String Matching Algorithms

Georgy Gimel'farb

(with basic contributions from M. J. Dinneen, Wikipedia, and web materials by Ch. Charras and Thierry Lecroq, Russ Cox, David Eppstein, etc.)

COMPSCI 369 Computational Science

1 String matching algorithms

2 Naïve, or brute-force search

3 Automaton search

4 Rabin-Karp algorithm

5 Knuth-Morris-Pratt algorithm

6 Boyer-Moore algorithm

7 Other string matching algorithms

Learning outcomes: Be familiar with string matching algorithms

Recommended reading:
http://www-igm.univ-mlv.fr/~lecroq/string/index.html

C. Charras and T. Lecroq: Exact String Matching Algorithms. Univ. de Rouen, 1997

String Matching (Searching)

String matching or searching algorithms try to find places where one or several strings (also called **patterns**) are found within a larger string (**searched text**):

```
...␣try␣to␣find␣places␣where␣one␣or␣several␣strings␣(also...
PATTERN: ace
...␣try␣to␣find␣places␣where␣one␣or␣several␣strings␣(also...
```

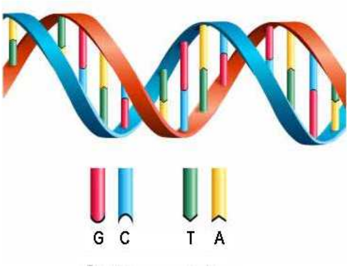
Formally, both the pattern and searched text are concatenation of elements of an **alphabet** (finite set) Σ

- Σ may be a usual human alphabet, for example, the Latin letters a through z or Greek letters α through ω
- Other applications may include binary alphabet, $\Sigma = \{0, 1\}$, or DNA alphabet, $\Sigma = \{A, C, G, T\}$, in bioinformatics

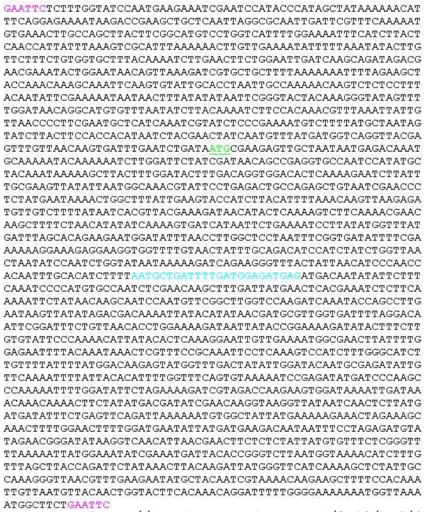
String Searching: DNA alphabet

DNA alphabet contains only four "letters", forming fixed pairs in the double-helical structure of DNA

- A – adenine: A pairs with T
- C – cytosine: C pairs with G
- G – guanine: G pairs with C
- T - thymine: T pairs with A



