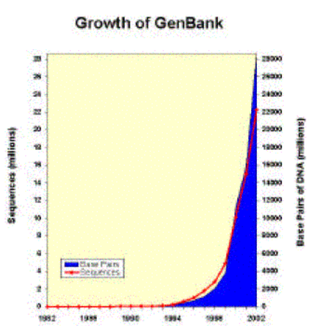
## String Matching and Suffix Tree

**Gusfield Ch1-7**

**EECS 458**
**CWRU**
**Fall 2004**

## GenBank




Size doubles every 14 months.

As of the end of 2002, GenBank contained approximately 28,507,990,166 bases in 22,318,883 sequence records

## BLAST (Altschul et al'90)

- Idea: true match alignments are very likely to contain a short segment that identical (or very high score).
- Consider every substring (seed) of length *w*, where *w*=12 for DNA and 4 for protein.
- Find the exact occurrence of each substring (how?)
- Extend the hit in both directions and stop at the maximum score.

## Problems

- Pattern matching: find the exact occurrences of a given pattern in a given structure ( string matching)
- Pattern recognition: recognizing approximate occurences of a given patttern in a given structure (image recognition)
- Pattern discovery: identifying significant patterns in a given structure, when the patterns are unknown (promoter discovery)

## Definitions

- String *S*[1..m]
- Substring *S*[i..j]
- Prefix *S*[1..i]
- Suffix *S*[i..m]
- Proper substring, prefix, suffix
- Exact matching problem: given a string *P* called pattern, and a long string *T* called text, find all the occurrences of *P* in *T*.

## Naïve method

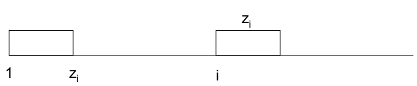
- Align the left end of P with the left end of T
- Compare letters of P and T from left to right, until
  - Either a mismatch is found (not an occurrence
  - Or P is exhausted (an occurrence)
- Shift P one position to the right
- Restart the comparison from the left end of P
- Repeat the process till the right end of P shifts past the right end of T
- Time complexity: worst case $\theta$(mn), where m=|P| and n=|T|
- Not good enough!

## Speedup

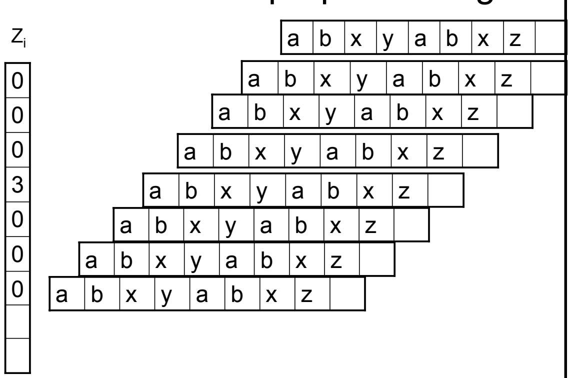
- Ideas:
  - When mismatch occurs, shift P more than one letter, but never shift so far as to miss an occurrence
  - After shifting, ship over parts of P to reduce comparisons
  - Preprocessing of P or T

## Fundamental preprocessing

- Can be on pattern *P* or text *T*.
- Given a string *S* (|*S*|=*m*) and a position *i* >1, define $Z_i$: the length of longest common prefix of *S* and *S*[i..m]
- Example, *S=abxyabxz*




## Fundamental preprocessing

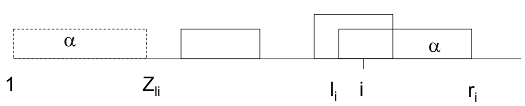


# Fundamental preprocessing

- Intention:
  - Concatenate P and T, inserted by an extra letter \$: S=P\$T
  - Every i, $Z_i \leq |P|$
  - Every $i>|P|+1$ and $Z_i=|P|$ records the occurrences of P in T
- Question: running time to compute all the Zs? The naïve method according to the definition runs in $\theta((m+n)^2)$ time!

