0.1 Palindromes

Recall that a *palindrome* is a string that reads the same forward and backward, i.e., $x = x^R$, where x^R is the *reverse* of string x. Once we know how to compute all periods of a string then we also know how to decide whether or not a string starts by a palindrome. Actually, we can compute all initial palindromes of w. For this, we run the algorithm on $w!w^R$ where ! is not in the alphabet. We could repeat this treatment to find all palindromes in w. Note however, that this algorithm is not online. As it turns out, much better palindrome detectors are known. In 1975, G. Manacher showed that in fact all palindromes in a string can be found in overall linear time [**?**]. Since the string a^m contains $\Theta(m^2)$ palindromes it is clear that this calls for a preliminary stipulation of a suitably compact, implicit description for palindromes. This turns out to be easy: let us say that a palindrome in y is *maximal* if the characters that respectively precede and follow it in y are different. Next, let us represent maximal palindromes by the pair of their center position in y and their length. Clearly, any non-maximal palindrome is contained in a maximal one centered at the same position, and is implicitly represented by the latter. We have thus reached the important objective of having a collective representation for all palindromes in a string that is linear in the size of that string. From now on, by the word palindrome we will intend to refer to a maximal palindrome. Consider first the computation of palindromes of even length, as described by the obvious pseudo-code in the figure. In an even palindrome, the center is represented by the interstice between two consecutive positions.

```
procedure initpalindrome ( y )
    begin
    i ← 1;
    while i1 ≤ ⌊|y|/2⌋ do
        test if y_1 y_2 ... y_2i is a palindrome;
        i = i + 1
    endwhile
    end
```

Figure 1: Finding all initial palindromes of y by direct search

The drawback of such a direct method is in that it does not profit in

```
procedure initpalindromesmart ( y )
    begin
    i ← 2; j ← 0; R(1) ← 0;
    while i ≤ ⌊|y|/2⌋ do
        while y(i − j) = y(i + j + 1) do j = j + 1 ;
        if j = i then output(i) ; R(i) ← j ;
        k ← 1;
        while R(i − k) ≠ R(i) − k do
            R(i + k) ← min{R(i − k), R(i) − k}; k = k + 1;
        j ← max{j − k, 0}; i = i + k;
    endwhile
    end
```

Figure 2: Finding all initial palindromes of y by linear search

any way from the information gathered as each *center* value i is processed. Manacher's algorithm takes instead advantage of that. Let us label by i the space between positions i and $i+1$ of y, and let $R(i)$ be the *radius* of the palindrome centered at i, i.e., the number of consecutive characters to the right of i that match characters symmetric on the left. The algorithm keeps track of the radius j of i for all centers i processed so far. The crux of the savings is induced by the innermost while loop, the rationale of which is that at each starting of the loop we can compute the radii for all consecutive positions k for which $R(i-k) \neq R(i) - k$.

This property yields the linear time procedure. The algorithm scans the string from left to right keeping track of the radii at each interstice. Assume that the current center is at position$i$ and let the h be the displacement of the mismatch from i. As we have computed the radii of all positions to the left of i, we can now scan the positions to the right of i looking for the next candidate center of an initial palindrome. Clearly, as long as the scan is confined inside the radius from i, any such center has a mirror on the left of i, for which the radius m is already known. Let n be the number of positions separating the new candidate center i' from the mismatch. The key is to compare m and n. Clearly, if $m < n$ the candidate position cannot be a center, since the symmetry about i just reproduces a smaller radius about i'. The condition $m > n$ equally disqualifies i', since the mismatch about i is incompatible with a match of the same character about i'. The following

lemma establishes the correctness of the algorithm formally.

Lemma 0.1.1 *Let k be such that $1 \leq k \leq R(i)$ and $R(i) \neq R(i) - k$. Then $R(i+k) = min\{R(i-k), R(i)-k\}$.*

Proof. Consider the two possible cases separately. In case $R(i+k) < R(i)-k$, then the palindrome centered at $i-k$ is fully inside the palindrome centered at i. Therefore, by symmetry the longest palindrome at $i+k$ has the same radius as the one at $i-k$, i.e., $R(i+k) = R(i-k)$. On the other hand, if $R(i+k) > R(i)-k$, then the characters delimiting the longest palindrome at $i+k$ must differ, whence $R(i+k) < R(i)-k$, a contradiction. ◇

It is left to the reader as an exercise to prove that the algorithm is easily modified to find *all* (i.e., odd and even, prefix as well as infix) palindromes in a string.

