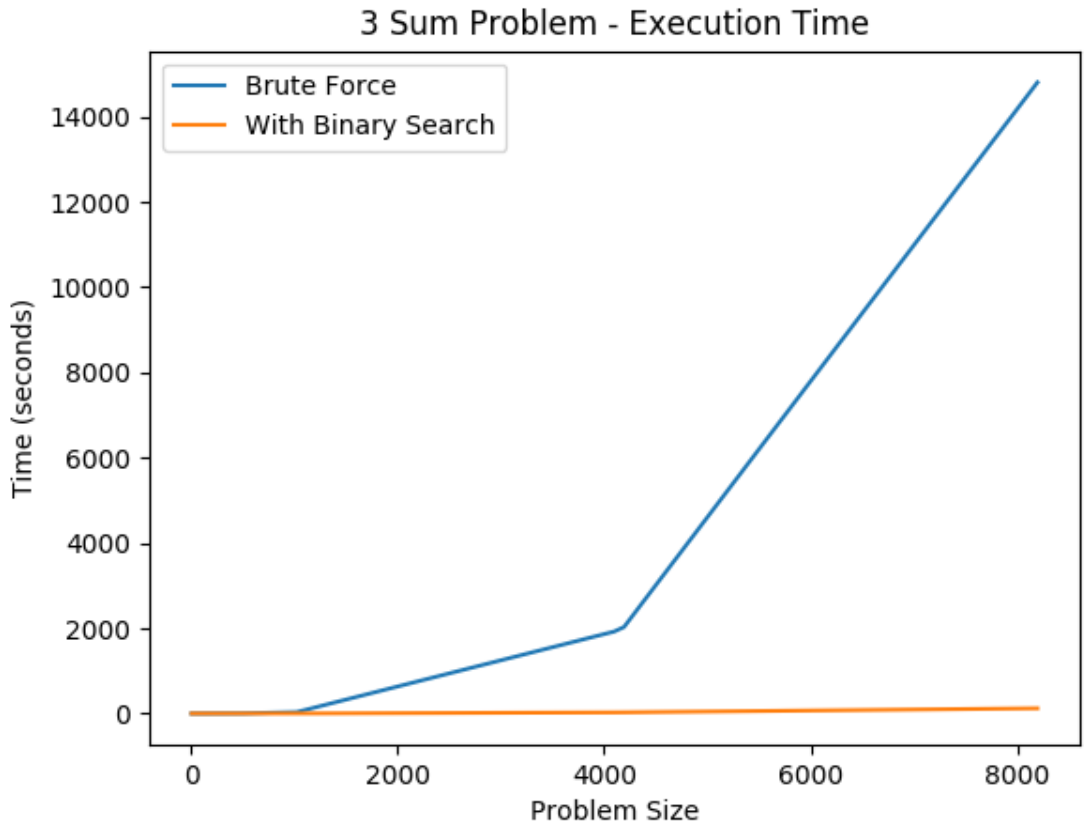
Yilin Yang

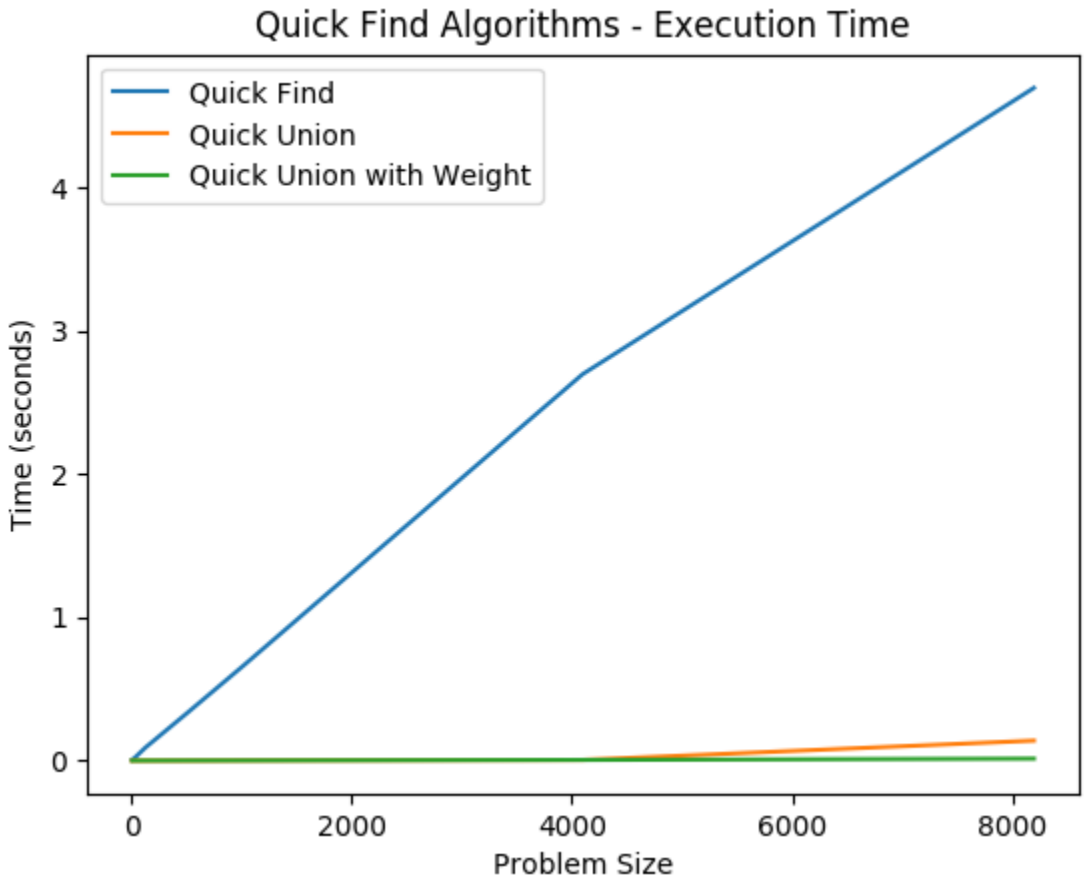
Data Structures HW 1 Report

Q1.



Shown above is the execution time for two solutions for solving the 3 Sum Problem. The objective of the problem is to find, given a list of N integers, a set of three integers whose sum is equal to zero. Two approaches were used; (1) a brute force strategy and (2) a refined binary search strategy. In the brute force strategy, the performance efficiency is of N^3 as the strategy iterates through the list of size N three times in search of integers that sum to zero. Similar to the brute force strategy, the binary search strategy also iterates through the list, however it only does so twice. In place of the third iteration, a binary search locates an integer that is the opposite of the sum of the two integers found by the two prior loops. The elimination of the third iteration already reduces the complexity from N^3 to at most N^2. However, we must take into account the performance cost of the binary search. The complexity of the binary search (as a divide and conquer strategy) is logarithmic since for each iteration of the search, half the problem size can be eliminated. The number of new operations that need to be performed with each new addition to the problem size increases at a slower rate than the problem size itself, therefore the complexity is $N^2\log(N)$.

Q2.



Shown above is the execution time for three variations of the Quick Find Algorithm. The objective of the problem is to determine if two points are connected and, if they are not, to connect them. Three approaches were used; (1) default Quick Find, (2) Quick Union, and (3) Quick Union with weighted decision making. For all implementations, the performance efficiency can be described as either N or log(N). Regardless of the problem size, each strategy performs approximately the same number of operations (in the beginning, no points are connected, however later in time some points may have