CSCI3320 Programming Assignment #1

Problems Description: Note that the maximum contiguous subsequence sum algorithms in the text do not give any indication of the actual sequence. In this assignment, you will modify these algorithms to return the **minimum contiguous subsequence sum including the starting and ending indexes** of the sum. Also, your program will measure their execution times with various input sizes.

(a) Modify Algorithms 2, 3 & 4 so that they return in the value of the **minimum** subsequence sum and the starting and ending indices of the subsequence. Your algorithm will return '0' if there is no positive number in the input array.

Your algorithm should take input size (N) from the user and generate a random sequence of N integers ranging from -9999 to 9999. **If N is less than 50**, your program **must print** the randomly generated numbers, and find the minimum subsequence **independently** (**do not refer** the indexes found by other algorithms. Your program should measure (and print) the execution time of these algorithms (Algorithms 2, 3 & 4) in milliseconds or nanoseconds.

```
< Sample Execution Scenarios >
Please enter the size of the problem (N): 15

Please enter the size of the problem (N): 15
673 -869 -153 214 -139 40 65 -925 -639 -696 956 823 -714 500 967

Algorithm 2:
MimSum: -3102, S_index: 1, E_index: 9
Execution Time: 82048 nanoseconds

Algorithm 3:
MimSum: -3102, S_index: 1, E_index: 9
Execution Time: 23165 nanoseconds

Algorithm 4:
MimSum: -3102, S_index: 1, E_index: 9
Execution Time: 11345 nanoseconds
```

< Note: all three algorithms must be executed for the same input numbers >

(b) Measure and compare the executions times of your algorithms at least 10 times and find **the average execution times** of these algorithms. The execution time of each test case must be recorded in a table. A template for input size =800 is shown below for your reference. You have to add a similar table for each input size.

< The recorded video uses a different range of input sizes. Please note that you only need to select a range of input sizes for your experiment that allows you to observe the crossover points where the execution times of the algorithms intersect. Feel free to use different input sizes for your test. The input sizes provided below are just one possible range.>

- Plot the average execution times of **Algorithm 2 & Algorithm** 3 for input sizes 100, 200, 400, 800, 1600, 3200, and 64000 (**adjust the input sizes as needed**).
- Plot the average execution times of Algorithm 3 & Algorithm 4 for input sizes 100, 200, 400, 800, 1600, 3200, and 64000 (**adjust the input sizes as needed**).

(Note: Your graph should display execution time on the vertical axis (Y-axis) and input size on the horizontal axis (X-axis). **It must also highlight the crossover points where the execution times of the two algorithms intersect**. Feel free to adjust the input sizes in your graph as needed.)

(To generate a reasonable graph, please feel free to use different input sizes as needed)

Input size (800)	`Algorithm 2`	`Algorithm 3`	`Algorithm 4`
Test 1			
Test 2			
Test 3			
Test 4			
Test 5			
Test 6			
Test 7			
Test 8			
Test 9			
Test 10			
Average			

(c) Compare the Big-Oh complexities of Algorithm 2, 3 & 4 with your experimental results (b). **If there is any major discrepancy between your results and the algorithm complexities in Big-Oh, discuss where the discrepancy came from**.

- **For Python Programmers:** Use 'PyCharm' for your IDE. For the installation of PyCharm, please refer the following webpage: https://www.jetbrains.com/help/pycharm/installation-guide.html.
  Do not use any non-standard Python library (such as NumPy).

  Deliverable: A **zipped file** including
  - source codes with compiling/running instructions,
  - Excel documents with two graphs (one for Algorithm 2 & 3, and the other for Algorithm 3 & 4) for the averaged execution times of three methods (**Note that** your Excel file **must** include all tables of ten measured execution times per each algorithm and each input size included in your graph)
  - a MS-Word document (Optional): Clear justification on the experimental results if there is any major discrepancy between the execution time graphs and the complexity analysis of the algorithms).

  (Upload **a SINGLE zipped file** via Canvas. Note that the zipped file must includes all electric copies of your source codes with brief running instruction, the graphs, and the MS-Word document)

  Grading Criteria:
  - Finding correct indexes (Your program must print all random numbers when N is smaller than 50. Otherwise, you will most 50%)
    - Algorithm 2: 10%, Algorithm 3: 30%, Algorithm 3: 10%
  - Experiments (50%)
    - Generating random numbers and retuning execution time for each test-case (10%)
    - Reporting measured execution-time tables for all required test cases (20%)

- Including two graphs (20%): For the full points, **<u>the graphs must highlight the crossover points where the execution times of the two algorithms intersect</u>**.

For full credit, your code should be **well documented with comments**, and the style of your code should follow the guidelines given below:

Your programs must contain enough comments. **Programs without comments or with insufficient and/or vague comments may cost you 30%.**

Every user-defined method or function (that you added in the source files) should have a comment header describing inputs, outputs, and what it does. An example function comment is shown below:

```
def quadratic_min _subsequence(array: List[int]):
    """
    FUNCTION quadratic_min_subsequence:
        Find the subsequence in the array with the minimum sum
        using an algorithm with O(n^2) time complexity.

    INPUT_Parameters:
        array (List[int]): Array to evaluate for the largest subsequence.

    Returns:
        ArraySubSequence: The subsequence with the largest sum in the array.

    """
```

Inline comments should be utilized as necessary (but not overused) to **make algorithms clear** to the reader.

Not-compile programs receive 0 point. By **not-compile**, I mean any reason that could cause an unsuccessful compilation, including missing files, incorrect filenames, syntax errors in your programs, and so on. Double check your files before you submit, since I will **not** change your program to make it work.

Compile-but-not-run programs receive no more than 50%. Compile-but-not-run means you have attempted to solve the problem to certain extent, but you failed to make it working properly. A meaningless or vague program receives no credit even though it compiles successfully.

Programs delivering incorrect result, incomplete result, or incompatible output receive no more than 70%