Once there is an algorithm to find palindromes in the string, one can also use this to decide whether or not the string itself is the product of the concatenation of strings all of which are palindromes. The set of strings with this property have historical and substantive interest in the theory of formal languages, where it has sometimes been referred to as the language *PALSTAR*. An easier problem is to test membership of a string in the set *E-PALSTAR* of words that are concatenation of even palindromes. For this, the natural idea is to greedily parse the input string looking for the earliest even palindrome, cut it away and proceed with the rest. This procedure takes linear time, by Manacher's algorithm. However, it remains to be seen that the greedy strategy is guaranteed to work, since we could cut a prefix of a larger palindrome with prejudice on the outcome. Luckily (see Exercise section), it can be proven that the greedy parse always works with even palindromes. A more elaborate construction, due to Galil and Seiferas, shows that membership in PALSTAR can be decided in linear time. Surprisingly, the task becomes more involved if one wants to test in linear time whether a string is formed by concatenation of a given *fixed* number of palindromes. It is known how to do this for up to 4 palindromes, but not beyond that number.

0.2 Automata

Although language membership and decision problems are not the focus of this discussion, a number of issues that emerge in pattern discovery relate to the theory of automata, particularly, finite automata and their corresponding languages and regular expressions. We thus offer a concise recap here. Interestingly enough, the existence of a connection between automata theory and string searching was recognized rather early, but it took some time to see how this could be exploited in the design of algorithms. As D. Knuth would report a few years later, he had been quite pleasantly surprised when in the early seventies he discovered that he could use concepts from finite automata in the design of fast string searching. In his own words, this was the "first time in Knuth's experience that automata theory had taught him how to solve a real programming problem better that he could solve it before' [**?**]'.

A *finite automaton* $\mathcal{A}$ is defined as a quintuple $\{\Sigma, Q, \delta, q_0, F\}$ where Σ is the input alphabet, Q is a finite set of states, q_0 is the *initial state* in which all computations begin, F is a subset of Q consisting of *final* or *accepting* states, and the "engine" δ is the *transition table* implementing a mapping $\Sigma \times Q \rightarrow Q$ that specifies for each character $a \in \Sigma$ and $q \in Q$ the new state entered by $\mathcal{A}$ when reading a on the input while in state q. Given an automaton $\mathcal{A}$, the set of strings accepted by it consists of precisely those strings that put the automaton in a final state when it is started in state q_0 reading the first character of the input. The sets accepted by a finite automata are called *regular* sets. A key observation at this junction is that, because it makes one transition on each input character consumed, a finite automaton always processes its input in linear time.

One of the most beautiful constructions of theoretical computer science consists of showing that regular sets are susceptible to a purely syntactic characterization that ignores any concept of a machine. Such a characterization is based on the notion of a *regular expression*. Regular expressions describe sets of strings resulting by some finite number of applications of the concatenation $(\cdot)$, union $(+)$ and *star* operator $(*)$ to the characters of an alphabet, where the $*$ operator is the reflexive transitive closure of concatenation. For instance, the regular expression a^* denotes the set of all strings consisting of 0 or more "a"s, $(a \cdot b)^* + b$ the set of strings consisting of the character b or any number of repetitions of ab, such as $abab ab, abababab$, etc., or even λ. The regular expression $(0 \cdot 1)^* + (0 \cdot 1 \cdot 1)^*$ denotes the set

of all strings in either one of the forms 01010101... or 011011011011.... And $(aba + a)^*$ denotes the set of strings that result form the concatenation of any number of copies of aba or a. Thus, $\lambda, abaaba, abaa, aaa, abaaaaba$ belong to this set, but say $abbaaba, abab, bba$ do not. Usually the concatenation operator is omitted, so that regular expressions resemble algebraic expressions in which the $*$ operator replaces a fixed power. Regular expressions are inductively defined as follows.

- λ, and each $a \in \Sigma$ are regular expressions;
- If r, r_1 and r_2 are regular expressions then so are $r_1 + r_2, r_1 \cdot r_2$ and r^*.

The following classical theorem holds.

