Contents

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.

```
class Solution {
public:
  vector<int> twoSum(vector<int>& nums, int target) {
    for (auto i = nums.begin(); i != nums.end(); ++i) {
      int other_part = target - (*i);
      auto itr = find(nums.begin(), nums.end(), other_part);

      if (itr != nums.end() && itr != i)
        return {static_cast<int>(i - nums.begin()), static_cast<int>(itr -
        ↪  nums.begin())};
    }

```

```
    return {0, 1};
  }
};
```

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:

```
class Solution {
public:
  /*Notes:
    The compare object used to sort vector of iterators
  */
  struct Compare {
    bool operator()(vector<int>::iterator a, vector<int>::iterator b) {
      return (*a < *b);
    }

  };

  /*Notes:
    A binary search to find target value in a vector of iterators;
    if found: return the index value of that iterator
    if not found: return -1
  */
  int findTarget(int target, const vector<vector<int>::iterator>&
  ↪  itr_vector, const vector<int>::iterator& current_itr) {
    int start_index = 0;
    int end_index = itr_vector.size() - 1;
    int middle;
    int result = -1;

    while (start_index <= end_index) {
      // update middle
      middle = (start_index + end_index) / 2;
      // 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
         ↪  to
62      // are in ascending order
63      sort(itr_vector.begin(), itr_vector.end(), Compare());
64
65      // go over nums, and find the pair
66      for (auto i = nums.begin(); i != nums.end() - 1; ++i) {
67        int other_part = target - (*i);
68        int other_part_index = findTarget(other_part, itr_vector, i);
69
70        if (other_part_index != -1) // found
```

```
71          return {static_cast<int>(i - nums.begin()),
            ↪  static_cast<int>(itr_vector[other_part_index] - nums.begin())};
72      }
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%)

```
class Solution {
public:
  int hammingDistance(int x, int y) {
    int result = 0;

    while (x != 0 || y != 0) {
      if (x % 2 != y % 2)
        result++;

      x = x >> 1;
```

```
      y = y >> 1;
    }

    return result;
  }
};
```

2.3.1.2 Time(94.5%)

```
class Solution {
public:
  int hammingDistance(int x, int y) {
    int result = 0;
    x ^= y;

    while (x) {
      if (x % 2)
        result++;
      x = x >> 1;
    }

    return result;
  }
};
```

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%

```
class Solution:
    def hammingDistance(self, x: int, y: int) -> int:
        result = 0
        while x or y:
            if x % 2 != y % 2:
                result += 1
            x = x >> 1
            y = y >> 1
        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 direcly 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

You have to work the way other round.

3.2.1 Grouping

Reference

The idea of grouping.

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) {
```

```
    int result = 0;
    int a = x ^ y;

    while (a != 0) {
      if (a % 2)
        result++;
      a = a >> 1;
    }

    return result;
  }

  int totalHammingDistance(vector<int>& nums) {
    int count = 0;
    for (int i = 0; i < nums.size() - 1; ++i) {
      for (int j = i + 1; j < nums.size(); ++j)
        count += hammingDistance(nums[i], nums[j]);
    }
    return count;
  }
};
```

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.....

```
class Solution {
public:
  int totalHammingDistance(vector<int>& nums) {
    vector<int> LSB_ones;
    vector<int> LSB_zeros;
    int count = 0;
    int non_zero_count = 1; // loop continue until no non-zero num in nums

    while (non_zero_count) {
      // clear temp container, reset non-zero count
      LSB_ones.clear();
      LSB_zeros.clear();
      non_zero_count = 0;

      // collect number, divide into two groups
      for (auto& i : nums) {
        if (i % 2 == 0)
          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 Tasks [1/4]

- ☒ 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%)

```
class Solution {
public:
  int numJewelsInStones(string J, string S) {
    int numJewl = 0;
    for (auto s : S)
      if (isJewels(s, J))
        numJewl++;
    return numJewl;
  }

  bool isJewels(char s, string J) {
    for (auto j : J)
      if (s == j)
        return true;

    return false;
  }
};
```