CS 351 - Assignment 2

Petros Blankenstein

Complexity Analysis:

Time Complexity Derivation - When n = 50, the comparison count is 19600. The triple for loop goes through n, then n-1, then n-2, so n(n-1)(n-2), and then we divide that by 6 because we aren't accounting for the order of the numbers. This gives us (n(n-1)(n-2))/6). Multiplying the items in the numerator together, we can get (n-1)(n-2) = n^2 - 3n + 2, then we multiply that by the final n to get n^3 - 3n^2 + 2n. To get the big O notation, we want to take the highest order operation, which is n^3, so the big o notation is O(n^3).

Space Complexity Analysis - The space complexity is simple, we don't copy the array or save any parts of it we just loop through it, so complexity is O(1). The only space we're using is saving integer variables such as "count", "n", "comparisons", "i", etc..

Best, Average, and Worst Case Analysis -

Performance Experiments:

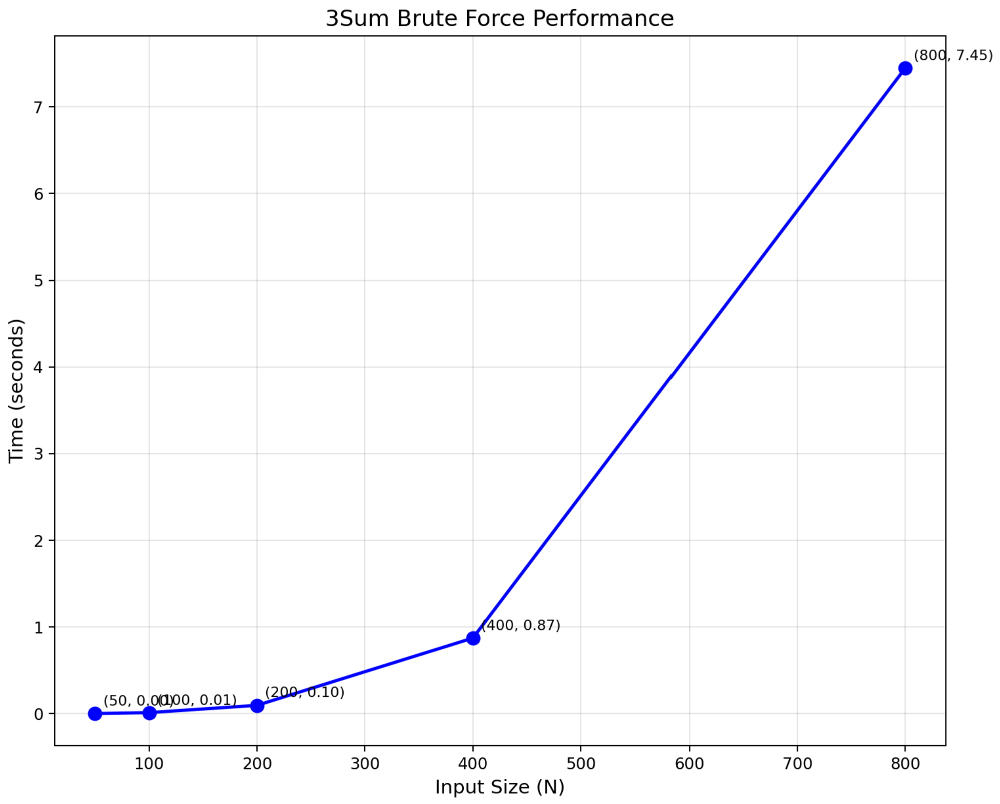
Test Data Generation - You can input a random seed at the beginning, but it defaults to "54321" as the seed (this is what I used for all tests). The arrays inputted were random, using randint with a range of -100 to 100.

