CS 302 – Assignment #01

Purpose: Refresh concepts regarding C++ simple I/O, functions, object oriented programing, dynamic allocation, variable scoping, and compilation/linking. Verify installation of development environment. Introduce algorithmic design patterns.
Due: Wednesday (9/05) → Must be submitted on-line before class.
Points: Part A → 50 pts Part B → 50 pts

Reading/References

Chapter 1, Data Structures and Algorithms

Assignment:

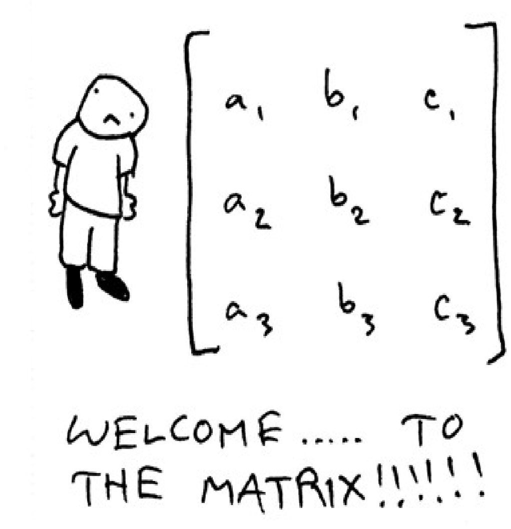
Solve the maximal matrix sum path problem using two different approaches.

Part A:

Given a matrix of numbers, with an order of 5, as shown (on right), we wish to find a path from the top to the bottom at the most cost. Start from any element in the first row, at each step, going down left or down right is the only option. The 'cost' is a summation of the numbers used along the way.

9	**1**	**4**	**8**	**5**
7	**2**	**1**	**3**	**4**
4	**7**	**6**	**2**	**5**
8	**1**	**9**	**5**	**3**
2	**8**	**3**	**1**	**4**

For example, starting from the 9, a maximal path would be (9, 2, 6, 5, 4) which would yield a sum of 26. However, starting from the 8, a maximal path would be (8, 1, 7, 9, 8) which would yield a sum of 33 which turns out to be the maximal path for this matrix.



Toothpaste For Dinner.com

Data Structure

In order to store the sizes and values efficiently, we will use dynamically allocated two-dimensional arrays. The algorithmic approach and the matrix order will be read from the command line.

Algorithm 1

The simplest approach is just try every possible combination of elements. This can be done fairly easily via recursion. This approach is generically referred to as brute force. For each item, take the best of either the left or right path. We must test this for each element of the first row.

$$pthSum[r,c] = \begin{cases} 0 & if\ \ c < 0 \ \| \ c \geq order \\ r & if\ \ r = order - 1 \\ max\big(mat[r][c] + pthSum(r+1, c-1), & \\ \qquad mat[r][c] + pthSum(r+1, c+1)\big) & otherwise \end{cases}$$

The base case for the recursion would be when the column is off the grid, thus 0 or the row is on the bottom on which case we return the current value.

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 sub-problems, solving each of those sub-problems, and storing their solutions. 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.

For example, given the following matrix

479	**665**	**154**	**269**	**501**
998	**992**	**904**	**763**	**254**
591	**869**	**843**	**683**	**708**
410	**88**	**352**	**566**	**497**
252	**486**	**565**	**115**	**585**

We can build a temporary array (right) one row at a time, starting with the first row which is initialized from the matrix.

The next row is based on the current value from the original matrix and the largest sum of either the left of right cell of the previous row. For example, cell (1,1) is based on the max of 992+479 (1471) and 992+154 (838) which is 1471. This is done for each cell in the row, ignoring invalid matrix locations.

479	**665**	**154**	**269**	**501**
998	**992**	**904**	**763**	**254**
591	**869**	**843**	**683**	**708**
410	**88**	**352**	**566**	**497**
252	**486**	**565**	**115**	**585**

479	**665**	**154**	**269**	**501**
1663	**1471**	**1569**	**1264**	**523**

This process is repeated for each row until the temporary matrix is filled.

479	**665**	**154**	**269**	**501**
998	**992**	**904**	**763**	**254**
591	**869**	**843**	**683**	**708**
410	**88**	**352**	**566**	**497**
252	**486**	**565**	**115**	**585**

479	**665**	**154**	**269**	**501**
1663	**1471**	**1569**	**1264**	**523**
2062	**2532**	**2314**	**2252**	**1972**
2942	**2402**	**2884**	**2880**	**2749**
2654	**3428**	**3445**	**2999**	**3465**

The final maximal path sum is the largest value in the bottom row of the temporary array, 3465 in this example.

