

# Chapter 1

## Pattern Matching and Text Compression Algorithms

**Maxime Crochemore**

*King's College London, UK and Université Paris-Est, France*

**Thierry Lecroq**

*University of Rouen, France*

1.1	Processing Texts Efficiently .....	4
1.2	String-Matching Algorithms .....	5
1.2.1	Karp–Rabin Algorithm .....	5
1.2.2	Knuth–Morris–Pratt Algorithm .....	7
1.2.3	Boyer–Moore Algorithm .....	8
1.2.4	Quick Search Algorithm .....	14
1.2.5	Experimental Results .....	14
1.2.6	Aho–Corasick Algorithm .....	15
1.3	Two-Dimensional Pattern Matching Algorithms .....	20
1.3.1	Zhu–Takaoka Algorithm .....	20
1.3.2	Bird/Baker Algorithm .....	22
1.4	Suffix Trees .....	24
1.4.1	McCreight Algorithm .....	28
1.5	Suffix Arrays .....	31
1.5.1	Kärkkäinen–Sanders Algorithm .....	32
1.5.2	Substring Search .....	36
1.5.3	Longest Common Prefixes .....	36
1.5.4	Substring Search with the Length of the Common Prefixes .....	38
1.6	Alignment .....	39
1.6.1	Global alignment .....	40
1.6.2	Local alignment .....	43
1.6.3	Longest Common Subsequence of Two Strings .....	46
1.6.4	Reducing the Space: Hirschberg Algorithm .....	47
1.7	Approximate String Matching .....	48
1.7.1	Shift-Or Algorithm .....	49
1.7.2	String Matching with $k$ Mismatches .....	51
1.7.3	String Matching with $k$ Differences .....	51
1.7.4	Wu–Manber Algorithm .....	54
1.8	Text Compression .....	55
1.8.1	Huffman Coding .....	56
1.8.1.1	Encoding .....	57
1.8.1.2	Decoding .....	60
1.8.2	Lempel–Ziv–Welsh (LZW) Compression .....	62
1.8.2.1	Compression Method .....	62
1.8.2.2	Decompression Method .....	63

1.8.2.3	Implementation .....	64
1.8.3	Mixing several methods .....	65
1.8.3.1	Run Length Encoding .....	66
1.8.3.2	Move To Front .....	66
1.8.3.3	Integrated example .....	66
1.9	Research Issues and Summary .....	68
	Defining Terms .....	70

## 1.1 Processing Texts Efficiently

The present chapter describes a few standard algorithms used for processing texts. They apply, for example, to the manipulation of texts (text editors), to the storage of textual data (text compression), and to data retrieval systems. The algorithms of this chapter are interesting in different respects. First, they are basic components used in the implementations of practical software. Second, they introduce programming methods that serve as paradigms in other fields of computer science (system or software design). Third, they play an important role in theoretical computer science by providing challenging problems.

Although data are stored in various ways, text remains the main form of exchanging information. This is particularly evident in literature or linguistics where data are composed of huge corpora and dictionaries. This applies as well to computer science where a large amount of data are stored in linear files. And this is also the case in molecular biology where biological molecules can often be approximated as sequences of nucleotides or amino acids. Moreover, the quantity of available data in these fields tends to double every 18 months. This is the reason why algorithms should be efficient even if the speed of computers increases at a steady pace.

Pattern matching is the problem of locating a specific pattern inside raw data. The pattern is usually a collection of strings described in some formal language. Two kinds of textual patterns are presented: single strings and approximated strings. Two algorithms for matching patterns in images that are extensions of string-matching algorithms are also presented.

In several applications, texts need to be structured before being searched. Even if no further information is known about their syntactic structure, it is possible and indeed extremely efficient to build a data structure that supports searches. From among several existing data structures equivalent to represent indexes, suffix trees and suffix arrays are presented, along with their construction algorithms.

The comparison of strings is implicit in the approximate pattern searching problem. Since it is sometimes required to compare just two strings (files or molecular sequences) the basic method based on longest common subsequences is introduced.

Finally, the chapter contains two classical text compression algorithms.

---

```

BF( $x, m, y, n$ )
1  ▷ Searching
2  for  $j \leftarrow 0$  to  $n - m$ 
3      do  $i \leftarrow 0$ 
4          while  $i < m$  and  $x[i] = y[i + j]$ 
5              do  $i \leftarrow i + 1$ 
6          if  $i \geq m$ 
7              then OUTPUT( $j$ )

```

---

FIGURE 1.1: The brute force string-matching algorithm.

Variants of these algorithms are implemented in practical compression software, in which they are often combined together or with other elementary methods. An example of mixing different methods is presented there.

The efficiency of algorithms is evaluated by their running times, and sometimes by the amount of memory space they require at run time as well.

---

## 1.2 String-Matching Algorithms

String matching is the problem of finding one, or more generally, all the **occurrences** of a pattern in a text. The pattern and the text are both strings built over a finite alphabet (a finite set of symbols). Each algorithm of this section outputs all occurrences of the pattern in the text. The pattern is denoted by  $x = x[0..m-1]$ ; its length is equal to  $m$ . The text is denoted by  $y = y[0..n-1]$ ; its length is equal to  $n$ . The alphabet is denoted by  $\Sigma$  and its size is equal to  $\sigma$ .

String-matching algorithms of the present section work as follows: they first align the left ends of the pattern and the text, then compare the aligned symbols of the text and the pattern—this specific work is called an attempt or a scan—and after a whole match of the pattern or after a mismatch they shift the pattern to the right. They repeat the same procedure again until the right end of the pattern goes beyond the right end of the text. This is called the scan and shift mechanism. Each attempt is associated with the position  $j$  in the text, when the pattern is aligned with  $y[j..j+m-1]$ .

The brute force algorithm consists in checking, at all positions in the text between 0 and  $n - m$ , whether an occurrence of the pattern starts there or not. Then, after each attempt, it shifts the pattern exactly one position to the right. This is the simplest algorithm, which is described in Fig. 1.1.

The time complexity of the brute force algorithm is  $O(mn)$  in the worst case but its behavior in practice is often linear on specific data.

---

```

REHASH( $a, b, h$ )
1  return  $((h - a \times d) << 1) + b$ 

```

---

FIGURE 1.2: Function REHASH.

### 1.2.1 Karp–Rabin Algorithm

Hashing provides a simple method for avoiding a quadratic number of symbol comparisons in most practical situations. Instead of checking at each position of the text whether the pattern occurs, it seems to be more efficient to check only if the portion of the text aligned with the pattern “looks like” the pattern. In order to check the resemblance between these portions a hashing function is used. To be helpful for the string-matching problem the hashing function should have the following properties:

- efficiently computable,
- highly discriminating for strings,
- $hash(y[j + 1..j + m])$  must be easily computable from  $hash(y[j..j + m - 1])$ :  
 $hash(y[j + 1..j + m]) = REHASH(y[j], y[j + m], hash(y[j..j + m - 1]))$ .

For a word  $w$  of length  $k$ , its symbols can be considered as digits, and function  $hash(w)$  is defined by

$$hash(w[0..k - 1]) = (w[0] \times 2^{k-1} + w[1] \times 2^{k-2} + \dots + w[k - 1]) \bmod q$$

where  $q$  is a large number. Then, REHASH has a simple expression

$$REHASH(a, b, h) = ((h - a \times d) \times 2 + b) \bmod q$$

where  $d = 2^{k-1}$  where  $q$  is the computer word-size (see Fig. 1.2).

During the search for the pattern  $x$ ,  $hash(x)$  is compared with  $hash(y[j - m + 1..j])$  for  $m - 1 \leq j \leq n - 1$ . If an equality is found, it is still necessary to check the equality  $x = y[j - m + 1..j]$  symbol by symbol.

In the algorithms of Fig. 1.2 and Fig. 1.3 all multiplications by 2 are implemented by shifts (operator  $<<$ ). Furthermore, the computation of the modulus function is avoided by using the implicit modular arithmetic given by the hardware that forgets carries in integer operations. Thus,  $q$  is chosen as the maximum value of an integer of the system.

The worst-case time complexity of the Karp–Rabin algorithm is quadratic (as it is for the brute force algorithm) but its expected running time is  $O(m + n)$ .

**Example 1.1:** Let  $x = \mathbf{ing}$ . Then  $hash(x) = 105 \times 2^2 + 110 \times 2 + 103 = 743$  (symbols are assimilated with their ASCII codes).

---

```

KR( $x, m, y, n$ )
1  ▷ Preprocessing
2   $d \leftarrow 1$ 
3  for  $i \leftarrow 1$  to  $m - 1$ 
4      do  $d \leftarrow d \ll 1$ 
5   $h_x \leftarrow 0$ 
6   $h_y \leftarrow 0$ 
7  for  $i \leftarrow 0$  to  $m - 1$ 
8      do  $h_x \leftarrow (h_x \ll 1) + x[i]$ 
9           $h_y \leftarrow (h_y \ll 1) + y[i]$ 
10 ▷ Searching
11 if  $h_x = h_y$  and  $x = y[0..m - 1]$ 
12     then OUTPUT(0)
13  $j \leftarrow m$ 
14 while  $j < n$ 
15     do  $h_y \leftarrow \text{REHASH}(y[j - m], y[j], h_y)$ 
16         if  $h_x = h_y$  and  $x = y[j - m + 1..j]$ 
17             then OUTPUT( $j - m + 1$ )
18      $j \leftarrow j + 1$ 

```

---

FIGURE 1.3: The Karp–Rabin string-matching algorithm.

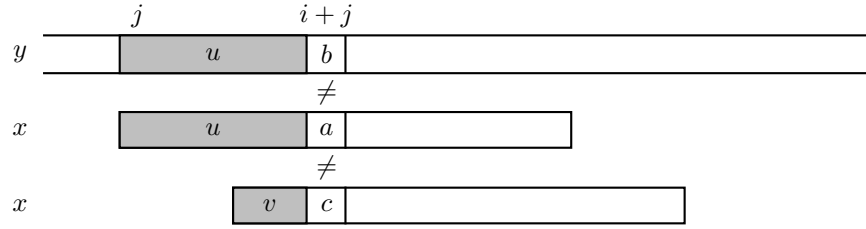
$y$	=	s	t	r	i	n	g		m	a	t	c	h	i	n	g
hash =		806	797	776	743	678	585	443	746	719	766	709	736	743		

### 1.2.2 Knuth–Morris–Pratt Algorithm

This section presents the first discovered linear-time string-matching algorithm. Its design follows a tight analysis of the brute force algorithm, and especially of the way this latter algorithm wastes the information gathered during the scan of the text.

Let us look more closely at the brute force algorithm. It is possible to improve the length of shifts and simultaneously remember some portions of the text that match the pattern. This saves comparisons between characters of the text and of the pattern, and consequently increases the speed of the search.

Consider an attempt at position  $j$ , that is, when the pattern  $x[0..m - 1]$  is aligned with the segment  $y[j..j + m - 1]$  of the text. Assume that the first mismatch (during a left to right scan) occurs between symbols  $x[i]$  and  $y[i + j]$  for  $0 \leq i < m$ . Then,  $x[0..i - 1] = y[j..i + j - 1] = u$  and  $a = x[i] \neq y[i + j] = b$ . When shifting, it is reasonable to expect that a **prefix**  $v$  of the pattern matches some **suffix** of the portion  $u$  of the text. Moreover, if one wants to avoid another immediate mismatch, the letter following the prefix  $v$  in the pattern

FIGURE 1.4: Shift in the Knuth–Morris–Pratt algorithm ( $v$  suffix of  $u$ ).

must be different from  $a$ . (Indeed, it should be expected that  $v$  matches a suffix of  $ub$ , but elaborating along this idea goes beyond the scope of the chapter.) The longest such prefix  $v$  is called the **border** of  $u$  (it occurs at both ends of  $u$ ). This introduces the notation: let  $next[i]$  be the length of the longest (proper) border of  $x[0..i-1]$  followed by a character  $c$  different from  $x[i]$ . Then, after a shift, the comparisons can resume between characters  $x[next[i]]$  and  $y[i+j]$  without missing any occurrence of  $x$  in  $y$  and having to backtrack on the text (see Fig. 1.4).

**Example 1.2:** Here

$y =$	.	.	.	<u>a</u>	<u>b</u>	<u>a</u>	<u>b</u>	<u>a</u>	<u>a</u>	<u>b</u>	.	.	.	.	.
$x =$				<u>a</u>	<u>b</u>	<u>a</u>	<u>b</u>	<u>a</u>	<u>b</u>	<u>a</u>					
$x =$									<u>a</u>	<u>b</u>	<u>a</u>	<u>b</u>	<u>a</u>	<u>b</u>	<u>a</u>

Compared symbols are underlined. Note that the empty string is the suitable border of **ababa**. Other borders of **ababa** are **aba** and **a**.

The Knuth–Morris–Pratt algorithm is displayed in Fig. 1.5. The table  $next$  it uses is computed in  $O(m)$  time before the search phase, applying the same searching algorithm to the pattern itself, as if  $y = x$  (see Fig. 1.6). The worst-case running time of the algorithm is  $O(m+n)$  and it requires  $O(m)$  extra space. These quantities are independent of the size of the underlying alphabet.

### 1.2.3 Boyer–Moore Algorithm

The Boyer–Moore algorithm is considered as the most efficient string-matching algorithm in usual applications. A simplified version of it, or the entire algorithm, is often implemented in text editors for the search and substitute commands.

The algorithm scans the characters of the pattern from right to left beginning with the rightmost symbol. In case of a mismatch (or a complete match of the whole pattern) it uses two precomputed functions to shift the pattern to the right. These two shift functions are called the *bad-character shift* and the *good-suffix shift*. They are based on the following observations.

---

```

KMP( $x, m, y, n$ )
1  ▷ Preprocessing
2   $next \leftarrow \text{PREKMP}(x, m)$ 
3  ▷ Searching
4   $i \leftarrow 0$ 
5   $j \leftarrow 0$ 
6  while  $j < n$ 
7      do while  $i > -1$  and  $x[i] \neq y[j]$ 
8          do  $i \leftarrow next[i]$ 
9           $i \leftarrow i + 1$ 
10          $j \leftarrow j + 1$ 
11         if  $i \geq m$ 
12             then OUTPUT( $j - i$ )
13              $i \leftarrow next[i]$ 

```

---

FIGURE 1.5: The Knuth–Morris–Pratt string-matching algorithm.

---

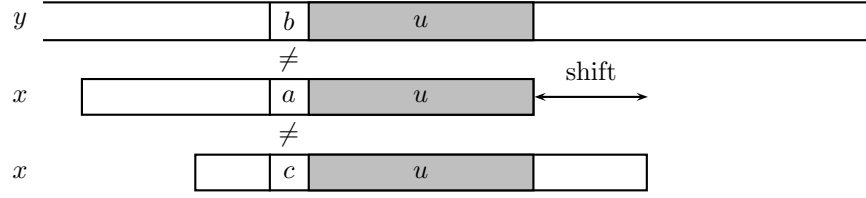
```

PREKMP( $x, m$ )
1   $i \leftarrow -1$ 
2   $j \leftarrow 0$ 
3   $next[0] \leftarrow -1$ 
4  while  $j < m$ 
5      do while  $i > -1$  and  $x[i] \neq x[j]$ 
6          do  $i \leftarrow next[i]$ 
7           $i \leftarrow i + 1$ 
8           $j \leftarrow j + 1$ 
9          if  $x[i] = x[j]$ 
10             then  $next[j] \leftarrow next[i]$ 
11             else  $next[j] \leftarrow i$ 
12  return  $next$ 

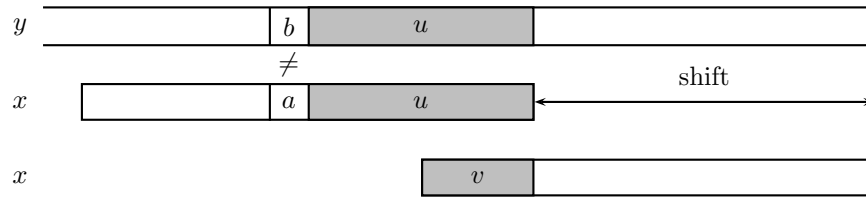
```

---

**FIGURE 1.6:** Preprocessing phase of the Knuth–Morris–Pratt algorithm: computing  $next$ .



**FIGURE 1.7:** The good-suffix shift, when  $u$  reappears preceded by a character different from  $a$ .



**FIGURE 1.8:** The good-suffix shift, when the situation of Fig. 1.7 does not happen. Only a suffix of  $u$  reappears as a prefix of  $x$ .

Assume that a mismatch occurs between the character  $x[i] = a$  of the pattern and the character  $y[i+j] = b$  of the text during an attempt at position  $j$ . Then,  $x[i+1..m-1] = y[i+j+1..j+m-1] = u$  and  $x[i] \neq y[i+j]$ . The good-suffix shift consists in aligning the **segment**  $y[i+j+1..j+m-1]$  with its rightmost occurrence in  $x$  that is preceded by a character different from  $x[i]$  (see Fig. 1.7). If there exists no such segment, the shift consists in aligning the longest suffix  $v$  of  $y[i+j+1..j+m-1]$  with a matching prefix of  $x$  (see Fig. 1.8).

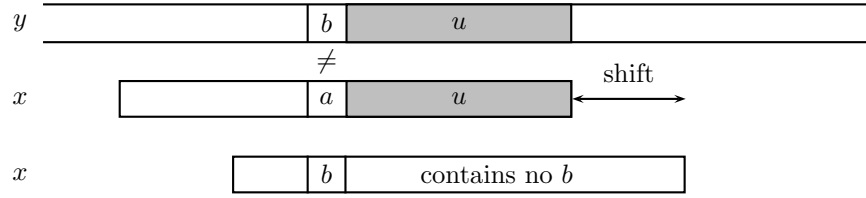
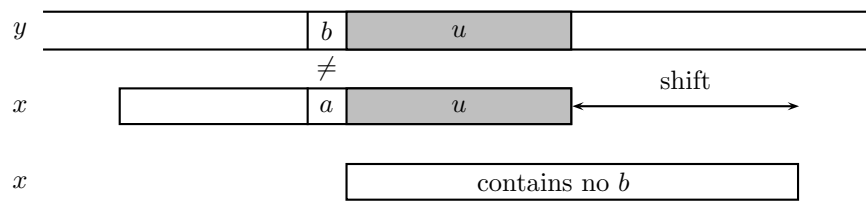
**Example 1.3:** Here

$y =$	.	.	.	a	b	b	a	a	b	b	a	b	b	a	.	.	.
$x =$	a	b	b	a	a	b	<u>b</u>	<u>a</u>	<u>b</u>	<u>b</u>	<u>a</u>						
$x =$				<u>a</u>	<u>b</u>	<u>b</u>	<u>a</u>	<u>a</u>	<u>b</u>	<u>b</u>	<u>a</u>	<u>b</u>	<u>b</u>	<u>a</u>			

The shift is driven by the suffix **abba** of  $x$  found in the text. After the shift, the segment **abba** in the middle of  $y$  matches a segment of  $x$  as in Fig. 1.7. The same mismatch does not recur.

**Example 1.4:** Here

$y =$	.	.	.	a	b	b	a	a	b	b	a	b	b	a	b	b	a	.	.
$x =$				b	b	a	b	<u>b</u>	<u>a</u>	<u>b</u>	<u>b</u>	<u>a</u>							
$x =$								<u>b</u>	<u>b</u>	<u>a</u>	<u>b</u>	<u>b</u>	<u>a</u>	<u>b</u>	<u>b</u>	<u>a</u>			

FIGURE 1.9: The bad-character shift,  $b$  appears in  $x$ .FIGURE 1.10: The bad-character shift,  $b$  does not appear in  $x$  (except possibly at  $m-1$ ).

The segment **abba** found in  $y$  partially matches a prefix of  $x$  after the shift, as in Fig. 1.8.

The bad-character shift consists in aligning the text character  $y[i+j]$  with its rightmost occurrence in  $x[0..m-2]$  (see Fig. 1.9). If  $y[i+j]$  does not appear in the pattern  $x$ , no occurrence of  $x$  in  $y$  can overlap the symbol  $y[i+j]$ , then the left end of the pattern is aligned with the character at position  $i+j+1$  (see Fig. 1.10).

**Example 1.5:** Here

$y =$	.	.	.	.	.	.	a	b	c	d	.	.	.	.		
$x =$	c	d	a	h	g	f	e	b	c	d	.	.	.	.		
$x =$							c	d	a	h	g	f	e	b	c	d

The shift aligns the symbol **a** in  $x$  with the mismatch symbol **a** in the text  $y$  (Fig. 1.9.).

**Example 1.6:** Here

$y =$	.	.	.	.	.	a	b	c	d	.	.	.	.	.	
$x =$	c	d	h	g	f	e	b	c	d	.	.	.	.	.	
$x =$							c	d	h	g	f	e	b	c	d

The shift positions the left end of  $x$  right after the symbol **a** of  $y$  (Fig. 1.10.).

---

```

BM( $x, m, y, n$ )
1  ▷ Preprocessing
2   $gs \leftarrow \text{PREGS}(x, m)$ 
3   $bc \leftarrow \text{PREBC}(x, m)$ 
4  ▷ Preprocessing
5   $j \leftarrow 0$ 
6  while  $j \leq n - m$ 
7      do  $i \leftarrow m - 1$ 
8          while  $i \geq 0$  and  $x[i] = y[i + j]$ 
9              do  $i \leftarrow i - 1$ 
10         if  $i < 0$ 
11             then OUTPUT( $j$ )
12          $j \leftarrow \max\{gs[i + 1], bc[y[i + j] - m + i + 1]\}$ 

```

---

FIGURE 1.11: The Boyer–Moore string-matching algorithm.

---

```

PREBC( $x, m$ )
1  for  $a \leftarrow \text{firstLetter}$  to  $\text{lastLetter}$ 
2      do  $bc[a] \leftarrow m$ 
3  for  $i \leftarrow 0$  to  $m - 2$ 
4      do  $bc[x[i]] \leftarrow m - 1 - i$ 
5  return  $bc$ 

```

---

FIGURE 1.12: Computation of the bad-character shift.

