University of Waterloo
CS240, Fall 2018
Assignment 5

Due Date: Wednesday, November 28, at 5:00pm

Please read `http://www.student.cs.uwaterloo.ca/~cs240/f18/guidelines.pdf` for guidelines on submission. All problems are written problems; submit your solutions electronically as a PDF with file name `a05wp.pdf` using MarkUs. We will also accept individual question files named `a05q1w.pdf`, `a05q2w.pdf`, `a05q3.pdf`, and `a05q4w.pdf` if you wish to submit questions as you complete them.

There are 78 possible marks available. The assignment will be marked out of 75.

Problem 1 KMP [6+6+6=18 marks]

a) Compute the failure array for the pattern $P =$`ababac`.

b) Show how to search for pattern $P =$`ababac` in the text $T =$`abcaabaabababacabcaa` using the KMP algorithm. Indicate in a table such as Table 1 which characters of P were compared with which characters of T. Follow the example on Slide 11 of Module 9. Place each character of P in the column of the compared-to character of T. Put round brackets around characters if an actual comparison was not performed. You may need to add extra rows to the table.

c) Consider a pattern P and a text T. Assume that you are given the failure array for the string $P\Phi T$: the concatenation of P, a character Φ that is not contained in P, and T. Explain how to use the failure array to find the first occurrence of P in T. Note that you only have access to P and the failure array for $P\Phi T$: the text T is not available.

a	b	c	a	a	b	a	a	b	a	b	a	b	a	c	a	b	c	a	a

Table 1: Table for KMP problem.

Problem 2 Boyer-Moore [3+7+5=15 marks]

a) Compute the last-occurrence function L for the pattern P = `ratatat`. Give your answer as a table as shown on Slide 18 of Module 9. Note: $\Sigma = \{a, r, t\}$.

b) Compute the suffix skip array S for the pattern P = `ratatat`. Give your answer as a table as shown on Slide 19 of Module 9.

c) Show how to search for pattern P = `ratatat` in the text T = `ratarrtaaratatatat` using the Boyer-Moore algorithm. Indicate in a table such as Table 2 which characters of P were compared with which characters of T. Follow the example on Slide 17 of Module 9. Place each character of P in the column of the compared-to character of T. Put round brackets around characters which are known to be matched based on the suffix skip array (even if the algorithm matches them again.) Put square brackets around characters which are known to be matched based on the last occurrence function (even if the algorithm matches them again.) Note that some lines in the table might require both square and round brackets.

r	a	t	a	r	r	t	a	a	r	a	t	a	t	a	t	a	t

Table 2: Table for Boyer-Moore problem.

Problem 3 Huffman Coding [7+2+7+4 = 20 marks]

a) Construct the Huffman code for the following text T = `bananablues` over alphabet $\Sigma = \{\texttt{a}, \texttt{b}, \texttt{e}, \texttt{l}, \texttt{n}, \texttt{s}, \texttt{u}\}$.

To make the answer to this part question unique, strictly follow these conventions for constructing the code: To break ties, first use the smallest letter (according to the alphabetical order), or the tree containing the smallest alphabetical letter. When combining two trees of different values, place the lower-valued tree on the left (corresponding to a `0`-bit). When combining trees of equal value, place the one containing the smallest letter to the left.

The final code (a table with the codewords for all letters) is sufficient for full marks. You are highly encouraged, though, to provide intermediate steps to get partial marks in the presence of mistakes.

b) Use your Huffman code to decode the following bitstring:

`1000100010010110011110110111111110010`

It is sufficient to list the final decoded string.

(If you get stuck or obtain an unreadable sequence of letters you probably made a mistake in 3(a). Make sure to fix it before moving on.)

c) Huffman codes are in general not unique, but we can make them so using tie breaking rules (as ours above). To allow decoding, we have to store the code alongside the compressed file. A simple way would be to store all codewords explicitly, or to store the characters frequencies, but it actually suffices to store the *lengths* of the codewords only (which requires much fewer bits, namely at most $|\Sigma|\lceil\log(|\Sigma|)\rceil$ bits) using so-called *canonical Huffman codes*. These schemes uniquely determine the codewords given the sorted list of codeword lengths. A concise implementation of a canonical Huffman code scheme is given below:

```
canonicalHuffmanCode(symbol, codelen, s):
    // Assumes symbol[i], i=0,...,s-1 has codeword length codelen[i].
    // Assumes codelen is sorted ascendingly.
    int code = 0
    for i = 0,...,s-1
        print (symbol[i], binaryString(code, codelen[i])
        // binaryString returns code in binary with codelen[i] digits/bits
        if (i == s-1) break
        code = (code + 1) << ( codelen[i+1] - codelen[i] )
        // << is the left-shift operation
```

For example, the codeword lengths $\{1, 3, 3, 3, 3\}$ result in the codewords `0`, `100`, `101`, `110` and `111` being printed.

Compute the canonical Huffman code for the text $T =$ `bananablues` based on the codeword lengths you obtained in 3(a). When sorting the characters by codeword length, break ties by the alphabetical order of the characters. Also draw the corresponding code trie for the canonical code.

d) Prove that Huffman's algorithm for constructing code tries generally cannot be used to directly construct canonical Huffman codes.

To do so, show that any tree corresponding to the codewords of the canonical Huffman code from part 3(c) *cannot* be the result of an execution of Huffman's tree construction algorithm. Your argument should not depend on any tie-breaking conventions.

(We therefore have to use a two-stage approach: First, we compute codeword lengths using Huffman's algorithm. Second, we compute the actual codewords to use with `canonicalHuffmanCode`).

Problem 4 Suffix Trees [10+3+2+10 = 25 marks]

a) Construct the suffix tree for T = `abracadabra`. Use the end of word character \$. Children of a node should be ordered alphabetically, that is, the labels on the edges are ordered alphabetically. Note that $\$ < a$.

b) Define a *repeat* in a string to be any pair of positions $i \neq j$, so that there is an $\ell \geq 1$ with $T_{i,i+\ell-1} = T_{j,j+\ell-1}$ i. e. two positions at which the same substring of length ℓ is found. We call ℓ the length of the repeat at (i, j) and $R = T_{i,i+\ell-1}$ the repeated pattern.

Find a repeated pattern R of maximal length in the example text T = `abracadabra` and list its length and positions.

c) Traverse your suffix tree from 4(a) along your maximal repeated pattern R from 4(b) Where does the search stop; at a leaf, at an internal node or inside an edge? Mark the reached position in your suffix tree in 4(a).

d) Let T be a binary string of length n (a string composed of the characters 0 and 1.) Describe an algorithm that finds a repeated pattern R in T of maximal length. (You algorithm does not have to output the positions (i, j).)

For full credit, your algorithm should have run time $O(n)$, but slower algorithms will receive partial credit.