Chapter 2

Strings and Pattern Matching

As explained in section, 1.1, strings form one of Scheme's built-in compound data types. In this chapter we discuss the string data structure in more detail. We present an overview of the most important procedures that have been built into Scheme for manipulating strings.

The bulk of our attention is devoted to the study of algorithms designed to find some string (called a pattern) in another string (called a text) that is typically much longer than the first one. Finding patterns in texts is also known as *pattern matching*. Several dozens of pattern matching algorithms have been invented. Most of them exhibit different performance characteristics depending on the properties of the pattern and the text. Studying a representative selection of these algorithms is the central theme of this chapter. At the end of the chapter, we present an algorithm — called QuickSearch — that doesn't necessarily have the best worst-case performance characteristic but that seems to beat all other algorithms in practice. Let us first start by establishing some terminology that allows us to talk about strings.

2.1 Strings in Scheme

A string is a compound data element of the data type `string` which is built into Scheme. Strings are finite sequences of characters. The simplest way to get hold of a string is to use the literate constructor "..." as explained in section 1.1. For example, "Madam, I'm Adam" is a string consisting of 14 characters including two whitespace characters. However, "..." is not the only way to create new strings. Procedural constructors `make-string` and `string` can be used as well. For example, given the character `#\a`, then (`make-string 10 #\a`) creates a string that consists of 10 a's. Given a list of characters, the `string` procedure can be used to create a string by juxtaposing the characters contained in the list. E.g., (`string #\S #\c #\h #\e #\m #\e`) creates a new string that is

identical to the string obtained by evaluating the literate expression "Scheme".

Having constructed strings using one of these constructors, Scheme's primitive procedures `string-length` and `string-ref` act as accessors for strings. `string-length` returns the *length* of a string. The length is defined as the number of characters contained by the string. For example, (`string-length` "hello") returns 5. Using a slightly more mathematical notation, we denote the length of a string s by $|s|$. For example $|$`''hello''`$| = 5$. The *empty string* is a string that has no characters. It is denoted by "" and its length is — by definition — zero. `string-ref` is used to access the individual characters of a string given an index i which denotes the location of the character in the string. i may vary from 0 to $|s| - 1$. E.g., given the string (`define my-string` "Scheme"), then the expression (`string-ref my-string 3`) evaluates to #\e. Strings are said to be *immutable data values*. This means that it is impossible to change the composition of a string after construction[1].

After having described the constructors and accessors for strings, let us now have a look at the operations Scheme provides on the `string` data type. Below we merely list a few operations that are extremely frequently used. For a complete list, we refer to the R6RS.

Conversion Operations: Two primitive procedures can be used to convert strings to lists of characters and the other way around. E.g., (`string->list` "Scheme") results in the Scheme list (#\S #\c #\h #\e #\m #\e). Conversely, (`list->string` (`list` #\a #\S #\t #\r #\i #\n #\g)) yields the string "aString". Both operations are in $O(n)$ where n is the length of the list or the length of the string at hand. This is because the Scheme evaluator has to process the entire sequence of characters in both cases.

Comparison Operations: The following procedures can be used to compare strings with one another. String comparison can be done in two ways. *Case sensitive comparison* makes a distinction between upper case and lower case characters. *Case insensitive comparison* does not distinguish between upper case and lower case characters. Whether or not a comparison procedure distinguishes cases is reflected by the presence or absence of `ci` (= case insensitive) in the name of the procedure. The table in figure 2.1 shows a complete overview of Scheme's string comparison procedures.

These comparison procedures are based on the *lexicographic ordering* that is defined for strings. In general, this means that shorter strings come before longer strings that start with the same sequence of characters. For example "hello" comes before "hello `world`". Furthermore, it means that #\a comes before #\b as one would expect. The exact order for individual characters is prescribed by an extension[2] of the ASCII code — the American Standard Code for Information Interchange. This code assigns

[1] Since R6RS, Scheme's standard libraries feature so-called mutable strings as well. We refer to the R6RS for more details.

[2] The extension is called the Unicode standard. Again, we refer to the R6RS for more details.

String comparison operation	Functionality
(`string=? s1 s2`)	String equality
(`string-ci=? s1 s2`)	String equality, case insensitive
(`string<? s1 s2`)	String before
(`string>? s1 s2`)	String after
(`string<=? s1 s2`)	String before-equal
(`string>=? s1 s2`)	String after-equal
(`string-ci<? s1 s2`)	String before, case insensitive
(`string-ci>? s1 s2`)	String after, case insensitive
(`string-ci<=? s1 s2`)	String before-equal, case insensitive
(`string-ci>=? s1 s2`)	String after-equal, case insensitive

Figure 2.1: Scheme's string comparison procedures

an number between 0 and 255 to all regularly occurring characters. The Scheme procedures `char->integer` and `integer->char` can be used to convert characters to their ASCII value and the other way around. For example, (`char->integer` #\`a`) yields 97 and (`integer->char` 65) yields #\A. These ASCII values are used to define the lexicographic ordering: a character `c1` is "smaller" than a character `c2` if (< (`char->integer c1`) (`char->integer c2`)) where < is the usual Scheme procedure for comparing numbers.

String addition and substraction: Strings can be "added" using the `string-append` procedure. It takes an arbitrary number of argument strings and it produces a new string that is the concatenation (also known as the juxtaposition) of the argument strings. E.g., (`string-append` "peek-" "a" "-boo") yields the string "peek-a-boo". Strings can also be "substracted" using the `substring` procedure. `substring` takes three arguments: a string, a starting index and an end index. It "reads" the characters from the string, starting at the start index and ending and the end index. It returns a *new* string that consists of the corresponding characters in the input string. For example, the expression (`substring "Scheme is the greatest language!" 14 22`) yields "greatest". `substring` does not modify its argument string.

