

Contents

I	Arrays	3
1	1. Two Sum	5
1.1	Problem Statement	5
1.2	Analysis	5
1.3	Solution	5
1.3.1	C++	5
1.3.1.1	$O(N^2)$ Time (35.43%)	5
1.3.1.2	$O(N \log N)$ Time (99.24%)	6
2	461. Hamming Distance	9
2.1	Problem Statement	9
2.2	Analysis	9
2.3	Solution	9
2.3.1	C++	9
2.3.1.1	Time(14.63%)	9
2.3.1.2	Time(94.5%)	10
2.3.1.3	Questions	10
2.3.2	Python	10
2.3.2.1	Faster than 97.37%	10
3	477. Total Hamming Distance	11
3.1	Problem Statement	11
3.2	Analysis	11
3.2.1	First Attempt (too slow)	11
3.2.2	Grouping	12
3.3	Solution	12
3.3.1	C++	12
3.3.1.1	Not Accepted (too slow)	12
3.3.1.2	Time (6.59%) Space (5.13%)	13
3.4	todos [0/4]	14
4	771. Jewels and Stones	15
4.1	Problem Statement	15
4.2	Analysis	15
4.3	Solution	15

4.3.1	C++	15
4.3.1.1	N^2 Time (96.35%) Space (79.64%)	15
II	Trees	17
5	104. Maximum Depth of Binary Tree	19
5.1	Problem Statement	19
5.2	Analysis	19
5.2.1	Recursion	19
5.3	Solution	19
5.3.1	C++	19
5.3.1.1	Recursion. Time (88.44%) Space (91.28%)	19
5.4	todos [/]	20
6	617. Merge Two Binary Trees	21
6.1	Problem Statement	21
6.2	Analysis	21
6.2.1	Approach #1 Recursive Method	21
6.2.2	Approach #2 Iterative Method	21
6.3	Solution	21
6.3.1	C++	21
6.3.1.1	Recursion Time (97.09%) Space(37.01%)	21
6.3.1.2	Iterative	22
6.4	todos [/]	22

Part I

Arrays

Chapter 1

1. Two Sum

1.1 Problem Statement

[Link](#)

1.2 Analysis

1.3 Solution

1.3.1 C++

1.3.1.1 $O(N^2)$ Time (35.43%)

Idea: traverse the vector. For each encountered value, calculate the corresponding value it needs to add up to the target value. And then traverse the vector to look for this value.

The time complexity is $O(N^2)$, because for each value in the vector, you'll go through the vector and search its corresponding part so they add up to the target. This is linear searching, which has $O(N)$ complexity.

```
1  class Solution {
2  public:
3      vector<int> twoSum(vector<int>& nums, int target) {
4          for (auto i = nums.begin(); i != nums.end(); ++i) {
5              int other_part = target - (*i);
6              auto itr = find(nums.begin(), nums.end(), other_part);
7
8              if (itr != nums.end() && itr != i)
9                  return {static_cast<int>(i - nums.begin()), static_cast<int>(itr -
10                     ↪ nums.begin())};
11      }
```

```

12     return {0, 1};
13 }
14 };

```

1.3.1.2 $O(N \log N)$ Time (99.24%)

Idea: the searching part is optimized. First we sort the vector. In order to keep the original relative order of each element, we sort a vector of iterators that referring each element in the original vector `nums`. Then, we can use this sorted vector to perform binary search, whose time complexity is $\log N$. The total time complexity is reduced to $O(N \log N)$.

I made some bugs when writting this code, because I didn't realize the following assumption:

- duplicates allowed
- each input would have *exactly* one solution

Code:

```

1  class Solution {
2  public:
3      /*Notes:
4       The compare object used to sort vector of iterators
5      */
6      struct Compare {
7          bool operator()(vector<int>::iterator a, vector<int>::iterator b) {
8              return (*a < *b);
9          }
10
11 };
12
13 /*Notes:
14  A binary search to find target value in a vector of iterators;
15  if found: return the index value of that iterator
16  if not found: return -1
17 */
18 int findTarget(int target, const vector<vector<int>::iterator>&
19   ↪ itr_vector, const vector<int>::iterator& current_itr) {
20     int start_index = 0;
21     int end_index = itr_vector.size() - 1;
22     int middle;
23     int result = -1;
24
25     while (start_index <= end_index) {
26         // update middle
27         middle = (start_index + end_index) / 2;
28         // check value

