Data Structures & Algorithms – Lab #2

Aim: Getting Familiar with Algorithms with different rate of growth and Big-O Notation for Algorithm Analysis

Topics:

1. Classic Problem of Anagram Detection for Strings
2. Comparing Three Algorithms for Anagram Detection
3. Big-O Notation

Anagram Detection Problem for Strings

Solution 1 – Execute the following function, which is the first solution to Anagram Detection problem for strings. Specify its Big-O Notation.

```
def anagramDetection1(s1,s2):
    anagram = True
    if len(s1) != len(s2):
        anagram = False

    s2list = list(s2)
    s1index = 0

    while s1index < len(s1) and anagram:
        s2index = 0
        found = False
        while s2index < len(s2list) and not found:
            if s1[s1index] == s2list[s2index]:
                found = True
            else:
                s2index = s2index + 1

        if found:
            s2list[s2index] = None
        else:
            anagram = False

        s1index = s1index + 1

    return anagram

print(anagramDetection1('listen','silent'))
```

The time complexity of this algorithm is $O(n^2)$, as it uses nested loops. When time complexity of an algorithm is $O(n^2)$, it means running time of algorithm grows based on the square of input size, which is *n*.

Solution 2 – Execute the following function, which is the second solution to Anagram Detection problem for strings. Specify its order of magnitude using Big-O Notation.

```
def anagramDetection2(s1,s2):
    s1List = list(s1)
    s2List = list(s2)

    s1List.sort()
    s2List.sort()

    ind = 0
    matches = True

    while ind < len(s1) and matches:
        if s1List[ind] == s2List[ind]:
            ind = ind + 1
        else:
            matches = False

    return matches


print(anagramDetection2('listen','silent'))
```

The time complexity of this algorithm is O(n log n). The big-O notation of while loop is O(n) while the big-O notation of listing sorting a list using list() method in worst case is O(n log n). So the time complexity of the whole code is O(n log n).

Solution 3 – Execute the following function, which is the third solution to Anagram Detection problem for strings. Specify its order of magnitude using Big-O Notation.

```
def anagramDetection3(s1,s2):
    countS1 = [0]*26
    countS2 = [0]*26

    for i in range(len(s1)):
        ind = ord(s1[i])-ord('a')
        countS1[ind] = countS1[ind] + 1

    for i in range(len(s2)):
        ind = ord(s2[i])-ord('a')
        countS2[ind] = countS2[ind] + 1

    j = 0
    anagram = True
    while j<26 and anagram:
        if countS1[j]==countS2[j]:
            j = j + 1
        else:
            anagram = False

    return anagram

print(anagramDetection3('listen','silent'))
```

The time complexity of this algorithm is O(n). There is no nested loop in this code. We used three separate loops. The big-O notation of each loop is O(n), so the time complexity of the whole code is O(n).

The ord() function in Python is a built-in function that takes a single Unicode character as input and returns its corresponding integer Unicode code (number).

Solution 3 is the fastest algorithm for Anagram Detection problem for strings, while solution 1 is the slowest one.