Contents

I Arrays **7**

1 1. Two Sum **9**
1.1 Problem Statement . . . 9
1.2 Analysis . . . 9
1.2.1 $O(N^2)$ method . . . 9
1.3 Solution . . . 9
1.3.1 C++ . . . 9
1.3.1.1 $O(N^2)$ Time (35.43%) . . . 9
1.3.1.2 $O(N^2)$ modified . . . 10
1.3.1.3 $O(N \log N)$ Time (99.24%) . . . 11
1.4 todos [1/5] . . . 13

2 136. Single Number **15**
2.1 Problem Statement . . . 15
2.2 Analysis . . . 15
2.2.1 Use Unordered-set . . . 15
2.3 Solution . . . 15
2.3.1 C++ . . . 15
2.3.1.1 Unordered-set. time (16.06%) space (15.74%) . . . 15
2.4 todos [0/4] . . . 16

3 169. Majority Element **17**
3.1 Problem Statement . . . 17
3.2 Analysis . . . 17
3.2.1 `unordered_map` . . . 17
3.3 Solutions . . . 17
3.3.1 C++ . . . 17
3.3.1.1 `unordered_map` (59%, 42%) . . . 17
3.4 todos [/] . . . 18

4 283. Move Zeros **19**
4.1 Problem Statement . . . 19
4.2 Analysis . . . 19
4.3 Solution . . . 19
4.3.1 C++ . . . 19

4.3.1.1 Use bubble sort idea. time (5%) space (75%) 19
4.3.1.2 Use erase(), remove zeros. time (35%) space (34%) 20
4.4 todos [0/2] 20

5 442. Find All Duplicates in an Array 23
5.1 Problem Statement 23
5.2 Analysis 23
5.2.1 Label Duplicate Number 23
5.3 Solution 23
5.3.1 C++ 23
5.3.1.1 Label duplicate number (96%, 16%) 23
5.4 todos [/] 24

6 448. Find All Numbers Disappeared in an Array 25
6.1 Problem Statement 25
6.2 Analysis 25
6.2.1 Label Appearance of Numbers 25
6.2.2 Use Unordered-set 25
6.3 Solution 26
6.3.1 C++ 26
6.3.1.1 Label appearance of numbers (97%, 15%) 26
6.3.1.2 Use unordered-set (13%, 7%) 26
6.4 todos [/] 27

7 461. Hamming Distance 29
7.1 Problem Statement 29
7.2 Analysis 29
7.3 Solution 29
7.3.1 C++ 29
7.3.1.1 Time(14.63%) 29
7.3.1.2 Time(94.5%) 30
7.3.1.3 Questions 30
7.3.2 Python 30
7.3.2.1 Faster than 97.37% 30

8 477. Total Hamming Distance 31
8.1 Problem Statement 31
8.2 Analysis 31
8.2.1 First Attempt (too slow) 31
8.2.2 Grouping 32
8.3 Solution 33
8.3.1 C++ 33
8.3.1.1 Not Accepted (too slow) 33
8.3.1.2 Grouping. time (6.59%) space (5.13%) 34
8.3.1.3 Grouping_example. time (88.24%, 49.76%) 35

8.4 todos [3/4] 36

9 581. Shortest Unsorted Continuous Subarray 37
9.1 Problem Statement 37
9.2 Analysis 37
9.2.1 Use sorting 37
9.3 Solution 38
9.3.1 C++ 38
9.3.1.1 use sorting 38
9.4 todos [1/3] 38

10 771. Jewels and Stones 41
10.1 Problem Statement 41
10.2 Analysis 41
10.3 Solution 41
10.3.1 C++ 41
10.3.1.1 N^2 Time (96.35%) Space (79.64%) 41

II Hash Table 43

11 242. Valid Anagram 45
11.1 Problem Statement 45
11.2 Analysis 45
11.2.1 Sort 45
11.2.2 Hash Table 45
11.3 Solution 45
11.4 todos [/] 45

III Linked List 47

12 160. Intersection of Two Linked Lists 49
12.1 Problem Statement 49
12.2 Analysis 49
12.2.1 Brutal force 49
12.2.2 Hash table 49
12.2.3 Skip longer list 49
12.2.4 Two pointer 50
12.3 Solution 50
12.3.1 C++ 50
12.3.1.1 brutal force 50
12.3.1.2 hash table 51
12.3.1.3 skip longer lists 52
12.4 todos [1/3] 53

13 206. Reverse Linked List **55**
13.1 Problem Statement . . . 55
13.2 Analysis . . . 55
13.2.1 Using Stack (my) . . . 55
13.2.2 Recursion (my) . . . 55
13.3 Solution . . . 55
13.3.1 C++ . . . 55
13.3.1.1 Using Stack. time (96%), space (5%) . . . 55
13.3.1.2 Using Recursion. time (18%), space (21%) . . . 56
13.4 todos [/] . . . 57

IV Trees **59**

14 101. Symmetric Tree **61**
14.1 Problem Statement . . . 61
14.2 Analysis . . . 61
14.2.1 Recursion . . . 61
14.3 Solution . . . 62
14.3.1 C++ . . . 62
14.3.1.1 recursion (75%, 64%) . . . 62
14.4 todos [1/5] . . . 63

15 104. Maximum Depth of Binary Tree **65**
15.1 Problem Statement . . . 65
15.2 Analysis . . . 65
15.2.1 Recursion . . . 65
15.3 Solution . . . 65
15.3.1 C++ . . . 65
15.3.1.1 Recursion. Time (88.44%) Space (91.28%) . . . 65
15.4 todos [0/1] . . . 66

16 112. Path Sum **67**
16.1 Problem Statement . . . 67
16.2 Analysis . . . 67
16.2.1 Recursion . . . 67
16.3 Solution . . . 67
16.3.1 C++ . . . 67
16.3.1.1 recursion (88%, 92%) . . . 67
16.4 todos [1/3] . . . 68

17 113. Path Sum II **69**
17.1 Problem Statement . . . 69
17.2 Analysis . . . 69
17.2.1 DFS . . . 69

17.3 Solution . . . 70
17.3.1 C++ . . . 70
17.3.1.1 DFS (80%, 40%) . . . 70
17.4 todos [1/4] . . . 72

18 226. Invert Binary Tree **73**
18.1 Problem Statement . . . 73
18.2 Analysis . . . 73
18.2.1 Recursion . . . 73
18.3 Solution . . . 73
18.3.1 C++ . . . 73
18.3.1.1 Recursion. time (91.95%) space (5.15%) . . . 73
18.4 todos [0/2] . . . 74

