

Problem Set 2: String Data Structures

This problem set is all about string data structures (tries, suffix trees, and suffix arrays), their applications, and their properties. After working through this problem set, you'll have a deeper understanding of how these algorithms and data structures work, how to generalize them, and some of their nuances. We hope you have a lot of fun with this one!

Due Tuesday, April 28 at 2:30PM Pacific time.

Problem One: An Old Google Interview Question

You are given a collection of strings w_1, w_2, \dots, w_k whose combined total length is n . Design an $O(n)$ -time algorithm that determines whether any of these strings are prefixes of one another. As usual, argue that your algorithm is correct and meets this runtime bound.

Problem Two: Suffix Array Search

Suffix arrays are one of those data structures that make a lot more sense once you're sitting down and writing code with them. To help you get a better handle on suffix arrays, in the remainder of this assignment we're going to ask you to code up some functions that operate on suffix arrays so that you can see what they look like and feel like in practice.

We've provided C++ starter files at `/usr/class/cs166/assignments/a2` and a `Makefile` that will build the project. To get warmed up with the starter files, we'd like you to begin by writing some code that makes you a client of a suffix array.

In the file `Search.cpp`, implement the function

```
vector<size_t> searchFor(const string& pattern,
                        const string& text,
                        const SuffixArray& sa);
```

that takes as input a pattern string, a text string, and a suffix array for that text, then returns a `vector<size_t>` containing all indices where that pattern string matches the text string. Your algorithm should run in time $O(n \log m + z)$, where m is the length of the text string to search in, n is the length of the text string to search for, and z is the number of matches.

To run fully automated correctness tests, execute `./test-search` from the command-line. That program will generate suffix arrays for a bunch of strings, then compare the output of your `searchFor` function against a naive, brute-force search.

For an interactive program that will let you get a sense for what your code is doing, run `./explore` and see what you find!

Some notes to keep in mind:

- Notice that the time for reporting matches is $O(z)$, which doesn't depend on the length of the string being searched for. This means that you will need to spend time $O(1)$ reporting each match.
- By convention, if you search for the empty string in a string of length m , you should get back $m+1$ total matches: one before each character, and one at the end of the string.

Step Two: Implement Induced Sorting

There are two points in this algorithm where you'll need to use induced sorting, so we recommend factoring out the logic to do that into its own helper function (or helper functions, depending on how you want to approach this). The first time you call this function, you'll place the LMS suffixes into the ends of their buckets sorted only by their first character. The second time around, you'll pass the LMS suffixes in in sorted order and position them at the ends of their buckets in the same relative order in which they were passed into the function.

As a refresher, induced sorting first creates an empty suffix array, then makes three linear scans:

- A reverse pass over the input array of LMS suffixes, placing each one into the next free slot at the end of its bucket.
- A forward pass over the suffix array, finding *L*-type suffixes and placing them into the first free slot at the front of their buckets.
- A reverse pass over the suffix array, finding *S*-type suffixes and placing them into the first free slot at the end of their buckets.

If you use the sample string shown above and pass the list of LMS suffixes into your induced sorting function in the order in which they appear in the original string, you should see the following after each pass:

After Pass 1:

\$	A			C								G					T				
21	-	8	13	-	-	-	-	-	-	-	5	10	-	-	-	-	-	-	-	-	
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21

After Pass 2:

\$	A			C								G					T				
21	-	8	13	20	19	-	-	-	-	-	5	10	18	4	9	-	-	7	12	17	3
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21

After Pass 3:

\$	A			C								G					T				
21	13	0	8	20	19	14	5	10	15	1	6	11	18	4	9	16	2	7	12	17	3
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21

Some notes on this part of the algorithm:

- To make this step run efficiently, you'll need to maintain some sort of mapping from characters to the start and end of the range demarcating their buckets. Make sure you can construct this in linear time.
- Remember that the third pass of induced sorting may overwrite the original LMS suffixes placed at the end of each bucket. (You can see this above in the A and C buckets.) This means that you'll need to reset the end of each bucket range before the third pass over the array.

You may also want to test out the sample string from lecture, since you also know what to expect there.

Step Three: Form the Reduced String

The purpose of this first induced sort was to get the LMS blocks into sorted order. If you'll recall, an LMS block is a substring of the original string that spans from one LMS character to another. (The sentinel by itself is also defined to be an LMS block). The goal now is to form the *reduced string* by taking the first LMS suffix and replacing each LMS block with that block's sorted index.

The good news is that by reading off the LMS suffixes in the order in which they appear in the induced-sorted array, you'll get back the LMS blocks in sorted order. Your next task is to assign each block a number corresponding to its sorted index, with all identical blocks receiving the same number. Because the blocks are stored in sorted order, you just need to check whether each block is equal to the one before it. If so, it's a duplicate and gets the same number as the block before it. If not, it gets the next block number. (The sentinel as a block by itself always gets index 0).

To check whether two blocks are equal, just do a linear scan over their characters and see if they match. (In lecture, we talked about a modified comparison method in which we compared both the characters and their *L/S* types; when you're coding this up, you *do not* need to do this. You can just compare the characters themselves.) Remember that the end of a block is delimited by the next LMS character in the string. Fortunately, you don't need to do any special bounds-checking here – thanks to the sentinel, your scans will never be able to run off the string.

