Problem Set 3: 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, May 4 at noon Pacific time.

Problem One: The Anatomy of Suffix Trees

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 possible," 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 where the path from the root of the suffix tree to that node spells out the string `moment`. (The period after the string `moment` is the period at the end of the sentence and isn't part of the string `moment` itself.) Briefly justify your answer.

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

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

Problem Two: Longest *k*-Repeated Substrings

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 Three: Representing Suffix Trees

In lecture, we claimed that a suffix tree for a string T can be represented in $O(m)$ space, where $m = |T|$. This glosses over an important detail: the $O(m)$ term hides a dependency on the number of distinct characters in T. The details matter and are more important than they might seem.

Some preliminary terms and definitions for this problem:

- We'll let m denote the length of the string T our suffix tree is built for. We'll let Σ denote the alphabet the characters in string T are drawn from, including the sentinel character `$`. We are interested in queries of the following form: given a pattern string P, determine whether P is a substring of T. We'll assume that $|P| = n$, following the convention from lecture.
- We'll say that a suffix tree representation is $\langle \boldsymbol{s}, \boldsymbol{q} \rangle$ if it uses s space and has query time q.
- Throughout this problem, we'll use a standard space-saving technique. We will write a copy of T separately from the suffix tree and represent edge labels as pairs of integer indices into T denoting each label's start and end position. Storing edges this way requires only $O(1)$ space per edge.

Here is an initial representation of a suffix tree. Assume that the characters in Σ are conveniently numbered 0, 1, 2, …, $|\Sigma| - 1$, so we can use characters as indices into an array. Each internal node in the tree maintains an array of $|\Sigma|$ pointers, one for each possible character. Each array slot corresponds to a character c and is either a null pointer (if there is no child whose edge starts with that character), or stores a pointer to the child whose edge starts with c, alongside the label used by that pointer.

i. Explain why this is a $\langle O(m \cdot |\Sigma|), O(n) \rangle$ suffix tree representation.

Now, consider the following alternative representation of a suffix tree. Each suffix tree internal node will store its child pointers in a BST. Specifically, let s be a node in the suffix tree that has an edge labeled l to a node t in the suffix tree. Then s's BST will contain a node with key l that, in addition to the standard BST left and right pointers, stores a pointer to t. The keys in each BST are sorted by edge label.

ii. Explain why this is a $\langle O(m), O(n \log |\Sigma|) \rangle$ suffix tree representation.

We'll see a third way of representing a suffix tree, but to do so, we first need to introduce a new data structure. A ***weight-equalized tree*** is a binary search tree in which each key x_i has an associated weight $w_i > 0$. A weight-equalized 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.

iii. As a warm-up, draw a weight-equalized tree containing the following key/weight pairs.

$$(a, 1) \quad (b, 3) \quad (c, 4) \quad (d, 5) \quad (e, 8) \quad (f, 7) \quad (g, 9)$$

iv. Suppose that the total weight in a weight-equalized 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-equalized tree each have weight at most εW.

Now, our third representation of a suffix tree. Associate each suffix tree node s with its ***size***, denoted **size(*s*)**, the number of suffix tree leaves reachable from s. For example, size(s) is 1 if s is a leaf, and size(s) is m if s is the root of the suffix tree. We represent the suffix tree as follows: each suffix tree node s stores its child pointers in a weight-equalized tree. If s has an edge labeled l to a suffix tree node t, then s's weight-equalized tree has l as a key with weight size(t). As above, BST nodes are sorted by edge labels.

v. Explain why this is a $\langle O(m), O(n + \log m) \rangle$ suffix tree representation. We want you to formally prove that the query time is correct. As a hint, split the cost of performing the lookup into two pieces: the cost of traversing links within the original suffix tree, and the cost of traversing links within the weight-equalized trees encountered along the way.

(Continued on the next page…)

And now, one final representation of a suffix tree. Color the nodes of the suffix tree one of two colors. If a node has size $|\Sigma|$ or greater, we paint it red. Otherwise, we paint it blue. The net effect of this is that the suffix tree now consists of a red tree containing the root node and many of its descendants, plus a large number of small blue trees hanging off that red tree.

Focus purely on the red tree, ignoring blue nodes. It contains some number of leaves, corresponding to internal suffix tree nodes where all the children have size less than $|\Sigma|$. It contains some number of nodes with exactly one child, which correspond to suffix tree nodes where all but one of its children have size less than $|\Sigma|$. And it contains some number of branching nodes, corresponding to suffix tree nodes with two or more children of size $|\Sigma|$ or greater.

vi. Prove that the number of red leaves and red branching nodes is at most $2(m+1) / |\Sigma|$.

Here's our final suffix tree representation. We'll represent the red tree using the following approach. Each red leaf or red branching node will be represented using the array-based strategy from part (i). Each red node with exactly one child will have that child represented by writing down the edge label using our standard space-saving technique. We will represent all blue subtrees using the weight-equalized strategy from part (v). Each red node will then store a BST, along the lines of in part (ii), containing the suffix tree edges pointing from red nodes to blue nodes. When performing a search, we will always first attempt to follow edges between red nodes, and will only try following edges to blue nodes if no such edge exists.

vii. Prove that this is a $\langle O(m), O(n + \log |\Sigma|)\rangle$ representation of a suffix tree.

That very last strategy might remind you a bit of our hybrid RMQ approaches. We split a larger object (our suffix tree) into a "summary structure" (the red tree) and a bunch of "block structures" (the blue trees hanging off of it). We then choose different data structures for the summary and blocks, and tune the block size so that in putting everything together we end up with something that's asymptotically superior to each of the pieces we built the structure from. This style of approach isn't a coincidence – turns out that lots of efficient data structures can be built this exact way!

Problem Four: 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. We'd like you to play around with them in the context of implementing the searching algorithm for suffix arrays we saw in Thursday's lecture.

We've provided C++ starter files at `/usr/class/cs166/assignments/a3` 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. Additionally, your algorithm should return the matches in the same relative order that they appear in the suffix array.

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.
- You only need to edit the file `Search.cpp`.
- Our provided starter files include two algorithms for building suffix arrays: the Manber-Myers algorithm ($O(m \log n)$) and DC3 ($O(m)$). You're welcome to check those out if you're curious.