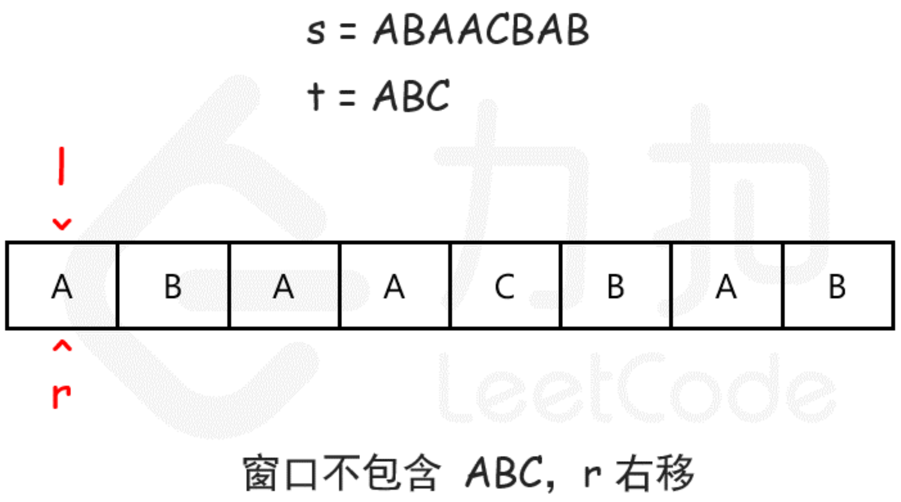
LeetCode 刷题记录

滑动窗口问题

核心思想: 我们可以用滑动窗口的思想解决这个问题，在滑动窗口类型的问题中都会有两个指针。一个用于「延伸」现有窗口的 r 指针，和一个用于「收缩」窗口的 l 指针。在任意时刻，只有一个指针运动，而另一个保持静止。我们在 ss上滑动窗口，通过移动 r 指针不断扩张窗口。当窗口包含 t 全部所需的字符后，如果能收缩，我们就收缩窗口直到得到最小窗口。



```
// 基本框架
int left = 0, right = 0;
while (right < s.size()) {
    // 增大窗口
    window.add(s[right]);
    right++;

    while (window needs shrink) {
        // 缩小窗口
        window.remove(s[left]);
        left++;
    }
}
```

3. 无重复字符的最长子串 #todo

```
 // 维护了一个滑动窗口，窗口内的都是没有重复的字符，需要尽可能的扩大窗口的大小。由于窗口在不停向右滑动，所以只关心每个字符最后出现的位置，并建立映射。窗口的右边界就是当前遍历到的字符的位置，为了求出窗口的大小，需要一个变量 left 来指向滑动窗口的左边界，这样，如果当前遍历到的字符从未出现过，那么直接扩大右边界，如果之前出现过，那么就分两种情况，在或不在滑动窗口内，如果不在滑动窗口内，那么就没事，当前字符可以加进来，如果在的话，就需要先在滑动窗口内去掉这个已经出现过的字符了，去掉的方法并不需要将左边界 left 一位一位向右遍历查找，由于 HashMap 已经保存了该重复字符最后出现的位置，所以直接移动 left 指针就可以了。维护一个结果 res，每次用出现过的窗口大小来更新结果 res，就可以得到最终结果
int lengthOfLongestSubstring(string s)
{
    int res = 0, n = s.size();
    //窗口的右边界就是当前遍历到的字符的位置，为了求出窗口的大小，需要一个变量 left 来指向滑动窗口的左边界
    int left = -1; // left 指向该无重复子串左边的起始位置的前一个
    //如果当前遍历到的字符从未出现过，那么直接扩大右边界，如果之前出现过，那么就分两种情况，在或不在滑动窗口内，如果不在滑动窗口内，那么就没事，当前字符可以加进来，如果在的话，就需要先在滑动窗口内去掉这个已经出现过的字符了，去掉的方法并不需要将左边界 left 一位一位向右遍历查找，由于 HashMap 已经保存了该重复字符最后出现的位置，所以直接移动 left 指针就可以了
    unordered_map<int, int> m;
    for (int i = 0; i < n; ++i)
    {
        //两个条件 m.count(s[i]) && m[s[i]] > left，因为一旦当前字符 s[i] 在 HashMap 已经存在映射，说明当前的字符已经出现过了，而若 m[s[i]] > left 成立，说明之前出现过的字符在窗口内，那么如果要加上当前这个重复的字符，就要移除之前的那个，所以让 left 赋值为 m[s[i]]，由于 left 是窗口左边界的前一个位置
        if (m.count(s[i]) && m[s[i]] > left)
        {
            left = m[s[i]];
        }
        m[s[i]] = i;
        res = max(res, i - left);
    }
    return res;
}
```

参考 https://www.cnblogs.com/grandyang/p/4480780.html

53. Maximum Subarray

```
class Solution {
public:
    int maxSubArray(vector<int>& nums)
    {
        if (nums.empty())
            return 0;
        int res = INT_MIN;
        int n = nums.size();
        int cur = 0;
        for(int i = 0; i < n; i++)
        {
            cur += nums[i];
            res = max(cur, res);
            cur = cur < 0 ? 0 : cur;
        }
        return res;
    }
};
```

76. 最小覆盖子串 # todo 20210419

```
string minWindow(string s, string t)
{
    vector<int> m(128,0);  // m 可以理解 需要多少个 如m[a] = -1,说明多了一个a, m[a] = 0,正好,
m[a]=1说明缺一个a
    // 记录t中每个字符出现的次数
    for (char c : t)
        m[c]++;
    int count = 0;
    int left = 0;
    int min_len = INT_MAX; // 记录窗口的最小值,
    int min_left = -1; // 记录窗口的最小值对应的左边界
    for(int i = 0; i < s.size(); i++)
    {
        //减1后的映射值仍大于等于0, 说明当前遍历到的字母是T串中的字母
        if (--m[s[i]] >= 0)
            count ++;
        while (count == t.size()) // 形成窗口了, 并且当前窗口包含t中的所有字符
        {
            if (i - left + 1 < min_len)
            {
                min_len = i - left + 1;
                min_left = left;
            }
            // 开始收缩左边界
            if (++m[s[left]] > 0)
            {
                count--;
            }
            ++left;
        }
    }
    return  min_left == -1 ? "" : s.substr(min_left, min_len);
}
```

239. 滑动窗口最大值

```
vector<int> maxSlidingWindow(vector<int> &nums, int k)
{
    if (nums.empty() || nums.size() < k || k < 1)
    {
        vector<int> res;
        return res;
    }
    deque<int> q;
    vector<int> res;
    // 核心是保持队列单调有序即可
    for (int i = 0; i < nums.size(); i++)
    {
        while (!q.empty() && nums[i] >= nums[q.back()])
            q.pop_back();
        q.push_back(i);  // 入队列相当于窗口多了一个数

        if (q.front() <= i - k) // 检查队首元素是否过期，如果过期则弹出
            q.pop_front();

        if (i >= k - 1)  // 开始形成窗口
            res.push_back(nums[q.front()]);
    }
    return res;
}
```

424. 替换后的最长重复字符 #todo 20210419

见https://www.cnblogs.com/grandyang/p/5999050.html

```
/*
如果没有k的限制，让我们求把字符串变成只有一个字符重复的字符串需要的最小置换次数，那么就是字符串的总长度减去出现次数最多的字符的个数。如果加上k的限制，我们其实就是求满足 (子字符串的长度减去出现次数最多的字符个数)<=k 的最大子字符串长度即可，搞清了这一点，我们也就应该知道怎么用滑动窗口来解了吧。我们用一个变量 start 记录滑动窗口左边界，初始化为0，然后遍历字符串，每次累加出现字  符的个数，然后更新出现最多字符的个数，然后我们判断当前滑动窗口是否满足之前说的那个条件，如果不满足，我们就把滑动窗口左边界向右移动一个，并注意去掉的字符要在 counts 里减一，直到满足条件，我们更新结果 res 即可。需要注意的是，当滑动窗口的左边界向右移动了后，窗口内的相同字母的最大数貌似可能会改变啊，为啥这里不用更新 maxCnt 呢？这是个好问题，原因是此题让求的是最长的重复子串，maxCnt 相当于卡了一个窗口大小，我们并不希望窗口变小，虽然窗口在滑动，但是之前是出现过跟窗口大小相同的符合题意的子串，缩小窗口没有意义，并不会使结果 res 变大，所以我们才不更新 maxCnt 的
*/

// 解法一
int characterReplacement(string s, int k)
{
    int res = 0, maxCnt = 0, left = 0;
    vector<int> m(128,0); // 用来记录窗口中每个字符出现的次数
    for (int i = 0; i < s.size(); i++)
    {
        maxCnt = max(maxCnt, ++m[s[i]]);
        // 判断当前窗口 left...i 是否满足条件
        if (i - left + 1 - maxCnt > k)  // 不满足 从左开始收缩窗口
        {
            --m[s[left]];
            left ++;
        }
        res = max(res, i - left + 1);
    }
    return res;
}

// 解法二
int characterReplacement(string s, int k)
{
    int res = 0, maxCnt = 0;
    vector<int> counts(26, 0);
    int right = 0;
    int left = 0;
    while(right < s.size())
    {
        maxCnt = max(maxCnt, ++counts[s[right] - 'A']);
        while (right - left + 1 - maxCnt > k) // 缩减窗口直到不满足条件为止
        {
            --counts[s[left] - 'A'];
            ++left;
        }
        res = max(res, right - left + 1);
        right ++;
    }
    return res;
}
```

438. 找到字符串中所有字母异位词

```
vector<int> findAnagrams(string s, string p)
{
    if (s.empty() || s.size() < p.size())
        return {};
    vector<int> res, m1(256, 0), m2(256, 0);
    for (int i = 0; i < p.size(); ++i)
    {
        ++m1[s[i]];
        ++m2[p[i]];
    }
    if (m1 == m2)
        res.push_back(0);
    // 在s上形成窗口 进行滑动 窗口大小为p.size()
    for (int i = p.size(); i < s.size(); ++i)
    {
        ++m1[s[i]];
        --m1[s[i - p.size()]];
        if (m1 == m2)
            res.push_back(i - p.size() + 1);
    }
    return res;
}
```

567. 字符串的排列 (和438 差不多)

解法一 其他解法见 [LeetCode] Permutation in String 字符串中的全排列

```
// 解法一
// 先来分别统计s1和s2中前n1个字符串中各个字符出现的次数，其中n1为字符串s1的长度，
这样如果二者字符出现次数的情况完全相同，说明s1和s2中前n1的字符互为全排列关系，那么符合题意了，
直接返回true。如果不是的话，那么我们遍历s2之后的字符，对于遍历到的字符，对应的次数加1，
由于窗口的大小限定为了n1，所以每在窗口右侧加一个新字符的同时就要在窗口左侧去掉一个字符，
每次都比较一下两个哈希表的情况，如果相等，说明存在
bool checkInclusion(string s1, string s2)
{
    if (s1.size() < s2.size())
            return false;
    vector<int> m1(128), m2(128);
    for (int i = 0; i < s1.size(); ++i)
    {
        ++m1[s1[i]];
        ++m2[s2[i]];
    }
    if (m1 == m2)
        return true;
    for (int i = s1.size(); i < s2.size(); i++)
    {
        ++m2[s2[i]];
        --m2[s2[i - s1.size()]];
        if (m1 == m2)
            return true;
    }
    return false;
}
```

双指针问题

todo: 11和42的区别

11. 盛最多水的容器 #todo 20210419

```
int maxArea(vector<int>& height)
{
    if (height.empty())
        return 0;
    int res = 0;
    int left = 0;
    int right = height.size() -1;
    while (left < right)
    {
        res  = max(res, min(height[left], height[right]) * (right - left));
        // todo: 为什么是谁小谁移动
        if (height[left] <= height[right])
            left++;
        else
            right--;
    }
    return res;
}
```

42. 接雨水 #todo

```
// 解法二: 还不太懂
int trap(vector<int>& height)
 {
    if (height.empty())
        return 0;

    int n = height.size();
    int res = 0;
    int left_max = height[0];
    int right_max = height[n-1];

    int l = 1;
    int r = n - 2;
    while(l <= r)
    {
        if (left_max <= right_max)
        {
            res += max(0, left_max - height[l]);
            left_max = max(left_max, height[l]);
            l++;
        }
        else
        {
            res += max(0, right_max - height[r]);
            right_max = max(height[r], right_max);
            r--;
        }
    }
    return res;
}

int trap(vector<int> &height)
{
    if (height.empty())
        return 0;
    int res = 0;
    int i = 0;
    stack<int> monoStack; // 因为要求一个数左边比他大和右边比他大,所以应该是一个单调递减的栈,
    这个栈需要保持严格单调递减
    while (i < height.size())
    {
        // 如果满足入栈条件,则直接入栈
        if (monoStack.empty() || height[i] < height[monoStack.top()])
        {
            monoStack.push(i++);
        }
        else// 如果不满足入栈条件,则弹出栈顶元素,这个时候可以结算当前元素,栈顶元素的下一个元素则为
        左边界，当前遍历到的height[i]则为右边界
        {
            int tmp = monoStack.top();
            monoStack.pop();
            if (monoStack.empty())
                continue;
            int h = min(height[i], height[monoStack.top()]);
            res = res + (h - height[tmp]) * (i - monoStack.top() - 1);
        }
    }
    return res;
}
```

75. Sort Colors

```
class Solution
{
public:
    void sortColors(vector<int> &nums)
    {
        if (nums.size() < 0)
            return;
        int left = 0; // 小于区域的下一个位置
        int right = nums.size() -1; // 大于区域的上一个位置
        int index = 0;
        while(index <= right)
        {
            if(nums[index] < 1)
                swap(nums[index++], nums[left++]);
            else if (nums[index] == 1)
            {
                index ++;
            }
            else
                swap(nums[index], nums[right--]);
       }
    }
};
```

167. Two Sum II - Input array is sorted

```
vector<int> twoSum(vector<int>& numbers, int target)
{
    //使用双指针，一个指针指向值较小的元素，一个指针指向值较大的元素。指向较小元素的指针从头向尾遍历，指向较大元素的指针从尾向头遍历。
    //如果两个指针指向元素的和 sum == target，那么得到要求的结果；
    //如果 sum > target，移动较大的元素，使 sum 变小一些；
    //如果 sum < target，移动较小的元素，使 sum 变大一些。
    vector<int> res;
    int n = numbers.size();
    if (n <= 1)
        return res;

    int left = 0;
    int right = n-1;
    while(left < right)
    {
        if (numbers[left] + numbers[right] == target)
        {
            res.push_back(left+1);
            res.push_back(right+1);
            break;
        }
        else if (numbers[left] + numbers[right] < target)
            left++;
        else
            right--;
    }
    return res;
}
```

240. Search a 2D Matrix II

```
bool searchMatrix(vector<vector<int>> &matrix, int target)
{
    if (matrix.empty())
        return false;
    int m = matrix.size();
    int n = matrix[0].size();

    int row = 0;
    int col = n - 1;
    // 从右上角或者左下角开始比较
    while (row < m && col >= 0)
    {
        if (target == matrix[row][col])
            return true;
        else if (matrix[row][col] < target)
            row++;
        else
            col--;
    }
    return false;
}
```

524. Longest Word in Dictionary through Deleting #todo

```
string findLongestWord(string s, vector<string> &d)
{
    string res = "";
    for (string str : d)
    {
        int i = 0;
        for (char c : s)
        {
            if (i < str.size() && c == str[i])
                ++i;
        }
        if (i == str.size() && str.size() >= res.size())
        {
            if (str.size() > res.size() || str < res)
            {
                res = str;
            }
        }
    }
    return res;
}
```

单调栈系列问题

单调栈的两种写法 LeetCode Monotone Stack Summary 单调栈小结

```
// 写法一
int trap(vector<int>& height)
{
    if (height.empty())
        return 0;

    * int res = 0;
    int i = 0;
    stack<int> monoStack;
    while( i < height.size())
    {
        if (monoStack.empty() || height[i] <=height[monoStack.top()])
        {
            monoStack.push(i++);
        }
        else
        {
            int tmp =  monoStack.top();
            monoStack.pop();
            if (monoStack.empty())
                continue;

            int h = min(height[i], height[monoStack.top()]);
            res = res + (h - height[tmp]) * (i- monoStack.top() -1);
        }
    }
   return res;
}
// 写法二
int largestRectangleArea(vector<int> height)
{
    stack<int> monoStack;
    int res = 0;
    height.push_back(0);
    for (int i = 0; i < height.size(); i++)
    {

        while(!monoStack.empty() && height[i] <= height[monoStack.top()])  // 但栈非空
        时，且当前元素大于栈顶元素时，进行弹出操作，并且结算该弹出元素  弹出谁就结算谁
        {
            int h = height[monoStack.top()];
            monoStack.pop();
            int left = monoStack.empty() ? -1 : monoStack.top();
            // int left = monoStack.size() > 0 ? monoStack.top() : -1;
            res = max(res, h * (i - left -1));
        }
        monoStack.push(i);
    }
    return res;
}
```

42. Trapping Rain Water

```
int trap_1(vector<int> &height)
{
    if (height.empty())
        return 0;
    int res = 0;
    int i = 0;

    stack<int> monoStack;
    // height.push_back(0);
    for (int i = 0; i < height.size(); i++)
    {
        while (!monoStack.empty() && height[i] > height[monoStack.top()]) // 但栈非空时，
且当前元素大于栈顶元素时，进行弹出操作，并且结算该弹出元素
        // 栈顶元素的下一个元素则为左边界，当前遍历到的height[i]则为右边界
        {
            int tmp = monoStack.top();
            monoStack.pop();
            if (monoStack.empty())
                continue;

            int h = min(height[i], height[monoStack.top()]);
            res = res + (h - height[tmp]) * (i - monoStack.top() - 1);
        }
        monoStack.push(i);
    }
    return res;
}

int trap(vector<int> &height)
{
    if (height.empty())
        return 0;
    int res = 0;
    int i = 0;
    stack<int> monoStack; // 因为要求一个数左边比他大和右边比他大,所以应该是一个单调递减的栈， 这个
栈需要保持严格单调递减
    while (i < height.size())
    {
        // 如果满足入栈条件,则直接入栈
        if (monoStack.empty() || height[i] < height[monoStack.top()])
        {
            monoStack.push(i++);
        }
        else // 如果不满足入栈条件,则弹出栈顶元素,这个时候可以结算当前元素,栈顶元素的下一个元素则为左边界, 当前遍历到的height[i] 则为右边界
        {
            int tmp = monoStack.top();
            monoStack.pop();
            if (monoStack.empty())
                continue;
            int h = min(height[i], height[monoStack.top()]);
            res = res + (h - height[tmp]) * (i - monoStack.top() - 1);
        }
    }
    return res;
}

// 解法二: 还不太懂
int trap(vector<int>& height)
 {
    if (height.empty())
        return 0;

    int n = height.size();
    int res = 0;
    int left_max = height[0];
    int right_max = height[n-1];

    int l = 1;
    int r = n - 2;
    while(l <= r)
    {
        if (left_max <= right_max)
        {
            res += max(0, left_max - height[l]);
            left_max = max(left_max, height[l]);
            l++;
        }
        else
        {
            res += max(0, right_max - height[r]);
            right_max = max(height[r], right_max);
            r--;
        }
    }
   return res;
}
```

84. Largest Rectangle in Histogram

```
int largestRectangleArea(vector<int> &heights)
{
    int res = 0;
    stack<int> st;
    heights.push_back(0);
    for (int i = 0; i < heights.size(); ++i)
    {
        while (!st.empty() && heights[st.top()] >= heights[i])
        {
            int cur = st.top();
            st.pop();
            res = max(res, heights[cur] * (st.empty() ? i : (i - st.top() - 1)));
        }
        st.push(i);
    }
    return res;
}
```

85. 最大矩形

```
class Solution {

    int largestRectangleArea(vector<int> height)
    {
        stack<int> monoStack;
        int res = 0;
        height.push_back(0);
        for (int i = 0; i < height.size(); i++)
        {
            while(!monoStack.empty() && height[i] <= height[monoStack.top()])  // 但栈非
            空时，且当前元素大于栈顶元素时，进行弹出操作，并且结算该弹出元素
            {
                int h = height[monoStack.top()];
                monoStack.pop();
                int left = monoStack.empty() ? -1 : monoStack.top();
                // int left = monoStack.size() > 0 ? monoStack.top() : -1;
                res = max(res, h * (i - left -1));
            }
            monoStack.push(i);
        }
        return res;
    }
public:
    int maximalRectangle(vector<vector<char>>& matrix) {
        if (matrix.empty())
            return 0;
        vector<int> heights(matrix[0].size(), 0);
        int res = 0;
        int m = matrix.size();
        int n = heights.size();

        for(int i = 0; i < m; i++)
        {
            for(int j = 0; j < n; j++)
            {
                if (matrix[i][j] == '1')
                {
                    heights[j] += 1;
                }
                else
                {
                    heights[j] = 0;
                }
            }
            res = max(largestRectangleArea(heights), res);
        }
        return res;
    }
};
```

