

# Text Retrieval: Theory and Practice

*Ricardo Baeza-Yates*

Center for Web Research

[www.cwr.cl](http://www.cwr.cl)

Depto. de Ciencias de la Computación

Universidad de Chile

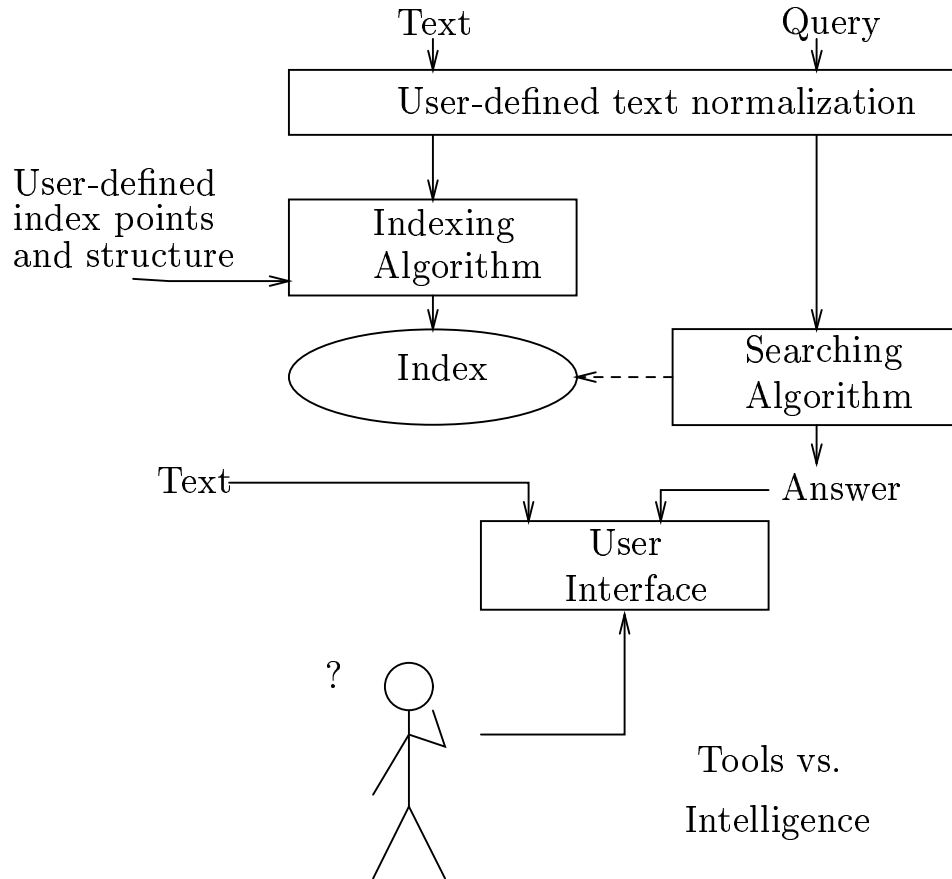
Santiago, CHILE

[rbaeza@dcc.uchile.cl](mailto:rbaeza@dcc.uchile.cl)

# Outline

1. Text Retrieval
2. Theory vs. Practice
3. Problem
4. String searching
5. From automata to algorithms
6. Indices
7. Hybrid solutions
8. Compression
9. Web Applications
10. Future and advanced topics

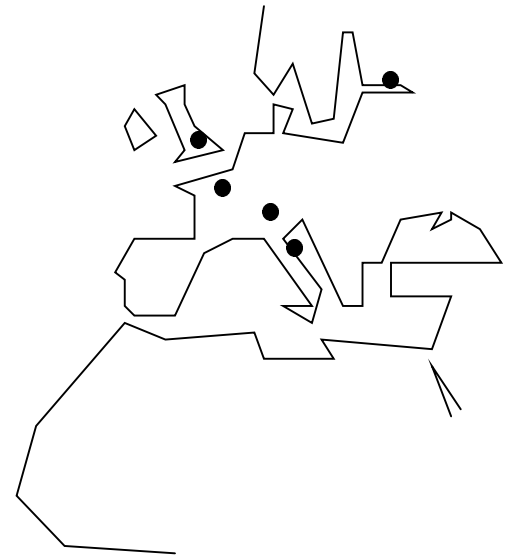
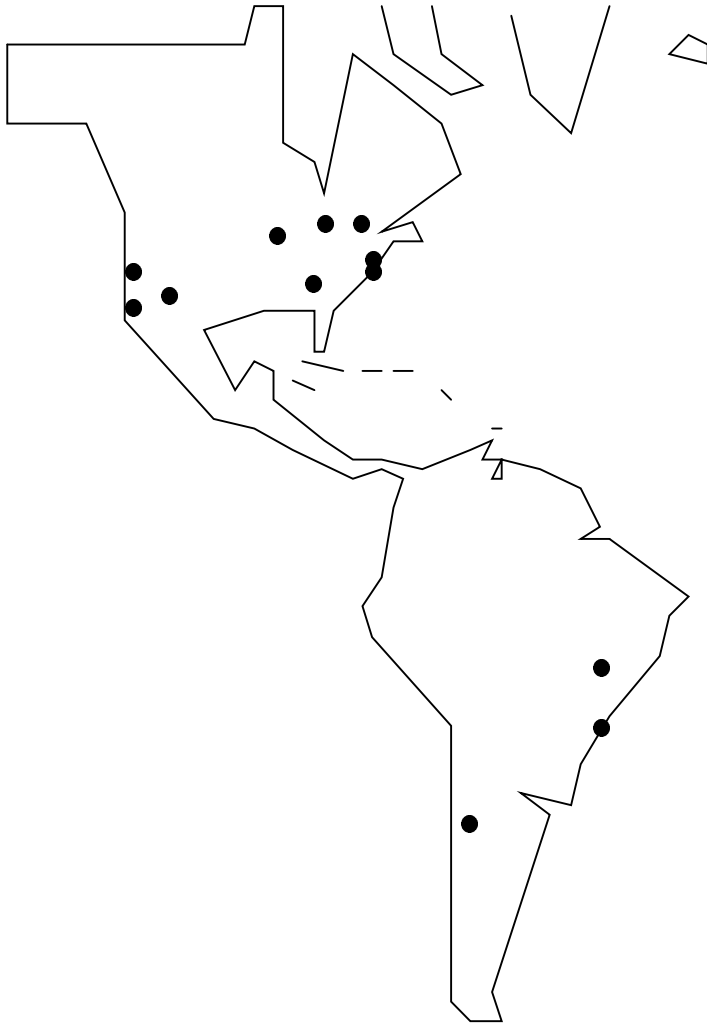
# User's point of view



Applications to other areas:

Web retrieval, XML processing, NL processing, text mining, multimedia search, bioinformatics, ....

# A Geographical View



Text Research Map

+ Melbourne, Australia

## Theory vs. Practice

How we can measure the goodness of an algorithm?

- Asymptotic worst case behavior
- Asymptotic average case behavior
- Practical behavior

D. Knuth [IFIP'89 Invited speech]

- Balance between theory and practice
- Software is hard

*The best theory*  
*is*  
*inspired by practice*  
—  
*The best practice*  
*is*  
*inspired by theory*

## Problem

- $\Sigma$ : finite alphabet of size  $\sigma$
- Text  $t \in \Sigma^*$  of length  $n$
- Pattern  $p \in \Sigma^*$  of length  $m$  ( $m \ll n$ )  
 $m$  is considered bounded
- Problem: find all occurrences of  $p$  in  $t$
- Space complexity  $S$ : the extra space used for the search (index)  
RAM model: words of size  $O(\log n)$
- Time complexity  $C$ : time needed to find the pattern  
or equivalent measure (for example, comparisons)
  - Worst-case
  - Average-case (uniform text and pattern)

## Search Models

1.  $p$  is a word
2.  $p$  is any sequence starting in an index-point

Some data structures assume the first model

## Answer Models

- exact match
- approximate match (distance function needed)
- closest match or all matches at a certain distance

## Computation Models

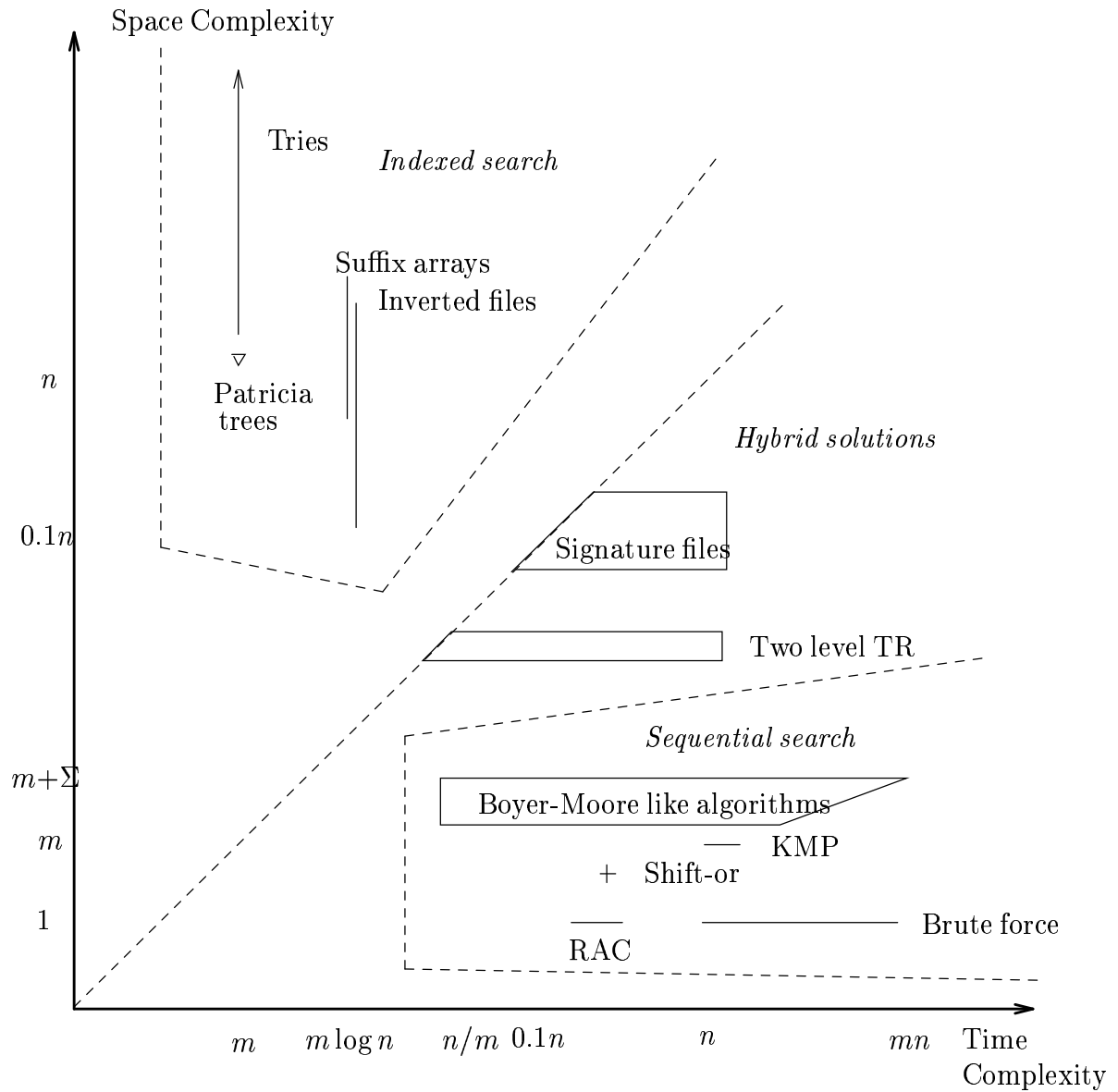
- Text-Pattern comparisons
- Arithmetical/Bitwise operations

