(appeared in GInfo 15/7, November 2005)

Suffix arrays – a programming contest approach

Adrian Vladu and Cosmin Negruşeri

Summary

An important algorithmic area often used in practical tasks is that of algorithms on character strings. Many programming contests have used problems that could have been easily solved if one efficiently managed to determine if one string was a subsequence of another string, or had found an order relation within a string's suffixes. We shall present a versatile structure that allows this along with other useful operations on a given string.

Keywords: suffix sorting, suffix arrays, suffix trees

1 Introduction

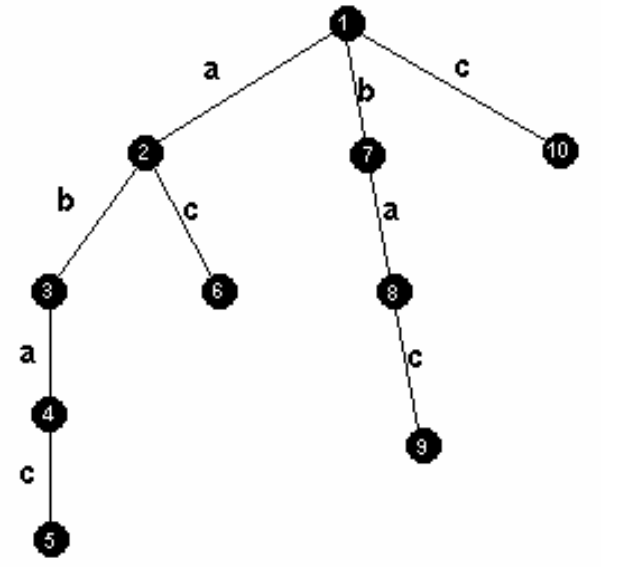
What are suffix arrays?

In order to let the reader gain a better vista on suffix arrays, we shall make a short presentation of two data structures called **trie**, respectively suffix tree [1] – which is a special case of a trie. A trie is a tree meant to store strings. Each of its nodes will have a number of sons equal to the size of the alphabet used by the strings that are needed to be stored. In our case, with strings containing only small letters of the English alphabet, each node will have at most 26 sons. Every edge going from the father toward its sons is labeled with a different letter of the alphabet. The labels on a path starting from the root and ending in a leaf will form a string stored in that tree. As it can be easily seen, finding whether a string is contained in this data structure is very efficient and is done in O(M) time, where M is the string's length. Therefore, the searching time does not depend on the number of words stored in the structure, this making it an ideal structure for implementing dictionaries.

Let's see what a suffix trie is:

Given a string $A = a_0a_1\ldots a_{n-1}$, denote by $A_i = a_ia_{i+1}\ldots a_{n-1}$ the suffix of A that begins at position i. Let n = length of A. The suffix trie is made by compressing all the suffixes of $A_1\ldots A_{n-1}$ into a trie, as in the figure below.

The suffix trie corresponding to the string "abac" is:



Operations on this structure are very easily done:

- checking whether a string W is a substring of A – it is enough to traverse the nodes starting from the root and going through the edges labeled correspondingly to the characters in W (complexity O(|W|))
- searching the longest common prefix for two suffixes of A – choose nodes u and v in the trie, corresponding to the ends of the two suffixes, then, with a LCA algorithm (least common ancestor), find the node corresponding to the end of the searched prefix. For example, for "abac" and "ac", the corresponding nodes are 5 and 6. Their least common ancestor is 2, that gives the prefix "a". The authors are strongly recommending [2] for an O($\sqrt{n}$) solution, [3] for an accessible presentation of a solution in O(lg n) or O(1), and [4] for a state of the art algorithm.
- finding the k-th suffix in lexicographic order - (complexity O(n), with a corresponding preprocessing). For example, the 3rd suffix of "abac" is represented in the trie by the 3rd leaf.

Even if the idea of a suffix trie would be very pleasing at first sight, the simplist implementation, where at every step one of the strings suffixes is inserted into the structure leads to an $O(n^2)$ complexity algorithm.There is a structure called suffix tree[1] that can be built in linear time, which is a suffix trie where the chains containing only nodes with the out-degree equal to 1 were compressed into a single edge (in the example above, these are represented by the chains 2 -3 – 4 – 5 and 1 – 7 – 8 – 9). Implementing the linear algorithm is scarcely possible in a short time, such as during a contest, this determining us to search another structure, easier to implement.