162. Find Peak Element

```
int findPeakElement(vector<int>& nums)
{
    if (nums.empty())
        return 0;

    stack<int> monoStack;
    for (int i = 0; i < nums.size(); i++)
    {
        if (!monoStack.empty() && nums[i] < nums[monoStack.top()])
        {
           int temp = monoStack.top();
           monoStack.pop();
           return temp;
        }
        else
        {
            monoStack.push(i);
        }

    }

    if (monoStack.size() == nums.size())
```

```
        return monoStack.top();

    return 0;
}
```

581. 最短无序连续子数组 #todo 20210419

```
class Solution {
public:
    int findUnsortedSubarray(vector<int>& nums)
    {
        stack<int> stack;
        int left = nums.size()-1;
        int right = 0;
        for(int i = 0; i < nums.size(); i++)
        {
            while(!stack.empty() && nums[i] < nums[stack.top()])
            {
                int temp = stack.top();
                stack.pop();
                left = min(left, temp);
            }
            stack.push(i);
        }

        while(!stack.empty())
            stack.pop();

        for(int i = nums.size() - 1; i >= 0; i--)
        {
            while(!stack.empty() && nums[i] > nums[stack.top()])
            {
                int temp = stack.top();
                stack.pop();
                right = max(right, temp);
            }
            stack.push(i);
        }
        return right - left > 0 ? right - left + 1 : 0;
    }
};
```

739. 每日温度

```
vector<int> dailyTemperatures(vector<int>& T)
{
    int n = T.size();
    vector<int> res(n);
```

```
        stack<int> s;
        for (int i = 0; i < n; ++i)
        {
            while (!s.empty() && T[i] > T[s.top()])
            {
                int pre_index = s.top();
                res[pre_index] = i - pre_index; // 弹出谁就结算谁
                s.pop();
            }
            s.push(i);
        }
        return res;
    }
```

402. Remove K Digits

https://www.cnblogs.com/grandyang/p/5883736.html

768. Max Chunks To Make Sorted II

https://www.cnblogs.com/grandyang/p/8850299.html

二分查找

34. 在排序数组中查找元素的第一个和最后一个位置

```
class Solution
{
public:
    int lower_bound(vector<int> &nums, int target)
    {
        int left = 0;
        int right = nums.size();
        while (left < right)
        {
            int mid = left + (right - left) / 2;
            if (target <= nums[mid])
                right = mid;
            else
            {
                left = mid + 1;
            }
        }
        return right;
    }

    int upper_bound(vector<int> &nums, int target)
    {
        int left = 0;
        int right = nums.size();
        while (left < right)
        {
            int mid = left + (right - left) / 2;
            if (target < nums[mid])
                right = mid;
            else
            {
                left = mid + 1;
            }
        }
        return right;
    }

    vector<int> searchRange(vector<int> &nums, int target)
    {
        if (nums.empty())
            return vector<int>{-1,-1};
        int low = lower_bound(nums, target);
        int up = upper_bound(nums, target) - 1;

        if (low == nums.size() || nums[low] != target)
            return vector<int>{-1,-1};
        return vector<int>{low,up};
    }
};
```

50. Pow(x, n)

```
double myPow(double x, int n) {
    //迭代的解法，让i初始化为n，然后看i是否是2的倍数，不是的话就让 res 乘以x。然后x乘以自己，i每次循环缩小一半，直到为0停止循环。最后看n的正负，如果为负，返回其倒数，
    double res = 1.0;
    for (int i = n; i != 0; i /= 2)
    {
        if (i % 2 != 0) res *= x;
        x *= x;
    }
    return n < 0 ? 1 / res : res;
}
```

69. Sqrt(x) #todo

```
int mySqrt(int x)
{
    if (x <= 1) return x;
    int left = 0, right = x;
    while (left < right) {
        int mid = left + (right - left) / 2;
        if (x / mid >= mid) left = mid + 1;
        else right = mid;
    }
    return right - 1;
}
```

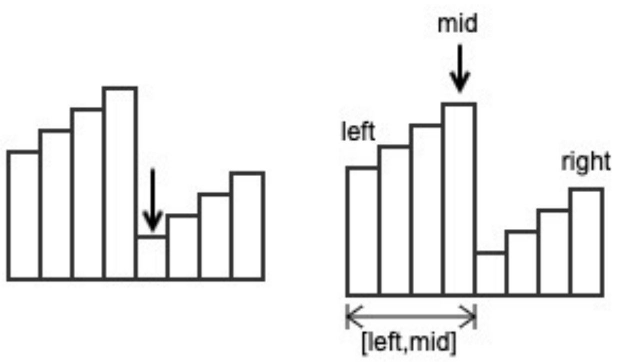
33. 搜索旋转排序数组 #todo

```
int search(vector<int>& nums, int target)
{
    int search(vector<int>& nums, int target)
    {
        int left = 0, right = nums.size() -1;
        while(left <= right)
        {
            int mid = left + (right - left);
            if (nums[mid] == target)
                return mid;
            if (nums[left] <= nums[mid])  // [left, mid] •递增序列
            {
                if (nums[left] <= target && target < nums[mid])  // 加等号，因为 left 可能是 target
                    right = mid - 1; // 在左侧 [left,mid) 查找  因为到了这个地方 num[mid] 已经不可能等于target了
                else
                    left = mid + 1;
            }
            else // [mid,right] 连续递增
            {
                if (nums[mid] < target && target <= nums[right]) // 加等号，因为 right 可能是 target
                    left = mid + 1; // 在右侧 (mid,right] 查找  因为到了这个地方 num[mid] 已经不可能等于target了
                else
                    right =  mid -1;
            }
        }
        return -1;
    }
}
```

81. 搜索旋转排序数组 II #todo

```
bool search(vector<int> &nums, int target)
{
    if (nums.empty())
        return false;

    int left = 0;
    int right = nums.size()-1;

    while(left <= right)
    {
        int mid = left + (right - left) / 2;
        if (nums[mid] == target)
            return true;

        if (nums[mid] < nums[right]) // 说明 [mid...right]是有序的
        {
            if (nums[mid] < target && target <= nums[right])
                left = mid + 1;
            else
                right = mid - 1;
        }
        else if (nums[mid] > nums[right]) // 说明[left...mid]是有序的
        {
            if(nums[left] <= target && target < nums[mid])
                right = mid-1;
            else
                left = mid + 1;
        }
        else if (nums[mid] == nums[right])
        {
            --right;
        }
    }
    return false;
}
```

153. 寻找旋转排序数组中的最小值



```
// 如果数组没有翻转，即 nums[left] <= nums[right]，则 nums[left] 就是最小值，直接返回
// 若 nums[left] <= nums[mid]，说明区间 [left,mid] 连续递增，则最小元素一定不在这个区间里，可以直接排除。因此，令 left = mid+1，在 [mid+1,right] 继续查找
//否则，说明区间 [left,mid] 不连续，则最小元素一定在这个区间里。因此，令 right = mid，在 [left,mid] 继续查找
//[left,right] 表示当前搜索的区间。注意 right 更新时会被设为 mid 而不是 mid-1，因为 mid 无法被排除。这一点和「33 题 查找特定元素」是不同的

int findMin(vector<int> &nums)
{
    int left = 0;
    int right = nums.size()  -1;
    while(left <= right)
    {
        if (nums[left] <= nums[right])
            return nums[left];
        int mid = left + (right - left ) / 2;
        if (nums[left] <= nums[mid]) //  [left,mid] 连续递增，则在 [mid+1,right] 查找
        {
            left = mid + 1
        }
        else // [left,mid] 不连续，在 [left,mid] 查找
        {
            right = mid
        }
    }
    return -1;
}
```

154. 寻找旋转排序数组中的最小值 II

```
int findMin(vector<int> &nums)
{
    int left = 0;
    int right = nums.size()  -1;
    while(left <= right)
    {
        if (nums[left] < nums[right] || left == right)
            return nums[left];
        int mid = left + (right - left ) / 2;
        if (nums[left] < nums[mid]) //  [left,mid] 连续递增，则在 [mid+1,right] 查找
        {
            left = mid + 1;
        }
        else if (nums[left] == nums[mid])
        {
            left++;
        }
        else // [left,mid] 不连续，在 [left,mid] 查找
        {
            right = mid;
        }
    }
    return -1;
}
```

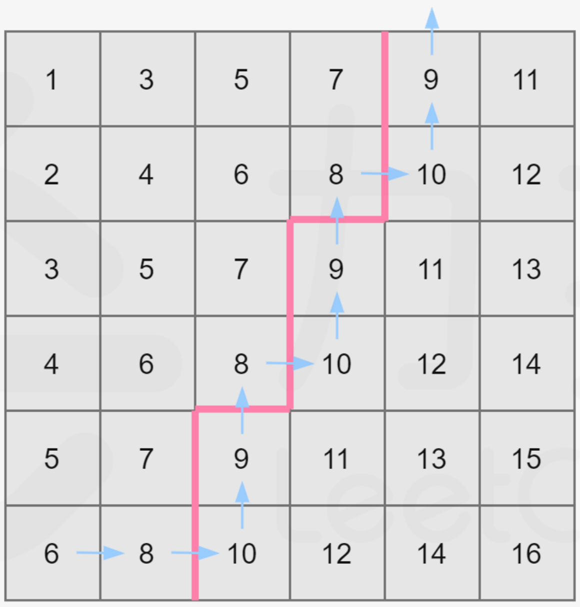
278. 第一个错误的版本

```
int firstBadVersion(int n) {
    int left = 1, right = n;
    while (left < right) {
        int mid = left + (right - left) / 2;
        if (isBadVersion(mid)) right = mid;
        else left = mid + 1;
    }
    return left;
}
```

287. 寻找重复数 #TODOz

```
// 解法一：二分查找
int findDuplicate(vector<int>& nums)
{
    // 把1~n的数的数字从中间的数字mid分为两部分，1~mid, mid+1~n;
    // 如果1~mid中的数据数目超过mid,那么1~mid中间肯定存在重复数字，否则mid+1~n肯定存在重复数字
    int left = 1, right = nums.size();
    while (left < right){
        int mid = left + (right - left) / 2, cnt = 0;
        for (int num : nums) {
            if (num <= mid) ++cnt;
        }
        // 根据抽屉原理，小于等于 4 的个数如果严格大于 4 个
        // 此时重复元素一定出现在 [1, 4] 区间里
        if (cnt <= mid) left = mid + 1;
        else right = mid;    // 重复元素位于区间 [left, mid]
    }
    return left;
}
```

378. 有序矩阵中第K小的元素



```
class Solution {
public:
    // 计算矩阵中有多少个数字小于target
    int search_less_equal(vector<vector<int>>& matrix, int target) {
        int n = matrix.size(), i = n - 1, j = 0, res = 0;
        while (i >= 0 && j < n) {
            if (matrix[i][j] <= target) {
                res += i + 1;
                ++j;
            } else {
                --i;
            }
        }
        return res;
    }

    int kthSmallest(vector<vector<int>>& matrix, int k)
    {
        int left = matrix[0][0],
        right = matrix.back().back();
        while (left < right)
        {
            int mid = left + (right - left) / 2;
            int cnt = search_less_equal(matrix, mid);
            if (cnt < k)
                left = mid + 1;
            else
                right = mid;
        }
        return left;
    }
};
```

704. Binary Search

```
int search(vector<int> &nums, int target)
{
    int left = 0, right = nums.size();
    while (left < right)
    {
        int mid = left + (right - left) / 2;
        if (nums[mid] == target)
            return mid;
        else if (nums[mid] < target)
            left = mid + 1;
        else
            right = mid;
    }
    return -1;
}
```

```

// 形成自己的套路，right 初始化为nums.size(), while判断就可以用left < right了
int find(vector<int>& nums, int target) {
    int left = 0, right = nums.size();
    while (left < right) {
        int mid = left + (right - left) / 2;
        if (nums[mid] == target) return mid;
        else if (nums[mid] < target) left = mid + 1;
        else right = mid;
    }
    return -1;
}

int find(vector<int>& nums, int target) {
    int left = 0, right = nums.size();
    while (left < right) {
        int mid = left + (right - left) / 2;
        if (nums[mid] < target) left = mid + 1;
        else right = mid;
    }
    return right;
}Kth Smallest Element in a Sorted Matrix](https://leetcode.com/problems/kth-smallest-
element-in-a-sorted-matrix/)
```

排序

归并排序

[493. Reverse Pairs]

```
class Solution
{
    int merge(vector<int> &arr, int left, int mid, int right)
    {
        vector<int> help;
        int index = 0;
        int i = left;
        int j = mid + 1;
        int count = 0;
        // todo: 核心是这个循环
        while (i <= mid && j <= right)
        {
            if (arr[i] > 2LL * arr[j])
            {
                count += mid - i + 1;
                ++j;
            }
            else
                ++i;
        }

        i = left;
        j = mid + 1;
        index = 0;
        while (i <= mid && j <= right)
        {
            // help[index++] = arr[i] < arr[j] ? arr[i++] : arr[j++];
            // int temp =

            help.push_back(arr[i] < arr[j] ? arr[i++] : arr[j++]);
        }

        while (i <= mid)
        {
            help.push_back(arr[i++]);
        }
        while (j <= right)
        {
            help.push_back(arr[j++]);
        }

        for (index = 0; index < help.size(); index++)
        {
            arr[left + index] = help[index];
        }
        return count;
    }

    int mergeSort(vector<int> &arr, int l, int r)
    {
        if (l == r)
        {
            return 0;
        }
        int mid = l + ((r - l) >> 1);
        int left = mergeSort(arr, l, mid);
        int right = mergeSort(arr, mid + 1, r);
        int count = merge(arr, l, mid, r);
        return left + right + count;
    }

public:
    // 思路：利用归并排序，但是代码写不出来
    int reversePairs(vector<int> &nums)
    {
        if (nums.size() <= 1)
            return 0;

        return mergeSort(nums, 0, nums.size() - 1);
    }
};
```

912. 排序数组

```
class Solution {
public:
    vector<int> sortArray(vector<int>& nums) {
      mergeSort(nums, 0, (int)nums.size() - 1);
      return nums;
    }
    void mergeSort(vector<int>& nums, int start, int end) {
      if (start >= end) return;
      int mid = (start + end) / 2;
      mergeSort(nums, start, mid);
      mergeSort(nums, mid + 1, end);
      merge(nums, start, mid, end);
    }
    void merge(vector<int>& nums, int start, int mid, int end) {
        vector<int> tmp(end - start + 1);
        int i = start, j = mid + 1, k = 0;
        while (i <= mid && j <= end) {
          if (nums[i] < nums[j]) tmp[k++] = nums[i++];
          else tmp[k++] = nums[j++];
        }
        while (i <= mid) tmp[k++] = nums[i++];
        while (j <= end) tmp[k++] = nums[j++];
        for (int idx = 0; idx < tmp.size(); ++idx) {
          nums[idx + start] = tmp[idx];
        }
    }
};
```

快速排序

```
 int partition(vector<int> &nums, int left, int right)
{
    int small = left -1;
    for(int i = left; i < right; i++)
    {
        if (nums[i] < nums[right])
            swap(nums[i], nums[++small]);
    }
    swap(nums[++small], nums[right]);
    return small;
}

void quickSort(vector<int> &nums, int left, int right)
{
    if (left < right)
    {
        int index = partition(nums, left, right);
        quickSort(nums, left, index - 1);
        quickSort(nums, index + 1, right);
    }
}

vector<int> MySort(vector<int>& arr) {
    // write code here
    quickSort(arr, 0, arr.size()-1);
    return arr;
}
```

75. Sort Colors 快排partion

```

    void sortColors(vector<int> &nums)
    {
        if (nums.size() < 0)
            return;
        int left = 0; // 小于区域的下一个位置
        int right = nums.size() -1; // 大于区域的上一个位置
        int index = 0;
        while(index <= right)
        {
            if(nums[index] < 1)
                swap(nums[index++], nums[left++]);
            else if (nums[index] == 1)
            {
                index ++;
            }
            else
                swap(nums[index], nums[right--]);
        }
    }
};
```

324. Wiggle Sort II #todo

核心思想， 如果当前数小于num,当前数和小于区域的下一个数交换, 如果当前数大于num,当前数和大于区域的前一
个数交换

```
class Solution {
public:

    int partrition(vector<int>& nums,int left,int right)
    {
        int i=left,j=right;
        int pivot=nums[i];//基准值
        while(i<j){
            //当尾部元素大于等于基准值时向前移动后面的指针
            while(i<j&&nums[j]>=pivot)
                j--;
            if(i<j){
                nums[i]=nums[j];
                i++;
            }
            //当前面的元素小于基准值时向后移动前面的指针
            while(i<j&&nums[i]<pivot)
                i++;
            if(i<j){
                nums[j]=nums[i];
                j--;
            }
        }
        nums[i]=pivot;
        return i;
    }

    int partrition_2(vector<int>& input, int start, int end)
    {
        int small = start - 1; // small 指向小于区域的最后一个元素
        for(int i = start;i < end; i++)
        {
            if(input[i] < input[end])
            {
                small++;
                if (i != small)
                    swap(input[small], input[i]);
            }
        }
        small++;
        swap(input[small],input[end]);
        return small;
    }


    int quickSelect(vector<int> &nums, int begin, int end, int k)
    {
        int index = partrition_2(nums, begin, end);

        while(index != k)
        {
            if (index > k)
            {
                end = index -1;
                index = partrition_2(nums, begin, end);
            }
            else
            {
                begin = index + 1;
                index = partrition_2(nums, begin, end);
            }
        }
        return index;
    }
    void wiggleSort(vector<int>& nums)
    {
        // 先找到中位数，


        int mid = nums[quickSelect(nums, 0, nums.size()-1, nums.size()/2)];
        auto midptr = nums.begin() + nums.size() / 2;
        int i = 0, j = 0, k = nums.size() - 1;
        // 3-way-partition
        // 然后根据中位数将原数组被分为3个部分，左侧为小于中位数的数，中间为中位数，右侧为大于中位数的
数
        while (j < k)
        {
            if (nums[j] > mid)
            {
                swap(nums[j], nums[k]);
                k--;
            }
            else if (nums[j] < mid)
            {
                swap(nums[j], nums[i]);
                i++;
                j++;
            }
            else
            {
                j++;
            }
        }

        if (nums.size() % 2)
            ++midptr;

        vector<int> tmp1(nums.begin(), midptr);
        vector<int> tmp2(midptr, nums.end());

        for (int i = 0; i < tmp1.size(); i++)
        {
            nums[2 * i] = tmp1[tmp1.size() - 1 - i];
        }
        for (int i = 0; i < tmp2.size(); i++)
        {
            nums[2 * i + 1] = tmp2[tmp2.size() - 1 - i];
        }
    }
};
```

链表

2. 两数相加

```
ListNode* addTwoNumbers(ListNode* l1, ListNode* l2)
{
    ListNode *fakeHead = new ListNode(-1), *cur = fakeHead;
    int carry = 0;
    while (l1 || l2)
    {
        int val1 = l1 ? l1->val : 0;
        int val2 = l2 ? l2->val : 0;
        int sum = val1 + val2 + carry;
        carry = sum / 10;
        cur->next = new ListNode(sum % 10);
        cur = cur->next;
        if (l1) l1 = l1->next;
        if (l2) l2 = l2->next;
    }
    if (carry) cur->next = new ListNode(1);
    return fakeHead->next;
}
```

24. 两两交换链表中的节点

```
ListNode *swapPairs(ListNode *head)
{
    ListNode *dummy = new ListNode(-1), *pre = dummy;
    dummy->next = head;
    while (pre->next && pre->next->next)
    {
        ListNode *t = pre->next->next;
        pre->next->next = t->next;
        t->next = pre->next;
        pre->next = t;
        pre = t->next;
    }
    return dummy->next;
}
```

