CHAPTER 1

Algorithms on Words

1.0. Introduction

This chapter is an introductory chapter to the book. It gives general notions, notation, and technical background. It covers, in a tutorial style, the main notions in use in algorithms on words. In this sense, it is a comprehensive exposition of basic elements concerning algorithms on words, automata and transducers, and probability on words.

The general goal of "stringology" we pursue here is to manipulate strings of symbols, to compare them, to count them, to check some properties, and perform simple transformations in an effective and efficient way.

A typical illustrative example of our approach is the action of circular permutations on words, because several of the aspects we mentioned above are present in this example. First, the operation of circular shift is a transduction which can be realized by a transducer. We include in this chapter a section (Section 1.5) on transducers. Transducers will be used in Chapter 3. The orbits of the transformation induced by the circular permutation are the so-called conjugacy classes. Conjugacy classes are a basic notion in combinatorics on words. The minimal element in a conjugacy class is a good representative of a class. It can be computed by an efficient algorithm (actually in linear time). This is one of the algorithms which appear in Section 1.2. Algorithms for conjugacy are again considered in Chapter 2. These words give rise to Lyndon words which have remarkable combinatorial properties already emphasized in Lothaire (1997). We describe in Section 1.2.5 the Lyndon factorization algorithm.

The family of algorithms on words has features which make it a specific field within algorithmics. Indeed, algorithms on words are often of low complexity but intricate and difficult to prove. Many algorithms have even

Applied Combinatorics on Words, eds. Jean Berstel and Dominique Perrin.
Published by Cambridge University Press.

a linear time complexity corresponding to a single pass scanning of the input word. This contrasts with the fact that correctness proofs of these algorithms are frequently complex. A well-known example of this situation is the Knuth–Morris–Pratt string searching algorithm (see Section 1.2.3). This algorithm is compact, and apparently simple but the correctness proof requires a sophisticated loop invariant.

The field of algorithms on words still has challenging open problems. One of them is the minimal complexity of the computation of a longest common subword of two words which is still unknown. We present in Section 1.2.4 the classic quadratic dynamic programming algorithm. A more efficient algorithm is mentioned in the Notes.

The field of algorithms on words is intimately related to formal models of computation. Among those models, finite automata and context-free grammars are the most used in practice. This is why we devote a section (Section 1.3) to finite automata and another one to grammars and syntax analysis (Section 1.6). These models, and especially finite automata, regular expressions, and transducers, are ubiquitous in the applications. They appear in almost all chapters.

The relationship between words and probability theory is an old one. Indeed, one of the basic aspects of probability and statistics is the study of sequences of events. In the elementary case of a finite sample space, such as in tossing a coin, the sequence of outcomes is a word. More generally, a partition of an arbitrary probability space into a finite number of classes produces sequences over a finite set. Section 1.8 is devoted to an introduction to these aspects. They are developed later in Chapters 6 and 7.

We have chosen to present the algorithms and the related properties in a direct style. This means that there are no formal statements of theorems, and consequently no formal proofs. Nevertheless, we give precise assertions and enough arguments to show the correctness of algorithms and to evaluate their complexity. In some cases, we use results without proof and we give bibliographic indications in the Notes.

For the description of algorithms, we use a kind of programming language that is close to the usual programming languages. Doing this, rather than relying on a precise programming language, gives more flexibility and improves readability.

The syntactic features of our programs concerning the control structure and the elementary instructions, make the language similar to a language such as Pascal. We take some liberty with real programs. In particular, we often omit declarations and initializations of variables. The parameter handling is C–like (no call by reference). In addition to arrays, we also use implicitly data structures such as sets and stacks and pairs or triples of

variables to simplify notation. All functions are global, and there is nothing resembling classes or other features of object-oriented programming. However, we use overloading for parsimony. The functions are referenced in the text and in the index by their name, like LONGESTCOMMONPREFIX() for example.

1.1. Words

We briefly introduce the basic terminology on words. Let $\mathcal{A}$ be a finite set usually called the *alphabet*. In practice, the elements of the alphabet may be characters from some concrete alphabet, but also more complex objects. They may themselves be words on another alphabet, as in the case of syllables in natural language processing (presented in Chapter 3). In information processing, any kind of record can be viewed as a symbol in some huge alphabet. This has the consequence that some apparently elementary operations on symbols, like the test for equality, often need a careful definition and may require a delicate implementation.

