CSE 101 Winter 2016
Programming Assignment Three (Dynamic Programming!)

Due: Friday February 23, 11:59 PM

Link to starter code: https://github.com/UCSD-CSE101/W18-PA3

PLEASE READ THE FAQ BEFORE POSTING ON PIAZZA

https://docs.google.com/document/d/1eM-kwmpA2y2w1721HER0JWWydamuGMrbcVgkvuExoDs/edit?usp=sharing

TwoD_Array.hpp

Many of the questions in this PA will make use of 2d arrays. However, accessing and creating large 2d arrays in C++ is slow, and may cause you to run out of stack space. We have created a templated wrapper class TwoD_Array so that you can access values without worrying about memory or the underlying implementation. **Please be very familiar with the implementation of TwoD_Array** as you will need to use it to achieve faster runtimes for each of the dynamic programming questions!

Constructor: You can create a TwoD_Array on the heap or the stack using the following syntax:

```
TwoD_Array<int> * arr = new TwoD_Array<int>(numRows, numCols)
TwoD_Array<int> arr = TwoD_Array<int>(numRows, numCols)
```

This will create a TwoD_Array of type int with numRows rows and numCols columns. *Please note that no additional checks were added, so if you call the constructor with a negative value for whatever reason, then it will segfault.*

Element Accessing: The **at** method returns a reference to the value at that index and can be used to both set and get the value.

```
arr.at(0, 0) = 5
int x = arr.at(0, 0)
```

Again, no checks have been added, so if you call at with negative indices, it may still silently run without segfaulting. In these cases, your program will not be doing what you think, so please be careful and save yourself the debugging time by thinking about these edge cases in advance.

Dimension Getting: `getNumRows()` and `getNumCols()` will return the number of rows and columns of the array

Printing: You can use `printOut()` to print the array row by row to std::out. This can be used to aid in your debugging. ***Please make sure you do not include calls to*** `printOut` ***in your final submission.***

Question 1: Rod Cutting [25 points]

```
Input:     The length of rod to cut
           Map from rod lengths to prices

Output:    Maximum possible profit made by cutting the rod up
```

For this question, you have a rod of length n, and a set of prices that you can sell different lengths of rod for. Your goal is to cut up the rod and sell the pieces in a way that maximizes the total amount of money you get. You do not need to return the actual cuts made, just the total profit made by selling them.

A naive solution would generate every possible configuration and check its profit. If we have a maximal number of possible rod lengths to sell, this will run in O(2^n) time, since there will be 2^n total possible sets of cuts.

Your program should use dynamic programming to improve on the naive solution and should run in O(n^2) time, where n is the initial length of the rod. Don't worry, the map from rod lengths to prices is considered a size of n, so you don't have to worry about exceeding the runtime here.

To Make:

```
make TestRodCut
```

To Run:

```
build/TestRodCut testcases/input_file
```

Example Input:

```
8          // The initial length of rod
1 1        // A length of rod and the price it sells at
2 5        // <- this means a rod of length 2 sells for 5 profit
6 17
7 19
```

Example Output:

```
Maximum profit: 22
```

In the example above, we can cut the rod of length 8 into two rods of length 2 and 6, and sell them for a total profit of 22. Although all possible lengths of rod may not have prices, you will be always be guaranteed the ability to sell a rod of length 1 so you will always be able to cut and sell all of the rod.

Question 2: Grid Sum [25 Points]

Constructor Input:	*N* by *N* square matrix of integers
Query Input:	x_{min} , y_{min} , x_{max} , y_{max} with $x_{min} \leq x_{max}$ and $y_{min} \leq y_{max}$. This 4-tuple designates a rectangular submatrix of the constructor input matrix.
Query Output:	Sum of all integers in each specified rectangular submatrix.

For this question, you will perform a **precomputation** on an *N* by *N* square matrix of integers, and answer subsequent queries for the sum of all integers in a rectangular submatrix of the inputted matrix. **The precomputation must have a time complexity of *O(N²)* and a space complexity of *O(N²)* where *N* is the number of elements in a row of the *N* by *N* matrix** -- these constraints are achievable with dynamic programming. The queries afterwards for the sum of a rectangular submatrix of integers must be answered with time complexity *O(1)*. The matrix is indexed with (0, 0) as the upper-left corner, and (*N* - 1, *N* - 1) as the lower-right corner. For a coordinate (*x*, *y*), *x* indicates the row index and *y* indicates the column index.

