CS1027

ASSIGNMENT 2

Computer Science Fundamentals II

This assignment is due on Thursday, October 29th, 2020 by 11:55pm.
See the bottom of this document for submission details.

Learning Outcomes

To gain experience with

- Solving problems with the stack data type
- The design of algorithms in pseudocode and their implementations in Java
- Handling exceptions

Introduction

It's Halloween! You know what that means, right? It's time to go trick-or-treating! With all the restrictions in place this year, it's more important than ever to plan out the ideal routes. To minimize the number of people on the doorsteps at once, everyone has agreed to only trick-or-treat for so long and to take turns. The hero of our assignment, Bryan the Bold, needs your help making sure that his friends and him can collect as much candy as possible before their designated trick-or-treating time ends. He brought a map of the neighbourhoods he's going to go trick or treating in and heard that you've been studying some computer science recently that will help him plan this all out. To help Bryan the Bold out, you will create a Java program to help him navigate the neighbourhood. The program needs to determine whether Bryan or a friend can reach enough candy to fill his bag from a starting point on the map, given a maximum distance from the start point (Bryan and friends can only run so fast!).

You are given a map of the neighbourhood, which is divided into rectangular cells to simplify the task of computing the required path. There are different types of map cells:

- The starting point that Bryan wants to use
- Map cells where the candy is located,
- Map cells indicating roadblocks or other barriers
- Map cells containing pathways. There are 3 types of pathways:
    - Cross paths. A cross path located in cell L can be used to interconnect all the neighbouring map cells of L. A cell L has at most 4 neighbouring cells that we denote as the north, south, east, and west neighbours. The cross path can be used to interconnect all neighbours of L;
    - Vertical paths. A vertical path can be used to connect the north and south neighbours of a map cell; and

- Horizontal paths. A horizontal path can be used to connect the east and west neighbours of a map cell.

Map cells containing candy are allowed to function like cross paths; Bryan the Bold can sneak through backyards if need be (but not road blocks).

Figure 1 shows an example of a map divided into cells.

Each map cell has up to 4 neighbouring cells indexed from 0 to 3.

Given a cell, the north neighbouring cell has index 0 and the remaining neighbouring cells are indexed in clockwise order. For example, in Figure 1 the neighbouring cells of cell 8 are indexed from 0 to 3 as follows: neighbour with index 0 1 is cell 5, neighbour with index 1 is cell 9, neighbour with index 2 is cell 11, and neighbour with index 3 is cell 7 (note that even though there are 4 neighbours, only 2 of them are accessible: cell 5 and cell 11, because cell 8 is a vertical path and thus does not connect on the east or west sides).

Some cells have fewer than 4 neighbours and the indices of these neighbours might not be consecutive numbers; for example, cell 6 in Figure 1 has 2 neighbours indexed 1 and 2 (no 0 or 3 neighbours).

A path from Bryan (cell number 1 in the figure) to a candy (cell number 7) is the following: 1, 2, 3, 4, 5, 6, 7. A path from Bryan to another candy (cell number 10) is the following: 1, 2, 3, 4, 9, 10. Notice that the two other candies in the figure are inaccessible because of the path types of their neighbouring cells.

Valid Paths

When looking for a path the program must satisfy the following conditions:

- The path can go from Bryan, a cross path cell, or a candy cell to the following neighbouring cells:
  - A candy cell,
  - A cross path cell,
  - The north cell or the south cell, if such a cell is a vertical path, or
  - The east cell or the west cell, if such a cell is a horizontal path.

- The path can go from a vertical path cell to the following neighbouring cells:
  - The north cell or the south cell, if such a cell is either Bryan, a cross path cell, a candy cell, or a vertical path cell.
- The path can go from a horizontal path cell to the following neighbouring cells:
  - The east cell or the west cell, if such a cell is either Bryan, a cross path cell, a candy cell, or a horizontal path cell.

