Problem Set 6 Solutions

Problem 1 Show how to modify the KPM pattern matching algorithm so as to find every occurrence of a pattern string P that appears as a substring in T, while still running in O(n+m) time. (Be sure to catch even those matches that overlap.)

Solution: Instead of returning when a match is found, store the index and set i = i+1 and j = f(m-1).
The modified KMP algorithm is shown as follows.

Algorithm ***KMPMatch(T, P)***

```
{   F = failureFunction(P);
    i = 0;
    j = 0;
    Construct an empty stack S; // S stores the locations of all the occurrences of P in T
    while ( i < n )
      if ( T[i] = P[j] )
        { if ( j = m–1 )
            {
              S.push(i – j) ; //An occurrence and i-j is the location of this occurrence
              i=i+1;
              j=f(m-1);
            }
          else
            { i = i + 1;
              j = j + 1;}
            }
       else
         if ( j > 0 )
           j = F[j – 1];
         else
           i = i + 1;
   return S;  //  if S is empty, no match is found
}
```

Since constructing an empty stack and the push operation take O(1) time, the time complexity of this algorithm is still O(n+m).

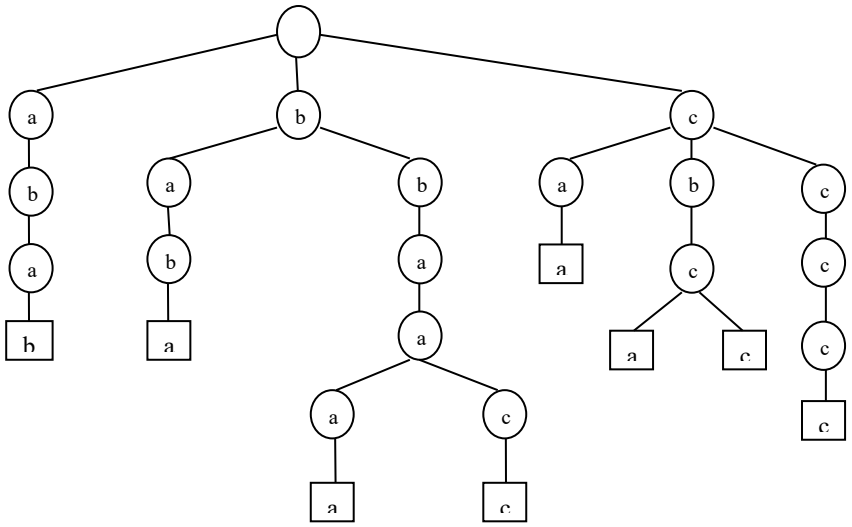
Problem 2 A pattern P of length m is said to be a circular substring of a text T of length n if P is equal to the concatenation of a suffix of T and a prefix of T, where neither the suffix and nor the prefix is an empty string. For example, if T=aacabca, all the circular substrings of length 3 of T are caa and aaa. Give an O(n+m)-time algorithm for determining whether P is a circular substring of T.

Solution: Generate a new text T′ = T[n−m+1. . . n-1]+T[0 . . .m-2]. Run KMP algorithm on T′ and P.

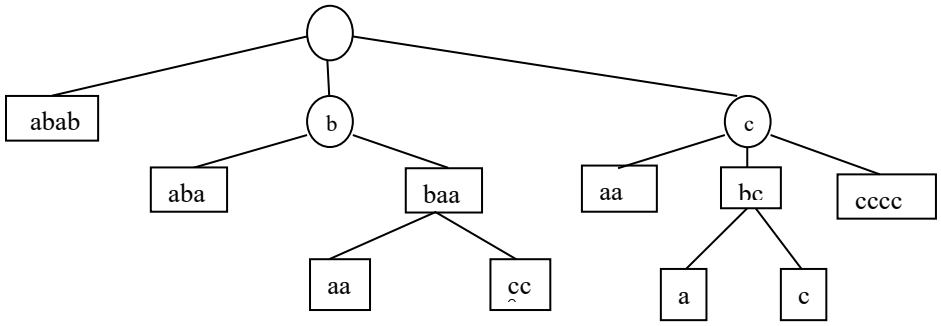
Problem 3 Draw a standard trie and a compressed trie for the following set of strings:

{abab, baba,ccccc, bbaaaa, caa, bbaacc, cbcc, cbca}.