This concludes our overview of Scheme's built-in string processing procedures. This overview is far from complete and the R6RS lists a huge number of built-in string processing procedures that are worthwhile studying whenever one has to deal with strings when writing Scheme applications.

There is one important aspect of string processing that is not included in Scheme's list of built-in string processing procedures. It is known as the *pattern matching problem* and it is a frequently occurring problem in computer science. The reason for not including a default Scheme procedure that solves the pattern matching problem in the list of standard Scheme procedures is that there exists no standard solution to the problem. Different solutions each have their

advantages and disadvantages. A study and comparison of these solutions to the pattern matching problem is the topic of the rest of this chapter.

2.2 The Pattern Matching Problem

The pattern matching problem is formulated as follows. Suppose that we have a string `t` (also known as "the text" or "the haystack") and suppose that we have another string `p` (also known as "the pattern" or "the needle"). The pattern matching problem is the question of finding a procedure (`match t p`) that is capable of computing an index `i` such that (`string=? p (substring t i (+ i (string-length p)))`). In other words, we are looking for the index i in `t` that corresponds to the location of `p` in `t`. It is said that `t` *matches* `p` *at position* i. The position is also known as the *offset* or *shift* of `p` in `t`.

For example, if the haystack is "`madam, I'm adam`" and the needle is "`adam`" then `match` should return 1 since "`adam`" occurs in the haystack starting at index 1 (remember that string indexes start counting from 0). This example shows that a text `t` may match a pattern `p` several times at various offsets. We speak about different *occurrences* of the pattern in the text. Clearly, if we have developed an algorithm that is capable of finding a single occurrence of the pattern in the text, then we also have an algorithm to find multiple occurrences in the text. All we have to do is apply the original algorithm multiple times.

The pattern matching problem has an obvious application in word processing programs. Anyone who has ever used such a program is acquainted with the "find" command which allows one to look for a pattern in the text file one is editing. In this application of pattern matching, the pattern usually consists of a small sequence of characters. More recently, the pattern matching problem has also found applications in bioinformatics, a new branch of computer science that uses computer science techniques in the context of biology and biotechnology. One frequently occurring problem in bio-informatics consists of finding sequences of genetic codes in long strings of DNA. This is a formulation of the pattern matching problem where the "text" consist of millions of characters whilst the "pattern" (i.e. the genetic code that is searched for) consists of a few hundreds of thousands of "characters". This example shows that, in contrast to the "find" command customary found in text processors, patterns can be quite long as well.

Before we start exploring the realm of pattern matching algorithms, we need to develop some additional terminology and notation that can be used to talk about strings in precise ways. Given a string `s`, then a string `e` is said to be a *prefix* of `s` if there exists a string `u` such that (`string=? (append e u) s`). `u` is then said to be a *suffix* of `s`. If `e` is a non-empty string (i.e. $|\texttt{e}| > 0$) and $\texttt{e} \neq \texttt{s}$ then `e` is said to be a *proper prefix* of `s`. Similarly, if `u` is a non-empty string (i.e. $|\texttt{u}| > 0$) and $\texttt{u} \neq \texttt{s}$, then `u` is said to be a *proper suffix* of `s`. In what follows, we shall mean "proper prefix" (resp. proper suffix) whenever we say "prefix" (resp. suffix). Sometimes a slightly more mathematical notation is preferred. Whenever we want to express that a string s consists of two

parts, u and v, then we write $s = u.v$. In other words, the dot notation is a mathematical shorthand for Scheme's `string-append`. In what follows, we sometimes need a prefix of a string s that is exactly k characters long. We shall indicate such prefix by $s_{0\rightarrow k-1}$. Similarly, a suffix consisting of k characters will be referred to as $s_{|s|-k\rightarrow|s|-1}$. The k'th character of the string s is simply denoted by s_k. For example, let us consider the string s of the form "Madam, I'm Adam". Then $s_{0\rightarrow 4}$ is the string "Madam". It is a prefix of length 5. Similarly, $s_{15-4\rightarrow 15-1} = s_{11\rightarrow 14} =''$ Adam$''$ is a suffix of s of length 4. $s_6 = \#\backslash$space is the 6'th character of s.

2.3 The Brute-Force Algorithm

We start our study of pattern matching algorithms by presenting the "brute-force algorithm". The name of the algorithm comes from the fact that it doesn't use any clever tricks to speed up the matching process in any way. Instead it simply considers *all* potential matches, one by one, until a match has been found. The brute-force algorithm is simple to program which is probably the reason why it is not a very efficient algorithm.

The following procedure is an implementation of the brute-force algorithm in Scheme. The procedure takes a text `t` and a pattern `p` as arguments. It either returns #f when the pattern does not occur in the text, or a number indicating the offset of the pattern in the text.

```
(define (match t p)
  (define n-t (string-length t))
  (define n-p (string-length p))
  (let loop
    ((i-t 0)
     (i-p 0))
    (cond
      ((> i-p (- n-p 1))
       i-t)
      ((> i-t (- n-t n-p))
       #f)
      ((eq? (string-ref t (+ i-t i-p)) (string-ref p i-p))
      (loop i-t (+ i-p 1)))
      (else
       (loop (+ i-t 1) 0))))))
```

