COMP 3190 Assignment 1

Material covered:

- solving simple problems using symbolic AI;
- state-space search;
- uninformed and informed search strategies.

Notes:

- Submit your solution in a single **zip file**. Other file formats will receive a mark of zero.
- Your zip file must contain a directory structure as follows. At the top level, place a **`README.txt`** file containing any notes to the marker about your solution, and nothing else. Then, place two subfolders `Q1` and `Q2` with your complete solutions for each of those two questions. Ensure the `main()` methods are in files named `A1Q1.java` and `A1Q2.java`.
- Your assignment submission **must** run correctly as downloaded. Do not use further subfolders (e.g. no Java packages). The marker **will not** reorganize or rename files to make it run.
- For programming questions, hand in only your source code. Do not hand in data files or compiled `.class` files unless requested in the question.
- Do not generate excessive (debug) output, beyond what is requested in the question.
- Remember to provide a signed Honesty Declaration to your instructor by the due date, and to follow the terms of the declaration.

Question 1 (15 marks): Aztec Math

Write a Java program that will solve Aztec Math Puzzles using search. An Aztec math puzzle is a number triangle using the values 1-9 where each number (except the top) is aligned with exactly two others below it. The number above must be computed as the sum, difference, product, or quotient of the two numbers below, with the operands in either order. Furthermore, all the numbers within a row must be different. The puzzle is initially populated with given numbers, and blanks to be filled that meet the constraints described. Some puzzles will have unique solutions.

Your solution must read in the initial state of the puzzle **from standard input** (`System.in`) as a single line of comma-separated integers, with 0 representing blanks in the puzzle.

Do not read more than a single line of input.

Running your program once should accept a single input, print the initial state, and solve the problem four times, using the following strategies:

- an informed search strategy of your choice;
- depth-first search;
- breadth-first search; and
- iterative deepening.

After applying each of the strategies, print out the solution (if any) and the number of states that were examined. Note that this number is not an absolute metric, just a way of comparing your searches. Count it in a consistent fashion across the four strategies. For IDS, also print the depth of the tree where a solution was found.

For the informed search, make sure you describe (briefly in your output and in greater detail in your `README`) how you informed your search. In your `README`, compare the performance of the informed search to the uninformed searches in terms of the number of states considered.

Here are some example inputs:

```
3,0,0,0,9,0,7,0,0,8
6,0,0,4,0,8,0,0,0,0,0,0,0,1,0,2,0,7,0,0,4
2,0,0,0,0,0,0,0,0,0,0,0,0,0,0
```

Question 2 (25 marks): Mission: Possible?

A secret agent. A world filled with bombs. Can she disarm them all... before it's too late?

Write a complete Java program that will read a text-based world on a square grid **from standard input** (`System.in`), in the format shown below. It will print out a copy of the initial world. It will calculate and execute a plan. It will print a copy of the world showing the path of the agent after the plan is executed, as a count of the number of times each location was visited (1-9, overflowing to a-z). It will also print out the total movement cost of the plan, the number of states it searched to find the solution, and the number of bombs disarmed and not disarmed.

The first line of input will be the number of rows and columns in the world. The world will always be surrounded by walls. The following symbols are used:

- `@`: the initial position of the secret agent (there will only be one of these)
- terrain by movement cost:
  - : (space) cost = 1
  - `.`: cost = 2
  - `:`: cost = 3
  - `!`: cost = 4
  - `$`: cost = 5
- `#`: wall (cannot move through)
- bombs: `A` to `Z` where the timer is set to 10 * the ordinal value of the letter (`A` is 1, and therefore has timer value 10)

For example, the input may look like this:

```
10,20
####################
#              @$$A#
#                  #
#                  #
#     .........    #
#     .:::::::.    #
#     .:!!!!!:.    #
# ####.:!$$$!:.    #
#   C#.:!$F$!:.    #
####################
```

Here are the rules of the world:

- The agent can move 8-connected (horizontally, vertically, or diagonally).
- The total cost of a path is the sum of the movement cost of the terrain of every location on the path.
- The terrain underneath the initial position of the agent and all bombs is movement cost 1 (blank space).
- Bombs explode if the agent cannot find a plan that reaches them with a total cost less or equal to the timer value. For example, the bomb in the upper-right corner will explode if the agent moves through the cost 5 terrain `$$` (5 + 5 + 1 > 10). Similarly, if the agent attempts to reach bomb F before C, C will explode since no path that includes A and F (or even just F) is less than or equal to 30.

The goal is first to maximize the number of bombs disarmed. If a bomb cannot be reached in time, there is no reason to include it in your plan. The secondary goal is to minimize the total cost of the plan. All of this has to be achieved within a reasonable amount of time: the program will not be allowed to run for more than 15 seconds.

Here is an example of a solution to the problem above:

```
####################
#              @$$1#
#     111111111122 #
#    1             #
#   1 .........    #
#  1  .:::::::.    #
# 2111.:!!!!!:.    #
#2####1:!$$$!:.    #
# 221#.1111$!:.    #
####################
bombs disarmed: 3
bombs exploded: 0
cost of plan: 45
states examined: 10000
```

Make no assumptions about the dimensions of the world, it may be any size. It may also be any configuration, and contain any number of bombs. Some may not be reachable in time, or at all.

The code will be tested on a variety of inputs of a variety of sizes. Note that much of your mark on this question will be proportional to the the quality of the solution: you want to achieve the most successful and lowest cost solution you can in a reasonable amount of time, while examining the minimal number of states. The "best" solution will require an informed search.