Let's see which are the suffixes of A, by a depth first traversal of the trie. Noting that during the depth first search we have to consider the nodes in the ascending lexicographic order of the edges linking them to their father, we gain the following suffix array:

abac = $\mathbf{A_0}$
ac = $\mathbf{A_2}$
bac = $\mathbf{A_1}$
c = $\mathbf{A_3}$

It is easy to see that these are sorted ascending. To store them, it is not necessary to keep a vector of strings; it is enough to maintain the index of every suffix in the sorted array. For the example above, we get the array **P = (0, 2, 1, 3),** this being the suffix array for the string "abac".

2 The suffix array data structure

2.1. How do we build a suffix array?

The first method we may think of is sorting the suffixes of A using an O(n lg n) sorting algorithm. Since comparing two suffixes is done in O(n), the final complexity will reach $O(n^2 \lg n)$. Even if this seems daunting, there is an algorithm of complexity O(n lg n), relatively easy to understand and code. If asymptotically its building time is greater that that of a suffix tree practice taught us that in reality constructing a suffix array is much faster, because of the big constant that makes the linear algorithm to be slower than we might think. Moreover, the amount of memory used implementing a suffix array with O(n) memory is 3 to 5 times smaller than the amount needed by a suffix tree.

The algorithm is mainly based on maintaining the order of the string's suffixes sorted by their 2^k long prefixes. We shall execute $m = [\log_2 n]$ (ceil) steps, computing the order of the prefixes of length 2^k at the k^{th} step. It is used an m x n sized matrix. Let's denote by A_i^k the subsequence of A of length 2^k starting at position i. The position of A_i^k in the sorted array of A_j^k subsequences (j = 1, n) is kept in $P_{(k, i)}$.

When passing from step k to step k + 1, all the pairs of subsequences A_i^k and $A_{i+2^k}^k$ are concatenated, therefore obtaining the substrings of length 2^{k+1}. For establishing the new order relation among these, the information computed at the previous step must be used. For each index i it is kept a pair formed by $P_{(k, i)}$ and $P_{(k, i + 2^k)}$. The fact that $i + 2^k$ may exceed the string's bounds must not bother us, because we shall fill the string with the "$" character, about which we shall consider that it's lexicographically smaller than any other character. After sorting, the pairs will be arranged conforming to the lexicographic order of the strings of length 2^{k+1}. One last thing that must be remembered is that at a certain step k there may be several substrings $A_i^k = A_j^k$, and these must be labeled identically ($P_{(k, i)}$ must be equal to $P_{(k, j)}$).

An image tells more that a thousand words:

bobocel

step 0:
0404123
bobocel

step 1:
0405123
bobocel

obocel$

step 2:
0516234
bobocel
obocel$
bocel$$
ocel$$$

step 3:
0516234
bobocel
obocel$
bocel$$
ocel$$$
cel$$$$
el$$$$$
l$$$$$$
$$$$$$$

Below is a pseudo-code showing the main steps that must be followed:

n ← length(**A**)
for i ← **0**, **n – 1**
    $\mathbf{P_{(0, i)}}$ ← position of $\mathbf{A_i}$ in the ordered array of **A**'s characters
cnt ← 1
for **k** ← **1**, $\mathbf{[\log_2 n]}$ (ceil)
    for **i** ← **0**, **n – 1**
        $\mathbf{L_{(i)}} \leftarrow (\mathbf{P_{(k-1, i)}}, \mathbf{P_{(k-1, i+cnt)}}, \mathbf{i})$
    sort **L**
    compute $\mathbf{P_{(k, i)}}$ **, i = 0, n - 1**
    cnt ← **2 * cnt**

To be noticed that a certain way of numbering the substrings is not necessarily needed, while a valid order relation among them is kept. In order to reach the O(n lg n) complexity, radix sort is recommended (two times count sort), getting an O(n) time complexity per sort operation. To make the implementation easier one may use the **sort()** function from STL(Standard Template Library, a library containing data structures and algorithms in C++). The complexity may be raised to $O(n \lg^2 n)$ worst case, but the implementation will become much easier, the differences being scarcely noticeable for strings with lengths smaller than 100 000.

