001 two_sum（easy）
011 Container With Most Water（easy）
015 3Sum(medium)
016 3Sum Closest(medium)
026 删除排序数组中的重复项(easy)
027 移除元素(easy)
031 Next Permutation(medium)
033 搜索旋转排序数组(medium)
034 在排序数组中查找元素的第一个和最后一个位置(medium)
035 搜索插入位置(easy)
039 组合总和(medium)
040 组合总和 II(medium)
048 旋转图像(medium)
053 最大子序和(easy)
054 螺旋矩阵(middle)
066 加1(easy)
073 矩阵置零(middle)
075 颜色分类(middle)
078 子集(middle)
080 删除排序数组中的重复项 II(middle)
081 搜索旋转排序数组 II(middle)
088 合并两个有序数组（easy）
090 子集 II(middle)
118 杨辉三角 (easy)
119 杨辉三角 II(easy)
120 三角形最小路径和 (middle)
121 买卖股票的最佳时机 (easy)
122 买卖股票的最佳时机 II
153 寻找旋转排序数组中的最小值 (middle)
154 寻找旋转排序数组中的最小值 II(hard)
162 寻找峰值(midle)
167 两数之和 II - 输入有序数组(easy)
169 多数元素(easy)
189 旋转数组(easy)

## 001 two_sum ( easy )

- **题目**
  Given an array of integers, return indices of the two numbers such that they add up to a specific target.
  You may assume that each input would have exactly one solution, and you may not use the same element twice.
  Example:
  Given nums = [2, 7, 11, 15], target = 9,
  Because nums[0] + nums[1] = 2 + 7 = 9,
  return [0, 1].
- **思路**
  遍历数组，每读到一个值，判断target - nums[i]在哈希表里是否存在：不存在，则把当前的值和i存到哈希表；存在，则返回结果。
- **基础知识**
  创建的哈希表无序的即可，所以用ordered_map，否则用map。
- **代码实现**

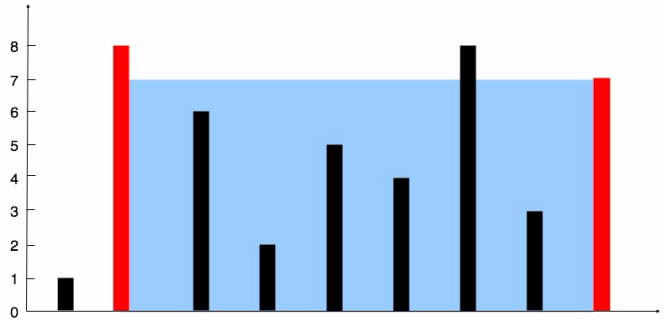
```
class Solution {
public:
    vector<int> twoSum(vector<int>& nums, int target) {
        unordered_map<int, int> numIndexMap;
        for (int i = nums.size()- 1; i >=0; --i) {
            if(numIndexMap.find(target - nums[i]) !=
numIndexMap.end())
                return {i, numIndexMap[target - nums[i]]};
            numIndexMap[nums[i]] = i;
        }
        return {};
    }
};
```

## 011 Container With Most Water ( easy )

- **题目**
  Given n non-negative integers a1, a2, ..., an , where each represents a point at

coordinate (i, ai). n vertical lines are drawn such that the two endpoints of line i is at (i, ai) and (i, 0). Find two lines, which together with x-axis forms a container, such that the container contains the most water.

**Note**: You may not slant the container and n is at least 2.




The above vertical lines are represented by array [1,8,6,2,5,4,8,3,7]. In this case, the max area of water (blue section) the container can contain is 49.

Example:

Input: [1,8,6,2,5,4,8,3,7]

Output: 49

- **思路**

  要使水的容积最大，分别从两侧往中间遍历，保存最大容积res

- **代码实现**

```
class Solution {
public:
    int maxArea(vector<int>& height) {
        int front = 0, back = height.size()-1,res = 0;

        while(front < back){
            int min_h = min(height[front],height[back]);
            res = max(res ,(back - front)*min_h);
            while(front < back && min_h >= height[front])
                front++;
            while(front <back && min_h >= height[back])
                back--;
        }

        return res;
    }
};
```

## 015 3Sum(medium)

- **题目**

  Given an array nums of n integers, are there elements a, b, c in nums such that a + b + c = 0? Find all unique triplets in the array which gives the sum of zero.

  **Note**:

  The solution set must not contain duplicate triplets.

  **Example**:

```
Given array nums = [-1, 0, 1, 2, -1, -4],
A solution set is:
[
  [-1, 0, 1],
  [-1, -1, 2]
]
```

- **思路**

  思路一：每次找出一个数，另外两个数用哈希表的方法找出另外两个数，此方法重复比较多，而且空间复杂度高

  思路二：首先对数组进行快速排序。从头开始遍历数组，如果当前元素nums[i]和q前一个元素nums[i-1]相同，则跳过，这样得到第一个元素nums[i]，接着在 i+1 到数组末尾筛选出另外两个数（原理是由最小的数加最大的数和第一个元素作比较，来决定最小的数前移或最大的数后移），如果筛选出来后还得继续筛选和去重复的值。

- **代码实现**

```
class Solution {
public:
    vector<vector<int>> threeSum(vector<int>& nums) {
        vector<vector <int> > res;

        sort(nums.begin(),nums.end());
        if(nums.empty() || nums.back() < 0 || nums.front() > 0)
            return {};

        for(int i = 0, len = nums.size() - 2; i < len; i++){
            if(nums[i] > 0)
```

```
                break;
            if(i > 0 && nums[i] == nums[i-1])
                continue;

            for(int j = i+1,k = len + 1,target = -nums[i]; j < k ;){
                if(nums[j]+nums[k] > target)
                    --k;
                else if(nums[j]+nums[k] < target)
                    ++j;
                else{
                    res.push_back({nums[i],nums[j],nums[k]});
                    while(j < k && nums[j] == nums[j+1])
                        ++j;
                    while(j < k && nums[k] == nums[k-1])
                        --k;
                    ++j;
                    --k;
                }
            }

        }
        return res;

    }
};
```