# Fundamental preprocessing

- Goal: linear time to compute all the Zs
- Z-box: for $Z_i>0$, it is the box starting at i with length $Z_i$ (ending at $i+Z_i-1$),
- $r_i$: the rightmost end of a $Z_j$-box ($j+Z_j-1$) for all $1<j \leq i$ such that $Z_j>1$.
- $l_i$: the left node of $Z_j$-box ending at $r_i$




# Fundamental preprocessing

- Computing $Z_k$:
  - Given $Z_i$ for all $1<i<k$
  - Let $r=r_{k-1}$ and $l=l_{k-1}$
  1. $k>r$: compute $Z_k$ explicitly (updating accordingly r and l if $Z_k>0$)
  2. $k\leq r$: k is in the Z-box starting at l (substring S[l..r]), therefor S[k]=S[k-l+1], S[k+1]=S[k-l+2], …, S[r]=S[$Z_l$]. In other words, $Z_k \geq \min\{Z_{k-l+1}, r-k+1\}$

## Fundamental preprocessing

- A) $Z_{k-l+1} < (r-k+1)$: $Z_k = Z_{k-l+1}$, and r, l remain unchanged
- B) $Z_{k-l+1} \geq (r-k+1)$: $Z_k \geq (r-k+1)$ and start comparison between S[r+1] and S[r-k+2] until a mismatch is found (updating r and l accordingly if $Z_k \geq r-k+1$)



## Fundamental preprocessing

- Conclusions:
  1. $Z_k$ is correctly computed
  2. There are a constant number of operations besides comparisons for each k
     - |S| iterations
     - Whenever a mismatch occurs, the iteration terminates
     - Whenever a match occurs, r is increased
  3. In total at most |S| mismatches and at most |S| matches
  4. Running time $\theta(|S|)$ and space $\theta(|S|)$

## Fundamental preprocessing

- Th: there is a $\theta(n+m)$-time and space algorithm which finds all the occurrences of P in T, where m=|P| and n=|T|.
- Notes:
  - Alphabet-independent
  - Space requirement can be reduced to $\theta(m)$
  - Not well suited for multiple patterns searching
  - Strictly linear, every letter in T has to be compared at least once

## Projects

- Topics
- Meeting: 3 times, as a group
- Presentations: 25 minutes/student (~20m talk + 5m questions)
- Term paper: single space, 11pt, 1in margin. 5-6p, 9-10p 10-12p, exclude references

## The Boyer-Moore algorithm: an example

- P=abxabxab, T=daaabxababxabxab

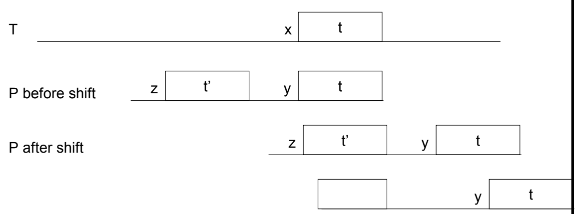
```
d a a a b x a b a b x a b x a b
a b x a b x a b
    a b x a b x a b

d a a a b x a b a b x a b x a b
a b x a b x a b
      a b x a b x a b
```

## The Boyer-Moore algorithm

- Rule 1: right-to-left comparison
- Rule 2: Bad character rule
  - For each $x \in \Sigma$, R(x) denotes the right-most occurrence of x in P (0 if doesn't appear)
  - When a mismatch occurs, T[k] against P[i], shift P right by max{1, i-R(T[k])} places. This takes T[k] against P[R(T[k])]
  - $|\Sigma|$ space to store R-values
- Rule 3: good suffix rule

# The Boyer-Moore algorithm

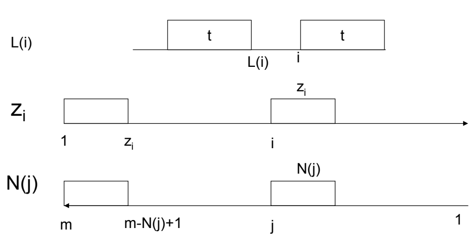



# The Boyer-Moore algorithm

- Rule 3: good suffix rule
  - When a mismatch occurs, T[k] against P[i]
  - Find the rightmost occurrence of P[(i+1)..m] in P such that the letter to the left differs P[i]
  - Shift P right such that this occurrence of P[(i+1)..m] is against T[(k+1)..(m+k-i)]
  - If there is no occurrence of P[(i+1)..m], find the longest prefix of P matches a suffix of P[(i+1)..m], shift P right such that this prefix is against the corresponding suffix

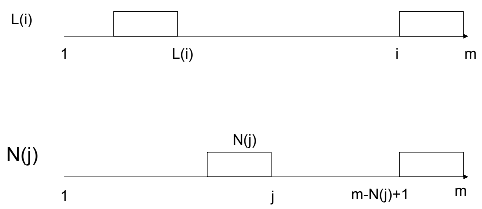
# Preprocessing for the good suffix rule

- Let L(i) denote the largest position less than m such that string P[i..m] matches a suffix of P[1..L(i)]
- Let N(j) denote the longest suffix of substring P[1..j] that is also a suffix of P
- Recall $Z_i$ the length of longest substring of P starts I and matches a prefix of S.