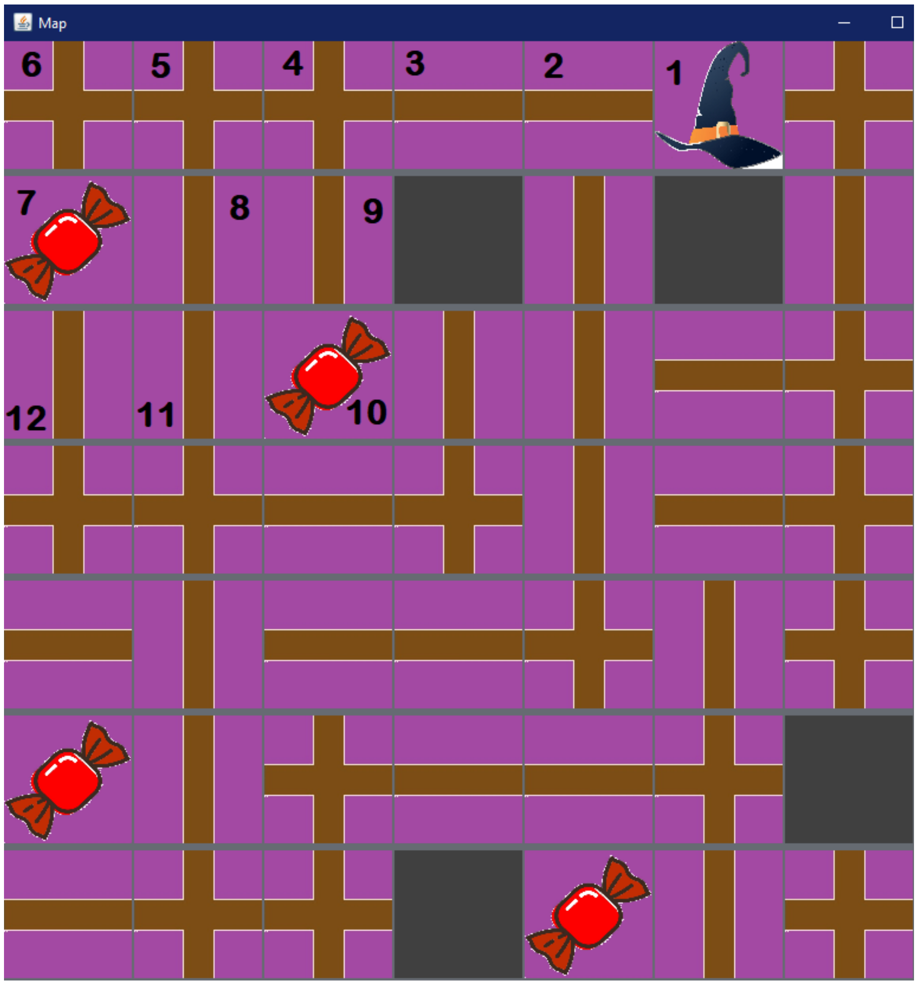


Figure 1. A sample of one of Bryan's neighbourhood maps represented as a grid of cells

Limitation:
If while looking for a path the program finds that from the current cell there are several choices as to which adjacent cell to use to continue the path, your program must select the next cell for the path in the following manner:

- The program prefers a candy cell over the other cells.
- If there is no candy cell adjacent to the current cell, then the program must prefer the cross path cell over the other cells; and
- If there is no cross-path cell, then the program chooses the smallest indexed cell that satisfies the conditions described above.

- When considering path length, adding a new cell to the stack or removing a cell from the stack counts as adding to the path length; Bryan takes time to move to and from a cell.

Provided files

The following is a list of files provided to you for this assignment. You do not need alter these files in any way and should assume we will mark them with unmodified ones. Studying these will help improve your understanding of Java.

- Map.java – This class represents the map of the neighbourhood, including Bryan's starting location and the candy houses. The public methods from this class you might use include:

  - *Map (String inputFile) throws InvalidMapException, FileNotFoundException, IOException*. This method reads the input file and displays the map on the screen. An *InvalidMapException* is thrown if the *inputFile* has the wrong format.
  - *MapCell getStart()*. Returns a *MapCell* object for the cell that Bryan starts in.
  - *Int bagSize()*. Returns the number of candies that can fit in Bryan's bag.
  - **Note:** Lines 45-47 set the delay times for showing the path. You may wish to enable line 47 when testing many maps to have it go quicker.

- MapCell.java – This class represents the cells of the map. Objects of this class are created inside the class *Map* when its constructor reads the map file. The methods that you might use from this class are:

  - *MapCell neighbourCell(int i) throws InvalidNeighbourIndexException*. As explained in Section 1, each cell of the map has up to four neighbouring cells, indexed from 0 to 3. This method returns either a *MapCell* object representing the i-th neighbour of the current cell or null if such a neighbour does not exist. Remember that if a cell has fewer than 4 neighbouring cells, these neighbours do not necessarily need to appear at consecutive index values. So, it might be, for example, that *this.getNeighbour(2)* is null, but *this.getNeighbour(i)* for all other values of i are not null. An *InvalidNeighbourIndexException* exception is thrown if the value of the parameter i is negative or larger than 3.

