COMP9024 18s2 | Data Structures and Algorithms

Week 11 Problem Set

String Algorithms

1. (String processing)

   Implement a C-function

   ```
   int *lastOccurrence(char *pattern, char *alphabet) { ... }
   ```

   that computes the last-occurrence function for the Boyer-Moore algorithm. The function should return a newly created dynamic array indexed by the numeric codes of the characters in the given alphabet (a non-empty string of ASCII-characters).

   Ensure that your function runs in *O(m+s)* time, where *m* is the size of the pattern and *s* the size of the alphabet.

   Hint: You can obtain the numeric code of a `char c` through type conversion: `(int)c`.

   Answer:

   ```
   #define ASCII_SIZE 128

   int *lastOccurrenceFunction(char *pattern, char *alphabet) {
      int *L = malloc(ASCII_SIZE * sizeof(int));
      assert(L != NULL);

      int i, s = strlen(alphabet);
      for (i = 0; i < s; i++)                // for all chars in the alphabet
         L[(int)alphabet[i]] = -1;           // ... initialise L[] to -1

      int m = strlen(pattern);
      for (i = 0; i < m; i++)
         L[(int)pattern[i]] = i;             // set L[]

      return L;
   }
   ```

2. (Boyer-Moore algorithm)

   Use your answer to Exercise 1 to write a C-program that:

   - prompts the user to input
     - an alphabet (a string),
     - a text (a string),
     - a pattern (a string);
   - computes and outputs the last-occurrence function for the pattern and alphabet;
   - uses the Boyer-Moore algorithm to match the pattern against the text.

   An example of the program executing could be

   ```
   prompt$ ./boyer-moore
   Enter alphabet: abcd
   Enter text: abacaabadcabacabaabb
   Enter pattern: abacab

   L[a] = 4
   L[b] = 5
   L[c] = 3
   L[d] = -1

   Match found at position 10.
   ```

   If no match is found the output should be: `No match.`

   Hints:

   - You may assume that
     - the pattern and the alphabet have no more than 127 characters;

    - the text has no more than 1023 characters.
- To scan `stdin` for a string with whitespace, such as `"a pattern matching algorithm"`, you can use:

```
#define MAX_TEXT_LENGTH 1024
#define TEXT_FORMAT_STRING "%[^\n]%*c"

char T[MAX_TEXT_LENGTH];
scanf(TEXT_FORMAT_STRING, T);
```

  This will read every character as long as it is not a newline `'\n'`, and `"%*c"` ensures that the newline is read but discarded.

We have created a script that can automatically test your program. To run this test you can execute the `dryrun` *program that corresponds to the problem set and week. It expects to find a program named* `boyer-moore.c` *in the current directory. You can use dryrun as follows:*

```
prompt$ ~cs9024/bin/dryrun prob11
```

Answer:

```
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>
#include <string.h>

#define MAX_TEXT_LENGTH 1024
#define MAX_PATTERN_LENGTH 128
#define MAX_ALPHABET_LENGTH 128
#define TEXT_FORMAT_STRING "%[^\n]%*c"
#define PATTERN_FORMAT_STRING "%[^\n]%*c"
#define ALPHABET_FORMAT_STRING "%[^\n]%*c"

#define MIN(x,y) ((x < y) ? x : y)    // ternary operator (cond ? t1 : t2)
                                      // => evaluates to t1 if (cond)≠0, else to t2

int boyerMoore(char *text, char *pattern, int *L) {
   int n = strlen(text);
   int m = strlen(pattern);

   int i = m-1;
   int j = m-1;
   do {
      if (text[i] == pattern[j]) {
         if (j == 0) {
            return i;
         } else {
            i--;
            j--;
         }
      } else {                           // character jump
         i = i + m - MIN(j, 1+L[(int)text[i]]);
         j = m - 1;
      }
   } while (i < n);
   return -1;                            // no match
}

int main(void) {
   char T[MAX_TEXT_LENGTH];
   char P[MAX_PATTERN_LENGTH];
   char S[MAX_ALPHABET_LENGTH];

   printf("Enter alphabet: ");
   scanf(ALPHABET_FORMAT_STRING, S);
   printf("Enter text: ");
   scanf(TEXT_FORMAT_STRING, T);
   printf("Enter pattern: ");
   scanf(PATTERN_FORMAT_STRING, P);
   putchar('\n');
```

```
   int *L = lastOccurrenceFunction(P, S);
   int i, s = strlen(S);
   for (i = 0; i < s; i++)
      printf("L[%c] = %d\n", S[i], L[(int)S[i]]);

   int match = boyerMoore(T, P, L);
   free(L);
   if (match > -1)
      printf("\nMatch found at position %d.\n", match);
   else
      printf("\nNo match.\n");

   return 0;
}
```

3. (Knuth-Morris-Pratt algorithm)

Develop, in pseudocode, a modified KMP algorithm that finds *every* occurrence of a pattern *P* in a text *T*. The algorithm should return a queue with the starting index of every substring of *T* equal to *P*.

