f''^{(2)}(8)-1=12 \leq m=12 \Rightarrow k''=2$
  - $\triangleright i=1, f''^{(0)}(8)-1 = 7 < j \leq f''^{(1)}(8)-1=11.$
  - $\triangleright i=2, f''^{(1)}(8)-1 = 11 < j \leq f''^{(2)}(8)-1=12.$
  - $\triangleright$  We set  $G(9)$  and  $G(11)=f''^{(1)}(8) - 1 = 12-1 = 11$ .

$j$	1	2	3	4	5	6	7	8	9	10	11	12
$P$	A	T	C	A	C	A	T	C	A	T	C	A
$f'$	10	11	12	8	9	10	11	12	13	13	13	14
$G'$	8	8	8	8	8	8	3	8	11	6	11	1

We essentially have to decide the maximum number of steps.  
We can move the window right when a mismatch occurs. This  
is decided by the following function:

$$\max\{G'(j), j-B(T_{s+j-1})\}$$

# Example



$j$	1	2	3	4	5	6	7	8	9	10	11	12
$P$	A	T	C	A	C	A	T	C	A	T	C	A
$f'$	10	11	12	8	9	10	11	12	13	13	13	14
$G'$	8	8	8	8	8	8	3	8	11	6	11	1

$\Sigma$	A	C	G	T
$B$	12	11	0	10

We compare  $T$  and  $P$  from right to left. Since  $T_{12} = "T" \neq P_{12} = "A"$ , the largest movement =  $\max\{G'(j), j-B(T_{s+j-1})\} = \max\{G'(12), 12-B(T_{12})\} = \max\{1, 12-10\} = 2$ .

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
T	G	A	T	C	G	A	T	C	A	C	A	T	A	T	C	A	C	A	T	C	A	T	C

mismatch

P	A	T	C	A	C	A	T	C	A	T	C	A
	1	2	3	4	5	6	7	8	9	10	11	12

Shift →

P	A	T	C	A	C	A	T	C	A	T	C	A
	1	2	3	4	5	6	7	8	9	10	11	12

j	1	2	3	4	5	6	7	8	9	10	11	12
P	A	T	C	A	C	A	T	C	A	T	C	A
f'	10	11	12	8	9	10	11	12	13	13	13	14
G'	8	8	8	8	8	8	3	8	11	6	11	1

$\Sigma$	A	C	G	T
B	12	11	0	10

After moving, we compare  $T$  and  $P$  from right to left. Since  $T_{14} = "T" \neq P_{12} = "A"$ ,  
the largest movement =  $\max\{G'(j), j-B(Ts+j-1)\} = \max\{G'(12), 12-B(T_{14})\}$   
=  $\max\{1, 12-10\} = 2$ .

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
T	G	A	T	C	G	A	T	C	A	C	A	T	A	T	C	A	C	A	T	C	A	T	C

mismatch

P	A	T	C	A	C	A	T	<span style="background-color: red;">C</span>	A	T	C	A
	1	2	3	4	5	6	7	8	9	10	11	12

Shift

P	A	T	C	A	C	A	T	<span style="background-color: red;">C</span>	A	T	C	A
	1	2	3	4	5	6	7	8	9	10	11	12

$j$	1	2	3	4	5	6	7	<span style="background-color: yellow;">8</span>	9	10	11	12
$P$	A	T	C	A	C	A	T	<span style="background-color: red;">C</span>	A	T	C	A
$f'$	10	11	12	8	9	10	11	12	13	13	13	14
$G'$	8	8	8	8	8	8	3	8	11	6	11	1

$\Sigma$	A	C	G	<span style="background-color: yellow;">T</span>
$B$	12	11	0	10

After moving, we compare  $T$  and  $P$  from right to left. Since  $T_{12} = \text{"T"} \neq P_8 = \text{"G"}$ , the largest movement =  $\max\{G'(8), j - B(T_{12})\} = \max\{8, 8 - 10\} = 8$ .

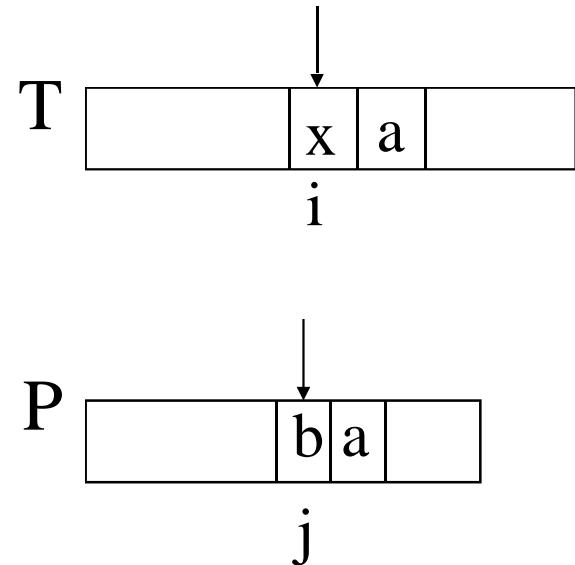
# Time Complexity

- The preprocessing phase in  $O(m+\Sigma)$  time and space complexity and searching phase in  $O(mn)$  time complexity.
- The worst case time complexity for the ***Boyer-Moore*** method would be  $O((n-m+1)m)$ .
- It was proved that this algorithm has  $O(m)$  comparisons when  $P$  is not in  $T$ . However, this algorithm has  $O(mn)$  comparisons when  $P$  is in  $T$ .

# The Boyer-Moore Algorithm

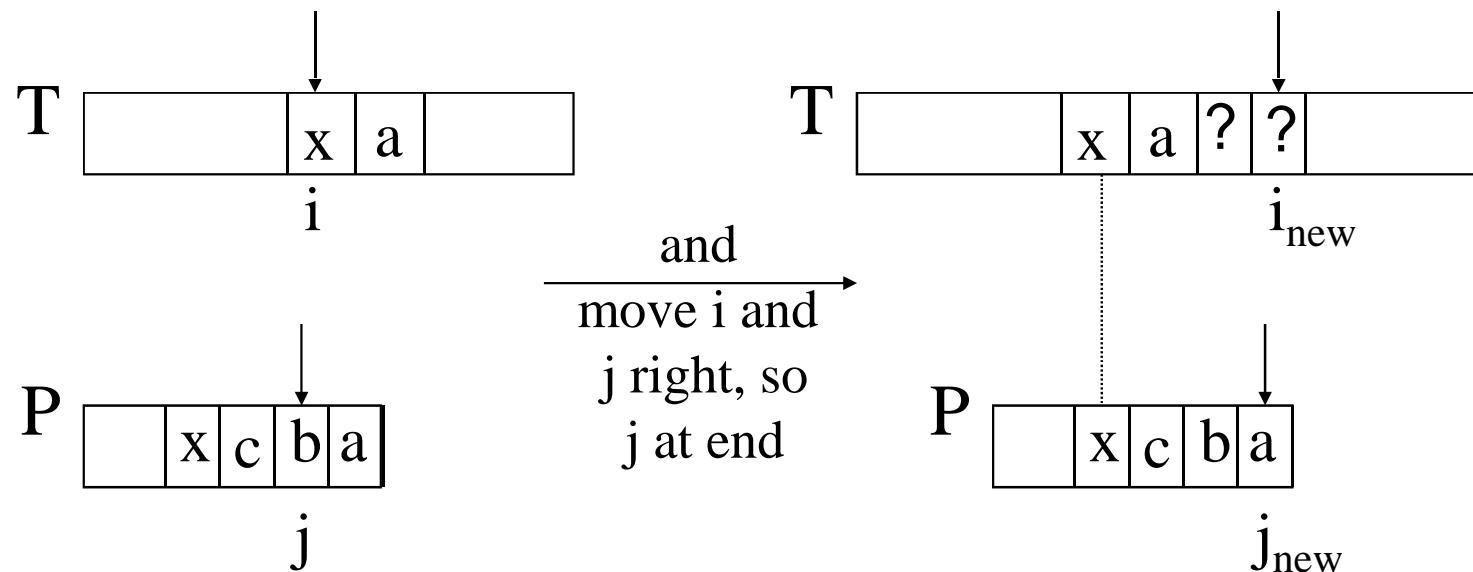
- The Boyer-Moore pattern matching algorithm is based on two techniques.
- 1. The *looking-glass* technique
  - find P in T by moving *backwards* through P, starting at its end

- 2. The *character-jump* technique
  - when a mismatch occurs at  $T[i] == x$
  - the character in pattern  $P[j]$  is not the same as  $T[i]$
- There are 3 possible cases, tried in order.



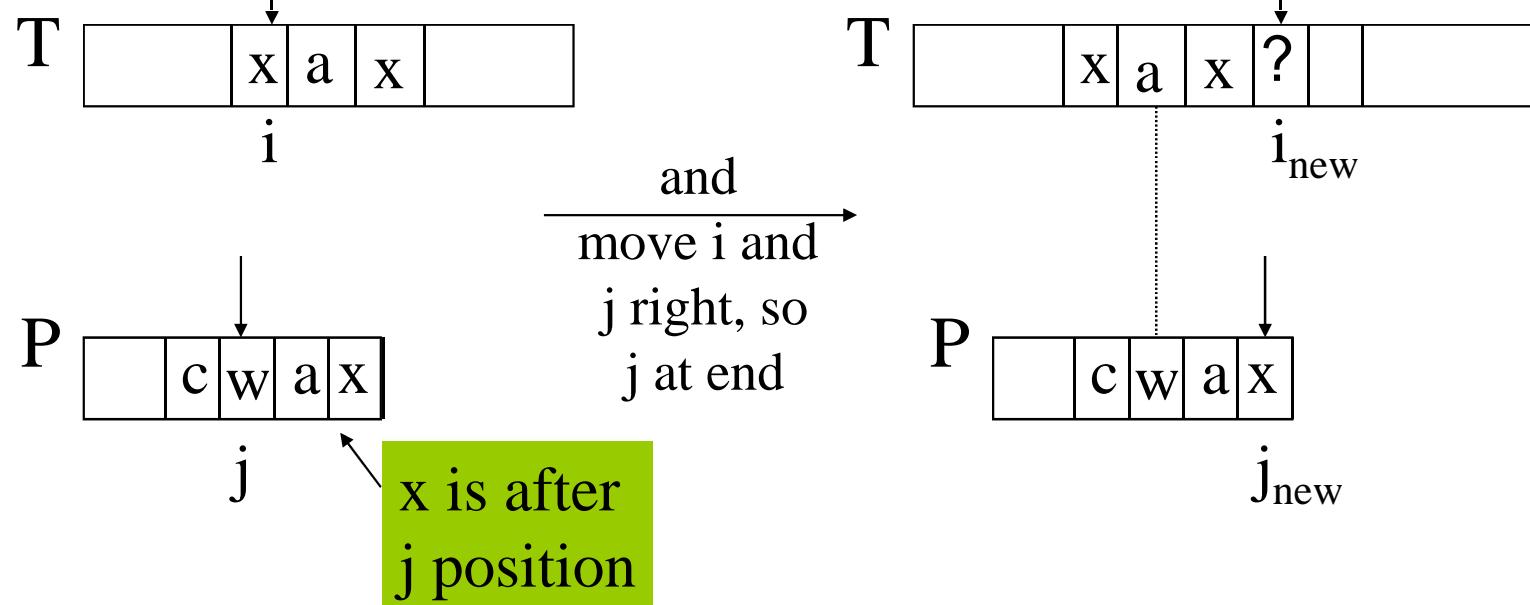
# Case 1

- If  $P$  contains  $x$  somewhere, then try to *shift P* right to align the last occurrence of  $x$  in  $P$  with  $T[i]$ .



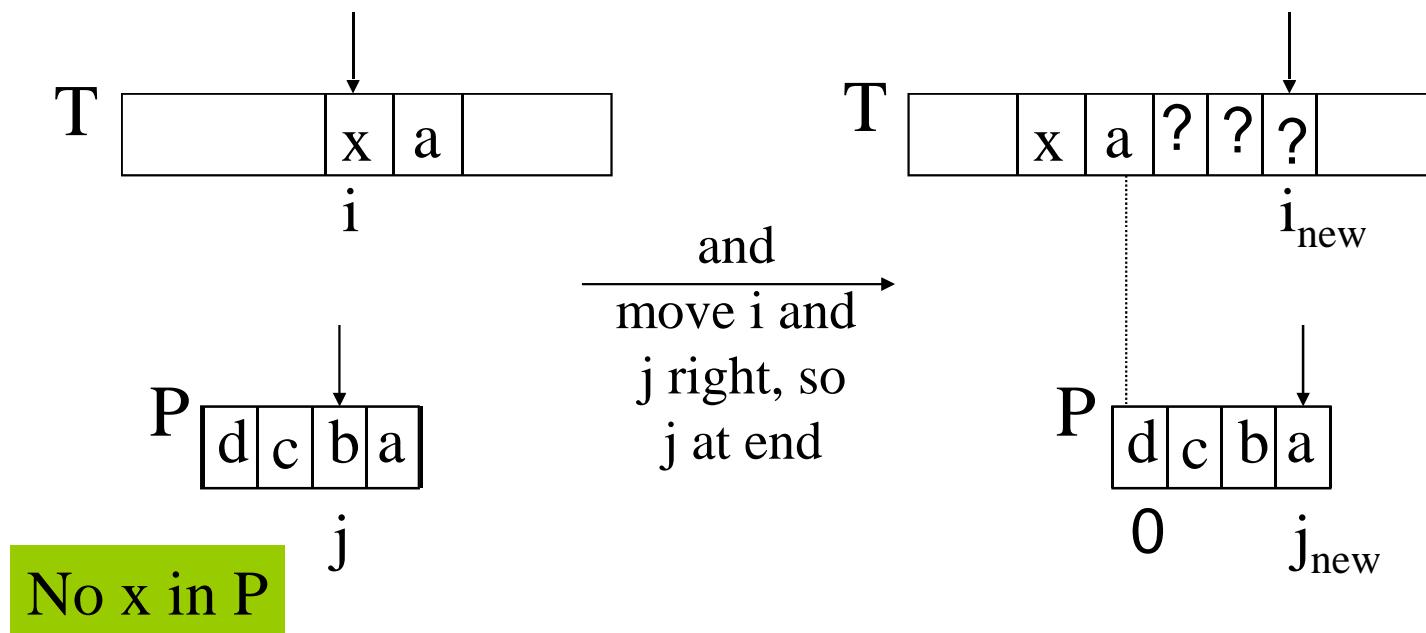
## Case 2

- If  $P$  contains  $x$  somewhere, but a shift right to the last occurrence is *not* possible, then  
*shift P* right by 1 character to  $T[i+1]$ .

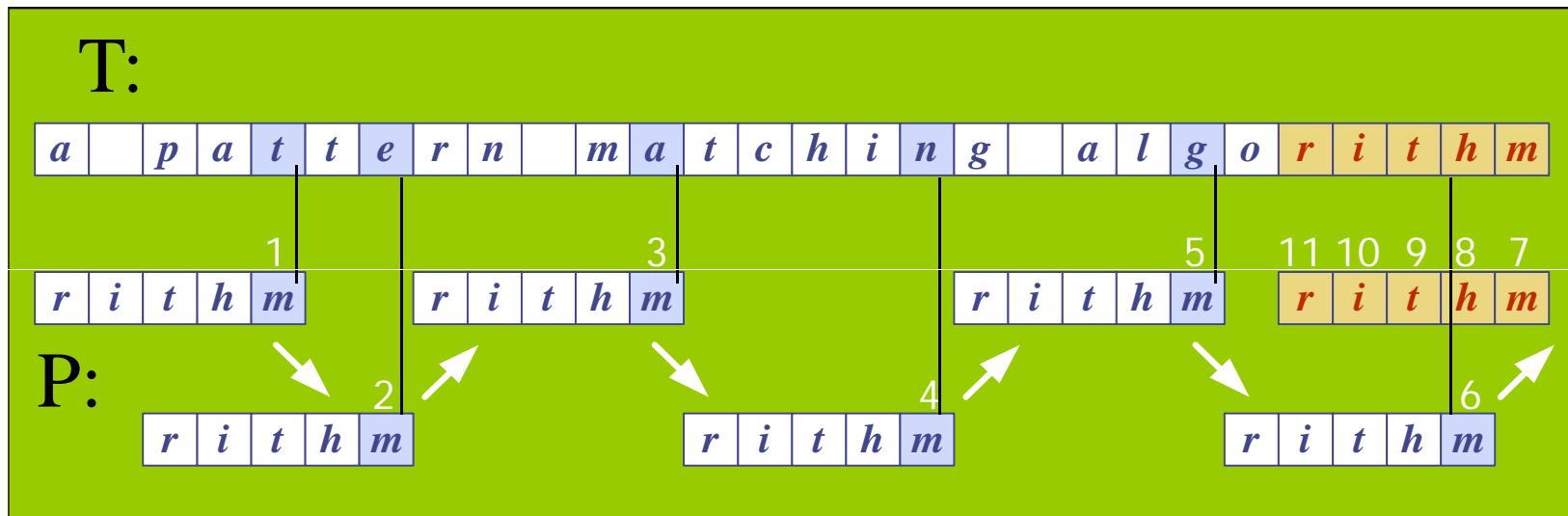


## Case 3

- If cases 1 and 2 do not apply, then *shift P* to align  $P[0]$  with  $T[i+1]$ .



# Boyer-Moore Example (1)



# Last Occurrence Function

- Boyer-Moore's algorithm preprocesses the pattern  $P$  and the alphabet  $A$  to build a last occurrence function  $L()$ 
  - $L()$  maps all the letters in  $A$  to integers
- $L(x)$  is defined as: //  $x$  is a letter in  $A$ 
  - the largest index  $i$  such that  $P[i] == x$ , or
  - -1 if no such index exists

# L( ) Example

- $A = \{a, b, c, d\}$
- $P$ : "abacab"

$P$	a	b	a	c	a	b
	0	1	2	3	4	5



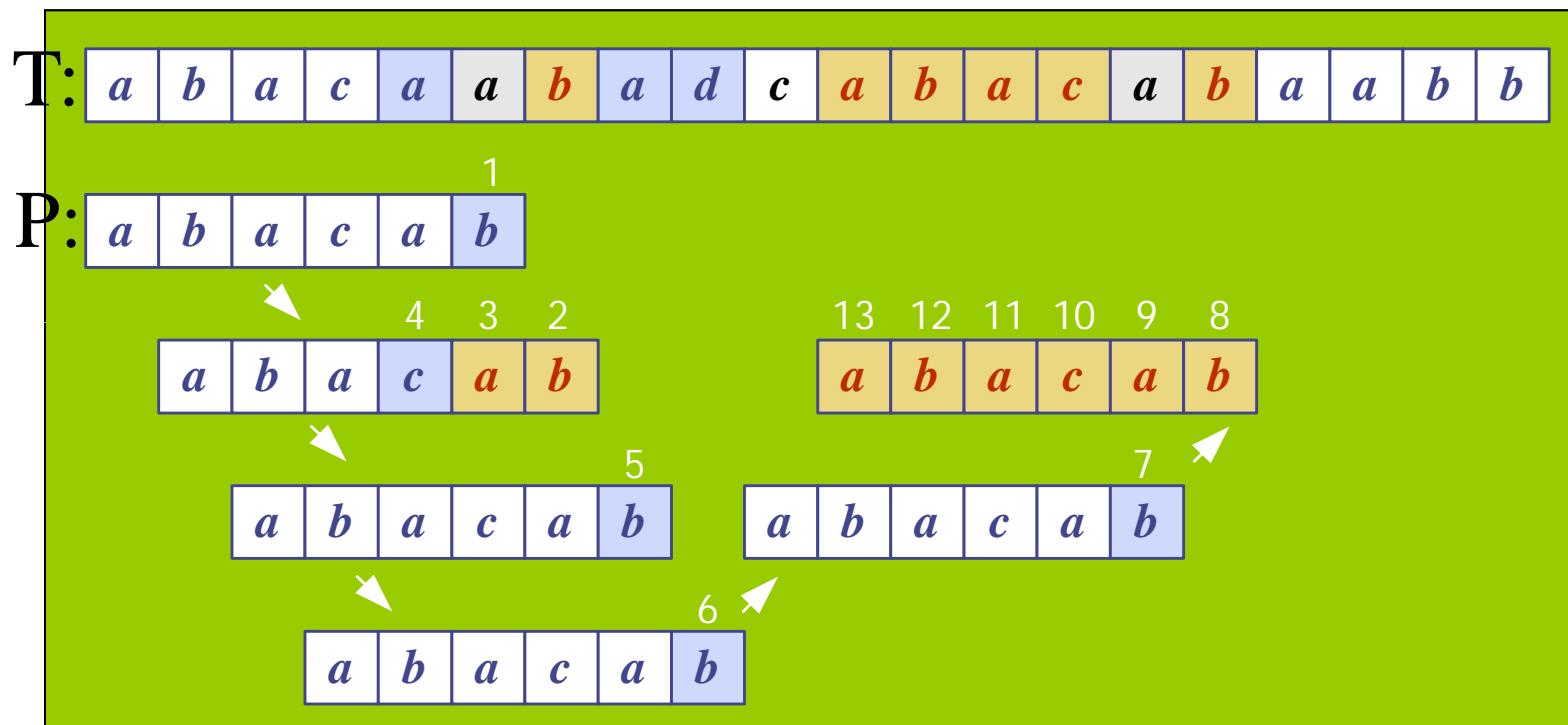
$x$	$a$	$b$	$c$	$d$
$L(x)$	4	5	3	-1

$L()$  stores indexes into  $P[]$

# Note

- In Boyer-Moore code,  $L()$  is calculated when the pattern  $P$  is read in.
- Usually  $L()$  is stored as an array
  - something like the table in the previous slide

# Boyer-Moore Example (2)



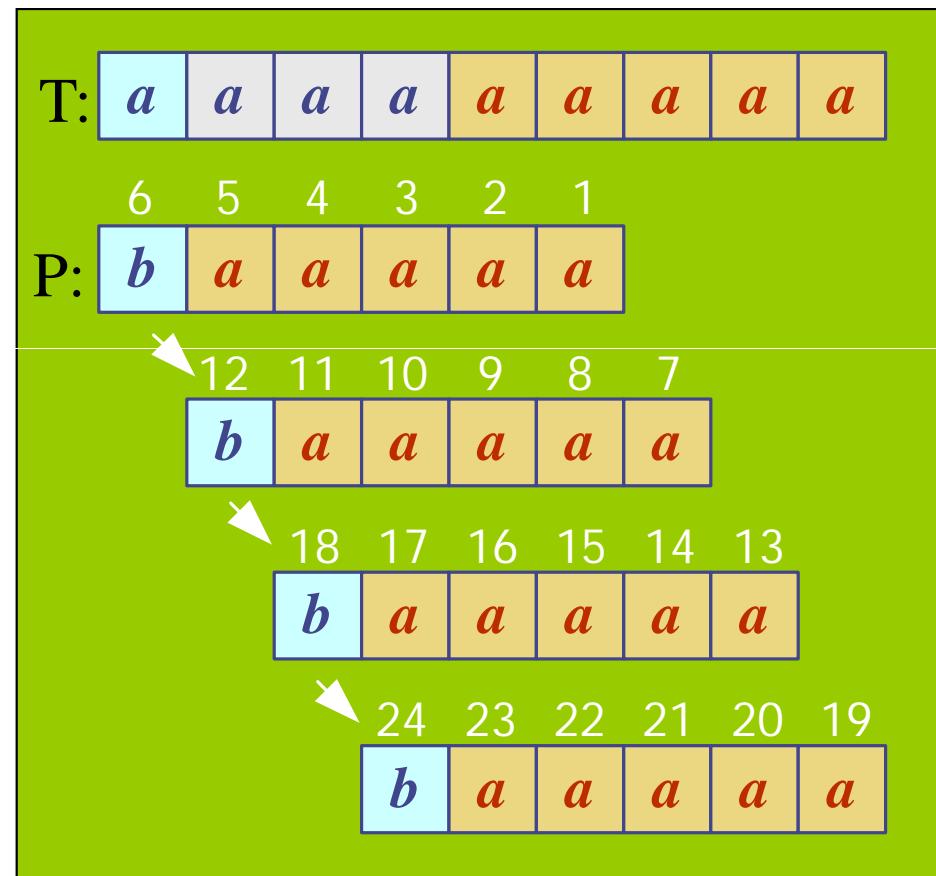
$x$	$a$	$b$	$c$	$d$
$L(x)$	4	5	3	-1

# Analysis

- Boyer-Moore worst case running time is  $O(nm + A)$
- But, Boyer-Moore is fast when the alphabet ( $A$ ) is large, slow when the alphabet is small.
  - e.g. good for English text, poor for binary
- Boyer-Moore is *significantly faster than brute force* for searching English text.

# Worst Case Example

- T: "aaaaaa...a"
- P: "baaaaa"



# Computational Number Theory

Topics:  
Primes, Factoring,  
GCD, Euclid, EEA, mod,  
bigint, RSA, DLOG

# Number-Theoretic Algorithms

## **Topics:**

- **Primes**
- **Modular math**
- **GCD**
- **Extended GCD**
- **Inverse mod**
- **Power mod**
- **Chinese Remainder Theorem**
- **Large Primes**
- **RSA**
- **Discrete Log**

# Prime factorization

1. There are infinite number of primes

Primes = { 2, 3, 5, 7, 11, ... }

2. Every number can be uniquely written as a product of its prime factors:

Eg.  $100 = 4 \cdot 25 = 2 \cdot 2 \cdot 5 \cdot 5$

# Homework: primes.c

Write a program **primes.c** to print the first n  
primes numbers

\$ primes 100

Prime numbers less than 100:

2 3 5 7 11 13 17 19 23 29 31 37 41 43 47 53  
59 61 67 71 73 79 83 89 97

# Homework: write **is-prime.c**, **factor.c**

```
$ is-prime 100  
100 is not a prime  
$ is-prime 229  
229 is a prime
```

```
$ factor 100  
100: 2 2 5 5  
$ factor 101  
101: 101
```

# Modular Math

$$(a \equiv b) \text{ mod } n \quad a \equiv_n b$$

# Modular math

In C, we write:  $5 \% 3 == 2$ , read as 5 mod 3 is 2.

In Math, we write:  $(5 \equiv 2) \text{ mod } 3$ , or  $5 \equiv_3 2$

$$5 \text{ mod } 3 = 2$$

$$3 * 3 \text{ mod } 5 = 4$$

$$3 * 3 * 3 \text{ mod } 5 = 2$$

$$(3 * 3 \text{ mod } 5) * 3 \text{ mod } 5 = 2$$

$$(3^4) \text{ mod } 5 = 1 = ((3^2) \text{ mod } 5)^2 \text{ mod } 5 = 1$$

$$= (9 \text{ mod } 5)^2 \text{ mod } 5 = 1$$

# GCD (Greatest Common Divisor)

## Examples:

1.  $\text{gcd}(10,15)=\text{gcd}(2*5, 3*5) = 5 * \text{gcd}(2, 3) = 5$
2.  $\text{gcd}(20,24)=\text{gcd}(2*2*5, 2*2*2*3) = 2*2*\text{gcd}(5,2*3) = 4$
3.  $\text{gcd}(2,2)=2$

Numbers are called **relatively prime** if their  $\text{gcd}(a,b)$  is 1. E.g.  $\text{gcd}(9,8)=1$

Any two **prime** numbers are **relatively prime**. E.g.  $\text{gcd}(13,17)=1$

# Properties of gcd

1.  $\text{gcd}(a,0)=a$
2.  $\text{gcd}(a,ka)=a$
3.  $\text{gcd}(a,b)=\text{gcd}(b,a)$
4.  $\text{gcd}(a,-b)=\text{gcd}(a,b)$
5.  $\text{gcd}(ka,kb) = k * \text{gcd}(a,b)$
6.  $\text{gcd}(a, b) = \text{gcd}(b, a \bmod b)$
7.  $\text{gcd}(\text{gcd}(a,b),c) = \text{gcd}(a,\text{gcd}(b,c)) = \text{gcd}(a,b,c).$

# Euclid's GCD Algorithm

To compute  $\text{gcd}(a,b)$  where  $a \geq 0$  and  $b \geq 0$ .

Using this property of gcd to write an recursive algorithm:  $\text{gcd}(a, b) = \text{gcd}(b, a \% b)$   
i.e. keep removing multiples until remainder is zero, last multiple is the gcd.

```
int gcd(a, b)
    if (b == 0)
        return a
    else
        return gcd(b, a mod b)
}
```

# GCD Algorithm Example

$\text{gcd}(a, b) = b > 0? \text{gcd}(b, a \% b)) : a$

## **Example**

$\text{gcd}(a=30, b=21)$   
=  $\text{gcd}(a=21, b=9 \text{ is } 30\%21)$   
=  $\text{gcd}(a=9, b=3 \text{ is } 21\%9)$   
=  $\text{gcd}(a=3, b=0 \text{ is } 9\%3)$   
= 3

# Modular math mod n

In modular arithmetic the set of congruence classes relatively prime to a modulus number  $n$ ,

form a group under multiplication called the **multiplicative group of integers modulo  $n$** .

It is also called the group of primitive residue classes **modulo  $n$** .

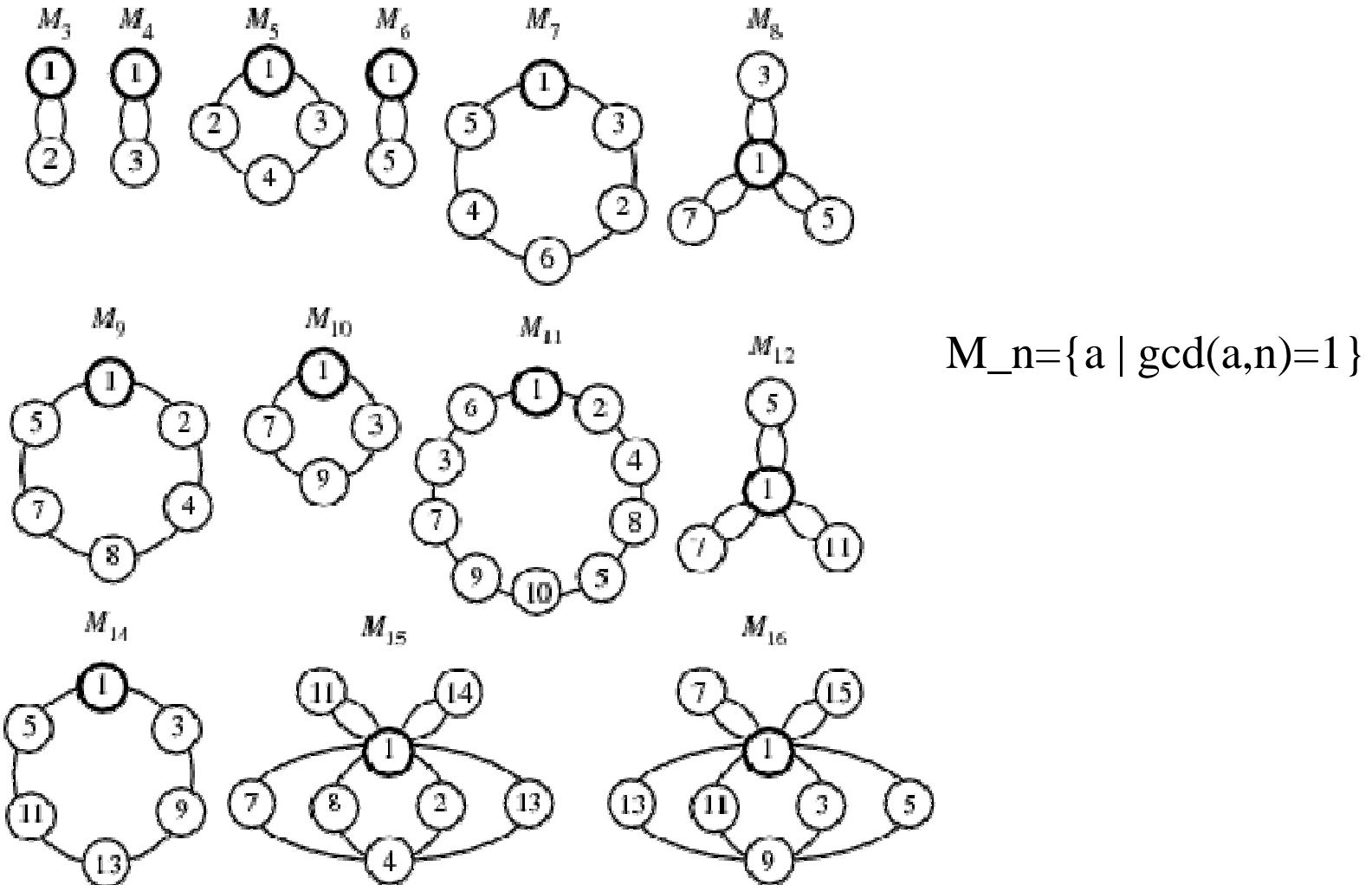
Since  $\gcd(a, n) = 1$  and  $\gcd(b, n) = 1$  implies  
 $\gcd(ab, n) = 1$

the set of classes relatively prime to  $n$  is closed under multiplication.

# Prime, mod 5 multiplicative group

	-----+-----
%5	1 2 3 4
	-----+-----
1	1 2 3 4
2	2 4 1 3
3	3 1 4 2
4	4 3 2 1
	-----+-----

# Mod n group examples



**GCD  
EEA**

**Extended Euclid Algorithm  
(EEA) for GCD**

# Extended Euclid Algorithm (EEA)

```
function eea(a, b)
    if b = 0:
        return(a, 1, 0)

    (d', x', y') = eea(b, a mod b)
    (d, x, y) = (d', y', x' - ⌊a/b⌋ y')
    return (d, x, y)
```

$$\rightarrow \gcd(a, 0) = a = 1 \cdot a + 0 \cdot b$$

$$\begin{aligned} \rightarrow d' &= bx' + (a \text{ mod } b) y' \\ d &= bx' + (a - \lfloor a/b \rfloor b) y' \\ &= ay' + b(x' - \lfloor a/b \rfloor y') \\ &= ax + b y \end{aligned}$$

# Extended Euclid Algorithm (EAA)

eea(a,b) returns [g, x, y]

Where:  $g = \gcd(a,b)$  and  $g = a^*x + b^*y$

## Example:

$\gcd(a=30, b=21)$  is  $3 = 30^*5 + 21^*(-7)$ .  
 $\text{eea}(a=30,b=21)$  is  $[g=3, x=5, y=-7]$

# Compute gcd(181, 39)

$181 = 39 * 4 + 25$	$(25 = 181 - 39*4)$
$39 = 25 * 1 + 14$	$(14 = 39 - 25)$
$25 = 14 * 1 + 11$	$(11 = 25 - 14)$
$14 = 11 * 1 + 3$	$(3 = 14 - 11)$
$11 = 3 * 3 + 2$	$(2 = 11 - 3*3)$
$3 = 2 * 1 + 1$	$(1 = 3 - 2*1)$
$2 = 2 * 1 + 0$	
gcd is last nonzero remainder 1.	

use back-substitution to  
find x, y such that:

$$181 * x + 39 * y = 1 = \text{gcd}$$

$$\begin{aligned} \text{gcd } 1 &= 3 - 2 \quad \dots \text{subst 2} \\ &= 3 - (11 - 3*3) \\ &= -11 + 4*3 \quad \dots \text{subst 3} \\ &= -11 + 4*(14 - 11) \\ &= 4*14 - 5*11 \quad \dots \text{subst 11} \\ &= 4*14 - 5*(25 - 14) \\ &= -5*25 + 9*14 \quad \dots \text{subst 14} \\ &= -5*25 + 9*(39 - 25) \\ &= 9*39 - 14*25 \quad \dots \text{subst 25} \\ &= 9*39 - 14*(181 - 39*4) \\ &= 65*39 - 14*181 \end{aligned}$$

hence x = 65, y = -14.

# Exercise find gcd of

Find the  $\text{gcd}(55, 700)$  and write the gcd as an integer combination of 55 and 700.

# Solution to gcd(55, 700)

$$700 = 55 \cdot 12 + 40 \quad (40=700\%55)$$

$$55 = 40 \cdot 1 + 15$$

$$40 = 15 \cdot 2 + 10$$

$$15 = 10 \cdot 1 + 5 \quad .. \text{ hence gcd}(55, 700) = 5$$

$10 = 5 \cdot 2 + 0$  .. zero means stop.

$$\begin{aligned} \gcd 5 &= 15 - 10 \\ &= 15 - (40 - 15 \cdot 2) \\ &= -40 + 3 \cdot 15 \\ &= -40 + 3(55 - 40) \\ &= 3 \cdot 55 - 4 \cdot 40 \\ &= 3 \cdot 55 - 4(700 - 55 \cdot 12) \\ &= 51 \cdot 55 - 4 \cdot 700 \end{aligned}$$

back substituting to find x, y

x        y

Hence  $\gcd(55, 700) = 5 = 51 \cdot 55 - 4 \cdot 700$ .

# Time complexity

Worst case:

gcd of successive Fibonacci numbers.

$\text{gcd}(F_n, F_{n+1})$  .. Takes  $\log(n)$  time

See Cormen

# Exercises

1. Compute eea(7, 3)
2. Compute eea(31, 20)

# Inverse mod n

# Inverse mod n

$$a * a^{-1} = 1 \text{ mod } n$$

If a and b are inverses mod n:

$$a * b = 1 \text{ mod } n$$

$$1/a = b \text{ mod } n$$

$$1/b = a \text{ mod } n$$

**Example:**  $5 * 2 = 10 = 1 \text{ mod } 9$

so 5 is inverse of 2 mod 9.

# Inverse modulo primes

All non-zero b have an **inverse** (mod prime p)  
where b is not a multiple of p.

For example, inverse mod 5

$$1 * 1 = 1 \text{ mod } 5$$

$$2 * 3 = 1 \text{ mod } 5$$

$$3 * 2 = 1 \text{ mod } 5$$

$$4 * 4 = 1 \text{ mod } 5$$

# Example, inverse mod 7

The multiplicative inverse of 1 is 1,  
the multiplicative inverse of 2 is 4 (and vice versa)  
the multiplicative inverse of 3 is 5 (and vice versa)  
and the multiplicative inverse of 6 is 6.

Note that 0 doesn't have a multiplicative inverse. This corresponds to the fact in ordinary arithmetic that 1 divided by 0 does not have an answer.

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

# Computing inverse mod n

Given  $a$  and  $n$ , with  $\gcd(a, n) = 1$ ,

Finding  $x$  satisfying  $ax \equiv 1 \pmod{n}$

(finding the inverse  $x$  of  $a$  mod  $n$ )

is the same as solving for  $x$  in  $ax + ny = 1$ .

We can use  $\text{eea}(a,n) = (g,x,y)$ , to get  $x$ .

