Assignment 2

420-ENM-MT Algorithm, Pseudo-code and Design

Due: December 6th 2018 before class

Instructions

- This assignment is worth **5% of your final grade**;
- You must submit four (4) files on **Omnivox**. Submit theory questions as a single **pdf** and coding questions as **`.py`** files;
- **Read every question carefully**;
- Some problems require you to write code. Make sure to read the instructions and submit the corresponding `.py` files with your assignment;
- Your code will be tested with **Python 3.7**;
- Using online resources is allowed, but you should cite your sources for any algorithm or code you did not write yourself. You must prove you understood the solution by properly commenting it, otherwise you will be given a grade of zero for **plagiarism**.

Problem 1 (10 points)

For each of the following situations, give the time-complexity of the described algorithm in big-O notation.

a) (1 point) An algorithm takes a list of length n as input and needs around $5n + 8$ operations to return.

b) (1 point) An algorithm takes an integer k as input and then proceeds to do at most $k^2 + k$ operations.

c) (2 points) A program takes an array-list of length n as argument and prints it in reverse.

d) (2 points) An application has a list of all its n registered user in alphabetical order. It uses Binary Search to check if a username already exists.

e) (2 points) To generate a three dimensional grid, a program takes a list of length n as input and returns a new list of all combinations of three elements from the list.

f) (2 points) To return the smallest element of a sequence, a program uses Insertion Sort and then outputs the first element of the sorted sequence.

Problem 2 (10 points)

The following function takes as input an array-list of numbers. Read the pseudo-code and answer the questions below.

```
function secret_function(numbers):
      new_numbers = InsertionSort(numbers)

      length = length of numbers

      low = length
      high = length

      for each index from 0 to (length - 1):
            if new_numbers[index] >= 0:
                  low = index
                  stop looping

      for each index from low to (length - 1):
            if new_numbers[index] > 10:
                  high = index
                  stop looping

      return new_number[low:high]
```

a) (2 points) Write the output after calling

```
secret_function([-5, 10, 9, 11, 3, 5, 0, 21])
```

b) (2 points) In one or two sentences, describe what the above function does.

c) (3 points) What is the time-complexity of `secret_function`?

d) (3 points) Suggest a way to improve the time-complexity of the function and give the new time-complexity.

Problem 3 (10 points)
In the previous assignment, you were asked to implement parentheses matching for an IDE. This algorithm allowed to detect a parentheses mismatch, but only for the characters **( )**. You are now asked to extend that algorithm to implement bracket matching.

Bracket matching is similar to parentheses matching, but it allows more than one type of parentheses. You are asked to implement bracket matching for the characters **( )**, **[ ]** and **{ }**.

Using **pseudo-code**, implement a function called `match` which takes a string of brackets as argument and returns `True` if the brackets match and `False` if the brackets mismatch.

Examples:

```
match("( [ ] )") → True
match("[ { } ( ) ]") → True
match("( ( ] ]") → False
match("} [ ] {") → False
match("{ ( } )") → False
```

Hint: This can only be solved with a stack.

Problem 4 (10 points)

In class, we covered an efficient sorting algorithm called Merge Sort. Here is a Python implementation of this algorithm.

```
def mergesort(sequence):
    if len(sequence) <= 1:
        return sequence[:]
    else:
        mid = len(sequence) // 2

        left = mergesort(sequence[:mid])
        right = mergesort(sequence[mid:])

        return merge(left, right)
```

One of the key principle of Merge Sort is that we can implement a function `merge` which takes two sorted lists as argument and merge them together into one bigger sorted list. In particular, this function runs in **`O(n)`** time-complexity.

Write the **Python** function `merge` that is needed to implement Merge Sort.

Answer this question by filling in the file `merge.py`, then submit it with your assignment.

Problem 5 (15 points)
Real life data is often not totally random. Some sorting algorithms take advantage of this fact by looking for increasing or decreasing subsequence in a list beforehand. This prevent the algorithm from spending time sorting an already sorted subsequence.

Write a **Python** function `is_sorted` that takes a list of integers as argument and computes whether it is in increasing order, decreasing order or not sorted at all.

The function should **return** `1` if the list is sorted in **increasing** order.
The function should **return** `-1` if the list is sorted in **decreasing** order.
The function should **return** `0` if the list is **not sorted**.
Finally, the function should **return** `1` if a list is either **empty or constant**.

Examples:

- `is_sorted([ ]) → 1`
- `is_sorted([1, 1, 1, 1]) → 1`
- `is_sorted([0, 3, 5, 7, 13]) → 1`
- `is_sorted([7, 3, -4, -17] → -1`
- `is_sorted([1, 3, 5, 3, 2]) → 0`

You are not allowed to use any Python library as well as built-in sorting methods or functions.

Your algorithm must be `O(n)`, but you are not required to prove its time-complexity.

Answer this question by filling in the file `is_sorted.py`, then submit it with your assignment.

Problem 6 (15 points)
A retail company wants to repurpose some of its shops to increase its profit. They estimate that they could save in distribution costs by avoiding selling the same item in shops which are located close to one another.

Since the company is selling a very broad range of items, it is not possible for a human accountant to analyze the whole inventory and extract duplicate items.

You are asked to write an algorithm to find the ids of items two shops have in common in store.

Write a **Python** function `find_duplicates` that takes two list of unique integers, representing item ids, as arguments and outputs a list of all ids which were found in both lists. The output ids must be in increasing order.

Example:

- `find_duplicates([1, 2, 3], [2, 4, 5])` → `[2]`
- `find_duplicates([7, 2, 5], [2, 4, 7, 8])` → `[2, 7]`
- `find_duplicates([1, 5, 2, 6], [8, 9, 10])` → `[ ]`

The company has millions of different items for sale and hundreds of stores to compare, so it is important that the algorithm be efficient. You are asked to implement an algorithm that is at least **`O(n log(n))`**, where `n` is the total number of items. You do not need to prove the time complexity of your algorithm.

You are not allowed to any Python library, but using built-in functions is permitted.

Answer this question by filling in the file `find_duplicates.py`, then submit it with your assignment.