CPSC 331 — Fall 2013

Assignment #1 — On the Correctness of Simple Algorithms and Programs

1 About This Exercise

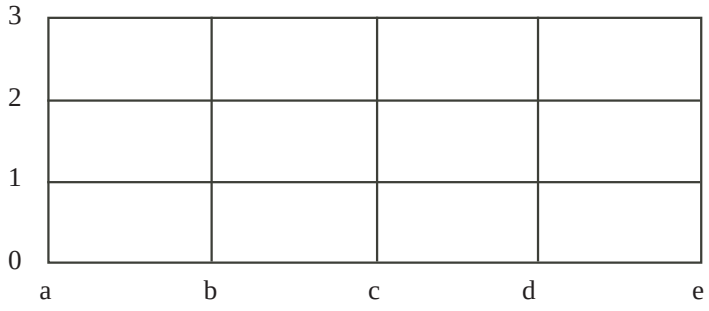
This assignment concerns material covered in

- lectures up to and including the lecture of Wednesday, September 18, and and used to solve problems in tutorials up to Thursday, September 24;
- reading exercise #1, and
- the first five Java exercises.

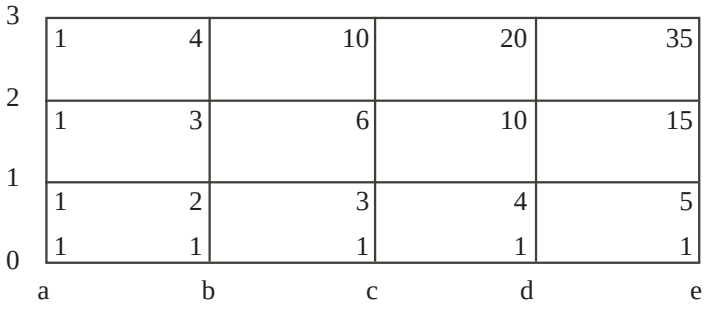
This assignment can be completed by groups of up to three students and it is due by 11:59pm on Friday, October 4. Please see the main "Assignments" course page for information about requirements for the reports and code required for assignments as well about how to submit your assignment using Blackboard.

2 The Problems To Be Solved

Suppose the streets in a city are arranged in a grid and that you wish to travel to a location that is northeast of the location where you are now. For example, you might wish to travel from the location marked $(a, 0)$ to the location marked $(e, 3)$ on the following map.



Suppose as well that you do not want to backtrack, so that you can only move north or east. In this case there are only finitely many ways to reach an intersection to the north and/or east of the place where you start. In particular, the number of routes you can follow from $(a, 0)$ to reach an intersection is shown, near the intersection, on the following copy of the map.



Indeed, if n and m are nonnegative integers and *Routes*(n, m) is the number of ways to travel n blocks to the east and m blocks north, then

$$\mathit{Routes}(n,m) = \begin{cases} 1 & \text{if } n = 0 \text{ or } m = 0 \text{ (or both)}, \\ \mathit{Routes}(n-1,m) + \mathit{Routes}(n,m-1) & \text{if } n > 0 \text{ and } m > 0 \end{cases}$$

— because if $n > 0$ and $m > 0$ then there $\mathit{Routes}(n-1,m)$ ways to travel this number of blocks (in each direction) in which the last street you travel on is headed *east*, and there $\mathit{Routes}(n,m-1)$ ways to travel this number of block (in each direction) in which the last strew you travel on is headed *north*.

In this question you will consider *two* algorithms (and corresponding Java programs) that solve the following computational problem.

Problem:	**Route Counting**
Precondition:	A pair of nonnegative integers, n and m, are given as input.
Postcondition:	The value $\mathit{Routes}(n,m)$ (as defined above) is returned as output.

1. Write a simple recursive algorithm, `routes1`, that solves the "Route Counting" problem. (This should be easy!)

2. Note that $n + m$ is a bound function for the recursive algorithm that you wrote — or, least, it should be! Write a proof of the correctness of your algorithm. (Consider standard induction on the bound function $n + m$ — the resulting proof should not be hard to write.)

3. Now write a Java program `Routes1` — that is part of the package `cpsc331.assignment1` — to implement your recursive algorithm. When executed, this should read the input n and m from the command line and it should return $Routes(n, m)$ as output if n and m are both nonnegative integers. It should display the error message

```
Sorry!  You must provide exactly two nonnegative integers as input.
```

if the inputs are invalid (that is, one or both are negative or not an integer at all, if there are not enough command-line inputs, or if there are too many of them).

A few sample runs of the program should therefore look like the following.

```
> java cpsc331.assignment1.Routes1  4  3
> 35

> java cpsc331.assignment1.Routes1  5  0
> 1

> java cpsc331.assignment1.Routes1  3
> Sorry! You must provide exactly two nonnegative integers as input.
```

Your program should include (at least) two methods as described below.

