# 2 Character Strings

The resolution of problems involving *character strings* (or simply *strings*) rapidly became an important domain in the study of algorithms. These problems occur, of course, in text processing, such as spell checking and the search for *factors (substrings)* or for more general *patterns*. With the development of bioinformatics, new problems arose in the alignment of DNA chains. This chapter presents a selection of string-processing algorithms that we consider important.

A character string could be represented internally by a list of symbols, but, in general, we use the Python native type `str`, which behaves essentially like a list. The characters can be encoded with two bytes for a *Unicode* encoding, but are usually encoded with only one byte, using the *ASCII* encoding: each integer between 0 and 127 corresponds to a distinct character, and the codes are organised in such a way that the successive symbols '0' to '9', 'a' to 'z' and 'A' to 'Z' are consecutive. Thereby, if a string `s` contains only capitals, we can recover the rank of the $i$th letter by computing `ord(s[i])-ord('A')`. Conversely, the $j$th capital letter of the alphabet numbered from~0—is obtained with `chr(j+ord('A'))`.

When we speak of a *factor* (or *substring*) of a string, we require the characters to be consecutive, in contrast with the more general notion of a *subsequence*.

## 2.1 Anagrams

*Definition*

A word $w$ is an *anagram* of a word $v$ if a permutation of the letters transforming $w$ into $v$ exists. Given a set of $n$ words of length at most $k$, we would like to detect all possible anagrams.

**input:** below the car is a rat drinking cider and bending its elbow while this thing is an arc that can act like a cat which cried during the night caused by pain in its bowel[1]

**output:** {bowel below elbow}, {arc car}, {night thing}, {cried cider}, {act cat}

[1] Believe it or not: the authors did not consume cider in order to produce this sample input.

*Complexity*

The proposed algorithm solves this problem in time $O(nk \log k)$ on average, and in $O(n^2 k \log k)$ in the worst case, due to the use of a dictionary.

*Algorithm*

The idea is to compute a signature for each word, so that two words are anagrams of each other if and only if they have the same signature. This signature is simply a new string made up of the same letters sorted in lexicographical order.

The data structure used is a dictionary associating with each signature the list of words with this signature.

```
def anagrams(S):                  # S is a set of strings
    d = {}                        # maps s to list of words with signature s
    for word in S:                # group words according to the signature
        s = ''.join(sorted(word)) # calculate the signature
        if s in d:
            d[s].append(word)     # append a word to an existing signature
        else:
            d[s] = [word]         # add a new signature and its first word
    # -- extract anagrams, ingoring anagram groups of size 1
    return [d[s] for s in d if len(d[s]) > 1]
```

*Problem*

Anagram verifier [spoj:ANGRAM]

## 2.2 T9—Text on 9 Keys

```
input:  2665687
output: bonjour
```

*Application*

Mobile telephones with keys offer an interesting input mode, sometimes called T9. The 26 letters of the alphabet are distributed over the keys 2 to 9, as in Figure 2.1. To input a word, it suffices to input the corresponding sequence of digits. However, as several words could begin with the same digits, a dictionary must be used to propose the most probable word. At any moment, the telephone displays the prefix of the most probable word corresponding to the sequence of digits entered.

*Definition*

The first part of the problem instance is a dictionary, composed of pairs $(m, w)$ where $m$ is a word over the alphabet of 26 lower-case letters, and $w$ is a weight of the importance of the word. The second part is a series of sequences of digits from 2 to 9. For each sequence $s$, the word in the dictionary of maximal weight is to be displayed. A word $m$ corresponds to $s$ if $s$ is a prefix of the sequence $t$ obtained from $m$ by replacing each letter by the corresponding digit, according to the correspondence table given in Figure 2.1. For example, ‘bonjour’ corresponds to 26, but also to 266 or 2665687.

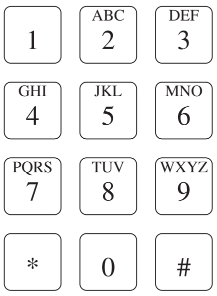

**Figure 2.1** The keys of a mobile phone.

*Algorithm*

The complexity is $O(nk)$ for the initialisation of the dictionary, and $O(k)$ for each request, where $n$ is the number of words in the dictionary and $k$ an upper bound on the length of the words.

In a first phase, we compute for each prefix $p$ of a word in the dictionary the total weight of all the words with prefix $p$. This weight is stored in a dictionary `total_weight`. In a second phase, we store in a dictionary `prop[seq]` the prefix to be proposed for a given sequence `seq`. A scan over the keys in `total_weight` allows the determination of the prefix with greatest weight.

A principal ingredient is the function `code_word`, which for a given word returns the corresponding sequence of digits.