## 016 3Sum Closest(medium)

- **题目**
  Given an array nums of n integers and an integer target, find three integers in nums such that the sum is closest to target. Return the sum of the three integers. You may assume that each input would have exactly one solution.
  **Example**:
  Given array nums = [-1, 2, 1, -4], and target = 1.
  The sum that is closest to the target is 2. (-1 + 2 + 1 = 2).
- **思路**
  首先对数组进行快速排序，接着遍历数组0到len-2个数据，当前nums[i]作为第一个值，接着在端点j=i+1和端点k =len-1寻找最接近的两个数，计算出当前两个端

点值的和还有与target的差，比较之前的值保存差最小的。若nums[i] +nums[j] +nums[k] >target,则 --k；若nums[i] +nums[j] +nums[k] > target，则j++；否则说明当前三个数的和等于target，直接返回结果，程序提前结束。

- **代码实现**

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

        int close_sum = nums[0] + nums[1] +nums[2];
        int diff = abs(target - close_sum);

        for(int i = 0,len = nums.size()-2; i < len ; i++){
            for(int j = i+1,k =len +1; j < k;){
                int new_sum = nums[i] + nums[j] + nums[k];
                int new_diff = abs(target - new_sum);
                if( new_diff < diff){
                    close_sum = new_sum;
                    diff = new_diff;
                }
                if(new_sum > target)
                    --k;
                else if(new_sum < target)
                    ++j;
                else
                    return new_sum;
            }
        }

        return close_sum;

    }
};
```

## 026 删除排序数组中的重复项(easy)

- **题目**

给定一个排序数组，你需要在 原地 删除重复出现的元素，使得每个元素只出现一次，返回移除后数组的新长度。
不要使用额外的数组空间，你必须在 原地 修改输入数组 并在使用 O(1) 额外空间的条件下完成（不需要考虑数组中超出新长度后面的元素）。

- **代码实现**

```
class Solution {
public:
    int removeDuplicates(vector<int>& nums) {
        if (nums.empty())
            return 0;
        int newIndex = 0;
        for (int i = 1; i < nums.size(); ++i) {
            if (nums[i] != nums[newIndex])
                nums[++newIndex] = nums[i];
        }
        return newIndex+1;
    }
};
```

## 027 移除元素(easy)

- **题目**
给你一个数组 nums 和一个值 val，你需要 原地 移除所有数值等于 val 的元素，并返回移除后数组的新长度。
不要使用额外的数组空间，你必须仅使用 O(1) 额外空间并原地修改输入数组。
元素的顺序可以改变。不需要考虑数组中超出新长度后面的元素。

- **代码实现**

```
class Solution {
public:
    int removeElement(vector<int>& nums, int val) {
        int newIndex = 0;
        for(int i = 0; i < nums.size(); ++i) {
            if(nums[i] != val)
                nums[newIndex++] = nums[i];
        }
```

```
        return newIndex;
    }
};
```

## 031 Next Permutation(medium)

- **题目**
  实现获取下一个排列的函数，算法需要将给定数字序列重新排列成字典序中下一个更大的排列。
  如果不存在下一个更大的排列，则将数字重新排列成最小的排列（即升序排列）。
  必须原地修改，只允许使用额外常数空间。
  以下是一些例子，输入位于左侧列，其相应输出位于右侧列。

```
1,2,3 → 1,3,2
3,2,1 → 1,2,3
1,1,5 → 1,5,1
```

- **思路**

  这道题让我们求下一个排列顺序，由题目中给的例子可以看出来，如果给定数组是降序，则说明是全排列的最后一种情况，则下一个排列就是最初始情况，可以参见之前的博客 Permutations。我们再来看下面一个例子，有如下的一个数组

  1 2 7 4 3 1

  下一个排列为：

  1 3 1 2 4 7

  那么是如何得到的呢，我们通过观察原数组可以发现，如果从末尾往前看，数字逐渐变大，到了2时才减小的，然后我们再从后往前找第一个比2大的数字，是3，那么我们交换2和3，再把此时3后面的所有数字转置一下即可，步骤如下：

  1 2 7 4 3 1

  1 2 7 4 3 1

  1 3 7 4 2 1

  1 3 1 2 4 7

- **代码实现**

```
class Solution {
public:
    void nextPermutation(vector<int>& nums) {
```

```
        int len = nums.size();
        if(len < 2)
            return ;
        int p_min = len -2;
        while(p_min >= 0 && nums[p_min] >= nums[p_min + 1])
            --p_min;

        if(p_min >= 0){
            int p_miner = len -1;
            while(nums[p_miner] <= nums[p_min])
                --p_miner;

            nums[p_miner] = nums[p_miner] - nums[p_min];
            nums[p_min] += nums[p_miner];
            nums[p_miner] = nums[p_min] - nums[p_miner];
        }
        while(++p_min < --len){        //reverse(nums.begin() + p_min + 1, nums.end());
            nums[len] = nums[len] - nums[p_min];
            nums[p_min] += nums[len];
            nums[len] = nums[p_min] - nums[len];
        }

    }
};
```

## 033 搜索旋转排序数组(medium)

- **题目**
  设按照升序排序的数组在预先未知的某个点上进行了旋转。
  ( 例如，数组 [0,1,2,4,5,6,7] 可能变为 [4,5,6,7,0,1,2] )。
  搜索一个给定的目标值，如果数组中存在这个目标值，则返回它的索引，否则返回 -1 。
  你可以假设数组中不存在重复的元素。
  你的算法时间复杂度必须是 O(log n) 级别。

  > 示例 1:
  > 输入: nums = [4,5,6,7,0,1,2], target = 0
  > 输出: 4

> 示例 2:
> 输入: nums = [4,5,6,7,0,1,2], target = 3
> 输出: -1

- **思路**
  类似二分搜索的方法
  如果target在左边段，nums[mid]在右边段，则later = mid - 1；
  如果target在右边段，nums[mid]在左边段，则former = mid +1；
  其他情况下，nums[mid] < targent ,former =mid+1; nums[mid] > targent ,later =mid -1

- **代码实现**

