EECE 230X – Introduction to Computation and Programming
Programming Assignment 6

- This programming assignment consists of 4 problems.
- Prerequisites: Topic 8
- Related material: Program efficiency

Problem 1. Efficiency of algorithms in Programming Assignments 3 and 4

In this problem, you are not asked to write a code. Submit your answers in a text file.

We analyzed in Topic 8 the efficiency of algorithms in Programming Assignments 2. In this problem, you will analyze the efficiency of other algorithms from Programming Assignments 3 and 4. Please refer to the solutions posted on moodle.

a) **Programming Assignment 3.** Analyze the **worst case** running time of each part of Problems 1-3 in Programming Assignment 3. For Problems 1,2,3.c, and 3.d you are asked to express the time in terms of Θ notation. For Parts (a) and (b) for Problem 3, it is enough to find an upper bound on the running time using the Big O notation.

a) **Programming Assignment 4.** Analyze the **worst case** and the **best case** running times of the following parts of Programming Assignment 4: Problem 1,2.b,2.c, and all three parts of Problem 3. Use the Θ notation for 1, 2.b,2.c, 3.c, and the Big O notation for 3.a and 3.b.

Problem 2. Sorted 2-SUM, 3-SUM

Recall the 2-SUM problem from Solving Session 2 (PSS 2) (associated with Topic 4) and consider the special case when the list of numbers is sorted.

Sorted 2-SUM problem: Given a list of n integers *sorted* in nondecreasing order and target integer t, check whether or not the list contains two integers whose sum is equal to the target t. In this problem, we are only interested in a YES/NO answer.

In PSS 2, we solved this problem in $O(n^2)$ time using nested loops. In Parts (a) and (b) of this problem, we will take advantage of the fact that the list is sorted to solve it more efficiently. We will use two different approaches.

a) **Sorted 2-SUM in $O(n \log n)$ time using binary search.** Implement the function `sorted2Sum(L,t)`, which given a list `L` of integers and target integer `t`, checks whether or not the list contains two elements (possibly the same element used twice) whose sum is equal to the target.

   Aim for $O(n \log n)$ time. The key is to use the binarySearch function in Topic 8.

   (*Hint:*]i[$L - t$ rof hcraes yranib, i hcae roF)

Test program:

```
print("a) ")
L = [-6, 1, 3, 5, 7, 8, 9, 11]
print(sorted2Sum(L, 14)) # 7+7
print(sorted2Sum(L, 12)) # 1+11
print(sorted2Sum(L, 15)) # 7+8
print(sorted2Sum(L, 3))  # -6+3
print(sorted2Sum(L, 0))
print(sorted2Sum(L, 7))
print(sorted2Sum(L, 21))
```

Output:

```
a)
True
True
True
True
False
False
False
```

b) **Sorted 2-SUM in $O(n)$ time.** Repeat Part (a) but now do it faster: aim for $O(n)$ time.

Instead of using binary search, the idea is to use two counters: i initialized to 0 and j to $n-1$. Compare $L[i]+L[j]$ to t, and accordingly decide how to move i and j.

Call your function `linearSorted2Sum(L,t)`. Test your function on the examples in Part (a):

```
print("b) ")
L = [-6, 1, 3, 5, 7, 8, 9, 11]
print(linearSorted2Sum(L, 14)) # 7+7
print(linearSorted2Sum(L, 12)) # 1+11
print(linearSorted2Sum(L, 15)) # 7+8
print(linearSorted2Sum(L, 3))  # -6+3
print(linearSorted2Sum(L, 0))
print(linearSorted2Sum(L, 7))
print(linearSorted2Sum(L, 21))
```

You should get the same output as in Part (a).

c) **3-SUM in $O(n^2)$ time.** In the 3-SUM problem, we are given a list of n integers (not necessarily sorted) and a target integer t. We would like to check whether or not the list contains 3 integers whose sum is equal to the target t.

The direct solution of this problem based on trying all possible triples requires $O(n^3)$ steps.

Your asked to do it in $O(n^2)$ steps.

(*Hint:* First sort the list using Selection Sort (Programming Assignment 4) or Insertion Sort in $O(n^2)$ steps. Then use the idea of Part (a) and your function in Part (b).)

Call your function `fast3Sum(L,t)`.

Test program:

```
print("c) ")
L = [-6, 1, 3, 5, 7, 9, 11]
print(fast3Sum(L,2))  # e.g., -6+1+7
print(fast3Sum(L,5))  # e.g., 1+1+3
print(fast3Sum(L,7))  # e.g., 1+1+5
print(fast3Sum(L,15)) # e.g., 1+3+11
print(fast3Sum(L,19)) # e.g., 3+7+9
print(fast3Sum(L,0))
print(fast3Sum(L,1))
print(fast3Sum(L,18))
print(fast3Sum(L,20))
print(fast3Sum(L,28))
```

Output:

```
c)
True
True
True
True
True
True
False
False
False
False
```

Problem 3. Binary search modified: first and last occurrences in $O(\log n)$ time

The `binarySearch` function we implemented in Topic 8 returns the index of an occurrence of given element `x` in a sorted list `L`. If `x` appears multiple times in `L`, we don't have any guarantee on whether the returned index is the first or last occurrence.

a) **Find first occurrence ($\star$).** Modify binary search to find the index of the first occurrence of `x`. Your asked to do that without affecting the $O(\log n)$ worst case running of binary search, where n is the length of `L`. That is, implement the function `binarySearchFirstOccurrence(L,x)`, which given sorted list `L` of numbers and a number `x`, return the index of the first occurrence of `x` in `L` if `x` appears in `L`. Else, the function should return `-1`.

