Problem Set 2: String Data Structures

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

Due Tuesday, April 19 at the start of lecture.

Problem One: Trie Representations (4 Points)

In a trie, each node stores a collection of pointers to child nodes. The performance of the basic operations on tries and the space efficiency of a trie depend directly on what data structure is used to represent those pointer collections. This problem explores several possibilities and their tradeoffs.

In what follows, imagine you have a trie storing n words and having N total nodes. Denote the alphabet of the trie as Σ. When performing a query, let w be the query string.

i. Fill in the table below by writing the time complexities of the specified operations and the space complexity of the overall trie assuming that the indicated data structure is used to store the child pointers. In the case of a hash table, assume that all hash table operations run in their expected, amortized cost of O(1) per operation. Make sure you account for the size of the alphabet.

	lookup	***insert***	***delete***	Space
Array of $\|\Sigma\|$ pointers, one for each character.	$\mathbf{O(\|w\|)}$			$\mathbf{\Theta(N \cdot \|\Sigma\|)}$
Balanced BST.				
Hash table (assume each operation runs in its expected runtime).				

When filling out this table, you can assume that allocating a block of uninitialized memory of any size takes time O(1) and that deallocating a block of memory of any size takes time O(1).

ii. There's another useful operation on tries called a ***successor search***. In this operation, the input is some string w, which may or may not be a word in the trie, and the output is the lexicographically first word in the trie that comes lexicographically after w. For example, if you had a trie for the set of words {a, art, bee, beet} and did a successor query for the word "ant," the query would return "art," which is the first word that comes after "ant." Given a query for "bee," the result would be "beet" (since beet is the first word that comes strictly after "bee.") Finally, given a query for "cat," the result would be null, since there are no words in the trie that come after "cat."

Briefly describe, at a high-level, how you'd implement the successor search operation on a trie. Then, for each of the above representations of the child pointers of the nodes in a trie, determine the runtime of the successor search. In doing so, you can denote by $\sigma(w)$ the string returned by the successor search for w.

The rest of this problem explores ***weight-balanced trees*** and their use in tries.

Suppose that you have a set of keys $x_1, \ldots, x_n$ that you'd like to store in a binary search tree. Each key is associated with a positive weight $w_1, \ldots, w_n$. We'll define the ***weight*** of a BST to be the sum of the weights of all the keys in that tree. 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.

Now, suppose that you have a static trie in which the set of words is fixed and no insertions or deletions may be performed. Consider the following representation of such a trie: each node in the trie stores its child pointers in a weight-balanced tree where the weight associated with each node is the number of strings stored in the subtrie associated with the given child pointer.

iii. Suppose that the total weight in a weight-balanced tree is W. Prove that there is a constant ε such that $0 < \varepsilon < 1$ with the following property: the left subtree and right subtree of a weight-balanced tree each have weight at most εW.

iv. Prove that any lookup in this representation of a trie requires time $O(|w| + \log n)$.

v. What is the space usage of a static trie represented this way?

Problem Two: The Knuth-Morris-Pratt Algorithm (4 Points)

Consider a special case of Aho-Corasick where there is just one pattern string P that you'd like to search for. In this special case, it's possible to significantly optimize the Aho-Corasick matching algorithm into another algorithm called the ***Knuth-Morris-Pratt algorithm***, or ***KMP*** for short. This problem will ask you to reason about Aho-Corasick and to see where KMP comes from.

i. What shape will the Aho-Corasick matching automaton have if there is just one pattern string?

ii. Of the three types of links used in Aho-Corasick (trie edges, suffix links, and output links), which of these types of links – if any – can safely be eliminated if there is only one pattern string? Why?

iii. Based on your answers to parts (i) and (ii) of this problem, write an implementation of the Aho-Corasick algorithm in the special case of a single pattern string. Your algorithm should not represent the matching automaton as a trie, but should instead represent everything implicitly as arrays, in a style similar to how a binary heap (a tree structure) is typically represented as an array, with links between the nodes represented implicitly through manipulations of the indices. Your algorithm should match the normal time bounds for the Aho-Corasick algorithm (linear preprocessing time, linear query time).

   To complete this part of the assignment, download the starter files from

   `/usr/class/cs166/assignments/ps2`

   and implement the appropriate types and functions in the `kmp.c` source file. We recommend testing your implementation extensively, as there are a number of edge cases to watch out for. You may want to use our provided test harness as a starting point, though you will almost certainly need to introduce your own tests on top of ours.

   To receive full credit on this part of the assignment, your code must compile will no warnings and should run cleanly under `valgrind` (that is, it should never cause memory errors). We will test your code on the `corn` machines, so we recommend you test there before submitting.

iv. Give a quick write-up of your implementation. How did you choose to represent the matching automaton? How does your implementation simulate trie links, suffix links, and/or output links?

Problem Three: *k*-Approximate Matching (3 Points)

The normal string matching problem is too restrictive in many genomics applications where, due either to sequencing errors or random mutations, a pattern string P might not exactly match anywhere in T even though it *almost* matches. The ***k-approximate matching problem*** is the following: given a string T of length m, a string P of length n, and a value $k \geq 0$, determine whether there is a substring of T of length n that matches P in all but at most k places. For example, given the strings

T = `thatthatisisthatthatisnotisnotisthatitititis` P = `matisse`

then P can be placed at the following location relative to T such that it matches in all but 3 places:

```
thatthatisisthatthatisnotisnotisthatitititis
      matisse
```

However, there is no way to place P such that it matches in all but 2 places.

Design an $O(mk + n)$-time algorithm for solving the k-approximate matching problem and prove that your algorithm is correct. In the common case where $k \ll n$, this can be significantly faster than the naïve algorithm.

Problem Four: Variations on DC3 (2 Points)

Consider a variation on DC3 called DCk. In this variation, instead of computing the relative order of the suffixes at indices of T that are nonzero mod 3, we compute the relative order of the suffixes at indices that are nonzero mod k using an appropriate generalization of the approach used by DC3. We then use that ordering to sort the suffixes at positions that are zero mod k, then merge together the suffixes using a generalization of the DC3 approach

i. Will DC2 correctly compute a suffix array in time $O(m)$? If it does, why isn't it used instead of DC3? If it doesn't, why not? Justify your answer.

ii. Will DC4 correctly compute a suffix array in time $O(m)$? If it does, why isn't it used instead of DC3? If it doesn't, why not? Justify your answer.