19 437. Path Sum III **75**
19.1 Problem Statement . . . 75
19.2 Analysis . . . 75
19.2.1 Double recursion (~50%, 50%) . . . 75
19.3 Solution . . . 76
19.3.1 C++ . . . 76
19.3.1.1 double recursion . . . 76
19.4 todos [1/3] . . . 77

20 543. Diameter of Binary Tree **79**
20.1 Problem Statement . . . 79
20.2 Analysis . . . 79
20.2.1 Direct recursion . . . 79
20.3 Solution . . . 80
20.3.1 C++ . . . 80
20.3.1.1 direct recursion (5%, 5%) . . . 80
20.4 todos [/] . . . 80

21 617. Merge Two Binary Trees **81**
21.1 Problem Statement . . . 81
21.2 Analysis . . . 81
21.2.1 Recursive Method . . . 81
21.2.2 Iterative Method (using stack) . . . 81
21.3 Solution . . . 81
21.3.1 C++ . . . 81
21.3.1.1 Recursion Time (97.09%) Space(37.01%) . . . 81
21.3.1.2 Iterative . . . 82
21.4 todos [0/2] . . . 82

22 938. Range Sum of BST **83**
22.1 Problem Statement . . . 83

22.2 Analysis . . . 83
22.2.1 Recursion (brutal and stupid) . . . 83
22.2.2 DFS . . . 83
22.3 Solution . . . 84
22.3.1 C++ . . . 84
22.3.1.1 recursion (stupid) . . . 84
22.3.1.2 recursion (better) . . . 84
22.3.1.3 DFS (slow, 5% and 6%) . . . 85
22.4 todos [1/3] . . . 86

V Other **87**

23 70. Climbing Stairs **89**
23.1 Problem Statement . . . 89
23.2 Analysis . . . 89
23.3 Solution . . . 89
23.3.1 C++ . . . 89
23.3.1.1 use a vector to hold intermediate result (77%, 64%) . . . 89
23.4 todos [/] . . . 90

Part I

Arrays

Chapter 1

1. Two Sum

1.1 Problem Statement

Link

1.2 Analysis

1.2.1 $O(N^2)$ method

Each input would have exactly one solution, and no same element can be used twice. We can go through the array. For each element we encountered (`nums[i]`), we calculate the counter part (the number that is needed so `nums[i] + counter_part = target`): simply: `target - nums[i]`. Then we go through the rest of the array to find out if such element exist. We go from `i + 1` to the end. We don't have to go from the beginning because if there is such an element, we would find it earlier. If no such element found, we continue to the next element, calculate its counter part and search again.

Strictly speaking, the time required will be smaller than $O(N^2)$ algorithm. The "inner" iteration's size is not N, its size is decreasing.

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.

```cpp
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 $O(N^2)$ modified

This is modified implementation. Although the algorithm is the same as the first $O(N^2)$ solution. This solution is much clearer.

```cpp
/*test cases:
[2,7,11,15]
9

[2,3]
5

[1,113,2,7,9,23,145,11,15]
154

[2,7,11,15,1,8,13]
3
*/


class Solution {
public:
  vector<int> twoSum(vector<int>& nums, int target) {
    int index_1 = -1;
    int index_2 = -1;

    for (int i = 0; i < nums.size(); i++) {
      int other_part = target - nums[i];

```

```
25        for (int j = i + 1; j < nums.size(); j++) {
26          if (nums[j] == other_part) {
27            index_1 = i;
28            index_2 = j;
29            break;
30          }
31        }
32
33        if (index_1 != -1)
34          break;
35      }
36
37      return {index_1, index_2};
38    }
39  };
```

1.3.1.3 $O(N \log N)$ 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>&
    ↪  itr_vector, const vector<int>::iterator& current_itr) {
19    int start_index = 0;
20    int end_index = itr_vector.size() - 1;
21    int middle;
22    int result = -1;
23
24    while (start_index <= end_index) {
25      // update middle
26      middle = (start_index + end_index) / 2;
27      // 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  };
```

1.4 todos [1/5]

- ☒ try sort directly method (using a copy array)
- ☐ write down my own analysis: sort copy array and iter_array
- ☐ check the solution and understands, implement each idea
- ☐ write the analysis of each idea
- ☐ generalize this problem

Chapter 2

136. Single Number

2.1 Problem Statement

Link

2.2 Analysis

2.2.1 Use Unordered-set

2.3 Solution

2.3.1 C++

2.3.1.1 Unordered-set. time (16.06%) space (15.74%)

```cpp
class Solution {
public:
  int singleNumber(vector<int>& nums) {
    unordered_set<int> unique_num;

    for (auto num : nums) {
      auto itr = unique_num.find(num);

      if (itr == unique_num.end())
        unique_num.insert(num);
      else
        unique_num.erase(itr);
    }

    return *unique_num.begin();
  }
```

```
17 };
```

2.4 todos [0/4]

- ☐ write your solution step (in analysis part), analysis time and space complexity
- ☐ think about possible improvements
- ☐ read solution, do additional work (internalize it and write analysis and code)
- ☐ read discussion, do additional work (internalize it and write analysis and code)

Chapter 3

169. Majority Element

3.1 Problem Statement

Link

3.2 Analysis

3.2.1 `unordered_map`

This is a problem that record the frequency of the element. I use the number as key and the appearing times as value, build an unordered_map that store this information. As long as a number's appearing times is more than `size / 2`, it will be the majority element.

3.3 Solutions

3.3.1 C++

3.3.1.1 `unordered_map` (59%, 42%)

Not very fast.

```cpp
class Solution {
public:
  int majorityElement(vector<int>& nums) {
    unordered_map<int, int> frequency_count;

    for (auto num : nums) {
      if (frequency_count.find(num) != frequency_count.end()) {
        frequency_count[num] += 1;
        if (frequency_count[num] > nums.size() / 2)
          return num;
```

```
11        }
12
13        else
14          frequency_count.insert(make_pair(num, 1));
15      }
16
17      return nums[0];
18    }
19  };
```

3.4 todos [/]

- ☐ think about other solution (use about 30 min)
- ☐ read discussion and contemplate other solution
- ☐ generalize the problem

Chapter 4

283. Move Zeros

4.1 Problem Statement

Link

4.2 Analysis

Make sure you know well the problem statement. For example, in this problem, there is no requirement for the zero element be kepted.

