COMP 2402 AB- Fall 2023 Assignment #5

Due: Wednesday, December 6, 23:59

Submit early and often. Late submissions (up to 12 hours) will be accepted.

Academic Integrity

You may:

- Discuss general approaches with course staff and your classmates,
- Use code and/or ideas from the textbook,
- Use a search engine / the internet to look up basic Java syntax.

You may not:

- Send or otherwise share code or code snippets with classmates,
- Use code not written by you unless it is code from the textbook (and you should cite it in comments),
- Use a search engine / the internet to look up approaches to the assignment,
- Use code from previous iterations of the course unless it was solely written by you,
- Use the internet to find source code or videos that give solutions to the assignment.

If you ever have any questions about what is or is not allowable regarding academic integrity, please do not hesitate to reach out to course staff. We will be happy to answer. Sometimes, it is difficult to determine the exact line, but if you cross it, the punishment is severe and out of our hands. Any student caught violating academic integrity, whether intentionally or not, will be reported to the Dean and be penalized. Please see Carleton University's Academic Integrity page.

Grading

This assignment will be tested and graded by a computer program (and **you can submit as many times as you like; your highest grade is recorded**). For this to work, there are some important rules you must follow:

- Keep the directory structure of the provided **zip** file. If you find a file in the subdirectory `comp2402a5`, leave it there.
- Keep the package structure of the provided **zip** file. If you find a `package comp2402a5;` directive at the top of a file, leave it there.

- Do not rename or change the visibility of any methods already present. If a method or class is public, leave it that way.
- Submit early and often. The submission server compiles and runs your code and gives you a mark. You can submit as often as you like, and only your best submission will count. There is no excuse for submitting code that does not compile or does not pass tests.
- Write efficient code. The submission server places a limit on how much time it will spend executing your code, even on inputs with a million lines. For some questions, it also limits how much memory your code can use. If you choose and use your data structures correctly, your code will efficiently execute within the time/memory limit. Choose the wrong data structure or use it the wrong way, and your code will be too slow for the submission server to grade (resulting in a grade of 0).

Submitting and Testing

For every assignment on Brightspace, you will find a URL "Assignment # submission server" (replace # with the corresponding assignment number) that will take you to the submission page. For more details on how to submit your work, follow the instructions on the document "Submission instructions". If you have issues, please post to Discord to the teaching team (or the class), and we'll see if we can help.

Warning: Do not wait until the last minute to submit your assignment. There is a hard 5-second limit on the time each test has to complete. For the largest tests, even an optimal implementation takes 3 seconds and may take longer if the server is heavily loaded.

Start by downloading and decompressing the Assignment 5 Zip File (**comp2402a5.zip**), which contains a skeleton of the code you need to write. The skeleton code in the **zip** file compiles fine. Here's what it looks like when you unzip and compile it from the command line:

```
alina@euclid:~$ unzip comp2402a5.zip
Archive:  comp2402a5.zip
  inflating: comp2402a5/AdjacencyLists.java
  inflating: comp2402a5/Algorithms.java
  inflating: comp2402a5/Graph.java
  inflating: comp2402a5/MixAndBoom.java
  inflating: comp2402a5/SnakesAndLadders.java
alina@euclid:~$ javac comp2402a5/*.java
```

This assignment consists of two main parts. For **part 1**, you need to implement the `doIt()` method in the `MixAndBoom` class. For **part 2**, you need to implement the `doIt()` method in the `SnakesAndLadders` class. Note that **since there is no `Graph` interface in Java, one has been provided for you**. However, using them is optional, i.e. you are not required to make use of the provided `Graph`, `AdjacencyLists`, or `Algorithms` classes in your solution. Also, you can make changes to them if you choose.
You can download some sample input and output files as a zip file **a5-io.zip**. Once you understand how to test your code with these input files, write your own tests that test your program more thoroughly (the provided tests are not exhaustive!)

The Assignment

Part 1: Mix And Boom [40 marks] Professor Mixiten Kaboom is a famous chemist constantly inventing new chemicals. Unfortunately, for some strange reason, some of his chemicals tend to explode when mixed with some of the other chemicals. To make things even worse, he has only two storage compartments in his lab. The only way to store the chemicals safely is to divide them between the compartments so that no two chemicals that explode when mixed are stored in the same compartment. Earlier, when the number of chemicals was small, the professor could do it easily. But now that he has accumulated so many chemicals over the years, the task has become too time-consuming. He asks for your help solving this problem. Given N chemicals and a list of chemical pairs that explode when mixed, you will have to write a program that determines if it is possible to store them using the two storage compartments.