```
class Solution {
public:
    int search(vector<int>& nums, int target) {
        int former = 0,later = nums.size() - 1;
        while(former <= later){
            int mid = (former + later)>>1;
            if(nums[mid] == target) return mid;
            if(target < nums[0] && nums[mid] >=nums[0])
                 former = mid+1;
            else if(target >= nums[0] && nums[mid] < nums[0])
                later = mid -1;
            else{
                if(target < nums[mid] )
                    later = mid -1;
                else
                    former = mid +1;
            }
        }
        return -1;
    }
};
```

## 034 在排序数组中查找元素的第一个和最后一个位置(medium)

- **题目**
  给定一个按照升序排列的整数数组 nums，和一个目标值 target。找出给定目标

值在数组中的开始位置和结束位置。
你的算法时间复杂度必须是 O(log n) 级别。
如果数组中不存在目标值，返回 [-1, -1]。

> 示例 1:
> 输入: nums = [5,7,7,8,8,10], target = 8
> 输出: [3,4]
> 示例 2:
> 输入: nums = [5,7,7,8,8,10], target = 6
> 输出: [-1,-1]

- **思路**
  使用两个二分查找，分别查找出目标值的左边界和右边界。二分查找中要想**满足low<high成为判断退出的条件**，至少需要 执行low = mid +1才可以实现离开循环（high 可以等于 mid 或者 mid - 1）。

> 分析为什么至少需要low = mid +1，因为low和high最后的大部分情况都是相邻，即（mid =（low + high）>>1 ）== low，如果low的值不变，那么将会变成死循环

- **代码实现**

```
class Solution {
public:
 vector<int> searchRange(vector<int>& nums, int target) {
  int low = 0, high = nums.size() - 1;
  while (low < high){
   int mid = (low + high) >> 1;
   if (nums[mid] < target)
    low = mid + 1;  //至少需要 low = mid +1才可以实现离开循环!
   else
    high = mid;
  }
  if (high == -1 || nums[low] != target)
   return {-1,-1};
  vector <int> res(2);
  res[0] = low,high = nums.size(); //考虑到后半段全为目标值的情况，所以把high开始定义为nums.size()
  while (low < high){
   int mid = (low + high) >> 1;
```

```
    if (nums[mid] <= target)
     low = mid + 1; //至少需要 low = mid +1才可以实现离开循环!
    else
     high = mid;
   }
   res[1] = high - 1;
   return res;
  }
};
```

## 035 搜索插入位置(easy)

- **题目**
  给定一个排序数组和一个目标值，在数组中找到目标值，并返回其索引。如果目标值不存在于数组中，返回它将会被按顺序插入的位置。
  你可以假设数组中无重复元素。

  > 示例 1:
  > 输入: [1,3,5,6], 5
  > 输出: 2:

- **思路**
  二分查找的边界考察
- **代码实现**

```
class Solution {
public:
 vector<vector<int>> combinationSum(vector<int>& nums, int target) {
  sort(nums.begin(), nums.end());
  if (nums.size() == 0 ||nums[0] > target)
   return{};
  vector<vector<int>> res;
  vector<int> one_res;
  SumDfs(nums, target, 0, res, one_res);
  return res;
 }

 void SumDfs(vector<int>& nums, int target, int start,
vector<vector<int>> &res, vector<int> &one_res){
```

```
  for (int i = start; i < nums.size(); ++i){
   if (nums[i] > target)
    break;
   else if (nums[i] < target){
    one_res.push_back(nums[i]);
    SumDfs(nums, target - nums[i], i, res, one_res);
    one_res.pop_back();
   }
   else{
    one_res.push_back(nums[i]);
    res.push_back(one_res);
    one_res.pop_back();
    break;
   }
  }
 }
};
```

## 039 组合总和(medium)

- **题目**
  给定一个无重复元素的数组 candidates 和一个目标数 target ，找出 candidates 中所有可以使数字和为 target 的组合。
  candidates 中的数字可以无限制重复被选取。
  说明：
  所有数字（包括 target）都是正整数。
  解集不能包含重复的组合。

  > 示例 1:
  > 输入: candidates = [2,3,6,7], target = 7,
  > 所求解集为:
  > [
  > [7],
  > [2,2,3]
  > ]

- **思路**
  对于输出所有符合要求的解，肯定是通过递归来实现。这类题思路大同小异

首先需要写一个递归函数，加入三个变量，第一个start保存当前递归到的下标，one_res为一个解，res为全部解。
这类题开始都先通过快排，优化程序判断次数。

- **代码实现**

```
class Solution {
public:
 vector<vector<int>> combinationSum(vector<int>& nums, int target) {
  sort(nums.begin(), nums.end());
  if (nums.size() == 0 ||nums[0] > target)
   return{};
  vector<vector<int>> res;
  vector<int> one_res;
  SumDfs(nums, target, 0, res, one_res);
  return res;
 }

 void SumDfs(vector<int>& nums, int target, int start,
vector<vector<int>> &res, vector<int> &one_res){
  for (int i = start; i < nums.size(); ++i){
   if (nums[i] > target)
    break;
   else if (nums[i] < target){
    one_res.push_back(nums[i]);
    SumDfs(nums, target - nums[i], i, res, one_res);
    one_res.pop_back();
   }
   else{
    one_res.push_back(nums[i]);
    res.push_back(one_res);
    one_res.pop_back();
    break;
   }
  }
 }
};
```

## 040 组合总和 II(medium)

- **题目**
  给定一个数组 candidates 和一个目标数 target ，找出 candidates 中所有可以使数字和为 target 的组合。
  candidates 中的每个数字在每个组合中只能使用一次。
  说明：
  所有数字（包括目标数）都是正整数。
  解集不能包含重复的组合。

  > 示例 1:
  > 输入: candidates = [10,1,2,7,6,1,5], target = 8,
  > 所求解集为:
  > [
  > [1, 7],
  > [1, 2, 5],
  > [2, 6],
  > [1, 1, 6]
  > ]

- **思路**
  与上一体的实现很思路一致。此题的区别在于每个数字在组合中只能使用一次。
  修改的内容：递归的for循环里排除重复的值；递归的时候 start + 1

- **代码实现**