4.3 Solution

4.3.1 C++

4.3.1.1 Use bubble sort idea. time (5%) space (75%)

Too slow, time complexity is $O(N^2)$.

```
class Solution {
public:
  void moveZeroes(vector<int>& nums) {
    bool swapped;

    // swap array
    do {
      swapped = false;

      for (auto iter = nums.begin(); iter != nums.end() - 1; ++iter) {
        if (*iter == 0) {
          if (*(iter + 1) == 0)
            continue;
```

```
14            swap(*iter, *(iter + 1));
15            swapped = true;
16          }
17        }
18      } while (swapped);
19    }
20  };
```

4.3.1.2 Use erase(), remove zeros. time (35%) space (34%)

Still slow. Since the `erase()` function will reallocate each element after the deleted one. Worst case time complexity should be $O(N^2)$.

```
1   class Solution {
2   public:
3
4     void moveZeroes(vector<int>& nums) {
5       int zero_count = 0;
6       for (auto iter = nums.begin(); iter != nums.end(); ++iter)
7         if (*iter == 0)
8           zero_count++;
9
10      if (zero_count == 0)
11        return;
12
13      auto iter = nums.begin();
14      int zero_deleted = 0;
15
16      while (zero_deleted < zero_count) {
17        if (*iter == 0) {
18          iter = nums.erase(iter);
19          nums.push_back(0);
20          zero_deleted++;
21        }
22
23        else
24          ++iter;
25      }
26    }
27  };
```

4.4 todos [0/2]

- ☐ try to think another Solution

- ☐ read the solution page and study

Chapter 5

442. Find All Duplicates in an Array

5.1 Problem Statement

Link

5.2 Analysis

5.2.1 Label Duplicate Number

Label the appearing frequency of each element, using the fact that `1 <= a[i] <= n`, where n is the size of array. Then count the number that appeared twice.

5.3 Solution

5.3.1 C++

5.3.1.1 Label duplicate number (96%, 16%)

This one use an extra vector to hold the labeling information.

```
1  class Solution {
2  public:
3    vector<int> findDuplicates(vector<int>& nums) {
4      vector<int> duplicate;
5      vector<int> frequency_count(nums.size(), 0);
6
7      for (int i = 0; i < nums.size(); i++) {
8        frequency_count[nums[i] - 1]++;
9      }
10
11     for (int i = 0; i < frequency_count.size(); i++)
```

```
12        if (frequency_count[i] > 1)
13          duplicate.push_back(i + 1);
14
15      return duplicate;
16    }
17  };
```

5.4 todos [/]

- ☐ think about the way to use original vector to hold labeling information
- ☐ read other solutions
- ☐ generalize the problem

Chapter 6

448. Find All Numbers Disappeared in an Array

6.1 Problem Statement

Link

6.2 Analysis

6.2.1 Label Appearance of Numbers

This is similar with Problem 442. Label the appearing frequency of each element, using the fact that `1 <= a[i] <= n`, where n is the size of array. Then count the number that appearing frequency is 0.

You can use either a new vector to hold the labeling information, or the original passed-in vector.

6.2.2 Use Unordered-set

Use an unordered-set to store all appeared number. Then traverse from 1 to N to find out if one number is in the set, if not, it is one disappearing number, push to result. This method's time complexity is $O(N)$ on average, but $O(N^2)$ for worst cases, due to the time complexity of `insert()` and `find()` in unordered-set.

6.3 Solution

6.3.1 C++

6.3.1.1 Label appearance of numbers (97%, 15%)

Space can be optimized by using original passed-in vector.

```cpp
class Solution {
public:
  vector<int> findDisappearedNumbers(vector<int>& nums) {
    vector<int> appear_label(nums.size(), 0);
    vector<int> disappear;

    // label appeared number
    for (int i = 0; i < nums.size(); i++) {
      appear_label[nums[i] - 1] = 1;
    }

    // find out unlabelled number
    for (int i = 0; i < appear_label.size(); ++i)
      if (appear_label[i] == 0)
        disappear.push_back(i + 1);

    return disappear;
  }
};
```

6.3.1.2 Use unordered-set (13%, 7%)

```cpp
class Solution {
public:
  vector<int> findDisappearedNumbers(vector<int>& nums) {
    vector<int> disappear;
    unordered_set<int> appeared; // extra space used

    for (auto num : nums)  // total average O(N), worst: O(N^2)
      appeared.insert(num);  // average: O(1), worst: O(N)

    for (int i = 1; i <= nums.size(); ++i) {
      if (appeared.find(i) == appeared.end())  // find(), average: O(1),
      ↪  worst: O(N)
        disappear.push_back(i);
    }

    return disappear;
```

```
16    }
17  };
18
19  // Total complexity: average: O(N), worst: O(N^2), still bound by O(N^2)
```

6.4 todos [/]

- ☐ think about using the original vector to hold labeling information
- ☐ read other solutions
- ☐ generalize the problem

Chapter 7

461. Hamming Distance

7.1 Problem Statement

Link

7.2 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
```

7.3 Solution

7.3.1 C++

7.3.1.1 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;
  }
};
```

7.3.1.2 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;
  }
};
```

7.3.1.3 Questions

Why the second solution is faster than the previous one?

- Bitwise XOR used.

7.3.2 Python

7.3.2.1 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 8

477. Total Hamming Distance

8.1 Problem Statement

Link

8.2 Analysis

This problem is similar with P461, but you can't 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

8.2.1 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.

8.2.2 Grouping

Reference

The idea of grouping is we count the total hamming distance as a whole. And we only count those valid bits (bits that will contribute to the total Hamming distance). Specifically, at any giving time, we divide 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. Another confusing case is when some numbers are trimmed to zero during the process. We still keep those zeros in array, because they still can be used to count total Hamming distance. For example, number 9 and 13317:

```
---------------------------
bit#:   1234 5678 9abc defg
---------------------------
9:      0000 0000 0000 1001
13317:  0011 0100 0000 0101
---------------------------
```

After four times of trimming:

```
-----------------------
bit#:   1234 5678 9abc
-----------------------
```

```
9:       0000 0000 0000
13317:   0011 0100 0000
----------------------
```

The difference at bit 3, 4, 6 should still be counted toward the total Hamming distance.

At each round, we first go over the list and divide the numbers into two groups. This process is $O(N)$. To calculate the contribution to total Hamming distance at this round is just a matter of multiplication, so the time complexity is $O(1)$. Thus, for one round, time complexity is $O(N)$. There are potentially `8 * sizeof(int)` bits to be trimmed, this is the number of rounds we are going to run, which is a constant not related to N. Thus the total complexity is: $O(N)$.

Additional notes (2019/5/26) It is not a good idea to **TRIM** the numbers, which may add additional complexities. We can just use a for loop to compare all `8 * sizeof(int)` bits on integer. The range of iterating number (i) is from 0 to 31. At each iteration, we compare the value at i-th bit (starting from zero) with 1. To achieve this, we need use two operators (bitwise **AND** and left shift). Notice that the bitwise **AND** is 1 only if both bits are 1.

8.3 Solution

8.3.1 C++

8.3.1.1 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  };
```

8.3.1.2 Grouping. 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  };
```

There are many reasons why this solution is expensive. Some of them are listed below:

- There is no need to actually use two vectors to **STORE** each number in two vectors. You just need to count the number.

8.3.1.3 Grouping_example. time (88.24%, 49.76%)

This is from the discussion (grouping idea).