connections and no union operations will be necessary). The extent of their operations will vary however. In Quick Find for example, every element will have their index updated when their root has their connection updated. This requires access to every element in the list of points, or N accesses, requiring a relatively large amount of time.

Quick Union avoids this by only updating a single index whenever a root changes. However, in the worst case with a very tall tree, this may still require N accesses. Using a weighting system, it is possible to avoid building large trees by favoring uniting smaller trees into larger trees, creating wider trees as a result. This lowers the accesses to log(N).

Q3.

Let us formally define Big Oh notation as F(N) <= c * G(N) for all values where N > Nc. We may use this to calculate values for c and Nc for various growth rates.

For Q1, we have growth rates of N^3 and $N^2\log(N)$. N^3 is trivial as it is not a sum or product of several terms, therefore no simplification is needed; the trend of N^3 is N^3. So, our F(N) and G(N) are both N^3. There are no coefficients at all in this case, so our c value must be 1, and consequentially, so is our Nc value. For $N^2\log(N)$, some simplification is required. The growth rate is a product of two terms, N^2 and log(N), of which N^2 is larger and will therefore emerge as the dominant trend given a large enough problem size. Here our F(N) and G(N) become $N^2\log(N)$ and N^2 respectively. To satisfy the relation described in the definition, c must be log(N). This holds true for Nc = 1.

For Q2, the procedure remains the same. For Quick Find and Quick Union, their growth rates are linear while Quick Union with weighting is log(N). With unspecified coefficients (Quick Find has a faster growth rate than Quick Union despite both being linear) a definite value. We can assume that Quick Find, Quick Union, and Quick Union with weighting have c values of x, y, and 1, respectively, where x > y > 1. Nc = 1 for all cases.

