Contents

1 0. Template

1.1 Problem Statement

[[][]]

1.2 Analysis

1.3 Solution

1.4 todos [0/4]

- ☐ write down your own solution and analysis
- ☐ time complexity analysis of your own solution
- ☐ check solution/discussion page for more ideas, implement them, and write down corresponding analysis (including time and space complexity analysis
- ☐ generalize this problem

2 1. Two Sum

2.1 Problem Statement

Link

2.2 Analysis

2.2.1 $O(N^2)$ method (brutal force)

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.

The time complexity is: $N + (N-1) + (N-2) + \cdots + 2 + 1$. Which is $\frac{N(N-1)}{2}$, so the time complexity is $O(N^2)$.

2.2.2 Sort a copy of the array

In the above solution, we use linear search to find out if the counter_part is in the array or not. If we have an ordered version of the array, we can use binary search to finish this task. It's time complexity is $O(\log N)$. This method requires extra space to hold the sorted version of the array. After we get the sorted version (you can use `std::sort()` to finish this job), we have two ways to get the index:

1. we start from the beginning of the original array, for each encountered element, we calculate the counterpart of it. Then we search this counterpart in sorted array (using binary search). If we found an counterpart, we traverse the original array to find out the index of this counterpart. If the index is the same as the index of the current element, it means we used the same element, which can not be considered as a solution. Otherwise, we have found the indexes.
2. we start from the beginning of the sorted array. For each encountered element, we calculate the corresponding counterpart. Then we search the sorted array, from the next element to the end. This is because the counterpart can not appear before the current element, otherwise, the previous element will search to the current element. If we found a counterpart exists, we will traverse the original array to find out the index of the two elements.

2.3 Solution

2.3.1 C++

$O(N^2)$ **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};
  }
};
```

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

```
/*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];

      for (int j = i + 1; j < nums.size(); j++) {
        if (nums[j] == other_part) {
          index_1 = i;
          index_2 = j;
          break;
        }
      }

      if (index_1 != -1)
        break;
    }

    return {index_1, index_2};
  }
};
```

$O(N \log N)$ **Time (8 ms)** 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
      if (*itr_vector[middle] == target) {
        if (itr_vector[middle] == current_itr) {
          start_index += 1;
          end_index += 1;
          continue;
        }

        result = middle;
        break;
      }

      else if (*itr_vector[middle] > target) {
        end_index = middle - 1;
        continue;
      }

      else if (*itr_vector[middle] < target) {
        start_index = middle + 1;
        continue;
      }

    }

    return result;
  }


  vector<int> twoSum(vector<int>& nums, int target) {
    // create a vector of iterators
    vector<vector<int>::iterator> itr_vector;
    for (auto i = nums.begin(); i != nums.end(); ++i)
      itr_vector.push_back(i);

    // sort the vector of iterators, so the values these iterators referred
     ↪  to
    // are in ascending order
    sort(itr_vector.begin(), itr_vector.end(), Compare());

    // go over nums, and find the pair
    for (auto i = nums.begin(); i != nums.end() - 1; ++i) {
      int other_part = target - (*i);
      int other_part_index = findTarget(other_part, itr_vector, i);

      if (other_part_index != -1) // found
        return {static_cast<int>(i - nums.begin()),
         ↪  static_cast<int>(itr_vector[other_part_index] - nums.begin())};
    }

    // for syntax
    return {0, 1};

  }
};
```

sort a copy of the array (way 1, 8 ms)

```
class Solution {
public:
  int binarySearch(const vector<int>& copy, int num) {
    int middle;
    int begin = 0;
    int end = copy.size() - 1;

    while (begin <= end) {
      middle = (begin + end) / 2;

      if (copy[middle] == num)
        return middle;

      if (copy[middle] > num) {
        end = middle - 1;
        continue;
      }

      if (copy[middle] < num) {
        begin = middle + 1;
        continue;
      }
    }

    return -1;
  }

  vector<int> twoSum(vector<int>& nums, int target) {
    vector<int> copy = nums;
    sort(copy.begin(), copy.end());
    int counter_part;
    int first_index;
    int second_index;
    int index;

    for (int i = 0; i < nums.size(); i++) {
      counter_part = target - nums[i];
      index = binarySearch(copy, counter_part);
      if (index != -1) {

        for (int j = 0; j < nums.size(); j++)
          if (nums[j] == copy[index]) {
            second_index = j;
            break;
          }

        if (i != second_index) {
          first_index = i;
          break;
        }

      }
    }

    return {first_index, second_index};
  }
};
```

sort a copy of the array (way 2, 4 ms)

```
class Solution {
public:
  int binarySearch(int target, int index, vector<int> copy) {
    int start_index = index;
    int end_index = copy.size() - 1;
    int middle;

    while (start_index <= end_index) {
      middle = (start_index + end_index) / 2;

      if (copy[middle] == target)
        return middle;

      else if (copy[middle] < target)
        start_index = middle + 1;

      else
        end_index = middle - 1;
    }

    return -1;
  }


  vector<int> twoSum(vector<int>& nums, int target) {
    vector<int> copy = nums;
    sort(copy.begin(), copy.end());
    int index_1 = -1;
    int index_2 = -1;

    for (int i = 0; i < copy.size(); i++) {
      int counter_part = target - copy[i];
      int counter_part_index = binarySearch(counter_part, i + 1, copy);

      if (counter_part_index != -1) { // match found, now try to find the
       ↪  actual index of the two values
        int index = 0;

        while (index_1 == -1 || index_2 == -1) {
          if (index_1 == -1 && nums[index] == copy[i])
            index_1 = index;

          else if (index_2 == -1 && nums[index] == copy[counter_part_index])
            index_2 = index;

          index++;
        }

        break;
      }
    }

    if (index_1 > index_2)
      return {index_2, index_1};
    else
      return {index_1, index_2};
  }
};
```

2.4 todos [2/4]

- ☒ 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
  - ☐ two pass hash table
  - ☐ one pass hash table
  - ☐ write the analysis of each idea
- ☐ generalize this problem

3 27. Remove Element

3.1 Problem Statement

Link

3.2 Analysis

3.2.1 Two pointers

Idea is similar with 283. Move Zeros. Using two pointers (iterators): `a` and `b`. Use iterator `a` to scan through the array. If target encountered, use the iterator `b` to scan element starting from next element. If the iterator `b` find non-target element, swap elements pointed by iterator `a` and `b`. If `b` didn't find any non-target element, it means there is none in the remaining part of the array. We can return.

Another thing should be kept is the number of non-target element we encountered during the iteration. This is the length of the sub-array that we should return. We update this number each time we encounter a non-target element or we swap a target and a non-target element.

3.3 Solution

3.3.1 C++

two pointers

```
class Solution {
public:
  int removeElement(vector<int>& nums, int val) {
    int length = 0;
    auto itr_b = nums.begin();

    for (auto itr_a = nums.begin(); itr_a != nums.end(); ++itr_a) {
      if (*itr_a != val) {
        length++;
        continue;
      }

      // target encountered
      auto itr_b = itr_a + 1;
      while (itr_b != nums.end() && *itr_b == val)
        ++itr_b;

      if (itr_b == nums.end())
        return length;
      swap(*itr_a, *itr_b);
      length++;
    }

    return length;
  }
};
```

3.4 todos [1/2]

- ☒ write down your solution and analysis
- ☐ check solution

4 70. Climbing Stairs

4.1 Problem Statement

Link

4.2 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.

