Austin Stau

Professor Scott Graham

CSC-482

November 6, 2021

Lab0x04

The tests with N=<8 is different than the rest because two iterations of the test gave me inconsistent results when forcing triplets, so I made it so that I force the triplets into specific elements in the array until N > 8. The issue was that there were so few elements to randomly take from that it would often overwrite the previous one. So, for this I made it so that elements one, two and three were chosen, but again just for when N=<8. Generally, I show small scale verification tests by eye to see what the results are. For my large scale you can see in the time tests the triplets that were found, and the number of trials and every trial is used with same -max value and max value with the same number of forced triplets.

**Brute Force:**

For verification purposes I am using the same list of numbers, just goes from 1 to N, and odd numbers are made negative. This is to make sure I’m getting the correct results.

Text

Description automatically generated with low confidence

Here is the small test verification, I had a list of size n every even number I negate and then I shuffle everything.

Here is N = 32

A picture containing text

Description automatically generated

From these verification tests we can see that on a small scale this does work, now I will give the full-time test with all the qualifications. I show the count of each triplet success as well. I will be using a high number as well for the test so in the range of -1000 to 1000.

Graphical user interface

Description automatically generated with medium confidence

With this time test we can see that it is hitting the expected doubling ratio of 8. I evaluated this to t(n)~c\*n^3 because it is three nested for loops which gives us the n^3. My theoretical doubling ratios are calculated by n^3/(n/2)^3. To justify this, I have three for loops each at T(n)~c\*n so n\*n\*n would give me n^3.

**Fast Algorithm:**

For this algorithm I started off with trying to use recursion and two nested loops which would give me t(n)~c\*n^2 log(n).

So, my first algorithm had two nested loops as normal, but then used the third array addition in the if statement look at arr[n]. Then the recursion part comes in with n-1 if n > 1. For verification we can see if it works:

Text

Description automatically generated

You can see there is some issues and positives, it didn’t find the first triplet sum, but found more in the second one. Found more in the N = 16 as well.

So, I am finding the same triplets with the forced list of numbers for the small-scale tests, we can see that they are the same so theoretically these should work well. Let’s check a time test for these and see what our timing is, this is using random numbers now and generating three forced triplets:

Text

Description automatically generated with medium confidence

So, with this time test I was getting the same exact time, I am thinking this is because I was basically doing the same thing as three nested loops, but with just recursion.

Here was the code I used:

Text

Description automatically generated

My second idea was to use a binary search along with two nested loops, this should give me my time complexity of n^2\*log(n). Also looking at some tips given in the discord helped me come to this conclusion.

Unfortunately, in my experimenting I tried the recursive use of binary search and that worked but only let me print out one triplet to see my data. Also tried the iterative without recursion but was also left with issues. So, this was not going to work out.

Onto my final idea which was to use two nested loops and then use an interesting approach I saw online which was found here: <https://tutorialspoint.dev/data-structure/hashing-data-structure/find-triplets-array-whose-sum-equal-zero>

Here they show C++ version where they use hashing, so I tried a modified version of this to get values. Where I take the negative of the addition of 2 numbers from elements defined by the two loops in the main array. Once I have the negative, I have another array that checks to see if the negation of these two addition is present inside of it. If not, it adds the value of the second number into its array and does this again. Here are some verification steps:

Text

Description automatically generated

You can see that not only does it find triplets but unique ones at that. For the time test here are my results:

Graphical user interface

Description automatically generated with low confidence

At N = 32768 I get a doubling ratio of over 8 for some reason, but for everything else I am around the mark of little above 4, sometimes less. To calculate my experimental time I used (n^2\*log(n))/((n/2)^2\*log(n/2)), this gave me a little above 4 in my ratio. To justify this, I have two for loops which would consist of the n^2, then the log(n) would come from the pseudo hashing part where I take sections and fill an array. This doesn’t quite take n time, but my results are showing an n^2 time. This could be just variation. I can see that I am finding less triplets than my previous brute force method, I think that may be because I am finding unique triplets now instead of with brute force method which found duplicates.

