(Sol) HW09 String Matching

1. [6 points] (Credit: Exercise 32.4-5 Cyclic Rotations in CLRS book.) Give a linear-time algorithm to determine if a text T is a cyclic rotation of another string T'. For example, `arc` and `car` are cyclic rotations of each other. For simplicity, you may assume the time complexity of KMP algorithm for n-letter string is $O(n)$.

    a. [4 points] Give pseudocode of your algorithm.
    Answer: Given two patterns P_1 and P_2, P_2 is a cyclic rotation of P_1 if and only if we can find the pattern P_1 within the string P_2P_2, which is the concatenation of P_2 and P_2, and length(P_1) = length(P_2). So, to determine if P_1 and P_2 are cyclic rotations of each other in linear time, use KMP to see if the pattern P_1 appears within the text P_2P_2, and if length(P_1) = length(P_2).
    (Note for grading: the answer doesn't have to include the code for KMP for full credit. The code below is only for a reference.)

```
public class CyclicRotation {

    public static boolean isCyclicRotation(String P1, String P2) {
        if (P1.length() != P2.length()) {
            return false;
        }
        String P2P2 = " "+P2+P2;
        P1 = " "+P1;
        return KMPbool(P2P2.toCharArray(), P1.toCharArray());
    }

    public static boolean KMPbool(char[] text, char[] pattern) {
        int m = pattern.length-1;
        int n = text.length-1;
        int[] pi = new int[m+1];
        computePi(pattern, pi);
        int q = 0;
        for (int i=1; i<=n; i++) {
            while (q > 0 && pattern[q+1]!=text[i]) {
                q=pi[q];
            }
            if (pattern[q+1]==text[i]) {
                q=q+1;
            }
            if (q==m) {
                return true;
            }
```

```
        }
        return false;
    }

    public static void computePi(char[] P, int[] pi) {
        int m = P.length-1;
        pi[1] = 0;
        int k = 0;
        for (int q=2; q<=m; q++) {
            while (k>0 && P[k+1]!=P[q]) {
                k = pi[k];
            }
            if (P[k+1]==P[q]) {
                k=k+1;
            }
            pi[q] = k;
        }
    }
}
```

b. [2 points] Analyze the running time of your algorithm. The time complexity of your algorithm must be at most $O(n)$ for full credit, where n is the length of T.
Answer: The time complexity of KMP is $O(n)$ when the length of text T is n. Comparing the length of P_1 and P_2 is $O(1)$, and the length of text P_2P_2 is $2n$, so using KMP for P_2P_2 is $O(2n) = O(n)$.

2. [6 points] (Credit: This is slightly modified from Problem 32.1 in CLRS book.) We still allow the pattern P to contain occurrences of a gap character $\diamond$ that can match an arbitrary string of characters (even one of zero length). Give a polynomial-time algorithm to determine if such a pattern P of length m occurs in a given text T of length n. Note that you do not need to find all occurrences, you just need to determine if the pattern occurs at all. The time complexity of your algorithm must be at most $O(mn)$ for full credit.

   a. [4 points] Give pseudocode of your algorithm.
   Answer: This problem is made easier by the fact that we do not need to find all occurrences of the pattern, just determine if the pattern exists at all. The pattern $ab \diamond ba \diamond c$ occurs in a string if the pattern ab occurs, followed by the pattern ba, followed by the pattern c.
   So, all we need to do is find the first occurrence of ab, and then find the first occurrence of ba after that, and then the first occurrence of c after that. Our algorithm is thus as follows:

- Split our pattern into subpatterns divided by $\diamond$ characters. So the pattern $ba \diamond ccb \diamond aab$ would be split into the subpatterns ba, ccb and aab.
- Find the first occurrence of the first subpattern ba(If none exists, fail.)
- Find the first occurrence of the second subpattern ccb, after the first subpattern (if none exists, fail)
- Find the first occurrence of the third subpattern aab, after the second subpattern (if none exists, fail)
- .. and so on

```
m = length of P
n = length of T
split pattern P into a list of subpatterns PS.
//e.g. PS[1] is the first subpattern of P
i=1, j=1, k=1
while (i<=n) {
  while (i<=n and j<=length of PS[k] and T[i] == PS[k][j]) {
    i++;
    j++;
  }
  if (j > length of PS[k] && k <= the number of subpatterns in PS) {
    k++;
    j=1;
  } else {
    return false;
  }
  if (k > the number of subpatterns in PS) {
    return true;
  }
}
return false;
```

b. [2 points Analyze the running time of your algorithm.
Answer: the outer while loop runs for n times at most, and the inner while loop runs for m times at most, so the overall time complexity is $O(mn)$.

3. [3 points] Analyze the running time of your solution for the GitHub assignment.