Contents

1 0. Template

1.1 Problem Statement

[[]]

1.2 Analysis

1.3 Solution

1.4 todos [0/5]

- ☐ 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
- ☐ read solutions to refine your code
- ☐ 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.

```
1 class Solution {
2 public:
3   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)
^^Ireturn {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++) {
^^Iif (nums[j] == other_part) {
^^I  index_1 = i;
^^I  index_2 = j;
^^I  break;
^^I}
```

```
      }

      if (index_1 != -1)
^^Ibreak;
    }

    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) {
^^Iif (itr_vector[middle] == current_itr) {
^^I  start_index += 1;
^^I  end_index += 1;
^^I  continue;
^^I}

^^Iresult = middle;
^^Ibreak;
      }

      else if (*itr_vector[middle] > target) {
^^Iend_index = middle - 1;
^^Icontinue;
      }

      else if (*itr_vector[middle] < target) {
^^Istart_index = middle + 1;
^^Icontinue;
      }

    }

    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
```

```
^^Ireturn {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)
^^Ireturn middle;

      if (copy[middle] > num) {
^^Iend = middle - 1;
^^Icontinue;
      }

      if (copy[middle] < num) {
^^Ibegin = middle + 1;
^^Icontinue;
      }
    }

    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) {

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

^^Iif (i != second_index) {
^^I  first_index = i;
^^I  break;
^^I}

      }
    }

    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)
^^Ireturn middle;

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

      else
^^Iend_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
^^Iint index = 0;

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

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

^^I  index++;
^^I}

^^Ibreak;
      }
    }

    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 15. 3Sum

3.1 Problem Statement

Link

3.2 Analysis

3.2.1 Two Pointers

The idea is same as P167, except we now want three sum. To use two pointers, we have to first sort the array, which takes $O(N \log N)$ time. Then, we traverse the array. For each number we encountered (`nums[i]`), we calculate the target 2sum that is: `0 - nums[i]`. Then we search if any two numbers in `nums[i+1:]` add to this target. If so, we push it in the result vector. The searching method is two pointers, the same as P167.

Another thing should notice is to avoid duplicate. We'll take `[-4,-1,-1,0,0,0,0,0,1,2]` as an example. First we start with `-4`, the target we are looking for is `4`:

```
-4 -1 -1 0 0 0 0 0 1 2
   ^                 ^
-1 + 2 = 1 < 4

-4 -1 -1 0 0 0 0 0 1 2
         ^           ^
0 + 2 = 2 < 4

-4 -1 -1 0 0 0 0 0 1 2
                  ^ ^
1 + 2 = 3 < 4

No match
```

There is no match for `4`, now we move to `-1`, the target is `1`:

```
-4 -1 -1 0 0 0 0 0 1 2
      ^             ^
-1 + 2 = 1, target found
```

We have found a target, now we push `{-1, -1, 2}` into the result vector. Notice that the array is sorted, it suggests we must move two pointers until they are both pointing to different numbers. This is because to sum to `1`, if you include `-1`, then the other number must be `2`, if you meet

other pair `(-1,2)`, they are duplicate. So you have to move two pointers until you meet two new numbers or two pointers cross each other. To do this:

```
while (++low < high && nums[low] == nums[low-1]);
while (--high > low && nums[high] == high[high+1]);
```

Now we are at:

```
-4 -1 -1 0 0 0 0 0 1 2
         ^           ^
0 + 1 = 1, target found
```

This is another pair, we push it in the result vector. Then we continue move on using same strategy. We will end up with `low > high`, loop terminates. Notice that we only look forward: we don't look back to find pairs, this is because if there is any pair in the back, they must have been found by previous loops and stored in the result vector.

The next number is still `-1`, if the current number is the same as the previous number, we should skip it to avoid duplicate result.

By the above procedures, we can avoid duplicate in our final result vector.

The time complexity of this approach is $O(N^2)$. The Outer loop will traverse each number until we checked `nums[size-3]`, because `nums[size-3]` is the last term that has three numbers available.

Additional notes In this discussion page, we can add a simple `if` statement inside the outer loop (the loop that traverse each number and find valid pairs) to increase the performance:

```
target = -nums[i];
if (target < 0)
  break;
```

Basically, if `target < 0`, it means `nums[i] > 0`, and all `nums[j] > 0` for `j > i`, the sum of three positive numbers can't be zero, so we can just stop here.

3.3 Solution

3.3.1 C++

two pointers

```
class Solution {
public:
  vector<vector<int>> threeSum(vector<int>& nums) {
    // check capacity
    if (nums.size() < 3)
      return {};

    int low, high, prev, target;
```

```
    int size = nums.size();
    vector<vector<int>> ret;

    sort(nums.begin(), nums.end());

    for (int i = 0; i < size - 2; i++) {
      // check if duplicate encountered
      if (i != 0 && nums[i] == prev)
^^Icontinue;

      target = -nums[i];
      low = i + 1;
      high = size - 1;
      while (low < high) {
^^Iif (nums[low] + nums[high] == target) {
^^I  ret.push_back({nums[i], nums[low], nums[high]});
^^I  while (++low < high && nums[low] == nums[low-1]);
^^I  while (--high > low && nums[high] == nums[high+1]);
^^I}
^^Ielse if (nums[low] + nums[high] > target)
^^I  high--;

^^Ielse
^^I  low++;
      }

      prev = nums[i];
    }

    return ret;
  }
};
```

3.4 todos [2/5]

- ☒ 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
- ☐ read solutions to refine your code
- ☐ generalize this problem

4 27. Remove Element

4.1 Problem Statement

Link

4.2 Analysis

4.2.1 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.

4.3 Solution

4.3.1 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) {
^^Ilength++;
^^Icontinue;
      }

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

      if (itr_b == nums.end())
^^Ireturn length;

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

```
    return length;
  }
};
```

4.4 todos [1/2]

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

5 62. Unique Paths

5.1 Problem Statement

Link

5.2 Analysis

5.2.1 Recursion (top-down)

The robot can only travel down or right, one step at a time. The final goal is at `m × n`, the robot starts at `1 × 1`. So it has to go `m - 1` in right direction, `n - 1` in down direction to reach the final goal.

The last step to the final goal could be a moving-right step or a moving-down step. So, the total number of unique paths to reach `(m, n)` is:

```
uniquePath(m, n) = uniquePath(m - 1, n) + uniquePath(m, n - 1);
```

`uniquePath(m - 1, n)` is the number of ways to reach `(m - 1, n)`, then do a moving-down to reach `(m, n)`.

`uniquePath(m, n - 1)` is the number of ways to reach `(m, n - 1)`, then do a moving-right to reach `(m, n)`.

The base case is when `m == 1` or `n == 1`, because this means the number of ways to reach to goal in a one dimensional grid. When `m == 1`, the robot can't move right. When `n == 1`, the robot can move down. So there is only one possible path. In this case, we return 1.

The recursion method will calculate redundant terms. For example, when calculating `uniquePath(m - 1, n) + uniquePath(m, n - 1)`, we are calling `uniquePath(m - 1, n - 1)` twice.

5.2.2 Recursion (top-down with bookkeeping by 2D array)

The first recursion method does a lot of unnecessary redundant calculations. We can avoid this by recording each calculated result. Before calling function recursively to get the result, we first look for the value if it is calculated. If not, we call recursive function to calculate.

For a 2D grid, we need a 2D array `p` to record results. `p[i][j]` means the unique ways to reach `(i, j)`.

5.2.3 Iteration (bottom-up with bookkeeping by 2D array)

To avoid the recursive stack, we can approach the problem in an iterative way. The reason why we need to call recursive function to calculate `uniquePath(m, n)` is, when we need `uniquePath(m, n)`, `uniquePath(m - 1, n)` and `uniquePath(m, n - 1)` is not ready. We can first calculate them before we need `uniquePath(m, n)`.

We still use a 2D array `p` to keep all the intermediate result.

The number of unique ways to reach `(i, j)` is the number of unique ways to reach `(i - 1, j)` plus the number of unique ways to reach `(i, j - 1)`, this is illustrated below:

```
   p2

p1 p

p = p1 + p2
```

For each row (iterate each row), we calculate the unique ways to reach each slot (iterate each column in each row), and record that number. So, in the above illustration, if `p` is at slot `(i,j)`, then `p2 == p[i-1][j]`, `p1 == p[i][j - 1]`. When `i == 1` or `j == 1`, `p[i][j] == 1`.

In this way, we can calculate the intermediate result in a bottom-up fashion.

5.2.4 Iteration (bottom-up with bookkeeping by 1D array)

Take a closer look at the 2D bookkeeping array in the above example, we find that when we calculate the path number at certain slot `x`, we are only using the last calculated row (shown in `o`) and the previous calculated value (`p`), as shown below:

```
••••••••••
ooooOooooo
•••px•••••
x = p + O
```

Thus, we can use a 1D array to keep the information. After we calculate `x`, we update the value in the 1D array so we can use when we calculate path number in the next row. Furthermore, after we update, the previous calculated value (`p`) is just this value:

```
••••••••••
••••••••••
••••x•••••
x = p + O    ooopOooooo (bookkeeping 1D array)
O = x  (O now is set to the just-calculated value, x)
```

The base case when `i == 1` or `j == 1` can be considered by initializing the bookkeeping array with 1 (check the evolving solution code for details).

5.3 Solution

5.3.1 C++

recursion

```cpp
class Solution {
public:
  int uniquePaths(int m, int n) {
    // base case
    if (m == 1 || n == 1)
      return 1;

    return uniquePaths(m - 1, n) + uniquePaths(m, n - 1);
  }
};
```

recursion (with bookkeeping by 2D array)

```cpp
#define dimension 101
class Solution {
public:
  /* const int dimension = 101; */

  int count(int m, int n, int p[dimension][dimension]) {
    // base case
    if (m == 1 || n == 1) {
      p[m][n] = 1;
      return 1;
    }

    // check if result has been calculated
    if (p[m - 1][n] == -1)
      p[m - 1][n] = count(m - 1, n, p);
    if (p[m][n - 1] == -1)
      p[m][n - 1] = count(m, n - 1, p);

    return p[m - 1][n] + p[m][n - 1];
  }

  int uniquePaths(int m, int n) {
    // initialize 2D array
    int p[dimension][dimension];
    for (int i = 0; i < dimension; i++)
      for (int j = 0; j < dimension; j++)
^^Ip[i][j] = -1;
```

```
    return count(m, n, p);
  }
};
```

iterative (with bookkeeping by 2D array)

```
#define dimension 101
class Solution {
public:
  int uniquePaths(int m, int n) {
    // initialize 2D array
    int p[dimension][dimension];

    // bottom up counting
    for (int i = 1; i <= m; i++)
      for (int j = 1; j <= n; j++)
^^Iif (i == 1 || j == 1)
^^I  p[i][j] = 1;
^^Ielse
^^I  p[i][j] = p[i - 1][j] + p[i][j - 1];

    return p[m][n];
  }
};
```

iterative (with bookkeeping by 1D array)

```
#define DIMENSION 100
class Solution {
public:
  int uniquePaths(int m, int n) {
    // initialize 1D array
    int p[DIMENSION];
    for (int i = 0; i < DIMENSION; i++)
      p[i] = 1;

