Tutorial 11

Data Structures 1, 9. 5. 2025 `https://iuuk.mff.cuni.cz/~chmel/2425/ds1en/`

Exercise 1 (Suffix and LCP arrays in practice)
Together, we will trace the building algorithms for both arrays for the string `barokoarokoko`.
After that, you can try it yourselves for the strings `CAGTAGCTGTA`, `TATGTCAGTATCTC`.

Solution

i	X[i]	R[i]	L[i]	sufix
0	13	3	0	ε
1	1	1	5	arokoarokoko
2	6	12	0	arokoko
3	0	10	0	barokoarokoko
4	11	5	2	ko
5	4	8	2	koarokoko
6	9	2	0	koko
7	12	13	1	o
8	5	11	1	oarokoko
9	10	6	3	oko
10	3	9	3	okoarokoko
11	8	4	0	okoko
12	2	7	4	rokoarokoko
13	7	0	–	rokoko

i	SA[i]	LCP[i]	Suffix	RA[SA[i]]	RA[SA[i]+k]	tempRA[SA[i]]
0	11	-	$	0	0	0
1	10	-	A$	1	0	1
2	4	-	AGCTGTA$	2	7	2
3	1	-	AGTAGCTGTA$	3	6	3
4	0	-	CAGTAGCTGTA$	4	2	4
5	6	-	CTGTA$	5	1	5
6	5	-	GCTGTA$	6	9	6
7	8	-	GTA$	7	0	7
8	2	-	GTAGCTGTA$	8	5	8
9	9	-	TA$	9	0	9
10	3	-	TAGCTGTA$	10	11	10
11	7	-	TGTA$	11	0	11

i	SA[i]	LCP[i]	Sorted Suffix T[SA[i]:]	Phi[i]	PLCP[i]	Positional Suffix T[i]
0	11	0	$	1	0	CAGTAGCTGTA$
1	10	0	A$	4	2	AGTAGCTGTA$
2	4	1	AGCTGTA$	8	3	GTAGCTGTA$
3	1	2	AGTAGCTGTA$	9	2	TAGCTGTA$
4	0	0	CAGTAGCTGTA$	10	1	AGCTGTA$
5	6	1	CTGTA$	6	0	GCTGTA$
6	5	0	GCTGTA$	0	1	CTGTA$
7	8	1	GTA$	3	1	TGTA$
8	2	3	GTAGCTGTA$	5	1	GTA$
9	9	0	TA$	2	0	TA$
10	3	2	TAGCTGTA$	11	0	A$
11	7	1	TGTA$	-1	0	$

i	SA[i]	LCP[i]	Suffix	RA[SA[i]]	RA[SA[i]+k]	tempRA[SA[i]]
0	12	-	$	0	0	0
1	6	-	AGTATC$	1	10	1
2	9	-	ATC$	2	0	2
3	1	-	ATGTCAGTATC$	3	5	3
4	11	-	C$	4	0	4
5	5	-	CAGTATC$	5	2	5
6	7	-	GTATC$	6	4	6
7	3	-	GTCAGTATC$	7	6	7
8	8	-	TATC$	8	0	8
9	0	-	TATGTCAGTATC$	9	11	9
10	10	-	TC$	10	0	10
11	4	-	TCAGTATC$	11	8	11
12	2	-	TGTCAGTATC$	12	1	12

i	SA[i]	LCP[i]	Sorted Suffix T[SA[i]:]	Phi[i]	PLCP[i]	Positional Suffix T[i]
0	12	0	$	8	3	TATGTCAGTATC$
1	6	0	AGTATC$	9	2	ATGTCAGTATC$
2	9	1	ATC$	4	1	TGTCAGTATC$
3	1	2	ATGTCAGTATC$	7	2	GTCAGTATC$
4	11	0	C$	10	2	TCAGTATC$
5	5	1	CAGTATC$	11	1	CAGTATC$
6	7	0	GTATC$	12	0	AGTATC$
7	3	2	GTCAGTATC$	5	0	GTATC$
8	8	0	TATC$	3	0	TATC$
9	0	3	TATGTCAGTATC$	6	1	ATC$
10	10	1	TC$	0	1	TC$
11	4	2	TCAGTATC$	1	0	C$
12	2	1	TGTCAGTATC$	-1	0	$

Exercise 2 (Yes, this story is more than two years old)
You managed to save a bunch of elephants from a volcano[1]. Thanks to your expert knowledge of various elephant species, you noticed that you actually have two different species of elephants. Since each species has different dietary preferences, you want to separate them so that none of the elephants get upset with you. Unfortunately, you are not able to distinguish them directly, but you know that they differ in their DNA. Moreover, as true computer science students, you have a DNA sequencing device with you, which also automatically generates the suffix and LCP arrays of the given DNA string.
With each elephant's consent, you sequence the relevant portion of its DNA, which can be represented as a string. After thoroughly studying the manual *How Elephants Differ*, you discovered that the main difference in their DNA is the length of the longest repeating substring of DNA bases. Specifically, if the length of such a substring is odd, the elephant is an Indian elephant, whereas an African elephant has its longest such substring of even length. For each elephant, determine whether it is an African or Indian elephant.
If you're not interested in the story: given a string and its suffix and LCP arrays, find the length of the longest repeating substring.