## Not all numbers have inverse mod n

if  $\gcd(a,n)=1$ , then a has an inverse mod n

The inverse x of a mod n can be found from  
eea(a,n):

Let  $\gcd(a,n)= 1$

$$a * x + n * y = 1$$

$$a * x + 0 = 1, \text{ mod } n$$

$$a * x = 1, \text{ mod } n$$

$$a^{-1} \equiv_n x$$

# Power Mod

# Power Mod, compute $a^p \bmod n$

Brute force:

```
int power_mod_(a,p)
    int z = 1;
    for(i = 1..p)
        z = z * a;
    return z % n
```

Problems:

1. z overflows
2. Slow.

# Power mod, compute $a^p \bmod n$

Brute force:

```
for(i=1..p)
```

```
    z = z * a mod n // keep z small.
```

Problems:

1. still slow, e.g. when  $p$  is a 20 digit number, billion billion iterations.

# Fast Power mod O(log(p))

```
// Return: bp mod n, complexity O(log2(p)).  
// Usage: pow-mod(2,100, 5) for 2100 mod 5  
int pow-mod(int b, int p, int n){  
    int r = 1;  
    while (p > 0) {  
        if (p & 1) // same as p % 2, odd(p)  
            r = (r * b) % n;  
        p >>= 1; // same as: p=p/2, shift-right  
        b = (b * b) % n;  
    }  
    return r;  
}
```

# Chinese Remainder Theorem

CRT

# CRT (chinese remainder theorem)

Problem: solve for  $x$  in

$$x = a_1 \bmod m_1$$

$$x = a_2 \bmod m_2$$

$$x = a_3 \bmod m_3$$

...

Where  $m_1, m_2, m_3, \dots$  are mutually relatively prime, ie.  $\gcd(m_1, m_2) = 1, \gcd(m_1, m_3) = 1, \dots$

# CRT

Compute  $M = m_1 * m_2 * m_3 \dots$

Compute  $M_j = M/m_j$

Note:  $M_j$  is an integer and  $\gcd(M_j, m_j)=1$

Compute inverse  $b_j$  of  $M_j$  such that:

$$M_j * b_j = 1 \pmod{m_j}$$

$$M_j * b_j = 0 \pmod{m_i} \quad (i \neq j).$$

Compute  $x = \sum_{j=1..}(M_j * b_j * a_j) \pmod{M}$ .

# CRT exercise

From *Number Theory* [Niven and Zuckerman]

Example: find the least positive integer  $x$ :

$$x \equiv 5 \pmod{7},$$

$$x \equiv 7 \pmod{11},$$

$$x \equiv 3 \pmod{13}.$$

# CRT solution

Given  $x = 5 \pmod{7}$ ,  $x = 7 \pmod{11}$ ,  $x = 3 \pmod{13}$ .

So  $a_1=5$ ,  $a_2=7$ ,  $a_3=3$ ;  $m_1=7$ ,  $m_2=11$ ,  $m_3=13$

$$M = 7 \cdot 11 \cdot 13 = 1001$$

- $M_1 = M/m_1 = 1001/7 = 143$
- $M_2 = M/m_2 = 1001/11 = 91$
- $M_3 = M/m_3 = 1001/13 = 77$
- $b_1 = \text{inv}(M_1) \pmod{m_1} = \text{inv}(143) \pmod{7} = 5 = -2$
- $b_2 = \text{inv}(91) \pmod{11} = 4$
- $b_3 = \text{inv}(77) \pmod{13} = 12 = -1$

$$x = M_1 \cdot b_1 \cdot a_1 + M_2 \cdot b_2 \cdot a_2 + M_3 \cdot b_3 \cdot a_3$$

$$x = 143 \cdot (-2) \cdot 5 + 91 \cdot 4 \cdot 7 + 77 \cdot (-1) \cdot 3 = 887, \text{ also}$$

$$x = (143 \cdot 5 \cdot 5 + 91 \cdot 4 \cdot 7 + 77 \cdot 12 \cdot 3) \% 1001 = 887$$

# CRT program

M=1001

$m_0 = 7, M_0 = 143 = M/(m_0 = 7), b_0 = -2 = \text{inv}(M_0 = 143) \bmod (m_0 = 7)$ .

$m_1 = 11, M_1 = 91 = M/(m_1 = 11), b_1 = 4 = \text{inv}(M_1 = 91) \bmod (m_1 = 11)$ .

$m_2 = 13, M_2 = 77 = M/(m_2 = 13), b_2 = -1 = \text{inv}(M_2 = 77) \bmod (m_2 = 13)$ .

crtin=887

887 mod 7 is 5

887 mod 11 is 7

887 mod 13 is 3

crt\_inv:  $+(5*-2*143)+(7*4*91)+(3*-1*77)= 887$

# More Primes

# Euler phi $\varphi$ function

$\phi(n)$  = count of numbers relatively prime to n,  
these form a multiplicative group  $M_n$ .

Examples:

$$\phi(6)=2 \dots \{1,5\}$$

$$\phi(5)=4 \dots \{1,2,3,4\}$$

$$\phi(p) = p-1, \text{ if } p \text{ is a prime} \dots \{1,2,\dots,p-1\}$$

# Perfect number

A **perfect number** is a positive integer that is equal to the sum of its proper positive divisors. Eg.  $6 = 1+2+3$

Q. Can you find more perfect numbers?

# Primality testing

Primality testing of  $n$  is easier than factoring  $n$ .  
Factoring  $n$  is hard.

Worse case  $n$  will have 2 factors near  $\sqrt{n}$ ,  
because  $n = \sqrt{n} * \sqrt{n}$ ,

Brute force will try to divide  $n$  by each factor, from 1  
to  $\sqrt{n}$ .

# Is Prime, naive algorithm

```
IsPrime(n) {  
    for f in 2..sqrt(n) {  
        if (n mod f == 0) // f divides n?  
            return false; // n is f * p/f  
    }  
    return true; // n is a prime.  
}
```

Complexity:  $O(\sqrt{n})$  .. number of bits( $n$ )/2s

Requires BigInts if  $n > 2^{\text{wordsize}}$

# Euler and Fermat's theorem

Euler's theorem:  $a^{\phi(n)} \equiv 1 \pmod{n}$ .

Proof from Lagrange's theorem about groups.

Fermat's theorem:

If  $p$  is a prime, then  $\phi(p) = p-1$ .

Fast test:  $a^{p-1} \equiv 1 \pmod{p}$

So if a number passes Fermat's test, it is likely to be a prime.

# Pseudo-prime

```
Pseudo-prime(n) {  
    if(  $2^{n-1} \neq 1 \pmod n$  )  
        return composite; // sure.  
    else // pseudo prime.  
        return almost prime  
}
```

Exceptions: *Carmichael numbers*, rare composite numbers that pass above test.

# Rabin Miller primality test

```
Rabin-miller-pseudo-prime(n){  
    for j = 1 .. s { // Try s times.  
        a = random(1..n-1)  
        if( a^(n-1) != 1 mod n ) { // fails fermat's test.  
            // also detect squareroots of 1.  
            return composite;  
        }  
    }  
    return almost prime; // error rate is 1/2^s  
}
```

# Pollard Rho

If the function prints a factor, it is correct.

If it doesn't print anything, it may be still searching.

Instead of  $\sqrt{n}$  complexity, *pollard-rho* one has complexity  $\sqrt{\sqrt{n}}$  .. (birthday paradox).

Fast when there are small factors.

It factored the 9<sup>th</sup> Fermat number:  $F_8 = 1238926361552897$  in 2 hours.

See [https://en.wikipedia.org/wiki/Pollard%27s\\_rho\\_algorithm](https://en.wikipedia.org/wiki/Pollard%27s_rho_algorithm)

<https://www.cs.colorado.edu/~srirams/courses/csci2824-spr14/pollardsRho.html>

# Factoring with *Pollard Rho* heuristic

**Inputs:**  $n$ , the integer to be factored; and  $f(x)$ , a pseudo-random function modulo  $n$ , e.g.  
 $f(x) = (x^2 + 1) \text{mod } n$ .

**Output:** a non-trivial factor of  $n$ , or failure.

$x \leftarrow 2, y \leftarrow 2; d \leftarrow 1$

While  $d = 1$ :

$x \leftarrow f(x)$  #  $x, y$  are two consecutive rand num

$y \leftarrow f(f(y))$

$d \leftarrow \text{GCD}(|x - y|, n)$

If  $d == n$  return failure.

else return  $d$  // found a factor of  $n$

# RSA

# RSA steps

1. Choose two large primes, p and q, kept secret.  
(generate large random number and fast test for primes).
  1. Compute  $n = p * q$ , assume factoring n is hard.
  2. Compute  $\phi(n) = (p - 1)(q - 1)$
  3. Pick prime  $e$ , such that  $\gcd(e, \phi(n)) = 1$ .
  4. Determine  $d = e^{-1} \pmod{\phi(n)}$
  5. Publish *public-key* ( $e, n$ )
  6. Hide *private-key* ( $d, n$ )

# RSA Encrypt and Decrypt

1. Encrypt plaintext= $m$  using public-key to:  
Ciphertext =  $c = m^e \pmod{n}$ .
2. Decrypt  $c$  using private key:  $m = c^d \pmod{n}$ .

# RSA example

1. Choose two large prime,  $p = 61$  and  $q = 53$ .
2.  $n = p * q = 61 * 53 = 3233$
3.  $\varphi(p * q) = (61 - 1)*(53 - 1) = 3120$ .
4. Let  $e = 17$ ,  $\gcd(e, n)=1$ .  
17 or any small prime will do.
5. Compute  $d=2753$  ( $17*d=1 \bmod \varphi(3233)$ )  
(Hard without knowing factors of  $n$ ).
6. public key is ( $n = 3233$ ,  $e = 17$ ),  
encryption function is  $m^{17}(\bmod 3233)$ .  
(using powermod, fast)
7. private key is ( $n = 3233$ ,  $d = 2753$ )  
decryption function is  $c^{2753}(\bmod 3233)$ .

# RSA encrypt/decrypt:

1. Encrypt ‘A’ = ascii 65,  
using ( $e=17, n=3233$ )

$$c = 65^{17} \pmod{3233} = 2790.$$

1. Decrypt 2790,  
using ( $d=2753, n=3233$ )

$$m = 2790^{2753} \pmod{3233} = 65, \text{ to ascii, ‘A’}.$$

# Discrete Log & Diffie Hellman

# Discrete Logarithms (dlog) Definition

Discrete logarithms are logarithms defined with regard to *multiplicative cyclic groups*.

If  $G$  is a *multiplicative cyclic group* and  $g$  is a *generator* of  $G$ , then all  $h$  in  $G$  can be written as:

1.  $h = g^x$ , for some  $x$  in  $G$  .. Easy, powermod
2.  $x = \text{dlog}_g(h)$  .. Hard

## dlog example

2 is a generator of the group  $Z_5^*$

$$2^4 = 16 \% 5 \equiv_5 1$$

$$\text{dlog}_{2 \text{ in } Z_5}(1) = 4$$

# Diffie hellman

Diffie hellman is based on the principle that computing  $d\log$  is hard.

Given  $g$  and  $g^a$ , it is hard to compute  $a$ .

## Diffie Hellman: shared-secret over insecure network

Alice and Bob want to come up with a shared-secret password.  
Alice and Bob agree on:  $g$  and modulus  $p$  (hacker Catie sees  $g$ ,  $p$ ).

Alice picks secret:  $a$ .

Alice computes:  $A \equiv_p g^a$

Alice sends  $A$  to Bob.

Bob picks secret:  $b$ .

Bob computes:  $B \equiv_p g^b$

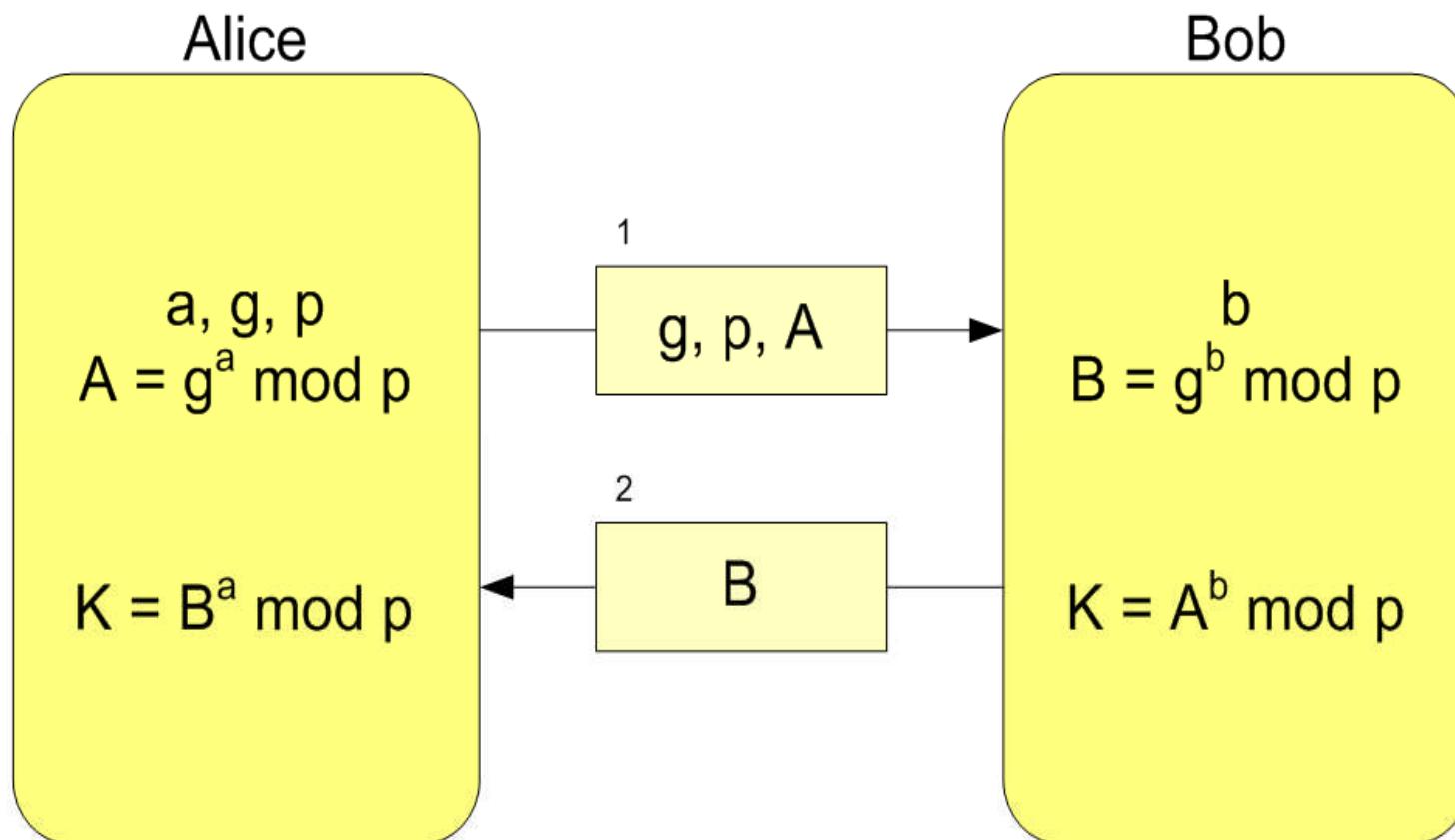
Bob sends  $B$  to Alice.

Alice computes:  $K \equiv_p B^a = (g^b)^a$

Bob computes:  $K \equiv_p A^b = (g^a)^b$

Alice and Bob use  $K$  as their *common shared-secret password*.  
MITM: Catie knows  $A$  and  $B$ ,  $g$ ,  $p$  but cannot compute  $a$ ,  $b$  or  $K$ .

# Diffie Hellman



$$K = A^b \text{ mod } p = (g^a \text{ mod } p)^b \text{ mod } p = g^{ab} \text{ mod } p = (g^b \text{ mod } p)^a \text{ mod } p = B^a \text{ mod } p$$

# Diffie Hellman example

Alice and Bob choose:  $p=23$  (prime),  $g=11$  (generator).

Alice chooses secret  $a=6$ , computes  $A=g^a=11^6 \equiv 9 \pmod{23}$

Bob chooses secret  $b=5$ . computes  $B=g^b=11^5 \equiv 5 \pmod{23}$ .

Alice sends Bob:  $A=9$

Bob sends Alice:  $B=5$ .

Alice computes:  $K=B^a=5^6 \equiv 8 \pmod{23}$ .

Bob computes:  $K=A^b=9^5 \equiv 8 \pmod{23}$ .

Now only they both share a secret  $K$ .

# Diffie Hellman example

## Diffie-Hellman Key Exchange



Alice



Bob

Bob and Alice know and have the following :  
 $p = 23$  (a prime number)  $g = 11$  (a generator)

Alice chooses a secret random number  $a = 6$

Alice computes :  $A = g^a \bmod p$   
 $A = 11^6 \bmod 23 = 9$

Alice receives  $B = 5$  from Bob

Secret Key =  $K = B^a \bmod p$

$K = 5^6 \bmod 23 = 8$

Bob chooses a secret random number  $b = 5$

Bob computes :  $B = g^b \bmod p$   
 $B = 11^5 \bmod 23 = 5$

Bob receives  $A = 9$  from Alice

Secret Key =  $K = A^b \bmod p$

$K = 9^5 \bmod 23 = 8$

The common secret key is : 8

N.B. We could also have written :  $K = g^{ab} \bmod p$

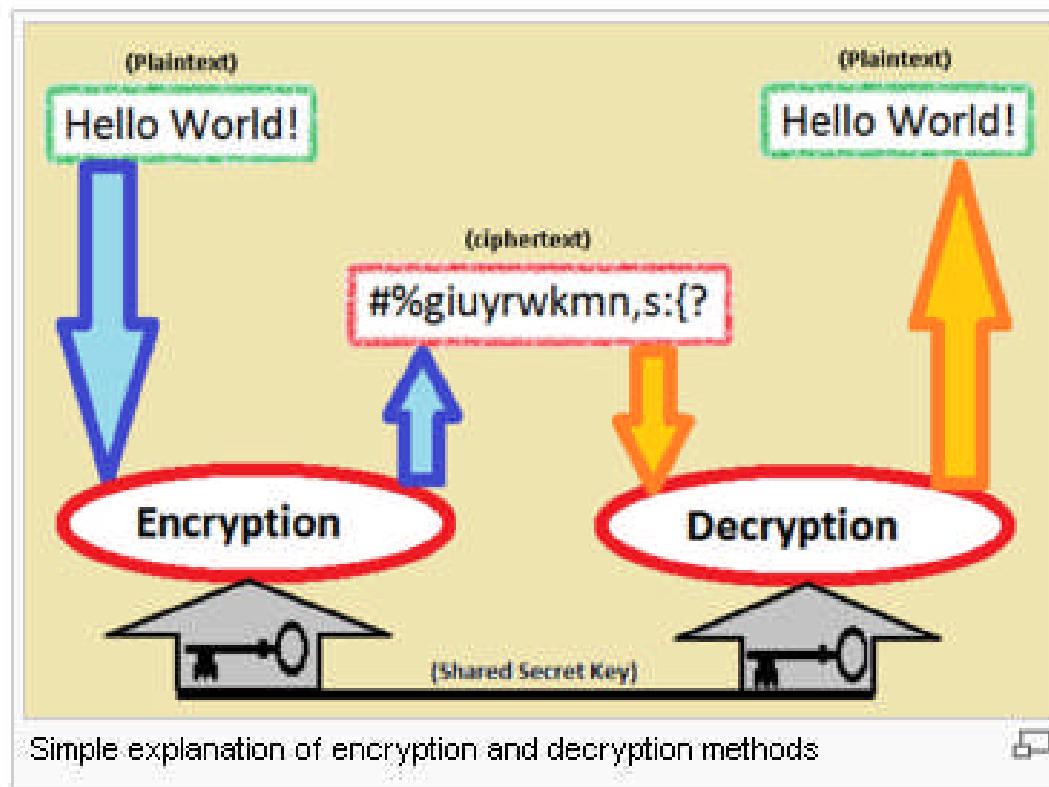
# Applications of Data Encryption

- SSL
- RSA
- Diffie Hellman

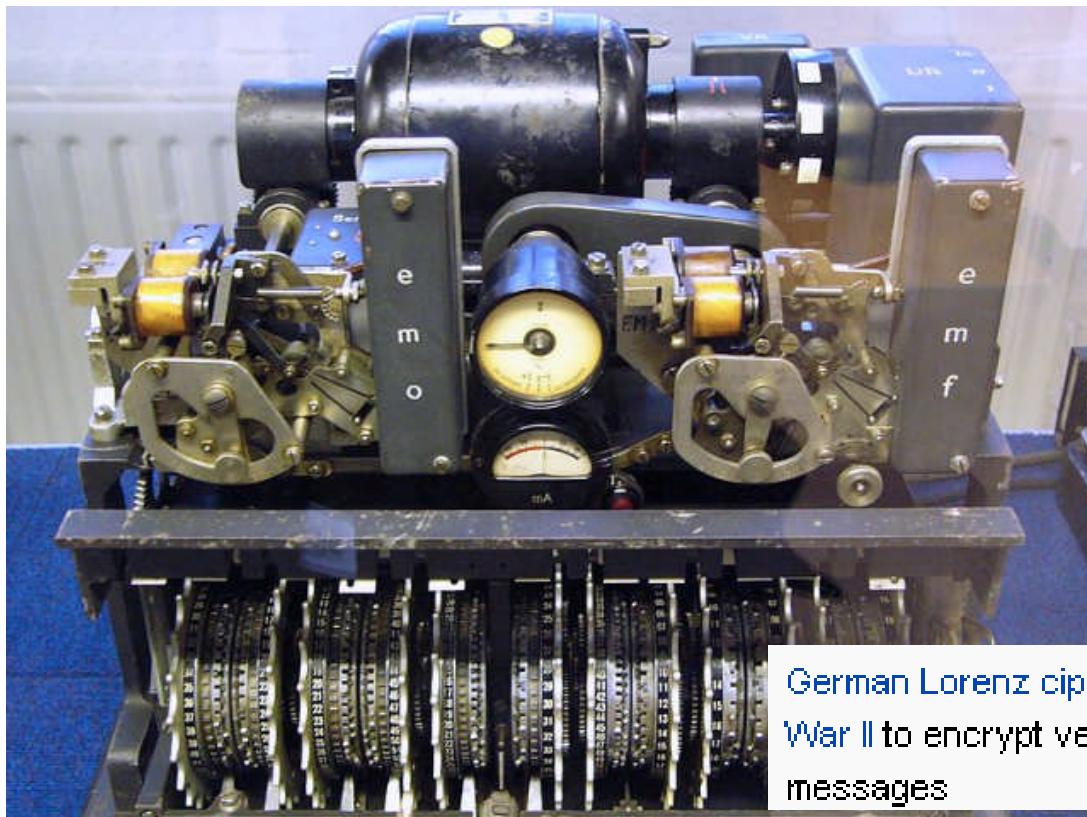
# Protecting data with Encryption

- We need to lock the data
- Make sure it is not tampered
- Make sure it available only to the recipients
- Make sure it can be used only once at the right time and place and person.

# Encryption



# Encryption before electronic computers



German Lorenz cipher machine, used in [World War II](#) to encrypt very-high-level [general staff](#) messages

# Symmetric key encryption

- Symmetric key encryption uses one key, called secret key - for both encryption and decryption. Users exchanging data must keep this key secret. Message encrypted with a secret key can be decrypted only with the same secret key.
- Examples: [DES](#) - 64 bits, [3DES](#) - 192 bits, [AES](#) - 256 bits, IDEA - 128 bits, Blowfish, Serpent

# Symmetric vs Public key encryption

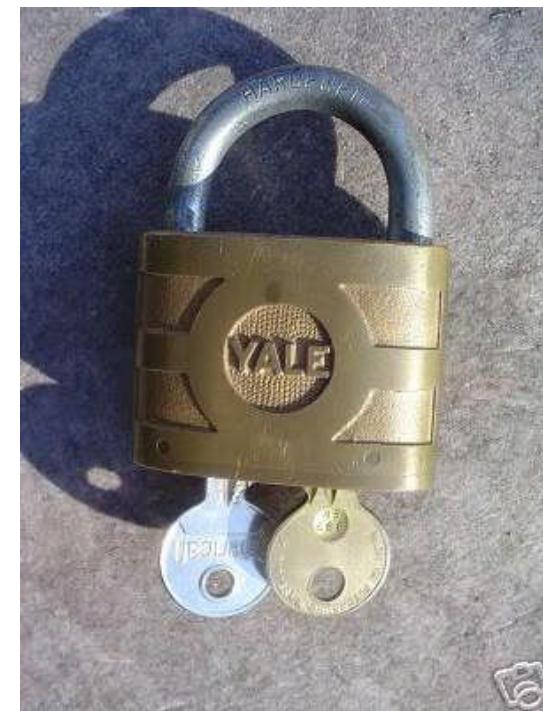
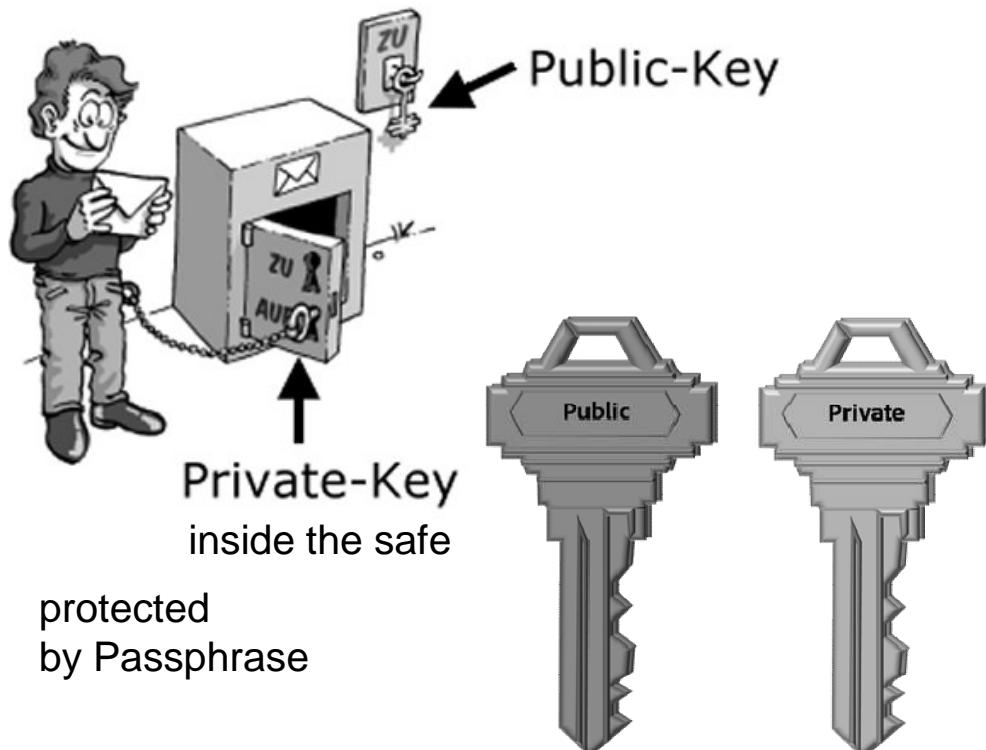


**A) Secret key (symmetric) cryptography.** SKC uses a single key for both encryption and decryption.

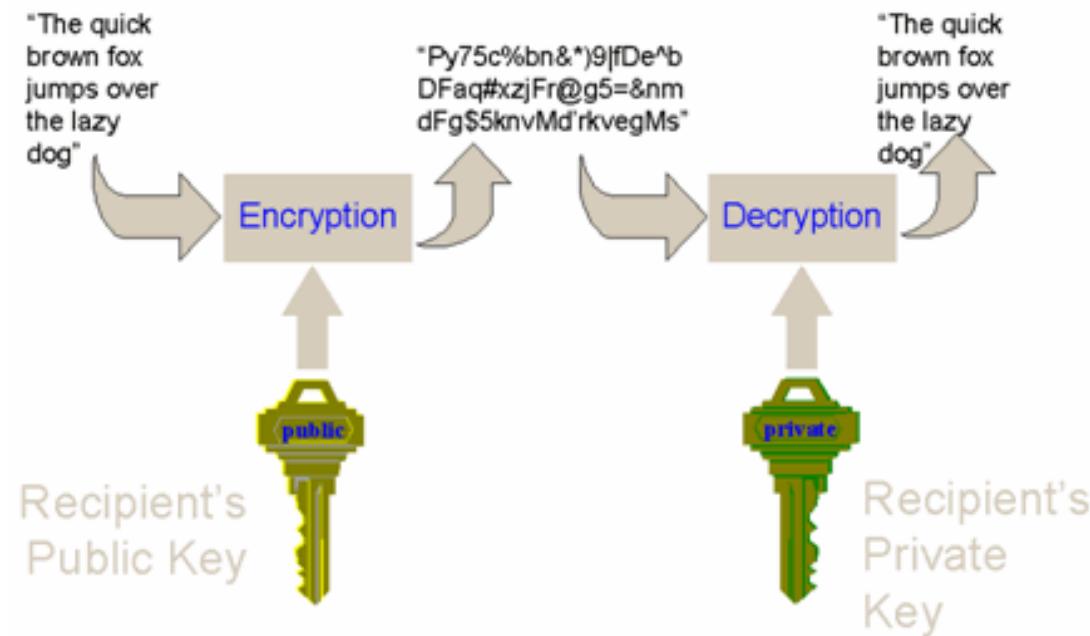


**B) Public key (asymmetric) cryptography.** PKC uses two keys, one for encryption and the other for decryption.

# Two keys per lock



# Public key cryptography



# Public key Cryptography

- The two main branches of public key cryptography are:
- **Public key encryption**: a message encrypted with a recipient's public key cannot be decrypted by anyone except by a matching private key. This is used for confidentiality.
- Digital signatures: a message signed with a sender's private key can be verified by anyone using sender's public key, thereby verifying the sender and message is untampered.

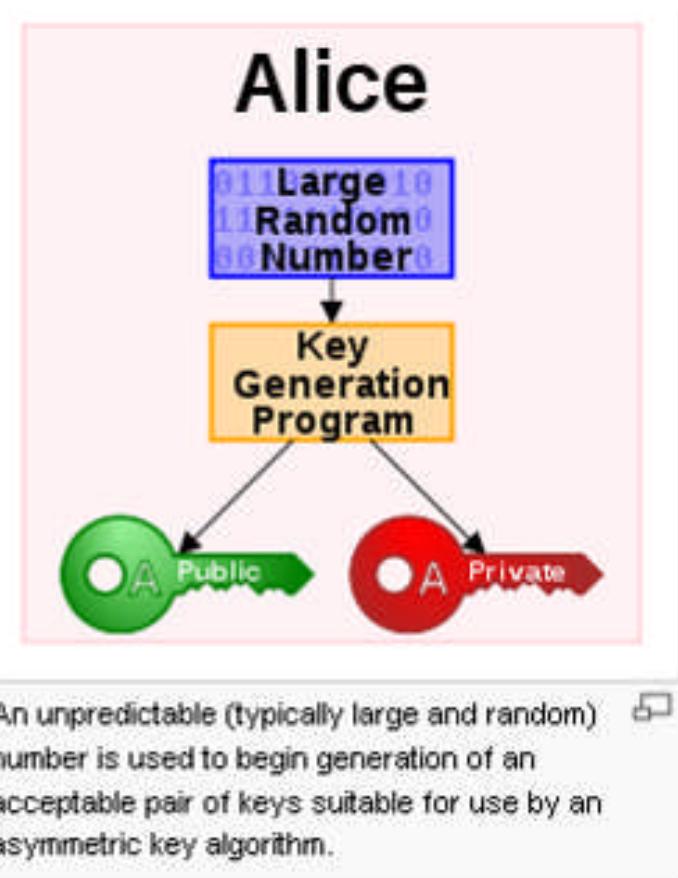
# Mailbox and wax seal analogy

- An analogy to public-key encryption is that of a locked [mailbox](#) with a mail slot. The mail slot is exposed and accessible to the public; its location (the street address) is in essence the public key.
- Anyone knowing the street address can go to the door and drop a written message through the slot; however, only the person who possesses the key can open the mailbox and read the message.
- An analogy for digital signatures is the sealing of an envelope with a personal [wax seal](#). The message can be opened by anyone, but the presence of the seal authenticates the sender.

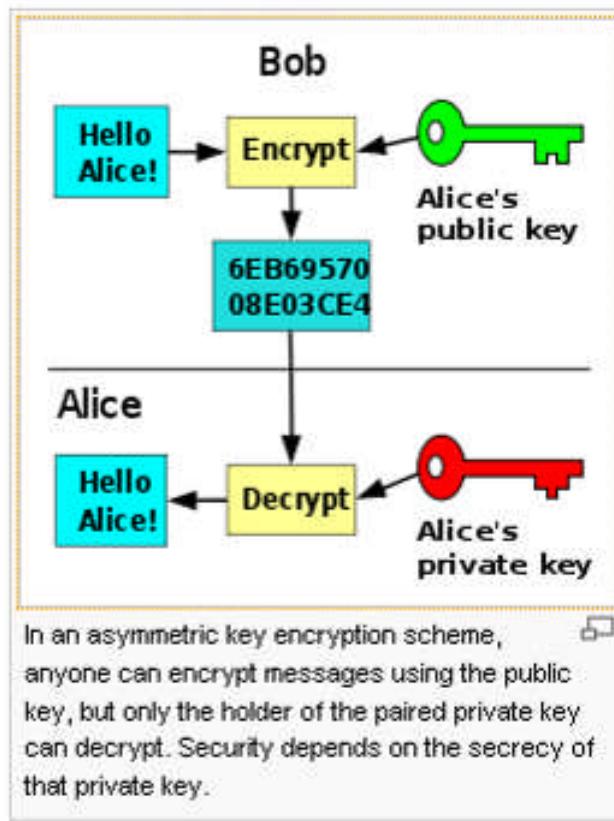
# Web of trust

- A central problem for use of public-key cryptography is confidence (ideally proof) that a public key is correct, belongs to the person or entity claimed (i.e., is 'authentic'), and has not been tampered with or replaced by a malicious third party.
- The usual approach to this problem is to use a public-key infrastructure (PKI), in which one or more third parties, known as certificate authorities, certify ownership of key pairs. Another approach, used by PGP, is the "web of trust" method to ensure authenticity of key pairs.