    // bottom up counting
    for (int i = 1; i < n; i++)
      for (int j = 1; j < m; j++)
^^Ip[j] += p[j - 1];

    return p[m - 1];
  }
};
```

5.4 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
  - ☐ read Li Jianchao's post
- ☐ generalize this problem

6 64. Minimum Path Sum

6.1 Problem Statement

Link

6.2 Analysis

6.2.1 Recursion (no bookkeeping)

At each block, you can move right (except the last column) or move down (except the last row). After moving, you can call the function again to calculate the new minSum of the remaining grid resulted from your choice. We need a helper function which accepts the `grid`, and two integer parameters `row` and `col` to indicate the current grid.

There 2 base cases for the recursion function:

1. we reached the last column. Since we can't move right any more, we just move to the last row and calculate the path sum and return.
2. we reached the last row. Since we can't move down any more, we just move to the last column and calculate the path sum and return.

Otherwise, we call function recursively to see which direction gives us smaller sum. This method will calculate redundant sub-grids. For example:

```
|1|3|1|
|1|5|1|
|4|2|1|
move right: minSum = 1 + |3|1|
                         |5|1|
                         |2|1|

move down: minSum = 1 + |1|5|1|
                        |4|2|1|


grid |5|1| is calculated twice.
     |2|1|
```

we are at slot 1. We can either chose moving right or down. The subsequent calculation is shown above, with redundant calculation.

6.2.2 Recursion (bookkeeping)

To avoid redundant calculation, we use a 2D vector to hold the calculated value. Before invoking recursive function to get some value, we check this 2D vector to see if that value is calculated or not. We update this 2D array after each calculation. This can effectively prevent redundant calculation. The space complexity is `m × n`.

6.2.3 Iterative (bookkeeping using original array)

The iterative way is build the min path sum array in a bottom-up way. Notice that we can use the given 2D vector to do the bookkeeping. Just alter the value in a slot, so the value in slot becomes the minimum path sum to reach that slot. We iterate each row, for each row, we iterate each column.

The space complexity is $O(1)$), Time complexity is $O(M \times N)$.

6.3 Solution

6.3.1 C++

recursion (no bookkeeping)

```
class Solution {
public:
  int minSum(vector<vector<int>>& grid, int row, int col) {
    // base case 1: can't move right
    if (col == grid[0].size() - 1) {
      int sum = 0;
      for (int i = row; i < grid.size(); i++)
^^Isum += grid[i][col];
      return sum;
    }

    // base case 2: can't move down
    if (row == grid.size() - 1) {
      int sum = 0;
      for (int i = col; i < grid[0].size(); i++)
^^Isum += grid[row][i];
      return sum;
    }

    // can move right or down
    return grid[row][col] + min(minSum(grid, row + 1, col), minSum(grid,
    ↪  row, col + 1));
  }
```

```
  int minPathSum(vector<vector<int>>& grid) {
    return minSum(grid, 0, 0);
  }
};
```

recursion (bookkeeping)

```
class Solution {
public:
  int minSum(vector<vector<int>>& grid, int row, int col,
  ↪  vector<vector<int>>& val) {
    // check if val[row][col] is calculated or not
    if (val[row][col] != -1)
      return val[row][col];

    // base case 1: can't move right
    if (col == grid[0].size() - 1) {
      val[row][col] = 0;
      for (int i = row; i < grid.size(); i++)
^^Ival[row][col] += grid[i][col];
      return val[row][col];
    }

    // base case 2: can't move down
    if (row == grid.size() - 1) {
      val[row][col] = 0;
      for (int i = col; i < grid[0].size(); i++)
^^Ival[row][col] += grid[row][i];
      return val[row][col];
    }

    // can move right or down
    val[row][col] = grid[row][col] + min(minSum(grid, row + 1, col, val),
    ↪  minSum(grid, row, col + 1, val));
    return val[row][col];
  }

  int minPathSum(vector<vector<int>>& grid) {
    // initialize a value grid to hold calculated value
    vector<vector<int>> val(grid.size(), vector<int>(grid[0].size(), -1));

    return minSum(grid, 0, 0, val);
  }
};
```

iterative (bookkeeping)

```
class Solution {
public:
  int minPathSum(vector<vector<int>>& grid) {
    // calculate first row and first col
    for (int i = 1; i < grid.size(); i++)
      grid[i][0] = grid[i - 1][0] + grid[i][0];

    for (int i = 1; i < grid[0].size(); i++)
      grid[0][i] = grid[0][i - 1] + grid[0][i];

    // calculate min sum to reach each block
    for (int i = 1; i < grid.size(); i++)
      for (int j = 1; j < grid[0].size(); j++)
^^Igrid[i][j] = grid[i][j] + min(grid[i - 1][j], grid[i][j - 1]);

    return grid[grid.size() - 1][grid[0].size() - 1];
  }
};
```

6.4 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

7 70. Climbing Stairs

7.1 Problem Statement

Link

7.2 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.

7.3 Solution

7.3.1 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];
  }
};
```

7.4 todos [/]

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

8 79. Word Search

8.1 Problem Statement

Link

8.2 Analysis

8.2.1 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)
```

8.3 Solution

8.3.1 Recursion and record visited slot (my first approach)

Python

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

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

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

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

^^I# perform next search, according to where the current function call
 ↪  coming from
^^Iif 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:]):
^^I    return True
^^Ielse:
^^I    visited.remove((row, col))  # remove record of failing search
```

```
^^I    return False


    def exist(self, board, word: str) -> bool:
^^Ivisited = set()

^^Ifor row in range(len(board)):
^^I    for col in range(len(board[row])):
^^I^^Iif self.search(board, visited, row, col, word):
^^I^^I    return True

^^Iprint('no match')
^^Ireturn False
```

8.4 todos [1/2]

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

9 94. Binary Tree Inorder Traversal

9.1 Problem Statement

Link

9.2 Analysis

9.2.1 Recursion

For a given node, the inorder traversal is:

- visit its left subtree
- visit its own value
- visit its right subtree

We call the function recursively to visit its left and right subtree. In order to save space, we pass the result vector by reference. The base case is when the `TreeNode` pointer is `nullptr`, for this case, we return directly. Otherwise, we call the function to traverse its left subtree first (the traversal result will be recorded in the vector), then add its own value to the vector, then call the function again to traverse its right subtree. We need to build a helper function to do the actual traversal.

The time complexity of this method is $O(N)$. Since the recursive function is: $T(N) = 2\dot{T}(\frac{N}{2} + 1)$. (or we only visit each node once).

9.2.2 Traversal using Stack (with book keeping visited nodes)

We use a stack `s` to do a DFS traverse of the tree. A hash table will be used to record the visited nodes.

We use a while loop to perform the traversal. The termination condition of the while loop is when the stack is empty (which means no more node to visit). Before the loop, we first push the root into the stack.

Inside the loop, we check whether the left child of `s` exists or not. If it exists and it is not visited, we push it to the stack. Otherwise, it means the left branch of `s.top()` has been visited, we now record its value (`s.top()->val`) and pop it out of the stack. Then we check if its right subtree exists and not visited, if so, we push it into the stack.

This algorithm requires additional space to hold the visiting record.

9.2.3 Traversal using Stack (without book keeping visited nodes)

In fact, we don't need to book keeping the visited nodes. Because we pushed each visited nodes in the stack. We use a `TreeNode` pointer `curr` to hold the next node we are going to push into the stack. Initially, `curr = root`.

The traversal while loop's stop condition is:

1. the next node we are going to push into stack is `nullptr`
2. the stack is empty

The first thing we do in this loop is to push all left subtree of `curr` into the stack, until we meet the left most node, we use a while loop to do this and update `curr` during each run:

```
while (curr != nullptr) {
  s.push(curr);
  curr = curr->left;
}
```

When this loop ends, `curr` is `nullptr`, which is the left child of `s.top()`. We record `s.top()`'s value into the returning vector and pop it out of the stack (because it is already visited). Then we update `curr` so it points to right child of the previous `s.top()`. Because after visiting the node, we should visit its right child, if there is any. These operations are shown below:

```
// now curr == nullptr
curr = s.top();
ret.push_back(curr->val);
s.pop();
curr = curr->right;
```

Whether or not we push `curr` into the stack depends on wether or not its empty. This will be determined in the next run of the while loop. In this way, all tree nodes are visited.

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:
  void in(vector<int>& v, TreeNode* t) {
    // check empty case
    if (t == nullptr)
      return;

    // traverse left subtree
    in(v, t->left);

    // record current node
    v.push_back(t->val);

    // traverse right subtree
    in(v, t->right);
  }

  vector<int> inorderTraversal(TreeNode* root) {
    vector<int> ret;
    in(ret, root);
    return ret;
  }
};
```

stack (use hash table to record visited nodes)

```cpp
class Solution {
public:

  vector<int> inorderTraversal(TreeNode* root) {
    if (root == nullptr)
      return {};
```

```
    stack<TreeNode*> s;  // to hold TreeNode
    s.push(root);
    unordered_set<TreeNode*> visited;  // hold visited nodes
    vector<int> ret;
    TreeNode* temp;

    while(!s.empty()) {
      if (s.top() -> left != nullptr && visited.find(s.top() -> left) ==
       ↪  visited.end()) {
^^Is.push(s.top() -> left);
^^Ivisited.insert(s.top());
      }

      else {
^^Iret.push_back(s.top() -> val);
^^Itemp = s.top() -> right;
^^Is.pop();
^^Iif (temp != nullptr && visited.find(temp) == visited.end()) {
^^I  s.push(temp);
^^I  visited.insert(temp);
^^I}
      }
    }

    return ret;
  }
};
```

stack (no book keeping)

```
class Solution {
public:

  vector<int> inorderTraversal(TreeNode* root) {
    if (root == nullptr)
      return {};

    stack<TreeNode*> s;
    vector<int> ret;
    TreeNode* curr = root;

    while (curr != nullptr || !s.empty()) {
      // trace to end of left subtree
      while (curr != nullptr) {
^^Is.push(curr);
```

```
^^Icurr = curr->left;
      }

      // curr is now nullptr, s.top() has no left child
      curr = s.top();
      s.pop();
      ret.push_back(curr->val);
      curr = curr->right;

    }

    return ret;
  }
};
```

9.4 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
  - ☒ stack
  - ☐ Morris traversal
- ☐ generalize this problem

10 XXXXX95. Unique Binary Search Trees II

10.1 Problem Statement

Link

10.2 Analysis

10.2.1 Analysis of failure

This problem requires building up the structures of all unique BST which stores values 1, 2, …, n. This problem can be solved with dynamic programming, so I think may be I can find the recursive relation in this problem. This is the first step.

Even the brutal force recursive solution is fine, as long as I can solve it. The solution function `generateTrees()` returns a vector of `TreeNode*`, which should be the root of each structurally unique BST storing values of `1, 2, ..., n`.

the relation between `generateTrees(n)` and `generateTrees(n-1)` I To solve it in a recursive way, we should analyze whether solving a smaller part of the problem helps solving the bigger one. `generateTrees(n-1)` will give me a vector of `TreeNode*`, which contains pointers to all root

node of structurally unique BST that storing values of `1, 2, ..., n-1`. Then, how do I use this result to build `generateTrees(n)`? Now I have all possible BST of `1, 2, ..., n-1`, and I want to insert a new value `n` to those trees to make new BST. I have following choices:

1. for each `n-1` trees, insert it to the left child of `n`'s node, making a new BST. We can make `n-1` new BSTs from this strategy. We add them to the back of the vector.
2. try to insert node `n` into each `n-1` subtree. **This is where I think this approach doesn't work. Because there are multiple ways to insert node `n` to a subtree. For example, I can insert to the first right subtree, and re-arrange the rest of previous right subtree to the left, or I can insert to the second right subtree, and re-arrange the rest of previous right subtrees to its left, creating another new BST, or I can just insert to the right most leaf branch**. This needs a lot of calculations and arrangements, so I give up on this idea.

the relation between `generateTrees(n)` and `generateTrees(n-1)` II Think in another way. Initially, I made a vector that holds all root nodes for the whole collection. Then, I call a recursive helper function to build each subtree (e.g. if the root value is `k`, then build the tree by inserting the rest nodes, `1, 2, ..., k-1, k+1, ..., n` to this subtree). We have considered all possible number of structurally unique binary search tree for a given root value `k`. For example, if `n == 4`, then the initial vector would be:

```
[
  [1],
  [1],
  [1],
  [1],
  [1],
  [2],
  [2],
  [3],
  [3],
  [4],
  [4],
  [4],
  [4],
  [4],
]
```

then, we call the recursive helper function to build the rest of these binary tree. The function should accept:

1. a pointer to `TreeNode`, this node is the root of the BST we are going to build.
2. an integer `min`, which is the minimum value in the BST
3. an integer `max`, which is the maximum value in the BST

However, for the same root node, we have multiple choices for the left and right subtree. How to count them one by one? How to prevent making identical subtrees? I don't know how to solve this problem. For example, for the first node `[1]`, we chose `[2]` as the right subtree. And when

working the second node `[1]`, how do we know that structure with `[2]` as the immediate subtree of `[1]` has already been counted? How do we make sure that all BSTs rooted at `[1]` can have unique structure?

10.3 Solution

10.4 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

11 101. Symmetric Tree

11.1 Problem Statement

Link

11.2 Analysis

11.2.1 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`.

11.3 Solution

11.3.1 C++

recursion (75%, 64%)

```
1 /**
2  * Definition for a binary tree node.
3  * 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)
^^Ireturn 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;
  }
};
```

11.4 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

12 104. Maximum Depth of Binary Tree

12.1 Problem Statement

Link

12.2 Analysis

12.2.1 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.

12.3 Solution

12.3.1 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);
  }
};
```

12.4 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

13 108. Convert Sorted Array to Binary Search Tree

13.1 Problem Statement

Link

13.2 Analysis

13.2.1 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 $O(N)$.

13.3 Solution

13.3.1 C++

recursion

```
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);
  }
};
```

13.4 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

14 110. Balanced Binary Tree

14.1 Problem Statement

Link

14.2 Analysis

14.2.1 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.

14.3 Solution

14.3.1 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:
  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);
  }
};
```

14.4 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

15 111. Minimum Depth of Binary Tree

15.1 Problem Statement

Link

15.2 Analysis

15.2.1 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)`.

15.3 Solution

15.3.1 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;
  }
};
```

15.4 todos [1/4]

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

16 112. Path Sum

16.1 Problem Statement

Link

16.2 Analysis

16.2.1 Recursion

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

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

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

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

16.3 Solution

16.3.1 C++

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)
^^Ireturn false;

      if (root->left == nullptr && root->right == nullptr) {
^^Iif (root->val == sum)
^^I    return true;
^^Ielse
^^I    return false;
      }

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

16.4 todos [1/3]

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

17 113. Path Sum II

17.1 Problem Statement

Link

17.2 Analysis

17.2.1 DFS

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

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

17.3 Solution

17.3.1 C++

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())) {
```

```
^^IcheckVal(v, sum);
^^ItraceBack(v);
^^Icontinue;
      }

      if (!isVisited(v.back()->left)) {
^^Ivisited_nodes.insert(v.back()->left); // mark as visited
^^Iv.push_back(v.back()->left);
^^Icontinue;
      }

      if (!isVisited(v.back()->right)) {
^^Ivisited_nodes.insert(v.back()->right); // mark as visited
^^Iv.push_back(v.back()->right);
^^Icontinue;
      }
    }

    return results;
  }
};
```

17.4 todos [1/4]

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

18 121. Best Time to Buy and Sell Stock

18.1 Problem Statement

Link

18.2 Analysis

18.2.1 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.

18.2.2 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.

18.3 Solution

18.3.1 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) {
^^Iprofit = prices[j] - prices[i];
```

```
^^Iif (profit > max_profit)
^^I  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])
^^Iminprice = prices[i];

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

    return maxprofit;
  }
};
```

18.4 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

19 122. Best Time to Buy and Sell Stock II

19.1 Problem Statement

Link

19.2 Analysis

19.2.1 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.

19.2.2 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).

19.3 Solution

19.3.1 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])
^^Icontinue;

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

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

    }

    return max_profit;
  }
};
```

19.4 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

20 136. Single Number

20.1 Problem Statement

Link

20.2 Analysis

20.2.1 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.

20.3 Solution

20.3.1 C++

hash table I: time (16.06%) space (15.74%)

```
1  class Solution {
2  public:
3    int singleNumber(vector<int>& nums) {
4      unordered_set<int> unique_num;
5
6      for (auto num : nums) {
```

```
      auto itr = unique_num.find(num);

      if (itr == unique_num.end())
^^Iunique_num.insert(num);
      else
^^Iunique_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()) {
^^Isum += num;
^^Irecord.insert(num);
      }

      else
^^Isum -= num;
    }

    return sum;
  }
};
```

20.4 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)

21 160. Intersection of Two Linked Lists

21.1 Problem Statement

Link

21.2 Analysis

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

21.2.1 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))*.

21.2.2 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.

21.2.3 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 $O(1)$.

21.2.4 Two pointer

21.3 Solution

21.3.1 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)
^^Ireturn 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())
^^Ireturn 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++)
^^IheadA = headA -> next;
    }

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

      for (int i = 1; i <= skip; i++)
^^IheadB = 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;

  }
};
```

21.4 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

22 167. Two Sum II - Input array is sorted

22.1 Problem Statement

22.2 Analysis

22.2.1 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.

22.2.2 Two Pointers (by Sol)

We can use two pointers starting from begin and end of the array. Take `[2,7,8,11,15]` as an example, the `target` we are looking at is `15`. Initially, two pointers are pointing at begin and end:

```
2  7  8  11  15
^           ^
```

We check if their sum is the target. If the sum is larger than target, we move the right pointer left. For example, `2 + 15 == 17 > 15`, so we move the right pointer:

```
2  7  8  11  15
^        ^
```

If the sum is smaller than target, we move the left pointer right:

```
2  7  8  11  15
   ^     ^
```

Now, `7 + 11 > 15`, so we move right pointer left:

```
2  7  8  11  15
   ^  ^
```

Since we move the pointer one bit at a time, eventually, we'll meet the two nodes that sum to the target (the condition of this while loop is when `low < high` is true, no duplicate allowed, so we stop if `low == high`).

The time complexity is $O(N)$, the worst case is when two pointers must traverse to adjacent elements, say `k` and `k+1`, then total moving steps are: `k + n - (k + 1) - 1`, which is `n - 2` steps. So time complexity is linear with N.

22.3 Solution

22.3.1 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)
^^Ireturn middle;

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

      else
^^Iend_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])
^^Icontinue;

      int counter_part = target - nums[i];
```

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

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

      if (counter_part_index != -1) { // match found
^^Iindex_1 = i;
^^Iindex_2 = counter_part_index;
^^Ibreak;
      }

    }

/*     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

*/
```

22.4 todos [1/3]

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

23 169. Majority Element

23.1 Problem Statement

Link

23.2 Analysis

23.2.1 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.

23.3 Solutions

23.3.1 C++

`unordered_map` **(59%, 42%)** Not very fast.

```
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()) {
^^Ifrequency_count[num] += 1;
^^Iif (frequency_count[num] > nums.size() / 2)
^^I  return num;
      }

      else
^^Ifrequency_count.insert(make_pair(num, 1));
    }

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

23.4 todos [1/3]

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

24 198. House Robber

24.1 Problem Statement

Link

24.2 Analysis

24.2.1 recursion I (book keeping sub-max profit)

Intuitively, we can use recursion to solve this problem. Adjacent houses are not allowed to loot. Given a list of positive integers `nums`, which represents the amount of money of each house, we can say that the max ammount loot way must be either including the first house or not including the first house. So, we just need to compare following two results:

- `nums[0] + rob(nums[2:])`
- `nums[1] + rob(nums[3:])`

The base case is when the `nums[]` passed in has 3, 2, 1 or 0 house(s) left. We can determine the max value easily.

This method is easy to understand, but it has a disadvantage. It will have many redundant calculation. For example, to calculate `rob(nums[2:])`, we need to compare:

- `nums[2] + rob(nums[4:])`
- `nums[3] + rob(nums[5:])`

To calculate `rob(nums[3:])`, we need to compare:

- `nums[3] + rob(nums[5:])`
- `nums[4] + rob(nums[6:])`

We can see that we have calculated `rob(nums[5:])` twice.

To improve this, we have to record what has been calculated. We can use a hash table to store the result of `rob(nums[i:])`, which means the max profit we can get from looting the subarray `nums[i:]`. The hashed key can be `i`, and the value is the calculated profit.

Each time before we recursively calling `rob()` to calculate a profit from looting a subarray, we first check whether this value has been calculated or not. If it is calculated, we use it directly, otherwise we calculate it and add to the hash table so future function call can use it, instead of calculating again.

(In fact, we don't need to use a hash table, a simple array is enough)

24.2.2 recursion II (cleaner, book keeping)

The above recursion works from the begining to the end. We can think in different ways to approach this problem. Reference.

When robbing house `i`, the robber has two options:

1. rob house `i`
2. don't rob house `i`

If choosing 1, the total profit is: `rob(i - 2) + nums[i]`, where `nums[i]` is the profit of robbing house `i`, `rob(i - 2)` is the maximum profit gained from robbing previous houses (to house `i - 2`).

If choosing 2, the total profit is `rob(i - 1)`. Thus, we have following relation:

```
rob(i) = max(rob(i - 2) + nums[i], rob(i - 1))
```

Similarliy, when we calculate `rob(i - 2)` and `rob(i - 1)`, we calculate redundant terms. For example, we calculated `rob(i - 2)` and `rob(i - 3)` twice. So, we use an array `p[i]` to record the value of `rob(i)`.

To implement this recurrent relation, we need a helper function with following header:

```
int rob(const vector<int>& nums, int i)
```

where `i` is the index of house to rob. To calculate the maximum profit, we pass in the last house:

```
rob(nums, nums.size() - 1);
```

The base case of this helper function is simplily `i < 0`, when this is the case, it means back calculating reached beyongd the first house, we return 0.

We use an array `p[]` to Each time before calling `rob()` recursively to calculate `rob(i)`, we check if this has been calculated or not (by checking the value of `p[i]`, we can initialize `p[]` to all -1). If it is already calculated, we return it directly, otherwise, we call `rob()` recursively to calculate.

24.2.3 recursion III (combine I and II)

This approach combines thought in I and II. It requires slightly more edge cases than II, these two methods is essentially the same, especially when comes to efficiency. The code is shown in Solution section.

24.2.4 dynamic programming

Reference

This particular problem and most of others can be approached by following steps:

1. find recursive relation
2. recursive (top-down)
3. recursive + bookkeeping (top-down)
4. iterative + bookkeeping (bottom-up)
5. iterative + N variables (bottom-up)

A problem might be suitable for dynamic programming if it can be splitted into smaller one and solve in similar way (dynamic programming is very similar to recursion). First, we try to find the recursive relation. The reason why we don't rely on recursion solely is that, for some problems, recursive functions are performing redundant work. One example is using recursion to calculate Fibonacci sequence. An other example is this problem (we analyzed this in the recursion I, II section).

At first glance we approach this kind problem in a top-down manner, i.e. from big problem to small problem. For instance, we have:

```
rob(i) = max(rob(i - 2) + nums[i], rob(i - 1))
```

In order to get `rob(i)`, we calculate `rob(i - 2)` and `rob(i - 1)`. This is a top-down approach.

To reduce redundant calculating, we bookkeeping those already calculated value, so we can use it directly if it is asked again. This leads to step 3, and it is how recursion I, II, III works. However, they are still using top-down approach, it involves recursive calls so it is expensive. Look closer to their calculation step, we can find that in order to calculate a top-value (`rob(i)`), they have to recursively call themselves to calculate a down-value first (`rob(i - 1)` and `rob(i - 2)`). The idea is, if we can calculate the down-value first, record them in somewhere, then the top-value can be calculated without a recursive call, just visit those calculated down-value. This leads to step 4, a bottom-up, iterative approach to get the final answer. Bottom-up means we first calculate bottom-value (or down-value), then calculate up-value (or top-value). It is an iterative approach because we don't need to use recursive call to calculate each value, the bottom-up order guarantees that each time we calculate an up-value, we have components (bottom-parts) ready to use.

Back to this problem. We use recursive ideas from recursion II (my original recursion cal be also used, but it needs to calculate from end to begin, I'll try this idea later). To calculate `rob(i)` we have to calculate `rob(i - 2)` and `rob(i - 1)` first. Thinking in this way is thinking in top-down recursive way. How about thinking in this way: we first calculate `rob(i - 2)` and `rob(i - 1)`, then we try to solve `rob(i)`, this is exactly how we approach it via iterative bottom-up approach. You may ask: what is the difference? The core difference is at the time of calculating `rob(i)`, the value of `rob(i - 1)` and `rob(i - 2)` is calculated or not. If it is not calculated, we have to use recursive call to calculate. If it is calculated, we simplily use the value (so no recursion involed). To do the bookkeeping of calculated `rob(i - 1)` and `rob(i - 2)`, we still use an array `p[]`.

We take `nums == [2,7,9,3,1]` as an example. The maximum profit we can get from robbing until house 0 (`nums[0]`) is obviously the value stored in house 0, so we have:

```
nums: 2  7  9  3  1
p:    2
```

The maximum profit we can get from robbing until house 1 is 7. Because we have two options:

1. rob house 0 and leave house 1 un-robbed
2. rob house 1 and leave house 0 un-robbed.

Option 2 has higher profit, so we have:

```
nums: 2  7  9  3  1
p:    2  7
```

The maximum profit we can get from robbing until house 2 is 11. Because we have two options:

1. rob house 2 (profit 9) and all the profit from robbing house until house 0 (2), the total is 11
2. don't rob house 2, you get all the profit from robbing house until house 1 (7), the total is 7

Option 1 has higher profit, so we fill 11 to `p[2]`:

```
nums: 2  7  9  3  1
p:    2  7  11
```

The maximum profit we can get from robbing until house 3 is 11. Because we have two options:

1. rob house 3 (profit 3) and all the profit from robbing house until house 1 (7), the total is 10
2. don't rob house 3, you get all the profit from robbing house until house 2 (11), the total is 11

Option 2 has higher profit, so we fill 11 to `p[3]`:

```
nums: 2  7  9  3  1
p:    2  7  11 11
```

The maximum profit we can get from robbing until house 4 is 12. Because we have two options:

1. rob house 4 (profit 1) and all the profit from robbing house until house 2 (11), the total is 12
2. don't rob house 4, you get all the profit from robbing house until house 3 (11), the total is 11

Option 1 has higher profit, so we fill 12 to `p[4]`.

Now, the problem has been solved. The maximum profit we can get is 12. Pay attention that `p[i]` is the maximum possible profit you can get from robbing house `i`.

There is one more thing we can optimize, which is step 5. If you take a closer look to `p[]`, you'll find all we used is `p[i - 2]` and `p[i - 1]`. In fact, we don't need previous `p[j], j < i - 2` because they are accounted in `p[i - 2]` and `p[i - 1]`. So, we could just use two variables to represent the accumulating maximum profit. We just need to update these two variables after we get each maximum profit along the intermediate steps. We name the two variables as `p1` and `p2`

To depict this process, take `[2,7,9,3,1]` as an example:

```
index:0  1  2  3  4
nums: 2  7  9  3  1
p:    2  7
      p1 p2 ?
```

The array `p` is used as reference. Now we are determining value at `?`, it should be: `max(nums[2] + p1, p2)`, let it be `r`, then we move to next house:

```
index:0  1  2  3  4
nums: 2  7  9  3  1
p:    2  7  11
      p1 p2 r
         p1 p2 ?
```

Now we are determining value at `?`. We update so the current `p1` holds previous `p2`, current `p2` holds calculated `r`. This is how we use two variables to do bookkeeping. Similarliy, when we move to next:

```
index:0  1  2  3  4
nums: 2  7  9  3  1
p:    2  7  11 11
      p1 p2 r  r
         p1 p2 r
            p1 p2 ?
```

After iterating over the `nums[]`, we just return `p2`, because it holds the most recent calculated maximum profit.

24.3 Solution

24.3.1 C++

recursion I (book keeping intermediate result)

```cpp
class Solution {
public:
  unordered_map<int, int> maxResults;

  int maxNonadjacentArray(vector<int>::iterator start, const vector<int>&
  ↪  nums) {
    if (start >= nums.end())
      return 0;

    if (start + 1 == nums.end())
      return *start;

    int max_first, max_second;
    int max_2, max_3, max_4;
    unordered_map<int, int>::iterator itr;

    // try to find max_2, if not, calculate it and insert into maxResults
    itr = maxResults.find(start + 2 - nums.begin());

    if (itr == maxResults.end()) {
      max_2 = maxNonadjacentArray(start + 2, nums);

      maxResults.insert({{start + 2 - nums.begin(), max_2}});
    }

    else
      max_2 = (*itr).second;

    // try to find max_3, if not, calculate it and insert into maxResults
    itr = maxResults.find(start + 3 - nums.begin());
```

```
    if (itr == maxResults.end()) {
      max_3 = maxNonadjacentArray(start + 3, nums);

      maxResults.insert({{start + 3 - nums.begin(), max_3}});
    }

    else
      max_3 = (*itr).second;

    // try to find max_4, if not, calculate it and insert into maxResults
    itr = maxResults.find(start + 4 - nums.begin());

    if (itr == maxResults.end()) {
      max_4 = maxNonadjacentArray(start + 4, nums);

      maxResults.insert({{start + 4 - nums.begin(), max_4}});
    }

    else
      max_4 = (*itr).second;

    // calculate max_first and max_second
    max_first = *start + max(max_2, max_3);

    max_second = *(start + 1) + max(max_3, max_4);

    return max(max_first, max_second);
  }

  int rob(vector<int>& nums) {
    return maxNonadjacentArray(nums.begin(), nums);
  }
};
```

recursion II(better, cleaner approach)

```
class Solution {
public:
  vector<int> p;

  int rob(vector<int>& nums) {
    p.resize(nums.size());
    for (int i = 0; i < nums.size(); ++i)
      p[i] = -1;

    return rob(nums, nums.size() - 1);
```

```
  }

private:
  int rob(vector<int>& nums, int i) {
    if (i < 0)
      return 0;

    if (p[i] >= 0)
      return p[i];  // i-th profit already calculated

    // i-th profit not calculated
    p[i] = max(nums[i] + rob(nums, i - 2), rob(nums, i - 1));
    return p[i];
  }
};
```

recursion III (combine I and II)

```
/*Notes:
This solution combines the idea in review I (my original thought, count
↪  from the beginning of the array) and review II (1. use an array to
↪  hold intermediate result; 2. use index rather than range to regulate
↪  range; 3. shorter and better edge cases)
*/
class Solution {
public:
  vector<int> p;

  int rob(vector<int>& nums) {
    // edge case 1
    if (nums.empty())
      return 0;
    // edge case 2
    if (nums.size() == 1)
      return nums[0];

    p.resize(nums.size());
    for (int i = 0; i < nums.size(); ++i)
      p[i] = -1;

    return max(nums[0] + rob(nums, 2), nums[1] + rob(nums, 3));
  }

private:
  int rob(vector<int>& nums, int i) {
    // base case: out of bound
```

```
    if (i > nums.size() - 1)
      return 0;

    // base case 2: (i + 1) out of bound
    if (i == nums.size() - 1)
      return nums[i];

    if (p[i] < 0)  // not calculated
      p[i] = max(nums[i] + rob(nums, i + 2), nums[i + 1] + rob(nums, i +
       ↪  3));

    return p[i];
  }
};
```

dynamic programming (bottom-up, array bookkeeping)

```
class Solution {
public:
  int rob(vector<int>& nums) {
    if (nums.empty())
      return 0;

    if (nums.size() == 1)
      return nums[0];

    vector<int> p(nums.size());

    p[0] = nums[0];
    p[1] = max(nums[0], nums[1]);

    for (int i = 2; i < nums.size(); i++) {
      p[i] = max(nums[i] + p[i - 2], p[i - 1]);
    }

    return p[p.size() - 1];
  }
};
```

dynamic programming (bottom-up, N-variable bookkeeping)

24.4 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

25 206. Reverse Linked List

25.1 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.

25.2 Analysis

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

25.2.1 Using Stack (my)

25.2.2 Recursion (my)

25.3 Solution

25.3.1 C++

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

```
/**
 * Definition for singly-linked list.
 * struct ListNode {
 *     int val;
 *     ListNode *next;
 *     ListNode(int x) : val(x), next(NULL) {}
 * };
 */
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) {
^^Inodes.push(head);
^^Ihead = head->next;
      }
```

```
23
24        else { // head is pointing the last node
25  ^^Inodes.push(head);
26  ^^Ibreak;
27        }
28      }
29
30      // start re-connect
31      head = nodes.top();
32      nodes.pop();
33      ListNode* last_node = head;
34
35      while (!nodes.empty()) {
36        last_node->next = nodes.top();
37        last_node = last_node->next;
38        nodes.pop();
39      }
40
41      last_node->next = nullptr;
42
43      return head;
44    }
45  };
```

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

25.4 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

26 226. Invert Binary Tree

26.1 Problem Statement

Link

26.2 Analysis

26.2.1 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.

26.3 Solution

26.3.1 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;
  }
};
```

26.4 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

27 242. Valid Anagram

27.1 Problem Statement

Link

27.2 Analysis

27.2.1 Sort

27.2.2 Hash Table

27.3 Solution

27.4 todos [/]

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

28 268. Missing Number

28.1 Problem Statement

28.2 Analysis

28.2.1 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.

28.3 Solution

28.3.1 C++

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

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

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

    return sum;
  }
};
```

28.4 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

29 283. Move Zeros

29.1 Problem Statement

Link

29.2 Analysis

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

29.2.1 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

29.3 Solution

29.3.1 C++

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

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

    // swap array
    do {
      swapped = false;

      for (auto iter = nums.begin(); iter != nums.end() - 1; ++iter) {
^^Iif (*iter == 0) {
^^I  if (*(iter + 1) == 0)
^^I    continue;
^^I  swap(*iter, *(iter + 1));
^^I  swapped = true;
^^I}
      }
    } 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 $O(N^2)$.

```
class Solution {
public:

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

    if (zero_count == 0)
      return;

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

```
15
16      while (zero_deleted < zero_count) {
17        if (*iter == 0) {
18  ^^Iiter = nums.erase(iter);
19  ^^Inums.push_back(0);
20  ^^Izero_deleted++;
21        }
22
23        else
24  ^^I++iter;
25      }
26    }
27  };
```

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) {
^^Iauto itr_b = itr_a;
^^Iwhile (itr_b < nums.end() && *itr_b == 0)
^^I  ++itr_b;

^^Iif (itr_b == nums.end())
^^I  return;

^^Iswap(*itr_a, *itr_b);
      }
    }
  }
};
```

29.4 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

30 344. Reverse String

30.1 Problem Statement

Link

30.2 Analysis

30.2.1 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'.

30.3 Solution

30.3.1 Python

direct swap

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

30.3.2 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]);
    }
  }
};
```

30.4 todos [1/2]

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

31 389. Find the Difference

31.1 Problem Statement

Link

31.2 Analysis

31.2.1 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.