To improve readability, the implementation below is in $O(nk^2)$.

```
t9 = "22233344455566677778889999"
#     abcdefghijklmnopqrstuvwxyz   mapping on the phone


def letter_to_digit(x):
    assert 'a' <= x <= 'z'
    return t9[ord(x) - ord('a')]


def code_word(word):
    return ''.join(map(letter_to_digit, word))
def predictive_text(dic):
    # total_weight[p] = total weight of words having prefix p
    total_weight = {}
    for word, weight in dic:
        prefix = ""
        for x in word:
            prefix += x
            if prefix in total_weight:
                total_weight[prefix] += weight
            else:
                total_weight[prefix] = weight
    #   prop[s] = prefix to display for s
    prop = {}
    for prefix in total_weight:
        code = code_word(prefix)
        if (code not in prop
                or total_weight[prop[code]] < total_weight[prefix]):
            prop[code] = prefix
    return prop


def propose(prop, seq):
    if seq in prop:
        return prop[seq]
    return None
```

*Problem*

T9 [poj:1451]

## 2.3 Spell Checking with a Lexicographic Tree

*Application*

How should the words of a dictionary be stored in order to implement a spell checker? For a given word, we would like to quickly find a dictionary word close to it in the sense of the *Levenshtein edit distance*, see Section 3.2 on page 63. If we store the dictionary words in a hash table, then we lose all proximity information between words. It is better to store them in a *lexicographic tree*, also known as a *prefix tree* or a *trie* (pronounced like ‘try’).

*Definition*

A *trie* is a tree that stores a set of words. The edges of a node towards its children are labelled by distinct letters. Each word in the dictionary then corresponds to a path from the root to a node in the tree. The nodes are marked to distinguish those corresponding to words in the dictionary from those that are only strict prefixes of such words, see Figure 2.2.

*Spell Checking*

With such a structure, it is easy to find a dictionary word that is at a distance `dist` from a given word, for the *Levenshtein edit distance* defined in Section 3.2 on page 63. It suffices to simulate the edit operations at each node, and invoke recursive calls to the search with the parameter `dist - 1`.

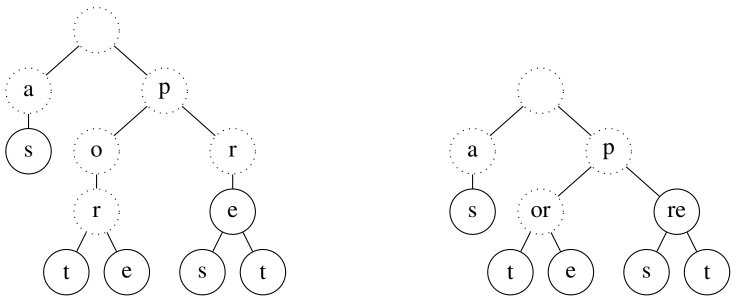

**Figure 2.2** A lexicographic tree for the words `as`, `port`, `pore`, `pré`, `près`, `prêt` (but without the accents). In this figure, the label of each edge is indicated in the node of the child. The circles with solid boundaries mark the nodes that correspond to words in the dictionary. On the right, we show a *Patricia trie* for the same dictionary.

```python
from string import ascii_letters    # in Python 2 one would import letters


class TrieNode:
    def __init__(self):                          # each node will have
        self.is_word = False                     # 52 children -
        self.s = {c: None for c in ascii_letters}  # most will remain empty


def add(T, w, i=0):  # Add a word to the trie
    if T is None:
        T = TrieNode()
    if i == len(w):
        T.is_word = True
    else:
        T.s[w[i]] = add(T.s[w[i]], w, i + 1)
    return T
def Trie(S):  # Build the trie for the words in the dictionary S
    T = None
    for w in S:
        T = add(T, w)
    return T


def spell_check(T, w):  # Spell check a word against the trie
    assert T is not None
    dist = 0
    while True:
        u = search(T, dist, w)
        if u is not None:  # Match at distance dist
            return u
        dist += 1          # No match - try increasing the distance
def search(T, dist, w, i=0):
    if i == len(w):
        if T is not None and T.is_word and dist == 0:
            return ""
        else:
            return None
    if T is None:
        return None
    f = search(T.s[w[i]], dist, w, i + 1)       # matching
    if f is not None:
        return w[i] + f
    if dist == 0:
        return None
    for c in ascii_letters:
        f = search(T.s[c], dist - 1, w, i)      # insertion
        if f is not None:
            return c + f
        f = search(T.s[c], dist - 1, w, i + 1)  # substitution
        if f is not None:
            return c + f
    return search(T, dist - 1, w, i + 1)        # deletion
```

*Variant*
A more complex structure exists that merges nodes as long as they have a single child. A node is thus labelled with a word, rather than by a simple letter, see Figure 2.2. This structure, optimal in memory and in traversal time, is known as a *Patricia trie* (see Morrison, 1968).

*Problem*
Spell checker [icpcarchive:3872]

## 2.4 Searching for Patterns

```
input: lalopalalali lala
output       ^
```

*Definition*
Given a string $s$ of length $n$ and a pattern $t$ of length $m$, we want to find the first index $i$ such that $t$ is a factor of $s$ at the position $i$. The response should be $-1$ if $t$ is not a factor of $s$.

*Complexity*
$O(n+m)$ (see Knuth et al., 1977).

*Naive Algorithm*
This consists of testing all possible alignments of $t$ under $s$ and for each alignment $i$ verifying character by character if $t$ corresponds to $s[i..i+m-1]$. The complexity is $O(nm)$ in the worst case. The following example illustrates the comparisons performed by the algorithm for an example. Each line corresponds to a choice of $i$, and indicates the characters of the 'pattern motif' that match, or an $\times$ for a difference.

| | l | a | l | o | p | a | l | a | l | a | l | i |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | l | a | l | × | | | | | | | | |
| 1 | | × | | | | | | | | | | |
| 2 | | | l | × | | | | | | | | |
| 3 | | | | × | | | | | | | | |
| 4 | | | | | × | | | | | | | |
| 5 | | | | | | × | | | | | | |
| 6 | | | | | | | l | a | l | a | | |

Observe that after handling a few $i$, we already know about a good portion of the string $s$. We could use this information to skip the comparison of $t[0]$ with $s[i]$ in the above example. This observation is extended to a study of the *boundaries* of the strings, leading to the optimal Knuth–Morris–Pratt algorithm, described in the next section.

