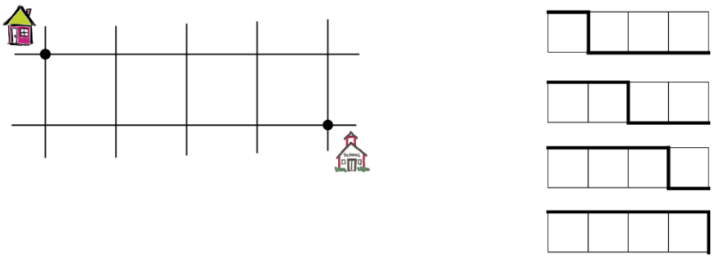
Assignment objectives

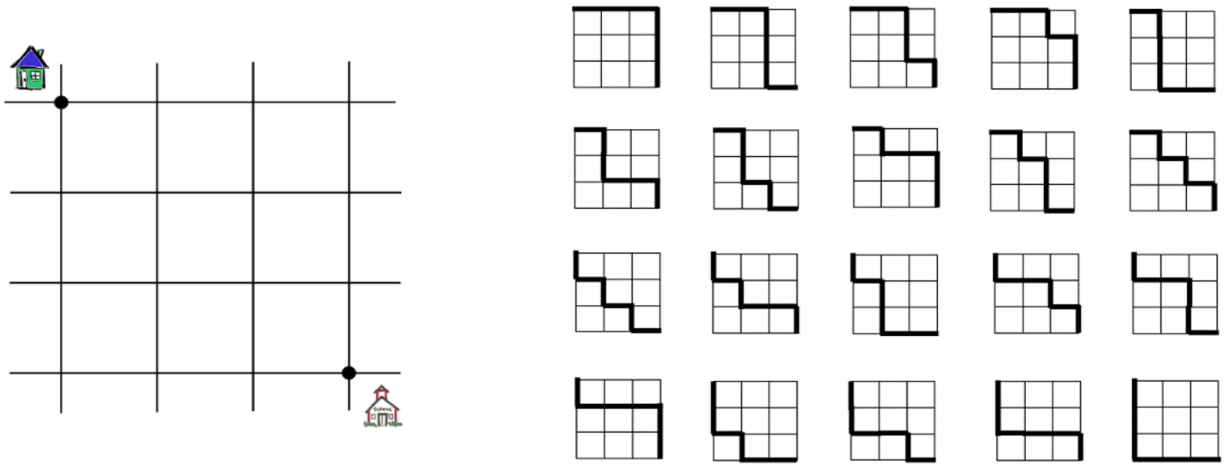
Practice the technique of *dynamic programming* and examine its time savings over straightforward recursion for applicable problems.

Introduction

Young Johnny's daily walk to school is 5 blocks, as shown in the diagram below. He has a choice of five different routes he can follow:



Young Alice lives 6 blocks from the same school, but she can walk **20** different ways:



Question: Given any two such locations on a grid of city streets, how many different paths are there to walk from the first location to the second location? Assume that all walks proceed down and to the right.

We will calculate the answer to this question two ways:

- By a straightforward recursive formula
- By a dynamic programming algorithm

Expressing the solution recursively

Define *SW(m, n)* to be the number of different ways to can walk *m* blocks down and *n* blocks over on a grid of city streets. There is no walking up or left. The starting point for a walk is

(0, 0). Then *SW(1, 4)* = 5 is the number of ways Johnny can walk to school, and *SW(3, 3)* = 20 is the number of ways that Alice can walk to school.

By definition *SW(0, 0)* is 1, because there is one and only one way to go to where you started: do nothing.

For any locations along the topmost (horizontal) street—SW(0, x)—or the leftmost (vertical) street—SW(y, 0)—there is only 1 possible path: a straight line from the starting point.

Key observation: For all the other grid points, in order to arrive there, you *must* have walked through either the point immediately above, or the point immediately to the left. Furthermore, arriving at these two previous points consists of two entirely different sets of paths. (Some of them may have had common segments, but their ending points are different.)

Thus the total number of ways you can arrive to a point is equal to the number of ways you could have arrived in the grid point immediately above *plus* the number of ways you could have arrived in the grid point immediately to the left.

Thus the function *SW(m, n)* can be expressed recursively as follows:

Base cases:

- If m = 0: *SW(0, n)* = 1 (walking *n* blocks straight to the right)
- If n = 0: *SW(m, 0)* = 1 (walking *m* blocks straight down)

Recursive case:

- If *m, n* are both > 0: *SW(m, n) = SW(m-1, n) + SW(m, n-1)*

Program design requirements

Please pay careful attention to all EXACT public identifier/method names, capitalization and spelling!