31.2.2 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 $O(N)$.

31.2.3 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 $O(N)$.

31.3 Solution

31.3.1 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)
^^Ireturn *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()) {
^^Idifference = ch;
^^Ibreak;
      }

      record.erase(itr);
    }

    return difference;

  }
};
```

31.4 todos [1/3]

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

32 406. Queue Reconstruction by Height

32.1 Problem Statement

Link

32.2 Analysis

32.2.1 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.

32.2.2 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?

The idea is to put each person directly into the final position. We need an empty array to start working with, declare one named `rp[]`. Assume a person is `[a,b]`. It means the person's height is `a`, and there should be `b` persons with a height greater or equal to `a`. If this person is the shortest person, we can directly place it at `rp[b]`. Why? because there are `b` empty slots in front of this slot, and since this person is the shortest person, any other person placed in these slots have a height higher than the shortest person.

When we place the rest of the persons `[a,b]`, we must be aware that `rp[b]` might not be the final position, because we don't know the number of person with greater or equavalent height of that person in front of this person, so we must count from `rp[0]`, until we have met enough slots that:

1. is empty (which means in future a valid person will be placed here)
2. contains a person with valid height

The slots found by this way may not be used because it might be occupied by other person. So we must navigate the array until we find the first empty slot.

To be familiar with this algorithm, try to work out the following example (already sorted):

```
[[3,4],[4,0],[5,2],[7,1],[8,1],[9,0]]
```

Time complexity: sort the array takes $O(N \log N)$. When inserting each person, we may take upto $O(k)$ time to count and find an available slot, where k is the current number of empty or "shorter" slots in the array. Thus the total time complexity is $O(N^2)$.

32.3 Solution

32.3.1 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++) {
^^Icount = 0;
^^I// count # of people who has greater or equal height with people[i]
^^Ifor (int j = 0; j < i; j++)
^^I  if (people[j][0] >= people[i][0])
^^I    count++;

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

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

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

```
};
```

place person direcly into final position

```
class Solution {
public:
  // comparing functors used by sort()
  struct Comp {
    bool operator()(const vector<int>& a, const vector<int>& b) const {
      if (a[0] == b[0])
^^Ireturn a[1] < b[1];

      return a[0] < b[0];
    }
  };

  vector<vector<int>> reconstructQueue(vector<vector<int>>& people) {
    // sort people
    sort(people.begin(), people.end(), Comp());

    // create another array and initialize
    int size = people.size();
    vector<vector<int>> rp(size);
    for (int i = 0; i < size; i++) rp[i] = { -1, -1 };
    int count;
    int j;

    // put each people to right position
    for (int i = 0; i < size; i++) {
      for (j = 0, count = 0; count < people[i][1]; j++)  // count valid
       ↪  slots
^^Iif (rp[j][0] == -1 && rp[j][1] == -1 || rp[j][0] >= people[i][0])
^^I  count++;
      while(!(rp[j][0] == -1 && rp[j][1] == -1))  // move to next empty
       ↪  slot
^^Ij++;
      rp[j] = people[i];
    }

    return rp;
  }
};
```

32.4 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
  - ☐ read solution and work on that
- ☐ generalize this problem

33 412. Fizz Buzz

33.1 Problem Statement

Link

33.2 Analysis

33.2.1 Naive solution (check each condition and add string)

33.2.2 String concatenation

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

33.3 Solution

33.3.1 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) {
^^Iif (i % 5 == 0)
^^I  ret.push_back("FizzBuzz");
^^Ielse
^^I  ret.push_back("Fizz");
      }

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

      else
^^Iret.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++) {
^^Iif (i % check_num[j] == 0)
^^I  temp += add_term[j];
      }

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

    return ret;
  }
};
```

33.4 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

34 437. Path Sum III

34.1 Problem Statement

Link

34.2 Analysis

34.2.1 Double recursion (~50%, 50%)

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

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

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

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

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

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

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

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

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

Case 3 is when a path only contains the `root`. Pay attention that if `root->left` has a path sum to zero, and `root->val == sum`, then `left->root` and `root` are considered two different pathes.

If we think in a recursive way, this will account for those paths that from a node but not reaching leaf. The last node in the path is the node that `node->val is equal to passed in sum`.

34.3 Solution

34.3.1 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);

  }
};
```

34.4 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

35 442. Find All Duplicates in an Array

35.1 Problem Statement

Link

35.2 Analysis

35.2.1 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.

35.3 Solution

35.3.1 C++

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

```
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)
^^Iduplicate.push_back(i + 1);

    return duplicate;
  }
};
```

35.4 todos [/]

- ☐ think about the way to use original vector to hold labeling information

☐ read other solutions

☐ generalize the problem

36 448. Find All Numbers Disappeared in an Array

36.1 Problem Statement

Link

36.2 Analysis

36.2.1 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.

36.2.2 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.

36.3 Solution

36.3.1 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;

    // 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)
^^Idisappear.push_back(i + 1);

    return disappear;
```

```
  }
};
```

Use unordered-set (13%, 7%)

```
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)
^^Idisappear.push_back(i);
    }

    return disappear;
  }
};

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

36.4 todos [/]

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

37 461. Hamming Distance

37.1 Problem Statement

Link

37.2 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
```

37.3 Solution

37.3.1 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)
^^Iresult++;

      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)
^^Iresult++;
      x = x >> 1;
    }

    return result;
  }
};
```

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

- Bitwise XOR used.

37.3.2 Python

Faster than 97.37%

```
class Solution:
    def hammingDistance(self, x: int, y: int) -> int:
^^Iresult = 0
^^Iwhile x or y:
^^I    if x % 2 != y % 2:
^^I^^Iresult += 1
^^I    x = x >> 1
^^I    y = y >> 1
^^Ireturn result
```

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

38 476. Number Complemen

38.1 Problem Statement

Link

38.2 Analysis

38.2.1 bit manipulation (my solution)

The complement is the set negation of the number. But all leading zeros in the binary form is changed into one, you have to use a mask to get rid of them. Example:

```
 5: ...000000 101
~5: ...111111 010
 2: ...000000 010
```

We use a `mask` initialized as `~0`, then for each leading zero in `num`, we set bit in corresponding position in `mask` as 0. This process is repeated until we encounter the first non-zero bit in `num`, then the `mask` would be:

```
   5: ...000000 101
  ~5: ...111111 010
mask: ...000000 111
   2: ...000000 010
```

Then, we just simplily do a set intersection between `~num` and `mask`, to get the complement of `num`.

Time complexity: $O(1)$. The input is just one integer, at most 32 bit has to traverse. So the time complexity is constant.

38.3 Solution

38.3.1 C++

bit manipulation (my solution)

```
class Solution {
public:
  int findComplement(int num) {
    int mask = ~0;
    int bit = 1 << 31;

    while (!(num & bit)) {
      mask &= ~bit;
      bit >>= 1;
    }

    return (~num) & mask;
  }
};
```

38.4 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 analysis
- ☐ generalize this problem

39 477. Total Hamming Distance

39.1 Problem Statement

Link

39.2 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

39.2.1 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.

39.2.2 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.

39.3 Solution

39.3.1 C++

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

```
1  class Solution {
2  public:
3    int hammingDistance(const int& x, const int& y) {
4      int result = 0;
5      int a = x ^ y;
6
7      while (a != 0) {
8        if (a % 2)
9  ^^Iresult++;
10       a = a >> 1;
11     }
12
13     return result;
14   }
15
16   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)
^^Icount += 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) {
^^Iif (i % 2 == 0)
^^I  LSB_zeros.push_back(i);
^^Ielse
^^I  LSB_ones.push_back(i);

^^I// update i and non_zero_count
^^Ii = i >> 1;
^^Iif (i)
^^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++) {
^^I   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()`.

39.4 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

40 494. Target Sum

40.1 Problem Statement

Link

40.2 Analysis

40.2.1 Recursion, no bookkeeping

The first step is to find out the recursive relation. For each number in the array: `nums[i]`, we have two choices: give `+` sign to it, or give `-` sign to it. If we give `+` sign, then we have to search `S - nums[i]` in the remaining array. If we give `-` sign, then we have to search `S + nums[i]` in the remaining array. The total number of ways to achieve target sum is the sum of these two (giving `+` and `-`).

The base case is the last digit. We just need to compare if `nums[i] == S` or `nums[i] == -S`. If any of this two conditions is true, we found one way to sum to target (it could be two, when `S == -S -> S == 0`.

There will be redundant calculations in recursive calls. We'll deal with this in next method.

Time complexity (from solution): size of recursion tree will be 2^n, This is because for each number, we may assign it `+` or `-`, two choices. So, the total number of choices are 2^n.

Space complexity (from solution): $O(n)$, the depth of the recursion tree can go upto n.

40.2.2 Recursion, bookkeeping (hash table)

We use a hash table to hold calculated result. Specifically, we keep the number of ways to sum to target `S`, at an index of `start`. To use a hash table, we first determine the key. The key can be a `pair` of integers: `pair<int, int> p(start, S)`. `p.first` is the starting index, `p.second` is the target sum. We need to provide hash function that hashes a `pair` object to a `size_t` value:

```
struct hash_pair {
  size_t operator()(const pair<int, int>& p) const {
    return p.first ^ p.second;
  }
}
```

Then, the hash table is declared as (we are using an `unordered_map` to use key-value pair):

```
unordered_map<pair<int, int>, int, hash_pair> val;
```

Then, before calculating, we check if this value has been calculated or not. After calculating, we update the key-value pair into the hash table.

Time complexity: how do we know how many operations we performed? Sometimes we can return the `val[][]` directly, sometimes we have to call recursive function to calculate. We can think it in this way: we are filling the result record array `val[n][sum]`. `n` depends on the size of `nums` array, while `sum` depends on the range of possible `sum`.

40.2.3 Recursion, bookkeeping (2D array)

We can use a 2D array to hold pair `<start, sum>`. Although some `sum` generated during the calculation may be negative, the sum of elements in the given array will not exceed 1000 (this means even all numbers are selected `+` sign, the total sum will not exceed 1000). So, `sum + 1000` will not exceed 2000. For `sum` in range of `[-1000,1000]`, `sum + 1000` is in range of `[0,2000]`. So the index range would be `[0,2000]`, we use an array with 2001 slots to keep the record.

Before calculating any value of `<start, sum>`, we check if the value has been calculated or not. If so, we return it directly. Otherwise, we call recursive function to calculate, then update the value: `val[start][sum + 1000]`.

If based on the above recursion method, we have to check whether intermediate `sum` excessed the range `sum > 1000 || sum < -1000`, if so, return 0.

There is an other recursion method, which accumulates the sum in the process (instead of subtracting or adding in my method). After the final digit contributes to the sum, we compare if the accumulated sum is equal with target sum, if so, return 1 to count this particular path that sum to the target. If not, return 0 (my code needs to check if intermediate `sum` exceeds the range in the process, otherwise, index will fall out of range). For details, compare my recursion and solution's recursion.

