**CSCI 385 – Data Structures and Algorithm Analysis**

**Homework #3 – Measuring Performance**

**Due: Thursday October 17, 2019 11:59 pm**

**(100 pts)**

1. Consider the following problem: Given a list of N integers, count the number of triples that sum to zero.

Algorithm:

Let *a* be the list of N integers

cnt = 0

i = 0

while (i < N)

j = i + 1

while (j < N)

k = j + 1

while (k < N)

if (*a*[i] + *a*[j] + *a*[k] = 0)

cnt = cnt + 1

endif

k = k + 1

endwhile

j = j + 1

endwhile

i = i + 1

endwhile

return cnt

Implement the algorithm above (**TripleSum**) using the following specification. Create a program from which you can quantitatively measure the running time of this algorithm using a number of trials. You will need to create a “stopwatch” that will measure the elapsed time (in seconds) that it takes to identify all of the triples that sum to 0 in the list. Your program should create multiple lists of randomly generated integers between the values *-10000* and *10000* to test. The first list size should be of size *500*, but for each subsequent trial, the list size should be doubled (up to a list of 128,000 elements). The output for each trial should be the size of the list, the count of the number of triples that sum to 0, and the elapsed time for the trial (in seconds). Develop a **detailed design** and a **comprehensive test plan** that is all implementation ready. Clearly document as part of your design the data structures you are utilizing to solve the problem. Your source code should be **fully documented** and conform to **good programming style**.

Your submission should show that you have fully implemented your developed test plan and contain the following ***in the order listed below*** for Question #1: (40 pts)

* Data Structure documentation
* Detailed Design
* Test Plan
* Source Code
* Script of the program running, both implementing your test plan and running the trials

1. Take the results from your trials and plot the data where the x-axis is the problem size (*N*) and the y-axis is the running time (*seconds*). What conclusion can you draw about the running time of the algorithm based on the input size? Given the data collected, can you estimate the running time for an input size of *N*=256,000? How about *N*=1,000,000? (5 pts)
2. Explain fully how you generated *negative random integers* for your array in question 1 and why it was necessary to do this. (5 pts)
3. The algorithm above is a brute-force solution to the *TripleSum* problem where the *if statement* is executed exactly

(1.1)

times. The question is – can we improve on this? Consider the following problem: Given a list of N *distinct* integers, count the number of pairs that sum to zero. Solving this problem, the same as **TripleSum** would require a nested loop and the following number of executions of the *if statement*:

(1.2)

If we use a *mergesort* and *binary search* on the list, it is possible to reduce the running time of the algorithm. Let’s call this new algorithm **PairSumFast**.

Algorithm:

Let *a* be the list of N integers, a[i] ≠ 0

cnt = 0

sort(a)

while (i < N)

if (binsearch.res*(-a*[i],*a*) > i)

cnt = cnt + 1

endif

i = i + 1

endwhile

Provide an analysis of *PairSumFast* by first arguing that the algorithm is correct, then specifying running times using Ο(N), Ω(N), and Θ(N) notations. (you will need to research the running time of *mergesort* and *binary search* to answer this question). (15 pts)

1. (2A) We can use the same ideas implemented in *PairSumFast* to improve the Ο(N3) running time of *TripleSum* by recognizing that the pair a[i] and a[j] are part of a triple that sums to zero iff the value **-(a[i]+a[j])** is in the array (and not a[i] or a[j]). Develop an algorithm **TripleSumFast** that incorporates these new ideas. Provide an analysis of *TripleSumFast* by first arguing that your algorithm is correct, then specifying the running time using Ο(N), Ω(N), and Θ(N) notations. (20 pts)
2. When analyzing an algorithm, we often assume that the order of growth of the program’s running time is insensitive to the input values. Consider the *TripleSum* algorithm from Question 1, and assume that the problem we are solving is: *Does the input contain a triple that sums to 0?* How does this change the algorithm? What conclusions can you draw from analyzing this new algorithm? (5 pts)
3. Provide a proof that the number of different triples that can be chosen from *N* items is precisely given in (1.1) above. *Hint: use induction and 1.2 above* (10 pts)

Develop an “electronic packet” of this assignment by providing the exercise number and description, followed by your answer. Start each question on a new page and include a header at the top of each page that contains your name and program track. All exercise answers should be submitted in ***one electronic file*** and submitted to the D2L Assignments file labeled ***Assign3***.