**Fastest Algorithm:**

For this I tried modifying my previous one, but I really came up with nothing, it was running quite efficient in my opinion. So, I tried my hand at finding something that only used one for loop instead of two. I first intended doing some sort of recursion, but recursion was getting messy when trying to do a continuous finding of triplets. It worked fine if I just needed to find one, but I wanted to find all.

Then, I looked online to find a solution and I came across this website:

<https://www.techiedelight.com/find-triplet-given-with-given-sum/>

This had a perfect algorithm for the solution I wanted. Some insights I missed when I was trying this on my own was that I could approach this problem in a few ways. In the beginning I was trying to just find the solution in terms of n^2, but there were more unique ways to approach this. This works similar to the previous pseudo hashing one where it finds the remaining value

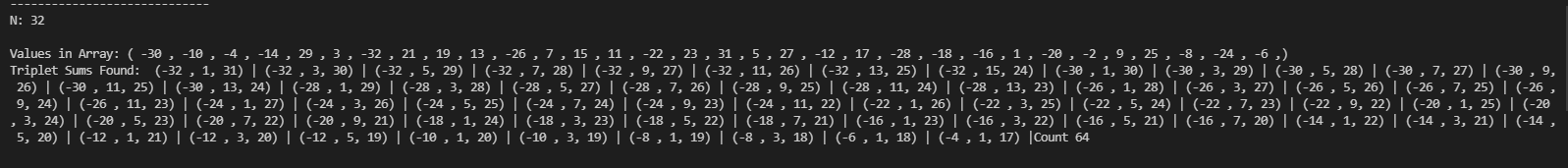
For verification I have:

Text

Description automatically generated

This show actually more triplets that it found than the previous algorithm, and there doesn’t seem to be any duplicates at least on the small scale.

Here is N = 32 for this list as well:



So, this algorithm is finding more valid triplets and at a quick glance they look to be unique.

Here is the time test:

Graphical user interface

Description automatically generated with medium confidence

We can see this runs actually below the 4.0 expected doubling ratio just like my last algorithm, but is finding more triplets and as we saw in the small verification tests this turned out to be true as well. So, this algorithm is probably the most consistent and the fastest out of the bunch.

To calculate this, I used (n^2)/(n/2)^2. I justify this by having one for loop that iterates at T(n)~c\*n , the while loop is basically the same as another for loop running T(n)~c\*n thus making this T(n)~c\*n^2.

|  |  |  |  |
| --- | --- | --- | --- |
|  | **Brute 3sum** | **Faster 3sum** | **Fastest 3sum** |
| **N** | **Time** | **Time** | **Time** |
| 4 | Nan | Nan | Nan |
| 8 | Nan | 2.937500 | 2.133333 |
| 16 | 3.764706 | 2.680851 | 1.875000 |
| 32 | 10.687500 | 2.0 | 4.266667 |
| 64 | 8.243055 | 3.015873 | 2.937500 |
| 128 | 7.268852 | 2.652632 | 4.638298 |
| 256 | 7.439024 | 3.829329 | 4.426564 |
| 512 | 7.680667 | 3.789116 | 3.941364 |
| 1024 | 8.348754 | 3.769231 | 3.623632 |
| 2048 | 7.872123 | 3.90000 | 2.906448 |
| 4096 | 8.020569 | 3.313953 | 3.477204 |
| 8192 | 7.934227 | 4.122807 | 3.500000 |
| 16384 | 0 | 3.727600 | 3.794872 |
| 32768 | 0 | 8.458904 | 3.587838 |
| 65536 | 0 | 3.862888 | 4.133710 |
| 131072 | 0 | 3.909516 | 3.779727 |

For the brute force anything past 8,192 was taking a very long time, which you can check in the testing above. Everything else I went to 131,072 which gave me good results.