The Boyer–Moore algorithm is shown in Fig. 1.11. For shifting the pattern, it applies the maximum between the bad-character shift and the good-suffix shift. More formally, the two shift functions are defined as follows. The bad-character shift is stored in a table  $bc$  of size  $\sigma$  and the good-suffix shift is stored in a table  $gs$  of size  $m + 1$ . For  $a \in \Sigma$

$$bc[a] = \begin{cases} \min\{i \mid 1 \leq i < m \text{ and } x[m - 1 - i] = a\} & \text{if } a \text{ appears in } x, \\ m & \text{otherwise.} \end{cases}$$

Let us define two conditions,

$$\begin{cases} \text{cond}_1(i, s): & \text{for each } k \text{ such that } i < k < m, s \geq k \text{ or } x[k - s] = x[k], \\ \text{cond}_2(i, s): & \text{if } s < i \text{ then } x[i - s] \neq x[i]. \end{cases}$$

Then, for  $0 \leq i < m$ ,

$$gs[i + 1] = \min\{s > 0 \mid \text{cond}_1(i, s) \text{ and } \text{cond}_2(i, s) \text{ hold}\}$$

and  $gs[0]$  is defined as the length of the smallest period of  $x$ .

To compute the table  $gs$ , a table  $suff$  is used. This table can be defined as

---

```

SUFFIXES( $x, m$ )
1   $\text{uff}[m-1] \leftarrow m$ 
2   $g \leftarrow m-1$ 
3  for  $i \leftarrow m-2$  downto 0
4      do if  $i > g$  and  $\text{uff}[i+m-1-f] \neq i-g$ 
5          then  $\text{uff}[i] \leftarrow \min\{\text{uff}[i+m-1-f], i-g\}$ 
6          else if  $i < g$ 
7              then  $g \leftarrow i$ 
8               $f \leftarrow i$ 
9              while  $g \geq 0$  and  $x[g] = x[g+m-1-f]$ 
10                 do  $g \leftarrow g-1$ 
11                  $\text{uff}[i] \leftarrow f-g$ 
12 return  $\text{uff}$ 

```

---

FIGURE 1.13: Computation of the table  $\text{uff}$ .

---

```

PREGS( $x, m$ )
1   $gs \leftarrow \text{SUFFIXES}(x, m)$ 
2  for  $i \leftarrow 0$  to  $m-1$ 
3      do  $gs[i] \leftarrow m$ 
4   $j \leftarrow 0$ 
5  for  $i \leftarrow m-1$  downto  $-1$ 
6      do if  $i = -1$  or  $\text{uff}[i] = i+1$ 
7          then while  $j < m-1-i$ 
8              do if  $gs[j] = m$ 
9                  then  $gs[j] \leftarrow m-1-i$ 
10                  $j \leftarrow j+1$ 
11  for  $i \leftarrow 0$  to  $m-2$ 
12      do  $gs[m-1-\text{uff}[i]] \leftarrow m-1-i$ 
13 return  $gs$ 

```

---

FIGURE 1.14: Computation of the good-suffix shift.

follows: for  $i = 0, 1, \dots, m - 1$ ,

$$\text{suff}[i] = \text{longest common suffix between } x[0..i] \text{ and } x.$$

It is computed in linear time and space by the function SUFFIXES (see Fig. 1.13).

Tables  $bc$  and  $gs$  can be precomputed in time  $O(m + \sigma)$  before the search phase and require an extra space in  $O(m + \sigma)$  (see Fig. 1.12 and Fig. 1.14). The worst-case running time of the algorithm is quadratic. However, on large alphabets (relative to the length of the pattern) the algorithm is extremely fast. Slight modifications of the strategy yield linear-time algorithms (see the bibliographic notes). When searching for  $a^m$  in  $(a^{m-1}b)^{\lfloor n/m \rfloor}$  the algorithm makes only  $O(n/m)$  comparisons, which is the absolute minimum for any string-matching algorithm in the model where the pattern only is preprocessed.

### 1.2.4 Quick Search Algorithm

The bad-character shift used in the Boyer–Moore algorithm is not very efficient for small alphabets, but when the alphabet is large compared with the length of the pattern, as it is often the case with the ASCII table and ordinary searches made under a text editor, it becomes very useful. Using it alone produces a practically very efficient algorithm that is described now.

After an attempt where  $x$  is aligned with  $y[j..j + m - 1]$ , the length of the shift is at least equal to one. Thus, the character  $y[j + m]$  is necessarily involved in the next attempt, and thus can be used for the bad-character shift of the current attempt. In the present algorithm, the bad-character shift is slightly modified to take into account the observation as follows ( $a \in \Sigma$ ):

$$bc[a] = 1 + \begin{cases} \min\{i \mid 0 \leq i < m \text{ and } x[m - 1 - i] = a\} & \text{if } a \text{ appears in } x, \\ m & \text{otherwise.} \end{cases}$$

Indeed, the comparisons between text and pattern characters during each attempt can be done in any order. The algorithm of Fig. 1.15 performs the comparisons from left to right. It is called Quick Search after its inventor and has a quadratic worst-case time complexity but good practical behavior.

**Example 1.7:** Here

$y =$	s	t	r	i	n	g	-	m	a	t	c	h	i	n	g
$x =$	i	n	g												
$x =$				i	<u>n</u>	<u>g</u>									
$x =$							i	n	g						
$x =$										i	n	g			
$x =$											i	<u>n</u>	<u>g</u>		

The Quick Search algorithm makes only nine comparisons to find the two occurrences of **ing** inside the text of length 15.

---

```

QS( $x, m, y, n$ )
1  ▷ Preprocessing
2  for  $a \leftarrow \text{firstLetter}$  to  $\text{lastLetter}$ 
3      do  $bc[a] \leftarrow m + 1$ 
4  for  $i \leftarrow 0$  to  $m - 1$ 
5      do  $bc[x[i]] \leftarrow m - i$ 
6  ▷ Searching
7   $j \leftarrow 0$ 
8  while  $j \leq n - m$ 
9      do  $i \leftarrow 0$ 
10         while  $i \geq 0$  and  $x[i] = y[i + j]$ 
11             do  $i \leftarrow i + 1$ 
12         if  $i \geq m$ 
13             then OUTPUT( $j$ )
14          $j \leftarrow bc[y[j + m]]$ 

```

---

FIGURE 1.15: The Quick Search string-matching algorithm.

### 1.2.5 Experimental Results

Fig. 1.16 and Fig. 1.17 present the running times of three string-matching algorithms: the Boyer–Moore algorithm (BM), the Quick Search algorithm (QS), and the Backward Oracle Matching algorithm (BOM). The Backward Oracle Matching algorithm can be viewed as a variation of the Boyer–Moore algorithm where factors (segments) rather than suffixes of the pattern are recognized. The BOM algorithm uses a data structure to store all the factors of the reversed pattern: a factor oracle but a suffix automaton or a **suffix tree** (see section 1.4) can also be used.

Tests have been performed on various types of texts. Fig. 1.16 shows the results when the text is a DNA sequence on the four-letter alphabet of nucleotides **A**, **C**, **G**, **T**. In Fig. 1.17 English text is considered.

For each pattern length, a large number of searches with random patterns were run. The average time according to the length is shown in the two figures. The running times of both preprocessing and searching phases are added. The three algorithms are implemented in a homogeneous way in order to keep the comparison significant.

For the genome, as expected, the QS algorithm is the best for short patterns. But for long patterns it is eventually less efficient than the BM algorithm. In this latter case, the BOM algorithm achieves the best results. For rather large alphabets the three algorithms have similar behaviors; however, the QS is always faster.

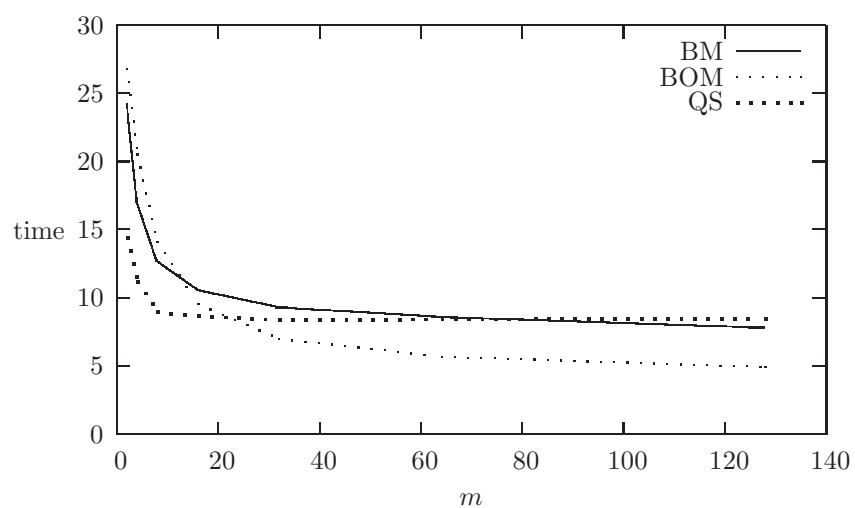


FIGURE 1.16: Running times for a DNA sequence.

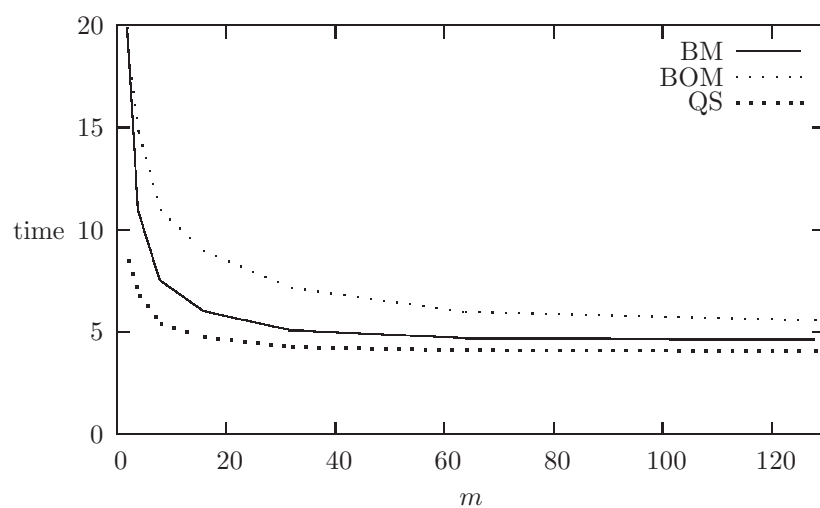


FIGURE 1.17: Running times for an English text.

---

```

PREAC( $X, k$ )
1  Create a new node  $root$ 
2  ▷ creates a loop on the root of the trie
3  for  $a \in \Sigma$ 
4      do  $child(root, a) \leftarrow root$ 
5  ▷ enters each pattern in the trie
6  for  $i \leftarrow 0$  to  $k - 1$ 
7      do ENTER( $X[i], root$ )
8  ▷ completes the trie with failure links
9  COMPLETE( $root$ )
10 return  $root$ 

```

---

FIGURE 1.18: Preprocessing phase of the Aho–Corasick algorithm.

### 1.2.6 Aho–Corasick Algorithm

The UNIX operating system provides standard text (or file) facilities. Among them is the series of **grep** commands that locate patterns in files. The algorithm underlying the **fgrep** command of UNIX is described in this section. It searches files for a finite set of strings, and can, for instance, output lines containing at least one of the strings.

If one is interested in searching for all occurrences of all patterns taken from a finite set of patterns, a first solution consists in repeating some string-matching algorithm for each pattern. If the set contains  $k$  patterns, this search runs in time  $O(kn)$ . The solution described in the present section and designed by Aho and Corasick runs in time  $O(n \log \sigma)$ . The algorithm is a direct extension of the Knuth–Morris–Pratt algorithm, and the running time is independent of the number of patterns.

Let  $X = \{x_0, x_1, \dots, x_{k-1}\}$  be the set of patterns, and let  $|X| = |x_0| + |x_1| + \dots + |x_{k-1}|$  be the total size of the set  $X$ . The Aho–Corasick algorithm first builds a **trie**  $T(X)$ , a digital tree recognizing the patterns of  $X$ . The trie  $T(X)$  is a tree in which edges are labeled by letters and in which branches spell the patterns of  $X$ . A node  $p$  in the trie  $T(X)$  is identified with the unique word  $w$  spelled by the path of  $T(X)$  from its root to  $p$ . The root itself is identified with the empty word  $\varepsilon$ . Notice that if  $w$  is a node in  $T(X)$  then  $w$  is a prefix of some  $x_i \in X$ . If  $w$  is a node in  $T(X)$  and  $a \in \Sigma$  then  $child(w, a)$  is equal to  $wa$  if  $wa$  is a node in  $T(X)$ ; it is equal to UNDEFINED otherwise.

The function PREAC in Fig. 1.18 returns the trie of all patterns. During the second phase, where patterns are entered in the trie, the algorithm initializes an output function *out*. It associates the singleton  $\{x_i\}$  with the nodes  $x_i$  ( $0 \leq i < k$ ), and associates the empty set with all other nodes of  $T(X)$  (see Fig. 1.19).

Finally, the last phase of function PREAC (Fig. 1.18) consists in building the failure link of each node of the trie, and simultaneously completing the output function. This is done by the function COMPLETE in Fig. 1.20. The

---

```

ENTER( $x, root$ )
1   $r \leftarrow root$ 
2   $i \leftarrow 0$ 
3  ▷ follows the existing edges
4  while  $i < |x|$  and  $child(r, x[i]) \neq \text{UNDEFINED}$  and  $child(r, x[i]) \neq root$ 
5      do  $r \leftarrow child(r, x[i])$ 
6       $i \leftarrow i + 1$ 
7  ▷ creates new edges
8  while  $i < |x|$ 
9      do Create a new node  $s$ 
10      $child(r, x[i]) \leftarrow s$ 
11      $r \leftarrow s$ 
12      $i \leftarrow i + 1$ 
13   $out(r) \leftarrow \{x\}$ 

```

---

FIGURE 1.19: Construction of the trie.

---

```

COMPLETE( $root$ )
1   $q \leftarrow \text{empty queue}$ 
2   $\ell \leftarrow \text{list of the edges } (root, a, p) \text{ for any character } a \in \Sigma$ 
    $\text{and any node } p \neq root$ 
3  while the list  $\ell$  is not empty
4      do  $(r, a, p) \leftarrow \text{FIRST}(\ell)$ 
5       $\ell \leftarrow \text{NEXT}(\ell)$ 
6       $\text{ENQUEUE}(q, p)$ 
7       $fail(p) \leftarrow root$ 
8  while the queue  $q$  is not empty
9      do  $r \leftarrow \text{DEQUEUE}(q)$ 
10      $\ell \leftarrow \text{list of the edges } (root, a, p) \text{ for any character } a \in \Sigma$ 
        $\text{and any node } p$ 
11     while the list  $\ell$  is not empty
12         do  $(r, a, p) \leftarrow \text{FIRST}(\ell)$ 
13          $\ell \leftarrow \text{NEXT}(\ell)$ 
14          $\text{ENQUEUE}(q, p)$ 
15          $s \leftarrow fail(r)$ 
16         while  $child(s, a) = \text{UNDEFINED}$ 
17             do  $s \leftarrow fail(s)$ 
18          $fail(p) \leftarrow child(s, a)$ 
19          $out(p) \leftarrow out(p) \cup out(child(s, a))$ 

```

---

FIGURE 1.20: Completion of the output function and construction of failure links.

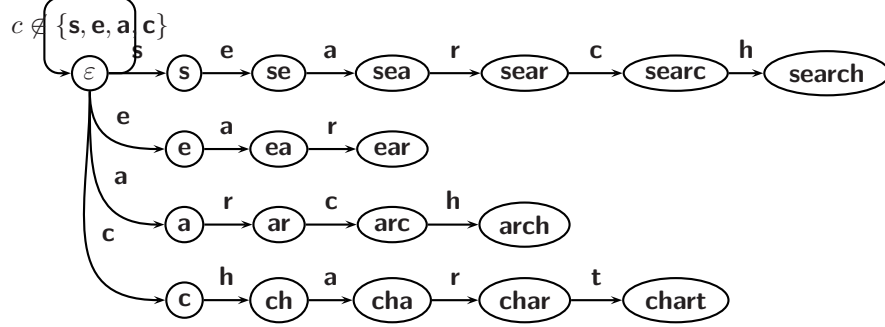
failure function  $fail$  is defined on nodes as follows ( $w$  is a node):

$fail(w) = u$  where  $u$  is the longest proper suffix of  $w$  that belongs to  $T(X)$ .

Computation of failure links is done during a breadth-first traversal of  $T(X)$ . Completion of the output function is done while computing the failure function  $fail$  using the following rule:

if  $fail(w) = u$  then  $out(w) = out(w) \cup out(u)$ .

**Example 1.8:** Here  $X = \{\text{search, ear, arch, chart}\}$



nodes	$\varepsilon$	s	se	sea	sear	searc	search	e	ea	ear
$fail$	$\varepsilon$	$\varepsilon$	e	ea	ear	arc	arch	$\varepsilon$	a	ar
nodes	a	ar	arc	arch	c	ch	cha	char	chart	
$fail$	$\varepsilon$	$\varepsilon$	c	ch	$\varepsilon$	$\varepsilon$	a	ar	$\varepsilon$	

nodes	sear	search	ear	arch	chart
$out$	ear	{search, arch}	ear	arch	chart

To stop going back with failure links during the computation of the failure links, and also to overpass text characters for which no transition is defined from the root, a loop is added on the root of the trie for these symbols. This is done at the first phase of function PREAC.

After the preprocessing phase is completed, the searching phase consists in parsing all the characters of the text  $y$  with  $T(X)$ . This starts at the root of  $T(X)$  and uses failure links whenever a character in  $y$  does not match any label of outgoing edges of the current node. Each time a node with a nonempty output is encountered, this means that the patterns of the output have been discovered in the text, ending at the current position. Then, the position is output.

An implementation of the Aho–Corasick algorithm from the previous discussion is shown in Fig. 1.21. Note that the algorithm processes the text in an on-line way, so that the buffer on the text can be limited to only one symbol. Also note that the instruction  $r \leftarrow fail(r)$  in Fig. 1.21 is the exact analogue

---

```

AC( $X, k, y, n$ )
1  ▷ Preprocessing
2   $r \leftarrow \text{PREAC}(X, k)$ 
3  ▷ Searching
4  for  $j \leftarrow 0$  to  $n - 1$ 
5      do while  $\text{child}(r, y[j]) = \text{UNDEFINED}$ 
6          do  $r \leftarrow \text{fail}(r)$ 
7           $r \leftarrow \text{child}(r, y[j])$ 
8          if  $\text{out}(r) \neq \emptyset$ 
9              then  $\text{OUTPUT}((\text{out}(r), j))$ 

```

---

FIGURE 1.21: The complete Aho–Corasick algorithm.

of instruction  $i \leftarrow \text{next}[i]$  in Fig. 1.5. A unified view of both algorithms exists but is beyond the scope of the chapter.

The entire algorithm runs in time  $O(|X| + n)$  if the *child* function is implemented to run in constant time. This is the case for any fixed alphabet. Otherwise a  $\log \sigma$  multiplicative factor comes from access to the children nodes.

---

### 1.3 Two-Dimensional Pattern Matching Algorithms

In this section only two-dimensional arrays are considered. Arrays may be thought of as bit map representations of images, where each cell of arrays contains the codeword of a pixel. The string-matching problem finds an equivalent formulation in two dimensions (and even in any number of dimensions), and algorithms of section 1.2 can be extended to operate on arrays.

The problem now is to locate all occurrences of a two-dimensional pattern  $X = X[0..m_1 - 1, 0..m_2 - 1]$  of size  $m_1 \times m_2$  inside a two-dimensional text  $Y = Y[0..n_1 - 1, 0..n_2 - 1]$  of size  $n_1 \times n_2$ . The brute force algorithm for this problem is given in Fig. 1.22. It consists in checking at all positions of  $Y[0..n_1 - m_1, 0..n_2 - m_2]$  if the pattern occurs. This algorithm has a quadratic (with respect to the size of the problem) worst-case time complexity in  $O(m_1 m_2 n_1 n_2)$ . In the next sections two more efficient algorithms are presented. The first one is an extension of the Karp–Rabin algorithm (previous section). The second one solves the problem in linear time on a fixed alphabet; it uses both the Aho–Corasick and the Knuth–Morris–Pratt algorithms.

#### 1.3.1 Zhu–Takaoka Algorithm

As for one-dimensional string matching, it is possible to check if the pattern occurs in the text only if the *aligned* portion of the text looks like the pattern.

---

```

BF2D( $X, m_1, m_2, Y, n_1, n_2$ )
1  ▷ Searching
2  for  $j_1 \leftarrow 0$  to  $n_1 - m_1$ 
3      do for  $j_2 \leftarrow 0$  to  $n_2 - m_2$ 
4          do  $i \leftarrow 0$ 
5              while  $i < m_1$ 
6                  and  $x[i, 0 \dots m_2 - 1] = y[j_1 + i, j_2 \dots j_2 + m_2 - 1]$ 
7                      do  $i \leftarrow i + 1$ 
8                  if  $i \geq m_1$ 
9                      then OUTPUT( $j_1, j_2$ )

```

---

**FIGURE 1.22:** The brute force two-dimensional pattern matching algorithm.

To do that, the idea is to use vertically the hash function method proposed by Karp and Rabin. To initialize the process, the two-dimensional arrays  $X$  and  $Y$  are translated into one-dimensional arrays of numbers  $x$  and  $y$ . The translation from  $X$  to  $x$  is done as follows ( $0 \leq i < m_2$ ):

$$x[i] = \text{hash}(X[0, i]X[1, i] \cdots X[m_1 - 1, i])$$

and the translation from  $Y$  to  $y$  is done by ( $0 \leq i < m_2$ ):

$$y[i] = \text{hash}(Y[0, i]Y[1, i] \cdots Y[m_1 - 1, i]).$$

The fingerprint  $y$  helps to find occurrences of  $X$  starting at row  $j = 0$  in  $Y$ . It is then updated for each new row in the following way ( $0 \leq i < m_2$ ):

$$\begin{aligned} & \text{hash}(Y[j + 1, i]Y[j + 2, i] \cdots Y[j + m_1, i]) \\ &= \text{REHASH}(Y[j, i], Y[j + m_1, i], \text{hash}(Y[j, i]Y[j + 1, i] \cdots Y[j + m_1 - 1, i])) \end{aligned}$$

(functions *hash* and *REHASH* are described in the section on the Karp–Rabin algorithm).