Theorem 0.2.1 *A language can be recognized by a finite automaton if and only if it can be denoted by a regular expression*

We will not prove this theorem but we can use it immediately in connection with fast string searching. In its simplest formulation, this problem consists of being assigned a textstring x and a pattern string y over some alphabet Σ and having to decide whether or not y occurs within x. The key observation is that this problem amounts to recognizing the set of strings denoted by a regular expression, namely, $\Sigma^* \cdot y \cdot \Sigma^*$, for which there must be a finite automaton and hence (since the automaton makes one transition on each input character consumed) also a solution taking time linear in the input string. Before we delve into this special case, we examine the more general problem of recognizing regular expressions.

0.3 Word Sets and Regular Expressions

Searching for regular expressions or slight variations thereof is a widespread facility of operating systems and software in general, and applications appear in e.g. grep, sed, awk, perl, vi, shells, lex, yacc, etc. The problem we consider here is, given a regular expression $\mathcal{E}$, to preprocess it in order to locate all occurrences of words of the associated language $lang(\mathcal{E})$ that occur in any given word x. We will see later that regular expressions denoting a finite number of words pose already quite interesting algorithmic problems. However, note that, in principle, a regular expression describes an infinite set of strings.

The classical solution to the general problem is composed of two phases. First, transform the regular expression $\mathcal{E}$ into a nondeterministic automaton that recognizes the language described by $\mathcal{E}$, according to a construction due to Thompson (cfr [**?**]). Following that, simulate the automaton thus obtained on the input word y in such a way that it recognizes each prefix of y that belongs to $\Sigma^* \cdot lang(\mathcal{E})$. Both phases are linear in the input. In particular, a nondeterministic automaton taking space linear in the length of the regular expression is easily built by iterated serial/parallel composition of smaller automata over the alphabet $\Sigma \cup \{\lambda\}$, using transitions on the empty symbol λ as connectors. Composition of constituent automata under each of the operations induced by $+$, $\cdot$ or * can be implemented to work in constant time. Combined with a prudent parsing of $\mathcal{E}$ this leads to the following result:

Theorem 0.3.1 Let $\mathcal{E}$ be a regular expression. The nondeterministic automaton recognizing $lang(\mathcal{E})$ can be computed and stored in time and space $O(|\mathcal{E}|)$.

The derivation of such an automaton proves one half of a central theorem of Kleene, recalled earlier, which set the equivalence between the languages recognized by finite automata and those described by a regular expression.

However, it is well known that the transformation of a nondeterministic automaton into a deterministic one is accompanied by an exponential explosion in the number of states. This poses a problem in the searches, since the search for end-positions of words in $lang(\mathcal{E})$ is performed by a simulation of a deterministic automaton recognizing $\Sigma^* lang(\mathcal{E})$. To circumvent this, the determinization is just simulated at search time: at any given time during the search, the automaton will not be in a single state, but rather in a set of states, the search itself taking care of dynamically maintaining knowledge of this set. A central notion for this process, related to λ-transitions, is that of λ-*closure* for a set of states S. This is the set of states reachable from Q solely through λ-transitions. Once the closure of a set of states is known, it is possible to compute effectively the transitions induced by any input symbol.

The simulation of a regular-expression-matching automaton consists of repeating the two operations "closure" and "transitions on a set of states". With careful implementation, based on standard manipulation of sets and queues, the time and the space required to perform either part is linear in the size of sets of states involved. This leads to the following

Theorem 0.3.2 Given a regular expression $\mathcal{E}$, testing whether a word y belongs to $lang(\mathcal{E})$ can be done in time $O(|\mathcal{E}| \times |y|)$ and space $O(|\mathcal{E}|)$.

Note that the original problem is different, in that it requires that the answer to the test be reported for each substring of the text x, and not only on x itself. But no transformation of the automaton for $(\mathcal{E})$ is necessary. A mere transformation of the search phase of the algorithm is sufficient: at each iteration of the closure computation, the initial state is integrated to the current set of states. By doing so, each substring of x is tested, and the following is established.

Theorem 0.3.3 Let $\mathcal{E}$ be a regular expression and x be a word. Finding all end-positions of subwords of x that are recognized by the automaton associated with $(\mathcal{E})$ can be performed in time $O(|\mathcal{E}|\,|x|)$ and space $O(|\mathcal{E}|)$. The time spent on each symbol of x is $O(|\mathcal{E}|)$.