```
class Solution {
public:
 vector<vector<int>> combinationSum2(vector<int>& nums, int target) {
  sort(nums.begin(), nums.end());
  if (nums.size() == 0 ||nums[0] > target) return{};
  vector<vector<int>> res;
  vector<int> one_res;
  SumDfs(nums, target, 0, res, one_res);
  return res;
 }

 void SumDfs(vector<int>& nums, int target, int start,
vector<vector<int>> &res, vector<int> &one_res){
  for (int i = start; i < nums.size(); ++i){
   if(i > start && nums[i] == nums[i - 1]) continue;
```

```
            if (nums[i] > target) break;
    else if (nums[i] < target){
     one_res.push_back(nums[i]);
     SumDfs(nums, target - nums[i], i + 1, res, one_res);
     one_res.pop_back();
    }
    else{
     one_res.push_back(nums[i]);
     res.push_back(one_res);
     one_res.pop_back();
     break;
    }
   }
  }
};
```

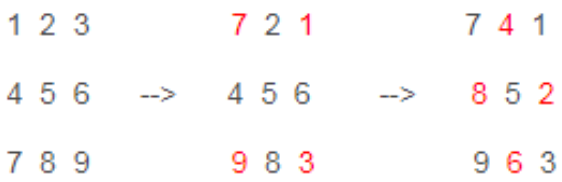
## 048 旋转图像(medium)

- **题目**
  给定一个 n × n 的二维矩阵表示一个图像。
  将图像顺时针旋转 90 度。
  说明：
  你必须在原地旋转图像，这意味着你需要直接修改输入的二维矩阵。请不要使用另一个矩阵来旋转图像。

  > 示例 1:
  > 给定 matrix =
  > [
  > [1,2,3],
  > [4,5,6],
  > [7,8,9]
  > ],
  > 原地旋转输入矩阵，使其变为:
  > [
  > [7,4,1],
  > [8,5,2],
  > [9,6,3]
  > ]

- **思路**

  做该题，得出一个对矩阵的表示方法（从特殊到普遍）。

  > a. 首先去四个角的值（0,0），（0,2），（2,2），（2,0）
  >
  > b. 特殊推断普遍的过程：（**0**,0）->（0,**2**）等价于（x1, y1）->(x2, y2)
  > 选取值1 和 2 的时候，x1与y2的值是保持不变的，x2 等于 y1
  > 即（i , j）->（j , len - 1 - i），遍历中 i 的值是不变的，++j
  > 其他三个同理可推得。

> 要抓住变换过程中，什么值是不变的，什么值是变的，变的关系是什么。

- **代码实现**

```
class Solution {
public:
    void rotate(vector<vector<int>>& matrix) {
        if(matrix.size() <= 1) return;
        for(int len = matrix.size(), i = 0; i < len / 2; ++i){
            for(int j = i; j < len - 1 - i; ++j){
                int temp = matrix[i][j];
                matrix[i][j] = matrix[len - 1 - j][i];
                matrix[len - 1 - j][i] = matrix[len - 1 - i][len - 1 - j];
                matrix[len - 1 - i][len - 1 - j] = matrix[j][len - 1 - i];
                matrix[j][len - 1 - i] = temp;
            }
        }
    }
};
```

## 053 最大子序和(easy)

- **题目**

  给定一个整数数组 nums ，找到一个具有最大和的连续子数组（子数组最少包含

一个元素），返回其最大和。
示例:

> 输入: [-2,1,-3,4,-1,2,1,-5,4],
> 输出: 6
> 解释: 连续子数组 [4,-1,2,1] 的和最大，为 6。进阶:如果你已经实现复杂度为 O(n) 的解法，尝试使用更为精妙的分治法求解。

- **思路**
  常规方法，用current_sum保存临时序列和，res_sum保持当前最大序列和，遍历数组的过程中，例如遍历到i，若当前current_sum大于0，则current_sum += nums[i]，否则current_sum = nums[i]。再把res_sum保存为current_sum 和 res_sum中较大的值。
  分治方法：二分查找的思路，采用递归 。之前排过序的数组查找的复杂度是很低，但这个复杂度觉得大于常规方法了。不推荐。
- **代码实现**

```
class Solution {
public:
    int maxSubArray(vector<int>& nums) {
        int resSum = INT_MIN, curSum = 0;
        for (int i = 0; i < nums.size(); ++i) {
            curSum = curSum > 0 ? curSum + nums[i] : nums[i];

            if (resSum < curSum)
                resSum = curSum;
        }
        return resSum;
    }
};
```

## 054 螺旋矩阵(middle)

- **题目**
  给定一个包含 m x n 个元素的矩阵（m 行, n 列），请按照顺时针螺旋顺序，返回矩阵中的所有元素。

> 示例 1:

> 输入:
> [
> [ 1, 2, 3 ],
> [ 4, 5, 6 ],
> [ 7, 8, 9 ]
> ]
> 输出: [1,2,3,6,9,8,7,4,5]

- **思路**
  这道题调试了很久，能发现自身对矩阵遍历比较弱。总结下**矩阵遍历的要点方法**：
  1、首先是需要遍历的次数，通过矩阵的长宽中的较小值除以 2 来计算
  2、每一轮遍历，对角线两个点的坐标，这个决定了每一轮的遍历区间
  3、最后一轮遍历有两种情况，剩下一排或者两排得判断清楚。目前用的方式是，通过矩阵的长和宽减去遍历次数 * 2，即为剩下的排数，当排位为1时即是单排，为2时即为双排。
- **代码实现**

```
class Solution {
public:
 vector<int> spiralOrder(vector<vector<int>>& matrix) {
  if (matrix.empty() || matrix[0].empty())
   return{};
  vector<int> res;
  int len_x = matrix[0].size() - 1, len_y = matrix.size() - 1;
  int cycle = len_x < len_y ? len_x / 2 : len_y / 2;
  int len_x1 = len_x, len_y1 = len_y;
  for (int i = 0; i <= cycle; --len_x, --len_y, ++i){
   int temp = i;
   while (temp <= len_x)
    res.push_back(matrix[i][temp++]);
   temp = i + 1;
   while (temp <= len_y)
    res.push_back(matrix[temp++][len_x]);
   if (i == cycle && (len_x1 - 2 * i == 0 || len_y1- 2 * i == 0))
    break;
   temp = len_x - 1;
   while (temp >= i)
```