Here you can see an extremely short implementation for suffix arrays in $O(n \lg^2 n)$.

```
# include <cstdio>
#include <cstring>
#include <algorithm>
using namespace std;

#define MAXN  65536
#define MAXLG 17

char A[MAXN];
struct entry {
    int nr[2], p;
} L[MAXN];
int P[MAXLG][MAXN], N, i, stp, cnt;

int cmp(struct entry a, struct entry b)
{
    return a.nr[0] == b.nr[0] ? (a.nr[1] < b.nr[1] ? 1 : 0) : (a.nr[0] < b.nr[0] ? 1 : 0);
}

int main(void)
{
    gets(A);
    for (N = strlen(A), i = 0; i < N; i ++)
        P[0][i] = A[i] - 'a';
    for (stp = 1, cnt = 1; cnt >> 1 < N; stp ++, cnt <<= 1)
    {
        for (i = 0; i < N; i ++)
        {
            L[i].nr[0] = P[stp - 1][i];
            L[i].nr[1] = i + cnt < N ? P[stp - 1][i + cnt] : -1;
            L[i].p = i;
        }
        sort(L, L + N, cmp);
        for (i = 0; i < N; i ++)
            P[stp][L[i].p] = i > 0 && L[i].nr[0] == L[i - 1].nr[0] && L[i].nr[1] == L[i - 1].nr[1] ?
P[stp][L[i - 1].p] : i;
    }
    return 0;
}
```

The suffix array will be found on the last row of matrix P. Searching the k^{th} suffix is now immediate, so we won't return to this aspect. The quantity of memory used may be reduced, using only the last two lines of the matrix P. It is a trade-off, as in this case the structure will not be able any more to execute efficiently the following operation.

2.2. Computing the longest common prefix (LCP)

Given two suffixes of a string A, compute their longest common prefix. We have shown before that with a suffix tree this can be achieved in O(1), with a corresponding pre-calculation. Let's see if a suffix array can reach the same performance.

Let two suffixes A_i si A_j. Using matrix P, one can iterate descending from the biggest k down to0 and check whether $A_i^k = A_j^k$. If the two prefixes are equal, a common prefix of length 2^k had been found. We only have left to update i and j, increasing them both by 2^k and check again if there are any more common prefixes.

The LCP computing function's code is extremely simple:

```
int lcp(int x, int y)
{
    int k, ret = 0;
    if (x == y) return N - x;
    for (k = stp - 1; k >= 0 && x < N && y < N; k --)
        if (P[k][x] == P[k][y])
            x += 1 << k, y += 1 << k, ret += 1 << k;
    return ret;
}
```

The complexity is O(lg n) for computing one of these prefixes. Reducing this query to an O(1) complexity, is based on the following observation: lcp(x, y) = minimum { lcp(x, x + 1), lcp(x + 1, x + 2), … lcp(y – 1, y) }. The proof is immediate, if we look at the corresponding suffix tree. Therefore it is enough to compute the longest common prefix for all the consecutive pairs of suffixes (O(n lg n) time) and introduce an additional structure that allows minimum range queries in O(1). The most efficient structure is that of RMQ(range minimum query), that we won't discuss in here, being studied in detail in [3], [4] and [5]. With another O(n lg n) preprocessing required by the new structure, we can now answer to the lcp queries in O(1). The structure needed by RMQ is also using O(n lg n) memory, thus the final time and memory are O(n lg n).

2.3. Searching

Since the suffix array offers the order of A's suffixes, searching a string W into A is easily done with a binary search. Since comparing is done in O(|W|), the search will have an O(|W| lg n) worst case complexity. Paper [6] offers both the data structure and the searching algorithm some refinements that allow reducing the time to O(|W| + lg n), but we do not find this very useful during a programming contest

3 Applications in contest problems

We tried to gather as many problems as possible that can be solved using the suffix arrays. Going through all the problems at the first reading would seem rather difficult for a reader who had the first contact with suffix arrays by reading this paper. To make the lecture easier, the problems are arranged in an increasing difficulty order.