Before explaining and analysing this procedure, let us first introduce some additional terminology and notation. We often need to refer to the length of `t` (resp. `p`). Depending on the font used, it is denoted by `n-t` (resp. `n-p`) or n_t (resp. n_p). All algorithms presented in this chapter use two index variables, namely `i-t` and `i-p`. They are used in iterations in order to denote the offset of the pattern in the text and the index used to designate a character inside the pattern. Depending on the font used, these numbers are denoted by i_t (resp. i_p) or

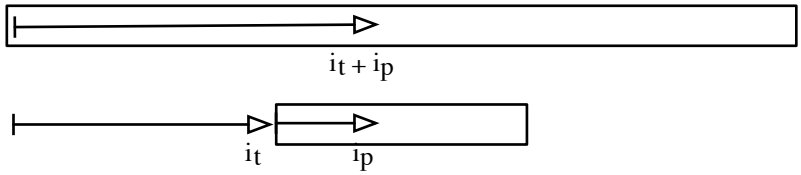

Figure 2.2: Indexes in String Matching Algorithms

`i-t` (resp. `i-p`). The particular values of these index variables are also referred to as *an alignment* of the pattern and the text. Figure 2.2 shows an alignment of a pattern and the text at offset i_t. It shows a phase in the algorithm where the $(i_t + i_p)$'th character of the text is compared with the i_p'th character of the pattern. In the description of the executions of the algorithms, we often refer to "the current characters" in a given alignment. This is the couple of characters consisting of the character that resides in location `i-p` in `p` and the one residing in location (+ `i-t i-p`) in `t`. In other words, the current characters is the couple of characters that are compared in a one particular iteration of the pattern matching procedure. When the current characters do not match, we speak of a *mismatch*.

During the execution of our algorithms, we say that "we move the pattern to the right" whenever we increment `i-t` by some amount. For example, moving the pattern one position to the right means that we replace `i-t` by (+ `i-t 1`) in the next iteration of the loop. Clearly, the quality of an algorithm can be measured by the amount of positions that we can move to the right in one single iteration of the loop without overshooting potential successful alignments.

The brute-force algorithm is a Scheme loop that makes `i-t` vary between `0` and the last offset where an occurrence of the pattern could possibly start, i.e. (− `n-t n-p`). For every value of `i-t`, `i-p` varies from `0` to (− `n-p 1`). The conditional in the body of the loop has four branches:

- The first test checks whether the algorithm has consumed the entire pattern. If this is the case, then the previous iteration of the loop has successfully compared the last character of the pattern with its corresponding character in the text. This means that we have found a match at offset `i-t` which is the location of the first character in the text against which we are aligning `p`.

- If the first test has failed (i.e., the pattern hasn't been entirely matched yet), then we check to see whether we have consumed the entire text. If this is the case, the text has been entirely consumed without having found a match. As a result, we return #f.

- Having checked both the pattern and the text against their boundaries, the algorithm subsequently checks whether the current character in `p` matches

the corresponding character in `t`. If this is indeed the case, we simply continue the loop by considering the next pattern character in the current alignment. This is done by incrementing `i-p` by one in the next iteration of the loop.

- If the current characters do not match, we start from scratch by trying to match the pattern with the next alignment in `t`. In other words, we cycle through the loop with the call (`loop` (+ `i-t 1`) `0`). This resets `i-p` to zero such that the process of checking the pattern restarts entirely.

Performance

What can we say about the efficiency of the brute-force algorithm? From the code we can see that in the worst case, the loop is executed for i_t varying from 0 to $n_t - i_t$ *and* that for every such i_t, it is being executed with i_p varying from 0 to $n_p - 1$. This means that the loop is executed $O(n_t.n_p)$ times[3]. Since the body of the look is in $O(1)$, we thus conclude that the brute-force algorithm has a worst-case performance characteristic that is in $O(n_t.n_p)$. In practice, the brute-force algorithm does not perform too bad in simple cases where the pattern is extremely small (say, a few characters). In other words, if n_p is small, then the brute-force algorithm exhibits linear behaviour. This situation occurs quite frequently when launching the "find" command in word processors. However, when used with large values for n_t and n_p (such as in bioinformatics applications), the brute-force algorithm has a horrible performance.

2.4 The Knutt-Morris-Pratt Algorithm

The reason why the brute-force algorithm is so slow is that, whenever a mismatch occurs, it only shifts the pattern one single position to the right. Moreover, after doing so it reconsiders the entire pattern from scratch, i.e. starting at index 0. This can be sped up considerably. Several algorithms have been invented to do so. One of the best known algorithms is the Knutt-Morris-Pratt algorithm (or KMP for short). The KMP algorithm was discovered in 1977 by Knuth and Pratt, and independently by Morris. The general idea of the KMP algorithm is as follows.

In order to explain the KMP algorithm, let us refer to figure 2.3(a) which shows a non-matching alignment of a pattern and a text. The figure shows the situation where a prefix *part* of the pattern has been found to match part the text. It also assumes that the first character of *rest*1 does not match the first character of *rest*2. In other words, the text is of the form *pre.part.rest*1.*post* and the pattern is of the form *part.rest*2. For example, given the text "Madam, `I'm Adam`" and given the pattern "`I'm cool!`", then *part* corresponds to "`I'm `" (including the whitespace), *res*1 corresponds to "Adam" and *res*2 corresponds to "cool".

[3]Remember from section 1.5.2 that we can omit constant factors like a in product expressions like $O(a.f)$. However, in the expression $O(n_t.n_p)$ none of the factors is a constant. Both variables depend on the input strings.

