EECE 230X – Introduction to Computation and Programming
Programming Assignment 4

- This programming assignment consists of 3 problems.
- Prerequisites: Topic 5
- Related material: Lists, tuples, strings, and function

Problem 1. Check if a string is a palindrome

A string is called a *palindrome* if it reads the same both forward and backward. For instance, each of the following strings is a palindrome "abba","abcba","a", "tx1aa1xt", "civic", "anna". The empty string is also a palindrome. On the other hand, none of the following strings is a palindrome: "cbba","accba","abad".

Write a program which asks the user to enter a string and checks whether or not it is palindrome.

Read the user input into a string s. Then use the indexing operator to compare the entries of s (see the slide entitled "Strings: indexing operator []" in the "Lists, Tuples, and Strings" presentation on moodle): first compare $s[0]$ and $s[n-1]$, where n is the length of s, then $s[1]$ and $s[n-2]$, and so on.

You are asked to solve this problem without using an auxiliary string and without slicing $[::-1]$ (see Problem 3.b in Programming Assignment 3). Using the slicing operator, it has a one line solution: `s == s[::-1]`.

Sample Input/Output:

```
Enter string: anna
YES palindrome
---------------
Enter string: civic
YES palindrome
---------------
Enter string: a
YES palindrome
---------------
Enter string: tx1aa1xt
YES palindrome
---------------
Enter string:
YES palindrome
---------------
Enter string: Civic
NOT a  palindrome
---------------
Enter string: ab
NOT a  palindrome
```

Problem 2. Selection Sort

In this problem you will implement a function to sort a list of numbers in non-decreasing order using the selection sort algorithm. The idea of Selection of Sort is as follows.

To sort a length-n list $L = [L[0], \ldots, L[n-1]]$:

1. Find the index *minIndex* of smallest element in L (find the index of first occurrence if you have more than one element equal to the min).
2. Exchange $L[minIndex]$ (swap it) with $L[0]$, hence now $L[0]$ is the (a) smallest element of L, i.e., the number stored in $L[0]$ is in its correct position in the desired sorted order
3. Find the index *minIndex* of the smallest element in $[L[1], \ldots, L[n-1]]$
4. Exchange $L[minIndex]$ with $L[1]$, hence now $L[1]$ is the (a) second smallest element of L, i.e., the number stored in $L[1]$ is in its correct position in the desired sorted order.
5. Find the index *minIndex* of the smallest element in $[L[2], \ldots, L[n-1]]$
6. Exchange $L[minIndex]$ with $L[2]$, hence now $L[2]$ is the (a) third smallest element of L, i.e., the number stored in $L[2]$ is in its correct position in the desired sorted order.
7. And so on until the whole list L is sorted

You will implement this algorithm in Part (c), but first you will try it on an example in part (a) and you will implement a function to do Steps $1, 3, 5, \ldots$ in Part (b).

a) **Try it on an example.** Illustrate the algorithm on `L=[5, 2, 4, 6, 1, 3]`. There is no need to submit this part.

b) **Function to find the index of min.** Implement a function `indexOfMin(L, start, end)`, which takes as input arguments a list `L` of integers or floats and two indices `start` and `end` such that $0 \leq start \leq end <$`len(L)`. It supposed to return the index of the smallest element in $[L[start], \ldots, L[end]]$. Return the index of first occurrence of a min if you have more than one elements equal to the min.

Test script:

```
L = [10,4,3,7,3,15]
print(indexOfMin(L,0,5))
print(indexOfMin(L,1,5))
print(indexOfMin(L,3,4))
print(indexOfMin(L,4,4))
```

Output:

```
2
2
4
4
```

c) **Selection sort function.** Using your function in Part (b), implement a function `selectionSort(L)`, which given a list `L` of integers floats, sorts `L` in non-decreasing order as explained in the selection sort algorithm. Note that unlike the function `indexOfMin`, the function `slectionSort` modifies the list `L`.

Notes:

- ⋆ Do not use the slicing operator in the implementing of `selectionSort` as it results in creating copies.
- ⋆ *Swapping*: Say that we want to swap, i.e., exchange, $L[7]$ and $L[2]$. If you set $L[7]$ to $L[2]$, and then set $L[2]$ to $L[7]$, you loose the old value of $L[7]$.