Task 1: hidden password (ACM 2003, abridged)

Consider a string of length n (1 <= n <= 100000). Determine its minimum lexicographic rotation. For example, the rotations of the string "alabala" are:
alabala
labalaa
abalaal
balaala
alaalab
laalaba
aalabal
and the smallest among them is "aalabal".

Solution:

Usually, when having to handle problems that involve string rotations, one would rather concatenate the string with itself in order to simplify the task. After, the minimal sequence of length n is requested. As their order is determined by the order of the string's suffixes – although there is a linear solution presented in [10]- suffix arrays are one easy gimmick that can solve the problem instantly.

Problem 2: array (training camp 2004)

Consider an array $c_1c_2...c_n$ consisting of n($1 \le n \le 30\ 000$)) elements from the set {A, B}. Concatenate the array with itself and obtain an array of length 2n. For an index k ($1 \le k \le 2n$) consider the subsequences of length at most n that end on position k, and among these let s(k) be the smallest lexicographic subsequence. Determine the index k for which s(k) is longest. Hint: Let X and Y bet two arrays as defined previously and « o » the concatenation operator. In this problem you will consider that X > X o Y.

Solution:

The searched subsequence is the smallest lexicographic rotation of the given array. Denote by S_i^k the substring of length k that begins on position i. Let S_i^n be the smallest substring of length n in the string obtained by concatenation. Supposing by absurd that s(i + n - 1) < n would mean that there is an i' ($i < i' \le j$) so that $S_{i',\ j-i'+1}$ is smaller than S_i^n. From the condition in the problem's text, we have $S_{i',}^{j-i'+1} > S_{i',}^{n}$; but $S_{i',}^{n} > S_i^n$ => contradiction.

Although there is an O(n) algorithm that would easily solve this, the method used during the contest by one of the authors(and that gained a maximum score) used suffix arrays, as in the previous task.

Problem 3: substr (training camp 2003)

You are given a text consisting of N characters (big and small letters of the English alphabet and digits). A substring of this text is a subsequence of characters that appear on consecutive positions in the text. Given an integer K, find the length of the longest substring that appears in the text at least K times ($1 \leq N \leq 16384$).

Solution:

Having the text's suffixes sorted, iterate with a variable i from 0 to N – K and compute the longest common prefix of suffix i and suffix i + K – 1. The biggest prefix found during this operation represents the problem's solution.

Problem 4: guess (training camp 2003)

You and the Peasant play a totally uninteresting game. You have a large string and the Peasant asks you questions like "does the following string is a substring of yours?" The Peasant asks you many questions and you have to answer as quick s you can. Because you are a programmer, you think that it would be better to know all the substrings that appear in your string. But before doing all this work, you are wondering how many distinct substrings are in your string ($1 \leq$ your string's length $\leq 10\,000$)

Solution:

This is actually asking to compute the number of nodes (without root) of a string's corresponding suffix trie. Each distinct sequence of the string is determined by the unique path traversed in the suffix trie when searching it. As in the example above, „abac" has the substrings „a", „ab", „aba", „abac", „ac", „b", „ba", „bac" and „c", determined by the path starting from the root and going toward nodes 2, 3, 4, 5, 6, 7, 8 and 9 in this order. Since building the suffix trie is not always a pleasant job and has a quadratic complexity, an approach using suffix arrays would be much more useful. Get the sorted array of suffixes in O(n lg n), then search the first position where the matching between every pair of consecutive suffixes fails (using the lcp function), and add the number of remaining characters to the solution.

Problem 5: seti (ONI 2002 – abridged)

Given a string of length N ($1 \leq N \leq 131072$) and M strings of length at most 64, count the number of matchings of every small string in the big one.

Solution:

Do as in the classical suffix arrays algorithm, only that it is sufficient to stop after step 6, where an order relation between all the strings of length $2^6 = 64$ was established. Having sorted the substrings of length 64, each query is solved by two binary searches. The overall complexity is O(N lg 64 + M * 64 * lg N) = O(N + M lg N).

Problem 6: common subsequence (Polish Olympiad 2001 and Top Coder 2004 - abridged)
There are given three strings S_1 , S_2 şi S_3, of lengths m, n and p (1 <= m, n, p <= 10000). Determine their longest common substring. For example, if S_1 = abababca S_2 = aababc and S_3= aaababca, their longest common substring would be ababc.