```
    res.push_back(matrix[len_y][temp--]);
   temp = len_y - 1;
   while (temp > i)
    res.push_back(matrix[temp--][i]);
  }
  return res;
 }
};
```

## 066 加1(easy)

- **题目**
  给定一个由整数组成的非空数组所表示的非负整数，在该数的基础上加一。最高位数字存放在数组的首位， 数组中每个元素只存储单个数字。你可以假设除了整数 0 之外，这个整数不会以零开头。

  > 示例 1:
  > 输入: [1,2,3]
  > 输出: [1,2,4]

- **思路**
  循环中只需判断当前是否为9，为9进位赋为0进行下一位判断，不为0则加1返回结果。

- **代码实现**

```
class Solution {
public:
    vector<int> plusOne(vector<int>& digits) {
        for (int i = digits.size() - 1; i >= 0; --i) {
            if (digits[i] == 9)
                digits[i] = 0;
            else {
                ++digits[i];
                return digits;
            }
        }
        if (digits.front() == 0)
            digits.insert(digits.begin(), 1);
        return digits;
```

```
    }
};
```

## 073 矩阵置零(middle)

- **题目**
  给定一个 m x n 的矩阵，如果一个元素为 0，则将其所在行和列的所有元素都设为 0。请使用原地算法。

  > 示例 1:
  > 输入:
  > [
  > [1,1,1],
  > [1,0,1],
  > [1,1,1]
  > ]
  > 输出:
  > [
  > [1,0,1],
  > [0,0,0],
  > [1,0,1]
  > ]
  > 进阶:
  > 一个直接的解决方案是使用 O(mn) 的额外空间，但这并不是一个好的解决方案。
  > 一个简单的改进方案是使用 O(m + n) 的额外空间，但这仍然不是最好的解决方案。
  > 你能想出一个常数空间的解决方案吗？

- **思路**
  1、O(mn) 的额外空间，申请一个同等大小的矩阵matrix_copy，先扫原数组的值，第一轮把有0的行赋为0，第二轮把有0的列赋为0，最后把matrix_copy中为0的全部赋值到原数组。
  2、申请长度为m和n的数组，分别保存行和列中0的标志位。如果第 i行有 0，则把行数组的第i个值赋为true。遍历原数组，对行列数组进行更新。根据更新后的行列数组更新元数组为0的行和列。
  3、不额外申请内存，把原数组的第0行和第0列来保存标志位。首先遍历 0行和0

列是否有0，若有则把对应的flag设为true，接着从1行1列开始遍历原数组，若有0，则把0行和0列对应的值改为0。再遍历原数组，比如martix[i][j]，若对应在0行或0列的值为0时，则martix[i][j] = 0，更新整个原数组；最后根据0行和0列的flag更新0行0列的值。

- **代码实现**

```
class Solution {
public:
    void setZeroes(vector<vector<int>>& matrix) {
        if(matrix.empty() || matrix.empty()) return;
        bool flag_zero_x = false, flag_zero_y = false;
        int len_x = matrix[0].size(), len_y = matrix.size();
        for(int i = 0; i < len_x; ++i)
            if(matrix[0][i] == 0){
                flag_zero_x = true;
                break;
            }
        for(int i = 0; i < len_y; ++i)
            if(matrix[i][0] == 0){
                flag_zero_y = true;
                break;
            }
        for(int i = 1; i < len_y; ++i)
            for(int j = 1; j < len_x; ++j){
                if(matrix[i][j] == 0){
                    matrix[i][0] = 0;
                    matrix[0][j] = 0;
                }
            }
        for(int i = 1; i < len_y; ++i)
            for(int j = 1; j < len_x; ++j){
                if(matrix[i][0] == 0 || matrix[0][j] == 0)
                    matrix[i][j] = 0;
            }
        if(flag_zero_x)
            for(int i = 0; i < len_x; ++i) matrix[0][i] = 0;
        if(flag_zero_y)
             for(int i = 0; i < len_y; ++i) matrix[i][0] = 0;
    }
```

```
};
```

## 075 颜色分类(middle)

- **题目**
  给定一个包含红色、白色和蓝色，一共 n 个元素的数组，原地对它们进行排序，使得相同颜色的元素相邻，并按照红色、白色、蓝色顺序排列。
  此题中，我们使用整数 0、 1 和 2 分别表示红色、白色和蓝色。
  注意:
  不能使用代码库中的排序函数来解决这道题。

  > 示例:
  > 输入: [2,0,2,1,1,0]
  > 输出: [0,0,1,1,2,2]
  > 进阶：
  > 一个直观的解决方案是使用计数排序的两趟扫描算法。
  > 首先，迭代计算出0、1 和 2 元素的个数，然后按照0、1、2的排序，重写当前数组。
  > 你能想出一个仅使用常数空间的一趟扫描算法吗？

- **思路**
  使用两个指针，分别保存头和尾指针，遍历数组，遇到0则和头指针的值交换，头指针后移一位；遇到2与尾指针的值交换，尾指针前移一位，继续判断当前的值，因为交换过来的值可能为0或2。

- **代码实现**

```
class Solution {
public:
    void sortColors(vector<int>& nums) {
        for(int i = 0,red = 0,blue = nums.size() - 1; i <= blue; ++i)
{
            if(nums[i] == 0)
                swap(nums[red++], nums[i]);
            else if(nums[i] == 2)
                swap(nums[i--], nums[blue--]);
        }
    }
};
```

## 078 子集(middle)

- **题目**
  给定一组不含重复元素的整数数组 nums，返回该数组所有可能的子集（幂集）。
  说明：解集不能包含重复的子集。

  > 示例:
  > 输入: nums = [1,2,3]
  > 输出:
  > [
  > [3],
  > [1],
  > [2],
  > [1,2,3],
  > [1,3],
  > [2,3],
  > [1,2],
  > []
  > ]

- **思路**
  这个题型挺好的。
  讲解一下查找的思路：比如有五个数，{1， 2， 3， 4， 5}
  1、{}
  2、{} ... {1}
  3、{}、{1}... {2}、{1，2}
  4、{}、{1}、{2}、{1，2}...{3}、{1,3}、{2,3}、{1,2,3}
  每一轮都在之前的基础上多几个子集

- **代码实现**