Continuing from our worked example above, the LMS suffixes you should get back from the induced-sorted array will be in the order 21, 13, 8, 5, 10. Those correspond to these LMS blocks:

\$ ACCGTGCC\$ AGC CCTA CCTA

If you've done everything properly at the end of this step, you should end up with this reduced string:

3 2 3 1 0

To see where this comes from, notice that the first LMS suffix in the original string is

CCTAGCCTACCGTGCC\$

which is then broken down into

CCTA AGC CCTA ACCGTGCC\$ \$

which, once the blocks are mapped to their index, works out to 32310.

Some things to watch out for in this step:

- This step involves maintaining indices into several different arrays. There are indices into the original text string, into the suffix array, into the partially-sorted list of LMS suffixes, and into the newly-formed reduced string. Be careful to keep all of these separated – naming your variables intelligently can help a lot.
- Make sure that your logic to assign numbers to the LMS blocks runs in $O(m)$ time. Because the LMS blocks are in sorted order, you only need to compare adjacent blocks and don't need to check all pairs against one another.
- Once you've determined what number each LMS block should be assigned, you need to find a way to put that block label into the right spot in the reduced string in time $O(1)$. There are many ways to do this. Be creative!
- If you've done everything properly, the reduced string should always end with a 0. There is no need to manually append anything. (Do you see why this is guaranteed to happen?)

Again, feel free to use the running example from lecture as another way of checking your work.

Step Four: Sort the LMS Substrings

You now have your reduced string. Your task at this point is to form the suffix array for that reduced string, which will then let you sort the LMS suffixes. (Oh yeah... that's why we're doing this!)

There are two possibilities here. First, imagine that your reduced string had no duplicated blocks in it. In that case, that reduced string is some permutation of $0, 1, 2, \dots, k - 1$, where k is the total number of blocks. In this case, you can compute the suffix array directly in time $\Theta(k)$.

The other option is that there's at least one duplicated block. In that case, it's not immediately obvious how to compute the suffix array for the reduced string, but that's okay! We can use recursion to do this for us! The "real" way to do this is to insert a recursive call to SA-IS on the reduced string. But since you're still getting this algorithm up and running, we recommend, for the moment, cheating and placing a call to another one of the suffix array construction algorithms packaged with the starter files. The `dc3` function uses another linear-time suffix array construction algorithm (Difference Cover 3) and it runs pretty fast, so feel free to place in a call to `dc3` here.

To check your work, if you're following along with the running example, you should get back this suffix array:

4 3 1 2 0

Once you've done this, you can use this suffix array to sort the LMS substrings. How? Well, if you have an array consisting of all the LMS suffixes in the order in which they appear in the original string, you just need to reorder them so that their order matches the order given by the suffix array. You might find this step pretty similar to the one you did in the case where all the blocks were unique.

To check your work, you should end up with the LMS suffixes in this order:

21 13 8 10 5

Some notes:

- The SA-IS algorithm is unusual in that the optimization described here (manually forming the suffix array when all the blocks are distinct) serves as the base case for the recursion. ***Don't skip this step***, since otherwise you'll end up with nonterminating recursion later on!
- Remember that the suffix array is set up so that $SA[i]$ is the index of the i th-smallest suffix, rather than the position that the suffix starting at position i occupies in sorted order.

Step Five: The Last Induced Sort, and Making Things Recursive

Home stretch! Now that you have the LMS suffixes in their proper order, you can make one last call to induced sorting given this ordering of the LMS suffixes. That will properly put all the suffixes in the right order, and you have your suffix array!

Here's what the suffix array should look like at the end of each of the three passes:

After Pass 1:

\$	A			C								G					T				
21	-	13	8	-	-	-	-	-	-	-	10	5	-	-	-	-	-	-	-		
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21

After Pass 2:

\$	A			C								G					T				
21	-	13	8	20	19	-	-	-	-	-	10	5	18	9	4	-	-	12	7	17	3
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21

After Pass 3:

\$	A			C								G					T				
21	13	0	8	20	19	14	10	5	15	1	11	6	18	9	4	16	2	12	7	17	3
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21

That last one is the final suffix array!

If your output matches ours, go back to where you compute the suffix array for the reduced string. Replace the call to `dc3` with a recursive call to `sais`. Try running the algorithm one more time to see if it still works. If so, great! If not, take some time to debug what's going on.

Step Six: Test, Test, Test!

At this point you have an initial draft of your implementation, and all that's left to do is to test it and smoke out any remaining bugs.

Run the program

```
./test-sais
```

from the command-line. This will subject your implementation to a pretty serious battery of tests, starting with the sample string shown above and the one from lecture, then progressing to much bigger examples. It'll compare your answer against the suffix array generated by our DC3 implementation and let you know if anything fails.

And then that's it! You're done! Change the compiler flags in the `Makefile` to disabling debugging and ramp the optimizer to `-Ofast`. Now run the `./explore` program and try feeding in some of the sample data files. Find any interesting strings or patterns in them?