4.3 Solution

4.3.1 C++

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];
  }
};
```

4.4 todos [/]

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

5 79. Word Search

5.1 Problem Statement

Link

5.2 Analysis

5.2.1 Recursion and record visited slot (my first approach)

Pay attention to "the same letter cell may not be used more than once". Not only the immediate previous letter can't be used, all the used letters can't be used. So, a set is used to record all visited slot.

Recursive way to solve this problem would be using a recursive function to search the adjacent cells. We need to build a helper with following abilities: we pass in a coordinate (row and column) and a word, this function will return a boolean value representing if the word can be found in path starting at the passed-in coordinate.

Specifically, the function accepts following parameters:

- `board`: we need to access the original letter board
- `visited`: this should be a hash table containing visited cells (represented by row and column number in the board) during the current search. Pay attention that, if a search doesn't get the target word (search failed), you have to remove the corresponding record of the caller-cell. This is because other paths may still need this cell in their search. This hash table should be checked to make sure no cells be used more than once.
- `row`, col: the coordinate of the current searching cell.

- `word`: the target word to search starting from cell at `(row, col)`

Imagin the actual search process. You are at a certain cell, and you have a word to search. For example, you are going to search "APPLE" start from a cell containing some characters. If the character is not the first letter of the word, you should return false, like the following example:

```
X A X
C B D
X F X
```

You are at cell-B. The four cell-Xs are irrelevant because you can't access them at cell-B. You check if cell-B containing the first letter in "APPLE", in this case, not. So you return false. If you encountered a cell that containing A, you can proceed, like following example:

```
X 1 X
4 A 2
X 3 X
```

If the current cell contains the right letter, we count it as one cell in the total path of the word search. We add the coordinate of this cell to the `visited` hash table.

we have to choose an adjacent cell to search. In fact, you have to search each adjacent cells (1, 2, 3, 4) by calling the `search()` function. The parameter `word` will be changed, so that the first letter is cut off, and pass the remaining part to the recursive function. Then, if `search(cell_1) or search(cell_2) or search(cell_3) or search(cell_4)` is true, we have found the word in some paths from cell_1 or 2 or 3 or 4. If not, no word could be found in paths starting from this cell. So, we need to remove the visited record of this cell from the hash table `visited`.

The base case of this recursive function is as follows:

- `word` is empty: in this case, the word was found in the last path (because the last word has been cut off). So we can return true. This should always be checked first in all base cases.
- `(row, col)` is visited (can be found in `visited`): return false
- `row` or `col` is out of board boundary: return false
- the first letter in `word` can't match `board[row][col]`: return false

The representation of adjacent cell is simple, for example:

```
X 1 X
4 A 2
X 3 X
if A: (R, C)
then:
   1 (R - 1, C)
   2 (R, C + 1)
```

```
3 (R + 1, C)
4 (R, C - 1)
```

5.3 Solution

5.3.1 Recursion and record visited slot (my first approach)

Python

```python
class Solution:
    def search(self, board, visited, row, col, words):
        # check words length, if it is empty, it means already been found
        if not words:
            return True

        # check if row and col has been visited or not
        if (row, col) in visited:
            return False

        # check row and col, to see if it is valid
        if row < 0 or row >= len(board) or col < 0 or col >= len(board[0]):
            print(row, col, 'out bound')
            return False

        # check if the current char in board[row][col] matches the first
        ↪ char in words
        if words[0] != board[row][col]:
            # visited.remove((row, col))  # remove record of failing search
            return False
        visited.add((row, col))

        # perform next search, according to where the current function call
        ↪ coming from
        if self.search(board, visited, row - 1, col, words[1:]) or
        ↪ self.search(board, visited, row, col + 1, words[1:]) or
        ↪ self.search(board, visited, row + 1, col, words[1:]) or
        ↪ self.search(board, visited, row, col - 1, words[1:]):
            return True
        else:
            visited.remove((row, col))  # remove record of failing search
            return False


    def exist(self, board, word: str) -> bool:
        visited = set()
        for row in range(len(board)):
            for col in range(len(board[row])):
                if self.search(board, visited, row, col, word):
                    return True

        print('no match')
        return False
```

5.4 todos [1/2]

- ☒ write down your own solution and analysis
- ☐ time complexity analysis of your solution
- ☐ check discussion page for more solutions

6 101. Symmetric Tree

6.1 Problem Statement

Link

6.2 Analysis

6.2.1 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`.

6.3 Solution

6.3.1 C++

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;
  }
};
```

6.4 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

7 104. Maximum Depth of Binary Tree

7.1 Problem Statement

Link

7.2 Analysis

7.2.1 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.

7.3 Solution

7.3.1 C++

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);
  }
};
```

7.4 todos [0/3]

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

8 108. Convert Sorted Array to Binary Search Tree

8.1 Problem Statement

Link

8.2 Analysis

8.2.1 Recursion

This is kind of related to binary search. In order to build a balanced binary search tree, the root node should be the middle num in the array of numbers. Then the left child should be the middle number of all numbers smaller than root, while the right child should be the middle number of all numbers larger than root, and so on.

To solve this problem recursively, first, build the root, then call the function again to build up the left subtree (passing only the first half of the array). Then, call the function again to build up the right subtree (passing only the second half of the array). In order to pass portion of the original array, we can use iterators to regulate the range the function is working on. Thus the function is like:

```
TreeNode* helper(vector<int>::iterator itr_1, vector<int>::iterator itr_2);
```

where `itr_1` is the beginning of the array, `itr_2` is the end of the array.

After analyzing, you'll find two base cases:

1. `itr_1 > itr_2`, in this case, no node can be built, return a `nullptr`
2. `itr_1 == itr_2`: this is the edge case, only one node can be built. Build a leaf node and return it

Remeber to update the length of the sub-array to work on.

Time complexity: a total of N nodes will be created, each creation time is constant, thus the time complexity is $O(N)$.

8.3 Solution

8.3.1 C++

recursion

```cpp
class Solution {
public:
  TreeNode* helper(vector<int>::iterator itr_1, vector<int>::iterator
   ↪  itr_2) {
    if (itr_1 > itr_2)
      return nullptr;

    if (itr_1 == itr_2) {
      TreeNode* new_node = new TreeNode(*itr_1);
      return new_node;
    }

    auto mid_itr = itr_1 + (itr_2 - itr_1) / 2;
    TreeNode* new_node = new TreeNode(*mid_itr);

    new_node->left = helper(itr_1, mid_itr - 1);
    new_node->right = helper(mid_itr + 1, itr_2);

    return new_node;
  }

  TreeNode* sortedArrayToBST(vector<int>& nums) {
    if (nums.empty())
      return nullptr;

    return helper(nums.begin(), nums.end() - 1);
  }
};
```

8.4 todos [2/4]

- ☒ write down your own solution and analysis
- ☒ time complexity analysis of your own solution
- ☐ check solution/discussion page for more ideas, implement them, and write down corresponding analysis (including time and space complexity analysis
- ☐ generalize this problem

9 110. Balanced Binary Tree

9.1 Problem Statement

Link

9.2 Analysis

9.2.1 Recursion

The balanced tree should satisfy the following conditions:

1. its left subtree is balanced
2. its right subtree is balanced
3. the height difference of its left subtree and right subtree is within the allowed maximum difference.

So, we can use two recursive function to finish these works. One will give the height of a tree (used in 3). Another will determine if a subtree is balanced or not (used in 1 and 2), we use the function we are trying to develop itself.

9.3 Solution

9.3.1 C++

recursion

```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:
  int depth(TreeNode* t) {
    if (t == nullptr)
      return 0;

    return max(depth(t->left), depth(t->right)) + 1;
  }

  bool isBalanced(TreeNode* root) {
    if (root == nullptr)
      return true;

    if (depth(root->left) - depth(root->right) > 1 || depth(root->left) -
     ↪  depth(root->right) < -1)
      return false;
    return isBalanced(root->left) && isBalanced(root->right);
  }
};
```

9.4 todos [1/4]

- ☒ write down your own solution and analysis
- ☐ read discussion page to get more ideas, try to implement them
- ☐ write down analysis of those other solutions
- ☐ generalize the problem

10 111. Minimum Depth of Binary Tree

10.1 Problem Statement

Link

10.2 Analysis

10.2.1 Recursion

The idea is similar with Maximum Depth of Binary Tree. We may use:

```
return min(minDepth(root->left), minDepth(root->right)) + 1;
```

However, there is one situation needs further consiferation:

```
  1
 /
2
```

The above tree's node 1 has only one child. The other child is `nullptr`. In this case the above code will choose the right child rather than the left. To deal with this problem, we can use following strategy:

- if one child is `nullptr`, then return the `minDept()` of the other child.
- otherwise, return the minimum of `minDept(left)` and `minDept(right)`.

10.3 Solution

10.3.1 C++

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:
/*   bool isLeaf(TreeNode* t) {
    if (t == nullptr)
      return false;

    return (t->left == nullptr && t->right == nullptr);
  } */

  int minDepth(TreeNode* root) {
    if (root == nullptr)
      return 0;

    if (root->left == nullptr)
      return minDepth(root->right) + 1;

    if (root->right == nullptr)
      return minDepth(root->left) + 1;

    return min(minDepth(root->left), minDepth(root->right)) + 1;
  }
};
```

10.4 todos [1/4]

- ☒ write down your own solution and analysis
- ☐ try DFS
- ☐ read discussion and explore more ideas
- ☐ try to implement other ideas

11 112. Path Sum

11.1 Problem Statement

Link

11.2 Analysis

11.2.1 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`.