Note that your algorithm should still run in *O(n+m)* time, and it should find every match, including those that "overlap".

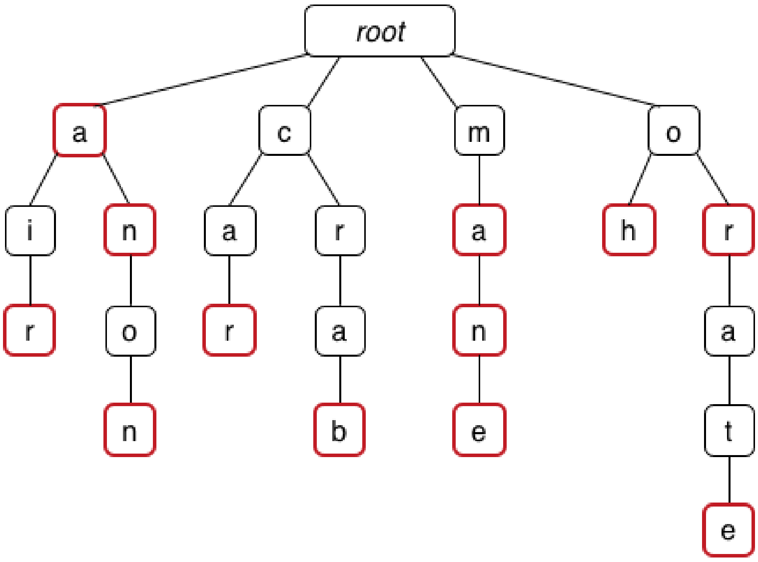
Answer:

```
KMPMatchAll(T,P):
   Input  text T of length n, pattern P of length m
   Output queue with all starting indices of substrings of T equal to P

   F=failureFunction(P)
   i=0, j=0
   Q=empty queue
   while i<n do
      if T[i]=P[j] then
         if j=m-1 then
            enqueue i-j into Q     // match found
            i=i+1, j=F[m-1]        // continue to search for next match
         else
            i=i+1, j=j+1
         end if
      else
         if j>0 then
            j=F[j-1]
         else
            i=i+1
         end if
      end if
   end while
   return Q                        // if Q is empty, no match found
```

4. (Tries)

a. Consider the following trie, where finishing nodes are shown in red:



What words are encoded in this trie?

b. If the following keys were inserted into an initially empty trie:

```
jaws boots axe boo so jaw sore boot boon
```

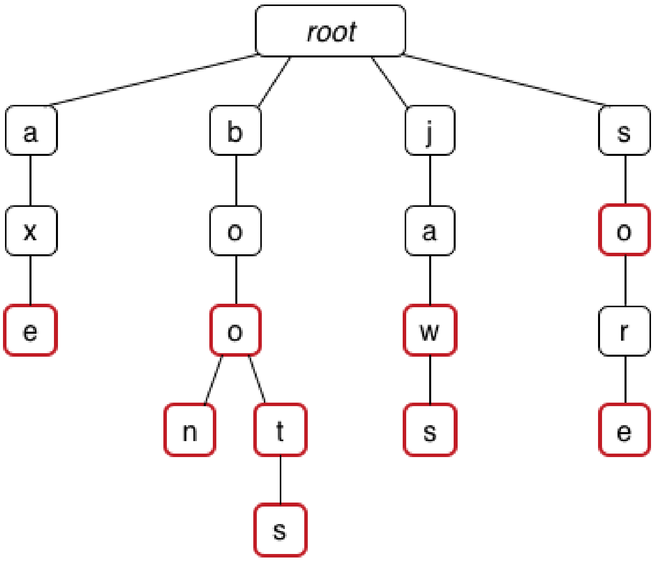
what would the final trie look like? Does the order of insertion matter?

c. Answer question b. for a compressed trie.

Answer:

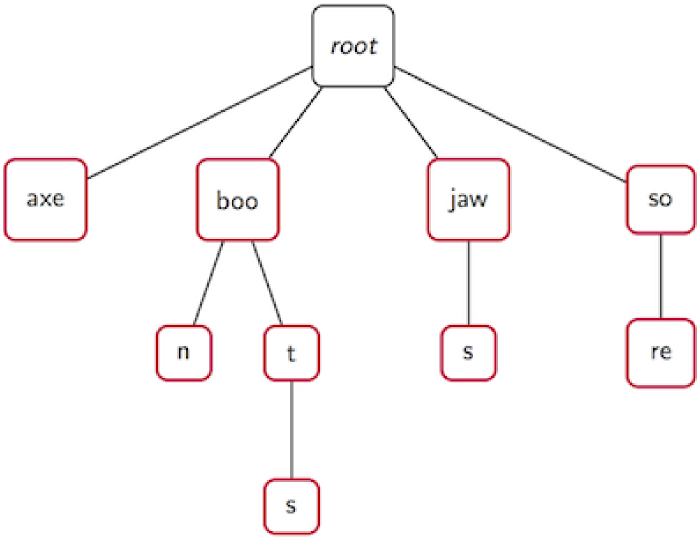
a. In alphabetical order: a, air, an, anon, car, crab, ma, man, mane, oh, or, orate.

b. The trie after all keys are inserted:



The order of insertion does not matter. The same trie will always result from insertion of the same set of words.

c. The compressed trie after all keys are inserted:



Again, the order of insertion does not matter.