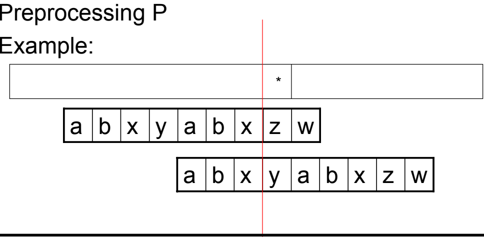
## Preprocessing for the good suffix rule

- Thm: L(i) is the largest index j less than m such that N(j)≥|P[i..m]|=m-i+1.




## The Knuth-Morris-Pratt Algorithm

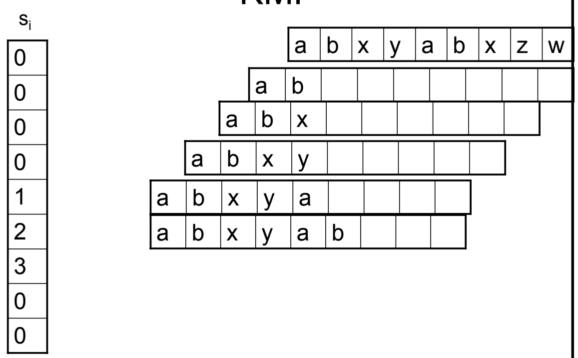
- History:
  - Best known
  - Not the method of choice, inferior in practice
  - Can be generalized for multiple string matching
- Preprocessing P
- Example:




## KMP

- Idea:
  - left to right comparison,
  - Shift P more places without missing occurrence
- A prefix of P matches a proper suffix of P[1..i] and the next letters do not match!
- Define $s_i$ of P, 2<=i<=m the length of longest proper suffix of P[1..i] that matches a prefix of P, $s_1$=0.

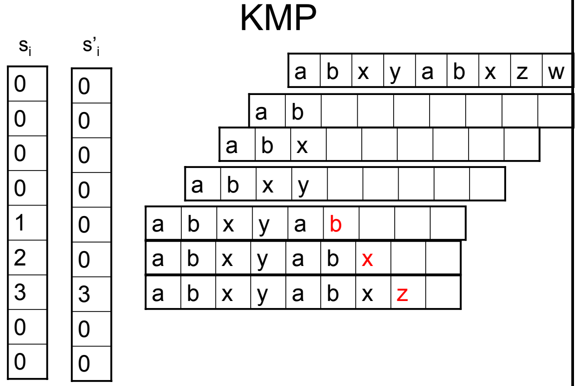
## KMP




## KMP

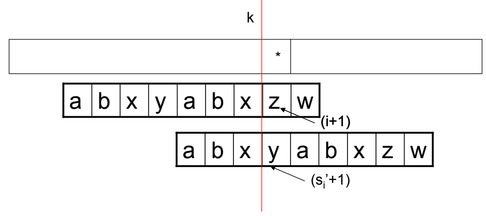
- Define $s_i$', 2<=i<=m, the length of the longest proper suffix of P[1..i] that matches a prefix of P with the additional condition that character P[i+1] differs from P[$s_i$'+1].
- Obviously $s_i' \le s_i$ for any i

## KMP



# KMP

- Shifting rule: shift P to the right $(i-s_i')$ spaces ( i=n if an occurrence)



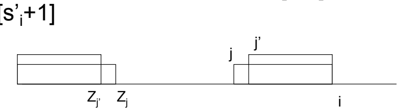

- Since $P[1..s_i']$ matches $P[i-s_i'+1, i]$ and $P[i-s_i'+1,i]$ matches $T[k-s_i'+1,k]$, thus we skip $s_i'$ comparisons.

