CSCI-1200 Data Structures — Spring 2023
Lab 8 — Recursion

This lab gives you practice in the use of recursion to solve problems. All three checkpoints addressed in this lab deal with finding and counting the number of paths between points on a rectilinear grid. A starting point (x, y) with non-negative integer coordinates is provided. You are only allowed to move horizontally and vertically along the grid. Hence, from (x, y) you may move to $(x + 1, y)$, $(x - 1, y)$, $(x, y - 1)$, $(x, y + 1)$. Your goal is to return to the origin $(0, 0)$ in such a way that you **never** increase the distance to the origin. The distance is counted as the minimum number of total vertical and horizontal steps to reach the origin. In the first checkpoint the grid will be "free". In the second and third checkpoints, some of the grid locations will be "blocked" in the sense that you can not move to that point.

Stop now and play with small examples. Draw pictures of a grid. Think about the implications of the rules before you proceed with the checkpoints.

Checkpoint 1

estimate: 10-30 minutes

Did you notice that the rules prevent certain moves from occurring? What are they? If you don't get them right you will not be able to do the lab correctly. Confirm your understanding with one of the lab TAs.

Now, write a simple recursive program (from scratch) that reads a starting location, (x, y) and returns the total number of different paths back to the origin when the entire grid is "free". Two paths are different if there is at least one step on the path that is different even if most of the steps are the same. As a hint, when $x == 0$ and $y == 0$ there is 1 path, when $x == 2$ and $y == 1$ there are 3 paths, and when $x == 2$ and $y == 2$ there are 6 paths. When you're confident your program is debugged try even bigger values. For what values does the program take a few seconds to complete on your computer? If you increase these values by 1, how does it impact the running time? Is this what you expected from Big O Notation?

To complete this checkpoint show a TA your program to count *all paths* back to the origin. Be prepared to discuss the running time of your program for different input values.

Checkpoint 2

estimate: 20-40 minutes

Please download the files:

http://www.cs.rpi.edu/academics/courses/spring23/csci1200/labs/08_recursion/start.cpp
http://www.cs.rpi.edu/academics/courses/spring23/csci1200/labs/08_recursion/grid1.txt
http://www.cs.rpi.edu/academics/courses/spring23/csci1200/labs/08_recursion/grid2.txt
http://www.cs.rpi.edu/academics/courses/spring23/csci1200/labs/08_recursion/grid3.txt
http://www.cs.rpi.edu/academics/courses/spring23/csci1200/labs/08_recursion/grid4.txt

Starting from the supplied program, `start.cpp`, write a program to count the number of paths when some of the grid locations are blocked. When a location is blocked, no path can go through it. Before writing your own code, compile and run `start.cpp`. Use the files `grid1.txt`, etc. as input. These have a sequence of blocked locations, ending with the point location $(0, 0)$ (which isn't blocked, but signals the end of the input). The next location is the location for the initial (x, y). By changing this location you will be able to test your program in different ways.

To complete this checkpoint show a TA your code to find and count legal (unblocked) paths.

Checkpoint 3

estimate: 30-40 minutes

Modify your program from Checkpoint 2 so that it finds and outputs a legal path from the given (x, y) location to the origin. You should both print the sequence of coordinates of a solution path *AND* modify the `print_grid` function to draw the grid a second time with the path marked on the grid with '`$`' symbols. If there is more than one path, it does not matter which it outputs. Did you notice that all legal paths to the origin are the same length under the rules provided?

Now let's explore recursion with `gdb/lldb`. Review the handout from Lab 3, Checkpoint 3:
`http://www.cs.rpi.edu/academics/courses/spring23/csci1200/labs/03_debugging/lab_post.pdf`
Here's a table of equivalent commands in `gdb` (Linux, WSL) vs. `lldb` (Mac, Linux, WSL):
`https://lldb.llvm.org/lldb-gdb.html`

1. Compile your program with the `-g` flag, so it includes debug info, including line numbers:
   ```
   clang++ -Wall -g start.cpp -o start.out
   ```

2. Start `gdb` (or `lldb`):
   ```
   gdb ./start.out
   ```

3. Set a breakpoint at the first line of the `main` function:
   ```
   b main
   ```

4. Now run/launch the program. Note: Here's where we specify the necessary command line arguments.
   ```
   r grid2.txt
   ```

5. Place a breakpoint on the line in main where you first call your recursive function. Replace `<NUM>` with the line number in your source code.
   ```
   b <NUM>
   ```
   `gdb` will confirm with a message like this:
   ```
   Breakpoint 2 at 0xwhatever: file start.cpp, line <NUM>
   ```
   You can review your currently set breakpoints:
   ```
   info b
   ```
   If you happen to mistype the line number, you can delete the breakpoint and try again. `<BNUM>` is the breakpoint number with the incorrect line number.
   ```
   delete <BNUM>
   ```

6. Place another breakpoint at the first line of your recursive function. And then let the program run until it reaches the next breakpoint by typing:
   ```
   continue
   ```
   When `gdb` pauses, it should inform you of the current program line number.

7. Now let's step into your recursive function to get a closer look at recursion. Let the program run until it enters the first recursive function call:
   ```
   continue
   ```
   Inspect the data in the function arguments and the grid of booleans. Print the coordinates of the current location, which are passed in as function arguments named `x` and `y` (for example).
   ```
   print x
   print y
   ```

If you are using `grid2.txt` as your grid, these values should read $x = 9$ and $y = 7$.

8. Let's navigate within our recursive algorithm using `step` and `next`. NOTE: Though similar sounding, these are two different commands in `gdb/lldb`. `step` will enter into the details of a called function and allow you to walk through its code, while `next` will fully execute one line from the current function (skipping over all of the details of the function as it's executed).

   Use the `next` command to move down until we hit our next recursive call. Once on this line, let's `step` into the function call. After you step into the function, you may want to type `continue` so we can skip to the next breakpoint. Print out the coordinates of the current location, which should be different.

   Also, let's print the boolean values from locations in the grid adjacent to the current position. For example:

   ```
   print blocked_grid
   print blocked_grid[x][y]
   print blocked_grid[x-1][y+1]
   ```

   Because `gdb` supports using basic math on these variables, we can also print the grid values above, below, to the left, and to the right of our current position.

9. Use `step` and `continue` to go further and further into the recursion. Use `backtrace` to show the function calls currently on the *call stack*, including the specific line numbers. As you walk step by step through your algorithm and print data, do the variable values and sequence of locations and function calls match your expectations? Ask a TA or mentor if you have questions about this information.

10. Finally, delete all of the breakpoints and then let the program finish without interruption.

    ```
    continue
    ```

To complete this checkpoint and the entire lab, present your modified program to a TA or mentor. Demonstrate your skills with `gdb/lldb` to print out values of adjacent positions in the blocked grids vector, to use backtrace, and navigate within recursive function calls using `next`, `step`, `continue`. Be prepared to discuss how you will use `gdb/lldb` to help debug your future Data Structures homeworks.