40.2.4 Iteration, bookkeeping (2D array) + hashtable, backward (my)

Step-by-step example: `[1,1,1,1,1]` Normally, I can build up the solution based on recursion result (the 2D array bookkeeping one). The goal is to remove recursive call. To do so, we may take a closer look at **WHEN** do we make the recursive call: when the value of `val[start][sum]` is not calculated. This term represents the total number of ways to get `sum` starting at index `start`, the summation takes `nums[start]` into account. We can either assign `+` or `-` to to `nums[start]`. If we assign `+`, then we just need to find the total number of ways to sum to `sum - nums[start]` starting

from index `start+1`. If we assign `-`, then we just need to find the total number of ways to sum to `sum + nums[start]` starting from index `start+1`. That is, if `val[start+1][sum+nums[start]]` and `val[start+1][sum-nums[start]]` is known, we can calculate `val[start][sum]` directly:

```
val[start][sum] = val[start+1][sum+nums[start]] +
↪  val[start+1][sum-nums[start]]
```

This means we maybe able to calculate these two terms first, and calculate the rest terms iteratively: calculate from back and toward head.

Take `nums == {1,1,1,1,1}` as an example:

```
0 1 2 3 4
1 1 1 1 1
```

From `nums[4]`, we know:

```
val[4][1]  == 1  // when picking + for nums[4]
val[4][-1] == 1  // when picking - for nums[4]
```

this two spots are all we know. Now let `start == 3`:

```
val[3][sum] = val[4][sum+nums[3]] + val[4][sum-nums[3]]
->
val[3][sum] = val[4][sum+1] + val[4][sum-1]
```

since all non-zero term we know is `val[4][1]` and `val[4][-1]`, we can only use this two to make whatever possible non-zero term for `val[3][sum]`. The possible `sum` value is:

```
sum = 0:
  val[3][0] = val[4][1] + val[4][-1]
  val[3][0] =     1     +     1
            = 2

sum = 2:
  val[3][2] = val[4][3] + val[4][1]
  val[3][2] =     0     +     1
            = 1

sum = -2:
  val[3][-2] = val[4][-1] + val[4][-3]
  val[3][-2] =     1     +     0
             = 1
```

Now, we have more non-zero terms of `val[i][j]` to use:

```
val[4][1]  == 1
val[4][-1] == 1
val[3][0]  == 2
```

```
val[3][2]  == 1
val[3][-2] == 1
```

Now, let `start == 2`:

```
val[2][sum] = val[3][sum+nums[2]] + val[3][sum-nums[2]]
->
val[2][sum] = val[3][sum+1] + val[3][sum-1]
```

Similarliy, we want to find out all sum that can make either `val[3][sum+1]` or `val[3][sum-1]` non-zero, specifically:

```
sum+1 == 0, 2, -2 -> sum == -1, 1, -3
sum-1 == 0, 2, -2 -> sum == 1, 3, -1
```

The possible `sum` value is then: -3, -1, 1, 3. Corresponding `val[2][sum]` is:

```
sum == -3:
  val[2][-3] == val[3][-2] + val[3][-4]
             ==     1      +     0
             == 1

sum == -1:
  val[2][-1] == val[3][0] + val[3][-2]
             ==     2     +     1
             == 3

sum == 1:
  val[2][1] == val[3][2] + val[3][0]
             ==     1    +     2
             == 3

sum == 3:
  val[2][3] == val[3][4] + val[3][2]
             ==     0    +     1
             == 1
```

Now, we have following terms to use:

```
val[2][-3] == 1
val[2][-1] == 3
val[2][1]  == 3
val[2][3]  == 1
```

Let `start == 1`, then:

```
val[1][sum] = val[2][sum+nums[1]] + val[2][sum-nums[1]]
->
val[1][sum] = val[2][sum+1] + val[2][sum-1]
```

To find all possible `sum`:

```
sum + 1 == -3, -1, 1, 3 -> sum == -4, -2, 0, 2
sum - 1 == -3, -1, 1, 3 -> sum == -2, 0, 2, 4
```

So, the possible value of `sum` is -4, -2, 0, 2, 4. We have:

```
val[1][-4] == val[2][-3] + val[2][-5]
           == 1 + 0
           == 1
val[1][-2] == val[2][-1] + val[2][-3]
           == 3 + 1
           == 4
val[1][0]  == val[2][1] + val[2][-1]
           == 3 + 3
           == 6
val[1][2]  == val[2][3] + val[2][1]
           == 1 + 3
           == 4
val[1][4]  == val[2][5] + val[2][3]
           == 0 + 1
           == 1
```

Now, we have following terms to use:

```
val[1][-4] == 1
val[1][-2] == 4
val[1][0]  == 6
val[1][2]  == 4
val[1][4]  == 1
```

Finally, let `start == 0`, then:

```
val[0][sum] = val[1][sum+1] + val[1][sum-1]
```

Calculate the possible sum:

```
sum + 1 == -4, -2, 0, 2, 4 -> sum == -5, -3, -1, 1, 3
sum - 1 == -4, -2, 0, 2, 4 -> sum == -3, -1, 1, 3, 5
```

The possible `sum` is -5, -3, -1, 1, 3, 5. We have:

```
val[0][-5] == v[1][-4] + v[1][-6]
           == 1 + 0
           == 1
val[0][-3] == v[1][-2] + v[1][-4]
           == 4 + 1
           == 5
val[0][-1] == v[1][0] + v[1][-2]
           == 6 + 4
```

```
           == 10
val[0][1]  == v[1][2] + v[1][0]
           == 4 + 6
           == 10
val[0][3]  == v[1][4] + v[1][2]
           == 1 + 4
           == 5
val[0][5]  == v[1][6] + v[1][4]
           == 0 + 1
           == 1
```

Now, we have reached index 0, which means we have calculated all the possible `sum` from the starting of the array. Besides these `sum~s, for all other ~sum`, `val[0][sum] == 0`. These ~sum~s are:

```
val[0][-5] == 1
val[0][-3] == 5
val[0][-1] == 10
val[0][1]  == 10
val[0][3]  == 5
val[0][5]  == 1
```

Step-by-step example: `[1,999]` Following the code (except the 1000 + part). `size == 2`. After initialize step:

```
val[1][999] = 1
val[1][-999] = 1
old_sums: [999, -999]
```

Loop `i = size - 2 == 0` (the only loop). Find all possible ~new_sum~s:

```
old_sum == 999
  new_sum == 1000, 998
old_sum == -999
  new_sum == -998, -1000

new_sums: [1000, 998, -998, -1000]
```

The possible sum for index `i == 0` is: `[1000, 998, -998, -1000]`. Now go over each possible `new_sum` to calculate the number of ways to get to this sum:

```
val[0][1000] = val[1][1001] + val[1][999] == 1
val[0][998] = val[1][999] + val[1][997] == 1
val[0][-998] = val[1][-997] + val[1][-999] == 1
val[0][-1000] = val[1][-999] + val[1][-1001] == 1
```

Step-by-step example: `[1,1,0]` One of my algorithm does not work when trialing zero presents, I'm here to find out why. `size == 3`, after initialization:

```
val[2][0] == 1
old_sums: [0]
```

Loop `i == 1`, find all possible ~new_sum~s:

```
old_sum == 0:
  new_sum == 1, -1

new_sums: [1, -1]
```

Now, go over each possible `new_sum` to calculate the number of ways to get to this sum:

```
val[1][1] = val[2][2] + val[2][0] == 1
val[1][-1] = val[2][0] + val[2][-2] == 1
```

Now, loop `i == 0`, `old_sums: [1, -1]`, find all possible `new_sum`:

```
old_sum: 1
  new_sum == 2, 0
old_sum: -1
  new_sum == -2, 0

new_sums: [-2, 0, 2]
```

Now, go over each possible `new_sum` to calculate the number of ways to get to this sum:

```
val[0][-2] = val[1][-1] + val[0][-3] == 1
val[0][0] = val[1][1] + val[1][-1] == 2
val[0][2] = val[1][3] + val[1][1] == 1
```

Obviously, the result of this algorithm is not correct. Because we have an zero at the tail position. When initialization, I used:

```
 val[size-1][1000 + nums[size-1]] = 1;
 val[size-1][1000 - nums[size-1]] = 1;
```

So if `1000 + nums[size-1] == 1000 - nums[size-1]`, we'll only count one for this sum value (actually there are two ways to sum to this value: assigning + and -, this value is zero). To fix this, use `+=` instead of `=` (this works because every slot is initialized as zero).

Translate to algorithm First, we use a 2D array to hold calculated values that we can use to calculate other values. `val[i][s]` stores the number of ways to sum to `s`, starting at index `i` in `nums`. For `nums[i]`, you have two options:

1. assign `+` to `nums[i]`
2. assign `-` to `nums[i]`

If you choose 1, then `val[i][s]` is actually `val[i+1][s - nums[i]]`. Similarliy, if you choose 2, then `val[i][s]` is `val[i+1][s + nums[i]]`. So, the totle number of ways to sum to `s`, starting at index `i` is:

```
val[i+1][s - nums[i]] + val[i+1][s + nums[i]]
```

One thing should be pointed out is, there are finite number of `s` we can get from the sequence of numbers. For example, starting from the last number in `nums`, the sum we can get is:

- `nums[size-1]`: assigning `+` to the number
- `-nums[size-1]`: assigning `-` to the number

Generally, for an index `i`, there are finite number of `sum` that `val[i][sum] > 0` In the formula of `val[i][s]`, we must select `s` so that `val[i+1][s - nums[i]] > 0` or `val[i+1][s + nums[i]] > 0`. To determine all availabe `s`, we use the possible `sum` from number `nums[i+1]`. For each `old_sum`, we can have two `new_sum`:

1. `new_sum_1 = old_sum + nums[i]`
2. `new_sum_2 = old_sum - nums[i]`

After collecting all possible `new_sum`, we calculate `val[i][new_sum]` according to the given formula.

During implementation, we don't want to count duplicate `new_sum`, so we can use a hashtable to hold all the possible unique `new_sum`.

40.2.5 Iteration (from solution), forward iteration, bookkeeping (2D array)

The solution uses forward iteration. I found that many times, forward iteration is better than backward iteration, fewer boundary cases, conditions. Next time try to think both of them.

This approach uses similar idea. We use a 2D array to keep intermediate values. The difference is:

1. `val[i][j]` in my solution means the number of ways to sum to `j` **STARTING** from index `i`, these summing ways include the remaining elements in the array (after `nums[i]`).
2. `val[i][j]` in this solution means the number of ways to sum to `j` from `0` upto index `i`, these ways include the elements in front of `nums[i]`.

This difference makes this method iterate from beginning to end. To calculate the intermediate value:

```
val[i][old_sum - nums[i]] += val[i-1][old_sum];  // assign + to nums[i]
val[i][old_sum + nums[i]] += val[i-1][old_sum];  // assign - to nums[i]
```

The idea is, assume we know `val[i-1][j]`, then when we proceed to `nums[i]`, we can either assign $+$ or $-$ to `nums[i]`, to get new sum `j + nums[i]` and `j - nums[i]`. This two sums are calculated based on old sum `j`, so the number of ways to calculate to this two sums are `val[i-1][j]`.

40.2.6 Iteration, forward iteration, bookkeeping 1D array

After closer examining the above solution, you will see we only used the last calculated row of `val[][]`. This suggests we can use a 1D array to hold this last row and perform the calculation. Specifically, when we calculating the possible sum upto `i`,we need a 1D array storing the possible sum upto `i-1` (this 1D array is named `old_sums`), and another 1D array storing the possible sum upto `i` (which is the result of our calculation, this 1D array is named `new_sums`). After we calculated each `new_sums` and before we move to calculate the next, we update `old_sums`:

```
old_sums = new_sums;
```

And at beginning of each iteration, we have to reset `new_sums`:

```
new_sums.assign(2001, 0);  // make each element 0
```

This is to make sure `new_sums` only contains possible sum of the current number `i`.