11.3 Solution

11.3.1 C++

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);
    }
};
```

11.4 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

12 113. Path Sum II

12.1 Problem Statement

Link

12.2 Analysis

12.2.1 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.

12.3 Solution

12.3.1 C++

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;
}
```

```
};
```

12.4 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

13 121. Best Time to Buy and Sell Stock

13.1 Problem Statement

Link

13.2 Analysis

13.2.1 Brutal force (my initial solution)

This is my initial solution. For each stock price, traverse through the rest of the array and calculate each profits. If a profit is found to be larger than the current max profit, assign this value to the max profit. Repeat this to all of the rest points.

The time complexity is:$(N-1)+(N-2)+\ldots+1 = \frac{N(N-1)}{2} = O(N^2)$. The space complexity is $O(1)$, for memories used is not related to input size.

13.2.2 One pass

In the brutal force method, we are doing many unnecessary calculations. For example, the input `prices` is:

```
[7,1,5,3,6,4]
```

The second element is the lowest price. In brutal force method, when we deal with this element, we calculated:

```
5 - 1 = 4
3 - 1 = 2
6 - 1 = 5
4 - 1 = 3
```

5 is obtained as the maxprofit. Then, we move on to the next element, which is `5`. In brutal force, we still needs to calculate the following:

```
3 - 5 = -2
6 - 5 = 1
4 - 5 = -1
```

If we keep track of the `minprice`, we will know that these calculations are totally unnecessary. To obtain the maxprofit, we use two variables to hold the min price upto a point (`minprice`), and the current maximum profit calculated (`maxprofit`). Initially, we set the `minprice` as the first price in `prices`, and `maxprofit = 0`. For each price we encountered (`prices[i]`), we compare it with the `minprice`. If it is lower than the `minprice`, update the `minprice`. Then, we calculate `prices[i] - minprice`, if this is larger than the `maxprofit`, we update the `maxprofit`. In this approach, the `minprice` will always before the sell price.

Time complexity of this algorithm is $O(N)$, since only one pass is performed. Space complexity is $O(1)$, since the memory used is not related to input size.

13.3 Solution

13.3.1 C++

brutal force

```
class Solution {
public:
  int maxProfit(vector<int>& prices) {
    if (prices.size() < 2)
      return 0;

    int max_profit = 0;
    int profit;

    for (int i = 0; i < prices.size() - 1; ++i) {
      for (int j = i + 1; j < prices.size(); ++j) {
        profit = prices[j] - prices[i];
        if (profit > max_profit)
          max_profit = profit;
      }
    }

    return max_profit;
  }
};
```

one pass

```
class Solution {
public:
  int maxProfit(vector<int>& prices) {
    if (prices.size() < 2)
      return 0;

    int minprice = prices[0];
    int maxprofit = 0;

    for (int i = 0; i < prices.size(); i++) {
      if (minprice > prices[i])
        minprice = prices[i];

      if (prices[i] - minprice > maxprofit)
        maxprofit = prices[i] - minprice;
    }

    return maxprofit;
  }
};
```

13.4 todos [3/4]

- ☒ write down your own solution and analysis
- ☒ time complexity analysis of your own solution
- ☒ check solution/discussion page for more ideas, implement them, and write down corresponding analysis (including time and space complexity)
  - ☒ one pass
- ☐ generalize this problem

14 122. *Best Time to Buy and Sell Stock II

14.1 Problem Statement

Link

14.2 Analysis

14.2.1 Initial Analysis (greedy algorithm, failed)

When making decisions, how do you know higher profit lies ahead? How do you make the ideal choice when you only have local information, not the global information?

In the example of greedy algorithm (the Dijkstra's algorithm), the shortest distance a node can get is updated when calculating new distances from a node being visited to unvisited neighbor nodes. If the newly calculated distance is shorter than the old distance, it is updated to reflect the more "global view" of ideal solution. The global aspect of the data is used during decision making because the calculated distance is the sum from beginning to the target node.

Back to this problem. We may need to **STORE** the max profit to each node, just as we store the minimum accumulative distance in each node in Dijkstra's algorithm's example. In this problem, the concept of neighboring is different from the neighboring nodes in graph. A node and its neighbor can be treated as a buying node and a selling node. So, for a specific node, all nodes after it can be viewed as neighboring node.

14.2.2 Initial Analysis (plot and recoganize the pattern, greedy algorithm)

Take price sequence `[7,1,5,3,6,4]` as an example. We want to maximize the profit, the the rule of thumb for buying a stock at day `i` is, the price at the day `i + 1` is higher than the current day. The rule of thumb for selling a stock at day `i` is, the price at day `i + 1` is lower than day `i`. If `prices[i + 1] > prices[i]`, we can hold the stock and wait until day `i + 1` to sell it, to maximize the profit. For each buy-sell operation, we seek to maximize profit of it.

Problems which can be solved by greedy algorithm has following two requirements:

1. Greedy choice property: a global optimal solution can be achieved by choosing the optimal choice at each step
2. Optimal substructure: a global optimal solution contains all optimal solutions to the sub-problems

Back to our problem, in order to achieve the over all maximum profit, we need to achieve sub-maximum profit for each price change cycle. So, when buying stock, if `prices[i + 1] < prices[i]`, we may want to buy at day `i + 1` rather than day `i`. Also, when selling stock, if we find a price drop (i.e. `prices[i + 1] < prices[i]`, we may want to sell our stock at day `i` rather than day `i + 1`, so we can get more profit.

The algorithm steps are as follows:

- check if size of `prices` is less than two, if so, return 0
- define a variable `max_profit` and initialize it to 0
- go over the `prices` array. For a certain day `i`, if we find `prices[i + 1] <= prices[i]`, we don't buy the stock. And since we didn't buy it, we can't sell it, so we go to next day.
- if we find `prices[i + 1] > prices[i]`, we buy the stock. And we traverse from day `i + 1` to end of the array. If we find a day `j`, such that `prices[j + 1] < prices[j]`, it means the price will drop at day `j + 1`, so we'd better sell it on day `j`. So, the profit is updated by the profit earned in this transaction, which is: `prices[j] - prices[i]`.
- repeat the buying and selling process until we reach the end of the array.

The time complexity of this approach is $O(N)$. Since you only need to go over the array once. When you found a buying day, you continue to search a selling day, after you found a selling day, you don't go back, you start from the next day of the selling day. Thus the total time complexity is $O(N)$.

The space complexity of this approach is $O(N)$ (didn't copy the entire array, the memory used is not related to the total size of the array).

14.3 Solution

14.3.1 C++

plot and recoganize the pattern (greedy algorithm, my first try)

```
class Solution {
public:
  int maxProfit(vector<int>& prices) {
    if (prices.size() < 2)
      return 0;

    int max_profit = 0;

    for (int i = 0; i < prices.size() - 1; i++) {
      if (prices[i + 1] <= prices[i])
        continue;

      int j = i + 1;
      while (j < prices.size() - 1 && prices[j + 1] > prices[j])
        j++;

      max_profit += prices[j] - prices[i];
      i = j;

    }

    return max_profit;
  }
};
```

14.4 todos [2/4]

- ☒ write down your own solution and analysis
- ☒ time complexity analysis of your own solution
- ☐ check solution/discussion page for more ideas, implement them, and write down corresponding analysis (including time and space complexity analysis
  - ☐ brutal force
  - ☐ peak valley
  - ☐ simple one pass

☐ generalize this problem

15 136. Single Number

15.1 Problem Statement

Link

15.2 Analysis

15.2.1 Hash Table

A hash table can be used to store the appearing information of each element. We can traverse the array, and try to find if each element encountered is in the hash table or not. If so, we remove it from the hash table. If not, we insert it into the hash table. The final remaining element would be the single number. This leads to hash table solution 1. The average time complexity for each hash table operation (`insert()`, `find()`, `erase()`) are constant, the worst case for them are linear. Thus, the total average case is $O(N)$, the total worst case is $O(N^2)$.

We can improve this a little by using an integer `sum`. Its initial value is zero. Each time we encounter an element, we check if it is in the hash table. If so, we subtract it from `sum`. If not, we add it to `sum`, and insert it into the hash table. This approach doesn't have to call `erase()`, the subtraction does the job, and time complexity of this step is guaranteed constant. Although the total average and worst time complexity is the same as the above one, it can run faster for certain cases. This leads to hash table solution 2.