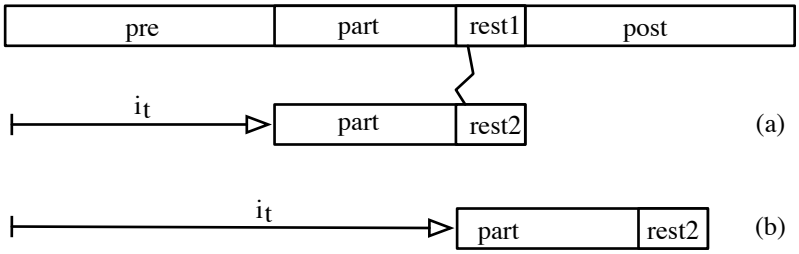

Figure 2.3: Basic (naive) idea of the KMP algorithm

The basic idea of KMP is depicted in figure 2.3(b). The idea is to shift the pattern to the right in order to realign it with *rest*1. In other words, we make the first character of the pattern align with the character of the text that gave rise to the mismatch. In our example, this would align the pattern "l'm `cool`" with "Adam" instead of "'m `Adam`" as would be done by the brute-force algorithm. In other words, by aligning the pattern with the character in the text that gave rise to the mismatch, we shift the pattern 3 positions to the right instead of just 1. We will show that this solution is not entirely correct. Nevertheless, this basic idea is important to understand the KMP algorithm. We shall henceforth refer to it as the naive KMP.

The reason why this is a naive solution, is that we can have repetition in the pattern. In order to understand the problem, let us have a look at figure 2.4(a). The grey zones in the pattern and the text indicate repetitions of a certain character sequence. As we can see from figure 2.4(b), we have shifted the pattern too far because the second occurrence of the grey zone *might* be the start of a successful alignment. By applying the naive algorithm, we have been *overshooting* a potential match. The reason is that the first occurrence of the grey zone in the pattern is moved *beyond* the second occurrence of the grey zone in the text such that its second occurrence in the text will never again be aligned with the first occurrence in the pattern. The correct shift is shown in figure 2.4(c).

For example, in the pattern "lala `or lalala`", a grey zone might be "la" or even "lala". Let us apply the aforementioned naive algorithm to this pattern and to the text "lalala `or lalalala is what I like to sing`". We start at $i_t = 0$ and $i_p = 0$ and we soon find that the prefix "lala" matches but that the third `l` of the text doesn't match the whitespace in the pattern. Applying the naive KMP algorithm would align the pattern against the third `l` (i.e. $i_t = 4$) thereby clearly overshooting the solution that we get when aligning the pattern using the offset $i_t = 2$. Notice that this problem only occurs when there is repetition in the pattern. The amount of characters overshot is indicated by k in figure 2.4(c). Notice that k only depends on the pattern and not on the text: the length of the repetitions can easily be determined by analysing the pattern

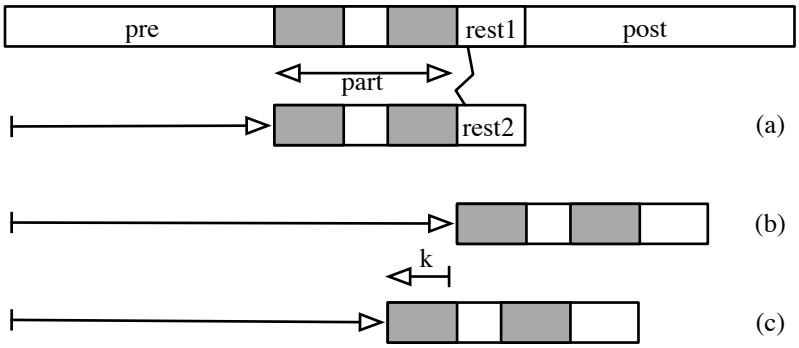

Figure 2.4: Patterns containing repetition

before the actual pattern matching algorithm starts. This *preprocessing phase* is the essence of the actual KMP algorithm.

The following Scheme procedure implements the Knuth-Morris-Pratt algorithm. The structure of the algorithm is identical to the structure of the brute-force algorithm. It is a loop, the body of which consists of four branches: either we have processed the entire pattern successfully, or we have consumed the entire text without finding an occurrence of the pattern, or the current alignment keeps on being successful, or — the `else` branch — a mismatch is encountered. The KMP algorithm differs from the brute-force algorithm in the fourth branch. Instead of shifting the pattern a single character to the right after a mismatch (i.e. $i_t + 1$), we shift it by i_p (as indicated by the naive solution) *minus* an amount that corresponds to the k explained above. We will also denote this k by $\sigma(i_p)$. In the Scheme code, this amount is obtained by calling (`sigma i-p`). `sigma` returns the amount of characters that *cannot* be skipped after a mismatch of p's i_p'th character without running the risk of overshooting a solution. `sigma` is known as the *failure function* for the given pattern. As already said, it is determined *before* the pattern matching loop is started. Assuming that we have a procedure `compute-failure-function` that determines σ, the KMP algorithm can be programmed as shown below. By convention we define (`sigma 0`) to be -1. This makes the fourth branch of the KMP algorithm evaluate the expression (+ `i-t` (− `i-p` (`sigma i-p`))) to (+ `i-t 1`) whenever `i-p` is 0. In other words, just like the brute-force algorithm, the KMP algorithm shifts the pattern one single position to the right whenever no characters match.

```
(define (match t p)
  (define n-t (string-length t))
  (define n-p (string-length p))
  (define sigma (compute-failure-function p))
  (let loop
    ((i-t 0)
```

```
 (i-p 0))
(cond
  ((> i-p (- n-p 1))
   i-t)
  ((> i-t (- n-t n-p))
   #f)
  ((eq? (string-ref t (+ i-t i-p)) (string-ref p i-p))
   (loop i-t (+ i-p 1)))
  (else
   (loop (+ i-t (- i-p (sigma i-p))) (if (> i-p 0)
                                         (sigma i-p)
                                         0)))))))
```