40.3 Solution

40.3.1 C++

recursion, no bookkeeping

```
class Solution {
public:
  int find(vector<int>& nums, int start, long S) {
    // base case, last digit
    if (start == nums.size() - 1) {
      int ret = 0;
      if (nums[start] == S)
^^Iret++;
      if (nums[start] == -S)
^^Iret++;
      return ret;
    }

    // other cases, call recursive function
    return find(nums, start + 1, S + nums[start]) + find(nums, start + 1,
    ↪  S - nums[start]);
  }

  int findTargetSumWays(vector<int>& nums, int S) {
    return find(nums, 0, S);
  }
};
```

recursion, with bookkeeping (hash table)

```
class Solution {
public:
  // hash function for std::pair<int, int>
  struct hash_pair {
    size_t operator()(const pair<int, int>& p) const {
      return p.first ^ p.second;
    }
  };


  int findSum(vector<int>& nums, int start, int S,
  ↪  unordered_map<pair<int, int>, int, hash_pair>& val) {
    pair<int, int> p(start, S);
    // check if calculated
    if (val.find(p) != val.end())
      return val[p];

    // base case, last digit
    if (start == nums.size() - 1) {
      int ret = 0;
      if (nums[start] == S)
^^Iret++;
      if (nums[start] == -S)
^^Iret++;

      val[p] = ret;
      return ret;
    }

    // other cases, call recursive function
    int ret = findSum(nums, start + 1, S + nums[start], val) +
    ↪  findSum(nums, start + 1, S - nums[start], val);
    val[p] = ret;
    return ret;
  }

  int findTargetSumWays(vector<int>& nums, int S) {
    // sum of elements not exceeding 1000
    if (S > 1000 || S < -1000)
      return 0;

    // create a hash table to hold the calculated value
    unordered_map<pair<int, int>, int, hash_pair> val;
    return findSum(nums, 0, S, val);
  }
```

```
};
```

recursion, with bookkeeping (2D array)

```
/*Notes:

*/
class Solution {
public:
  int find(vector<int>& nums, int start, int S, int val[20][2001]) {
    // check if S exceeded range
    if (S > 1000 || S < -1000)
      return 0;

    // check if calculated
    if (val[start][S + 1000] != 1001)
      return val[start][S + 1000];

    // base case, last digit
    if (start == nums.size() - 1) {
      int ret = 0;
      if (nums[start] == S)
^^Iret++;
      if (nums[start] == -S)
^^Iret++;

      val[start][S + 1000] = ret;
      return ret;
    }

    // other cases, call recursive function
    int ret = find(nums, start + 1, S + nums[start], val) + find(nums,
    ↪  start + 1, S - nums[start], val);

    val[start][S + 1000] = ret;
    return ret;
  }

  int findTargetSumWays(vector<int>& nums, int S) {
    // check S size
    if (S > 1000 || S < -1000)
      return 0;

    // create a 2D array
    int val[20][2001];
    for (int i = 0; i < 20; i++)
```

```
      for (int j = 0; j < 2001; j++)
^^Ival[i][j] = 1001;

    return find(nums, 0, S, val);
  }
};
```

iteration, with bookkeeping (2D array) + hashtable (my)

```
class Solution {
public:
  int findTargetSumWays(vector<int>& nums, int S) {
    // empty
    if (nums.empty())
      return (S == 0) ? 1 : 0;

    // large sum
    if (S > 1000 || S < -1000)
      return 0;

    int size = nums.size();
    int new_sum;
    unordered_set<int> new_sums, old_sums;

    // value storage array
    int val[20][2001];
    for (int i = 0; i < 20; i++)
      for (int j = 0; j < 2001; j++)
^^Ival[i][j] = 0;

    // initialize
    val[size-1][1000 + nums[size-1]] += 1;
    val[size-1][1000 - nums[size-1]] += 1;
    old_sums.insert(nums[size-1]);
    old_sums.insert(-nums[size-1]);

    // calculate all possible sum
    for (int i = size - 2; i >= 0; --i) {
      new_sums.clear();
      // find all possible new_sum when assigning + and - to nums[i]
      for (auto old_sum : old_sums) {
^^Inew_sum = old_sum + nums[i];  // assigning +
^^Inew_sums.insert(new_sum);
^^Inew_sum = old_sum - nums[i];  // assigning -
^^Inew_sums.insert(new_sum);
      }
```

```
      // update val[i][new_sum] for all possible new_sum (starting at i)
      for (auto new_sum : new_sums) {
^^Iint old_sum_1 = new_sum + nums[i];
^^Iint old_sum_2 = new_sum - nums[i];

^^Iif (old_sum_1 >= -1000 && old_sum_1 <= 1000)
^^I  val[i][1000 + new_sum] += val[i+1][1000 + old_sum_1];
^^Iif (old_sum_2 >= -1000 && old_sum_2 <= 1000)
^^I  val[i][1000 + new_sum] += val[i+1][1000 + old_sum_2];

^^I// cout << "val[" << i << "][" << new_sum << "] == " << val[i][1000 +
 ↪  new_sum] << endl;
      }

      // update old sum
      old_sums = new_sums;

    }

    return val[0][1000 + S];

  }
};
```

iteration, with bookkeeping (2D array), forward (sol)

```
class Solution {
public:
  int findTargetSumWays(vector<int>& nums, int S) {
    // empty case
    int n = nums.size();
    if (n == 0)
      return (S == 0)? 1 : 0;

    // val[i][1000 + j] is the number of ways to sum to j, from 0 upto i
    int val[20][2001];
    for (int i = 0; i < 20; i++)
      for (int j = 0; j < 2001; j++)
^^Ival[i][j] = 0;

    // initialization, should use +=, rather than directly assignment
    val[0][1000+nums[0]] += 1;
    val[0][1000-nums[0]] += 1;
```

```
    for (int i = 0; i < n - 1; i++) {
      // find each old_sum and calculate new_sum
      for (int j = 0; j < 2001; j++)
^^Iif (val[i][j] != 0) { // old_sum
^^I  val[i+1][j + nums[i+1]] += val[i][j];
^^I  val[i+1][j - nums[i+1]] += val[i][j];
^^I}
    }

    return (S > 1000 || S < -1000) ? 0: val[n-1][1000+S];
  }
};
```

iteration, with bookkeeping (1D array), forward

```
class Solution {
public:
  int findTargetSumWays(vector<int>& nums, int S) {
    // empty case
    int n = nums.size();
    if (n == 0)
      return (S == 0)? 1 : 0;

    // val[i][1000 + j] is the number of ways to sum to j, from 0 upto i
    vector<int> old_sums(2001, 0);
    vector<int> new_sums(2001, 0);

    // initialization, should use +=, rather than directly assignment
    old_sums[1000+nums[0]] += 1;
    old_sums[1000-nums[0]] += 1;
    new_sums = old_sums;

    for (int i = 0; i < n - 1; i++) {
      new_sums.assign(2001, 0);

      // find each old_sum and calculate new_sum
      for (int j = 0; j < 2001; j++)
^^Iif (old_sums[j] != 0) { // old_sum
^^I  new_sums[j + nums[i+1]] += old_sums[j];
^^I  new_sums[j - nums[i+1]] += old_sums[j];
^^I}

      old_sums = new_sums;
    }

    return (S > 1000 || S < -1000) ? 0: new_sums[1000+S];
```

```
    }
};
```

40.4 todos [2/5]

- ☒ 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)
- ☐ read solutions to refine your code
- ☐ generalize this problem

41 543. Diameter of Binary Tree

41.1 Problem Statement

Link

41.2 Analysis

41.2.1 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`

41.3 Solution

41.3.1 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));
  }
};
```

41.4 todos [/]

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

42 557. Reverse Words in a String III

42.1 Problem Statement

Link

42.2 Analysis

42.2.1 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.

42.3 Solution

42.3.1 Python

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

```
class Solution:
    def reverseWords(s: str) -> str:
^^Iwords = s.split()
^^Ireversed_list = []
^^Ifor word in words:
^^I    reversed_list.append(''.join(list(reversed(word))) + ' ')
^^Ireversed_list[-1] = reversed_list[-1][:-1]

^^Ireturn ''.join(reversed_list)
```

42.4 todos [1/2]

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

43 559. Maximum Depth of N-ary Tree

43.1 Problem Statement

43.2 Analysis

43.2.1 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.

43.2.2 DFS

43.3 Solution

43.3.1 C++

recursion

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

    Node() {}

    Node(int _val, vector<Node*> _children) {
^^Ival = _val;
^^Ichildren = _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)
^^Ichild_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":[{"$id":"5","children":[],"v
↪  al":5},{"$id":"6","children":[],"val":6}],"val":3},{"$id":"3","childr
↪  en":[],"val":2},{"$id":"4","children":[],"val":4}],"val":1}

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

*/
```

43.4 todos [1/5]

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

44 581. Shortest Unsorted Continuous Subarray

44.1 Problem Statement

Link

44.2 Analysis

44.2.1 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.

44.3 Solution

44.3.1 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]) {
^^Istart_differ_index = i;
^^Ibreak;
      }
    }

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

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

```
};
```

44.4 todos [1/3]

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

45 617. Merge Two Binary Trees

45.1 Problem Statement

Link

45.2 Analysis

45.2.1 Recursive Method

Use recursiion to solve this problem.

45.2.2 Iterative Method (using stack)

45.3 Solution

45.3.1 C++

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

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

```
21          return node;
22        }
23      }
24    };
```

Iterative

45.4 todos [0/2]

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

46 647. Palindromic Substrings

46.1 Problem Statement

Link

46.2 Analysis

46.2.1 Brutal force (accepted, slow)

This approach enumerates all substrings and determine if they are palindrome or not, and count the number in the process.

Each single character is a palindrome, so we can start from length `i = 2`. For each palindrome with length `i`, we start from index `j = 0` to `n - i - 2`, where `n` is the total length of the original string. `n - i - 2` is the last starting index of substring with a length of `i`. For each substring, we iterate over the characters, using `k`. Namely, `s[k]` is each character in substring starting at `j` with length `i`, the range of the index of this substring is `[j,j + i - 1]`.

To determine if a substring is a palindrome, we check the if the first letter is the same as the last, and the second letter is the same as the second from the last, etc. So, we compare:

```
s[j]          s[j + i - 1]
s[j + 1]      s[j + i - 2]
s[j + 2]      s[j + i - 3]
...
```

We only need to compare to the half way of the list. For `k` in range of `[j, j+1, j+2, ..., j + i/2 - 1]`. And the compared index from the end is: `j + i - 1 - (k - j)`, which is: `2*j + i - k - 1`. If any two pairs not equal, this substring is not a palindrome, we break the loop. If the loop is not breaked during the iteration from `k == j` to `k == j + i/2 - 1`, the substring is a palindrome.

Time complexity is high, I ran several tests (of `"aaa...aa"` with varying length) and counted the operation number:

```
1: 0
2: 1
5: 13
10: 95
15: 308
20: 715
25: 1378
30: 2360
35: 3723
105: 97838
525: 12093003
```

This is higher than $O(N^2(\log N)^3)$, lower than $O(N^2(\log N)^4)$.

46.3 Solution

46.4 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
  - ☐ Expand Around Center
  - ☐ Manacher's Algorithm
  - ☐ Dynamic Programming
- ☐ generalize this problem

47 653. Two Sum IV - Input is a BST

47.1 Problem Statement

Link

47.2 Analysis

47.2.1 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.