Solution:
If the strings were smaller in length, the problem could have been easily solved using dinamic programming, leading to a $O(n^2)$ complexity.

Another idea is to take each suffix of S_1 and try finding its maximum matching in the other two strings. A naive maximum matching algorithm gives an O(n^2) complexity, but using KMP [8], we can achieve this in O(n), and using this method for each of S_1's suffixes we would gain an O(n^2) solution.

Let's see what happens if we sort the suffixes of the three strings:
a$
ababababca$
ababca$
abca$
bababca$
babca$
bca$
ca$
aababc#
ababc#
abc#
babc#
bc#
c#

a@
aaababca@
aababca@
ababca@
abca@
babca@
bca@
ca@

Now we merge them (consider $ < # < @ < a …):

a$
a@
aaababca@

aababc#
aababca@
abababca$
ababc#
ababca$
ababca@
abc#
abca$
abca@
babababca$
babc#
babca$
babca@
bc#
bca$
bca@
c#
ca$
ca@

The maximal common substring corresponds to the longest common prefix of the three suffixes ababca$, ababc# and ababca@. Let's see where they appear in the sorted array:

a$
a@
aaababca@
aababc#
aababca@
abababca$
ababc#
ababca$
ababca@
abc#
abca$
abca@
babababca$
babc#
babca$
babca@
bc#
bca$
bca@

c#
ca$
ca@

This is where we can figure out that the solution is a sequence i..j in the array of sorted suffixes, with the property that it contains at least one suffix from every string, and the longest common prefix of the first and last suffix in the suffix is maximum, giving the solution for this problem. Other common substrings of the three strings would be common prefixes for some substring in the suffix array, e.g. bab for **bab**abca$ **bab**c# **bab**ca$, or a for **a**$ **a**@ **a**aababca@ **a**ababc#.

To find the sequence with the longest common prefix, go with two indexes, START and END, over the suffixes, where START goes from 1 to the number of suffixes, and END is the smallest index greater than START so that between START and END there are suffixes from all the three strings. In this way, the pair [START, END] will hit the optimal sequence [i..j]. This algorithm is linear because START takes N values while END is incremented at most N times.

In order to sort the array containing all the suffixes, it is not necessary to sort the suffixes of every string in part and then merge them, as this would increase the complexity if implemented without any smart gimmicks. We can concatenate the three strings into a single one (abababca$aababc@aaababca# for the example above) and then sort its suffixes.

Problem 7: the longest palindrome (USACO training gate)

Given a strings S of length n (n <= 20000) determine its longest substring that also is a palindrome.

Solution:

For a fixed i, computing the longest palindrome that is centered in i requires the longest common prefix of the substring S[i..n] and the reversed S[1..i]. Merge the sorted suffixes of string S and the reversed string S', then query the longest common prefix for S[i] and S'[n – i + 1] (S'[n – i + 1] = S[1..i]). Since this is done in O(lg n), the overall complexity is O(n lg n). The case where the searched palindromes have an even length is treated in an analogous way.

Problem 8: template (Polish Olympiad of Informatics 2004, abridged)

For a given string A, find the smallest length of a string B so that A can be obtained by sticking several B's together (when sticking them, they can overlap, but they have to match).

Example: ababbababbababababbababababbababbaba

Result: 8
The minimum length B is “ababbaba”

A can be obtained from B this way:

```
abababbababababbababababbababbaba
ababbaba
     ababbaba
            ababbaba
                   ababbaba
                         ababbaba
```

Solution:

The simplest solution uses suffix arrays a balanced tree and a max-heap. It’s obvious that the searched pattern is a prefix of A. Having sorted the suffixes of A, we shall add to B, step by step, one more ending character. At each step we keep to pointers L and R, representing the first and the last suffix in the array that have B as a prefix. The balanced tree will hold permanently the starting positions of the suffixes between L and R, and the heap will keep the distances between consecutive elements of the tree.

When inserting a new character into B, by two binary searches we get the new L’ and R’, with L ≤ L’ and R’ ≤ R. We must also update the tree and the heap. Introduce characters into B while the biggest(first) element in the heap is smaller or equal to the length of B. B’s final length offers the searched result. The final complexity is O(n lg n), where n is the length of A.