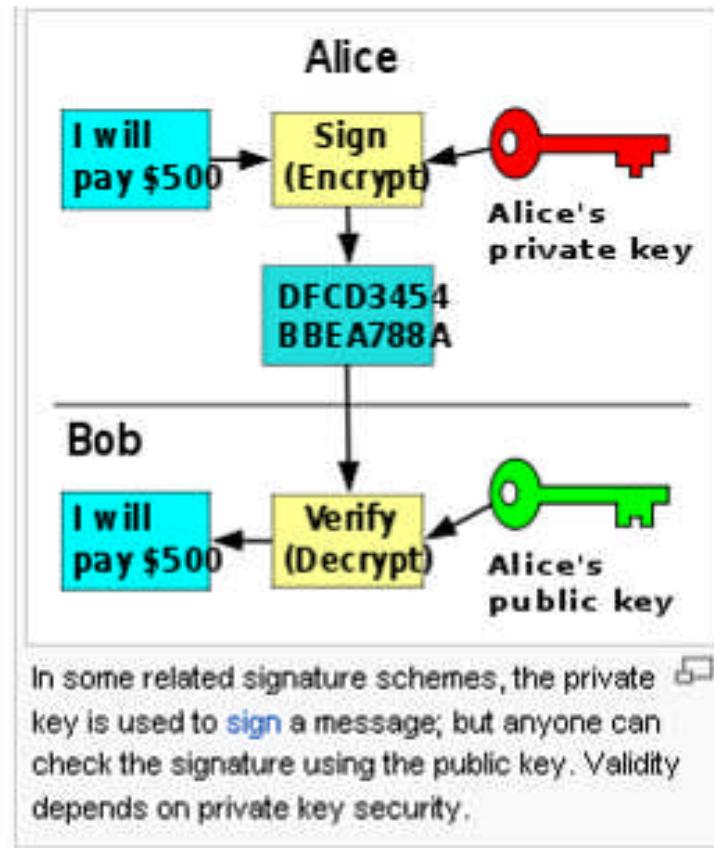
# Alice makes a public key for locking



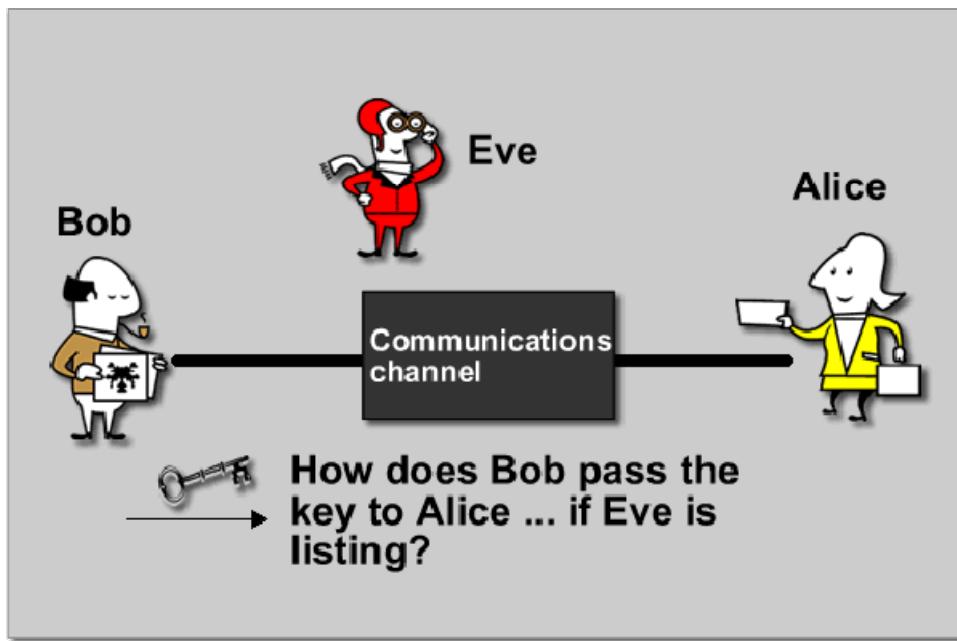
# PK Encryption: Anyone can lock, only Alice can unlock



# PK Signing: Only Alice can lock anyone can unlock



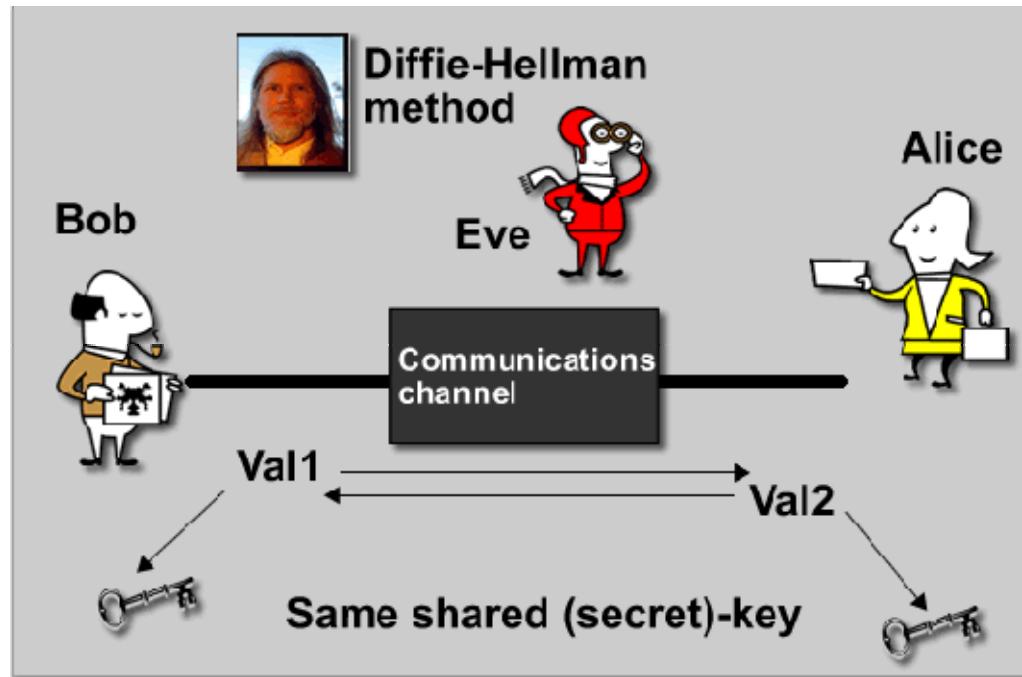
# Key exchange over insecure network

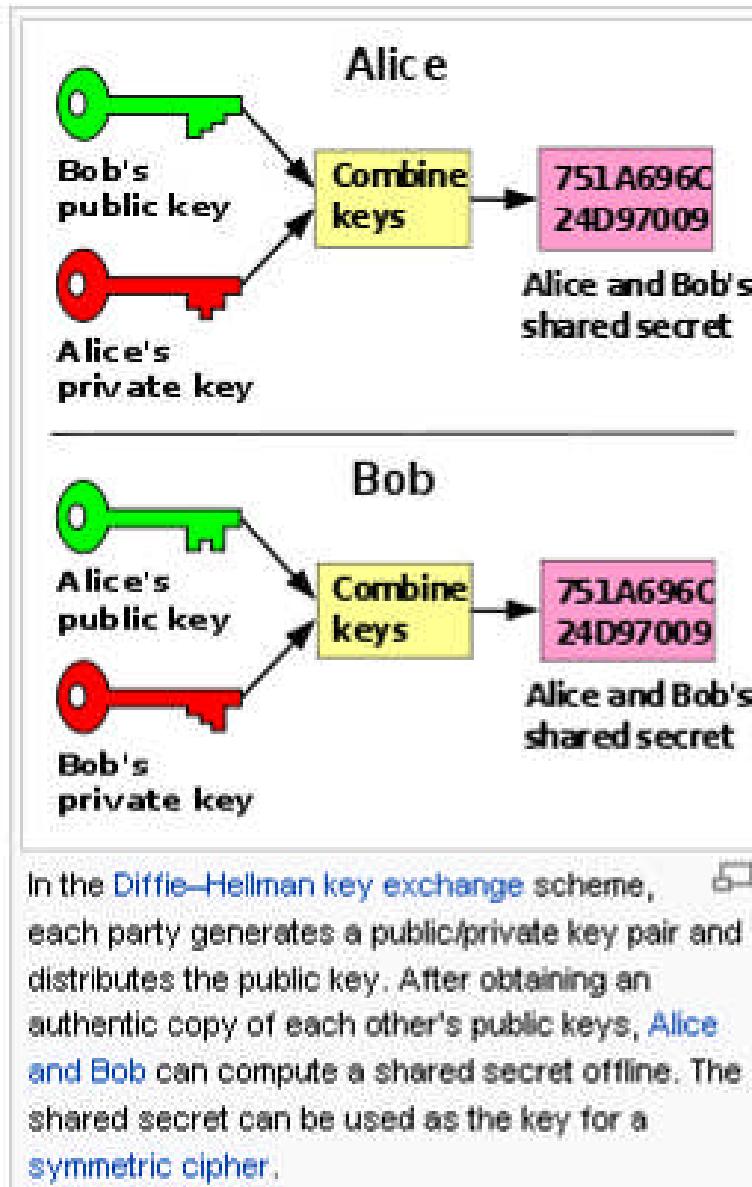


# Merkle Hellman and Diffie



# Diffie Hellman key exchange





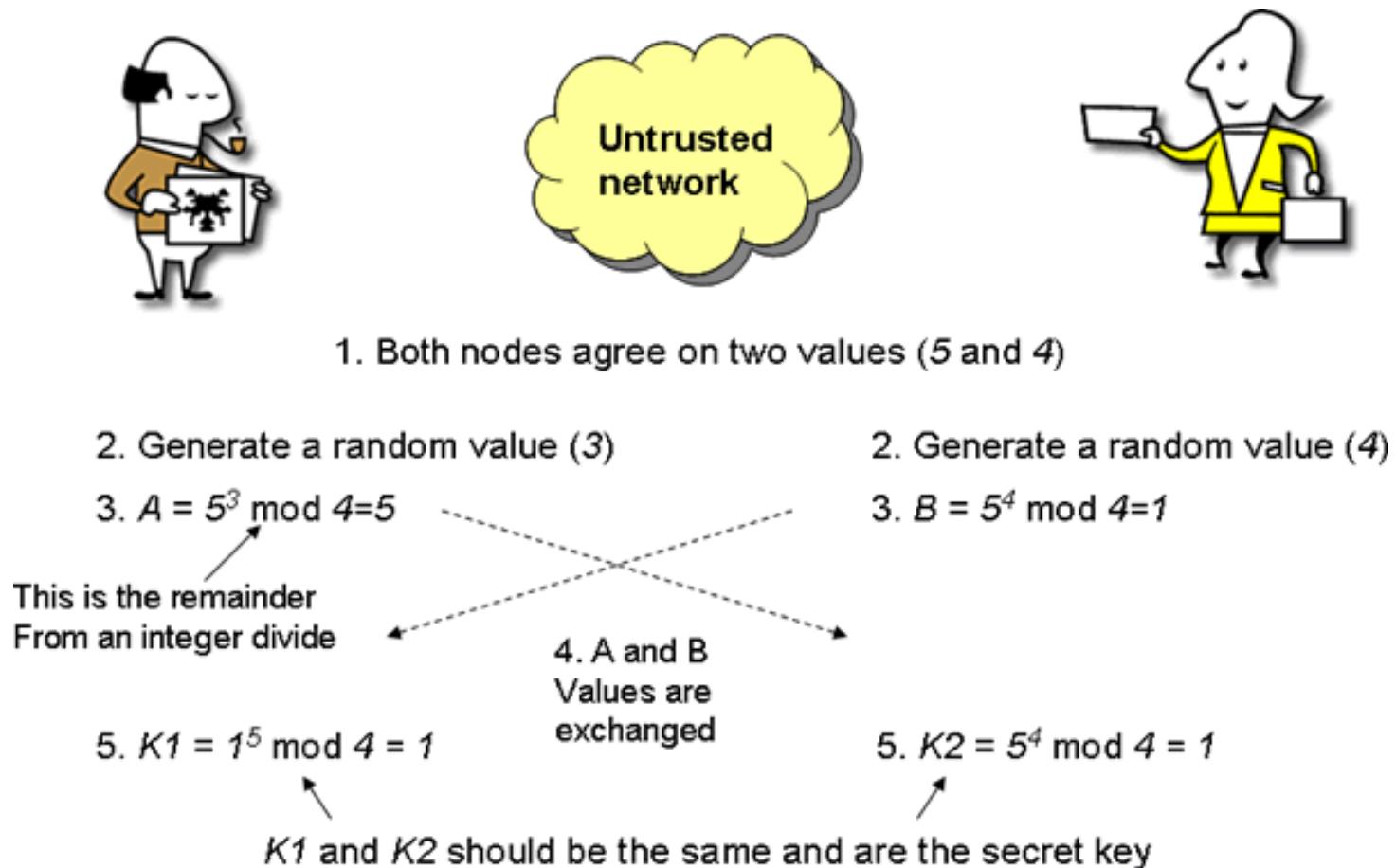
## Diffie Hellman overview

# Diffie Hellman overview

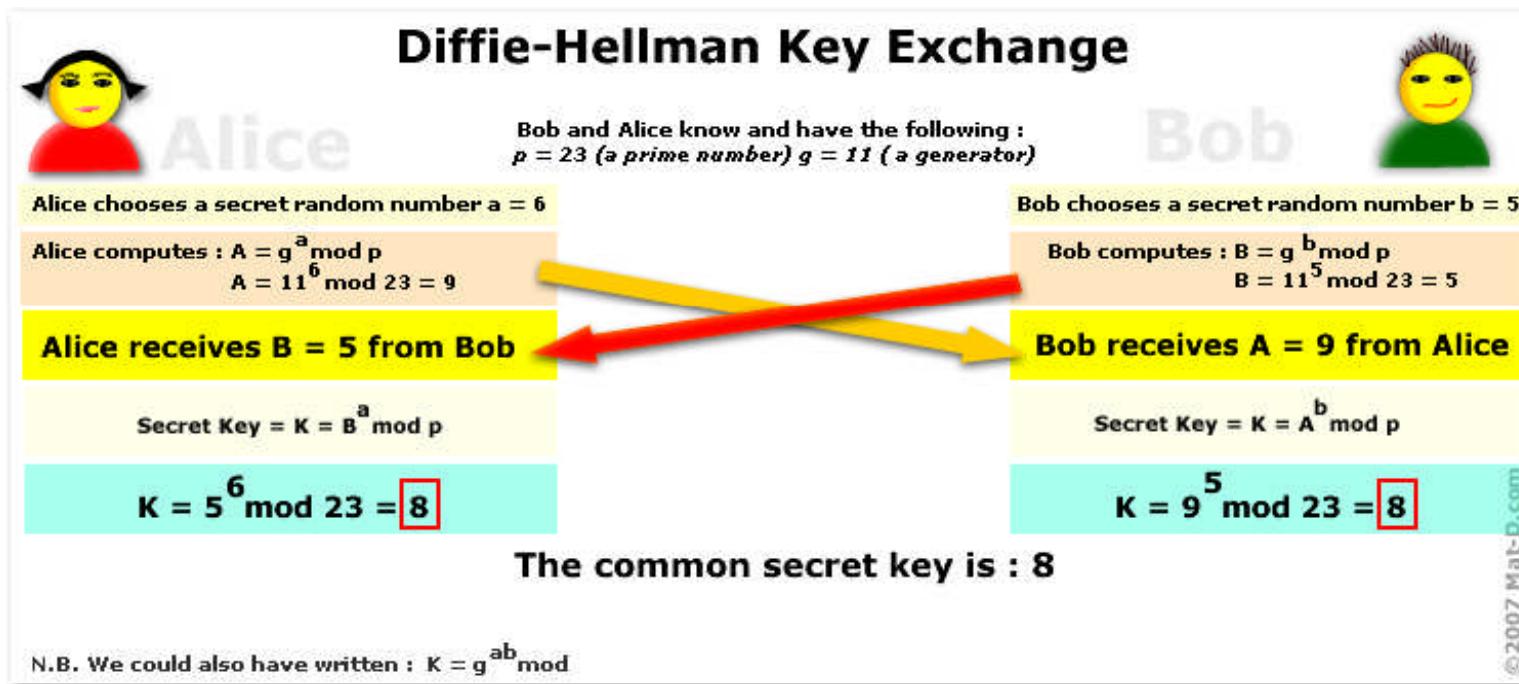


1. Both nodes agree on two values ( $g$  and  $n$ )
  2. Generate a random value ( $x$ )
  3.  $A = G^x \text{ mod } n$
  2. Generate a random value ( $y$ )
  3.  $B = G^y \text{ mod } n$
  4.  $A$  and  $B$   
Values are  
exchanged
  5.  $K1 = B^x \text{ mod } n$
  5.  $K2 = A^y \text{ mod } n$
- K1 and K2 should be the same and are the secret key*

# Diffie Hellman example 1



# Diffie Hellman example 2





# The RSA Cryptosystem

Clifford Cocks  
(Born 1950)

- A public key cryptosystem, now known as the RSA system was introduced in 1976 by three researchers at MIT.



Ronald Rivest  
(Born 1948)



Adi Shamir  
(Born 1952)



Leonard  
Adelman  
(Born 1945)

- It is now known that the method was discovered earlier by Clifford Cocks, working secretly for the UK government.
- The public encryption key is  $(n, e)$ , where  $n = pq$  (the modulus) is the product of two large (200 digits) primes  $p$  and  $q$ , and an exponent  $e$  that is relatively prime to  $(p-1)(q-1)$ . The two large primes can be quickly found using probabilistic primality tests, discussed earlier. But  $n = pq$ , with approximately 400 digits, cannot be factored in a reasonable length of time.

# HTTPS security guarantees

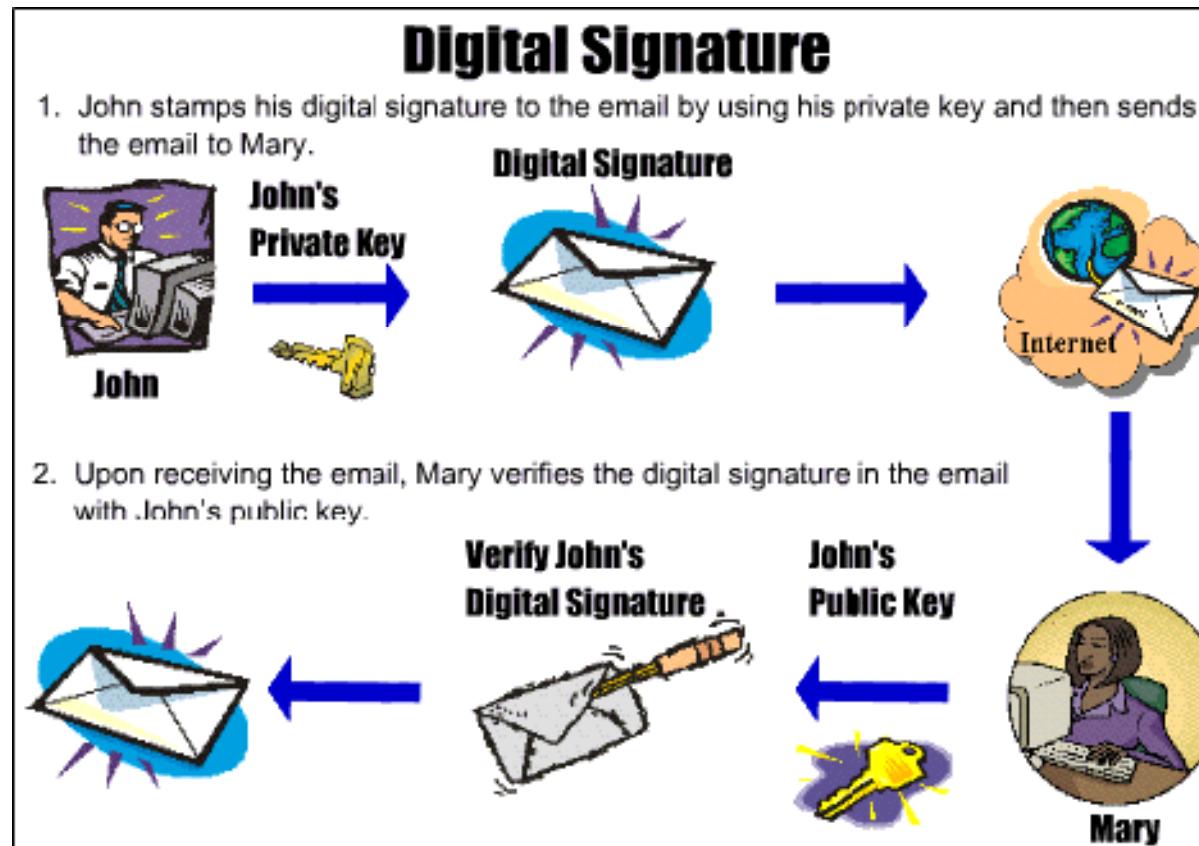
from eff.org

- Server authentication allows the browser and the user to have some confidence that they are talking to the true application server. Without this guarantee, there can be no guarantee of confidentiality or integrity.
- Data confidentiality means that eavesdroppers cannot understand the communications between the user's browser and the web server, because the data is encrypted.
- Data integrity means that a network attacker cannot damage or alter the content of the communications between the user's browser and the web server, because they are validated with a cryptographic message authentication code.

# Certificate

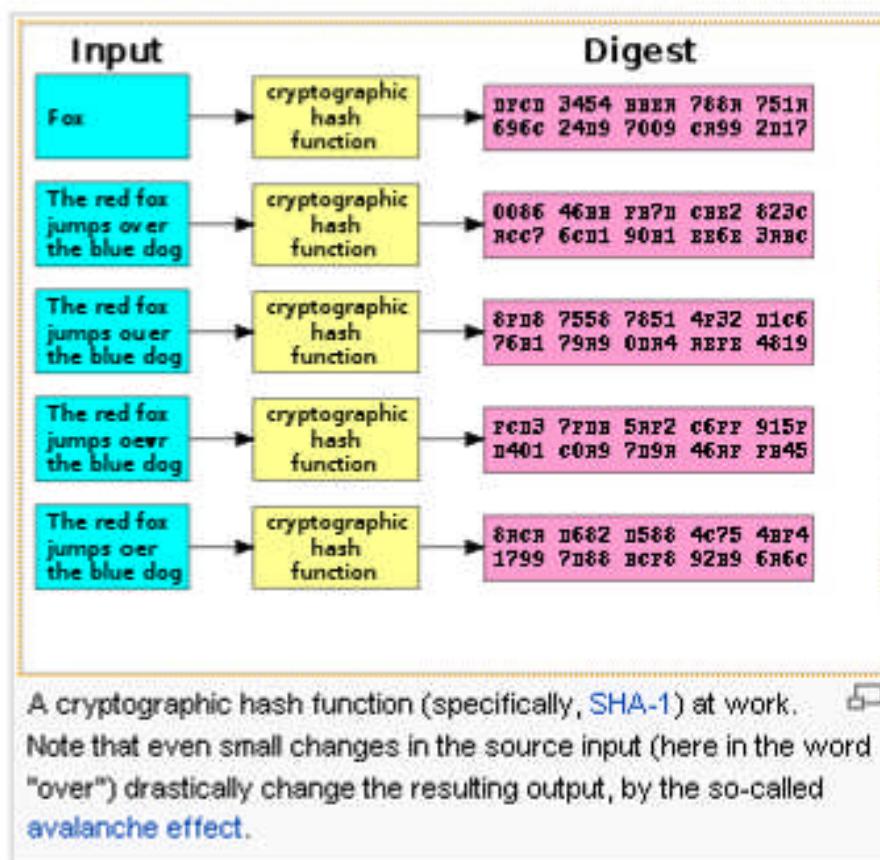


# Digital Signatures

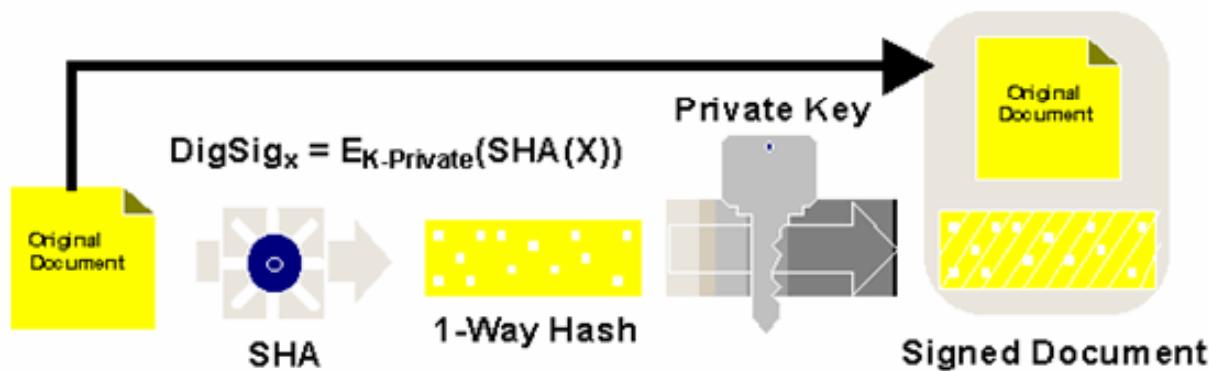


# Cryptographic hash functions

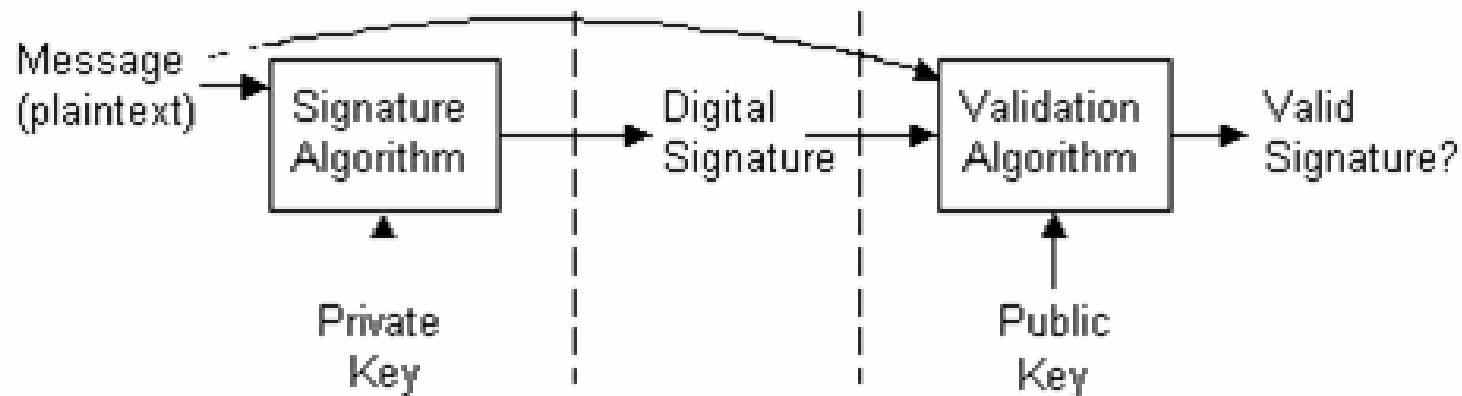
- One way functions - given output, very hard to compute input.
- Hard to find another input that gives same output.
- Hard to find 2 different inputs with same output.
- Examples:  
MD5 128 bits  
SHA 160  
SHA2 256,512



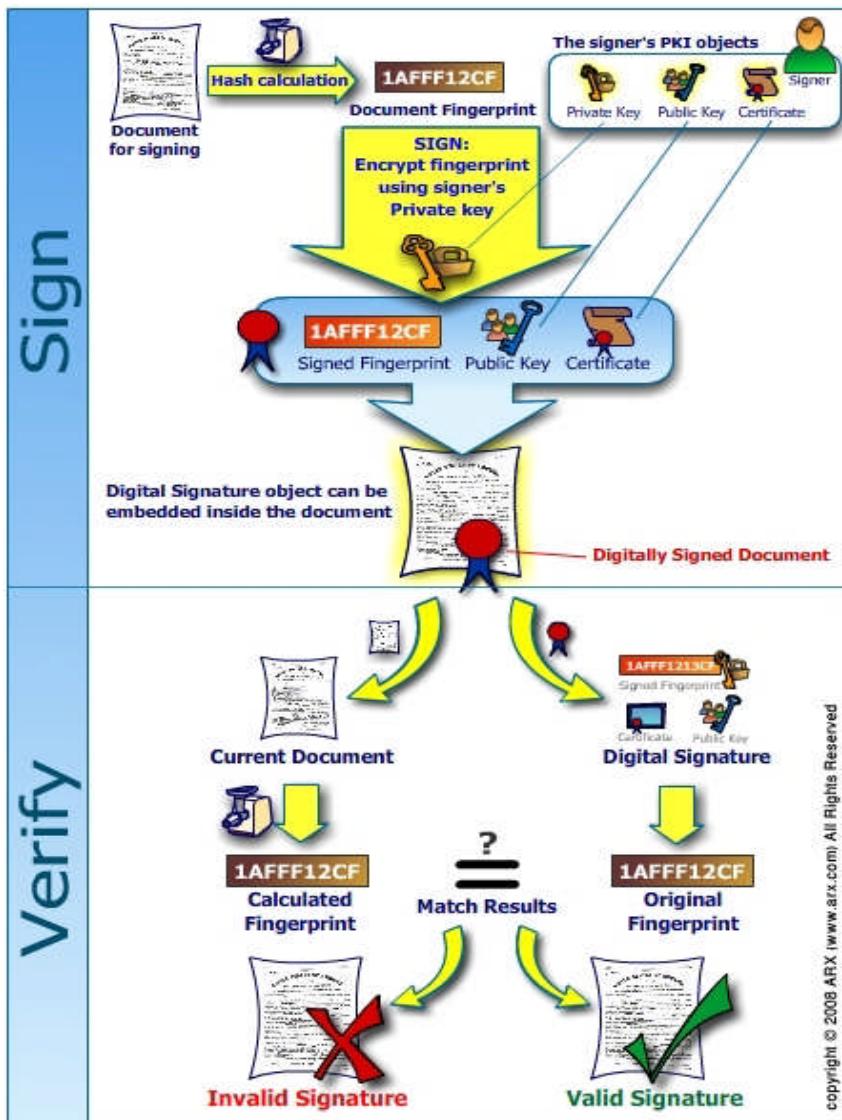
# Signing using SHA and RSA



# Digital Signatures



# SSL Cert



# SSL

- SSL is secure socket layer
- Used by https
- to secure e-commerce and logins

# https website

The screenshot shows the SBI Online Banking login page. At the top, the URL <https://www.onlinesbi.com/retail/login.htm#> is displayed in the browser's address bar, with the 'https' part circled in red. The page itself has a blue header with the SBI logo and navigation links for Home, Products & Services, and How Do I. The main content area is titled 'Login' and 'Welcome to Personal Banking'. It includes fields for 'Username \*' and 'Password \*', a 'Enable Virtual Keyboard' checkbox, and 'Login' and 'Reset' buttons. To the right is a 'Virtual Keyboard' tool. Below the login form is a note about security and a warning about phishing with a yellow exclamation mark icon. At the bottom, there are links for Complaints, Troubleshooting, Password Management, Security Tips, FAQ, About Phishing, Report Phishing, and Lock User Access. A VeriSign Secured badge is present, along with a note about encryption. A lightbulb icon provides tips for secure login.

To access your accounts...  
**Login to OnlineSBI**

(CARE: Username and password are case sensitive.)

**Username \***

**Password \***   
 Enable Virtual Keyboard

**Login** **Reset**

For better security use the Online Virtual Keyboard to login. [More...](#)

**NEVER respond to any popup,email, SMS or phone call, no matter how appealing or official looking, seeking your personal information such as username, password(s), mobile number, ATM Card details, etc. Such communications are sent or created by fraudsters to trick you into parting with your credentials.**

**VeriSign Secured**

This site is certified by VeriSign as a secure and trusted site. All information sent or received in this site is encrypted using 256-bit encryption

**Complaints** | [Trouble logging in](#) | [Password Management](#) | [Security Tips](#) | [FAQ](#) | [About Phishing](#) | [Report Phishing](#) | [Lock User Access](#)

**Mandatory fields are marked with an asterisk (\*)**

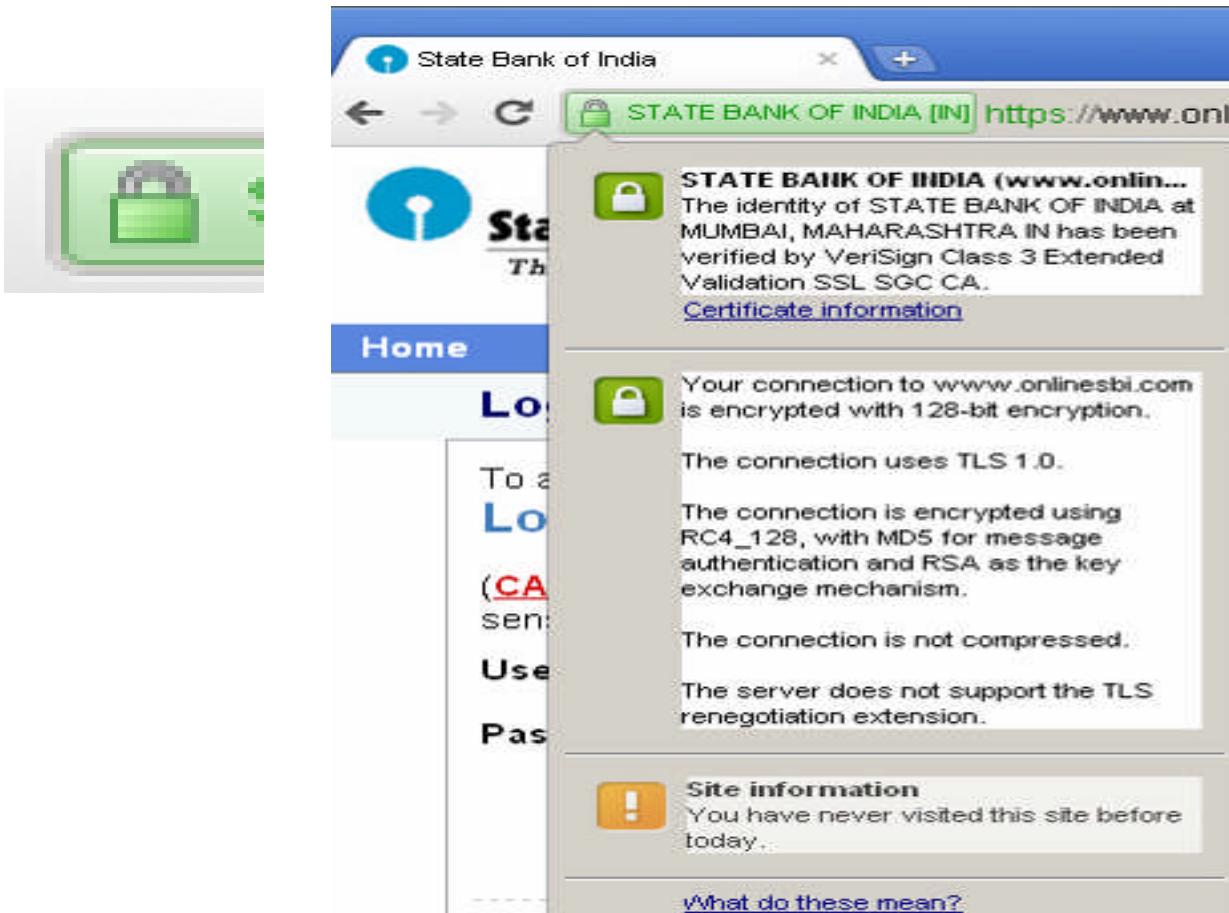
**Do not provide your username and password anywhere other than in this page**

**Your username and password are highly confidential. Never part with them. SBI will never ask for this information.**

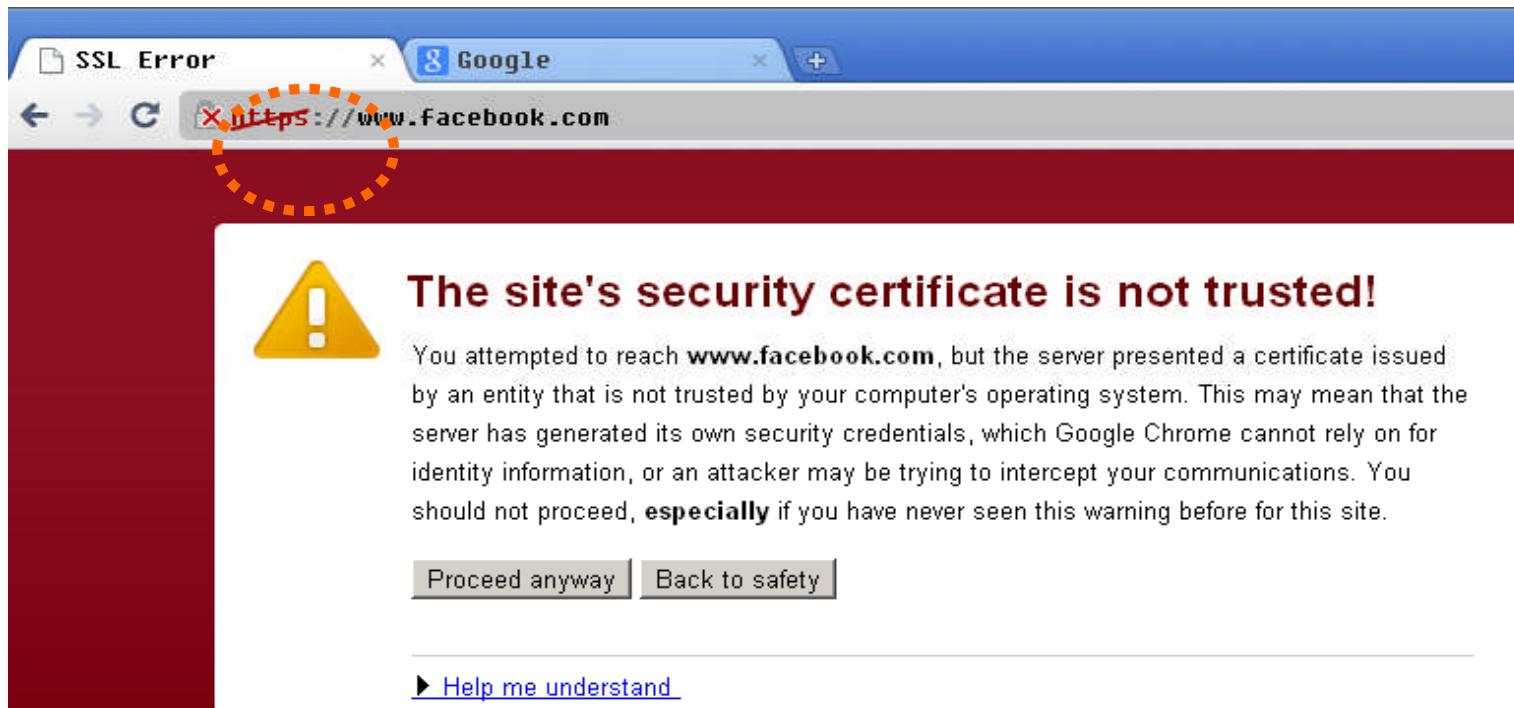
© Copyright OnlineSBI

Privacy Statement | Disclosure | Terms of Service (Terms & Conditions)

# SBI SSL certificate

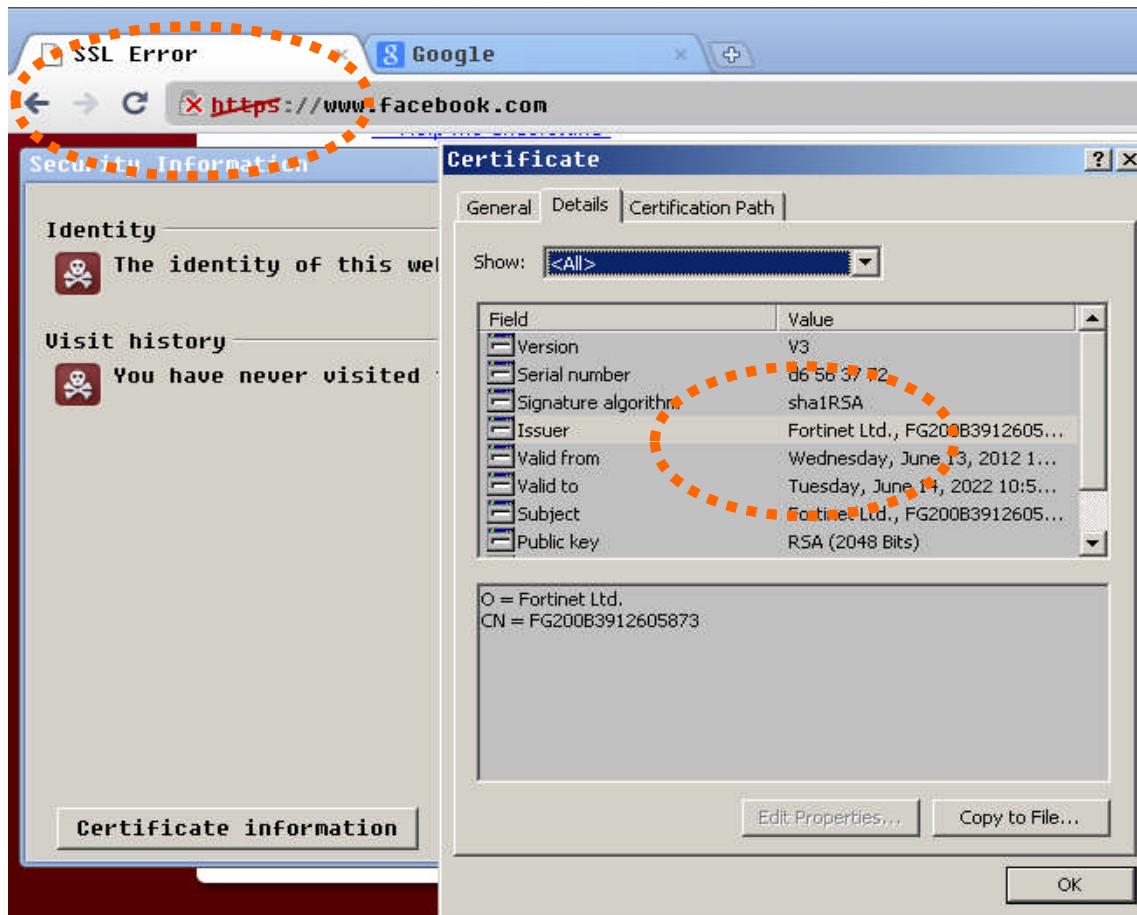


# `https:` browser warning on fake ssl certificate

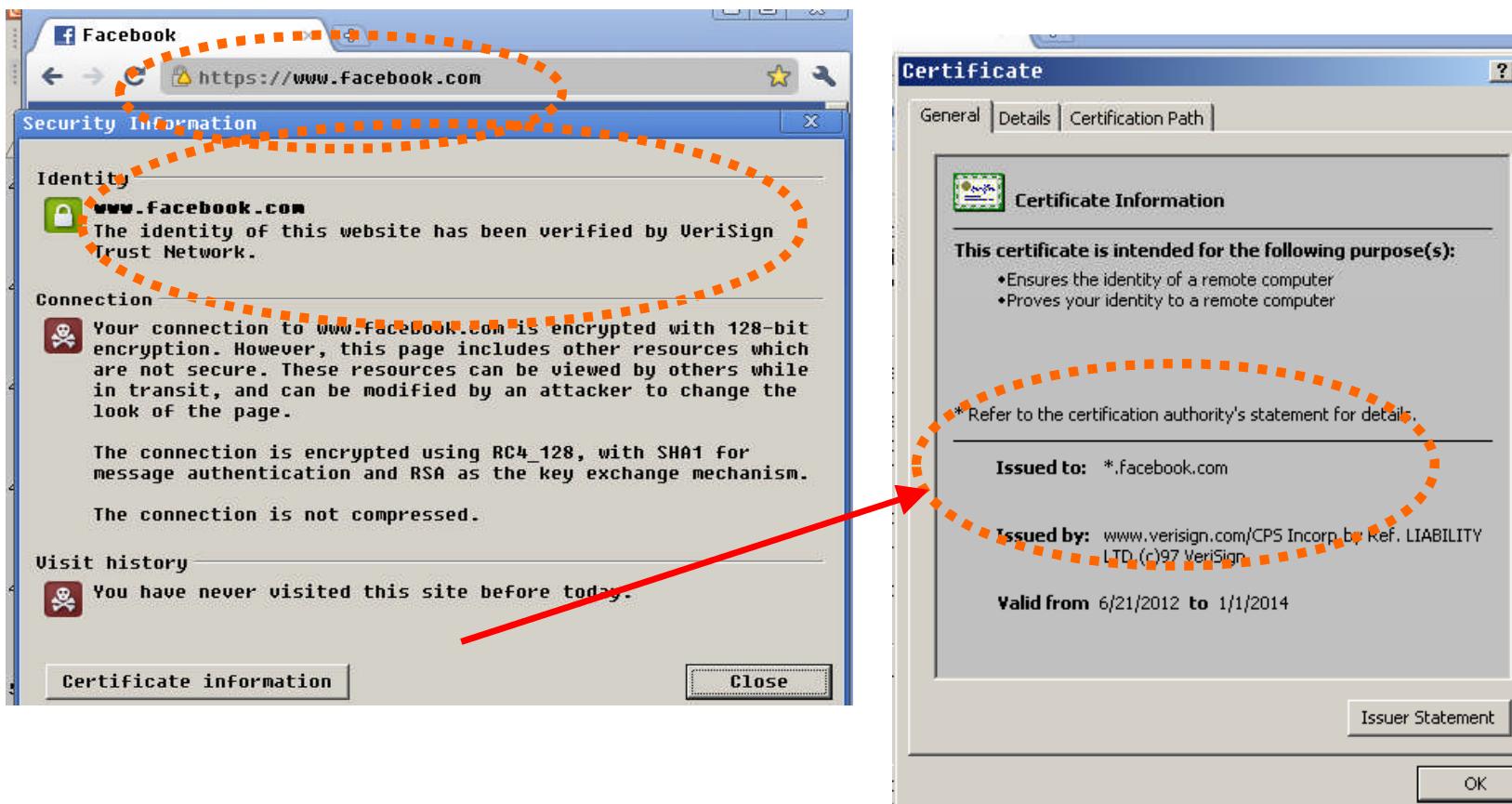


It is not possible to verify that you are communicating with www.facebook.com instead of an attacker who generated his own certificate claiming to be www.facebook.com. You should not proceed past this point.