15.3 Solution

15.3.1 C++

hash table I: 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();
  }
};
```

hash table II: 37%, 15%

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

    int sum = 0;

    for (auto num : nums) {
      if (record.find(num) == record.end()) {
        sum += num;
        record.insert(num);
      }

      else
        sum -= num;
    }

    return sum;
  }
};
```

15.4 todos [3/4]

- ☒ 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)
  - ☒ brutal force: use another array to hold
  - ☐ math
  - ☐ bit manipulation
- ☐ read discussion, do additional work (internalize it and write analysis and code)

16 160. Intersection of Two Linked Lists

16.1 Problem Statement

Link

16.2 Analysis

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

16.2.1 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))*.

16.2.2 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.

16.2.3 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 $O(1)$.

16.2.4 Two pointer

16.3 Solution

16.3.1 C++

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;
  }
};
```

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;
  }
};
```

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;

  }
};
```

16.4 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

17 167. Two Sum II - Input array is sorted

17.1 Problem Statement

17.2 Analysis

17.2.1 Using similar idea in 1. Two Sum.

In this problem, the array has already been sorted. So we can start from the array, for each encountered element, we calculate the corresponding counterpart. Then we use binary search to find out if it exists in the array. The range is from the next element to the last element.

The first submission was not passed. Then I made some optimization (just for this case). They are:

1. if the current element is the same as the previous element, we pass to next (because the previous element didn't find match, this one won't either)
2. if the counterpart is larger than the largest element in the array, or its smaller than the current element, we pass to next. Since in this situation, no match is possible.
3. in the binary search function, a constant reference was used to avoid copying of the original array.

17.3 Solution

17.3.1 C++

binary search (69%, 90%)

```
class Solution {
public:
  int binarySearch(int target, int index, const vector<int>& nums) {
    int start_index = index;
    int end_index = nums.size() - 1;
    int middle;

    while (start_index <= end_index) {
      middle = (start_index + end_index) / 2;
      if (nums[middle] == target)
        return middle;

      else if (nums[middle] < target)
        start_index = middle + 1;

      else
        end_index = middle - 1;
    }

    return -1;
  }


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

    for (int i = 0; i < nums.size(); i++) {
      //check if duplicate encountered
      if (i > 0 && nums[i] == nums[i - 1])
        continue;

      int counter_part = target - nums[i];

      // check range
      if (counter_part > nums.back() || counter_part < nums[i])
        continue;

      int counter_part_index = binarySearch(counter_part, i + 1, nums);

      if (counter_part_index != -1) { // match found
        index_1 = i;
        index_2 = counter_part_index;
        break;
      }

    }

/*      if (index_1 > index_2)
      return {index_2, index_1};
    else
      return {index_1, index_2}; */
    return {index_1 + 1, index_2 + 1};
  }
```

```
};

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

[1,2,3,4,5,6,7,8,9,10]
10

[2,5,7,9,11,16,17,19,21,32]
25

*/
```

17.4 todos [1/3]

- ☒ write down your own solution
- ☐ try to think about two-pointers method (check discussion page)
- ☐ check discussion page for more ideas

18 169. Majority Element

18.1 Problem Statement

Link

18.2 Analysis

18.2.1 `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.

18.3 Solutions

18.3.1 C++

`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;
      }

      else
        frequency_count.insert(make_pair(num, 1));
    }

    return nums[0];
  }
};
```

18.4 todos [1/3]

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

19 198. House Robber

19.1 Problem Statement

Link

19.2 Analysis

19.3 Solution

19.4 todos [0/4]

- ☐ write down your own solution and analysis
- ☐ time complexity analysis of your own solution
- ☐ check solution/discussion page for more ideas, implement them, and write down corresponding analysis (including time and space complexity analysis
- ☐ generalize this problem

20 206. Reverse Linked List

20.1 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.

20.2 Analysis

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

20.2.1 Using Stack (my)

20.2.2 Recursion (my)

20.3 Solution

20.3.1 C++

Using Stack. time (96%), space (5%) This method uses a stack to keep the reverse order. Additional memory is required.

```cpp
/**
 * Definition for singly-linked list.
 * struct ListNode {
 *     int val;
 *     ListNode *next;
 *     ListNode(int x) : val(x), next(NULL) {}
 * };
 */
class Solution {
public:
  ListNode* reverseList(ListNode* head) {
    stack<ListNode*> nodes;
    // check if head is nullptr
    if (head == nullptr)
      return head;

    // store the list in stack
    while (true) {
      if (head->next != nullptr) {
        nodes.push(head);
        head = head->next;
      }

      else { // head is pointing the last node
```

```
        nodes.push(head);
        break;
      }
    }

    // start re-connect
    head = nodes.top();
    nodes.pop();
    ListNode* last_node = head;

    while (!nodes.empty()) {
      last_node->next = nodes.top();
      last_node = last_node->next;
      nodes.pop();
    }

    last_node->next = nullptr;

    return head;
  }
};
```

Using Recursion. time (18%), space (21%) This approach is a very "akward" way to use recursion.

20.4 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

21 226. Invert Binary Tree

21.1 Problem Statement

Link

21.2 Analysis

21.2.1 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.

21.3 Solution

21.3.1 C++

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) {}
 * };
 */
class Solution {
public:
  TreeNode* invertTree(TreeNode* root) {
    if (root == nullptr)
      return root;

    TreeNode* temp = root->left;
    root->left = invertTree(root->right);
    root->right = invertTree(temp);

    return root;
  }
};
```

21.4 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

22 242. Valid Anagram

22.1 Problem Statement

Link

22.2 Analysis

22.2.1 Sort

22.2.2 Hash Table

22.3 Solution

22.4 todos [/]

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

23 268. Missing Number

23.1 Problem Statement

23.2 Analysis

23.2.1 math

The sum of 1 to n is `n * (n + 1) / 2`. So, we can calculate this first and then traverse the array, subtract each element from the sum. The final remaining number is equal to the missing number.

23.3 Solution

23.3.1 C++

math: time $O(N)$, space $O(1)$

```
class Solution {
public:
  int missingNumber(vector<int>& nums) {
    int sum = nums.size() * (nums.size() + 1) / 2;

    for (auto& num : nums) {
      sum -= num;
    }

    return sum;
  }
};
```

23.4 todos [1/3]

- ☒ write down your solution and analysis
- ☐ read solution page, understand and implement each solution, and write down analysis
  - ☐ sorting
  - ☐ hash table
  - ☐ bit manipulation
- ☐ generalize this problem

24 283. Move Zeros

24.1 Problem Statement

Link

24.2 Analysis

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

24.2.1 Two pointers

We can use two iterators to scan and find out zero and non-zero elements in the array, and swap them. For example, we have the following array:

```
[3,1,0,5,4,6,0,9]
 ^
```

We use an iterator to traverse this array, starting from the first element. If we find zero:

```
[3,1,0,5,4,6,0,9]
     ^
```

we will start another iterator to scan through the rest of the array, and find out the first non-zero element:

```
[3,1,0,5,4,6,0,9]
     ^ ^
     a b
```

as shown above, iterator `a` is pointing to an encountered zero, and iterator `b` is pointing to the first non-zero element after `a`. Then we swap these two elements:

```
[3,1,5,0,4,6,0,9]
     ^ ^
     a b
```

If the rest of the array are all zero, we can just return, because the array is already in shape, for example:

```
[3,1,5,0,0,0,0,0] end
       ^           ^
       a           b
```

We do this until `a` goes to the end of the array or `b` encountered `nums.end()` while searching first non-zero

24.3 Solution

24.3.1 C++

bubble sort. time (5%) space (75%) Too slow, time complexity is $O(N^2)$.

```cpp
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;
          swap(*iter, *(iter + 1));
          swapped = true;
        }
      }
    } while (swapped);
  }
};
```

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 $O(N^2)$.

