CSCI 2110 Computer Science III
Data Structures and Algorithms
ASSIGNMENT NO. 1

Date Given: Tuesday, September 19, 2017

Date Due: Tuesday, October 3, 2017 at 11.55 p.m. (5 minutes to midnight)

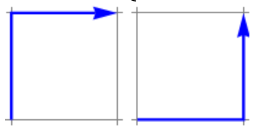
Submission on dal.ca/brightspace

Welcome to your first assignment. This assignment has two programming questions, and they are meant to hone in your skills with object-oriented programming, algorithm design and large data sets.

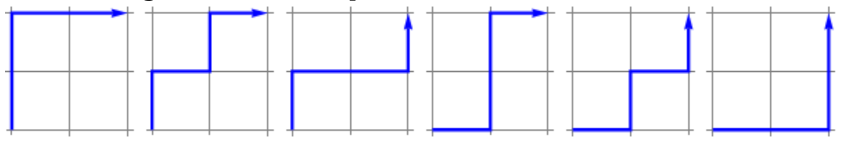
Programming Question 1- Generating Lattice Paths: Generating all possible paths to go from one point to another in a lattice (that is, a grid) is a popular problem in mathematics and has several applications, for example, in game design.

Consider a 1x1 grid. To get to the top right corner starting from the bottom left corner, there are exactly 2 possible paths. Here we assume that one can move either up (North direction) or right (East direction) only (diagonal movement is not allowed).

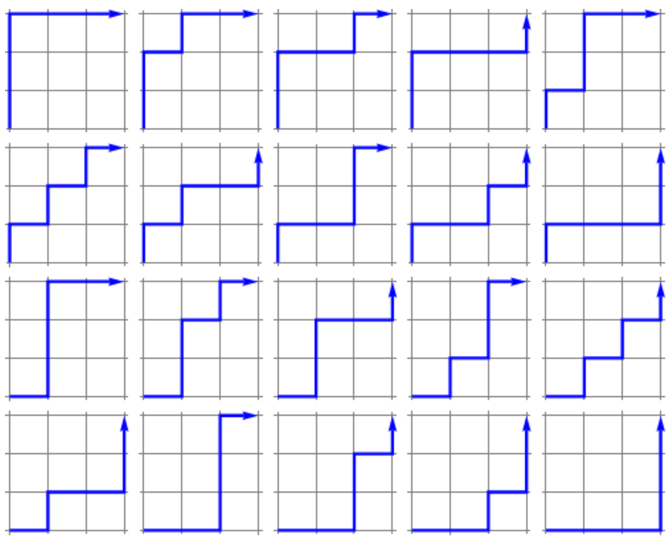
1x1 Grid (Paths are North-East and East-North)



For a 2 x 2 grid, there are 6 paths as shown below:



For a 3 × 3 grid, there are 20 paths:



For a 4x4 grid, there are 70 paths, and so on. The number of paths increases rapidly with the grid size, and is given by the formula

$$\frac{(2n)!}{n!^2}$$

where n is the lattice size.

a) Write a program that generates all possible paths of a lattice of a given size and stores the paths as Strings in an array list.
For example, if n =1, the array list should store [NE, EN] where N is north and E is east.
A sample screen dialog would be:

```
Enter the lattice size: 1
The lattice paths are: [NE, EN]
```

Note: You are not actually creating a lattice or a 2-d array in your program. You just need to think how to generate all the possible paths and design the algorithm accordingly.

b) Extend your program to generate all possible paths in a n x n grid if diagonal moves within a square (from bottom left to top right) are also allowed. For example, in a 1 x 1 grid, the possible paths would be:
[NE, EN, D] where D is the diagonal hop.

c) Determine the execution time to generate the lattice paths for both cases using the following code template:

```
long startTime, endTime, executionTime;
startTime = System.currentTimeMillis();

//include the code snippet (or call to the method) here

endTime = System.currentTimeMillis();
executionTime = endTime - startTime;

//display executionTime
```

Submit your code (.java files) for a) and b), sample outputs, and a text file containing the execution times for sample outputs.

Programming Question 2:

Pattern Matching: String matching or pattern matching, simple as it may seem, is an important problem in computer science with numerous applications, especially in the fields of bio-informatics, DNA sequencing and analysis.

Exercise 2(A).

Start off with this simple exercise. Write a program (**name it StringMatch.java**) that prompts the user to enter two strings and tests whether the second string is a substring of the first string. The number of times the substring appears should be displayed. ***Do not use the indexOf method in the String class (you are actually implementing a version of the indexOf method from scratch)***. Instead, you would run through the string character by character using charAt and check if the substring exists. If the second string is not a substring of the first string, then you should display something like "No matches – substring (aaa) was not found in (bbb)".

The following sample runs give you an idea what the program is supposed to do. You can use these as your test cases.

```
Enter a string: mississippi
Enter a substring: ssi
Substring ssi was found in mississippi 2 times

Enter a string: mississippi
Enter a substring: p
Substring p was found in mississippi 2 times
```

The following example shows that if the substring is longer than the first string, it should report no match.

```
Enter a string: wonder
Enter a substring: wonderful
No matches – substring wonderful was not found in wonder
```

The following example shows that your algorithm must be case-sensitive (that is, do not ignore case).

```
Enter a string: mississippi
Enter a substring: M
No matches – substring M was not found in mississippi
```

Exercise 2(B)
In this exercise, you will apply the string-matching algorithm that you developed in Exercise 2(A) to DNA analysis. The text file in the link given next to the assignment link contains 20,000 base pairs of DNA sequences from the human genome. (Source- Open source project: https://genome.ucsc.edu)

Download the file by clicking on the link next to the Assignment link.

Write a program (**name it DNAMatch.java**) that reads the text file, prompts the user for a test string and determines the matches of this string in the DNA sequence. Your program should also report the execution time for string matching (do not measure the execution time for reading the file).

Here are sample runs of the code (Note: these are for illustration purposes only. You may get different results. Also, try your program on different string sequences).

```
Enter the filename to read from: DNASequence.txt
Enter the test string: CC
Substring CC was found 1459 times

Execution time for string matching: 41 ms

Enter the filename to read from: DNASequence.txt
Enter the test string: GGGCT
Substring GGGCT was found 34 times
Execution time for string matching: 4 ms


Enter the filename to read from: DNASequence.txt
Enter the test string: G
Substring G was found 4871 times
Execution time for string matching: 349 ms


Enter the filename to read from: DNASequence.txt
Enter the test string: &^$%#()Q
```

```
Substring &^$%#()Q was not found.
Execution time for string matching: 20 ms
```

Run your program a number of times for different substrings and report your results in a table similar to the one below:

Substring	Number of matches found	Time taken (ms)

Submit StringMatch.java, DNAMatch.java, and the results table.