As mentioned, the drawback of performing regular-expression-matching by deterministic automata is that the automaton can have a number of states exponential in the length of $\mathcal{E}$. This is the situation, for example, when

$$\mathcal{E} = \mathsf{a}\overbrace{(\mathsf{a}+\mathsf{b})\cdots(\mathsf{a}+\mathsf{b})}^{m-1 \text{ times}}$$

for some $m \geq 1$; here, the minimal deterministic automaton recognizing $\Sigma^* lang(\mathcal{E})$ has exactly 2^m states since the recognition process has to memorize the last m symbols read from the input word x. However, not all states of the deterministic automaton for $\Sigma^* lang(\mathcal{E})$ are necessarily met during the search phase. This suggests a lazy construction of the deterministic automaton during the search as a possible practical alternative.

0.4 Searching for one string

It turns out that setting up a finite automaton for recognizing the language $\Sigma^* y$ or even $\Sigma^* y \Sigma^*$ does not require to go explicitly through the nondeterministic version of it. To illustrate this point, the figure below represents the completion of the construction of the smallest automaton recognizing $\Sigma^* y$ if State 7 is assumed to be the only final state. Adding the last state entails redirecting to it the transition on character b from state 6. This transition

was formerly taking to state 3. Now, it remains to define the transitions from the new state 7. But these states are precisely those previously reached through 6, under the respective characters a and b. The whole construction can be described more formally by the following pseudocode.

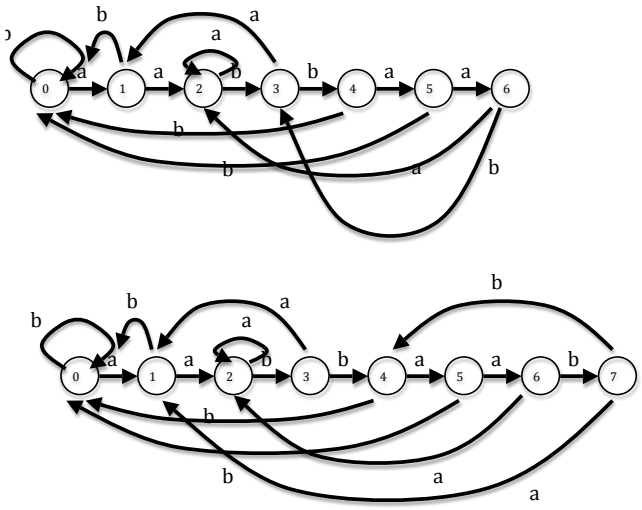

Figure 3: Building the smallest finite automaton recognizing *aabbaab*

procedure) builddfa
begin
$q_0 \leftarrow newcell; Q \leftarrow q_0; q_f \leftarrow q_0$;
for all $a \in \Sigma$ **do** $\delta(q_0, a) \leftarrow q_0$;
for $i = 1$ **to** m **do**
$temp \leftarrow \delta(q_f, y_i)$; $q \leftarrow q_f$; $q_f \leftarrow newcell$; $q_f \leftarrow \delta(q, y_i)$;
for all $a \in \Sigma$ **do** $\delta(q_f, a) \leftarrow \delta(temp, a)$;
end
output $\{\Sigma, Q, \delta, q_0, \{q_f\}\}$
end.

It is clear that building such an automaton takes time $\Theta(|\Sigma| \times |y|)$. The details are left for an exercise.

Various pattern matching techniques and tools (refer, e.g., to [**?**, **?**]) have been developed in more than three decades to detect and count all distinct occurrences of an assigned substring y (the *pattern*) within a longer string x (the *text*). None of these methods utilizes finite automata explicitly, and the astute reader has already figured out why. Below we describe one method resulting from multiple and partially interdependent efforts of Knuth, Morris and Pratt, that is the closest in structure to the automaton derivation above.

Before getting into that consider for a moment the cost of a direct method. The latter would start by aligning the pattern y and text x starting at the first position of both and proceed to check characters from either string pairwise. Let i and j be used to keep track of the pattern and text characters being considered, respectively. If a mismatch occurs between x_i and y_j, then j is reset to 1 and i is backed up at position $i - j$, then the process resumes. It is clear that such a strategy leads to $O(n \times m)$ character comparisons (easily the dominant cost of the algorithm), which in the worst case reaches precisely $n \times m$ (why?).

