Xin Yang (xy213)

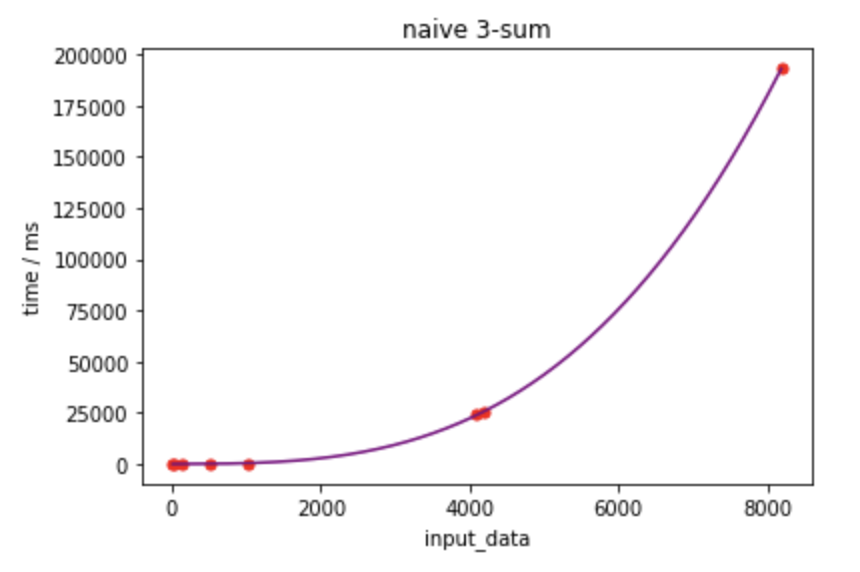
Homework 1 Report + Question 3

**Q1:**

Naive 3-sum:

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Size | 8 | 32 | 128 | 512 | 1024 | 4096 | 4192 | 8192 |
| Time/ms | 0 | 2 | 21 | 74 | 425 | 24180 | 25338 | 193000 |

Table 1 Naive 3-sum



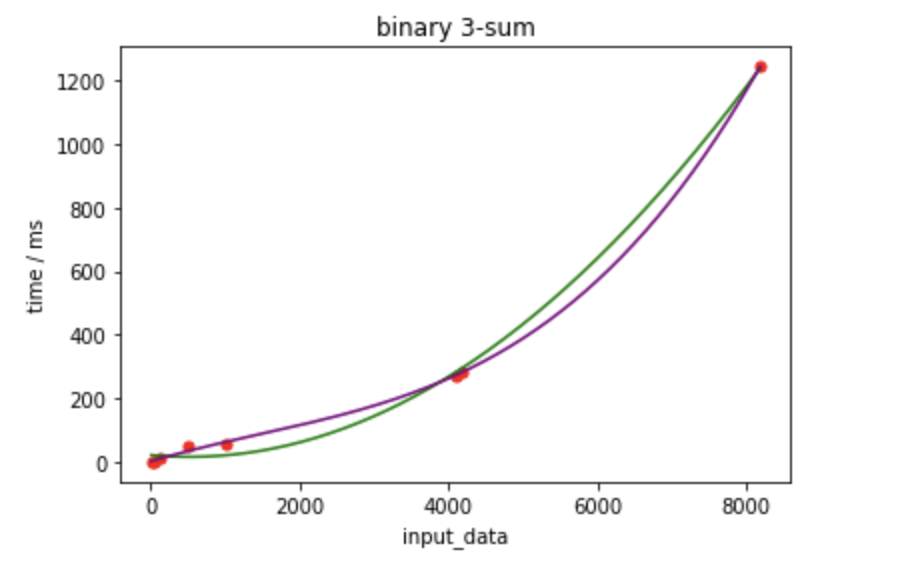
The time complexity is O(N³), as there are three nested loops and all other operations are O(1) and can be omitted when n becomes extremely large.

And the curve optimized as a cubic equation perfectly fits the time consumption dots.

Binary 3-sum:

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Size | 8 | 32 | 128 | 512 | 1024 | 4096 | 4192 | 8192 |
| Time/ms | 0 | 0 | 11 | 52 | 54 | 271 | 285 | 1243 |

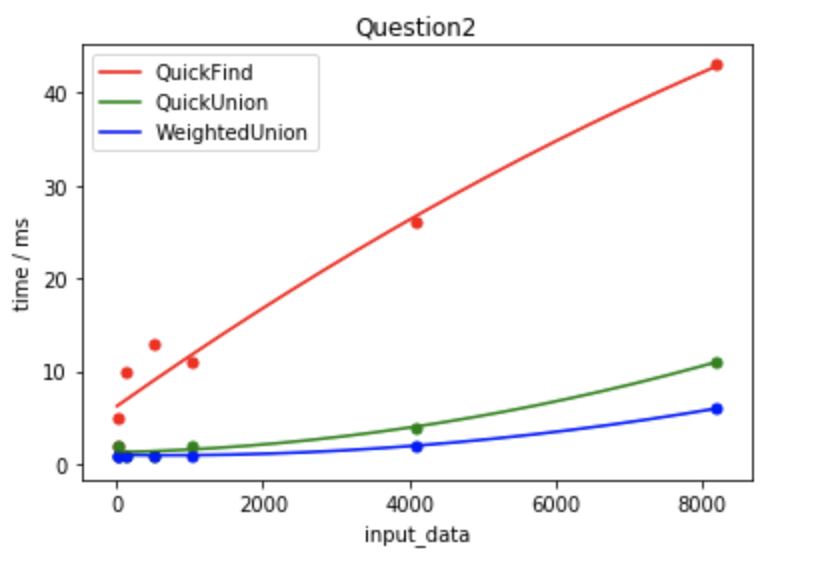
Table 2 Binary 3-sum



The binary 3-sum implemented the binary search to find the last digit, as binary search has a O(log(N)) complexity, the overall complexity is N²\*log(N), but since there can be duplicate results, we still need a loop inside to count how many numbers are the same as the third number, if there’re no duplicates, then the result is O(N²log(N)).

