EECE 230X – Introduction to Computation and Programming
Programming Assignment 8

- This programming assignment consists of 4 problems and 1 optional problems.
- Prerequisites: Topic 10
- Related material: Recursion II: Merge Sort, Divide-and-conquer, solving recurrences divide-and-conquer recurrences

Problem 1. Base case of recursive binary search modified

Modify the base case of recursive binary search by performing a linear (sequential) search when the sublist size becomes less than or equal to 5. Complete the following code:

```
def searchHybrid(L,x): # Wrapper function
    """ Assumes L is a list sorted in nondecreasing order
        returns index of an occurence of  x in L if found
        else returns -1
        Algorithm: hybrid binary-search linear-search"""

    def linearSearch(L,x,low,high):
        """ Assumes L is a list sorted in nondecreasing order
            returns index of an occurence of  x in L[low...high] if found
            else returns -1
            Algorithm: linear-search"""
        ....

    def recBinarySearchHybrid(L,x,low, high):
        """ Assumes L is a list sorted in nondecreasing order
            returns index of an occurence of  x in L[low...high] if found
            else returns -1
            Algorithm: hybrid binary-search linear-search: use linear search if size of L[low...high]
            is at most 5"""
        ...

    return recBinarySearchHybrid(L,x,...)
```

Test program:

```
print("searchHybrid([],3):",searchHybrid([], 3))
print("searchHybrid([5],3):",searchHybrid([5], 3))
print("searchHybrid([5], 5):",searchHybrid([5], 5))
L = [2*i for i in range(21)] # L = [0,2,....,40]
print("L=",L)
print("searchHybrid(L, i) for i =-1,..., 41:")
for i in range(-1,42):
    print(searchHybrid(L, i),end=" ")
```

Output:

```
searchHybrid([],3): -1
searchHybrid([5],3): -1
```

```
searchHybrid([5], 5): 0
L= [0, 2, 4, 6, 8, 10, 12, 14, 16, 18, 20, 22, 24, 26, 28, 30, 32, 34, 36, 38, 40]
searchHybrid(L, i) for i =-1,..., 41:
-1 0 -1 1 -1 2 -1 3 -1 4 -1 5 -1 6 -1 7 -1 8 -1 9 -1 10 -1 11 -1 12 -1 13 -1 14 -1 15 -1 16 -1 17 -1
18 -1 19 -1 20 -1
```

Problem 2. Check if a string is a palindrome recursively

Recall from Problem 1 in Programming Assignment 4 that a string is called a *palindrome* if it reads the same both forward and backward. In this problem, you are asked to write a recursive function which given a string, checks whether or not it is a palindrome. The use of for or while loops in this problem is strictly forbidden.

Think in terms of problem and subproblem.

Complete the following program:

```
def isPalindrome(s): # Wrapper function
    def isPalindromeRec(s,low,high):
    """ Recursive function which checks if substring s[low ... high] is palindrome
        returns a True/False value"""
        ....
    n  = len(s)
    return isPalindromeRec(s,0,n-1)

print(isPalindrome("anna"))
print(isPalindrome("civic"))
print(isPalindrome("a"))
print(isPalindrome("tx1aa1xt"))
print(isPalindrome(""))
print(isPalindrome("Civic"))
print(isPalindrome("ab"))
```

Output:

```
True
True
True
True
True
False
False
```

Problem 3. Applications of Sorting in $O(n \log n)$ time

Now that we know how to sort in $O(n \log n)$ using Merge Sort, we will explore few applications of fast sorting. Instead of using Merge Sort, we will use the `list.sort` method which has the same worst case running time, but is practically faster and is more convenient to use.

a) **Element Distinctness in $O(n \log n)$ time.** Recall the Element Distinctness Problem: given a list `L`, check if its elements are distinct. We solved this problem $O(n^2)$ time using nested loops, where `n=len(L)`. Do in $O(n \log n)$ time by first sorting the input list. To avoid modifying the original list, sort a clone of `L`. Call your function `elementDistinctnessFast(L)`. It is supposed to return a `True/False` value.

Test program:

```
print(elementDistinctnessFast([1,2,5,10,3,31,32,33,37,50,250]))
print(elementDistinctnessFast([1,2]))
print(elementDistinctnessFast([1]))
print(elementDistinctnessFast([]))
print(elementDistinctnessFast([1,2,5,2,10]))
print(elementDistinctnessFast([10,1,2,10]))
print(elementDistinctnessFast([1,2,5,10,3,31,32,33,37,5,250]))
print(elementDistinctnessFast([2,2]))
```

Output:

```
True
True
True
True
False
False
False
False
```

b) **2-SUM $O(n \log n)$ time.** Recall the 2-SUM problem from Problem Solving Session 1 (PSS 1) (associated with Topic 4) and Programming Assignment 6 (PA 6). In PSS 1, we solved the problem in $O(n^2)$ time. In PA 6, we solved the special case when input list `L` is sorted in $O(n)$ time, where `n=len(L)`. In this part, you are asked to solve the general case when `L` is not necessarily sorted in $O(n \log n)$ time (Yes: this part is very easy). Write a function `twoSumFast(L,t)`, which given a list `L` of integers and target integer `t`, checks in $O(n \log n)$ time whether or not the list contains two elements (possibly the same element used twice) whose sum is equal to the target. As in Part (a), to avoid modifying the original list, sort a clone of `L`.

