Learning outcomes.

Design, implement and evaluate algorithms following specifications.

Basic Graph exploration algorithms

0 Preamble

Description of a programming assignment is not a linear narrative and may require multiple readings before things start to click. You are encouraged to consult instructor/Teaching Assistants for any questions/clarifications regarding the assignment. Your programs must be in Java.

1 Problem Description

Scarlet Overkill is developing a simple strategy game to keep the minions busy. The game includes a square grid gameboard with n rows and n columns. Some of the locations in the grid are occupied by "frogs". Each frog can be moved over one other frog and the direction can be up, down, right, left and diagonal. As a frog is moved to jump over another frog, the latter is removed from the gameboard. The goal of the game is to move the frogs (one frog at a time) in such a way that at the end there is one frog on the gameboard.

Consider the gameboard in Figure 1. Each location in the grid is labeled with a number (consider it the address of the location). If a location i is labeled with i^F then at that location there is a frog.

As you can imagine, the gameboard can be made larger and number of frogs on the gameboard can be made as many as needed; and there can be initial setup for which there may not be a strategy to achieve the game objective. You are "requested" by Scarlet Overkill to write a program, which can take as input the size of the gameboard and the initial setup and output the strategies (if any) that will realize the game objective.

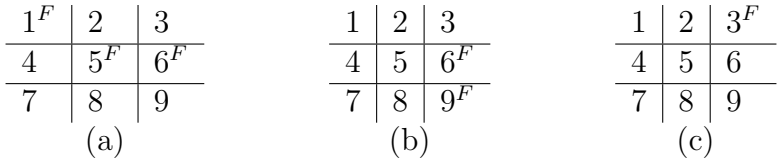

Figure 1: (a) Initial Gameboard setup; (b) Frog at location 1 is moved diagonally over frog at location 5; (c) Frog at location 9 is moved up over frog at location 6; goal achieved.

2 Encoding Description

The input to your program is a simple text file containing some numbers (integers). The first line will indicate the number of rows/columns in the gameboard. The second line will contain a sequence of numbers, in ascending order, separated by space, where each number indicates the location of a frog in the gameboard. For instance, the example gameboard in Figure 1 will be presented as

```
3
1 5 6
```

You can assume that the textfile will be correctly formatted and will contain semantically meaningful information. For instance, for a gameboard containing 3 rows/columns, there will not be any frog-location greater than 9 or less than 1; for a gameboard of containing 4 rows/columns, there will not be any frog-location greater than 16 or less than 1. There will be no two frogs placed in the same location. All locations will be integers. In short, you do not need to worry about the input file being meaningless (no input validation is needed).

You are required to implement the following classes and methods.

1. A class `GamePlan` which includes the following attributes and methods

   (a) `int goalLoc;`

   This attribute holds the location of the final frog when the game objective is realized. For the example in Figure 1, `goalLoc` is equal to 3.

   (b) `int numOfPlans;`

   This attribute holds the number of plans which lead to the location of final frog to be `goalLoc`. For instance, for `goalLoc` equal to 3, there is one plan for the example in Figure 1.

   (c) `ArrayList<int[]> plan;`

   This attribute holds one possible plan that can realize the game objective resulting in the final frog at location `goalLoc`. Note that `plan` is of type `ArrayList`. Each element in the array list describes the location of frogs (using type `int[]`). The sequence of such elements form a plan. The 0-th element in the array list describes the initial locations of all frogs and the last element in the array list describes the location of the final frog.

That is,

- `int[]`: describes the gameboard status, i.e., location of frogs.
- `ArrayList<int[]>`: describes a sequence of gameboard status. This array list will contain $x_0, x_1, \ldots, x_k$ gameboard status where a valid frog move updates the status from x_i to x_{i+1}. The x_0 is the initial set up. The final status x_k is is the location of the final frog.