Solution: The standard trie:



The compressed trie:



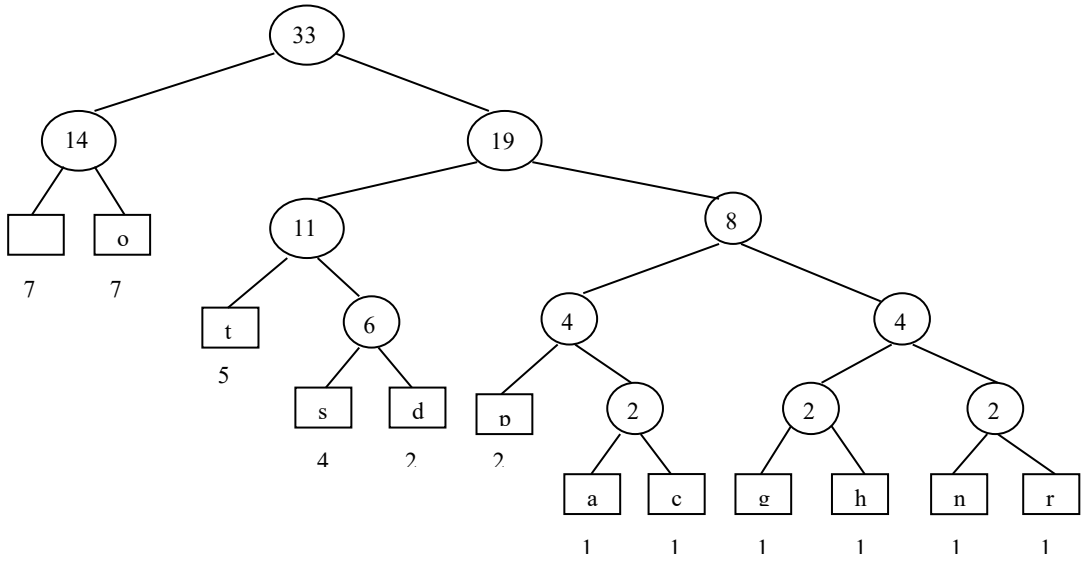
Problem 4 Draw the frequency array and Huffman tree for the following string:

"dogs do not spot hot pots or cats".

Solution: 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

The Huffman tree:



Problem 5 Give an efficient algorithm for deleting a string from a compressed trie and analyse its running time.

Solution:

```
Algorithm stringDeletion(T, s)
  Input: A compressed trie T and a string s
  Output: T without s
  {
   search the compressed trie for s; // work out the search procedure yourself
    if ( s was not found )
       {  print "s is not found";
          return 0;  // unsuccessful deletion
        }
    else
       {  let u be the node where s was just found;
          if ( s is not equal to the whole string ended at u or u has a child )
             {  print " s is a partial string";
                return 0;   // unsuccessful deletion
             }
          v=the parent of u;
          delete u;
          if ( v has a child c )
             { // p.string denotes the string stored at a node p
               v.string=v.string+c.string;
               delete node c;  // merge v and c into a single node
             }
          return 1;   // successful deletion
       }
```

```
}
```

Running time analysis: O(dm), where d is the size of the alphabet and m the depth of the external node that represents the string.

Problem 6 Give a sequence $S=(x_0, x_1, x_2, \ldots, x_{n-1})$ of numbers, describe an $O(n^2)$-time algorithm for finding a longest subsequence $T=(x_{i0}, x_{i1}, x_{i2}, \ldots, x_{ik-1})$ of the numbers, such that $i_j<i_{j+1}$ and $x_{ij}>x_{ij+1}$. That is, T is a longest decreasing subsequence of S.

Solution:

```
Algorithm subsequence(T, P)
  Input: A text T and a pattern P
  Output: The longest decreasing subsequence L of S
  {
   create an identical copy S1 of S;
   sort S1 in non-ascending order;
   remove all duplicates in S1 so that all numbers are distinct;
   find the longest common subsequence L of S and S1;
   return L;
  }
```

Time complexity analysis:

1. It takes O(n) time to create S1.
2. Sorting S1 using a heap-based priority queue takes O(n log n) time.
3. Removing all duplicates in S1 takes $O(n^2)$ time.
4. Finding the longest common subsequence of S and S1 takes $O(n^2)$ time.

Therefore, this algorithm takes $O(n)+O(n \log n)+O(n^2)+O(n^2)=O(n^2)$ time.

Problem 7 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.

Solution: We can use a greedy approach to solve this problem. The idea is to put the character with the highest frequency first. We use a priority queue based a max heap to store all characters with frequencies as keys where the highest frequency character is stored at the root. We repeatedly do the following until the priority queue becomes empty:

1. Remove the highest frequency character from the heap and append it to the result.
2. Decrease frequency of the character and temporarily move this character out of priority queue so that it is not picked next time.

```
Algorithm rearrangeString(s):
  Input:  a string s
  Output: a permutation of s such that no two adjacent chars are the same or 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=removeMax(P); // remove the char with the max frequency
append c to s_new;
c.key=c.key-1;
while P is not empty
    {  d=removeMax(P);
       append d to s_new;
       d.key=d.key-1
       if  ( c.key>0 )
           insert(P, c);        // insert c back into the priority queue
       c=d;
    }
if ( c.key>0 )
  return false;
else
  return s_new;
```

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 heap-based priority queue takes O(n) time.
3. Both removeMax() and insert() take O(log n) time. Hence, the while-loop takes O(n·log n) time.

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