# KMP

```
KMP(){
   m=|P|,n=|T|, s', q=0;
   for (i=0; i<n;i++){
     while q>0 and P[q+1] <> T[i]
       q=s_q'
     if(P[q+1]==T[i])
       q++;
     if(q=m) {
       find a pattern at position i-m, q=s_q'
  }
```

# KMP

- Correctness
- $Z_j$: the length of the longest common prefix of P and P[j..n]
- $s'_i$ the length of the longest proper suffix of P[1..i] that matches a prefix of P, with the additional condition that the character P[i+1] differs from $P[s'_i+1]$




- Therefore, $s'=\max\{ Z_j \mid Z_j = i-j+1\}$

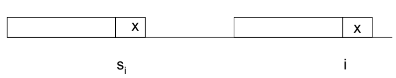
# KMP

- Running time:
  - In total s phases (of comparison/shift), $s \le n$
  - Every 2 consecutive phases overlap one letter (the mismatched one) from T
  - Therefore, in total $n+s \le 2n$
- Questions:
  - Any letter from T is skipped for comparison?
  - The times of comparison for a letter is at most a constant time? (real time algorithm)

# Original preprocessing of KMP: calculating s' directly

- $s_i$: the length of longest proper suffix of P[1..i] that matches a prefix of P
- $s_i'$, the length of the longest proper suffix of P[1..i] that matches a prefix of P and P[i+1] <> P[$s_i'$+1].
- So $s_i' = s_i$ if P[i+1] <> P[$s_i$+1]

  $s_i' = s'_{si}$ otherwise

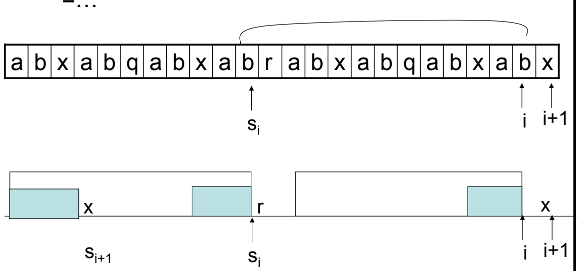



# Calculate s

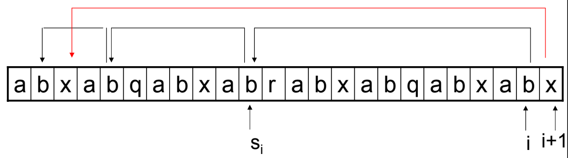
$s_{i+1} = s_i+1$ if P[$s_i$+1]=P[i+1]

$= s_{\{si\}}+1$ if P[$s_{si}$+1]=P[i+1]

$=\ldots$



## Calculate s



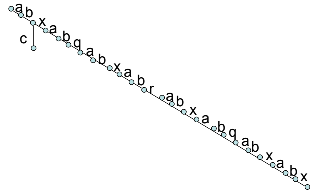

- The preprocessing can be done in linear time.

## Multiple pattern matching problem

- Given a set of pattern $\mathrm{P} = \{P_1, P_2, \ldots, P_k\}$ and a text (databases) T, find all the occurrences of all the patterns…
- Keyword tree: given a set of pattern $\mathrm{P} = \{P_1, P_2, \ldots, P_k\}$, the keyword tree K is a tree:
  - Rooted, directed
  - Each edge is labeled with one letter
  - Edges coming out of a node have distinct labels
  - Every pattern is spelled out (map to one node)
  - Every leaf maps to some pattern