## 2.5 Maximal Boundaries—Knuth–Morris–Pratt

```
entrée: abracadabra
sortie: abra abra
```

An important classic problem on character strings is the detection of occurrences of a pattern $t$ in a string $s$. The Knuth–Morris–Pratt algorithm solves this problem in optimal linear time $O(|t| + |s|)$ (see Knuth et al., 1977). The essential ingredient of this algorithm is the search for the boundaries of a string, which is the subject of this section. At first sight this problem seems to be totally artificial and without any applications. However, do not be fooled—this problem is at the heart of several classic problems concerning character strings, and we will present a few of these. We begin by introducing a few formal notions. Note that in what follows, the terms ‘word’ and ‘character string’ are used interchangeably.

*Definition*
The *boundary* of a word $w$ is a word that is at the same time a strict prefix and a strict suffix of $w$, where by *strict* we mean that the length of the boundary must be strictly less than the length of $w$. The *maximal boundary* of $w$ is its longest boundary, and is denoted $\beta(w)$. For example, $abaababaa$ has for boundaries $abaa$, $a$ and the empty word $\varepsilon$, hence $\beta(abaababaa) = abaa$. See Figure 2.3 for an illustration.

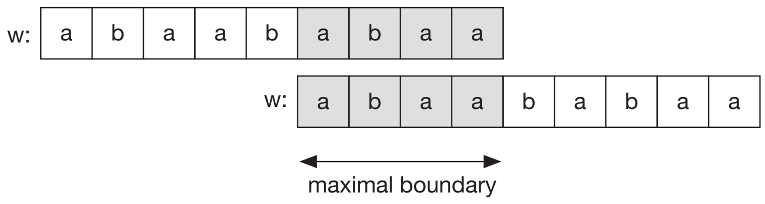

**Figure 2.3** Intuitively, the maximal boundary represents the longest *overlap* of a word with itself: take two copies of the word $w$ one over the other, and slide the second to the right until the letters of the two words coincide. The portion of $w$ corresponding to this overlap is then its maximal boundary.

Given a string $w = w_0 \cdots w_{n-1}$, we would like to compute its maximal boundary. As we will see further on, this problem is recursive in nature, and hence the solution is based on the computation of the maximal boundaries of each of its prefixes. As a boundary is completely described by its length, we will in fact compute the lengths of the maximal boundaries. We thus seek to efficiently construct the sequence of lengths $f_i = |\beta(w_0 \cdots w_i)|$ for $i = 0, \ldots, n-1$, see Figure 2.4.

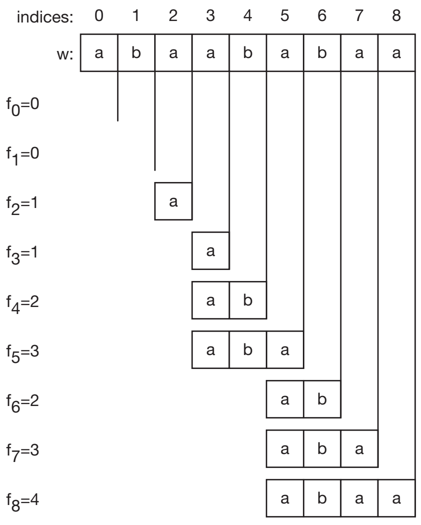

**Figure 2.4** The array $f$ of boundary lengths illustrated with an example. For a given string $w$, $f_i$ is the largest value $k$ such that the suffix $w_{i-k+1}\ldots w_i$ is equal to the prefix $w_0 \ldots w_{k-1}$.

### *Key Observation*

The relation of being a boundary is transitive: if $v$ is a boundary of a boundary $b$ of $u$, then $v$ is also a boundary of $u$. Moreover, if $v$ is a boundary of $u$ shorter than $\beta(u)$, then $v$ is a boundary of $\beta(u)$, see Figure 2.5. Hence, the iterated application of $\beta$ to a word $w$ generates all the boundaries of $w$. For example, with $w = abaababa$, we have $\beta(w) = aba$, then $\beta(\beta(w)) = a$ and finally $\beta(\beta(\beta(w))) = \varepsilon$. This will prove very useful for our algorithm.

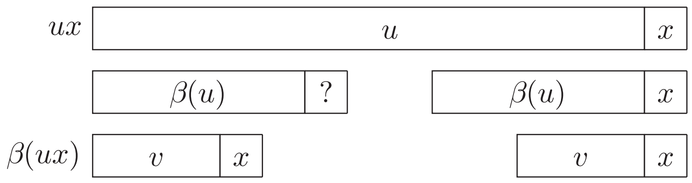

**Figure 2.5** Visual explanation of the proof: every boundary of $uw_i$ can be written $vw_i$ where $v$ is a boundary of $u$, hence knowing the lengths of the boundaries of $u$ allows the identification of the longest boundary of $uw_i$.