# Firewall proxy claiming to be facebook over ssl



# Good ssl certificate, with security flaws



# Applications of Number theory: **GnuPG**

# GPG (Gnu privacy guard) PGP (pretty good privacy)

- GPG
- C:\> gpg --version
- C:\> gpg –gen-key
  - » gpg: key 43F2B829 marked as ultimately trusted
  - » public and secret key created and signed.
- ~/.gnupg

# GPG

- gpg --export --armor > Public-key.asc
- gpg --import file.asc
- gpg --sign-key RedHat
- gpg –list-keys

See <http://www.gnupg.org/gph/en/manual.html> for more help.

# GPG usage

- Generate a private key:

```
gpg --gen-key
```

- Get your public key as ascii text:

```
gpg --armor --output pubkey.txt  
-export you@gmail.com
```

# GPG usage

Send your keys to a key-server

```
gpg --send-keys youremail -keyserver  
hkp://subkeys.pgp.net
```

Import Friend's key

```
gpg --import friend.asc OR  
gpg --search-keys friend@gmail.com \  
--keyserver hkp://subkeys.pgp.net
```

# GPG usage

Encrypt message.txt for your friend

Check you have his/her key, else get it:

```
$ gpg --list-keys | grep friend
```

...

```
$ gpg --encrypt -recipient \
friend@gmail.com message.txt
```

Reading mail from your friend

```
$ gpg --decrypt reply.txt
```

# GPG usage

# Signing a file

```
$ gpg --armor --detach-sign myfile.zip
```

# Verify the signature

```
$ gpg --verify myfile.asc myfile.zip
```

**Send your key to Keyserver,  
<http://pgp.mit.edu/>**

## MIT PGP Public Key Server

Help: [Extracting keys](#) / [Submitting keys](#) / [Email interface](#) / [About this server](#) / [FAQ](#)  
Related Info: [Information about PGP](#) / [MIT distribution site for PGP](#)

---

### Extract a key

Search String:

Index:  Verbose Index

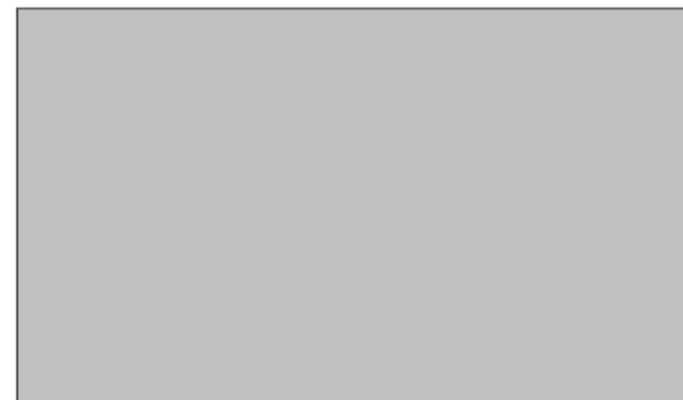
Show PGP fingerprints for keys

Only return exact matches

---

### Submit a key

Enter ASCII-armored PGP key here:



# Homework

- Read <http://www.pgpi.org/doc/pgpintro/>  
(google for current link to "pgp intro")
- Install gnupg (if you don't have it)
- Create your public key
- Upload your key to mit keyserver
- Sign your friends public key
- Send a signed/encrypted test email to a friend.
- Decrypt and verify signature in the email you receive

# Basic Number Theory

- Primes
- Factoring
- RSA
- Random numbers

# Primes

```
$ OpenSSL  
> prime 17 (is 17 a prime?)  
    11 is prime  
    (Hex 11 = 16 + 1 = 17 Dec).  
$ gpg --gen-prime 1 30  
    # 30 bit prime.  
    3756E197 # Hex  
    # Decimal 928440727
```

# Factoring

Using cygwin on windows or linux:

```
$ factor 1111111111111111  
3 31 37 41 271 2906161  
  
$ factor 928440727  
928440727  
  
$ factor 928440729  
3 3 337 443 691
```

# Factoring with OpenSSL

```
$ factor 928440727
```

```
928440727: 928440727
```

```
$ factor 928440729
```

```
928440729: 3 3 337 443 691
```

# OpenSSL to generate public key

Generate a new public/private keypair in openssl

```
$ openssl genrsa -out key.pem
Generating RSA private key, 512 bit long
modulus
..+++++++
e is 65537 (0x10001)
```

Extract the modulus, e, primes from your key:

```
$ openssl rsa -in key.pem -noout -text
publicExponent: 65537 (0x10001)
Modulus=....long-string-of-hex-digits...
```

# Random numbers

- PRNG are predictable, /dev/urandom
  - Use a fixed function and time to generate random numbers.
  - Example:  $r = \text{md5}(\text{time} + \text{hostname} + \text{process id})$
- RNG /dev/random, uses system entropy
  - Use physical system to generate random number
  - Example:  $r = \text{md5}(\text{mouse and keyboard delays})$

# Brute force cracking passwords

WORD,	SIZE, BITS, CRACK-TIME
1 Single word,	8 char, 24-bits, Seconds
2 Random [a-z]	8 char, 37-bits, Minutes
3 Random [a-z]	16 char, 75-bits, Decades

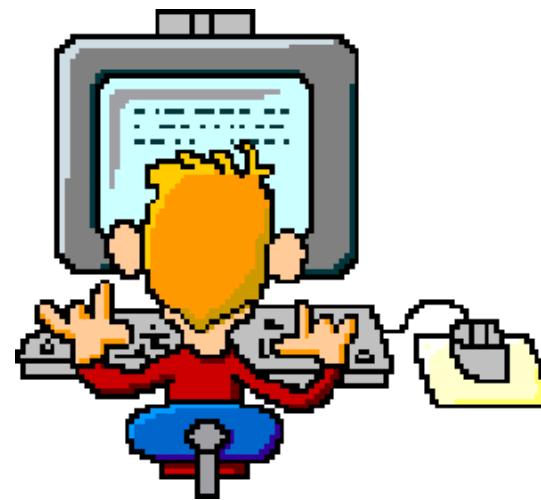
Completely random printable string

- | SIZE, BITS, CRACK-TIME            |
|-----------------------------------|
| • 6 char, 40-bits, Minutes        |
| • 8 char, 52-bits, Hours          |
| • 12 char, 78-bits, Decades       |
| • 15 char, 97-bits, Centuries     |
| • 20 char, 130-bits, Un-crackable |

# Homework on Number Theory

## Primes, Factor, EEA, BigInt, 1000!

## Diffie Hellman, RSA



# Homework

Write a program **primes.c** to print the first n  
primes numbers

\$ primes 100

Prime numbers less than 100:

2 3 5 7 11 13 17 19 23 29 31 37 41 43 47 53  
59 61 67 71 73 79 83 89 97

# Animation: Sieve of Erosthenes algorithm

	2	3	4	5	6	7	8	9	10	Prime numbers
11	12	13	14	15	16	17	18	19	20	
21	22	23	24	25	26	27	28	29	30	
31	32	33	34	35	36	37	38	39	40	
41	42	43	44	45	46	47	48	49	50	
51	52	53	54	55	56	57	58	59	60	
61	62	63	64	65	66	67	68	69	70	
71	72	73	74	75	76	77	78	79	80	
81	82	83	84	85	86	87	88	89	90	
91	92	93	94	95	96	97	98	99	100	
101	102	103	104	105	106	107	108	109	110	
111	112	113	114	115	116	117	118	119	120	

# Print primes, sieve of Erosthenes algorithm

```
Print_Primes( n ) {  
    int composite[n];  
    for( i = 2 ; i < n; i++ )  
        composite[i] = 0;  
    for( i = 2 ; i < n; i++ ){  
        if( composite[i] )  
            continue;  
        print i  
        for(k=i*2; k < n; k += i)  
            composite[k] = 1;  
    }  
}
```

# Homework: Write is-prime.c, factor.c

```
$ is-prime 100  
100 is not a prime
```

```
$ is-prime 101  
101 is a prime
```

```
$ factor 100  
100: 2 2 5 5
```

```
$ factor 101  
101: 101
```

# Is Prime, naive algorithm

```
is_prime(n) {
    for f in 2 to sqrt(n) {
        if (n mod f == 0)
            return "n is composite, factor is f"
    }
    return "n is a prime, no factors found"
}
```

## Homework: Write Euclid's GCD, Ext-GCD, Modular Inverse in C

```
def gcd(a, b):                      # python code
    if (b == 0): return a
    else: return gcd(b, a mod b)

def eea(a, b):                      # ext euclid gcd
    if a == 0: return (b, 0, 1)
    else:
        g, y, x = egcd(b % a, a)
        return (g, x - (b // a) * y, y)

def mod_inv(a, m):                  # (a * mod_inv(a,m) ) mod m == 1
    g, x, y = eea(a, m)
    if g != 1: Error "no modular inverse"
    else: return x % m
```

# HW: Code Fast Power Mod in C

```
// Return: b^p mod n
// Usage: pow_mod(2,100, 5) for 2100 mod 5
// Trick 1. resize 2p mod n, in each step
// Trick 2. don't loop 1..p, use binary(p)
pow_mod(int b, int p, int n)
    int result = 1
    while (p > 0) {
        if (p % 2)
            result = (result * b) % n
        p /= 2          // trick1. p = p>>1 , log(p) time.
        b = (b * b) % n // trick 2. b bounded by mod n
    }
    return result
```

# BIGINT 1000!

# Homework, compute $1000!$ exactly in C

1. Store bigint as an array of digits.

2. Start with

```
void fact(int n)
    bigint f = 1;
    for(i = 1; i<=1000; i++) // f=1*2*...*1000
        mult(f, i, f)
    print(f);
```

3. Write helper functions

```
void mult(bigint *a, int i) { ??? } /* homework*/
void print(bigint *n) /* homework */
```

# Solution 1. 1000!

```
/*
```

What: compute n! exactly using bigint.

Date: 2013-03-22

By moshahmed/at/gmail

Usage:

```
$ gcc -g -Wall factorial-bigint.c -lm -o fact
```

```
$ fact 1000
```

```
1000!= 402387260077093773543702....000
```

```
*/
```

# Solution 2. 1000!

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#define N 10000
typedef struct { char digits[N]; } bigint;
// To make it easy to extend a bigint
// on right side during multiplication.
// Store bigint value as reverse(digits),
// So digits="321" => value=123.
```

# Solution 3. 1000!

```
void print(bigint *b) {  
    int len=strlen(b->digits);  
    while(len>=0) // Index hi to low.  
        printf("%c", b->digits[len--]);  
    printf("\n");  
}
```

## 4. 1000!

```
void mult(bignum *a, int b) {  
    int carry=0, len = strlen(a->digits), k;  
    // HOME WORK  
    // Sample computation to do:  
    // Input: a = "027\0", b=7  
    // means: compute: 720 * 7  
    // output: a = 5040  
}
```

# Homework, algorithm for bigint multiply

```
mult( bigint *a, int b) // Do: a = a * b
    int carry=0
    for( i= a->len; i>0; i--)
        x = b * a->digit[i] + carry
        b -> digit[i] = x % 10
        carry = x / 10
    while carry > 0 do
        b->digit[i++] = carry % 10
        carry /= 10
```

# Main function to compute 1000!

```
int main(int argc, char *argv[]){
    int i, input = 1000;
    bigint fact;
    fact.digits[0]='1';
    fact.digits[1] = '\0'; // initialize fact=1
    if ( argc > 1 ) input = atoi(argv[1]);
    for(i=1;i<=input;i++)
        mult(&fact, i);
    printf("%d!=",input); print(&fact);
    return 0;
}
```

# Homework, C code for bigint arithmetic functions

Write functions to

1. bigint \*add(bigint \*a, bigint \*b):  
    // return result = a + b
2. bigint \* mult(bigint \*a, bigint \*b)  
    // return result = a \*b; multiply a digit at a time.
3. minus(bigint \*a, bigint \*b)  
    // return result = a – b; subtract using borrow
4. bigint \*random( ) // return a random bigint
5. print(bigint \*a)
6. Use these to multiply 2 random hundred digit numbers,  
    and print the result.

# Diffie Hellman Computation

# Verify this Diffie Hellman calculation

Alice and Bob choose:  $p=23$  (prime),  $g=11$  (generator).

Alice chooses secret  $a=6$ , computes  $A=g^a \equiv 11^6 \equiv 9 \pmod{23}$

Bob chooses secret  $b=5$ . computes  $B=g^b \equiv 11^5 \equiv 5 \pmod{23}$ .

Alice sends Bob:  $A=9$

Bob sends Alice:  $B=5$ .

Alice computes:  $K=B^a \equiv 5^6 \equiv 8 \pmod{23}$

Bob computes:  $K=A^b \equiv 9^5 \equiv 8 \pmod{23}$ .

-----

Repeat above calculations and compute  $K$ ,

i.e. " $(g^a)^b \pmod{p}$ "

" $(g^b)^a \pmod{p}$ "

where  $p=101$ ,  $g=7$ ,  $a=4$ ,  $b=12$ ,

# Solution using Pari and Maple

Q. Compute K, with g=7, a=4, b=12, p=101

i.e. " $(g^a)^b \bmod p$ " and " $(g^b)^a \bmod p$ "

c:\> pari

```
gp > lift(Mod((7^4)^12, 101))
```

```
68
```

```
gp > lift(Mod((7^12)^4, 101))
```

```
68
```

c:\> maple

```
(7**4)**12 mod 101;
```

```
68
```

```
(7**12)**4 mod 101;
```

```
68
```

# RSA Computation

Using pari,maple/mathematica

# Verify this RSA calculation using Maple / Pari / bigint / math package

1. Choose two large prime,  $p = 61$  and  $q = 53$ .
2.  $n = p * q = 61 * 53 = 3233$
3.  $\phi(p * q) = (61 - 1)*(53 - 1) = \phi(3233) = 3120$
4. Let  $e = 17$ ,  $\gcd(e, n)=1$ .
5. Compute  $d=2753 = \text{inv}(17, 3120)$  (inverse modulus n).  
i.e. ( $d * 17 = 1 \pmod{3120}$ )
  1. Hard to find  $d$ , without knowing factors of  $n$ .
  2. public key is ( $n = 3233$ ,  $e = 17$ ),  
encryption function is  $m^{17}(\pmod{3233})$ .
  3. private key is ( $n = 3233$ ,  $d = 2753$ )  
decryption function is  $c^{2753}(\pmod{3233})$ .

**Repeat above:**  $e=23$ ,  $p=11$ ,  $q=13$

i.e.  $n = p * q$ ,  $\phi=(p-1)*(q-1)$ ,  $d = \text{inv}(e, \phi)$

# Solution for RSA keys using Pari, Mathematica and Maple

Given:  $e=23$ ,  $p=11$ ,  $q=13$ , compute RSA keys:

$$n = p \cdot q = 11 \cdot 13 = 143$$

$$\phi = (p-1) \cdot (q-1) = 10 \cdot 12 = 120$$

$$d = \text{inv}(e, n) = \text{inv}(23, 120) = 47$$

c:\> pari

```
gp> bezout(23,120)[1]  
47
```

\| bezout is gcdext(x,y): returns [u,v,d]  
\| such that d=gcd(x,y) and u\*x+v\*y=d.

c:\> mathematica

```
in[1] PowerMod[23, -1, 120]  
out[1] 47
```

//  $23^{-1} \equiv_{120} 27$

c:\> maple

```
23^(-1) mod 120  
47
```

```
public key(n, e) = ( 143, 23 ) // e for encrypt
```

```
private key(n, d) = ( 143, 47 ) // d for decrypt
```

# RSA encrypt/decrypt a character

1. Encrypt (  $\text{ord}(\text{'A'}) = 65$  ), using  
key=(e=17,n=3233)  
 $c = 65^{17} \pmod{3233} = 2790.$
2. Decrypt 2790, using (d=2753,n=3233)  
 $m = 2790^{2753} \pmod{3233} = 65$ , to get 'A' = chr(65)
3. Repeat above, encrypt 'C' and decrypt it.  
Show all the numbers.

## Solution: RSA encrypt/decrypt 'C':

Solution: perl -e "print ord('C')" = 67

$$c = 67^e \bmod n = 67^{23} \bmod 143 = 111$$

$$d = 111^d \bmod n = 111^{47} \bmod 143 = 67 = 'C'$$

perl -e "print chr(67)" = 'C'

c:\> pari

gp> lift(Mod(67^23, 143)) = 111

gp> lift(Mod(111^47, 143)) = 67

c:\> maple

> 67^23 mod 143 # prints 111

> 111^47 mod 143 # prints 67

# Computational Geometry

References: Cormen, Shamos, Berge.

# Applications

- Graphics, Video
- GIS, maps, layered maps
- Transportation
- CAD/CAM (computer aided design/manufacturing).
- Robotics, motion planning
- Routing VLSI chips
- Molecular chemistry

# Geometry

**Plane Geometry:** the geometry that deals with figures in a two-dimensional PLANE.

**Solid Geometry:** the geometry that deals with figures in three-dimensional space.

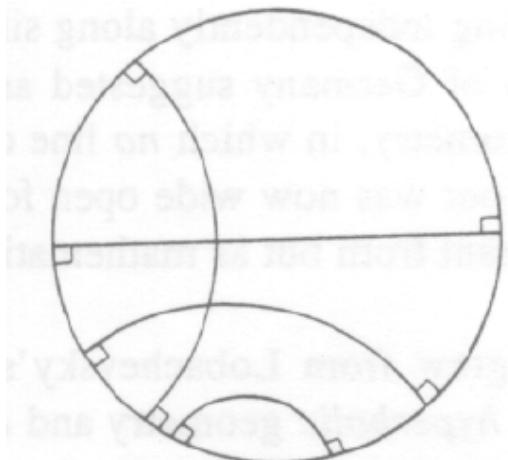
**Spherical Geometry:** the geometry that deals with figures on the surface of a sphere.

**Euclidean Geometry:** the geometry (plane and solid) based on Euclid's postulates.

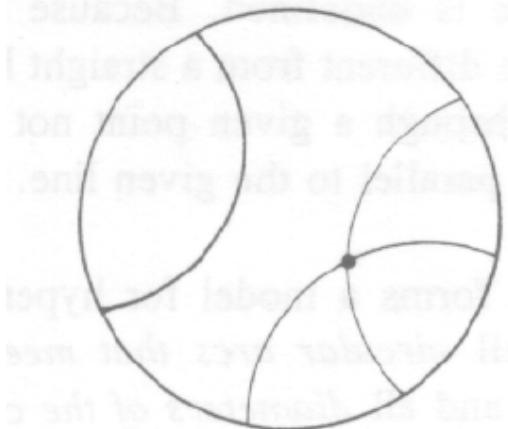
**Non-Euclidean Geometry:** any geometry that changes Euclid's postulates.

**Analytic Geometry:** the geometry that deals with the relation between ALGEBRA and geometry, using GRAPHS and EQUATIONS of lines, curves, and surfaces to develop and prove relationships.

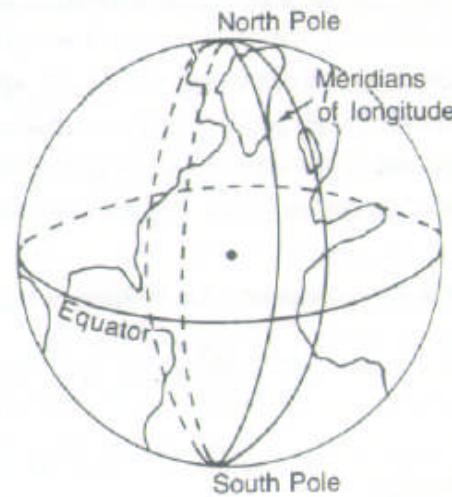
# Non-Euclidean Geometry



Models for hyperbolic geometry

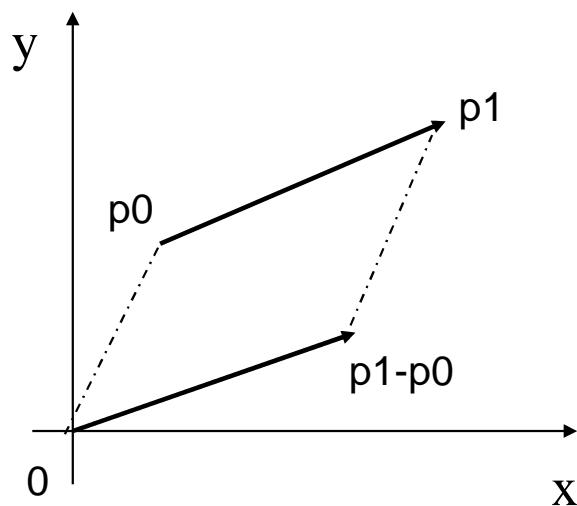


Models for elliptic geometry



# Points, Lines, Vectors

- We will use x-y plane coordinates for points.
- $p_1 = (x_1, y_1)$ ,  $p_0 = (x_0, y_0)$ , etc..
- Lines segments: have start and end points
- Vectors: have direction and length (magnitude).
- $|v|$  is the magnitude of vector  $v$ .



# Basic Geometric Objects in the Plane

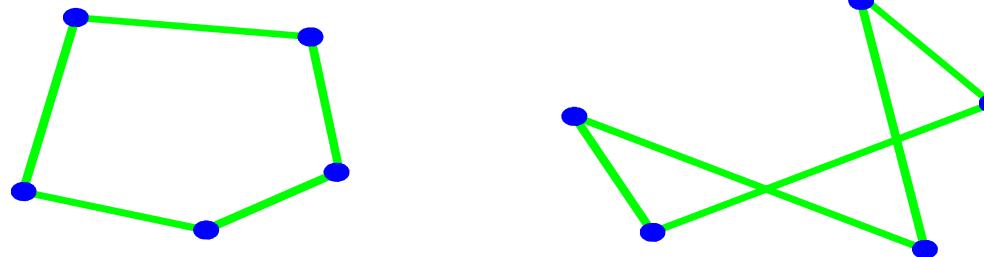
*point* : denoted by a pair of coordinates (x,y)



*segment* : portion of a straight line between two points

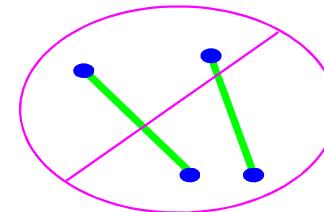
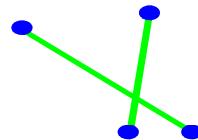


*polygon* : circular sequence of points (**vertices**) and segments (**edges**)

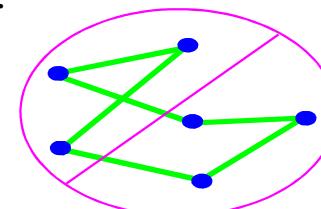
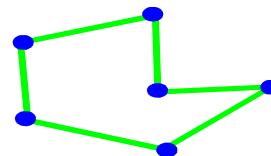


# Some Geometric Problems

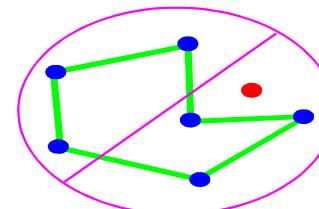
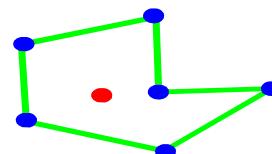
**Segment intersection** Given two segments, do they intersect



**Simple closed path:** given a set of points, find a non-intersecting polygon with vertices on the points.

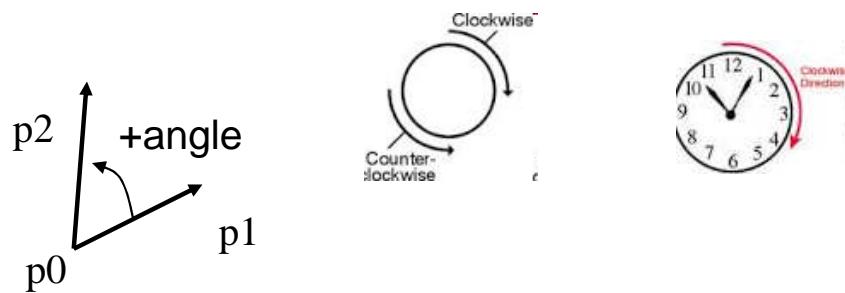


**Inclusion in polygon** Is a point inside or outside a polygon ?



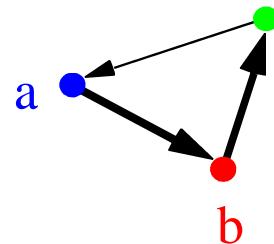
# Q1 Lines: clockwise/ccw?

- Given directed lines  $p_0p_1$  and  $p_0p_2$ ,
- Is  $p_0-p_1$  clockwise (cw) turn from  $p_0p_2$ ?
- Clockwise (cw) is considered negative angle.
- Anti-clockwise is also called counter clockwise (ccw), is considered positive angle.

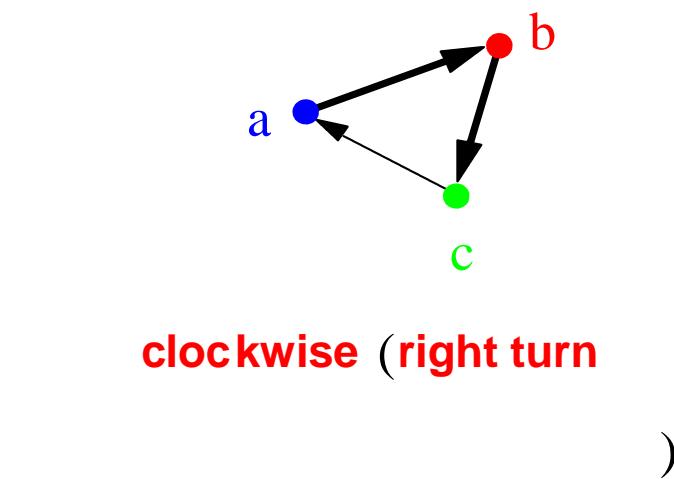


# Orientation in the Plane

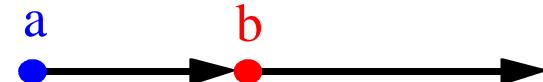
- The orientation of an ordered triplet of points in the plane can be
  - counterclockwise (left turn)
  - clockwise (right turn)
  - collinear (no turn)
- Examples:



counter clockwise (left turn)



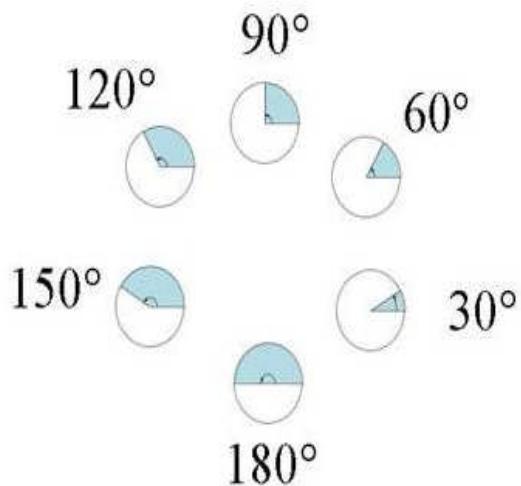
clockwise (right turn)



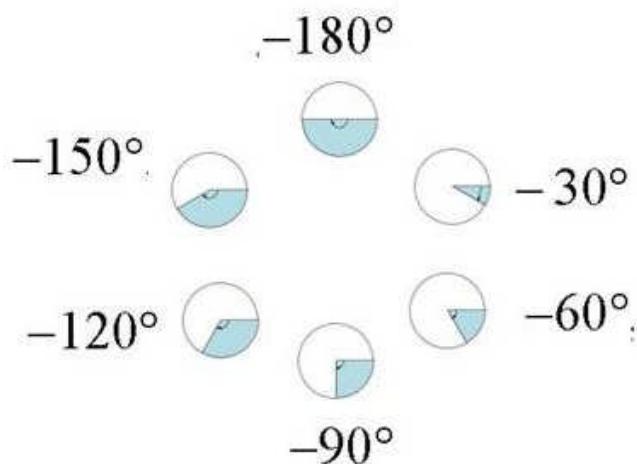
collinear no turn

# CW and CCW angles

Anti- Clockwise rotation of angles

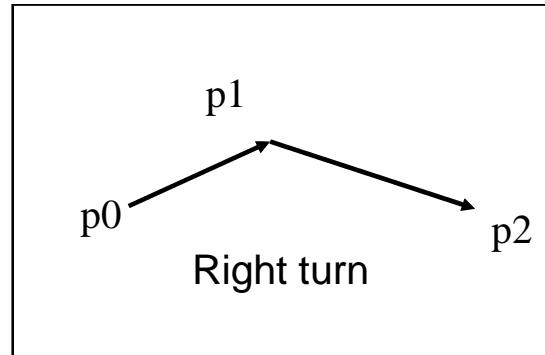
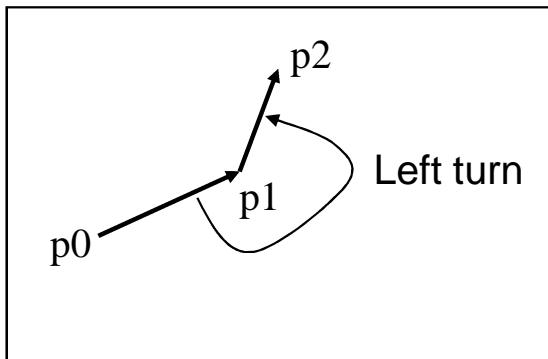


Clockwise rotation of angles



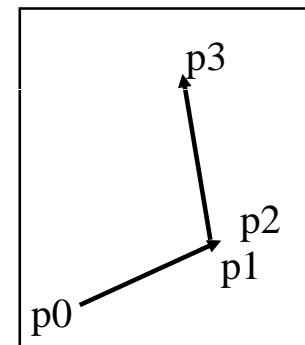
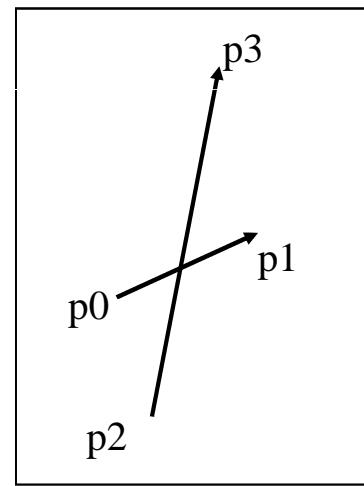
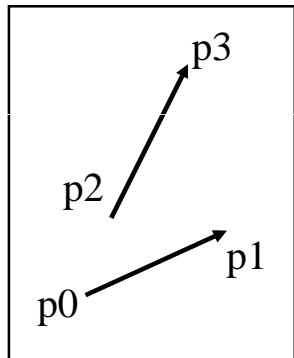
## Q2 Line: Left/Right turn?

- Given two line segments  $p_0p_1$  and  $p_1p_2$ , if we go from  $p_0p_1$  to  $p_1p_2$ ,
- Is there a left or a right turn at  $p_1$ ?



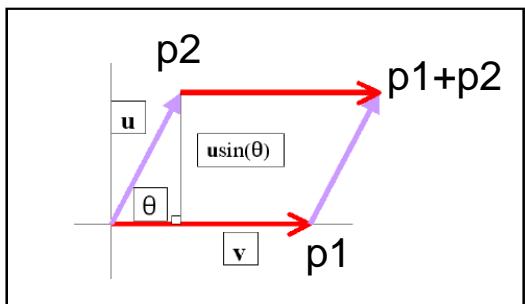
# Q3 Lines: intersect?

Do line segments  $p_0p_1$  and  $p_2p_3$  intersect?

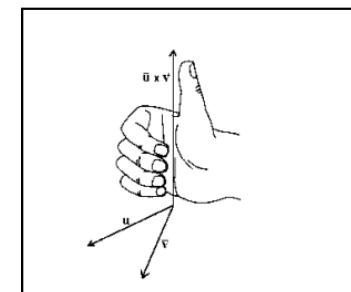


# Cross product of 2 vectors

- Given vectors  $p_1, p_2$ , their *cross-product*  $p_3$  is
- $p_3 = p_1 \times p_2 = (x_1.y_2) - (x_2.y_1) = - p_2 \times p_1$
- Use the "*right hand rule*" for direction of  $p_3$ .
- Magnitude* of  $p_3$  is area of the parallelogram formed by  $p_1$  and  $p_2$ , area=  $(|p_1|.|p_2|.sin(t))$ , where  $t$  is angle between the vectors.

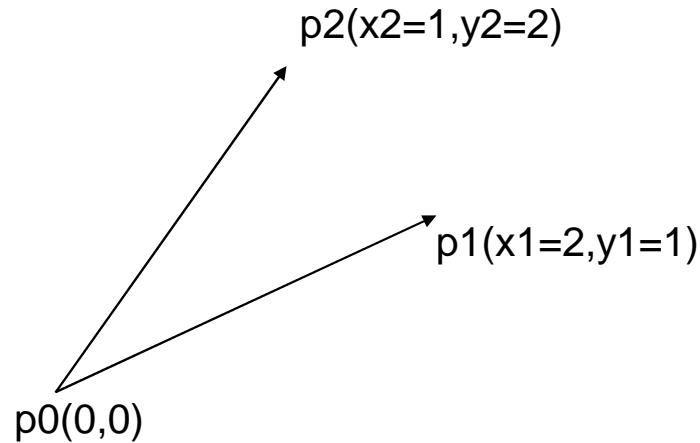


$$\begin{aligned} p_1 \times p_2 &= \det \begin{pmatrix} x_1 & x_2 \\ y_1 & y_2 \end{pmatrix} \\ &= x_1 y_2 - x_2 y_1 \\ &= - p_2 \times p_1 \end{aligned}$$



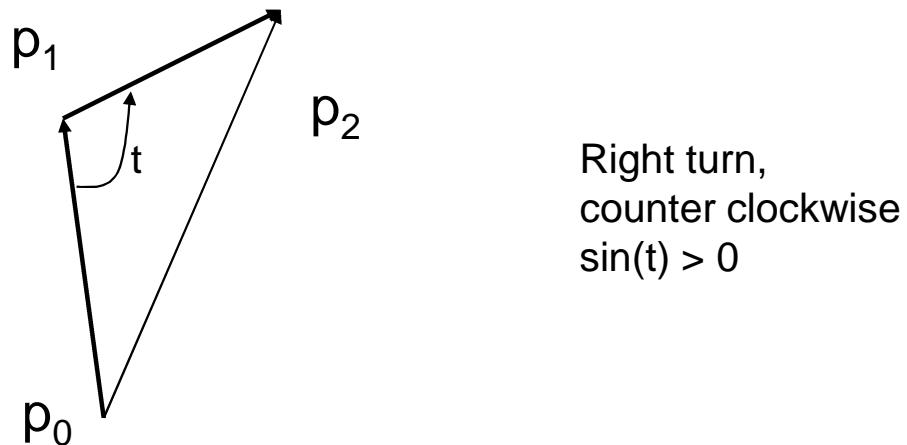
# Example: cross product

- $p_1 \times p_2 = (x_1.y_2) - (x_2.y_1)$   
 $= 2*2-1*1=4-1=3$ , (ccw).
- $p_2 \times p_1 = -3$  (clockwise)



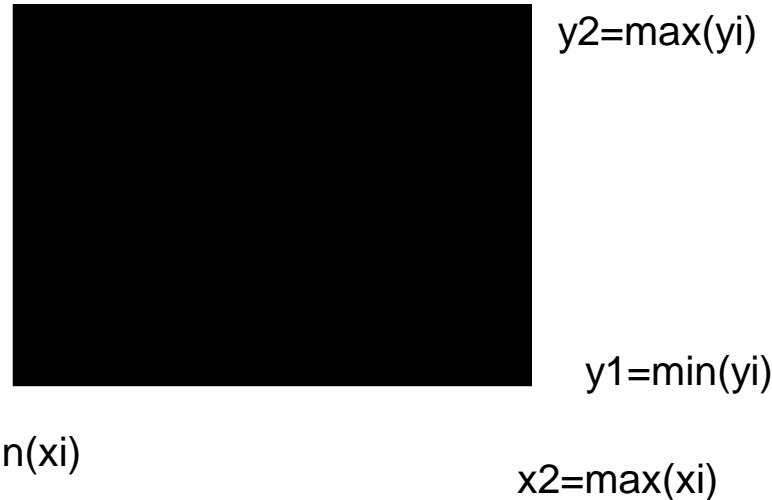
# Left or Right turn?

- Given  $p_0p_1$  and  $p_1p_2$ ,
- Going from  $p_0$  to  $p_1$  to  $p_2$ ,
- Do we turn left/right at  $p_1$ ?
- Compute  $d = (p_1 - p_0) \times (p_2 - p_0)$
- if  $d > 0$ , then right turn
- if  $d < 0$ , then left turn



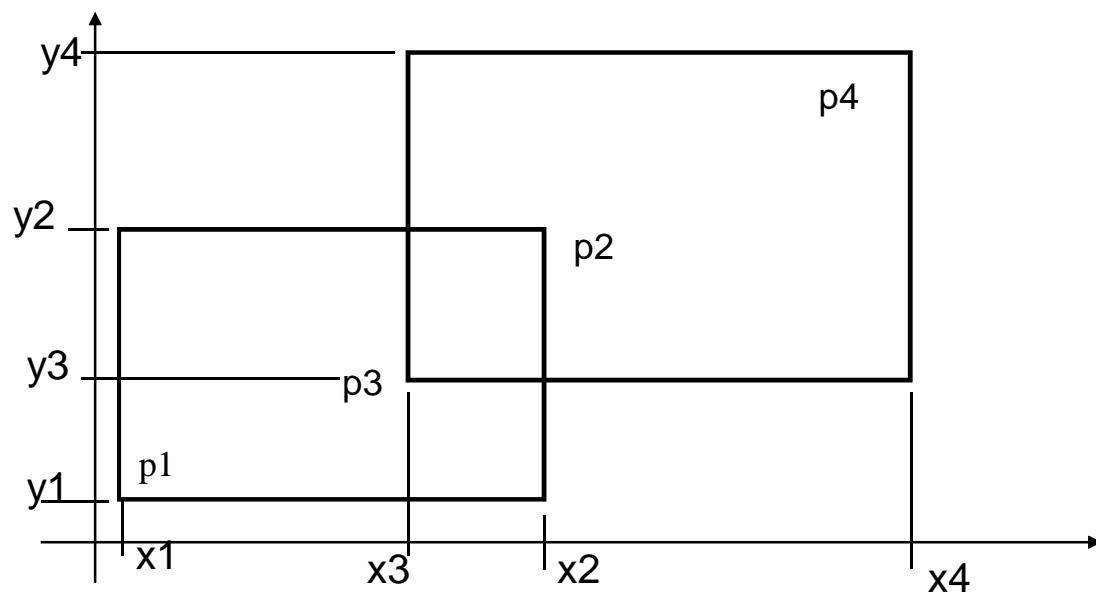
# Bounding box

- bounding box is the smallest x-y rectangle containing the object



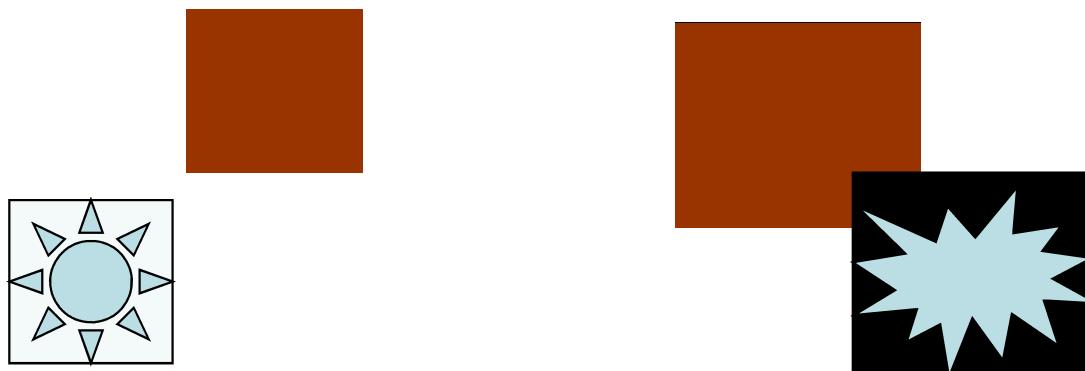
# Intersection of rectangles

- $\text{Rect}(p1, p2) \times \text{Rect}(p3, p4)$  iff
  - $((x1 \leq x3 \leq x2 \leq x4) \parallel$
  - $(x3 \leq x1 \leq x4 \leq x2) \quad ) \ \&\&$
  - Similar conditions for  $y1..y4$ .



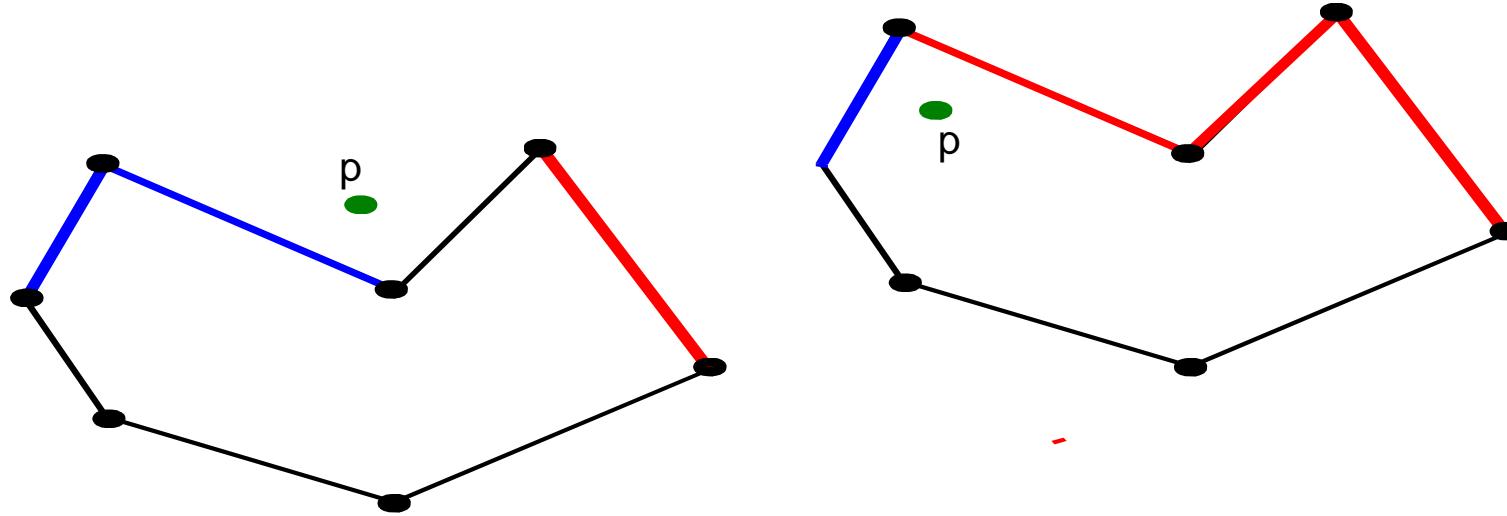
# Quick Intersection check

- Collision detection
- Approximate objects by boxes.
  - If bounding box of two objects don't intersect, the objects cannot intersect. (common case).
  - But, If boxes intersect, more complex test is required for intersection.



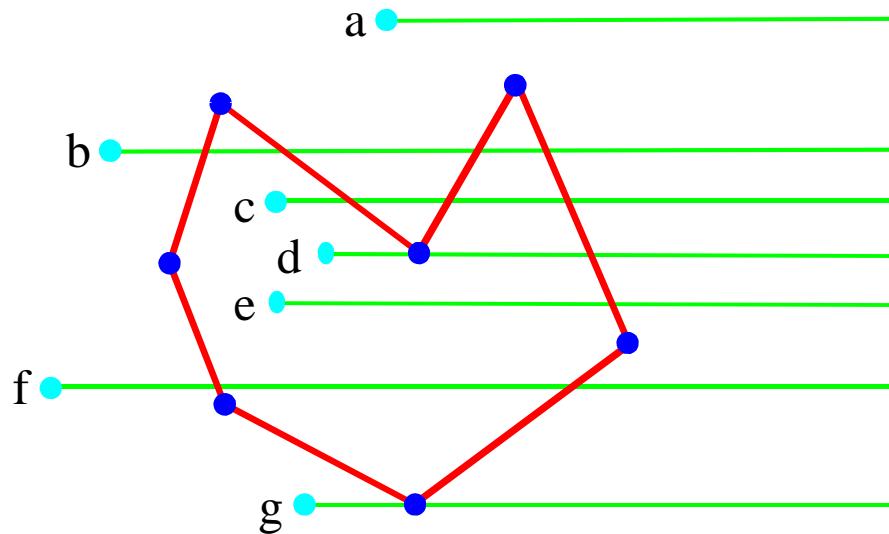
# Point Inclusion

- Given a polygon and a point  $p$ ,
- Is the point inside or outside the polygon?
- Orientation helps solving this problem in linear time



# Point Inclusion: algorithm

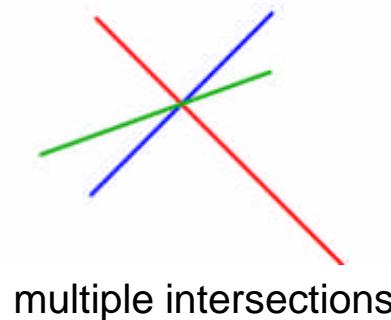
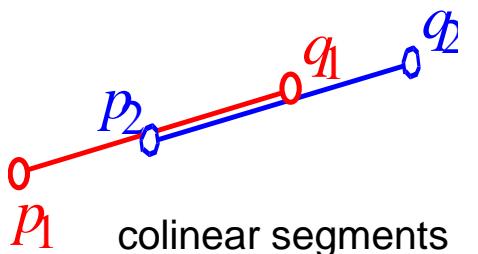
- Draw a horizontal line to the right of each point and extend it to infinity
- Count the number of times a line intersects the polygon.
  - even  $\Rightarrow$  point is outside
  - odd  $\Rightarrow$  point is inside



Degeneracy: What about points d and g (inside/outside)?

# Degeneracy

- Degeneracies are input configurations that involve tricky special cases.
- When implementing an algorithm, degeneracies should be taken care of separately -- the general algorithm might fail to work.
- E.g. whether two segments intersect, we have degeneracy if two segments are collinear.
- E.g. The general algorithm of checking for orientation would fail to distinguish whether the two segments intersect.



# Computational Aspects

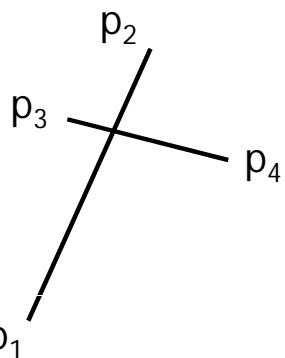
- We will use +,-,\*,< on real numbers.
- Avoid real divisions, causes instability.
- Avoid real equality, as the representation is approximation.
- Instead of `(x1 == x2)` USE `eq(x1,x2)`

