Text indexing with suffix arrays

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.

Download the file `suffarray.py` from Moodle, which contains the skeletons of the functions you will have to implement plus some functions to help you:

`str_compare(a,b)`: A function to compare two strings a and b using the lexicographic order. Returns a negative value if $a < b$, a positive value if $a > b$, and zero if $a = b$.

`str_compare_m(a,b,m)`: A function for the m-th lexicographic order string comparison. Same as `str_compare` with an extra integer argument m: checks the lexicographic order of the first m characters (e.g., it returns 0 if the first m characters of a and b match even if $a \neq b$).

`longest_common_prefix(a,b)`: A 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 the functions `testi()` checking the results of your implementation on question i.

1 Suffix arrays

A suffix array for a text T of characters within an alphabet Σ is a data structure that stores all the possible suffixes of a text T, ordered lexicographically according to Σ.[1] An illustrative example is given in Figure 1.

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`, which stores the text (string) T for which we construct the suffix array. It also has an attribute `self.N`, storing the length of T, and an attribute

[1] Technically, we do not store the suffixes but the positions of their starting character within the text.

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: The suffix array of the string **abracadabra**. On the left: a list of all suffixes and the index i of their first character within the string. On the right: the suffixes stored in lexicographic order.

`self.suffixId`, which is a list storing the suffix array as the starting positions of each suffix, ordered lexicographically (like in Figure 1). The class also has a method `suffix(self,i)`, which returns the suffix string in the suffix array at index `i`.

Question 1. Complete the constructor `__init__(self,t)` by sorting the suffixes, `self.suffixId`, in 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).

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: pattern matching and longest repeated substrings.

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 occurrences 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 indices 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:

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

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 variables $l = -1$ and $r = N$, and 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 repeat (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 $T[SA[k]:] \geq_m S$.

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

Question 3 (Optional exercise)**.** Prove that the time complexity (that is, the number of character comparisons) 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 Section 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 (included) and R (excluded).

Question 4. Implement the function `KWIC(sa,S,c=15)` (short for *keyword-in-context*), which, given a `suffix_array` object `sa` (of a text T) and a string `S`, returns the list of occurrences of `S` within T. The integer `c` corresponds to the context length: you have to add the `c` characters before and after an occurrence of `S`, showing the context in which the pattern is appearing. For example, on the text `abracadabra` with the string `cad` and context length `c` $= 3$, it should return `['bracadabr']` (3 additional letters, `bra`, that are neighboring the `cad` occurrence on the left and similarly, on the right). Note that you may have to add *less* than `c` characters of context at the very beginning or the very 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 text T of length N, a *repeated substring* of T is a string S of length m such that there exist at least two **distinct** indices i and j such that

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

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

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` from `suffarray.py` 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 compute the longest common prefix (using our previous function) for each pair of suffixes of T starting at distinct positions i and j, and return the longest of them. 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 neighbors 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 and bold, the longest common prefix shared with the next suffix in the array. No green letters means that this prefix is empty, e.g, the longest common prefix between "adabra" and "bra" is empty.

Question 5. Implement the function `longest_repeated_substring(sa)`, which, given a `suffix_array` object `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; this is not true in general).

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 longest common prefix array (lcp array[2]) to the suffix array (see e.g. the Kasai algorithm).

[2] `https://en.wikipedia.org/wiki/LCP_array`