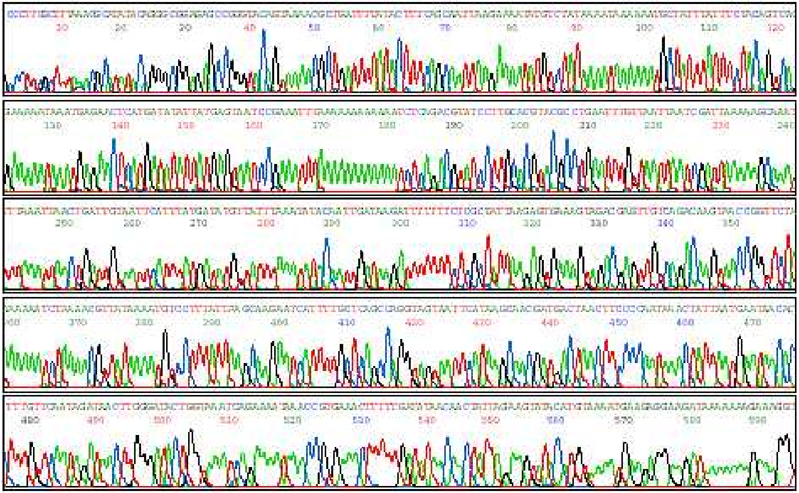
http://www.biotechnologyonline.gov.au/popups/img_helix.html



```
GAATTCTCTTTGGTATCCAATGAAGAAATCGAATCCATACCCATAGCTATAAAAAACAT
TTCAGGAGAAAATAAGACCGAAGCTGCTCAATTAGGCGCAATTGATTCGTTTCAAAAAT
GTGAAACTTGCCAGCTTACTTCGGCATGTCCTGGTCATTTTGGAAAATTTCATCTTACT
CAACCATTATTTAAAGTCGCATTTAAAAAACTTGTTGAAAATATTTTTAAATATACTTG
TTCTTTCTGTGGTGCTTTACAAAATCTTGAACTTCTGGAATTGATCAAGCAGATAGACG
AACGAAATACTGGAATAACAGTTAAAGATCGTGCTGCTTTTAAAAAAATTTTAGAAGCT
ACCAAACAAAGCAAATTCAAGTGTATTGCACCTAATTGCCAAAAACAAGTCTCTCCTTT
ACAATATTCGAAAAATAATAACTTTATATATAATTCGGGTACTACAAAGGGTATAGTTT
TGGATAACAGGCATGTGTTTAATATCTTACAAAATCTTCCACAAACGTTTAAATTATTG
TTAACCCCTTCGAATGCTCATCAAATCGTATCTCCCGAAAATGTCTTTTATGCTAATAG
TATCTTACTTCCACCACATAATCTACGAACTATCAATGTTTATGATGGTCAGGTTACGA
GTTTGTTAACAAGTGATTTGAATCTGATAATGCGAAGAGTTGCTAATAATGAGACAAAT
GCAAAAATACAAAAAATCTTGGATTCTATCGATAACAGCCGAGGTGCCAATCCATATGC
TACAAATAAAAAGCTTACTTTGGATACTTTGACAGGTGGACACTCAAAAGAATCTTATT
TGCGAAGTTATATTAATGGCAAACGTATTCCTGAGACTGCCAGAGCTGTAATCGAACCC
TCTATGAATAAAACTGGCTTTATTGAAGTACCATCTTACATTTTAAACAAGTTAAGAGA
TGTTGTCTTTTATAATCACGTTACGAAAGATAACATACTCAAAAGTCTTCAAAACGAAC
AAGCTTTTCTAACATATATCAAAAGTGATCATAATTCTGAAAATCCTTATATGGTTTAT
GATTTAGCACAGAAGAATGGATATTTAACCTTGGCTCCTAATTTCGGTGATATTTTCGA
AAAAAGGAAAGAGGAAGGTGGTTTTGTAACTATTTGCAGACATCCATCTATCTGGTTAA
CTAATATCCAATCTGGTATAATAAAAAGATCAGAAGGGTTTACTATTAACATCCCAACC
ACAATTTGCACATCTTTTAATGCTGATTTTGATGGAGATGAGATGACAATATATTCTTT
CAAATCCCCATGTGCCAATCTCGAACAAGCTTTGATTATGAACTCACGAAATCTCTTCA
AAAATTCTATAACAAGCAATCCAATGTTCGGCTTGGTCCAAGATCAAATACCAGCCTTG
AATAAGTTATATAGACGACAAAATTATACATATAACGATGCGTTGGTGATTTTAGGACA
ATTCGGATTTCTGTTAACACCTGGAAAAGATAATTATACCGGAAAAGATATACTTTCTT
GTGTATTCCCAAAACATTATACACTCAAAGGAATTGTTGAAAATGGCGAACTTATTTTG
GAGAATTTTACAAATAAACTCGTTTCCGCAAATTCCTCAAAGTCCATCTTTGGGCATCT
TGTTTTATTTTATGGACAAGAGTATGGTTTGACTATATTGGATACAATGCGAGATATTG
TTCAAAATTTTATTACACATTTTGGTTTCAGTGTAAAAATCCGAGATATGATCCCAAGC
CCAAAAATTTTGGATATTCTAGAAAAGATCGTAGACCAAGAAGTGGATAAAATTGATAA
ACAAACAAAACTTCTATATGACGATATCGAACAAGGTAAGGTTATAATCAACTCTTATG
ATGATATTTCTGAGTTCAGATTAAAAAATGTGGCTATTATGAAAAAGAAACTAGAAAGC
AAACTTTTGGAACTTTTGGATGAATATTATGATGAAGACAATAATTTCCTAGAGATGTA
TAGAACGGGATATAAGGTCAACATTAACGAACTTCTCTCTATTATGTGTTTCTCGGGTT
TTAAAAATTATGGAAATATCGAAATGATTACACCGGGTCTTAATGGTAAAACATCTTTG
TTTAGCTTACCAGATTCTATAAACTTACAAGATTATGGGTTCATCAAAAGCTCTATTGC
CAAAGGGTTAACGTTTGAAGAATATGCTACAATCGTAAAACAAGAAGCTTTTCCACAAA
TTGTTAATGTTACAACTGGTACTTCACAAACAGGATTTTTGGGGAAAAAAATGGTTAAA
ATGGCTTCTGAATTC
```