82. 删除排序链表中的重复元素 II

```
ListNode *deleteDuplicates(ListNode *head)
{
    if (!head || !head->next)
        return head;
    ListNode *dummy = new ListNode(-1), *pre = dummy;
    dummy->next = head;
    while (pre->next)
    {
        ListNode *cur = pre->next;
        while (cur->next && cur->next->val == cur->val)
        {
            cur = cur->next;
        }
        if (cur != pre->next)
            pre->next = cur->next;
        else
            pre = pre->next;
    }
    return dummy->next;
}
```

83. 删除排序链表中的重复元素

```
ListNode* deleteDuplicates(ListNode* head) {
    if (head == NULL || head->next == NULL)
        return head;

    ListNode *p = head;
    while(p && p->next)
    {
        if (p->val == p->next->val)
        {
            p->next = p->next->next;
        }
        else
        {
            p = p->next;
        }

    }
    return head;
}
```

138. 复制带随机指针的链表

```
Node* copyRandomList(Node* head)
{
    if (head == nullptr)
        return head;

    Node *res = new Node(head->val, nullptr, nullptr);
    Node *node = res;
    Node *cur = head->next;
    unordered_map<Node *, Node *> m;
    m[head] = node;
    // 第一遍遍历生成所有新节点时同时建立一个原节点和新节点的 HashMap
    while(cur)
    {
        Node *temp = new Node(cur->val, nullptr, nullptr);
        node->next = temp;
        m[cur] = temp;
        node = node->next;
        cur = cur->next;
    }
    // 第二遍给随机指针赋值时
    node = res; cur = head;
    while(cur)
    {
        node->random = m[cur->random];
        cur = cur->next;
        node = node->next;
    }
    return res;
}
```

K路归并

21. 合并两个有序链表

```
class Solution {
public:
    ListNode* mergeTwoLists(ListNode* l1, ListNode* l2)
    {
        if (l1 == NULL && l2 == NULL)
            return NULL;
        ListNode *pHead = new ListNode(-1);
        ListNode *tail = pHead;

        ListNode *p = l1;
        ListNode *q = l2;
        // 依次摘下一个节点相当于尾插法
        while (p && q)
        {
            if (p->val < q->val)
            {
                tail->next = p;
                p = p->next;
            }
            else
            {
                tail->next = q;
                q = q->next;
            }
            tail= tail->next;
        }
        tail->next = p ? p : q;
        return pHead->next;
    }
};
```

23. 合并K个升序链表 #Todo 最小堆的做法 需要熟悉STL

```
// 解法一
class Solution
{
public:
    ListNode *mergeKLists(vector<ListNode *> &lists)
    {
        if (lists.empty())
            return NULL;
        int n = lists.size();
        while (n > 1)
        {
            int k = (n + 1) / 2;
            for (int i = 0; i < n / 2; ++i)
            {
                lists[i] = mergeTwoLists(lists[i], lists[i + k]);
            }
            n = k;
        }
        return lists[0];
    }
    ListNode *mergeTwoLists(ListNode *l1, ListNode *l2)
    {
        ListNode *pHead = new ListNode(-1), *cur = pHead;
        while (l1 && l2)
        {
            if (l1->val < l2->val)
            {
                cur->next = l1;
                l1 = l1->next;
            }
            else
            {
                cur->next = l2;
                l2 = l2->next;
            }
            cur = cur->next;
        }
        if (l1)
            cur->next = l1;
        if (l2)
            cur->next = l2;
        return pHead->next;
    }
};

// 解法二 （https://github.com/grandyang/leetcode/issues/23）
class Solution {
public:
    // 利用了最小堆这种数据结构，首先把k个链表的首元素都加入最小堆中，它们会自动排好序。然后每次取出最小的那个元素加入最终结果的链表中，然后把取出元素的下一个元素再加入堆中，下次仍从堆中取出最小的元素做相同的操作，以此类推，直到堆中没有元素了，此时k个链表也合并为了一个链表，返回首节点即可
    ListNode* mergeKLists(vector<ListNode*>& lists) {
        auto cmp = [](ListNode*& a, ListNode*& b) {
            return a->val > b->val;
        };

        priority_queue<ListNode*, vector<ListNode*>, decltype(cmp) > q(cmp);
        for (auto node : lists) {
            if (node) q.push(node);
        }
        ListNode *dummy = new ListNode(-1), *cur = dummy;
        while (!q.empty()) {
            auto t = q.top(); q.pop();
            cur->next = t;
            cur = cur->next;
            if (cur->next) q.push(cur->next);
        }
        return dummy->next;
    }
};
```

快慢指针

剑指 Offer 22. 链表中倒数第k个节点

```
ListNode* getKthFromEnd(ListNode* head, int k)
{
    if (head == nullptr || k < 0)
        return head;

    ListNode *fast = head;
    ListNode *slow = head;

    for(int i = 0; i < k; i++)
    {
        fast = fast->next;
    }
    while (fast)
    {
        fast = fast->next;
        slow = slow->next;
    }
    return slow;
}
```

19. 删除链表的倒数第 N 个结点 # todo 注意细节

```
ListNode* removeNthFromEnd(ListNode* head, int n)
{
    if (head == nullptr)
        return head;

    ListNode *fast = head;
    ListNode *slow = head;
    for(int i = 0; i < n; i++)
    {
        fast = fast->next;
    }
    // todo: 注意细节
    if (fast == nullptr) // 防止n等链表长度 正好删除第一个节点
        return head->next;
    while(fast->next)
    {
        fast = fast->next;
        slow = slow->next;
    }
    slow->next= slow->next->next;
    return head;
}
```

61. 旋转链表 #todo

```
ListNode *rotateRight(ListNode *head, int k)
{
    if (!head) return NULL;
    int n = 0;
    ListNode *cur = head;
    while (cur)
    {
        ++n;
        cur = cur->next;
    }
    k %= n;
    ListNode *fast = head, *slow = head;
    for (int i = 0; i < k; ++i)
    {
        if (fast) fast = fast->next;
    }
    if (!fast) return head;
    while (fast->next)
    {
        fast = fast->next;
        slow = slow->next;
    }
    fast->next = head;
    fast = slow->next;
    slow->next = NULL;
    return fast;
}
```

141. Linked List Cycle

```
bool hasCycle(ListNode *head)
{
    if (head == nullptr || head->next == nullptr || head->next->next == nullptr)
        return false;

    ListNode *slow =  head->next;
    ListNode *fast = head->next->next;

    while(slow != fast)
    {
        if (fast->next == nullptr || fast->next->next == nullptr)
            return false;
        slow = slow->next;
        fast =  fast->next->next;
    }
    return true;
}
```

142. Linked List Cycle II

```
 ListNode *detectCycle(ListNode *head)
{
    if (head == nullptr || head->next == nullptr || head->next->next == nullptr)
        return nullptr;

    ListNode *slow = head->next;
    ListNode *fast = head->next->next;

    while(slow != fast)
    {
        if (fast->next == nullptr || fast->next->next == nullptr)
            return nullptr;

        slow = slow->next;
        fast = fast->next->next;


    }
    fast = head;
    while(slow != fast)
    {
        slow = slow->next;
        fast = fast->next;
    }
    return slow;
}
```

143. Reorder List #TODO

```
/*
解法一：
1. 使用快慢指针来找到链表的中点，并将链表从中点处断开，形成两个独立的链表。
2. 将第二个链翻转。
3. 将第二个链表的元素间隔地插入第一个链表中。
*/
void reorderList_1(ListNode *head)
{
    if (!head || !head->next || !head->next->next)
        return;
    ListNode *fast = head, *slow = head;
    while (fast->next && fast->next->next)
    {
        slow = slow->next;
        fast = fast->next->next;
    }
    ListNode *mid = slow->next;
    slow->next = NULL;
    // 反转链表
    ListNode *last = mid, *pre = NULL;
    while (last)
    {
        ListNode *temp = last->next;
        last->next = pre;
        pre = last;
        last = temp;
    }
    while (head && pre)
    {
        ListNode *temp = head->next;
        head->next = pre;
        pre = pre->next;
        head->next->next = temp;
        head = temp;
    }
}
/**
 * 解法二：
 * 其实可以借助栈的后进先出的特性来做，如果我们按顺序将所有的结点压入栈，那么出栈的时候就可以倒序了，实际上就相当于翻转了链表。由于只需将后半段链表翻转，那么我们就要控制出栈结点的个数，还好栈可以直接得到结点的个数，我们减1除以2，就是要出栈结点的个数了。
 */
void reorderList(ListNode *head)
{
    if (!head || !head->next || !head->next->next)
        return;
    stack<ListNode *> st;
    ListNode *cur = head;
    while (cur)
    {
        st.push(cur);
        cur = cur->next;
    }
    int cnt = ((int)st.size() - 1) / 2;
    cur = head;
    while (cnt > 0)
    {
        auto t = st.top();
        st.pop();
        ListNode *next = cur->next;
        cur->next = t;
        t->next = next;
        cur = next;
        cnt--;
    }
    st.top()->next = NULL;
}
```

160. 相交链表

```
/**
 * Definition for singly-linked list.
 * struct ListNode {
 *     int val;
 *     ListNode *next;
 *     ListNode(int x) : val(x), next(NULL) {}
 * };
 */
class Solution {
    int getLen(ListNode *head)
    {
        int len = 0;
        while(head)
        {
            len++;
            head = head->next;

        }
        return len;
    }
public:
    ListNode *getIntersectionNode(ListNode *headA, ListNode *headB)
    {
        if (headA == nullptr || headB == nullptr)
            return nullptr;

        int len1 = getLen(headA);
        int len2 = getLen(headB);

        ListNode *lon = headA;
        ListNode *small = headB;

        if (len1 < len2)
        {
            lon = headB;
            small = headA;
        }

        for(int i = 0; i < abs(len1 - len2); i++)
        {
            lon = lon->next;
        }

        while(lon && small && lon->val != small->val)
        {
            lon = lon->next;
            small = small->next;
        }
        return lon;
    }
};
```

234. 回文链表

```
/**
 * struct ListNode {
 *  int val;
 *  struct ListNode *next;
 * };
 */

class Solution {
public:
    /**
     *
     * @param head ListNode类 the head
     * @return bool布尔型
     */
    ListNode * reverse(ListNode *head)
    {
        ListNode *pre = nullptr;
        ListNode *cur = head;
        while(cur)
        {
            ListNode *temp = cur->next;
            cur->next = pre;
            pre = cur;
            cur = temp;
        }
        return pre;
    }
    bool isPail(ListNode* head) {
        // write code here

        if (head->next == nullptr || head->next->next == nullptr)
            return head;
        ListNode *fast = head;
        ListNode *slow = head;
        // 找到中间结点 slow 中间结点或者是中间结点的前一个
        while(fast->next && fast->next->next)
        {
            slow = slow->next;
            fast = fast->next->next;
        }

        slow->next = reverse(slow->next);

        slow = slow->next;
        while (slow != NULL)
        {
            if (head->val != slow->val)
```

```
                return false;
            head = head->next;
            slow = slow->next;
        }
        return true;
    }
};
```

链表排序

86. 分隔链表 # todo?

```
 // 将所有小于给定值的节点取出组成一个新的链表，此时原链表中剩余的节点的值都大于或等于给定值，只要将原链表直接接在新链表后
class Solution
{
public:
    ListNode *partition(ListNode *head, int x)
    {
        if (!head) return head;
        ListNode *large_head = new ListNode(-1);
        ListNode *small_head = new ListNode(-1);
        large_head->next = head;
        ListNode *cur = large_head, *p = small_head;
        while (cur->next)
        {
            if (cur->next->val < x)
            {
                p->next = cur->next;
                p = p->next;
                cur->next = cur->next->next;
            }
            else
            {
                cur = cur->next;
            }
        }
        p->next = large_head->next;
        return small_head->next;
    }
};
```

147. Insertion Sort List #todo 对链表使用插入排序

```
ListNode *insertionSortList(ListNode *head)
{
    if (head == nullptr)
        return head;
    ListNode *fakeHead = new ListNode(-1);

    ListNode *p = nullptr;
    fakeHead->next = nullptr;
    while (head)
    {
        p = head->next;
        ListNode *q = fakeHead;

        if (fakeHead->next == nullptr)
        {
            fakeHead->next = head;
            head->next = nullptr;
            head = p;
        }
        else
        {
            while (q->next && q->next->val < head->val)
            {
                q = q->next;
            }
            head->next = q->next;
            q->next = head;
            head = p;
        }
    }
    return fakeHead->next;
}
```

148. Sort List 对链表使用归并的方式排序

```
// 合并两个有序链表
ListNode *merge(ListNode *l1, ListNode *l2)
{
    ListNode *fakeHead = new ListNode(-1);
    ListNode *p = fakeHead;

    while (l1 != nullptr && l2 != nullptr)
    {
        if (l1->val < l2->val)
        {
            p->next = l1;
            l1 = l1->next;
        }
        else
        {
            p->next = l2;
            l2 = l2->next;
        }
```

```
            p = p->next;
        }
        if (l1 != nullptr)
        {
            p->next = l1;
        }
        if (l2 != nullptr)
        {
            p->next = l2;
        }

        return fakeHead->next;
    }

    //O(nlogn)对链表排序 归并和递归的方式
    ListNode *sortList(ListNode *head)
    {
        if (head == nullptr || head->next == nullptr)
                return head;

        // 统一用这种方式寻找中间节点
        ListNode *fast = head, *slow = head;
        while(fast->next && fast->next->next)
        {
            slow = slow->next;
            fast = fast->next->next;
        }

        fast = slow->next;
        slow->next = nullptr;

        return merge(sortList(head), sortList(fast));
    }
```

链表翻转

2. 两数相加 #todo

```
ListNode *addTwoNumbers(ListNode *l1, ListNode *l2)
{
    ListNode *fakeHead = new ListNode(-1), *cur = fakeHead;
    int carry = 0; // 表示进位

    while (l1 || l2)
    {
        int val1 = l1 ? l1->val : 0;
        int val2 = l2 ? l2->val : 0;
        int sum = val1 + val2 + carry;
        carry = sum / 10;
        cur->next = new ListNode(sum % 10);
        cur = cur->next;
        if (l1)
            l1 = l1->next;
        if (l2)
            l2 = l2->next;
    }
    if (carry)
        cur->next = new ListNode(1);

    return fakeHead->next;
}
```

25. K 个一组翻转链表 #todo

```
// 每k组翻转
ListNode* reverseKGroup(ListNode* head, int k)
{
    ListNode *fakeHead = new ListNode(-1);
    fakeHead->next = head;

    ListNode *pre = fakeHead;
    ListNode *cur = fakeHead;

    int num = 0;
    while(cur->next)
    {
        ++num;
        cur = cur->next;
    }
    while(num >= k)
    {
        cur = pre->next;
        for(int i = 1; i < k; i++)
        {
            // 把cur后面的一个节点temp摘下来，然后用头插法插入到pre后面
            ListNode *temp = cur->next;
            cur->next = temp->next;
            temp->next = pre->next;
            pre->next = temp;
        }
        pre = cur;
        num -= k;
    }
    return fakeHead->next;
}
```

92. 反转链表 II

```
ListNode *reverseBetween(ListNode *head, int m, int n)
{
    ListNode *dummy = new ListNode(-1), *pre = dummy;
    dummy->next = head;
    for (int i = 0; i < m - 1; ++i)
        pre = pre->next;
    ListNode *cur = pre->next;
    for (int i = m; i < n; ++i)
    {
      // 这个地方依然是头插法
        ListNode *t = cur->next;
        cur->next = t->next;
        t->next = pre->next;
        pre->next = t;
    }
    return dummy->next;
}
```

206. 反转链表

```
// 非递归
ListNode *reverseList(ListNode *head)  // 没有额外使用头结点的方式
{
    if(!head)
        return nullptr;
    ListNode *pre = nullptr;
    ListNode *cur = head;
    while(cur)
    {
        ListNode *temp = cur->next;
        cur->next = pre;
        pre = cur;
        cur = temp;
    }
    return pre;
}

// 递归
ListNode* reverseList(ListNode* head)
{
    if (!head || !head->next) return head;
    ListNode *newHead = reverseList(head->next);
    head->next->next = head;
    head->next = NULL;
    return newHead;
}
```

328. 奇偶链表

```
class Solution
{
public:
    /*
    * 解法一：
    * 可以使用两个指针来做，pre指向奇节点，cur指向偶节点，然后把偶节点cur后面的那个奇节点提前到pre的后面，然后pre和cur各自前进一步，此时cur又指向偶节点，pre指向当前奇节点的末尾，以此类推直至把所有的偶节点都提前了即可
    */
    ListNode *oddEvenList_1(ListNode *head)
    {
        if (!head || !head->next)
            return head;
        ListNode *pre = head, *cur = head->next;
        while (cur && cur->next)
        {
            ListNode *tmp = pre->next;
            pre->next = cur->next;
            cur->next = cur->next->next;
            pre->next->next = tmp;
            cur = cur->next;
            pre = pre->next;
        }
        return head;
    }
    /*
    * 解法二：
    * 用两个奇偶指针分别指向奇偶节点的起始位置，另外需要一个单独的指针even_head来保存偶节点的起点位置，然后把奇节点的指向偶节点的下一个(一定是奇节点)，此奇节点后移一步，再把偶节点指向下一个奇节点的下一个(一定是偶节点)，此偶节点后移一步，以此类推直至末尾，此时把分开的偶节点的链表连在奇节点的链表后即可
    */
    ListNode *oddEvenList(ListNode *head)
    {
        if (!head || !head->next)
            return head;
        ListNode *odd = head, *even = head->next, *even_head = even;
        while (even && even->next)
        {
            // odd = odd->next = even->next;
            // even = even->next = odd->next;
            odd->next = even->next;
            odd = odd->next;
            even->next = odd->next;
            even = even->next;
        }
        odd->next = even_head;
        return head;
    }
};
```

动态规划

Leetcode 题解 - 动态规划

1.坐标型动态规划

状态: f(x)表示从起点走到坐标x, f[x][y]表示我从起点走到坐标x,y; 方程: 研究走到x, y这个点之前的一步; 初始化: 起点; 答案: 终点

62. 不同路径

```
int uniquePaths(int m, int n)
{
    // int dp[m][n];
    // dp[0][0] = 0;
    // dp[0][1] = 1;
    // dp[1][0] = 1;
    // dp[1][1] = 2;

    // dp[i][j] 表示从[0][0]--->[i][j] 有多少种走法
    // 第0行和第0列 在边界上所以只有一种方法
    vector<vector<int>> dp(m, vector<int>(n, 1));

    for(int i = 1; i < m; i++)
    {
        for(int j = 1; j < n; j++)
        {
            dp[i][j] = dp[i-1][j] + dp[i][j-1];
        }
    }
    return dp[m-1][n-1];
}
```

63. 不同路径 II

难度中等531

```
int uniquePathsWithObstacles(vector<vector<int>> &obstacleGrid)
{
    int m = obstacleGrid.size();
    int n = obstacleGrid[0].size();
    if (obstacleGrid.empty() || obstacleGrid[0].empty() || obstacleGrid[0][0] == 1)
    {
        return 0;
    }
    vector<vector<int>> dp(m, vector<int>(n, 0));
    // 先初始化边界
    for (int i = 0; i < m; i++)
    {
        if (obstacleGrid[i][0] != 1)
            dp[i][0] = 1;
        else
            break;
    }
    for (int j = 0; j < n; j++)
    {
        if (obstacleGrid[0][j] != 1)
            dp[0][j] = 1;
        else
            break;
    }

    // dp[i][j] 表示从[0][0]--->[i][j] 有多少种走法
    for (int i = 1; i < m; i++)
    {
        for (int j = 1; j < n; j++)
        {
            if (obstacleGrid[i][j] == 1)
                dp[i][j] = 0;
            else
                dp[i][j] = dp[i - 1][j] + dp[i][j - 1];
        }
    }
    return dp[m - 1][n - 1];
}
```

64. 最小路径和

```
int minPathSum(vector<vector<int>> &grid)
{
    int m = grid.size();
    int n = grid[0].size();

    // dp[i][j] 表示从[0][0]-->[i][j]的最短路径和
    vector<vector<int>> dp(grid); // 这里使用grid直接初始化是为了累加数组方便

    for (int j = 1; j < n; ++j)
    {
        dp[0][j] += dp[0][j - 1];
    }
    for (int j = 1; j < m; ++j)
    {
        dp[j][0] += dp[j - 1][0];
    }
    for (int i = 1; i < m; i++)
    {
        for (int j = 1; j < n; j++)
        {
            dp[i][j] = min(dp[i - 1][j], dp[i][j - 1]) + grid[i][j];
        }
    }
    return dp[m - 1][n - 1];
}
```

