Programming Design, Spring 2016
Homework 11

Instructor: Ling-Chieh Kung
Department of Information Management
National Taiwan University

Please upload one PDF file for Problem 1 and two CPP files for Problems 2 and 3 (optional) to PDOGS at http://pdogs.ntu.im/judge/. Each student must submit her/his individual work. No hard copy. No late submission. The due time of this homework is ***2:00 am, May 23, 2016***. Please answer in either English or Chinese. The maximum point of this homework is 120.

Before you start, please read Chapter 11 of the textbook.[1]

The TA who generates the testing data and grades this homework is ***Chien Huang***.

Problem 1

(20 points; 5 points each) Please answer the following questions regarding `MyVector` introduced in class.

(a) (5 points) Consider the two overloaded `[]` on page 30 of the slides. Explain why we need two versions, and when will each version be invoked.

(b) (5 points) Consider the overloaded `=` on page 39 of the slides. Explain why we should return `MyVector&` instead of `MyVector`.

(c) (5 points) Consider the overloaded `+` on page 44 of the slides. Explain why we should return `MyVector` instead of `MyVector&`.

(d) (5 points) Overload the subtraction operator `-` for `MyVector`.

Problem 2

(55 points) Enabled by modern information technology, sharing economy is very popular nowadays. For example, Uber helps passengers and (amateur) drivers find each other. Today we will write a program for such a car-sharing platform to connect passengers and drivers.

In the system, there are passengers (he) and drivers (she), represented by two classes `Passenger` and `Driver`. Consider `Passenger` first:

```
class Passenger
{
private:
  static int offlineCnt;
  static int searchingCnt;
  static int travelingCnt;
  int id;
  int status;
  int x;
  int y;
  int ratingCnt;
  int* ratings;
};
```

[1] The textbook is *C++ How to Program: Late Objects Version* by Deitel and Deitel, seventh edition.

These attributes are explained below:

1. Each registered passenger has a unique ID in the system. It is an integer stored in `id`.
2. A passenger has three possible types of status: offline, searching, and traveling. One is in the searching status when he opens the app to search for a driver; one is in the traveling status is he is in his way to his destination; otherwise he is offline. For simplicity, after a passenger is assigned a driver by the system (but before the driver reaches the passenger), his status is traveling. The variable `status` is used to store one's status.
3. A passenger's location is recorded as a point on the two-dimensional space. We assume that all passengers (and drivers) locate on the grid points on a plane, so a location can be expressed by integers `x` and `y`. Of course when one is offline, the system does not track his location. In this case, `x` and `y` records the location when this passenger closes the app.
4. A passenger gets rated by a driver after a trip. Each rating is an integer between 1 and 5. The number of times a passenger is rated is recorded in `ratingCnt`.
5. All the ratings one has received is recorded in a dynamic vector pointed by `ratings`, where its ith element is the ith rating the passenger received. Note that `ratingCnt` and `ratings` can be used to calculate one's average rating. If one has no rating, his average rating is defined to be 3.
6. The three static variables records the numbers of offline, searching, and traveling passengers. Note that they are static rather than instance variables because they are attributes of the class, not an object. Also note that it is better to put them as static variables of `Passenger` than to put them as local or global variables outside `Passenger` (why?).

Of course, you are allowed to add more members into `Passenger`.

The class `Driver` is very similar to `Passenger`:

```
class Driver
{
private:
  static int offlineCnt;
  static int searchingCnt;
  static int travelingCnt;
  int id;
  int status;
  int x;
  int y;
  int ratingCnt;
  int* ratings;
public:
  bool operator>(const Driver&) const;
};
```

A driver also has the same nine attributes. While you are allowed to add more members into `Driver`, you are required to overload the operator `>`, which compares the average rating of two drivers. More precisely, if `d1` and `d2` are two `Driver` objects, `d1 > d2` should return true if one of the following is true:

- `d1` has a strictly higher average rating;
- They have the same rating and `d1` has a smaller driver ID.