```
class Solution {
public:
    vector<vector<int>> subsets(vector<int>& nums) {
        vector <vector<int>> res(1);
        sort(nums.begin(), nums.end());
```

```
        for(int i = 0; i < nums.size(); ++i){
            for(int j = 0, Size = res.size(); j < Size; ++j){
                res.push_back(res[j]);
                res.back().push_back(nums[i]);
            }
        }
        return res;
    }
};
```

## 080 删除排序数组中的重复项 II(middle)

- **题目**
  给定一个排序数组，你需要在原地删除重复出现的元素，使得每个元素最多出现两次，返回移除后数组的新长度。
  不要使用额外的数组空间，你必须在原地修改输入数组并在使用 O(1) 额外空间的条件下完成。

  > 示例 1:
  > 给定 nums = [1,1,1,2,2,3],
  > 函数应返回新长度 length = 5, 并且原数组的前五个元素被修改为 1, 1, 2, 2, 3 。
  > 你不需要考虑数组中超出新长度后面的元素。

- **思路**
  用count保存当前数字出现的次数。
- **代码实现**

```
class Solution {
public:
    int removeDuplicates(vector<int>& nums) {
        if(nums.size() < 3)
            return nums.size();
        int cur_p = 0, count = 1;
        for(int i = 1; i < nums.size(); ++i){
            if(nums[i] == nums[i - 1] && count < 2){
                ++count;
                nums[++cur_p] = nums[i];
```

```
            }
            else if(nums[i] != nums[i - 1]){
                count = 1;
                nums[++cur_p] = nums[i];
            }
        }
        return cur_p + 1;
    }
};
```

## 081 搜索旋转排序数组 II(middle)

- **题目**
  假设按照升序排序的数组在预先未知的某个点上进行了旋转。
  ( 例如，数组 [0,0,1,2,2,5,6] 可能变为 [2,5,6,0,0,1,2] )。
  编写一个函数来判断给定的目标值是否存在于数组中。若存在返回 true，否则返回 false。

  > 示例 1:
  > 输入: nums = [2,5,6,0,0,1,2], target = 0
  > 输出: true

- **思路**
  **重点题**
  这个题得明确思路，还是挺重要的，因为每次对判断的思路都不够清晰。
  如果 nums[mid] < nums[high]时，若target > nums[mid] && target <= nums[high], 那么 low = mid + 1（target在最右侧有序区域）,其他情况下high = mid - 1
  如果nums[mid] > nums[high]时，若target < nums[mid] && target >= nums[low], 那么 high = mid - 1（target在最左侧有序区域）,其他情况都是low = mid + 1；
  如果nums[mid] == nums[high]，那么 high--。（为了解决最左侧和最右侧有重复数字的情况，因为判断不出mid是属于左侧区域还是右侧区域）

  > 我们可以观察出规律，如果中间的数小于最右边的数，则右半段是有序的，若中间数大于最右边数，则左半段是有序的，我们只要在有序的半段里用首尾两个数组来判断目标值是否在这一区域内，这样就可以确定保留哪半边了

- **代码实现**

```
```

```cpp
class Solution {
public:
    bool search(vector<int>& nums, int target) {
        if(nums.size() < 1)
            return false;
        int low = 0, high = nums.size() - 1;
        while(low <= high){
            int mid = (low + high) >> 1;
            if(nums[mid] == target)
                return true;
            if(nums[mid] < nums[high]){
                if(target <= nums[high] && target > nums[mid])
 //target位于最右侧有序区域
                    low = mid + 1;
                else
                    high = mid - 1;
            }
            else if(nums[mid] > nums[high]){
                if(target >= nums[low] && target < nums[mid])
//target位于最左侧有序区域
                    high = mid - 1;
                else
                    low = mid + 1;
            }
            else
                --high;
        }
        return false;
    }
};
```

## 088 合并两个有序数组（easy）

- **题目**
  给你两个有序整数数组 nums1 和 nums2，请你将 nums2 合并到 nums1 中，使 nums1 成为一个有序数组。
  说明:
  初始化 nums1 和 nums2 的元素数量分别为 m 和 n 。你可以假设 nums1 有足够的空间（空间大小大于或等于 m + n）来保存 nums2 中的元素。

- **代码实现**

```
class Solution {
public:
    void merge(vector<int>& nums1, int m, vector<int>& nums2, int n)
{
        int curIndex = m + n - 1;
        --m, --n;
        while (m >= 0 && n >= 0)
            nums1[curIndex--] = nums1[m] > nums2[n] ? nums1[m--] :
nums2[n--];
        while (n >= 0)
            nums1[curIndex--] = nums2[n--];
    }
};
```

## 090 子集 II(middle)

- **题目**
  给定一个可能包含重复元素的整数数组 nums，返回该数组所有可能的子集（幂集）。
  说明：解集不能包含重复的子集。

  > 示例:
  > 输入: [1,2,2]
  > 输出:
  > [
  > [2],
  > [1],
  > [1,2,2],
  > [2,2],
  > [1,2],
  > []
  > ]

- **思路**
  拿题目中的例子 [1 2 2] 来分析，根据之前 Subsets 里的分析可知，当处理到第一个2时，此时的子集合为 [], [1], [2], [1, 2]，而这时再处理第二个2时，如果在 []

和 [1] 后直接加2会产生重复，所以只能在上一个循环生成的后两个子集合后面加2，发现了这一点，题目就可以做了，我们用 last 来记录上一个处理的数字，然后判定当前的数字和上面的是否相同，若不同，则循环还是从0到当前子集的个数，若相同，则**新子集个数减去之前循环时子集的个数**当做**起点**来循环，这样就不会产生重复了。

[]

[1]

[2]、[1 2]

[2 2]、[1 2 2]

每一轮都在之前的基础上多几个子集

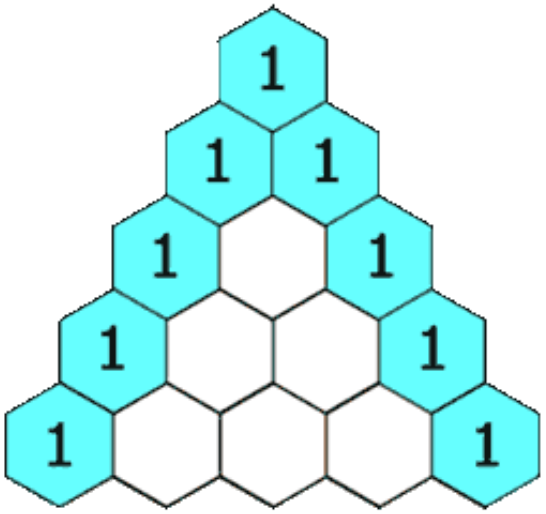
- **代码实现**

