CS/ECE 374 FALL 2018 Homework 5 Problem 3	Zhe Zhang (zzhan157@illinois.edu) Ray Ying (xinruiy2@illinois.edu) Anqi Yao (anqiyao2@illinois.edu)

1. Recall that a *palindrome* is any string that is exactly the same as its reversal, like `I`, or `DEED`, or `RACECAR`, or `AMANAPLANACATACANALPANAMA`. For technical reasons, in this problem, we will only be interested in palindromes that are of length at least one, hence we will not treat the string ϵ as a palindrome.

   Any string can be decomposed into a sequence of palindrome substrings. For example, the string `BUBBASEESABANANA` ("Bubba sees a banana.") can be broken into palindromes in the following ways (among many others):

   `BUB` • `BASEESAB` • `ANANA`

   `B` • `U` • `BB` • `A` • `SEES` • `ABA` • `NAN` • `A`

   `B` • `U` • `BB` • `A` • `SEES` • `A` • `B` • `ANANA`

   `B` • `U` • `B` • `B` • `A` • `S` • `E` • `E` • `S` • `A` • `B` • `ANA` • `N` • `A`

   `B` • `U` • `B` • `B` • `A` • `S` • `E` • `E` • `S` • `A` • `B` • `A` • `N` • `A` • `N` • `A`

   Since any string $w \neq \epsilon$ can always be decomposed to palindromes we may want to find a decomposition that optimizes some objective. Here are two example objectives. The first objective is to decompose w into the smallest number of palindromes. A second objective is to find a decomposition such that each palindrome in the decomposition has length at least k where k is some given input parameter. Both of these can be cast as special cases of an abstract problem. Suppose we are given a function called $cost()$ that takes a positive integer h as input and outputs an integer (h). Given a decomposition of w into $u_1, u_2, \ldots, u_r$ (that is, $w = u_1 u_2 \ldots u_r$) we can define the cost of the decomposition as $\sum_{i=1}^{r} (|u_i|)$.

   For example if we define $(h) = 1$ for all $h \geq 1$ then finding a minimum cost palindromic decomposition of a given string w is the same as finding a decomposition of w with as few palindromes as possible. Suppose we define () as follows: $(h) = 1$ for $h < k$ and $(h) = 0$ for $h \geq k$. Then finding a minimum cost palindromic decomposition would enable us to decide whether there is a decomposition in which all palindromes are of length at least k; it is possible iff the minimum cost is 0.

   Describe an efficient algorithm that given black box access to a function (), and a string w, outputs the value of a minimum cost palindromic decomposition of w.

Solution:

1. In order to find the efficient algorithm for the minimum cost function, there will be two steps we need to perform. One is finding the different ways of decomposition and the second is finding which one decomposition will have the minimum cost when perform cost().

   - In the first step, we define a function $palindromList(w, idx, slist)$, which takes the string w which is given to us, an int value idx which show the current location and

an list that is a sequence of number, the $i-1$th index(index starts from o) in the list will be the length of $u_{(i)}$.
The basic idea is we traverse the whole string from the beginning. Having a pointer start from the first char in the string, keep going to the end of the string.
We check if the substring that from beginning to the char the pointer is pointing at is a palindrome. Once it finds a palindrome, it will first push the length of the substring into a newly created list and then run palindromList function on the remaining substring and newly created list. If the it reaches the end and still not able to find a palindrome, it just returns.

The function is provided below:

```
biglist = [] defining as a global variable
check(w,idx) it's a function that return a boolean whether w(0,idx) is
palindrome
w(a,b) is the notion for the string w from index a till index b
set the initial value of idx = o and slist = NULL

palindromList(w, idx, slist):

    while(idx < len(w)):

      - if(idx == len(w) - 1):
          * if(check(w,idx) == true):
              · tempList = slist
              · tempList.push(len(w(0,idx)))
              · biglist.push(tempList)
          * return
        if(check(w,idx) == true):
          * tempList = slist
          * tempList.push(len(w(0,idx)))
          * palindromList(w(idx+1), o, tempList)

        idx++

    return
```

- After first step, we have to compute the minimum cost by use the labels in each small list in the biglist, since cost function will only take the length parameter, the item we stored in the small list is each component palindrome's length. We simply take a small list at a time and find the small list that has least cost time and return it.

Combine two steps, the total time algorithm will be $O(n^3)$. Since we have to do palindromList function for every index from start to end, if each time the process takes $O(1)$, there is $(1+n)*n/2$ steps for the first u_1 choosing the starting string length 1, the string length is n, then the total time for palindromList is $n*(1+n)*n/2$, which is $O(n^3)$. The selecting smallest will be the number of list generated, which is at most $O(n^3)$ since we only did that much action. So the total algorithm time will be $O(n^3)$.

■