```
#include <math.h>
#define eps          1.e-10
#define eq(x1,x2)    (fabs(x1-x2)< eps)
```

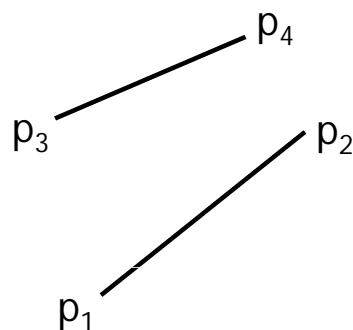
# 2 line intersection

From Cormen chapter 33

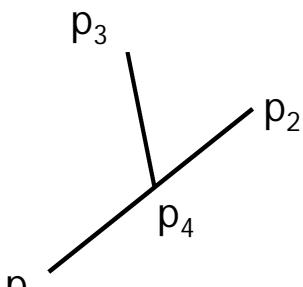
# Two Segments Intersect in 5 ways



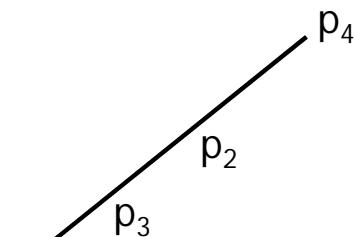
(a)  $p_1p_2, p_3p_4$  intersect



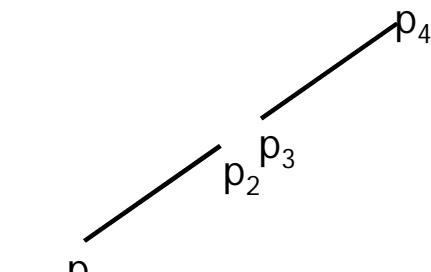
(b)  $p_1p_2, p_3p_4$  do not intersect



(c)  $p_1p_2, p_3p_4$  intersect



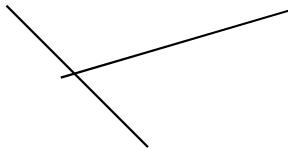
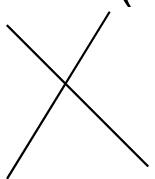
(d)  $p_1p_2, p_3p_4$  intersect



(e)  $p_1p_2, p_3p_4$  do not intersect

# Two line segments intersect

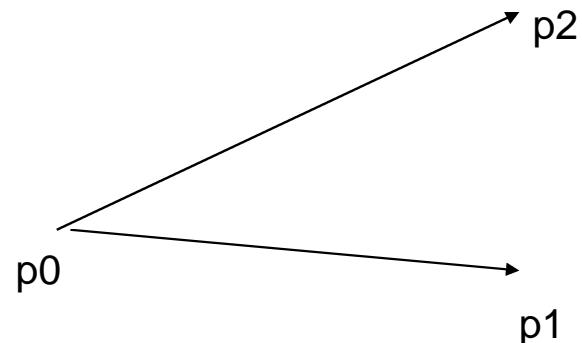
- check whether each segment **straddles** the line containing the other.
- A segment  $p_1p_2$  **straddles** a line if point  $p_1$  lies on one side of the line and point  $p_2$  lies on the other side.
- **Two line segments intersect** iff either hold:
  - Each segment straddles other.
  - endpoint of one segment lies on the other segment.  
(boundary case.)



# Direction

DIRECTION( $p_0, p_1, p_2$ )

**return**  $(p_2 - p_0) \times (p_1 - p_0)$



# On-Segment

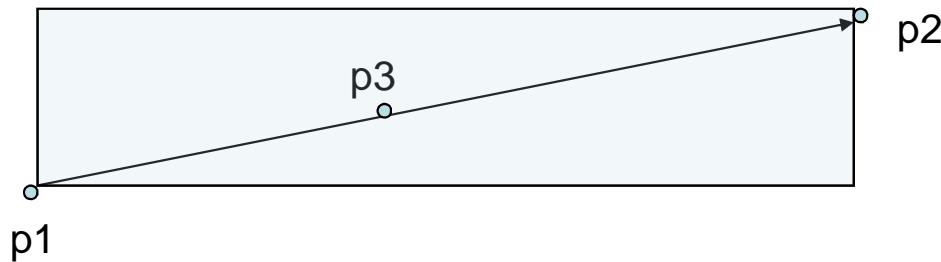
ON-SEGMENT( $p_1, p_2, p_3$ )

return

$$\min(x_1, x_2) \leq x_3 \leq \max(x_1, x_2)$$

&&

$$\min(y_1, y_2) \leq y_3 \leq \max(y_1, y_2)$$



# Algorithm