### *Algorithm and Proof*

Suppose that we know the maximal boundaries for the first $i$ prefixes of $w$, i.e. up to the prefix $u = w_0 \cdots w_{i-1}$, for $1 \le i < n$. Consider the next prefix $uw_i$: if its maximal boundary is not the empty word, then it is of the form $vw_i$ where $v$ is a boundary of $u$. Thus, to find the longest boundary of $uw_i$ it suffices to verify whether for each boundary $v$ of $u$ in order of decreasing length, the word $vw_i$ is a boundary of $uw_i$, see Figure 2.5. We already know that $vw_i$ is a suffix of $uw_i$ and that $v$ is a prefix of $u$, hence it suffices to compare the two letters $w_k \overset{?}{=} w_i$ where $k = |v|$: indeed, $w_k$ is the letter following the prefix $v$ in $uw_i$. How can we iterate over the boundaries $v$ of $u$ in order of decreasing size? We simply successively apply the function $\beta$ to the word $u$, until we hit the empty word. Note that to perform this test, we only need to know the lengths of the boundaries $v$ of $u$, which have been stored in the array $f$.

### *Illustration*

We use a window that exposes a letter $w_i$ of the first copy of $w$ and a letter $w_k$ of the second copy. Three cases must be considered:

1. If the two characters are equal, then we know that $w_0 \cdots w_k$ is the longest boundary of $w_0 \cdots w_i$, and set $f_i = k + 1$, since $|w_0 \cdots w_k| = k + 1$. The window is then shifted one step to the right in order to process the next prefix, see Figure 2.6; this corresponds to an increment of both $i$ and $k$.
2. If not, in the case $w_k \neq w_i$ and $k > 0$, we move on to the next smaller boundary $\beta(w_0 \dots w_{k-1})$, of size $f_{k-1}$. This boils down to shifting to the right the second copy of $w$, until its contents coincide with the first along the whole of the left part of the window, see Figure 2.6.
3. The final case corresponds to $k = 0$ and $w_0 \neq w_i$. At this point, we know that the maximal boundary of $w_0 \dots w_i$ is the empty word. We thus set $f_i = 0$.

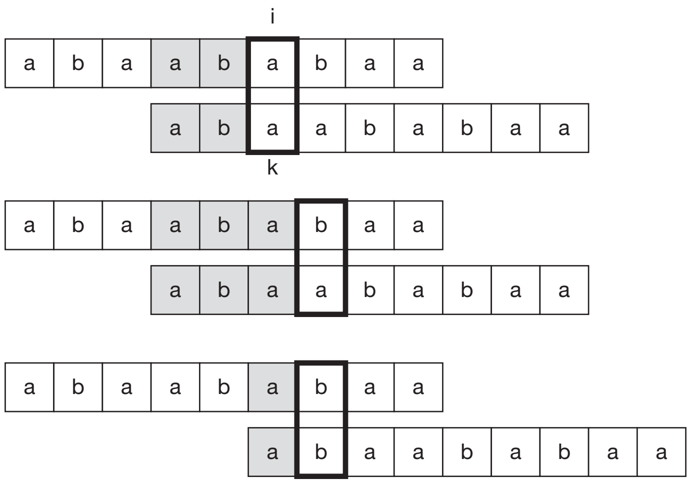

**Figure 2.6** An example of the execution of the Knuth–Morris–Pratt algorithm. When the window reveals two identical characters, we set $f_i = k + 1$ and shift to the right, which comes down to incrementing $i$ and $k$. However, when the window exposes two different characters, the bottom word must be shifted to the right, so that its prefix determined by the window is of length $f_{k-1}$.

### *Code*

All these cases can be captured in just a few lines of code. An outer `while` loop looks for the boundary of a suitable prefix (case 2 above). We exit this loop for one of two reasons. First, if $k = 0$ (no non-empty boundary was found), then only a single character remains to test: $w_k \overset{?}{=} w_i$. If yes, then $f_i = 1$ (case 1); if not, $f_k = 0$ (case 3). Second, we exit because a non-empty boundary is found where $w_k = w_i$ and $f_i = k + 1$: we increment $k$ and then assign $k$ to $f_i$. We thus ensure that at each start of the loop `for`, the value of $k$ corresponds to the length of the longest boundary of the preceding prefix.

```python
def maximum_border_length(w):
    n = len(w)
    f = [0] * n                 # init f[0] = 0
    k = 0                       # current longest border length
    for i in range(1, n):       # compute f[i]
        while w[k] != w[i] and k > 0:
            k = f[k - 1]        # mismatch: try the next border
        if w[k] == w[i]:        # last characters match
            k += 1              # we can increment the border length
        f[i] = k                # we found the maximal border of w[:i + 1]
    return f
```

### *Complexity*

This algorithm consists of a `while` loop nested in a `for` loop, which suggests a quadratic complexity. However, the behaviour of the algorithm should be considered using the example illustrated above. For each comparison $w_k \overset{?}{=} w_i$ in the algorithm, either the word on the bottom is shifted to the right or the window is shifted to the right. Each of these movements can only be executed at most $|w|$ times, which shows the linear complexity in $|w|$: we speak of amortised complexity, as the long iterations are on average compensated by other shorter iterations.

