AY 2019 Term 2 Take-Home Exam

Date / Time	18 December 2019, 08:00 – 18 December 2019 23:59
Course	**[M1522.600] Computer Programming**
Instructor	Youngki Lee

- You can refer to the Internet or other materials during the exam, but you ***SHOULD NOT*** discuss the question with anyone else and need to code **ALONE**.
- We will use the automated copy detector to check the possible plagiarism of the code between the students. The copy checker is reliable so that it is highly likely to mark a pair of code as the copy even though two students quickly discuss the idea without looking at each other's code. Of course, we will evaluate the similarity of a pair compared to the overall similarity for the entire class.
- We will do the manual inspection of the code. In case we doubt that the code may be written by someone else (outside of the class), we reserve the right to request an explanation about the code. We will ask detailed questions that cannot be answered if the code is not written by yourself.
- If one of the above cases happen, you will get 0 marks for the exam and may get a further penalty. Please understand that we will apply these methods for the fairness of the exam.
- Download and unzip “Exam.zip” file from the ETL. “Exam.zip” file contains skeleton codes for C++ problems (in the “cpp” directory) and Java problems (in the “java” directory).
- Do not modify the overall directory structure after unzipping the file, and fill in the code in appropriate files. It is okay to add new directories or files if needed.
- When you submit, compress the “cpp” and “java” directories in a single zip file named “20XX-XXXXX.zip” and upload it to ETL as you submit the solution for the lab tests. Contact the TA if you are not sure how to submit. Double-check if your final zip file is properly submitted.
- Do not use external libraries.

Question 1: The Prime Tournament [C++; 50 Marks]

Objective: A tournament where players compete with each other on a game involving prime numbers.

Description: There is a game called “Prime Battle”, where two players hold number cards and compete with each other. It was named so because prime numbers are used to determine who is a winner for each match.

In this problem, we will first implement “Prime Battle” by completing logics involving prime numbers. Next, we will define several types of players regarding their risk-taking strategies. Finally, we will make a tournament consisting of many players and determine who wins the first prize.

Notes

1. We will only use standard input and output throughout this problem. `main.cpp` is also given that inputs from user and outputs in a valid format. Being a take-home exam, the remaining part from logic to design is up to you. Be sure not to change the behavior of `main.cpp` and please do not leave debugging messages printed so that your program is correctly graded.
2. **Indicate your OS (e.g., Windows, Linux, MacOS)** in the first line of `main.cpp` as a comment.
3. The program you should implement will be quite complex. We strongly recommend you to modularize your program. Especially, please write your program in different files and leave `main.cpp` as a bridge interface that only deals with input and output.
4. Please make sure you understand the concept of prime numbers and integer factorization. We provide two important functions in `main.cpp` to solve this problem: one for the primality tests and another for integer factorization.
5. We provide the example test cases for Question 1-1 and Question 1-2 under the “testcases” directory. For instance, “1.in” and “1.out” under the “testcases/1-1” directory describes the first test case of Question 1-1. You need to create your own test cases from Question 1-3 to Question 1-6.

Question 1-1: Battle of Two Numbers [8 Marks]

Objective: Implement `pair<int,int> number_fight(int a, int b)` in main.cpp

Description: Using integer factorization, we can make two positive integers fight with each other. The rule is as follows:

- Given two integers A and B, factorize them (denoted as FA, FB)
    - For convenience, let's define the factorization of 1 is { 1 }
- Make the list of **distinct** prime numbers that are in both FA and FB (denoted as FG)
- Calculate the product of numbers in FG (denoted as G)
    - If FG is empty, then set G as 1
- Divide A and B by G, respectively.

In other words, we will factorize two numbers and remove all intersecting prime factors. But if they share the same prime factor more than once, we will just take one, not all of them.