## Algorithmic point of view

Input data:

- Raw pattern and text
  - sequential, on-line, real-time algorithms
- Preprocessing of the pattern
  - pattern is known in advance
- Preprocessing of the text → index
  - Inverted files
  - Tries
  - Patricia trees
  - Suffix arrays
- Hybrid solutions:
  - Filters: signature files
  - Two Level TR
- Compression: index, text, both!

# Space-Time Trade-Offs



## Existing Space-Time Trade-Offs

- Linear search and constant space:  $O(n)$
- Linear search and bounded space:  $O(nm)$
- Linear search and linear space:  $O(n^2)$
- Sub-linear search and bounded space:  $O(n \log m)$
- Sub-linear search and sub-linear space:  $O(nm^2 / \log^2 n)$
- Logarithmic search and linear space:  $O(mn \log n)$
- Bounded time search and linear space:  $O(mn)$
- Bounded time search and super-linear space:  $O(mn^2)$

## Examples

| Class complexity  | Worst-case   | Average-case                                  |
|-------------------|--|---|
| $n$               | Random access compression  | Brute-force                                   |
| $n \log m$        | —  | Boyer-Moore                                   |
| $nm^2 / \log^2 n$ | Shift-or   | Shift-or                                      |
| $mn$              | Brute-force<br>Knuth-Morris-Pratt<br>Boyer-Moore<br>Patricia trees | Knuth-Morris-Pratt<br>Tries<br>Patricia trees |
| $mn \log n$       | Inverted files<br>Suffix arrays                                    | Inverted files<br>Suffix arrays               |
| $n^2$             | Signature files<br>Two-level TR                                    | Signature files<br>Two-level TR               |
| $mn^2$            | Tries  | —   |

## String searching: Definition

- Basic problem: find exact occurrences of a pattern in a text
- Variations
  - Allow mismatches
  - Allow mismatches, insertions and deletions
- Examples:

text

text

text: This is a text example ...

t ext

ex

Example: *grep* command in Unix.

# Complexity

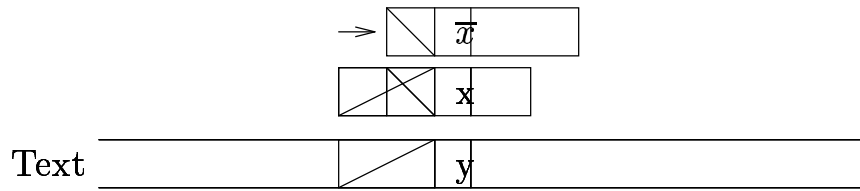
$n$ : size of the text

$m$ : size of the pattern

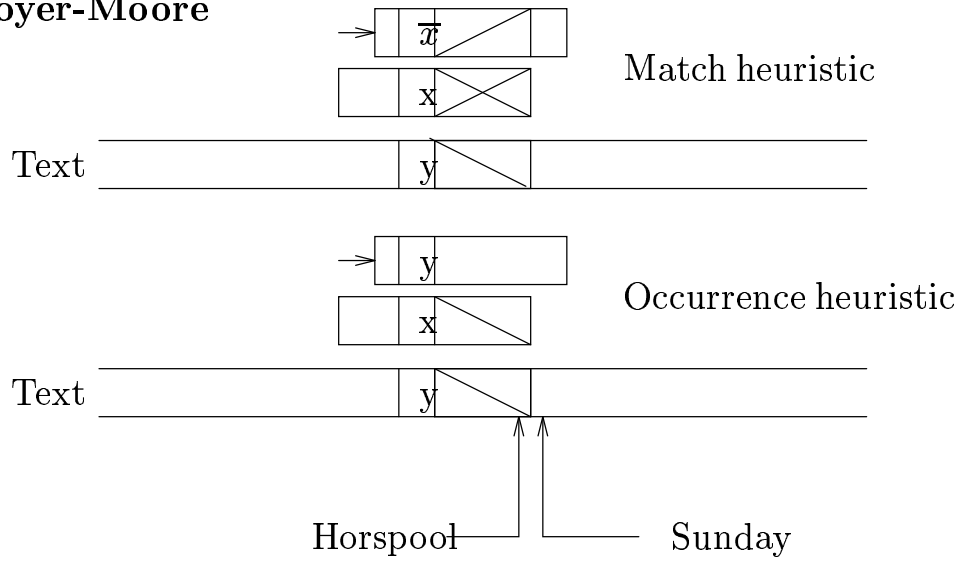
- Raw text
  - Worst case:  
lower and upper bound of  $n + O(n/m)$  comparisons
  - Average case:  $O(\log m n/m)$  lower and upper bound
- Preprocessed text:
  - Index construction:  $O(n)$  time and space (finite alphabet)
  - Worst case:  $O(m)$  comparisons
  - Average case:  $\min(m, \log n)$  comparisons

# Classical Algorithms

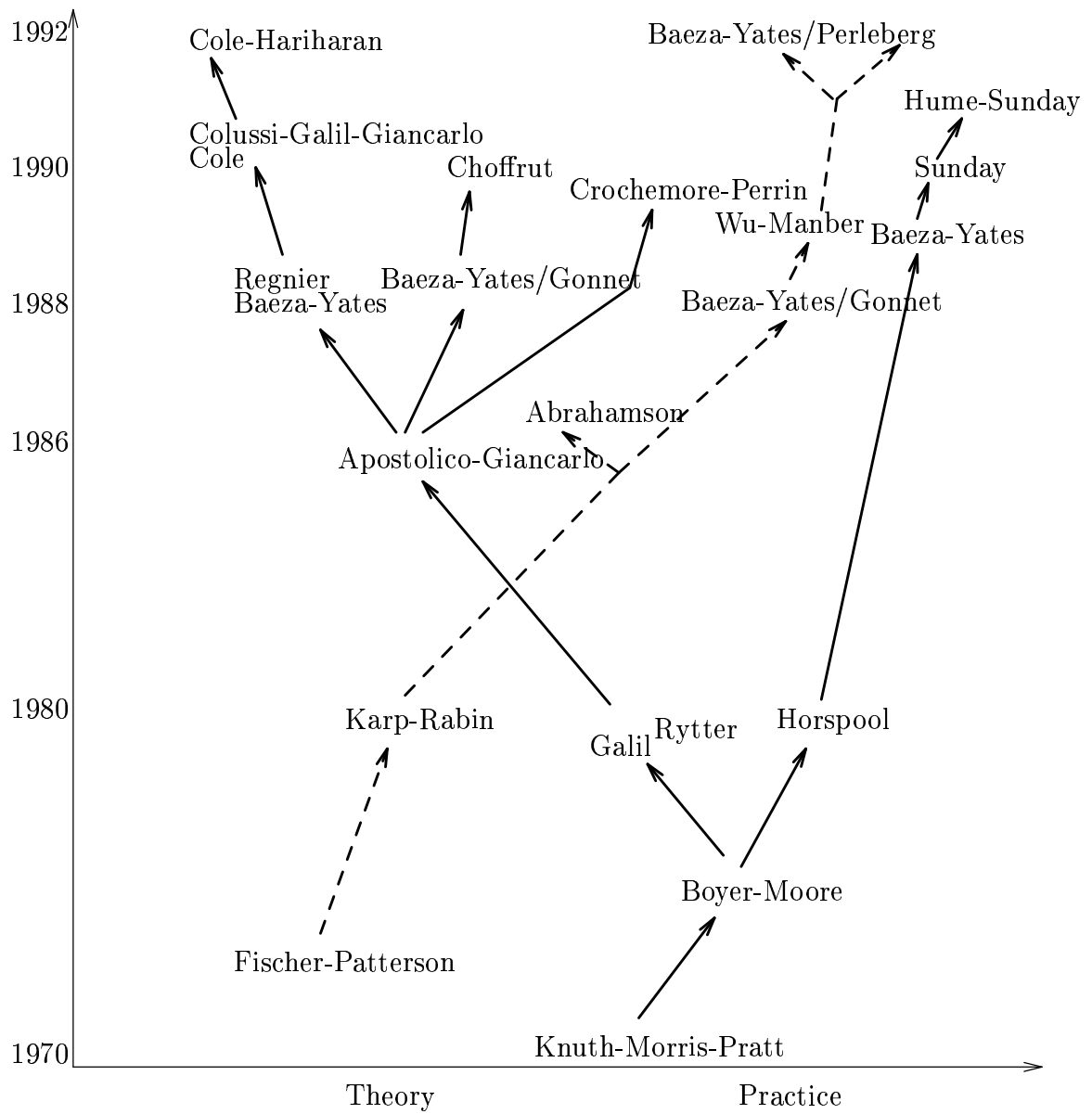
## Knuth-Morris-Pratt



## Boyer-Moore



# String Searching: Historical View



# Knuth-Morris-Pratt Algorithm

Fascinating story... from theory and practice

Preprocessing:

$$\text{next}[j] = \max\{i | (\text{pattern}[k] = \text{pattern}[j - i + k] \text{ for } k = 1, \dots, i - 1) \\ \text{and } \text{pattern}[i] \neq \text{pattern}[j]\} ,$$

for  $j = 1, \dots, m$ .