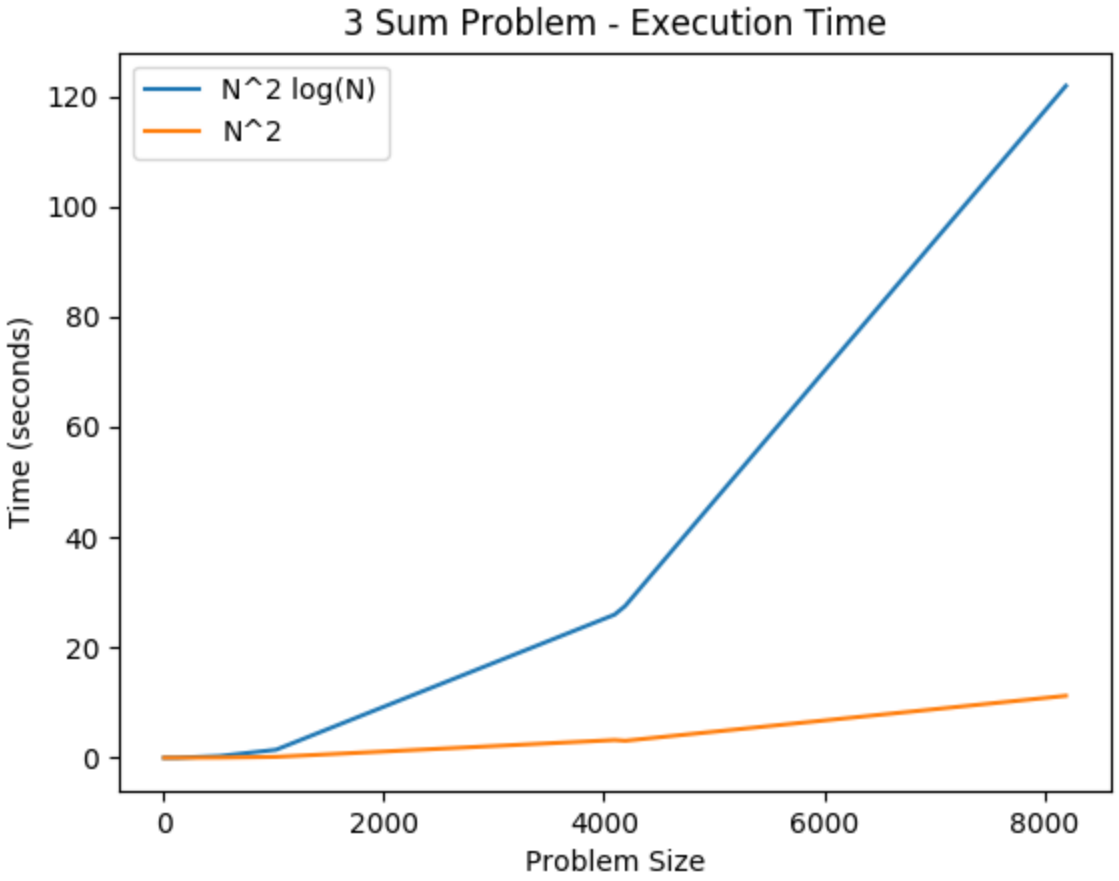
Q4.

A program was developed to locate the farthest pair of double values in a given list such that their difference is the largest possible out of all provided numbers. This is done through a single for loop that accesses each element in the list once and records the smallest and largest numbers encountered and subtracts them. The performance is linear N as a result; the number of operations performed is directly proportional to the problem size.

Q5.

A quadratic implementation for the 3 Sum Problem was developed, the execution time of which is shown below compared to the binary search implementation. The brute force strategy had a performance of N^3 as it iterated through the list of numbers three times by using three for loops. The binary search strategy had a performance of $N^2\log(N)$ as it iterated through the list of numbers only twice. However, it still had to factor the cost of the binary search itself, which was log(N). The implementation specifically used the two for loops to locate two numbers, then used binary search to find a third number that summed to the opposite of those two numbers.

For the quadratic solution, a solution was first developed for a 2 sum problem: finding two numbers who sum to zero. This solution was linear; a single loop took a number from the first and last index of the list and checked if their sum was zero and, if not, repeated using incremented indices. This 2 sum problem solver was nested into a larger for loop that iterated through the list of numbers. Instead of summing to zero, the 2 sum problem solver instead solved for a value equal to the opposite of the number indicated by the parent loop. Thus using only two iterations, the 3 sum problem was solved with a performance of N^2.


3 Sum Problem - Execution Time
N^2 log(N)
N^2
Time (seconds)
Problem Size
0
20
40
60
80
100
120
0
2000
4000
6000
8000