(*Hint:* You only need to do simple modifications of the conditions in the `if-elif-else` structure.)

Test program:	*Output:*
`print(binarySearchFirstOccurrence([], 3))`	`-1`
`print(binarySearchFirstOccurrence([5], 3))`	`-1`
`print(binarySearchFirstOccurrence([5], 5))`	`0`
`print(binarySearchFirstOccurrence([3,5,5,5], 5))`	`1`
`print(binarySearchFirstOccurrence([3,5,5,5,5], 1))`	`-1`
`print(binarySearchFirstOccurrence([3,5,5,5,5], 2))`	`-1`
`print(binarySearchFirstOccurrence([3,5,7,7,7,15,26,30,33], 7))`	`2`
`print(binarySearchFirstOccurrence([3,5,7,7,7,15,26,30,33], 33))`	`8`
`print(binarySearchFirstOccurrence([3,5,7,7,7,15,26,30,33], 12))`	`-1`
`print(binarySearchFirstOccurrence([3,3,5,7,15,26,30,33], 26))`	`5`
`print(binarySearchFirstOccurrence([3,3,3,3,3,3,3,3,3,3], 3))`	`0`

b) **Find first and last occurrence.** Similarly implement the function `binarySearchLastOccurrence`. Then implement the function `binarySearchFistAndLastOccurrences(L,x)`, which given sorted list `L` of numbers and a number `x`, returns the tuple `(`i_1`,` i_2`)`, where i_1 and i_2 are the indices of respectively the first and last occurrences of `x` in `L`, if `x` appears in `L`. Else, your function should return `(-1,-1)`. The worst case running time of your function should be $O(\log n)$.

Test program:	*Output:*
`print(binarySearchFirstAndLastOccurrences([], 3))`	`(-1, -1)`
`print(binarySearchFirstAndLastOccurrences([5], 3))`	`(-1, -1)`
`print(binarySearchFirstAndLastOccurrences([5], 5))`	`(0, 0)`
`print(binarySearchFirstAndLastOccurrences([3,5,5,5], 5))`	`(1, 3)`
`print(binarySearchFirstAndLastOccurrences([3,5,5,5,5], 1))`	`(-1, -1)`
`print(binarySearchFirstAndLastOccurrences([3,5,5,5,5], 2))`	`(-1, -1)`
`print(binarySearchFirstAndLastOccurrences([3,5,7,7,7,15,26,30,33], 7))`	`(2, 4)`
`print(binarySearchFirstAndLastOccurrences([3,5,7,7,7,15,26,30,33], 33))`	`(8, 8)`
`print(binarySearchFirstAndLastOccurrences([3,5,7,7,7,15,26,30,33], 12))`	`(-1, -1)`
`print(binarySearchFirstAndLastOccurrences([3,3,5,7,15,26,30,33], 26))`	`(5, 5)`
`print(binarySearchFirstAndLastOccurrences([3,3,3,3,3,3,3,3,3,3], 3))`	`(0, 9)`

Problem 4. Merging two sorted lists in linear time

In this problem, we are interested in merging two lists of numbers which are sorted in non-decreasing order. We would like to take advantage of the fact that they are sorted to merge them efficiently.

Implement the function `merge(L,R)`, which given two sorted lists `L` and `R` of numbers, returns a list `C` consisting of the elements of `L` and `R` merged in sorted order.

Aim for linear time, i.e., $O(m+n)$, where $m =$`len(L)` and $n =$`len(R)`. Use the `list.append` method and the fact that its amortized cost is $O(1)$, i.e., the cost of sequence of k `list.append` operations starting with the empty list is $O(k)$.

Test program:

```
print(merge([1,3,5,7,9],[2,4,6,8,10]))
print(merge([1,5,7],[2,3,5,8,9,9]))
print(merge([1,5,7],[20,30,50,80,90,90]))
print(merge([10,50,70],[2,3,5,8,9,9]))
print(merge([1,5,7],[]))
print(merge([],[2,3,5,8,9,9]))
print(merge([1,2,3],[1,2,3]))
```

Output:

```
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
[1, 2, 3, 5, 5, 7, 8, 9, 9]
[1, 5, 7, 20, 30, 50, 80, 90, 90]
[2, 3, 5, 8, 9, 9, 10, 50, 70]
[1, 5, 7]
[2, 3, 5, 8, 9, 9]
[1, 1, 2, 2, 3, 3]
```