```
1  class Solution {
2  public:
3    int totalHammingDistance(vector<int>& nums) {
4      if (nums.size() <= 0) return 0;
5
6      int res = 0;
7
8      for(int i=0;i<32;i++) {
9        int setCount = 0;
10       for(int j=0;j<nums.size();j++) {
11           if ( nums[j] & (1 << i) ) setCount++;
12       }
13
14       res += setCount * (nums.size() - setCount);
15     }
16
17     return res;
18   }
19 };
```

This solution is a lot faster than my version, altough we use the same idea. I used a lot more steps to do the book keeping, which the example solution uses spaces and time efficiently. Specifically:

- I have defined two vectors to actually store the **TWO** groups. My thinking is simple: if the idea involves two groups, then I want to actually implement two groups to closely follow the idea. This reflects the lack of ability to generalize a problem and find what matters most to solve the problem. In this specific example, what matters most, is to **KNOW** the number of element in just **ONE** group, there are ways to know this without actually spending time and spaces to keep the whole record of the two groups.
- my end point would be "there is no non-zero number in the array", I have to declare a new integer to keep track of the number of non-zero number, and I have to use an if expression to determine if a number is non-zero after trimming the least significant bit. The example code only traverse all the bits of an integer (i.e. 32 bits in total, or 4 bytes) using a for loop.

In line 11, the code reads: `if ( nums[j] & (1 « i) ) setCount++;`. The operators used are bitwise AND, bitwise left shift. This is to compare the i-th bit of `num[j]` with 1. If it is

1, then at this bit, the number should be counted in group G_1. For example, if `num[j] == 113`, `i == 5`, then we compare:

```
          ↓
113:     0111 0001
1 << i:  0010 0000
```

Also, we don't have to count integer numbers in G_0, since: $N_{G_0} = N - N_{G_1}$, where N is the total number of integers, which is equal to `nums.size()`.

8.4 todos [3/4]

- ☒ 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 9

581. Shortest Unsorted Continuous Subarray

9.1 Problem Statement

Link

9.2 Analysis

9.2.1 Use sorting

Let's compare the original array with its sorted version. For example, we have:

```
original: 2 6 4 8 10  9  15
sorted:   2 4 6 8  9 10  15
            ↑            ↑
            1            6
```

They started to differ at 1, and ended differ at 6. The continuous unsorted subarray is bound to this range, thus we can calculate the length by `end_differ_index - start_differ_index`.

Steps to solve this problem:

1. build a sorted array
2. create two integers: `end_differ_index - start_differ_index`, they represent the position where differ between original array and sorted array starts and ends. The default value would be zero, which means no difference.
3. start from beginning, traverse the array to find out the first position where two arrays differ. Store it in `start_differ_index`.
4. start from ending, traverse the array (to the beginning) to find out the last position where two arrays are the same. Store it in `end_differ_index`.

5. return `end_differ_index - start_differ_index`, which is the length of the shortest unsorted continuous subarray. If the original array is already sorted, this value would be zero.

9.3 Solution

9.3.1 C++

9.3.1.1 use sorting

```
class Solution {
public:
  int findUnsortedSubarray(vector<int>& nums) {
    vector<int> nums_sorted = nums;
    sort(nums_sorted.begin(), nums_sorted.end());

    int start_differ_index = 0;
    int end_differ_index = 0;

    // determine start_differ_index
    for (int i = 0; i < nums.size(); i++) {
      if (nums[i] != nums_sorted[i]) {
        start_differ_index = i;
        break;
      }
    }

    // determine end_differ_index
    for (int i = nums.size() - 1; i >= 0; i--) {
      if (nums[i] != nums_sorted[i]) {
        end_differ_index = i + 1;
        break;
      }
    }

    // return result
    return end_differ_index - start_differ_index;
  }
};
```

9.4 todos [1/3]

- ☒ write down your analysis and solution
- ☐ check solution page, study, understand and implement them

- ☐ study first solution (brutal force)

Chapter 10

771. Jewels and Stones

10.1 Problem Statement

Link

10.2 Analysis

10.3 Solution

10.3.1 C++

10.3.1.1 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;
  }
};
```

Part II

Hash Table

Chapter 11

242. Valid Anagram

11.1 Problem Statement

Link

11.2 Analysis

11.2.1 Sort

11.2.2 Hash Table

11.3 Solution

11.4 todos [/]

- ☐ write down your analysis and solution
- ☐ check solution and discussion section, read and understand other ideas and implement them
- ☐ generalize the problem

Part III

Linked List

Chapter 12

160. Intersection of Two Linked Lists

12.1 Problem Statement

Link

12.2 Analysis

Assume the size of list `A` is (m), and the size of list `B` is (n).

12.2.1 Brutal force

We can traverse list `A`. For each encountered node, we traverse list `B` to find out if there is a same node. The time complexity should be $(O(mn))$. Since we don't use other spaces to store any information, the space complexity is $(O(1))$.

12.2.2 Hash table

First, we traverse list `A` to store all the address information of each node in a hash table (e.g. Unordered-set). Then we traverse list `B` to find out if each node in `B` is also in the hash table. If so, it is an intersection.

12.2.3 Skip longer list

Let's consider a simpler situation: list `A` and list `B` has the same size. We just need to traverse the two lists one node at a time. If there is an intersection, it must be at the same relative position in the list.

In this problem, we may not have lists that with same size. However, if two linked lists intersect at some point, the merged part's length does not exceed the size of the shorter list. This means we can skip some beginning parts of the longer list because intersection could not possibly happen there. For example:

```
List A: 1 5 2 8 6 4 9 7 3
List B:       4 8 1 9 7 3
                    ↑
```

List `A` and `B` intersect at node 9. We can skip the `[1, 5, 2]` part in list `A` and then treat them as list with same size:

```
List A': 8 6 4 9 7 3
List B : 4 8 1 9 7 3
              ↑
```

So the steps to solve this problem are:

1. traverse list `A` and `B` to find out the size of two lists
2. skip beginning portion of longer list so that the remaining part of the longer list has the same size as the shorter list
3. check the two lists and find possible intersection

The time complexity: $O(n)$ or $O(m)$, depends on which is bigger. The space used is not related to the input size, thus space complexity is $O(1)$.