For example in Figure 1, the `plan` array list for `goalLoc` $= 3$ contains the following elements

```
{1, 5, 6}  {6, 9} {3}
```

You are required to arrange the frog-locations in each `int[]` array in ascending order. *The array list is null if there are no plans.*

(d)
```
public int getgoalLoc() {
return goalLoc;
}
```

(e)
```
public int getnumOfPlans() {
return numOfPlans;
}
```

(f)
```
public ArrayList<int[]> getPlan() {
   return plan;
}
```

You can include any attributes and any helper functions as needed to correctly assign values to the above attributes of the class `GamePlan`.

2. A class `FrogGame` with a default constructor. Include the following methods as per their signature and implement their semantics.

   (a) `public void readInput(String FileName) throws IOException`
   The objective of the method is to read the input gameboard instance from a textfile (its name is the parameter) and populate the attributes as you see fit.

   (b) `public ArrayList<GamePlan> getGamePlans()`
   The objective of this method is to generate an array-list of gameplans where each element of the list is of type `GamePlan` one for each location of the final frog.
   For instance, for the initial gameboard configuration (see Figure 1), there are two different gameplans corresponding to the location of the final frog being at 7 or at 3.

3 Submission Requirements

1. You shall use default package (that is, no package at all). Though it is not recommended in practice for mid-to-large projects, organize your classes as follows:

(a) Prepare a file `FrogGame.java`. In this file you will write the specified class `public class FrogGame` and its methods, and other helper classes. You can have as many helper classes and methods as you want. For instance,

```
// import directives
import java.io.IOExceptions;
//

// primary class for this file
public class FrogGame {
   // specified attributes
   // any other attributes

   // specified methods
   // Any other helper methods

}

class GamePlan {
   // specified attributes
   // any other attributes

   // specified methods
   // Any other helper methods
}

class MyClass {
   // Any helper class needed
}
```

(b) Prepare another file with any name suitable for your setup which only contains the `main` method and declares the object of type `FrogGame` and invokes it methods. You can use the following sample file containing the main method (there is no constraint on how/what you write in this file as this file will not be part of your submission).

```
// appropriate import directives

public class HW {
   public static void main(String[] args) throws exception {

      FrogGame fg = new FrogGame();
```

```
      fg.readInput(args[0]);

      ArrayList<GamePlan> gps = fg.getGamePlans();  // get all game plans

      GamePlan gp;  // gameplans for a particular goal location
      ArrayList<int[]> plan; // actual plan

      // for printing
      if (gps == null) System.out.println("No plans"); // For no plans
      else {

       for (int i=0; i<gps.size(); i++) {
         gp = gps.get(i);      // get plans for a goal location
         System.out.println("**For Goal Location** " + gp.getgoalLoc());
         System.out.println("Number of plans: " + gp.getnumOfPlans());
         System.out.println("One of the plans: ");
         plan = gp.getPlan();  // these are all the plans

         for (int k=0; k<plan.size(); k++) {  // get each location in the plan
           for (int l=0; l<plan.get(k).length; l++) {
              System.out.print(plan.get(k)[l] + " ");
           }
           System.out.println();
         }
       }
      }
   }
}
```

If we consider the input file in `input.txt` containing

```
3
1 5 6
```

then for a correct implementation of `FrogGame` and `GamePlan` (assumming FrogGame.java and HW.java are in the same working directory), the output for
`javac HW.java; java HW input.txt` is

```
**For Goal Location** 3
Number of plans: 1
One of the plans:
1 5 6
6 9
3
```

```
**For Goal Location** 7
Number of plans: 1
One of the plans:
1 5 6
1 4
7
```

You are required to submit the `FrogGame.java` file and nothing else. (Double check that all necessary helper classes are present in this file.)

4 Some more example scenarios

Consider the following initial setup where the frogs are located at

```
6 10 11 13
```

The output (using the above driver class `HW`) is as follows for different grid sizes.