Solution2 (Mircea Paşoi):

For the string S, compute for every i from 1 to n the length of the longest prefix of S with S[i..n]. This can be done using suffix arrays.
For example, if S is the string and T is the array keeping the maximal matchings, then:

```
S = a b b a a b b a a
T = 9 0 0 1 5 0 0 1 1
```

For every possible k in the pattern (1 <= k <= n) check whether the maximum distance d between the indexes of the two farthest elements with values greater than equal to k in the string T is greater than k. For example:

```
k=9: 9 - - - - - - - -    => d=9, good.
k=8: 9 - - - - - - - -    => d=9, not good.
k=7: 9 - - - - - - - -    => d=9, not good.
k=6: 9 - - - - - - - -    => d=9, not good
k=5: 9 - - - 5 - - - -    => d=5, good.
k=4: 9 - - - 5 - - - -    => d=5, not good.
k=3: 9 - - - 5 - - - -    => d=5, not good.
k=2: 9 - - - 5 - - - -    => d=5, not good..
k=1: 9 - - 1 5 - - 1 1    => d=3, not good.
```

The smallest k where the distance d is small enough represents the length of the searched pattern (in this case k = 5).

To get an algorithm of good complexity, this step must be done efficiently. We may use a segment tree, walk with k from 1 to n and delete from the tree the elements smaller than k, while updating the tree so that it will answer queries like "what is the greatest distance between two consecutive elements in the structure. The algorithm has a complexity of O(n log n). A detailed presentation of the segment tree can be found in [9] and [10].

Problem 9: (Baltic Olympiad of Informatics 2004)

A string *s* is called an *(K,L)-repeat* if S is obtained by concatenating $K \geq 1$ times some seed string *T* with length $L \geq 1$. For example, the string

S = abaabaabaaba

is a (4,3)-repeat with

T = aba

as its seed string. That is, the seed string *T* is 3 characters long, and the whole string *S* is obtained by repeating *T* 4 times.

Write a program for the following task: Your program is given a long string *U* consisting of characters 'a' and/or 'b' as input. Your program must find some *(K,L)*-repeat that occurs as substring within *U* with *K* as large as possible. For example, the input string

U = babbabaabaabaabab

contains the underlined (4,3)-repeat S starting at position 5. Since *U* contains no other contiguous substring with more than 4 repeats, your program must output this underlined substring.

Solution:

We want to find out for a fixed L how to get the greatest K so that in the string U there will be a substring S which is a (K, L) - repeat. Check this example: U = bab**aab**aabaabaaab L = 3 and a fixed substring X = aab that begins on position 4 of U. We can try to extend the sequence aab by repeating it from its both ends as much as possible, as can be seen below:

```
b a b a a b a a b a a b a a a b
a a b a a b
  a b a a b a a b a a b a a b a
```

Extending the prefix of length L this way to the left, then to the right (in our example the prefix of length 3) of the obtained sequence, we get the longest repetition of a string of length L satisfying the

property that the repetition contains X as a substring the (in the case where the repetition is (1, L) this is not true, but it's a trivial case) .

Now we see that in order to identify within U all the (K, L) repetitions with a fixed L, it is sufficient to partition the string in n/L chunks and then extend them. It will not be possible to do this in O(1) for every chunk, thus the final algorithm will have a final complexity of O(n/1 + n/2 + n/3 + .. + n/n) (every chunk can be repeated partially or totally only at left or right, and we will not extend every chunk separately, but we will merge the adjacent chunks into a single one; if we had p consecutive chunks of same length, their maximum extensions would be found in O(p)). But we know that the sum 1 + ½ + 1/3 + ¼ + … + 1/n – ln n is convergent toward a known constant c, known as Euler's constant, and c < 1, so we can easily figure out that O(1 + ½ + … + 1/n) = O(ln n) so the algorithm would have an O(n lg n) complexity if the extensions would have been computed easily.

