CS176–Fall 2015 — Solutions to Homework 1

Manohar Jois

September 24, 2015

Problem 1

(a) Assume for string S we can build a suffix array A as well as an auxiliary structure L that stores the length of the longest common prefixes between suffixes represented by entries i and $i-1$ in A ($L[1]$ is undefined), both in linear time. Once we build these, we scan the consecutive pairs of L and group together entries that are at least k, outputting the index and frequency when we encounter an entry less than k. Pseudocode might look like the following:

```
build suffix array A and LCP array L
index = A[1], count = 1
for i in [2, |S|]:
    if L[i] >= k:
        count++
    else:
        if index + k - 1 <= |S|:  # dont go past end of string
            output (index, count)
        index = A[i], count = 1
```

All instances of a k-mer will be represented in a single cluster within the suffix array, and it is enough to check if a suffix's LCP with its lexicographic predecessor is at least k to check if it's the same k-mer. With linear time construction of A and L as well as a linear scan through L, it's clear that the runtime of this algorithm is $O(n)$.

(b) Simply concatenate the first $k-1$ characters of S to the end of S and run the same algorithm as in part (a). Clearly all possible k-mers represented by a circular jump from the end to the beginning of the string will be captured within this new string. The runtime will be $O(n+k-1) = O(n)$ because if $k > n$ the problem reduces to the case where $k = n$.

Problem 2

Build a generalized suffix tree that includes suffixes for all k strings. While building the tree, if a node v has a child whose edge to v contains only the terminal character, then the path to v represents a suffix of all strings indicated by that child node. Insert these string numbers into a list for that node $L(v)$. The construction of the tree and the L lists takes linear time.

Now traverse the tree with DFS while maintaining k stacks. For every node v on the way down, push v into every stack indicated by its list $L(v)$. On the way up past v, pop the top off these same stacks.

When you reach a leaf that represents an entire string j, the top of each stack will be k nodes u_i that represent the longest suffix-prefix match between strings i and j (except possibly j with j, but this case is trivial: the entire string j itself is the best match). By construction of the stacks, each node u_i is the deepest node that represents a complete suffix of string i along the path to the leaf of string j, so it must represent the longest suffix-prefix match.

There are at most n indices across all L-lists because there are at most n complete suffixes. This means there are at most n pushes and n pops across all k stacks. For all k strings we stop at the leaf representing it and output k substrings representing matches with the other strings. Linear construction and traversal along with $O(k^2)$ work to produce output means that the runtime of this algorithm is $O(n + k^2)$.

Problem 3

Start with the typical approach to exact string matching using the Z-algorithm as usual, where our text to process becomes $X = P\$T$. Compute the Z-scores for $2 \leq i \leq |P|+|T|+1$. Now use the Z-scores themselves corresponding to the text T as the string Y. If we search for the pattern S in text Y using the typical Z-algorithm approach, the indices returned should correspond to the indices of the desired super patterns in T.

The key point is to show that any substring in T with the Z-scores indicated in S is a super pattern over S and P. The Z-scores indicate the size of substrings starting at each index that match prefixes of P. The only catch is that the Z-scores could show that the matching portion of T extends past the region of the portion. To fix this, we can change the mechanic of character comparison in the second exact matching pass. Instead of comparing integer characters by character equality, we compare the integers (k_i, z_i) themselves. If $z_i \geq k_i$, then the characters are "equal." This ensures that the list S can match up with the Z-scores even when a match extends past the super pattern.

Since this is simply 2 instances of the linear-time, Z-score-based exact string matching algorithm, it is also linear time.

Problem 4

(a) n must be even. $Z_3(S) = n > 0$ and $Z_{n+2}(S) = 0$ imply that $S[1] = S[3] \neq S[n+2]$. The first inequality also implies that $S[n+2] = S[n] = S[n-2] = \ldots = S[3]$. If n is odd, this second implication contradicts the first because 3 is on the chain of every other number.

(b) $Z_i(S) = 0$ for even i, undefined for $i = 1$ (as always), 0 for $i = n+3$, and $n+3-i$ for any odd i in between. This is because if $n = 2$, then $S[1:2] = S[3:4]$ and $S[5] \neq S[3]$. Every time we increase n by 2, we must make $S[3 : n+2]$ identical to $S[1 : n]$, so our only option is to copy $S[n-1 : n]$ to $S[n+1 : n+2]$. The Z-scores for all odd indices i decrease by 2 every time because $S[i : n+2]$ exactly and non-extensibly matches smaller and smaller prefixes of S.

Problem 5

(a) S must be 8 characters in length, with positions 1 and 5 starting with the same character. Positions 2 and 6 must also start with the same character, different from 1 and 5. Same for positions 3, 4 and 7. With position 8 being the terminating character, the string S is, without loss of generality, ABCCABC\$. So $|LCP(1,5)|$, $|LCP(2,6)|$ and $|LCP(3,7)|$ are 3, 2 and 1, respectively.

(b) The suffix array A_S must be $[8,5,1,6,2,7,4,3]$. The terminating character by itself is always the smallest, and each suffix beginning with the same character will have its indices clustered together in the suffix array. Index 7 is less than index 3 because the eigth character is \$. This determines the relative ranks of indices 6 and 2, and in turn of indices 5 and 1. Since 5 must be before 1, it must be $A_S[2]$. 6 and 2 must follow because 7 has to be near 3. The relative order of 4 and 7 can be determined by the order of 5 and 8.