Use the function `linearSorted2Sum(L,t)` as given in the solutions of PA 6.

Test program:

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

Output:

```

True
True
True
True
False
False
False
```

c) **4-SUM $O(n^2 \log n)$ time.** In the 4-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 4 integers whose sum is equal to the target t.

Write an efficient function `fourSumFast(L,t)` to solve the 4-SUM problem. Aim for $O(n^2 \log n)$ time (this is not a typo: $O(n^2 \log n)$ not $O(n^3 \log n)$).

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

(*Hint:* .dneppa esU .L ni stnemele fo sriap fo smus lla fo gntisisnoc tsil wen a tcursnoC)

Test program:

```
L = [13,  5, 7, 9, 112,16,27,31]
print(fourSumFast(L,24))   # 5+5+5+9
print(fourSumFast(L,40))   # 13+13+5+9
print(fourSumFast(L,62))   # 13+13+5+31
print(fourSumFast(L,98))   # 13+27+27+31
print(fourSumFast(L,0))
print(fourSumFast(L,10))
print(fourSumFast(L,29))
print(fourSumFast(L,89))
```

Output:

```

True
True
True
True
False
False
False
False
```

Problem 4. Ternary Merge Sort

Instead of dividing the input list into a left sublist and right sublist as in Merge Sort, Ternary Merge Sort divides the input list $A[\texttt{low} \ldots \texttt{high}]$ into three sublists of roughly the same size: $A[\texttt{low} \ldots \texttt{third}]$, $A[\texttt{third} + 1 \ldots \texttt{twoThird}]$, and $A[\texttt{twoThird} + 1 \ldots \texttt{high}]$, where $\texttt{third} = \texttt{low} + (\texttt{high} - \texttt{low})//3$ and

`twoThird` $=$ `low` $+ 2 *$ (`high` $-$ `low`)//3. Then it proceeds recursively as in Merge Sort by first recursively sorting each of the three sublists. Implement Ternary Merge Sort. Call your function `ternaryMergeSort(A, low,high)`, where `low = 0` and `high= len(A)-1` in the initial call. Instead of implementing a new function to merge three sorted lists, use the `merge` function (Programming Assignment 6 and Topic 10) as many times as needed. Pay attention to the base case: it is slightly more complex than that of Merge Sort.

What is the running time of Ternary Merge Sort? Derive a recurrence for the running time of Ternary Merge Sort. Solve it using Master Theorem. Include the recurrence and its solution as comments in your code.

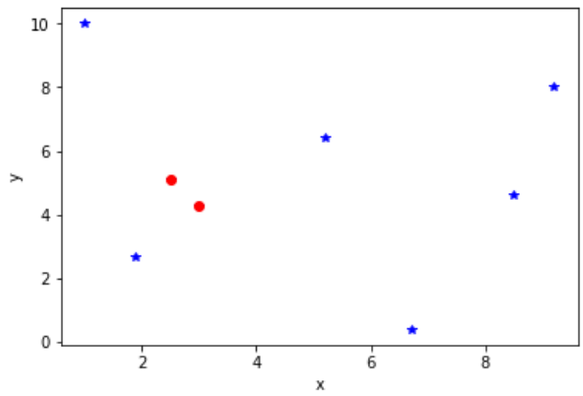
Test it on the following lists:

```
A = []
A= [1]
A = [1,2]
A = [2,1]
A = [5,6,1,3,2,1,7,9,5,15,100, 2,17,56]
A = [20-i for i in range(20)] #, i.e., [20,19, ..., 1]
```

Problem 5 ($\star\star$) (Optional). Closest pair of points

Write a function, which given a list L of n tuples representing planar points, finds the Euclidean distance between the closest pair of points. The Euclidean distance between points (x_1, y_1) and (x_2, y_2) is $d((x_1, y_1), (x_2, y_2)) = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$

Example, if `L = [(3.0, 4.3), (9.2, 8.0), (1.0, 10.0), (5.2, 6.4), (8.5, 4.6), (6.7, 0.4), (2.5, 5.1), (1.9, 2.7)]`, then the closest pair are `(3.0, 4.3)` and `(2.5, 5.1)`, as shown in the below figure. The desired output is thus $d((3.0, 4.3), (2.5, 5.1)) = 0.9433981132056602$.



The nested-loops solution for this problem takes $\Theta(n^2)$ time. Our objective is to do it faster. Aim for $O(n \log n)$ time.

(*Hint:* divide-and-conquer)

Note that the 1-dimensional version of this problem (distance is the absolute value) has a simple $O(n \log n)$ solution: first sort, then traverse the point while keeping track of the minimum distance between two contiguous points.