### *Application: Find the Longest Boundary Palindrome*

```
input:  lilipolilil
output: lil     lil
```

A word $x = x_0x_1 \cdots x_{n-1}$ is a palindrome if $x = \overleftarrow{x}$, where $\overleftarrow{x} = x_{n-1} \cdots x_1x_0$. Given a word $x$, we look for the longest palindrome $u$ such that $x$ can be written in the form $x = uvu$ for a word $v$. This problem comes down to seeking the longest boundary of $x\overleftarrow{x}$.

### *Application: Find a Pattern $t$ in the String $s$*

```
input: fragil supercalifragilisticexpialidocious
output:                 ^
```

The most important application of the maximal boundary algorithm is the search for the first occurrence of a pattern $t$ within a string $s$ whose length is greater than $|t|$. The naive approach consists of testing all the possible positions of $t$ with respect to $s$. More precisely, for each position $i = 0 \ldots n - m$ in $s$ we test if the substring $s[i : i + m]$ is equal to $t$. This equality test involves comparing for every $j = 0 \ldots m - 1$ the character $s[i + j]$ with $t[j]$. These two nested loops have complexity $O(nm)$.

The Knuth–Morris–Pratt algorithm selects a letter # occurring neither in $t$ nor in $s$ and we consider the lengths $f_i$ of the maximal boundaries of the prefixes of $w = t\#s$. Note that these boundaries can never be longer than $t$, due to the character #, hence $f_i \leq |t|$. However, if ever $f_i = |t|$, then we have found an occurrence of $t$ in $s$. In the positive case, the answer to the problem is the index $i - 2|t|$: the length of $t$ is subtracted twice from $i$, once to arrive at the start of the boundary and another time to obtain the index in $s$ rather than in $w$, see Figure 2.7. Note that this algorithm is a bit different from the classic presentation of the Knuth–Morris–Pratt algorithm, composed of a pre-computation of the array $f$ of boundaries of $t$ followed by a very similar portion seeking the maximal alignments of $t$ with the prefixes of $s$.

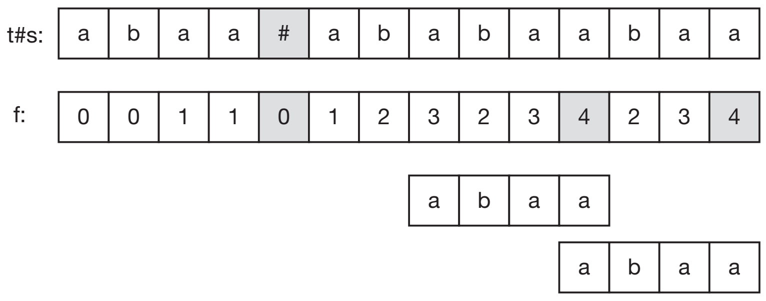

**Figure 2.7** An execution of the maximal boundary search algorithm. Each occurrence of $t$ in $s$ corresponds to a boundary of length $|t|$ in the array $f$ of lengths of maximal boundaries.

```
def knuth_morris_pratt(s, t):
    sep = '\x00'                    # special unused character
    assert sep not in t and sep not in s
    f = maximum_border_length(t + sep + s)
    n = len(t)
    for i, fi in enumerate(f):
        if fi == n:                 # found a border of the length of t
            return i - 2 * n        # beginning of the border in s
    return -1
```

### *Method Provided by the Language Python*

Note that the language Python provides an integrated method `find`, allowing the search of a pattern $t$ in a string $s$, simply with the expression `s.find(t)`. A second optional parameter permits the search to be started at a certain index in $s$. This functionality will be useful in the problem of the determination of the period of a word, presented below.

*Note*

All standard libraries of common programming languages propose a function that looks for a pattern `needle` in a string `haystack`. However, the function provided by Java 10 has worst-case complexity $\Theta(nm)$, which is astonishingly expensive. We invite the reader to measure, for their favourite language, the execution time as a function of $n$ of the search for the pattern $a^n b$ in the string $a^{2n}$, and protest if it is not linear.

*Application: Determine the Largest Integer Power of a Word*

```
input:  blablabla
output: (bla)^{3}
```

Given a word $x$, we would like to identify the largest integer $k$ such that $x$ can be written as $z^k$ for a word $z$. If $k > 1$, then we say that $x$ is not *primitive*.

We can use our Swiss army knife of the computation of boundaries to solve this problem: if $x$ can be written as $y^\ell$, then all the $y^p$ for $p = 0, \ldots, \ell - 1$ are boundaries of $x$. It remains to prove that $y^{\ell-1}$ is the maximal boundary of $x$. So, if $n$ is the length of $u$ and if $n - f_n$ divides $n$, the word is not primitive and the largest value of $k$ we seek is $n/(n - f_n)$, see Figure 2.8.

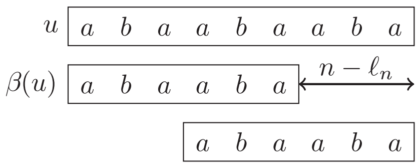

**Figure 2.8** Knowing the maximal boundary of a periodic word allows the determination of its smallest period, here *aba*.