For example, let's consider two numbers 24 and 36

- A = 24, B = 36
- FA = {2, 2, 2, 3}, FB = {2, 2, 3, 3}
- FG = {2, 3}
- G = 6
- A ← 24 / 6 = 4, B ← 36 / 6 = 6

So, after one fight, 24 and 36 will become 4 and 6, respectively. Implement this algorithm in `pair<int,int> number_fight(int a, int b)`.

- Input
    - `a`, `b`: positive integers less than 10^6
- Output
    - `std::pair<int,int>` that contains the changed value of `a` and `b` after one fight.
    - The first element of the pair is for `a`, and the second for `b`.

Question 1-2: Fight or Flight? [8 Marks]

Objective: Implement `pair<int,int> number_vs_number(int a, int b)` in main.cpp

Description: Did you notice that fighting always makes numbers small, or the same at best? This is why we sometimes choose not to fight even when we are very offended. The same holds for the world of numbers. Now numbers itself has two options: to fight or not to fight.

So, when two numbers are matched to each other, there can be a total of 4(=2×2) cases.

- If both choose to fight, it is the same as the previous question.
- If both choose not to fight, nothing changes.
- If only one of them chooses to fight, we will call this *attack*. This is what happens when one attacks another.
    - Let's define A' and B' as follows: if both A and B choose to fight, then the values will change into A' and B'.

- When A attacks B, we define **the *damage* D** of attack as (B - B')
    - If B has 7 as its prime factor, then both A and B get the half damage.
        - A ← A - floor(D / 2), B ← B - floor(D / 2)
        - If A or B becomes less than 1, set it as 1.
    - If B does not have 7 as its prime factor, then B gets all damage.
        - B ← B - D (= B')

According to the new rule, which option is good depends on the situation. In principle, both can choose not to fight for the sake of both. However, sometimes they may have to attack for self-defense. Maybe it reminds you of the Prisoner's Dilemma?

Now, a number A will choose to fight or not based on the following algorithm

1. Assuming the opponent B will fight, the number compares two outcomes of choosing to fight or not.
    - Let's denote the outcome as A1(fight) and A2(not fight), respectively.
    - If A1 >= A2, it makes a *conditional decision* to fight with B. Otherwise, it makes a conditional decision not to fight with B.
        - The real decision is not made yet. Please read the following procedure 2 and 3
2. Assuming the opponent B will not fight, it does the same to make another conditional decision.
3. If two conditional decisions in 1 and 2 agree, it will adopt that decision. Otherwise, it will choose to fight if and only if, before fighting, the opponent(=B) is **strictly bigger** than itself(=A).

The same decision-making procedure holds for B.

Consider the following example.

(A, B) = (14, 12)	12(=B) chooses to fight	12(=B) chooses not to fight
14(=A) chooses to fight	(A, B) ← (7, 6)	(A, B) ← (14, 6)
14(=A) chooses not to fight	(A, B) ← (11, 9)	(A, B) ← (14, 12)

In the perspective of A,

1) Assuming B will fight, A should choose not to fight to get the better result(=11)
2) Assuming B will not fight, A should choose to fight as two outcomes are the same(=14)
3) As the two decisions do not agree with each other, A will choose not to fight as B is not bigger than itself.

Similarly, we can conclude that B will choose to fight. This results in (A, B) becoming (11, 9) after one battle.

Implement the fight of two numbers in `pair<int,int> number_vs_number(int a, int b)`

- Input
  - `a`, `b`: positive integers less than 10^6
- Output
  - `std::pair<int,int>` that contains the changed value of `a` and `b`. Unlike question 1-1, they will choose whether to fight or not according to the algorithm described above.
  - The first element of the pair is for `a`, and the second for `b`.

Question 1-3: Player Types [12 Marks]

Objective: Implement `pair<multiset<int>, multiset<int>> player_battle(string type_a, multiset<int> a, string type_b, multiset<int> b)` in `main.cpp`

Description: So far, we have discussed the battle between two numbers. Now we're going to think about a battle between two players. Each player has several numbers in their hands, and it will select one of the numbers and send it to the fight. After one battle, two cards are sent back to their owners with their values (possibly) changed. The rule of the battle between two numbers is the same as question 1-2.