**Example 1.9:**

$X =$ 

a	a	a
b	b	a
a	a	b

$Y =$ 

a	b	a	b	a	b	b
a	a	a	a	b	b	b
b	b	b	a	a	a	b
a	a	a	b	b	a	a
b	b	a	a	a	b	b
a	a	b	a	b	a	a

$x =$ 

681	681	680
-----	-----	-----

$y =$ 

680	684	680	683	681	685	686
-----	-----	-----	-----	-----	-----	-----

Next value of  $y$  is 

681	681	681	680	684	683	685
-----	-----	-----	-----	-----	-----	-----

. The occurrence of  $x$  at position 1 on  $y$  corresponds to an occurrence of  $X$  at position (1, 1) on  $Y$ .

---

```

KMP-IN-LINE( $X, m_1, m_2, Y, n_1, n_2, x, y, next, j_1$ )
1   $i_2 \leftarrow 0$ 
2   $j_2 \leftarrow 0$ 
3  while  $j_2 < n_2$ 
4      do while  $i_2 > -1$  and  $x[i_2] \neq y[j_2]$ 
5          do  $i_2 \leftarrow next[i_2]$ 
6           $i_2 \leftarrow i_2 + 1$ 
7           $j_2 \leftarrow j_2 + 1$ 
8          if  $i_2 \geq m_2$ 
9              then DIRECT-COMPARE( $X, m_1, m_2, Y, n_1, n_2, j_1, j_2 - 1$ )
10              $i_2 \leftarrow next[m_2]$ 

```

---

FIGURE 1.23: Search for  $x$  in  $y$  using KMP algorithm.

---

```

DIRECT-COMPARE( $X, m_1, m_2, Y, row, column$ )
1   $j_1 \leftarrow row - m_1 + 1$ 
2   $j_2 \leftarrow column - m_2 + 1$ 
3  for  $i_1 \leftarrow 0$  to  $m_1 - 1$ 
4      do for  $i_2 \leftarrow 0$  to  $m_2 - 1$ 
5          do if  $X[i_1, i_2] \neq Y[i_1 + j_1, i_2 + j_2]$ 
6              then return
7  OUTPUT( $j_1, j_2$ )

```

---

FIGURE 1.24: Naive check of an occurrence of  $x$  in  $y$  at position ( $row$ ,  $column$ ).

Since the alphabet of  $x$  and  $y$  is large, searching for  $x$  in  $y$  must be done by a string-matching algorithm for which the running time is independent of the size of the alphabet: the Knuth–Morris–Pratt suits this application perfectly. Its adaptation is shown in Fig. 1.23.

When an occurrence of  $x$  is found in  $y$ , then one still has to check if an occurrence of  $X$  starts in  $Y$  at the corresponding position. This is done naively by the procedure of Fig. 1.24.

The Zhu–Takaoka algorithm as explained above is displayed in Fig. 1.25. The search for the pattern is performed row by row starting at row 0 and ending at row  $n_1 - m_1$ .

### 1.3.2 Bird/Baker Algorithm

The algorithm designed independently by Bird and Baker for the two-dimensional pattern matching problem combines the use of the Aho–Corasick algorithm and the Knuth–Morris–Pratt (KMP) algorithm. The pattern  $X$  is divided into its  $m_1$  rows  $R_0 = X[0, 0 \dots m_2 - 1]$  to  $R_{m_1-1} = x[m_1 - 1, 0 \dots m_2 - 1]$ . The rows are preprocessed into a trie as in the Aho–Corasick algorithm described earlier.

---

```

ZT( $X, m_1, m_2, Y, n_1, n_2$ )
1  ▷ Preprocessing
2  ▷ Computes  $x$ 
3  for  $i_2 \leftarrow 0$  to  $m_2 - 1$ 
4      do  $x[i_2] \leftarrow 0$ 
5      for  $i_1 \leftarrow 0$  to  $m_1 - 1$ 
6          do  $x[i_2] \leftarrow (x[i_2] << 1) + X[i_1, i_2]$ 
7  ▷ Computes the first value of  $y$ 
8  for  $j_2 \leftarrow 0$  to  $n_2 - 1$ 
9      do  $y[j_2] \leftarrow 0$ 
10     for  $j_1 \leftarrow 0$  to  $m_1 - 1$ 
11         do  $y[j_2] \leftarrow (y[j_2] << 1) + Y[j_1, j_2]$ 
12   $d \leftarrow 1$ 
13  for  $i \leftarrow 1$  to  $m_1 - 1$ 
14      do  $d \leftarrow d << 1$ 
15   $next \leftarrow \text{PREKMP}(X', m_2)$ 
16  ▷ Searching
17   $j_1 \leftarrow m_1 - 1$ 
18  while  $j_1 < n_1$ 
19      do  $\text{KMP-IN-LINE}(X, m_1, m_2, Y, n_1, n_2, x, y, next, j_2)$ 
20          if  $j_1 < n_1 - 1$ 
21              then for  $j_2 \leftarrow 0$  to  $n_2 - 1$ 
22                  do  $y[j_2] \leftarrow \text{REHASH}(Y[j_1 - m_1 + 1, j_2],$ 
23                       $Y[j_1 + 1, j_2], y[j_2])$ 
24           $j_1 \leftarrow j_1 + 1$ 

```

---

**FIGURE 1.25:** The Zhu-Takaoka two-dimensional pattern matching algorithm.

---

```

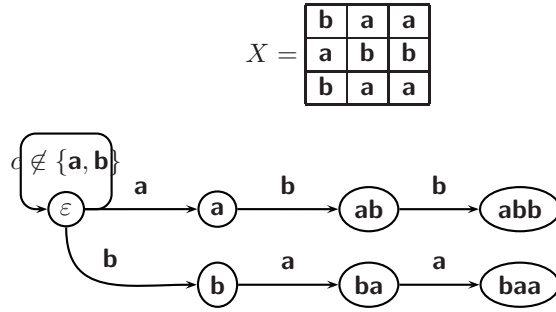
PRE-KMP-FOR-B( $X, m_1, m_2$ )
1   $i \leftarrow 0$ 
2   $next[0] \leftarrow -1$ 
3   $j \leftarrow -1$ 
4  while  $i < m_1$ 
5      do while  $j > -1$  and  $X[i, 0..m_2 - 1] \neq X[j, 0..m_2 - 1]$ 
6          do  $j \leftarrow next[j]$ 
7           $i \leftarrow i + 1$ 
8           $j \leftarrow j + 1$ 
9          if  $X[i, 0..m_2 - 1] \neq X[j, 0..m_2 - 1]$ 
10             then  $next[i] \leftarrow next[j]$ 
11             else  $next[i] \leftarrow j$ 
12 return  $next$ 

```

---

FIGURE 1.26: Computes the function  $next$  for rows of  $X$ .

**Example 1.10:** Pattern  $X$  and the trie of its rows:



The search proceeds as follows. The text is read from the upper left corner to the bottom right corner, row by row. When reading the character  $Y[j_1, j_2]$  the algorithm checks whether the portion  $Y[j_1, j_2 - m_2 + 1..j_2] = R$  matches any of  $R_0, \dots, R_{m_1-1}$  using the Aho–Corasick machine. An additional one-dimensional array  $a$  of size  $n_1$  is used as follows:  $a[j_2] = k$  means that the  $k - 1$  first rows  $R_0, \dots, R_{k-2}$  of the pattern match, respectively, the portions of the text:  $Y[j_1 - k + 1, j_2 - m_2 + 1..j_2], \dots, Y[j_1 - 1, j_2 - m_2 + 1..j_2]$ . Then, if  $R = R_{k-1}$ ,  $a[j_2]$  is incremented to  $k + 1$ . If not,  $a[j_2]$  is set to  $s + 1$  where  $s$  is the maximum  $i$  such that

$$R_0 \cdots R_i = R_{k-s+1} \cdots R_{k-2} R.$$

The value  $s$  is computed using the KMP algorithm vertically (in columns). If there exists no such  $s$ ,  $a[j_2]$  is set to 0. Finally, if at some point  $a[j_2] = m_1$  an occurrence of the pattern appears at position  $(j_1 - m_1 + 1, j_2 - m_2 + 1)$  in the text.

The Bird/Baker algorithm is presented in Figs. 1.26 and 1.27. It runs in time  $O((n_1 n_2 + m_1 m_2) \log \sigma)$ .

---

```

B( $X, m_1, m_2, Y, n_1, n_2$ )
1  ▷ Preprocessing
2  for  $i \leftarrow 0$  to  $m_2 - 1$ 
3      do  $a[i] \leftarrow 0$ 
4   $root \leftarrow \text{PREAC}(m_1)$ 
5   $next \leftarrow \text{PRE-KMP-FOR-B}(X, m_1, m_2)$ 
6  for  $j_1 \leftarrow 0$  to  $n_1 - 1$ 
7      do  $r \leftarrow root$ 
8          for  $j_2 \leftarrow 0$  to  $n_2 - 1$ 
9              do while  $child(r, Y[j_1, j_2]) = \text{UNDEFINED}$ 
10                 do  $r \leftarrow fail(r)$ 
11                  $r \leftarrow child(r, Y[j_1, j_2])$ 
12                 if  $out(r) \neq \emptyset$ 
13                     then  $k \leftarrow a[j_2]$ 
14                     while  $k > 0$  and  $X[k, 0..m_2 - 1] = out(r)$ 
15                         do  $k \leftarrow next[k]$ 
16                      $a[j_2] \leftarrow k + 1$ 
17                     if  $k \geq m_1 - 1$ 
18                         then  $\text{OUTPUT}(j_1 - m_1 + 1, j_2 - m_2 + 1)$ 
19                     else  $a[j_2] \leftarrow 0$ 

```

---

**FIGURE 1.27:** The Bird/Baker two-dimensional pattern matching algorithm.

---

## 1.4 Suffix Trees

The suffix tree  $S(y)$  of a string  $y$  is a trie (as described earlier) containing all the suffixes of the string, and having the properties described subsequently. This data structure serves as an index on the string: it provides a direct access to all segments of the string, and gives the positions of all their occurrences in the string.

Once the suffix tree of a text  $y$  is built, searching for  $x$  in  $y$  remains to spell  $x$  along a branch of the tree. If this walk is successful the positions of the pattern can be output. Otherwise,  $x$  does not occur in  $y$ .

Any kind of trie that represents the suffixes of a string can be used to search it. But the suffix tree has additional features which imply that its size is linear. The suffix tree of  $y$  is defined by the following properties:

- All branches of  $S(y)$  are labeled by all suffixes of  $y$ .
- Edges of  $S(y)$  are labeled by strings.
- Internal nodes of  $S(y)$  have at least two children (when  $y$  is not empty).
- Edges outgoing an internal node are labeled by segments starting with different letters.

---

```

SUFFIX-TREE( $y, n$ )
1   $T_{-1} \leftarrow$  one node tree
2  for  $j \leftarrow 0$  to  $n - 1$ 
3      do  $T_j \leftarrow$  INSERT( $T_{j-1}, y[j \dots n - 1]$ )
4  return  $T_{n-1}$ 

```

---

FIGURE 1.28: Construction of a suffix tree for  $y$ .

---

```

INSERT( $T_{j-1}, y[j \dots n - 1]$ )
1  locate the node  $h$  associated with  $head_j$  in  $T_{j-1}$ , possibly breaking an edge
2  add a new edge labeled  $tail_j$  from  $h$  to a new leaf representing  $y[j \dots n - 1]$ 
3  return the modified tree

```

---

FIGURE 1.29: Insertion of a new suffix in the tree.

- The preceding segments are represented by their starting positions on  $y$  and their lengths.

Moreover, it is assumed that  $y$  ends with a symbol occurring nowhere else in it (the dollar sign is used in examples). This avoids marking nodes, and implies that  $S(y)$  has exactly  $n$  leaves (number of nonempty suffixes). The other properties then imply that the total size of  $S(y)$  is  $O(n)$ , which makes it possible to design a linear-time construction of the trie. The algorithm described in the present section has this time complexity provided the alphabet is fixed, or with an additional multiplicative factor  $\log \sigma$  otherwise.

The algorithm inserts all nonempty suffixes of  $y$  in the data structure from the longest to the shortest suffix, as shown in Fig. 1.28. Two definitions to explain how the algorithm works are introduced:

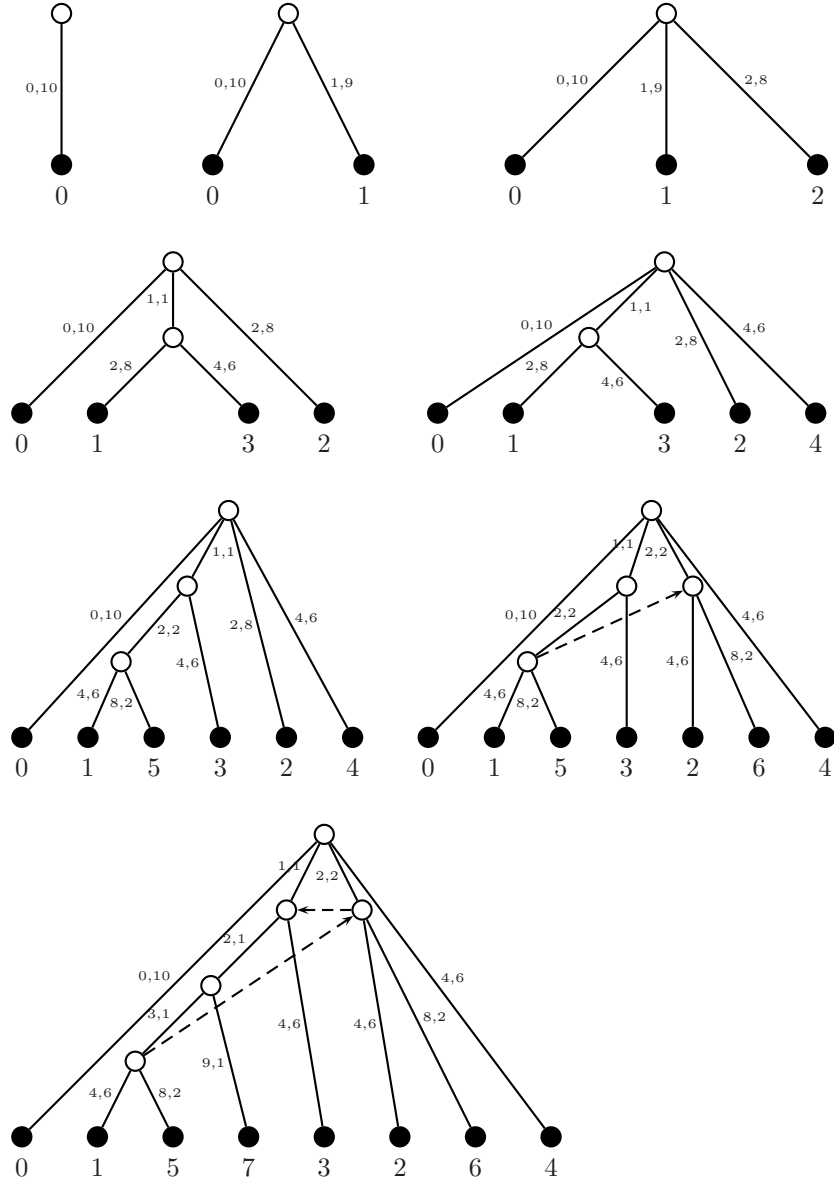
- $head_j$  is the longest prefix of  $y[j \dots n - 1]$  which is also a prefix of  $y[i \dots n - 1]$  for some  $i < j$ .
- $tail_j$  is the word such that  $y[j \dots n - 1] = head_j tail_j$ .

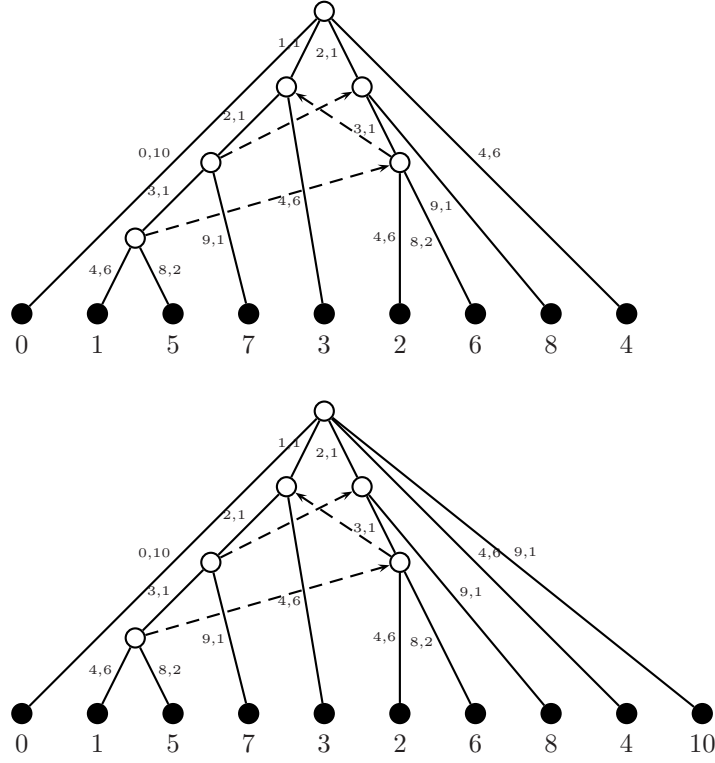
The strategy to insert the  $i$ th suffix in the tree is based on these definitions and described in Fig. 1.29.

The second step of the insertion (Fig. 1.29) is clearly performed in constant time. Thus, finding the node  $h$  is critical for the overall performance of the algorithm. A brute-force method to find it consists in spelling the current suffix  $y[j \dots n - 1]$  from the root of the tree, giving an  $O(|head_j|)$  time complexity for the insertion at step  $j$ , and an  $O(n^2)$  running time to build  $S(y)$ . Adding short-cut links leads to an overall  $O(n)$  time complexity, although there is no guarantee that insertion at step  $j$  is realized in constant time.

**Example 1.11:** The different tries during the construction of the suffix tree of  $y = \text{CAGATAGAG}$ . Leaves are black and labeled by the position of the suffix they represent. Plain arrows are labeled by pairs: the pair  $(j, \ell)$  stands

for the segment  $y[j \dots j + \ell - 1]$ . Dashed arrows represent the nontrivial suffix links.





### 1.4.1 McCreight Algorithm

The key to get an efficient construction of the suffix tree  $S(y)$  is to add links between nodes of the tree: they are called *suffix links*. Their definition relies on the relationship between  $head_{j-1}$  and  $head_j$ : if  $head_{j-1}$  is of the form  $az$  ( $a \in \Sigma$ ,  $z \in \Sigma^*$ ), then  $z$  is a prefix of  $head_j$ . In the suffix tree the node associated with  $z$  is linked to the node associated with  $az$ . The suffix link creates a shortcut in the tree that helps with finding the next head efficiently. The insertion of the next suffix, namely,  $head_j tail_j$ , in the tree reduces to the insertion of  $tail_j$  from the node associated with  $head_j$ .

The following property is an invariant of the construction: in  $T_j$ , only the node  $h$  associated with  $head_j$  can fail to have a valid suffix link. This effectively happens when  $h$  has just been created at step  $j$ . The procedure to find the next head at step  $j$  is composed of two main phases:

**A Rescanning:** Assume that  $head_{j-1} = az$  ( $a \in \Sigma$ ,  $z \in \Sigma^*$ ) and let  $d'$  be the associated node. If the suffix link on  $d'$  is defined, it leads to a node  $d$  from which the second step starts. Otherwise, the suffix link on  $d'$  is found by rescanning as follows. Let  $c'$  be the parent of  $d'$ , and let  $(j, \ell)$

---

```

M(y, n)
1  root ← INIT(y, n)
2  head ← root
3  tail ← child(root, y[0])
4  n ← n - 1
5  while n > 0
6      do if head = root                                ▷ Phase A (rescanning)
7          then d ← root
8              (j, ℓ) ← label(tail)
9              γ ← (j + 1, ℓ - 1)
10         else γ ← label(tail)
11             if link(head) ≠ UNDEFINED
12                 then d ← link(head)
13                 else (j, ℓ) ← label(head)
14                     if parent(head) = root
15                         then d ← RESCAN(root, j + 1, ℓ - 1)
16                         else d ← RESCAN(link(parent(head)), j, ℓ)
17                     link(head) ← d
18         (head, γ) ← SCAN(d, γ)                        ▷ Phase B (scanning)
19         create a new node tail
20         parent(tail) ← head
21         label(tail) ← γ
22         (j, ℓ) ← γ
23         child(head, y[j]) ← tail
24         n ← n - 1
25  return root

```

---

FIGURE 1.30: Suffix tree construction.

---

```

INIT( $y, n$ )
1  create a new node  $root$ 
2  create a new node  $c$ 
3   $parent(root) \leftarrow \text{UNDEFINED}$ 
4   $parent(c) \leftarrow root$ 
5   $child(root, y[0]) \leftarrow c$ 
6   $label(root) \leftarrow \text{UNDEFINED}$ 
7   $label(c) \leftarrow (0, n)$ 
8  return  $root$ 

```

---

FIGURE 1.31: Initialization procedure.

be the label of edge  $(c', d')$ . For the ease of the description, assume that  $az = av(y[j \dots j + \ell - 1])$  (it may happen that  $az = y[j \dots j + \ell - 1]$ ). There is a suffix link defined on  $c'$  and going to some node  $c$  associated with  $v$ . The crucial observation here is that  $y[j \dots j + \ell - 1]$  is the prefix of the label of some branch starting at node  $c$ . Then, the algorithm rescans  $y[j \dots j + \ell - 1]$  in the tree: let  $e$  be the child of  $c$  along that branch, and let  $(k, m)$  be the label of edge  $(c, e)$ . If  $m < \ell$ , then a recursive rescan of  $q = y[j + m \dots j + \ell - 1]$  starts from node  $e$ . If  $m > \ell$ , the edge  $(c, e)$  is broken to insert a new node  $d$ ; labels are updated correspondingly. If  $m = \ell$ ,  $d$  is simply set to  $e$ . If the suffix link of  $d'$  is currently undefined, it is set to  $d$ .

**B Scanning:** A downward search starts from  $d$  to find the node  $h$  associated with  $head_j$ . The search is dictated by the characters of  $tail_{j-1}$  one at a time from left to right. If necessary a new internal node is created at the end of the scanning.