Example:

|         |   |   |   |   |   |   |   |   |   |   |   |   |
|---------|---|---|---|---|---|---|---|---|---|---|---|---|
|         | a | b | r | a | c | a | d | a | b | r | a |   |
| next[j] | 0 | 1 | 1 | 0 | 2 | 0 | 2 | 0 | 1 | 1 | 0 | 5 |

Extension to multiple patterns: Aho-Corasick

# Algorithm

```
search( text, n, pat, m ) // Search pat[1..m] in text[1..n]
char text[], pat[];
int n, m;
{
    int next[MAX_PATTERN_SIZE];

    pat[m+1] = CHARACTER_NOT_IN_THE_TEXT;
    kmp( pat, m+1, pat, m+1, next ); // Preprocess pattern
    kmp( text, n, pat, m, next );    // Search text
    pat[m+1] = END_OF_STRING;
}
```

```
kmp( text, n, pat, m, next )
char text[], pat[];
int n, m, next[];
{
    static dosearch = 0;
    int i, j;

    i = 1;
    if( !dosearch ) // Preprocessing
        j = next[1] = 0;
    else j = 1;
    do {
        if( j == 0 || text[i] == pat[j] )
        {
            i++; j++;
            if( !dosearch ) { // Preprocessing
                if( text[i] != pat[j] ) next[i] = j;
                else next[i] = next[j];
            }
        }
        else j = next[j];

        if( dosearch && j > m ) { // Search
            Report_match_at_position( i-m );
            j = next[m+1];
        }
    }
    while( i <= n ) ;
    dosearch = 1;
}
```

Worst case complexity:  $2n + O(\text{matches})$

# Boyer-Moore-Horspool-Sunday Algorithm

Match heuristic can be extended: BM automata, suffix automata

In practice the occurrence heuristic is the key issue:

$$d[x] = \min\{s \mid s = m+1 \text{ or } (1 \leq s \leq m \text{ and } pattern[m+1-s] = x)\} .$$

```
search( text, n, pat, m ) // Search pat[1..m] in text[1..n]
char text[], pat[];
int n, m;
{
    int d[MAX_ALPHABET_SIZE], i, j, k, lim;

    // Preprocessing
    for( k=0; k<MAX_ALPHABET_SIZE; k++ )
        d[k] = m+1;
    for( k=1; k<=m; k++ )
        d[pat[k]] = m+1-k;
    // Search
    lim = n-m+1;
    for( k=1; k <= lim; k += d[text[k+m]] )
    {
        i=k;          // Could optimal order
        for( j=1; j<=m && text[i] == pat[j]; j++ )
            i++;
        if( j == m+1 )
            Report_match_at_position( k );
    }
}
```

Complexity ranges between  $n/(m+1)$  and  $O(nm)$

# Randomized algorithm: Karp-Rabin

```
#define Q1 33554393
#define Q 16647143
#define D 7

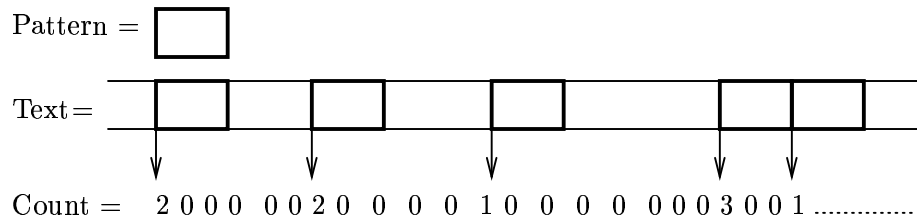
rksearch( text, n, pat, m )
char text[], pat[];
int n, m;
{
    int h1, h2, dM, i, j;

    dM = 1;
    for( i=1; i<m; i++ ) dM = (dM << D) % Q;
    h1 = h2 = 0;
    for( i=1; i<=m; i++ )
    {
        h1 = ((h1 << D) + pat[i] ) % Q;
        h2 = ((h2 << D) + text[i] ) % Q;
    }
    i = 1;
    while( i <= n-m+1 )
    {
        if( h1 == h2 ) // probabilistic match
        {
            for(j=1; j<=m && text[i-1+j] == pat[j]; j++ ); // check
            if( j > m ) // real match
                matches++;
            else printf( "Collision...\n" );
        }
        h2 = (h2 + (Q << D) - text[i]*dM ) % Q;
        h2 = ((h2 << D) + text[i+m] ) % Q;
        i++;
    }
}
```

Complexity is linear and in practice not good

## Counting: Baeza-Yates/Perleberg, 1992

- Idea: Count the number of matches for all possible positions of the pattern
- Straight implementation: Brute force algorithm with  $O(mn)$  worst and average case time



## Improvements

- Preprocess the pattern computing which characters of the alphabet should update a counter
- We need only the last  $m$  counters
- $O(m + |\Sigma|)$  extra space

## Example

```
Pattern = t h a n  
Text =   t h i s   i s   a n   e x a m p l e   t h a t .....  
        |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  
        |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  
Count =  2 0 0 0  0 0 2 0  0 0 0  1 0  0 0  0 0 0 0  0 0 0 3 0 0 1 .....
```

Each step is:

```
For all j such that pattern[j] = text[i]
    increment count[i-j+1]
```

## Code

```
for (i=0; i<n; i++) {  
    if ((off1=(aptr=&alpha[c=*t++])->offset) >= 0) {  
        count[(i+off1)&MOD256]--;  
        for (aptr=aptr->next; aptr!=NULL; aptr=aptr->next)  
            count[(i+aptr->offset)&MOD256]--;  
    }  
    if (count[i&MOD256] <= k) printf("%d",count[i&MOD256]);  
    count[i&MOD256] = m;  
}
```

## Running time

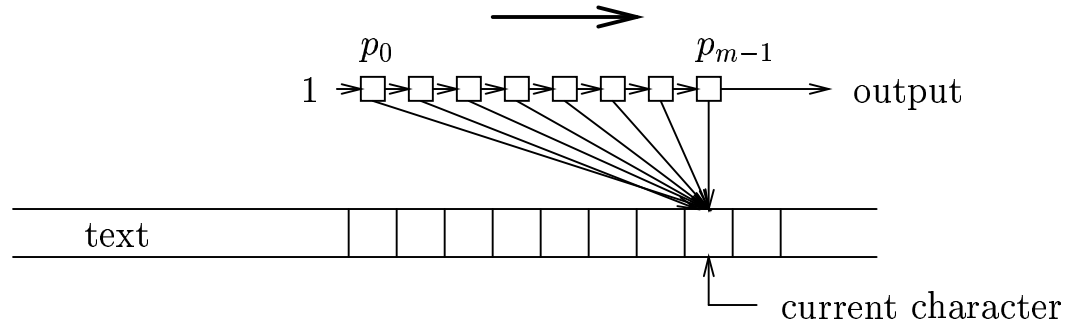
- $O(R + n + m + |\Sigma|)$  total cost
- $R$  is the number of text-pattern symbol matches

$$0 \leq R \leq f_{max}n \leq m n$$

- On average  $R = O(\frac{m}{|\Sigma|}n)$
- Cost is independent of number of mismatches
- Not suitable for  $m \gg \Sigma$  (e.g. DNA)