```

```
28     if (*itr_vector[middle] == target) {
29         if (itr_vector[middle] == current_itr) {
30             start_index += 1;
31             end_index += 1;
32             continue;
33         }
34
35         result = middle;
36         break;
37     }
38
39     else if (*itr_vector[middle] > target) {
40         end_index = middle - 1;
41         continue;
42     }
43
44     else if (*itr_vector[middle] < target) {
45         start_index = middle + 1;
46         continue;
47     }
48
49 }
50
51 return result;
52 }
53
54
55 vector<int> twoSum(vector<int>& nums, int target) {
56     // create a vector of iterators
57     vector<vector<int>::iterator> itr_vector;
58     for (auto i = nums.begin(); i != nums.end(); ++i)
59         itr_vector.push_back(i);
60
61     // sort the vector of iterators, so the values these iterators referred
62     → to
63     // are in ascending order
64     sort(itr_vector.begin(), itr_vector.end(), Compare());
65
66     // go over nums, and find the pair
67     for (auto i = nums.begin(); i != nums.end() - 1; ++i) {
68         int other_part = target - (*i);
69         int other_part_index = findTarget(other_part, itr_vector, i);
70
71         if (other_part_index != -1) // found
```

```
71         return {static_cast<int>(i - nums.begin()),  
72                 ↪ static_cast<int>(itr_vector[other_part_index] - nums.begin())};  
73     }  
74     // for syntax  
75     return {0, 1};  
76  
77 }  
78 };
```


Chapter 2

461. Hamming Distance

2.1 Problem Statement

[Link](#)

2.2 Analysis

To compare two numbers bitwisely, we may need the fact that a number mod 2 is equal to the last digit of its binary form. For example:

$x = 1 \ (0 \ 0 \ 0 \ 1)$

$y = 4 \ (0 \ 1 \ 0 \ 0)$

$x \% 2 = 1$

$y \% 2 = 0$

2.3 Solution

2.3.1 C++

2.3.1.1 Time(14.63%)

```
1  class Solution {
2  public:
3      int hammingDistance(int x, int y) {
4          int result = 0;
5
6          while (x != 0 || y != 0) {
7              if (x % 2 != y % 2)
8                  result++;
9
10             x = x >> 1;
```

```

11     y = y >> 1;
12 }
13
14     return result;
15 }
16 };

```

2.3.1.2 Time(94.5%)

```

1  class Solution {
2  public:
3      int hammingDistance(int x, int y) {
4          int result = 0;
5          x ^= y;
6
7          while (x) {
8              if (x % 2)
9                  result++;
10             x = x >> 1;
11         }
12
13         return result;
14     }
15 };

```

2.3.1.3 Questions

Why the second solution is faster than the previous one?

- Bitwise XOR used.

2.3.2 Python

2.3.2.1 Faster than 97.37%

```

1  class Solution:
2      def hammingDistance(self, x: int, y: int) -> int:
3          result = 0
4          while x or y:
5              if x % 2 != y % 2:
6                  result += 1
7              x = x >> 1
8              y = y >> 1
9          return result

```

However, this algorithm is exactly the same as C++'s first version. Why such huge speed variance?

Chapter 3

477. Total Hamming Distance

3.1 Problem Statement

Link

3.2 Analysis

This problem is similar with P461, but you can't directly solve it using that idea (see the first solution). The size of the input is large:

- Elements of the given array are in the range of 0 to 10^9
- Length of the array will not exceed 10^4

3.2.1 First Attempt (too slow)

My first attempt is just go over all the combinations in the input array: (x_i, x_j) and call the function that calculate the hamming distance of two integers (P461), the code is shown in solution section. However, this approach is too slow to pass the test.

The time complexity of the function that calculates the hamming distance of two integers is not huge, just $O(1)$. The real time consuming part is the combination. It is simply:

$$\binom{N}{2} = \frac{N(N-1)}{2} \sim O(N^2)$$

Inside these combinations, we included many bit-pairs that do not contribute to the total Hamming distance count, for example, the combination of number 91 and 117 is:

```
-----  
bit#: 1234 5678  
-----  
91:    0101 1011
```

117: 0111 0101

The bit at 1, 2, 4, 8 are not contributing to the total Hamming distance count, but we still include it and spend time verifying. This flaw can be solved in the grouping idea.

3.2.2 Grouping

Reference

The idea of grouping is we count the total hamming distance as a whole. Specifically, at any giving time, we devide the array into two groups G_0, G_1 . The rule of grouping is:

- a number n that $n\%2 = 0$, goes to G_0
- a number n that $n\%2 = 1$, goes to G_1

The result of $n\%2$ will give you the least significant bit, or the last bit of an integer in binary form. By the definition of Hamming distance, we know that any combinations that contains number pairs only from G_0 or only from G_1 will not contribute to the total Hamming distance count (just for this grouping round, which only compares the least significant bit of those numbers). On the other hand, any combination that contains one number from G_0 and one number from G_1 will contribute 1 to the total Hamming distance. So, for this round, we only have to count the combination of such case, which is simply:

$$N_{G_0} \times N_{G_1}$$

Then, we trim the current least significant bit and re-group the numbers into new G_0 and G_1 . This is because at each bit the numbers are different. We do this until **ALL** numbers are **ZERO**. For example, if at one round, there are no numbers in G_1 , all numbers are in G_0 , then although the contribution to total Hamming distance of this round is zero, we have to move on to trim the least significant bit and re-group the numbers.

