École Polytechnique — CSE202 Design and Analysis of Algorithms

Text indexing with Suffix Array

In this tutorial, we will study text indexation using a suffix array data structure. We have seen in the past tutorials different methods for pattern recognition within a text. You will see that by paying a preprocessing cost to build an index of a text file, some string computations such as pattern recognition can be done very efficiently. This is particularly useful when we know that a text will be queried many times.

Open a new project `CSE202_TD12` in Spyder. Download the file `suffarray.py` that contains the skeletons of the functions you will have to implement plus some functions to help you :

`str_compare(a,b)` **:** function to compare two strings a and b using the lexicographic order. Returns a -1 if $a < b$, 1 if $a > b$, and 0 if $a = b$.

`str_compare_m(a,b,m)` **:** function for m-th lexicographic order string comparison. Same as `str_compare` with an extra integer argument m : check the lexicographic order of the first m characters (i.e return 0 if the first m characters of a and b matches even if $a \neq b$).

`longest_common_prefix(a,b)` **:** function that returns the length of the longest common prefix between strings a and b.

You can use the test file `test.py` which contains functions `testi()` checking the results of your implementation on question i.

1 Suffix Array

A suffix array for a text T of characters within an alphabet Σ is a data structure that stores all the possible suffixes within a text T, ordered lexicographically accordingto Σ (technically, we do not store the suffixes but the position of their starting characters within the text). Below in Figure 1 is an illustrative example.

Introduced by Manber and Myers, the purpose of a suffix array is to play the role of an index to accelerate some queries within the text such as finding substrings matching a given pattern, or computing the longest repeated substrings.

The file `suffarray.py` contains the skeleton of a `suffix_array` class that you will have to implement. This class contains an attribute `self.T` that is a reference (string object) to text string T indexed by the suffix array. It also has an attribute `self.N` storing the length of T and an attribute `self.suffixId` that is a list storing the suffix array as the position of each suffix ordered lexicographically (like in Figure 1). The class also has a method `suffix(self,i)` that returns the suffix string in the suffix array at index i.

i	a	b	r	a	c	a	d	a	b	r	a
0	a	b	r	a	c	a	d	a	b	r	a
1	b	r	a	c	a	d	a	b	r	a	
2	r	a	c	a	d	a	b	r	a		
3	a	c	a	d	a	b	r	a			
4	c	a	d	a	b	r	a				
5	a	d	a	b	r	a					
6	d	a	b	r	a						
7	a	b	r	a							
8	b	r	a								
9	r	a									
10	a										

$\Rightarrow$

SA											
10	a										
7	a	b	r	a							
0	a	b	r	a	c	a	d	a	b	r	a
3	a	c	a	d	a	b	r	a			
5	a	d	a	b	r	a					
8	b	r	a								
1	b	r	a	c	a	d	a	b	r	a	
4	c	a	d	a	b	r	a				
6	d	a	b	r	a						
9	r	a									
2	r	a	c	a	d	a	b	r	a		

FIGURE 1 – Suffix array of the string **abracadabra**. On the left : list of all suffixes and the index i of their first character within the string. On the right : the suffixes stored in lexicographic order.

Question 1. Complete the constructor `__init__(self,t)` by sorting the suffixes according to the lexicographic order. To this end, use the Python `sort` method on lists using an appropriate key function that you have to define (we will use here Python's own string comparison).

Question 2 (Optional exercise)**.** If we assume that the time complexity of comparing two strings of size N is $O(N)$, prove that the complexity of the method we implemented for creating the suffix array is $O(N^2 \log N)$. If we denote by M the size of the alphabet Σ, prove that a binary encoding of the text T requires $O(N \log_2 M)$ space. Show that our suffix array requires $O(N \log_2 N)$ space.

Remark. It is good to know that both the time and space complexity of creating our naive suffix array implementation can be drastically reduced. For the time complexity, there exist approaches that reduce to $O(N \log N)$ complexity (worst case), even down to $O(N)$. The space complexity can also be compressed to $O(N)$ by using appropriate structures for representing the suffix array (see Compressed Suffix Array, or FM-Index). We will not focus on these improvements in this tutorial and concentrate on the usage of suffix arrays.

2 Applications

We will study two typical string problems : the pattern matching and the longest repeated substring.

2.1 Searching for a pattern

We already saw in past tutorials and in the course several algorithms for pattern matching. We will see how we can achieve a better complexity than these previous methods once a suffix array has been computed.

Consider a suffix array SA of a text T, and consider a pattern string S of length m. Finding occurences of S within T boils down to finding suffixes of T prefixed by S. Since the suffix array is ordered lexicographically, we know that if S is a prefix of the suffix induced by $SA[i]$ then other occurrences of S may appear in $SA[i-1]$ or $SA[i+1]$. In other words, we can find a pair (L, R) of index such that S is a prefix of all suffixes induced by $SA[i]$ for all $i \in [L, R-1]$. Formally, these are defined as follows :

$$L = \min\{k : S \leq_m T[SA[k]:]\} \text{ or } N \text{ if empty,}$$
$$R = \min\{k : S <_m T[SA[k]:]\} \text{ or } N \text{ if empty}$$

where $\leq_m$ designates the m-th lexicographic order (this corresponds to `str_compare_m`). If $R \leq L$, then there are no occurrences of S within T. Finding L and R can be done via a dichotomic search within SA.