The input begins with an integer in a single line, which is the number of chemicals the professor has. The chemicals are numbered from 1 to N. Next, there is a sequence of lines, each containing a pair of integers (between 1 and N inclusive), representing two chemicals that explode when mixed. If it is possible to store the chemicals safely, your program should output “yes”, and “no” otherwise.

Part 2: Snakes And Ladders [60 marks] A worldwide classic board game. Navigate your token from start to finish, avoid the snakes (?? Something for you to think about), and take shortcuts going up the ladders. The board is a grid of squares numbered from 1 to the final square. For our assignment, assume the board is an N-by-N matrix.

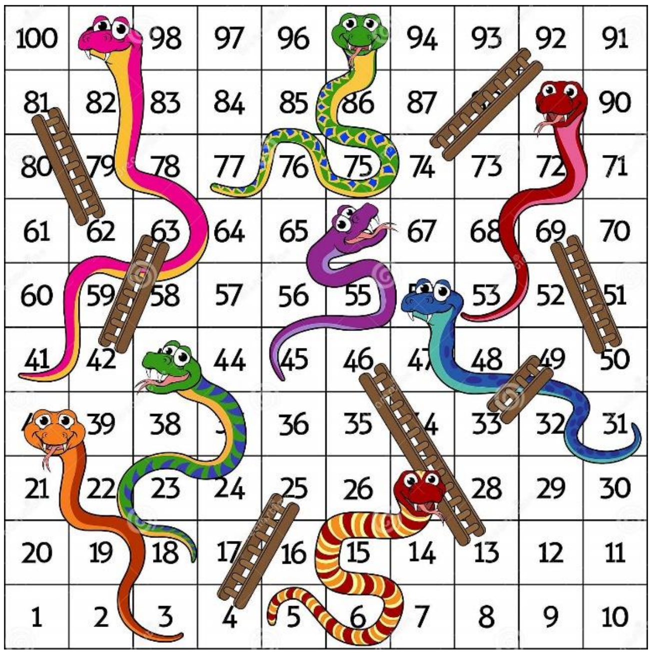
The rules of the board game "Snakes and Ladders" are simple:

- Each player starts at location 1 and takes turns to move their token forward.
- Players take turns rolling a six-sided dice to determine how many squares they will move on their turn.
- Move your token forward along the board the number of squares shown on the dice.
- If your token lands on a square at the base of a ladder, you can immediately climb the ladder to the square at the top.
- If your token lands on a square with the mouth of a snake, you must slide down to the square at the snake's tail.
- The first player to reach or exceed the final square is the winner of the game.

Imagine that you have a magical six-sided dice that rolls any number you want between 1 and 6. Given a board of size N-by-N, a configuration of snakes and ladders, and the magical six-sided dice, you must determine the **minimum** number of dice rolls you will need to go from location 1 to location N*N.

The input begins with an integer in a single line, which represents the board dimension N. Next, there is a sequence of lines, each containing a pair of integers (between 1 and N*N inclusive), representing either a ladder or a snake. A ladder is represented by a pair where the first number is smaller than the second, and the opposite for snakes. Your program should output a single integer representing the minimum number of dice rolls required to reach the location N*N.

Example of the board (taken from www.dreamstime.com):



Tips, Tricks, and FAQs

How should I approach each problem?

- You may have already figured out that this assignment is about graphs. Think about how you can pose each problem as a graph-related problem. Once you do that, you can start thinking about how your algorithm should work to solve each of these problems.
- Draw an example graph on paper and simulate your algorithm on it step by step. Try different examples and see if it works for all of them.
- Once you have the algorithms figured out, you can start implementing them. Luckily, you can use the provided `Graph` implementation to start testing. Some basic algorithms worth looking into are already provided there.

How should I test my code?

- The sample input/output files are quite simple, and they are meant to be examples that show how the input/output is formatted. Make your own input files and test your solutions with them.
- Test for both correctness and speed.
- All the tests on the server are heavily memory constrained. Be careful with how much memory your solution takes up.
- In some cases, your solution may be acceptably fast but too memory-intensive, which may lead to timeout because of how the JVM garbage collector behaves on the server.
- Hint: you can save memory usage by not storing unnecessary information when the graph is sparse.