If `d1 > d2`, we say `d1` has *higher rating priority* than `d2`.

Six kinds of events may happen:

1. (Event P) A passenger signs up: A new `Passenger` variable should be created. The initial status is offline.

2. (Event D) A driver signs up: A new `Driver` variable should be created. The initial status is offline.

3. (Event O) A driver opens the app: The driver's status changes from offline to searching. Her current location (x, y) will be given.

4. (Event S) An offline passenger opens the app and starts to search for a driver: The passenger's status changes from offline to searching, the system finds an available driver (whose status is searching) and assign it to the passenger, and then both the passenger and driver change from searching to traveling. When he opens the app, his current location (x, y) will be given.

5. (Event A) A passenger arrives its destination: The passenger's status becomes offline, and the driver's status becomes searching. Their destination location (x, y) will be given. The two ratings r_P and r_D given by the passenger and driver, respectively, will also be given.

6. (Event C) A driver closes the app: Her status changes from searching to offline. A driver is not allowed to close the app when she is traveling.

For simplicity, we assume that drivers do not move unless she is assigned to a passenger and brings him to the destination. In other words, after she opens the app, she stays there; after she gets to a destination, she stays there.

The most interesting part (if any) of these events is event S. The system finds a driver in the following way. First, it checks the searching passenger's average rating. If it is strictly below 2, it will disallow the passenger to find a driver, and the passenger will give up and close the app. Otherwise, it will find all the searching drivers whose Manhattan distance from the passenger is no greater than M, i.e.,

$$|x^P - x^D| + |y^P - y^D| \leq M,$$

where (x^P, y^P) and (x^D, y^D) are the locations of the searching passenger and a given searching driver, respectively. Among all drivers who are far from the searching passenger by no greater than M, the system assigns the one with the highest rating priority. If there is no such a driver, the system does the same search for Manhattan distances $2M$, $3M$, ..., until a driver is found and assigned. If there is no searching driver at all, the system will say sorry to the passenger, and the passenger will give up and close the app.

You will be given a sequence of events. By processing these events, you will update the statuses of the passengers and drivers. At the end you should print out the IDs of all searching drivers, in the order of their registration times.

Input/output formats

There are 15 input files. In each file, there are $n + 1$ lines, where $n \leq 10000$. The first line contains an integer M. Starting from the second line, each line contains first a letter s which is one of the following six letters: P, D, O, S, A, and C. Each letter represents an event type. For each type of event, some parameters follow:

1. (Event P) After the letter P, there is an integer as the passenger ID.

2. (Event D) After the letter D, there is an integer as the driver ID.

3. (Event O) After the letter O, there is an integer as the driver ID, an integer as x, and then an integer as y, where (x, y) is the driver's current location.

4. (Event S) After the letter S, there is an integer as the passenger ID, an integer as x, and then an integer as y, where (x, y) is the passenger's current location.

5. (Event A) After the letter A, there is an integer as the passenger ID, an integer as x, an integer as y, an integer r_P, and an integer r_D, where (x, y) is the passenger's (and driver's) destination location, r_P is the rating given by the passenger to the driver, and r_D is the rating given by the driver to the passenger.

6. (Event C) After the letter C, there is an integer as the driver ID.

For each line, there is a white space between any two consecutive values. You may assume that x and y are within -1000 and 1000, passenger IDs are non-repeating, driver IDs are non-repeating, and passenger and driver IDs are all within 1 and 99999. You may also assume that all events are reasonable, e.g., a traveling driver cannot close the app, a passenger must sign up before he does anything, a passenger cannot close the app twice before he opens it, etc.

