Assignment 5 – Sorting Algorithms

CSDS 233 – Introduction to Data Structures

General instruction: This assignment includes two parts, written and programming. Please write/type your answers neatly so they can be readable. Please submit a single PDF file for the written part and a zip file for the programming part before **11:59 P.M. on November 28th, 2023.** Please ensure that your written assignment is outside the zip file.

File name format: P5_YourCaseID_YourLastName.zip (or pdf).

Office Hours + Additional Office Hours

Kim Tran (kdt46@case.edu): Tuesday, 10:00am - 11:00am
Tammy Lin (txl619@case.edu): Wednesday, 10:00am - 11:00am
Additional Office Hours:

- Thursday, November 16th, 3-5pm in Glennan Lounge

Best of luck!

Written Assignment (50 points)

1. **[12 pts] Sorting Algorithms**
   Given the following list {9, 7, 4, 1, 8, 3, 6, 7} and sorting functions below, write out each step of the sorting and comparison process until a list is sorted.
   a. Merge sort
   b. Quick sort
   c. Selection sort

2. **[14 pt] Big-O**
   a. Given this array {9, 8, 7, 6, 3, 2, 0}, what is the tightest big-O for these cases when running quicksort? Solution should be in terms of n where n is the size of the array. Please explain how you got your answer.
      [1] when the pivot is the first element
      [2] when the pivot is the median value

   b. What is the tightest bound for this mystery sort function? Show time complexity of each loop as well as the overall time complexity for mystery sort.

```
public static void mysterySort(int arr[], int n)
        {
        boolean sorted = false;

        while (!sorted){
            sorted = true;
            int temp = 0;

            for (int i=1; i<=n-2; i=i+2) {
                if (arr[i] > arr[i+1]) {
                    temp = arr[i];
                    arr[i] = arr[i+1];
                    arr[i+1] = temp;
                    sorted = false;
                }
            }

            for (int i=0; i<=n-2; i=i+2){
                if (arr[i] > arr[i+1]){
                    temp = arr[i];
                    arr[i] = arr[i+1];
                    arr[i+1] = temp;
                    sorted = false;
                }
            }
        }

        return;
    }
```

3. **[12 pts] Heap sort**

   Given the following array: {2, 10, 1, 7, 30, 2}, demonstrate how to sort this array using heapsort. Show the heap at each action throughout the algorithm.

4. **[12 pts] Pseudocode**

   Write pseudocode for the following: Given a string S, sort the characters in the string by frequency. For example: “ghjklghjlgjj” → “khhllgggjjjj”. If a frequency is the same, sort in alphabetical order. Characters in the string will all be lowercase a-z. Your pseudocode should look like code, but does not necessarily have to compile.

Programming Assignment (50 points)

Case Western wants to improve upon the current CaseCash plan and create an application that allows the following functionalities:

- Transfer of CaseCash from Student A to Student B
- Deposit CashCash into a Student account
- Withdraw CaseCash
- View student account balance
- Be able to sort students by name and account balance

Create the following classes and functions:

Student

Required fields:

`String name`

- The name of the student

`int balance`

- The current balance in the student account

Required methods:

`public Student(String name, int balance)`

- Public constructor that initializes the fields name and balance

`public int getBalance()`

- Returns the balance the student has in their account

`public void updateBalance(int newAmount)`

- Updates the balance the student has in their account

CaseCashSystem

Required fields:

`List<Student> students`

- The list of students with accounts created.

Required methods:

`public List<String> runSimulation(List<String> commands)`

- Function used to run the simulation provided by the commands. This function should parse the commands and call their respective helper function to complete a task. More details regarding what inputs look like are on the next page. Note: Every call to runSimulation() should clear the previous list of students (starts clean).

`public boolean init(String name, int initialBalance)`

- Initializes a student with a name and an initial account balance. This should return true if the student has not been created already, and return false if a student with this name already exists. InitialBalance cannot be negative. Corresponds to "***INIT***, `name, initialBalance`".

`public int getBalance(String name)`

- Return the balance of a given student. Corresponds to "***GET***, `name`".