http://www.insectscience.org/2.10/ref/fig5a.gif

String Searching: DNA alphabet



http://biology.kenyon.edu/courses/biol114/Chap08/longread_sequence.gif

String Searching (Matching) Problems

http://www-igm.univ-mlv.fr/∼lecroq/string/index.html

String matching: Find one, or more generally, all the occurrences of a pattern $x = [x_0 x_1 .. x_{m-1}]$; $x_i \in \Sigma$; $i = 0, \ldots, m-1$, in a text (string) $y = [y_0 y_1 .. y_{n-1}]$; $y_j \in \Sigma$; $j = 0, \ldots, n-1$

- Two basic variants:
  1. Given a pattern, find its occurrences in any initially unknown text
     - Solutions by preprocessing the pattern using finite automata models or combinatorial properties of strings
  2. Given a text, find occurrences of any initially unknown pattern
     - Solutions by indexing the text with the help of trees or finite automata
- In COMPSCI 369: only algorithms of the first kind
- Algorithms of the second kind: look e.g. at Google…

String Matching: Sliding Window Mechanism

- Sliding window: Scan the text by a window of size, which is generally equal to m
- An attempt: Align the left end of the window with the text and compare the characters in the window with those of the pattern
    - Each attempt (step) is associated with position j in the text when the window is positioned on $y_j..y_{j+m-1}$
- Shift the window to the right after the whole match of the pattern or after a mismatch

Effectiveness of the search depends on the order of comparisons:

1. The order is not relevant (e.g. naïve, or brute-force algorithm)
2. The natural left-to-right order (the reading direction)
3. The right-to-left order (the best algorithms in practice)
4. A specific order (the best theoretical bounds)

Single Pattern Algorithms (Summary)

Notation:

m – the length (size) of the pattern; n – the length of the searched text

String search algorithm	**Time complexity for**	
	preprocessing	matching
Naïve	0 (none)	$\Theta(n \cdot m)$
Rabin-Karp	$\Theta(m)$	avg $\Theta(n+m)$ worst $\Theta(n \cdot m)$
Finite state automaton	$\Theta(m\|\Sigma\|)$	$\Theta(n)$
Knuth-Morris-Pratt	$\Theta(m)$	$\Theta(n)$
Boyer-Moore	$\Theta(m+\|\Sigma\|)$	$\Omega(n/m)$, $O(n)$
Bit based (approximate)	$\Theta(m+\|\Sigma\|)$	$\Theta(n)$

See `http://www-igm.univ-mlv.fr/∼lecroq/string` for some animations of these and many other string algorithms

Naïve (Brute-Force) Algorithm

```
for ( j = 0; j <= n - m; j++ ) {
   for ( i = 0; i < m && x[i] == y[i + j]; i++ );
   if ( i >= m ) return j;
}
```

