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 Thursday, April 25 at 2:30PM
You are encouraged to work on this assignment in pairs.

Problem One: Time/Space Tradeoffs in Tries (10 Points)

If you code up a trie, you'll quickly run into a design choice: how do you represent the child pointers associated with each node? This decision shouldn't be taken lightly; it will have an impact on the amount of time required to perform operations on the trie and on the space used to encode the trie. In this problem, you'll analyze the time/space tradeoffs involved in three different trie representations.

Here's some notation we'll use throughout this problem:

- We'll have ***k*** denote the total number of words stored in our trie.
- We'll have ***t*** denote the total number of nodes in the trie.
- We'll have **Σ** denote the alphabet from which the strings in our trie are drawn.

An assumption that's common throughout Theoryland that's also usually true in practice is that, given some character $x \in \Sigma$, it's possible to map x to some integer in $\{0, 1, 2, \ldots, |\Sigma| - 1\}$ in time O(1), making it possible to do index-based lookups on characters efficiently. We'll also assume that each character fits comfortably into a machine word.

For the purposes of this problem, we'll imagine that each node in the trie is represented with some sort of structure that looks like this:

```
struct TrieNode {
    bool isWord;
    Collection<char, TrieNode*> children;
};
```

Here, the `isWord` data member is `true` if the node represents a word and is `false` otherwise. The children data member represents a mapping from characters to the children labeled with those nodes. Here, `Collection` is a placeholder for whatever type of collection we use to store the pointers.

Note that the above node structure means that we won't be using Patricia tries or endmarking our words with a sentinel character. We'll just be using a vanilla trie.

First, suppose you decide to store the child pointers in each trie node as an array of size $|\Sigma|$. Each pointer in that array points to the child labeled with that character and is null if there is no child with that label.

i. Give a big-Θ expression for the total memory usage of this trie. Briefly justify your answer.

ii. Give the big-O runtime of determining whether a string of length L in contained in a trie when the trie's children are represented as an array as in part (i). Briefly justify your answer.

Now, let's imagine that you decide to store the child pointers in each trie node in a binary search tree. (A trie represented this way is sometimes called a ***ternary search tree***.)

iii. Give a big-Θ expression for the space used by a ternary search tree. As a way of checking your work, your overall expression should have no dependence on $|\Sigma|$. (Whoa! That's unexpected!) Briefly justify your answer.

iv. Suppose each node in the ternary search tree stores its children in a balanced BST. Briefly explain why the cost of a lookup in the ternary search tree is $O(L \log |\Sigma|)$.

As you can see here, there's a time/space tradeoff involved in using ternary search trees as opposed to an array-based trie. The memory usage is lower for the ternary search tree than for the array-based trie, but the lookups are slightly slower.

(Continued on the next page)

If you're working in computational biology where Σ is often `{A, C, T, G}`, then the cost of an extra factor of $|\Sigma|$ or $\log |\Sigma|$ isn't going to be too big. On the other hand, if you're working with general text, where Σ might be all of Unicode, including emojis, the impact of a $|\Sigma|$ or $\log |\Sigma|$ term can be enormous. Are there ways of representing tries where neither the time nor space depends on $|\Sigma|$?

Amazingly, the answer is yes, using a tool called a ***weight-balanced tree***. A weight-balanced tree is a binary search tree in which each key x_i has an associated weight $w_i > 0$. A weight-balanced tree is then defined as follows: the root node is chosen so that the difference in weights between the left and right subtrees is as close to zero as possible, and the left and right subtrees are then recursively constructed using the same rule.

v. Draw a weight-balanced tree containing the following key/weight pairs.

$$(a, 2) \quad (b, 7) \quad (c, 1) \quad (d, 8) \quad (e, 4) \quad (f, 5) \quad (g, 9)$$

vi. Suppose that the total weight in a weight-balanced tree is W. Prove that there is a universal constant ε (that is, a constant that doesn't depend on W or the specific tree chosen) where $0 < \varepsilon < 1$ and the left subtree and right subtree of a weight-balanced tree each have weight at most εW.

Let's now bring things back around to tries. Imagine that we start with a ternary search tree, which already stores child pointers in a BST, but instead of using a regular balanced BST, we instead store the child pointers for each node in weight-balanced trees where the weight associated with each BST node is the number of strings stored in the subtrie associated with the given child pointer.