Let's say a player has N numbers in its hands. It also knows all the M numbers of its opponent. Which number should the player select to fight?

As a player knows the consequences of all N x M cases, it can adopt several strategies to get a better result. Here, ***the better(or the best) result(outcome)*** is defined by the sum of all numbers in its hands. The bigger the sum, the better. Especially, it has nothing to do with how small the opponent's numbers become.

In perspective of risk-managing strategy, there are 3 types of players:

1. **Maximize-Gain**: A player of this type always assumes and seeks the best case. It selects a number that can produce the best result if the opponent makes a choice the player wants.
2. **Minimize-Loss**: A player of this type always assumes the worst case among all possible outcomes of its choice, and wants to be as safe as possible. It selects a number that can produce the best result under the assumption of the worst-case outcome.
3. **Minimize-Regret**: It takes into account both (1) the worst-case outcome of **its choice** and (2) one best-case outcome among **all other choices**. The regret of choice is defined as (2) - (1), and it selects the number with the minimum regret.

   For all three types of players, they will select the smallest number if there is a tie.

Consider the following example, where A and B has 3 numbers each.

(A, B)	B = 8	B = 14	B = 18
A = 2	(1, 4) → Δ = (-1, -4)	(1, 11) → Δ = (-1, -3)	(1, 9) → Δ = (-1, -9)
A = 12	(6, 4) → Δ = (-6, -4)	(9, 11) → Δ = (-3, -3)	(2, 3) → Δ =(-10, -15)
A = 14	(11, 5) → Δ = (-3, -3)	(14, 14) → Δ = (0, 0)	(14, 9) → Δ = (0, -9)

1) If A seeks “Maximize-Gain”, then it will select 14 because it can achieve the best result(Δ = 0) if B selects 14 or 18.
2) If A seeks “Minimize-Loss”, then it will select 2 because, in this way, it can guarantee the loss to be at most 1 regardless of B’s choice.
3) If A seeks “Minimize-Regret”, then it will select 2 because the regret in this case(= 1) is the smallest. (The regret of each choice is 1, 10, and 2, respectively)

Given two players, each having a strategy among the above three, determine the choices they will make and return the state after one battle. Implement this in `pair<multiset<int>, multiset<int>> player_battle(string type_a, multiset<int> a, string type_b, multiset<int> b)`

- Input
  - `type_a`, `type_b`: `std::string`, one of “`Maximize-Gain`”, “`Minimize-Loss`”, “`Minimize-Regret`”. Each specifies the strategy type of the corresponding player.
  - `a`, `b`: `std::multiset<int>` that shows the numbers each player has. The sizes of sets are between 1 and 10 (inclusive), respectively, and integers inside are positive and less than 10^6.
- Output
  - `std::pair<std::multiset<int>, std::multiset<int>>` that shows the numbers of each player after one battle.
  - The first element of the pair is for player a, the second one for b.

Question 1-4: Player vs. Player [5 Marks]

Objective: Implement `pair<multiset<int>, multiset<int>> player_vs_player(string type_a, multiset<int> a, string type_b, multiset<int> b)` in main.cpp

Description: One battle is not enough for players. They want to fight until there is nothing left to do. From now on, we will define a battle between two players as consecutive matches done until it converges.

- In each match, both players select the card and have one battle by the same rule as question 1-3.

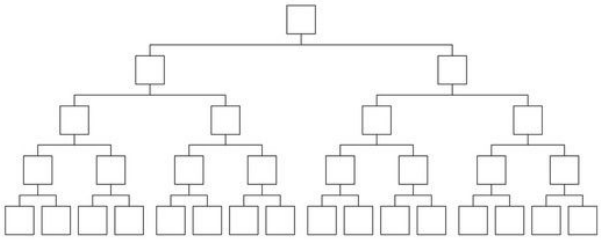
- They repeat matches until the state(numbers of both players) does not change anymore.