```cpp
class Solution {
public:

  void moveZeroes(vector<int>& nums) {
    int zero_count = 0;
    for (auto iter = nums.begin(); iter != nums.end(); ++iter)
      if (*iter == 0)
```

```
        zero_count++;

    if (zero_count == 0)
      return;

    auto iter = nums.begin();
    int zero_deleted = 0;

    while (zero_deleted < zero_count) {
      if (*iter == 0) {
        iter = nums.erase(iter);
        nums.push_back(0);
        zero_deleted++;
      }

      else
        ++iter;
    }
  }
};
```

two pointers. time (5%) space (80%)

```
class Solution {
public:
  void moveZeroes(vector<int>& nums) {
    for (auto itr_a = nums.begin(); itr_a != nums.end() - 1; ++itr_a) {
      if (*itr_a == 0) {
        auto itr_b = itr_a;
        while (itr_b < nums.end() && *itr_b == 0)
          ++itr_b;

        if (itr_b == nums.end())
          return;

        swap(*itr_a, *itr_b);
      }
    }
  }
};
```

24.4 todos [2/3]

- ☒ try to think another Solution
- ☒ write down two pointer solution

- ☐ read the solution page and study
    - ☐ analyze solutions, implement them
    - ☐ analyze your solution: time complexity, any extra work performed so it is slow

25 344. Reverse String

25.1 Problem Statement

Link

25.2 Analysis

25.2.1 Direct swap

The problem requires to reverse in place. So you can swap each "pairing element" in the array. For example:

`['a', 'b', 'c', 'd', 'e']`

You swap 'a' and 'e', 'b' and 'd'.

25.3 Solution

25.3.1 Python

direct swap

```
class Solution:
    def reverseString(self, s: List[str]) -> None:
        """
        Do not return anything, modify s in-place instead.
        """
        for i in range(int(len(s) / 2)):
            # c = s[i]
            # s[i] = s[-i - 1]
            # s[-i - 1] = c
            s[i], s[-i - 1] = s[-i - 1], s[i]
```

25.3.2 C++

direct swap

```
class Solution {
public:
  void reverseString(vector<char>& s) {
    for (int i = 0; i < s.size() / 2; ++i) {
      swap(s[i], s[s.size() - i - 1]);
    }
  }
};
```

25.4 todos [1/2]

- ☒ write down your own solution
- ☐ check discussion page

26 389. Find the Difference

26.1 Problem Statement

Link

26.2 Analysis

26.2.1 Sort

The two strings will have only one difference. We can just sort the two strings (use `std::sort()`, time complexity is $O(N \log N)$). Then we traverse the two strings, return the first character that is different. If no difference found to the end of the original string, we return the last character in the second string.

The time complexity would be $O(N \log N)$, which is the amount of time `std::sort()` requires.

26.2.2 Summation

We can add (the ASCII value) all characters in `s` together to get `sum_s`, then add all characters in `t` together to get `sum_t`. Then, the different character's ASCII value should be `sum_t - sum_s`.

The time complexity to add all characters for a certain string is $O(N)$. So the total time complexity is $O(N)$.

26.2.3 Hash Table

We can record all characters in `s` into a hash table that duplicates are allowed (unordered_multiset in C++). This is because `s` might have duplicates. Then, we go through `t` and try to find if the character we encountered is also in the hash table. If so, we have to erase it from the hash table (this is for situations like `s` has one certain character, but `t` has two of this). If not in the hash table, then this is the difference character.

This solution's time complexity is comparable to the sort method. Although the average case should be $O(N)$. It uses extra space to hold the hash table, so its space complexity is also high $O(N)$.

26.3 Solution

26.3.1 C++

sort

```
class Solution {
public:
  char findTheDifference(string s, string t) {
    sort(s.begin(), s.end());
    sort(t.begin(), t.end());

    for (auto s_i = s.begin(), t_i = t.begin(); s_i < s.end(); ++s_i,
     ↪  ++t_i) {
      if (*s_i != *t_i)
        return *t_i;
    }

    return t.back();
  }
};
```

sum

```
class Solution {
public:
  char findTheDifference(string s, string t) {
    int sum = 0;

    for (auto ch : s)
      sum += ch;

    for (auto ch : t)
      sum -= ch;

    return static_cast<char>(-sum);
  }
};
```

hash Table

```
class Solution {
public:
  char findTheDifference(string s, string t) {
    unordered_multiset<char> record;

    // go over s and record each character
    for (auto ch : s)
      record.insert(ch);

    // go over t and check each character
    char difference;

    for (auto ch : t) {
      auto itr = record.find(ch);

      if (itr == record.end()) {
        difference = ch;
        break;
      }

      record.erase(itr);
    }

    return difference;

  }
};
```

26.4 todos [1/3]

- ☒ write down your solution and analysis
- ☐ check discussion page, work on that
- ☐ generalize the problem

27 406. Queue Reconstruction by Height

27.1 Problem Statement

Link

27.2 Analysis

27.2.1 Swap person in the queue

The idea is similar with bubble sort. We traverse the array, for each person, we determine whether it needs swap or not. A person needs swap means, the number of people with equal or greater height than this person before it is different from the corresponding number indicated by this person. For example:

```
[[7,0], [4,4], [7,1], [5,0], [6,1], [5,2]]
```

The first person says its height is 7, and there are 0 person infront of it has a height greater or equal 7. This is true, this person doesn't need swap. Then we go to the next person, which is [4,4]. It says the person's height is 4 and there should be 4 persons in front of it whose height is at least 4. This is not the case, so person [4,4] needs to swap.

If the counted number is lower than the indicated number, we should move the current person forward, otherwise, we move the current person backward. To swap, we traverse forward or backward to determine the final position of where should we swap to. Then we do the swap. Take above array as example, we need to swap backward, and [7,1], [5,0], [6,1] are all having a height equal or greater than [4,4], so we swap [4,4] and [6,1]. Then we marked this round has been swapped.

We keep scanning and swapping the array until a run has not swapped, which means each person is at the right position.

*[?]*The time complexity of a single run is $O(N^2)$, plus additional costs to run until no swapped happened during a run.

27.2.2 Place person to the right position directly

From hint 1:

> What can you say about the position of the shortest person? If the position of the shortest person is i, how many people would be in front of the shortest person?

From hint 2:

> Once you fix the position of the shortest person, what can you say about the position of the second shortest person?

27.3 Solution

27.3.1 C++

swap

```
class Solution {
public:
  vector<vector<int>> reconstructQueue(vector<vector<int>>& people) {
    // check empty
    if (people.empty())
      return people;

    bool swapped = true;
    int size = people.size();
    int count;
    int swap_index;
    while (swapped) {
      swapped = false;
      for (int i = 0; i < size; i++) {
        count = 0;
        // count # of people who has greater or equal height with people[i]
        for (int j = 0; j < i; j++)
          if (people[j][0] >= people[i][0])
            count++;

        // check count
        if (count > people[i][1]) {
          swap_index = i;
          while (count > people[i][1]) {
            swap_index--;
            if (people[swap_index][0] >= people[i][0])
              count--;
          }
          swap(people[swap_index], people[i]);
          swapped = true;
        }

        else if (count < people[i][1]) {
          swap_index = i;
          while (count < people[i][1]) {
            swap_index++;
            if (people[swap_index][0] >= people[i][0])
              count++;
          }
          swap(people[swap_index], people[i]);
          swapped = true;
        }
      }
    }

    // when while loop terminates, no swap happened, all people in position
    return people;
  }
};
```

27.4 todos [0/4]

- ☐ write down your own solution and analysis
- ☐ time complexity analysis of your own solution
- ☐ check solution/discussion page for more ideas, implement them, and write down cor-

responding analysis (including time and space complexity analysis

- ☐ generalize this problem

28 412. Fizz Buzz

28.1 Problem Statement

Link

28.2 Analysis

28.2.1 Naive solution (check each condition and add string)

28.2.2 String concatenation

For each sub-condition satisfies, concatenate the specific string to it. This is neater than the naive solution.

28.3 Solution

28.3.1 C++

naive Solution

```
class Solution {
public:
  vector<string> fizzBuzz(int n) {
    vector<string> ret;

    for (int i = 1; i <= n; i++) {
      if (i % 3 == 0) {
        if (i % 5 == 0)
          ret.push_back("FizzBuzz");
        else
          ret.push_back("Fizz");
      }

      else if (i % 5 == 0)
        ret.push_back("Buzz");

      else
        ret.push_back(to_string(i));
    }

    return ret;
  }
};
```

string concatenation

```
class Solution {
public:
  vector<string> fizzBuzz(int n) {
    vector<string> ret;
    vector<int> check_num{3, 5};
    vector<string> add_term{"Fizz", "Buzz"};
    string temp;

    for (int i = 1; i <= n; i++) {
      temp.clear();
      for (int j = 0; j < check_num.size(); j++) {
        if (i % check_num[j] == 0)
          temp += add_term[j];
      }

      if (temp.empty())
        ret.push_back(to_string(i));
      else
        ret.push_back(temp);
    }

    return ret;
  }
};
```

28.4 todos [1/6]

- ☒ write down your own solution and analysis
- ☐ time complexity analysis of your own solution
- ⊟ check solution/discussion page for more ideas
  - ☒ string concatenation
  - ☐ hash
- ☐ implement them, and write down corresponding analysis
- ☐ time complexity of these Solutions
- ☐ generalize this problem

29 437. Path Sum III

29.1 Problem Statement

Link

29.2 Analysis

29.2.1 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:

```
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`.