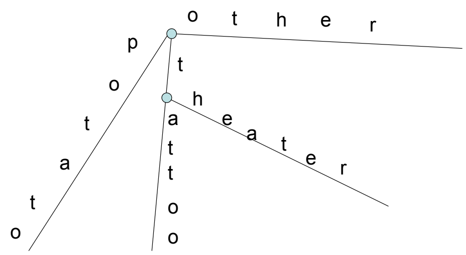
## An example

- abxabqabxabrabxabqabxabx



- abc

- P = {potato, tattoo, theater, other}

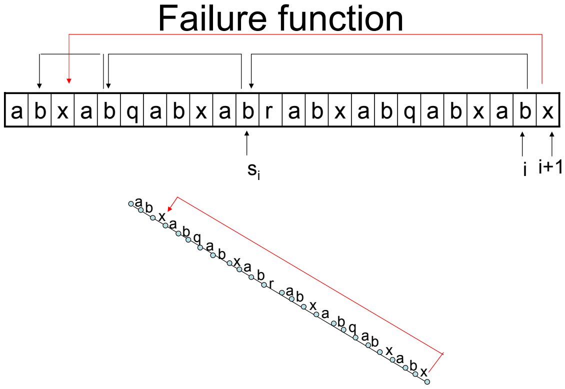


- Linear time construction
- Dictionary problem

## Multiple pattern matching problem

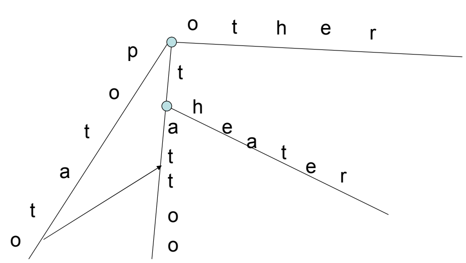
- Given a set of pattern P = {$P_1$, $P_2$,…, $P_k$} and a text (databases) T, find all the occurrences of all the patterns.
- The sum of the lengths of patterns: m, length of text: n. Previous algorithms imply a search algorithm of $\theta$(m+kn) time.
- There are algorithms running in $\theta$(m+n+l) time, where l is the total number of occurrences of all the patters
- Using keyword tree of P
- The same idea as in KMP

## Failure function



## Multiple pattern matching problem

- L(v) denote the string from the root to v and lp(v) denote the length of the longest proper suffix of string L(v) that is a prefix of some pattern in P.
- lp(v) for all the node v in K can be computed in linear time $\theta$(m)
- Use the same failure links v->n(v)




## Aho-Corasick algorithm

- Create the keyword tree
- Compute the lp(v) and n(v) for each node v in the keyword tree
- Search the text against the tree, when a mismatch, T shifts lp(v) spaces and starts to compare from n(v)
- Total running time $\theta$(m+n+l)

## Exact string matching applications

- Sequence-tagged-sites
- Exact string matching with wild cards
- Two-dimensional exact matching
- Regular expression pattern matching

## Suffix tree

- Introduction
- Construction
- Applications

- Reading: Gusfield Ch5-7

## Suffix tree

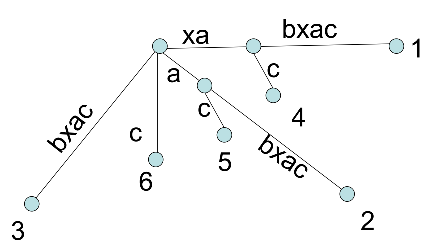
- Given a finite alphabet set $\Sigma$, a string S of length m, e.g., *S*=abxabc
- Suffix tree of S:
  - Rooted, directed
  - Edges labeled by substrings of S
  - Edges coming out of a node start with distinct letters
  - Exactly m leaves
  - Leaf i spells out suffix S[i..m]

## Keyword tree

- Keyword tree: given a set of pattern P = {$P_1$, $P_2$,…, $P_k$}, the keyword tree K is a tree:
  - Rooted, directed
  - Each edge is labeled with one letter
  - Edges coming out of a node have distinct labels
  - Every pattern is spelled out (map to one node)
  - Every leaf maps to some pattern

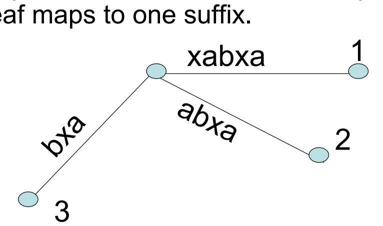
## Suffix tree

Example: xabxac




## Suffix tree

- What if a suffix is a prefix of some other suffix? E.g. xabxa
- Solution: append a new letter $
- Implicit suffix tree: if we don't require each leaf maps to one suffix.



## First construction algorithm

- For a string S:
  - Assume no suffix is a prefix of some other suffix.
  - Make every suffix as a pattern $P_i$=S[i..m]
  - Apply the linear time keyword tree construction algortihm
  - Concatenate "paths" into "edges"
- Running time:
  - Linear in the sum of the lengths of patterns $Sum_i|P_i|$ = m(m+1)/2
  - $\theta(m^2)$
  - Goal: design a linear time algorithm $\theta(m)$

## Why suffix tree

- 1st application: exact matching
- Suppose in $\theta(n)$ time we can build the suffix tree for the text T
- Given any pattern P, at any time
  - Match letters of P along the suffix tree, until
  - Either no more matches are possible, P doesn't occur anywhere in T
  - Or P is exhausted: the labels of the leaves in side the subtree under the last matching edge are the starting positions of the occurrences.