After the two phases A and B are executed, the node associated with the new head is known, and the tail of the current suffix can be inserted in the tree.

To analyze the time complexity of the entire algorithm one mainly has to evaluate the total time of all scannings, and the total time of all rescannings. Assume that the alphabet is fixed, so that branching from a node to one of its children can be implemented to take constant time. Thus, the time spent for all scannings is linear because each letter of  $y$  is scanned only once. The same holds true for rescannings because each step downward (through node  $e$ ) increases strictly the position of the segment of  $y$  considered there, and this position never decreases.

An implementation of McCreight's algorithm is shown in Fig. 1.30. The next figures (Figs. 1.31–1.34) give the procedures used by the algorithm, especially procedures RESCAN and SCAN.

The following notation will be used:

- $parent(c)$  is the parent node of the node  $c$ ,

---

```

RESCAN( $c, j, \ell$ )
1  ( $k, m$ )  $\leftarrow$  label( $child(c, y[j])$ )
2  while  $\ell > 0$  and  $\ell \geq m$ 
3      do  $c \leftarrow child(c, y[j])$ 
4           $\ell \leftarrow \ell - m$ 
5           $j \leftarrow j + m$ 
6          ( $k, m$ )  $\leftarrow$  label( $child(c, y[j])$ )
7  if  $\ell > 0$ 
8      then return BREAK-EDGE( $child(c, y[j]), \ell$ )
9  else return  $c$ 

```

---

FIGURE 1.32: The crucial rescan operation.

---

```

BREAK-EDGE( $c, k$ )
1  create a new node  $g$ 
2   $parent(g) \leftarrow parent(c)$ 
3  ( $j, \ell$ )  $\leftarrow$  label( $c$ )
4   $child(parent(c), y[j]) \leftarrow g$ 
5   $label(g) \leftarrow (j, k)$ 
6   $parent(c) \leftarrow g$ 
7   $label(c) \leftarrow (j + k, \ell - k)$ 
8   $child(g, y[j + k]) \leftarrow c$ 
9   $link(g) \leftarrow$  UNDEFINED
10 return  $g$ 

```

---

FIGURE 1.33: Breaking an edge.

- $label(c)$  is the pair  $(i, l)$  if the edge from the parent node of  $c$  to  $c$  itself is associated with the factor  $y[i \dots i + l - 1]$ ,
- $child(c, a)$  is the only node that can be reached from the node  $c$  with the character  $a$ ,
- $link(c)$  is the suffix node of the node  $c$ .

---

## 1.5 Suffix Arrays

Suffix trees are very powerful however they are very space consuming. In most cases they can be replaced by suffix arrays that are more space economical. The technique involves binary search in the sorted list of the text suffixes. In this section, we show how to lexicographically sort the suffixes of a string  $y$  of length  $n$ . The resulting permutation, as an array, constitutes the

---

```

SCAN( $d, \gamma$ )
1  ( $j, \ell$ )  $\leftarrow \gamma$ 
2  while  $child(d, y[j]) \neq \text{UNDEFINED}$ 
3      do  $g \leftarrow child(d, y[j])$ 
4           $k \leftarrow 1$ 
5          ( $s, lg$ )  $\leftarrow label(g)$ 
6           $s \leftarrow s + 1$ 
7           $\ell \leftarrow \ell - 1$ 
8           $j \leftarrow j + 1$ 
9          while  $k < lg$  and  $y[j] = y[s]$ 
10             do  $j \leftarrow j + 1$ 
11                  $s \leftarrow s + 1$ 
12                  $k \leftarrow k + 1$ 
13                  $\ell \leftarrow \ell - 1$ 
14             if  $k < lg$ 
15                 then return (BREAK-EDGE( $g, k$ ), ( $j, \ell$ ))
16          $d \leftarrow g$ 
17 return ( $d, (j, \ell)$ )

```

---

FIGURE 1.34: The scan operation.

suffix array of the string. It is usually enhanced with the LCP array described further in the section to get an efficient string searching algorithm.

The goal of the sorting is to compute a permutation  $p$  of the indices on  $y$  that satisfies the condition

$$y[p[0]..n-1] < y[p[1]..n-1] < \cdots < y[p[n-1]..n-1]. \quad (1.1)$$

Note that the inequalities are strict since two suffixes occurring at distinct positions cannot be identical.

The implementation of a standard lexicographic sorting method (repeated radix sort), leads to an algorithm whose execution time is  $O(n^2)$  since the sum of the lengths of the suffixes of  $y$  is quadratic.

Actually, the ordering is not entirely sufficient to get an efficient search. The precomputation and the utilization of common prefixes to the suffixes are extra elements that make the technique very efficient. The search for a string of length  $m$  in a string length  $n$  then takes  $O(m + \log n)$  time.

To describe the sorting algorithm, for  $u \in \Sigma^*$ , we denote by

$$first_k(u) = \begin{cases} u & \text{if } |u| \leq k, \\ u[0..k-1] & \text{otherwise,} \end{cases}$$

the beginning of order  $k$  of the string  $u$ .

### 1.5.1 Kärkkäinen–Sanders Algorithm

In this section, the alphabet of  $y$  is assumed to be a bounded segment of integers, as it can be considered in most real applications. With this condition, not only letters can be sorted in linear time but suffixes of  $y$  as well.

The present algorithm for sorting the suffixes proceeds in four steps as follows, where  $k$  is an appropriate non-negative integer.

- Step 1** Positions  $i$  on the input string  $y$  are sorted for two third of them, namely for  $i = 3k$  or  $i = 3k + 1$ , according to  $\text{first}_3(y[i..n-1])$ . Let  $t[i]$  be the rank of  $i$  in the sorted list.
- Step 2** Suffixes of the  $2/3$ -shorter word  $z = t[0]t[3] \cdots t[3k] \cdots t[1]t[4] \cdots t[3k+1] \cdots$  are recursively sorted. Let  $s[i]$  be the rank of suffix at position  $i$  on  $y$  in the sorted list of them derived from the sorted list of suffixes of  $z$ .
- Step 3** Suffixes  $y[j..n-1]$ , for  $j$  of the form  $3k+2$ , are sorted using the table  $s$ .
- Step 4** The final step consists in merging the ordered lists obtained at the second and third steps.

A careful implementation of the algorithm leads to a linear running time. It relies on the following elements. The first step can be executed in linear time by using a mere radix sort. Since the order of suffixes  $y[j+1..n-1]$  is already known from  $s$ , the third step can be done in linear time by just radix sorting pairs  $(y[j], s[j+1])$ . Comparing suffixes at positions  $i$  ( $i = 3k$  or  $i = 3k + 1$  for the first list) and  $j$  ( $j = 3k + 2$  for the second list) remains to compare pairs of the form  $(y[i], s[i+1])$  and  $(y[j], s[j+1])$  if  $i = 3k$  or pairs of the form  $(y[i]y[i+1], s[i+2])$  and  $(y[j]y[j+1], s[j+2])$  if  $i = 3k + 1$ . This is done in constant time and then the merge at the fourth step can thus be realized in linear time.

An example of a run of the algorithm is shown in Figure 1.35: (1) String  $y = \text{abaaabaaabb}$  and its two sets of positions  $P_{01}$  and  $P_2$ . (2) Step 1. Strings  $\text{first}_3(y[i..n-1])$  for  $i \in P_{01}$  and their ranks:  $t[i]$  is the rank of  $i$  in the sorted list. (3) Step 2. Positions in  $P_{01}$  sorted according to their associated suffixes in  $z$ , resulting in  $L_{01}$  and the table of ranks  $s$ . (4) Step 3. Positions  $j$  in  $P_2$  sorted according to pairs  $(y[j], s[j+1])$  resulting in  $L_2$ . (5) Step 4. Pairs used for comparing positions when merging the sorted lists  $L_{01}$  and  $L_2$ . (6) Permutation  $p$  corresponding to the sorted suffixes of  $y$ .

The algorithm SKEW-SUFFIX-SORT (Fig. 1.36) describes the method presented above in a more precise way. To shorten the presentation of the algorithm, the definition of  $s$  (see Line 11) is extended to positions  $n$  and  $n+1$  that are considered Lines 12 and 13 (call to COMP, Fig. 1.37).

The recursivity of the algorithm (Line 8) yields the recurrence relation  $T(n) = T(2n/3) + O(n)$ , for  $n > 3$ , and  $T(n) = O(1)$ , for  $n \leq 3$ , because all other lines execute in constant time, i.e.  $O(n)$  time. The recurrence has

	$i$	0	1	2	3	4	5	6	7	8	9	10	
(1)	$y[i]$	<b>a</b>	<b>b</b>	<b>a</b>	<b>a</b>	<b>a</b>	<b>b</b>	<b>a</b>	<b>a</b>	<b>a</b>	<b>b</b>	<b>b</b>	
	$P_{01} = \{0, 1, 3, 4, 6, 7, 9, 10\}$ and $P_2 = \{2, 5, 8\}$												
	$i$	$i \bmod 3 = 0$				$i \bmod 3 = 1$							
(2)	$first_3(y[i..n-1])$	0	3	6	9	1	4	7	10				
	$t[i]$	<b>aba</b>	<b>aab</b>	<b>aaa</b>	<b>bb</b>	<b>baa</b>	<b>aba</b>	<b>aab</b>	<b>b</b>				
		2	1	0	5	4	2	1	3				
(3)	$z = 2\ 1\ 0\ 5\ 4\ 2\ 1\ 3$ and $L_{01} = (6, 3, 7, 0, 4, 10, 1, 9)$												
	$s[i]$	3				1	0	7	6				
						3	1	0	7				
	$j$	$j \bmod 3 = 2$											
(4)	$(y[j], s[j+1])$	2				5	8						
		<b>(a, 1)</b>				<b>(b, 0)</b>	<b>(a, 7)</b>						
	$L_2 = (2, 8, 5)$												
	$i$	$i \bmod 3 = 0 \text{ or } 1$											
		6	3	7	0	4	10	1	9				
(5)	$(y[i]y[i+1], s[i+2])$					<b>(aa, 7)</b>		<b>(ab, 0)</b>		<b>(b, -1)</b>	<b>(ba, 1)</b>		
	$(y[i], s[i+1])$	<b>(a, 2)</b>		<b>(a, 4)</b>		<b>(a, 6)</b>		<b>(b, 5)</b>					
	$j$	$j \bmod 3 = 2$											
		2	8	5									
	$(y[j]y[j+1], s[j+2])$	<b>(aa, 4)</b>		<b>(ab, 5)</b>		<b>(ba, 2)</b>							
	$(y[j], s[j+1])$	<b>(a, 1)</b>		<b>(a, 7)</b>		<b>(b, 0)</b>							
(6)	$i$	0	1	2	3	4	5	6	7	8	9	10	
	$p[i]$	2	6	3	7	0	4	8	10	1	5	9	

FIGURE 1.35: Building the Suffix Array of  $y = \mathbf{abaaabaaabb}$ .

---

```

SKEW-SUFFIX-SORT( $y, n$ )
1  if  $n \leq 3$ 
2    then return permutation of the sorted suffix of  $y$ 
3  else  $P_{01} \leftarrow \{i \mid 0 \leq i < n \text{ and } (i \bmod 3 = 0 \text{ or } i \bmod 3 = 1)\}$ 
4    if  $n \bmod 3 = 0$ 
5      then  $P_{01} \leftarrow P_{01} \cup \{n\}$ 
6     $t \leftarrow$  table of ranks of positions  $i$  in  $P_{01}$  according to
       $\text{first}_3(y[i..n-1])$ 
7     $z \leftarrow t[0]t[3] \cdots t[3k] \cdots t[1]t[4] \cdots t[3k+1] \cdots$ 
8     $q \leftarrow \text{SKEW-SUFFIX-SORT}(z, \lfloor 2n/3 \rfloor + 1)$ 
9     $L_{01} \leftarrow (3q[j] \text{ if } 0 \leq q[j] \leq \lfloor n/3 \rfloor + 1, 3q[j] + 1 \text{ otherwise}$ 
      with  $j = 0, 1, \dots, |z| - 1)$ 
10    $s \leftarrow$  table of ranks of positions in  $L_{01}$ 
11    $(s[n], s[n+1]) \leftarrow (-1, -1)$ 
12    $L_2 \leftarrow$  list of positions  $j = 3k + 2, 3k + 2 < n$ 
      sorted according to  $(y[j], s[j+1])$ 
13    $L \leftarrow$  merge of  $L_{01}$  and  $L_2$  using COMP()
14    $p \leftarrow$  permutation of positions on  $y$  corresponding to  $L$ 
15   return  $p$ 

```

---

FIGURE 1.36: Linear time suffix array construction on a bounded alphabet.

---

```

COMP( $i, j$ )
1  if  $i \bmod 3 = 0$ 
2    then if  $(y[i], s[i+1]) < (y[j], s[j+1])$ 
3      then return -1
4      else return 1
5  else if  $(y[i..i+1], s[i+2]) < (y[j..j+1], s[j+2])$ 
6    then return -1
7    else return 1

```

---

FIGURE 1.37: Constant time comparison of  $y[i..n-1]$  and  $y[j..n-1]$  during the merge.

---

```

BINARY-SEARCH( $y, n, p, x, m$ )
1   $\ell \leftarrow -1$ 
2   $r \leftarrow n$ 
3  while  $\ell + 1 < r$ 
4      do  $i \leftarrow \lfloor (\ell + r)/2 \rfloor$ 
5           $k \leftarrow |lcp(x, y[p[i]..n-1])|$ 
6          if  $k = m$ 
7              then return  $i$ 
8          elseif  $x \leq y[p[i]..n-1]$ 
9              then  $r \leftarrow i$ 
10         else  $\ell \leftarrow i$ 
11 return  $(\ell, r)$ 

```

---

FIGURE 1.38: Binary search of  $x$  is the suffix array of  $y$ .

solution  $T(n) = O(n)$ , Algorithm SKEW-SUFFIX-SORT computes the suffix array of a string of length  $n$  in time  $O(n)$ .

### 1.5.2 Substring Search

Searching for a pattern  $x$  of length  $m$  in the string  $y$  using its suffix array can be done by a simple binary search (see Fig. 1.38). It returns:

- $(-1, 0)$  if  $x < y[p[0]..n-1]$ ;
- $(n-1, n)$  if  $x > y[p[n-1]..n-1]$ ;
- $i$  if  $x$  is a prefix of  $y[p[i]..n-1]$ ;
- $(i, i+1)$  if  $y[p[i]..n-1] < x < y[p[i+1]..n-1]$ .

The loop line 3 iterates at most  $\log n$  times and each test line 8 can perform at most  $m$  character comparisons. Thus the time complexity of algorithm is  $O(m \times \log n)$ .

### 1.5.3 Longest Common Prefixes

In this section the second element that constitutes a suffix array is described: the array  $LCP$  storing the maximal lengths of prefixes common to suffixes in the sorted list. We first introduce the following notation for two strings  $u, v \in \Sigma^*$ :

$$lcp(u, v) = \text{longest common prefix of } u \text{ and } v.$$

We start by the first half of the array  $LCP$  storing the maximal lengths of prefixes common to consecutive suffixes in the sorted list. It is defined as follows, for  $1 \leq i \leq n-1$ :

$$LCP[i] = lcp(y[p[i-1]..n-1], y[p[i]..n-1]).$$

---

```

DEF-HALF-LCP( $y, n, p$ )
1  for  $i \leftarrow 0$  to  $n - 1$ 
2      do  $\text{Rank}[p[i]] \leftarrow i$ 
3   $\ell \leftarrow 0$ 
4  for  $j \leftarrow 0$  to  $n - 1$ 
5      do  $\ell \leftarrow \max\{0, \ell - 1\}$ 
6           $i \leftarrow \text{Rank}[j]$ 
7          if  $i \neq 0$ 
8              then  $j' \leftarrow p[i - 1]$ 
9                  while  $j + \ell < n$  and  $j' + \ell < n$ 
                      and  $y[j + \ell] = y[j' + \ell]$ 
10                     do  $\ell \leftarrow \ell + 1$ 
11          else  $\ell \leftarrow 0$ 
12           $\text{LCP}[i] \leftarrow \ell$ 
13 return  $\text{LCP}$ 

```

---

FIGURE 1.39: Linear computation of the first LCP values.

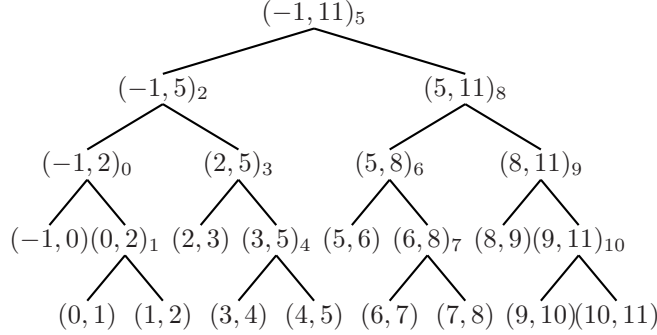
A direct computation using letter by letter comparisons leads to an execution time in  $O(n^2)$  since the sum of lengths of the suffixes is quadratic.

We describe an algorithm that performs the computation in linear time. Indeed, the suffixes of  $y$  are not independent of each other. This dependence allows to reduce the computation time by the mean of a quite simple algorithm, based on the following property: let  $i, j, i'$  be positions on  $y$  for which  $j = p[i]$  and  $j - 1 = p[i']$ , then  $\text{LCP}[i'] - 1 \leq \text{LCP}[i]$ .

Using this result, in order to compute  $\text{LCP}[i]$ , i.e. the length of the longest common prefix between  $y[p[i]..n-1]$  and  $y[p[i-1]..n-1]$  when  $0 < i \leq n$ , one can start the letter by letter comparison exactly at the position at which stops the previous computation, the one of  $\text{LCP}[i']$ . Knowing that, it is sufficient to proceed by considering the suffixes from the longest to the shortest, and not in the lexicographic order that seems more natural. This is what Algorithm DEF-HALF-LCP (see Fig. 1.39) realizes. It computes the values  $\text{LCP}[i]$  for  $0 \leq i \leq n$ . Note that to determine the position  $i$  associated with position  $j$ , the algorithm uses the reverse of permutation  $p$  which is computed in a first step (Lines 1–2). This function is represented by the table called *Rank* since it indicates the rank of each suffix in the sorted list of suffixes of  $y$ .

Applied to string  $y$  of length  $n$  and to the permutation  $p$  of its suffixes, Algorithm DEF-HALF-LCP computes the *LCP* array in time  $O(n)$ .

The number of pairs  $(\ell, r)$  considered by algorithm BINARY-SEARCH( $y, n, p, x, m$ ) is bounded by  $2n + 1$ : there are  $n + 1$  pairs of the form  $(i, i + 1)$  that constitute the leaves of a binary tree thus there are at most  $n$  internal nodes. Fig. 1.40 shows an example for a string of length 11. Small numbers close to pairs  $(\ell, r)$  are  $i = \lfloor (\ell + r)/2 \rfloor$ .



**FIGURE 1.40:** Pairs considered during the binary search in a string of length 11.

---

```

DEF-LCP( $y, \ell, r, LCP$ )
1  ▷  $\ell < r$ 
2  if  $\ell + 1 = r$ 
3    then return  $LCP[r]$ 
4  else  $i \leftarrow \lfloor (\ell + r)/2 \rfloor$ 
5     $LCP[n + 1 + i] \leftarrow \min\{\text{DEF-LCP}(y, \ell, i, LCP),$ 
6       $\text{DEF-LCP}(y, i, r, LCP)\}$ 
7    return  $LCP[n + 1 + i]$ 

```

---

**FIGURE 1.41:** Linear computation of the last LCP values.

It is thus possible to extend array  $LCP$  as follows:

$$LCP[n + 1 + i] = \text{lcp}(p[\ell]..n - 1, y[p[r]..n - 1])$$

where  $i = \lfloor (\ell + r)/2 \rfloor$ .

The length of the longest common prefix between  $y[p[\ell]..n - 1]$  and  $y[p[r]..n - 1]$  when  $\ell < r - 1$  is equal to the minimum value among all  $LCP[i]$  where  $\ell < i \leq r$ . Then these values can be computed by a call to the algorithm  $\text{DEF-LCP}(y, 0, n - 1, LCP)$  (see Fig. 1.41).

#### 1.5.4 Substring Search with the Length of the Common Prefixes

Let  $\ell, r, i$  be three integers,  $0 \leq \ell < i < r < n$ . If  $y[p[\ell]..n - 1] < x < y[p[r]..n - 1]$  let  $lb = |\text{lcp}(x, y[p[\ell]..n - 1])|$  and  $le = |\text{lcp}(x, y[p[r]..n - 1])|$  satisfying  $lb \leq le$ . We then have:

$$|\text{lcp}(y[p[i]..n - 1], y[p[r]..n - 1])| < le \text{ implies } y[p[i]..n - 1] < x < y[p[r]..n - 1]$$

and

$$|\text{lcp}(y[p[i]..n - 1], y[p[r]..n - 1])| > le \text{ implies } y[p[\ell]..n - 1] < x < y[p[i]..n - 1].$$

---

```

SEARCH( $y, n, p, LCP, x, m$ )
1  ( $\ell, lb$ )  $\leftarrow (-1, 0)$ 
2  ( $r, le$ )  $\leftarrow (n, 0)$ 
3  while  $\ell + 1 < r$ 
4      do  $i \leftarrow \lfloor (\ell + r)/2 \rfloor$ 
5          if  $\ell + 1 = i$ 
6              then  $g \leftarrow LCP[i]$ 
7              else  $g \leftarrow LCP[n + 1 + \lfloor (\ell + i)/2 \rfloor]$ 
8          if  $i + 1 = r$ 
9              then  $h \leftarrow LCP[r]$ 
10             else  $h \leftarrow LCP[n + 1 + \lfloor (i + r)/2 \rfloor]$ 
11             if  $lb \leq h$  and  $h < le$ 
12                 then ( $\ell, lb$ )  $\leftarrow (i, h)$ 
13             elseif  $lb \leq le$  and  $le < h$ 
14                 then ( $r, le$ )  $\leftarrow (i, h)$ 
15             elseif  $le \leq g$  and  $g < lb$ 
16                 then ( $r, le$ )  $\leftarrow (i, g)$ 
17             elseif  $le \leq lb$  and  $lb < g$ 
18                 then ( $\ell, lb$ )  $\leftarrow (i, g)$ 
19             else  $\ell \leftarrow \max\{lb, le\}$ 
20                  $\ell \leftarrow \ell + |lcp(x[\ell \dots m - 1], y[p[i] + [\ell \dots n - 1]])|$ 
21                 if  $\ell = m$ 
22                     then return  $i$ 
23                 elseif  $x < y[p[i] \dots n - 1]$ 
24                     then ( $r, le$ )  $\leftarrow (i, \ell)$ 
25                 else ( $\ell, lb$ )  $\leftarrow (i, \ell)$ 
26 return ( $\ell, r$ )

```

---

**FIGURE 1.42:** Search for  $x$  in the suffix array of  $y$  using the length of the common prefixes.

The search can then now be done in  $O(m + \log n)$  with the algorithm SEARCH( $y, n, p, LCP, x, m$ ) (see Fig. 1.42).