```
class Solution {
public:
    vector<vector<int>> subsetsWithDup(vector<int>& nums) {
        if(nums.empty())
            return {};
        sort(nums.begin(), nums.end());
        vector <vector<int>> res(1);
        int last = nums[0];
        for(int i = 0, size = 1; i < nums.size(); ++i){
            if(last != nums[i]){
                last = nums[i];
                size = res.size();
            }
            int Size = res.size();
            for(int j = Size - size; j < Size; ++j){
                res.push_back(res[j]);
                res.back().push_back(nums[i]);
            }
        }
        return res;
    }
};
```

## 118 杨辉三角 (easy)

- **题目**

给定一个非负整数 numRows，生成杨辉三角的前 numRows 行。




> 输入: 5
> 输出:
> [
> [1],
> [1,1],
> [1,2,1],
> [1,3,3,1],
> [1,4,6,4,1]
> ]

- **思路**

  主要在于二维vector的大小设置。

  ...

- **代码实现**

```
class Solution {
public:
    vector<vector<int>> generate(int numRows) {
        vector<vector<int>> res(numRows, vector<int>());
        for (int i = 0; i < numRows; ++i) {
            res[i].resize(i + 1, 1);
            for (int j = 1; j < i; ++j) {
                res[i][j] = res[i - 1][j - 1] + res[i - 1][j];
            }
        }
        return res;
```

```
    }
};
```

## 119 杨辉三角 II(easy)

- **题目**
  给定一个非负索引 k，其中 k ≤ 33，返回杨辉三角的第 k 行。
  在杨辉三角中，每个数是它左上方和右上方的数的和。

  > 示例:
  > 输入: 3
  > 输出: [1,3,3,1]

- **思路**
  通过两个for循环，用一维数组把之前的每一行结果都计算出来
  ...

- **代码实现**

```
class Solution {
public:
    vector<int> getRow(int rowIndex) {
        vector<int> resRow(rowIndex + 1, 1);
        for (int i = 1; i < rowIndex; ++i)
            for (int j = i; j > 0; --j)
                resRow[j] += resRow[j-1];
        return resRow;
    }
};
```

## 120 三角形最小路径和 (middle)

- **题目**
  给定一个三角形，找出自顶向下的最小路径和。每一步只能移动到下一行中相邻的结点上。

  > 例如，给定三角形：
  > [
  > [2],

> [3,4],
> [6,5,7],
> [4,1,8,3]
> ]
> 自顶向下的最小路径和为 11（即，2 + 3 + 5 + 1 = 11）。

- **思路**
  个人第一思路是新建一个dp，长度为nums.size() - 1。从第一层往下遍历保存每一层的和。最后再找出dp中最小的值。
  看到博客上的想法**短而精悍**。直接容器dp复制nums的最后一行，从下往上遍历，最后dp[0] 即为最小的和。
  ...
- **代码实现**

```
class Solution {
public:
    int minimumTotal(vector<vector<int>>& nums) {
        int high = nums.size() - 1;
        vector <int> dp(nums.back());
        for(int i = dp.size() - 2; i >= 0; --i)
            for(int j = 0; j <= i; ++j){
                dp[j] = min(dp[j], dp[j + 1]) + nums[i][j];
            }
        return dp[0];
    }
};
```

## 121 买卖股票的最佳时机 (easy)

- **题目**
  给定一个数组，它的第 i 个元素是一支给定股票第 i 天的价格。如果你最多只允许完成一笔交易（即买入和卖出一支股票一次），设计一个算法来计算你所能获取的最大利润。
  注意：你不能在买入股票前卖出股票。
- **代码实现**

```
class Solution {
```

```
public:
    int maxProfit(vector<int>& prices) {
        int tempMaxProfit = 0;
        for (int i = 0, buyPrice = INT_MAX; i < prices.size(); ++i) {
            buyPrice = prices[i] < buyPrice ? prices[i] : buyPrice; // 取当前最低买入价格
            if (tempMaxProfit < prices[i] - buyPrice)
                tempMaxProfit = prices[i] - buyPrice;
        }
        return tempMaxProfit;
    }
};
```

## 122 买卖股票的最佳时机 II

给定一个数组，它的第 i 个元素是一支给定股票第 i 天的价格。设计一个算法来计算你所能获取的最大利润。你可以尽可能地完成更多的交易（多次买卖一支股票）。
注意：你不能同时参与多笔交易（你必须在再次购买前出售掉之前的股票）。

- **代码实现**

```
    int maxProfit(vector<int>& prices) {
        int tempMaxProfit = 0;
        for (int i = 1; i < prices.size(); ++i) {
            if (prices[i] > prices[i-1])
                tempMaxProfit += prices[i] - prices[i-1];
        }
        return tempMaxProfit;
    }
```

## 153 寻找旋转排序数组中的最小值 (middle)

- **题目**
  假设按照升序排序的数组在预先未知的某个点上进行了旋转。
  ( 例如，数组 [0,1,2,4,5,6,7] 可能变为 [4,5,6,7,0,1,2] )。
  请找出其中最小的元素。
  你可以假设数组中不存在重复元素。

> 示例 1:
> 输入: [3,4,5,1,2]
> 输出: 1

- **思路**
  新学到一个知识点，首先判断该数组是否进行了旋转
  接着二分查找，搜到只剩两个数
  ...
- **代码实现**