Solution
Find the largest value in the LCP array, and this corresponds to the length of the longest repeating substring.

Exercise 3 (In search of palindromes)
Given a string α, find, using the LCP array or suffix array the longest palindromic substring of α.

Solution

[1]See the story from Advent of Code 2022, Days 16–18: `https://adventofcode.com/2022/day/16`.

We get a character # that does not appear in the string, and then we consider the LSP, rank and suffix arrays of $\alpha\#\alpha^R$. Then, we build an interval tree that computes the minimum over each subtree of the LCP array.
Then, we split the potential palindroms into palindromes of odd and even length. For palindromes of odd length, we iterate over all indices $i \leq |\alpha|$, and determine the value $\mathrm{LCP}(R[i], R[2|\alpha|+1-i])$ using the tree. For palindromes of even length, we similarly iterate over all indices $i \leq |\alpha|$, and determine the value of $\mathrm{LCP}(R[i], R[2|\alpha|-i])$, and additionally check that $\alpha[i] = \alpha[i-1]$.

Exercise 4 (The elephants are somewhat curious)
When the elephants found out everything your sequencer can do, they became interested in one more thing: they read that a substring P of length n is associated with increased intelligence, and so they are curious whether they have such a predisposition. Given the relevant portion of their genome G of length m, first recall the trivial algorithm for finding a needle in a haystack in time $\mathcal{O}(n \log m)$ (you may assume the suffix array is already constructed).
Next, suppose that for any two strings x, y, you are able to compute $\mathrm{LCP}(x, y)$ in constant time. Modify the trivial algorithm so that it runs in only $\mathcal{O}(\log m)$ time.

Solution
Trivial: binary search over the suffix array and compare strings to see in which direction to go.
Better: ask for LCP(needle, the middle suffix of the haystack) and then look at the next character, and then go left/right based on that.

Bonusy

Exercise 5 (Advent of code intensifies)
But the elephants are also realists, so they are aware that the general LCP cannot be computed in constant time. On the other hand, they have discovered a bunch of other substrings that are allegedly associated with all kinds of traits, so they would be very happy if the algorithm could be as fast as possible.
But what if we could compute the LCP of any two *suffixes* in the genome (haystack) in constant time?[2] In that case, find an algorithm for searching for a needle of length n in a haystack of length m that runs in time $\mathcal{O}(n + \log m)$.
Hint: *In the suffix array, all suffixes of G are sorted lexicographically, so we can perform binary search on it, and we are interested in finding a suffix whose prefix is exactly P. The key is to ensure that, during comparisons, we only compare the relevant parts of the strings so that all comparisons together cost only $\mathcal{O}(n)$. The goal is to use information about common prefixes between elements of the suffix array to maintain information about the common prefix of P with relevant strings in S.*

Solution
During binary search, we always maintain a minimum index i_l and a maximum index i_r in S, which define the shrinking interval where we continue the search, and a middle index $i_m = \lfloor \frac{i_l + i_r}{2} \rfloor$, used to divide the interval. At the same time, we keep track of the length of the common prefix of P with the boundary suffixes: $d_l = \mathrm{LCP}(P, S[i_l])$, $d_r = \mathrm{LCP}(P, S[i_r])$, which we initially compute by comparing character by character.
During each step of the binary search, we choose the boundary with the longer common prefix with P based on comparing d_l and d_r, and then compute the LCP of this boundary with the middle suffix $S[i_m]$. From the result, we may be able to determine $\mathrm{LCP}(P, S[i_m])$: without loss of generality, assume $d_l \geq d_r$. Then, if the resulting $\mathrm{LCP}(S[i_l], S[i_m]) > d_l$, we continue the search in the right half. If the resulting LCP $\mathrm{LCP}(S[i_l], S[i_m]) < d_l$, we also continue in the right half. If they are equal, we proceed by comparing the next character.

Exercise 6 (Actually, the assumption can be weakened)
We know that it is possible to compute the LCP array when given S in linear time (using Kasai's algorithm). Show, that we can then also compute all possible values that the algorithm in the previous exercise uses.

Solution
The tree of the bianry search is fixed and it has linear size, so we can compute values for the given nodes bottom-up: we have leaves immediately from the LCP array, internal nodes are minima of their children.

[2] This assumption is stronger than just having the LCP array, because we are also able to compute the LCP of lexicographically non-adjacent suffixes.

Cheatsheet

- For a string α with Python-style substring notation (i.e., indexing starts at zero, $\alpha[i : j]$ includes all characters from index i to $j - 1$, omitting one coordinate means "from the beginning"or "to the end", respectively), we have the following:
- The suffix array S: $S[i]$ gives the index such that $\alpha[S[i] :]$ is the i-th lexicographically smallest suffix of the string α
- The rank array R: $R[i]$ gives the lexicographic rank of the suffix $\alpha[i :]$
- The LCP array L: $L[i]$ gives the number of characters at the beginning that are shared by $\alpha[S[i] :]$ and $\alpha[S[i + 1] :]$ (i.e., the i-th and $(i + 1)$-th lexicographically smallest suffixes of α)