## 1.6 Alignment

Alignments are used to compare strings. They are widely used in computational molecular biology. They constitute a mean to visualize resemblance between strings. They are based on notions of distance or similarity. Their computation is usually done by dynamic programming. A typical example of this method is the computation of the longest common subsequence of two strings. The reduction of the memory space presented on it can be applied to similar problems. Three different kinds of alignment of two strings  $x$  and  $y$  are

considered: global alignment (that consider the whole strings  $x$  and  $y$ ), local alignment (that enable to find the segment of  $x$  that is closer to a segment of  $y$ ) and the longest common subsequence of  $x$  and  $y$ .

An **alignment** of two strings  $x$  and  $y$  of length  $m$  and  $n$  respectively consists in aligning their symbols on vertical lines. Formally an alignment of two strings  $x, y \in \Sigma^*$  is a word  $w$  on the alphabet  $(\Sigma \cup \{\varepsilon\}) \times (\Sigma \cup \{\varepsilon\}) \setminus \{(\varepsilon, \varepsilon)\}$  ( $\varepsilon$  is the empty word) whose projection on the first component is  $x$  and whose projection of the second component is  $y$ .

Thus an alignment  $w = (\bar{x}_0, \bar{y}_0)(\bar{x}_1, \bar{y}_1) \cdots (\bar{x}_{p-1}, \bar{y}_{p-1})$  of length  $p$  is such that  $x = \bar{x}_0 \bar{x}_1 \cdots \bar{x}_{p-1}$  and  $y = \bar{y}_0 \bar{y}_1 \cdots \bar{y}_{p-1}$  with  $\bar{x}_i \in \Sigma \cup \{\varepsilon\}$  and  $\bar{y}_i \in \Sigma \cup \{\varepsilon\}$  for  $0 \leq i \leq p-1$ . The alignment is represented as follows

$$\begin{array}{cccc} \bar{x}_0 & \bar{x}_1 & \cdots & \bar{x}_{p-1} \\ \bar{y}_0 & \bar{y}_1 & \cdots & \bar{y}_{p-1} \end{array}$$

with the symbol  $-$  instead of the symbol  $\varepsilon$ .

**Example 1.12:**

$$\begin{array}{cccccc} \mathbf{A} & \mathbf{C} & \mathbf{G} & - & - & \mathbf{A} \\ \mathbf{A} & \mathbf{T} & \mathbf{G} & \mathbf{C} & \mathbf{T} & \mathbf{A} \end{array}$$

is an alignment of **ACGA** and **ATGCTA**.

### 1.6.1 Global alignment

A global alignment of two strings  $x$  and  $y$  can be obtained by computing the distance between  $x$  and  $y$ . The notion of distance between two strings is widely used to compare files. The **diff** command of UNIX operating system implements an algorithm based on this notion, in which lines of the files are treated as symbols. The output of a comparison made by **diff** gives the minimum number of operations (substitute a symbol, insert a symbol, or delete a symbol) to transform one file into the other.

Let us define the edit distance between two strings  $x$  and  $y$  as follows: it is the minimum number of elementary edit operations than enable to transform  $x$  into  $y$ . The elementary edit operations are:

- the substitution of a character of  $x$  at a given position by a character of  $y$ ,
- the deletion of a character of  $x$  at a given position,
- the insertion of a character of  $y$  in  $x$  at a given position.

A cost is associated to each elementary edit operation. For  $a, b \in \Sigma$ :

- $Sub(a, b)$  denotes the cost of the substitution of the character  $a$  by the character  $b$ ,
- $Del(a)$  denotes the cost of the deletion of the character  $a$ ,
- $Ins(a)$  denotes the cost of the insertion of the character  $a$ .

This means that the costs of the edit operations are independent of the positions where the operations occur. The edit distance of two strings  $x$  and  $y$  can now be defined by

$$edit(x, y) = \min\{\text{cost of } \gamma \mid \gamma \in \Gamma_{x,y}\}$$

where  $\Gamma_{x,y}$  is the set of all the sequences of edit operations that transform  $x$  into  $y$ , and the cost of an element  $\gamma \in \Gamma_{x,y}$  is the sum of the costs of its elementary edit operations.

In order to compute  $edit(x, y)$  for two strings  $x$  and  $y$  of length  $m$  and  $n$  respectively, a two-dimensional table  $T$  of  $m + 1$  rows and  $n + 1$  columns is used such that

$$T[i, j] = edit(x[i], y[j])$$

for  $i = 0, \dots, m - 1$  and  $j = 0, \dots, n - 1$ . It follows  $edit(x, y) = T[m - 1, n - 1]$ .

The values of the table  $T$  can be computed by the following recurrence formula:

$$\begin{aligned} T[-1, -1] &= 0, \\ T[i, -1] &= T[i - 1, -1] + Del(x[i]), \\ T[-1, j] &= T[-1, j - 1] + Ins(y[j]), \\ T[i, j] &= \min \begin{cases} T[i - 1, j - 1] + Sub(x[i], y[j]), \\ T[i - 1, j] + Del(x[i]), \\ T[i, j - 1] + Ins(y[j]), \end{cases} \end{aligned}$$

for  $i = 0, 1, \dots, m - 1$  and  $j = 0, 1, \dots, n - 1$ .

The value at position  $(i, j)$  in the table  $T$  only depends on the values at the three neighbor positions  $(i - 1, j - 1)$ ,  $(i - 1, j)$  and  $(i, j - 1)$ .

The direct application of the above recurrence formula gives an exponential time algorithm to compute  $T[m - 1, n - 1]$ . However the whole table  $T$  can be computed in quadratic time, technique known as “dynamic programming”. This is a general technique that is used to solve the different kinds of alignments.

The computation of the table  $T$  proceeds in two steps. First it initializes the first column and first row of  $T$ , this is done by a call to a generic function MARGIN which is a parameter of the algorithm and that depends on the kind of alignment that is considered. Second it computes the remaining values of  $T$ , that is done by a call to a generic function FORMULA which is a parameter of the algorithm and that depends on the

---

```

GENERIC-DP( $x, m, y, n, \text{MARGIN}, \text{FORMULA}$ )
1  MARGIN( $T, x, m, y, n$ )
2  for  $j \leftarrow 0$  to  $n - 1$ 
3      do for  $i \leftarrow 0$  to  $m - 1$ 
4          do  $T[i, j] \leftarrow \text{FORMULA}(T, x, i, y, j)$ 
5  return  $T$ 

```

---

FIGURE 1.43: Computation of the table  $T$  by dynamic programming.

---

```

MARGIN-GLOBAL( $T, x, m, y, n$ )
1   $T[-1, -1] \leftarrow 0$ 
2  for  $i \leftarrow 0$  to  $m - 1$ 
3      do  $T[i, -1] \leftarrow T[i - 1, -1] + \text{Del}(x[i])$ 
4  for  $j \leftarrow 0$  to  $n - 1$ 
5      do  $T[-1, j] \leftarrow T[-1, j - 1] + \text{Ins}(y[j])$ 

```

---

FIGURE 1.44: Margin initialization for the computation of a global alignment.

kind of alignment that is considered. Computing a global alignment of  $x$  and  $y$  can be done by a call to **GENERIC-DP** with the following parameters  $(x, m, y, n, \text{MARGIN-GLOBAL}, \text{FORMULA-GLOBAL})$  (see Fig. 1.43, 1.44 and 1.45). The computation of all the values of the table  $T$  can thus be done in quadratic space and time:  $O(m \times n)$ .

An optimal alignment (with minimal cost) can then be produced by a call to the function

**ONE-ALIGNMENT**( $T, x, m - 1, y, n - 1$ )

(see Fig. 1.46). It consists in tracing back the computation of the values of the table  $T$  from position  $[m - 1, n - 1]$  to position  $[-1, -1]$ . At each cell  $[i, j]$  the algorithm determines among the three values  $T[i - 1, j - 1] + \text{Sub}(x[i], y[j])$ ,  $T[i - 1, j] + \text{Del}(x[i])$  and  $T[i, j - 1] + \text{Ins}(y[j])$  which has been used to produce the value of  $T[i, j]$ . If  $T[i - 1, j - 1] + \text{Sub}(x[i], y[j])$  has been used it adds  $(x[i], y[j])$  to the optimal alignment and proceeds recursively with the cell at  $[i - 1, j - 1]$ . If  $T[i - 1, j] + \text{Del}(x[i])$  has been used it adds  $(x[i], -)$  to the optimal alignment and proceeds recursively with cell at  $[i - 1, j]$ . If  $T[i, j - 1] +$

---

```

FORMULA-GLOBAL( $T, x, i, y, j$ )
1  return  $\min \begin{cases} T[i - 1, j - 1] + \text{Sub}(x[i], y[j]) \\ T[i - 1, j] + \text{Del}(x[i]) \\ T[i, j - 1] + \text{Ins}(y[j]) \end{cases}$ 

```

---

FIGURE 1.45: Computation of  $T[i, j]$  for a global alignment.

---

```

ONE-ALIGNMENT( $T, x, i, y, j$ )
1  if  $i = -1$  and  $j = -1$ 
2    then return  $(\varepsilon, \varepsilon)$ 
3  else if  $i = -1$ 
4    then return ONE-ALIGNMENT( $T, x, -1, y, j - 1$ )  $\cdot (\varepsilon, y[j])$ 
5  elseif  $j = -1$ 
6    then return ONE-ALIGNMENT( $T, x, i - 1, y, -1$ )  $\cdot (x[i], \varepsilon)$ 
7  else if  $T[i, j] = T[i - 1, j - 1] + \text{Sub}(x[i], y[j])$ 
8    then return ONE-ALIGNMENT( $T, x, i - 1, y, j - 1$ )  $\cdot$ 
       $(x[i], y[j])$ 
9    elseif  $T[i, j] = T[i - 1, j] + \text{Del}(x[i])$ 
10     then return ONE-ALIGNMENT( $T, x, i - 1, y, j$ )  $\cdot$ 
       $(x[i], \varepsilon)$ 
11   else return ONE-ALIGNMENT( $T, x, i, y, j - 1$ )  $\cdot$ 
       $(\varepsilon, y[j])$ 

```

---

FIGURE 1.46: Recovering an optimal alignment.

$\text{Ins}(y[j])$  has been used it adds  $(-, y[j])$  to the optimal alignment and proceeds recursively with cell at  $[i, j - 1]$ . Recovering all the optimal alignments can be done by a similar technique.

**Example 1.13:**

$T$	$j$	-1	0	1	2	3	4	5
$i$		$y[j]$	A	T	G	C	T	A
-1	$x[i]$	0	1	2	3	4	5	6
0	A	1	0	1	2	3	4	5
1	C	2	1	1	2	2	3	4
2	G	3	2	2	1	2	3	4
3	A	4	3	3	2	2	3	3

The values of the above table have been obtained with the following unitary costs:  $\text{Sub}(a, b) = 1$  if  $a \neq b$  and  $\text{Sub}(a, a) = 0$ ,  $\text{Del}(a) = \text{Ins}(a) = 1$  for  $a, b \in \Sigma$ .

### 1.6.2 Local alignment

A local alignment of two strings  $x$  and  $y$  consists in finding the segment of  $x$  that is closer to a segment of  $y$ . The notion of distance used to compute global alignments cannot be used in that case since the segments of  $x$  closer to segments of  $y$  would only be the empty segment or individual characters.

This is why a notion of similarity is used based on a scoring scheme for edit operations.

A score (instead of a cost) is associated to each elementary edit operation. For  $a, b \in \Sigma$ :

- $Sub_S(a, b)$  denotes the score of substituting the character  $b$  for the character  $a$ ,
- $Del_S(a)$  denotes the score of deleting the character  $a$ ,
- $Ins_S(a)$  denotes the score of inserting the character  $a$ .

This means that the scores of the edit operations are independent of the positions where the operations occur. For two characters  $a$  and  $b$ , a positive value of  $Sub_S(a, b)$  means that the two characters are close to each other, and a negative value of  $Sub_S(a, b)$  means that the two characters are far apart.

The edit score of two strings  $x$  and  $y$  can now be defined by

$$sco(x, y) = \max\{\text{score of } \gamma \mid \gamma \in \Gamma_{x,y}\}$$

where  $\Gamma_{x,y}$  is the set of all the sequences of edit operations that transform  $x$  into  $y$  and the score of an element  $\sigma \in \Gamma_{x,y}$  is the sum of the scores of its elementary edit operations.

In order to compute  $sco(x, y)$  for two strings  $x$  and  $y$  of length  $m$  and  $n$  respectively, a two-dimensional table  $T$  of  $m + 1$  rows and  $n + 1$  columns is used such that

$$T[i, j] = sco(x[i], y[j])$$

for  $i = 0, \dots, m - 1$  and  $j = 0, \dots, n - 1$ . Therefore  $sco(x, y) = T[m - 1, n - 1]$ .

The values of the table  $T$  can be computed by the following recurrence formula:

$$\begin{aligned} T[-1, -1] &= 0, \\ T[i, -1] &= 0, \\ T[-1, j] &= 0, \\ T[i, j] &= \max \begin{cases} T[i - 1, j - 1] + Sub_S(x[i], y[j]), \\ T[i - 1, j] + Del_S(x[i]), \\ T[i, j - 1] + Ins_S(y[j]), \\ 0, \end{cases} \end{aligned}$$

for  $i = 0, 1, \dots, m - 1$  and  $j = 0, 1, \dots, n - 1$ .

Computing the values of  $T$  for a local alignment of  $x$  and  $y$  can be done by a call to **GENERIC-DP** with the following parameters ( $x, m, y, n, \text{MARGIN-LOCAL}, \text{FORMULA-LOCAL}$ ) in  $O(mn)$  time and space complexity (see Fig. 1.43, 1.47 and 1.48). Recovering a local alignment can be done in a way similar to what is done in the case of a global alignment (see Fig. 1.46)

---

```

MARGIN-LOCAL( $T, x, m, y, n$ )
1   $T[-1, -1] \leftarrow 0$ 
2  for  $i \leftarrow 0$  to  $m - 1$ 
3      do  $T[i, -1] \leftarrow 0$ 
4  for  $j \leftarrow 0$  to  $n - 1$ 
5      do  $T[-1, j] \leftarrow 0$ 

```

---

FIGURE 1.47: Margin initialization for computing a local alignment.

---

```

FORMULA-LOCAL( $T, x, i, y, j$ )
1  return  $\max \begin{cases} T[i-1, j-1] + \text{Sub}_S(x[i], y[j]) \\ T[i-1, j] + \text{Del}_S(x[i]) \\ T[i, j-1] + \text{Ins}_S(y[j]) \\ 0 \end{cases}$ 

```

---

FIGURE 1.48: Recurrence formula for computing a local alignment.

but the trace back procedure must start at a position of a maximal value in  $T$  rather than at position  $[m-1, n-1]$ .

**Example 1.14:** Computation of an optimal local alignment of

$x = \mathbf{EAWACQGKL}$  and  $y = \mathbf{ERDAWCQPGKWY}$  with scores:

$\text{Sub}_S(a, a) = 1$ ,  $\text{Sub}_S(a, b) = -3$  and  $\text{Del}_S(a) = \text{Ins}_S(a) = -1$  for  $a, b \in \Sigma$ ,  $a \neq b$ .

$T$	$j$	-1	0	1	2	3	4	5	6	7	8	9	10	11
$i$		$y[j]$	E	R	D	A	W	C	Q	P	G	K	W	Y
-1	$x[i]$	0	0	0	0	0	0	0	0	0	0	0	0	0
0	E	0	1	0	0	0	0	0	0	0	0	0	0	0
1	A	0	0	0	0	1	0	0	0	0	0	0	0	0
2	W	0	0	0	0	0	2	1	0	0	0	0	1	0
3	A	0	0	0	0	1	1	0	0	0	0	0	0	0
4	C	0	0	0	0	0	0	2	1	0	0	0	0	0
5	Q	0	0	0	0	0	0	1	3	2	1	0	0	0
6	G	0	0	0	0	0	0	0	2	1	3	2	1	0
7	K	0	0	0	0	0	0	0	1	0	2	4	3	2
8	L	0	0	0	0	0	0	0	0	0	1	3	2	1

The corresponding optimal local alignment is:

---

```

A W A C Q - G K
A W - C Q P G K

```

---

---

```

FORMULA-LCS( $T, x, i, y, j$ )
1  if  $x[i] = y[j]$ 
2    then return  $T[i - 1, j - 1] + 1$ 
3    else return  $\max\{T[i - 1, j], T[i, j - 1]\}$ 

```

---

FIGURE 1.49: Recurrence formula for computing an *lcs*.

### 1.6.3 Longest Common Subsequence of Two Strings

A subsequence of a word  $x$  is obtained by deleting zero or more characters from  $x$ . More formally  $w[0..i-1]$  is a subsequence of  $x[0..m-1]$  if there exists an increasing sequence of integers  $(k_j \mid j = 0, \dots, i-1)$  such that for  $0 \leq j \leq i-1$ ,  $w[j] = x[k_j]$ . A word is an  $\text{lcs}(x, y)$  if it is a **longest common subsequence** of the two words  $x$  and  $y$ . Note that two strings can have several longest common subsequence. Their common length is denoted by  $\text{llcs}(x, y)$ .

A brute-force method to compute an  $\text{lcs}(x, y)$  would consist in computing all the subsequences of  $x$ , checking if they are subsequences of  $y$ , and keeping the longest one. The word  $x$  of length  $m$  has  $2^m$  subsequences, and so this method could take  $O(2^m)$  time, which is impractical even for fairly small values of  $m$ .

However  $\text{llcs}(x, y)$  can be computed with a two-dimensional table  $T$  by the following recurrence formula:

$$\begin{aligned}
 T[-1, -1] &= 0, \\
 T[i, -1] &= 0, \\
 T[-1, j] &= 0, \\
 T[i, j] &= \begin{cases} T[i-1, j-1] + 1 & \text{if } x[i] = y[j], \\ \max(T[i-1, j], T[i, j-1]) & \text{otherwise,} \end{cases}
 \end{aligned}$$

for  $i = 0, 1, \dots, m-1$  and  $j = 0, 1, \dots, n-1$ . Then  $T[i, j] = \text{llcs}(x[0..i], y[0..j])$  and  $\text{llcs}(x, y) = T[m-1, n-1]$ .

Computing  $T[m-1, n-1]$  can be done by a call to **GENERIC-DP** with the following parameters  $(x, m, y, n, \text{MARGIN-LOCAL}, \text{FORMULA-LCS})$  in  $O(mn)$  time and space complexity (see Fig. 1.43, 1.47 and 1.49).

It is possible afterward to trace back a path from position  $[m-1, n-1]$  in order to exhibit an  $\text{lcs}(x, y)$  in a similar way as for producing a global alignment (see Fig. 1.46).

**Example 1.15:** The value  $T[4, 8] = 4$  is  $\text{llcs}(x, y)$  for  $x = \mathbf{AGCGA}$  and  $y = \mathbf{CAGATAGAG}$ . String **AGGA** is an  $\text{lcs}$  of  $x$  and  $y$ .

---

```

LLCS( $x, m, y, n$ )
1  for  $i \leftarrow -1$  to  $m - 1$ 
2    do  $C[i] \leftarrow 0$ 
3  for  $j \leftarrow 0$  to  $n - 1$ 
4    do  $last \leftarrow 0$ 
5      for  $i \leftarrow -1$  to  $m - 1$ 
6        do if  $last > C[i]$ 
7          then  $C[i] \leftarrow last$ 
8        elseif  $last < C[i]$ 
9          then  $last \leftarrow C[i]$ 
10       elseif  $x[i] = y[j]$ 
11         then  $C[i] \leftarrow C[i] + 1$ 
12               $last \leftarrow last + 1$ 
13  return  $C$ 

```

---

FIGURE 1.50:  $O(m)$ -space algorithm to compute  $llcs(x, y)$ .

$T$	$j$	-1	0	1	2	3	4	5	6	7	8
$i$		$y[j]$	<b>C</b>	<b>A</b>	<b>G</b>	<b>A</b>	<b>T</b>	<b>A</b>	<b>G</b>	<b>A</b>	<b>G</b>
-1	$x[i]$	0	0	0	0	0	0	0	0	0	0
0	<b>A</b>	0	0	1	1	1	1	1	1	1	1
1	<b>G</b>	0	0	1	2	2	2	2	2	2	2
2	<b>C</b>	0	1	1	2	2	2	2	2	2	2
3	<b>G</b>	0	1	1	2	2	2	2	3	3	3
4	<b>A</b>	0	1	2	2	3	3	3	3	4	4

#### 1.6.4 Reducing the Space: Hirschberg Algorithm

If only the length of an  $lcs(x, y)$  is required, it is easy to see that only one row (or one column) of the table  $T$  needs to be stored during the computation. The space complexity becomes  $O(\min(m, n))$  as can be checked on the algorithm of Fig. 1.50. Indeed, the Hirschberg algorithm computes an  $lcs(x, y)$  in linear space and not only the value  $llcs(x, y)$ . The computation uses the algorithm of Fig. 1.50.

