Theory Assignment 1

Chenting Zhang

October 11, 2023

1 Problem 1

1.1 1.

Stack is last-in first-out data structure:

(a) Push(1) [1]

(b) Push(2) [2,1]

(c) Push(3) [3,2,1]

(d) Push(4) [4,3,2,1]

(e) Push(5) [5,4,3,2,1]

(f) Pop() [4,3,2,1]

(g) Pop() [3,2,1]

(h) Push(34) [34,3,2,1]

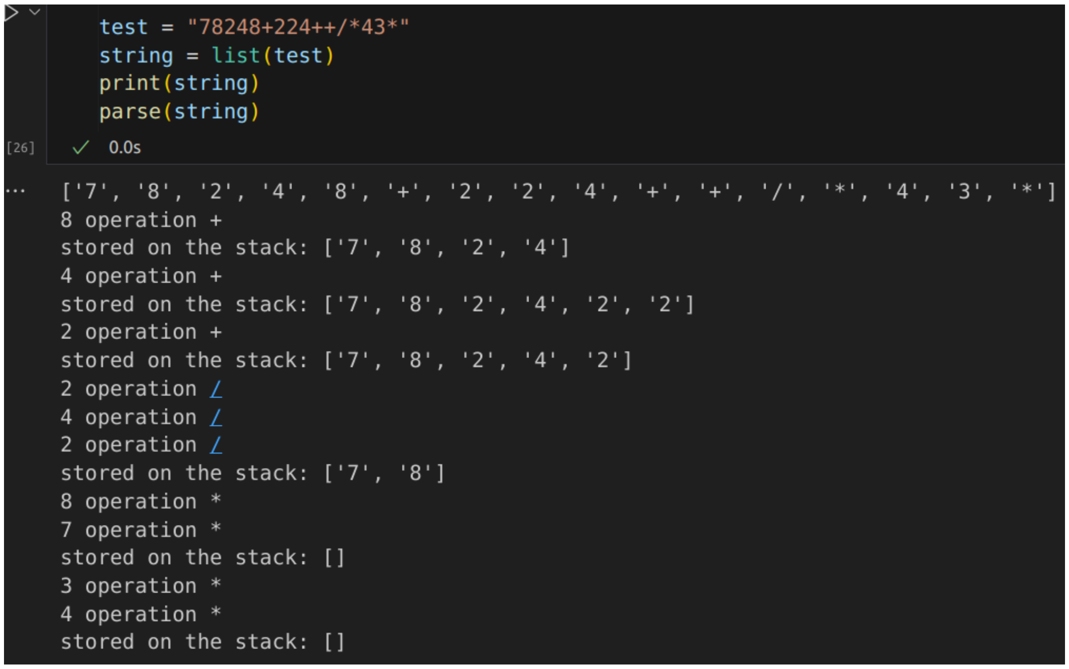
(i) Pop() [3,2,1]

1.2 2.

```python
def parse(input):
    string = list(input)
    stack = []
    for element in string:
        if element in ["0","1","2","3","4","5","6","7","8","9"]:
            stack.append(element)

        elif element == "+":
            pop_element = stack.pop()
            print(pop_element,'operation +')
            print("stored on the stack:", stack)
        elif element == "*":
            while True:
                pop_element = stack.pop()
                print(pop_element,'operation *')
                if not stack:
                    print("stored on the stack:", stack)
                    break
        elif element == "/":
            for _ in range(3):
                pop_element = stack.pop()
                print(pop_element, "operation /")
            print("stored on the stack:", stack)
        else:
            print("invalid input")
```

The test code and result is illustrated as follows:



```
test = "78248+224++/*43*"
string = list(test)
print(string)
parse(string)
```

```
[26]  ✓ 0.0s
```

```
['7', '8', '2', '4', '8', '+', '2', '2', '4', '+', '+', '/', '*', '4', '3', '*']
8 operation +
stored on the stack: ['7', '8', '2', '4']
4 operation +
stored on the stack: ['7', '8', '2', '4', '2', '2']
2 operation +
stored on the stack: ['7', '8', '2', '4', '2']
2 operation /
4 operation /
2 operation /
stored on the stack: ['7', '8']
8 operation *
7 operation *
stored on the stack: []
3 operation *
4 operation *
stored on the stack: []
```

The algorithm's output is $[8, 4, 2, 2, 4, 2, 8, 7, 3, 4]$. The stored stack after each operation(except push numbers) is