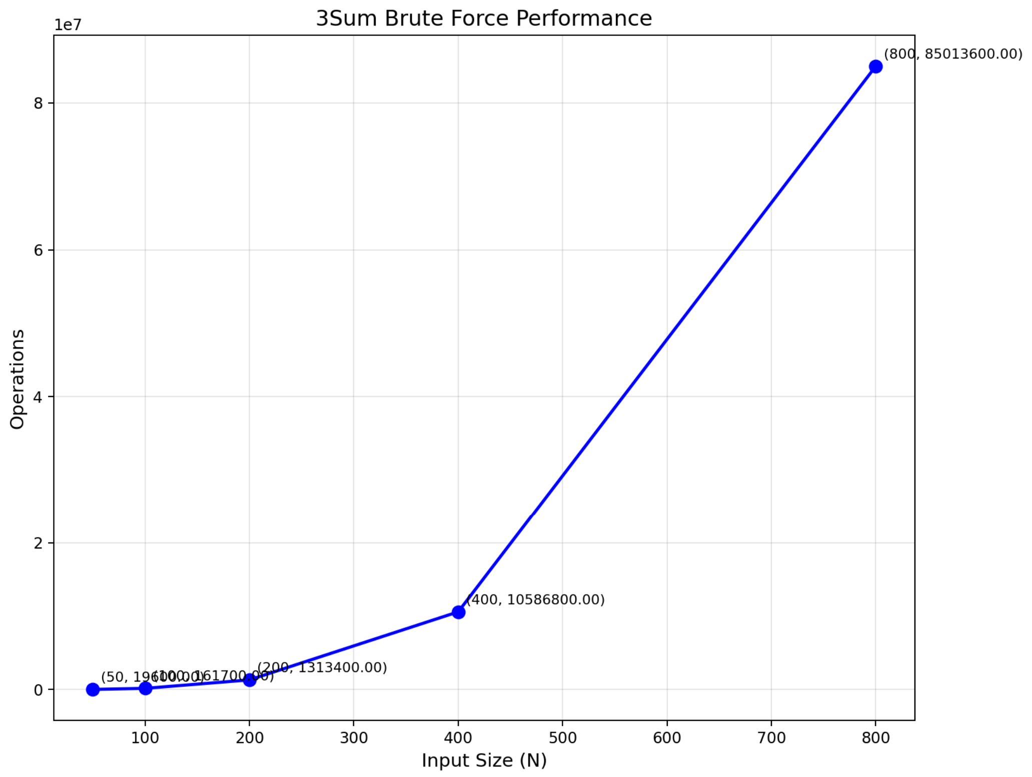
Runtime Measurements -

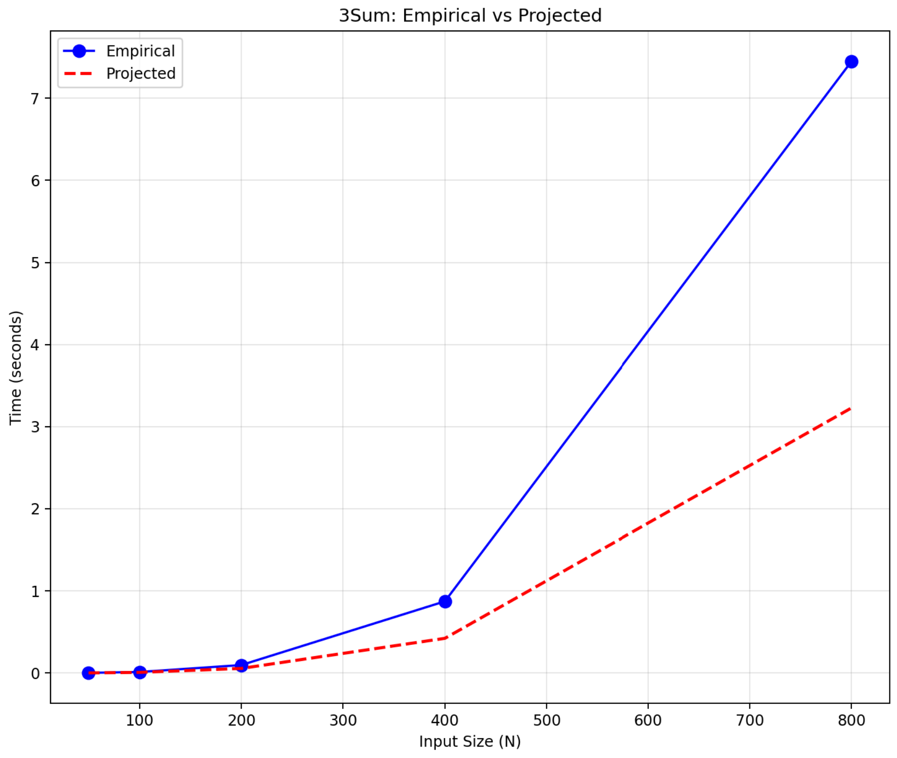
Array Size	Runtime	Operations Count
50	0.0015897400036919862	19600