Main features of this easy (but slow) $O(nm)$ algorithm:

- No preprocessing phase
- Only constant extra space needed
- Always shifts the window by exactly 1 position to the right
- Comparisons can be done in any order
- mn expected text characters comparisons

Naïve Algorithm: An Example

Pattern: abaa; *searched string*: ababbaabaaab

```
ababbaabaaab             ............

abaa________  step 1     ABA#        mismatch: 4th letter
_abaa_______  step 2     _#...       mismatch: 1st letter
__abaa______  step 3     __AB#.      mismatch: 3rd letter
___abaa_____  step 4     ___#...     mismatch: 1st letter
____abaa____  step 5     ____#...    mismatch: 1st letter
_____abaa___  step 6     _____A#..   mismatch: 2nd letter
______abaa__  step 7     ______ABAA  success
_______abaa_  step 8     _______#... mismatch: 1st letter
________abaa  step 9     ________.#..mismatch: 2nd letter
```

Runs with 9 window steps and 18 character comparisons

Automaton Based Search

Main features:

- Building the minimal deterministic finite automaton (DFA) accepting strings from the language $L = \Sigma^* x$
  - L is the set of all strings of characters from Σ ending with the pattern x
  - Time complexity $O(m|\Sigma|)$ of this preprocessing ($m = |x|$, i.e. the size of x)
- Time complexity $O(n)$ of the search in a string y of size n if the DFA is stored in a direct access table
- Most suitable for searching within many different strings y for same given pattern x

$$x = x_0x_1x_2x_3 \Rightarrow \left.\begin{array}{l}\epsilon\\ x_0\\ x_0x_1\\ x_0x_1x_2\\ x_0x_1x_2x_3\end{array}\right\} m+1 \text{ DFA states} \longrightarrow x = \text{abaa} \Rightarrow \left.\begin{array}{l}\epsilon \ \ \text{empty}\\ \text{a}\\ \text{ab}\\ \text{aba}\\ \text{abaa}\end{array}\right\}$$

Bulding the Minimal DFA for $L = \Sigma^* x$

- The DFA $(Q, \Sigma, \delta : Q \times \Sigma \rightarrow Q, q_0 \in Q, F \subseteq Q)$ to recognise the language $L = \Sigma^* x$:
    - Q – the set of all the prefixes of $x = x_0 \cdots x_{m-1}$:

$$Q = \{\epsilon,\ x_0,\ x_0 x_1,\ \ldots,\ x_0 \cdots x_{m-2},\ x_0 \cdots x_{m-1}\}$$

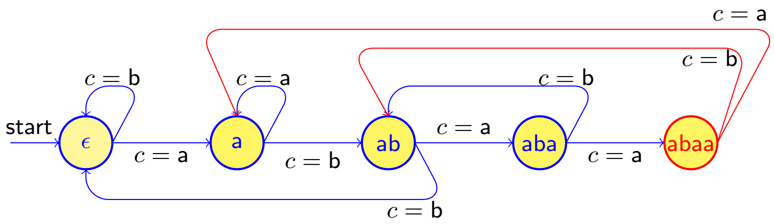
    - $q_0 = \epsilon$ – the state representing the empty prefix
    - $F = \{x\}$ – the state representing the pattern(s) x
    - δ – the state+character to state transition function
        - For $q \in Q$ and $c \in \Sigma$, $\delta(q, c) = qc$ if and only if $qc \in Q$
        - Otherwise $\delta(q, c) = p$ such that p is the longest suffix of qc, which is a prefix of x (i.e. $p \in Q$)
- Once the DFA is built, searching for the word x in a text y consists of parsing y with the DFA beginning with the initial state q_0
- Each time a unique final state F is encountered an occurrence of x is reported

Automaton Search: x = abaa and y = ababbaabaaab

$$\Sigma = \{\text{a}, \text{b}\};\ Q = \{\epsilon, \text{a}, \text{ab}, \text{aba}, \text{abaa}\};\ q_0 = \epsilon;\ F = \{x\} = \{\text{abaa}\}$$