1. $[7, 8, 2, 4]$
2. $[7, 8, 2, 4, 2, 2]$
3. $[7, 8, 2, 4, 2]$
4. $[7, 8]$
5. $[]$
6. $[]$

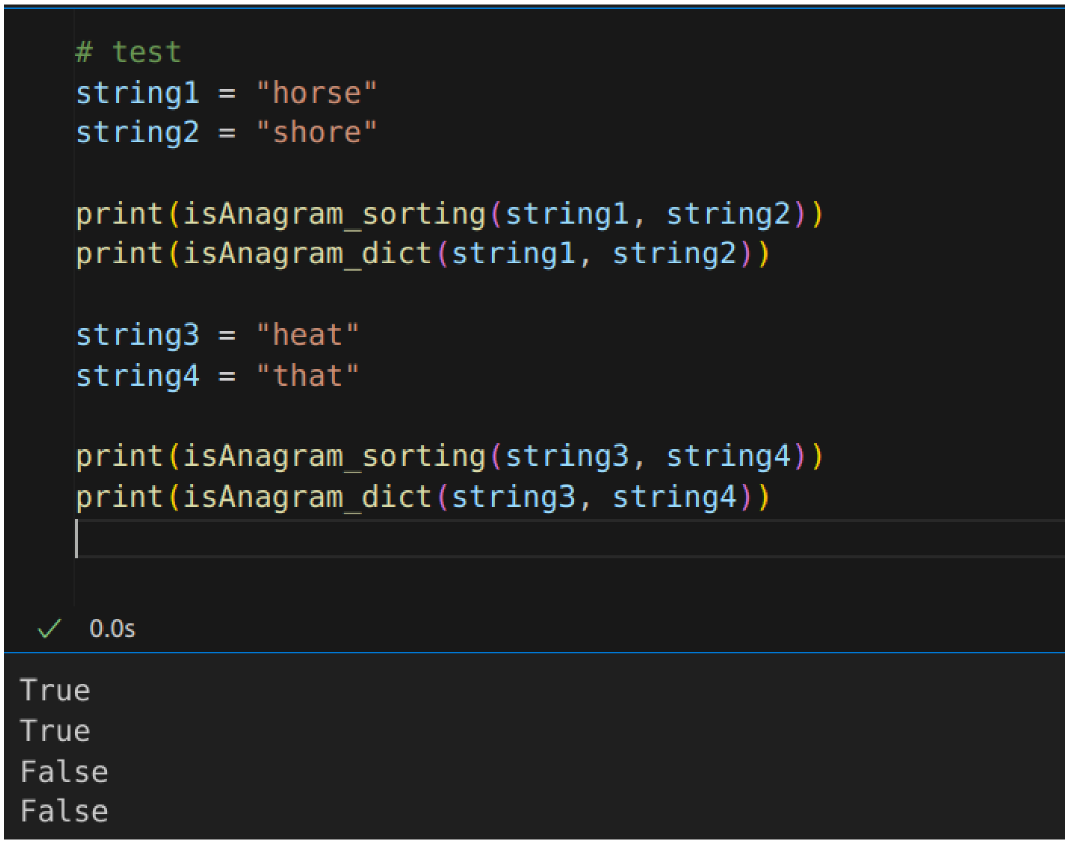
(python list operation)
Note: in python list representation of stack, the first index indicating the first one coming inside the stack, like stored in the bottom of the stack. The "append" operation is equivalent to "push".

2 Problem 2

```python
def isAnagram_sorting(s1, s2):
    if sorted(s1) == sorted(s2):
        return True
    else:
        return False

def isAnagram_dict(s1, s2):
    l1 = list(s1)
    l2 = list(s2)
    if set(l1) == set(l2):
        return True
    else:
        return False
```

Test result is illustrated as follows:



```
# test
string1 = "horse"
string2 = "shore"

print(isAnagram_sorting(string1, string2))
print(isAnagram_dict(string1, string2))

string3 = "heat"
string4 = "that"

print(isAnagram_sorting(string3, string4))
print(isAnagram_dict(string3, string4))

✓ 0.0s
True
True
False
False
```

Figure 1: test

Here I used two different kinds of algorithms to implement this. One is sorting, the logic is firstly, split the string and store each of the character in a list, then I called python built-in function sort, in which it sorted the character by its ASCII value.