```
SEGMENTS-INTERSECT( $p_1, p_2, p_3, p_4$ )
1  $d_1 \leftarrow \text{DIRECTION}(p_3, p_4, p_1)$ 
2  $d_2 \leftarrow \text{DIRECTION}(p_3, p_4, p_2)$ 
3  $d_3 \leftarrow \text{DIRECTION}(p_1, p_2, p_3)$ 
4  $d_4 \leftarrow \text{DIRECTION}(p_1, p_2, p_4)$ 
5 if (( $d_1 > 0$  and  $d_2 < 0$ ) or ( $d_1 < 0$  and  $d_2 > 0$ ))
   and (( $d_3 > 0$  and  $d_4 < 0$ ) or ( $d_3 < 0$  and  $d_4 > 0$ ))
6 then return TRUE
7 elseif  $d_1 = 0$  and ON-SEGMENT( $p_3, p_4, p_1$ ) // p1 on p3p4?
8 then return TRUE
9 elseif  $d_2 = 0$  and ON-SEGMENT( $p_3, p_4, p_2$ ) // p2 on p3p4?
10 then return TRUE
11 elseif  $d_3 = 0$  and ON-SEGMENT( $p_1, p_2, p_3$ ) // p3 on p1p2?
12 then return TRUE
13 elseif  $d_4 = 0$  and ON-SEGMENT( $p_1, p_2, p_4$ ) // p4 on p1p2?
14 then return TRUE
15 else return FALSE
```

$(p_1 - p_3) \times (p_4 - p_3) < 0$

$(p_3 - p_1) \times (p_2 - p_1) > 0$

$(p_4 - p_1) \times (p_2 - p_1) < 0$

$(p_2 - p_3) \times (p_4 - p_3) > 0$

(a)

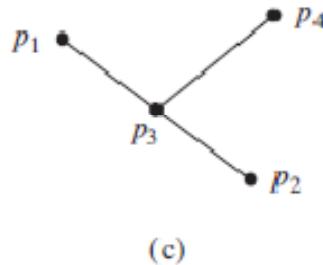
$(p_1 - p_3) \times (p_4 - p_3) < 0$

$(p_2 - p_3) \times (p_4 - p_3) < 0$

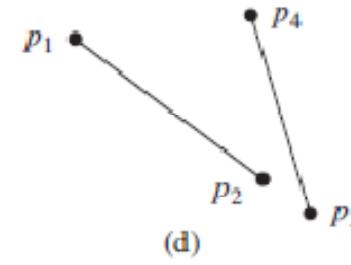
$(p_3 - p_1) \times (p_2 - p_1) > 0$

$(p_4 - p_1) \times (p_2 - p_1) < 0$

(b)



(c)



(d)

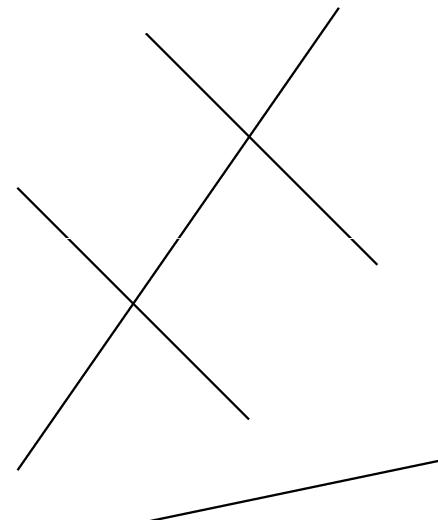
**Figure 33.3** Cases in the procedure SEGMENTS-INTERSECT. (a) The segments  $\overline{p_1p_2}$  and  $\overline{p_3p_4}$  straddle each other's lines. Because  $\overline{p_3p_4}$  straddles the line containing  $\overline{p_1p_2}$ , the signs of the cross products  $(p_3 - p_1) \times (p_2 - p_1)$  and  $(p_4 - p_1) \times (p_2 - p_1)$  differ. Because  $\overline{p_1p_2}$  straddles the line containing  $\overline{p_3p_4}$ , the signs of the cross products  $(p_1 - p_3) \times (p_4 - p_3)$  and  $(p_2 - p_3) \times (p_4 - p_3)$  differ. (b) Segment  $\overline{p_3p_4}$  straddles the line containing  $\overline{p_1p_2}$ , but  $\overline{p_1p_2}$  does not straddle the line containing  $\overline{p_3p_4}$ . The signs of the cross products  $(p_1 - p_3) \times (p_4 - p_3)$  and  $(p_2 - p_3) \times (p_4 - p_3)$  are the same. (c) Point  $p_3$  is colinear with  $\overline{p_1p_2}$  and is between  $p_1$  and  $p_2$ . (d) Point  $p_3$  is colinear with  $\overline{p_1p_2}$ , but it is not between  $p_1$  and  $p_2$ . The segments do not intersect.

# N line intersection

From Cormen chapter 33

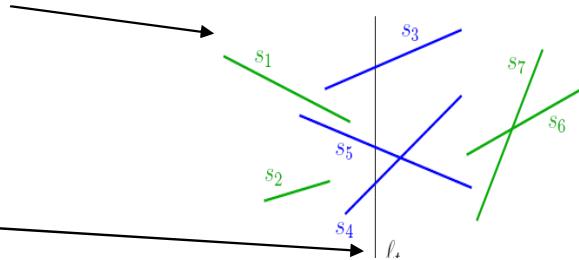
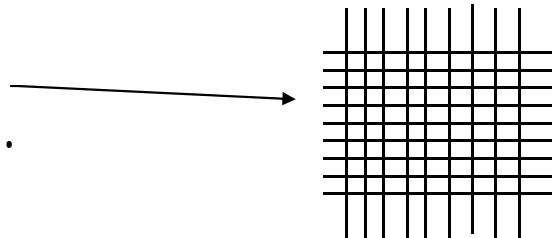
# Segment intersection

- Given: a set of  $n$  distinct segments  $s_1 \dots s_n$ , represented by coordinates of endpoints
- Detect if any pair  $s_i \neq s_j$  intersects.
- Report all intersection.



# N segment intersection

- Brute force, check all  $n \times n$  intersections in  $O(n^2)$  time.
- worst-case is  $O(n^2)$
- However: if the number of *intersections*  $m$  is usually small, we can compute in  $O(n \log n)$ .
- Idea: **Sweep** left to right, looking for intersections.
- Ignoring segments that are too far away for speedup.

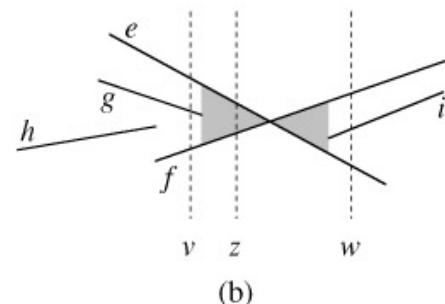
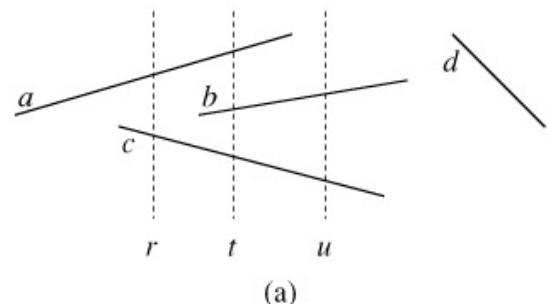


# Sweeping

- **Sweeping**: An imaginary **sweep line** passes through the given set of geometric objects.
- Two segments  $s_1, s_2$  are **comparable** if some **vertical sweep line**  $v(x)$  intersects both of them.
- $(s_1 >_x s_2)$  if  $s_1$  is above  $s_2$  at  $v(x)$ .

# Order segments

- For example, we have the relationships  $a >_r c$ ,  $a >_t b$ ,  $b >_t c$ ,  $a >_t c$ , and  $b >_u c$ . Segment  $d$  is not comparable with any other segment. When segments  $e$  and  $f$  intersect, their orders are reversed: we have  $e >_v f$  but  $f >_w e$ .



# Moving the sweep line

- Sweeping algorithms typically manage two sets of data:
  - The **sweep-line status** gives the relationships among the objects intersected by the sweep line.
  - The **event-point schedule** is a sequence of x-coordinates, ordered from left to right, that defines the halting positions of the sweep line.
  - We call each such halting position an **event point**.
  - Changes to the sweep-line status occur only at event points.
- The sweep-line status is a total order  $T$ , for which we require the following operations:
  - **INSERT( $T, s$ )**: insert segment  $s$  into  $T$ .
  - **DELETE( $T, s$ )**: delete segment  $s$  from  $T$ .
  - **ABOVE( $T, s$ )**: return the segment immediately above segment  $s$  in  $T$ .
  - **BELLOW( $T, s$ )**: return the segment immediately below segment  $s$  in  $T$ .
  - If there are  $n$  segments in the input, each of the above operations take  $O(\lg n)$  time using **red-black trees**.

# Segment-intersection pseudo-code

**ANY-SEGMENTS-INTERSECT( $S$ )**

$T \leftarrow \{ \}$

EP = sort the endpoints of the segments in  $S$  from left to right, breaking ties by putting left endpoints before right endpoints and breaking further ties by putting points with lower  $y$ -coordinates first

for each point  $p$  in EP (the sorted list of endpoints)

    if  $p$  is the left endpoint of a segment  $s$  then INSERT( $T, s$ )

        if (ABOVE( $T, s$ ) exists and intersects  $s$ ) or

            (BELOW( $T, s$ ) exists and intersects  $s$ ) then return TRUE

        if  $p$  is the right endpoint of a segment  $s$  then

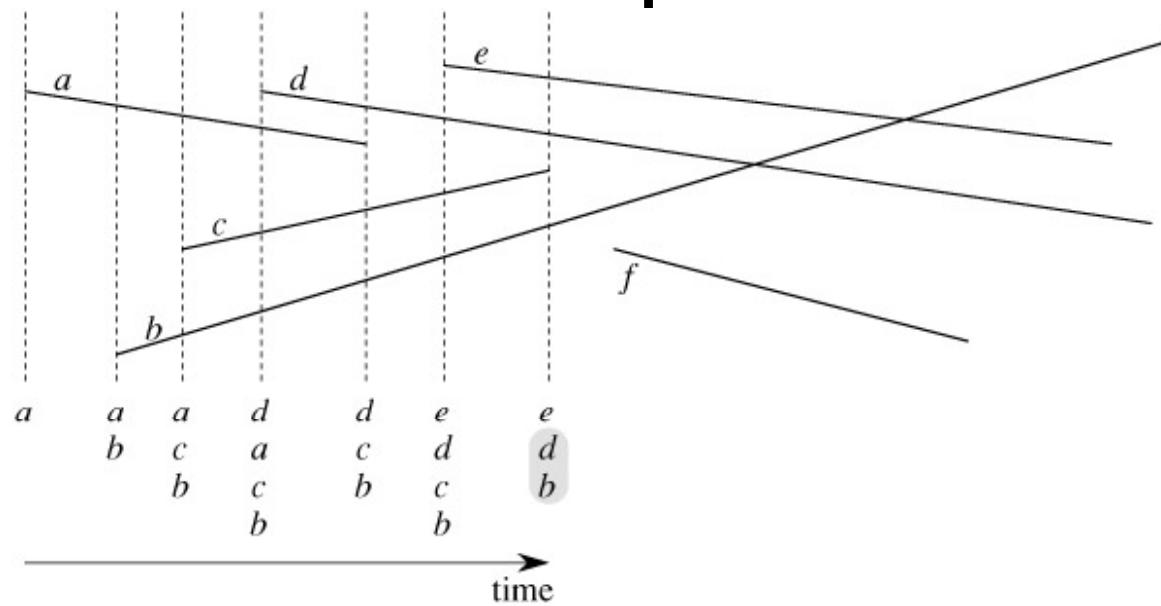
            if both ABOVE( $T, s$ ) and BELOW( $T, s$ ) exist and

                ABOVE( $T, s$ ) intersects BELOW( $T, s$ ) then return TRUE

            DELETE( $T, s$ )

return FALSE // no intersections

# Example



Each dashed line is the *sweep line* at an event point, and the ordering of segment names below each sweep line is the total order *T* at the end of the *for* loop in which the corresponding event point is processed. The intersection of segments *d* and *b* is found when segment *c* is deleted.

# Running time

- If there are  $n$  segments in set  $S$ , then ANY-SEGMENTS-INTERSECT runs in time  $O(n \lg n)$ .
- Line 2 takes  $O(n \lg n)$  time, using merge sort or heapsort. Since there are  $2n$  event points, the **for** loop iterates at most  $2n$  times.
- Each iteration takes  $O(\lg n)$  time, since each red-black-tree operation takes  $O(\lg n)$  time and, using the method of line intersection, each intersection test takes  $O(1)$  time.
- The total time is thus  $O(n \lg n)$ .

# Closest pair of points

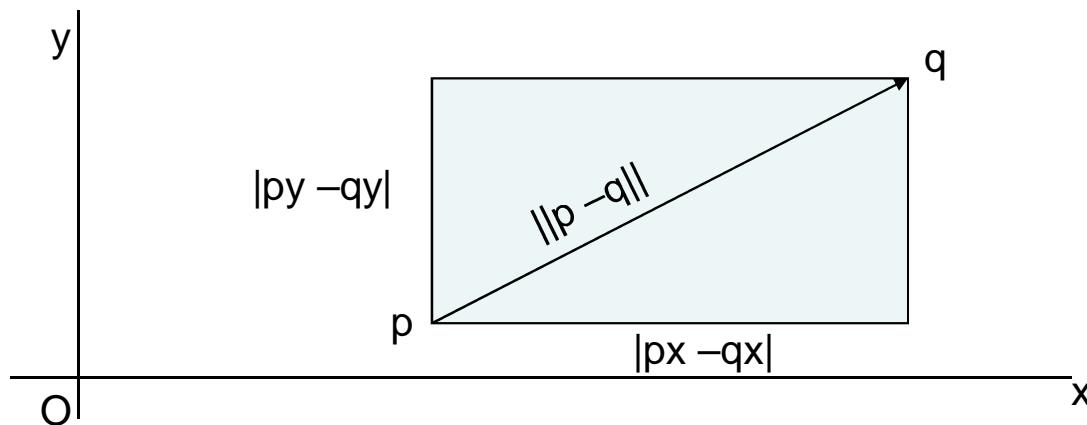
From Cormen chapter 33

# Distance

**Euclidean distance:**  $\|p - q\|$  is the shortest distance between points p and q in the plane,  $ed(p,q) = \sqrt{ \text{sqr}(p_x - q_x) + \text{sqr}(p_y - q_y) }$

**Mahanttan distance:**

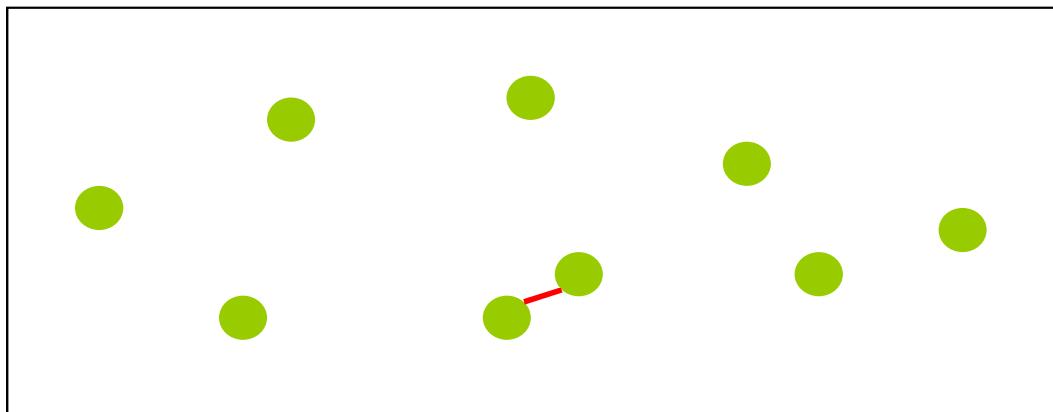
$$M(p,q) = |px - qx| + |py - qy|$$



# The closet pair problem

**Given:** a set of points  $P=\{p_1 \dots p_n\}$  in the plane, such that  $p_i=(x_i, y_i)$ .

**Find** a pair  $p_i \neq p_j$  with minimum  $\|p_i - p_j\|$ ,  
where  $\|p - q\| = \sqrt{\text{sqr}(p_x - q_x) + \text{sqr}(p_y - q_y)}$

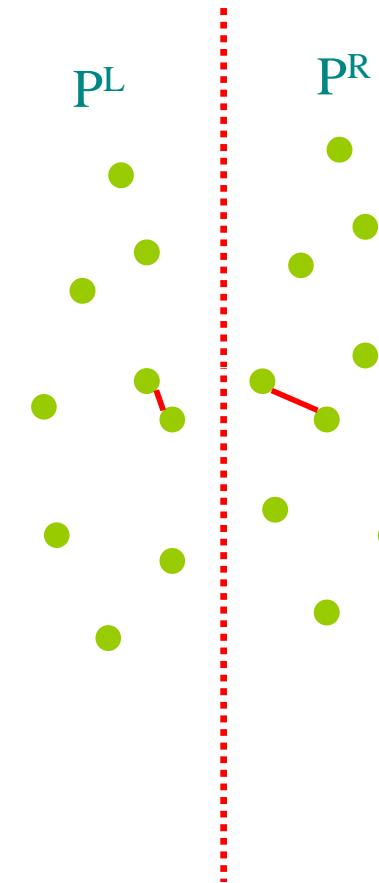


# Closest Pair Brute force

- Find a closest pair among  $p_1 \dots p_n$
- Easy to do in  $O(n^2)$  time
  - For all  $p_i \neq p_j$ , compute  $\|p_i - p_j\|$  and choose the minimum
- Want  $O(n \log n)$  time

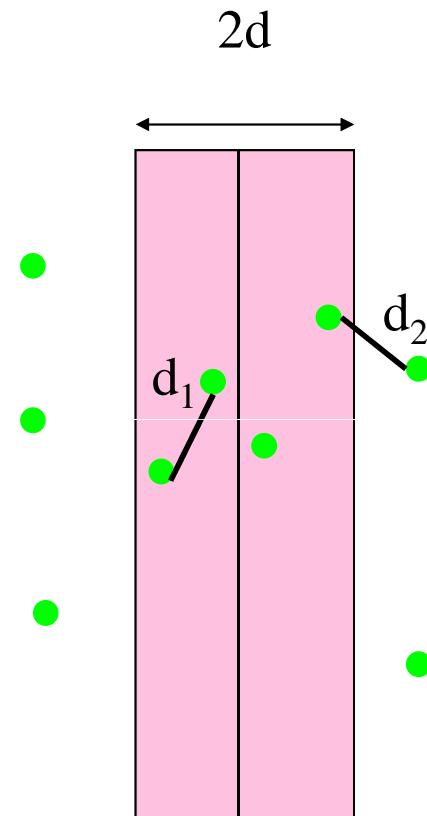
# Divide and conquer

- Divide:
  - Compute the median of  $x$ -coordinates
  - Split the points into  $P^L$  and  $P^R$ , each of size  $n/2$
- Conquer: compute the closest pairs for  $P^L$  and  $P^R$
- Merge the results (the hard part)



# Merge

- Let  $d = \min(d_1, d_2)$
- Observe:
  - Need to check only pairs which cross the dividing line
  - Only interested in pairs within distance  $< d$
- Suffices to look at points in the  $2d$ -width strip around the median line

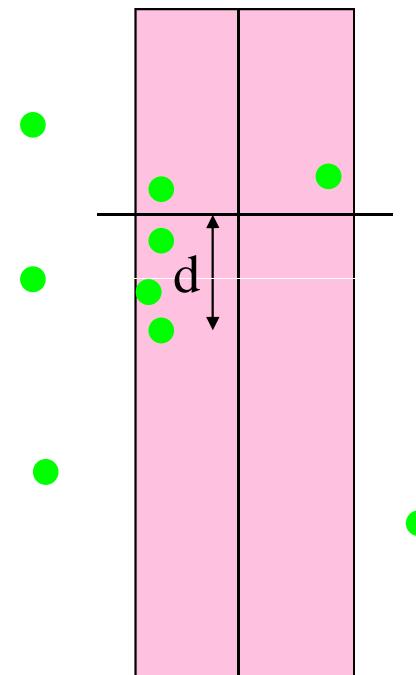


# Scanning the strip

- $S = \text{Points in the strip.}$   
 $Q = \text{Sort } S \text{ by their y-coordinates, Let } q_i = (x_i, y_i)$   
 $Q = \{q_1 \dots q_k : k \leq n, \text{ sorted points in strip by Y-coord }\}$
- $d_{\min} = d$
- **For**  $i=1$  **to**  $k$  **do**
  - $j=i-1$
  - While** ( $y_i - y_j < d$ ) **do** // is this  $O(n^2)$  ?
    - If**  $\|q_i - q_j\| < d$  **then**
      - $d_{\min} = \|q_i - q_j\|$
      - $j--$
  - Report  $d_{\min}$  (and the corresponding pair)

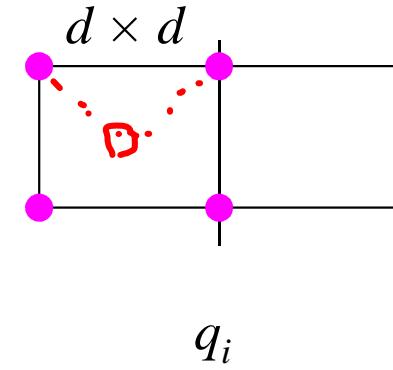
# Merge complexity is not $O(n^2)$

- Can we have  $O(n^2)$   $q_j$ 's that are within distance  $d$  from  $q_i$  ?