70. Climbing Stairs

```
int climbStairs(int n)
{
    if (n <= 1)
        return 1;
    vector<int> dp(n, 0);
    dp[0] = 1;
    dp[1] = 2;
    for (int i = 2; i < n; i++)
    {
        dp[i] = dp[i - 1] + dp[i - 2];
    }
    return dp[n - 1];
}
```

120. Triangle #todo

```
int minimumTotal(vector<vector<int>>& triangle)
{
    int n = triangle.size();
    for (int i = n-2; i >=0; --i)
    {
        for (int j = 0; j <= i;  j++)
        {

            triangle[i][j] += min(triangle[i+1][j+1], triangle[i+1][j]);
        }
    }
    return triangle[0][0];
}
```

2.单序列动态规划

状态: f[i]表示前i个位置/数字/字符, 第i个; 方程: f[i] = f(f[j]), j是i之前的一个位置; 初始化: f[0]; 答案: f[n-1]; 小技巧: 一般有N个数字/字符, 就开N+1个位置的数组, 第0个位置单独留出来作初始化.(跟坐标相关的动态规划除外)

32. 最长有效括号

```
int longestValidParentheses(string s)
{
    int n = s.length();
    vector<int> dp(n, 0);  //dp[i]表示以s[i]结尾的最长有效字符串长度
    int res = 0;
    for(int i = 1; i < n; i++)
    {
        if (s[i] == '(') // 如果遇到左括号，说明以当前字符结尾不可能形成有效括号字符串，所以dp[i] = 0;
            dp[i] = 0;

        if (s[i] == ')')  // 当前字符为右括号时，那么找到前一个字符位置形成的最长有效括号字符串的长度，在这个长度之前的字符串如果是左括号，那么可以形成有效括号字符串
            // 即 dp[i] = dp[i-1] + 2, 需要注意的是 需要加上 前一个字符位置形成的有效括号字符串;
        {
            int pre = i - dp[i-1] - 1;
            if (pre >= 0 && s[pre] == '(')
            {
                dp[i] = dp[i-1] + 2 + (pre > 0 ? dp[pre-1] : 0);
            }
        }
        res = max(res, dp[i]);
    }
    return res;
}
```

45. Jump Game II 好像不是dp #todo 20210415

```
int jump(vector<int>& nums)
{
    int level = 0;
    int cur_begin = 0;
    int cur_end = 0;
    int next_end = 0;

    while(cur_end < nums.size() -1)
    {
        for(int i = cur_begin; i <= cur_end; i++)
        {
            next_end = max(next_end, i + nums[i]);
        }
        ++level;
        cur_begin = cur_end + 1;
        cur_end = next_end;
    }

    return level;
}
```

55. 跳跃游戏

```
bool canJump(vector<int>& nums)
{
  // dp[i] 表示达到i位置时剩余的跳力，若到达某个位置时跳力为负了，说明无法到达该位置
    // 所以当前位置的剩余跳力（dp 值）和当前位置新的跳力中的较大那个数决定了当前能到的最远距离，而下一个位置的剩余跳力（dp 值）就等于当前的这个较大值减去1
    vector<int> dp(nums.size(), 0);
    for (int i = 1; i < nums.size(); ++i)
    {
        dp[i] = max(dp[i - 1], nums[i - 1]) - 1;
        if (dp[i] < 0)
            return false;
    }
    return true;
}
```

91. 解码方法 #todo 20210415

```
int numDecodings(string s)
{
    int n = s.size();
    if (n <= 0  || s[0] == '0')
        return 0;
    // dp[i] 表示前i个数字的组合数目
    int dp[n+1] = {0};
    dp[0] = 1;
    dp[1] = s[0] != '0' ? 1 : 0;
    for(int i = 2; i <= n; i++)
    {
        int d1 = s[i-1] - '0';
        int d2 = s[i-2] - '0';
        int num = d2*10 + d1;
        if(!(d1 >= 1 && d1 <= 9) && (num >= 10 && num <= 26))
        {
           dp[i] = dp[i-2];
        }
        if (!(num >= 10 && num <= 26) && (d1 >= 1 && d1 <= 9))
        {
            dp[i] = dp[i-1];
        }
        if((d1 >= 1 && d1 <= 9) && (num >= 10 && num <= 26))
        {
            dp[i] = dp[i-1] + dp[i-2];
        }
    }
    return dp[n];
}
```

132. 分割回文串 II

```
// 解法1:
int minCut(string s)
{
    if (s.empty())
        return 0;
    int n = s.size();
    //p[i][j] 表示区间 [i, j] 内的子串是否为回文串,
    vector<vector<bool>> p(n, vector<bool>(n));
    // dp[i]表示子串 [0...i] 范围内的最小分割数
    vector<int> dp(n);
  // 两个for循环 子串
    for (int j = 0; j < n; j++)
    {
        dp[j] = j;
        for(int i = 0; i <= j; i++)
        {
            p[i][j] = s[i] == s[j] && (j - i < 2 || p[i+1][j-1]);

            if (p[i][j])
            {
                dp[j] = (i == 0) ? 0 : min(dp[i-1] + 1, dp[j]);
            }
        }
    }
    return dp[n-1];
}


// 解法2:
int minCut_2(string s)
{
    int n = s.size();
    if (n <= 0)
        return 0;

    // dp[i]表示s[i...n-1]的最小分割次数
    vector<int> dp(n + 1, 0);
    dp[n] = -1;
    vector<vector<bool>> p(n, vector<bool>(n, false));

    for (int i = n - 1; i >= 0; i--)
    {
        dp[i] = INT_MAX;
        for (int j = i; j < n; j++)
        {
            if (s[i] == s[j] && (j - i < 2 || p[i + 1][j - 1])) // 判断s[i...j]是不是回文
子串
            {
                p[i][j] = true;
                dp[i] = min(dp[j + 1] + 1, dp[i]);
            }
        }
    }
    return dp[0];
}
```

139. 单词拆分 #TODO 20210415

```
bool wordBreak(string s, vector<string> &wordDict)
{
    if (wordDict.size() == 0)
        return false;

    int n = s.size();
    // bool dp[n+1];
    vector<bool> dp(n + 1, false);
    dp[0] = true;
    //其中 dp[i] 表示子串 s[0...i-1] 内的子串是否可以拆分
    // 子串
    for (int i = 1; i <= n; i++)
    {
        for (int j = 0; j < i; j++)
        {
            if (dp[j] && count(wordDict.begin(), wordDict.end(), s.substr(j, i - j)) !=
0)
            {
                dp[i] = true;
                break;
            }
        }
    }
    return dp[n];
}
```

198. 打家劫舍

难度中等1395

```
int rob(vector<int> &nums)
{
    int n = nums.size();
    if (n <= 0)
        return 0;
    if (n == 1)
    {
        return nums[0];
    }
    if (n == 2)
    {
        return max(nums[0], nums[1]);
    }
    int dp[n] = {0};
    dp[0] = nums[0];
    dp[1] = max(nums[0], nums[1]);
    for (int i = 2; i < n; i++)
    {
        dp[i] = max(dp[i - 2] + nums[i], dp[i - 1]);
    }
    return dp[n - 1];
}
```

279. 完全平方数

```
int numSquares(int n)
{
    if (n <= 0)
        return 0;
    // dp[i] 表示数字i最少有几个平方数的和
    vector<int> dp(n + 1, INT_MAX);
    dp[0] = 0;
    for (int i = 1; i <= n; i++)
    {
        for (int j = 1; j * j <= i; j++)
        {
            dp[i] = min(dp[i], dp[i - j * j] + 1);
        }
    }
    return dp[n];
}
```

300. 最长递增子序列

难度中等1533

```
int lengthOfLIS(vector<int>& nums)
{
    if (nums.empty())
        return 0;

    vector<int> dp(nums.size()+1, 1);
    dp[0] =0;
    int res= INT_MIN;
    for(int i = 1; i <= nums.size(); i++)
    {
        // dp[i] = 1;
        for(int j = 1; j < i; j++)
        {
            if(nums[i-1] > nums[j-1])
            {
                dp[i] = max(dp[i], dp[j]+1);
            }
        }
        res = max(res, dp[i]);
    }
    return res;
}
```

312. 戳气球

```
int maxCoins(vector<int>& nums)
{
    int n = nums.size();
    nums.insert(nums.begin(), 1);
    nums.push_back(1);
    //  dp，其中 dp[i][j] 表示打爆区间 [i,j] 中的所有气球能得到的最多金币
    // dp[i][j] 的值，这个区间可能比较大，但是如果知道了所有的小区间的 dp 值，然后聚沙成塔，逐步的就能推出大区间的 dp 值了。还是要遍历这个区间内的每个气球，就用k来遍历吧，k在区间 [i, j] 中，假如第k个气球最后被打爆，那么此时区间 [i, j] 被分成了三部分，[i, k-1]，[k]，和 [k+1, j]，只要之前更新过了 [i, k-1] 和 [k+1, j] 这两个子区间的 dp 值，可以直接用 dp[i][k-1] 和 dp[k+1][j]，那么最后被打爆的第k个气球的得分该怎么算呢，你可能会下意识的说，就乘以周围两个气球被 nums[k-1] * nums[k] * nums[k+1]，但其实这样是错误的，为啥呢？dp[i][k-1] 的意义是什么呢，是打爆区间 [i, k-1] 内所有的气球后的最大得分，此时第 k-1 个气球已经不能用了，同理，第 k+1 个气球也不能用了，相当于区间 [i, j] 中除了第k个气球，其他的已经爆了，那么周围的气球只能是第 i-1 个，和第 j+1 个了，所以得分应为 nums[i-1] * nums[k] * nums[j+1]，分析到这里，状态转移方程应该已经跃然纸上了吧，如下所示：
//dp[i][j] = max(dp[i][j], nums[i - 1] * nums[k] * nums[j + 1] + dp[i][k - 1] + dp[k + 1][j])                 ( i ≤ k ≤ j )
    vector<vector<int>> dp(n + 2, vector<int>(n + 2, 0));
    for (int len = 1; len <= n; ++len) {
        for (int i = 1; i <= n - len + 1; ++i) {
            int j = i + len - 1;
            for (int k = i; k <= j; ++k) {
                dp[i][j] = max(dp[i][j], nums[i - 1] * nums[k] * nums[j + 1] + dp[i][k
- 1] + dp[k + 1][j]);
            }
        }
    }
    return dp[1][n];
}
```

343. 整数拆分 #todo 20210415

```
int cuttingRope(int n)
{
    if (n <= 2)
        return 1;

    // dp[i] 表示分拆数字i能得到的最大乘积
    vector<int> dp(n+1, 0);

    for(int i = 3; i <= n; i++)
    {
        for (int j = 0; j < i; j++)
        {
            dp[i] = max(max(dp[i], j*dp[i-j]), j*(i-j)); //dp[i] = max(dp[i], j*dp[i-
j],j*(i-j) )
        }
    }
    return dp[n];
}
```

3.双序列动态规划

状态: f[i][j]表示第一个sequence的前i个数字/字符, 配上第二个sequence的前j个;

方程: f[i][j] = 研究第i个和第j个的匹配关系;

初始化: f[i][0]和f[0][i];

答案: f[n][m], 其中n = s1.length(); m = s2.length();

10. 正则表达式匹配

```
bool isMatch(string s, string p)
{
    int m = s.length();
    int n = p.length();
    // dp[i][j] 表示 s[0...i-1]和p[0...j-1] 匹配
    vector<vector<bool>> dp(m + 1, vector<bool>(n + 1, false));

    dp[0][0] = true;
    for (int i = 1; i <= m; i++)
        dp[i][0] = false;

    for(int j = 1; j <= n; j++)
    {
        dp[0][j] = j > 1 && '*' == p[j - 1] && dp[0][j - 2];
    }

    for(int i = 1; i <= m; i++)
    {
        for(int j = 1; j <= n; j++)
        {
            if (p[j-1] != '*') // 如果p[j-1]的位置不是*的情况下， 如果s[i-1] == p[j-1] ||
p[j-1] == '.', 则dp[i][j] 匹配
                dp[i][j] = dp[i-1][j-1] && (s[i-1] == p[j-1] || p[j-1] == '.');
            else  // // (s[i-1] == p[j-2] || '.' == p[j-2]) && dp[i-1][j]; 可以认为p[j-1]
直接丢掉
            {
                dp[i][j] = dp[i][j-2] || (s[i-1] == p[j-2] || '.' == p[j-2]) && dp[i-1]
[j];
            }
        }
    }
    return dp[m][n];
}
```

44. 通配符匹配

难度困难662

```
bool isMatch(string s, string p)
{
    int m = s.size();
    int n = p.size();

    // dp[i][j] 表示s[0...i-1] 和 p[0...j-1]是否能够匹配
    // vector<vector<bool>> dp(m+1, vector<bool>(n+1, false));
    vector<vector<bool>> dp(m + 1, vector<bool>(n + 1, false));
    dp[0][0] = true; // 空串能匹配

    for (int j = 1; j <= n; j++)
    {
        if (p[j - 1] == '*')
            dp[0][j] = dp[0][j - 1];
    }
    // * 可以匹配空串和任意字符串
    // 如果我们不使用这个星号，那么就会从 dp[i][j-1] 转移而来；如果我们使用这个星号，那么就会从
dp[i-1][j]dp[i-1][j] 转移而来。

    // dp[i-1][j] 表示 s[0...i-2] 和p[0...j-1]匹配成功，  因为星号可以匹配任意字符串，再多加一个任
意字符也没问题
    // dp[i][j-1] 表示 s[0...i-1] 和p[0...j-2]匹配成功，星号可以匹配空串
    for (int i = 1; i <= m; i++)
    {
        for (int j = 1; j <= n; j++)
        {
            if (p[j - 1] == '*')
                dp[i][j] = dp[i - 1][j] || dp[i][j - 1];
            else
            {
                dp[i][j] = dp[i - 1][j - 1] && (s[i - 1] == p[j - 1] || p[j - 1] ==
'?');
            }
        }
    }
    return dp[m][n];
}
```

72. 编辑距离 #todo 增删改对应的到底是哪个

```
int minDistance(string word1, string word2)
{
    int m = word1.size();
    int n = word2.size();
    // dp[i][j] 表示word1[0...i-1] 变换到word2[0...j-1]所需要的最小步骤
    vector<vector<int>> dp(m + 1, vector<int>(n + 1, 0)); // 二维dp数组初始化大小为[m+1]
[n+1] 是为了初始化第0行和第0列

   for(int i = 0; i <= m; i++)
        dp[i][0] = i;
   for(int j = 0; j <=n; j++)
      dp[0][j] = j;

    for (int i = 1; i <= m; i++)
    {
        for (int j = 1; j <= n; j++)
        {
            if (word1[i - 1] == word2[j - 1])
            {
                dp[i][j] = dp[i - 1][j - 1];
            }
            else
            {
                dp[i][j] = min(dp[i - 1][j - 1], min(dp[i - 1][j], dp[i][j - 1])) + 1;
//dp[i-1][j-1] 表示替换操作，dp[i-1][j] 表示删除操作，dp[i][j-1] 表示插入操作。
            }
        }
    }
    return dp[m][n];
}
```

97. 交错字符串

```
bool isInterleave(string s1, string s2, string s3)
{
    int m = s1.size();
    int n = s2.size();
    int k = s3.size();

    if (m + n != k)
        return false;

    // dp[i][j] 表示s1[0...i-1] 和s2[0...j-1]能否交替表示成s3[0...i+j-1]
    vector<vector<bool>> dp(m + 1, vector<bool>(n + 1, false));
    dp[0][0] = true;
    for (int i = 1; i <= m; i++)
    {
        dp[i][0] = dp[i - 1][0] & (s1[i - 1] == s3[i - 1]);
    }

    for (int j = 1; j <= n; j++)
    {
        dp[0][j] = dp[0][j - 1] & (s2[j - 1] == s3[j - 1]);
    }

    for (int i = 1; i <= m; i++)
    {
        for (int j = 1; j <= n; j++)
        {
            if ((s1[i - 1] == s3[i + j - 1] && dp[i - 1][j]) || (s2[j - 1] == s3[i + j
- 1] && dp[i][j - 1]))
                dp[i][j] = true;
        }
    }
    return dp[m][n];
}
```

115. 不同的子序列

```
int numDistinct(string s, string t)
{
    int m = s.size();
    int n = t.size();

    // dp[i][j]表示S[0...j-1]中的子序列等于T[0..i-1]
    // 1 如果s[i-1]!=t[j-1] 则s[:i-1]中匹配t[:j-1]子序列个数==s[:i-2]中匹配t[:j-1]子序列个数
    //     dp[i][j] = dp[i-1][j]
    // 2 if s[i-1]==t[j-1]:
    //     dp[i][j] = dp[i-1][j-1] + dp[i-1][j]
    //     2.1 用s[i-1]     dp[i-1][j-1] 则s[:i-1]中匹配t[:j-2]子序列个数 ==s[:i-1]中匹配
t[:j-1]子序列个数
    //     2.2 不用s[i-1]   dp[i-1][j]   则s[:i-2]中匹配t[:j-1]子序列个数  ==s[:i-1]中匹配
t[:j]子序列个数

    vector<vector<long long>> dp(m + 1, vector<long long>(n + 1, 0));

    for (int j = 0; j <= m; j++)
    {
        dp[j][0] = 1;
    }

    for (int i = 1; i <= m; i++)
    {
        for (int j = 1; j <= n; j++)
        {
            if (s[i - 1] != t[j - 1])
                dp[i][j] = dp[i - 1][j];
            else
                dp[i][j] = dp[i - 1][j - 1] + dp[i - 1][j];
        }
    }
    return dp[m][n];
}
```

712. 两个字符串的最小ASCII删除和

```
int minimumDeleteSum(string s1, string s2)
{
    int m = s1.size();
    int n = s2.size();

    //dp[i][j]表示s1[0...i-1]和s2[0...j-1]需要的最小cost
    vector<vector<int>> dp(m+1, vector<int>(n+1, 0));

    for(int i = 1; i <= m; i++)
    {
        dp[i][0] = dp[i-1][0] + s1[i-1];
    }

    for(int j = 1; j <= n; j++)
    {
        dp[0][j] = dp[0][j-1] + s2[j-1];
    }

    for(int i = 1; i <= m; i++)
    {
        for(int j = 1; j <= n; j++)
        {
            if (s1[i-1] == s2[j-1]) // 表示不需要删除
                dp[i][j] = dp[i-1][j-1];
            else
            {
                dp[i][j] = min(dp[i-1][j] + s1[i-1], dp[i][j-1] + s2[j-1]);
            }
        }
    }
    return dp[m][n];
}
```

1143. 最长公共子序列

```
int longestCommonSubsequence(string word1, string word2)
{
    int m = word1.size();
    int n = word2.size();

    if (m == 0 && n == 0)
        return 0;

    // dp[i][j] 表示word1[0...i-1]和word2[0...j-1]上的最长公共子序列长度
    // 这里dp数组初始化长度为m+1,n+1,为了初始化方便考虑第0行和第0列
    vector<vector<int>> dp(m + 1, vector<int>(n + 1, 0));
    for (int j = 0; j <= n; j++)
    {
        dp[0][j] = 0;
    }
    for (int i = 0; i <= m; i++)
    {
        dp[i][0] = 0;
    }

    for (int i = 1; i <= m; i++) // 循环就得下标1开始
    {
        for (int j = 1; j <= n; j++)
        {
            if (word1[i - 1] == word2[j - 1])
                dp[i][j] = dp[i - 1][j - 1] + 1;
            else
                dp[i][j] = max(dp[i - 1][j], dp[i][j - 1]);
        }
    }
    return dp[m][n];
}
```

4.划分型动态规划

状态: f[i]表示前i个元素的最大值; 方程: f[i] = 前i个元素里面选一个区间的最大值; 初始化: f[0]; 答案: f[n - 1]

[188. Best Time to Buy and Sell Stock IV]

5.背包型动态规划

特点: 1). 用值作为DP维度, 2). DP过程就是填写矩阵, 3). 可以滚动数组优化 状态: f[i][S]前i个物品, 取出一些能否组成和为S; 方程: f[i][S] = f[i-1][S-a[i]] or f[i-1][S]; 初始化: f[i][0]=true; f[0][1...target]=false; 答案: 检查所有f[n][j]