For example, to find L : start with some range variable $l = -1$ and $r = N$, get the index k in the middle of l and r. If $S \leq_m T[SA[k]:]$, then set r to k and repeat (L must be in the first half of the suffix array), otherwise set l to k and repeats (L must be in the second half of the suffix array). Stop once $r = l + 1$ and return r. To find R, proceed similarly but set l to k when $T[SA[k]:] \leq_m S$.

Remark. You can check that $(l = -1 \vee T[SA[l]:] <_m S) \wedge (r = N \vee S \leq_m T[SA[r]:])$ is an invariant of this algorithm for L.

Question 3. Implement the methods `findL(self,S)` and `findR(self,S)` that compute respectively L and R as above for a given input string S, using a dichotomic search.

Question 4 (Optional exercise)**.** Prove that the time complexity (character comparison) of `findLR(self,S)` (which calls `findL(self,S)` and `findR(self,S)`, see in the file `suffaray.py`) is only $O(m \log N)$ where m is the length of S.

Remark. Using additional data structures (see part 3), it is possible to achieve a $O(m + \log N)$ time complexity.

Once we know the index range (L, R) containing a match of the pattern S, we can simply iterate through all suffixes of the suffix array appearing between L and R (excluded).

Question 5. Implement the function `KWIC(sa, S, c=15)`(Keyword-in-context) that, given a suffix array SA of a text T and a string S, returns the list of occurrences of S within the text T. The integer c corresponds to the context length : you have to add the c characters before and after an occurrence of S (showing in which context the pattern is appearing). For example, on the text `abracadabra` with the text `cad` and context $c = 3$, it should return `['bracadabr']`. Note that you may have to add *less* than c caracters of context at the very beginning or end of the text T. For example, on the text `abracadabra` with the text `bra` and context $c = 3$, it should returns `['abracad', 'adabra']`.

2.2 Longest repeated substring

Given a string T of length N, a repeated substring of T is a string S of length m such that there exist at least two distinct index i and j such that

$$S = T[i : i + m] = T[j : j + m].$$

A longest repeated substring is such a string S with longest size m. This problem (and its variants) has many applications for example in computer biology (finding longest repeated sequence of DNA fragment), data compression, cryptography, etc. We will see how we can take advantage of a suffix array to have an efficient algorithm for this task.

First, we will consider the basic problem of finding the longest common prefix between two strings : given two strings a and b, their longest common prefix is the longest string S of size m such that $S = a[0 : m] = b[0 : m]$ (this implies that $a[m] \neq b[m]$). You can use the function `longest_common_prefix` for this purpose.

We can then imagine a brute-force algorithm (without using a suffix array) to compute the longest repeated substring within a text. The idea is to search for the longest common prefix (using our previous function) of each pair of suffixes of T starting at distinct positions i and j, and remember and return the longest of these common prefixes. Since there are $\binom{N}{2}$ pairs i, j of indices, such a naive algorithm would have to call `longest_common_prefix` $O(N^2)$ times (where each call to `longest_common_prefix` has cost $O(N)$, since in the worst case N characters have to be read).

Using a suffix array, the number of calls to `longest_common_prefix` can be reduced to $N - 1$. The key observation is that each suffix finds its longest common prefix with one of its neighbor in the array (see Figure 2).

SA											
10	**a**										
7	**a**	**b**	**r**	**a**							
0	**a**	b	r	a	c	a	d	a	b	r	a
3	**a**	c	a	d	a	b	r	a			
5	a	d	a	b	r	a					
8	**b**	**r**	**a**								
1	b	r	a	c	a	d	a	b	r	a	
4	c	a	d	a	b	r	a				
6	d	a	b	r	a						
9	**r**	**a**									
2	r	a	c	a	d	a	b	r	a		

FIGURE 2 – Suffix array of the string **abracadabra**. In green, the longest common prefix shared with the next suffix in the array.

Question 6. Implement the function `longest_repeated_substring(sa)` that given a suffix array SA of a text T returns a longest repeated substring in T (in the tests we consider instances where the longest repeated substring is unique).

3 Going further

We can try to extend a bit the previous questions. For example :

— how to modify the function `longest_repeated_substring` in order to compute the longest substring which repeats at least k times ? You can start with $k = 3$.

— in order to improve the complexity of finding longest repeated substring and finding matching pattern, we can see how to add a least common prefix array (lcp array[1]) to the suffix array (see e.g. the Kasai algorithm).

1. `https://en.wikipedia.org/wiki/LCP_array`