CS 302 – Assignment #01

Purpose: Refresh concepts regarding C++ simple I/O, functions, object oriented programing, variable scoping, and compilation/linking. Verify installation of development environment.
Due: Tuesday (1/24) → Must be submitted on-line before class.
Points: Part A → 50 pts Part B → 50 pts

Reading/References

Chapter 1, Data Structures and Algorithms

Source: www.xkcd.com/1425

Assignment:

Part A:

Given a triangle of numbers, with a depth of 4, as shown (on right), we wish to find a path from the top (3 here) to the bottom row with the *least* cost. At each step, left or right is the only option. The 'cost' is a summation of the numbers used along the way. For example, taking the outside path on the right (3, 4, 6, 5) would yield a sum of 18. Taking the path (3, 7, 2, 2) would yield a sum of 14. However, taking the path (3, 4, 2, 2) would yield a sum of 11, which is the least cost path for this triangle.

3
7 4
4 2 6
8 2 9 5

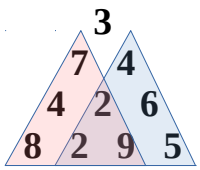
Algorithm 1

The simplest approach is just try every possible path, track, and report the lowest cost. This is referred to as a *brute-force* approach. The number of possible solutions is $2^{depth-1}$ which is 8 in this example. Since we have a binary choice at each step, we can iterate through all possibilities with an integer counter, and use the bits of the number to pick the direction left or right (for each number) as follows:

$index = index + (iteration \gg column \wedge 1)$ where iteration is the attempt number (from ***0*** to $2^{depth-1} - 1$ (inclusive) and the column would range from **0** to ***depth*-1**). However, the running sum is initialized with the triangle top (0,0), so the (***column*+1**, ***index***) element would be accessed. While running through the path, we sum up the numbers and check if the sum is smaller than the current minimum found. We retain the smallest sum or path found.

Algorithm 2

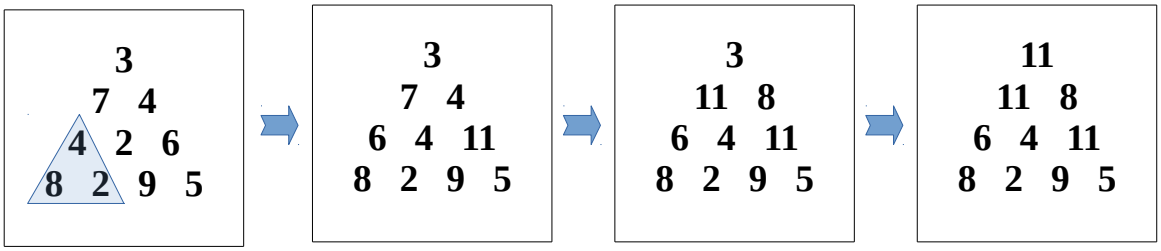
There is a more efficient, but slightly more conceptually complicated algorithm using a dynamic programming[1] approach. Dynamic programming is a method for solving a complex problems by breaking it down into a collection of simpler subproblems, solving each of those subproblems, and storing their solutions.



Standing at the top of the triangle we have to choose between going left and right. In order to make the optimal choice (which minimizes the overall sum), we would have to know the sum we can get if we go either way. So in order to answer the question we would basically have to solve the two smaller problems.

1 For more information, refer to: https://en.wikipedia.org/wiki/Dynamic_programming

We can break each of the sub-problems down in a similar way, and we can continue to do so until we reach a sub-problem at the bottom line consisting of only one number, then the question becomes trivial to answer, since the answer is the number it self. Once that question is answered we can move up one line, and answer the questions posed there with a solution which is ***number*** + *min*(***left***,***right***) for each entry in that row. For example, in the third row, first number (4), we would use 4+2 which is smaller than 4+8 and over-write the 4 with a 6. Once we know the answer to all 3 sub-problems on the next to last line, we can move up and answer the posed sub-problems by the same formula already applied. And we can continue to do so until we reach the original question of whether to go left or right. This process is shown as follows:

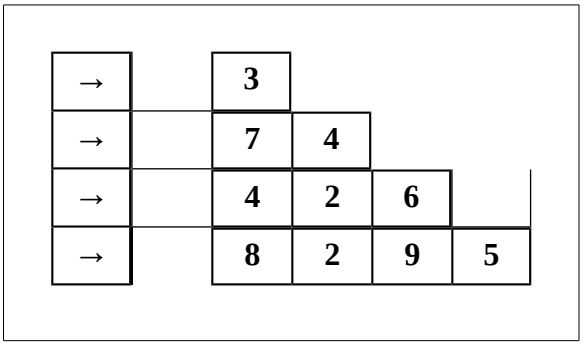


At this point, the minimum cost path is located at the top (11 in this example).

Data Structure

In order to store the triangle efficiently, we will use a dynamically allocated two-dimension jagged array. The size or depth of the triangle is the entry in the provided data file. All data files are correct, as such error checking the values is not necessary. The logical structure is shown on the right.

Note, pay very close attention to the data types.