[279. Perfect Squares]

```
int numSquares(int n)
{
    if (n <= 0)
        return 0;

    // dp[i] 表示数字i最少有几个平方数的和
    vector<int> dp(n+1, INT_MAX);
    dp[0] = 0;
    for(int i = 1; i <= n; i++)
    {
        for(int j = 1; j * j <= i; j++)
        {
            dp[i] = min(dp[i], dp[i-j*j]+1); //这里有两种选择，第j个要不要
        }
    }

    return dp[n];
}
```

[322. Coin Change]

```
int coinChange(vector<int> &coins, int amount)
{
    // int dp[amount+1] = {amount+1};
    // dp[i] 表示钱数为i时的最小硬币数的找零，注意由于数组是从0开始的，所以要多申请一位，数组大小为
amount+1，这样最终结果就可以保存在 dp[amount] 中了
    //vector<int> dp(amount + 1, amount + 1);
    vector<int> dp(amount + 1, INT_MAX-1);//用这种初始化的方式要好理解点
    int size = coins.size();
    dp[0] = 0;
    for (int i = 1; i <= amount; i++)
    {
        for (int j = 0; j < size; j++)
        {
            if (coins[j] <= i)
            {
                dp[i] = min(dp[i], dp[i - coins[j]] + 1); //这里有两种选择，第j个硬币要不要
            }
        }
    }
    return dp[amount] > amount ? -1 : dp[amount];
}
```

518. Coin Change 2 #todo 空间优化 377

```
int change(int amount, vector<int> &coins)
{
    //dp[i][j] 表示用前i个硬币组成钱数为j的不同组合方法
    vector<vector<int>> dp(coins.size() + 1, vector<int>(amount + 1, 0));
    dp[0][0] = 1;
    // 采用的方法是一个硬币一个硬币的增加，每增加一个硬币，都从1遍历到 amount，对于遍历到的当前钱数j，组成方法就是不加上当前硬币的拼法 dp[i-1][j]，还要加上去掉当前硬币值的钱数的组成方法
    for (int i = 1; i <= coins.size(); ++i)
    {
        for (int j = 0; j <= amount; ++j)
        {
            if(j >= coins[i - 1])
                dp[i][j] = dp[i - 1][j] +  dp[i][j - coins[i - 1]]; // 第i个硬币有 使用和不使用两种情况
            else
                dp[i][j] = dp[i - 1][j];
        }
    }
    return dp[coins.size()][amount];
}
```

416. Partition Equal Subset Sum # todo 空间优化

```
// 解法一：需要空间优化
bool canPartition(vector<int>& nums)
{
    int sum = 0;
    for(int i = 0; i < nums.size(); i++)
    {
        sum += nums[i];
    }
    int target  = sum / 2;
    if (sum % 2 == 1)
        return false;
    // 状态定义：dp[i][j]表示从数组的 [0, i] 这个子区间内挑选一些正整数，每个数只能用一次，使得这些数的和恰好等于 j。
    // 状态转移方程：很多时候，状态转移方程思考的角度是「分类讨论」，对于「0-1 背包问题」而言就是「当前考虑到的数字选与不选」。
    //    不选择 nums[i]，如果在 [0, i - 1] 这个子区间内已经有一部分元素，使得它们的和为 j ，那么 dp[i][j] = true；
    //    选择 nums[i]，如果在 [0, i - 1] 这个子区间内就得找到一部分元素，使得它们的和为 j - nums[i]。
    int n = nums.size();
    vector<vector<bool>>  dp(n + 1, vector<bool>(target + 1, false));
    // base case
    for (int i = 0; i <= n; i++)
        dp[i][0] = true;

    for (int i = 1; i <= n; i++) {
        for (int j = 0; j <= target; j++) {
            if (j - nums[i - 1] < 0) {
               // 背包容量不足，不能装入第 i 个物品
                dp[i][j] = dp[i - 1][j];
            } else {
                // 装入或不装入背包
                dp[i][j] = dp[i - 1][j] || dp[i - 1][j-nums[i-1]];
            }
        }
    }
    return dp[n][target];
}

bool canPartition(vector<int> &nums)
{
    int sum = 0;
    for (int i = 0; i < nums.size(); i++)
    {
        sum += nums[i];
    }
    if (sum % 2 == 1)
        return false;
    int targetSum = sum / 2;
    // dp[i] 表示原数组是否可以取出若干个数字，其和为i
    vector<bool> dp(targetSum + 1, false);
    dp[0] = true;
    for (int i = 1; i < nums.size(); i++)
    {
        for (int j = targetSum; j > 0; j--)
        {
            if (j >= nums[i])
            {
                dp[j] = dp[j] || dp[j - nums[i]]; // 两种情况 分别是使用当前数字nums[i] 和不
使用当前数字nums[i]
            }
        }
    }
    return dp[targetSum];
}

/* 计算 nums 中有几个子集的和为 sum */
```

474. Ones and Zeroes

```
int findMaxForm(vector<string> &strs, int m, int n)
{
    //dp[i][j]表示有i个0和j个1时能组成的最多字符串的个数
    vector<vector<int>> dp(m + 1, vector<int>(n + 1, 0));
    for (string str : strs)
    {
        int zeros = 0, ones = 0;
        for (char c : str)
            (c == '0') ? ++zeros : ++ones;
        for (int i = m; i >= zeros; --i)
        {
            for (int j = n; j >= ones; --j)
            {
                dp[i][j] = max(dp[i][j], dp[i - zeros][j - ones] + 1);
            }
        }
    }
    return dp[m][n];
}
```

494. Target Sum

```
class Solution {
public:
    /*
    sum(A) - sum(B) = target
    sum(A) = target + sum(B)
    sum(A) + sum(A) = target + sum(B) + sum(A)
    2 * sum(A) = target + sum(nums)

    int n = nums.size();
    vector<vector<bool>>  dp(n + 1, vector<bool>(target + 1, 0));
    // base case
    for (int i = 0; i <= n; i++)
        dp[i][0] = 1;

    for (int i = 1; i <= n; i++) {
        for (int j = 0; j <= target; j++) {
            if (j - nums[i - 1] < 0) {
               // 背包容量不足，不能装入第 i 个物品
                dp[i][j] = dp[i - 1][j];
            } else {
                // 装入或不装入背包
                dp[i][j] = dp[i - 1][j] + dp[i - 1][j-nums[i-1]];
            }
        }
    }
    return dp[n][target];
}




    */
    int findTargetSumWays(vector<int>& nums, int S) {

    // dp[i][j] 选前i个数 组成和为j的方法 思路有了 实际代码写不出来
    // dp[i][j] = dp[i-1][j-num[i]] + dp[ i - 1 ][ j + nums[ i ] ]
    vector<vector<bool>> dp(nums.size(), vector<bool>(S+1));
    dp[0][0]=true;
    if (nums[0] <= S)  // 第一行
    {
        dp[0][nums[0]] = 1;
    }
    for (int i = 1; i < nums.size(); i++)
    {
        for (int j = 0; j <= S; j++)
        {
            if (j-num[i] > 0 &&  j + nums[i] < S  )
                dp[i][j] =dp[i-1][j-num[i]] + dp[ i - 1 ][ j + nums[ i ] ]
            else
                dp[i][j] = dp[i - 1][j];
        }
    }
    return dp[nums.size() - 1][target];



    }
};

class Solution {
private:
    int targetSum(vector<int> & nums, int sum)
    {
        // int dp[sum+1] = {0};
        vector<int> dp(sum+1, 0);
        dp[0] = 1;
        for(int i = 0; i < nums.size(); i++)
        {
            for(int j = sum; j >= nums[i]; j--)
            {
                dp[j] += dp[j-nums[i]];
            }
        }
        return dp[sum];


    }
public:
    int findTargetSumWays(vector<int>& nums, int target)
    {
        // sum(A) - sum(B) = target
        // sum(A) = target + sum(B)
        // sum(A) + sum(A) = target + sum(B) + sum(A)
        // 2 * sum(A) = target + sum(nums)
        int sum = accumulate(nums.begin(), nums.end(), 0);
        return sum < target || (target + sum) % 2 == 1 ? 0 : targetSum(nums, (target +
sum) / 2);
    }
};
```

6.区间型动态规划

特点: 1). 求一段区间的解max/min/count; 2). 转移方程通过区间更新; 3). 从大到小的更新; 这种题目共性就是区间最后求[0, n-1]这样一个区间逆向思维分析, 从大到小就能迎刃而解

区间 DP 是在一个区间上进行的一系列的动态规划，在一个线性的数据上对区间进行状态转移，dp[i][j]表示i到j的区间。dp[i][j]可以由子区间的状态转移而来，关键是 dp[i][j]表示什么，然后找 dp[i][j]和子区间的关系

[5. 最长回文子串](#)

```
string longestPalindrome(string s) // todo: 时间上还得优化
{
    if (s.empty())
        return "";
    int len = 0;  // 记录最长回文子串的长度
    int left = 0, right = 0; // 记录最长回文子串的左右边界
    // dp[i][j] 表示 s[i...j]上是否为回文子串
    vector<vector<bool>> dp(s.size(), vector<bool>(s.size(), false));、
    for (int i = 0; i < s.size(); i++)
    {
        for (int j = 0; j <= i; j++)
        {
            dp[j][i] = s[i] == s[j] && ( i - j < 2 || dp[j+1][i-1]);
            if (dp[j][i] && i - j + 1 > len)
            {
                len = i - j +1;
                left = j;
                right = i;
            }
        }
    }
    return s.substr(left,right - left + 1);
}

int getLongestPalindrome(string str, int n) {
    // write code here
    if (str.empty() || n <= 0)
        return 0;


    // dp[i][j]表示区间[i...j]上是否是回文子串
    vector<vector<bool>> dp(str.size(), vector<bool>(str.size(), false));
    int max_len = 0;
    int max_left = 0;
    int max_right = 0;

    for(int i = 0; i < n; i++)
    {
        for(int j = 0; j <= i; j++)
        {
            dp[j][i] = str[i] == str[j] && ( i - j < 2 || dp[j+1][i-1]);
            if (dp[j][i] && i-j+1 > max_len)
            {
                max_len = i - j +1;
            }

        }
    }

    return max_len;

}
```

131. 分割回文串

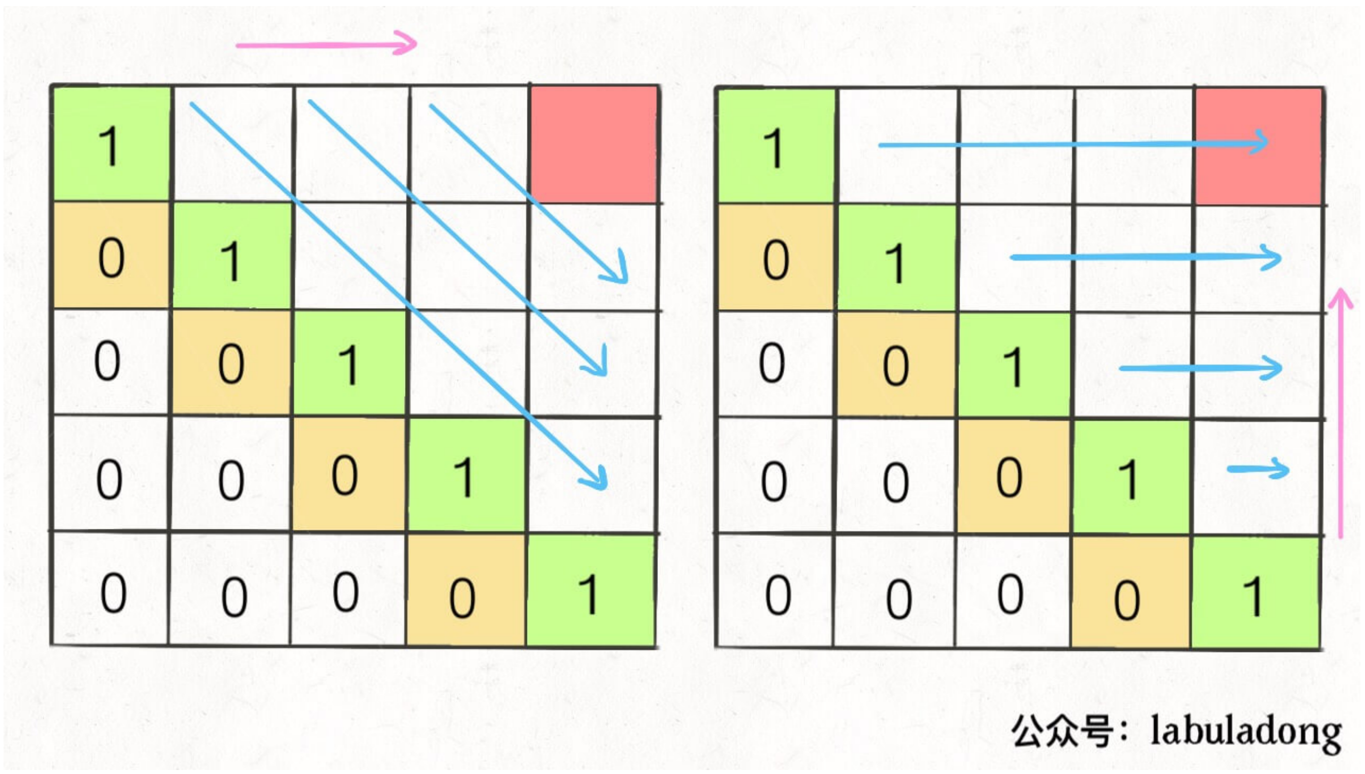
```
bool isPalindrome(const string& s, int start, int end)
{
    while(start <= end) {
        if(s[start++] != s[end--])
            return false;
    }
    return true;
}
void dfs(string &s, vector<string> &temp, int start,vector<vector<string>> &res)
{
    if(start == s.size())
    {
        res.push_back(temp);
        return;
    }

    for(int i = start; i < s.size(); i++)
    {
        if (isPalindrome(s, start, i))
        {
            temp.push_back(s.substr(start, i - start +1)); // index
            dfs(s, temp, i+1, res);
            temp.pop_back();
        }
    }
}

vector<vector<string>> partition(string s)
{
    vector<vector<string> > res;
    if(s.empty()) return res;

    vector<string> temp;
    dfs(s, temp, 0, res);

    return res;
}
```

132. 分割回文串 II

```
class Solution
{
public:
     // 解法1:
    int minCut(string s)
    {
        if (s.empty())
```

```
            return 0;
        int n = s.size();
        //p[i][j] 表示区间 [i, j] 内的子串是否为回文串,
        vector<vector<bool>> p(n, vector<bool>(n, false));
        // dp[i]表示子串 [0, i] 范围内的最小分割数
        vector<int> dp(n);
        for (int i = 0; i < n; ++i)
        {
            dp[i] = i;
            for (int j = 0; j <= i; ++j)
            {
                if (s[i] == s[j] && (i - j < 2 || p[j + 1][i - 1]))
                {
                    p[j][i] = true;
                    dp[i] = (j == 0) ? 0 : min(dp[i], dp[j - 1] + 1);
                }
            }
        }
        return dp[n - 1];
    }

    // 解法2:
    int minCut_2(string s)
    {
        int n = s.size();
        if (n <= 0)
            return 0;

        // dp[i]表示s[i...n-1]的最小分割次数
        vector<int> dp(n + 1, 0);
        dp[n] = -1;
        vector<vector<bool>> p(n, vector<bool>(n, false));

        for (int i = n - 1; i >= 0; i--)
        {
            dp[i] = INT_MAX;
            for (int j = i; j < n; j++)
            {
                if (s[i] == s[j] && (j - i < 2 || p[i + 1][j - 1])) // 判断s[i...j]是不是
回文子串
                {
                    p[i][j] = true;
                    dp[i] = min(dp[j + 1] + 1, dp[i]);
                }
            }
        }
        return dp[0];
    }
};
```

516. 最长回文子序列 后面两题 逆向遍历 为什么？？？ 重点是画图



```
int longestPalindromeSubseq(string s)
{
    int n = s.size();
    // dp[i][j]表示[i,j]区间内的字符串的最长回文子序列
    vector<vector<int>> dp(n, vector<int>(n));

    //如果s[i]==s[j]，那么i和j就可以增加2个回文串的长度，我们知道中间dp[i + 1][j - 1]的值，那么其加上2就是dp[i][j]的值。如果s[i] != s[j]，那么我们可以去掉i或j其中的一个字符，然后比较两种情况下所剩的字符串谁dp值大，就赋给dp[i][j]

    for (int i = 0; i < n; i++)
    {
        dp[i][i] = 1;
        for (int j = i-1; j >= 0; j--)
        {
            if (s[i] == s[j])
            {
                dp[j][i] = dp[j+1][i-1] + 2;
            }
            else
            {
                dp[j][i] = max(dp[j][i-1], dp[j+1][i]);
            }
        }
    }
    return dp[0][n-1];
}
```

647. 回文子串

```
int countSubstrings(string s)
{
    int n = s.size();
    if (n <= 0)
        return 0;
    int res = 0;
    // dp[i][j] 表示区间s[i...j]上是否为回文子串
    vector<vector<bool>> dp(n, vector<bool>(n, false));

    for (int i = 0; i < n; i++)
    {
        // dp[i][i] = true;
        for (int j = 0; j <= i; j++)
        {
            dp[j][i] = s[i] == s[j] && (i - j < 2 || dp[j + 1][i - 1]);
            if (dp[j][i])
                res++;
        }

    }
    return res;
}
```

7.博弈型动态规划状态

定义一个人的状态; 方程: 考虑两个人的状态做状态更新; 初始化: 暂无; 答案: 先思考最小状态, 再思考大的状态 -> 往小的递推, 适合记忆话搜索 动态规划, 循环(从小到大递推), 记忆化搜索(从大到小搜索, 画搜索树); 什么时候 用记忆化搜索: 1). 状态转移特别麻烦, 不是顺序性, 2). 初始化状态不是很容易找到; 题目类型: 1). 博弈类问题, 2). 区间类问题; 适合解决题目: 1). 状态特别复杂, 2). 不好初始化

486. Predict the Winner

```
class Solution
{
    // 作为先发者 在i...j范围上先发获得的收益
    int f(vector<int> &nums, int i, int j)
    {
        if (i == j) // 如果只有一个数并且又是先发者，则直接拿走该数
            return nums[i];

        else
            return max(s(nums, i + 1, j) + nums[i], s(nums, i, j - 1) + nums[j]);
    }
    // / 作为后发者 在i...j范围上后发获得的收益
    int s(vector<int> &nums, int i, int j)
    {
        if (i == j)
        {
            return 0;
        }
        else // 对方也是绝顶聪明,作为后发者,此时只能选先发者拿完之后 剩下最小的
            return min(f(nums, i + 1, j), f(nums, i, j - 1));
    }
public:
    bool PredictTheWinner(vector<int> &nums)
    {

        if (nums.empty())
            return false;
        int sum = 0;
        for (int i = 0; i < nums.size(); i++)
            sum += nums[i];

        int res = f(nums, 0, nums.size() - 1);
        return sum - res > res ? false : true;
    }
};
```

其他

221. 最大正方形

```
int maximalSquare(vector<vector<char>>& matrix)
{
    if (matrix.empty() || matrix[0].empty())
      return 0;
    int m = matrix.size(), n = matrix[0].size(), res = 0;
    // dp[i][j] 表示到达 (i, j) 位置所能组成的最大正方形的边长
    // p[i][j]，由于该点是正方形的右下角，所以该点的右边，下边，右下边都不用考虑，关心的就是左边，上边，和左上边。这三个位置的dp值 suppose 都应该算好的，还有就是要知道一点，只有当前 (i, j) 位置为1，dp[i][j] 才有可能大于0，否则 dp[i][j] 一定为0。当 (i, j) 位置为1，此时要看 dp[i-1][j-1]，dp[i][j-1]，和 dp[i-1][j] 这三个位置，我们找其中最小的值，并加上1，就是 dp[i][j] 的当前值了
    vector<vector<int>> dp(m, vector<int>(n, 0));
    for (int i = 0; i < m; ++i) {
        for (int j = 0; j < n; ++j) {
            if (i == 0 || j == 0) dp[i][j] = matrix[i][j] - '0';
            else if (matrix[i][j] == '1') {
                dp[i][j] = min(dp[i - 1][j - 1], min(dp[i][j - 1], dp[i - 1][j])) + 1;
            }
            res = max(res, dp[i][j]);
        }
    }
    return res * res;
}
```

