Question 1: Grid Sum [25 Points]

Initial Input Matrix: \( N \) by \( N \) square matrix of integers

Subsequent Input: One or more lines with 4-tuples \((x_{\text{min}}, y_{\text{min}}, x_{\text{max}}, y_{\text{max}})\) with \(x_{\text{min}} \leq x_{\text{max}}\) and \(y_{\text{min}} \leq y_{\text{max}}\). Each 4-tuple indicates a rectangular submatrix of the input matrix.

Subsequent 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 \( O(N^2) \) and a space complexity of \( O(N^2) \) 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.

For this programming assignment, a TwoD_Array class has been provided to you which allocates one contiguous block of memory and allows you to use a one-dimensional array as if it were two-dimensional. We do this for performance issues since we are working with large two-dimensional arrays, and if each row of the array were scattered throughout the memory space, it will be more difficult for the machine to find a particular element upon request. The TwoD_Array has a templated type and can only store homogeneous data (i.e., all integers, all characters, etc. depending on what you used as the template type). 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! The input for this question will also be given as a TwoD_Array of type \(<\text{int}>\).

Breakdown of TwoD_Array<\text{T}> (which can have its array type templated out as \text{int}, char, etc.):

1) Constructor \rightarrow TwoD_Array<\text{int}> myArr(500, 1000);

The constructor above will create a local TwoD_Array of type \text{int} with 500 rows and 1000 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.
2) Element Accessor → myArr.at(50, 50);
   The accessor returns a reference to the element at the position specified by row index, column
   index in that order. Since it returns a reference, you can do the above to get the value, and you
   can also do: myArr.at(50, 50) = 100; to set the array value at index(50, 50) to 100.
   Again, no checks were 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.

3) Dimension Getters → myArr.getNumRows(), myArr.getNumCols()
   As you would expect, these getters get you the total number of rows and total number of
   columns, respectively.

4) Array Verifier → myArr.printOut();
   The above will print out the two-dimensional array row by row. This is provided to help you with
   your debugging efforts. Please make sure you do not include calls to printOut in your final
   submission.

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. As in the previous assignment, modify the values for $N$ and $Q$ at the top of the script to
set the parameters for the test case. After execution completes, a file named GENGRIDNQ will be
saved to the current directory.

To Generate a Test Case:
   python gridsum_gen.py

To Make:
   make TestGridSum

To Run:
   ./TestGridSum.out input_file
Sample Input:

```
5
3 8 3 27 9
9 22 38 2 87
3 7 90 15 73
4 5 19 45 13
8 23 17 59 4
0 0 1 2
0 0 4 4
2 3 4 4
1 2 1 4
3 3 3 3
3 3 4 4
0 0 0 0
2 0 4 2
```

First line denotes N
Subsequent N lines define the N by N matrix starting from the first line as row 0 and ending with the last line as row N-1

The next lines represent the queries to execute
Query format is “x₁ y₁ x₂ y₂” with the guarantee that x₁ ≤ x₂ and y₁ ≤ y₂ and x₁, x₂, y₁, y₂ ≥ 0

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
```
Question 2: String Reconstruction [25 Points]

Input: An upper-case string of English letters, followed by a dictionary of valid words and each word’s unigram probability (basically, a probability for each word)

Output: The most likely string decomposition of the input string into valid words based on a unigram probability model

In lecture, you learned about a relatively elegant dynamic programming method of determining whether a string can be decomposed into a sequence of valid words (i.e., words that exist in a given dictionary). In this problem, you will be tackling this challenge while racing against the clock and using a “unigram probability model” to efficiently deduce the most likely decomposition of the string, if a decomposition exists.

To Make:
make TestStrRecon

To Run:
./TestStrRecon.out input_file

Sample Input #1:
THEMESOBIG
THE 0.001
OF 0.0005
TO 0.0006
A 0.0002
AND 0.0006
IN 0.0003
SO 0.1
BIG 0.01
OBIG 0.001
O 0.1
THEMES 0.1
MESS 0.01

Sample Output #1:
Most likely sentence: 'THEMES SO BIG'

Sample Input #2:
NOTPOSSIBLE
NOT 0.001

Sample Output #2:
Most likely sentence: ''
Notice that in the first case, there are three possible outputs - “THE MESS O BIG” or “THEMES SO BIG” or “THE MESS OBIG”. However, because “THEMES SO BIG” is more likely based on the unigram probability model, you must output “THEMES SO BIG”. Computations for this problem are expected to complete within $O(N^2)$ runtime (disregarding dictionary lookup time), without strict time limits.

Notes:
- For a string of non-trivial length, multiplying together all of the unigram probabilities will quickly exhaust the precision C++ decimal data formats provide. Hence, scientists often use the log probability to compare probabilities of such precision. However, for the sake of simplifying calculations we will not be using log probabilities to calculate the probabilities, and you will not be tested on cases where determining the most likely decomposition requires a precision greater than the 53 bits (~15 digits) of decimal precision double provides.
- It is helpful to compute the probability that a certain word is selected as you build your DP table, rather than after the DP table is complete!
- You are expected to return an empty string if a decomposition is not possible with the given dictionary.
- You may assume that the maximum-probability decomposition is unique for each input sentence that has at least one valid decomposition.
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 substrings of the first string in \(O(2^n)\), and then check each substring against the other string to find the longest common subsequence with a total time complexity of \(O(2^n \times m)\). 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 \(O(n \times m)\). You can find a more detailed description of the recurrence in lecture 10, slide 9.

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:
./TestLCS.out 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
   AAAAAAAA

Sample Output #3:
The LCS is: ‘AAAA’

Given input strings $s1$ and $s2$ of lengths $n$ and $m$ respectively, you are expected to return the LCS in $O(n\times m)$ 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 Finding Stamps [25 Points]

Input: Positive scalar value (in cents) for the amount of postage needed, List of unique and positive stamp values (in cents) with infinite supply for each

Output: Minimum number of stamps needed to make the postage

For this question, you are given a required postage (integer representing a value in cents) and a list of allowed stamp denominations (integers representing stamp values in cents). You must achieve the required postage using only stamps from the list of stamp denominations. You may use any of the stamp denominations as many times as you'd like. For this question, it is guaranteed that you will always be given a stamp with the 1-cent denomination, so it will always be possible to create the required postage. Please keep this 1-cent guarantee in mind and include the 1-cent stamp in your test cases.

You will implement two approaches to solving the problem of finding the minimum number of stamps needed to achieve the required postage.

The naive solution (recursively testing all possible orderings of stamps in order to find the one that uses the minimum number of stamps) 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 postage amount, and the value as the corresponding lowest number of stamps to make that postage. 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 stamps. It must have time complexity $O(P \times S)$ where $P$ represents the postage value in cents and $S$ represents the number of available stamp denominations. Revisit the knapsack problem (lecture 10, slide 18) 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 postage value up to
\( P = 1,000,000 \) and distinct number of stamps up to \( S = 100 \). 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 qualitative analysis in the form of a write-up PDF comparing the different solutions to finding the minimum number of stamps needed to make the required postage.

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 \( P \) and \( S \)) 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 stamps 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 due to memory constraints.)

To Make:
make TestStamps

To Run:
./TestStamps 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):

./TestStamps.out stamps_ex 1
Result of find_stamps_naive: 3

./TestStamps.out stamps_ex 2
Result of find_stamps.memoized: 3

./TestStamps.out stamps_ex 3
Result of find_stamps_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!