  - boolean methods: *isBlocked(), isCrossPath(), isVerticalPath(), isHorizontalPath(), isBryan(), isCandy(),* return true if this *MapCell* object represents a cell corresponding to a blocked path, a cross path, a vertical path, a horizontal path, Bryan, or a candy respectively.
  - *boolean isMarked()* returns true if this MapCell object represents a cell that has been marked as *inStack* or *outOfStack.*
  - *void markInStack()* marks this *MapCell* object as *inStack.*
  - *void markOutStack()* marks this *MapCell* object as *outOfStack.*

- Other classes provided:
  - *ArrayStackADT.java, CellColors.java, CellComponent.java, InvalidNeighbourIndexException.java, CellLayout.java, InvalidMapException.java, IllegalArgumentException.java.*

Classes to implement

A description of the classes that you are required to implement for full marks in this assignment are given below. You may implement more classes if you want; you must submit the code for these with your assignment.

In all these classes, you may implement more private (helper) methods as desired, but you may **not** implement more public methods. You may **not** add static methods or instance variables. Penalties will be applied if you implement these.

For this assignment, you may not use Java's *Stack* class, although reading and understanding the code there may help you better understand stacks. You also may not use any other pre-made Java collections libraries. The data structure required for this assignment is an array, described below:

ArrayStack.java

This class implements a stack using an array. The header for this class must be this:

public class ArrayStack<T> implements ArrayStackADT<T>

This class will have the following private instance variables:

- T[] stack.
  - This array stores the data items of the stack.

- int top.
  - This variable stores the position of the last data item in the stack. In the constructor this variable must be initialized to +1, this means the stack is empty.
  - Note that this is different from the way in which the variable top is used in the lecture notes.

- String sequence – used for TestSearch.java

This class needs to provide the following public methods:

- ArrayStack()
  - Creates an empty stack.
  - The default initial capacity of the array used to store the items of the stack is 5.

- ArrayStack(int initialCapacity).
  - Creates an empty stack using an array of length equal to the value of the parameter.

- void push(T dataItem)
  - Adds dataItem to the top of the stack. If the array storing the data items is full, you will increase its capacity as follows:
    - If the capacity of the array is smaller than 50, then the capacity of the array will be increased by a factor of 10.
    - Otherwise, the capacity of the array will increase by 50. So, if, for example, the size of the array is 225 and the array is full, when a new item is added the size of the array will increase to 275.

  At the end of this method, at the following. This is used in TestSearch to make sure you are following a correct path.:

```
if (dataItem instanceof MapCell) {

            sequence += "push" + ((MapCell)dataItem).getIdentifier();

        }
        else {

            sequence += "push" + dataItem.toString();

        }
```

- T pop() throws EmptyStackException
  - Removes and returns the data item at the top of the stack. An EmptyStackException is thrown if the stack is empty.
  - If after removing a data item from the stack the number of data items remaining is smaller than one fourth of the length of the array and the length of the array is larger than 50 you need to shrink the size of the array by 25;
  - To do this create a new array of size equal to half of the size of the original array and copy the data items there.
  - For example, if the stack is stored in an array of size 100 and it contains 25 data items, after performing a pop operation the stack will contain only 24 data items. Since 24 < 100/4 then the size of the array will be reduced to 75. When creating an EmptyStackException an appropriate String message must be passed as parameter.

At the end of pop(), add the following lines:

```
if (result instanceof MapCell) {
                        sequence += "pop" + ((MapCell)result).getIdentifier();
                }
                else {
                        sequence += "pop" + result.toString();
                }
```

- T peek() throws EmptyStackException.
  - Returns the data item at the top of the stack without removing it. An EmptyStackException is thrown if the stack is empty.
- boolean isEmpty().
  - Returns true if the stack is empty and returns false otherwise.
- int size()
  - Returns the number of data items in the stack.
- int length()
  - Returns the capacity of the array stack.
- String toString()
  - Returns a String representation of the stack of the form: "Stack: elem1, elem2, ..." where element i is a String representation of the i-th element of the stack.
  - If, for example, the stack is stored in an array called s, then element 1 is s[0].toString(), element 2 is s[1].toString(), and so on.

You can implement other methods in this class, if you want to, but they must be declared as private.