分治

395. Longest Substring with At Least K Repeating Characters

深度优先搜索DFS

经典DFS

```
int dirs[8][2] = {1,1,1,0,1,-1,0,1,0,-1,-1,1,-1,0,-1,-1};

void dfs(const vector<vector<int> >& nums, vector<vector<bool> >& visit, int i, int j,
int& value)
{
    if ( nums[i][i] == 0 || visit[i][j])
        return;
    visit[i][j] = true;
    value += nums[i][j];
    for(int k = 0; k < 8; k++)
    {
        int x = i + dirs[k][0];
        int y = j + dirs[k][1];
        if (x < 0 || x > nums.size() || y < 0 || y > nums[0].size() || visit[x][y])
            continue;
        dfs(nums, visit, x, y, value);
    }
}
```

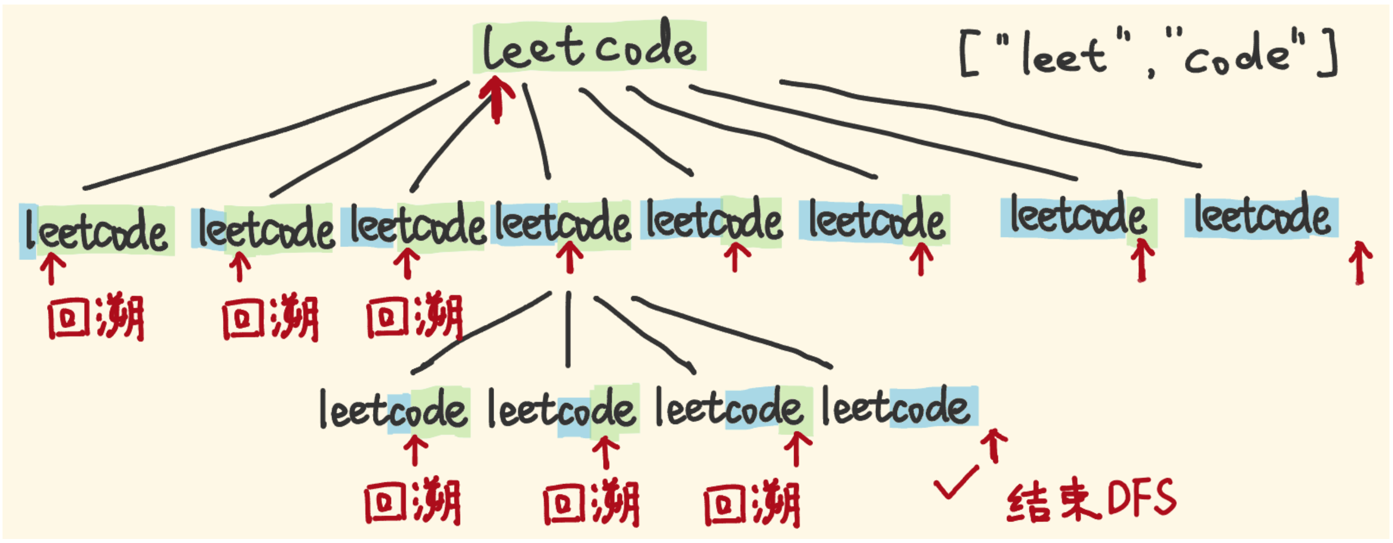
79. 单词搜索

```
class Solution
{
    bool dfs(vector<vector<char>> &board, string &word, int i, int j, int pos)
    {
        if (i >= board.size() || j >= board[0].size() || i < 0 || j < 0 || pos >=
word.size() || word[pos] != board[i][j])
            return false;
        if (pos == word.size() - 1 && word[pos] == board[i][j])
            return true;
    // 这个地方修改临时值和回溯思想不一样，只是为了不重复访问，需要一个和原数组等大小的 visited 数组，
是 bool 型的，用来记录当前位置是否已经被访问过，因为题目要求一个 cell 只能被访问一次
        char temp = board[i][j];
        board[i][j] = '0';
        bool flag = dfs(board, word, i, j + 1, pos + 1) ||
                    dfs(board, word, i, j - 1, pos + 1) ||
                    dfs(board, word, i + 1, j, pos + 1) ||
                    dfs(board, word, i - 1, j, pos + 1);
        board[i][j] = temp;
        return flag;
    }

public:
    bool exist(vector<vector<char>> &board, string word)
    {
        if (board.size() == 0)
            return false;
        for (int i = 0; i < board.size(); i++)
        {
            for (int j = 0; j < board[0].size(); j++)
            {
                if (dfs(board, word, i, j, 0))
                    return true;
            }
        }
        return false;
    }
};
```

130. 被围绕的区域

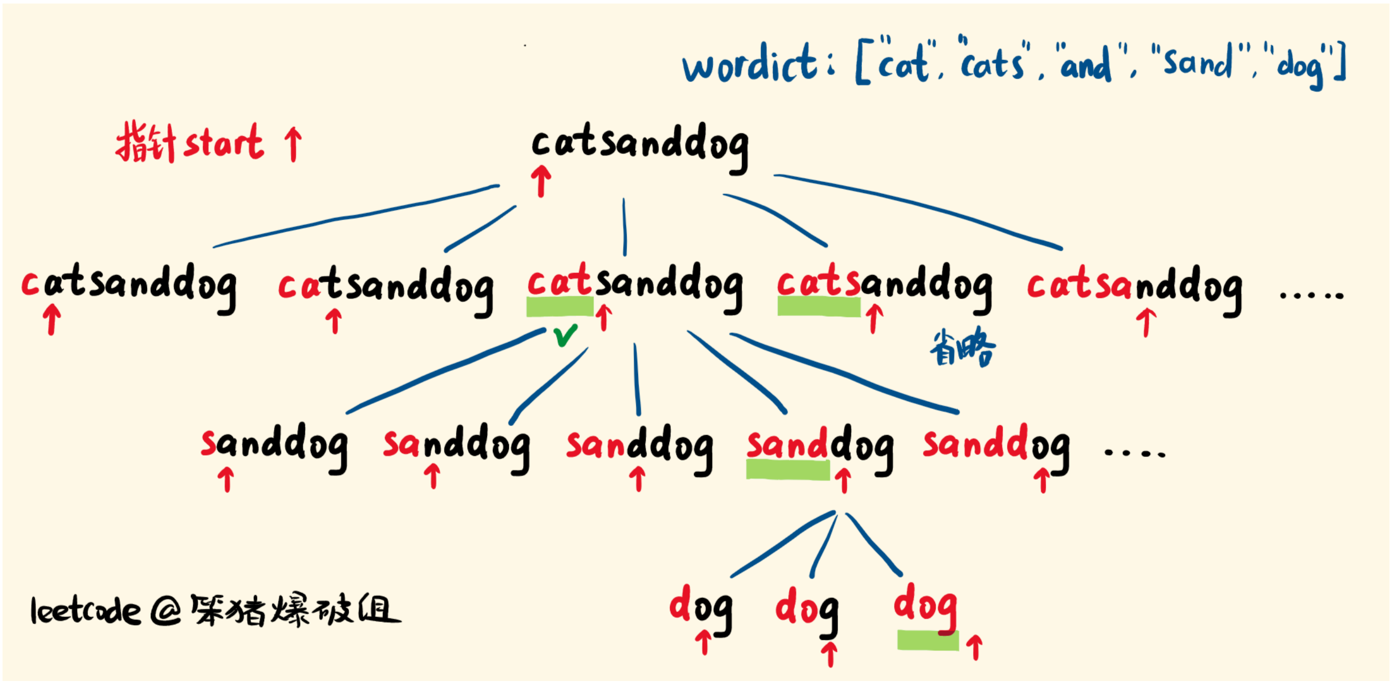
```
class Solution {
public:
   void solve(vector<vector<char> >& board)
   {

       if (board.empty() || board[0].empty())
           return;

       int m = board.size();
       int n = board[0].size();
       for (int i = 0; i < m;  ++i)
        {
            for (int j = 0; j < n; ++j)
            {
                // 从出边界出发，将边界上为O的并且连在一起的都变成$
                if ((i == 0 || i == m - 1 || j == 0 || j == n - 1) && board[i][j] ==
'O')
                    solveDFS(board, i, j, m, n);
            }
        }
       // 先将没有变成$也就是不和边界上的O连在一起的改为X,
        for (int i = 0; i < board.size(); ++i) {
            for (int j = 0; j < board[i].size(); ++j) {
                if (board[i][j] == 'O') board[i][j] = 'X';
                if (board[i][j] == '$') board[i][j] = 'O';
            }
        }
    }
    void solveDFS(vector<vector<char> > &board, int i, int j, int m, int n) {

        if (i >= m || i < 0 || j >= n || j < 0 || board[i][j] != 'O') return;

        board[i][j] = '$';
        solveDFS(board, i - 1, j, m, n);
        solveDFS(board, i, j + 1, m, n);
        solveDFS(board, i + 1, j, m, n);
        solveDFS(board, i, j - 1, m, n);

    }
};
```

139. 单词拆分 回溯+记忆化搜索



```
// 解法一：暴力递归超时了
bool dfs(string &s, unordered_set<string> &wordSet, int i, vector<int> &memo)
{
    if ( i >= s.size())
        return true;
    for(int j = i+1; j <= s.size(); j++)
    {
        if (wordSet.count(s.substr(i, j - i)) && dfs(s, wordSet, j, memo))
            return true;
    }
    return false;
}

bool wordBreak(string s, vector<string> &wordDict)
{
    unordered_set<string> wordSet(wordDict.begin(), wordDict.end());
    vector<int> memo(s.size(), -1);
    return dfs(s, wordSet, 0, memo);
}

// 解法一：增加记忆化搜索
bool wordBreak(string s, vector<string> &wordDict)
{
    unordered_set<string> wordSet(wordDict.begin(), wordDict.end());
    vector<int> memo(s.size(), -1);
    return dfs(s, wordSet, 0, memo);
}
// dfs 表示字符串s[start,n] 是否可分
bool dfs(string s, unordered_set<string> &wordSet, int start, vector<int> &memo)
{
    if (start >= s.size())
        return true;
    if (memo[start] != -1)
        return memo[start];
    for (int i = start + 1; i <= s.size(); ++i)
    {
        if (wordSet.count(s.substr(start, i - start)) && dfs(s, wordSet, i, memo))
        {
            return memo[start] = 1;
        }
    }
    return memo[start] = 0;
}
```

140. Word Break II 注意和139的区别


wordict: ["cat", "cats", "and", "sand", "dog"]
指针start ↑
catsanddog
catsanddog catsanddog catsanddog catsanddog catsanddog .....
省略
sanddog sanddog sanddog sanddog sanddog ....
dog dog dog
leetcode @笨猪爆破组

```
// 解法一 暴力递归超时了  想改成不带返回值得形式
vector<string> dfs(string s, unordered_set<string> &wordSet, int start)
```

```
{
    if (start == s.size())
    {
        return {""};
    }
    vector<string> res;
    for (int i = start; i < s.size(); ++i)
    {
        if (wordSet.count(s.substr(start, i - start+1)) > 0)
        {
            vector<string> tmp = dfs(s, wordSet, i+1);
            for (auto str : tmp)
            {
                res.push_back(s.substr(start, i - start + 1) + (str.empty() ? "" : " ")
+ str);
            }
        }
    }
    return res;
}

vector<string> wordBreak(string s, vector<string> &wordDict)
{
    unordered_set<string> wordSet(wordDict.begin(), wordDict.end());
    return dfs(s, wordSet, 0);
}

// 解法二：增加记忆化搜索
vector<string> dfs(string s, unordered_set<string> &wordSet, int start,
unordered_map<int, vector<string>> &memo)
{
    if (memo.count(start) > 0)
    {
        return memo[start];
    }
    if (start == s.size())
    {
        return {""};
    }
    vector<string> res;
    for (int i = start; i < s.size(); ++i)
    {
        if (wordSet.count(s.substr(start, i - start + 1)) > 0)
        {
            vector<string> tmp = dfs(s, wordSet, i+1, memo);
            for (auto str : tmp)
            {
                res.push_back(s.substr(start, i - start + 1) + (str.empty() ? "" : " ")
+ str);
```

```
            }
        }
    }
    memo[start] = res;
    return res;
}

vector<string> wordBreak(string s, vector<string> &wordDict)
{
    unordered_set<string> wordSet(wordDict.begin(), wordDict.end());
    unordered_map<int, vector<string>> memo;
    return dfs(s, wordSet, 0, memo);
}
```

200. Number of Islands

```
class Solution
{
    void dfs(vector<vector<char>> &grid, int i, int j, int m, int n)
    {
        if (i < 0 || i >= m || j < 0 || j >= n || grid[i][j] != '1')
            return;
        grid[i][j] = '2';
        dfs(grid, i + 1, j, m, n);
        dfs(grid, i - 1, j, m, n);
        dfs(grid, i, j + 1, m, n);
        dfs(grid, i, j - 1, m, n);
    }

public:
    int numIslands(vector<vector<char>> &grid)
    {
        if (grid.empty())
            return 0;
        int m = grid.size();
        int n = grid[0].size();
        int res = 0;
        for (int i = 0; i < m; i++)
        {
            for (int j = 0; j < n; j++)
            {
                if (grid[i][j] == '1')
                {
                    res++;
                    dfs(grid, i, j, m, n);
                }
            }
        }
        return res;
    }
};
```

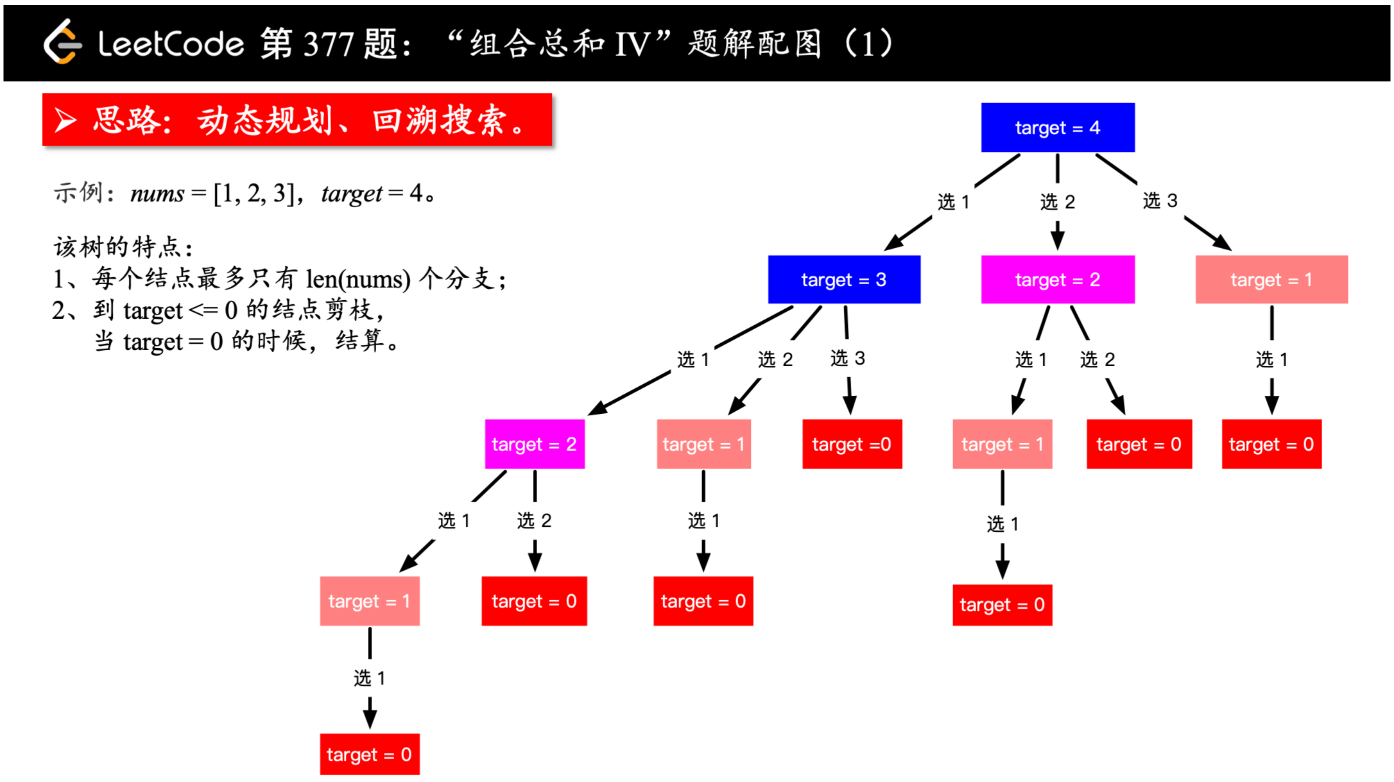
212. Word Search II dfs+字典树 #todo

```
class Solution {
public:
    struct TrieNode {
        TrieNode *child[26];
        string str;
        TrieNode() : str("") {
            for (auto &a : child) a = NULL;
        }
    };
    struct Trie {
        TrieNode *root;
        Trie() : root(new TrieNode()) {}
        void insert(string s) {
            TrieNode *p = root;
            for (auto &a : s) {
                int i = a - 'a';
                if (!p->child[i]) p->child[i] = new TrieNode();
                p = p->child[i];
            }
            p->str = s;
        }
    };
    vector<string> findWords(vector<vector<char>>& board, vector<string>& words) {
        vector<string> res;
        if (words.empty() || board.empty() || board[0].empty()) return res;
        vector<vector<bool>> visit(board.size(), vector<bool>(board[0].size(), false));
        Trie T;
        for (auto &a : words) T.insert(a);
        for (int i = 0; i < board.size(); ++i) {
            for (int j = 0; j < board[i].size(); ++j) {
                if (T.root->child[board[i][j] - 'a']) {
                    search(board, T.root->child[board[i][j] - 'a'], i, j, visit, res);
                }
            }
        }
        return res;
    }
    void search(vector<vector<char>>& board, TrieNode* p, int i, int j,
vector<vector<bool>>& visit, vector<string>& res) {
        if (!p->str.empty()) {
            res.push_back(p->str);
            p->str.clear();
        }
        int d[][2] = {{-1, 0}, {1, 0}, {0, -1}, {0, 1}};
        visit[i][j] = true;
        for (auto &a : d) {
            int nx = a[0] + i, ny = a[1] + j;
            if (nx >= 0 && nx < board.size() && ny >= 0 && ny < board[0].size() &&
!visit[nx][ny] && p->child[board[nx][ny] - 'a']) {
                search(board, p->child[board[nx][ny] - 'a'], nx, ny, visit, res);
            }
        }
        visit[i][j] = false;
    }
};
```

329. Longest Increasing Path in a Matrix 记忆化搜索

```
int dirs[4][2] = {0, 1, 1, 0, 0, -1, -1, 0};

// 表示从 (i, j) 出发的最长路径长度
int dfs(vector<vector<int>>& matrix, vector<vector<int>>& dp, int i, int j)
{

    if (dp[i][j] > 0)
        return dp[i][j];
    int res = 1;
    for(int k = 0; k < 4; k++)
    {
        int x = i + dirs[k][0];
        int y = j + dirs[k][1];
        if (x >= 0 && x < matrix.size() && y >= 0 && y < matrix[0].size() && matrix[x]
[y] > matrix[i][j])
        {
            int dist = 1 + dfs(matrix, dp, x, y);
            res = max(dist, res);
        }
    }
    dp[i][j] = res;
    return res;
}
int longestIncreasingPath(vector<vector<int>>& matrix)
{
    if (matrix.empty())
        return 0;
    // dp[i][j]表示数组中以(i,j)为起点的最长递增路径的长度，初始将dp数组都赋为0，当我们用递归调用
时，遇到某个位置(x, y), 如果dp[x][y]不为0的话，我们直接返回dp[x][y]即可，不需要重复计算。
    vector<vector<int> > dp( matrix.size(), vector<int>(matrix[0].size(), 0) );
    int longest = INT_MIN;
    for(int i = 0; i < matrix.size(); i++)
    {
        for(int j = 0; j < matrix[0].size(); j++)
        {
            int len = dfs(matrix, dp, i, j);
            longest = max(longest,len);
        }
    }
    return longest;
}
```

377. Combination Sum IV #todo 优化成动态规划


LeetCode 第 377 题：“组合总和 IV”题解配图（1）

➢ 思路：动态规划、回溯搜索。

示例：*nums* = [1, 2, 3]，*target* = 4。

该树的特点：
1、每个结点最多只有 len(nums) 个分支；
2、到 target <= 0 的结点剪枝，
   当 target = 0 的时候，结算。

```
// 记忆化搜索
int dfs(vector<int> &nums, int target, unordered_map<int, int> &memo)
{
    if(memo.count(target)) return memo[target];
    if (target < 0) return 0;
    if (target == 0) return 1;

    int res = 0;
    for(int i = 0; i < nums.size(); i++)
    {
        res += dfs(nums, target - nums[i], memo);
    }
    memo[target] = res;
    return res;
}
int combinationSum4(vector<int>& nums, int target)
{
    unordered_map<int, int> memo;
    return dfs(nums, target, memo);
}

// 优化成动态规划
int combinationSum4(vector<int>& nums, int target)
{
    vector<int> dp(target + 1);
    dp[0] = 1;
    for (int i = 1; i <= target; ++i) {
        for (auto a : nums) {
            if (i >= a) dp[i] += dp[i - a];
        }
    }
    return dp.back();
}
```