# Bit Parallelism: Baeza-Yates/Gonnet, 1989

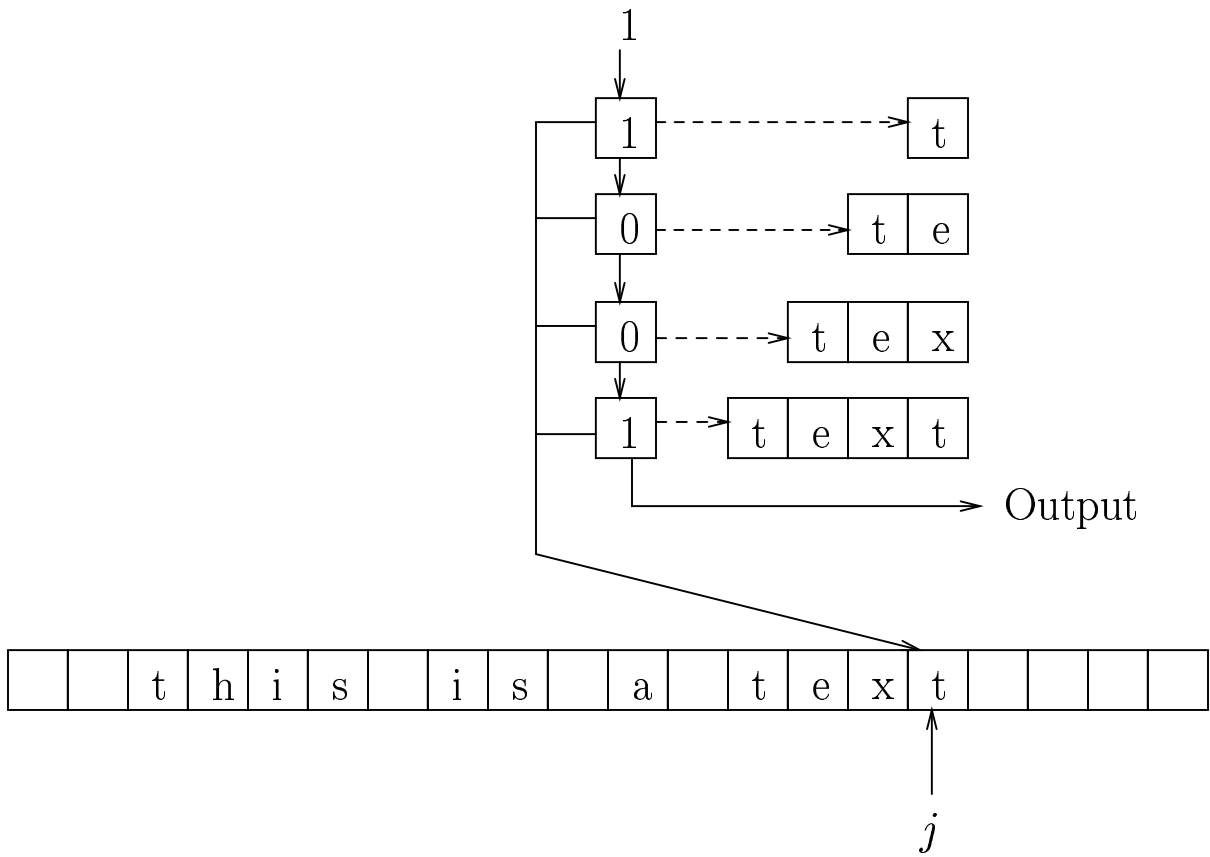
Parallel algorithm using  $m$  processors



Processor  $i$ :  $1$  if  $p_0, \dots, p_i = t_{j-i}, \dots, t_j$   
 $0$  otherwise

$$p_i^{j+1} \leftarrow p_{i-1}^j \ \& \ (\text{pattern}[i] \ =? \ \text{text}[j])$$

# Example:



## Bit sequence simulation

One bit per processor  $\rightarrow$  simulation with a bit vector!

$$\vec{p} \leftarrow ( \textit{shift left } \vec{p} ) \ \& \ \begin{pmatrix} p \\ a \quad =? \textit{ text}[j] \\ t \end{pmatrix}$$

For finite alphabets, all possible comparisons can be precomputed before the search

$$\vec{p} \leftarrow ( \textit{shift left } \vec{p} ) \ \& \ T[\textit{text}[j]]$$

In the example:

|   | T[t] | T[e] | T[x] | T[*] |
|---|------|------|------|------|
| t | 1    | 0    | 0    | 0    |
| e | 0    | 1    | 0    | 0    |
| x | 0    | 0    | 1    | 0    |
| t | 1    | 0    | 0    | 0    |

$0 \longleftrightarrow 1 \rightarrow$  shift-and/or algorithm

Handbook of Algorithms and Data Structures, 2<sup>nd</sup> Ed, 1991

# Complexity

For the uniform cost RAM model, we have

$$\text{word size} = O(\log_2 n)$$

- Preprocessing time:  $O(\Sigma + m)$
- Search time:  $O(mn/\log n)$
- Space needed:  $O(\Sigma m/\log n)$  words

## Code:

```
// Preprocessing
for( i=0; i<MAXSYM; i++ ) T[i] = ~0;
for( lim=0, j=1; *pattern != EOS; lim |= j, j <<= B, pattern++ )
    T[*pattern] &= ~j;
lim = ~(lim >> B);
// Search
matches = 0; state = ~0;           // Initial state
for( ; *text != EOS; text++ )
{
    state = (state << B) | T[*text]; // Next state
    if( state < lim )
        // Match at current position-len(pattern)+1
}
```

## Extensions

- Every pattern element is a class of symbols

Just change  $T$ !

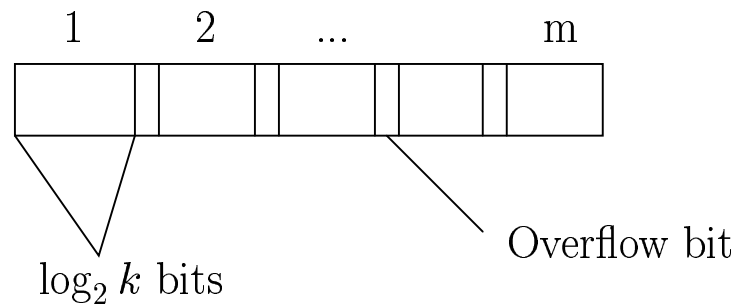
- Don't care symbols on the text:  $T[*] = 11\dots 1$

- Multiple patterns: just one longer sequence

- Mismatches: count the number of mismatches

+ instead of  $\&$ , using  $\log k + 1$  bits per position

$k$  is maximal number of mismatches allowed



- Insertions and deletions [Wu & Manber, 1991]

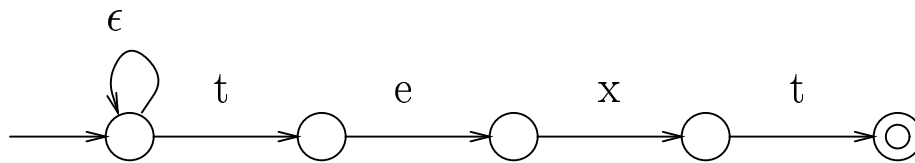
$k + 1$  bit sequences

Agrep: fastest approximate search tool for Unix

# From Automata to Bit-parallelism

Exploit automata structure

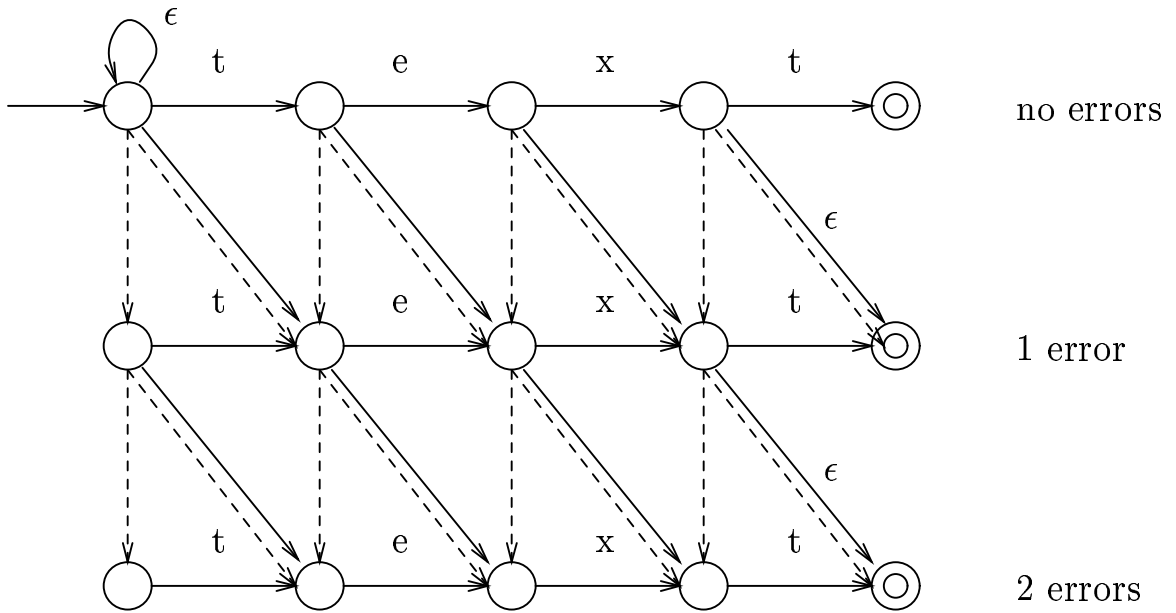
Consider the NFA to search for **text**



- Processors in 1  $\iff$  Active states of standard simulation
- Be careful with  $\epsilon$ -closure
- Related to hardware implementations

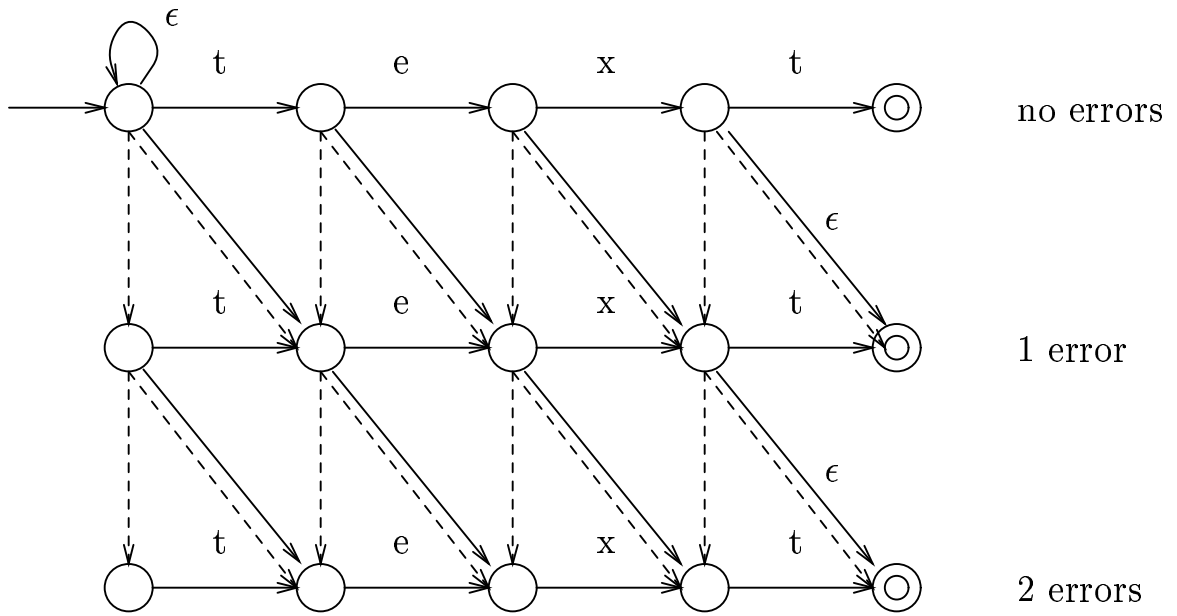
# Approximate string searching

Consider the NFA for searching **text** with at most  $k = 2$  errors



Longest positional match wanted?