We denote as usual by $\mathcal{A}^*$ the set of words over $\mathcal{A}$ and by ε the empty word. For a word w, we denote by $|w|$ the length of w. We use the notation $\mathcal{A}^+ = \mathcal{A}^* - \{\varepsilon\}$. The set $\mathcal{A}^*$ is a monoid. Indeed, the concatenation of words is associative, and the empty word is a neutral element for concatenation. The set $\mathcal{A}^+$ is sometimes called the *free semigroup* over $\mathcal{A}$, while $\mathcal{A}^*$ is called the *free monoid*.

A word w is called a *factor* (resp. a *prefix*, resp. a *suffix*) of a word u if there exist words x, y such that $u = xwy$ (resp. $u = wy$, resp. $u = xw$). The factor (resp. the prefix, resp. the suffix) is *proper* if $xy \neq \varepsilon$ (resp. $y \neq \varepsilon$, resp. $x \neq \varepsilon$). The prefix of length k of a word w is also denoted by $w[0\,..\,k-1]$.

The set of words over a finite alphabet $\mathcal{A}$ can be conveniently seen as a tree. Figure 1.1 represents $\{a, b\}^*$ as a binary tree. The vertices are the elements of $\mathcal{A}^*$. The root is the empty word ε. The sons of a node x are the words xa for $a \in \mathcal{A}$. Every word x can also be viewed as the path leading from the root to the node x. A word x is a prefix of a word y if it is an ancestor in the tree. Given two words x and y, the longest common prefix of x and y is the nearest common ancestor of x and y in the tree.

A word x is a *subword* of a word y if there are words $u_1, \ldots, u_n$ and $v_0, v_1, \ldots, v_n$ such that $x = u_1 \cdots u_n$ and $y = v_0 u_1 v_1 \cdots u_n v_n$. Thus, x is obtained from y by erasing some factors in y.

Given two words x and y, a *longest common subword* is a word z of maximal length that is both a subword of x and y. There may exist several

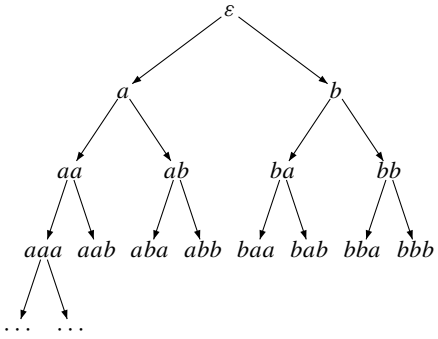

Figure 1.1. The tree of the free monoid on two letters.

longest common subwords for two words x and y. For instance, the words abc and acb have the common subwords ab and ac.

We denote by alph w the set of letters having at least one occurrence in the word w.

The set of factors of a word x is denoted $F(x)$. We denote by $F(\mathcal{X})$ the set of factors of words in a set $\mathcal{X} \subset \mathcal{A}^*$. The *reversal* of a word $w = a_1 a_2 \cdots a_n$, where $a_1, \ldots, a_n$ are letters, is the word $\tilde{w} = a_n a_{n-1} \cdots a_1$. Similarly, for $\mathcal{X} \subset \mathcal{A}^*$, we denote $\tilde{\mathcal{X}} = \{\tilde{x} \mid x \in \mathcal{X}\}$. A *palindrome word* is a word w such that $w = \tilde{w}$. If $|w|$ is even, then w is a palindrome if and only if $w = x\tilde{x}$ for some word x. Otherwise w is a palindrome if and only if $w = xa\tilde{x}$ for some word x and some letter a.

An integer $p \geq 1$ is a *period* of a word $w = a_1 a_2 \cdots a_n$ where $a_i \in \mathcal{A}$ if $a_i = a_{i+p}$ for $i = 1, \ldots, n - p$. The smallest period of w is called *the* period or the *minimal* period of w.

A word $w \in \mathcal{A}^+$ is *primitive* if $w = u^n$ for $u \in \mathcal{A}^+$ implies $n = 1$.

Two words x, y are *conjugate* if there exist words u, v such that $x = uv$ and $y = vu$. Thus conjugate words are just cyclic shifts of one another. Conjugacy is thus an equivalence relation. The conjugacy class of a word of length n and period p has p elements if p divides n and has n elements otherwise. In particular, a primitive word of length n has n distinct conjugates.