For example, consider the following input:

```
4
P 1
P 2
D 1
D 4
D 3
O 1 0 0
S 1 10 10
S 2 5 5
A 1 300 300 5 1
C 1
O 1 10 11
O 4 10 10
O 3 6 6
S 2 10 9
S 1 10 10
D 2
O 2 -1 -1
C 3
```

Let's process these events:

- First two passengers and three drivers sign up. Then driver 1 opens the app at $(0, 0)$. When passenger 1 then searches for a driver, driver 1 is assigned. As no other drivers are available, when passenger 2 then searches for a driver, he will not be served and will give up. Passenger 1 (and driver 1) then arrives $(300, 300)$, driver 1 gets a rating 5, and passenger 1 gets a rating 1. Driver 1 then closes the app and later reopens it at $(10, 11)$.
- Then drivers 4 and 3 open the apps at $(10, 10)$ and $(6, 6)$. When passenger 2 later searches for a driver at $(10, 9)$, all the three drivers are available. First we calculate the three drivers' distances to the passenger, which are 2, 1, and 7. As $M = 4$, the system will pick one driver between drivers 1 and 4. As driver 1's average rating (5) is greater than driver 4's (3, by default), driver 1 will be assigned, even though driver 4 is closer to the passenger.
- Now passenger 1 searches for a driver again. Unfortunately, his average rating (1) is strictly lower than 2. Therefore, his order is rejected, and he gives up.
- Driver 2 signs up and opens the app at $(-1, -1)$, and driver 3 closes the app.

As we may see, at the end only drivers 4 and 2 are available. Therefore, the output should be

```
4 2
```

In general, the IDs of available drivers should be printed out according to their registration times, where two consecutive values should be separated by a white space. As driver 4 signs up earlier than driver 2, the output should be `4 2`, not `2 4`.

What should be in your source file

Your .cpp source file should contain C++ codes that will both read testing data and complete the above task. For this problem, you are allowed to use only techniques covered so far. NO other techniques are allowed. Finally, you should write relevant comments for your codes.

Grading criteria

You must use the given classes to store the given input information. If you fail to do so, you will get no point. If you do, you will be graded according to the following rule:

- 45 points will be based on the correctness of your output. PDOGS will compile your program, feed testing data into your program, and check the correctness of your outputs. Each fully correct set of outputs gives you 3 points.
- 10 points will be based on how you write your program, including the logic and format. Please try to write a robust, efficient, and easy-to-read program.

Problem 3

(45 points) This problem is a complication of Problem 2. Now we are interested in knowing the amount of money each driver earns. By completing a trip of traveling distance d, a driver collects revenue

$$p = \begin{cases} a & \text{if } d \leq A \\ a + b(d - A) & \text{otherwise} \end{cases},$$

where $A > 0$, $a > 0$, and $b > 0$ are given pricing parameters. We assume that a driver always drives from the starting location to the destination along a shortest path. Therefore, if the starting location is (x_0, y_0) and the destination location is (x_1, y_1), then we have $d = |x_0 - x_1| + |y_0 - y_1|$.

Given all the events as in Problem 2 and pricing parameters, your program should print out the total revenues earned by all drivers in the order of their registration times.

Input/output formats

There are 15 input files. In each file, there are $n + 2$ lines of input. The first line contains three integers A, a, and b, separated by two white spaces. The last $n + 1$ lines follow the format given in Problem 2. Given the input, the output should be the total revenues earned by each individual driver in the order of their registration times, where two consecutive values are separated by a white space.

For example, for the input

```
100 50 2
4
P 1
P 2
D 1
D 4
D 3
O 1 0 0
S 1 10 10
```

```
S 2 5 5
A 1 300 300 5 1
C 1
O 1 10 11
O 4 10 10
O 3 6 6
S 2 10 9
S 1 10 10
D 2
O 2 -1 -1
C 3
```

the output should be

```
1010 0 0 0
```

where $1010 = 50 + 2 \times (580 - 100) = 1010$.

What should be in your source file

Your .cpp source file should contain C++ codes that will both read testing data and complete the above task. For this problem, you are allowed to use any technique. You are not even required to use the classes defined in Problem 2. Finally, you should write relevant comments for your codes.

Grading criteria

45 points will be based on the correctness of your output. PDOGS will compile your program, feed testing data into your program, and check the correctness of your outputs. Each fully correct set of outputs gives you 3 points.