The second way is to use hashset data structure to store the distinct element in a hashset, it ensures uniqueness of Elements and has no guaranteed order which enables it to detect if two strings are anagram.

3 Problem 3

3.1 1.

To design an algorithm running in time $O(nk)$. We can iterate each element in L for each element in Q to find the number of smaller elements. The code implementation is written as follows:

```python
def algorithm1(L, Q):
    res = []
    for target in Q:
        count = 0
        for element in L:
            if element < target:
                count += 1
        res.append(count)
    return res
```

There are two for loops with size n and k, thus the time complexity is $O(nk)$.

3.2 2.

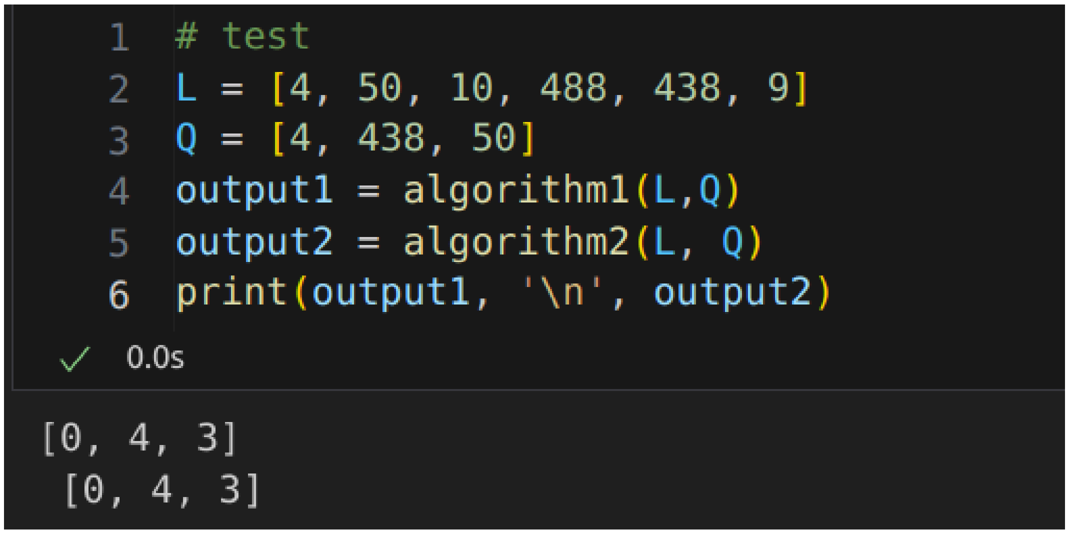
To design an algorithm running in time $O(nlog(n) + klog(n))$. We should firstly design a sorting algorithm with time complexity $O(nlog(n))$. And then find the position of each element in list Q by using binary search which cost $klog(n)$. Thus, we could firstly implement mergeSort which satisfies the requirement and then doing binarysearch. The code implementation is written as follows:

```python
def mergeSort(list):
    if len(list) > 1:
        mid = len(list)//2
        lefthalf = list[:mid]
        righthalf = list[mid:]
        mergeSort(lefthalf)
        mergeSort(righthalf)
        i,j,k = 0,0,0
        while i < len(lefthalf) and j < len(righthalf):
            if lefthalf[i] < righthalf[j]:
                list[k] = lefthalf[i]
                i+=1
            else:
                list[k] = righthalf[j]
                j+=1
            k+=1
        while i < len(lefthalf):
            list[k] = lefthalf[i]
            i+=1
            k+=1
        while j < len(righthalf):
            list[k] = righthalf[j]
            j+=1
            k+=1
    return list

def binarysearch(element, list):
    left = 0
    right = len(list) - 1
    while left <= right:
        mid = left + (right - left) // 2
        if list[mid] == element:
            return mid
        elif list[mid] < element:
            left = mid + 1
        else:
            right = mid - 1
    return -1

def algorithm2(L, Q):
    sorted_list = mergeSort(L)
    res = []
    for target in Q:
        res.append(binarysearch(target, sorted_list))
    return res
```

The test results of both algorithms are illstrated as follows:



```
# test
L = [4, 50, 10, 488, 438, 9]
Q = [4, 438, 50]
output1 = algorithm1(L,Q)
output2 = algorithm2(L, Q)
print(output1, '\n', output2)
```

```
[0, 4, 3]
 [0, 4, 3]
```

Figure 2: test

3.3 3.