12.2.4 Two pointer

12.3 Solution

12.3.1 C++

12.3.1.1 brutal force

```
/**
 * Definition for singly-linked list.
 * struct ListNode {
 *     int val;
 *     ListNode *next;
 *     ListNode(int x) : val(x), next(NULL) {}
 * };
 */
class Solution {
public:
  ListNode* exist(ListNode* ptr, ListNode* head) {
    while (head != nullptr && ptr != head) {
      head = head->next;
    }

    return head;
  }
```

```
  ListNode* getIntersectionNode(ListNode *headA, ListNode *headB) {
    while (headA != nullptr) {
      ListNode* result = exist(headA, headB);

      if (result != nullptr)
        return result;

      headA = headA->next;
    }

    return headA;
  }
};
```

12.3.1.2 hash table

```
/**
 * Definition for singly-linked list.
 * struct ListNode {
 *     int val;
 *     ListNode *next;
 *     ListNode(int x) : val(x), next(NULL) {}
 * };
 */
class Solution {
public:

  ListNode* getIntersectionNode(ListNode *headA, ListNode *headB) {
    unordered_set<ListNode*> A_record;

    // record A's node
    while (headA != nullptr) {
      A_record.insert(headA);
      headA = headA->next;
    }

    // go over B and find if there is any intersection
    while (headB != nullptr) {
      if (A_record.find(headB) != A_record.end())
        return headB;

      headB = headB->next;
    }
```

```
    return headB;
  }
};
```

12.3.1.3 skip longer lists

```
/**
 * Definition for singly-linked list.
 * struct ListNode {
 *     int val;
 *     ListNode *next;
 *     ListNode(int x) : val(x), next(NULL) {}
 * };
 */
class Solution {
public:

  ListNode* getIntersectionNode(ListNode *headA, ListNode *headB) {
    int size_A = 0;
    int size_B = 0;

    ListNode* start_A = headA;
    ListNode* start_B = headB;

    // count the number of nodes in A and B
    while (start_A != nullptr) {
      start_A = start_A -> next;
      size_A++;
    }

    while (start_B != nullptr) {
      start_B = start_B -> next;
      size_B++;
    }

    // skip the first portion of the list
    if (size_A > size_B) {
      int skip = size_A - size_B;

      for (int i = 1; i <= skip; i++)
        headA = headA -> next;
    }

    else if (size_A < size_B) {
      int skip = size_B - size_A;
```

```
      for (int i = 1; i <= skip; i++)
        headB = headB -> next;
    }

    // now A and B has same relative length, check possible intersection
    while (headA != nullptr && headA != headB) {
      headA = headA -> next;
      headB = headB -> next;
    }

    return headA;

  }
};
```

12.4 todos [1/3]

- ☒ write down your analysis and solution
- ☐ read the two pointer solution, understand, implement, record
- ☐ read discussion page to see if there is any other solution

Chapter 13

206. Reverse Linked List

13.1 Problem Statement

Link

Notice that the `head` in this linked list is actually the first node in the list. Not like what you learned in COP 4530.

13.2 Analysis

This problem should have some simpler solution. My two solutions are just akward.

13.2.1 Using Stack (my)

13.2.2 Recursion (my)

13.3 Solution

13.3.1 C++

13.3.1.1 Using Stack. time (96%), space (5%)

This method uses a stack to keep the reverse order. Additional memory is required.

```
/**
 * Definition for singly-linked list.
 * struct ListNode {
 *     int val;
 *     ListNode *next;
 *     ListNode(int x) : val(x), next(NULL) {}
 * };
 */
```

```
9  class Solution {
10 public:
11   ListNode* reverseList(ListNode* head) {
12     stack<ListNode*> nodes;
13     // check if head is nullptr
14     if (head == nullptr)
15       return head;
16
17     // store the list in stack
18     while (true) {
19       if (head->next != nullptr) {
20         nodes.push(head);
21         head = head->next;
22       }
23
24       else { // head is pointing the last node
25         nodes.push(head);
26         break;
27       }
28     }
29
30     // start re-connect
31     head = nodes.top();
32     nodes.pop();
33     ListNode* last_node = head;
34
35     while (!nodes.empty()) {
36       last_node->next = nodes.top();
37       last_node = last_node->next;
38       nodes.pop();
39     }
40
41     last_node->next = nullptr;
42
43     return head;
44   }
45 };
```

13.3.1.2 Using Recursion. time (18%), space (21%)

This approach is a very "akward" way to use recursion.

13.4 todos [/]

- ☐ try to think another way to work this problem
- ☐ read solution, write down thinking process
- ☐ time complexity analysis of your code and solution code

Part IV

Trees

Chapter 14

101. Symmetric Tree

14.1 Problem Statement

Link

14.2 Analysis

14.2.1 Recursion

We need to define a method to describe how two nodes are equal "symmetrically", i.e. if two subtrees with root node `a` and `b`, we say subtree `a` is "equal" with subtree `b`, if the two subtrees are symmetric.

By this method, we need a helper function that accepts two `Treenode` pointer (`a` and `b`). Its return type is bool. It can tell whether the two subtrees started by the two `Treenode` passed in are symmetrically equal or not. We use this function recursively. Two subtrees are symmetrically equal, if:

1. `a->val == b->val`, the root must have the same value
2. `a->left` is symmetrically equal to `b->right`
3. `a->right` is symmetrically equal to `b->left`

Case 2, 3 can be determined by calling this function recursively. Case 1 can be determined directly. Also, we have to be aware of the base case (when `a == nullptr` or `b == nullptr`.

14.3 Solution

14.3.1 C++

14.3.1.1 recursion (75%, 64%)

```cpp
/**
 * Definition for a binary tree node.
 * struct TreeNode {
 *     int val;
 *     TreeNode *left;
 *     TreeNode *right;
 *     TreeNode(int x) : val(x), left(NULL), right(NULL) {}
 * };
 */
class Solution {
public:
  bool isSymmetric(TreeNode* root) {
    if (root == nullptr)
      return true;

    return isSym(root->left, root->right);
  }

  bool isSym(TreeNode* a, TreeNode* b) {
    if (a == nullptr) {
      if (b == nullptr)
        return true;
      return false;
    }

    if (b == nullptr)
      return false;

    if (a->val != b->val)
      return false;

    if (isSym(a->left, b->right) && isSym(a->right, b->left))
      return true;

    return false;
  }
};
```

14.4 todos [1/5]

- ☒ write down your analysis (recursion)
- ☐ think about iterative solution
- ☐ write down analysis (iterative solution)
- ☐ check solution and discussion to find out any other idea
- ☐ generalize this problem

Chapter 15

104. Maximum Depth of Binary Tree

15.1 Problem Statement

Link

15.2 Analysis

15.2.1 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.