Apart from shifting the pattern more than one character to the right, the KMP algorithm also differs from the brute-force algorithm in that it skips some prefix of the pattern. This is taken care of by the final `if`-test in the reiteration. If `i-p` did not increase in the looping process (i.e. it is equal to zero), then none of the pattern's characters have been checked yet. This means that we still have to check the entire pattern in the next iteration. In other words, `i-p` has to stay zero. However, if `i-p` is strictly bigger than zero (i.e. at least one character of the pattern matches the text in the current alignment), then we do not need to reconsider those characters and we take `i-p` to be (`shift i-p`) in the next alignment. They correspond to the grey zone in figure 2.4 about which we already know that it matches the text in the new alignment that will be checked in the reiteration. This is explained further below. Let us now first have a look at an example to understand the behaviour of the pattern matching procedure.

Suppose that we run the procedure as follows:

```
(match "I'm singing lalala down in lalaland" "lalaland")
```

The listing shown below shows the evolution of the indices i_t and i_p throughout the evaluation of the algorithm. The behaviour of the procedure shows that i_t gradually climbs to the value 12 where a first near-match occurs. The near-match is 6 characters long (namely "lalala"). When the algorithm subsequently finds out that #\n does not match #\space, it doesn't simply restart the matching process at $i_t = 13$ which is what the brute-force algorithm would do. Instead, it tries to align the pattern with the text at position $i_t + i_p = 18$, minus the amount of characters in the pattern that might be overshot. This happens to be 4 since "lalaland" aligns with "lalala down..." by shifting the former 4 characters to the left starting from the non-matching whitespace. Therefore the matching process restarts at $18 - 4 = 14$, i.e. "lala down...". But also notice that i_p does not restart from 0 either. Since we have shifted the pattern 4 characters to the left (because we know they will match), we don't need to reconsider them again. Hence the matching restarts at $i_p = 4$, i.e. we immediately proceed by checking the third #\l of "lalaland" against its corresponding character in the text. In other words, we match "lalaland" against "lala down..." starting

at $i_p = 4$. Then #\l in the pattern does not match the #\space. Again, i_t is set to $i_t + i_p - k = 14 + 4 - 2 = 16$ because realigning the start of the pattern with the character causing the mismatch (i.e. #\space) overshoots the second occurrence of "la" in "lala down...". In order to avoid the overshooting, we have to substract 2 from this naive realignment. Hence, the next alignment is against "la down..." (i.e. $i_t = 16$). Furthermore, since we just checked the "la", we don't need to check this again. Hence, $i_p = 2$. This reasoning is repeated once more for this new alignment with $i_t + i_p - k = 16 + 2 - 0$ and $i_p = 0$. From that point on, the algorithm gradually makes i_t climb to 27 where a successful match is found.

```
i-t = 0      i-p = 0
i-t = 1      i-p = 0
i-t = 2      i-p = 0
i-t = 3      i-p = 0
i-t = 4      i-p = 0
i-t = 5      i-p = 0
i-t = 6      i-p = 0
i-t = 7      i-p = 0
i-t = 8      i-p = 0
i-t = 9      i-p = 0
i-t = 10     i-p = 0
i-t = 11     i-p = 0
i-t = 12     i-p = 0, 1, 2, 3, 4, 5, 6
i-t = 14     i-p = 4
i-t = 16     i-p = 2
i-t = 18     i-p = 0
i-t = 19     i-p = 0
i-t = 20     i-p = 0
i-t = 21     i-p = 0
i-t = 22     i-p = 0
i-t = 23     i-p = 0
i-t = 24     i-p = 0
i-t = 25     i-p = 0
i-t = 26     i-p = 0
i-t = 27     i-p = 0, 1, 2, 3, 4, 5, 6, 7, 8
```

Let us now explain how to obtain the `sigma` function that contains the information about repetitions in the pattern. Consider the following hypothetical situation during a particular execution of the KMP matching algorithm:

```
text    = b a b b a b b a b b a b c
pattern = b a b b a b b a b c
```

We observe that the pattern matches the text for 9 characters, after which we encounter a mismatch of `b` against `c`. The naive solution would simply shift the pattern in order to make its first character align with the character in the text that gave rise to the mismatch:

```
text    = b a b b a b b a b b a b c
pattern =                   b a b b a b b a b c
```

However, as we have explained, this is too naive a solution. Since the pattern contains internal repetition, some characters need to be reconsidered in order not to overshoot the solution. At this point we have two options. In the first option, we shift the pattern three characters to the left w.r.t. the positioning of the naive solution. In the second option, we shift the pattern six characters to the left w.r.t. that positioning. They are both shown below.

```
text    = b a b b a b b a b b a b c
pattern =             b a b b a b b a b c
pattern =       b a b b a b b a b c
```

Clearly, the second option is the correct one since the first option gives rise to an overshot solution. The point is that we have to shift the pattern *the maximal amount* of characters to the left in order to be sure. This is the k we were intuitively referring at in figure 2.4. Hence, given a shift i_t and given the fact that i_p characters successfully match (but the $(i_p + 1)$'th character residing at index i_p does not match), then we have to shift the pattern to position $i_t + i_p - k = i_t + i_p - \sigma(i_p)$ for the maximal $k = \sigma(i_p)$. If none of the characters match, we have $i_p = 0$ which requires $\sigma(i_p) = -1$ (such that $i_t + i_p - \sigma(i_p) = i_t + 0 - (-1)$). Hence, if none of the characters match, we shift the pattern one position to the right, exactly like the brute-force algorithm. Let us now figure out how to come up with the other values for σ.