47.3 Solution

47.4 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

48 657. Robot Return to Origin

48.1 Problem Statement

Link

48.2 Analysis

48.2.1 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.

48.3 Solution

48.3.1 Python

count instruction number

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

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

48.4 todos [1/2]

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

49 696. Count Binary Substrings

49.1 Problem Statement

Link

49.2 Analysis

49.2.1 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 `'00101000111100101'` 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.

49.2.2 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])`

49.2.3 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).

49.3 Solution

49.3.1 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:
^^Inum_first = []
^^Inum_second = []
^^Iresult = 0

^^Inum_first.append(s[0])

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

^^I    elif s[i] != num_first[0]:
^^I^^Inum_second.append(s[i])

^^I    elif s[i] == num_first[0]:  # another un-grouped num_first
↪  occured, we have to stop here
^^I^^Iresult += min(len(num_first), len(num_second))
^^I^^Inum_first = num_second[:]
^^I^^Inum_second.clear()
```

```
^^I^^Inum_second.append(s[i])

^^Iresult += min(len(num_first), len(num_second))

^^Ireturn result
```

count length of single number substring

```
class Solution:
    def countBinarySubstrings(self, s: str) -> int:
^^Icount = 1
^^Ilengths = []
^^Ians = 0

^^Ifor i in range(1, len(s)):
^^I    if s[i - 1] != s[i]:
^^I^^Ilengths.append(count)
^^I^^Icount = 1
^^I    else:
^^I^^Icount += 1

^^Ilengths.append(count)

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

^^Ireturn ans
```

count the total number on the fly

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

^^Ians += min(prev, cur)
```

```
^^Ireturn ans
```

49.4 todos [2/3]

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

50 739. Daily Temperature

50.1 Problem Statement

Link

50.2 Analysis

50.2.1 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 $O(N^2)$.

50.2.2 Record information (accepted, very slow)

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: $N \log N$.

50.2.3 SOL. Next array

This method is inspired by solution: next array. In the record information method, we recorded the appearing index of each temperature. However, after a closer look you'll find that it is not necessary to keep all that information (not to say we have to search the first larger index!). In fact, we can process the array in reverse order. We still have to keep the record of temperature and their appearing index, but since we are processing in a reverse order, we only need to keep the last appearing index from the end. When we are dealing with a temperature, all we care is the first appearing index after that temperature, so that's all we need to record. And we can build this record and analyze each slot in one run.

Don't need to use a hash table since each temperature `[30,100]` is unique, so we can use it directly.

50.3 Solution

50.3.1 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]) {
^^Icount = j - i;
^^Ibreak;
      }

    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]) {
^^Icount = temp[T[i] - 30][j] - i;
^^Ibreak;
      }
    ret[i] = count;
  }

  return ret;
}
```

next array

```
/*Notes:
Inspired by sol-next array
use array instead of hash table to store the next occurence of a
↪  temperature (faster?)
*/
class Solution {
public:
  vector<int> dailyTemperatures(vector<int>& T) {
    // initialize the temperature table
    int temp[71];
    for (int i = 0; i < 71; i++)
      temp[i] = -1;

    // iterate from the back
    int size = T.size();
    vector<int> ret(size);
    int count;
```

```
    for (int i = size - 1; i >= 0; --i) {
      count = size + 1;
      for (int j = T[i] + 1; j <= 100; j++)
^^Iif (temp[j - 30] != -1 && count > temp[j - 30] - i)
^^I  count = temp[j - 30] - i;

      ret[i] = (count == size + 1) ? 0 : count;
      temp[T[i] - 30] = i;
    }

    return ret;
  }
};
```

50.4 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

51 746. Min Cost Climbing Stairs

51.1 Problem Statement

Link.

Clarification: the top of the stairs is the one after the last index.

51.2 Analysis

51.2.1 Recursion (without bookkeeping)

We can build a recursive function that takes two arguments:

```
int count(int start, vector<int>& cost)
```

This function helper can give us the minimum cost to climb to top when starting at index `start`. The base case is the when `start == size - 1` or `start == size - 2`, in these cases, the minimum cost is `cost[start]`, since we can climb one or two steps. In other cases, the minimum cost is got by recursively calling the function:

```
min(count(start+1, cost), count(start+2, cost))
```

This method will do many redundant calculation.

51.2.2 Iteration (with bookkeeping, 1D array, backward bookkeeping)

Let `n` be the size of `cost[]`. We can use a vector of size `n` to hold min cost of each step, i,e, `val[i]` is the min cost to reach the top starting at index `i`. We use a backward iteration to calculate each term. At `n-1` step, `val[n-1] = cost[n-1]`. At `n-2` step, `val[n-2] = cost[n-2]` (because we can take two steps). For all other steps `i (i >= 0)`, the min cost can be calculated as the minimum of `cost[i] + val[i+1]` and `cost[i] + val[i+2]`.

This method prevents redundant caculation. Time complexity of iteration approach is $O(N)$.

51.2.3 Iteration (variable bookkeeping, backward iteration)

We can use three variables to hold the calculated intermediate value. Since the intermediate value contains the previous min cost (min cost is accumulative), we don't need 1D array to keep each intermediate result.

The iteration is backward.

51.2.4 Iteration (variable bookkeeping, forward iteration)

It is also possible to iterate from beginning to end of the `cost[]` array. Just need a little tweak. Code is shown in solution section.

51.3 Solution

51.3.1 C++

recursion

```
class Solution {
public:
  int count(int start, vector<int>& cost) {
    // base case: start is the last step
    if (start == cost.size() - 1 || start == cost.size() - 2)
      return cost[start];

    // normal case
    return min(count(start+1, cost), count(start+2, cost)) + cost[start];
  }

  int minCostClimbingStairs(vector<int>& cost) {
    return min(count(0, cost), count(1, cost));
  }
};
```

iteration (bookkeeping, 1D array, backward)

```
class Solution {
public:
  int minCostClimbingStairs(vector<int>& cost) {
    int size = cost.size();

    if (size == 2)
      return min(cost[0], cost[1]);

    vector<int> val(size);
    val[size-1] = cost[size-1];
    val[size-2] = cost[size-2];

    for (int i = size - 3; i >= 0; i--)
      val[i] = min(cost[i] + val[i+1], cost[i] + val[i+2]);

    return min(val[0], val[1]);

  }
};
```

iteration (bookkeeping, variables, backward)

```
class Solution {
public:
  int minCostClimbingStairs(vector<int>& cost) {
    int n = cost.size();
    if(n == 2)
      return min(cost[0], cost[1]);

    int a;
    int b = cost[n-2];
    int c = cost[n-1];

    for (int i = n - 3; i >= 0; i--) {
      a = min(cost[i] + b, cost[i] + c);
      if (i == 0)
^^Ibreak;
      c = b;
      b = a;
    }

    return min(a, b);
  }
```

```
};
```

iteration (bookkeeping, variable, forward)

```cpp
class Solution {
public:
  int minCostClimbingStairs(vector<int>& cost) {
    int a = cost[0];
    int b = cost[1];
    int c;

    for (int i = 2; i < cost.size(); i++) {
      c = cost[i] + min(a, b);
      a = b;
      b = c;
    }

    return min(a, b);
  }
};
```

51.4 todos [2/5]

- ☒ 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
- ☐ read solutions to refine your code
- ☐ generalize this problem

52 771. Jewels and Stones

52.1 Problem Statement

Link

52.2 Analysis

52.2.1 Brutal force

52.3 Solution

52.3.1 C++

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

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

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

    return false;
  }
};
```

52.4 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

53 804. Unique Morse Code Words

53.1 Problem Statement

Link

53.2 Analysis

53.2.1 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.

53.3 Solution

53.3.1 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)
^^Itranslate += 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"]

*/
```

53.4 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

54 824. Goat Latin

54.1 Problem Statement

Link

54.2 Analysis

54.2.1 Modify the word directly

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

54.3 Solution

54.3.1 Python

modify the word directly

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

^^Ivowel = ('a', 'e', 'i', 'o', 'u')

^^Ifor i in range(len(words)):

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

^^I    for j in range(i + 1):
^^I^^Iwords[i] += 'a'

^^I    words[i] += ' '

^^Iwords[-1] = words[-1][:-1]

^^Ireturn ''.join(words)
```

54.4 todos [1/6]

- [x] 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

55 876. Middle of the Linked List

55.1 Problem Statement

Link

55.2 Analysis

55.2.1 Use an array to hold each node

We iterate the linked list and keep each node in a vector, then we return the middle one directly. Time complexity: $O(N)$, space complexity: $O(N)$.

55.2.2 Iterate twice

We iterate the linked list to get the size, then we iterate again to get the middle node. Time complexity: $O(N)$, space complexity: $O(1)$.

55.2.3 Fast and slow pointer (from solution)

55.3 Solution

55.3.1 C++

use array to hold each node

```cpp
class Solution {
public:
  ListNode* middleNode(ListNode* head) {
    if (head == nullptr)
      return nullptr;

    vector<ListNode*> list;
    while (head != nullptr) {
      list.push_back(head);
      head = head->next;
    }

    return list[list.size() / 2];
  }
};
```

iterate twice

```cpp
class Solution {
public:
  ListNode* middleNode(ListNode* head) {
    if (head == nullptr)
      return head;

    // get size of linked list
    int size = 0;
    ListNode* temp = head;
    while (temp != nullptr) {
      size++;
      temp = temp->next;
    }

    // get middle node
    temp = head;
    int middle = size / 2;
    while (middle-- != 0)
      temp = temp->next;

    return temp;
  }
};
```

55.4 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
  - ☐ fast and slow pointer
- ☐ generalize this problem

56 929. Unique Email Addresses

56.1 Problem Statement

56.2 Analysis

56.2.1 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

56.3 Solution

56.3.1 Python

use string member functions

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

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

^^Ireturn len(unique_emails)
```

56.4 todos [1/2]

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

57 938. Range Sum of BST

57.1 Problem Statement

Link

57.2 Analysis

57.2.1 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.

57.2.2 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.

57.3 Solution

57.3.1 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) {
^^Iif (nodes.top()->val >= L && nodes.top()->val <= R) {
^^I  sum += nodes.top()->val;
^^I  nodes.pop();
^^I  continue;
^^I}
      }
```

```
      // 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()) {
^^Ivisited_nodes.insert(nodes.top()->left);  // mark as visited
^^Inodes.push(nodes.top()->left);
^^Icontinue;
      }

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

      // 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)
^^Isum += nodes.top()->val;

      nodes.pop();
    }

    return sum;

  }
};
```

57.4 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

58 1021. Remove Outermost Parenthese

58.1 Problem Statement

Link

58.2 Analysis

58.2.1 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.

58.2.2 Two pointers

58.3 Solution

58.3.1 C++

Stack

```
class Solution {
public:
```

```
  string removeOuterParentheses(string S) {
    stack<char> ch_stack;
    string result;
    string temp;

    for (char ch : S) {
      if (ch == '(') {
^^Ich_stack.push(ch);
^^Itemp += ch;
^^Icontinue;
      }

      if (ch == ')' && ch_stack.size() == 1) {
^^Ich_stack.pop();

^^I// record temp to result, not including the first '('
^^Ifor (int i = 1; i < temp.size(); i++)
^^I  result += temp[i];

^^I// clear temp cache
^^Itemp.clear();
^^Icontinue;
      }

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

    }

    return result;
  }
};
```

58.4 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

59 1108. Defanging an IP Address

59.1 Problem Statement

Link

59.2 Analysis

59.2.1 Direct replace

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

59.3 Solution

59.3.1 Python

direct replace

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

59.4 todos [1/2]

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