1.1.1. Ordering

There are three order relations frequently used on words. We give the definition of each of them.

The *prefix order* is the partial order defined by $x \leq y$ if x is a prefix of y.

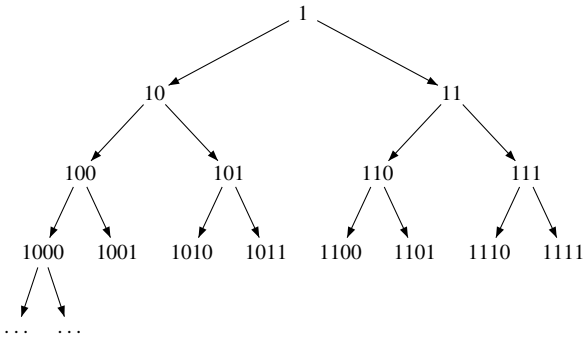

Figure 1.2. The tree of integers in binary notation.

Two other orders, the *radix order* and the *lexicographic order* are refinements of the prefix order which are defined for words over an ordered alphabet $\mathcal{A}$. Both are total orders.

The *radix order* is defined by $x \leq y$ if $|x| < |y|$ or $|x| = |y|$ and $x = uax'$ and $y = uby'$ with a, b letters and $a \leq b$. If integers are represented in base k without leading zeroes, then the radix order on their representations corresponds to the natural ordering of the integers. If we allow leading zeroes, the same holds provided the words have the same length (which can always be achieved by padding).

For $k = 2$, the tree of words without leading zeroes is given in Figure 1.2. The radix order corresponds to the order in which the vertices are met in a breadth-first traversal. The index of a word in the radix order is equal to the number represented by the word in base 2.

The *lexicographic order*, also called *alphabetic order*, is defined as follows. Given two words x, y, we have $x < y$ if x is a proper prefix of y or if there exist factorizations $x = uax'$ and $y = uby'$ with a, b letters and $a < b$. This is the usual order in a dictionary. Note that $x < y$ in the radix order if $|x| < |y|$ or if $|x| = |y|$ and $x < y$ in the lexicographic order.

1.1.2. Distances

A *distance* over a set E is a function d that assigns to each element of E a nonnegative number such that:

(i) $d(u, v) = d(v, u)$,
(ii) $d(u, w) \leq d(u, v) + d(v, w)$ (triangular inequality)
(iii) $d(u, v) = 0$ if and only if $u = v$.

Several distances between words are used (see Figures 1.3 and 1.4). The most common is the *Hamming distance*. It is only defined on words of equal

(a) Hamming distance (b) Subword distance

Figure 1.3. The Hamming distance is 3 and the subword distance is 2.

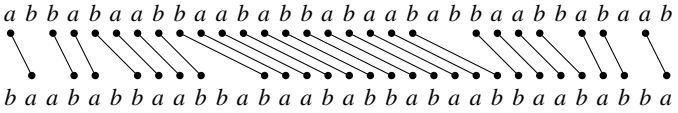

Figure 1.4. The Hamming distance of these two Thue–Morse blocks of length 32 is equal to their length; their subword distance is only 6.

length. For two words $u = a_0 \cdots a_{n-1}$ and $v = b_0 \cdots b_{n-1}$, where a_i, b_i are letters, it is the number $d_H(u, v)$ of indices i with $0 \leq i \leq n-1$ such that $a_i \neq b_i$. In other terms

$$d_H(u, v) = \text{Card}\{i \mid 0 \leq i \leq n-1 \text{ and } a_i \neq b_i\}.$$

Thus the Hamming distance is the number of *mismatches* between u and v. It can be verified that d_H is indeed a distance. Observe that $d_H(u, v) = n - p$ where p is the number of positions at which u and v coincide. In a more general setting, a distance between letters is used instead of just counting each mismatch as 1.

The Hamming distance takes into account the differences at the same position. In this way, it can be used as a measure of modifications or errors caused by a modification of a symbol by another one, but not of a deletion or an insertion. Another distance is the subword distance which is defined as follows. Let u be a word of length n and v be a word of length m, and p be the length of a longest common subword of u and v. The *subword distance* between u and v is defined as $d_S(u, v) = n + m - 2p$. It can be verified that $d_S(u, v)$ is the minimal number of insertions and suppressions that change u into v. The name *indel* (for *in*sertion and *del*etion) is used to qualify a transformation that is either an insertion or a deletion.