29.3 Solution

29.3.1 C++

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);
  }
};
```

29.4 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

30 442. Find All Duplicates in an Array

30.1 Problem Statement

Link

30.2 Analysis

30.2.1 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.

30.3 Solution

30.3.1 C++

Label duplicate number (96%, 16%) This one use an extra vector to hold the labeling information.

```cpp
class Solution {
public:
  vector<int> findDuplicates(vector<int>& nums) {
    vector<int> duplicate;
    vector<int> frequency_count(nums.size(), 0);

    for (int i = 0; i < nums.size(); i++) {
      frequency_count[nums[i] - 1]++;
    }

    for (int i = 0; i < frequency_count.size(); i++)
      if (frequency_count[i] > 1)
        duplicate.push_back(i + 1);

    return duplicate;
  }
};
```

30.4 todos [/]

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

31 448. Find All Numbers Disappeared in an Array

31.1 Problem Statement

Link

31.2 Analysis

31.2.1 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.

31.2.2 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.

31.3 Solution

31.3.1 C++

Label appearance of numbers (97%, 15%) Space can be optimized by using original passed-in vector.

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

```

```cpp
    // 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;
  }
};
```

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;
  }
};

// Total complexity: average: O(N), worst: O(N^2), still bound by O(N^2)
```

31.4 todos [/]

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

32 461. Hamming Distance

32.1 Problem Statement

Link

32.2 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
```

32.3 Solution

32.3.1 C++

Time(14.63%)

```cpp
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;
  }
};
```

Time(94.5%)

```cpp
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;
  }
};
```

Questions Why the second solution is faster than the previous one?

- Bitwise XOR used.

32.3.2 Python

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?

33 477. Total Hamming Distance

33.1 Problem Statement

Link

33.2 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

33.2.1 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.

33.2.2 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.

33.3 Solution

33.3.1 C++

Not Accepted (too slow) This algorithm is too slow.

```
class Solution {
public:
  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;
  }
};
```

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. . . ..

```
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);
        else
          LSB_ones.push_back(i);

        // update i and non_zero_count
        i = i >> 1;
        if (i)
          non_zero_count++;
      }

      // update count
      count += LSB_ones.size() * LSB_zeros.size();
    }

    return count;
  }
};
```

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.

Grouping_example. time (88.24%, 49.76%) This is from the discussion (grouping idea).

```
class Solution {
public:
  int totalHammingDistance(vector<int>& nums) {
    if (nums.size() <= 0) return 0;

    int res = 0;

    for(int i=0;i<32;i++) {
      int setCount = 0;
      for(int j=0;j<nums.size();j++) {
          if ( nums[j] & (1 << i) ) setCount++;
      }

      res += setCount * (nums.size() - setCount);
    }

    return res;
  }
};
```

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()`.

33.4 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

34 543. Diameter of Binary Tree

34.1 Problem Statement

Link

34.2 Analysis

34.2.1 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`

34.3 Solution

34.3.1 C++

direct recursion (5%, 5%)

```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:
  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));
  }
};
```

34.4 todos [/]

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

35 557. Reverse Words in a String III

35.1 Problem Statement

Link

35.2 Analysis

35.2.1 Python-use `reversed()` and `split()`

This approach is trivial, including using two built-in methods: `reversed()` and `split()`. First, the input string is splitted into a list. Each element in the list is a word in the string (separated by whitespaces). Then, these substrings were reversed and added to another empty list to form the sentence. A space should be added after each word. You have to get rid of tailing space.

35.3 Solution

35.3.1 Python

use `reversed()` and `split()`

```python
class Solution:
    def reverseWords(s: str) -> str:
        words = s.split()
        reversed_list = []
        for word in words:
            reversed_list.append(''.join(list(reversed(word))) + ' ')
        reversed_list[-1] = reversed_list[-1][:-1]
        return ''.join(reversed_list)
```

35.4 todos [1/2]

- ☒ write down your own solution and analysiss
- ☐ read discussion/solution page for more ideas

36 559. Maximum Depth of N-ary Tree

36.1 Problem Statement

36.2 Analysis

36.2.1 Recursion

The recursion idea is similar with 104. Maximum Depth of Binary Tree. In this problem, the tree is N-ary rather than binary. And the node struct of the tree is slightly different from the binary tree. A vector is used to keep record of all the child nodes of one parent node. So, when doing recursion, you traverse the vector and apply the recursive function for each child.

We are allowed to modify the tree node. So we can store the intermediate result in the node->val. Details are shown in the code.

36.2.2 DFS

36.3 Solution

36.3.1 C++

recursion

```
/*
// Definition for a Node.
class Node {
public:
    int val;
    vector<Node*> children;

    Node() {}

    Node(int _val, vector<Node*> _children) {
        val = _val;
        children = _children;
    }
};
```

```
*/
class Solution {
public:
  void maximum(Node* t) {
    // maximum value will be stored in t->val
    if (t->children.size() == 0) {
      t->val = 1;
      return;
    }

    // t has some children, get them maximum value first
    for (auto& child : t->children) {
      maximum(child);
    }

    // now each child's maximum depth is stored in their val
    // find out the maximum
    int child_max_depth = 0;
    for (const auto& child : t->children) {
      if (child->val > child_max_depth)
        child_max_depth = child->val;
    }

    t->val = child_max_depth + 1;
  }

  int maxDepth(Node* root) {
    if (root == nullptr)
      return 0;

    maximum(root);
    return root->val;
  }
};

/*cases:
{"$id":"1","children":[{"$id":"2","children":[{"$id":"5","children":[],"val
↪  ":5},{"$id":"6","children":[],"val":6}],"val":3},{"$id":"3","children":
↪  [],"val":2},{"$id":"4","children":[],"val":4}],"val":1}

{"$id":"1","children":[],"val":1}

*/
```

36.4 todos [1/5]

- ☒ write down your recursive solution
- ☐ think about DFS traversal, implement and write down analysis
- ☐ read discussion panel
- ☐ try new idea
- ☐ generalize

37 581. Shortest Unsorted Continuous Subarray

37.1 Problem Statement

Link

37.2 Analysis

37.2.1 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.

37.3 Solution

37.3.1 C++

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;
  }
};
```

37.4 todos [1/3]

- ☒ write down your analysis and solution
- ☐ check solution page, study, understand and implement them
- ☐ study first solution (brutal force)

38 617. Merge Two Binary Trees

38.1 Problem Statement

Link

38.2 Analysis

38.2.1 Recursive Method

Use recursiion to solve this problem.

38.2.2 Iterative Method (using stack)

38.3 Solution

38.3.1 C++

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

```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:
  TreeNode* mergeTrees(TreeNode* t1, TreeNode* t2) {
    if (t1 == nullptr)
      return t2;
    else if (t2 == nullptr)
      return t1;
    else {
      TreeNode* node = new TreeNode(t1->val + t2->val);
      node->left = mergeTrees(t1->left, t2->left);
      node->right = mergeTrees(t1->right, t2->right);
      return node;
    }
  }
};
```

Iterative

38.4 todos [0/2]

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

39 653. Two Sum IV - Input is a BST

39.1 Problem Statement

Link

39.2 Analysis

39.2.1 Recursion

We use a recursion to traverse the whole tree. For each node encountered, we use another recursive function to search the whole tree to see whether the counterpart exists in the tree. Be aware that we don't use same node twice, so you have to consider this case in this find counterpart function.

39.3 Solution

39.4 todos [/]

- ☐ write down your own solution and analysis
- ☐ try DFS method
- ☐ check solution page to find out more ideas and implemnent them
- ☐ write down analysis of additional solution
- ☐ generalize the problem

40 657. Robot Return to Origin

40.1 Problem Statement

Link

40.2 Analysis

40.2.1 Determine if instructions are paired

If a `'L'` is paired with a `'R'`, then the effect of their moves will be canceled. Same idea for `'U'` and `'D'`. So, we can just count the total number of the instructions and see if they can cancel each other.

40.3 Solution

40.3.1 Python

count instruction number