# Horizontal bit parallelism: Wu & Manber



$$\vec{R}'_k = (\text{shift } \vec{R}_k \& T[\text{text}[j]]) \text{ or } \vec{R}_{k-1} \text{ or } (\text{shift}(\vec{R}_{k-1} \text{ or } \vec{R}'_{k-1}))$$

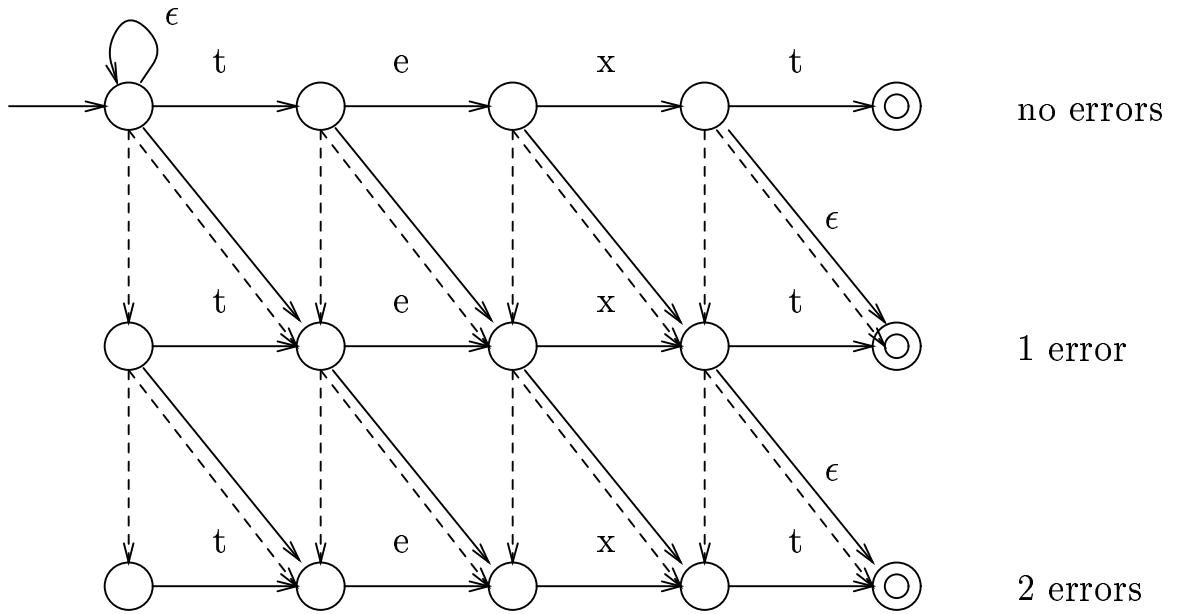
Initially  $\vec{R}_k = 11..100..0$  ( $k$  ones)

Drawback: Dependency on  $R'_{k-1}$

Complexity:  $O(kmn/\log n)$  search time

$O((k + \Sigma)m/\log n)$  space

Vertical bit parallelism:



Key information: highest (smallest error) active state per column

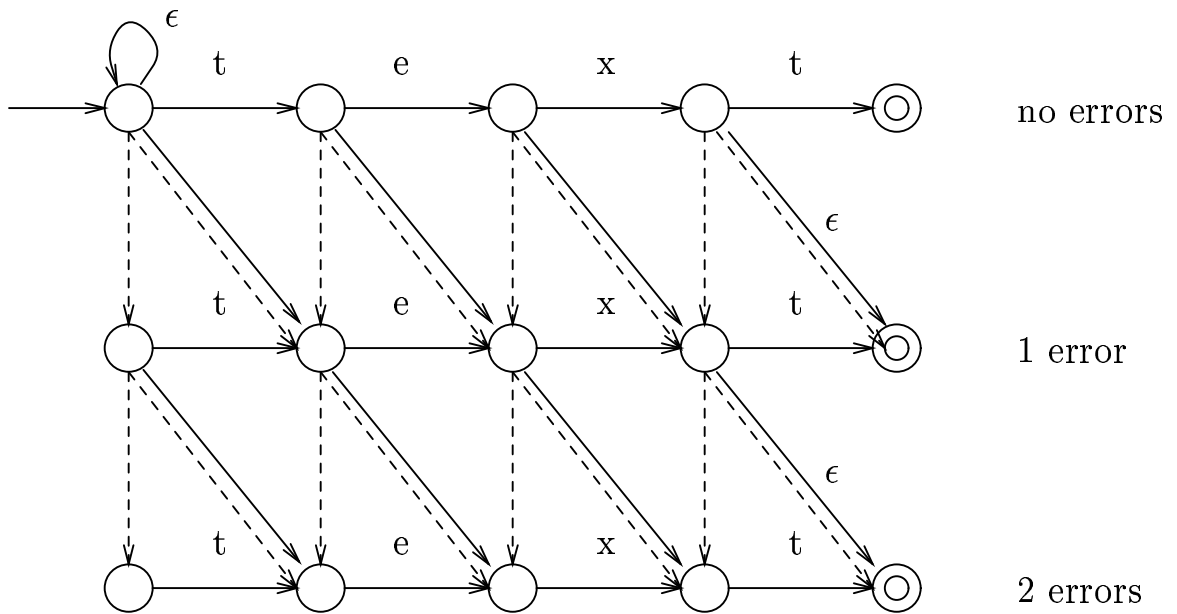
State of the search:  $m$  numbers on the range  $0 \dots k + 1$ .

$$R'_i = \min( R_{i-1} + (\text{text}[j] == \text{patt}[i - 1]), R_i + 1, R'_{i-1} + 1 )$$

Initially  $R_0 = 0$  and  $R'_0 = R_0$

Related to dynamic programming:

- Longest common subsequence, string editing
- Related to Ukkonen's automata approach



Each diagonal represents an  $\epsilon$ -closure (longest match)

State of the search:  $m - k + 1$  numbers on the range  $0 \dots k + 1$ .

$$R'_i = \min( (\ell - i)(\exists \ell \geq R_{i-1} + i - 1 \mid \text{text}[j] == \text{patt}[\ell]), \\ R_i + 1, R_{i+1} + 1, k + 1 ) \quad R'_0 = 0$$

Advantage: all  $R'_i$ s can be computed in parallel

Related to:

- Simulation of DP over a suffix array (also Ukkonen [CPM'93])
- Application: sequence comparison on biological DB

# Approximate String Matching: Dynamic Programming

- Minimum number of errors to match  $P_{1..i}$  to a suffix of  $T_{1..j}$

$$C[0, j] = 0$$

$$C[i, 0] = i$$

$$C[i, j] = \text{if } (P_i = T_j) \text{ then } C[i - 1, j - 1]$$

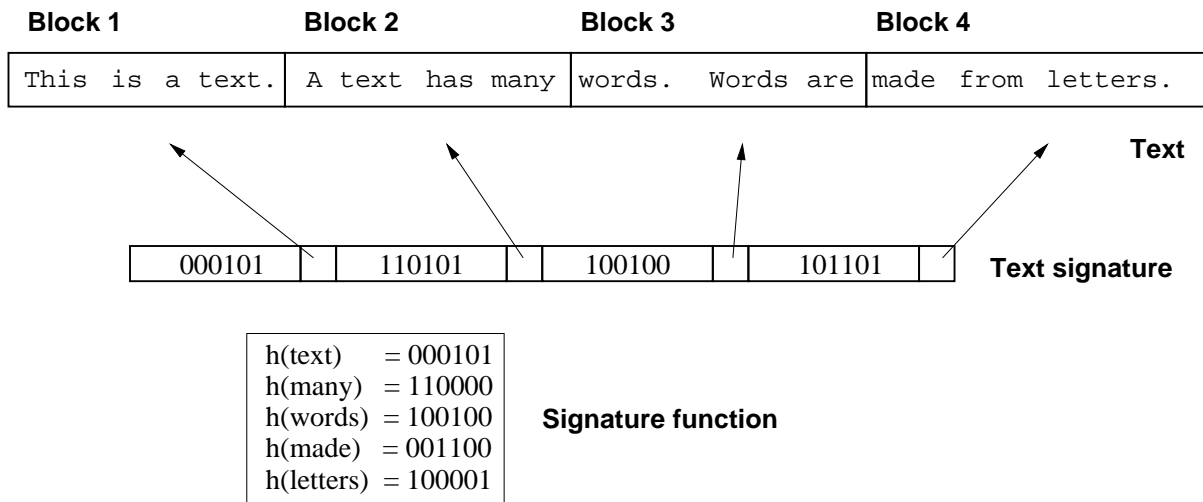
$$\text{else } 1 + \min(C[i - 1, j], C[i, j - 1], C[i - 1, j - 1])$$

- Example:

|          |   |          |          |          |          |          |          |          |
|----------|---|----------|----------|----------|----------|----------|----------|----------|
|          |   | <b>s</b> | <b>u</b> | <b>r</b> | <b>g</b> | <b>e</b> | <b>r</b> | <b>y</b> |
|          | 0 | 0        | 0        | 0        | 0        | 0        | 0        | 0        |
| <b>s</b> | 1 | 0        | 1        | 1        | 1        | 1        | 1        | 1        |
| <b>u</b> | 2 | 1        | 0        | 1        | 2        | 2        | 2        | 2        |
| <b>r</b> | 3 | 2        | 1        | 0        | 1        | 2        | 2        | 3        |
| <b>v</b> | 4 | 3        | 2        | 1        | 1        | 2        | 3        | 3        |
| <b>e</b> | 5 | 4        | 3        | 2        | 2        | 1        | 2        | 3        |
| <b>y</b> | 6 | 5        | 4        | 3        | 3        | <b>2</b> | <b>2</b> | <b>2</b> |

# Signature Files

- Hash every word to a small bit sequence
- Index: concatenation of the bit sequences
- Space: 5 to 10% of the text
- Search: hash the word pattern, search in the index
- Problem: false matches
- Superimposed coding: hash more than one word to one code



- Optimal size to minimize false matches

# Inverted Files

Idea: all words and their positions

|   |   |   |    |    |    |    |    |    |    |    |    |    |    |
|---|---|---|----|----|----|----|----|----|----|----|----|----|----|
| 1   | 6 | 9 | 11 | 17 | 19 | 24 | 28 | 33 | 40 | 46 | 50 | 55 | 60 |
| This is a text. A text has many words. Words are made from letters. |   |   |    |    |    |    |    |    |    |    |    |    |    |

| Vocabulary | Occurrences |
|------------|-------------|
| letters    | 60...       |
| made       | 50...       |
| many       | 28...       |
| text       | 11, 19...   |
| words      | 33, 40...   |

Text

**Inverted Index**

Vocabulary search: Hashing, sorted, etc.