3.3 Solution

3.3.1 C++

3.3.1.1 Not Accepted (too slow)

This algorithm is too slow.

```

1  class Solution {
2  public:
3      int hammingDistance(const int& x, const int& y) {
4          int result = 0;
5          int a = x ^ y;
6
7          while (a != 0) {
```

```

8         if (a % 2)
9             result++;
10        a = a >> 1;
11    }
12
13    return result;
14 }
15
16 int totalHammingDistance(vector<int>& nums) {
17     int count = 0;
18     for (int i = 0; i < nums.size() - 1; ++i) {
19         for (int j = i + 1; j < nums.size(); ++j)
20             count += hammingDistance(nums[i], nums[j]);
21     }
22     return count;
23 }
24 };

```

3.3.1.2 Time (6.59%) Space (5.13%)

This is the first version after I read and apply the idea of grouping numbers with different Least Significant bit. Although it is still slow, it is accepted....

```

1  class Solution {
2  public:
3      int totalHammingDistance(vector<int>& nums) {
4          vector<int> LSB_ones;
5          vector<int> LSB_zeros;
6          int count = 0;
7          int non_zero_count = 1; // loop continue until no non-zero num in nums
8
9          while (non_zero_count) {
10             // clear temp container, reset non-zero count
11             LSB_ones.clear();
12             LSB_zeros.clear();
13             non_zero_count = 0;
14
15             // collect number, divide into two groups
16             for (auto& i : nums) {
17                 if (i % 2 == 0)
18                     LSB_zeros.push_back(i);
19                 else
20                     LSB_ones.push_back(i);
21
22                 // update i and non_zero_count

```

```
23         i = i >> 1;
24         if (i)
25             non_zero_count++;
26     }
27
28     // update count
29     count += LSB_ones.size() * LSB_zeros.size();
30 }
31
32 return count;
33 }
34 };
```

3.4 todos [0/4]

- ☐ Write the analysis of grouping idea and my code
- ☐ Read code in reference of grouping idea, make notes
- ☐ Check other possible solution and make future plan
- ☐ Try to generalize this problem

Chapter 4

771. Jewels and Stones

4.1 Problem Statement

[Link](#)

4.2 Analysis

4.3 Solution

4.3.1 C++

4.3.1.1 N^2 Time (96.35%) Space (79.64%)

```
1  class Solution {
2  public:
3      int numJewelsInStones(string J, string S) {
4          int numJewl = 0;
5          for (auto s : S)
6              if (isJewels(s, J))
7                  numJewl++;
8          return numJewl;
9      }
10
11     bool isJewels(char s, string J) {
12         for (auto j : J)
13             if (s == j)
14                 return true;
15
16         return false;
17     }
18 };
```


Part II

Trees

Chapter 5

104. Maximum Depth of Binary Tree

5.1 Problem Statement

[Link](#)

5.2 Analysis

5.2.1 Recursion

A node's maximum depth, is the larger maximum depth of its left and right subtree plus one. Base case: if a node is nullptr, maximum depth is zero.

5.3 Solution

5.3.1 C++

5.3.1.1 Recursion. Time (88.44%) Space (91.28%)

```
1  /**
2   * Definition for a binary tree node.
3   * struct TreeNode {
4   *     int val;
5   *     TreeNode *left;
6   *     TreeNode *right;
7   *     TreeNode(int x) : val(x), left(NULL), right(NULL) {}
8   * };
9   */
10 class Solution {
11 public:
12     int maxDepth(TreeNode* root) {
13         // base case
```

```
14     if (root == nullptr)
15         return 0;
16
17     int left_depth = maxDepth(root->left);
18     int right_depth = maxDepth(root->right);
19
20     return (left_depth >= right_depth ? left_depth + 1 : right_depth + 1);
21 }
22 };
```

5.4 todos [/]

- read about the discussion page for more methods and ideas

Chapter 6

617. Merge Two Binary Trees

6.1 Problem Statement

[Link](#)

6.2 Analysis

6.2.1 Approach #1 Recursive Method

Use recursion to solve this problem.

6.2.2 Approach #2 Iterative Method

6.3 Solution

6.3.1 C++

6.3.1.1 Recursion Time (97.09%) Space(37.01%)

Recursion.

```
1  /**
2   * Definition for a binary tree node.
3   * struct TreeNode {
4   *     int val;
5   *     TreeNode *left;
6   *     TreeNode *right;
7   *     TreeNode(int x) : val(x), left(NULL), right(NULL) {}
8   * };
9   */
10 class Solution {
11 public:
```

```
12  TreeNode* mergeTrees(TreeNode* t1, TreeNode* t2) {
13      if (t1 == nullptr)
14          return t2;
15      else if (t2 == nullptr)
16          return t1;
17      else {
18          TreeNode* node = new TreeNode(t1->val + t2->val);
19          node->left = mergeTrees(t1->left, t2->left);
20          node->right = mergeTrees(t1->right, t2->right);
21          return node;
22      }
23  }
24  };
```

6.3.1.2 Iterative

6.4 todos [/]

- ☐ read the other solution (iterate the tree using stack), and understand it
- ☐ write code based on the other solution