Since we shift the pattern k characters to the left (w.r.t. the naive solution) and since we just finished the previous iteration of the loop by successfully checking those characters against the text, this necessarily implies that the first k characters of the pattern have to be identical to the last k characters in the part that was just checked, i.e. in $p_{0 \to i_p - 1}$. Hence, k is the length of a prefix of p that is also a suffix of $p_{0 \to i_p - 1}$. But because we need to shift the pattern the *maximal* number of positions to the left in order not to overshoot any match, we want $k = \sigma(i_p)$ to be the length of the *longest* such prefix of p that is also a suffix of $p_{0 \to i_p - 1}$. This also explains why there is no need to reconsider those $\sigma(i_p)$ characters in the next alignment. We know that they have already been checked against the text (since they are a suffix of the part which we just finished checking). Hence, i_p can safely restart from $\sigma(i_p)$ in the next iteration.

The following Scheme procedure establishes σ for any given pattern `p`.

```
(define (compute-failure-function p)
  (define n-p (string-length p))
  (define sigma-table (make-vector n-p 0))
  (let loop
    ((i-p 2)
     (k 0))
    (cond
      ((>= i-p n-p)
```

```
      (vector-set! sigma-table (- n-p 1) k))
     ((eq? (string-ref p k)
           (string-ref p (- i-p 1)))
      (vector-set! sigma-table i-p (+ k 1))
      (loop (+ i-p 1) (+ k 1)))
     ((> k 0)
      (loop i-p (vector-ref sigma-table k)))
     (else ; k=0
      (vector-set! sigma-table i-p 0)
      (loop (+ i-p 1) k))))
(vector-set! sigma-table 0 -1)
(lambda (q)
  (vector-ref sigma-table q)))
```

The failure function is a Scheme lambda that is returned from this procedure. It encapsulates a vector `sigma-table` that maps an index `q` (which is the `i-p` from the KMP algorithm) to the length of the longest prefix of `p` that is also a suffix of $p_{0 \to q-1}$. As explained, by convention $\sigma(0) = -1$. Furthermore, $\sigma(1) = 0$ since the length of the longest prefix of p that is also a suffix of p_0 is zero. The algorithm therefore starts looking for suffixes starting from $i_p = 2$ until $i_p = n_p - 1$. At the beginning of every iteration, k is the length of the longest prefix of p that is also a suffix of $p_{0 \to i_p - 2}$. The goal of the body is to determine the length of the longest prefix of p that is also a suffix of $p_{0 \to i_p - 1}$.

Suppose that we have found out — in the previous iteration of the loop — that $k = \sigma(i_p - 1)$ is the length of the longest prefix of p that is also a suffix of $p_{0 \to i_p - 2}$. In the next iteration of the loop, we have the following possibilities:

- If $i_p = n_p$ we have reached the end the pattern. The k computed so far is the length of the longest prefix of p that is also a suffix of $p_{0 \to n_p - 1}$. All we have to do is store it in the last position of the table and quit the algorithm.

- If the $(k+1)$'th character of p (residing at index k) continues matching the i_p'th character of p (residing at index $i_p - 1$), then this means that the suffix of $p_{0 \to i_p - 1}$ is just one character longer than the suffix of $p_{0 \to i_p - 2}$ computed by the previous iteration. Hence. $\sigma(i_p) = k+1$ and we continue the loop for the next i_p and the next k.

- If the character doesn't match, but $k > 0$, then we investigate the previously visited prefixes (by reducing k) that are also suffixes of $p_{0 \to i_p - 2}$ in order to see whether one of *those* prefixes *is* followed by the character at position $i_p - 1$. But the lengths of those prefixes already reside in σ from previous iterations. Hence we continue the loop by reducing k to $\sigma(k)$ in the hope that this prefix is a prefix for which the next character *is* equal to the character at position $i_p - 1$. Having found such a prefix, both k and i_p can start to increment again in subsequent iterations in order to find the *longest* such prefix.

- If the character doesn't match and $k = 0$, then we have unsuccessfully visited all possible prefixes of p. As such, the length of the longest prefix of p which is also a suffix of $p_{0 \to i_p - 1}$ turns out to be zero. Hence, we assign the value 0 to $\sigma(i_p)$.

Performance

Assume for a moment that σ has been successfully computed and focus on the main KMP algorithm. We focus on how the sum $i_t + i_p$ evolves since that sum is an indication for the progress of the matching process. In the brute-force algorithm, this sum continuously increases (by 1) as more characters of the pattern match the text. However, that sum is violently reset to $i_t + 1$ every time a mismatch is encountered. Let us now analyse the evolution of the sum for the KMP algorithm. In every execution of the loop, $i_t + i_p$ is incremented by 1 whenever the current characters match. When the characters do not match, i_t is replaced by $i_t + i_p - \sigma(i_p)$ and i_p is replaced by $\sigma(i_p)$. Hence, in this case the sum $i_t + i_p$ remains identical to $i_t + i_p$ in the previous iteration of the loop. Hence, the worst thing that could happen is an alternation between these two situations (i.e. a mismatch following every match) since that behaviour makes the sum increase in the slowest possible way. Since the sum can never exceed n_t, this causes the loop to execute at most $O(2.n_t) = O(n_t)$ times.

In order to see how long it takes to establish σ, we first observe that for all x, $\sigma(x) < x$. Indeed, the length of the longest prefix of p that is also a suffix of $p_{0 \to x-1}$ can never be longer than $x - 1$ since we are only interested in proper prefixes and suffixes. Let us focus on the number $i_p - k$ for a moment.