A common generalization of the Hamming distance and the subword distance is the *edit distance*. It takes into account the substitutions of a symbol by another in additions to indels (see Problem 1.1.2).

A related distance is the *prefix distance*. It is defined as $d(u, v) = n + m - 2p$ where $n = |u|, m = |v|$ and p is the length of the longest common prefix of u and v. It can be verified that the prefix distance is actually the length of the shortest path from u to v in the tree of the free monoid.

1.2. Elementary algorithms

In this section, we treat algorithmic problems related to the basic notions on words: prefixes, suffixes, factors.

1.2.1. Prefixes and suffixes

Recall that a word x is a *prefix* of a word y if there is a word u such that $y = xu$. It is said to be *proper* if u is nonempty. Checking whether x is a prefix of y is straightforward. Algorithm LONGESTCOMMONPREFIX below returns the length of the longest common prefix of two words x and y.

```
LONGESTCOMMONPREFIX(x, y)
1 ▷ x has length m, y has length n
2 i ← 0
3 while i < m and i < n and x[i] = y[i] do
4       i ← i + 1
5 return i
```

In the tree of a free monoid, the length of the longest common prefix of two words is the height of the least common ancestor.

As mentioned earlier, the conceptual simplicity of the above algorithm hides implementation details such as the computation of equality between letters.

1.2.2. Overlaps and borders

We introduce first the notion of overlap of two words x and y. It captures the amount of possible overlap between the end of x and the beginning of y. To avoid trivial cases, we rule out the case where the overlap would be the whole word x or y. Formally, the *overlap* of x and y is the longest proper suffix of x that is also a proper prefix of y. For example, the overlap of *abacaba* and *acabaca* has length 5. The *border* of a nonempty word w is the overlap of w and itself. Thus it is the longest word u which is both a proper prefix and

a proper suffix of w. The overlap of x and y is denoted by overlap(x, y), and the border of x by border(x). Thus border(x) = overlap(x, x).

As we shall see, the computation of the overlap of x and y is intimately related to the computation of the border. This is due to the fact that the overlap of x and y involves the computation of the overlaps of the prefixes of x and y. Actually, one has overlap(xa, y) = border(xa) whenever x is a prefix of y and a is a letter. Next, the following formula allows the computation of the overlap of xa and y, where x, y are words and a is a letter. Let z = overlap(x, y).

$$\text{overlap}(xa, y) = \begin{cases} za & \text{if } za \text{ is a prefix of } y, \\ \text{border}(za) & \text{otherwise.} \end{cases}$$

Observe that border(za) = overlap(za, y) because z is a prefix of y. The computation of the border is an interesting example of a nontrivial algorithm on words. A naive algorithm would check, for each prefix of w, whether it is also a suffix of w, and select the longest such prefix. This would obviously require a time proportional to $|w|^2$. We will see that it can be done in time proportional to the length of the word. This relies on the following recursive formula allowing the computation of the border of xa in terms of the border of x, where x is a word and a is a letter. Let u = border(x) be the border of x. Then for each letter a,

$$\text{border}(xa) = \begin{cases} ua & \text{if } ua \text{ is a prefix of } x, \\ \text{border}(ua) & \text{otherwise.} \end{cases} \tag{1.2.1}$$

The following algorithm (Algorithm BORDER) computes the length of the border of a word x of length m. It outputs an array b of $m + 1$ integers such that $b[j]$ is the length of the border of $x[0\,..\,j-1]$. In particular, the length of border(x) is $b[m]$. It is convenient to set $b[0] = -1$. For example, if $x = abaababa$, the array b is

	0	1	2	3	4	5	6	7	8
b :	−1	0	0	1	1	2	3	2	3

BORDER(x)

```
1  ▷ x has length m, b has size m + 1
2  i ← 0
3  b[0] ← −1
4  for j ← 1 to m − 1 do
5        b[j] ← i
```

```
 6        ▷ Here x[0..i − 1] = border(x[0..j − 1])
 7        while i ≥ 0 and x[j] ≠ x[i] do
 8              i ← b[i]
 9        i ← i + 1
10  b[m] ← i
11  return b
```

This algorithm is an implementation of Formula (1.2.1). Indeed, the body of the loop on j computes, in the variable i, the length of the border of $x[0\,..\,j]$. This value will be assigned to $b[j]$ at the next increase of j. The inner loop is a translation of the recursive formula.