- NO
- **Theorem:** there are at most 7  $q_j$ 's such that  $y_i - y_j \leq d$ .
- Proof by packing argument



# Proof: merge complexity.

- **Theorem:** there are at most 7  $q_j$ 's such that  $y_i - y_j \leq d$ .
- **Proof:** Each such  $q_j$  must lie either in the left or in the right  $d \times d$  square
- Note that within each square, all points have distance  $\geq d$  from others
- We can pack at most 4 such points into one square, so we have 8 points total (incl.  $q_i$ )

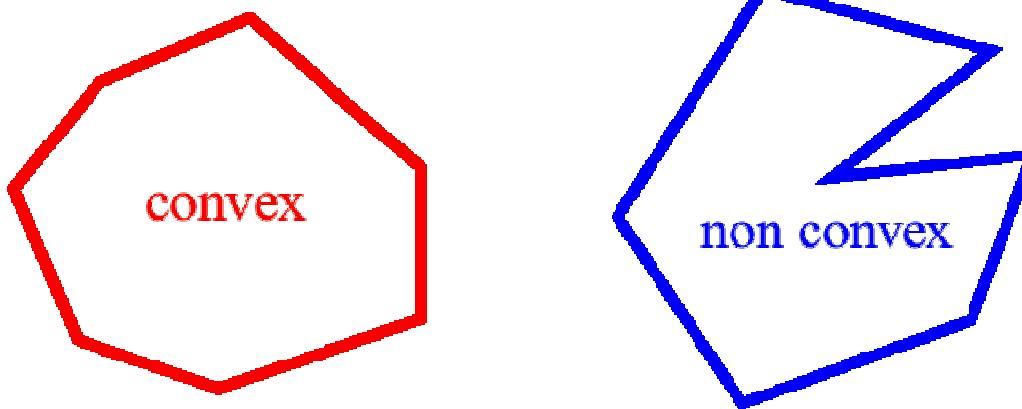


# Convex Hull

From Cormen chapter 33

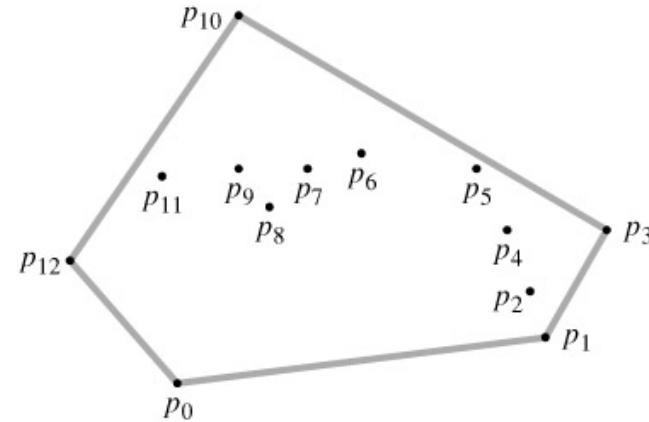
# What is convex

- A polygon  $P$  is said to be **convex** if:
  - $P$  is non-intersecting; and
  - for any two points  $p$  and  $q$  on the boundary of  $P$ , segment  $(p,q)$  lies entirely inside  $P$

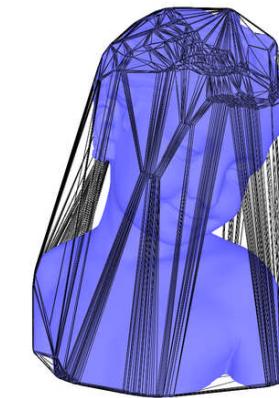
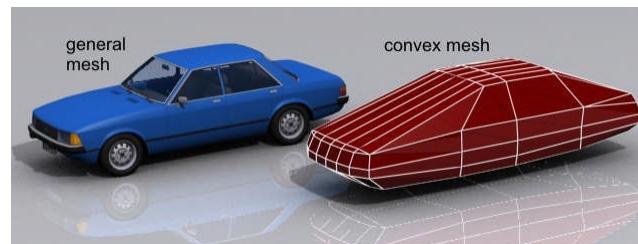
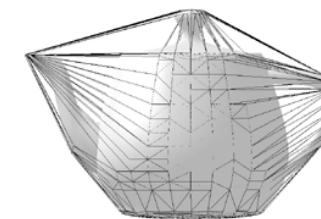
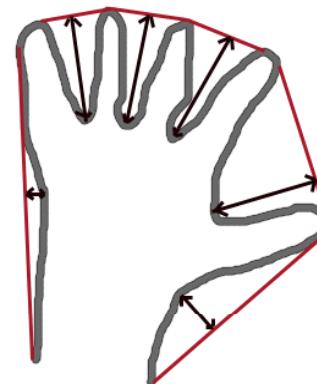
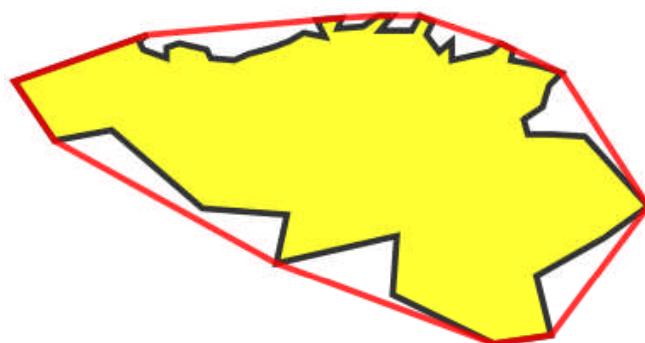


# convex hull

- The **convex hull** CH of a set Q of points is the smallest convex polygon P for which each point in Q is either on the boundary of P or in its interior.
- Example.  $\text{CH}(Q) = \{p_0, p_1, p_3, p_{10}, p_{12}\}$

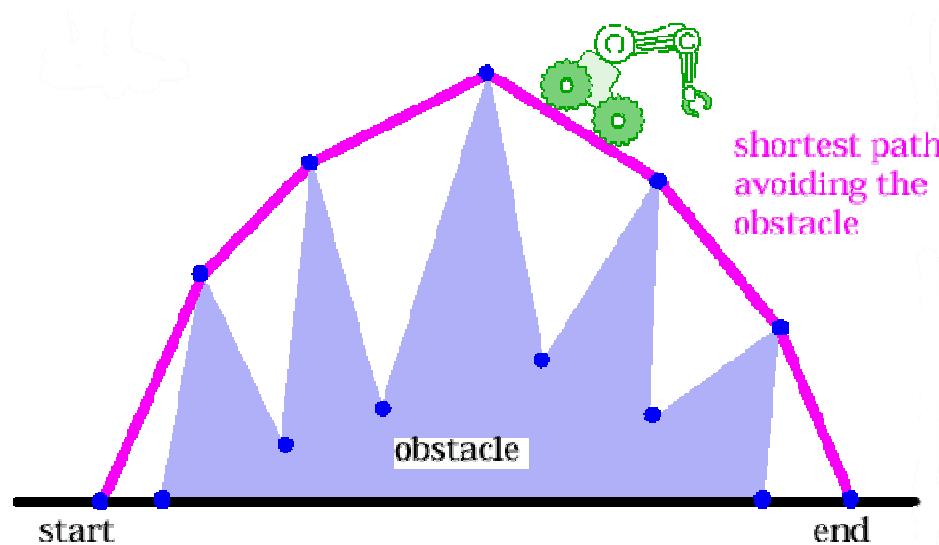


# Convex Hull examples



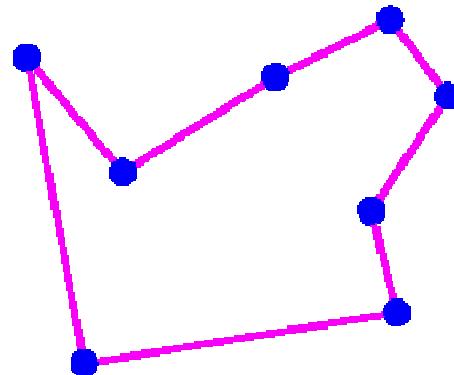
# Robot motion planning

- *In motion planning for robots, sometimes there is a need to compute convex hulls.*



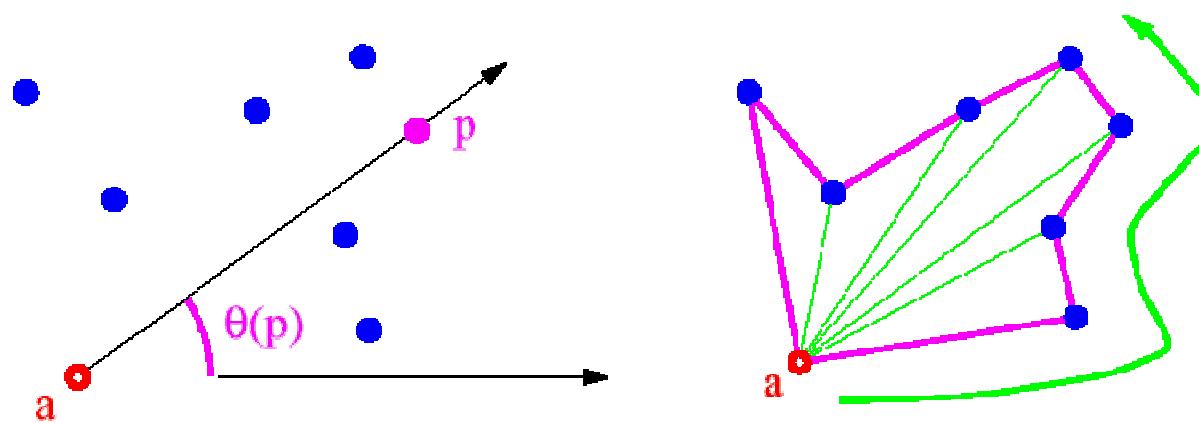
# Graham Scan

- ***Graham Scan algorithm.***
  - Phase 1: Solve the problem of finding the non-crossing closed path visiting all points



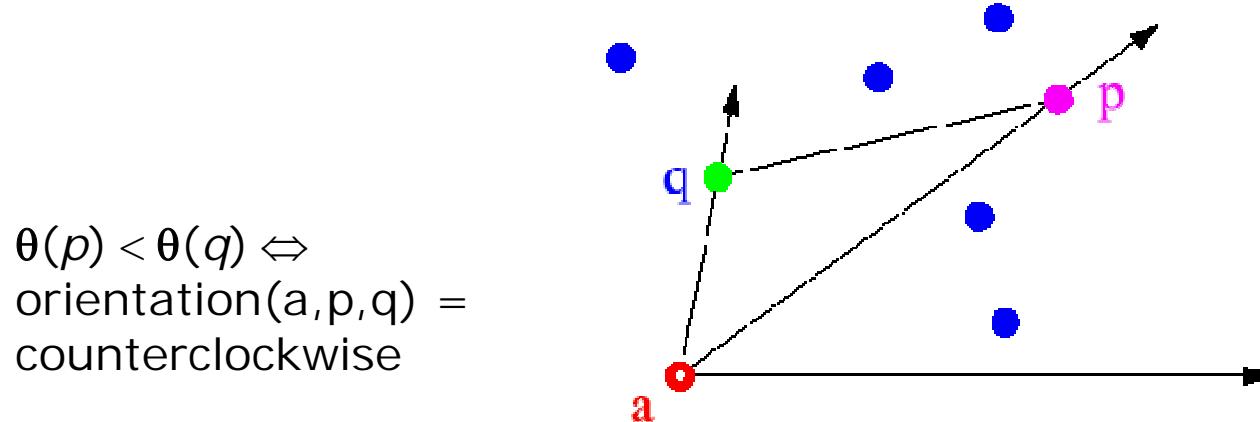
# Finding non-crossing path

- *How do we find such a non-crossing path:*
  - Pick the bottommost point  $a$  as the anchor point
  - For each point  $p$ , compute the angle  $\theta(p)$  of the segment  $(a,p)$  with respect to the  $x$ -axis.
  - Traversing the points by **increasing angle** yields a simple closed path



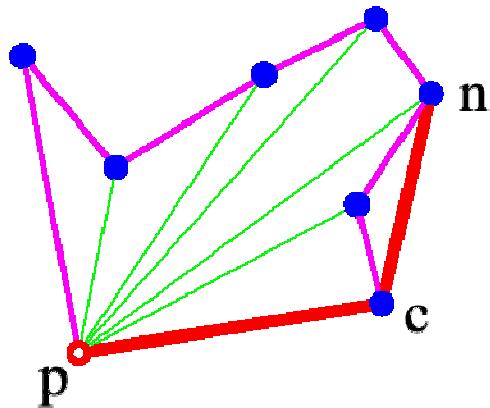
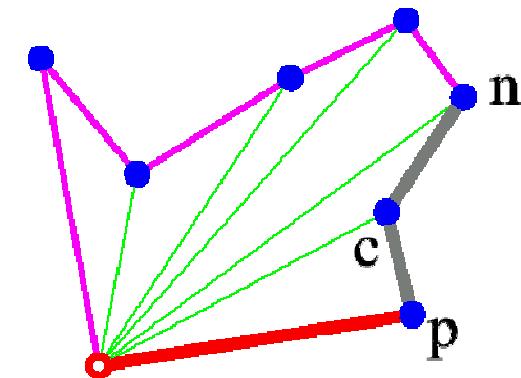
# Sorting by angle

- *How do we sort by increasing angle?*
  - *Observation:* We do not need to compute the actual angle!
  - We just need to compare them for sorting



# Rotational sweeping

- *Phase 2 of Graham Scan:  
Rotational sweeping*
  - The anchor point and the first point in the polar-angle order have to be in the hull
  - Traverse points in the sorted order:
    - Before including the next point  $n$  check if the new added segment makes a right turn
    - If not, keep discarding the previous point ( $c$ ) until the right turn is made



# Implementation and analysis

- *Implementation:*
  - Stack to store vertices of the convex hull
- *Analysis:*
  - Phase 1:  $O(n \log n)$ 
    - points are sorted by angle around the anchor
  - Phase 2:  $O(n)$ 
    - each point is pushed into the stack once
    - each point is removed from the stack at most once
  - Total time complexity  $O(n \log n)$

# Graham's scan algorithm

- **Graham's scan** solves the convex-hull problem by maintaining a **stack S** of candidate points.
- Each point of the input set  $Q$  is pushed once onto the stack, and the points that are not vertices of  $\text{CH}(Q)$  are eventually popped from the stack.
- When the algorithm terminates, stack  $S$  contains exactly the vertices of  $\text{CH}(Q)$ , in counterclockwise order of their appearance on the boundary.
- Graham scan takes  **$O(n \lg n)$**

# GRAHAM-SCAN

- The procedure GRAHAM-SCAN takes as input a set  $Q$  of points, where  $|Q| \geq 3$ .
- It calls the functions  $\text{TOP}(S)$ , which returns the point on top of stack  $S$  without changing  $S$ , and  $\text{NEXT-TO-TOP}(S)$ , which returns the point one entry below the top of stack  $S$  without changing  $S$ .
- The stack  $S$  returned by GRAHAM-SCAN contains, from bottom to top, exactly the vertices of  $\text{CH}(Q)$  in counterclockwise order.

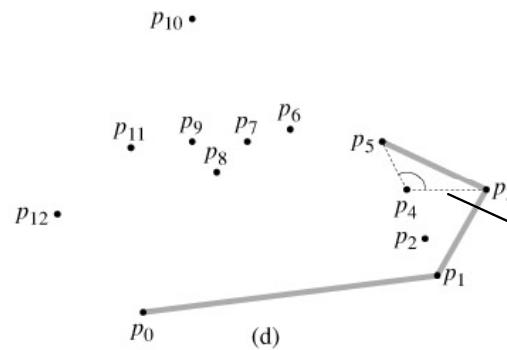
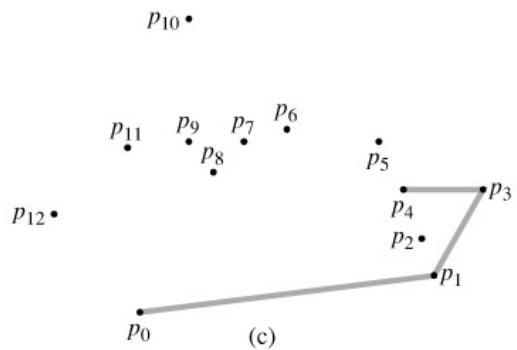
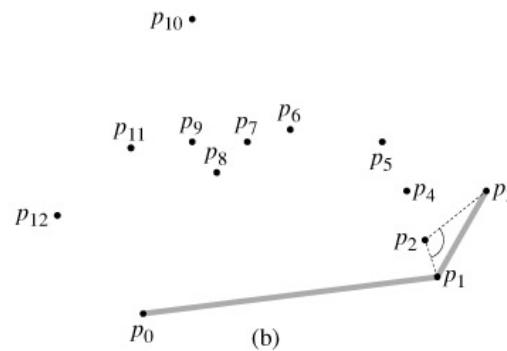
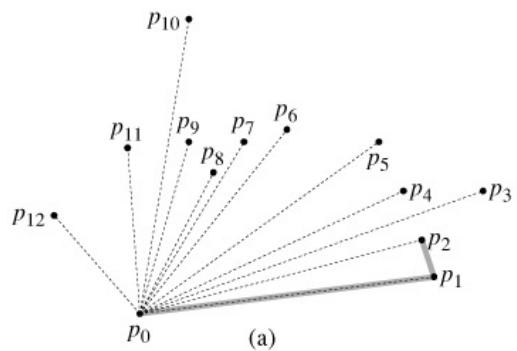
# Graham scan algorithm

## GRAHAM-SCAN(Q)

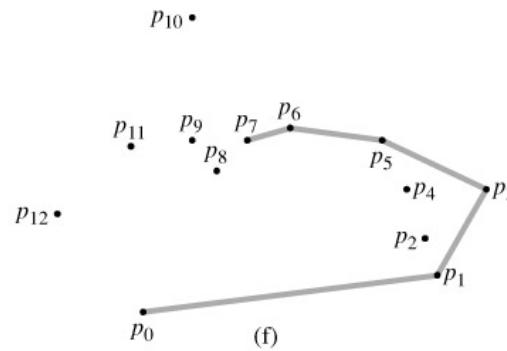
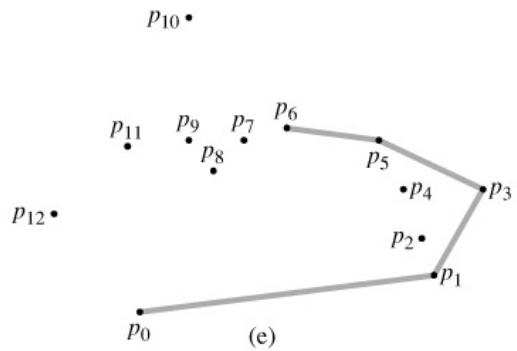
- 1 let  $p_0$  be the point in  $Q$  with the min  $y$ -coord,  
or the leftmost such point in case of a tie
- 2 let  $[p_1, p_2, \dots, p_m]$  be the remaining points in  $Q$ ,  
sorted by polar angle in counterclockwise order around  $p_0$   
(if more than one point has the same angle,  
keep only the farthest point from  $p_0$ )
- 3.. PUSH( $p_0$ ,  $S$ ) ; PUSH( $p_1$ ,  $S$ ) ; PUSH( $p_2$ ,  $S$ )
- 7 **for**  $i \leftarrow 3$  **to**  $m$
- 8 **do while** the angle formed by points NEXT-TO-TOP( $S$ ), TOP( $S$ ),  
and  $p_i$  makes a nonleft turn
- 9 **do {** POP( $S$ ) **}**
- 10 PUSH( $p_i$ ,  $S$ )
- 11 **return**  $S$

## Illustration:

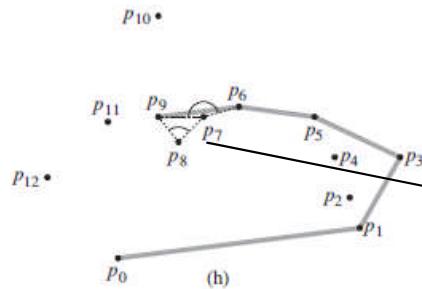
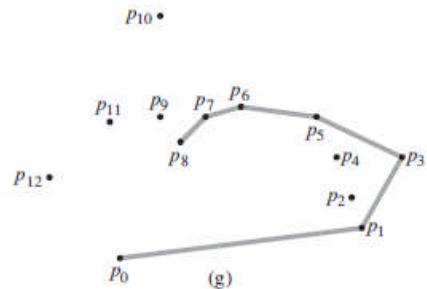
sorts the remaining points of  $Q$  by polar angle relative to  $p_0$ , using the method comparing cross products



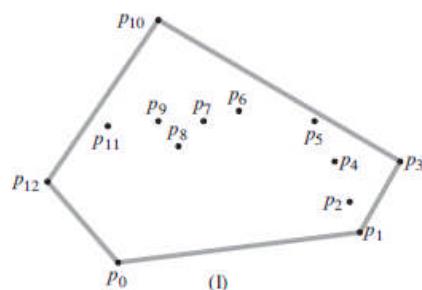
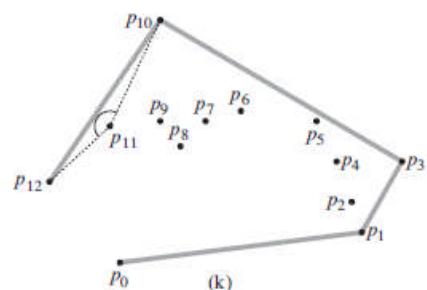
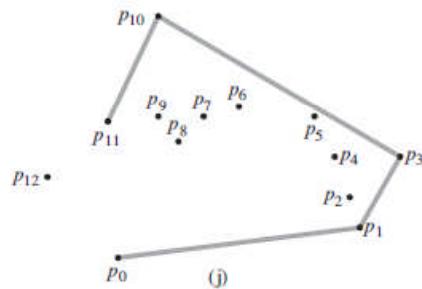
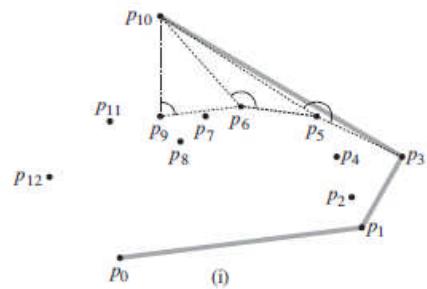
Right turn,  
pop( $p_4$ )



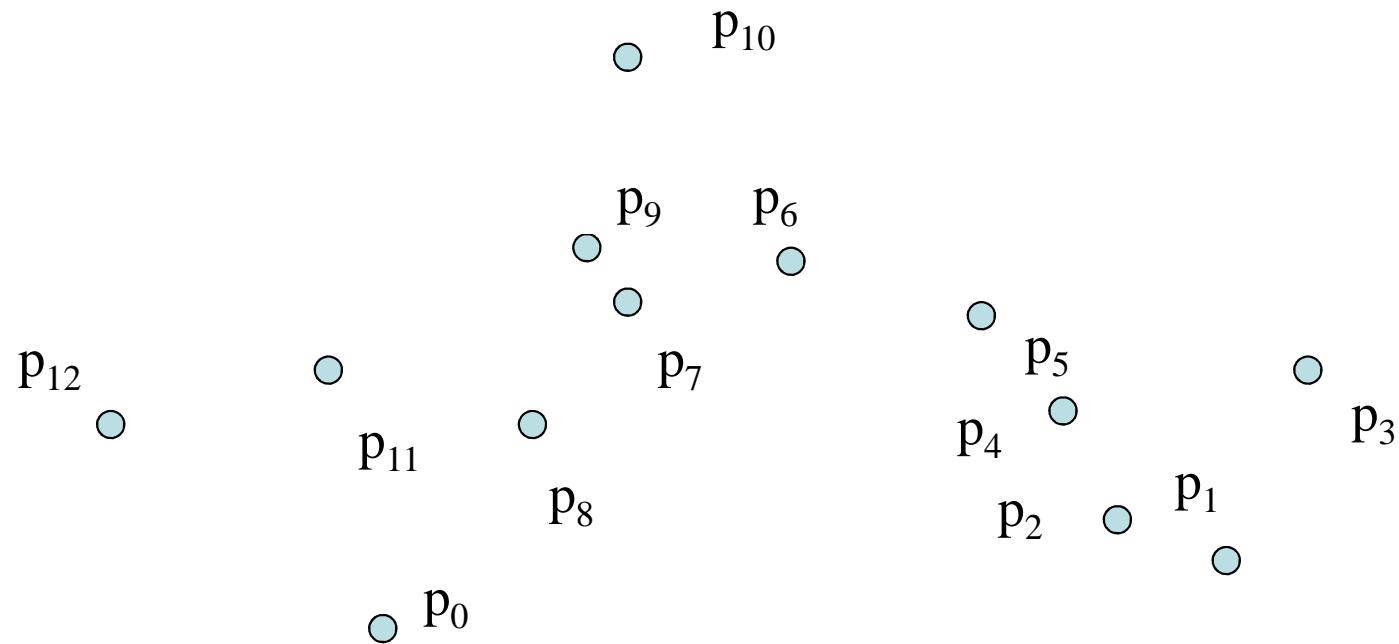
# Illustration continued



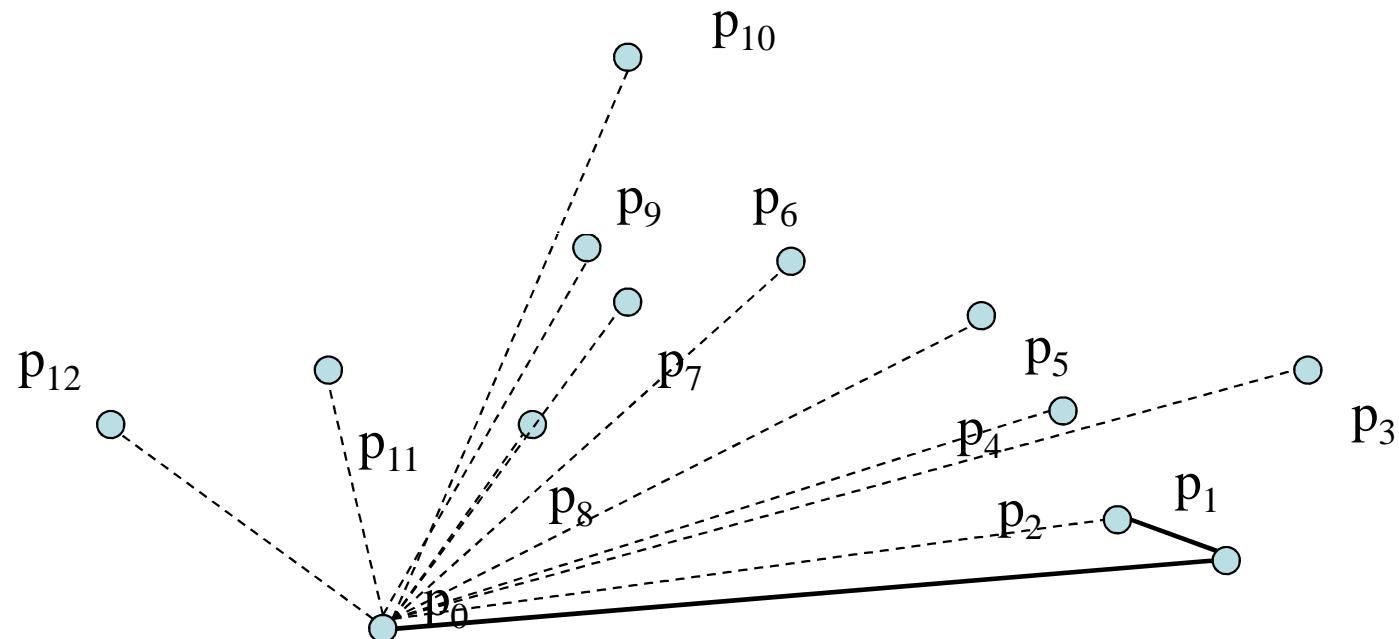
Step 8: while ..  
pop p8,  
pop p7



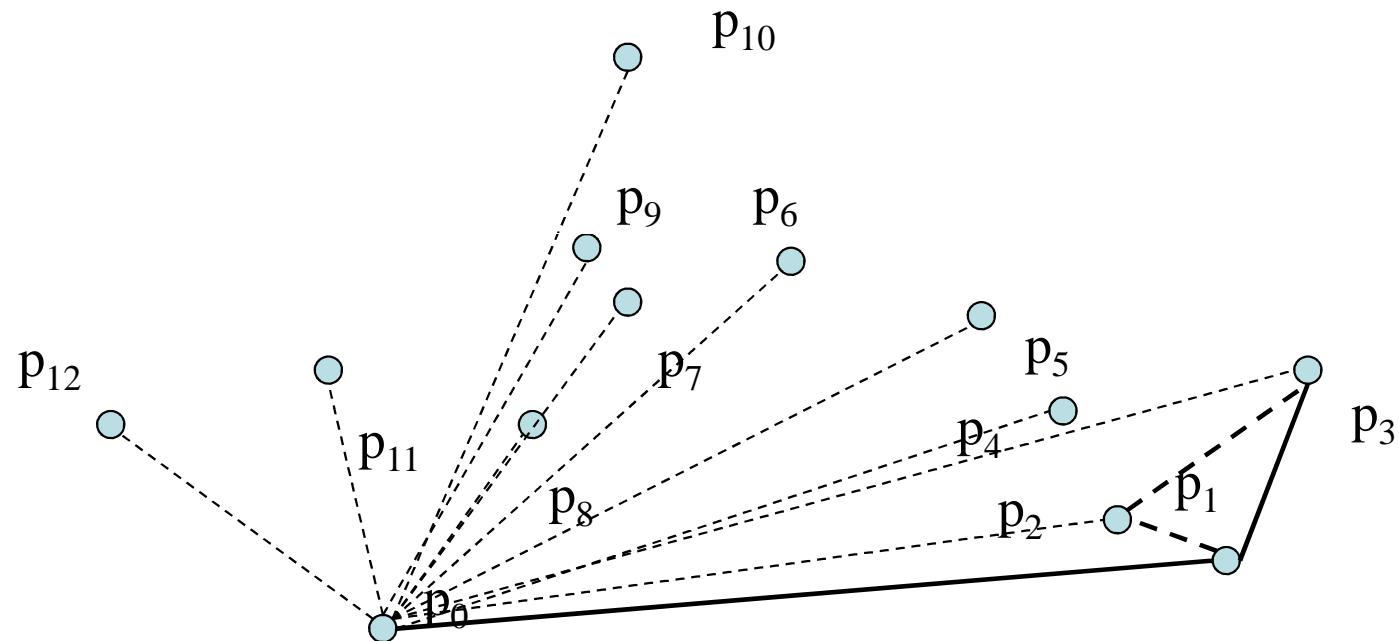
# Graham Scan - Example



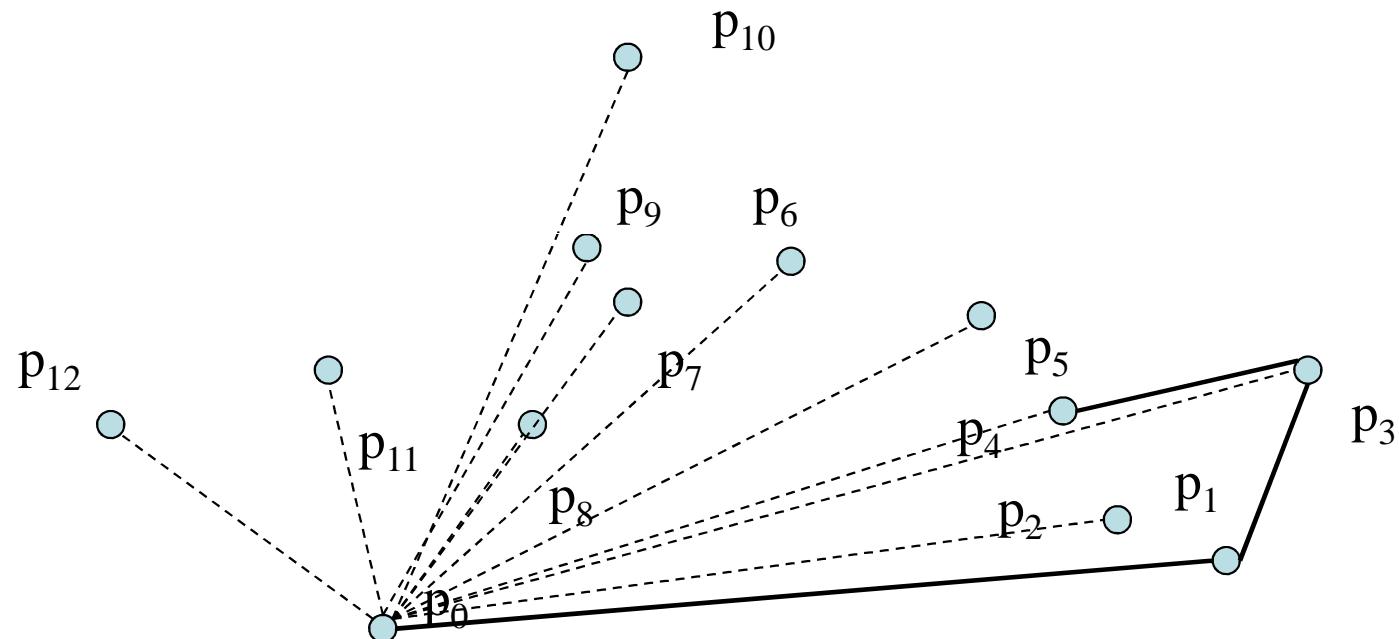
# Graham Scan - Example



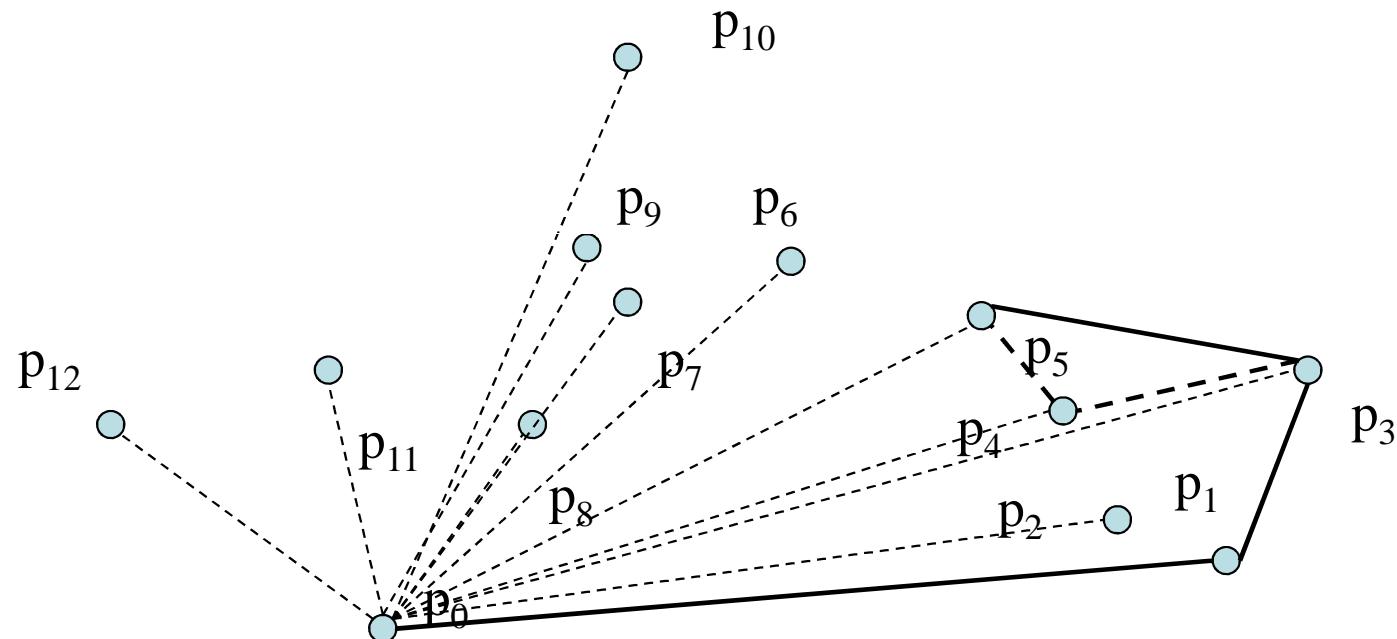
# Graham Scan - Example



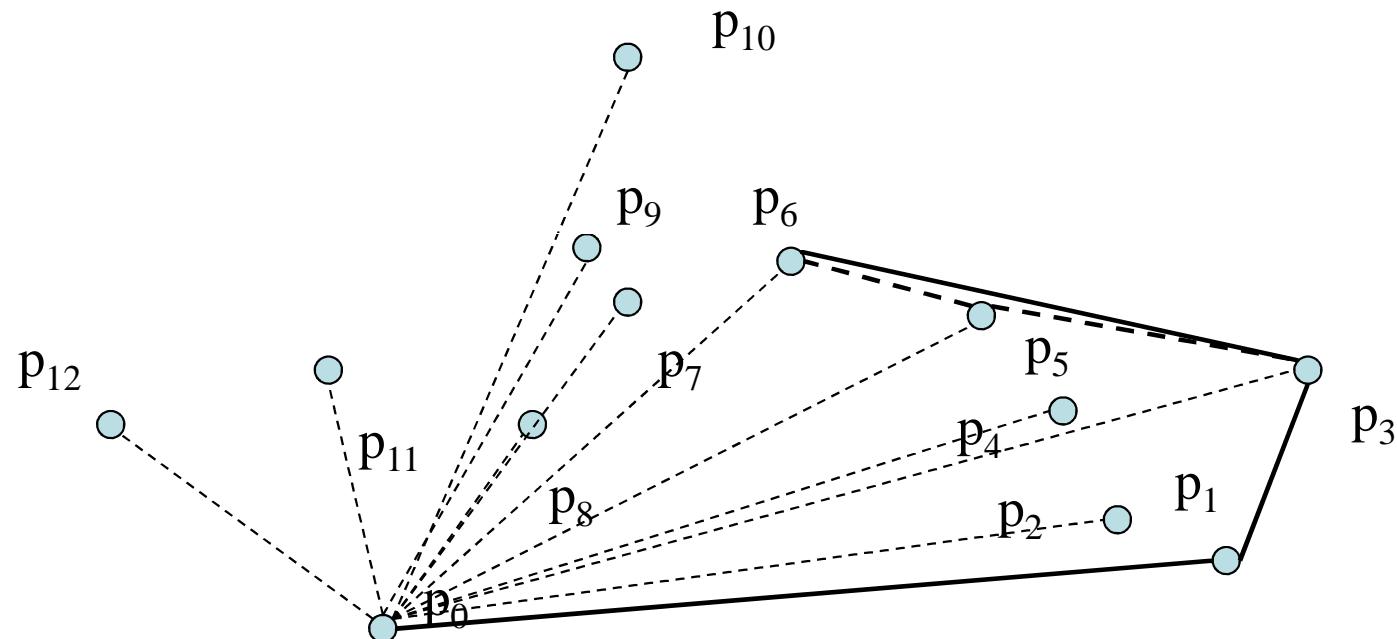
# Graham Scan - Example



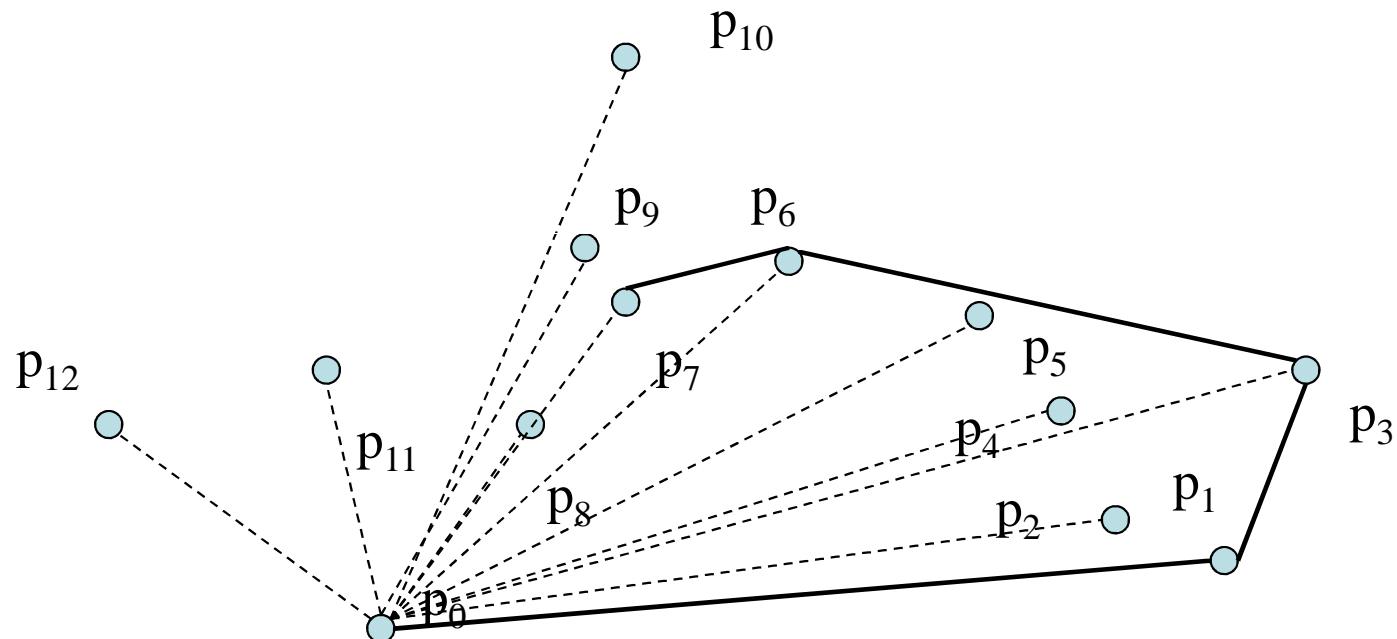
# Graham Scan - Example



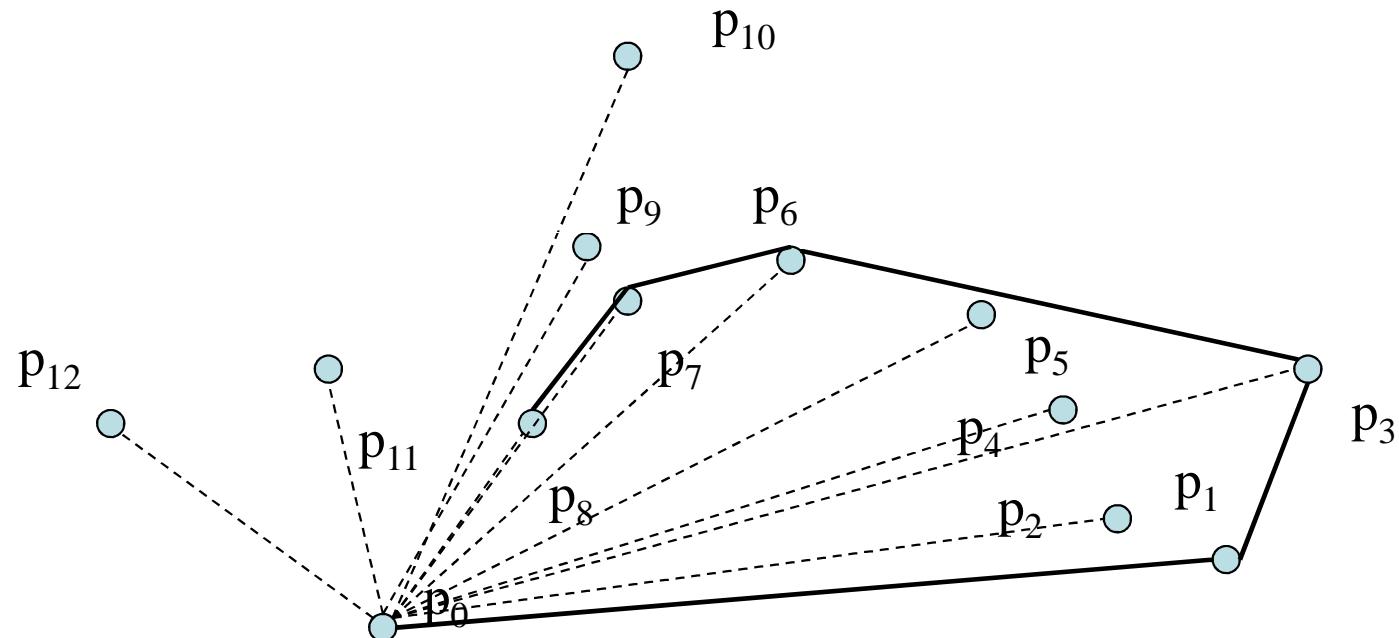
# Graham Scan - Example



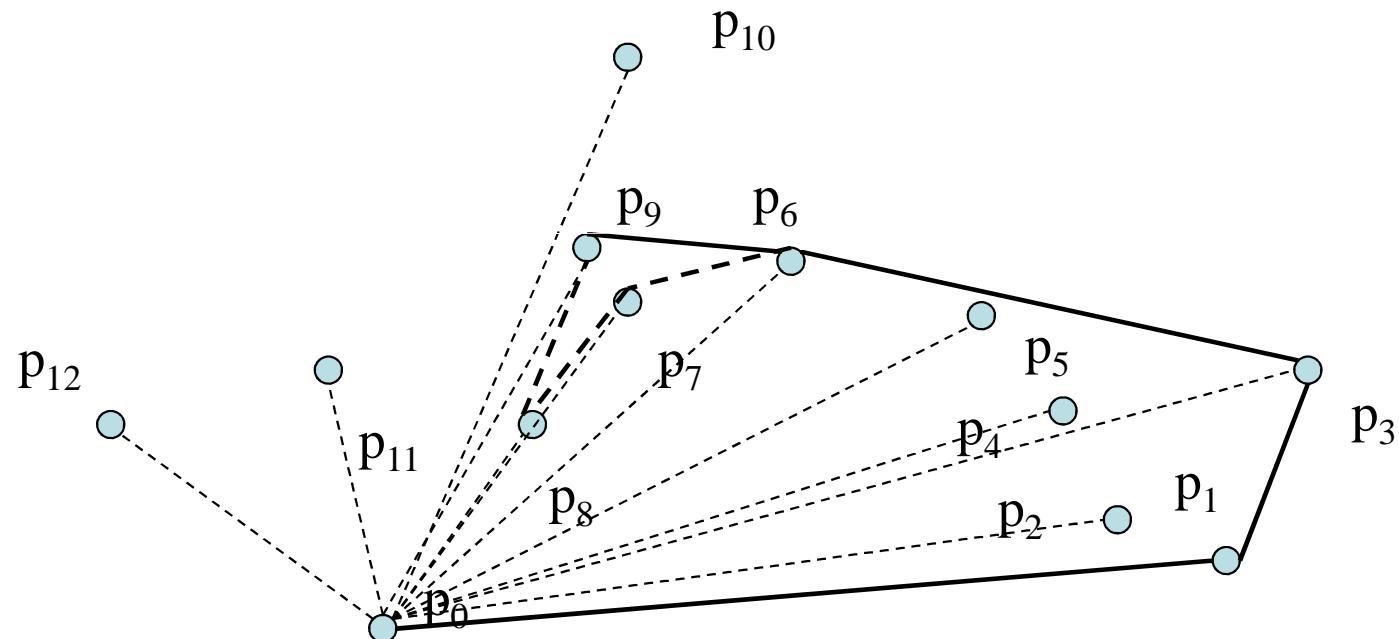
# Graham Scan - Example



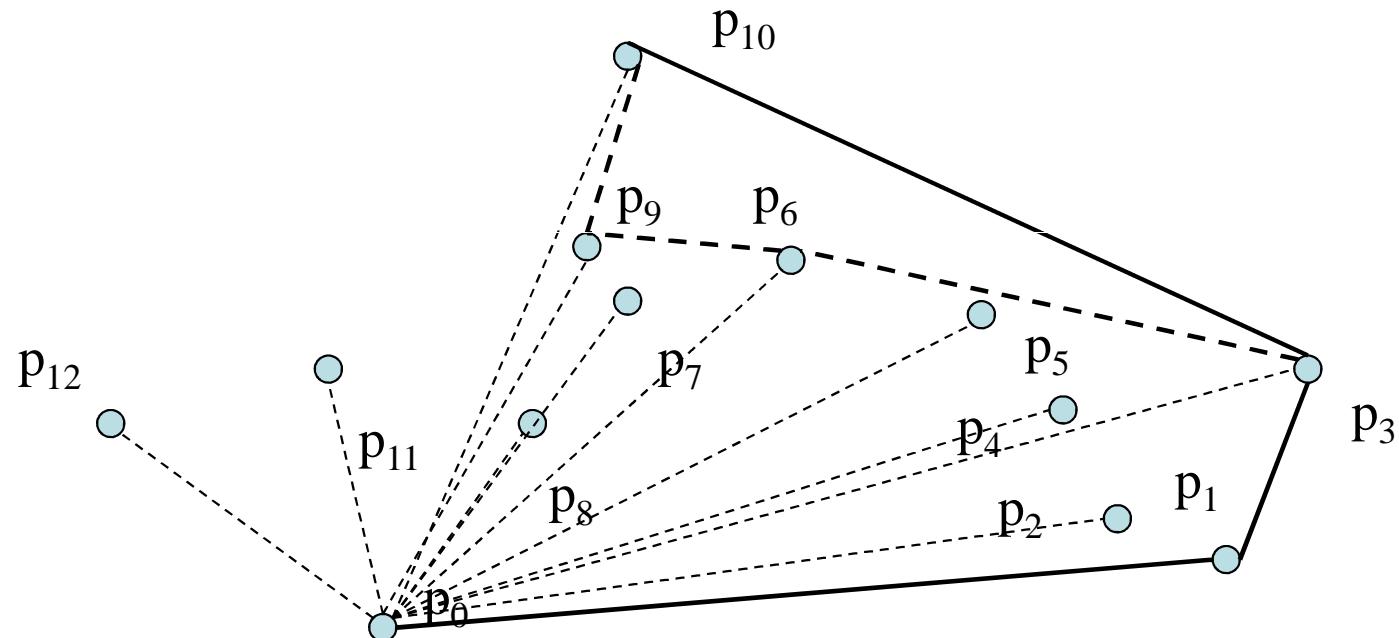
# Graham Scan - Example



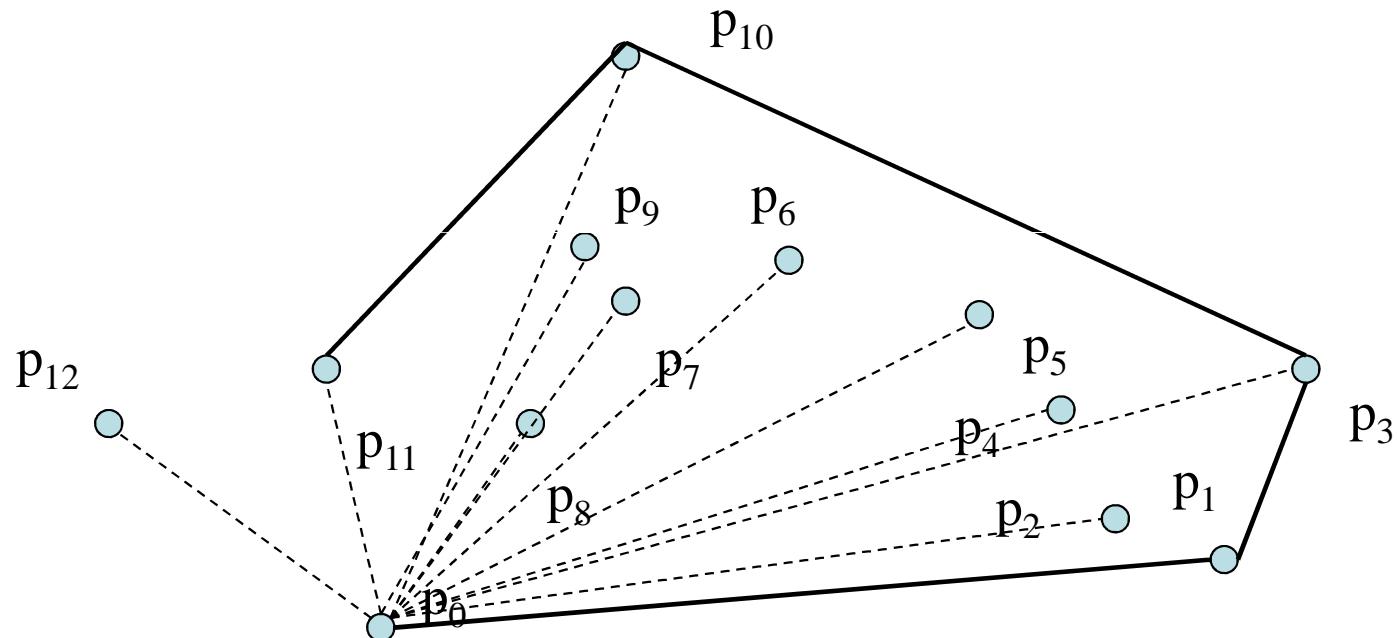
# Graham Scan - Example



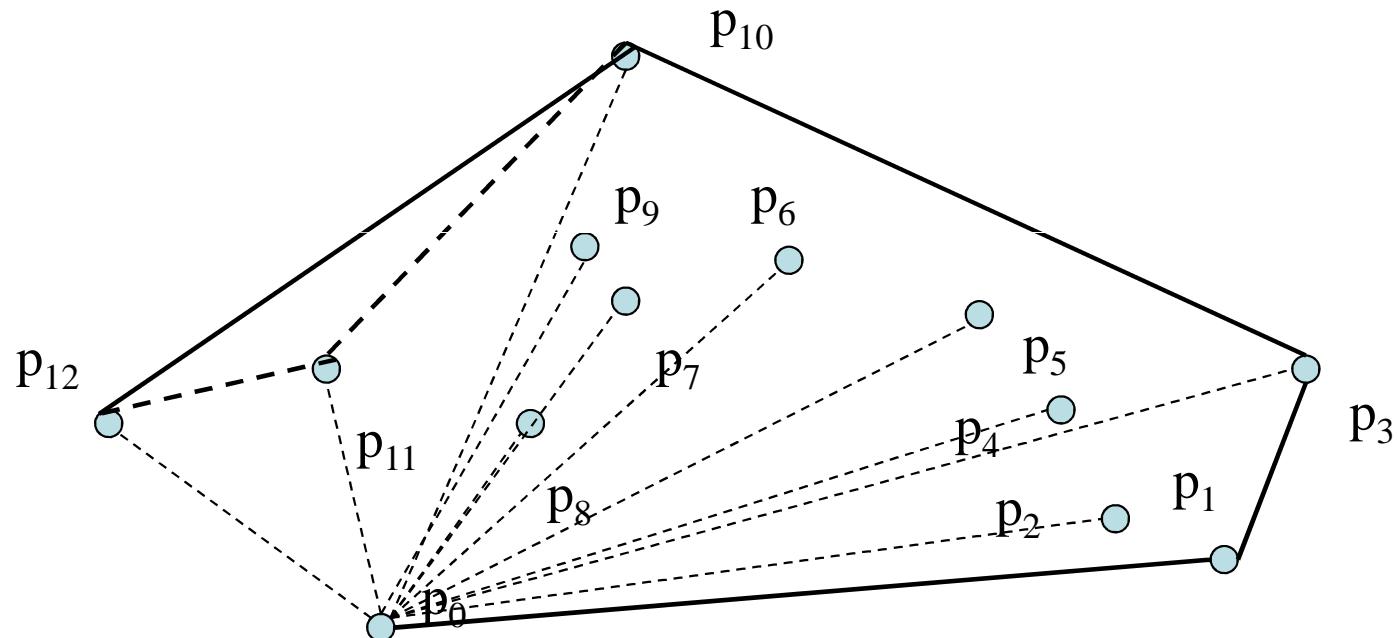
# Graham Scan - Example



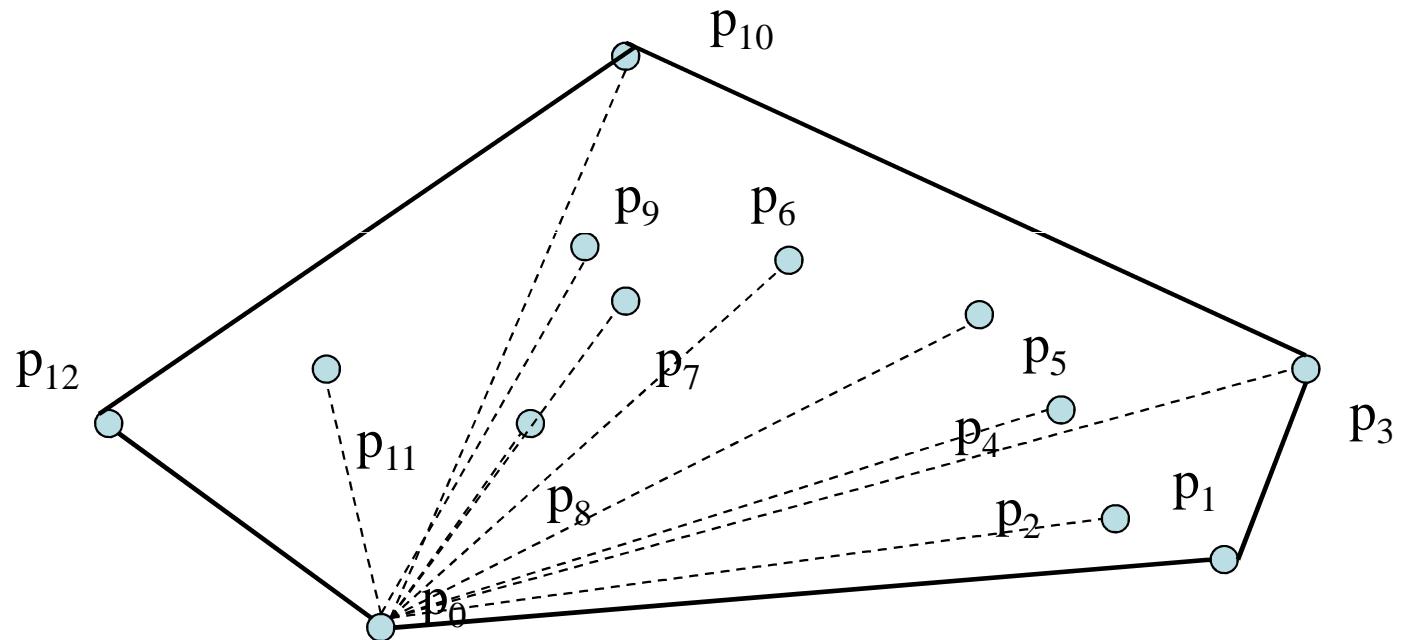
# Graham Scan - Example



# Graham Scan - Example

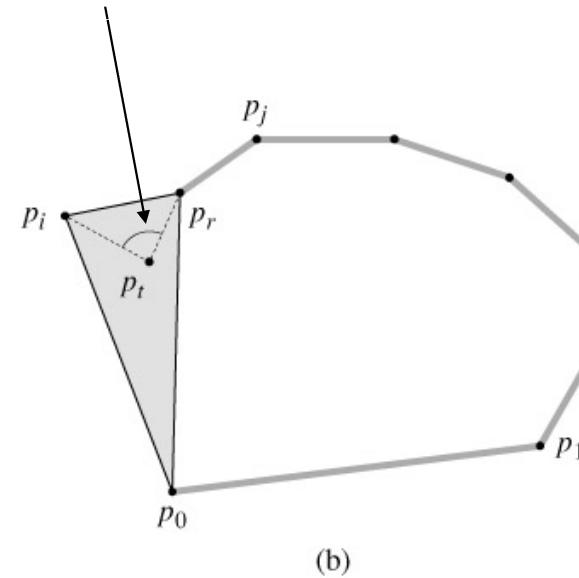
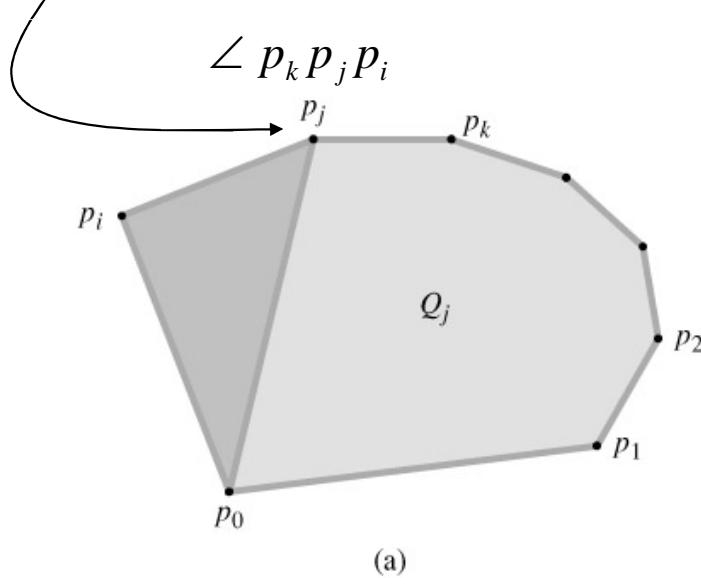


# Graham Scan - Example



# Proof.

- Focus on this moment just before  $p_i$  is pushed, to decide whether  $p_j$  stays in the CH
- Because  $p_i$ 's polar angle relative to  $p_0$  is greater than  $p_j$ 's polar angle, and because the angle  $p(k,j,i)$  makes a left turn (otherwise we would have popped  $p_j$ )



# Running time

- Line 1 takes  $\Theta(n)$  time.
- Line 2 takes  $O(n \lg n)$  time, using merge sort or heapsort to sort the polar angles and the cross-product method of to compare angles. (Removing all but the farthest point with the same polar angle can be done in a total of  $O(n)$  time.)
- Lines 3-5 take  $O(1)$  time. Because  $m \leq n - 1$ , the **for** loop of lines 6-9 is executed at most  $n - 3$  times. Since PUSH takes  $O(1)$  time, each iteration takes  $O(1)$  time exclusive of the time spent in the **while** loop of lines 7-8, and thus overall the **for** loop takes  $O(n)$  time exclusive of the nested **while** loop.
- We use aggregate analysis to show that the **while** loop takes  $O(n)$  time overall.
- continued ..

# Running time continued

- For  $i = 0, 1, \dots, m$ , each point  $p_i$  is pushed onto stack  $S$  exactly once. We observe that there is at most one POP operation for each PUSH operation.
- At least three points  $-p_0$ ,  $p_1$ , and  $p_m$ -are never popped from the stack, so that in fact at most  $(m - 2)$  POP operations are performed in total.
- Each iteration of the **while** loop performs one POP, and so there are at most  $m - 2$  iterations of the **while** loop altogether.
- Since the test in line 7 takes  $O(1)$  time, each call of POP takes  $O(1)$  time, and since  $m \leq n - 1$ , the total time taken by the **while** loop is  $O(n)$ .
- Thus, the running time of GRAHAM-SCAN is  **$O(n \lg n)$** .

# Convex Hull – Jarvis march

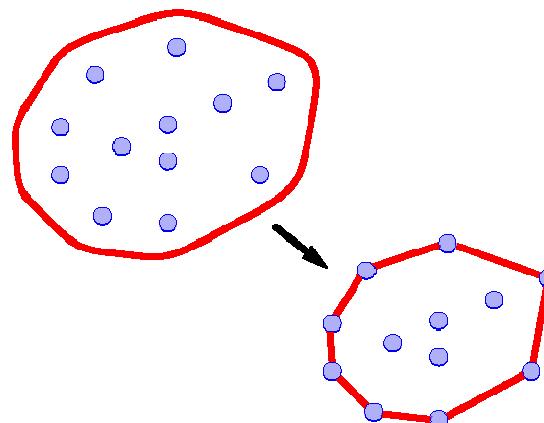
From Cormen chapter 33

# Jarvis's March (JM)

- JM computes the convex hull of a set  $Q$  of points by a technique known as **package wrapping (or gift wrapping)**.
