CPSC 331 — Assignment #1

Proving the Correctness of Simple Algorithms — and Implementing Them as Java Programs

About This Assignment

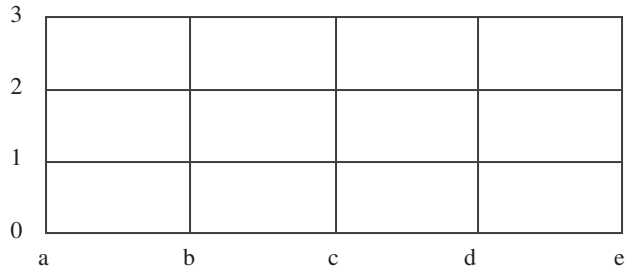
This assignment can be completed by groups of up to three students.[1] It is not necessary for these students all to be in either the same tutorial or lecture section. It is due by 11:59 pm on Friday, October 9.

Please read and make sure that you understand the following ***before*** you start work on this.

- Assignment Do's and Don'ts: Information about what is allowed — and what is not allowed — when working on assignments in this course.
- Expectations about the Quality of Students' Work on Assignments
- How to Submit a CPSC 331 Assignment

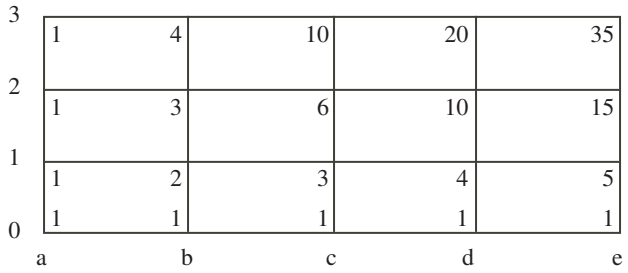
Problems To Be Solved

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 $(\mathrm{a}, 0)$ to the location marked $(\mathrm{e}, 3)$ on the following map.



[1] Submissions from groups of four or more students will not be marked.

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 $(\mathrm{a}, 0)$ 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

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

— because if $n > 0$ and $m > 0$ then there *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 *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.

Route Counting

Precondition: A pair of nonnegative integers, n and m, are given as input.
Postcondition: The value *Routes*(n,m) (as defined above) is returned as output.

A simple recursive algorithm `sRoute` that can be used to solve this problem is shown in Figure 1 on page 3.

1. Prove that the function $f(\mathtt{n},\mathtt{m}) = (\mathtt{n}+\mathtt{m})$ is a ***bound function*** for this recursive algorithm.
2. Prove that the `sRoute` algorithm correctly solves the "Route Counting" problem.
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

```
int sRoute(int n, int m) {
1. if ((n == 0) or (m == 0)) {
2.   return 1
   } else {
3.   return sRoute(n − 1, m) + sRoute(n, m − 1)
   }
}
```

Figure 1: A Recursive Solution for the "Route Counting" Problem

```
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 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 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.

Note: Two files that will help you to ***test*** your programs, before you submit them for assessment, are available.

- `testRoutes1.sh`: A shell script with tests for this program's main method. Instructions for the use of this kind of program are included in Java Development Exercise #6.
- `TestSNumberRoutes.java`: JUnit tests for the inner method of this program, which is also part of the `cpsc331.assignment1` package. Instructions for the use of this kind of program are included in Java Development Exercise #5.

4. For non-negative integers n and m let $T_{\text{Routes1}}(n, m)$ be the number of numbered steps that are executed by the algorithm in Figure 1, given n and m as inputs.

   Write a ***recurrence*** for $T_{\text{Routes1}}(n, m)$.

   Note: Please make sure that you have reviewed the definition of a ***recurrence*** and made sure that this is what you are giving. If you have answered this question correctly then you should be able to use your recurrence to confirm that $T_{\text{Routes1}}(0, 0) = T_{\text{Routes1}}(0, 1) = T_{\text{Routes1}}(1, 0) = 2$ and that $T_{\text{Routes1}}(1, 1) = 6$.

5. It is follows by its definition that $T_{\text{Routes1}}(n, m) \geq 0$ for all $n, m \in \mathbb{N}$.

   Use your recurrence, and the above observation (which you may use without proof) to show that

   $$T_{\text{Routes1}}(n, n) \geq 2^n$$

   for all $n \in \mathbb{N}$.

   Note: This has been designed to be easy to prove: You can actually "throw a lot away" (that is, use correct bounds that are correct, but much weaker than can actually be established) when proving this, and write a short and simple proof. However, you should consider using the case "$n = 1$" as a special case, along with the case "$n = 0$".

It follows that the algorithm in Figure 1 is too slow to be useful for at least some (reasonably sized) inputs. It turns out that the problem here is that — even though there is a reasonably small number of (distinct) instances of the problem that must be formed and recursively solved, the same instances get formed and solved, and over again.