`public boolean deposit(Student student, int amount)`

- Deposit money from a student account. Return true if deposit is successful, and return false if deposit fails (negative input). If false, the balance of the student account should not be changed at all. Corresponds to "***DEPOSIT***, `studentA, amount`".

`public boolean transfer(Student studentA, Student studentB, int amount)`

- Transfers the amount from studentA account to studentB account. This function should return true if transferring is successful, and return false if transferring money from A to B will result in a negative balance. The parameter amount cannot be negative. If false, the balance of account A and B should not be changed at all. Corresponds to "***TRANSFER***, `studentA, studentB, amount`".

`public List<String> sortName()`

- Returns a list of student names in alphabetical order. You are not allowed to use the Java sorting functions, and should write your own. This function should utilize **merge sort**. If merge sort is not implemented, significant points will be deducted. Corresponds to "***SORT***, `name`".

`public List<String> sortBalance()`

- Returns a list of student names in the order of smallest balance to largest balance in their account. You are not allowed to use the Java sorting functions, and should write your own. This function should utilize **quick sort**. If quick sort is not implemented, significant points will be deducted. Corresponds to "***SORT***, `balance`"

`public boolean withdrawal(Student student, int amount)`

- Remove money from a student account. Return true if remove is successful, and return false if removing will result in a negative balance. If false, the balance of the student account should not be changed at all. Corresponds to "***WITHDRAWAL***, `studentA, amount`".

Input Formats and Examples:

Inputs will always be of this format:

```
"INIT, name, initialBalance"
"GET, name"
"TRANSFER, studentA, studentB, amount"
"WITHDRAWAL, studentA, amount"
"DEPOSIT, studentA, amount"
"SORT, name or balance"
```

```
List<String> inputs = ["INIT, Tammy, 200",
                       "INIT, Kim, 300",
                       "INIT, Quyen, 400",
                       "SORT, name",
                       "SORT, balance",
                       "TRANSFER, Kim, Tammy, 100",
                       "SORT, name",
                       "SORT, balance"]

List<String> outputs = runSimulation(List<String> inputs);

System.out.println(outputs);
     >>> ["true",
          "true",
          "true",
          "[Kim, Quyen, Tammy]",
          "[Tammy, Kim, Quyen]",
          "true",
          "[Kim, Quyen, Tammy]",
          "[Kim, Tammy, Quyen]"]
```

Conditions + Additional Notes:

- ***INIT*** for a student must be called prior to making deposits and withdrawals for that student. Otherwise a function should return false.
- ***GET, TRANSFER, WITHDRAWAL***, and ***DEPOSIT*** being called assumes that the student exists. TAs will not be testing with invalid inputs. However, if you want to add validation, feel free.
- Student names are unique, meaning each student must have a different name.
- You can utilize `HashMap`, `HashSets`, `ArrayList`, `LinkedList` however you deem necessary. However, usage of all of these are not required.
- All inputs in `List<String> commands` for `runSimulation` will ALWAYS be one of the 6 input formats, you will not have to do any validation to check if inputs are formatted correctly.

Grading

Student Implementation (5 points)

- Student class is implemented correctly with correct field names and functions.

CaseCashSystem Implementation (30 points)

- [5 pt] Run simulation method implementation
- [10 pt] Sort name implementation
- [10 pt] Sort balance implementation
- [2 pt] Transfer, deposit, withdrawal function implementation
- [2 pt] Init and get function implementation
- [1 pt] Constructor implementation, required fields, and functions are present

JUnit Tests (5 points)

- [2.5 pt] Sort balance test (at least 3 cases)
- [2.5 pt] Sort name test (at least 3 cases)

Clean Code and Design (5 points)

- Code should have comments above all complex lines of code. This includes loops, if blocks, etc.

Encapsulation and Abstraction (5 points)

- [2pt] Fields and helper functions are private
- [3pt] Student class is abstracted in the CaseCashSystem class
    - This means that from another file such as the test files, you are not directly making students, but rather using the init function to create a new student.