Granularity depends in what we want to answer: file, word, byte

# Inverted Files: Space

- Vocabulary: Heaps' law

$$V = C \times n^\beta$$

- Posting file: linear space (one occurrence = one pointer)

- Word distribution: Zipf's law

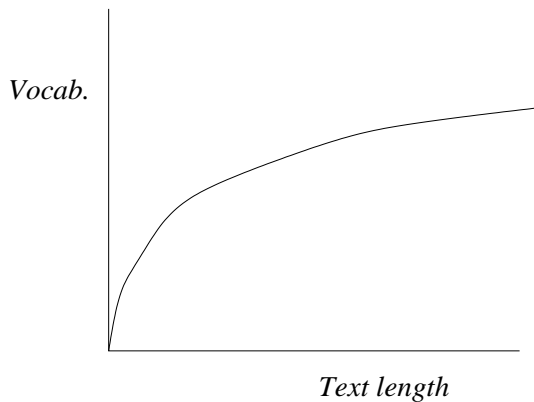
$$w_r = \frac{N}{r^\theta H_N(\theta)}$$

where

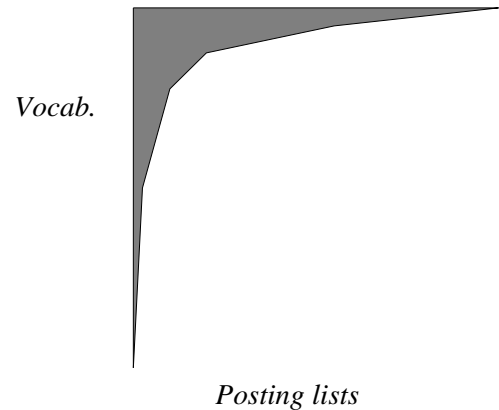
$$H_N(\theta) = \sum_{r=1}^k \frac{1}{r^\theta}$$

- Stopwords: half the posting file

- Linear space



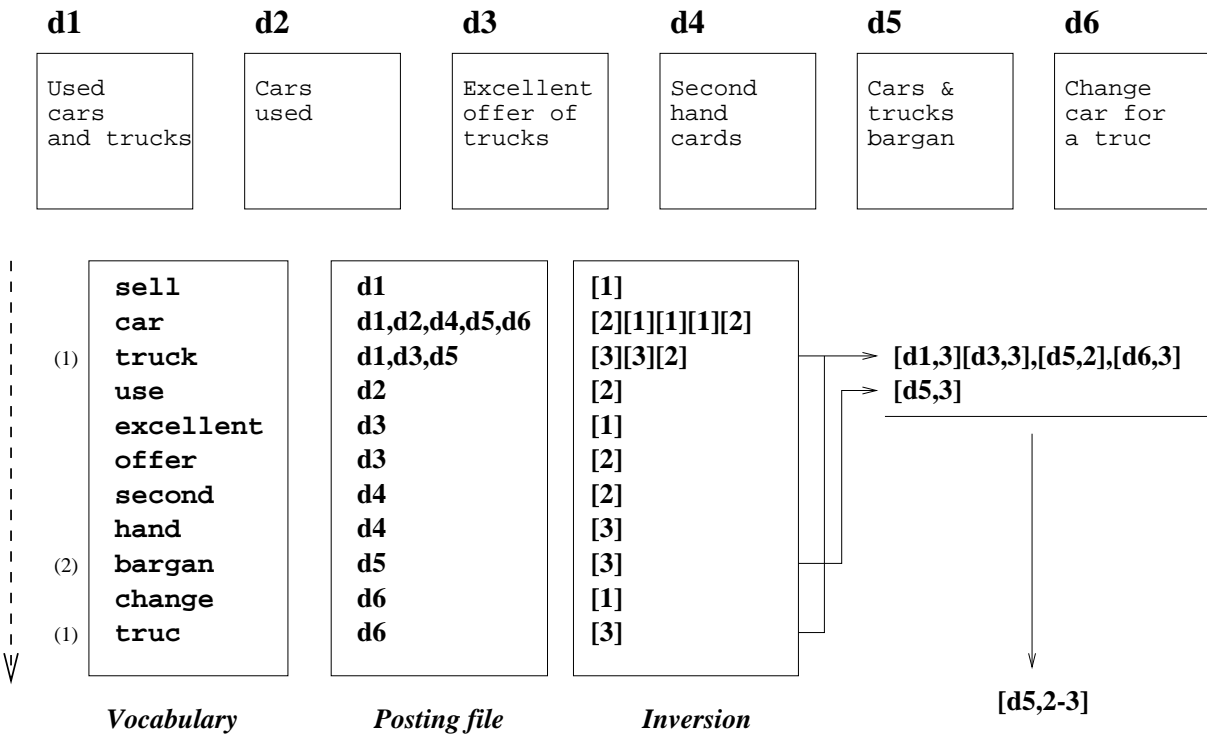
**Heaps' Law**



**Zipf's Law**

# Complex Patterns

Search the vocabulary sequentially and do set operations

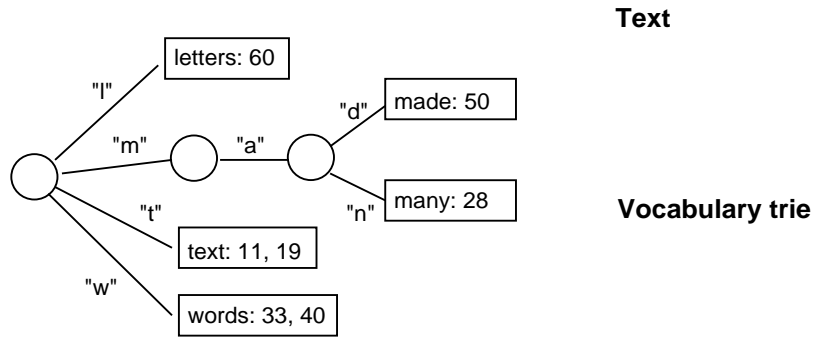


"bargain for trucks" [approximate]

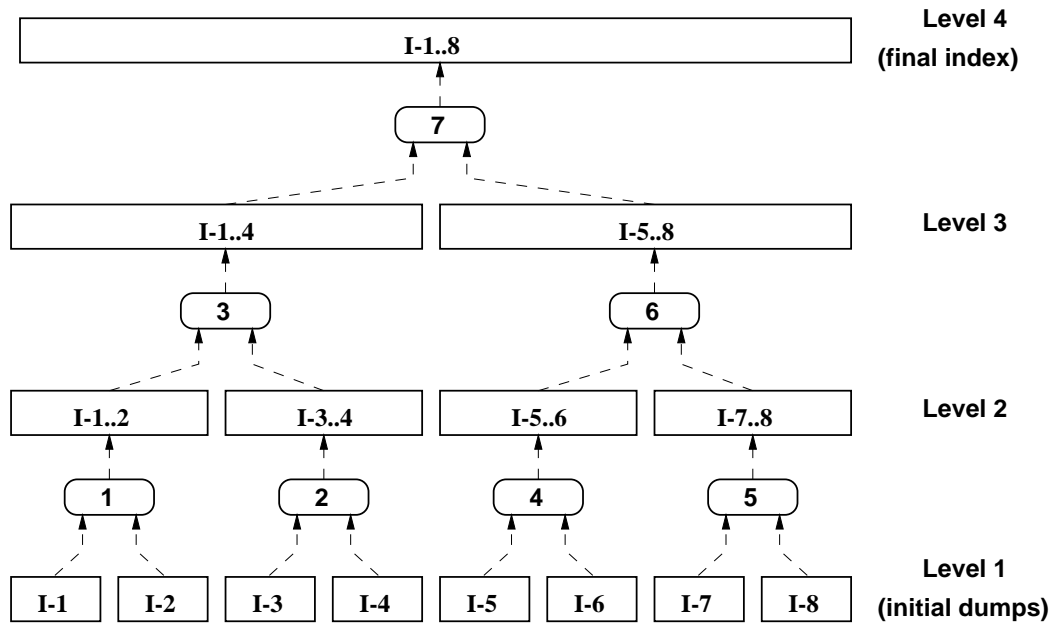
# Building Inverted Files

- Process text pieces as large as possible

1    6   9 11    17 19    24   28   33    40   46   50   55   60  
 This is a text. A text has many words. Words are made from letters.