Your programs will be tested with up to *N*=5,000 and number of queries *Q*=100,000. For running this upper bound case on ieng6, your program should finish in around 10 seconds, and will be given 20 seconds when grading to ensure no problems with file writing. You should see that on larger test cases, the executable calculates everything for a few seconds, then outputs the result of all of the queries at once. The tester does not output as it goes along because the time cost for switching back and forth from IO adds significant overhead to the overall running time. It is also guaranteed that the total sum of the entire grid can be stored within an integer, so you do not have to worry about overflow issues.

For generating larger test cases, a python script (gridsum_gen.py) has been provided for your convenience. This script is run similarly to graph_gen by providing the number of rows/cols and the number of queries as command line arguments and will save generated files in the grid/ directory

We have also provided a verification script (gridsum_verify.py) to calculate the correct outputs for a given query on a given graph. Run both scripts with the --help flag to learn more about how to specify inputs.

To Generate a Test Case:

```
    python gridsum_gen.py -n [numRows/Cols] -q [numQueries]
```

To Make:

```
    make TestGridSum
```

To Run:

```
    build/TestGridSum grid/input_file
```

Sample Input:

```
    5                  // First line denotes N
    3  8  3  27 9      // Subsequent N lines define the N by N matrix
    9  22 38 2  87     // starting from the first line as row 0
    3  7  90 15 73     // and ending with the last line as row N-1
    4  5  19 45 13
    8  23 17 59 4
    0 0 1 2            // The next lines represent the queries to
    0 0 4 4            // be executed.
    2 3 4 4            // Query format is "x1 y1 x2 y2",
    1 2 1 4            // where x1 ≤ x2 and y1 ≤ y2 and x1, x2, y1, y2 ≥ 0
    3 3 3 3
    3 3 4 4
    0 0 0 0
    2 0 4 2
```

Sample Output:

```
    Grid sum for (0,0) to (1,2): 83
    Grid sum for (0,0) to (4,4): 593
    Grid sum for (2,3) to (4,4): 209
    Grid sum for (1,2) to (1,4): 127
    Grid sum for (3,3) to (3,3): 45
    Grid sum for (3,3) to (4,4): 121
    Grid sum for (0,0) to (0,0): 3
    Grid sum for (2,0) to (4,2): 176
```

To Verify:

```
    python gridsum_verify.py -f grid/input_file -x 0,0,1,2
        Sum from (0, 0) to (1, 2) is 83
```

Question 3: Longest Common Subsequence [25 Points]

Input: Two sequences *X* and *Y* composed of (capital) English letters, and with respective lengths *n* and *m*, having a unique **longest common subsequence**

Output: The longest common subsequence between *X* and *Y*

The longest common subsequence problem is a canonical example of dynamic programming in practice, and is used in utilities such as the *diff* command used in *bash*, *git*, and more! To solve this problem naively for two strings of length *n* and *m*, we can first generate all subsequences of the first string in $O(2^n)$, and then check each subsequence against the other string to find the longest common subsequence with a total time complexity of $O(2^n*m)$. However, in lecture we discussed a dynamic programming approach that utilizes the recurrence relation that allows us to break down the problem by matching incrementally longer (prefix) substrings and "building" common subsequences thereby reducing the time complexity to $O(n*m)$. You can find a more detailed description of the recurrence in lecture 10, slide 10.

For this question, you will be given two strings (representing the two input sequences of English letters) that have a **unique** LCS (note that it is normally possible to have multiple LCS's). You must return the LCS using a dynamic programming implementation.

To Make:

```
make TestLCS
```

To Run:

```
build/TestLCS input_file
```

Sample Input #1:

```
AMLREZETS
FWFTLESSEHT
```

Sample Output #1:

```
The LCS is: 'LEET'
```

Sample Input #2:

```
AAAAA
BBBBB
```

Sample Output #2:

```
     The LCS is: ‘’
```

Sample Input #3:

```
     AAAAA
     AAAAAAAAAA
```

Sample Output #3:

```
     The LCS is: ‘AAAAA’
```

Given input strings *s1* and *s2* of lengths *n* and *m* respectively, you are expected to return the LCS in *O(n*m)* time and space complexity via the dynamic programming technique discussed in class. We will test strings of up to 5000 characters, and your LCS implementation should complete within 5 seconds on ieng6 for these upper bound cases. When grading, we will allow your program to run for up to 10 seconds.