StartSearch.java

This class will have the following private instance variable

- Map neighbourhoodMap;
- This variable will reference the object representing the neighbourhood map where Bryan and the candies are located. This variable must be initialized in the constructor for the class, as described below.

You must implement the following methods in this class:

- StartSearch(String filename, int maxPathLength)
  - This is the constructor for the class. It receives as input the name of the file containing the description of the neighbourhood map. In this method you must create an object of the class Map (described in the provided files) passing as parameter the given input file; this will display the map on the screen.
  - Read them if you want to know the format of the input files.
- static void main(String[] args).
  - This method will first create an object of the class *StartSearch* using the constuctor *StartSearch(args[0]).*
  - *Args[0]* will be the name of a map file, *args[1]* will be the number of cells that can be traveled to during his designated trick-or-treating time.
  - If and only if a size argument is provided, your algorithm should count how many candies can be found in a path that is at most *args[1]* long.
  - Otherwise, it should count how many candies are reachable on the map.
  - When you run the program, you will pass as command line arguments the name of the input file (see following section after Java classes), and the number of cells that Bryan can travel from his starting point to find candy.
  - Your main method then will try to find a path from Bryan to the candies according to the restrictions specified above.
  - The algorithm that looks for a path from the initial cell to the destinations must use a stack and it cannot be recursive.
  - Suggestions on how to look for this path are given in the next section. The code provided to you will show the path selected by the algorithm as it tries to reach the candy cells, so you can visually verify if your program works.
  - The main method should exit by printing out the number of candies found given the maximum path length if one is given, and the number of candies discoverable if no maximum path length is given.

- MapCell bestCell(MapCell cell).
    - The parameter is the current cell.
    - This method returns the best cell to continue the path from the current one, as specified early in the limitations.
    - Refer to those priorities when coding this section.
    - If several unmarked cells are adjacent to the current one and can be selected as part of the path, then this method must return one of them in the following order:
        - A candy cell
        - A cross path cell.
        - If there are several possible cross path cells, then the one with the smallest index is returned.
        - A vertical path cell or a horizontal path cell with smallest index
        - Read the description of the class MapCell below to learn how to get the index of a neighbouring cell.
        - If there are no unmarked cells adjacent to the current one that can be used to continue the path, this method returns null.
    - Your program must catch any exceptions that are thrown by the provided code.
    - For each exception caught, an appropriate message must be printed.
    - The message must explain what caused the exception to be thrown.

You can write more methods in this class but they must be declared as private.

Algorithm for Computing a Path

Below is a description for an algorithm that will look for a path for Bryan to the candies. It will be helpful to understand the algorithm deeply before attempting to implement it. There are better algorithms for finding candies given a maximum path length, but they may not pass all the tests. Implement the algorithm as described to make sure your code passes the test cases.

You must use a stack to keep track of which cells are in the path, and it cannot be recursive. Writing your algorithm in pseudocode will make coding it in Java easier and is helpful to show to the TAs and the instructors if you need help.

- Initialize the number of found candies to 0.
- Create a stack based on the command line inputs
- Get the start cell (Bryan) using the methods of the supplied class Map.
- Push the starting cell into the stack and mark the cell as inStack.
- You will use methods of the class MapCell to mark a cell.
- While the stack is not empty and Bryan's bag is not full perform the following steps:
    - Peek at the top of the stack to get the current cell.

    - Find the best unmarked neighbouring cell (use method bestCell from class StartSearch to do this).
    - If one exists:
        - Push the neighbouring cell into the stack and mark it as inStack.
        - (If max path length given) Increase the path length counter
        - If the best neighbouring cell is a candy, increase the number of candies found.
    - Otherwise, since there are no unmarked neighbouring cells that can be added to the path, pop the top cell (and increase the path length counter) from the stack and mark it as outOfStack.
- While the stack is not empty perform the following:
    - Pop the top cell from the stack and mark is as outOfStack.

Your program must print a message indicating how many candies were collected.
Note that your algorithm does not need to find the shortest path from Bryan to all the candies.

