CS 224 HW 3 - take two weeks. Some of the problems are somewhat vague and open ended. Do what you can.

1. In class we saw that BWT[i] = T[SA[i] - 1], where T is the input string, BWT[i] is the character at position i in the BWT string for T, and SA[i] is the i'th entry in the suffix array for T.

This shows how to compute the BWT from the SA. What about the other direction? If we have the BWT, what else would we need to compute the SA, and in linear time? I don't know the answer.

2. Read the BWT inversion method that I posted on the class website. That method derives the original string from the left to the right, the opposite of what is done in the FM approach. As was done in the FM method, it should be possible to define pointers that can be followed to spell out the original string in forward order. Analogous to what was done in the FM paper, define those pointers and write up a formula for them. Show how those pointers relate to the FM pointers.

3. We know how to go from a Suffix Tree to a SA via a lexicographic dfs, and we know how to go from a SA to the BWT, as detailed in problem 1. So it seems it should be possible to come up with concise statement of what the BWT string is in terms of the suffix tree for an input string. Actually, we did this in class on thursday. Given that, we should be able to describe what the FM BWT inversion algorithm is doing on the suffix tree. Do it. Similarly, can you see what the FM exact search algorithm looks like on the suffix tree - describe it as simply as possible.

4. The FM exact matching method finds occurrences of P in T working from the right end of P to the left end of P. It runs in time proportional to the length of P. Can you devise a similar algorithm that runs in time proportional to the length of P that works from the left end. It may be helpful to remember that there are methods that invert the BWT that work from the left end.

5. In the Ferragina and Manzini JACM paper (posted on the class website) in Section 2.3 they describe a compression algorithm for the BWT string. It first uses MTF and then a run-length-encoder, but just of the zeros. It seems to me that this is just the same as doing a run-length-encoding of runs of identical characters in the BWT string. Am I missing something? Can you see what the move to front is contributing? Maybe it is just recoding to a smaller alphabet, or at least one where the distribution of characters is more heavilly weighted towards small integers. That would be helpful for large alphabets (English), but not so important in small alphabets (DNA).

Discuss.

6. In this guided exercise, you will work out the original suffix array algorithm given in the Manber and Myers paper. The method involves a general technique called "successive refinement", which is valuable to know, in addition to its use in suffix arrays.

Successive refinement methods

Successive refinement is a general algorithmic technique that has been used for a number of string problems [**?**, **?**, **?**]. In the next several exercises, we introduce the ideas, connect successive refinement to suffix trees, and apply successive refinement to particular string problems.

Let S be a string of length n. The relation E_k is defined on pairs of suffixes of S. We say iE_kj if and only if suffix i and suffix j of S agree for at least their first k characters. Note that E_k is an equivalence relation and so it partitions the elements into equivalence classes. Also, since S has n characters, every class in E_n is a singleton. Verify the following two facts:

Fact 1: For any $i \neq j$, $iE_{k+1}j$ if and only if iE_kj and $i+1E_kj+1$.

Fact 2: Every E_{k+1} class is a subset of an E_k class and so the E_{k+1} partition is a refinement of the E_k partition.

We use a tree T, called the *refinement tree*, to represent the successive refinements of the classes of E_k as k increases from 0 to n. The root of T represents class E_0 and contains all the n suffixes of S. Each child of the root represents a class of E_1 and contains the elements in that class. In general, each node at level l represents a class of E_l and its children represent all the E_{l+1} classes that refine it.

Now modify T as follows. If node v represents the same set of suffixes as its parent node v', contract v and v' to a single node. In the new refinement tree, T', each non-leaf node has at least two children. What is the relationship of T' to the suffix tree for string S? Show how to convert a suffix tree for S into tree T' in $O(n^2)$ time.