- The algorithm runs in time  $O(n h)$ , where  $h$  is the number of vertices of  $\text{CH}(Q)$ .
- When  $h$  is  $o(\lg n)$ , JM is asymptotically faster than Graham's scan.
- JM wraps  $Q$  from left and right sides, going from the bottom to the highest point.

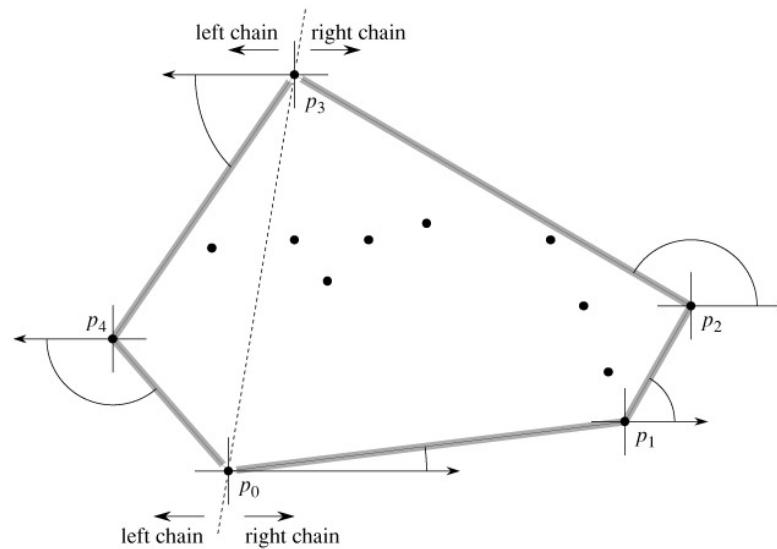
# Convex hull problem

- *Convex hull problem:* Let  $S$  be a set of  $n$  points in the plane. Compute the convex hull of these points.
- *Intuition:* rubber band stretched around the pegs
- *Formal definition:* the **convex hull** of  $S$  is the smallest convex polygon that contains all the points of  $S$



# Idea

- JM wraps Q from two sides:
  - right (going from the bottom to top)
  - left (going top to bottom)



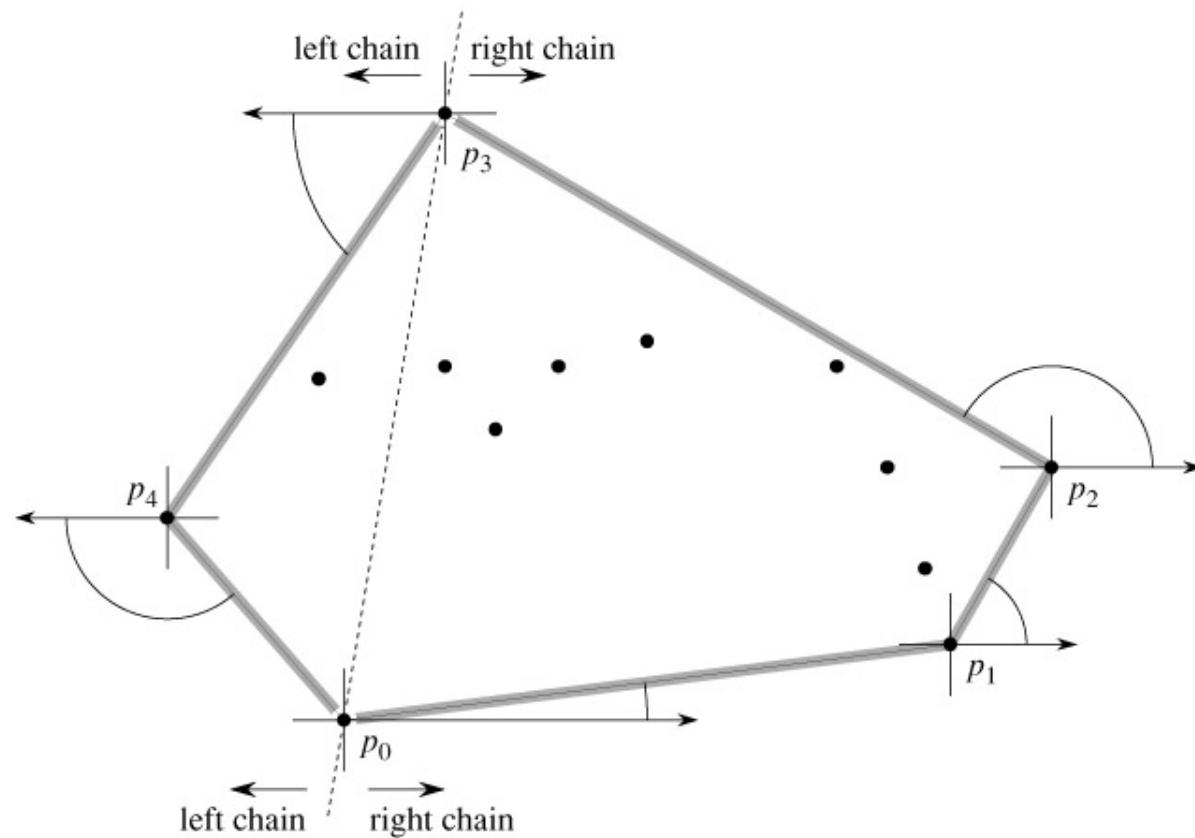
## JM right chain

- JM builds a sequence  $H = (p_0, p_1, \dots, p_{h-1})$  of the vertices of  $\text{CH}(Q)$ .
- Start with  $p_0$ , the next convex hull vertex  $p_1$  has the **smallest polar angle** with respect to  $p_0$ . (In case of ties, choose farthest point from  $p_0$ .)
- Similarly,  $p_2$  has the smallest polar angle with respect to  $p_1$ , and so on. (break ties by choosing the farthest such vertex).
- When we reach the highest vertex, say  $p_k$  we have constructed the ***right chain*** of  $\text{CH}(Q)$ .

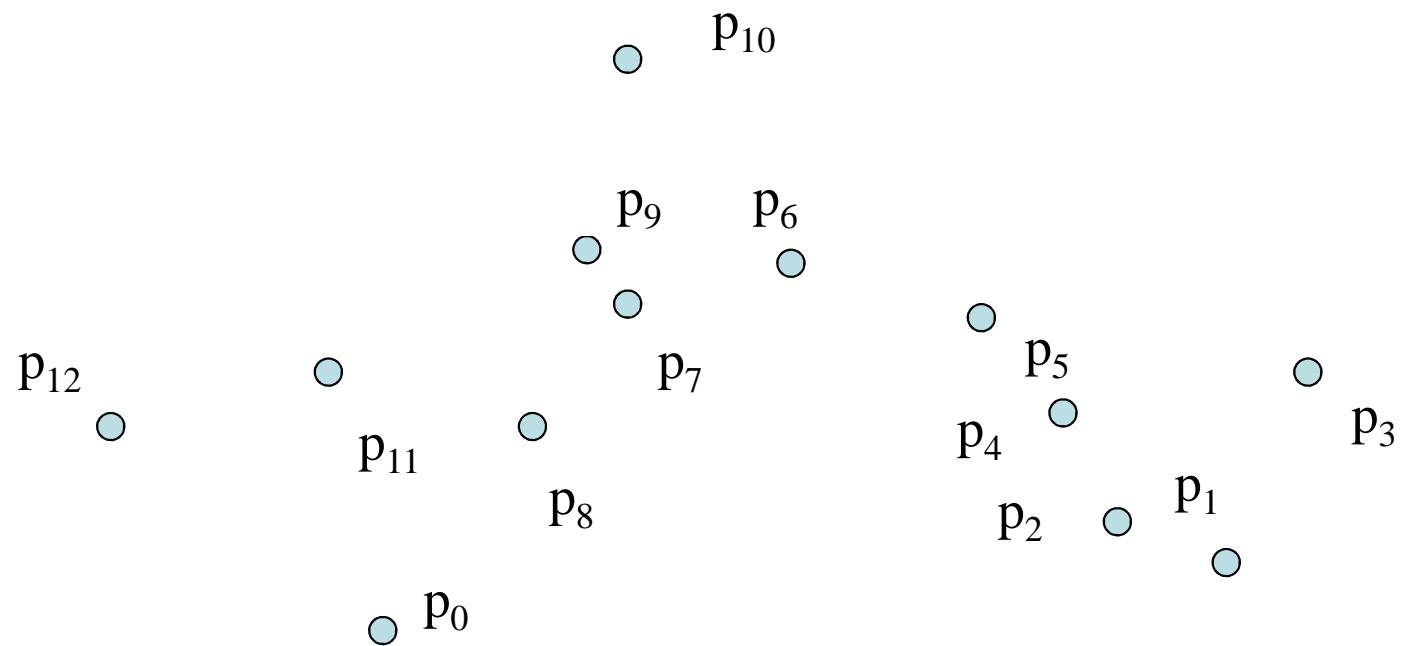
## JM left chain

- To construct the ***left chain***, we start at  $p_k$  and choose  $p_{k+1}$  as the point with the **smallest polar angle** with respect to  $p_k$ , but *from the negative x-axis*.
- We continue on, forming the left chain by taking polar angles from the negative x-axis, until we come back to our original vertex  $p_0$ .

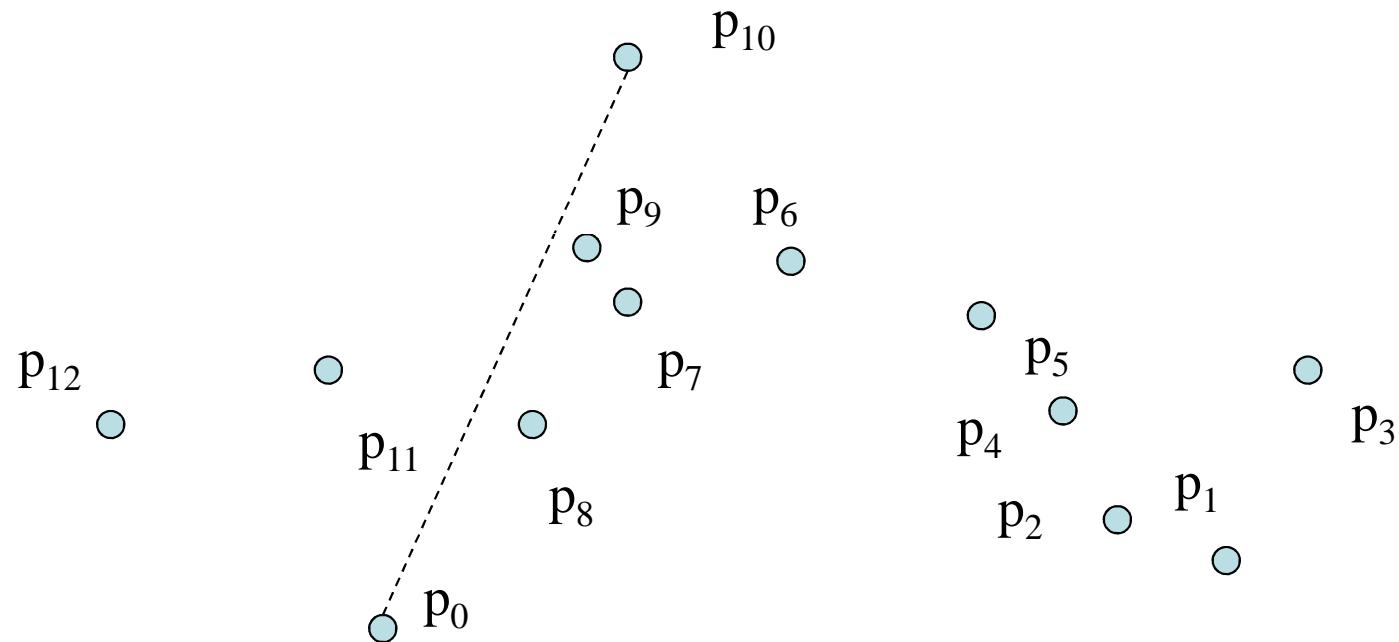
# JM Illustration



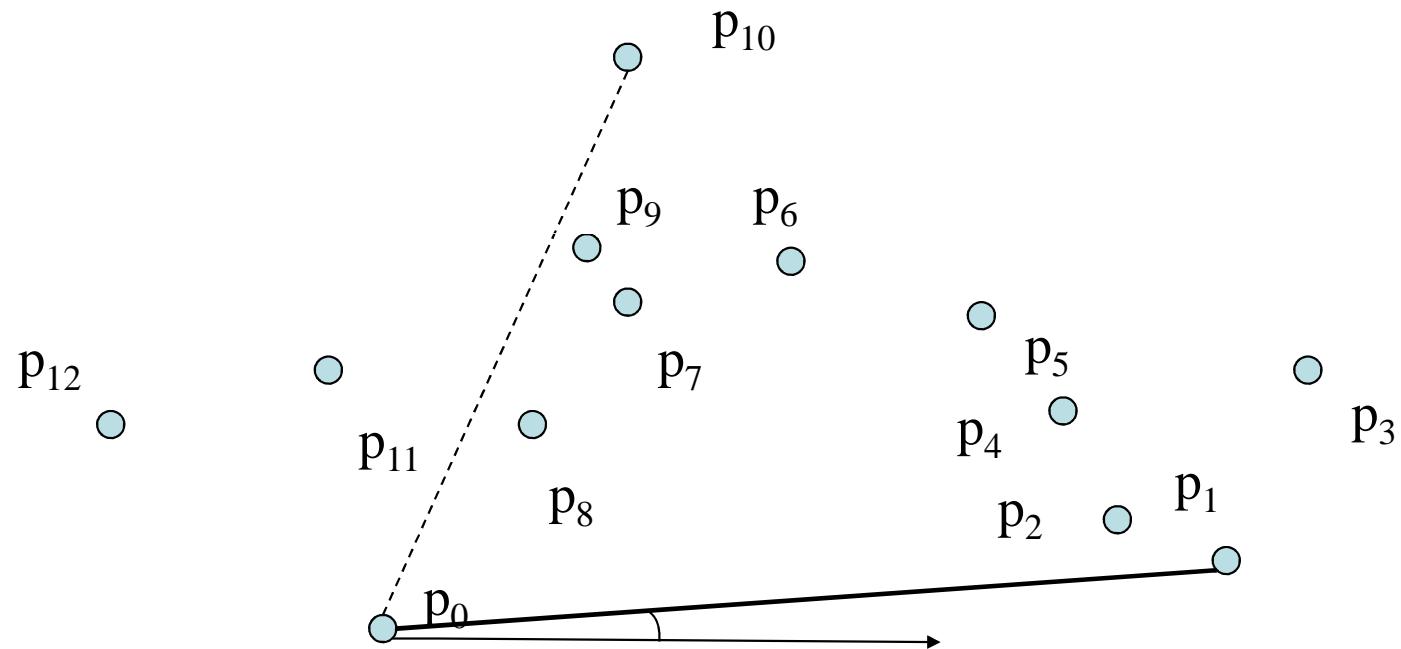
# Jarvis March - Example



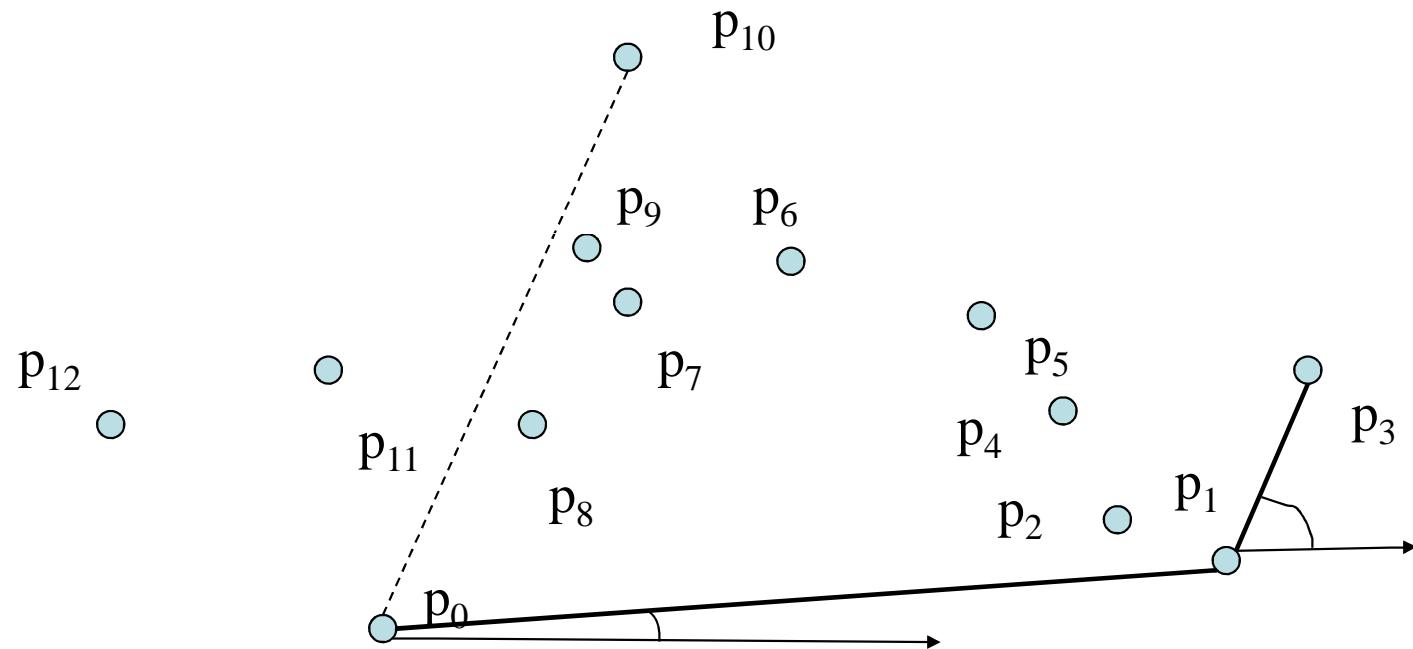
# Jarvis March - Example



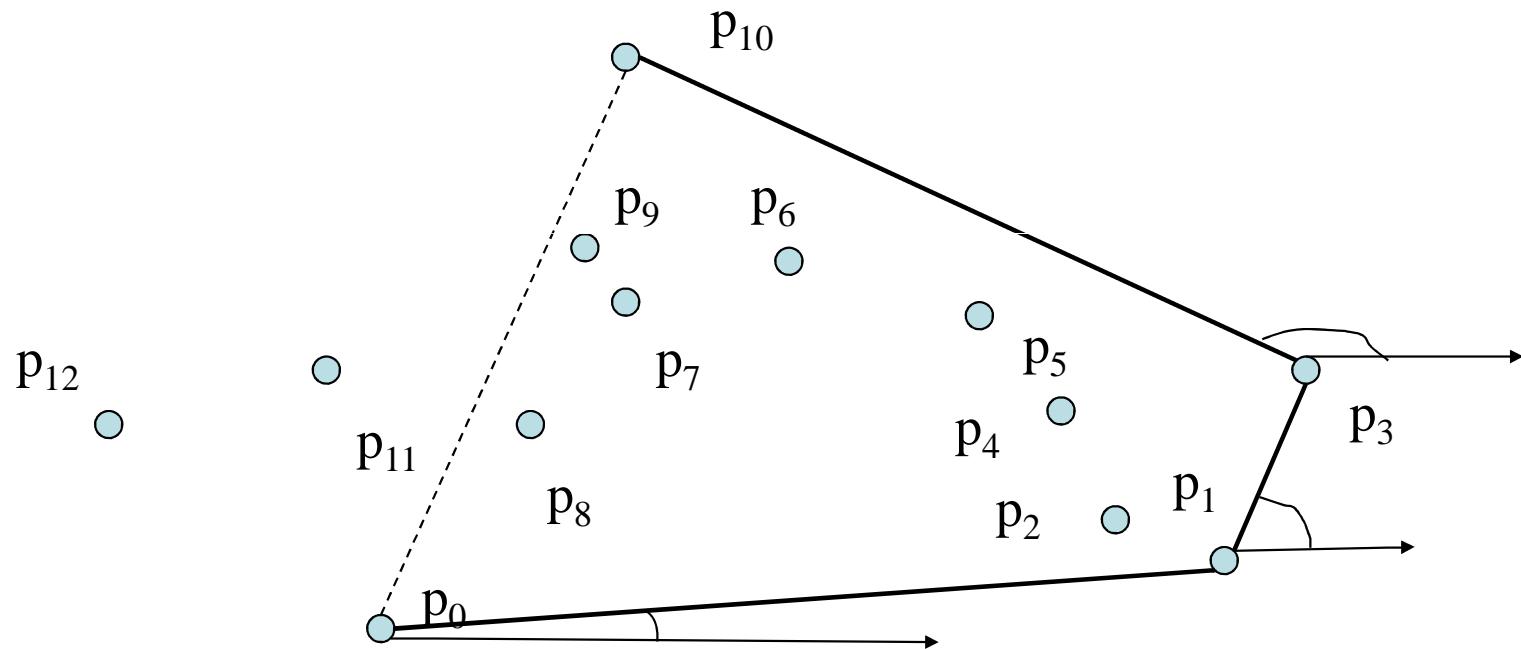
# Jarvis March - Example



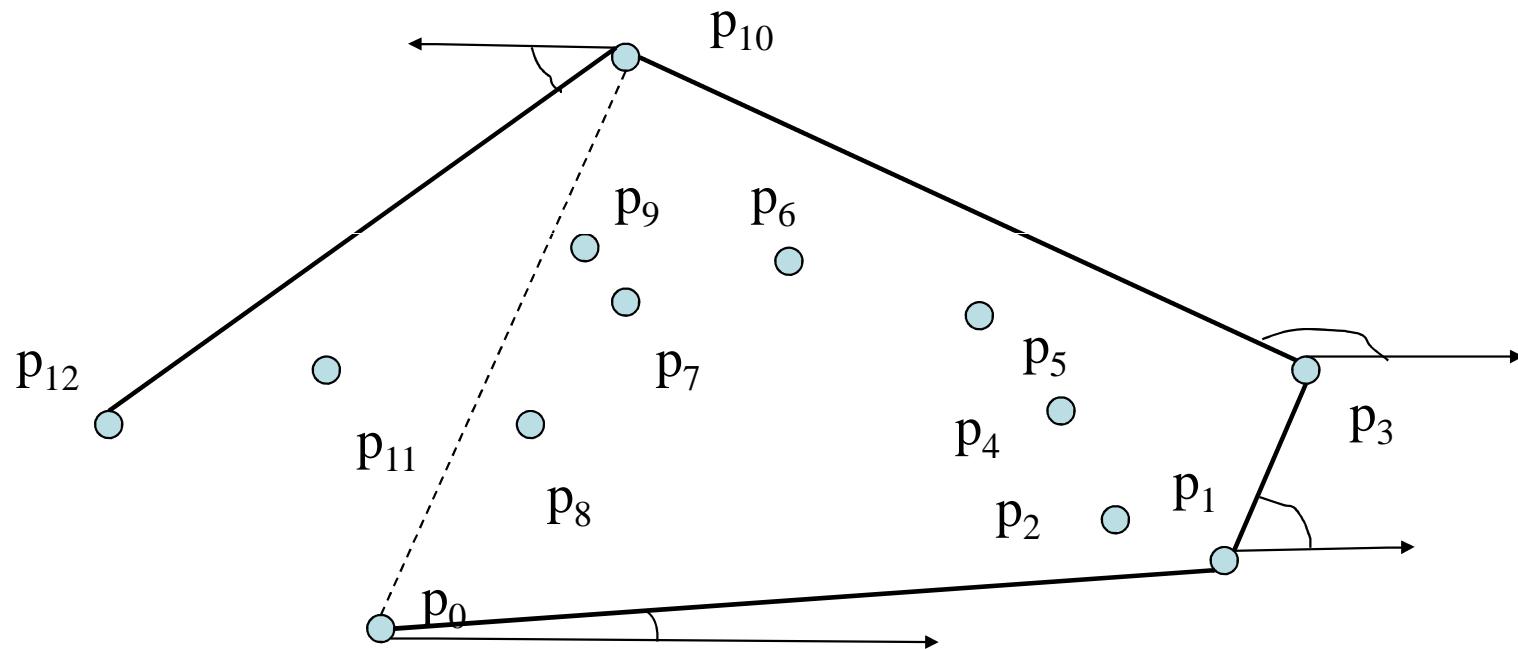
# Jarvis March - Example



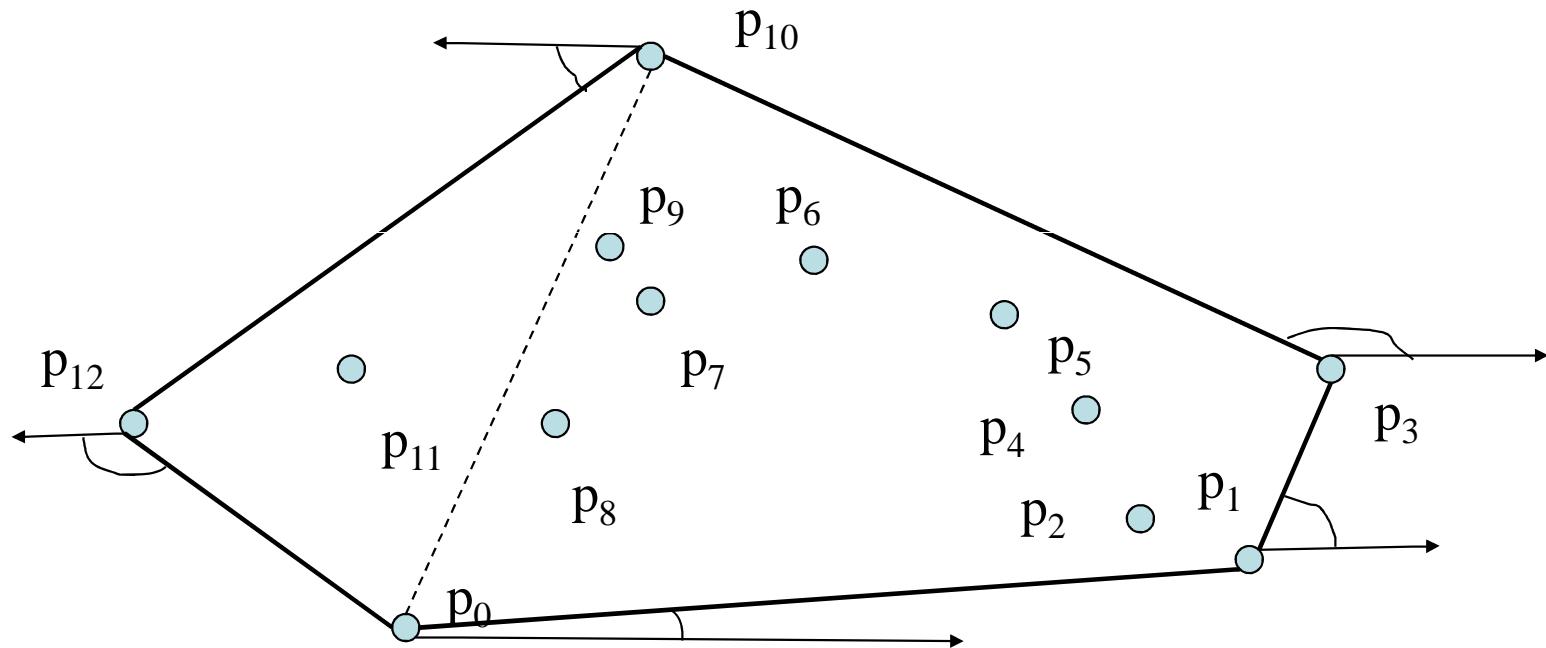
# Jarvis March - Example



# Jarvis March - Example



# Jarvis March - Example



# Running time

- If implemented properly, JM has a running time of  $O(nh)$ .
- where  $h$  is size of  $\text{CH}(Q)$ .
- For each of the  $h$  vertices of  $\text{CH}(Q)$ , we find the vertex with the minimum polar angle.
- Each comparison between polar angles takes  $O(1)$  time
- We can compute the minimum of  $n$  values in  $O(n)$  time if each comparison takes  $O(1)$  time.
- Thus, Jarvis's march takes  $O(nh)$  time.

# Convex hull - few other methods

## Divide & Conquer

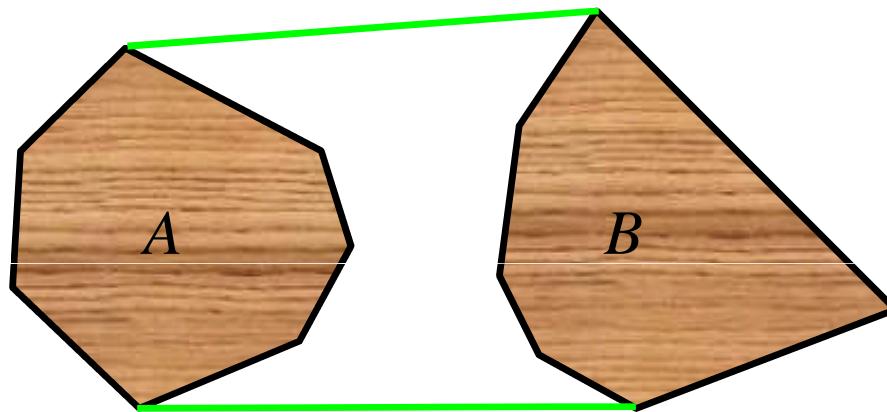
- Sort the points from left to right
- Let  $A$  be the leftmost  $\lceil n/2 \rceil$  points
- Let  $B$  be the rightmost  $\lfloor n/2 \rfloor$  points
- Compute convex hulls  $H(A)$  and  $H(B)$
- Compute  $H(A \cup B)$  by merging  $H(A)$  and  $H(B)$

Merging is tricky, but can be done in linear time

[Adapted from K.Lim Low]

# Convex hull - few other methods

upper tangent



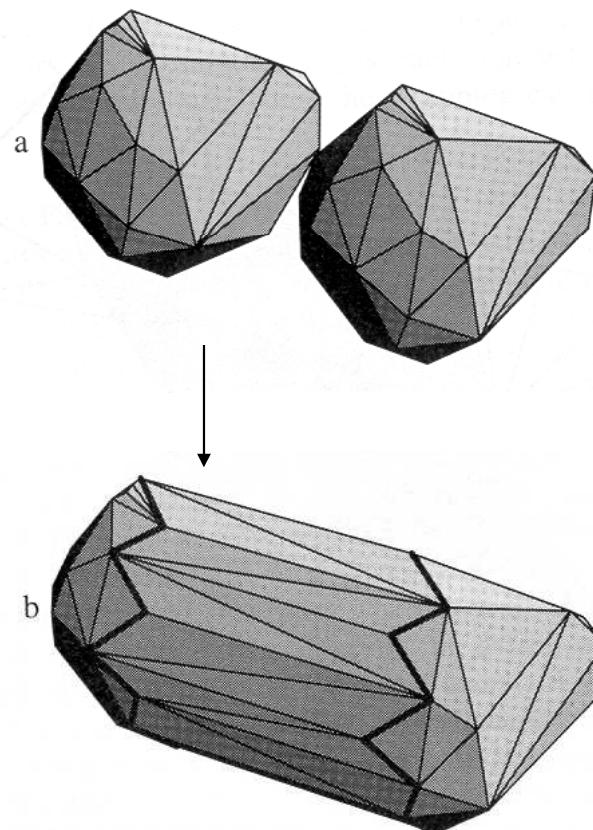
Need to find the upper and lower tangents  
They can be found in linear time

[Adapted from K.Lim Low]

# Convex hull – Divide and conquer in 3D

Merging still can be done in linear time!

We have  $O(n \log n)$  in 3D



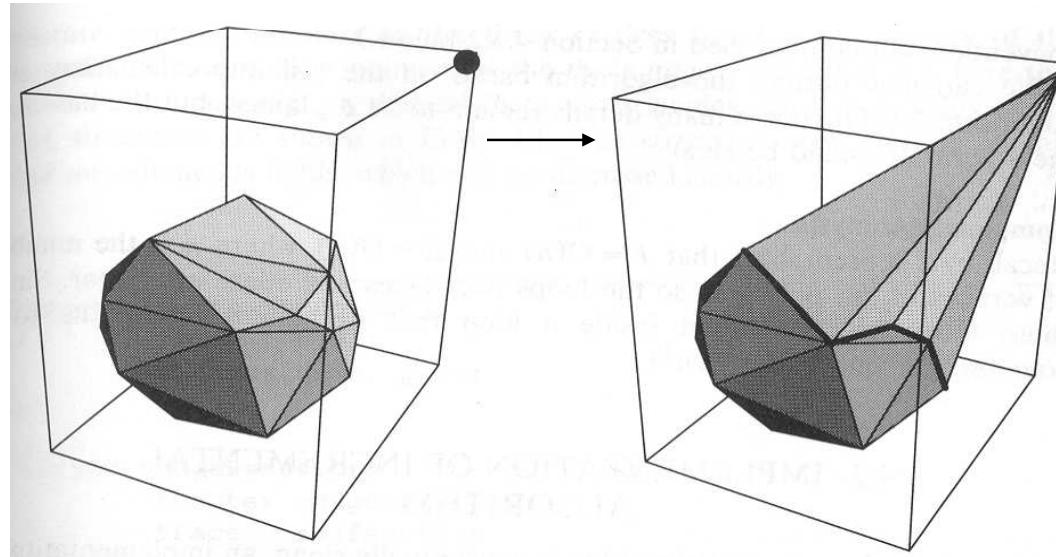
[Adapted from K.Lim Low]

# Convex hull - few other methods

Idea: incrementally add a point to a convex polyhedra  $P$

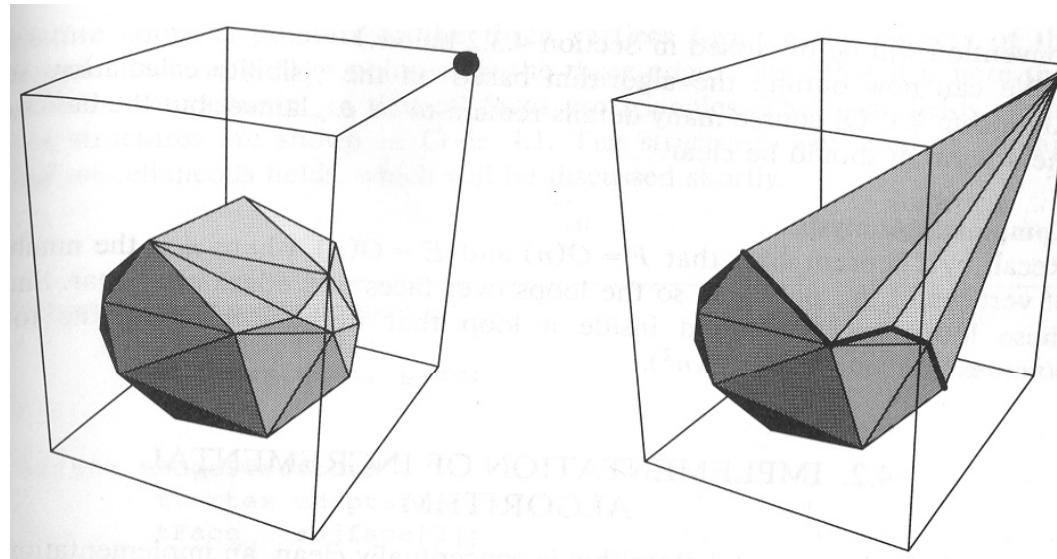
Two cases:

1. The new point is inside  $P \rightarrow$  do nothing
2. The new point is outside of  $P \rightarrow$  fix  $P$ !



[Adapted from F.Joskowitz]

# Convex hull



Naive implementation:  $O(n^2)$   
 $O(n \log n)$  randomized algorithm  
Can be made deterministic with the same running time.

[Adapted from F.Joskowitz]

# Complexity in 3 and more dimensions?

How many “faces” complex hull might have?

2D	up to	n
3D	up to	?
3D	up to	n
kD	up to	?

- Unfortunately, convex hull in  $d$  dimensions can have  $\Omega(n^{\lfloor d/2 \rfloor})$  facets (proved by Klee, 1980)
- No  $O(n \lg n)$  algorithm possible for  $d > 3$

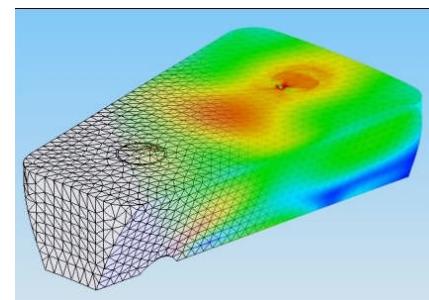
# Complexity in 3 and more dimensions?

- Gift wrapping method requires  $O(n^{\lfloor d/2 \rfloor + 1})$  time.
- There are algorithms that works in  $O( n \log n + n^{\lfloor d/2 \rfloor} )$  time.

# Polygon Triangulation

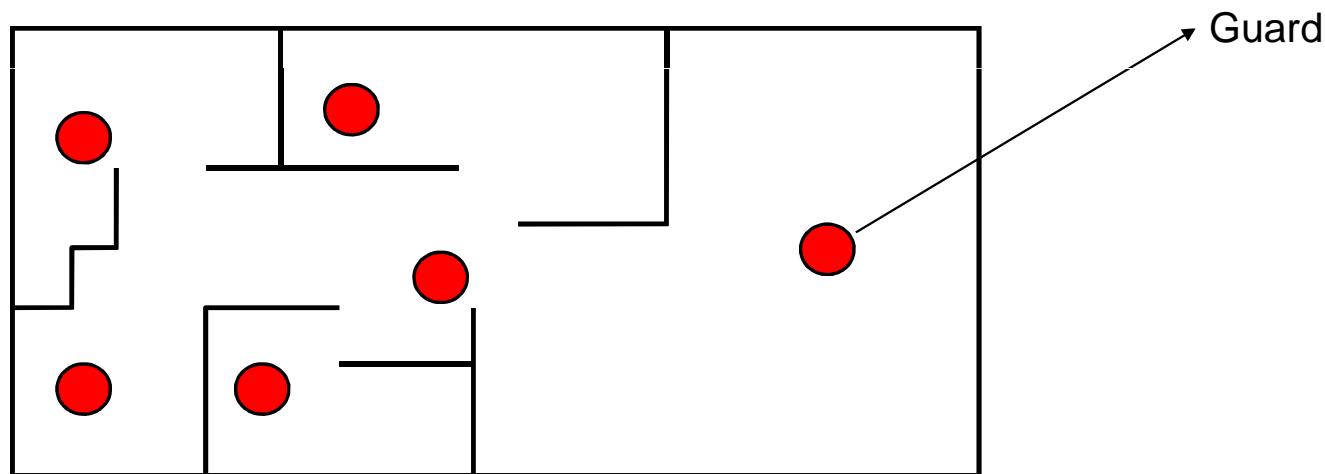
# Applications of Polygon Triangulation

- Guarding Art galleries
- Partitioning of shapes/polygons into triangular pieces for FEM (finite element modeling in CAD/CAM)



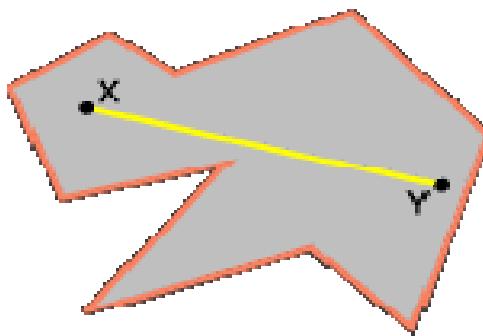
# Guarding art gallery

- Minimum number of guards needed to keep watch on every wall?



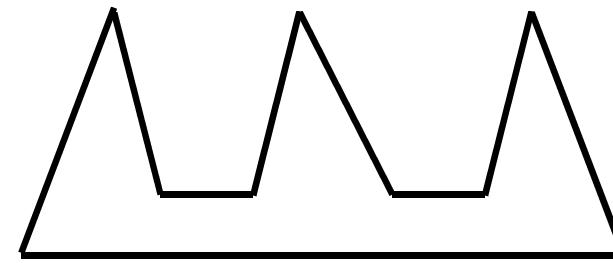
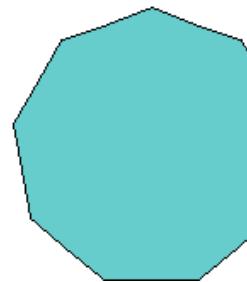
# Guarding art galleries

- The gallery is represented by a simple polygon
- A guard is represented by a point within the polygon
- Guards have a viewport of  $360^\circ$
- A polygon is completely guarded, if every point within the polygon is guarded by at least one of the watchmen



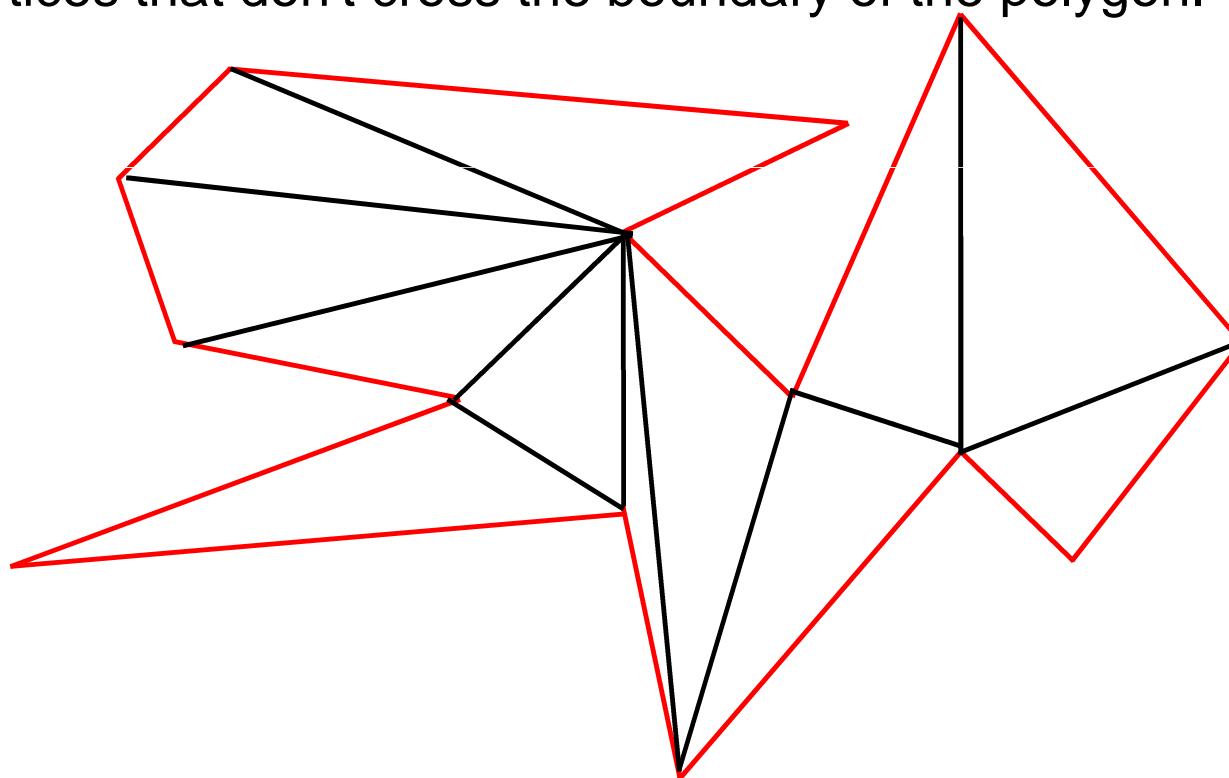
# Guarding art galleries

- Even if two polygons have the same number of vertices, one may be easier to guard than the other.
- Determine minimum number of guards for an arbitrary polygon with  $n$  vertices.

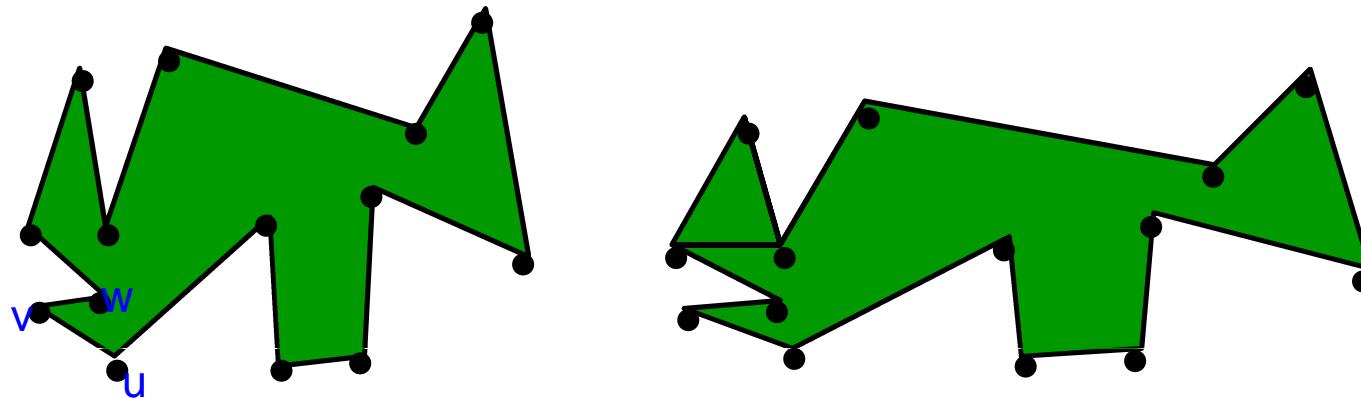


# Polygon Triangulation

- Given a **polygon**, we want to decompose it into triangles by adding *diagonals*: new line segments between the vertices that don't cross the boundary of the polygon.



# Triangulation of simple polygons



**Simple polygon:** polygon does not cross itself.  
Does every simple polygon have a triangulation?  
How many triangles in a triangulation?  
Minimize number of triangles.

# Theorem

**Theorem:** Every simple polygon admits a triangulation, and any triangulation of a simple polygon with  $n$  vertices consists of exactly  $n-2$  triangles.

**Proof.** by induction.

- The **base case  $n = 3$**  is trivial: there is only one triangulation of a triangle, and it obviously has only one triangle.
- Let  $P$  be a polygon with  $n$  edges. Draw a **diagonal** between two vertices. This splits  $P$  into two smaller polygons.
- One of these polygons has  $k$  edges of  $P$  plus the diagonal, by the **induction hypothesis**, this polygon can be broken into  $k - 1$  triangles.
- The other polygon has  $n - k + 1$  edges, and so by the **induction hypothesis**, it can be broken into  $n - k - 1$  triangles.
- Putting the two pieces back together, we have a total of  $(k - 1) + (n - k - 1) = n - 2$  triangles.

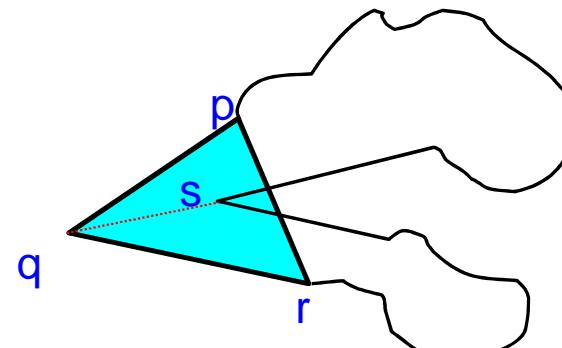
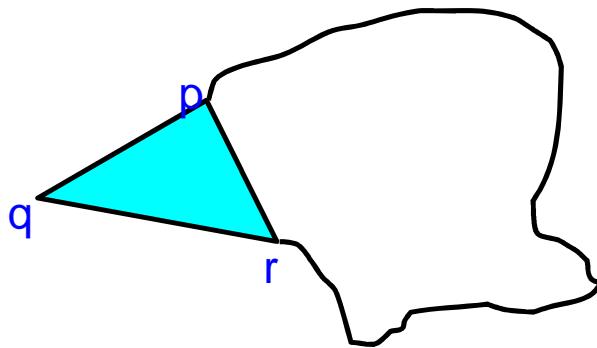
# Existence of diagonal

- How do we know that every polygon has a diagonal? [Meisters in 1975]
- **Lemma.** Every polygon with more than three vertices has a diagonal.
- **Proof.** Let  $P$  be a polygon with more than three vertices. Let  $q$  be the leftmost vertex. Let  $p$  and  $r$  be two neighboring vertices of  $q$ .

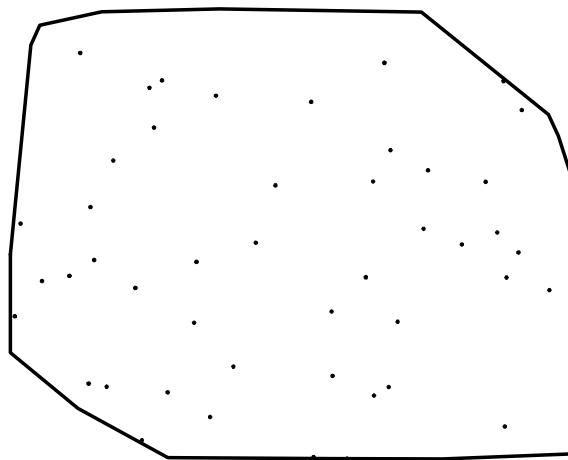
## Proof of Existence of diagonal continued

Case 1: pr completely in P. Then segment pr is a diagonal

Case 2: pr not completely in P. Let s be the vertex furthest away from the segment pr. Then the line qs is a diagonal.



# Partitioning problem



*Given:*  $n$  points are scattered inside a convex polygon  $P$  (in 2D) with  $m$  vertices.

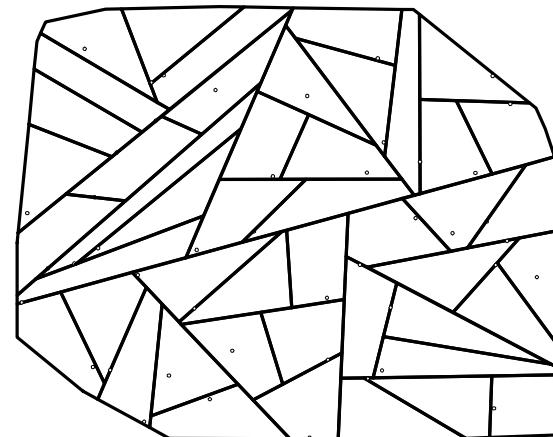
Does there exist a partition of  $P$  into  $n$  sub-regions satisfying the following:

- Each sub-region is a convex polygon
- Each sub-region contains at least one point
- All sub-regions have equal area

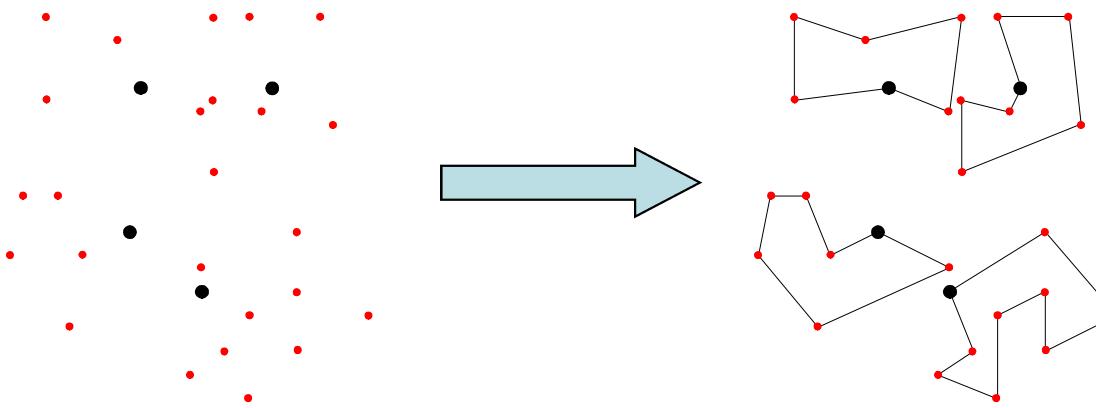
# Partitioning result

*Given:*  $n$  points are scattered inside a convex polygon  $P$  (in 2D) with  $m$  vertices. Does there exist a partition of  $P$  into  $n$  sub-regions satisfying the following:

An equitable partition always exists, we can find it in running time  $O(N n \log N)$ , where  $N = m + n$ .



# Partitioning Applications



*Example: Broadcast Network* problems, to connect *clients* to *servers*, in a fixed underlying network topology.

*Example: Multi-Depot Vehicle Routing Problem (MDVRP).*

*Definition:* A set of *vehicles* located at *depots* in the plane must visit a set of *customers* such that the maximum TSP cost is minimized (min-max MDVRP).