The algorithm computes the border of x (or the table b itself) in time $O(|x|)$. Indeed, the execution time is proportional to the number of comparisons of symbols performed at line 7. Each time a comparison is done, the expression $2j - i$ increases strictly. In fact, either $x[j] = x[i]$ and i, j both increase by 1, or $x[j] \neq x[i]$, and j remains constant while i decreases strictly (since $b[i] < i$). Since the value of the expression is initially 0 and is bounded by $2|x|$, the number of comparisons is at most $2|x|$.

The computation of the overlap of two words x, y will be done in the next section.

1.2.3. Factors

In this section, we consider the problem of checking whether a word x is a factor of a word y. This problem is usually referred to as a *string matching problem*. The word x is called the *pattern* and y is the *text*. A more general problem, referred to as *pattern matching*, occurs when x is replaced by a *regular expression* $\mathcal{X}$ (see Section 1.4). The evaluation of the efficiency of string matching or pattern matching algorithms depends on which parameters are considered. In particular, one may consider the pattern to be fixed (because several occurrences of the same pattern are looked for in an unknown text), or the text to be fixed (because several different patterns will be looked for in this text). When the pattern or the text is fixed, it may be subject to a preprocessing. Moreover, the evaluation of the complexity can take into account either only the computation time, or both time and space. This may make a significant difference on very large texts and patterns.

We begin with a naive quadratic string searching algorithm. To check whether a word x is a factor of a word y, it is clearly enough to test for each index $j = 0, \ldots, n - 1$ if x is a prefix of the word $y[j\,..\,n - 1]$.

```
NaiveStringMatching(x, y)
1 ▷ x has length m, y has length n
2 (i, j) ← (0, 0)
3 while i < m and j < n do
4       if x[i] = y[j] then
5             (i, j) ← (i + 1, j + 1)
6       else  j ← j − i + 1
7             i ← 0
8 return i = m
```

The number of comparisons required in the worst case is $O(|x||y|)$. The worst case is reached for $x = a^m b$ and $y = a^n$. The number of comparisons performed is in this case $m(n - m - 1)$.

We shall see now that it is possible to search a word x inside another word y in linear time, that is in time $O(|x| + |y|)$. The basic idea is to use a finite automaton recognizing the words ending with x. If we can compute some representation of it in time $O(|x|)$, then it will be straightforward to process the word y in time $O(|y|)$.

The wonderfully simple solution presented below uses the notion of border of a word. Suppose that we are in the process of identifying x inside y, the position i in x being placed in front of position j in y, as in the naive algorithm. We can then set $x = ubt$ where $b = x[i]$ and $y = wuaz$ where $a = y[j]$. If $a = b$, the process goes on with $i + 1$, $j + 1$. Otherwise, instead of just moving x one place to the right (i.e. $j = j - i + 1, i = 0$), we can take into account that the next possible position for x is determined by the border of u. Indeed, we must have $y = w'u'az$ and $x = u'ct'$ with u' both a prefix of u and a suffix of u since $w'u' = wu$ (see Figure 1.5). Hence the next comparison to perform is between $y[j]$ and $x[k]$ where $k - 1$ is the length of the border of u.

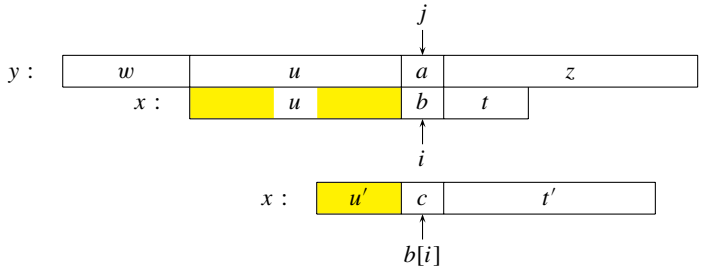

Figure 1.5. Checking $y[j]$ against $x[i]$: if they are different, $y[j]$ is checked against $x[b[i]]$.