For example, consider the trie from Slide 10 of our lecture on suffix trees. Focus on the node associated with the word `ant`. Then its child labeled `e` would have weight 3, since that subtrie contains three words, and its child labeled `i` would have weight 1, since that subtrie contains just one word.

vii. Prove that looking up a string of length L in a trie represented this way takes time $O(L + \log k)$. As a hint, imagine that you have two bank accounts, one with $O(L)$ dollars in it and one with $O(\log k)$ dollars with it. Explain how to charge each unit of work done by a search in a trie represented this way to one of the two accounts without ever overdrawing your funds.

Weight-balanced trees have a bunch of other wonderful properties, and we'll spin back around to them later in the quarter.

There are a bunch of other ways you can encode tries, each of which has its advantages and disadvantages. If you liked this problem, consider looking into this as a topic for your final project!

Problem Two: The Anatomy of Suffix Trees (2 Points)

Consider the following paragraph, which is taken from an English translation of the excellent short story "Before the Law" by Franz Kafka:

```
Before the law sits a gatekeeper. To this gatekeeper comes a man from
the country who asks to gain entry into the law. But the gatekeeper
says that he cannot grant him entry at the moment. The man thinks about
it and then asks if he will be allowed to come in later on. "It is pos-
sible," says the gatekeeper, "but not now."
```

Actually building a suffix tree for this text, by hand, would be quite challenging. But even without doing so, you can still infer much about what it would look like.

i. Without building a suffix tree or suffix array for the above text, determine whether the suffix tree for this passage contains a node labeled with the string `moment`. Briefly justify your answer.

ii. Repeat the above exercise for the string `man`.

iii. Repeat the above exercise for the string `gatekeeper`.

Problem Three: Longest *k*-Repeated Substrings (3 Points)

Design an O(m)-time algorithm that, given a string T of length m and a positive integer k, returns the longest substring that appears in T in at least k different places. The substrings are allowed to overlap with one another; for example, given the string `HAHAHAHAHA` and $k = 3$, you'd return `HAHAHA`.

As a note, the runtime of O(m) is independent of k. For full credit, your solution use should not use suffix trees, though you're welcome to use any other data structures you'd like.

Problem Four: Suffix Array Search (3 Points)

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/ps2` 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.

Problem Five: Implementing SA-IS (12 Points)

Your final task is to implement a function

```
SuffixArray sais(const vector<size_t>& text);
```

that takes as input a text string (described below), then uses the SA-IS algorithm to construct a suffix array for that string.

You might notice that the input to this function is a list of numbers rather than a string. To make things a bit easier, you'll receive your input as a sequence of numbers representing the rank order of the characters of the original string. For example, if the string in question is `abracadabra$`, we'd give you as input the sequence `[1, 2, 5, 1, 3, 1, 4, 1, 2, 5, 1, 0]`. You don't need to append a sentinel; the input will always be terminated with a `0`, representing the sentinel at the end of the string.

There's a fair amount of code to write here, but if you proceed slowly and test each piece of your code as you go, we think you'll find that it's not that bad. Our reference solution is under 250 lines and is well-commented and decomposed. (In other words, we weren't optimizing for line counts by sacrificing code quality.) Be sure to run your code in `valgrind` with optimization turned off as you're doing your development – it's a great way to catch off-by-one errors.

To help you get SA-IS working, we've put together a breakdown of the individual steps of the algorithm, along with some advice and a worked example that will help you check that each step is working properly.

Step One: Annotate each character and find the LMS characters.

Your first step is to take the input string and to tag each suffix as either *S*-type or *L*-type.The easiest way to do this is with a reverse scan of the characters in the input string.

As long as you're classifying suffixes this way, we recommend that you also create a list of the starting positions of all the LMS-type suffixes. You'll need this later on. Our reference solution builds up this list in the order in which those LMS suffixes appear.

You can test your code by running

```
./run-sais-on [sample-string]
```

from the command-line. Don't include a sentinel at the end of your string; our program will do that for you. As a test case, given the input string `ACGTGCCTAGCCTACCGTGCC`, your program should classify the characters as follows, with LMS strings marked with stars:

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

We also recommend checking your implementation against our sample DNA sequence from Tuesday's lecture, since we've already worked out all the intermediate steps for you. Some notes:

- Remember that the sentinel is defined to be *S*-type.
- In order for a suffix to be an LMS suffix, it must be preceded by an *L*-type suffix. This means that an *S*-type suffix at the beginning of the string is not considered an LMS suffix. (The one exception: the sentinel is ***always*** considered to be an LMS suffix.)

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 induce-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 induce-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, …, $k - 1$, 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 ith-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 `-O3`. Now run the `./explore` program and try feeding in some of the sample data files. Find any interesting strings or patterns in them?