  To resolve this issue, use a temporary variable *temp*:

  1. set a *temp* to $L[7]$
  2. set $L[7]$ to $L[2]$, then
  3. set $L[2]$ to *temp*.

  Another approach to swap is use tuples: $(L[7], L[2]) = (L[2], L[7])$.

Test script:

```
L =  [10,4,3,7,3,15]
selectionSort(L)
print(L)
L =  [1.1, 31.31, 5.15]
selectionSort(L)
print(L)
L =  [10, 10]
selectionSort(L)
print(L)
L =  [1]
selectionSort(L)
print(L)
L =  []
selectionSort(L)
print(L)
```

Output:

```
[3, 3, 4, 7, 10, 15]
[1.1, 5.15, 31.31]
[10, 10]
[1]
[]
```

Problem 3. Longest palindromic substring

Recall the definition of a palindromic string from Problem 1: a string is called a palindrome or *palindromic* if it reads the same both forward and backward. In this problem, given a strings s, we would like to find a palindromic substring string of s whose length is maximal.

Examples:

string	longest symmetric substring underlined
”aceexcivicgrfdds”	”aceexcivicgrfdds”
”civicgrfdds”	”civicgrfdds”
”aceexcivic”	”aceexcivic”
“123abba1”	“123abba1”
”12345”	any one of the 5 characters

If s contains more than one palindromic substring of maximal length (e.g., last example), we are satisfied with any of them as answer.

a) **Using slicing.** Implement the function `longestPalSubsA(s)`, which given a string s, returns a palindromic substring of `s` of maximal length.

You are asked to do this part using slicing (Problem 3.b in Programming Assignment 3): to check if the substring `s[i:j+1]` (recall that $s[i : j + 1]$ is the substring “s[i]s[i+1]...s[j]”) is palindromic, let `t = s[i:j+1]` and check if $t == t[:: -1]$.

(*Hint:* Use variables `start` and `end` to keep track of the start and end of longest palindromic substring found so far. Loop over $i = 0, \dots, n$ and $j = i, \dots, n$ and, depnding on $s[i : j + 1]$, update `start` and `end` if needed. Appropriately initialize `start` and `end`.)

Test script:

```
print(longestPalSubsA("aceexcivicgrfdds"))
print(longestPalSubsA("civicgrfdds"))
print(longestPalSubsA("aceexcivic"))
print(longestPalSubsA("civic"))
print(longestPalSubsA("123abba1"))
print(longestPalSubsA("abba1"))
print(longestPalSubsA("123abba"))
print(longestPalSubsA("12345"))
print(longestPalSubsA(""))
```

Output:

```
civic
civic
civic
civic
abba
abba
abba
#1,2,3, 4, or 5 are all  valid outputs
#empty line
```

b) **Without slicing and using triply-nested loops.** Repeat Part (a) without using slicing except for in the return statement, i.e., slicing is only allowed in `return s[start:end+1]`. Call your function `longestPalSubsB`. The direct implementation requires three nested loops.

It helps first to implement the function `isSubsequencePalindrome(s,start,end)`, which takes as input parameters a string `s` and two indices `start` and `end` such t that 0 ≤`start`≤`end`<`len(s)`. The function `isSubsequencePalindrome(s,start,end)` returns `True` if the substring `s[start:end+1]` is a palindrome, and `False` otherwise. Implement this function by appropriately modifying the code of Problem 1.

Test your function on the strings in Part (a).

c) **(⋆) Without slicing and without three nested loops.** Repeat Part (b) without slicing and without triply nested loops. Call your function `longestPalSubsC`. This part is difficult.

As in the max-sum problem in the previous assignment, emulating three nested loops using two or one loop doesn't count as valid solution. The aim is to do it **more efficiently**: the number of steps should scale quadratically NOT cubicly with the string length.

Don't try to read the following hint unless you are stuck (to read it you need to magnify the PDF file); try first to solve it on your own without help.

(*Hint:* elddim eht morf dnapxe)