Let us define

$$\begin{aligned}
T^*[i, n] = T^*[m, j] &= 0, \quad \text{for } 0 \leq i \leq m \quad \text{and} \quad 0 \leq j \leq n \\
T^*[m - i, n - j] &= llcs((x[i \dots m - 1])^R, (y[j \dots n - 1])^R) \\
&\quad \text{for } 0 \leq i \leq m - 1 \quad \text{and} \quad 0 \leq j \leq n - 1
\end{aligned}$$

---

```

HIRSCHBERG( $x, m, y, n$ )
1  if  $m = 0$ 
2    then return  $\varepsilon$ 
3  else if  $m = 1$ 
4    then if  $x[0] \in y$ 
5      then return  $x[0]$ 
6    else return  $\varepsilon$ 
7  else  $j \leftarrow \lfloor n/2 \rfloor$ 
8     $C \leftarrow \text{LLCS}(x, m, y[0..j-1], j)$ 
9     $C^* \leftarrow \text{LLCS}(x^R, m, y[j..n-1]^R, n-j)$ 
10    $k \leftarrow m-1$ 
11    $M \leftarrow C[m-1] + C^*[m-1]$ 
12   for  $j \leftarrow -1$  to  $m-2$ 
13     do if  $C[j] + C^*[j] > M$ 
14       then  $M \leftarrow C[j] + C^*[j]$ 
15      $k \leftarrow j$ 
16   return  $\text{HIRSCHBERG}(x[0..k-1], k, y[0..j-1], j) \cdot$ 
       $\text{HIRSCHBERG}(x[k..m-1], m-k,$ 
       $y[j..n-1], n-j)$ 

```

---

FIGURE 1.51:  $O(\min(m, n))$ -space computation of  $\text{lcs}(x, y)$ .

and

$$M(i) = \max_{0 \leq j < n} \{T[i, j] + T^*[m-i, n-j]\}$$

where the word  $w^R$  is the reverse (or mirror image) of the word  $w$ . The following property is the key observation to compute an  $\text{lcs}(x, y)$  in linear space:

$$M(i) = T[m-1, n-1], \quad \text{for } 0 \leq i < m.$$

In the algorithm shown in Fig. 1.51 the integer  $j$  is chosen as  $n/2$ . After  $T[i, j-1]$  and  $T^*[m-i, n-j]$  ( $0 \leq i < m$ ) are computed, the algorithm finds an integer  $k$  such that  $T[i, k] + T^*[m-i, n-k] = T[m-1, n-1]$ . Then, recursively, it computes an  $\text{lcs}(x[0..k-1], y[0..j-1])$  and an  $\text{lcs}(x[k..m-1], y[j..n-1])$ , and concatenates them to get an  $\text{lcs}(x, y)$ .

The running time of the Hirschberg algorithm is still  $O(mn)$  but the amount of space required for the computation becomes  $O(\min(m, n))$  instead of being quadratic when computed by dynamic programming.

---

## 1.7 Approximate String Matching

Approximate string matching is the problem of finding all approximate occurrences of a pattern  $x$  of length  $m$  in a text  $y$  of length  $n$ . Approximate

occurrences of  $x$  are segments of  $y$  that are close to  $x$  according to a specific distance: the distance between segments and  $x$  must be not greater than a given integer  $k$ . Two distances are considered in this section: the **Hamming distance** and the **Levenshtein distance**.

With the Hamming distance, the problem is also known as approximate string matching with  $k$  mismatches. With the Levenshtein distance (or edit distance), the problem is known as approximate string matching with  $k$  differences.

The Hamming distance between two words  $w_1$  and  $w_2$  of the same length is the number of positions with different characters. The Levenshtein distance between two words  $w_1$  and  $w_2$  (not necessarily of the same length) is the minimal number of differences between the two words. A difference is one of the following operations:

- A substitution: a character of  $w_1$  corresponds to a different character in  $w_2$ .
- An insertion: a character of  $w_1$  corresponds to no character in  $w_2$ .
- A deletion: a character of  $w_2$  corresponds to no character in  $w_1$ .

The *Shift-Or algorithm* of the next section is a method that is both very fast in practice and very easy to implement. It solves the Hamming distance and the Levenshtein distance problems. The method for the exact string-matching problem is initially described and then it is presented how to handle the cases of  $k$  mismatches and of  $k$  differences. The method is flexible enough to be adapted to a wide range of similar approximate matching problems.

### 1.7.1 Shift-Or Algorithm

An algorithm to solve the exact string-matching problem is first presented that uses a technique different from those developed in section 1.2, but which extends readily to the approximate string-matching problem.

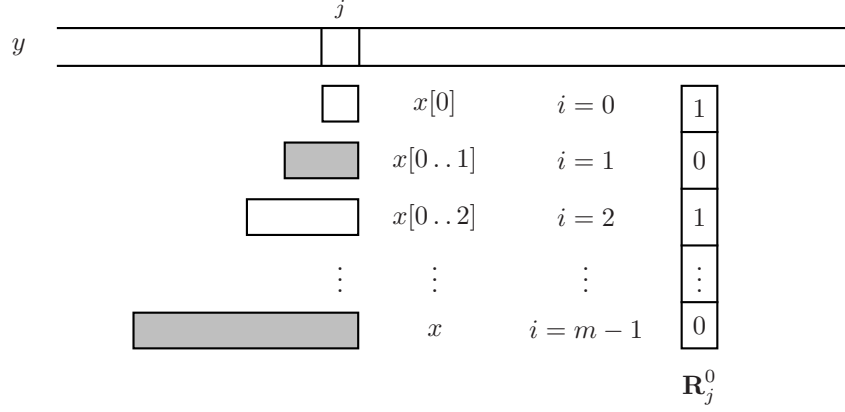
Let  $\mathbf{R}^0$  be a bit array of size  $m$ . Vector  $\mathbf{R}_j^0$  is the value of the entire array  $\mathbf{R}^0$  after text character  $y[j]$  has been processed (see Fig. 1.52). It contains information about all matches of prefixes of  $x$  that end at position  $j$  in the text. It is defined, for  $0 \leq i \leq m-1$ , by

$$\mathbf{R}_j^0[i] = \begin{cases} 0 & \text{if } x[0..i] = y[j-i..j], \\ 1 & \text{otherwise.} \end{cases}$$

Therefore,  $\mathbf{R}_j^0[m-1] = 0$  is equivalent to saying that an (exact) occurrence of the pattern  $x$  ends at position  $j$  in  $y$ .

The vector  $\mathbf{R}_j^0$  can be computed after  $\mathbf{R}_{j-1}^0$  by the following recurrence relation:

$$\mathbf{R}_j^0[i] = \begin{cases} 0 & \text{if } \mathbf{R}_{j-1}^0[i-1] = 0 \text{ and } x[i] = y[j], \\ 1 & \text{otherwise,} \end{cases}$$

FIGURE 1.52: Meaning of vector  $\mathbf{R}_j^0$ .

and

$$\mathbf{R}_j^0[0] = \begin{cases} 0 & \text{if } x[0] = y[j], \\ 1 & \text{otherwise.} \end{cases}$$

The transition from  $\mathbf{R}_{j-1}^0$  to  $\mathbf{R}_j^0$  can be computed very fast as follows. For each  $a \in \Sigma$ , let  $S_a$  be a bit array of size  $m$  defined, for  $0 \leq i \leq m-1$ , by

$$S_a[i] = 0 \quad \text{iff} \quad x[i] = a.$$

The array  $S_a$  denotes the positions of the character  $a$  in the pattern  $x$ . All arrays  $S_a$  are preprocessed before the search starts. And the computation of  $\mathbf{R}_j^0$  reduces to two operations, SHIFT and OR:

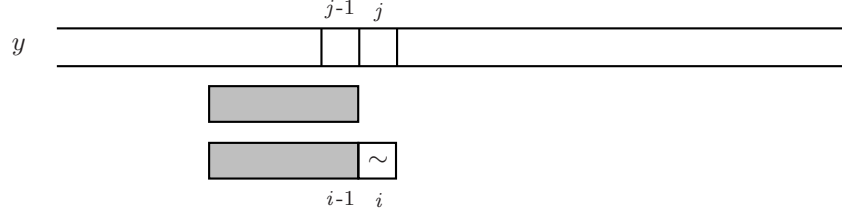
$$\mathbf{R}_j^0 = \text{SHIFT}(\mathbf{R}_{j-1}^0) \quad \text{OR} \quad S_{y[j]}.$$

**Example 1.16:** String  $x = \mathbf{GATAA}$  occurs at position 2 in  $y = \mathbf{CAGATAAGAGAA}$ .

	$S_A$	$S_C$	$S_G$	$S_T$
	1	1	0	1
	0	1	1	1
	1	1	1	0
	0	1	1	1
	0	1	1	1

	C	A	G	A	T	A	A	G	A	G	A	A
G	1	1	0	1	1	1	1	0	1	0	1	1
A	1	1	1	0	1	1	1	1	0	1	0	1
T	1	1	1	1	0	1	1	1	1	1	1	1
A	1	1	1	1	1	0	1	1	1	1	1	1
A	1	1	1	1	1	1	0	1	1	1	1	1

FIGURE 1.53: If  $\mathbf{R}_{j-1}^0[i-1] = 0$  then  $\mathbf{R}_j^1[i] = 0$ .

### 1.7.2 String Matching with $k$ Mismatches

The Shift-Or algorithm easily adapts to support approximate string matching with  $k$  mismatches. To simplify the description, the case where at most one substitution is allowed is presented first.

Arrays  $\mathbf{R}^0$  and  $S$  are used as before, and an additional bit array  $\mathbf{R}^1$  of size  $m$ . Vector  $\mathbf{R}_{j-1}^1$  indicates all matches with at most one substitution up to the text character  $y[j-1]$ . The recurrence on which the computation is based splits into two cases.

- There is an exact match on the first  $i$  characters of  $x$  up to  $y[j-1]$  (i.e.,  $\mathbf{R}_{j-1}^0[i-1] = 0$ ). Then, substituting  $x[i]$  to  $y[j]$  creates a match with one substitution (see Fig. 1.53). Thus,

$$\mathbf{R}_j^1[i] = \mathbf{R}_{j-1}^0[i-1].$$

- There is a match with one substitution on the first  $i$  characters of  $x$  up to  $y[j-1]$  and  $x[i] = y[j]$ . Then, there is a match with one substitution of the first  $i+1$  characters of  $x$  up to  $y[j]$  (see Fig. 1.54). Thus,

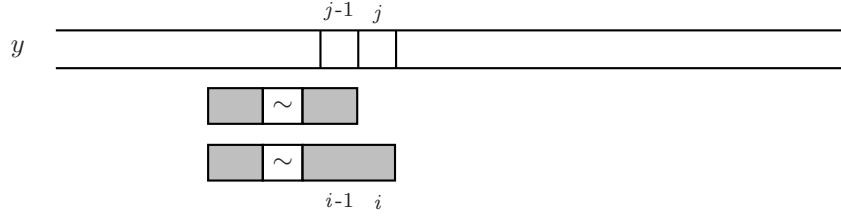
$$\mathbf{R}_j^1[i] = \begin{cases} \mathbf{R}_{j-1}^1[i-1] & \text{if } x[i] = y[j], \\ 1 & \text{otherwise.} \end{cases}$$

This implies that  $\mathbf{R}_j^1$  can be updated from  $\mathbf{R}_{j-1}^1$  by the relation:

$$\mathbf{R}_j^1 = (\text{SHIFT}(\mathbf{R}_{j-1}^1) \quad \text{OR} \quad S_{y[j]}) \quad \text{AND} \quad \text{SHIFT}(\mathbf{R}_{j-1}^0).$$

**Example 1.17:** String  $x = \mathbf{GATAA}$  occurs at positions 2 and 7 in  $y = \mathbf{CAGATAAGAGAA}$  with no more than one mismatch.

	C	A	G	A	T	A	A	G	A	G	A	A
G	0	0	0	0	0	0	0	0	0	0	0	0
A	1	0	1	0	1	0	0	1	0	1	0	0
T	1	1	1	1	0	1	1	1	1	0	1	0
A	1	1	1	1	1	0	1	1	1	1	0	1
A	1	1	1	1	1	1	0	1	1	1	1	0

FIGURE 1.54:  $\mathbf{R}_j^1[i] = \mathbf{R}_{j-1}^1[i-1]$  if  $x[i] = y[j]$ .

### 1.7.3 String Matching with $k$ Differences

In this section it is shown how to adapt the Shift-Or algorithm to the case of only one insertion, and then dually to the case of only one deletion. The method is based on the following elements.

One insertion is allowed: here, vector  $\mathbf{R}_{j-1}^1$  indicates all matches with at most one insertion up to text character  $y[j-1]$ .  $\mathbf{R}_{j-1}^1[i-1] = 0$  if the first  $i$  characters of  $x$  ( $x[0..i-1]$ ) match  $i$  symbols of the last  $i+1$  text characters up to  $y[j-1]$ . Array  $\mathbf{R}^0$  is maintained as before, and array  $\mathbf{R}^1$  is maintained as follows. Two cases arise.

- There is an exact match on the first  $i+1$  characters of  $x$  ( $x[0..i]$ ) up to  $y[j-1]$ . Then inserting  $y[j]$  creates a match with one insertion up to  $y[j]$  (see Fig. 1.55). Thus,

$$\mathbf{R}_j^1[i] = \mathbf{R}_{j-1}^0[i].$$

- There is a match with one insertion on the  $i$  first characters of  $x$  up to  $y[j-1]$ . Then if  $x[i] = y[j]$  there is a match with one insertion on the first  $i+1$  characters of  $x$  up to  $y[j]$  (see Fig. 1.55). Thus,

$$\mathbf{R}_j^1[i] = \begin{cases} \mathbf{R}_{j-1}^1[i-1] & \text{if } x[i] = y[j], \\ 1 & \text{otherwise.} \end{cases}$$

This shows that  $\mathbf{R}_j^1$  can be updated from  $\mathbf{R}_{j-1}^1$  with the formula

$$\mathbf{R}_j^1 = (\text{SHIFT}(\mathbf{R}_{j-1}^1) \text{ OR } S_{y[j]}) \text{ AND } \mathbf{R}_{j-1}^0.$$

**Example 1.18:** Here **GATAAG** is an occurrence of  $x = \mathbf{GATAA}$  with exactly one insertion in  $y = \mathbf{CAGATAAGAGAA}$

	C	A	G	A	T	A	A	G	A	G	A	A
G	1	1	1	0	1	1	1	1	0	1	0	1
A	1	1	1	1	0	1	1	1	1	0	1	0
T	1	1	1	1	1	0	1	1	1	1	1	1
A	1	1	1	1	1	1	0	1	1	1	1	1
A	1	1	1	1	1	1	1	0	1	1	1	1

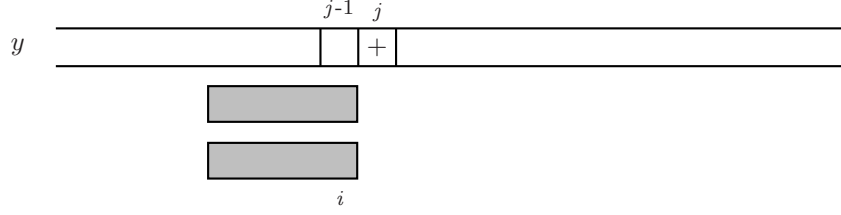


FIGURE 1.55: If  $\mathbf{R}_{j-1}^0[i] = 0$  then  $\mathbf{R}_j^1[i] = 0$ .

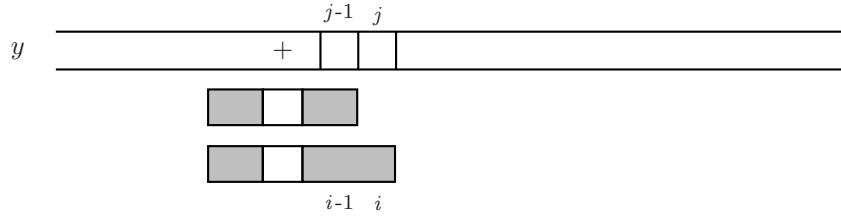


FIGURE 1.56:  $\mathbf{R}_j^1[i] = \mathbf{R}_{j-1}^1[i-1]$  if  $x[i] = y[j]$ .

One deletion is allowed: assume here that  $\mathbf{R}_{j-1}^1$  indicates all possible matches with at most one deletion up to  $y[j-1]$ . As in the previous solution, two cases arise.

- There is an exact match on the first  $i+1$  characters of  $x$  ( $x[0..i]$ ) up to  $y[j]$  (i.e.,  $\mathbf{R}_j^0[i] = 0$ ). Then, deleting  $x[i]$  creates a match with one deletion (see Fig. 1.57). Thus,

$$\mathbf{R}_j^1[i] = \mathbf{R}_j^0[i].$$

- There is a match with one deletion on the first  $i$  characters of  $x$  up to  $y[j-1]$  and  $x[i] = y[j]$ . Then, there is a match with one deletion on the

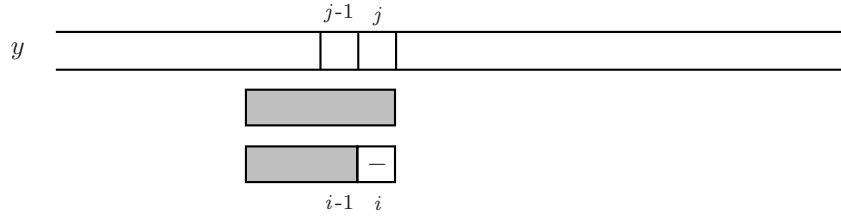
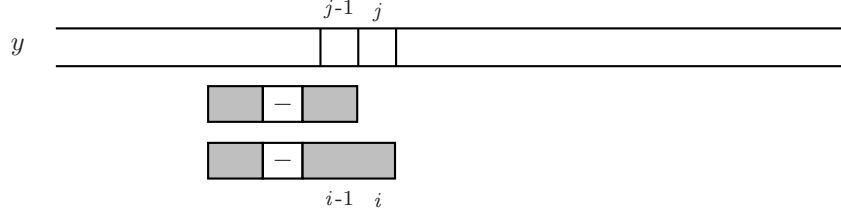


FIGURE 1.57: If  $\mathbf{R}_j^0[i] = 0$  then  $\mathbf{R}_j^1[i] = 0$ .

FIGURE 1.58:  $\mathbf{R}_j^1[i] = \mathbf{R}_{j-1}^1[i-1]$  if  $x[i] = y[j]$ .

first  $i + 1$  characters of  $x$  up to  $y[j]$  (see Fig. 1.57). Thus,

$$\mathbf{R}_j^1[i] = \begin{cases} \mathbf{R}_{j-1}^1[i-1] & \text{if } x[i] = y[j], \\ 1 & \text{otherwise.} \end{cases}$$

The discussion provides the following formula used to update  $\mathbf{R}_j^1$  from  $\mathbf{R}_{j-1}^1$ :

$$\mathbf{R}_j^1 = (\text{SHIFT}(\mathbf{R}_{j-1}^1) \text{ OR } S_{y[j]}) \text{ AND } \text{SHIFT}(\mathbf{R}_j^0).$$

**Example 1.19:** **GATA** and **ATAA** are two occurrences with one deletion of  $x = \mathbf{GATAA}$  in  $y = \mathbf{CAGATAAGAGAA}$

	C	A	G	A	T	A	A	G	A	G	A	A
G	0	0	0	0	0	0	0	0	0	0	0	0
A	1	0	0	0	1	0	0	0	0	0	0	0
T	1	1	1	0	0	1	1	1	0	1	0	1
A	1	1	1	1	0	0	1	1	1	1	1	0
A	1	1	1	1	1	0	0	1	1	1	1	1

#### 1.7.4 Wu–Manber Algorithm

In this section, a general solution for the approximate string-matching problem with at most  $k$  differences of the types: insertion, deletion, and substitution is presented. It is an extension of the problems presented above. The following algorithm maintains  $k + 1$  bit arrays  $\mathbf{R}^0, \mathbf{R}^1, \dots, \mathbf{R}^k$  that are described now. The vector  $\mathbf{R}^0$  is maintained similarly as in the exact matching case (section “Shift-Or Algorithm”). The other vectors are computed with the formula ( $1 \leq \ell \leq k$ )

$$\begin{aligned} \mathbf{R}_j^\ell = & (\text{SHIFT}(\mathbf{R}_{j-1}^\ell) \text{ OR } S_{y[j]}) \\ & \text{AND } \text{SHIFT}(\mathbf{R}_j^{\ell-1}) \\ & \text{AND } \text{SHIFT}(\mathbf{R}_{j-1}^{\ell-1}) \\ & \text{AND } \mathbf{R}_{j-1}^{\ell-1} \end{aligned}$$

---

```

WM( $x, m, y, n, k$ )
1  for each character  $a \in \Sigma$ 
2    do  $S_a \leftarrow 1^m$ 
3  for  $i \leftarrow 0$  to  $m - 1$ 
4    do  $S_{x[i]}[i] \leftarrow 0$ 
5   $\mathbf{R}^0 \leftarrow 1^m$ 
6  for  $\ell \leftarrow 1$  to  $k$ 
7    do  $\mathbf{R}^\ell \leftarrow \text{SHIFT}(\mathbf{R}^{\ell-1})$ 
8  for  $j \leftarrow 0$  to  $n - 1$ 
9    do  $T \leftarrow \mathbf{R}^0$ 
10      $\mathbf{R}^0 \leftarrow \text{SHIFT}(\mathbf{R}^0) \text{ OR } S_{y[j]}$ 
11     for  $\ell \leftarrow 1$  to  $k$ 
12       do  $T' \leftarrow \mathbf{R}^\ell$ 
13        $\mathbf{R}^\ell \leftarrow (\text{SHIFT}(\mathbf{R}^\ell) \text{ OR } S_{y[j]}) \text{ AND}$ 
14          $(\text{SHIFT}((T \text{ AND } \mathbf{R}^{\ell-1})) \text{ AND } T$ 
15          $T \leftarrow T'$ 
16     if  $\mathbf{R}^k[m - 1] = 0$ 
17       then OUTPUT( $j$ )

```

---

FIGURE 1.59: Wu–Manber approximate string-matching algorithm.

which can be rewritten into

$$\begin{aligned} \mathbf{R}_j^\ell &= (\text{SHIFT}(\mathbf{R}_{j-1}^\ell) \text{ OR } S_{y[j]}) \\ &\quad \text{AND } \text{SHIFT}(\mathbf{R}_j^{\ell-1} \text{ AND } \mathbf{R}_{j-1}^{\ell-1}) \\ &\quad \text{AND } \mathbf{R}_{j-1}^{\ell-1}. \end{aligned}$$

**Example 1.20:** Here  $x = \mathbf{GATAA}$  and  $y = \mathbf{CAGATAAGAGAA}$  and  $k = 1$ . The output 5, 6, 7, and 11 corresponds to the segments **GATA**, **GATAA**, **GATAAG**, and **GAGAA** which approximate the pattern **GATAA** with no more than one difference.