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

Class Descriptions

- Maximal Matrix Path Sum Problem, **bestPath**, Class

  The **bestPath** class will implement both algorithms and some support functions. A header file and implementation file will be required.

bestPath
-maxtrixOrder: int
-**matrix: int
-LIMIT=999: static const int
-MIN_ORDER=5: static const int
-MAX_ORDER=100: static const int
+bestPath()
+~bestPath()
+displayMatrix(): void
+createMatrix(const int): bool
+bestPthDY(): int
+bestPthREC(): int
-bestPthREC1(int, int): int

Function Descriptions

- The *bestPath()* constructor function will initialize class variables as appropriate.
- The *~bestPath()* destructor function should free the dynamically allocated memory.
- The *dissplayMatrix()* function should display the matrix in a formatted manner. Refer to the example for output formatting.
- The *createMatrix()* function should dynamically create the matrix of the passed order, and populate the arrays with random numbers. The order must be between MIN_ORDER and MAX_ORDER (inclusive). The random numbers should be generated using the C++ rand() function with the LIMIT as follows **`rand()%LIMIT+1`** (in cell order).
- The *bestPthDY()* function should solve the matrix maximal sum path problem by using a dynamic programming approach (see algorithm 2 explanation).
- The *bestPthREC()* function should solve the matrix maximal sum path problem by calling the private recursive function.
- The *bestPthREC1()* function should solve the matrix maximal sum path problem recursively (see algorithm 1 explanation).

You should not need any additional private functions.

Part B:

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

Enter the execution times into a spreadsheet (based on the item counts) and create a line chart plot of the execution times vs array length for each algorithm. The results are provided in minutes and seconds format and for clarity, however only seconds will be entered in the spreadsheet. Refer to the example for how the plot should look.

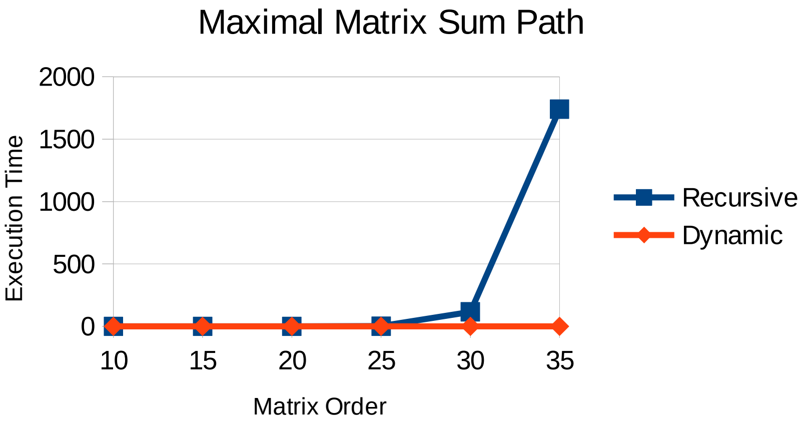
Create and submit a write-up with an explanation not to exceed ~500 words including the following:

- 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, due to different hardware, execution times for each submittal will be different (very different). You should use a word-processor (LibreOffice, MS Word, or Google Docs, etc.) but must submit the file in PDF format.

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%
ed-vm% ./main -dy 4
Error, invalid matrix order.
Program terminated.
ed-vm%
ed-vm% ./main -dy 101
Error, invalid matrix order.
Program terminated.
ed-vm%
ed-vm% ./main -dy 10
************************************************************
CS 302 - Assignment #1
Best Matrix Path Finder.

Algorithm: Dynamic Programming

Matrix:
Order:   10

  479  665  154  269  501  998  992  904  763  254
  591  869  843  683  708  410   88  352  566  497
  252  486  565  115  585  414  864   23  389  308
  546  586  973  418  573  193  416  566  815  179
  538  406  766  381  807  194  510  894  264   76
  111  515  281  675  630  865  807  213  887  914
  520  433  501  493  570  792  404  985   77  219
  883  334  343  649  714  151  561  942  763  825
  737  592  340   18  267  688  601   75  900  488
  988  421  639  208  632  209  719   37  913  795


Best Possible = 7523
ed-vm%
ed-vm% ./main -rc 5
************************************************************
CS 302 - Assignment #1
Best Matrix Path Finder.

Algorithm: Recursive

Matrix:
Order:   5

  479  665  154  269  501
  998  992  904  763  254
  591  869  843  683  708
  410   88  352  566  497
  252  486  565  115  585


Best Possible = 3465
ed-vm%
```