NATIONAL UNIVERSITY OF SINGAPORE
SCHOOL OF COMPUTING

Practice Paper 3 for Practical Assessment
CS1010J — Programming Methodology

Time Allowed: 1.5 hours

INSTRUCTIONS TO CANDIDATES

1. This assessment paper consists of 2 exercises on 6 pages.
2. This is an **OPEN BOOK** assessment. You may bring in any printed material and a calculator, but **not** other electronic devices such as laptops, iPads or thumb-drives.
3. In line with the university rules, any form of communication with other students, or the use of unauthorised materials is considered cheating and you are liable to the disciplinary action.
4. The first 10 minutes of the assessment is reserved for you to read questions and design algorithms. You are not allowed to type your programs in this period of time.
5. You are to use **DrJava** or **VS Code** to write your programs. No other IDEs are allowed.
6. You may assume that all input data are valid; hence no input validation is needed.
7. Once finish, you may submit your programs to **CodeCrunch** for automatic grading.
8. You are **NOT** allowed to use any Java language syntax not covered in weeks 1-9 (e.g., string functions or recursion).

ALL THE BEST!

Exercise 1: Check Array [40 marks]

Given an array of distinct positive integers, perform the following two tasks.

1. Report the number of **Armstrong Numbers** in the array.

   An Armstrong number is a positive integer that equals to the sum of cube of its digits.

   For example, 153 is an Armstrong number because the sum of cube of its digits ($1^3 + 5^3 + 3^3$) equals to 153. Other examples of Armstrong numbers include 1 (= 1^3), 370 (= $3^3 + 7^3 + 0^3$) and 407 (= $4^3 + 0^3 + 7^3$).

2. Report the second largest integer in the array.

   For example, if the given array is {5, 2, 1, 3, 6}, the second largest integer is 5.

In the given skeleton program **CheckArray.java**, you are supposed to complete the following three methods:

- **`boolean isAN(int num)`** that returns true if **`num`** is a Armstrong number, or false otherwise.
- **`int countAN(int[] arr)`** that returns the number of Armstrong numbers in the given array.
- **`int getSecondLargest(int[] arr)`** that returns the second largest integer in the given array. You may assume that **`arr`** contains at least two distinct integers.

Take note of the following coding restrictions:

1. You are **not** allowed to change the given **`main()`** method or method headers of other methods.
2. You are **not** allowed to change the given array **`arr`**, create additional array or write additional methods.

Two sample runs are shown below with user's input shown in **bold**.

Sample run #1

```
Enter the number of elements: 5
Enter elements: 5 3 1 2 6
Number of Armstrong number = 1
Second largest element = 5
```

Sample run #2

```
Enter the number of elements: 3
Enter elements: 5 2 370
Number of Armstrong number = 1
Second largest element = 5
```

Exercise 2: Play Chess [60 marks]

The game of chess is played by two players. One player is referred to as the ***White*** player and the other player as the ***Black*** player, depending on the color of the pieces that they choose.

In a chess tournament, each player plays **two** games with every opponent, with one game as the White player and the other game as the Black player. For each game played, a player scores 1 point if he wins, 0 point if losses, or 0.5 point if the game is drawn.

We use a two-dimensional *n*n* array, **`scores`**, to store the game results for a chess tournament with *n* chess players (indexed from 0 to *n*-1) such that **`scores[a][b]`** is the score for the game between player **`a`** (as the White player) and player **`b`** (as the Black player).

The following example shows the game results of 5 players.

Player	0	1	2	3	4
0	0	0.5	1	0	0.5
1	0	0	0.5	0	0.5
2	0.5	1	0	1	0
3	1	1	0.5	0	0.5
4	1	0.5	1	0.5	0

Table 1. Example game result of 5 players

The above table can be analyzed in the following way.

- Each row represents the game results of player **`a`** (as the White player) against every other player (as the Black players). So, for instance, **`scores[0][2]`** is 1, which indicates that player 0 (as the White player) has beaten away player 2 (as the Black player).
- Following the clue, we can easily understand that column 0 indicates the game results of player 0 as the Black player against 4 other players as the White players. For instance, **`scores[3][0]`** is 1, which means player 3 (as the White player) has beaten away player 0 (as the Black player). Hence, player 3 scores 1 point from this game and player 0 scores 0.
- Of course, a player cannot fight against himself and therefore all the diagonal slots are filled in with 0.
- The total scores player 0 will receive is 3.5 (2 points gained as the White player and 1.5 points gained as the Black player).