- In the second branch of the conditional, we observe that the reiteration replaces i_p by $i_p + 1$ and k by $k + 1$. This means that $i_p - k$ remains the same.
- In the third branch of the conditional, the reiteration keeps i_p and replaces k by $\sigma(k)$. Since $\sigma(k) < k$, this means that $i_p - k$ increases.
- The fourth branch of the conditional causes a reiteration with the same k but by replacing i_p by $i_p + 1$. Again, $i_p - k$ increases.

This case-by-case analysis shows that every iteration either keeps $i_p - k$ constant, or makes this number increase by some amount. Because $i_p - k \leq i_p$ and $i_p \leq n_p$, and because i_p and $i_p - k$ cannot stagnate at the same time (check this in the code!), this means that the loop can execute no more than $2.n_p$ times. We conclude that `compute-failure-function` is in $O(n_p)$.

The performance of the KMP algorithm is a combination of establishing the failure function and executing the actual algorithm. Since the former is in $O(n_p)$ and the latter is in $O(n_t)$, the entire KMP algorithm is in $O(n_t + n_p)$. For large n_p and n_t this is *considerably* better than the $O(n_p.n_t)$ result of the brute-force algorithm. Therefore, the KMP algorithm is quite successful in bio-informatics where the patterns are extremely long (about one hundred thousand "characters") and a lot of repetition occurs. For more mundane applications

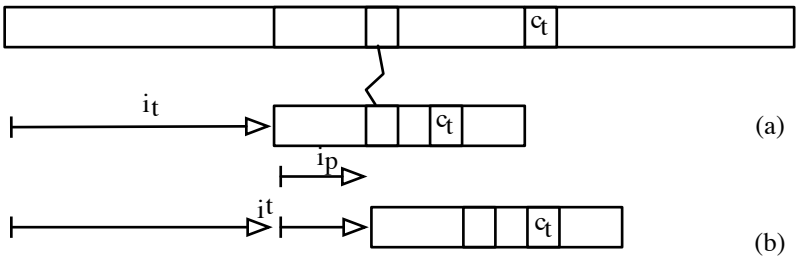

Figure 2.5: The QuickSearch Alorithm

such as finding a short word in a web page or in a word processing document, KMP does not significantly outperform the brute-force algorithm. In those cases, it appears that the average case behaviour for the brute-force algorithm is quite similar to the KMP algorithm.

2.5 The QuickSearch Algorithm

Many clever pattern matching algorithms have been discovered in the past decades. The QuickSearch algorithm was discovered in 1990 by D.M. Sunday. It outperforms the algorithms we have discussed so far. On top of that, QuickSearch is much simpler than the Knuth-Morris-Pratt algorithm and most other algorithms known in the literature. The fact that a simpler algorithm beats more complex algorithms is fairly exceptional in computer science.

In order to understand the QuickSearch algorithm, we invite the reader to have a look at figure 2.5. The figure shows the QuickSearch algorithm right before (figure 2.5(a)) and right after (figure 2.5(b)) a mismatch. Whenever a mismatch occurs, the algorithm considers the text character c_t that is located at the first position to the right of the current alignment. c_t is then searched for in the pattern. In case c_t does not occur in the pattern at all, then the pattern is shifted entirely beyond c_t. No intermediate alignment can result in a successful match since the text contains c_t and the pattern doesn't. However, if c_t *does* occur in the pattern, then we realign the pattern with the text in such a way that the occurrence of c_t in the pattern is aligned with the occurrence of c_t in the text. When there is more than one occurrence of c_t in the pattern then we have to select the *rightmost* such occurrence. If we would take any other occurrence then we run the risk of overshooting a solution by sliding the pattern too far. After the pattern has been realigned like this, we restart comparing the pattern against the text beginning with the very first character of the pattern.

A Scheme procedure implementing the QuickSearch algorithm is presented below. Again the procedure is conceived as a loop with a conditional body. The first branch checks whether the entire pattern has been checked. The second branch checks whether the entire text has been unsuccessfully consumed. The

third branch checks for an additional matching character and continues the loop by considering the next character. Finally, the fourth branch is applied whenever the third branch does not succeed, i.e. when a mismatch occurs. In that case, the algorithm realigns the pattern against the text by sliding the pattern to the right (by the amount indicated by applying the `shift` function to the character `c-t` that is depicted in figure 2.5) and by resetting i_p to zero in order to restart checking the pattern from scratch. The call to `mod` is to cover the exceptional case that occurs when the very last character of the text gives rise to a mismatch. In that case, there is no c_t left and trying to access it would cause us to read a non-existing character at position n_t.

```
(define (match t p)
  (define n-t (string-length t))
  (define n-p (string-length p))
  (define shift (compute-shift-function p))
  (let loop
    ((i-t 0)
     (i-p 0))
    (cond
      ((> i-p (- n-p 1))
       i-t)
      ((> i-t (- n-t n-p))
       #f)
      ((eq? (string-ref t (+ i-t i-p)) (string-ref p i-p))
       (loop i-t (+ i-p 1)))
      (else
       (let ((c-t (string-ref t (mod (+ i-t n-p) n-t))))
         (loop (+ i-t (shift c-t)) 0)))))))
```