Implement this in `pair<multiset<int>, multiset<int>> player_vs_player(string type_a, multiset<int> a, string type_b, multiset<int> b)`. Input and output format are the same as question 1-3.

Question 1-5: A Tournament [12 Marks]

Objective: Implement `int tournament(vector<pair<string, multiset<int>>> players)` in main.cpp

Description: Why only two players? In this problem, we will make a tournament that up to 16 players can participate in. Here, the definition of the battle between two players is the same as question 1-4.



- The number of players is between 1 and 16, inclusive. Each player has an integer ID.
- There are several rounds. Players are eliminated in every round, and it continues until there is only one player left in the tournament.
- In each round, players stand side by side in a line. In other words, they are arranged in a 1D list. This should be in the ascending order of players' IDs.
- Then we make a typical tournament:
    - The 1st and 2nd players compete with each other, and the winner goes to the next round. The 3rd with 4th, and so on.
    - If the number of players is odd, the last player will proceed to the next round without having a battle.
    - A match result of a player is the sum of all the numbers it has after the battle. A player with a bigger match result is considered a winner.
    - If there is a tie, the one with a smaller index is considered a winner. (e.g. If 1st and 2nd ties, 1st proceeds to the next round)
- In every battle of two players, they start with their initial state(=numbers each player had at the start of the tournament). Battles the players had before does not affect the starting condition.

For example, if there are 15 players in a tournament, it should be as follows.

1) The first round (15 players)
    a) Battles: (Player 0 with Player 1), …, (Player 12 with Player 13)

b) Player 14 proceeds to the next round
2) The second round (8 players)
3) ...

Implement this tournament algorithm in `int tournament(vector<pair<string, multiset<int>>> players)`

- Input
  - `std::vector<std::pair<std::string, std::multiset<int>>>` that contains information of players in a tournament. The size of the vector is between 1 and 16 (inclusive).
  - One `pair<string, multiset<int>>` describes one player. The first element of the pair is a strategy type of the player. The second one contains numbers the player has. The size of the set is between 1 and 10 (inclusive).
  - The player's ID is defined as the 0-based index in a vector. (e.g., the first player has an ID of 0)
- Output
  - The ID of the player

Question 1-6: Steady Winner [5 Marks]

Objective: Implement `int steady_winner(vector<pair<string, multiset<int>>> players)` in main.cpp

Description: Now, we will repeat tournaments to check who wins the most times. Let's say we have n players in a tournament. Then we will make a total of n tournaments as follows.

- A tournament of [Player 0, Player 1, Player 2, …, Player (n-1), Player n]
- A tournament of [Player 1, Player 2, …, Player (n-1), Player n, Player 0]
- A tournament of [Player 2, …, Player (n-1), Player n, Player 0, Player 1]
- …
- A tournament of [Player n, Player 0, Player 1, Player 2, …, Player (n-1)]

That is, we will make all tournaments from all *rotated* sequences of players. Return the player who won (the 1st prize) the most. If there is a tie, return the player with a smaller ID.

Note: Player IDs of this function has nothing to do with those of each tournament. Each tournament should be done by considering the ID of each player as the index of the player in the given list. It is this function that should track players' identities in each tournament and count who won the most.

Implement this in `int steady_winner(vector<pair<string, multiset<int>>> players)`. Input and output format are the same as question 1-5.

Problem 2. Course Registration System [Java; 50 Marks]

Objective: A course registration system based on a mileage bidding, not a first-come-first-served basis.

Description: Unlike Seoul National University, many universities run their course registration systems based on a mileage bidding. The idea can be briefly summarized as follows.

1. This system provides a certain amount of mileage (e.g., 72) to each student every semester. The student can place a bid for a course with a certain amount of mileage. The bid amount will be decided based on how much he wishes to take the course.
2. At a certain point, course registrations will be confirmed based on all students' bids. Students who bid higher mileages have the priority for the course registration.
3. You will implement key functions of the course registration system as guided by this document.