*Proof*

Suppose that $w = z^k$ for $k$ maximal and $\beta(w)$ can be written as $z^{k-1}q$ where $qb = z$ for a certain non-empty $b$. First, note that $z = qb = bq$ (see Figure 2.9), hence $|b| \le |q|$, otherwise $z^{k-1}b$ would be a larger boundary of $w$. Hence, $b$ is a boundary of $z$ and $bz = zb = bqb$. Thus, $bw = wb$, meaning that at the same time $b$ is a boundary of $w$ and $w$ is a boundary of $bw$. We now prove the key observation: as long as $b^\ell$ is smaller that $w$, then $b^\ell$ is a boundary of $w$. We already know this is true for $\ell = 1$, and if it is true for $b^\ell$, then $bb^\ell$ is a boundary of $bw = wb$; hence $b^{\ell+1}$ and $w$ are boundaries of $bw$. This means that either $b^{\ell+1}$ is a boundary of $w$, or $w$ is a boundary of $b^{\ell+1}$. Let $L$ be the largest integer for which $b^L$ is a boundary of $w$: then $w = b^L r = rb^L$ for a word $r$ strictly smaller than $b$. If $r$ is non-empty, then we have found a boundary larger than the maximal boundary $z^{k-1}q$, a contradiction. Hence, $r$ is empty and since $|b| < |z|$, the factorisation $b^L$ is suitable for $L > k$, again a contradiction. See Figure 2.9

```
def powerstring_by_border(u):
    f = maximum_border_length(u)
    n = len(u)
    if n % (n - f[-1]) == 0:       # does the alignment shift divide n ?
        return n // (n - f[-1])    # we found a power decomposition
    return 1
```

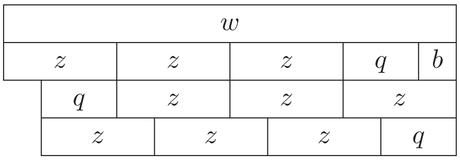

**Figure 2.9** If $w = z^4 = z^3qb$ and we suppose that the maximal boundary of $w$ is $z^3q$, then the boundary property allows us to see that $z = bq = qb$.

Note that a shorter implementation exists using the substring search function integrated with Python, see Figure 2.10.

```
def powerstring_by_find(u):
    return len(x) // (x + x).find(x, 1)
```

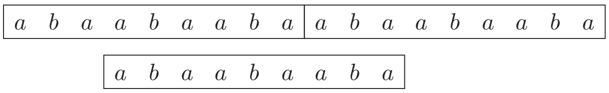

**Figure 2.10** Detection of the first non-trivial position of $x$ in $xx$ allows the identification of its smallest period, here 3.

### *Application: Conjugate of a Word*

```
input:  sweetsour  soursweet
output: sweet|sour sour|sweet
```

Another classic problem consists in detecting whether two words $x$ and $y$ are conjugate, i.e. if they can be written as $x = uv$ and $y = vu$ for words $u$ and $v$. In the affirmative case, we would like to find the decomposition of $x$ and $y$ minimising the length of $u$. This boils down to simply looking for the first occurrence of $y$ in the word $xx$.

### *Problems*

Find the maximal product of string prefixes [codility:carbo2013]
A Needle in the Haystack [spoj:NHAY]
Power strings [kattis:powerstrings]
Period [spoj:PERIOD]

## 2.6 Pattern Matching—Rabin–Karp

### *Complexity*

The expected time is $O(n+m)$, but the worst case complexity is $O(nm)$.

### *Algorithm*

The Rabin–Karp algorithm 1987 is based on a completely different idea from Knuth–Morris–Pratt. To find a pattern $t$ in a large string $s$, we slide a window of length `len(t)` over $s$ and verify if the content of this window is equal to $t$. Since a character-by-character test is very costly, we instead maintain a hash value for the contents of the current window and compare it to the hash value of the search string `t`. If ever the hash values coincide, we proceed to the expensive character-by-character test, see Figure 2.11. To obtain an interesting complexity, it is necessary to efficiently update the hash value as the window shifts. We thus use what is known as a *rolling* hash function.

If the hash function, with values in $\{0,1,\dots,p-1\}$, is well-chosen, we would expect a collision—i.e. when two distinct strings $u,v$ with the same size, selected uniformly, give the same hash value—to occur with probability on the order of $1/p$. In this case, the mean complexity of the algorithm is $O(n+m+m/p)$. Our implementation uses $p$ on the order of $2^{56}$, hence in practice the complexity is $O(n+m)$, but in the worst case, it is $O(nm)$.

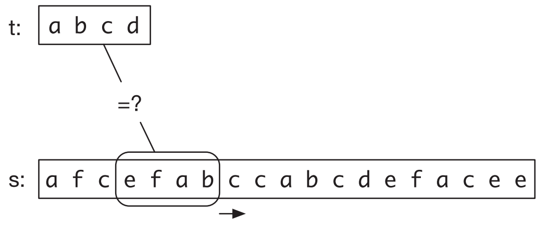

**Figure 2.11** The idea of the Rabin–Karp algorithm is to first compare the hash values between $t$ and a window on $s$ before performing a costly character-by-character comparison.

### *Rolling Hash Function*

Our hash function first transforms an $m$-character string into $m$ integers $x_0,\dots,x_{m-1}$, corresponding to their ASCII codes, lying between 0 and 127. The value of the hash function is then the multilinear expression

$$h(x_0,\dots,x_{m-1}) = (x_0\cdot 128^{m-1} + x_1\cdot 128^{m-2} + \cdots + x_{m-2}\cdot 128 + x_{m-1}) \bmod p$$

where all of the operations are performed modulo a large prime number $p$. In practice, care should be taken so that all of the values calculated can be manipulated on a 64-bit machine, where a machine word (CPU register) contains signed integer values between $-2^{63}$ and $2^{63}-1$. The largest intermediate value calculated by the algorithm is $128\cdot(p-1) = 2^7\cdot(p-1)$, hence for our implementation we have chosen $p < 2^{56}$.

