< Programming Assignment #3>

Problems Description: Your algorithm first takes an input size ($10 \leq N < 200000$) from the user. First, your program will generate a random sequence of N integers **ranging from -999,999,999 to 999,999,999** and store them in an array A (list in Python). **If N is less than 50,** your program must **use random numbers ranging from -99 to 99 and print all randomly generated numbers** on the screen. After generating random numbers, your program takes an input K from the user again, and determines if there are two numbers whose sum equals a given number K. For instance, if the random numbers are 8, 4, 1, 6 & 3 and K is 10, then the answer is 'yes' (since 4 + 6 = K).

Do the following:

(a) **Give $O(N^2)$ algorithm** to solve this problem.
(b) **Give O(N log N) algorithm** to solve the problem (Hint: Sort the array first using a Python sort and find (k-A[i]) from the array A for $0 \leq i < N$). Describe your idea and analyze the complexity of your approach using your Python code.
(c) **Give O(N) algorithm** to solve the problem (Hint: Use a hash table. **Assume the size of the hash table is 500,009**). Describe your idea and analyze the complexity of your approach using your Python code.
(d) Code these solutions and compute the running times of your algorithms. (Write your code in Java for (a), (b) and (c))
(e) Evaluate and contrast the worst-case execution times of your algorithms for identifying pairs with sums equal to K in a minimum of 10 iterations. Subsequently, create **a plot illustrating the averaged worst execution time across these approaches** for input sizes 1000, 2000, 4000, 8000 and 16000 (the range of your input sizes can vary based on the speed of your computer. Note: the execution times on the vertical axis and the input sizes on the horizontal axis in the graph). Make sure that you measured the worst-case execution time. An easy way to measure the worst-case execution time is to **make *k* greater than 2,000,000,000**.
    - Plot the average execution times of your $O(N^2)$ & O(N log N) algorithms for the selected input sizes in the first graph.
    - Plot the average execution times of your O(N log N) & O(N) algorithms for the selected input sizes in the second graph.

    (Your graphs must show the averaged worst-case executions time on the vertical axis and the input sizes on the horizontal axis)
(f) Turn in the softcopies of two graphs for the averaged worst-case execution times of your algorithms.

Here is a sample execution of the program:

Scenario #1:

```
Enter size of random array: 10
75 35 -42 -26 -37 61 29 -71 -68 65
Enter the K value: -33
Running the algorithms ...
```

```
< Quadratic Algorithm >
Yes, there are two numbers whose sum equals to K
K = -33, (35 + -68)
Execution time in nanoseconds: 35

< O(NlogN) Algorithm >
Yes, there are two numbers whose sum equals to K
K = -33, (35 + -68)
Execution time in nanoseconds: 55

< Linear Algorithm >
Yes, there are two numbers whose sum equals to K
K = -33, (35 + -68)
Execution time in nanoseconds: 95
```

Scenario #2:

```
Enter size of random array: 4000
Enter the K value: 2000000000    Running
the algorithms ...

< Quadratic Algorithm >
No, the algorithm could not find two numbers whose sum equals to K
Execution time in nanoseconds: 217000

< O(NlogN) Algorithm >
No, the algorithm could not find two numbers whose sum equals to K
Execution time in nanoseconds: 117800

< Linear Algorithm >
No, the algorithm could not find two numbers whose sum equals to K
Execution time in nanoseconds: 57800
```

Deliverable: A single zipped file that includes the followings:

- For (a), (b) and (c): In **a text file** (like MS Word), shown your algorithms with a brief explanation of your method and the complexity analysis of them ($O(N^2)$, O(NlogN), and O(N) algorithms),
- **Tables** containing ten worst-case execution time measurements for each algorithm across different input sizes,
  (A template of the table for input size =8000 is shown below for your reference. You must add a similar table per each input size.)

Input size (8000)	$O(N^2)$ algorithm	O(NlogN) algorithm	O(N) algorithm (**optional**)
Test 1			
Test 2			
Test 3			
Test 4			

Test 5			
Test 6			
Test 7			
Test 8			
Test 9			
Test 10			
Averaged Worst-case Time			

- **<u>Two graphs of measured worst-case executions times</u>**, and
- **<u>All source files.</u>**
- (optional) Running instructions (including any special compilation steps)

For Python Programmers: Use ‘PyCharm’ for your IDE.

For Java Programmers: Before submitting your assignment, remove all IDE-related headers such as a package header. Note: You must be able to compile your codes using a simple command (i.e. ‘javac assignment.java’, for more information, please see the following webpage: https://introcs.cs.princeton.edu/java/11hello/).

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_max_subsequence:
        Find the subsequence in the array with the maximum 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 a 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 60%.