Class name: Lab6

Public method: `long SW_Recursive(int m, int n)`

Calculates and returns SW(m, n) by using the recursive definition given above. Please note that the calculations of SW and the return value are of type `long`. No console output.

You can write a recursive helper method, or you can use SW_Recursive itself recursively.

Public method: `void RunRecursive (int first, int last)`

Runs SW_Recursive in a loop, using values of m & n that are equal to each other. Also uses one of the Java timing methods to measure the running time of each call to SW_Recursive. Produces console output that looks something like this (example showing output for first=0 and last=5):

```
SW_Recursive(0,0) = 1, time is 0 ms
SW_Recursive(1,1) = 2, time is 0 ms
SW_Recursive(2,2) = 6, time is 0 ms
SW_Recursive(3,3) = 20, time is 0 ms        ← Young Alice, 20 different walks!
SW_Recursive(4,4) = 70, time is 0 ms
SW_Recursive(5,5) = 252, time is 0 ms
```

You may use your own formatting and wording of the message, but the key information (SW values and calculation time) must be present.

NOTE: While you are testing SW_Recursive() and RunRecursive(), try calculating values up to about SW(20, 20).

Public method: `long SW_DynamicProg(int m, int n)`

Calculates SW(m, n) by using dynamic programming. Please note that the calculations of SW and the return value are of type `long`. No console output.

Use an array to store values of SW(m, n) so that they can be looked-up/used in successive calculations. It's OK to use an ArrayList for this, but IMHO a plain array is the easiest, and I'm not just saying that because of my natural bias.

Public method: `void RunDynamicProg (int first, int last)`

Runs SW_DynamicProg in a loop, using values of m & n that are equal to each other. Also uses one of the Java timing methods to measure the running time of each call to SW_DynamicProg. Produces console output that looks something like this (example showing output for first=20 and last=24):

```
SW_DynamicProg(20,20) = 137846528820, time is 0 ms
SW_DynamicProg(21,21) = 538257874440, time is 0 ms
SW_DynamicProg(22,22) = 2104098963720, time is 0 ms
SW_DynamicProg(23,23) = 8233430727600, time is 0 ms
SW_DynamicProg(24,24) = 32247603683100, time is 1 ms
```

You may use your own formatting and wording of the message, but the key information (SW values and calculation time) must be present.

NOTE: While you are testing SW_DynamicProg() and RunDynamicProg(), try calculating values up to at least SW(37, 37). (Note: The last few values will fail.)

Important Note

If you examine enough of these "SW" numbers (not just the (x, x) ones, but also others), you may recognize them. You may also know that there is a simple way to calculate them directly (involving factorials). *For the above required methods, you MUST NOT calculate the numbers this way.* The purpose of this assignment is not to "write a program that produces these SW numbers"; it is to "write a program that uses/demonstrates Dynamic Programming".

Submission information

Due date: As shown on Learning Hub.

Submit the following to the drop box on Learning Hub:

- Just your Java source code (*.java file).
- File name is not important.
- Please *do not zip* or otherwise archive your code. Plain Java files only.
- Please *do not zip* or include your entire project directory.

Marking information

This lab is worth 20 points. As usual, 4-5 points are reserved for compliance with the COMP 3760 Coding Requirements.

Virtual donut 1

This part is not required; it is just for fun (and  *).*

Ensure that your program can calculate the correct value of SW(37, 37), which is 1746130564335626209832. *Do not use/modify the two required SW functions for this bonus problem*; make a new function. Make it a public method so my test program can access it. Name it whatever you want. Write me a note with your submission so that I will know to look for it.

Virtual donut 2

This part is not required; it is just for fun (and  *).*

Compute the largest SW(x, x) that you can *by any algorithm*. This does not have to be Dynamic Programming. In fact, you can calculate SW numbers *very much larger* with other non-DP algorithms.

Submit the following information:

- The answer, in a text file of its own
- The number of digits
- The amount of computer time spent on the calculation (be sure to give the time units—seconds, minutes, hours?, days???)

Only six people will earn VD2—one in each set of COMP 3760. In addition, one person will receive *another*  for being the overall winner in all sets/both campuses combined. (I.e. one person will earn TWO donuts for this bonus question.)

Virtual donut 3

This part is not required; it is just for fun (and  *).*

Compute a larger number than I have ever done. My current record is SW(37373737, 37373737). (No kidding, that really is it.)