The polynomial form of this hash function allows us to calculate in constant time the value of $h(x_1,\dots,x_m)$ in terms of $x_0, x_m$ and $h(x_0,\dots,x_{m-1})$: removing the first character amounts to subtracting the first term, shifting the characters to the left corresponds to a multiplication by 128 and modifying the last character is done by adding a term. Consequently, shifting the window on $s$, updating the hash value and comparing it with that of $t$ takes constant time.

Note in the code below the addition of the value `DOMAIN * PRIME` (which is of course 0 mod $p$) to ensure that the calculations remain in the non-negative integers. This is not strictly necessary in Python but is required in languages such as C++, where the calculation of modulo can return a negative number.

```
PRIME = 72057594037927931     # < 2^{56}
DOMAIN = 128


def roll_hash(old_val, out_digit, in_digit, last_pos):
    val = (old_val - out_digit * last_pos + DOMAIN * PRIME) % PRIME
    val = (val * DOMAIN) % PRIME
    return (val + in_digit) % PRIME
```

The implementation of the algorithm begins with a function to compare character-by-character factors of length $k$ in $s$ at the position $i$ and in $t$ at the position $j$.

```
def matches(s, t, i, j, k):
    # tests if s[i:i + k] equals t[j:j + k]
    for d in range(k):
        if s[i + d] != t[j + d]:
            return False
    return True
```

Next, the implementation of the Rabin–Karp algorithm itself begins with the computation of the hash values of $t$ and of the first window on $s$, followed by a loop over all the factors of $s$, until a match is found.

```
def rabin_karp_matching(s, t):
    hash_s = 0
    hash_t = 0
    len_s = len(s)
    len_t = len(t)
    last_pos = pow(DOMAIN, len_t - 1) % PRIME
    if len_s < len_t:              # substring too long
        return -1
    for i in range(len_t):         # preprocessing
        hash_s = (DOMAIN * hash_s + ord(s[i])) % PRIME
        hash_t = (DOMAIN * hash_t + ord(t[i])) % PRIME
    for i in range(len_s - len_t + 1):
        if hash_s == hash_t:               # hashes match
            # check character by character
            if matches(s, t, i, 0, len_t):
                return i
        if i < len_s - len_t:
            # shift window and calculate new hash on s
            hash_s = roll_hash(hash_s, ord(s[i]), ord(s[i + len_t]),
                               last_pos)
    return -1                              # no match
```

This algorithm is less efficient than that of Knuth–Morris–Pratt: our experiments measured roughly a factor of 3 increase in the computation time. Nevertheless, its interest lies in the fact that its technique can be applied to solve several interesting variants of this problem.

### *Variant: The Search for Multiple Patterns*

The Rabin–Karp algorithm which verifies if a string $t$ is a factor of a given string $s$ can naturally be generalised to a set $\mathcal{T}$ of strings to be found in $s$, in the case where all of the strings of $\mathcal{T}$ have the same length. It suffices to store the hash values of the strings in $\mathcal{T}$ in a dictionary `to_search` and for each window on $s$ to verify whether the associated hash value is contained in `to_search`.

### *Variant: Common Factor*

Given two strings $s,t$ and a length $k$, we look for a string $f$ of length $k$ that is at the same time a factor of $s$ and of $t$. To solve this problem, we first consider all of the factors of length $k$ of the string $t$. These substrings are obtained in an analogous method to the Rabin–Karp algorithm, by sliding a window of size $k$ across $t$ and storing the resulting hash values in a dictionary `pos`. With each hash value, we associate the start positions of the corresponding windows.

Next, for each factor $x$ of $s$ of length $k$, we verify whether the hash value $v$ is in `pos`, in which case we compare character-by-character $x$ with the factors in $t$ at the positions of `pos[v]`.

For this algorithm, we must pay attention to the choice of the hash function. If $s$ and $t$ are of length $n$, it is necessary to choose $p \in \Omega(n^2)$, so that the number of collisions between the hash values of one of the $O(n)$ windows of $t$ with one of the $O(n)$ windows of $s$ is roughly constant. For a precise analysis, see Karp and Rabin (1987).

### *Variant: Common Factor with Maximal Length*

Given two strings $s,t$, finding the longest common factor can be done with a binary search on the length $k$ by using the previous algorithm. The complexity is $O(n \log m)$ where $n$ is the total length of $s$ and $t$, and $m$ is the length of the optimal factor.

```
def rabin_karp_factor(s, t, k):
    last_pos = pow(DOMAIN, k - 1) % PRIME
    pos = {}
    assert k > 0
    if len(s) < k or len(t) < k:
        return None
    hash_t = 0


    # First calculate hash values of factors of t
    for j in range(k):
        hash_t = (DOMAIN * hash_t + ord(t[j])) % PRIME
    for j in range(len(t) - k + 1):
        # store the start position with the hash value
        if hash_t in pos:
            pos[hash_t].append(j)
        else:
            pos[hash_t] = [j]
        if j < len(t) - k:
            hash_t = roll_hash(hash_t, ord(t[j]), ord(t[j + k]), last_pos)


    hash_s = 0
    # Now check for matching factors in s
    for i in range(k):          # preprocessing
        hash_s = (DOMAIN * hash_s + ord(s[i])) % PRIME
    for i in range(len(s) - k + 1):
        if hash_s in pos:       # is this signature in s?
            for j in pos[hash_s]:
                if matches(s, t, i, j, k):
                    return (i, j)
        if i < len(s) - k:
            hash_s = roll_hash(hash_s, ord(s[i]), ord(s[i + k]), last_pos)
    return None
```

*Problem*
Longest Common Substring [spoj:LCS]

## 2.7 Longest Palindrome of a String—Manacher

```
input: babcbabcbaccba
output:  abcbabcba
```

*Definition*
A word $s$ is a palindrome if the first character of $s$ is equal to the last, the second is equal to the next-to-last and so on.