Several string algorithms use successive refinement without explicitly finding or representing all the classes in the refinement tree. Instead, they construct only some of the classes or only compute the tree implicitly. The advantage is reduced use of space in practice, or an algorithm that is better suited for parallel computation [**?**]. The original suffix array construction method [**?**] is such an algorithm. In that algorithm, the suffix array is obtained as a byproduct of a successive refinement computation where the E_k partitions are computed only for values of k that are a power of two. First we need an extension of Fact 1:

Fact 3: For any $i \neq j$, $iE_{2k}j$ if and only if iE_kj and $i+kE_kj+k$.

From Fact 2, the classes of E_{2k} refine the classes of E_k.

The algorithm of [**?**] starts by computing the partition E_1. Each class of E_1 simply lists all the locations in S of one specific character in the alphabet, and the classes are arranged in lexical order of those characters. For example, for $S =$*mississippi\$*, E_1 has five classes: $\{12\}, \{2,5,8,11\}, \{1\}, \{9,10\}, \{3,4,6,7\}$. The class $\{2,5,8,11\}$ lists the position of all the i's in S and so comes before the class for the single m, which comes before the class for the s's etc. The end-of-string character \$ is considered to be lexically smaller than any other character.

How E_1 is obtained in practice depends on the size of the alphabet and the manner that it is represented. It certainly can be obtained with $O(n \log n)$ character comparisons.

For any $k \geq 1$, we can obtain the E_{2k} partition by refining the E_k partition, as suggested in Fact 3. However, it is not clear how to efficiently implement a direct use of Fact 3. Instead, we create the E_{2k} partition in $O(n)$ time, using a *reverse* approach to refinement. Rather than examining a class C of E_k to find how C should be refined, we use C *as a refiner* to see how it forces other E_k classes to split, or to stay together, as follows. For each number $i > k$ in C, locate and mark number $i-k$. Then, for each E_k class A, any numbers in A marked by C identify a complete E_{2k} class. The correctness of this follows from Fact 3.

Give a complete proof of the correctness of the reverse refinement approach to creating the E_{2k} partition from the E_k partition.

Each class of E_k, for any k, holds the starting locations of a k-length substring of S. The algorithm in [**?**] constructs a suffix array for S using the reverse refinement approach, with the added detail that the classes of E_k are kept in the lexical order of the strings associated with the classes.

In more detail, to obtain the E_2 partition of $S =$ *mississippi\$*, process the classes of E_1 in order, from the lexically smallest to the lexically largest class. Processing the first class, $\{12\}$, results in the creation of the E_2 class $\{11\}$. The second E_1 class $\{2,5,8,11\}$ marks indices $\{1,4,7\}$ and $\{10\}$, and hence creates the three E_2 classes $\{1\}, \{4,7\}, \{10\}$. Class $\{9,10\}$ of E_1 creates the two classes $\{8\}, \{9\}$. Class $\{3,4,6,7\}$ of E_1 creates classes $\{2,5\}, \{3,6\}$ of E_2. Each class of E_2 holds the starting locations of identical substrings of length one or two. These classes, lexically ordered by the substrings they represent, are: $\{12\},\{11\},\{8\},\{2,5\},\{1\},\{10\},\{9\},\{4,7\},\{3,6\}$. The classes of E_4, in lexical order are: $\{12\},\{11\},\{8\},\{2,5\},\{1\},\{10\},\{9\},\{7\},\{4\},\{6\},\{3\}$. Note that $\{2,5\}$ remain in the same E_4 class because $\{4,7\}$ were in the same E_2 class. The E_2 classes of $\{4,7\}$ and $\{3,6\}$ are each refined in E_4. Explain why.

Although the general idea of reverse refinement should now be clear, efficient implementation requires a number of additional details. Give complete implementation details and analysis, proving that the E_{2k} classes can be obtained from the E_k classes in $O(n)$ time. Be sure to detail how the classes are kept in lexical order.

Assume n is a power of two. Note that the algorithm can stop as soon as every class is a singleton and this must happen within $\log_2 n$ iterations. When the algorithm ends, the order of the (singleton) classes describes a permutation of the integers 1 to n. Prove that this permutation is the suffix array for string S. Conclude that the reverse refinement method creates a suffix array in $O(n \log n)$ time.