547. 省份数量

```
class Solution {
public:
    int findCircleNum(vector<vector<int>>& M)
  {
    int sum = 0;
    int m = M.size();
    vector<bool> visited(m, false);
    for (int i = 0; i < m; i++)
    {
      if (visited[i]) continue;
      dfs(M, i, visited);
      ++sum;
    }
    return sum;
  }

  void dfs(vector<vector<int>>& M, int i, vector<bool>& visited)
  {
    visited[i] = true;
    for (int j = 0; j < M.size(); j++)
    {
      if (M[i][j] == 0 || visited[j]) continue;
      dfs(M, j, visited);
    }
  }
};
```

576. Out of Boundary Paths 记忆化搜索

```
int dirs[4][2] = {0, 1, 1, 0, 0, -1, -1, 0};
int dfs(vector<vector<vector<uint>>> &dp, int x, int y, int step,int m, int n)
{
    if (x < 0 || y < 0 || x >= m || y >= n)    // 一旦超出边界直接返回1
        return 1;
    if (x-step >= 0 && x + step < m && y - step >= 0 && y + step < n)    // 不管从哪个方向
走step步之后 都到不了边界外
        return 0;
    if(step <= 0)  // 如果没得走了
        return 0;

    if (dp[step][x][y] > 0)
        return dp[step][x][y];
    int count = 0;
    for(int k = 0; k < 4; k++)
    {
        int i = x + dirs[k][0];
        int j = y + dirs[k][1];
        count += dfs(dp, i, j, step - 1, m, n);
        count %= 1000000007;
    }

    dp[step][x][y] = count;
    return count;
}
int findPaths(int m, int n, int N, int i, int j)
{
    // dp[k][i][j]表示总共走k步，从(i,j)位置走出边界的总路径数
    vector<vector<vector<uint>>> dp(N+1,vector<vector<uint>>(m,vector<uint>(n,0)));
    int count = dfs(dp, i, j, N, m, n) % 1000000007;
    return coun;
}
```

688. Knight Probability in Chessboard 记忆化搜索

```

vector<vector<int>> dirs{{-1,-2},{-2,-1},{-2,1},{-1,2},{1,2},{2,1},{2,-1},{1,-2}};
double dfs(vector<vector<vector<double>>> &dp, int i, int j, int k, int N)
{
    if (i < 0 || i >= N || j < 0 || j >=N)
        return 0.0;
    if (k == 0)
        return 1.0;

    if (dp[k][i][j] != 0.0)
        return dp[k][i][j];
    double count = 0.0;

    for (auto dir : dirs)
    {
        int x = i + dir[0];
        int y = j + dir[1];
        // if (x < 0 || x >= N || y < 0 || y >= N)
        //     continue;
        count += dfs(dp, x, y, k-1, N);;
    }
    dp[k][i][j] = count;
    return count;
}
double knightProbability(int N, int K, int r, int c)
{
    // dp[k][i][j]表示总共走k步，从(i,j)位置没有走出边界的总路径数
    if (K == 0)
        return 1;
    vector<vector<vector<double>>> dp(K+1,vector<vector<double>>(N,vector<double>(N,
0.0)));
    double total_step = dfs(dp, r, c, K, N);
    return dp[K][r][c] / pow(8, K);
}
```

827. Making A Large Island 类似于回溯

```
int dirs[4][2] = {0, 1, 1, 0, 0, -1, -1, 0};
int dfs(vector<vector<int>>& grid, vector<vector<bool>>& visited, int i, int j)
{
    int m = grid.size();
    int n = grid[0].size();
    if (i < 0 || i >= m || j < 0 || j >= n || visited[i][j] == true || grid[i][j] != 1
)
        return 0;
    visited[i][j]=true;
    int curSize = 1;

    for(int k = 0; k < 4; k++)
    {
        int x = i + dirs[k][0];
        int y = j + dirs[k][1];
        curSize +=  dfs(grid, visited, x, y);
   }
    return curSize;
    // return 1 + dfs(grid, visited, i+1,j) + dfs(grid, visited, i-1,j) + dfs(grid,
visited, i,j+1) + dfs(grid, visited, i,j-1);
}
int largestIsland(vector<vector<int>>& grid)
{
    if (grid.empty())
        return 0;
    int m = grid.size();
    int n = grid[0].size();
    if(grid==vector<vector<int>>(m,vector<int>(n,1))) return m*n;
    vector<vector<bool>> visited(m, vector<bool>(n, false));
    vector<vector<bool>> temp(m, vector<bool>(n, false));
    int res = 0;
    int maxSize = INT_MIN;
    for(int i = 0; i < m; i++)
    {
        for(int j = 0; j < n; j++)
        {
            if (grid[i][j] == 0)
            {
                grid[i][j] = 1;
                res = dfs(grid, visited, i, j);
                visited=temp;
                maxSize = max(res, maxSize);
                grid[i][j]=0;
            }
        }
    }
    return maxSize;
}
```

BFS

判断一棵二叉树是否完全二叉树 #TODO

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

    queue<TreeNode *> q;
    q.push(root);
    while(!q.empty())
    {
        root = q.front();
        q.pop();
        if (root->left && root->right)
        {
            q.push(root->left);
            q.push(root->right);
        }
        else if (root->left == nullptr && root->right)
            return false;
        else if (root->left && root->right == nullptr)
        {
            while(!q.empty())
            {
                root = q.front();
                q.pop();
                if (root->left || root->right)
                    return false;
            }
        }
    }
    return true;
}
```

102. Binary Tree Level Order Traversal

```
vector<vector<int>> levelOrder(TreeNode *root)
{
    vector<vector<int>> ret;
    if (root == NULL)
        return ret;

    queue<TreeNode *> q;
    q.push(root);
    while (!q.empty())
    {
        int size = q.size();
        vector<int> level;
        for (int i = 0; i < size; i++)
        {
            TreeNode *node = q.front();
            level.push_back(node->val);
            q.pop();
            if (node->left)
                q.push(node->left);
            if (node->right)
                q.push(node->right);
        }
        ret.push_back(level);
    }
    return ret;
}
```

103. Binary Tree Zigzag Level Order Traversal

```
//由于每层的结点数是知道的，就是队列的元素个数，所以可以直接初始化数组的大小,使用一个变量 leftToRight 来标记顺序，初始时是 true，当此变量为 true 的时候，每次加入数组的位置就是i本身，若变量为 false 了，则加入到 size-1-i 位置上，这样就直接相当于翻转了数组
vector<vector<int>> zigzagLevelOrder(TreeNode *root)
{
    vector<vector<int>> res;
    if (root == NULL)
        return res;
    queue<TreeNode *> q;
    q.push(root);
    bool leftToRight = true;
    while (!q.empty())
    {
        int size = q.size();
        vector<int> oneLevel(size); // 这个地方注意 要给定数组大小
        for (int i = 0; i < size; ++i)
        {
            TreeNode *t = q.front();
            q.pop();
            int idx = leftToRight ? i : (size - 1 - i);
            oneLevel[idx] = t->val;
            if (t->left)
                q.push(t->left);
            if (t->right)
                q.push(t->right);
        }
        leftToRight = !leftToRight;
        res.push_back(oneLevel);
    }
    return res;
}
```

111. 二叉树的最小深度

```
int minDepth(TreeNode *root)
{
    if (!root)
        return 0;

    queue<TreeNode *> q;
    q.push(root);
    int res = 1;
    while (!q.empty())
    {
        int size = q.size();
        for (int i = 0; i < size; i++)
        {
            root = q.front();
            q.pop();
            if (!root->left && !root->right)
                return res;
            if (root->left)
                q.push(root->left);

            if (root->right)
                q.push(root->right);
        }
        res++;
    }
    return -1;
}
```

116. 填充每个节点的下一个右侧节点指针

```
Node *connect(Node *root)
{
    if (!root) return NULL;
    queue<Node *> q;
    q.push(root);
    while (!q.empty())
    {
        int size = q.size();
        for (int i = 0; i < size; ++i)
        {
            Node *t = q.front();
            q.pop();
            if (i < size - 1)
                t->next = q.front();
            if (t->left) q.push(t->left);
            if (t->right) q.push(t->right);
        }
    }
    return root;
}
```

199. 二叉树的右视图

```
vector<int> rightSideView(TreeNode* root)
{
    vector<int> res;
    if (!root)
        return res;

    queue<TreeNode *> q;
    TreeNode *last = root;
    TreeNode *nlast = nullptr;
    q.push(root);
    while(!q.empty())
    {
        root = q.front();
        q.pop();

        if (root->left)
        {
            q.push(root->left);
            nlast = root->left;
        }
        if (root->right)
        {
            q.push(root->right);
            nlast = root->right;
        }

        if (root == last)
        {
            res.push_back(root->val);
            last = nlast;
        }

    }
    return res;
}
```

127. 单词接龙 todo: 还有种解法不是很懂

```
//用BFS来求最短路径的长度
int ladderLength(string beginWord, string endWord, vector<string> &wordList)
{
    // todo: 直接使用vector 会超时,主要是因为有大量删除操作
    unordered_set<string> wordSet(wordList.begin(), wordList.end());
    if (!wordSet.count(endWord))
        return 0;
    queue<string> q;
    q.push(beginWord);
    int res = 0;
    while (!q.empty())
    {
        for (int k = q.size(); k > 0; --k)
        {
            string word = q.front();
            q.pop();
            if (word == endWord)
                return res + 1;
            for (int i = 0; i < word.size(); ++i)
            {
                string newWord = word;
                for (char ch = 'a'; ch <= 'z'; ++ch)
                {
                    newWord[i] = ch;
                    if (wordSet.count(newWord) && newWord != word)
                    {
                        q.push(newWord);
                        wordSet.erase(newWord);
                    }
                }
            }
        }
        ++res;
    }
    return 0;
}
```

513. 找树左下角的值

```
/**
 * 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 findBottomLeftValue(TreeNode* root)
    {
        return levelOrder(root, depth(root));
    }
private:

    int depth(TreeNode *root)
    {
        if (root == NULL)
            return 0;
        int ldepth = depth(root->left);
        int rdepth = depth(root->right);
        return ldepth > rdepth ? ldepth + 1 : rdepth + 1;
    }
    int levelOrder(TreeNode *root, int d)
    {
        queue<TreeNode *> q;
        q.push(root);
        int level = 0;
        TreeNode *last = root;
        TreeNode *nlast = nullptr;
        int res;
        while(!q.empty())
        {
            root = q.front();
            q.pop();

            if (level == d-1)
                return root->val;

            if (root->left)
            {
                q.push(root->left);
                nlast = root->left;
            }

            if (root->right)
            {
                q.push(root->right);
                nlast = root->right;
            }


            if(root == last && !q.empty())
            {
                level ++;
                last = nlast;
            }
        }

        return res;

    }
};
```

207. 课程表

难度中等833【拓扑排序】

```
 bool canFinish(int numCourses, vector<pair<int, int>>& prerequisites)
{
    vector<vector<int>> graph(numCourses); // 构建图 领接表的形式
    vector<int> indegree(numCourses,0);    // 顶点的入度表

    for(int i = 0; i < prerequisites.size(); i++)
    {
        graph[prerequisites[i].second].push_back(prerequisites[i].first);
        ++indegree[prerequisites[i].first];
    }

    queue<int> q; // 所有入度为0的结点入队列
    for(int i = 0; i < numCourses; i++)
    {
        if (indegree[i] == 0)
        {
            q.push(i);
        }
    }
    int counter = 0;
    while(!q.empty())
    {
        int u = q.front();
        q.pop();
        ++counter;
        for(int i = 0; i < graph[u].size(); i++)
        {
            if (--indegree[graph[u][i]] == 0)
            {
               q.push(graph[u][i]);
            }
        }
    }
    return counter==numCourses;
}
```

301. 删除无效的括号

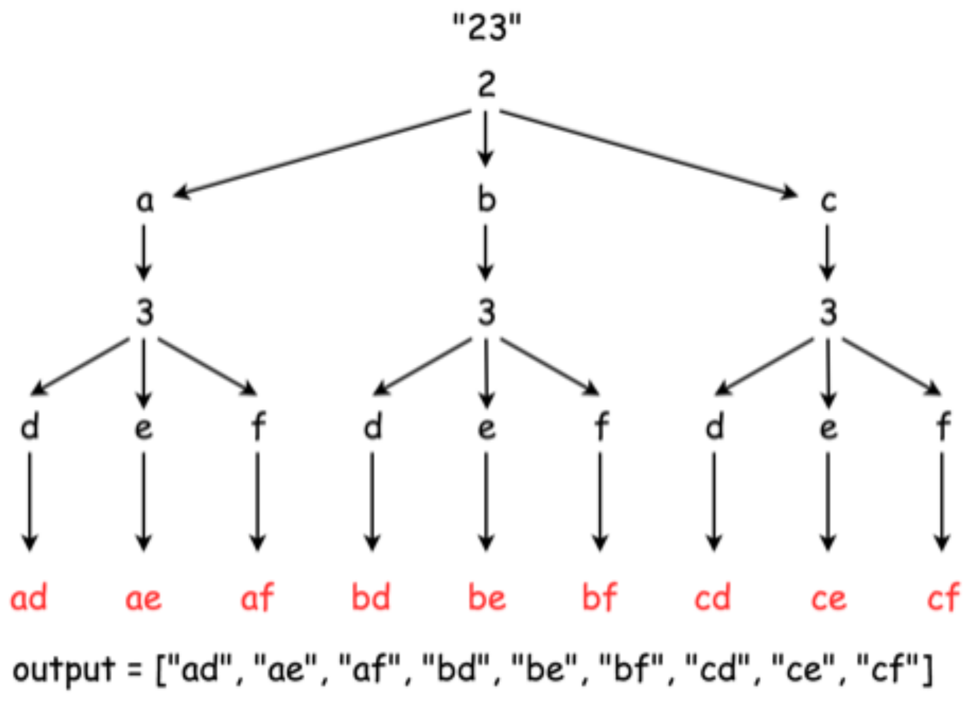
```
class Solution {
public:
    vector<string> removeInvalidParentheses(string s)
    {
        //可以用 BFS 来解，我把给定字符串排入队中，然后取出检测其是否合法，若合法直接返回，不合法的话，对其进行遍历，对于遇到的左右括号的字符，去掉括号字符生成一个新的字符串，如果这个字符串之前没有遇到过，将其排入队中，用 HashSet 记录一个字符串是否出现过。对队列中的每个元素都进行相同的操作，直到队列为空还没找到合法的字符串的话，那就返回空集
        vector<string> res;
        unordered_set<string> visited{{s}};
        queue<string> q{{s}};
        bool found = false;
        while (!q.empty()) {
            string t = q.front(); q.pop();
            if (isValid(t)) {
                res.push_back(t);
                found = true;
            }
            if (found) continue;
            for (int i = 0; i < t.size(); ++i) {
                if (t[i] != '(' && t[i] != ')') continue;
                string str = t.substr(0, i) + t.substr(i + 1);
                if (!visited.count(str)) {
                    q.push(str);
                    visited.insert(str);
                }
            }
        }
        return res;
    }
    bool isValid(string t) {
        int cnt = 0;
        for (int i = 0; i < t.size(); ++i) {
            if (t[i] == '(') ++cnt;
            else if (t[i] == ')' && --cnt < 0) return false;
        }
        return cnt == 0;
    }
};
```

542. 01 矩阵

```
class Solution {
public:
    vector<vector<int>> updateMatrix(vector<vector<int>>& matrix) {

        if(matrix.empty() || matrix[0].empty())
            return matrix;
        queue<pair<int,int>> q;
        int m = matrix.size();
        int n = matrix[0].size();
        for(int i = 0; i < m; i++)
        {
            for(int j = 0; j < n; j++)
            {
                if (matrix[i][j] == 0)
                {
                    q.push({i,j});
                }
                else
                {
                    matrix[i][j] = INT_MAX;
                }
            }
        }

        int dirs[4][2]= {-1, 0, 1, 0, 0, -1, 0, 1};

        while(!q.empty())   // 广度优先遍历
        {

            int x = q.front().first, y = q.front().second;
            q.pop();
            for(int i = 0; i < 4; i++)  // 遍历4个方向
            {
                int nx = x + dirs[i][0];
                int ny = y + dirs[i][1];
                if(nx >= 0 && ny >= 0 && nx < m &&  ny < n && matrix[nx][ny] > 
matrix[x][y])
                {
                    matrix[nx][ny] = matrix[x][y] + 1;
                    q.push({nx,ny});
                }

            }
        }

        return matrix;

    }
};
```

回溯

17. 电话号码的字母组合

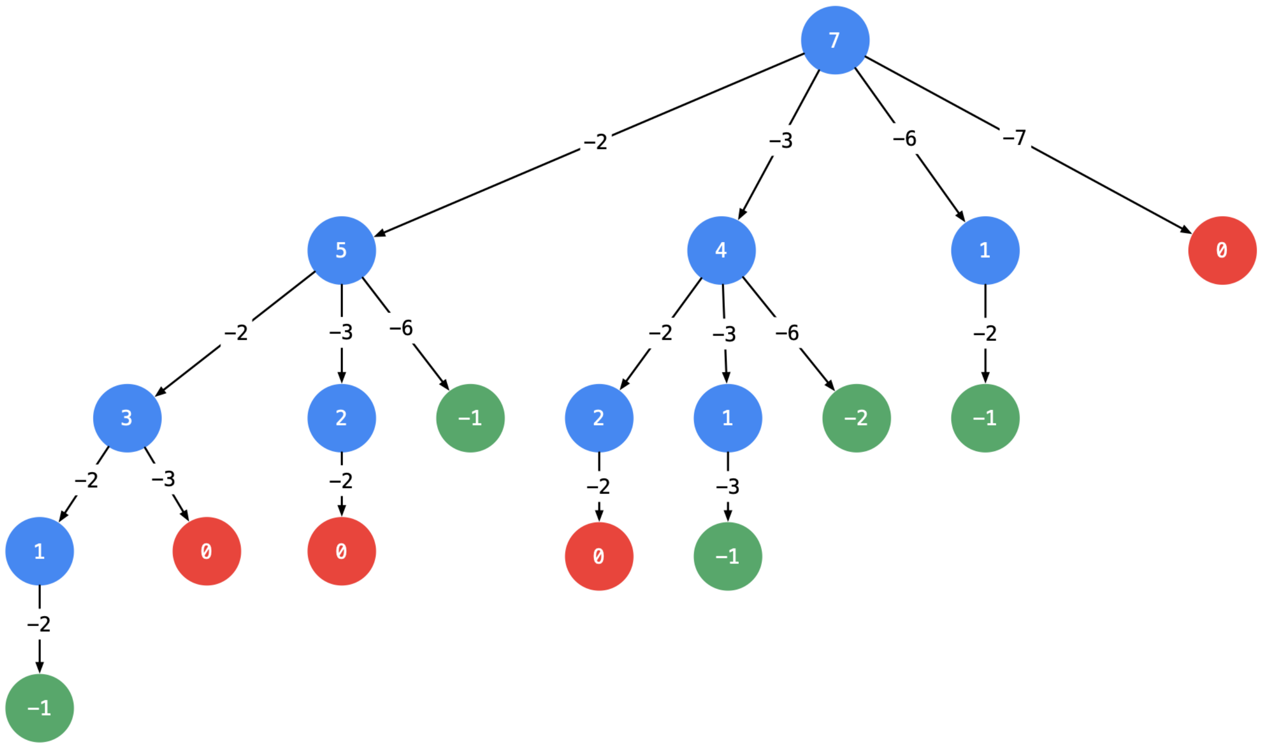

"23"
2
a
b
c
3
3
3
d
e
f
d
e
f
d
e
f
ad
ae
af
bd
be
bf
cd
ce
cf
output = ["ad", "ae", "af", "bd", "be", "bf", "cd", "ce", "cf"]