- The main method should check whether two integer inputs have actually been received. If this is not the case then it should display the error message listed above and terminate.

  Otherwise it should call the method that is described next, either catching any `IllegalArgumentException` thrown by this method (and then displaying the given error message in this case, too) or printing out the value that has been returned.

- The desired number of possible routes should be calculated by a method with signature

  ```
  public BigInteger count ( int n, int m )
  ```

  This method should throw an `IllegalArgumentException` if either (or both) of its inputs is negative. Otherwise it should use your algorithm from Question #2 to compute (and return) *Routes*(n, m).

  As the signature for this method indicates you should use Java's `BigInteger` data type to represent this value.

You may include other methods (that these call) as well, but this is not necessary. You *must* include both of the above, because these methods will be tested when your work is graded.

Unfortunately you should find that — like the first program to compute the Fibonacci numbers that we considered in class — this method is too slow to be useful.Indeed, it might not be advisable to run it if the sum of n and m is greater than 20.

4. Write another algorithm, `routes2`, that solves the "Routes Counting Problem" more efficiently when n and m are both positive. In this case the algorithm should declare and fill in the entries of an $(n+1) \times (m+1)$ integer array `A`: For $0 \leq i \leq n$ and $0 \leq j \leq m$, `A`$[i][j]$ should eventually be set to have value $Routes(i, j)$.

   The algorithm should never need to call itself recursively — because the values `A[i-1][j]` and `A[i][j-1]` should already be filled in (and set to $Routes(i-1, j)$ and $Routes(i, j-1)$, respectively) at the point in the computation when you wish to set the value of `A[i][j]`, for any pair of positive integers i and j.

   Instead, the array should use several `while` loops (almost certainly including one or more "inner" loops that are nested inside an "outer"' one to fill in the entries of the array `A`.

   You should also give loop invariants and bound functions for each of the loops in the algorithm. The loop invariants should be correct — but also complete enough that they could be used to establish the partial correctness of this algorithm. Unfortunately this means that some of the loop invariants (especially for the inner loop(s)) will need to be rather long, because they will need to include information about the inputs, several variables, and detailed information about which entries of the array `A` have already been filled in.

   You should also include describe

   - an assertion that holds at the *beginning* of the execution of the loop,
   - an assertion that holds at the *beginning* of *every* execution of the *body* of the loop,
   - an assertion that holds at the *end* of *every* execution of the *body* of the loop, and
   - an assertion that holds at the *end* of the execution of the loop.

   In each case, you should find that the assertion should state that the loop invariant of the corresponding loop is satisfied — and that it should include a *small* amount of additional information about the possible value of a variable in your algorithm.

   You may either include these loop invariants, bound functions and additional assertions as inline documentation or you may state them separately after the algorithm. If you do the latter, please make it clear *which* loop each loop invariant (*etc.*) corresponds to.

5. Now write *another* program, "`Routes2`," to compute $Routes(n, m)$ as well. The expected inputs and required output for this program are the same as for `Routes1` and this should also be part of the package `cpsc331.assignment1`.

   Once again, this program should include *two* methods as follows.

   - The main method should check whether two integer inputs have actually been received. If this is not the case then it should display the error method that has been described here and it should then terminate.

     Otherwise it should call the method that is described next, either catching any exception thrown by this method (and then displaying the given error message in this case, too) or printing out the value that has been returned.

     While the *behaviour* of this should be identical to that of the main method for your first program, `CountRoutes1`, the *implementation* will necessarily be a little different because of differences in the method(s) that it calls.

   - The desired number of possible routes should be calculated by a method with signature

     ```
     public static BigInteger count ( int n, int m )
     ```

     This method should throw an `IllegalArgumentException` if either (or both) of its inputs is negative. Otherwise it should compute the desired value by applying the algorithm (that fills in the entries of an array `A`) that gave when answering the previous question.

     The loop invariants, bound functions and additional assertions that you were asked to supply in the previous questions should be included as inline documentation for this method.

   You should find that this second program is *much* faster than the first. It will eventually fail as well — because it runs out of storage space.

   For full marks this program should include at least one assertion in which the method `count` from your `Routes1` program is called in order to check to be sure that the output returned (when n and m are both nonnegative integers) is correct. For a *small* number of bonus marks you may include additional assertions (that are executed when java is executed with the `-ea` flag) in order to make the program test itself more extensively.

   You should be notice (if you *do* include at least one assertion, as described above, that an execution of `Routes2` with assertion checking enabled is at least as slow as the execution of `Routes1` on the same input.