```
int sRoute2(int n, int m) {
 1.  if ((n == 0) or (m == 0)) {
 2.    return 1
     } else {
 3.    Set R to be an ((n + 1) × (m + 1)) array of integer's.
 4.    integer i := 0
 5.    while (i ≤ n) {
 6.      integer j := 0
 7.      while (j ≤ m) {
 8.        if ((i == 0) or (j == 0)) {
 9.          R[i][j] := 1
           } else {
10.          R[i][j] := R[i − 1][j] + R[i][j − 1]
           }
11.        j := j+1
         }
12.      i := i+1
       }
13.    return R[n][m]
     }
}
```

Figure 2: A Faster Algorithm for the "Route Counting" Problem

An optimization strategy that you will learn more about in CPSC 413, ***dynamic programming***, can be used to develop another algorithm. Consider the algorithm shown in Figure 2.

When answering the next question, you may assume that the following is a ***loop invariant*** for the *outer* loop in this algorithm (at lines 5–12) without proving it.

Loop Invariant for Outer Loop

(a) $\mathtt{n}$ and $\mathtt{m}$ are integer inputs such that $\mathtt{n} \geq 1$ and $\mathtt{m} \geq 1$.

(b) $\mathtt{R}$ is a (variable) $((\mathtt{n}+1) \times (\mathtt{m}+1))$ array of integers.

(c) $\mathtt{i}$ is an integer variable such that $0 \leq \mathtt{i} \leq \mathtt{n}+1$.

(d) $\mathtt{R}[k][\ell] =$ *Routes*(k, ℓ) for all integers k and ℓ such that $0 \leq k \leq \mathtt{i}-1$ and $0 \leq \ell \leq \mathtt{m}$.

6. State a loop invariant for the ***inner*** loop (at lines 7–11) and prove that your loop invariant is correct.

   Note: In order to be complete your loop invariant will need to be at least as long as the loop invariant for the *outer* loop — and include quite a bit (but not all) of the same information, as well information that the loop invariant for the outer loop does not include.

7. Now prove that the loop invariant for the outer loop, given above, is also correct.

8. Use this to prove that the algorithm in Figure 2 is partially correct — assuming that the process used above actually works: That is, you may assume that the loop invariant for the outer loop is always satisfied, at the beginning of an execution of the loop body, when you are proving the correctness of a loop invariant for the inner loop.[2]

9. State a bound function for the inner loop for this algorithm and show that your bound function is correct. Then use this to find an upper bound for the number of numbered steps included in an execution of the inner loop. (This upper bound might be a function of `n`, `m` and `i`).

10. State a bound function for the outer loop for this algorithm and show that your bound function is correct. Then use this to find an upper bound for the number of numbered steps included in an execution of the outer loop. (This upper bound might be a function of `n` and `m`).

11. Complete a proof that if the algorithm in Figure 2 is executed with non-negative integers `n` and `m` given as input (so that the precondition for the the "Route Counting" problem is satisfied) then the execution of the algorithm terminates — and give an upper bound, depending on `n` and `m`, for the number of numbered steps included in this execution of the algorithm.

    Note: If you have answered the above questions then you have proved that the algorithm shown in Figure 2 is correct.

12. 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 same error method as the main method for the `Routes1` program does in this case.

[2] It is possible to state and prove another "loop theorem"' concerning nested loops implying that this really does work.

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 using an implementation of the algorithm shown in Figure 2.

  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.

Note: Two files that will help you to ***test*** your programs, before you submit them for assessment, are available.

- `testRoutes2.sh`: A shell script with tests for this program's main method. Instructions for the use of this kind of program are included in Java Development Exercise #6.
- `TestFNumberRoutes.java`: JUnit tests for the inner method of this program, which is also part of the `cpsc331.assignment1` package. Instructions for the use of this kind of program are included in Java Development Exercise #5.