Note :

1. Please start with the skeleton code and test data given in the java folder. Import the code and data in the Intellij IDEA as we practiced in the lab classes (using "Project from Existing Sources" or "Import Project").
2. In this question, you will implement the methods in Server.java, which is located in the src/target directory. Do NOT change the signatures of the given functions, but you can add/modify other parts.

Directory Structure
src/ ├─ datastructure/ │ ├─ ErrorCodes.java │ ├─ Bidding.java │ ├─ Course.java │ ├─ Pair.java │ └─ Config.java ├─ target/ │ └─ **Server.java** // Implement this └─ TestCases.java // Test cases are in this file.

3. There are five java classes (ErrorCodes, Bidding, Course, Pair, Config) under the data structure package. The specific use of each class will be explained in the sub-questions. These classes determine the format of the output, and thus, you SHOULD NOT modify these five classes.
4. Feel free to add or modify other java files.
5. There are example test cases in the TestCases class. You can run them from their main function. It provides you with the basic test cases for each sub-question. However, it is encouraged to add more test cases to check the correctness of your code.

Problem 2.1. Search Course Information [16 Marks]

Objective: Implement the `List<Course> search(Map<String, Object> searchConditions, String sortCriteria)` method in the Server class.

Description: The method returns the list of courses matching the search conditions (given in the Map) following the sorting criteria (given as the String).

- Inputs:
  - The first parameter `Map<String, Object> searchConditions` can describe three different search conditions as below.

key	value type	value description
“dept”	String	The department providing the course
“ay”	Integer	The academic year for the course
“name”	String	Name of the course

    - If the map is null or empty, then it means there are no specific search criteria. In this case, you should return the entire courses sorted by the sorting criteria.
    - For the “dept” and “ay” criteria, the values should match exactly to pass the search conditions.
    - The “name” criteria is specially handled. For instance, when the search string is “Computer Engineering”, you should consider two keywords “Computer” and “Engineering” split by the space. Then, you should find the courses that contain both keywords, i.e., “Computer” and “Engineering” in the name. To generalize this, you first need to find multiple keywords split by the space from the search string, then find the courses containing all keywords.
    - If multiple conditions co-exist, you should search for the courses that match all the conditions.
    - Assume that the keys are one of the three strings, i.e., “dept”, “ay”, “name”.
  - The second parameter sortCriteria describes the sorting criteria of the returned `List<Course>`. It can have the following four values.

sortCriteria	Description
“id”	Sort by the course id in the ascending order.

“name”	Sort by the course name in the dictionary order. If the names are equal, sort by the course id in the ascending order.
“dept”	Sort by the department name in the dictionary order. If the department names are equal, sort by the course id in the ascending order.
“ay”	Sort by the academic year in the ascending order. If the academic years are equal, sort by the course id in the ascending order.

        - Assume that course id is unique for all the courses.
        - If sortCriteria is null or empty String, sort by the course id.
        - Assume that the sortCriteria string is one of “id”, “name”, “dept”, “ay”.
- Outputs:
    - `List<Course>`: The list of the class Course instances satisfying the given search condition with the elements sorted by the given criteria.
        - Note: use the Course class in the skeleton code.

Note:

1. To solve this problem, you first need to load all the course information using an appropriate data structure. Use the Course class (in the datastructure package) to store the information of the course (you should not modify this class as described above).

Class Course
- courseName : public String - college : public String - department : public String - academicDegree : public String - instructor : public String - location : public String - courseId : public int - academicYear : public int - credit : public int - quota : public int
- public Course(int courseId, String college, String department, String academicDegree, int academicYear, String courseName, int credit, String location, String instructor, int quota) - public String toString() - public boolean equals(Object obj)

2. Course information is given as files. They are organized as the following directory structure. The java directory include the example data following the directory structure below. Each text file contains attributes of the course separated by a vertical bar (see the table below for more details).