15.3 Solution

15.3.1 C++

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

```
/**
 * Definition for a binary tree node.
 * struct TreeNode {
 *     int val;
 *     TreeNode *left;
 *     TreeNode *right;
 *     TreeNode(int x) : val(x), left(NULL), right(NULL) {}
 * };
 */
class Solution {
public:
  int maxDepth(TreeNode* root) {
    // base case
```

```
    if (root == nullptr)
      return 0;

    int left_depth = maxDepth(root->left);
    int right_depth = maxDepth(root->right);

    return (left_depth >= right_depth ? left_depth + 1 : right_depth + 1);
  }
};
```

15.4 todos [0/1]

- ☐ read about the discussion page for more methods and ideas
- ☐ read about traversal method
- ☐ make notes in your data structure notes about DFS abd BFS

Chapter 16

112. Path Sum

16.1 Problem Statement

Link

16.2 Analysis

16.2.1 Recursion

The goal is to find a root-to-leaf path such that sum of all values stored in node is the given sum: `sum`. We can start from root. Notice that, if `root->left` or `root->right` has a path that can add up to `sum - root->val`, a path is found. This implies that we can recursively call the function itself and find if there is any path that can have `sum - root->val` target.

One thing should be noticed that is, we have to go down all the way to a leaf to find out the final answer that whether the path of this leaf to root satisfies or not. So there is only two base cases:

1. the pointer passed in is `nullptr`: return false
2. the pointer passed in is leaf: check if passed in `sum` is equal to `root->val`, if so, return true. Otherwise, return false.

For other situations, we continue call the function. Do not pass in `nullptr`.

16.3 Solution

16.3.1 C++

16.3.1.1 recursion (88%, 92%)

```
/**
 * Definition for a binary tree node.
```

```
 * struct TreeNode {
 *     int val;
 *     TreeNode *left;
 *     TreeNode *right;
 *     TreeNode(int x) : val(x), left(NULL), right(NULL) {}
 * };
 */
class Solution {
public:
    bool hasPathSum(TreeNode* root, int sum) {
      if (root == nullptr)
        return false;

      if (root->left == nullptr && root->right == nullptr) {
        if (root->val == sum)
            return true;
        else
            return false;
      }

      if (root->left == nullptr)
        return hasPathSum(root->right, sum - root->val);
      else if (root->right == nullptr)
        return hasPathSum(root->left, sum - root->val);
      else
        return hasPathSum(root->left, sum - root->val) ||
         ↪  hasPathSum(root->right, sum - root->val);
    }
};
```

16.4 todos [1/3]

- ☒ write down your recursion solution and analysis
- ☐ work on the DFS approach
- ☐ check discussion, find out other ideas, understand and implement them

Chapter 17

113. Path Sum II

17.1 Problem Statement

Link

17.2 Analysis

17.2.1 DFS

In DFS, you use a stack to keep track of your path. This problem requires you to find out all the path that satisfies the requirement. So you have to do book-keeping. The basic DFS idea is as follows.

1. we push the root into a stack: `v`.
2. use a while loop to find all combinations of root-to-leaf path: `while (!v.empty())`
3. if the top node in the stack is a leaf, then it suggests the current stack is holding a complete root-to-leaf path. We should check if this path adds to the target sum. If so we have to push this path into the result. Then, we have to trace backward, until we found a previous node that has unvisited child **OR** the stack is empty. Each time we push a node into the stack, we have to mark it as visited. We achieve this by using an unordered_set to record these nodes being pushed into the stack. Unordered_set has fast retrival rate using a key.
4. if the top node in the stack is not a leaf, then we have to continue to push its children into the stack. We first try inserting left child, and then right child. This depends on the visit history of the children. Only one child per loop. After inserting one child, we `continue`, beginning the next loop.
5. from the above analysis, we can see that we trace back, only when we meet a leaf node. This guarantees that the found path is root-to-leaf path.

6. after the while loop, the stack becomes empty, which means all nodes are visited. Then we return the result.

17.3 Solution

17.3.1 C++

17.3.1.1 DFS (80%, 40%)

```
/**
 * Definition for a binary tree node.
 * struct TreeNode {
 *     int val;
 *     TreeNode *left;
 *     TreeNode *right;
 *     TreeNode(int x) : val(x), left(NULL), right(NULL) {}
 * };
 */
class Solution {
public:
  // member variables
  vector<vector<int>> results;
  unordered_set<TreeNode*> visited_nodes;

  // helper functions
  bool isLeaf(TreeNode* t) {
    return (t->left == nullptr) && (t->right == nullptr);
  }

  void traceBack(vector<TreeNode*>& v) {
    while (!v.empty() && !hasUnvisitedChild(v.back()))
      v.pop_back();
  }

  bool hasUnvisitedChild(TreeNode* t) {
    return !(isVisited(t->left) && isVisited(t->right));
  }

  bool isVisited(TreeNode* t) {
    if (t == nullptr || visited_nodes.find(t) != visited_nodes.end())
      return true;

    return false;
  }
```

```
  void checkVal(const vector<TreeNode*>& v, int target) {
    int sum = 0;
    vector<int> result;
    for (auto node : v) {
      result.push_back(node->val);
      sum += node->val;
    }

    if (sum == target)
      results.push_back(result);
  }

  // solution function
  vector<vector<int>> pathSum(TreeNode* root, int sum) {
    if (root == nullptr)
      return results;

    vector<TreeNode*> v{root};
    visited_nodes.insert(root);

    while (!v.empty()) {
      // check the last node: to see if it is leaf
      if (isLeaf(v.back())) {
        checkVal(v, sum);
        traceBack(v);
        continue;
      }

      if (!isVisited(v.back()->left)) {
        visited_nodes.insert(v.back()->left); // mark as visited
        v.push_back(v.back()->left);
        continue;
      }

      if (!isVisited(v.back()->right)) {
        visited_nodes.insert(v.back()->right); // mark as visited
        v.push_back(v.back()->right);
        continue;
      }
    }

    return results;
  }
};
```

17.4 todos [1/4]

- ☒ write down your DFS solution and analysis
- ☐ work on the recursion approach
- ☐ check discussion, find out other ideas, understand and implement them
- ☐ generalize the problem

Chapter 18

226. Invert Binary Tree

18.1 Problem Statement

Link

18.2 Analysis

18.2.1 Recursion

To solve this problem recursively, we first invert the left subtree of a node by calling this function, then we invert the right subtree of this node by calling this function. Then we return a pointer to this node. Base case: `node == nullptr`, in this case we return the node directly, since the invert of a `nullptr` tree is itself.