## 1st application

- Conclusion:
  - Exact string matching done in $\theta$(m+n+l) time
  - Exact multiple string matching done in $\theta$(m+n+l) time,
  - L the number of occurrences
- Other applications:
  - Multiple keyword search
  - Longest repeating substring
  - Longest common substring of two/more strings

## Ukkonen's linear time construction

- Using implicit suffix tree of S[1..i]: $T_i$.
- Construct $T_i$ incrementally:
  - From $T_i$ to $T_{i+1}$
  - Need to add S[i+1] to every suffix of S[1..i] and the empty string, there are i+1 of them…
  - Append S[i+1] to suffix S[j..i] – becoming a suffix of S[j..(i+1)], j=1, 2… i, i+1.
  - Three suffix extension rules:

## Suffix extension rules

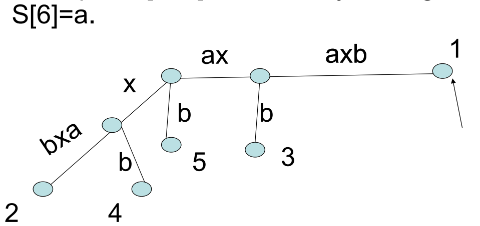
- Let $\beta$ denote the path of S[j..i]

1. $\beta$ ends at a leaf, append S[i+1] to the corresponding edge label
2. Some paths start from the end of $\beta$, but non of them starts with S[i+1], add a new leaf edge, labeled by S[i+1]
3. Some path from the end of $\beta$ starts with S[i+1], already in the tree, do nothing

- Straightforward implementation $\theta(m^3)$

## Ukkonen's algorithm, speedup

- How?
- Key thing to do: to locate the ending position of S[j..i] in $T_i$
- Example: S[1..5] = axaxb, try adding S[6]=a.

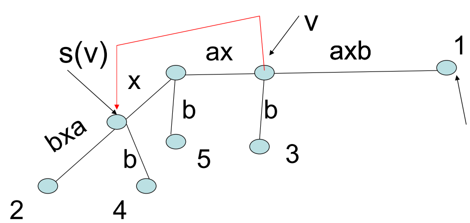


## Suffix links

- j=1: easy (use a pointer pointing to the longest path in $T_i$) append a to the edge
- Denote the leaf edge as (v, 1)
  1. If v is the root, j=2 is done straightforwardly
  2. v is not the root: there is another node, denoted as s(v), such that if root to v spells out S[1..l], then root to s(v) spells out S[2..l]
  3. When we have the information on s(v), continue search from it, not necessarily from the root again.
  - Repeat for every j

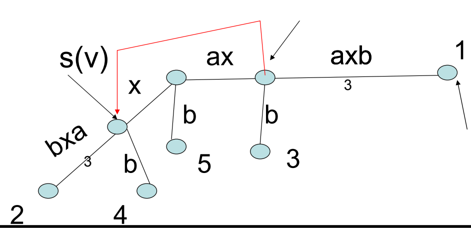
## Suffix links

- Notes:
  - need to record s(v) if a new node is created
  - It doesn't give a "faster" algorithm




## Trick1: skip/count

- Key: record the length of each edge,
- When search the path, by using the length information, for each step, we can go node by node, not character by character; only constant number of comparison at each node (assume constant alphabet set)



## Analysis of trick 1

- Every node has a depth, i.e., the number of nodes on the path from the root
- The depth of v is at most one greater than the depth of s(v),
- During the entire phase constructing $T_{i+1}$ from $T_i$ ($1 \le j \le i+1$): decrease node depth at most 2m (each j, decrease 1 to find v, and decrease 1 to find s(v); $i \le m$)
- The total number of node length that could be increased during a phase is bounded by the number of decrement (2m) and the maximum length of a path (m)
- So construction done in O(m)
- Applying suffix link and trick 1 gives an $O(m^2)$ time suffix tree building algorithm

## Two observations

- Observation1: The space for the total number of letters in the edge labels could reach $\theta(m^2)$
- An alternate way to represent labels is necessary: Use position intervals [start, end] to represent label S[start..end]
- Observation2: When S[j..i] doesn't end at a leaf and there is an extending edge whose label starts with S[i+1], we are done for j, we are done for the phase i (S[j+1,i+1], S[j+2,i+1],…, S[i,i+1],
- Trick 2: whenever this happens, $T_{i+1}$ is built

## Last trick

- Observation 3: once a leaf, always a leaf. If a leaf is created and labeled by an index j, rule 1 will always apply.
- Trick 3: use a global parameter e to denote the last position thus to skip the extensions (only need to update e once per iteration).