Directory Structure
data/ ├─ Courses/ │ └─ 2020_Spring │ ├─ Engineering │ │ ├─ **11.txt** │ │ ├─ **3.txt** │ │ └─ **…(Course ID).txt** │ ├─ Fine Arts │ │ ├─ **5.txt** │ │ ├─ **8.txt** │ │ └─ **…(Course ID).txt** │ ├─ Humanities │ │ └─ **…(Course ID).txt** │ └─ …(College Names) ├─ Users/ └─ Users_backup/

	Format	Example
File Path	data/Courses/2020_Spring/(College Name)/(Course ID).txt	data/Courses/2020_Spring/Humanities/12.txt
File Content	(department)|(academic degree)|(academic year)|(course name)|(credit)|(location)|(instructor)|(quota)	Aesthetics|Bachelor|4|Topics in French Aesthetics|3|014-207|Milne, Peter W|40

- Assume that each attribute does not include a vertical bar.

Problem 2.2 Placing a Bid [16 Marks]

Objective: Implement the following two methods in the Server class:

1) `int bid(int courseId, int mileage, String userid)`
2) `Pair<Integer, List<Bidding>> retrieveBids(String userid)`

Description: The bid method allows a student to place a bid for a course while the retrieveBids method returns the information of all placed bids. Note: we will ask you to implement the `bid` method first and the `retrieveBids` method later, but you may want to implement them in

reverse order after reading this question.

Firstly, implement the following bid method.

`int bid(int courseid, int mileage, String userid)`

- Inputs:
  - `int courseid`: Course ID that the user wants to place a bid.
  - `int mileage`: The amount of mileage the user wants to bid.
  - `String userid`: User ID to specify the user.
- Outputs:
  - `int`: Error code specifying the status of the method call.

ErrorCode	**Description**
SUCCESS	The bidding is successful.
USERID_NOT_FOUND	The given user ID does not exist in the system.
NO_COURSE_ID	The given course ID does not exist in the system.
OVER_MAX_MILEAGE	The sum of previously bid mileages and the current bid mileage exceeds the maximum allowable mileage. Note that the maximum mileage is defined as a static variable MAX_MILEAGE in the Config class.
OVER_MAX_COURSE_MILEAGE	The given mileage exceeds the maximum allowable mileage per course. Note that the maximum mileage per course is defined as a static variable MAX_MILEAGE_PER_COURSE in the Config class.
NEGATIVE_MILEAGE	The mileage is a negative integer.
IO_ERROR	IOException is thrown during the execution of the method. Handle this only if necessary. This will not be tested.

- The ErrorCode class (in the skeleton code) defines constant integer values matching to various types of errors. Use the class appropriately to return the error code.
- When multiple errors occur, return the error code with the lowest value.

Note:

1. Each user is assigned a maximum amount of mileage to be spent in total. The maximum mileage is defined in the Config class (as a static variable MAX_MILEAGE = 72).
2. There is maximum allowable mileage for each bid, which is defined in the Config class (as a static variable MAX_MILEAGE_PER_COURSE = 18).
3. If the user already has a bid for the same course, you should replace the existing bid with the new amount.
4. If the user bids 0 mileage for the previous bid course, then the previous bid should be canceled.

5. If the user bids 0 mileage for the new course, then ignore the bid.
6. The bid information should be stored in files (not in the memory only). The bid information should not be removed even if the server is stopped and restarted.
7. More specifically, the bid information of a user should be stored in a file (named “data/Users/(User ID)/bid.txt”), as described in the tables below.
8. Assume that a valid student has a directory named with his/her ID under the Users directory, and has the file bid.txt in it. If the corresponding directory for a student ID does not exist, it means that no such student exists. In such a case, the USERID_NOT_FOUND error should be returned. You may want to consider defining User class to manage User information.
9. We provide resetUserDirs() to clear all user directories and reload the default user information. You can freely use this method in TestCases.java to make appropriate test cases.