- Merge partial indexes



## **Two-level Text Retrieval:** Block addressed inverted files

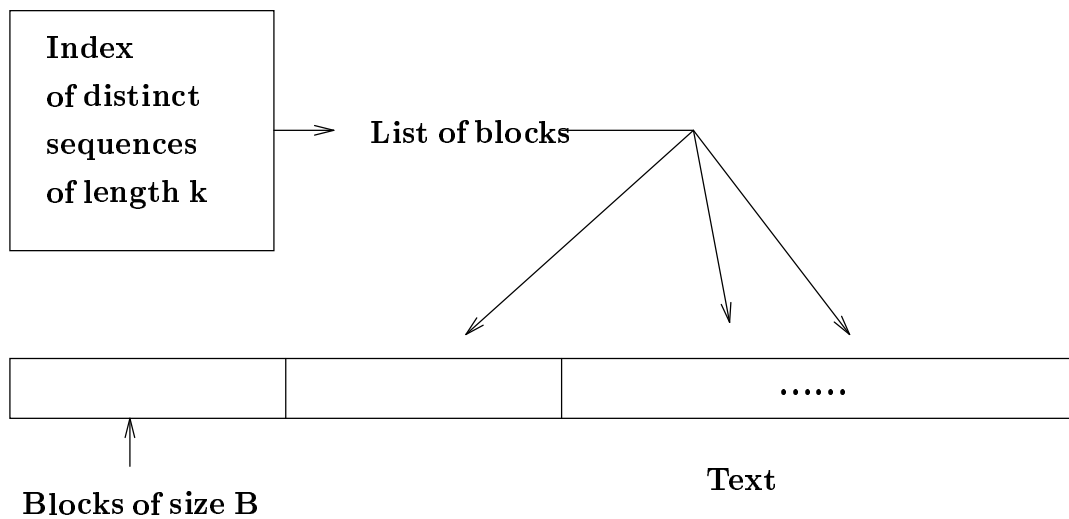
Idea: Used in PIRS (Personal Information Retrieval System)

[Wu and Manber 1993]

- The text is divided in 256 blocks of the same size
- An inverted file of all the different words of the text is built
- Each entry indicates only the blocks where the word appears  
→ 1 byte per block
- First we search in the inverted file  
next in the corresponding blocks using a fast sequential algorithm  
Complexity depends on the number of occurrences  
→ locality of reference is important
- For large texts, empirical results show that the index requires  
less than 5% of the text size
- This idea works reasonable well up to 200Mbs

# Generalization

- Index storing all different sequences of length  $k$  of the text starting on index-points
- Every entry points to all the blocks of the text having that sequence
- Block size is also a parameter, denoted by  $B$
- Searching is done as before taking care of the relation between  $m$  and  $k$



## Remarks

- $k$  models the granularity of the index
- If  $k = 0$ , we get sequential search
- If  $k = O(\log n)$  we have a full index
- Index has to provide fast search for prefix searching
- $B$  represents the granularity for accessing the text
  - equivalent to document size in traditional IR systems
- Size of  $B$  should be based on the granularity of secondary storage
  - important implementation issue

## Worst-case Analysis

- There are at most  $\sigma^k$  different sequences of length  $k$
- To address inside the index we need at most  $O(k \log \sigma)$  bits
- To address a block we need  $\log(n/B)$  bits
- We can use a sorted array with the sequences, each one having a pointer to a posting file, which stores block numbers
- Maximum number of blocks:  $n/B$
- Total space needed, in bits, is

$$S = \sigma^k(2k \log \sigma + \log(n/B)) + \sigma^k n/B \log(n/B) = O(\sigma^k \frac{n}{B} \log n)$$

which could be  $O(n^2)$  bits for  $k$  and  $B$  of  $O(\log n)$

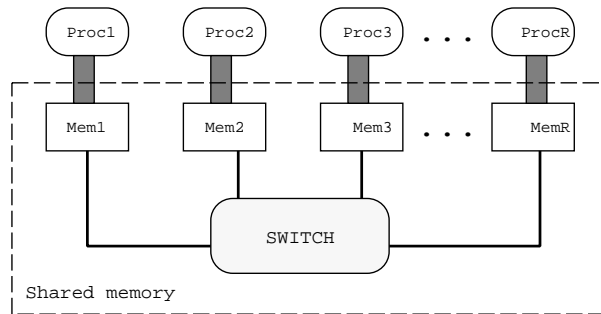
- The number of comparisons in the worst case is  $k \log(\sigma^k) = k^2 \log \sigma$  in the index (or  $mk$  for  $m \leq k$ )
- The search time in the blocks is  $O(n)$  in the worst case

## Inverted File Space in Practice

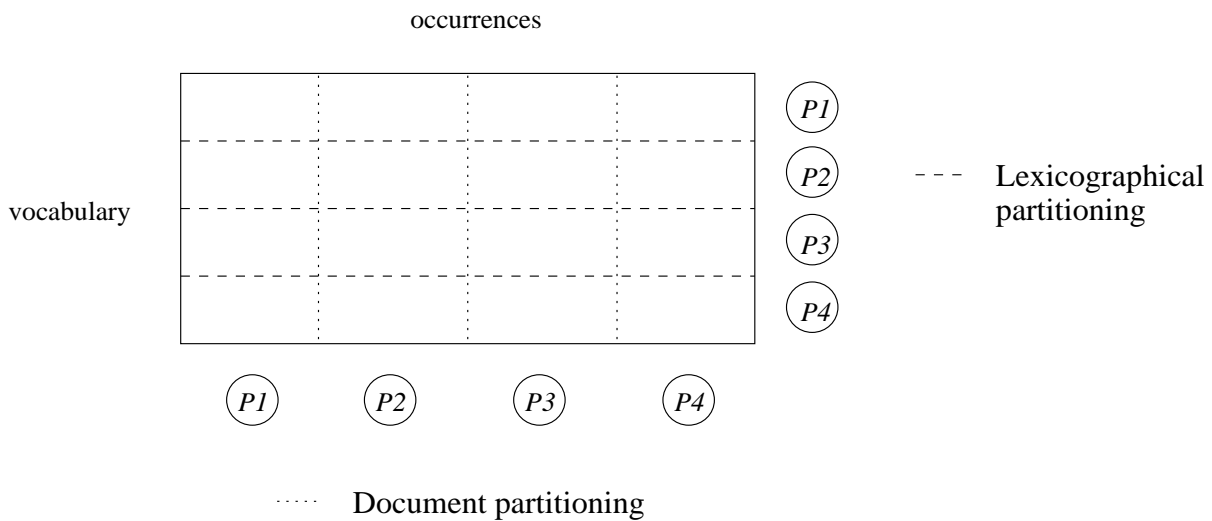
| Index                     | Small base<br>(1 MB) |     | Medium base<br>(200 MB) |      | Large base<br>(2 GB) |      |
|---------------------------|----------------------|-----|-------------------------|------|----------------------|------|
|                           |                      |     |                         |      |                      |      |
| Full<br>inverted          | 45%                  | 73% | 36%                     | 64%  | 35%                  | 63%  |
| Document<br>addressing    | 19%                  | 26% | 18%                     | 32%  | 26%                  | 47%  |
| Block (64K)<br>addressing | 27%                  | 41% | 18%                     | 32%  | 5%                   | 9%   |
| Block (256)<br>addressing | 18%                  | 25% | 1.7%                    | 2.4% | 0.5%                 | 0.7% |

# Distributed Inverted Files

Hardware architecture model:



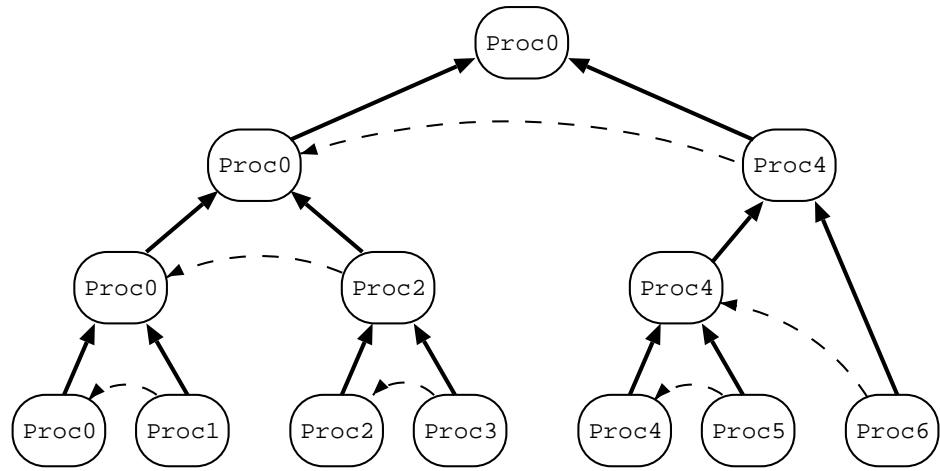
Possible solutions:



Two interesting measures:

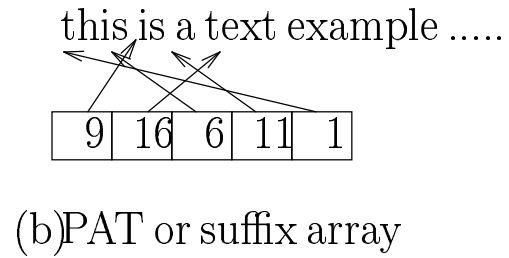
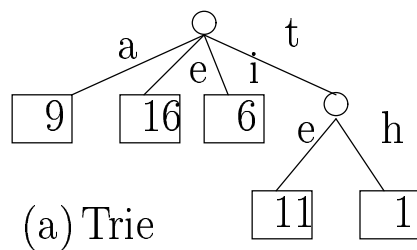
- *Throughput*: number of queries answered per second
- *Answer time*: time for a particular query

# Distributed Inversion



6

# Tries and Suffix Arrays



- Problem: space can be quadratic in a trie
- Patricia trees: cut unary trees
- To remember depth, a count is added at every node
- Space is now linear