I plot two curves, cubic equation and quadratic, as can be seen, the dots are separated between the two curves, which proves the complexity is between O(N²) and O(N³), and fits O(N²log(N)).

**Q2:**



|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| Size | 8 | 32 | 128 | 512 | 1024 | 4096 | 8192 |
| QuickFind | 2 | 5 | 10 | 13 | 11 | 26 | 43 |
| QuickUnion | 1 | 2 | 1 | 1 | 2 | 4 | 11 |
| WeightedUnion | 1 | 1 | 1 | 1 | 1 | 2 | 6 |

Table 3 Union Find

Quick Find:

The complexity of find is O(1) as the corresponding id[i] is the root of I, and the union costs O(N) to traversal the whole list to find the elements’ root which need to be changed. The overall complexity is O(Nn) but the figure didn’t fit well.

Quick Union:

The find function of quick union is O(N) as the worst case if all elements are connected as a linked list, under this circumstance, the find operation will go through all elements until it comes to the very first one, which is the root. The union function has also O(N) as the worst case to find out the root and modify the value. And the overall complexity is also O(Nn).

Weighted Union:

Weighted quick union can dynamically add small trees into large trees, which can make the tree grow horizontally instead of vertically, in this way the find operation will have a shorter average performance since the tree with more roots have less depth. Since union is also related to find operation, both complexities for find and union are O(lgN) in that the depth of tree will always smaller than lg(N). The overall result is O(nlgN).

The three curves can be plotted and compared in one image. We can tell that there’re deviations at the very beginning because the datasets are quite small and other uncontrollable factors can fluctuate the result. Though the overall trends are still correct with quick find took the most time and increases much greater than quick union and weighted quick union. The quick union is slower than weighted quick union, and this also meets the theory.

**Q3:**

Q1:

Naive Search:

The assignment in three loops are 3\*O(1), and the three nested loop each iterate n - 2 times, and assume i = 0, j from 1 to n - 2, there’re (n - 1)(n - 2)/2 possible ways of k, then we change i into 1, 2, …, n - 3, similarly we can find how many times the assignment sentences in the second loop, which is n\*(n - 1)/2!, and for the first loop is n, and let’s assume the sentence inside the if is x, we can get the f(n) as:

f(n) = 3 + n + 2n(n - 1)/2 + 3((n-1)(n-2) + (n-2)(n-3) + … + (n - (n - 2))(n - (n-1)))/3! + x

= 3 + n + n² - n + (n - 2) \* (n - 1) \* n/2 + x

= 3 + n + (n² - n) + (n^3 - 2 \* n² + 2) /2 + x

= n³/2 + 3 + x

~n³/2

~ O(N³)

Since g(n) = n³, c can be 1/2 as the coefficient of n³ in f(n) is 1/2, and when Nc is larger than 2, O(g(n)) will always no less than f(n).

Binary Search:

For the worst case, I’ll assert there will be no duplicates. Number of O(1) operations is 2, assignment of left and right is 2(n\*(n-1)/2) = n² – n. The most inner while is binary search, which has log(N) complexity, and add to the above situation, we can get:

f(n) = 2 + (n² - n)O(logN)

= n² \* O(logN) -nO(logN) + 2

~ O(n²logN)

Thus g(n) is n²logN, and c is 1, Nc is 2 still.

Q2:

In the worst case, each pair need to be connected. In this problem, N is 8192, and n is the number of pairs waiting for union.

Quick Find:

The find operation is O(1) and there’s nothing to talk about, the union we can find 3 O(1) operations and 3 operations in one loop, and the range is from 0 to n – 1,

Overall: n times traversal and in each round run O(1) find and O(N) connect,

f(n) = n \* (1 + N)

= n + Nn

~ O(Nn)

So c is (1 + N) and Nc is 1 + N.

Quick Union:

The find operation is O(N) by a simple while loop, and both in find and union there’re twice computation of finding roots.

Overall:

f(n) = n (2N + 2N)

= 4Nn

~ O(Nn)

C is 4 and Nc is 4N.

Weighted Union:

Both find and union have O(lgN) complexity because it’s closely related to the depth of tree, and the depth is no more than lgN supposing there’re N nodes.

f(n) = n(2lgN + 2lgN)

= 4nlgN

~ O(nlgN)

c = 4 and Nc is 4lgN.

Q4:

The algorithm is to maintain a max value and a min value, and we can get the max and min by going through the array only once, the (min, max) is the farthest pair. The time complexity is O(N).

Q5:

This is a O(N²) algorithm, and the main idea is to first fix the first number by traverse the array once, during each time of this traversal, set two pointers left and right, which left is initially the one after the first number, and the right is the last one of array. If the sum of first number, left and right is bigger than target, than we can move right forward to decrease the sum, opposite is the same, until left and right meet. The encounter of left and right is O(N), and the first number will also run O(N), thus this algorithms is O(N²).