The clever thing about the QuickSearch algorithm is that the `shift` function can be established upfront by preprocessing the pattern. This is shown below. The procedure `compute-shift-function` returns a lambda that encapsulates a shift table. The shift table is indexed by ASCII values that vary between the smallest and the biggest ASCII value of the characters occurring in the pattern. For example, consider the pattern "hello". Using `char->integer` we know that the smallest character is #\e with ASCII value 101. Similarly, the greatest value is the character #\o the ASCII value of which is 111. Therefore, the shift table has 11 entries, one for every character *between* #\e and #\o. The shift for all other characters (i.e. all characters that are smaller than #\e and greater than #\o) is the length of the entire pattern, i.e. n_p. This distinction is made in the body of the lambda expression, before the actual shift table is consulted. For all characters that *do* lie between #\e and #\o, the shift table contains the correct shift. For the characters that do not occur in the pattern (such as for example #\f, the shift is n_p as well. Indeed, when c_t does not occur in the pattern, we can safely slide the pattern beyond c_t. Therefore, the (`make-vector` ...) expression returned by `create-table` initialises all entries of the vector to n_p. The procedure creates this vector after having established the smallest and the

greatest ASCII values of the characters of the pattern. These ASCII values are kept in the variables `min-ascii` and `max-ascii` while `create-table` processes the entire pattern. The table contains `max-ascii` − `min-ascii` +1 entries. Notice that the table is a Scheme vector that is necessarily indexed starting at index 0 and ending at index `max-ascii` − `min-ascii`. This is the reason why we have to "normalize" a given ASCII value when looking it up in the table; i.e. we substract `min-ascii` in order to obtain an index that lies in the range of the vector.

After having created the table, (`fill-table 0`) traverses the pattern a second time, from left to right. For every character it encounters at a certain `index`, n_p minus `index` is written into the corresponding entry in the shift table. For instance, if an #\a is encountered at the third position in a pattern of length 10, then the table entry that corresponds to the #\a contains 7. This means that the pattern can be shifted 7 positions to the right after encountering a mismatch where c_t equals #\a. By filling the vector from left to right we guarantee that only the position in the pattern of the rightmost occurrence of a character is remembered in the vector.

```
(define (compute-shift-function p)
  (define n-p (string-length p))
  (define min-ascii (char->integer (string-ref p 0)))
  (define max-ascii min-ascii)

  (define (create-table index)
    (if (< index n-p)
      (begin
        (set! min-ascii (min min-ascii (char->integer (string-ref p index))))
        (set! max-ascii (max max-ascii (char->integer (string-ref p index))))
        (create-table (+ index 1)))
      (make-vector (− max-ascii min-ascii −1) n-p)))

  (define (fill-table index)
    (if (< index n-p)
      (let* ((ascii (char->integer (string-ref p index))))
        (vector-set! shift-table (− ascii min-ascii) (− n-p index))
        (fill-table (+ index 1)))))

  (define shift-table (create-table 0))
  (fill-table 0)
  (lambda (c)
    (let ((ascii (char->integer c)))
      (if (>= max-ascii ascii min-ascii)
        (vector-ref shift-table (− ascii min-ascii))
        n-p))))
```

Performance

This QuickSearch has the potential of exhibiting sublinear behaviour: $O(N+\frac{n_t}{n_p+1})$ in the best case. The first N comes from the work needed to establish the shift table. By analyzing the code of `compute-shift-function`, we observe that a vector has to be initialized (all entries are taken to be `n-p`). The size of the vector depends on the number of different characters in the pattern. After initializing the vector, the pattern is traversed in order to fill the vector. Hence $N = max(\delta_p, n_p)$ where δ_p is the number of different characters occurring in the pattern (this is also known as the *alphabet size* of the pattern). The second $\frac{n_t}{n_p+1}$ comes from the fact that, in the best case, the pattern immediately gives rise to a mismatch for each and every alignement and the character c_t following the alignment does not occur in the pattern. This results in a $n_p + 1$ shift for every comparison.

The fact that QuickSearch has the potential of exhibiting sublinear behaviour stems from the fact that it is able to skip large portions of the input text. In other words, the algorithm has the benefit of not checking every single character in the text. This is in sharp contrast with the brute-force algorithm and the KMP algorithm, which examine every single character in the text at least once. However, this can also be problematic. If the application of a pattern matching algorithm in a software application has an additional goal, then it might be desirable to check every single character. For example, suppose that —apart from finding the pattern — we also want to count the number of words in a text. This actually means that we need to count the number of whitespace characters. By skipping large portions of the text we clearly will not end up with the correct number of whitespace characters. In such cases, the KMP algorithm is a good solution.

It is not hard to come up with a worst-case example that shows that QuickSearch is actually in $O(n_p.n_t)$ (find it!). However, in practice, QuickSearch beats all other approaches.

2.6 Strings vs. Data Storage

In this chapter, we have studied ways to retrieve information from strings. Although strings are easily composed using `string-append`, the algorithms presented show that retrieving information from strings is not easy. As explained in section 1.4.3, one of the central themes of the course is how to implement dictionaries. Suppose that we were to implement dictionaries using strings. We might consider representing a dictionary as a large string that contains all the key-value pairs in textual format. `string-append` would be used to implement the `insert!` function. Indeed, inserting a key-value pair would boil down to appending the key-value pair to the string representing the dictionary. However, as the algorithms in this chapter show, the implementation of `find` would not be very efficient. As the dictionary grows, `find` gets slower and slower. Moreover, the efficiency of `find` not only depends on the *number* of key-value pairs, but also on the number of characters that make up those pairs.

The deeper reason why strings are not a good way to represent large collec-

tions of data is that strings are extremely structure-shy. As we will see in the rest of the course, richer data structures (i.e. data structures which exhibit more internal structure than mere "flat" strings) allow for much faster implementations of `find`. Strings are not a very good representation for storing information in a computer. In the optimal situation, they are merely used by programs in order to communicate with users (e.g. using `display`). Inside the guts of our programs, we try to shun strings as much as possible.