Consider the extreme scenario when Q contains all the elements of L which means all we want to get the rank of all the elements. In this case the first algorithm runs in $O(n^2)$ while the second algorithm rums in $O(nlog(n))$. Thus when the size of n and k are the same, the second algorithm is preferred. However, when we only want to get the rank of one element for example, we just need to iterate L once instead of sorting L, in which case the first algorithm is preferred.

4 Problem 4

4.0.1 1.

To design an algorithm in $O(n^2)$. The intuition is to iterate all the elements in both lists and find the smallest distance among them all. The code implementation is written as follows:

```
def algorithm_1(L1, L2):
    res = abs(L1[0] - L2[0])
    for num1 in L1:
        for num2 in L2:
            temp = abs(num1-num2)
            res = temp if temp < res else res
    return res
```

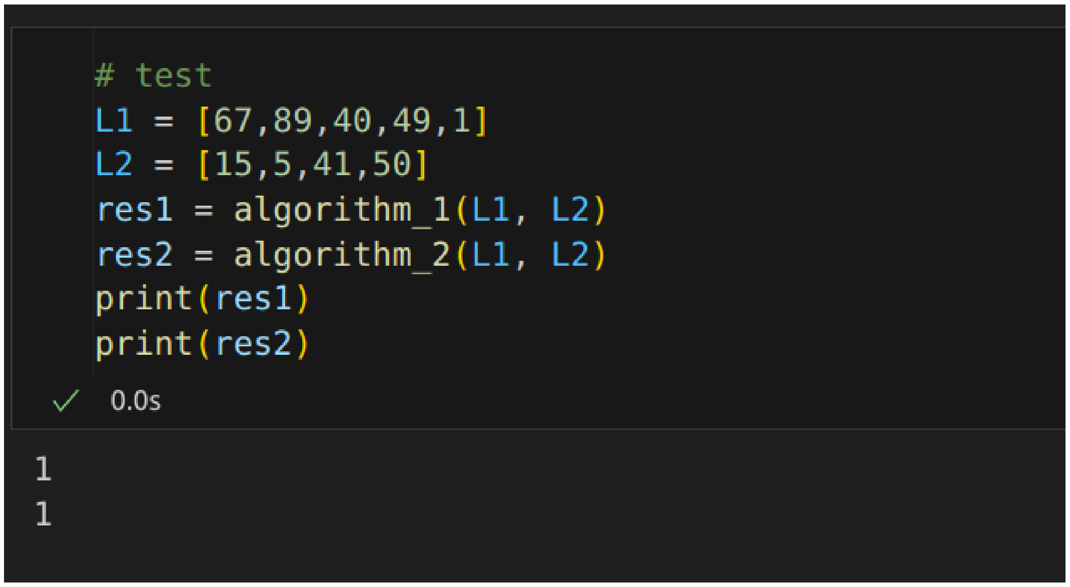
4.1 2.

To design an algorithm in $O(nlogn)$. We already know that mergesort takes $O(nlogn)$, in the following code, I firstly sorted list L1 and L2 respectively, which takes $O(n1log(n1)+n2log(n2)$ then I try to find the minumum distance between L1 and each element in L2. The searching procedure cost $O(n1+n2)$ time in total. Thus the overall time complexity is $O(n1logn1+n2logn2+n1+n2) = O(nlog(n))$

```
def algorithm_2(L1, L2):
    length_L1 = len(L1)
    mergeSort(L1)
    mergeSort(L2)
    p1, p2 = 0, 0
    res = []
    init = float('inf')
    while p1 < len(L1) and p2 < len(L2):
        p1 = 0
        temp = abs(L1[p1]- L2[p2])
```

```
            temp = min(init, temp)
            while L1[p1] < L2[p2]:
                p1 += 1
                temp_other = abs(L1[p1] - L2[p2])
                if temp < temp_other:
                    break
                else:
                    temp = temp_other
            res.append(temp)
            p2+=1
            minimum = min(res)
        return minimum
```

The test result is illustrated as follows:



```
# test
L1 = [67,89,40,49,1]
L2 = [15,5,41,50]
res1 = algorithm_1(L1, L2)
res2 = algorithm_2(L1, L2)
print(res1)
print(res2)
```

✓ 0.0s

```
1
1
```

Figure 3: test