Useful for complex queries: regular expressions in sublinear average time

# Suffix Arrays

to be at the beach or to be at work, that is the real question \$

↑ ↑ ↑ ↑ ↑     ↑ ↑ ↑ ↑ ↑     ↑     ↑ ↑ ↑ ↑ ↑

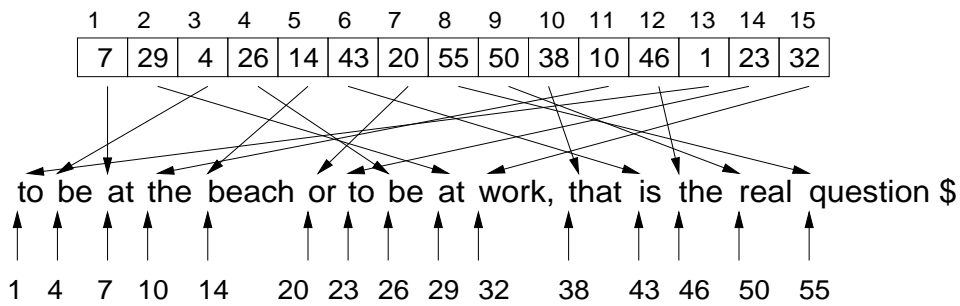
1 4 7 10 14     20 23 26 29 32     38     43 46 50 55

**Text**

**Index points**

- 1: to be at the beach or to be at work, that is the real question
- 4: be at the beach or to be at work, that is the real question
- 7: at the beach or to be at work, that is the real question
- 10: the beach or to be at work, that is the real question
- .....
- 55: question

**Suffixes**



**Suffix Array**

**Text**

**Index points**

# Suffix Array Search

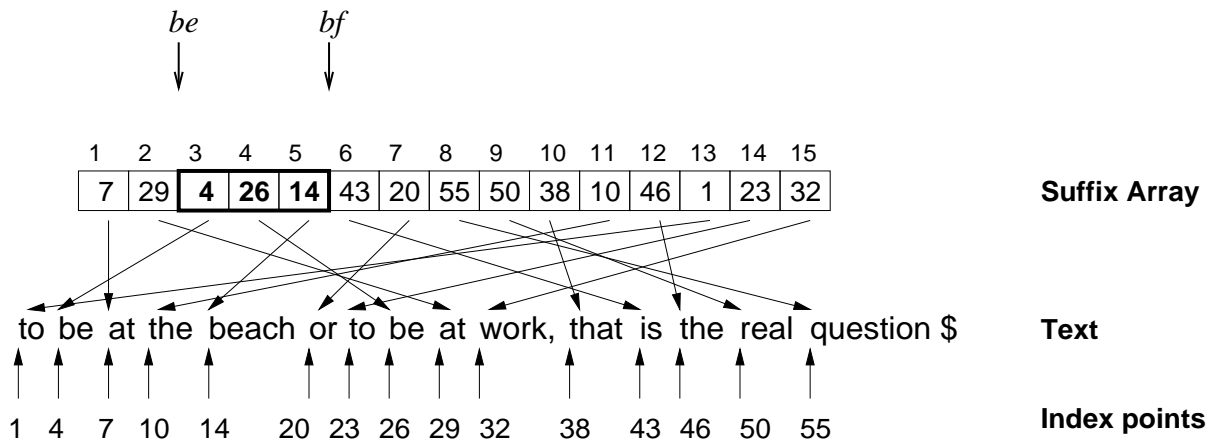
- Every substring is a prefix of a suffix
- The prefix relation can be used for lexicographic order

$$x : a \text{ prefijo de } y \iff (x : a \leq y) \wedge (y < x : (a + 1))$$

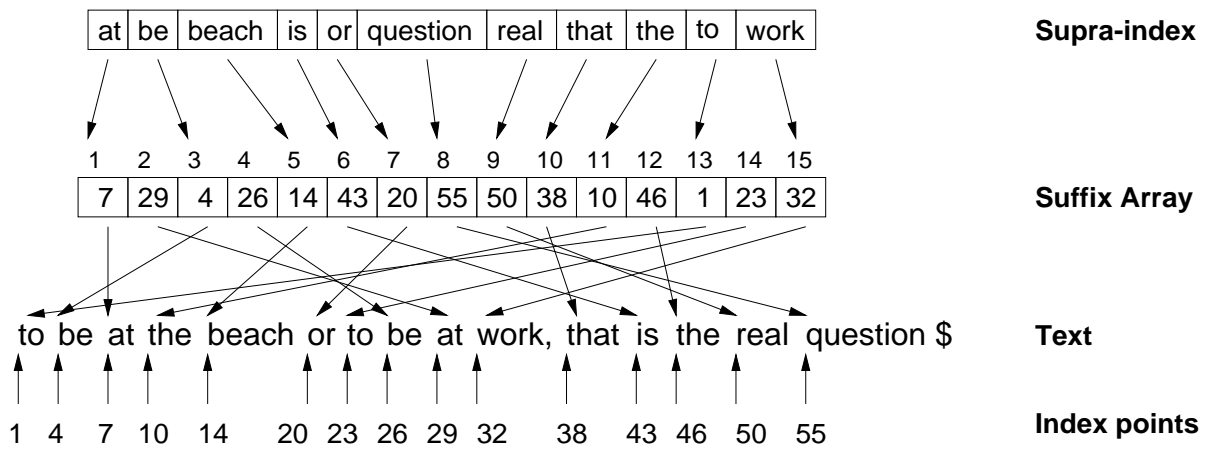
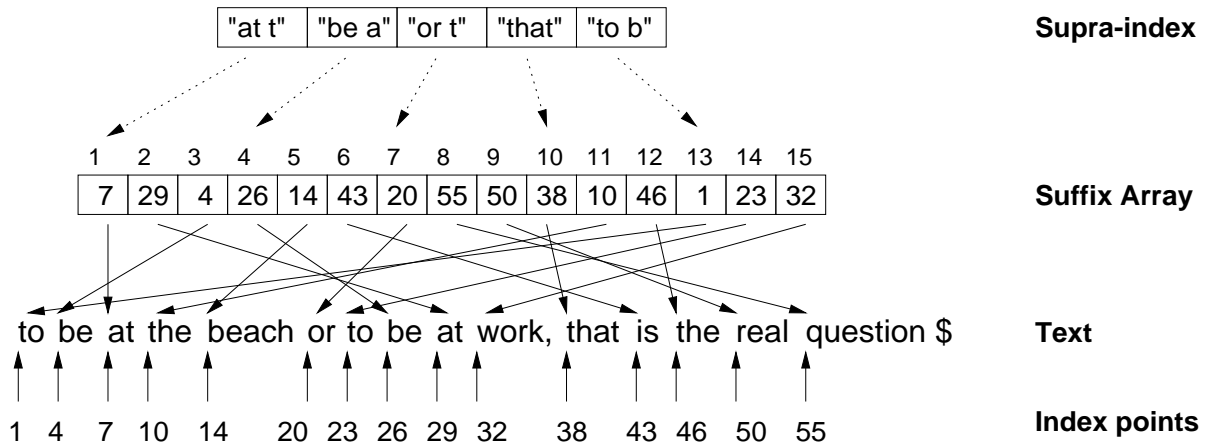
- Hence, two binary searches are enough to obtain the suffix array range where all occurrences of  $x$  appear

Number of occurrences: range size

- Time is logarithmic in the size of the array

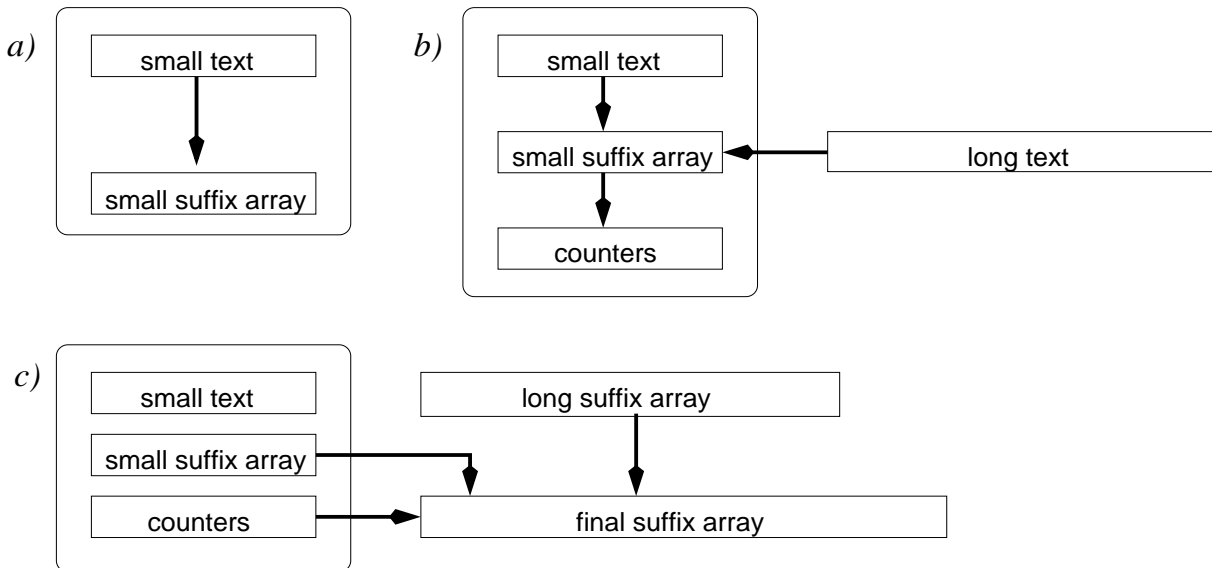


# Improvements



## Suffix Array: Construction

- In principle is a lexicographical order
- But suffixes are suffixes of suffixes
- However, random access to the text is the bottleneck
- Best solution: sequential scan with counting



## Future

- Further study on the power of non-comparison based algorithms

Many new algorithms

Examples: Bit-parallelism (agrep, shift-or) and counting.

- Problem reduction works for text searching

Example: Multiple string searching plus checking

- Two dimensional case [Baeza-Yates and Regnier, 1990]
- Approximate pattern matching [Wu and Manber, 1991]

- The final optimal algorithm depends on the input

Further study of input adaptive algorithms?

- Emphasize practical cases

- New uses for old concepts

Example:  $n$ -grams

- New text indices tailored to special cases

Example: approximate string matching

- Searching in metric spaces