	C	A	G	A	T	A	A	G	A	G	A	A
G	0	0	0	0	0	0	0	0	0	0	0	0
A	1	0	0	0	0	0	0	0	0	0	0	0
T	1	1	1	0	0	0	1	1	0	0	0	0
A	1	1	1	1	0	0	0	1	1	1	0	0
A	1	1	1	1	1	0	0	0	1	1	1	0

The method, called the Wu–Manber algorithm, is implemented in Fig. 1.59. It assumes that the length of the pattern is no more than the size of the memory word of the machine, which is often the case in applications.

The preprocessing phase of the algorithm takes  $O(\sigma m + km)$  memory space, and runs in time  $O(\sigma m + k)$ . The time complexity of its searching phase is  $O(kn)$ .

---

## 1.8 Text Compression

This section is interested in algorithms that compress texts. Compression serves both to save storage space and to save transmission time. Assume that the uncompressed text is stored in a file. The aim of compression algorithms is to produce another file containing the compressed version of the same text. Methods in this section work with no loss of information, so that decompressing the compressed text restores exactly the original text.

Two main strategies to design the algorithms are applied. The first strategy is a statistical method that takes into account the frequencies of symbols to build a uniquely decipherable code optimal with respect to the compression. The code contains new codewords for the symbols occurring in the text. In this method fixed-length blocks of bits are encoded by different codewords. *A contrario* the second strategy encodes variable-length segments of the text. To put it simply, the algorithm, while scanning the text, replaces some already read segments just by a pointer to their first occurrences.

Text compression software often use a mixture of several methods. An example of that is given in Section 1.8.3 which contains in particular two classical simple compression algorithms. They compress efficiently only a small variety of texts when used alone. But they become more powerful with the special preprocessing presented there.

### 1.8.1 Huffman Coding

The Huffman method is an optimal statistical coding. It transforms the original code used for characters of the text (ASCII code on 8 b, for instance). Coding the text is just replacing each symbol (more exactly each occurrence of it) by its new codeword. The method works for any length of blocks (not only 8 b), but the running time grows exponentially with the length. In the following, one assumes that symbols are originally encoded on 8 b to simplify the description.

The Huffman algorithm uses the notion of **prefix code**. A prefix code is a set of words containing no word that is a prefix of another word of the set. The advantage of such a code is that decoding is immediate. Moreover, it can be proved that this type of code does not weaken the compression.

A prefix code on the binary alphabet  $\{0, 1\}$  can be represented by a trie (see section on the Aho–Corasick algorithm) that is a binary tree. In the present method codes are complete: they correspond to complete tries (internal nodes have exactly two children). The leaves are labeled by the original characters, edges are labeled by 0 or 1, and labels of branches are the words of the code. The condition on the code implies that codewords are identified with leaves only. The convention is adopted that, from an internal node, the edge to its left child is labeled by 0, and the edge to its right child is labeled by 1.

---

```

COUNT(fin)
1  for each character  $a \in \Sigma$ 
2      do  $freq(a) \leftarrow 0$ 
3  while not end of file fin and  $a$  is the next symbol
4      do  $freq(a) \leftarrow freq(a) + 1$ 
5  END  $\leftarrow 1$ 

```

---

FIGURE 1.60: Counts the character frequencies.

In the model where characters of the text are given new codewords, the Huffman algorithm builds a code that is optimal in the sense that the compression is the best possible (the length of the compressed text is minimum). The code depends on the text, and more precisely on the frequencies of each character in the uncompressed text. The more frequent characters are given short codewords, whereas the less frequent symbols have longer codewords.

#### 1.8.1.1 Encoding

The coding algorithm is composed of three steps: count of character frequencies, construction of the prefix code, and encoding of the text.

The first step consists in counting the number of occurrences of each character in the original text (see Fig. 1.60). A special end marker (denoted by END) is used, which (virtually) appears only once at the end of the text. It is possible to skip this first step if fixed statistics on the alphabet are used. In this case the method is optimal according to the statistics, but not necessarily for the specific text.

The second step of the algorithm builds the tree of a prefix code using the character frequency  $freq(a)$  of each character  $a$  in the following way:

- create a one-node tree  $t$  for each character  $a$ , setting  $weight(t) = freq(a)$  and  $label(t) = a$ ,
- repeat (1), extract the two least weighted trees  $t_1$  and  $t_2$ , and (2) create a new tree  $t_3$  having left subtree  $t_1$ , right subtree  $t_2$ , and weight  $weight(t_3) = weight(t_1) + weight(t_2)$ ,
- until only one tree remains.

The tree is constructed by the algorithm BUILD-TREE in Fig. 1.61. The implementation uses two linear lists. The first list contains the leaves of the future tree, each associated with a symbol. The list is sorted in the increasing order of the weight of the leaves (frequency of symbols). The second list contains the newly created trees. Extracting the two least weighted trees consists in extracting the two least weighted trees among the two first trees of the list of leaves and the two first trees of the list of created trees. Each new tree is inserted at the end of the list of the trees. The only tree remaining at the end of the procedure is the coding tree.

---

```

BUILD-TREE()
1  for each character  $a \in \Sigma \cup \{\text{END}\}$ 
2      do if  $\text{freq}(a) \neq 0$ 
3          then create a new node  $t$ 
4               $\text{weight}(t) \leftarrow \text{freq}(a)$ 
5               $\text{label}(t) \leftarrow a$ 
6   $l\text{leaves} \leftarrow$  list of all the nodes in increasing order of weight
7   $l\text{trees} \leftarrow$  empty list
8  while  $\text{LENGTH}(l\text{leaves}) + \text{LENGTH}(l\text{trees}) > 1$ 
9      do  $(\ell, r) \leftarrow$  extract the two nodes of smallest weight (among the
          two nodes at the beginning of  $l\text{leaves}$  and the two nodes at the beginning of  $l\text{trees}$ )
10         create a new node  $t$ 
11          $\text{weight}(t) \leftarrow \text{weight}(\ell) + \text{weight}(r)$ 
12          $\text{left}(t) \leftarrow \ell$ 
13          $\text{right}(t) \leftarrow r$ 
14         insert  $t$  at the end of  $l\text{trees}$ 
15 return  $t$ 

```

---

FIGURE 1.61: Builds the coding tree.

---

```

BUILD-CODE( $t, \text{length}$ )
1  if  $t$  is not a leaf
2      then  $\text{temp}[\text{length}] \leftarrow 0$ 
3          BUILD-CODE( $\text{left}(t), \text{length} + 1$ )
4           $\text{temp}[\text{length}] \leftarrow 1$ 
5          BUILD-CODE( $\text{right}(t), \text{length} + 1$ )
6  else  $\text{codeword}(\text{label}(t)) \leftarrow \text{temp}[0.. \text{length} - 1]$ 

```

---

FIGURE 1.62: Builds the character codes from the coding tree.

After the coding tree is built, it is possible to recover the codewords associated with characters by a simple depth-first search of the tree (see Fig. 1.62;  $\text{codeword}(a)$  is then the binary code associated with the character  $a$ ).

In the third step, the original text is encoded. Since the code depends on the original text, in order to be able to decode the compressed text, the coding tree and the original codewords of symbols must be stored with the compressed text. This information is placed in a header of the compressed file, to be read at decoding time just before the compressed text. The header is made via a depth-first traversal of the tree. Each time an internal node is encountered a 0 is produced. When a leaf is encountered a 1 is produced followed by the original code of the corresponding character on 9 b (so that the end marker can be equal to 256 if all the characters appear in the original text). This part of the encoding algorithm is shown in Fig. 1.63.

After the header of the compressed file is computed, the encoding of the original text is realized by the algorithm of Fig. 1.64.

---

```

CODE-TREE(fout, t)
1  if t is not a leaf
2    then write a 0 in the file fout
3        CODE-TREE(fout, left(t))
4        CODE-TREE(fout, right(t))
5    else write a 1 in the file fout
6        write the original code of label(t) in the file fout

```

---

FIGURE 1.63: Memorizes the coding tree in the compressed file.

---

```

CODE-TEXT(fin, fout)
1  while not end of file fin and a is the next symbol
2    do write codeword(a) in the file fout
3  write codeword(END) in the file fout

```

---

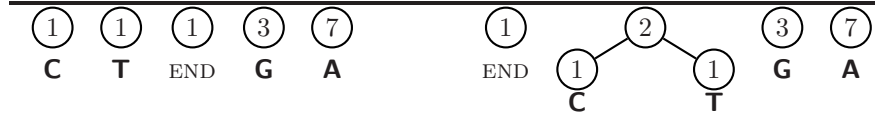
FIGURE 1.64: Encodes the characters in the compressed file.

A complete implementation of the Huffman algorithm, composed of the three steps just described, is given in Fig. 1.65.

**Example 1.21:** Here  $y = \text{CAGATAAGAGAA}$ . The length of  $y = 12 \times 8 = 96$  b (assuming an 8-b code). The character frequencies are

A	C	G	T	END
7	1	3	1	1

The different steps during the construction of the coding tree are




---

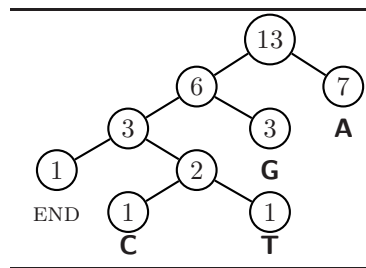
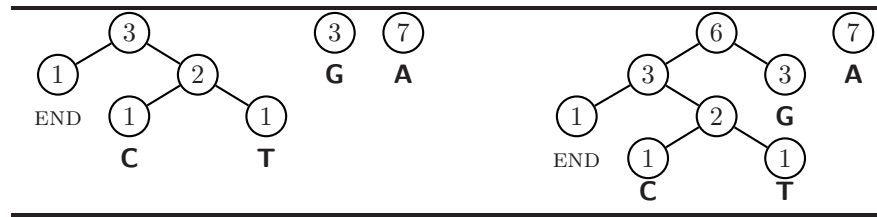
```

CODING(fin, fout)
1  COUNT(fin)
2   $t \leftarrow \text{BUILD-TREE}()$ 
3  BUILD-CODE(t, 0)
4  CODE-TREE(fout, t)
5  CODE-TEXT(fin, fout)

```

---

FIGURE 1.65: Complete function for Huffman coding.



A	C	G	T	END
1	0010	01	0011	000

character codewords:

The encoded tree is **0001** binary (END, 9)**01**binary (**C**, 9)**1**binary(**T**, 9)  
**1**binary (**G**, 9)**1**binary (**A**, 9), which produces a header of length 54 b,

0001 100000000 01 001000011 1 001010100 1 001000111 1 001000001

The encoded text

0010 1 01 1 0011 1 1 01 1 01 1 1 000

is of length 24 b. The total length of the compressed file is 78 b.

The construction of the tree takes  $O(\sigma \log \sigma)$  time if the sorting of the list of the leaves is implemented efficiently. The rest of the encoding process runs in linear time in the sum of the sizes of the original and compressed texts.

### 1.8.1.2 Decoding

Decoding a file containing a text compressed by the Huffman algorithm is a mere programming exercise. First, the coding tree is rebuilt by the algorithm of Fig. 1.66. Then, the uncompressed text is recovered by parsing the compressed text with the coding tree. The process begins at the root of the coding tree and follows a left edge when a 0 is read or a right edge when a 1 is read. When a leaf is encountered, the corresponding character (in fact the original codeword of it) is produced and the parsing phase resumes at the root of the tree. The parsing ends when the codeword of the end marker is read. An implementation of the decoding of the text is presented in Fig. 1.67.

The complete decoding program is given in Fig. 1.68. It calls the preceding

---

```

REBUILD-TREE(fin, t)
1  b ← read a bit from the file fin
2  if b = 1                                     ▷ leaf
3      then left(t) ← NIL
4             right(t) ← NIL
5             label(t) ← symbol corresponding to
                        the 9 next bits in the file fin
6  else create a new node ℓ
7             left(t) ← ℓ
8             REBUILD-TREE(fin, ℓ)
9             create a new node r
10            right(t) ← r
11            REBUILD-TREE(fin, r)

```

---

FIGURE 1.66: Rebuilds the tree read from the compressed file.

---

```

DECODE-TEXT(fin, fout, root)
1  t ← root
2  while label(t) ≠ END
3      do if t is a leaf
4          then label(t) in the file fout
5              t ← root
6          else b ← read a bit from the file fin
7              if b = 1
8                  then t ← right(t)
9                  else t ← left(t)

```

---

FIGURE 1.67: Reads the compressed text and produces the uncompressed text.

---

```

DECODING(fin, fout)
1  create a new node root
2  REBUILD-TREE(fin, root)
3  DECODE-TEXT(fin, fout, root)

```

---

FIGURE 1.68: Complete function for Huffman decoding.

functions. The running time of the decoding program is linear in the sum of the sizes of the texts it manipulates.

### 1.8.2 Lempel–Ziv–Welsh (LZW) Compression

Ziv and Lempel designed a compression method using encoding segments. These segments are stored in a dictionary that is built during the compression process. When a segment of the dictionary is encountered later while scanning the original text it is substituted by its index in the dictionary. In the model where portions of the text are replaced by pointers on previous occurrences, the Ziv–Lempel compression scheme can be proved to be asymptotically optimal (on large enough texts satisfying good conditions on the probability distribution of symbols).

The dictionary is the central point of the algorithm. It has the property of being prefix closed (every prefix of a word of the dictionary is in the dictionary), so that it can be implemented as a tree. Furthermore, a hashing technique makes its implementation efficient. The version described in this section is called the Lempel–Ziv–Welsh method after several improvements introduced by Welsh. The algorithm is implemented by the **compress** command existing under the UNIX operating system.

#### 1.8.2.1 Compression Method

The scheme of the compression method is first described. The dictionary is initialized with all the characters of the alphabet. The current situation is when one has just read a segment  $w$  in the text. Let  $a$  be the next symbol (just following  $w$ ). Then the method proceeds as follows:

- If  $wa$  is not in the dictionary, write the index of  $w$  to the output file, and add  $wa$  to the dictionary. Then  $w$  is reset to  $a$  and process the next symbol (following  $a$ ).
- If  $wa$  is in the dictionary, process the next symbol, with segment  $wa$  instead of  $w$ .

Initially, the segment  $w$  is set to the first symbol of the source text.

**Example 1.22:** Here  $y = \mathbf{CAGTAAGAGAA}$

C	A	G	T	A	A	G	A	G	A	A	<i>w</i>	out	added
	↑										<b>C</b>	67	<b>CA</b> , 257
		↑									<b>A</b>	65	<b>AG</b> , 258
			↑								<b>G</b>	71	<b>GT</b> , 259
				↑							<b>T</b>	84	<b>TA</b> , 260
					↑						<b>A</b>	65	<b>AA</b> , 261
						↑					<b>A</b>		
							↑				<b>AG</b>	258	<b>AGA</b> , 262
								↑			<b>A</b>		
									↑		<b>AG</b>		
										↑	<b>AGA</b>	262	<b>AGAA</b> , 262
											<b>A</b>		
												65	
												256	

### 1.8.2.2 Decompression Method

The decompression method is symmetrical to the compression algorithm. The dictionary is recovered while the decompression process runs. It is basically done in this way:

- Read a code  $c$  in the compressed file.
- Write in the output file the segment  $w$  which has index  $c$  in the dictionary.
- Add to the dictionary the word  $wa$  where  $a$  is the first letter of the next segment.

In this scheme, a problem occurs if the next segment is the word which is being built. This arises only if the text contains a segment  $azax$  for which  $az$  belongs to the dictionary but  $aza$  does not. During the compression process the index of  $az$  is written into the compressed file, and  $aza$  is added to the dictionary. Next,  $aza$  is read and its index is written into the file. During the decompression process the index of  $aza$  is read while the word  $az$  has not been completed yet: the segment  $aza$  is not already in the dictionary. However, since this is the unique case where the situation arises, the segment  $aza$  is recovered taking the last segment  $az$  added to the dictionary concatenated with its first letter  $a$ .

**Example 1.23:** Here the decoding is 67, 65, 71, 84, 65, 258, 262, 65, 256

---

```

COMPRESS(fin, fout)
1  count  $\leftarrow$  -1
2  for each character a  $\in$   $\Sigma$ 
3      do count  $\leftarrow$  count + 1
4          HASH-INSERT(D, (-1, a, count))
5  count  $\leftarrow$  count + 1
6  HASH-INSERT(D, (-1, END, count))
7  p  $\leftarrow$  -1
8  while not end of file fin
9      do a  $\leftarrow$  next character of fin
10         q  $\leftarrow$  HASH-SEARCH(D, (p, a))
11         if q = NIL
12             then write code(p) on 1 + log(count) bits in fout
13                 count  $\leftarrow$  count + 1
14                 HASH-INSERT(D, (p, a, count))
15                 p  $\leftarrow$  HASH-SEARCH(D, (-1, a))
16         else p  $\leftarrow$  q
17 write code(p) on 1 + log(count) bits in fout
18 write code(HASH-SEARCH(D, (-1, END))) on 1 + log(count) bits in fout

```

---

FIGURE 1.69: LZW compression algorithm.

read	written	added
67	<b>C</b>	
65	<b>A</b>	<b>CA</b> , 257
71	<b>G</b>	<b>AG</b> , 258
84	<b>T</b>	<b>GT</b> , 259
65	<b>A</b>	<b>TA</b> , 260
258	<b>AG</b>	<b>AA</b> , 261
262	<b>AGA</b>	<b>AGA</b> , 262
65	<b>A</b>	<b>AGAA</b> , 263
256		

### 1.8.2.3 Implementation

For the compression algorithm shown in Fig. 1.69, the dictionary is stored in a table *D*. The dictionary is implemented as a tree; each node *z* of the tree has the three following components:

- *parent*(*z*) is a link to the parent node of *z*.
- *label*(*z*) is a character.
- *code*(*z*) is the code associated with *z*.

---

```

UNCOMPRESS(fin, fout)
1  count  $\leftarrow$  -1
2  for each character a  $\in$   $\Sigma$ 
3      do count  $\leftarrow$  count + 1
4          HASH-INSERT(D, (-1, a, count))
5  count  $\leftarrow$  count + 1
6  HASH-INSERT(D, (-1, END, count))
7  c  $\leftarrow$  first code on  $1 + \log(\text{count})$  bits in fin
8  write string(c) in fout
9  a  $\leftarrow$  first(string(c))
10 while TRUE
11     do d  $\leftarrow$  next code on  $1 + \log(\text{count})$  bits in fin
12         if d > count
13             then count  $\leftarrow$  count + 1
14                 parent(count)  $\leftarrow$  c
15                 label(count)  $\leftarrow$  a
16                 write string(c)a in fout
17                 c  $\leftarrow$  d
18             else a  $\leftarrow$  first(string(d))
19                 if a  $\neq$  END
20                     then count  $\leftarrow$  count + 1
21                         parent(count)  $\leftarrow$  c
22                         label(count)  $\leftarrow$  a
23                         write string(d) in fout
24                         c  $\leftarrow$  d
25                     else break

```

---

FIGURE 1.70: LZW decompression algorithm.

The tree is stored in a table that is accessed with a hashing function. This provides fast access to the children of a node. The procedure HASH-INSERT( $(D, (p, a, c))$ ) inserts a new node  $z$  in the dictionary  $D$  with  $\text{parent}(z) = p$ ,  $\text{label}(z) = a$ , and  $\text{code}(z) = c$ . The function HASH-SEARCH( $(D, (p, a))$ ) returns the node  $z$  such that  $\text{parent}(z) = p$  and  $\text{label}(z) = a$ .

For the decompression algorithm, no hashing technique is necessary. Having the index of the next segment, a bottom-up walk in the trie implementing the dictionary produces the mirror image of the segment. A stack is used to reverse it. Assume that the function *string*(*c*) performs this specific work for a code *c*. The bottom-up walk follows the parent links of the data structure. The function *first*(*w*) gives the first character of the word *w*. These features are part of the decompression algorithm displayed in Fig. 1.70.

The Ziv–Lempel compression and decompression algorithms run both in linear time in the sizes of the files provided a good hashing technique is chosen. Indeed, it is very fast in practice. Its main advantage compared to Huffman coding is that it captures long repeated segments in the source file.

### 1.8.3 Mixing several methods

Simple compression methods are described and then an example of a combination of several of them, basis of the popular **bzip** software.

#### 1.8.3.1 Run Length Encoding

The aim of Run Length Encoding (RLE) is to efficiently encode repetitions occurring in the input data. Let us assume that it contains a good quantity of repetitions of the form  $aa \dots a$  for some character  $a$  ( $a \in \Sigma$ ). A repetition of  $k$  consecutive occurrences of letter  $a$  is replaced by  $\&ak$ , where the symbol  $\&$  is a new character ( $\& \notin \Sigma$ ).

The string  $\&ak$  that encodes a repetition of  $k$  consecutive occurrences of  $a$  is itself encoded on the binary alphabet  $\{0, 1\}$ . In practice, letters are often represented by their ASCII code. Therefore, the codeword of a letter belongs to  $\{0, 1\}^k$  with  $k = 7$  or  $8$ . Generally there is no problem in choosing or encoding the special character  $\&$ . The integer  $k$  of the string  $\&ak$  is also encoded on the binary alphabet, but it is not sufficient to translate it by its binary representation, because one would be unable to recover it at decoding time inside the stream of bits. A simple way to cope with this is to encode  $k$  by the string  $0^\ell \text{bin}(k)$ , where  $\text{bin}(k)$  is the binary representation of  $k$ , and  $\ell$  is the length of it. This works well because the binary representation of  $k$  starts with a 1 so there is no ambiguity to recover  $\ell$  by counting during the decoding phase. The size of the encoding of  $k$  is thus roughly  $2 \log k$ . More sophisticated integer representations are possible, but none is really suitable for the present situation. Simpler solution consists in encoding  $k$  on the same number of bits as other symbols, but this bounds values of  $\ell$  and decreases the power of the method.