Transitions $\delta(q, c)$:

$c \backslash q$	ϵ	a	ab	aba	abaa
a	a	a	aba	abaa	a
b	ϵ	ab	ϵ	ab	ab



See also: `http://www.ics.uci.edu/~eppstein/161/960222.html`

Automaton Search: $x =$ abaa and $y =$ ababbaabaaab

Automaton:

Initial state ϵ
Final states {abaa}
Transitions

$q_{\text{next}} = \delta(q_{\text{curr}}, c) =$

$q_{\text{curr}} \setminus c$	a	b
ϵ	a	ϵ
a	a	ab
ab	aba	ϵ
aba	abaa	ab
abaa	a	ab

Step	Text	Transition		
1	ababbaabaaab	ϵ	$\longrightarrow$	a
2	ababbaabaaab	a	$\longrightarrow$	ab
3	ababbaabaaab	ab	$\longrightarrow$	aba
4	ababbaabaaab	aba	$\longrightarrow$	ab
5	ababbaabaaab	ab	$\longrightarrow$	ϵ
6	ababbaabaaab	ϵ	$\longrightarrow$	a
7	ababbaabaaab	a	$\longrightarrow$	a
8	ababbaabaaab	a	$\longrightarrow$	ab
9	ababbaabaaab	ab	$\longrightarrow$	aba
10	ababbaabaaab	aba	$\longrightarrow$	abaa
11	ababbaabaaab	abaa	$\longrightarrow$	a
12	ababbaabaaab	a	$\longrightarrow$	ab

Runs with 12 steps and 12 character comparisons

Rabin-Karp Algorithm

Main features:

- Using hashing function
  (i.e., it is more efficient to check whether the window contents "looks like" the pattern than checking exact match)
- Preprocessing phase: time complexity $O(m)$ and constant space
- Searching phase time complexity:
  - $O(mn)$ for worst case
  - $O(n+m)$ for expected case
- Good for multiple patterns x being used

Rabin-Karp Hashing Details

Desirable hashing functions $\mathrm{hash}(\ldots)$ for string matching:

- Efficiency of computation
- High discrimination for strings
- Easy computation of $\mathrm{hash}(y_{j+1}..y_{j+m})$ from the previous window: i.e. $\mathrm{hash}(y_{j+1}..y_{j+m}) = \mathrm{rehash}\left(y_j, y_{j+m}, \mathrm{hash}(y_j..y_{j+m-1})\right)$

For a word w of length m, let $\mathrm{hash}(w)$ be defined as:

$$\mathrm{hash}(w_0..w_{m-1}) = \left(w_0 2^{m-1} + w_1 2^{m-2} + \ldots + w_{m-1} 2^0\right)_{\bmod q}$$

where q is a large number. Then $\mathrm{rehash}(a, b, h) = (2h - a2^m + b)_{\bmod q}$

- Preprocessing phase: computing $\mathrm{hash}(x)$
  - It can be done in constant space and $O(m)$ time
- Searching phase: comparing $\mathrm{hash}(x)$ with $\mathrm{hash}(y_j..y_{j+m-1})$ for $0 \leq j < n - m$
  - If an equality is found, still check the equality $x = y_j..y_{j+m-1}$ character by character

Rabin-Karp Algorithm

Using external hash and rehash functions:

```
int RabinKarp(String x, String y)
{
  m = x.length();
  n = y.length();
  hx = hash(x,0,m-1);
  hy = hash(y,0,m-1);
  for (int j = 0; j <= n - m; j++)
  {
   if (hx==hy && y.substring(j,j+m-1)==x) return j;
   hy = rehash(y[j],y[j+m],hy);
  }
  return -1; // not found
}
```