Class Descriptions

- Triangle Path Class

  The triangle path class will implement both minimum cost triangle path finding algorithms and some support functions. A header file and implementation file will be required.

trianglePath
-depth: int
-**triangleData: int
+trianglePath()
+~trianglePath()
+readTriangleData(string): bool
+displayTriangle(): void
+triangleSumBF(int): int
+triangleSumDY(): int

Function Descriptions

- The *trianglePath()* constructor function will initialize class variables as appropriate.
- The *~trianglePath()* destructor function should free the dynamically allocated memory.
- The *readTriangleData()* function should open the file, read the triangle depth, verify the depth is between 4 and 100 (inclusive) and, if, so, dynamically create the jagged two-dimension array, read the numbers into the array, and return true. If the file does not open or there is an error with the depth, the function should display an appropriate error message and return false.
- The *dissplayTriangle()* function should display a formatted triangle (see example). This is done by displaying (depth - currentRow) spaces before each number. The output should include the depth and number of possible solutions, and then the formatted triangle values. Refer to the example for output formatting.
- The *triangleSumBF()* function should find the minimum cost for a path from top to bottom using the brute force algorithm. Note, this will require initializing the minimum variable to maximum value for that data type using the max() function from numeric_limits (in <limits>). For example,

  ```
  smallestSumSoFar = numeric_limits<int>::max()
  ```

  will set the variable accordingly. You will need to include <limits> in the includes. Since the total number of iterations will exceed the size of an integer, you will need to use a long long data type for the looping. This will mean performing the appropriate type casting. Pay very close attention to the types as getting it wrong can cause very subtle, difficult to debug errors.
- The *triangleSumDY()* function should find the minimum cost for a path from top to bottom using the dynamical programming algorithm.

You may any additional private functions if needed.

Part B:

When completed, use the provided script file to execute the program on a series of different input files. The script will write the execution times to a text file.

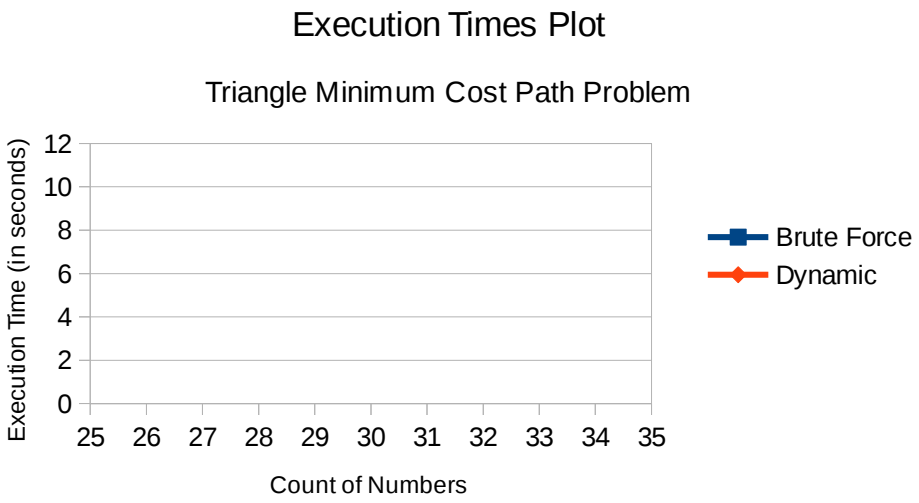
Enter the file number and execution times into a spreadsheet and create a line chart plot of the execution times for each algorithm. The results are provided in minutes and seconds format and for clarity, seconds will be entered in the spreadsheet. Refer to the example for how the plot should look.

Create and submit a write-up with an explanation not to exceed ~500 words including the following:

- Name, Assignment, Section
- Description of the machine used for obtaining the execution times (processor, RAM).
- Copy of the chart (cut-and-pasted from spreadsheet).
- Explanation of the results, comparing the algorithms
  - some comments about why the executions times were similar or were different.
- Note what the advantage of the 'jagged' array approach (as compared to a standard two-dimensional array).

Note, due to different hardware, execution times for each submittal will be different (very different). You should use a word-processor (open document, word format, or Google Docs) but submit the file version as a PDF file.

Example Plot:
Below is an example of the execution times plot (excluding the second, algorithm 2, execution times). This incomplete example is to show the appropriate format.



The final chart should be complete and show the times for both algorithms (instead of just one as shown in the example above.

Submission:
When complete, submit:

- Part A → A copy of the **source files** via the class web page (assignment submission link) by class time on or before the due date. The source files, with an appropriate *makefile*, should be placed in a ZIP folder.
- Part B → A copy of the write-up including the chart (see example). Must use PDF format. Other formats will not be accepted (and receive 0 pts).

Assignments received after the due date/time will not be accepted.

You may re-submit as many times as desired. Each new submission will require you to remove (delete) the previous submission. Make sure your program includes the appropriate documentation. See Program Evaluation Criteria for CS 302 for additional information.

Reminder: Copying code from someone else or from the net will result in a zero for the assignment and referral to the Office of Student Conduct.

Example Executions:

Below are some sample executions for the program. *Note*, the **ed-vm%** is the prompt.

```
ed-vm%  ./main -dy tri3.dat
************************************************************
CS 302 - Assignment #1

Depth: 15
Possible Solutions: 16384

               75
              95 64
             17 47 82
            18 35 87 10
           20 04 82 47 65
          19 01 23 75 03 34
         88 02 77 73 07 63 67
        99 65 04 28 06 16 70 92
       41 41 26 56 83 40 80 70 33
      41 48 72 33 47 32 37 16 94 29
     53 71 44 65 25 43 91 52 97 51 14
    70 11 33 28 77 73 17 78 39 68 17 57
   91 71 52 38 17 14 91 43 58 50 27 29 48
  63 66 04 68 89 53 67 30 73 16 69 87 40 31
 04 62 98 27 23 09 70 98 73 93 38 53 60 04 23

Final Least Cost Path = 447
ed-vm%
ed-vm%
ed-vm%
```