Command Line Arguments

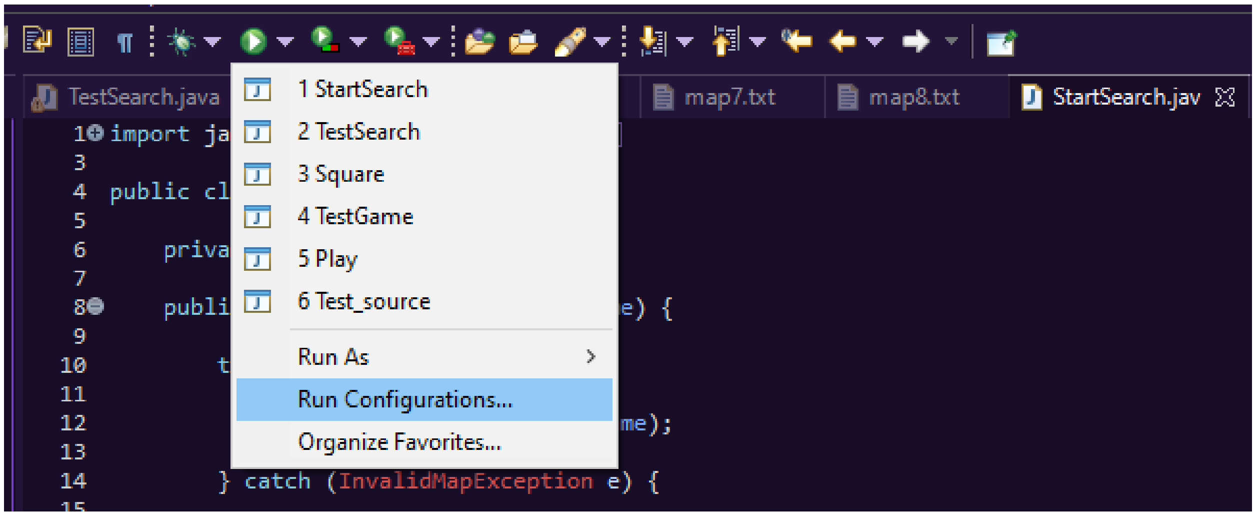
Your program must read the name of the input map file from the command line.
You can run the program with the following command:

java StartSearch nameOfMapFile (optional)maxPathLength

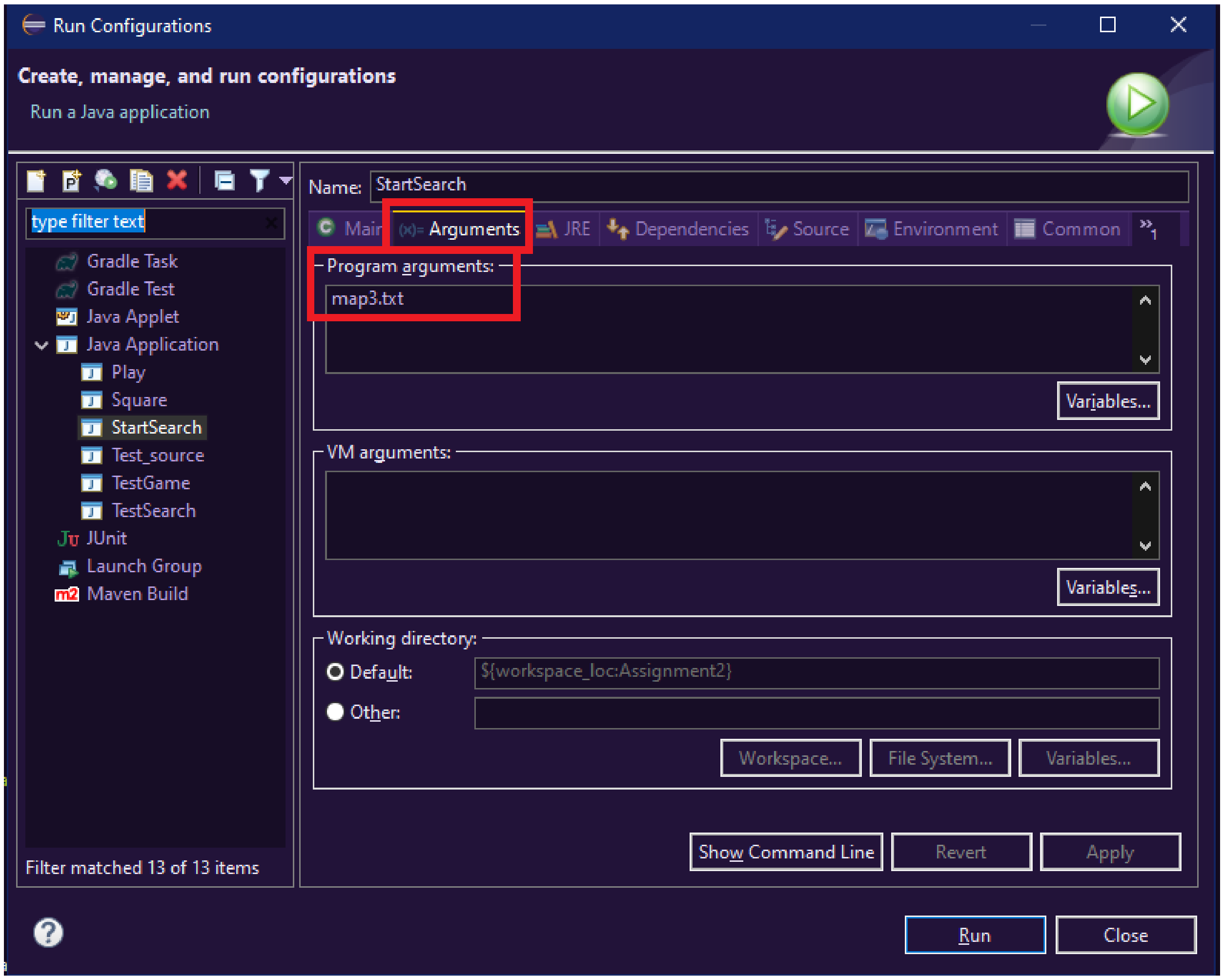
Where nameOfMapFile is the name of the file containing the desert map, and maxPathLength is the longest path that Bryan can take to find candies.
You can use the following code to verify that the program was invoked with the correct number of arguments:

```
public class StartSearch {
    public static void main (String[] args) {
        if (args.length < 1) {
            System.out.println("You must provide the name of the input file");
            System.exit(0);
        }
        String mapFileName = args[0];
        int maxPathLength = Integer.parseInt(args[1]);
        ...
```