Rabin-Karp with x = abaa and y = ababbaabaaab

hash(abaa)	=	1459	$\rightarrow$	`hx` — hash value for pattern x
		`hy` $\downarrow$	$\rightarrow$	hash value for substring y
$\mathrm{hash}(y_0..y_3)$	=	1460		ababbaabaaab
$\mathrm{hash}(y_1..y_4)$	=	1466		ababbaabaaab
$\mathrm{hash}(y_2..y_5)$	=	1461		ababbaabaaab
$\mathrm{hash}(y_3..y_6)$	=	1467		ababbaabaaab
$\mathrm{hash}(y_4..y_7)$	=	1464		ababbaabaaab
$\mathrm{hash}(y_5..y_8)$	=	1457		ababbaabaaab
$\mathrm{hash}(y_6..y_9)$	=	1459		ababbaabaaab : `return 6`
$\mathrm{hash}(y_7..y_10)$	=	1463		ababbaabaaab
$\mathrm{hash}(y_8..y_11)$	=	1456		ababbaabaaab

Rabin-Karp Algorithm (searching multiple patterns)

Extending the search for multiple patterns of the *same* length:

```
void RabinKarpMult(String[] x, String y)
{
  m = x[0].length();
  n = y.length();
  for( int i = 0; i < x.length; i++ )
     hx[i] = hash( x[i], 0, m-1);
  hy = hash(y, 0, m-1);
  for( int j = 0; j <= n - m; j ++ )  {
    for( int k = 0; k < x.length; k++ )
      if ( hx[k]==hy && y.substring(j,j+m-1) == x[k] )
                                           matchProcess( k, j );
        hy = rehash( y[j], y[j+m], hy );
    }
}
```

Knuth-Morris-Pratt Algorithm

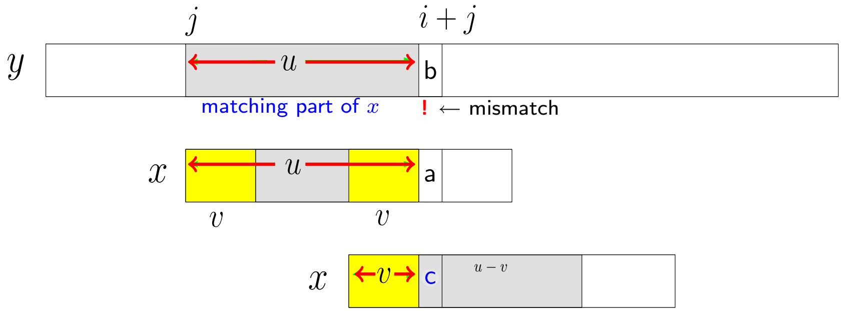
Searches for occurrences of a pattern x within a main text string y by employing the simple observation: *after a mismatch, the word itself allows us to determine where to begin the* ***next match*** *to bypass re-examination of previously matched characters*

- Preprocessing phase: $O(m)$ space and time complexity
- Searching phase: $O(n+m)$ time complexity (independent from the alphabet size)
- At most $2n-1$ character comparisons during the text scan
- The maximum number of comparisons for a single text character: $\leq \log_\eta m$ where $\eta = \frac{1+\sqrt{5}}{2}$ is the golden ratio

The algorithm was invented in 1977 by Knuth and Pratt and independently by Morris, but the three published it jointly

Knuth-Morris-Pratt Window Shift Idea

Let offset i; $0 < i < m$, be the first mismatched position for a pattern x matched to the text string y starting at index position j (i.e. $x_0..x_{i-1} = y_j..y_{j+i-1} = u$, but $x_i =$ a $\neq y_{j+i} =$ b):



The length of the largest substring v being a prefix and suffix of u, which are followed by different characters (like va and vc above), gives the next search index `next[i]`

Knuth-Morris-Pratt Preprocessing

All the shift distances `next[i]` can be actually computed for $0 \leq i \leq m$ in total time $O(m)$ where $m = |x|$

```
void computeNext( String x, int[] next ) {
 int i = 0;
 int j = next[0] = -1;  // end of window marker
 while ( i < x.length() ) {
    while ( j > -1 && x[i] != x[j] ) j = next[ j ];
    i++;
    j++;
    if ( x[ i ] == x[ j ] )
              next[ i ]= next[ j ];
    else next[ i ]= j;
  }
}
```