Now we can use the suffix trees. To find out how much the sequence U[i..j] can be extended to the right, we need to find the longest common prefix of U[i..j] and U[j + 1..n]. To extend it to the left, it's sufficient to reverse U, that would lead to the same problem. We have seen how to compute the longest common prefix in O(1) using the suffix array, that is built in O(n lg n) time, then do the required RMQ pre-calculation in O(n lg n) that allows the lcp queries to be answered in O(1). The final complexity is O(n lg n).

Problem 10: (ACM SEER 2004)

Given a string S, determine for each of its prefixes if it's a periodic string. Hence, for every i (2 <= i <= N) we are interested in the greatest K > 1 (if there exists such one) satisfying the property that S's prefix of length ican be also written as A^k, or A concatenated with itself k times, for some string A. We are also interested which is that k. (0 <= N <= 1000000)

Example: aabaabaabaab
Result:
2 2
6 2
9 3
12 4

Explanation: prefix aa has the period a; prefix aabaab has the period aab; prefix aabaabaab has the period aab; prefix aabaabaabaab has the period aab.

Solution:

See what happens when trying to match a string with one of its suffixes. Take a string and break it in two parts, getting a prefix and a suffix

```
S  =  aab aabaabaaaab
```

```
suf = aab aabaaaab
pre = aab
```

If the suffix matches some number of characters of the initial string which is >= |pre|, it means that pre is also a prefix of the suffix and we can also break the suffix in prefix and suffix1, and the string can be broken in prefix, prefix and suffix1. If the string matches some arbitrary number of characters from the string >= 2|pre| then the suffix matches suffix1 on a number of characters >= |pre| thus suffix1 can be written as prefix and suffix2, then suffix can be written as prefix prefix suffix2, so S can be written as prefix prefix prefix sufix2.

```
S    = aab aab aab aaaab
suf  = aab aab aaaab
suf1 = aab aaaab
pre = aab
```

Observe that if S matches at least k * |prefix| characters of its suffix, then S has a prefix of length (k + 1) * |prefix|, which is periodic.

Using the suffix array, we can find out for each suffix its maximum matching with the initial string. If the i^{th} suffix matches the first k * (i – 1) positions then we can update the information that says that the prefixes of length j * (i – 1) (where 2 <= j <= k) are periodic. For every suffix Si the update has a maximum complexity of O(n/(i – 1)). Thus the algorithm gets an overall complexity of O(n log n).

There is a similar solution using the KMP algorithm, that can be done in O(n), but it doesn't match this paper's purpose.

4 Conclusion

During the contests it is more likely to use the $O(n \lg^2 n)$ solution, slower, but easier to implement. If possible, try not to use more than O(n) memory. The running time of the two solutions is scarcely different for a relatively small input string, and during a contest, the solution's simplicity makes the implementation and debugging considerably easier.

We draw the final conclusion that the suffix arrays are a very useful data structure, extremely easy to implement. Thus it's not strange that during the last years many problems that were using it appeared in programming contests. For any questions or suggestions, please contact the authors using the following e-mail addresses:

azotlichid@yahoo.com
cosminn@gmail.com

References

[1] Mark Nelson - Fast String Searching With Suffix Trees

[2] Mircea Paşoi - Multe "smenuri" de programare in C/C++... si nu numai! (Many programming tricks in C/C++)– http://info.devnet.ro

[3] Emilian Miron – LCA – Lowest common ancestor – http://info.devnet.ro

[4] Michael A. Bender, Martin Farach–Colton - The LCA Problem Revisited

[5] Erik Demaine - MIT Advanced Data Structures – Lecture 11 - April 2, 2003

[6] Udi Mamber, Gene Myers - Suffix arrays : A new method for on-line string searches

[7] Mohamed Ibrahim Abouelhoda, Stefan Kurtz, Enno Ohlebusch - Replacing suffix trees with enhanced suffix arrays, Journal of Discrete Algorithms 2 (2004)

[8] Cormen, Leiserson, Rivest - Introduction to Algorithms

[9] Dana Lica, Arbori de intervale (Segment Trees), GInfo 15/4

[10]Negruşeri Cosmin - Căutari Ortogonale, Structuri de date şi aplicaţii (Orthogonal Searches, Data Structures and Applications), GInfo 15/5

[9] www.boi2004.lv

[10] E. W. Dijkstra – Manuscripts Archive - http://www.cs.utexas.edu/users/EDW