1. number of rows and columns is 4:

```
**For Goal Location** 7
Number of plans: 2
One of the plans:
6 10 11 13
11 13 14
11 15
7
**For Goal Location** 14
Number of plans: 1
One of the plans:
6 10 11 13
6 7 11
8 11
14
**For Goal Location** 9
Number of plans: 1
One of the plans:
6 10 11 13
6 7 11
3 6
9
```

2. number of rows and columns is 5 (number of rows and columns is 6):

```
No plans
```

3. number of rows and columns is 7:

```
**For Goal Location** 28
Number of plans: 2
One of the plans:
6 10 11 13
10 11 20
12 20
28
**For Goal Location** 4
Number of plans: 2
One of the plans:
6 10 11 13
10 11 20
12 20
4
```

Note that for a specific `goalLoc`, there can be more than one game plan. For instance, with initial frog locations

```
6 10 11 13
```

in a game board containing 4 rows and 4 columns, there are two game plans with final frog location at 7. The plans are

```
{6 10 11 13} {11 13 14} {11 15} {7}
{6 10 11 13} {6 9 13}   {5 6}   {7}
```

Your program can output any one of the above plans.

Also note that, if there are plans leading to different locations for the final frog, it is not necessary to order the plans with respect to the location of the final frog.

5 Suggestion: Use of Hashing in Java

In this assignment, if you want to use java `HashMap` to keep track of frog locations in the gameboard, review the `HashMap` class desciption from Java docs. One suggestion would be to use the `int[]` that captures the frog locations as the key for the map. You can develop a your own class (e.g., `class HashKey`) for the key and have an attribute (for this class) of a data type relevant to your implementation such that it includes frog locations. This class must have an `hashCode` function with return type `int`. You can use java's built-in hashCode for arrays for defining this function:

```
public int hashCode() {
    return Arrays.hashCode(<some data of type int[]>);
}
```

You will also have to write an `equals` method with input parameter of type `Object` and with return type `boolean`. This method returns true only when the input object properties (this will depend on your implementation) are equal to object's properties stored at the location identified by the hashcode of the key (frog locations).

```
public boolean equals(Object o) {
   ...
   return <true or false>
}
```

If you have not used HashMap or Map in previous classes (COMS 227, COMS 228), you can review the following for a quick start:
`https://www.baeldung.com/java-custom-class-map-key` or
`https://dzone.com/articles/things-to-keep-in-mind-while-using-custom-classes`.

6 Postscript

1. You must follow the given specifications, which includes method names, classnames, return types, input types. Any discrepancy may result in lower than expected grade even if "everything works".

2. In the given problem, there are several data structure/organization that are left for you to decide. Please do not ask questions related to such data structure/organization. Part of the exercise is to understand and assess a good way to organize data that will allow effective application of methods/algorithms.

3. You will have to think about how to model the problem into a graph-based problem and apply your knowledge of graph algorithms to address the original problem. Please do not ask questions about how to model the problem as a graph-based problem and/or what graph algorithms to use. Part of the exercise is to understand and assess a good way to represent/reduce a problem to a known problem for which we know an efficient algorithm.

4. Start reading and sketching the strategy for implementation **as early as possible**. That does not mean starting to "code" without putting much thought on what to code and how to code it. This will also help in resolving all doubts about the assignment before it is too late. Early detection of possible pitfalls and difficulties in the implementation will help in reducing the `endTime-startTime` for this assignment.

5. Both correctness and efficiency are important for any algorithm assignment. Writing a highly efficient incorrect solution *will* result in low grade. Writing a highly inefficient

correct solution *may* result in low grade. In most cases, the brute force algorithm is unlikely to be the most efficient. Use your knowledge from lectures, notes, book-chapters to design and implement algorithms that are correct and efficient.

6. Test your code extensively (starting with individual methods). Your submission will be assessed using test cases that are different from the ones provided as part of this assignment specification. Your grade will primarily depend on the number of these test cases for which your submission produces the correct result.