18.3 Solution

18.3.1 C++

18.3.1.1 Recursion. time (91.95%) space (5.15%)

I don't understand why my code require this amount of space. Needs to be analyzed.

```
/**
 * Definition for a binary tree node.
 * struct TreeNode {
 *     int val;
 *     TreeNode *left;
 *     TreeNode *right;
 *     TreeNode(int x) : val(x), left(NULL), right(NULL) {}
 * };
 */
```

```
10 class Solution {
11 public:
12   TreeNode* invertTree(TreeNode* root) {
13     if (root == nullptr)
14       return root;
15
16     TreeNode* temp = root->left;
17     root->left = invertTree(root->right);
18     root->right = invertTree(temp);
19
20     return root;
21   }
22 };
```

18.4 todos [0/2]

- ☐ analyze why my code requires a lot more space than the divide and conquer method
- ☐ read the discussion page for more solution

Chapter 19

437. Path Sum III

19.1 Problem Statement

Link

19.2 Analysis

19.2.1 Double recursion (~50%, 50%)

The tricky part is that the path does not need to start or end at the root or a leaf. However, it must go downwards (traveling only from parent nodes to child nodes), this is to say that we don't consider the situation that the path is like: `left_child -> node -> right_child`, which makes things easier.

The tricky part means we may have some paths deep below that sum to the target value, these paths are not connected to the root. In fact, we can conclude that, given a tree (or subtree) starting at `node`, the paths that sum to the target value are composed of following cases:

1. paths from **left** subtree of node that sum to the target, they are not connected to `node` though
2. paths from **right** subtree of node that sum to the target, they are also not connected to `node`.
3. any paths that containing `node` as their starting node. This includes path connecting `node` and `node->left`, paths connecting `node` and `node->right`, and also `node` alone if `node->val == target`.

Pay attention that we don't have to consider paths like `left->node->right`, as mentioned earlier. The function header of the solution function is:

```
1 int pathSum(TreeNode* root, int sum)
```

We can use this function to get the result of case 1 and case 2. Since these results are **NOT** containing the root. As for case 3, we can build a helper function `continuousSum()` to calculate. This function will also use recursive algorithm. The function header is:

```
int continuousSum(TreeNode* root, int sum)
```

It will return the total number of path that containing `root` and sum to the target `sum`. Pay attention that, these paths do not need to go from `root` to leaf. The base case is when `root == nullptr`, in this case, return zero. The total number can be calculated by calling itself, which is composed of following:

1. `continuousSum(root->left, sum - root->val)`
2. `continuousSum(root->right, sum - root->val)`
3. `+1 if root->val == sum`

Case 3 is when a path only contains the `root`. Pay attention that if `root->left` has a path sum to zero, and `root->val == sum`, then `left->root` and `root` are considered two different pathes. If we think in a recursive way, this will account for those paths that from a node but not reaching leaf. The last node in the path is the node that `node->val is equal to passed in sum`.

19.3 Solution

19.3.1 C++

19.3.1.1 double recursion

```
/**
 * Definition for a binary tree node.
 * struct TreeNode {
 *     int val;
 *     TreeNode *left;
 *     TreeNode *right;
 *     TreeNode(int x) : val(x), left(NULL), right(NULL) {}
 * };
 */
 /*Notes:
 calculate continuous sum and un-continuous sum
 */
class Solution {
public:
  int continuousSum(TreeNode* root, int sum) {
    if (root == nullptr)
      return 0;
```

```
    int count = continuousSum(root->left, sum - root->val) +
    ↪  continuousSum(root->right, sum - root->val);

    if (sum == root->val)
      count += 1;

    return count;
  }

  int pathSum(TreeNode* root, int sum) {
    if (root == nullptr)
      return 0;

    return continuousSum(root, sum) + pathSum(root->left, sum) +
    ↪  pathSum(root->right, sum);

  }
};
```

19.4 todos [1/3]

- ☒ write down your own solution (including analysis).
- ☐ check discussion panel, find out other solutions. Understand and write analysis, implement the solution
- ☐ write down these analysis

Chapter 20

543. Diameter of Binary Tree

20.1 Problem Statement

Link

20.2 Analysis

20.2.1 Direct recursion

Just as stated in the problem statement, the longest path between any two nodes may not pass through the root. So, for a given node, the longest path of this node may have three cases:

1. longest path is in its left subtree, and does not pass this node;
2. longest path is in its right subtree, and does not pass this node;
3. longest path passes through this node;

We can calculate the path of the above three cases, and find out which one is the longest. Calculate case 1 and 2 is easy, we can call the function recursively to find out the longest path of the left and right subtree. To calculate case 3, we use the fact that: longest path passing this node = height of left subtree + height of right subtree + 2, where "2" correspondes to the two edges connecting left and right subtree to the root node. The we compare these three values and return the largest one.

Also, we have to consider the base case:

1. this node is `nullptr`
2. its left subtree is `nullptr`
3. its right subtree is `nullptr`

20.3 Solution

20.3.1 C++

20.3.1.1 direct recursion (5%, 5%)

```
/**
 * Definition for a binary tree node.
 * struct TreeNode {
 *     int val;
 *     TreeNode *left;
 *     TreeNode *right;
 *     TreeNode(int x) : val(x), left(NULL), right(NULL) {}
 * };
 */
class Solution {
public:
  int height(TreeNode* t) {
    if (t == nullptr || (t->left == nullptr && t->right == nullptr))
      return 0;
    return max(height(t->left), height(t->right)) + 1;
  }

  int diameterOfBinaryTree(TreeNode* root) {
    if (root == nullptr || (root->left == nullptr && root->right ==
    ↪  nullptr))
      return 0;

    if (root->left == nullptr)
      return max(height(root), diameterOfBinaryTree(root->right));
    else if (root->right == nullptr)
      return max(height(root), diameterOfBinaryTree(root->left));
    else
      return max(max(height(root->left) + height(root->right) + 2,
      ↪  diameterOfBinaryTree(root->left)),
      ↪  diameterOfBinaryTree(root->right));
  }
};
```

20.4 todos [/]

- ☐ check solution and study
- ☐ implement solution by yourself
- ☐ write down different ways of thinking about this problem

Chapter 21

617. Merge Two Binary Trees

21.1 Problem Statement

Link

21.2 Analysis

21.2.1 Recursive Method

Use recursiion to solve this problem.

