CSE101 HW7

Yunfan Kang A14282934

1.(a)

(b)

procedure findMaxSum**(**int a**[])**

global int max **=** 0**;**

global int maxL **=** 0**;**

global int maxR **=** 0**;**

global int currentMaxL **=** 0**;**

findMaxSum**(**a**,** a**.**length**-**1**);**

**return** max**;**

**}**

procedure findMaxSum**(**int a**[],** int pos**){**

**if** **(**pos **==** **-**1**){**

**return** 0**;**

**}else{**

int currentMax**;**

int pre **=** findMaxSum**(**a**,** pos**-**1**)+**a**[**pos**];**

**if** **(**pre **>** a**[**pos**]){**

currentMax **=** pre**;**

**}else{**

currentMax **=** a**[**pos**];**

currentMaxL **=** pos**;**

**}**

**if** **(**max **<** currentMax**){**

max **=** currentMax**;**

maxR **=** pos**;**

maxL **=** currentMaxL**;**

**}**

**return** currentMax**;**

**}**

**}**

Idea and time complexity:

findMaxSum(int a[]) tries to return the contiguous subsequence of maximum sum of array a. Global variable max will keep track of the maximum sum found so far and maxL and maxR will store the index of the two end of the contiguous subsequence of maximum sum found so far. Global variable currentMaxL keeps track of the contiguous subsequence of maximum sum among sequences that end at a[pos]. findMaxSum(int a[], int pos) will first find the contiguous subsequence of maximum sum ending at a[pos] and compare its sum with max and update max, maxL, maxR if needed.

Base case of findMaxSum(int a[], int pos) is when pos = -1, which means the subarray we are going to consider is empty. In this case return 0. Otherwise, it will compare findMaxSum**(**a**,** pos**-**1**)+**a**[**pos**]** and a**[**pos**]** to decide the contiguous sequence of maximum sum ending at a[pos] and then compare the sum with max to decide whether to update the global variables. Hence T(n) = T(n – 1) + O(1), T(n) = O(n)

2.(a) Each element in the sequence can be either in or not in a subsequence, hence there are combinations. Hence there are subsequences.

(b)

procedure palindromicSubsequence**(**sequence a**[]){**

int table**[**a**.**length**][**a**.**length**];**

**for** **(**int i **=** 0**;** i **<** a**.**length**;** i**++){**

table**[**i**][**i**]** **=** 1**;**

**}**

**for** **(**int i **=** 1**;** i **<** a**.**length**;** i**++){**

**for** **(**int j **=** 0**;** j **<** a**.**length **-** i**;**j**++){**

**if** **(**a**[**j**]** **==** a**[**i**+**j**]){**

table**[**j**][**i**+**j**]** **=** table**[**j**+**1**][**i**+**j**-**1**]** **+** 2**;**

**}else{**

table**[**j**][**i**+**j**]** **=** Math**.**max**(**table**[**j**+**1**][**i**+**j**],** table**[**j**][**i**+**j**-**1**]);**

**}**

**}**

**}**

**return** **(**table**[**0**][**a**.**length**-**1**]);**

**}**Idea and time complexity:

Every character in the sequence is a palindromic subsequence of length 1, hence set all diagonal entries to be 1. Then Compute the value of table[j][i+j] based on its left, lower, and left lower entry of the table until the right upper half of the table is full. If a[j] = a[i+j], then the length of longest palindromic subsequence of a[i] to a[i+j](table[j][i+j]) is equal to that of a[j+1] to a[i + j -1](stored in table[j+1][i+j-1]) + 2. Otherwise, set the value of table[j][i+j] to be the larger one of table[j+1][i+j]and table[j][i+j-1].