5. (Text compression)

   Compute the frequency array and draw a Huffman tree for the following string:

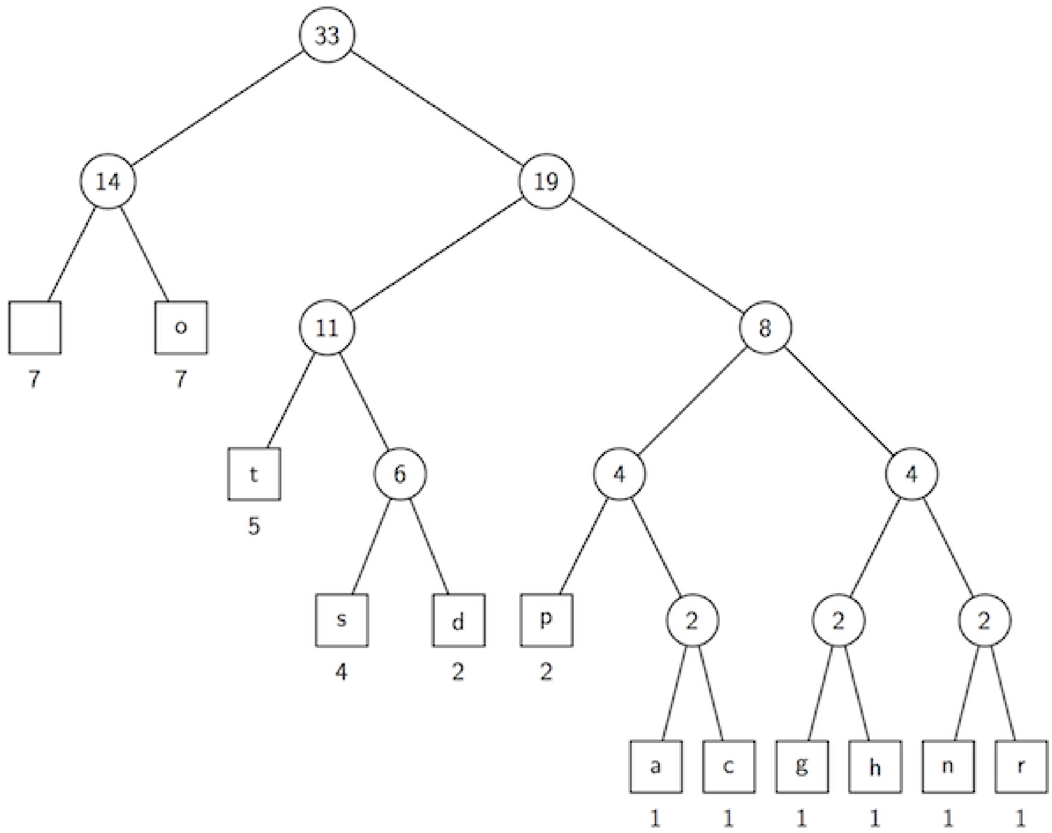
```
dogs do not spot hot pots or cats
```

Answer:

The frequency array:

Character		a	c	d	g	h	n	o	p	r	s	t
Frequency	7	1	1	2	1	1	1	7	2	1	4	5

A Huffman tree:



6. **Challenge Exercise**

   Given a string *s* with repeated characters, design an efficient algorithm for rearranging the characters in *s* so that no two adjacent characters are identical, or determine that no such permutation exists. Analyse the time complexity of your algorithm.

   Answer:

   The problem can be solved with a greedy approach: In each step, we select a character with the highest frequency that is different from the character selected before. Every time a character is selected, its frequency is reduced by one.

```
rearrangeString(S):
   Input  string S
   Output permutation of S such that no two adjacent chars are the same
          false if no such permutation exists

   compute frequency of each char in S
   P=priority queue of distinct chars in S with frequency as key
   S_new=empty string
   c=leave(P), append c to S_new, c.key=c.key-1
   while P is not empty do
      d=leave(P), append d to S_new, d.key=d.key-1
      if c.key>0 then
         join(P,c)          // insert c back into the priority queue
      end if
      c=d
   end while
   if c.key>0 then
      return false
   else
      return S_new
   end if
```

   Time complexity analysis:

   Let *n* be the size of the input string *s*.
   1. Computing the frequencies of all characters in *s* takes *O(n)* time.

2. Creating a priority queue for all distinct characters using a self-balancing BST takes *O(n·log n)* time.
3. Using a self-balancing BST, both `leave()` and `join()` take *O(log n)* time. Hence, the while-loop takes *O(n·log n)* time.

Therefore, the complexity of the algorithm is *O(n·log n)*.