Question 4: Analysis of Filling a USB Drive [25 Points]

```
Input:     Positive scalar value (in MB) for the total storage
                capacity of the USB drive
           List of unique and positive file sizes (in MB) with
                infinite supply for each

Output:    Minimum number of files needed to fill up the USB
```

For this question, you are given a **storage size** for a USB drive (integer representing amount of memory in MB) and a list of **file sizes** (integers representing file sizes in MB). Your goal is to totally fill the USB drive's memory using as few files as possible.

You must fill the USB drive using only files from the list of possible file sizes. You may use any of the files as many times as you'd like. For this question, it is guaranteed that you will always be given a file of size 1 MB, **so it will always be possible to fill the entire hard drive**. Please keep this 1 MB guarantee in mind and include the 1 MB file in your test cases.

You will implement two approaches to solving the problem of finding the minimum number of files needed to fill the USB drive.

The naive solution (recursively testing all possible orderings of files in order to find the one that uses the minimum number of files) has been provided to you through the starter code. It is highly recommended that you read through it and thoroughly understand it so that you can see the recurrence and overlapping subproblems associated with the naive solution. Since the naive implementation is recursive, it is constrained by stack space and it will segmentation fault on inputs that are large enough. For the purposes of this class, this recursive implementation is perfectly fine. However, please keep this in mind if you ever find yourself writing a recursive solution for a real application!

Your first implementation will improve upon the naive solution by memoizing its subproblems. We expect its runtime to be much faster after caching solutions to subproblems. It is recommended that you use a std::map<int, int> object to perform the memoization with the key as the USB drive size, and the value as the corresponding smallest number of files needed to fill that drive. When making your memoized implementation, you should start with the provided naive solution and add the necessary features. The memoization object has also been provided for you in the skeleton code and is placed above the method definition. Your implementation should still be recursive and is not expected to work on inputs that are limited by stack space.

Your second implementation will use dynamic programming (DP) to further improve the finding of the minimum number of files. It must have time complexity *O(N * F)* where *N* represents the USB drive size in MB and *F* represents the number of available file sizes. Revisit the knapsack problem (lecture 10, slide 29) as a starting point for your DP implementation.

It is highly recommended that you use the provided TwoD_Array class for your DP array to ensure that you do not run into memory issues (do not try to make your DP array locally in the stack space -- it may work for smaller inputs, but it will segmentation fault for larger inputs). We will test USB drive sizes up to *N*=1,000,000 and distinct number of files up to *F*=100. For this upper bound case, your executable should finish in roughly 5 seconds, and will be given 10 seconds to run on ieng6 when grading.

In addition to the above implementations, **you must provide a <u>qualitative</u> analysis in the form of a write-up PDF** comparing the different solutions to finding the minimum number of files required to fill up the USB.

You must also include the following points in your analysis:

- Give a rough estimate for the number of subproblems associated with the naive solution. What input scale (in *N* and *F*) is too much for the naive solution to handle?
- How does memoization improve the naive solution? Describe a test case for which you saw that the naive solution was unable to complete in a reasonable amount of time, but the memoized version was able to find the minimum number of files rather quickly.
- Compare the rough runtimes between your memoized implementation and your dynamic programming implementation. As in the above point, describe a test case for which the memoized version appears to "hang", but the dynamic programming version finishes rather quickly. (Note: this should be a test case for which the memoized version "hangs" instead of segmentation faulting to due memory constraints.)

To Make:

```
make TestUSB
```

To Run:

```
build/TestUSB testcases/input_file INDEX
```

Where INDEX *is* 1, 2, *or* 3 *corresponding with naive, memoized, or DP respectively.*

Sample Input:

```
13
10 1 6
```

Sample Output (for each implementation):

```
build/TestUSB usb1.txt 1
Result of find_usb_naive: 3

build/TestUSB usb1.txt 2
Result of find_usb_memoized: 3

build/TestUSB usb1.txt 3
Result of find_usb_dp: 3
```

As you would expect, the three solutions always output the same value. Please be careful to ensure that your solutions do the same!