```
class Solution:
    def judgeCircle(self, moves: str) -> bool:
        if len(moves) % 2 not 0:
            return false

        return moves.count('L') == moves.count('R') and moves.count('U') ==
         ↪  moves.count('D')
```

40.4 todos [1/2]

- ☒ write down your analysis and solution
- ☐ check discussion page

41 696. Count Binary Substrings

41.1 Problem Statement

Link

41.2 Analysis

41.2.1 Collect meta-substrings

First, we have to be aware that how to divide the input string into separate parts and count. The valid substring should have all 0s and 1s grouped together. So, we may want to divide the string into meta-substrings. A meta-substring is a substring that each 0 and 1 are grouped together. For example, an input of `'001010001111100101'` can be divided into following meta-substrings:

```
'001'
'10'
'01'
'1000'
'0001111'
'111100'
'001'
'10'
'01'
```

For each meta-substring, we count the number of valid substring. It is clear that the total number of valid substring equals to the length of shorter 1s or 0s. For example, `'0001111'` has following substrings: `'000111'`, `'0011'` and `'01'`.

My approach is to use two lists to hold each character. One is for the first appearing number, another is for the second appearing number. The next time we meet the first appearing number again, we stop and check the current meta-substring (which is stored in the two lists). Then, we copy the content in second appearing number list to the first appearing number list and continue to count.

The explanation is messy, you can check the code part.

41.2.2 Count length of single number substring

We can go through the string and count the length of each same-character contiguous blokcs. For example, for string `"011010011100"`, each substring and its length are:

```
substring length
0          1
11         2
0          1
1          1
00         2
111        3
00         2
```

View horizontally:

```
[0 11 0 1 00 111 00]
```

Then, we go through this list of substrings. Each breakpoint has an event of character change (either from 0 to 1, or from 1 to 0). We compare the length of the substrings at left and right side of the breakpoint. The number of valid substrings is the shorter length (explained in the first analysis part, collect meta substrings). Thus we have:

```
[0 11 0 1 00 111 00]
 ----
  1
[0 11 0 1 00 111 00]
   ----
    1
[0 11 0 1 00 111 00]
      ---
       1
[0 11 0 1 00 111 00]
        ----
         1
[0 11 0 1 00 111 00]
          ------
            2
```

```
[0 11 0 1 00 111 00]
             ------
               2
```

So, the total number valid substrings is $1+1+1+1+2+2=8$.

How to count? we use a variable to hold the appearing times of current number `count`. Initially we set it to 1. Then we start from the second character in the string (we set `count = 1` initially, so the first character in the string has been counted for. Also, we don't need to keep record of which character the counting corresponds to, because all we need to track is the breakpoint, where two characters differ from each other).

Start from the second character, and go through the string. If current character is different from the previous character, a breakpoint has been reached. We need to store the current counting to the list, and reset the count to 1 (which corresponds to the current character). By this means, we can't add the last character's data to the list. We do this by adding the `count` to the list after the iterative for loop (because `count` is now holding data corresponding to the last character in string).

The solution is leet code has another way to count (which is better).

After we get the list `lengths`, we can find smaller value for `(lengths[0], lengths[1])`, `(lengths[1], lengths[2])`, . . . , `(lengths[n - 1], lengths[n])`

41.2.3 Count total number on the fly

There is a way to get the final number of valid substrings without using extra space. It is similar with the above analysis. But we don't use a list to hold the length info of each single number substring.

We can calculate the number of valid substrings as soon as we get the length of adjacent 1-substring and 0-substring. So we use `prev` and `cur` to hold the length of previous substring and current substring (the previous substring is composed of character different from the current substring. If previous substring is 111.., the current substring is 000. . .).

Just like the second analysis, when we go througiteratingh the string, if a breakpoint is found, we update the final result (the number of valid substrings) by `min(prev, cur)`. Then, we update `prev` and `cur`: `prev = cur`, `cur = 1`. This is because, the moment we found a breakpoint, our current sequence becomes the previous, and we have to start counting occurance of the real current sequence.

After the iteration, we still need to add `min(prev, cur)` to the final answer as this is not included during the iteration (go through the for loop manually and you'll understand).

41.3 Solution

41.3.1 Python

count meta-strings

```
# count meta-substring
# a meta-substring is the substring that each 0 and 1 are grouped together
# for example, for an input '00101000111100101'
# you have following meta-substrings
# '001'
# '10'
# '01'
# '1000'
# '0001111'
# '111100'
# '001'
# '10'
# '01'
# the number of valid substring is the min of number of 1s or 0s
class Solution:
    def countBinarySubstrings(self, s: str) -> int:
        num_first = []
        num_second = []
        result = 0

        num_first.append(s[0])

        for i in range(1, len(s)):
            if len(num_second) == 0 and s[i] == num_first[0]:
                num_first.append(s[i])

            elif s[i] != num_first[0]:
                num_second.append(s[i])

            elif s[i] == num_first[0]:  # another un-grouped num_first
             ↪  occured, we have to stop here
                result += min(len(num_first), len(num_second))
                num_first = num_second[:]
                num_second.clear()
                num_second.append(s[i])

        result += min(len(num_first), len(num_second))

        return result
```

count length of single number substring

```
class Solution:
    def countBinarySubstrings(self, s: str) -> int:
        count = 1
        lengths = []
        ans = 0

        for i in range(1, len(s)):
            if s[i - 1] != s[i]:
                lengths.append(count)
                count = 1
            else:
                count += 1

        lengths.append(count)

        for i in range(len(lengths) - 1):
            ans += min(lengths[i], lengths[i + 1])

        return ans
```

count the total number on the fly

```
class Solution:
    def countBinarySubstrings(self, s: str) -> int:
        # prev: occuring times of previous num
        # cur: occuring times of current num
        ans, prev, cur = 0, 0, 1
        for i in range(1, len(s)):
            if s[i] != s[i - 1]:
                ans += min(prev, cur)
                prev, cur = cur, 1
            else:
                cur += 1

        ans += min(prev, cur)

        return ans
```

41.4 todos [2/3]

- ☒ write down your analysis and Solution
- ☐ time complexity analysis
- ☒ check solution page and re-implement them
    - ☒ Group by character
    - ☒ Linear scan

42 739. Daily Temperature

42.1 Problem Statement

Link

42.2 Analysis

42.2.1 Brutal force (time limit exceeded)

This solution is not accepted due to high time complexity. For each temperature, we scan the rest temperatures and find out the first one that is higher than it, then we compute the distance between these two. If none found, the distance is set to 0.

The time complexity is $O(N^2)$.

42.2.2 Record information (time limit exceeded)

In the brutal force, we have missed much information. We can, however, use an initial scan to hold the information that the indexes of temperatures that are higher than a certain temperature. We do it as follows:

Declare a 2D array temp[i][j], there are 71 entrys of temp[i], corresponding to the temperature range[30, 100]. Each temp[i] is a vector of int, which stores the index of temperatures in T[] that are higher than value i.

After we obtain such 2D array, we scan T, for each encountered temperature (T[i]), we traverse temp[T[i]], temp[T[i]] is a vector holding indexes that has higher temperature than T[i], we compare each temp[T[i]][j], and find out the first index that temp[T[i]][j] > i, the distance between these two are the value we are looking for. If none exist, the value should be 0.

In implementation, we only allocate 71 slots for temp[][] to record temperatures. So the relation between index and temperature should be:

```
index       temperature
  0             30
  1             31
        ......
  71            100
```

This solution is also very slow. The time complexity for building the 2D array is $O(71N)$, the time complexity for searching appropriate index is unfortunately, still $O(N^2)$ for extreme cases.

One point we can optimize is when we searching the 2D array, we use binary search instead of linear search. This can increase the speed to an acceptable range (be accepted as a solution), but it is still **VERY SLOW**. The time complexity of this part using binary search is: $N \log N$.

42.3 Solution

42.3.1 C++

brutal force (time limit exceeded)

```
vector<int> dailyTemperatures(vector<int>& T) {

  vector<int> counts;
  int count;

  for (int i = 0; i < T.size(); i++) {
    count = 0;

    for (int j = i + 1; j < T.size(); j++)
      if (T[j] > T[i]) {
        count = j - i;
        break;
      }

    counts.push_back(count);
  }

  return counts;