<u>Task 1:</u>

You are required to complete the following two methods in the given skeleton program:

1. **`int gamesWin(double[][] scores, int p)`** that takes a two-dimensional square matrix, **`scores`**, as input, and returns the number of games that player **`p`**

wins in the chess tournament. For example, player 3 wins 4 games in the example shown in Table 1 in the previous page (2 as the White player and 2 as the Black player).

2. **`double[] totalScore(double[][] scores)`** that takes a two-dimensional square matrix, **`scores`**, as input, and returns a one-dimensional array that stores the total score of each player. That is, suppose **`total`** is the returned array, then **`total[a]`** is the total score gained by player **`a`** in all the games he has played. Table 2 below shows the returned array for the example given in Table 1 in the previous page.

Array Index (Player ID)	Array Element (Score)
0	3.5
1	2.0
2	3.5
3	5.5
4	5.5

Table 2. Total scores of 5 players

Task 2:

Between two players A and B, we say that player A ***beats*** player B if:

1) Player A wins at least one game against player B, **and**
2) Player B doesn't win any game against player A.

Player A is considered to be ***stronger*** *than* player B if **any** of the following conditions hold:

1) The total score of player A is more than the total score of player B.
2) The total scores of player A and player B are the same and player A beats player B.
3) The total scores of player A and player B are the same, and neither of the players beats the other. However, the total number of games won by player A is more than the total number of games won by player B.

If neither player is stronger than the other, then player A is said to be *as **competent** as* player B.

For example, following the example given in Table 1 and Table 2,

- Player 0 (with a total score 3.5) is stronger than player 1 (with a total score 2.0).
- Player 0 is stronger than player 2 because they have the same total scores and player 0 beats player 2.
- Player 3 is stronger than player 4 because they have the same total scores, and they never beat each other. However, player 3 wins 4 games while player 4 only wins 2 games.

You are required to complete the following three methods in the given skeleton program:

1. **`boolean beat(double[][] scores, int a, int b)`** that returns true if player **`a`** beats player **`b`**, or false otherwise.
2. **`boolean stronger(double[][] scores, int a, int b)`** that returns true if player **`a`** is stronger than player **`b`**, or false otherwise.
3. **`int comparePlayer(double[][] scores, int a, int b)`** that takes as inputs, a two-dimensional square matrix, **`scores`**, and two player numbers, **`a`** and **`b`**. It returns one of the following three values:
   (i). 1 if player **`a`** is stronger than player **`b`**.
   (ii). -1 if player **`b`** is stronger than player **`a`**.
   (iii). 0 if two players are competent.

Complete the skeleton program **PlayChess.java** for the above two tasks. Take note of the following additional instructions/restrictions:

1. You may assume there are at least 2 players in a chess tournament.
2. **`main()`**, **`readScores()`** and **`printScores()`** methods are complete and given. You are **not** allowed to change them.
3. You are **not** allowed to change the method headers of other given methods.
4. You are **not** allowed to change the **`scores`** array in all the methods you write.
5. Your methods may invoke each other and you may create additional arrays or write additional methods as necessary.

Two sample runs of the program are given below with the user's input shown in **bold**. For your convenience of testing, test inputs are also appended to the back of the skeleton program.

Sample run #1

```
Enter the number of players: 5
Enter the scores:
  0  0.5    1    0  0.5
  0    0  0.5    0  0.5
0.5    1    0    1    0
  1    1  0.5    0  0.5
  1  0.5    1  0.5    0
Player  Score
  0      3.5
  1      2.0
  2      3.5
  3      5.5
  4      5.5
Enter ID of the two players to compare: 1 0
Player 0 is stronger than player 1
Enter ID of the two players to compare: 2 0
```

```
Player 0 is stronger than player 2
Enter ID of the two players to compare: 3 4
Player 3 is stronger than player 4
```

Sample run #2

```
Enter the number of players: 4
Enter the scores:
  0 0.5 0.5 0.5
0.5   0 0.5 0.5
0.5 0.5   0 0.5
0.5 0.5 0.5   0
Player  Score
  0      3.0
  1      3.0
  2      3.0
  3      3.0
Enter ID of the two players to compare: 1 0
Two players are as competent as each other
Enter ID of the two players to compare: 2 0
Two players are as competent as each other
Enter ID of the two players to compare: 1 2
Two players are as competent as each other
```

Example Rough Marking Scheme for an Exercise of 60 Marks

1. Style: 10 marks
   - Write program description and student number in the program header.
   - Write a short and meaningful description for every method (except main).
   - Apply consistent indentation and good naming of variables.
2. Correctness and design: 50 marks
   - (**In actual PA**) Graders will manually go through your program and award marks method by method.
3. Additional penalties:
   - Program is not compilable: deduct 5 marks.
   - Break coding restrictions: deduct 5 - 20 marks, depending on severity.

=== END OF PAPER ===