100	0.012163489995873532	161700
200	0.09623004999593832	1313400
400	0.8723295600007986	10586800
800	7.448667669997667	85013600

Visualization and Analysis -




3Sum Brute Force Performance
1e7
Operations
Input Size (N)
(50, 19600.00)
(100, 161700.00)
(200, 1313400.00)
(400, 10586800.00)
(800, 85013600.00)
0
2
4
6
8
100
200
300
400
500
600
700
800


3Sum: Empirical vs Projected
Empirical
Projected
Time (seconds)
Input Size (N)
0
1
2
3
4
5
6
7
100
200
300
400
500
600
700
800

Theoretical Comparison with Optimized Approaches:

Sorted Array + Two Pointers - First sort the array from smallest to largest. Iterate through the array, and have 2 pointers, one at the beginning of the array and one at the end. If adding those three numbers together, the number is larger than 0, move the leftmost pointer to the right by one, if it's smaller than 0, move the rightmost pointer to the left by one. The complexity is O(n^2) because you're looping through the array which is O(n) time, and then you're moving the pointers through it which is also O(n) time. Doing pointers for each of the array loops is O(n^2) because you're doing an O(n) complexity task for each O(n) complexity task. If n = 1000, Two pointers would take around 1,000,000 operations while the brute force method would take around 1,000,000,000 operations.

Hash Tables - First create the dictionary with the frequency of each number that appears in the array. Iterate through the array, and then use the dictionary like 2sum to find two numbers that when added together equal the number you add to the iterating number to get 0. Time complexity is O(n^2) because we're looping through the array and we're checking through the dictionary which are both O(n) time. However, for space, hash tables take up more memory than the other two methods as you have to create and store a dictionary with each number that appears in the array and how many times it does so. Between time and space, with a powerful machine you would rather do things in less time because usually you have enough memory to not have to worry about it as much, but with a less powerful or more limited machine, you may not have the option to use that much memory.

Reflection and Conclusions:

The brute force method quickly becomes impractical as it loops through the array three times (O(n^3) time complexity) to check each item in the array. It quickly slows down, taking nearly a second to sort an array of only 400 items and nearly seven and a half seconds for an array of 800 items. I can imagine that going any higher would turn into an actual unreasonable amount of time approaching hours or days. My empirical results did not perfectly match the theoretical predictions. As we can see in the "3Sum: Empirical vs Projected" graph. Differences could be attributed to things such as a poor implementation on my part (I don't expect this to be the case but you never know), miscalculating the theoretical predictions, algorithmic overhead, hardware performing poorly or throttling, or something else entirely. It is of utmost importance to consider algorithmic complexity before implementing your algorithm because if you implement an algorithm and it ends up having a high time or space complexity, it could be unrealistic to run on your machine or for your use case. Think about your application of the algorithm. What kind of machines will be running it? What constraints might they have? Things like that before you implement an O(n!) algorithm on accident (I'd like to see that) and then complain when it takes too long.