Directory Structure
data/ ├─ Courses/ ├─ Users/ │ ├─ 2010-22221 │ │ └─ **bid.txt** │ ├─ 2012-22221 │ │ └─ **bid.txt** │ └─ ...(UserID) └─ Users_backup/ └─ (Backup of the /data/Users)

	Format	Example
File Path	data/Users/(User ID)/bid.txt	data/Users/2018-22233/bid.txt
Content	(course_id)|(mileage) (course_id)|(mileage) (course_id)|(mileage) (course_id)|(mileage) ...	10|17 9|15 8|15 1|1 3|2

- The bids saved in the bid.tex don’t need to be sorted.

Secondly, implement the following retrieveBids method.

```
Pair<Integer,List<Bidding>> retrieveBids(String userid)
```

- Inputs:
    - `String userid`: User ID to specify the user.
- Outputs:
    - `Pair<Integer,List<Bidding>>`: The key is the error code (defined in ErrorCode class). The value is the list of previously placed bids. The list does not need to be sorted.

ErrorCode	Description
SUCCESS	Successfully retrieved the user's bids.
USERID_NOT_FOUND	The given user ID does not exist in the system.
IO_ERROR	IOException is thrown during the execution of the method. Handle this only if necessary. This will not be tested.

- When multiple errors occur, return the error code with the lowest value.

Note:

1. Use the Bidding class (described in the table below) and Pair class defined in the datastructure package to generate the output. Refer to the skeleton code for the Pair class (it is very simple!).

class Bidding
- courseId : public int - mileage : public int
- public Bidding(int courseId, int mileage) - public String toString() - public boolean equals(Object obj)

Problem 2.3. Clearing Bids [18 Marks]

Objective: Implement the following two methods in the Server class.

1) `boolean clearBids()`
2) `Pair<Integer,List<Course>>retrieveRegisteredCourse(String userid)`

Description: The clearBids method determines which students will be registered for all courses based on the placed bids at the moment.

`boolean clearBids()`

- Outputs:
    - boolean:
        - **false**, if IOException is thrown during the execution of the method.

- **true**, otherwise.

The clearing logic is as follows.

1. The basic rule is that students who bid larger mileage to a course will be registered for the course, if the quota is limited.
2. The quota of each course should have been retrieved when you loaded course information for Problem 2.1.
3. If multiple students bid the same mileage for a course and all of them cannot fit into the quota of the course, the student who placed a smaller number of bids in total has the priority. If this second criteria cannot break the tie, a student with the preceding user ID (in the dictionary order. e.g. 2012-12345 precedes 2012-12435) gets the priority.
4. The confirmed registrations (i.e., who is registered for which courses) should be stored in files. Similar to the bids, the information of the confirmed registrations should not be removed even if the server is stopped and restarted.
5. Feel free to create a file to store the confirmed courses for a user under the corresponding user directory. You can freely decide the file format.
6. All stored bids should be removed.
7. Similar to the bids, you can use resetUserDirs() to clear all user directories and reload the default user information. You can freely use this method in TestCases.java to make appropriate test cases.

Secondly, implement the retrieveRegisteredCourse method.

`Pair<Integer,List<Course>> retrieveRegisteredCourse(String userid)`

- inputs:
  - String userid: User ID specifying the user.
- outputs:
  - `Pair<Integer,List<Course>>`: The key is the error code. The value is the list of the Course instances that are confirmed to be registered. The list does not need to be sorted.

ErrorCode	Description
SUCCESS	Successfully returned the user's registered courses.
USERID_NOT_FOUND	The given user ID does not exist in the system.
IO_ERROR	IOException is thrown during the execution of the method. Handle this only if needed. This will not be tested.

Note:

- Assume that `retrieveRegisteredCourse` is called only after `boolean clearBids()` is called.