}
```

record information (binary search, accepted, very slow)

```
vector<int> dailyTemperatures(vector<int>& T) {
  // build the temp[][] 2D record array
  vector<vector<int>> temp(71);
  for (int i = 0; i < T.size(); i++)
    for (int j = 0; j < T[i] - 30; j++)
      temp[j].push_back(i);

  // find out answer
  vector<int> ret(T.size());
  int count;

  for (int i = 0; i < ret.size(); i++) {
    count = 0;

    for (int j = 0; j < temp[T[i] - 30].size(); j++)
      if (i < temp[T[i] - 30][j]) {
        count = temp[T[i] - 30][j] - i;
        break;
      }
    ret[i] = count;
  }

  return ret;
}
```

42.4 todos [2/4]

- ☒ write down your own solution and analysis
- ☒ time complexity analysis of your own solution
- ☐ check solution/discussion page for more ideas, implement them, and write down corresponding analysis (including time and space complexity analysis
  - ☐ next array
  - ☐ stack
- ☐ generalize this problem

43 771. Jewels and Stones

43.1 Problem Statement

Link

43.2 Analysis

43.2.1 Brutal force

43.3 Solution

43.3.1 C++

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;
  }
};
```

43.4 todos [0/4]

- ☐ write down your own solution and analysis
- ☐ check solution and discussion for other ideas
- ☐ implement other ideas, write down analysis
- ☐ generalize this problem

44 804. Unique Morse Code Words

44.1 Problem Statement

Link

44.2 Analysis

44.2.1 Hash Table

Generally speaking, this problem wants to find how many unique elements in a collection of elements. We can use a hash table to get this done. Since we don't need ordering, we can use an unordered_set.

To solve this problem, simplily translate the word first, then insert the translated Mores phrase into the hash table. `unordered_set` in C++ doesn't allow duplicates, so if an element that is identical to an element inside the hash table, it will not be inserted. At the end, we just simplily return the size of the hash table.

44.3 Solution

44.3.1 C++

hash Table

```
class Solution {
public:
  int uniqueMorseRepresentations(vector<string>& words) {
    // create a vector of string containing the mapping of letter to Morse
     ↪  code
    vector<string> letter_M{".-","-...","-.-.","-..",".","..-.","--.","....⌋
     ↪  ","..",".---","-.-",".-..","--","-.","---",".--.","--.-",".-.","...⌋
     ↪  ","-","..-","...-",".--","-..-","-.--","--.."};

    // go over the input list of words and translate each word into Morse
     ↪  code
    string translate;
    unordered_set<string> records;

    for (const string& word : words) {
      translate.clear();

      for (char ch : word)
        translate += letter_M[ch - 97];

      records.insert(translate);
    }


    // return the size of the hash table
    return records.size();

  }
};

/*cases:
["gin", "zen", "gig", "msg"]

["sut", "zen", "gin", "bmf", "sot", "xkf", "qms", "hin", "rvg", "apm"]

*/
```

44.4 todos [1/3]

- ☒ write down your own solution and analysis
- ☐ check discussion page for more space-efficient solution
- ☐ try to implement and write down your update

45 824. Goat Latin

45.1 Problem Statement

Link

45.2 Analysis

45.2.1 Modify the word directly

The problem has already given the instructions on how to modify the word.

45.3 Solution

45.3.1 Python

modify the word directly

```python
class Solution:
    def toGoatLatin(self, S: str) -> str:
        words = S.split()

        vowel = ('a', 'e', 'i', 'o', 'u')

        for i in range(len(words)):

            if words[i][0].lower() in vowel:
                words[i] += "ma"
            else:
                words[i] = words[i][1:] + words[i][0] + "ma"

            for j in range(i + 1):
                words[i] += 'a'

            words[i] += ' '

        words[-1] = words[-1][:-1]

        return ''.join(words)
```

45.4 todos [1/6]

- ☒ write down your own solution and analysis
- ☐ time complexity analysis of your own solution
- ☐ check solution/discussion page for more ideas

- ☐ implement them, and write down corresponding analysis
- ☐ time complexity of these Solutions
- ☐ generalize this problem

46 929. Unique Email Addresses

46.1 Problem Statement

46.2 Analysis

46.2.1 Python string manipulation

Use member functions provided by python string class to solve this problem.

1. split the string into two parts: before the @ and after the @
2. remove all '.' and characters after '+' in the first part
3. add modified first part + @ + second part (domain part) in a set. Set is used to keep uniqueness of the email address
4. After analyzing all the strings in the given list, return the length of the set, which is the number of unique email address

46.3 Solution

46.3.1 Python

use string member functions

```python
class Solution:
    def numUniqueEmails(self, emails: List[str]) -> int:
        unique_emails = set()

        for email in emails:
            email_list = email.split('@')
            first = email_list[0].replace('.', '').split('+')[0]
            unique_emails.add(first + '@' + email_list[1])

        return len(unique_emails)
```

46.4 todos [1/2]

- ☒ write down your solution and analysis
- ☐ read discussion page

47 938. Range Sum of BST

47.1 Problem Statement

Link

47.2 Analysis

47.2.1 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.

47.2.2 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.

47.3 Solution

47.3.1 C++

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));
  }
};
```

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);
  }
};
```

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;

  }
};
```

47.4 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

48 1021. Remove Outermost Parenthese

48.1 Problem Statement

Link

48.2 Analysis

48.2.1 Stack

We have to first understand a valid parentheses string and a primitive valid parentheses string. This is similar with base of a vector space.

A valid parentheses string can be viewed as a string that has balanced parenthese (by saying balance, I mean the number of '(' and ')' are the same, also their appearing sequence matches). We can use a stack to check the validity of a parentheses string.

Given a string of parentheses, we go from the first character and moving forward, recording each encountered character to a temporary string. When we encounter the first ')' which makes all the previous parentheses characters forming a valid parentheses string, they will make a primitive valid parentheses string. Because it cann't be splitted any further. We can then store the temp to our result, removing the outer parentheses in the process.

In detail, we need to use three constructs to finish this job:

1. a stack used to determine if a valid parentheses string has been encountered.

2. a temp string used to record the sequence of characters before encountering a valid parentheses string.
3. a result string used to collect all temp strings (after the outter parentheses are removed)

Steps:

1. construct two strings (`temp`, `result`) and one stack. The stack will be used to hold all '(' characters encountered.
2. traverse the string from the beginning
3. if we encounter a '(', push into the stack, also add this to temp (which will record the occuring sequence of the characters inside this primitive valid parentheses string)
4. if we encounter a ')', and we have more than one items in stack, we have not reached the end of the first valid parenthese string. We should add this to temp. Then we pop one item in the stack (so the most adjacent '(' is balanced by this ')')
5. if we encounter a ')' and we have only one item in stack, this is the ending ')' of the current primitive valid parentheses string. We pop the stack (so it is now empty and ready for the next recording). Then we traverse `temp` to store the sequence into the result. We start from `temp[1]`, because `temp[0]` is the starting '(' of the current primitive valid parentheses string, which we should trim off.

48.2.2 Two pointers

48.3 Solution

48.3.1 C++

Stack

```
class Solution {
public:
  string removeOuterParentheses(string S) {
    stack<char> ch_stack;
    string result;
    string temp;

    for (char ch : S) {
      if (ch == '(') {
        ch_stack.push(ch);
        temp += ch;
        continue;
      }

      if (ch == ')' && ch_stack.size() == 1) {
        ch_stack.pop();
        // record temp to result, not including the first '('
        for (int i = 1; i < temp.size(); i++)
          result += temp[i];

        // clear temp cache
        temp.clear();
        continue;
      }

      // if the current primitive valid parenthese not ending
      temp += ch;
      ch_stack.pop();

    }

    return result;
  }
};
```

48.4 todos [1/4]

- ☒ write down your own solution and analysis
- ☐ read discussion, collect possible solution ideas
- ☐ think about the possible solution, re-implement them
- ☐ write down analysis for these other solutions

49 1108. Defanging an IP Address

49.1 Problem Statement

Link

49.2 Analysis

49.2.1 Direct replace

Search the string, for each `'.'` encountered, replace it with `'[.]'`

49.3 Solution

49.3.1 Python

direct replace

```python
class Solution:
    def defangIPaddr(self, address: str) -> str:
        return address.replace('.', '[.]')
```

49.4 todos [1/2]

- ☒ write down your own solution
- ☐ check discussion page