21.2.2 Iterative Method (using stack)

21.3 Solution

21.3.1 C++

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

Recursion.

```
/**
 * Definition for a binary tree node.
 * struct TreeNode {
 *     int val;
 *     TreeNode *left;
 *     TreeNode *right;
 *     TreeNode(int x) : val(x), left(NULL), right(NULL) {}
 * };
 */
class Solution {
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  };
```

21.3.1.2 Iterative

21.4 todos [0/2]

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

Chapter 22

938. Range Sum of BST

22.1 Problem Statement

Link

22.2 Analysis

22.2.1 Recursion (brutal and stupid)

The tree is composed of left subtree, the node, and the right subtree. The base case is when the root is pointing to `nullptr`, in this case, we should return 0.

Thus, we call the function itself to find out the range sum of left subtree and right subtree first, then we check the `node->val`. If it is within the range, we add it to the whole sum, otherwise, we ignore it.

This algorithmm is easy to follow, but it does a lot of unnecessary work (didn't use the fact that this is a binary search tree, which satisfies: `node->left->val < node->val < node->right->val`, given "the binary search tree is guaranteed to have unique values"). If `node->val` is smaller than `L`, then we have no reason to check `rangeSumBST(node->left, L, R)`, since any value contained in this branch of subtree is bound to be smaller than `node->val`, thus not within the range `[L, R]`. Similarliy if `node->val` is greater than `R`, we don't have to check `rangeSumBST(node->right, L, R)`. This thought gives a better recursion algorithm.

22.2.2 DFS

DFS allows us to traverse the tree in a deapth first manner (go deep first). It will eventually go over all the nodes one by one. We use a stack to perform the DFS, we also need an associative container to hold record of visited nodes. The basic steps is this:

1. create a stack and an unordered_set

2. push the root (if it is not nullptr) into the stack
3. while the stack is not empty, we check the top node in the stack
    - if the top node is a leaf, then we check its value (to see if it is within the range, so we can add it to the total sum); Then we pop it ()
    - if the top node is not a leaf, we check if its left node is visited, if not, we visit it by pushing its left child into the stack, and record this in the Unordered_set, then we go to the next loop. If its left node was already visited, we check right child and do the same thing
4. if the top node has no unvisited child, we check its value to see if it satisfies the range, if so, we add it to the sum. Then, we pop it out of the stack, start next loop.

By DFS, we can traverse the whole tree's each node in a depth-first manner. We can get the range sum along the way.

22.3 Solution

22.3.1 C++

22.3.1.1 recursion (stupid)

```
class Solution {
public:
  int rangeSumBST(TreeNode* root, int L, int R) {
    // base case
    if (root == nullptr)
      return 0;

    return (root -> val <= R && root -> val >= L ? root -> val +
    ↪  rangeSumBST(root -> left, L, R) + rangeSumBST(root -> right, L, R)
    ↪  : rangeSumBST(root -> left, L, R) + rangeSumBST(root -> right, L,
    ↪  R));
  }
};
```

22.3.1.2 recursion (better)

```
class Solution {
public:
  int rangeSumBST(TreeNode* root, int L, int R) {
    // base case
    if (root == nullptr)
      return 0;

    if (root->val < L)
```

```
      return rangeSumBST(root -> right, L, R);

    if (root->val > R)
      return rangeSumBST(root->left, L, R);

    return root->val + rangeSumBST(root->left, L, R) +
     ↪  rangeSumBST(root->right, L, R);
  }
};
```

22.3.1.3 DFS (slow, 5% and 6%)

```
class Solution {
public:

  int rangeSumBST(TreeNode* root, int L, int R) {
    if (root == nullptr)
      return 0;

    int sum = 0;

    // use a set to keep track of visited nodes
    unordered_set<TreeNode*> visited_nodes;
    // use a stack to do DFS
    stack<TreeNode*> nodes;
    nodes.push(root);

    while (!nodes.empty()) {
      // check if top node is leaf or not
      if (nodes.top()->left == nodes.top()->right) {
        if (nodes.top()->val >= L && nodes.top()->val <= R) {
          sum += nodes.top()->val;
          nodes.pop();
          continue;
        }
      }

      // check if nodes.top() has unvisited child (first check left, then
       ↪  right)
      // if so, push it into the stack
      // otherwise, calculate sum
      if (nodes.top()->left != nullptr &&
       ↪  visited_nodes.find(nodes.top()->left) == visited_nodes.end()) {
        visited_nodes.insert(nodes.top()->left);  // mark as visited
        nodes.push(nodes.top()->left);
```

```
        continue;
      }

      if (nodes.top()->right != nullptr &&
       ↪  visited_nodes.find(nodes.top()->right) == visited_nodes.end()) {
        visited_nodes.insert(nodes.top()->right);
        nodes.push(nodes.top()->right);
        continue;
      }

      // up to here, both child of the nodes.top() node has been visited
      // add to sum if nodes.top()->val satisfies the condition
      if (nodes.top()->val >= L && nodes.top()->val <= R)
        sum += nodes.top()->val;

      nodes.pop();
    }

    return sum;

  }
};
```

22.4 todos [1/3]

- ☒ write down your analysis and solution (recursion and DFS)
- ☐ check solution's DFS, study and re-implement
- ☐ read discussion page, to gain more understanding of possible solution
- ☐ re-implement and write down analysis

Part V

Other

Chapter 23

70. Climbing Stairs

23.1 Problem Statement

Link

23.2 Analysis

This problem can be analyzed backward. Assume we have two steps left, we can use two 1 step to finish, or one 2 steps to finish. So the total number of ways to finish is: [number of ways to finish n - 1 stairs] + [number of ways to finish n - 2 stairs].

It is easy to think using recursion to do this, but it will cause a lot of unnecessary calculation (redundant calculation). This problem is identical to calculate Fibonacci number. Recursion is a bad implementation. The good way is to **Store** the intermediate results, so we can calculate next term easily.

23.3 Solution

23.3.1 C++

23.3.1.1 use a vector to hold intermediate result (77%, 64%)

```
class Solution {
public:
  int climbStairs(int n) {
    if (n == 1)
      return 1;
    else if (n == 2)
      return 2;

    vector<int> steps;
```

```
    steps.push_back(1);
    steps.push_back(2);  // steps required when n = 1 & 2

    int step;
    for (int i = 2; i < n; i++) {
      steps.push_back(steps[i - 1] + steps[i - 2]);
    }

    return steps[n - 1];
  }
};
```

23.4 todos [/]

- ☐ read each solution carefully, try to understand the idea and implement by yourself.
- ☐ generalize the problem