You can access run configurations as follows:



```
import ja
public cl
    priva
    publi                e) {
        t
                         me);
    } catch (InvalidMapException e) {
```



Name: StartSearch

Program arguments: map3.txt

VM arguments:

Working directory:
Default: ${workspace_loc:Assignment2}
Other:

In the text box for 'Program arguments', you can type your arguments in. Ex: map3.txt above, or map3.txt 20 for map 3 with a maximum of 20 steps.

Image Files and Sample Input Files Provided

- You are given several image files that are used by the provided Java code to display the
- different kinds of map cells on the screen.
- You are also given several input map files that you can use to test your program.
- In Eclipse put all these files inside your project file in the same folder where the src folder is.
- Do not put them **inside** the src folder as Eclipse will not find them there.
- If your program does not display the map correctly on your monitor, you might need to move these files to another folder, depending on how your installation of Eclipse has been configured.

Marking notes

Marking categories

- Functional specifications
  - Does the program behave according to specifications?
  - Does it produce the correct output and pass all tests?
  - Are the class implemented properly?
  - Are you using appropriate data structures?
- Non-functional specifications
  - Are there Javadocs comments and other comments throughout the code?
  - Are the variables and methods given appropriate, meaningful names?
  - Is the code clean and readable with proper indenting and white-space?
  - Is the code consistent regarding formatting and naming conventions?
- Penalties
  - Lateness: 10% per day
  - Submission error (i.e. missing files, too many files, ZIP, etc.): 5%
  - "package" line at the top of a file: 5%

Remember you must do all the work on your own. Do not copy or even look at the work of another student. All submitted code will be run through cheating-detection software.

Submission (due Thursday, October 29th, 2020 at 11:55pm ET)

ASSIGNMENT 2

Rules

- Please only submit the files specified below. Do not attach other files even if they were part of the assignment.
- Submit the assignment on time. Late submissions will receive a penalty of 10% per day.
- Forgetting to submit is not a valid excuse for submitting late.
- Submissions must be done through Gradescope
- Submitting the files in an incorrect submission page or website will receive a penalty.
- Assignment files are NOT to be emailed to the instructor(s) or TA(s). They will not be marked if sent by email.
- You may re-submit code if your previous submission was not complete or correct, however, re-submissions after the regular assignment deadline will receive a penalty.

Files to submit

- ArrayStack.java
- StartSearch.java
- Any custom classes used.

Grading Criteria

- Total Marks: [20]
- Functional Specifications:
  - [4] ArrayStack.java
  - [4] StartSearch.java
  - [8] Passing Tests
- Non-Functional Specifications:
  - [1] Meaningful variable names, private instance variables
  - [1] Code readability and indentation
  - [2] Code comments