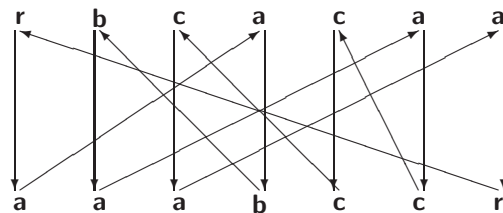
#### 1.8.3.2 Move To Front

The Move To Front (MTF) method may be regarded as an extension of Run Length Encoding or a simplification of Ziv–Lempel compression. It is efficient when the occurrences of letters in the input text are localized into relatively short segment of it. The technique is able to capture the proximity between occurrences of symbols and to turn it into a short encoded text.

Letters of the alphabet  $\Sigma$  of the input text are initially stored in a list that is managed dynamically. Letters are represented by their rank in the list, starting from 1, rank that is itself encoded as described above for RLE.

Letters of the input text are processed in an on-line manner. The clue of the method is that each letter is moved to the beginning of the list just after it is translated by the encoding of its rank.

The effect of MTF is to reduce the size of the encoding of a letter that reappears soon after its preceding occurrence.

FIGURE 1.71:  $BW(\mathbf{baccara})$ .

### 1.8.3.3 Integrated example

Most compression software combine several methods to be able to compress efficiently a large range of input data. An example of this strategy, implemented by the UNIX command **bzip**, is presented.

Let  $y = y[0]y[1] \cdots y[n-1]$  be the input text. The  $k$ -th rotation (or conjugate) of  $y$ ,  $0 \leq k \leq n-1$ , is the string  $y_k = y[k]y[k+1] \cdots y[n-1]y[0]y[1] \cdots y[k-1]$ .

The  $BW$  transformation is defined as  $BW(y) = y[p_0]y[p_1] \cdots y[p_{n-1}]$ , where  $p_i + 1$  is such that  $y_{p_i+1}$  has rank  $i$  in the sorted list of all rotations of  $y$ .

It is remarkable that  $y$  can be recovered from both  $BW(y)$  and a position on it, starting position of the inverse transformation (see Figure 1.71). This is possible due to the following property of the transformation. Assume that  $i < j$  and  $y[p_i] = y[p_j] = a$ . Since  $i < j$ , the definition implies  $y_{p_i+1} < y_{p_j+1}$ . Since  $y[p_i] = y[p_j]$ , transferring the last letters of  $y_{p_i+1}$  and  $y_{p_j+1}$  to the beginning of these words does not change the inequality. This proves that the two occurrences of  $a$  in  $BW(y)$  are in the same relative order as in the sorted list of letters of  $y$ . Figure 1.71 illustrates the inverse transformation. Top line is  $BW(y)$  and bottom line the sorted list of letters of it. Top-down arrows correspond to succession of occurrences in  $y$ . Each bottom-up arrow links the same occurrence of a letter in  $y$ . Arrows starting from equal letters do not cross. The circular path is associated with rotations of the string  $y$ . If the starting point is known, the only occurrence of letter **b** here, following the path produces the initial string  $y$ .

Transformation  $BW$  obviously does not compress the input text  $y$ . But  $BW(y)$  is compressed more efficiently with simple methods. This is the strategy applied for the command **bzip**. It is a combination of the  $BW$  transformation followed by MTF encoding and RLE encoding. Arithmetic coding, a method providing compression ratios slightly better than Huffman coding, may also be used.

Table 1.1 contains a sample of experimental results showing the behavior of compression algorithms on different types of texts from the Calgary Corpus: **bib** (bibliography), **book1** (fiction book), **news** (USENET batch file), **pic**

bytes	111,261	768,771	377,109	513,216	39,611	93,695	
Texts	<b>bib</b>	<b>book1</b>	<b>news</b>	<b>pic</b>	<b>progc</b>	<b>trans</b>	Average
<b>pack</b>	5.24	4.56	5.23	1.66	5.26	5.58	4.99
<b>gzip-b</b>	2.51	3.25	3.06	0.82	2.68	1.61	2.69
<b>bzip2-1</b>	2.10	2.81	2.85	0.78	2.53	1.53	2.46

**TABLE 1.1:** Compression results with three algorithms: Huffman coding (**pack**), Ziv–Lempel coding (**gzip-b**) and Burrows–Wheeler coding (**bzip2-1**).

(black and white fax picture), **progc** (source code in C) and **trans** (transcript of terminal session).

The compression algorithms reported in the table are: the Huffman coding algorithm implemented by **pack**, the Ziv–Lempel algorithm implemented by **gzip-b** and the compression based on the *BW* transform implemented by **bzip2-1**. Figures give the number of bits used per character (letter). They show that **pack** is the less efficient method and that **bzip2-1** compresses a bit more than **gzip-b**. Additional compression results can be found at <http://corpus.canterbury.ac.nz>.

## 1.9 Research Issues and Summary

The algorithm for string searching by hashing was introduced by Harrison in 1971, and later fully analyzed by Karp and Rabin (1987).

The linear-time string-matching algorithm of Knuth, Morris, and Pratt is from 1976. It can be proved that, during the search, a character of the text is compared to a character of the pattern no more than  $\log_{\Phi}(|x| + 1)$  (where  $\Phi$  is the golden ratio  $(1 + \sqrt{5})/2$ ). Simon (1993) gives an algorithm similar to the previous one but with a delay bounded by the size of the alphabet (of the pattern  $x$ ). Hancart (1993) proves that the delay of Simon’s algorithm is, indeed, no more than  $1 + \log_2 |x|$ . He also proves that this is optimal among algorithms searching the text through a window of size 1.

Galil (1981) gives a general criterion to transform searching algorithms of that type into real-time algorithms.

The Boyer–Moore algorithm was designed by Boyer and Moore (1977). The first proof of its linearity when restricted to the search for the first pattern occurrence is by Knuth *et al.* (1977). Cole (1994) proves that the maximum number of symbol comparisons is bounded by  $3n$ , and that this bound is tight.

Knuth *et al.* (1977) consider a variant of the Boyer–Moore algorithm in which all previous matches inside the current window are memorized. Each window configuration becomes the state of what is called the Boyer–Moore

automaton. It is still unknown whether the maximum number of states of the automaton is polynomial or not.

Several variants of the Boyer–Moore algorithm avoid the quadratic behavior when searching for all occurrences of the pattern. Among the more efficient in terms of the number of symbol comparisons are: the algorithm of Apostolico and Giancarlo (1986), Turbo–BM algorithm by Crochemore *et al.* (1992) (the two algorithms are analyzed in Lecroq (1995)), and the algorithm of Colussi (1994).

The general bound on the expected time complexity of string matching is  $O(|y| \log |x|/|x|)$ . The probabilistic analysis of a simplified version of the Boyer–Moore algorithm, similar to the Quick Search algorithm of Sunday (1990) described in the chapter, was studied by several authors.

String searching can be solved by a linear-time algorithm requiring only a constant amount of memory in addition to the pattern and the (window on the) text. This can be proved by different techniques presented in Crochemore and Rytter (2002).

Experimental results presented in this chapter have been realized using SMART (String Matching Algorithms Research Tool, <http://www.dmi.unict.it/~faro/smart/>) by Faro and Lecroq (2011). An exhaustive experimental study can be found in Faro and Lecroq (2010).

The Aho–Corasick algorithm is from Aho and Corasick (1975). It is implemented by the **fgrep** command under the UNIX operating system. Commentz-Walter (1979) has designed an extension of the Boyer–Moore algorithm to several patterns. It is fully described in Aho (1990).

On general alphabets the two-dimensional pattern matching can be solved in linear time, whereas the running time of the Bird/Baker algorithm has an additional  $\log \sigma$  factor. It is still unknown whether the problem can be solved by an algorithm working simultaneously in linear time and using only a constant amount of memory space (see Crochemore and Rytter 2002).

The suffix tree construction of section 1.4 is by McCreight (1976). An on-line construction is given by Ukkonen (1995). Other data structures to represent indexes on text files are: direct acyclic word graph (Blumer *et al.*, 1985), suffix automata (Crochemore, 1986), and suffix arrays (Manber and Myers, 1993). All these techniques are presented in (Crochemore and Rytter, 2002). The data structures implement full indexes with standard operations, whereas applications sometimes need only incomplete indexes. The design of compact indexes is still unsolved.

The suffix array of a string, with the associated search algorithm based on the knowledge of the common prefixes (Section 1.5), is from Manber and Myers (1993). The suffix array construction in Section 1.5.1 is from Kärkkäinen and Sanders (2003) (see also Kim *et al.* (2003) and Ko and Aluru (2003)). The method used in Section 1.5.3 to compute the longest common prefixes of sorted suffixes is from Kasai *et al.* (2001).

First algorithms for aligning two sequences are by Needleman and Wunsch (1970) and Wagner and Fischer (1974). Idea and algorithm for local alignment

is by Smith and Waterman (1981). Hirschberg (1975) presents the computation of the lcs in linear space. This is an important result because the algorithm is classically run on large sequences. Another implementation is given in Durbin *et al.* (1998). The quadratic time complexity of the algorithm to compute the Levenshtein distance is a bottleneck in practical string comparison for the same reason.

Approximate string searching is a lively domain of research. It includes, for instance, the notion of regular expressions to represent sets of strings. Algorithms based on regular expression are commonly found in books related to compiling techniques. The algorithms of section 1.7 are by Baeza-Yates and Gonnet (1992) and Wu and Manber (1992).

The statistical compression algorithm of Huffman (1951) has a dynamic version where symbol counting is done at coding time. The current coding tree is used to encode the next character and then updated. At decoding time a symmetrical process reconstructs the same tree, so the tree does not need to be stored with the compressed text, see Knuth (1985). The command **compact** of UNIX implements this version.

Several variants of the Ziv and Lempel algorithm exist. The reader can refer to Bell *et al.* (1990) for a discussion on them. Nelson (1992) presents practical implementations of various compression algorithms. The *BW* transform is from Burrows and Wheeler (1994).

## Defining Terms

**Alignment:** an alignment of two strings  $x$  and  $y$  is a word of the form  $(\bar{x}_0, \bar{y}_0)(\bar{x}_1, \bar{y}_1) \cdots (\bar{x}_{p-1}, \bar{y}_{p-1})$  where each  $(\bar{x}_i, \bar{y}_i) \in (\Sigma \cup \{\varepsilon\}) \times (\Sigma \cup \{\varepsilon\}) \setminus \{(\varepsilon, \varepsilon)\}$  for  $0 \leq i \leq p-1$  and both  $x = \bar{x}_0\bar{x}_1 \cdots \bar{x}_{p-1}$  and  $y = \bar{y}_0\bar{y}_1 \cdots \bar{y}_{p-1}$ .

**Border:** A word  $u \in \Sigma^*$  is a border of a word  $w \in \Sigma^*$  if  $u$  is both a prefix and a suffix of  $w$  (there exist two words  $v, z \in \Sigma^*$  such that  $w = vu = uz$ ). The common length of  $v$  and  $z$  is a period of  $w$ .

**Edit distance:** The metric distance between two strings that counts the minimum number of insertions and deletions of symbols to transform one string into the other.

**Hamming distance:** The metric distance between two strings of same length that counts the number of mismatches.

**Levenshtein distance:** The metric distance between two strings that counts the minimum number of insertions, deletions, and substitutions of symbols to transform one string into the other.

**Occurrence:** An occurrence of a word  $u \in \Sigma^*$ , of length  $m$ , appears in a word  $w \in \Sigma^*$ , of length  $n$ , at position  $i$  if for  $0 \leq k \leq m - 1$ ,  $u[k] = w[i + k]$ .

**Prefix:** A word  $u \in \Sigma^*$  is a prefix of a word  $w \in \Sigma^*$  if  $w = uz$  for some  $z \in \Sigma^*$ .

**Prefix code:** Set of words such that no word of the set is a prefix of another word contained in the set. A prefix code is represented by a coding tree.

**Segment:** A word  $u \in \Sigma^*$  is a segment of a word  $w \in \Sigma^*$  if  $u$  occurs in  $w$  (see occurrence), that is,  $w = vuz$  for two words  $v, z \in \Sigma^*$  ( $u$  is also referred to as a factor or a subword of  $w$ ).

**Subsequence:** A word  $u \in \Sigma^*$  is a subsequence of a word  $w \in \Sigma^*$  if it is obtained from  $w$  by deleting zero or more symbols that need not be consecutive ( $u$  is sometimes referred to as a subword of  $w$ , with a possible confusion with the notion of segment).

**Suffix:** A word  $u \in \Sigma^*$  is a suffix of a word  $w \in \Sigma^*$  if  $w = vu$  for some  $v \in \Sigma^*$ .

**Suffix tree:** Trie containing all the suffixes of a word.

**Trie:** Tree in which edges are labeled by letters or words.



---

## Bibliography

- [1] Aho, A. V. 1990. Algorithms for finding patterns in strings. In *Handbook of Theoretical Computer Science*, vol. A. *Algorithms and Complexity*, J. van Leeuwen, ed., pp. 255–300. Elsevier, Amsterdam.
- [2] Aho, A. V. and Corasick, M. J. 1975. Efficient string matching: an aid to bibliographic search. *Comm. ACM* 18(6):333–340.
- [3] Baeza-Yates, R. A. and Gonnet, G. H. 1992. A new approach to text searching. *Comm. ACM* 35(10):74–82.
- [4] Baker, T. P. 1978. A technique for extending rapid exact-match string matching to arrays of more than one dimension. *SIAM J. Comput.* 7(4):533–541.
- [5] Bell, T. C., Cleary, J. G., and Witten, I. H. 1990. *Text Compression*. Prentice-Hall, Englewood Cliffs, NJ.
- [6] Bird, R. S. 1977. Two-dimensional pattern matching. *Inf. Process. Lett.* 6(5):168–170.
- [7] Blumer, A., Blumer, J., Ehrenfeucht, A., Haussler, D., Chen, M. T., and Seiferas, J. 1985. The smallest automaton recognizing the subwords of a text. *Theor. Comput. Sci.* 40:31–55.
- [8] Boyer, R. S. and Moore, J. S. 1977. A fast string searching algorithm. *Comm. ACM* 20(10):762–772.
- [9] Breslauer, D., Colussi, L., and Toniolo, L. 1993. Tight comparison bounds for the string prefix matching problem. *Inf. Process. Lett.* 47(1):51–57.
- [10] Burrows, M. and Wheeler, D. 1994. A block sorting lossless data compression algorithm. Technical Report 124, Digital Equipment Corporation.
- [11] Cole, R. 1994. Tight bounds on the complexity of the Boyer-Moore pattern matching algorithm. *SIAM J. Comput.* 23(5):1075–1091.
- [12] Colussi, L. 1994. Fastest pattern matching in strings. *J. Algorithms* 16(2):163–189.
- [13] Crochemore, M. 1986. Transducers and repetitions. *Theor. Comput. Sci.* 45(1):63–86.

- [14] Crochemore, M. and Rytter, W. 2002. *Jewels of Stringology*. World Scientific.
- [15] Durbin, R., Eddy, S., and A. Krogh, A., and Mitchison G. 1998. *Biological sequence analysis probabilistic models of proteins and nucleic acids*. Cambridge University Press.
- [16] S. Faro and T. Lecroq. The Exact String Matching Problem: a Comprehensive Experimental Evaluation. Report arXiv:1012.2547, 2010.
- [17] S. Faro and T. Lecroq. SMART: a string matching algorithm research tool. <http://www.dmi.unict.it/~faro/smart/>, 2011.
- [18] Galil, Z. 1981. String matching in real time. *J. ACM* 28(1):134–149.
- [19] Hancart, C. 1993. On Simon’s string searching algorithm. *Inf. Process. Lett.* 47(2):95–99.
- [20] Hirschberg, D. S. 1975. A linear space algorithm for computing maximal common subsequences. *Comm. ACM* 18(6):341–343.
- [21] Hume, A. and Sunday, D. M. 1991. Fast string searching. *Software—Practice Exp.* 21(11):1221–1248.
- [22] J. Kärkkäinen and P. Sanders. Simple linear work suffix array construction. J. C. M. Baeten, J. K. Lenstra, J. Parrow and G. J. Woeginger, editors. *Proceedings of the 30th International Colloquium on Automata, Languages and Programming* Eindhoven, The Netherlands, Lecture Notes in Computer Science 2719, Springer-Verlag, 2003, 943–955.
- [23] Karp, R. M. and Rabin, M. O. 1987. Efficient randomized pattern-matching algorithms. *IBM J. Res. Dev.* 31(2):249–260.
- [24] T. Kasai, G. Lee, H. Arimura, S. Arikawa and K. Park. Linear-Time Longest-Common-Prefix Computation in Suffix Arrays and Its Applications. A. Amir and G. M. Landau, editors. *Proceedings of the 12th Annual Symposium on Combinatorial Pattern Matching*, Jerusalem, Israel, 2001. Lecture Notes in Computer Science 2089. Springer-Verlag, 2001, 181–192.
- [25] D. K. Kim, J. S. Sim, H. Park and K. Park. Linear-Time Construction of Suffix Arrays. R. A. Baeza-Yates, E. Chávez and M. Crochemore, editors. *Proceedings of the 14th Annual Symposium on Combinatorial Pattern Matching*, Morelia, Michocán, Mexico, 2003. Lecture Notes in Computer Science 2676. Springer-Verlag, 2003, 186–199.
- [26] Knuth, D. E. 1985. Dynamic Huffman coding. *J. Algorithms* 6(2):163–180.
- [27] Knuth, D. E., Morris, J. H., Jr, and Pratt, V. R. 1977. Fast pattern matching in strings. *SIAM J. Comput.* 6(1):323–350.

- [28] P. Ko and S. Aluru. Space Efficient Linear Time Construction of Suffix Arrays. R. A. Baeza-Yates, E. Chávez and M. Crochemore, editors. *Proceedings of the 14th Annual Symposium on Combinatorial Pattern Matching*, Morelia, Michocán, Mexico, 2003. Lecture Notes in Computer Science 2676. Springer-Verlag, 2003, 200–210.
- [29] Lecroq, T. 1995. Experimental results on string-matching algorithms. *Software—Practice Exp.* 25(7):727–765.
- [30] McCreight, E. M. 1976. A space-economical suffix tree construction algorithm. *J. Algorithms* 23(2):262–272.
- [31] Manber, U. and Myers, G. 1993. Suffix arrays: a new method for on-line string searches. *SIAM J. Comput.* 22(5):935–948.
- [32] Needleman, S. B. and Wunsch, C. D. 1970. A general method applicable to the search for similarities in the amino acid sequence of two proteins. *J. Mol. Biol.* 48:443–453.
- [33] Nelson, M. 1992. *The Data Compression Book*. M&T Books.
- [34] Simon, I. 1993. String matching algorithms and automata. In *First American Workshop on String Processing*, Baeza-Yates and Ziviani, eds., pp. 151–157. Universidade Federal de Minas Gerais.
- [35] Smith, T. F. and Waterman, M. S. 1981. Identification of common molecular sequences. *J. Mol. Biol.* 147:195–197.
- [36] Stephen, G. A. 1994. *String Searching Algorithms*. World Scientific Press.
- [37] Sunday, D. M. 1990. A very fast substring search algorithm. *Comm. ACM* 33(8):132–142.
- [38] Ukkonen, E. 1995. On-line construction of suffix trees. *Algorithmica* 14(3):249–260.
- [39] Wagner, R. A. and Fischer, M. 1974. The string-to-string correction problem. *J. ACM* 21(1):168–173.
- [40] Welch, T. 1984. A technique for high-performance data compression. *IEEE Comput.* 17(6):8–19.
- [41] Wu, S. and Manber, U. 1992. Fast text searching allowing errors. *Comm. ACM* 35(10):83–91.
- [42] Zhu, R. F. and Takaoka, T. 1989. A technique for two-dimensional pattern matching. *Comm. ACM* 32(9):1110–1120.

## Further Information

Problems and algorithms presented in the chapter are just a sample of questions related to pattern matching. They share the formal methods used to design solutions and efficient algorithms. A wider panorama of algorithms on texts may be found in a few books such as:

Adjeroh, D., Bell, T. and Mukherjee, A. 2008. *The Burrows-Wheeler Transform: Data Compression, Suffix Arrays, and Pattern Matching*. Springer Verlag.

Apostolico, A. and Galil, Z., editors. 1997. *Pattern Matching Algorithms*. Oxford University Press.

Bell, T. C., Cleary, J. G. and Witten, I. H. 1990. *Text Compression*. Prentice-Hall, Englewood Cliffs, NJ.

Crochemore, M., Hancart, C. and Lecroq, T. 2007. *Algorithms on Strings*. Cambridge University Press.

Crochemore, M. and Rytter, W. 2002. *Jewels of Stringology*. World Scientific.

Gusfield D. 1997. *Algorithms on Strings, Trees and Sequences: Computer Science and Computational Biology*. Cambridge University Press.

Navarro, G. and Raffinot M. 2002. *Flexible Pattern Matching in Strings: Practical On-line Search Algorithms for Texts and Biological Sequences*. Cambridge University Press.

Nelson, M. 1992. *The Data Compression Book*. M&T Books.

Salomon, D. 2000. *Data Compression: the Complete Reference*. Springer Verlag.

Smyth, W. F. 2003. *Computing Patterns in Strings*. Addison-Wesley Longman, 2003.

Stephen, G. A. 1994. *String Searching Algorithms*. World Scientific Press.

Research papers in pattern matching are disseminated in a few journals, among which are: *Communications of the ACM*, *Journal of the ACM*, *Theoretical Computer Science*, *Algorithmica*, *Journal of Algorithms*, *SIAM Journal on Computing*, *Journal of Discrete Algorithms*.

Finally, three main annual conferences present the latest advances of this field of research: Combinatorial Pattern Matching, which started in 1990. Data Compression Conference, which is regularly held at Snowbird. The scope of SPIRE (String Processing and Information Retrieval) includes the domain of data retrieval.

General conferences in computer science often have sessions devoted to pattern matching algorithms.

Several books on the design and analysis of general algorithms contain chapters devoted to algorithms on texts. Here is a sample of these books:

Cormen, T. H., Leiserson, C. E., and Rivest, R. L. 1990. *Introduction to Algorithms*. MIT Press.

Gonnet, G. H. and Baeza-Yates, R. A. 1991. *Handbook of Algorithms and Data Structures*. Addison-Wesley.

Animations of selected algorithms may be found at:

- <http://www-igm.univ-mlv.fr/~lecroq/string/> (Exact String Matching Algorithms),
- <http://www-igm.univ-mlv.fr/~lecroq/seqcomp/> (Alignments).