```
class Solution {
public:
    int findMin(vector<int>& nums) {
        if(nums.empty())
            return -1;
        int low = 0,high = nums.size() - 1;
        if(nums[low] > nums[high]){  //判断是否进行了旋转
            while(low != high -1){ //搜到只剩两个数
                int mid = (low + high) >> 1;
                if(nums[mid] >= nums[high])
                    low = mid;
                else
                    high = mid;
             }
             return nums[low] < nums[high]? nums[low]:nums[high]; //返回最后两个数中较小的那个数
        }
        return nums[0];
    }
};
```

## 154 寻找旋转排序数组中的最小值 II(hard)

- **题目**
  假设按照升序排序的数组在预先未知的某个点上进行了旋转。
  ( 例如，数组 [0,1,2,4,5,6,7] 可能变为 [4,5,6,7,0,1,2] )。
  请找出其中最小的元素。
  注意数组中可能存在**重复**的元素。

> 示例 1：
> 输入: [1,3,5]
> 输出: 1

- **思路**
  **重点题**
  对于有重复的旋转数组关键问题：数组首和尾存在相同数组，那么mid的值同时和首尾相同时很难判断mid是处与左半段还是右半段。
  比如：[3,3,1,3]，mid的值等于3，就不知道应该是low = mid 还是 high = mid。
  通过high- - 解决末尾有重复的问题
  ...
- **代码实现**

```
class Solution {
public:
    int findMin(vector<int>& nums) {
        if(nums.empty())
            return -1;
        int low = 0,high = nums.size() - 1;
        if(nums.size() == 1)
            return nums[0];
        if(nums[low] >= nums[high]){ //判断是否进行了旋转
            while(low != high -1){ //搜到只剩两个数
                int mid = (low + high) >> 1;
                if(nums[mid] > nums[high])
                    low = mid;
                else if(nums[mid] < nums[high])
                    high = mid;
                else
                    high--;  //解决首尾有相同值的问题
             }
             return nums[low] < nums[high]? nums[low]:nums[high]; //返回最后两个数中较小的那个数
        }
        return nums[0];
    }
};
```

## 162 寻找峰值(midle)

- **题目**
  峰值元素是指其值大于左右相邻值的元素。
  给定一个输入数组 nums，其中 nums[i] ≠ nums[i+1]，找到峰值元素并返回其索引。
  数组可能包含多个峰值，在这种情况下，返回任何一个峰值所在位置即可。
  你可以假设 nums[-1] = nums[n] = -∞。

  > 示例 1:
  > 输入: nums = [1,2,3,1]
  > 输出: 2
  > 解释: 3 是峰值元素，你的函数应该返回其索引 2。

- **思路**
  常规O（n）复杂度的方法，即遍历数组寻找答案。
  log N的解法为二分查找，重点在于找到那个极值，low和high都赋值为值较大的那个数。
  ...

- **代码实现**

```
class Solution {
public:
    int findPeakElement(vector<int>& nums) {

        int low = 0,high = nums.size() - 1;
        while(low < high){
            int mid = (low + high) >> 1;
            if(nums[mid] < nums[mid + 1])  //选择mid 和 mid + 1的原因是不会造成越界的问题。
                low = mid + 1;   //mid + 1的值比较大，low保存为mid + 1。这里用low = mid + 1而不是high = mid + 1的原因是为了能跳出循环，low必须得后移，high = mid + 1某些情况下会进入死循环状态。
            else
                high = mid; // mid 值较大
        }
        return low;
    }
};
```

## 167 两数之和 II - 输入有序数组(easy)

- **题目**
  给定一个已按照升序排列 的有序数组，找到两个数使得它们相加之和等于目标数。
  函数应该返回这两个下标值 index1 和 index2，其中 index1 必须小于 index2。
  说明:
  返回的下标值（index1 和 index2）不是从零开始的。
  你可以假设每个输入只对应唯一的答案，而且你不可以重复使用相同的元素。

  > 示例:
  > 输入: numbers = [2, 7, 11, 15], target = 9
  > 输出: [1,2]
  > 解释: 2 与 7 之和等于目标数 9 。因此 index1 = 1, index2 = 2

- **思路**
  第一种：用map来做，空间复杂度过高。
  第二种：因为原数组是有序的，所以用两个指针分别指向头和尾来寻找。

- **代码实现**

```
    vector<int> twoSum(vector<int>& numbers, int target) {
        int left = 0, right = numbers.size()-1;
        while (left < right) {
            if (numbers[left] + numbers[right] == target)
                break;
            numbers[left] + numbers[right] < target ? ++left : --
right;
        }
        return {left + 1, right + 1};
    }
```

## 169 多数元素(easy)

- **题目**
  给定一个大小为 n 的数组，找到其中的多数元素。多数元素是指在数组中出现次数大于 ⌊ n/2 ⌋ 的元素。你可以假设数组是非空的，并且给定的数组总是存在多数

元素。

- **代码实现**

```
    int majorityElement(vector<int>& nums) {
        int solider = nums[0], blood = 1;
        for (int i = 1; i < nums.size(); ++i) {
            if (blood == 0) {
                solider = nums[i];
                blood = 1;
            } else {
                solider == nums[i] ? ++blood : --blood;
            }
        }
        return solider;
    }
```

## 189 旋转数组(easy)

- **题目**
  给定一个数组，将数组中的元素向右移动 k 个位置，其中 k 是非负数。

  > 示例:
  > 输入: [1,2,3,4,5,6,7] 和 k = 3
  > 输出: [5,6,7,1,2,3,4]
  > 解释:
  > 向右旋转 1 步: [7,1,2,3,4,5,6]
  > 向右旋转 2 步: [6,7,1,2,3,4,5]
  > 向右旋转 3 步: [5,6,7,1,2,3,4]
  > 尽可能想出更多的解决方案，至少有三种不同的方法可以解决这个问题。
  > 要求使用空间复杂度为 O(1) 的 原地 算法。

- **思路**
  第一种：复制一个容器，再进行平移更新，空间复杂度过高。
  第二种：数组前n - k个元素翻转，后k个元素翻转，整个数组翻转。

- **代码实现**

```
class Solution {
```

```
public:
    void rotate(vector<int>& nums, int k) {
        if (nums.empty() || (k %= nums.size()) == 0)
            return;
        int numsLength = nums.size();
        reverse(nums.begin(), nums.begin() + numsLength - k);    // reverse函数的区间为[first, end)
        reverse(nums.begin() + numsLength - k, nums.end());
        reverse(nums.begin(), nums.end());
    }
};
```