The problem of the longest palindrome consists in determining the longest factor that is a palindrome.

*Complexity*
This problem can be solved in quadratic time with the naive algorithm, in time $O(n \log n)$ with suffix arrays, but in time $O(n)$ with Manacher's algorithm (1975), described here.

*Algorithm*
First, we transform the input $s$ by inserting a separator `#` around each character and by adding sentinels `^` and `$` around the string. For example, `abc` is transformed into `^#a#b#c#$`. Let $t$ be the resulting string. This allows us to process in an equivalent manner palindromes of both even and odd length. Note that with this transformation, every palindrome begins and ends with the separator `#`. Thus, the two ends of a palindrome have indices with the same parity, which simplifies the transformation of a solution on the string $t$ into one on the string $s$. The sentinels avoid special care for the border cases.

The word `nonne` contains a palindrome of length 2 (`nn`) and one of length 3 (`non`). Their equivalents in

```
 |-----|
^#n#o#n#n#e#$
     |---|
```

all begin and end with the separator `#`.

The output of the algorithm is an array $p$ indicating for each position $i$, the largest radius $r$ such that the factor from $i - r$ to $i + r$ is a palindrome. The naive algorithm is the following: for each $i$, we initialise $p[i] = 0$ and increment $p[i]$ until we find the longest palindrome $t[i - p[i], \ldots, i + p[i]]$ centred on $i$.

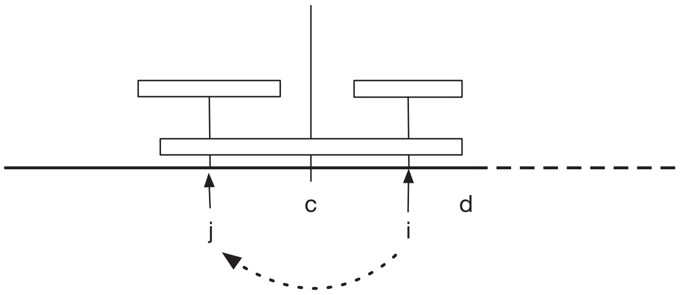

**Figure 2.12** Manacher's algorithm. Having already computed $p$ for the indices $< i$, we wish to compute $p[i]$. Suppose there is a palindrome centred on $c$ of radius $d - c$ with $d$ maximal, and let $j$ be the mirror image of $i$ with respect to $c$. By symmetry, the palindrome centred on $j$ of radius $p[j]$ must be equal to the word centred on $i$, at least up to the radius $d - i$. Consequently, $p[j]$ is a lower bound for the value $p[i]$.

Manacher's improvement concerns the initialisation of $p[i]$. Suppose we already know a palindrome centred on $c$ with radius $r$, hence terminating on the right at

$d = c + r$. Let $j$ be the mirror image of $i$ with respect to $c$, see Figure 2.12. There is a strong relation between $p[i]$ and $p[j]$. In the case where $i + p[j]$ is not greater than $d$, we can initialise $p[i]$ by $p[j]$. This is a valid operation, as the palindrome centred on $j$ of radius $p[j]$ is included in the first half of the palindrome centred on $c$ and of radius $d - c$; hence it is also found in the second half.

After having computed $p[i]$, we must update $c$ and $d$, to preserve the invariant that they code a palindrome with $d - c$ maximal. The complexity is linear, since each comparison of a character is responsible for an incrementation of $d$.

```
def manacher(s):
    assert set.isdisjoint({'$', '^', '#'}, s)  # Forbidden letters
    if s == "":
        return (0, 1)
    t = "^#" + "#".join(s) + "#$"
    c = 1
    d = 1
    p = [0] * len(t)
    for i in range(2, len(t) - 1):
        #                        -- reflect index i with respect to c
        mirror = 2 * c - i         # = c - (i-c)
        p[i] = max(0, min(d - i, p[mirror]))
        #                        -- grow palindrome centered in i
        while t[i + 1 + p[i]] == t[i - 1 - p[i]]:
            p[i] += 1
        #                        -- adjust center if necessary
        if i + p[i] > d:
            c = i
            d = i + p[i]
    (k, i) = max((p[i], i) for i in range(1, len(t) - 1))
    return ((i - k) // 2, (i + k) // 2)  # extract solution
```

## *Application*

A man is walking around town, and his smartphone registers all of his movements. We recover this trace and seek to identify a certain type of trajectory during the day, notably round trips between two locations that use the same route. For this, we extract from the trace a list of street intersections and check it for palindromes.

## *Problems*

Longest Palindromic Substring [spoj:LPS]
Casting Spells [kattis:castingspells]