A perusal of the automaton described earlier suggests where the source of inefficiency of the direct method might be: in order to improve this performance, we need to be able to shift the pattern farther following each mismatch. Specifically, the shift must move the pattern so as to bring it in line with the earliest suffix of $x_1x_2...x_{i-1}$ that has a chance of being completed into an occurrence. Such a shift corresponds to the quantity $j - bord(y_1y_2...y_{j-1})$, that is, to the period of $y_1y_2...y_{j-1}$! In conclusion, following a mismatch between x_i and y_j, the character of y that must be compared with x_i is obtained by resetting j to $next(j) = j - bord(y_1y_2...y_{j-1}) + 1$. The corresponding algorithm can be described as follows.

```
procedure stringsearch ( y )
    begin
    i ← 1; j ← 1;
    while  j < n do begin
        while  j = m + 1 or  (i > 0 and y_j ≠ x_i do
            j ← next[j];

        i ← i + 1; j ← j + 1;
        if j = m + 1 then "occurrence at i"
    endwhile
    end
```

This algorithm is interesting in several respects. For starters, the idea of using the table next does not look at first as a great strategy. In fact, if we count the number of character comparisons "locally", then for any fixed value of i the i-pointer may be stuck with the same value while the pattern still moves forward by only one character at a time, driven by the table *next*. To see this, consider the pattern $y = a^m$ faced with the text $x = ...a^{m-1}b...$ following the comparison between the last a of y with the b in x.: the table *next* will advance the pattern one position at a time, thereby charging $\Theta(m^2)$ comparisons to this single value of i. It would appear that iterating this cost through the process will amount to a total of $n \times m$, which is the same as in the direct search. While $O(n \times m)$ is certainly an upper bound for the algorithm, however, it is not a tight upper bound. This is due to the fact that the localized accounting does not do justice of the method: whereas $\Theta(m^2)$ comparisons may take place for some values of i, this may not occur *all the time*. This can be proved by an *amortized* accounting that derives additional elegance from the following feature: because we cannot take into account all possible input configurations with their respective character distribution and structure, we will tally not the character comparisons but rather the *effects* of character comparisons.

Theorem 0.4.1 *The algorithm* stringsearch *performs a number of character comparisons bounded by* $2n - m$.

Proof. Each character comparison results in either $y_j = x_i$ or $y_j \neq x_i$. In the first case i undergoes a unit increase, in the second, y undergoes a forward shift. Neither action is ever reverted throughout the execution, whence the first event can take place no more than n times and the second no more than $n - m$ times. ◇

To conclude this section, there are a number of issues that we are asked to consider. One is the relationship between the automaton and a similar transition diagram implementing the *next* table. Another, is the relationship between the computation of borders examined in the previous chapter and the process of searching.

One way to address the first issue is by asking what would be needed in order to transform the state transition diagram of the automaton into that of a "string searching machine" embodying the transitions that are operated by the algorithm. The figure below shows the structure of such a machine for the string *aabbaab* in our example.

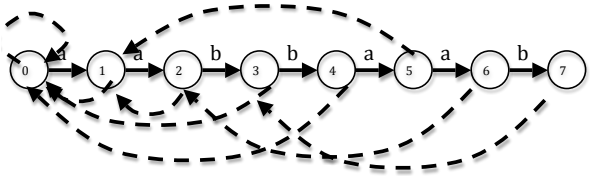

Figure 4: The string searching machine recognizing *aabbaab*

The main difference with the automaton is that this machine does not consume input on backward transitions. The machine may be taken as a living proof of the linearity theorem, in the following sense: the (not more than n) forward transitions advance the parsing of the text, and every (not more than $n-m$) backward transition shifts the pattern forward. This comes as no surprise, since the machine ultimately incarnates the computation of borders. Upon looking again at the procedure that computes borders it is striking to see the similarity between that procedure and string searching: in fact, the computation of borders is like running stringsearching with two copies of the patters serving as text and pattern, respectively. One of the beauties of the method is in that this preprocessing does not seem to need a precomputed table $next$. In fact, the values of the table that are needed in the course of the computation are always already in place when needed.