Knuth-Morris-Pratt Main Algorithm

The main search runs in time $O(n)$ where $n = |y|$.

```
int KMP(String x, String y) {
 int m = x.length(); int n = y.length();
 int[ m+1 ] next;
 computeNext( x, next );
 int i = 0; int j = 0;   // indices in x and y
 while ( j < n ) {
   while ( i > -1 && x[i] != y[j] ) i = next[ i ];
   i++;
   j++;
   if ( i >= m ) return j - i; // Match
 }
 return -1; // Mismatch
}
```

So the total time of the KMP algorithm is $O(m + n)$

KMP with $x =$ abaa and $y =$ ababbaabaaab

Preprocessing phase:

x	a	b	a	a	-
i	0	1	2	3	4
next$[i]$	-1	0	-1	1	1

Searching phase:

```
ababbaabaaab
ABAa           Shift by 2 (next[3]=1)

  .Ba.         Shift by 3 (next[2]=-1)

     Ab..      Shift by 1 (next[1]=0)

      ABAA     Shift by 3 (match found)
```

Boyer-Moore Algorithm

Main features of this “best practical choice” algorithm:

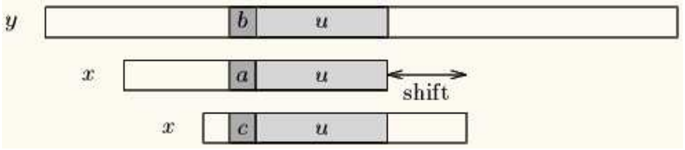
- Performing the comparisons from right to left
- Preprocessing phase: $O(m + |\Sigma|)$ time and space complexity
- Searching phase: $O(m + n)$ time complexity;
- $3n$ text character comparisons in the worst case when searching for a non periodic pattern
- $O(n/m)$ best performance

Two precomputed functions to shift the window to the right:

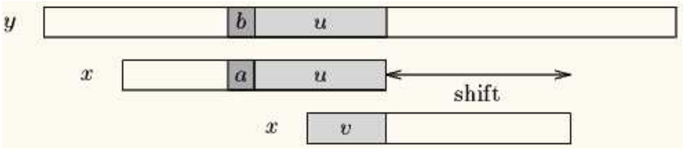
- The good-suffix shift (also called *matching shift*)
- The bad-character shift (also called *occurrence shift*)

Good-Suffix (Matching) Shifts

A mismatch $x_i \neq y_{j+i}$ for a matching attempt at position j, so that $x_{i+1}..x_{m-1} = y_{j+i+1}..y_{j+m-1} = u$
The shift: by aligning the segment u in y with its rightmost occurrence in x that is preceded by a character different from x_i:



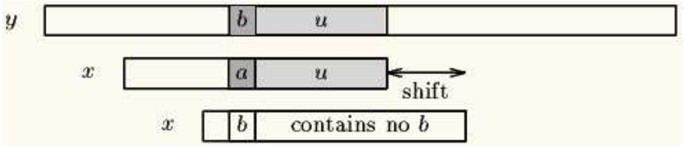
or if no such segment in x exists, by aligning the longest suffix of u in y with a matching prefix v of x:



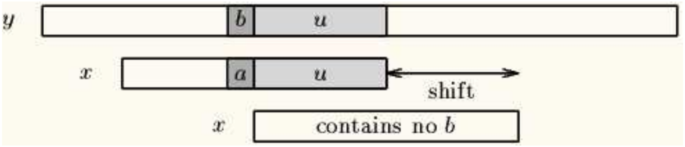
See details in http://www-igm.univ-mlv.fr/~lecroq/string/index.html

Bad-Character (Occurrence) Shifts

A mismatch $x_i \neq y_{j+i}$ for a matching attempt at position j, so that $x_{i+1}..x_{m-1} = y_{j+i+1}..y_{j+m-1} = u$
The shift: by aligning the text character y_{i+j} with its rightmost occurrence in $x_0..x_{m-2}$:



or if y_{j+i} does not occur in x, the left end of the window is aligned with the character y_{j+i+1} immediately after y_{j+i}:



See details in http://www-igm.univ-mlv.fr/~lecroq/string/index.html

Boyer-Moore with $x =$ abaa and $y =$ ababbaabaaab

Bad-character shifts `Bc[a]=1` and `Bc[b]=2`.
Good-suffix shifts `Gs` are 3, 3, 1, and 2, respectively.

```
ababbaabaaab

...a        Shift by 2

  ..aA      Shift by 1

   aBAA     Shift by 3

      ABAA  Shift by 3
```

Shift-AND Matching Algorithm

Also known as the **Baeza-Yates–Gonnet** algorithm and is related to the **Wu-Manber** k-differences algorithm.

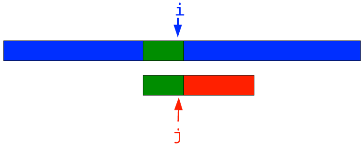
The main features of this bit based algorithm are:

- efficient if the pattern length is no longer than the memory-word size of the machine;
- preprocessing phase in $O(m + |\Sigma|)$ time and space complexity;
- searching phase in $O(n)$ time complexity;
- adapts easily to approximate string matching.

Shift-AND Matching Algorithm (continued)

Algorithm uses (for fixed i) a state vector $\hat{s}$, where

$$s[j] = 1 \text{ iff } y[i-j,\ldots,i] = x[0,\ldots,j]$$



For $c \in \Sigma$ let $T[c]$ be a (Boolean) bit vector of length $m = |x|$ that indicates where c occurs in x.
The next state vector at postions $i+1$ is computed very fast:

$$\hat{s} = ((\hat{s} << 1) + 1)\ \&\ T[y[i+1]]$$

A match is found whenever $s[m-1] = 1$.

Shift-AND with X=abaa and Y=ababbaabaaab

The character vectors for the pattern x are:

x \ $T[]$	a	b
a	1	0
b	0	1
a	1	0
a	1	0

The main search progresses as follows:

x \ $s[]$	a	b	a	b	b	a	a	b	a	a	a	b
a	1	0	1	0	0	1	1	0	1	1	1	0
b	0	1	0	1	0	0	0	1	0	0	0	1
a	0	0	1	0	0	0	0	0	1	0	0	0
a	0	0	0	0	0	0	0	0	0	$\boxed{1}$	0	0

Wu-Manber Approximation Matching Algorithm

The **Shift-AND** algorithm can be modified to detect string matching with at most k errors (or k differences).

The possibilities for matching $x[0,\ldots,j]$ with a substring of y that ends at position i with e errors:

1. *Match:* $x[j] = y[i]$ and a match with e errors between $x[0,\ldots,j-1]$ and a substring of y ending at $i-1$.
2. *Substitution:* a match with $e-1$ errors between $x[0,\ldots,j-1]$ and a substring of y ending at $i-1$.
3. *Insertion:* a match with $e-1$ errors between $x[0,\ldots,j]$ and a substring of y ending at $i-1$.
4. *Deletion:* a match with $e-1$ errors between $x[0,\ldots,j-1]$ and a substring of y ending at i.

Wu-Manber State Update Procedure

We introduce new state vectors $\hat{s_e}$ that represent the matches where $0 \leq e \leq k$ errors have occurred. [Note: $\hat{s} = \hat{s_0}$.]

The state updating is a generalization of the **Shift-AND** rule:

$$\begin{aligned} s'_e \quad = \quad & (((s_e << 1) + 1) \textbf{ AND } T[y[i+1]]) \textbf{ OR} \\ & ((s_{e-1} << 1) + 1) \textbf{ OR} \\ & ((s'_{e-1} << 1) + 1) \textbf{ OR} \\ & s_{e-1} \end{aligned}$$

Here, the s_* and s'_* denote the state at character position i and $i+1$, respectively, of the text string y

The **OR**'s in the above rule account for the 4 possible ways to approximate the pattern with errors