## Recall: Ukkonen's linear time construction

- Using implicit suffix tree of S[1..i]: $T_i$.
- Construct $T_i$ incrementally:
  - From $T_i$ to $T_{i+1}$
  - Need to add S[i+1] to every suffix of S[1..i] and the empty string, there are i+1 of them…
  - Append S[i+1] to suffix S[j..i] – becoming a suffix of S[1..(i+1)], j=1, 2… i, i+1.
  - Three suffix extension rules:

## Recall: Suffix extension rules

- Let $\beta$ denote the path of S[j..i]

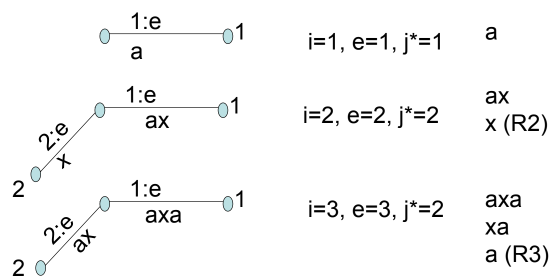
1. $\beta$ ends at a leaf, append S[i+1] to the corresponding edge label
2. Some paths start from the end of $\beta$, but non of them starts with S[i+1], add a new leaf edge, labeled by S[i+1]
3. Some path from the end of $\beta$ starts with S[i+1], already in the tree, do nothing

## Time complexity: amortized analysis

- Increment e to skip the first j* (from last phase) extensions. (Skip some j at the beginning)
- Apply trick 1 to continue until trick 2 can be applied at $j^{th}$ extension, (skip some j at last)
  - Set j*=j-1: update j* for the next phase use
- Next phase we can skip the first j* extensions…
- Every two consecutive phases overlap at most 1 index
- Linear time algorithm!
- (j* of $(i+1)^{th}$ run is the previous position that rule 3/trick 2 applied in $i^{th}$ run.

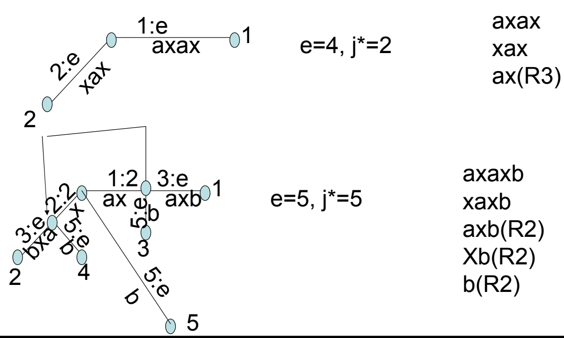
# Example

- S[1..6]=axaxba



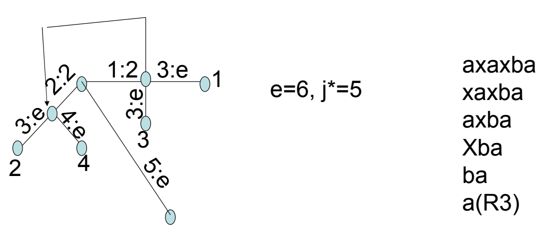

# Example

- S[1..6]=axaxba




# Example

- S[1..6]=axaxba



## The true suffix tree

- Append $ to S and execute on $T_m$
- Correctness: every suffix is spelled out by some root-to-leaf path, no suffix is a prefix of some other suffix
- Generalized suffix tree for a set of strings:
  - Concatenate strings into one, by adding some extra letters, but some synthetic suffixes
  - Build suffix tree for one string, then on top of it build for anther, then on top of it, build for anther…

## Applications

- Exact pattern search
- Longest common substring
- Tandem repeat
- Suffix array
- …