```
//Input: digits = "23"
//Output: ["ad","ae","af","bd","be","bf","cd","ce","cf"]
class Solution
{
public:
    vector<string> dict = {"abc", "def", "ghi", "jkl", "mno", "pqrs", "tuv", "wxyz"};
    void dfs(string &digits, int start, string &temp, vector<string> &res)
    {
        if (start == digits.size() && temp.size() == digits.size())
        {
            res.push_back(temp);
            return;
        }
        for(int i = start; i < digits.size(); i++)
        {
            int digit = digits[i] - '0' - 2;

            for(int j = 0; j < dict[digit].size(); j++)
            {
                temp.push_back(dict[digit][j]);
                dfs(digits, i+1, temp, res);
                temp.pop_back();
            }
        }
    }
public:
    vector<string> letterCombinations(string digits) {
        string temp;
        vector<string> res;
        if (digits.empty())
            return res;
        dfs(digits, 0, temp, res);
        return res;
    }
};
```

22. 括号生成

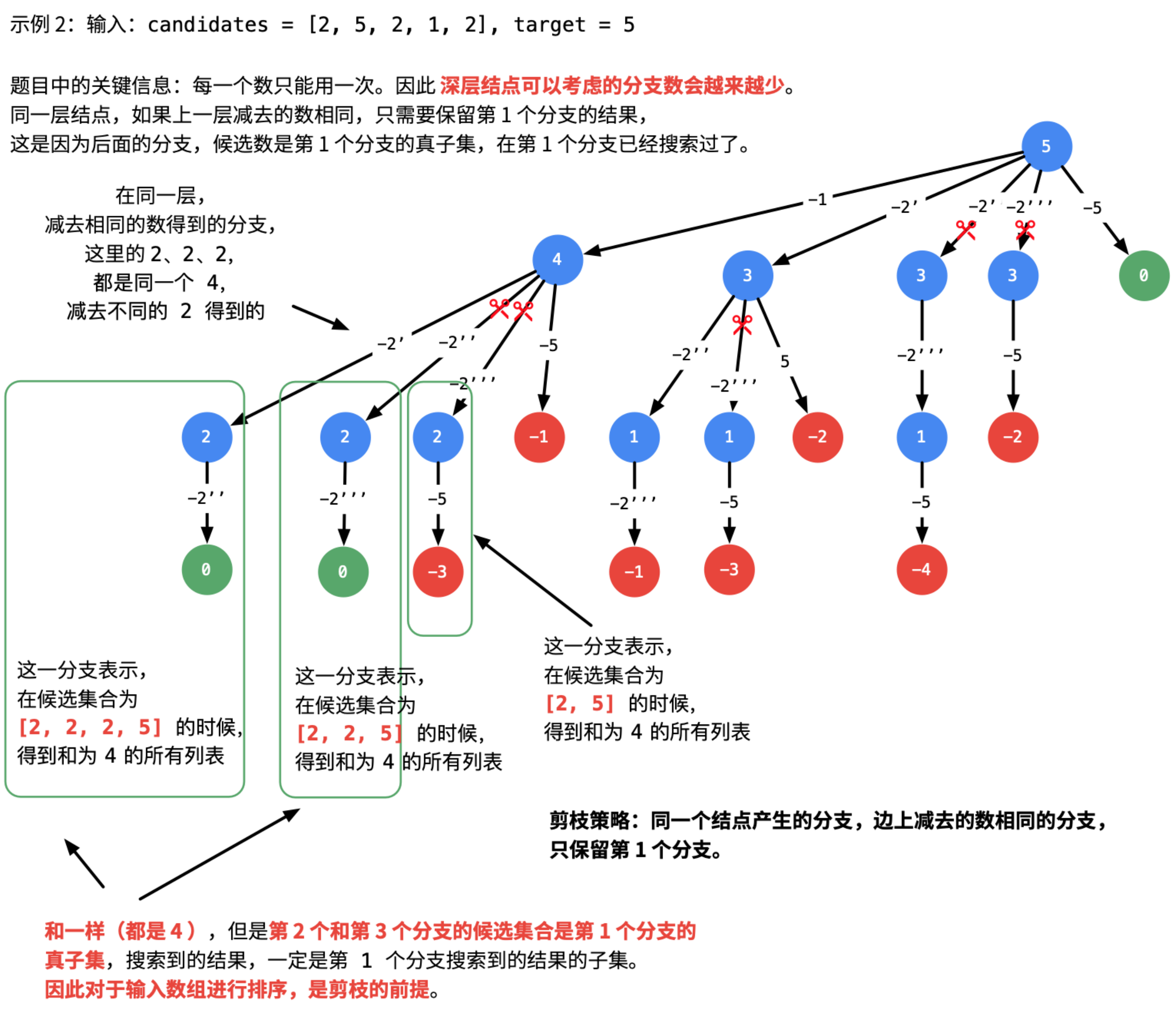
```
class Solution
{
    void dfs(vector<string> & res, string &temp, int left, int right, int n)
    {
        if (temp.size() == 2 *n)
        {
            res.push_back(temp);
            return;
        }

        if (left < n)
        {
            temp.push_back('(');
            dfs(res, temp, left + 1, right, n);
            temp.pop_back();
        }
        if (right < left)
        {
            temp.push_back(')');
            dfs(res, temp, left, right + 1, n);
            temp.pop_back();
        }
    }

public:
    vector<string> generateParenthesis(int n) {
        vector<string> res;
        string temp;
        dfs(res, temp, 0, 0, n);
        return res;
    }
};
```

39. 组合总和



```
class Solution
{
public:
    void backtrack(vector<int>& candidates, int target, vector<int> &temp,
vector<vector<int>> &res, int start)
    {
        if (target < 0)
            return;
        if (target == 0) // 满足条件了 直接返回
        {
            res.push_back(temp);
            return ;
        }

        for(int i = start; i < candidates.size(); i++)
        {
            if(target - candidates[i] < 0)
                break;
            temp.push_back(candidates[i]);
            backtrack(candidates, target - candidates[i], temp, res, i); // i表示每个数字
可以用多次
            temp.pop_back();
        }
    }
public:
    vector<vector<int>> combinationSum(vector<int>& candidates, int target)
    {
        sort(candidates.begin(), candidates.end()); // 减枝
        vector<vector<int>> res;
        vector<int> tmp; // 用来存放每一次满足条件的结果
        backtrack(candidates, target, tmp, res, 0);
        return res;
    }
};
```

[40. 组合总和 II](#)todo 去重策略不是很懂



示例 2：输入：candidates = [2, 5, 2, 1, 2]，target = 5

题目中的关键信息：每一个数只能用一次。因此 **深层结点可以考虑的分支数会越来越少**。
同一层结点，如果上一层减去的数相同，只需要保留第 1 个分支的结果，
这是因为后面的分支，候选数是第 1 个分支的真子集，在第 1 个分支已经搜索过了。

在同一层，
减去相同的数得到的分支，
这里的 2、2、2，
都是同一个 4，
减去不同的 2 得到的

这一分支表示，
在候选集合为
[2, 2, 2, 5] 的时候，
得到和为 4 的所有列表

这一分支表示，
在候选集合为
[2, 2, 5] 的时候，
得到和为 4 的所有列表

这一分支表示，
在候选集合为
[2, 5] 的时候，
得到和为 4 的所有列表

**剪枝策略：同一个结点产生的分支，边上减去的数相同的分支，
只保留第 1 个分支。**

和一样（都是 4），但是**第 2 个和第 3 个分支的候选集合是第 1 个分支的
真子集**，搜索到的结果，一定是第 1 个分支搜索到的结果的子集。
因此对于输入数组进行排序，是剪枝的前提。

```
class Solution
{
public:
    void backtrack(vector<int>& candidates, int target, vector<int> &temp,
vector<vector<int>> &res, int start)
    {
        if (target < 0)
```

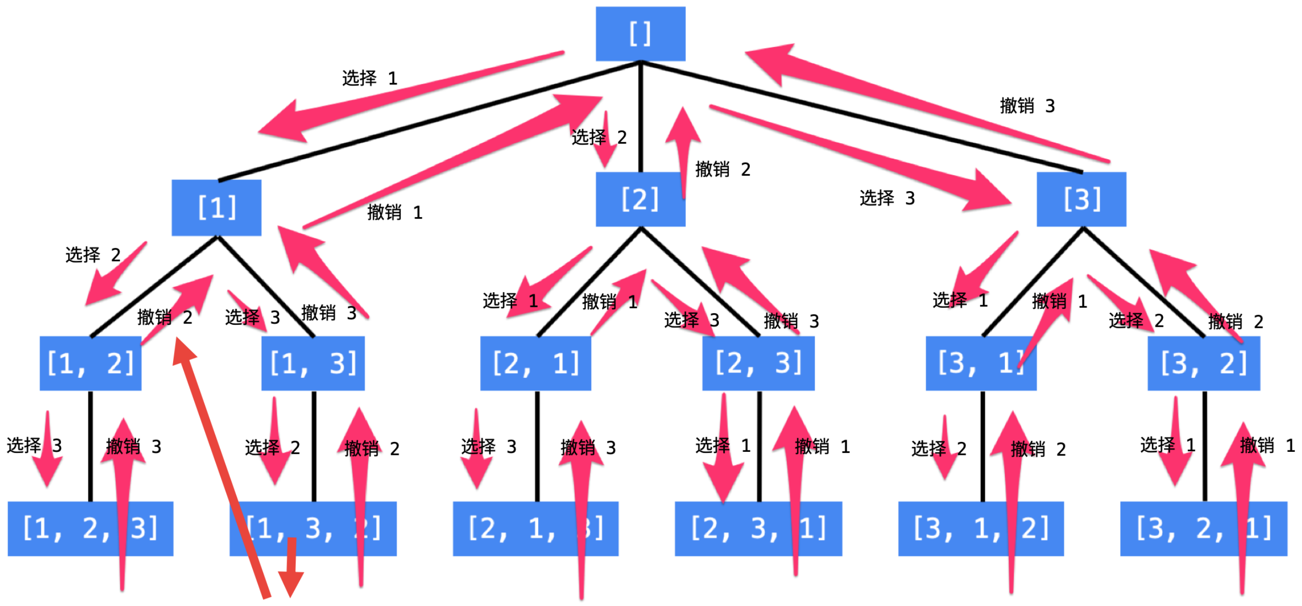
```
            return;
        if (target == 0)
        {
            res.push_back(temp);
            return ;
        }

        for(int i = start; i < candidates.size(); i++)
        {
            if(candidates[i] > target) return;
            if(i && candidates[i] == candidates[i-1] && i > start) continue; // check
duplicate combination
            temp.push_back(candidates[i]);
            backtrack(candidates, target - candidates[i], temp, res, i+1);
            temp.pop_back();
        }
    }
public:
    vector<vector<int>> combinationSum2(vector<int>& candidates, int target) {
        sort(candidates.begin(), candidates.end());
        vector<vector<int>> res;
        vector<int> tmp;
        backtrack(candidates, target, tmp, res, 0);
        return res;
    }
};
```

31. 下一个排列

```
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
void nextPermutation(vector<int>& num)
{
    int i, j, n = num.size();
    for (i = n - 2; i >= 0; --i) {
        if (num[i + 1] > num[i]) {
            for (j = n - 1; j > i; --j) {
                if (num[j] > num[i]) break;
            }
            swap(num[i], num[j]);
            reverse(num.begin() + i + 1, num.end());
            return;
        }

    }
    reverse(num.begin(), num.end());
}
```

46. 全排列

难度中等1294



正是因为在上一步撤销了对 2 的选择，在这一步才能选择 2，这是在深度优先遍历的过程中，需要状态重置的意义。
其它地方也是一样的，就不标注了。

```
class Solution
{
public:
    vector<vector<int>> res;
    vector<int> temp;

    void dfs(vector<int> &nums, vector<bool> &uesd)
    {

        if (temp.size() == nums.size())
        {
            res.push_back(temp);
            return;
        }
        for (int i = 0; i < nums.size(); i++)
        {
            if (!uesd[i])
            {
                temp.push_back(nums[i]);
                uesd[i] = true;
                dfs(nums,uesd);
                uesd[i] = false;
                temp.pop_back();
            }
        }
    }
    vector<vector<int>> permute(vector<int> &nums)
    {
        vector<bool> uesd(nums.size());
        dfs(nums,uesd);
        return res;
    }
};
```

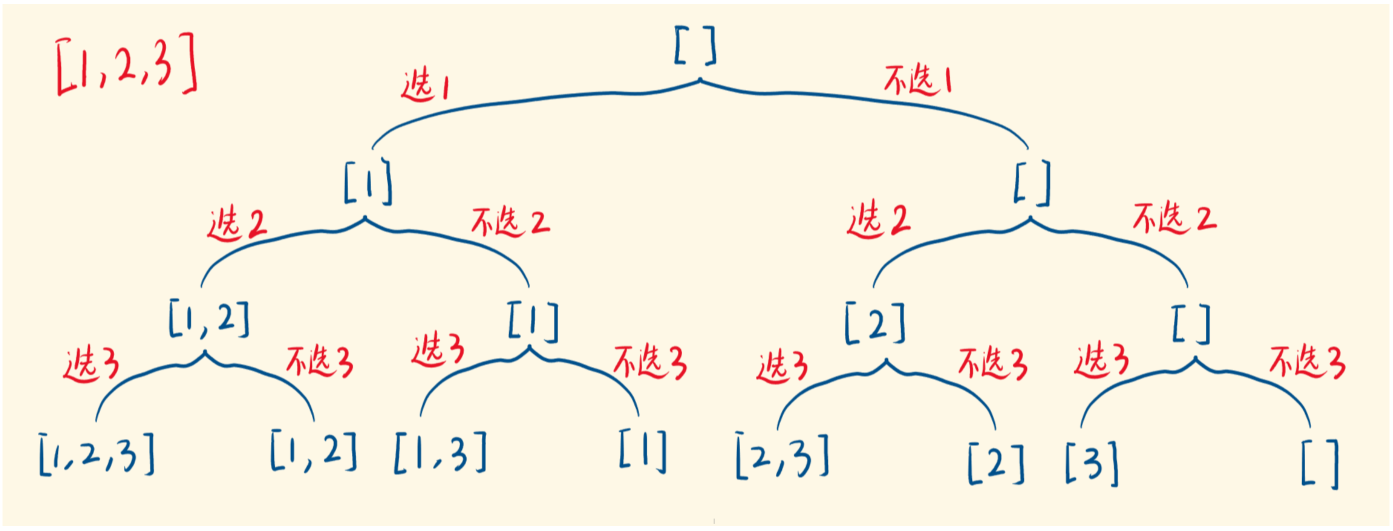
47. 全排列 II 同40

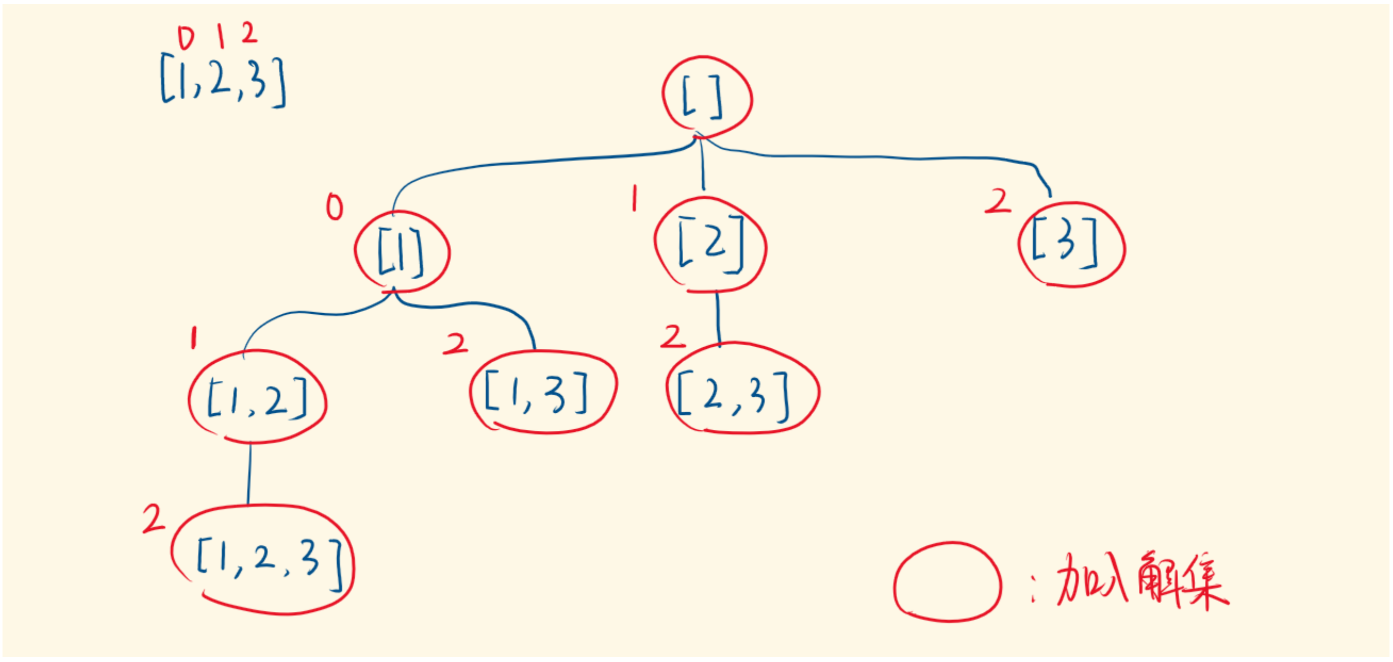
```
class Solution
{
public:
    void dfs(vector<vector<int>> &res, vector<int> &temp, vector<int> &nums, 
vector<bool> &uesd, int start)
    {

        if (temp.size() == nums.size())
        {
            res.push_back(temp);
            return;
        }

        for(int i = 0; i < nums.size(); i++)
        {
            if (!uesd[i])
            {
                if (i > 0 && nums[i] == nums[i-1] && uesd[i-1])
                    continue;
                uesd[i] = true;
                temp.push_back(nums[i]);
                dfs(res, temp, nums, uesd, i+1);
                uesd[i] = false;
                temp.pop_back();
            }
        }
    }
public:
    vector<vector<int>> permuteUnique(vector<int>& nums)
    {
        vector<vector<int>> res;
        vector<int> temp;
        vector<bool> uesd(nums.size());
        sort(nums.begin(), nums.end());
        dfs(res, temp, nums, uesd, 0);
        return res;

    }
};
```

77. Combinations

```
class Solution
{
private:
    void dfs(vector<vector<int>> &res,vector<int>& nums, int n, int k, int first)
    {
        if (nums.size() == k)
        {
            res.push_back(nums);
            return;
        }
        for (int i = first; i <= n; i ++)
        {
            nums.push_back(i);
            dfs(res, nums, n, k, i+1);
            nums.pop_back();
        }
    }
public:
    vector<vector<int>> combine(int n, int k) {
        vector<vector<int>> res;
        vector<int> nums;
        dfs(res, nums, n, k, 1);
        return res;
    }
};
```

78. 子集





```
class Solution
{   // 解法一:  每个元素，都有两种选择：选入子集，或不选入子集。
    void dfs(vector<int>& nums, int i, vector<int> &temp, vector<vector<int>> &res )
    {
        if (i == nums.size())
        {
            res.push_back(temp);
            return;
        }
        temp.push_back(nums[i]);  // 选择这个数
        dfs(nums, i+1, temp, res);  // 基于该选择，继续往下递归
        temp.pop_back();          // 上面的递归结束，撤销该选择
        dfs(nums, i+1, temp, res);  // 不选这个数，继续往下递归
    }
public:
    vector<vector<int>> subsets(vector<int>& nums)
    {
        vector<vector<int>> res;
        if (nums.empty())
            return res;

        vector<int> temp;
        dfs(nums,0, temp, res);
        return res;
    }
};

// 解法二:
//每次看看有几个数能选，然后选一个
//用 for 枚举出当前可选的数，比如选第一个数：有 1、2、3 可选。
//选了 1，选第二个数，有 2、3 可选；如果选 2，选第二个数，只有 3 可选，如下图
class Solution
{
public:
    void dfs(vector<vector<int>> &res, vector<int> &temp, vector<int> &nums, int index)
    {
        res.push_back(temp);
        for(int i = index; i < nums.size(); i++)
        {
            temp.push_back(nums[i]);
            dfs(res, temp, nums, i+1);
            temp.pop_back();
        }
    }
    vector<vector<int>> subsets(vector<int>& nums)
    {
        vector<vector<int>> res;
        vector<int> temp;
        dfs(res, temp, nums, 0);
        // res.push_back({});
        return res;
    }
};
```

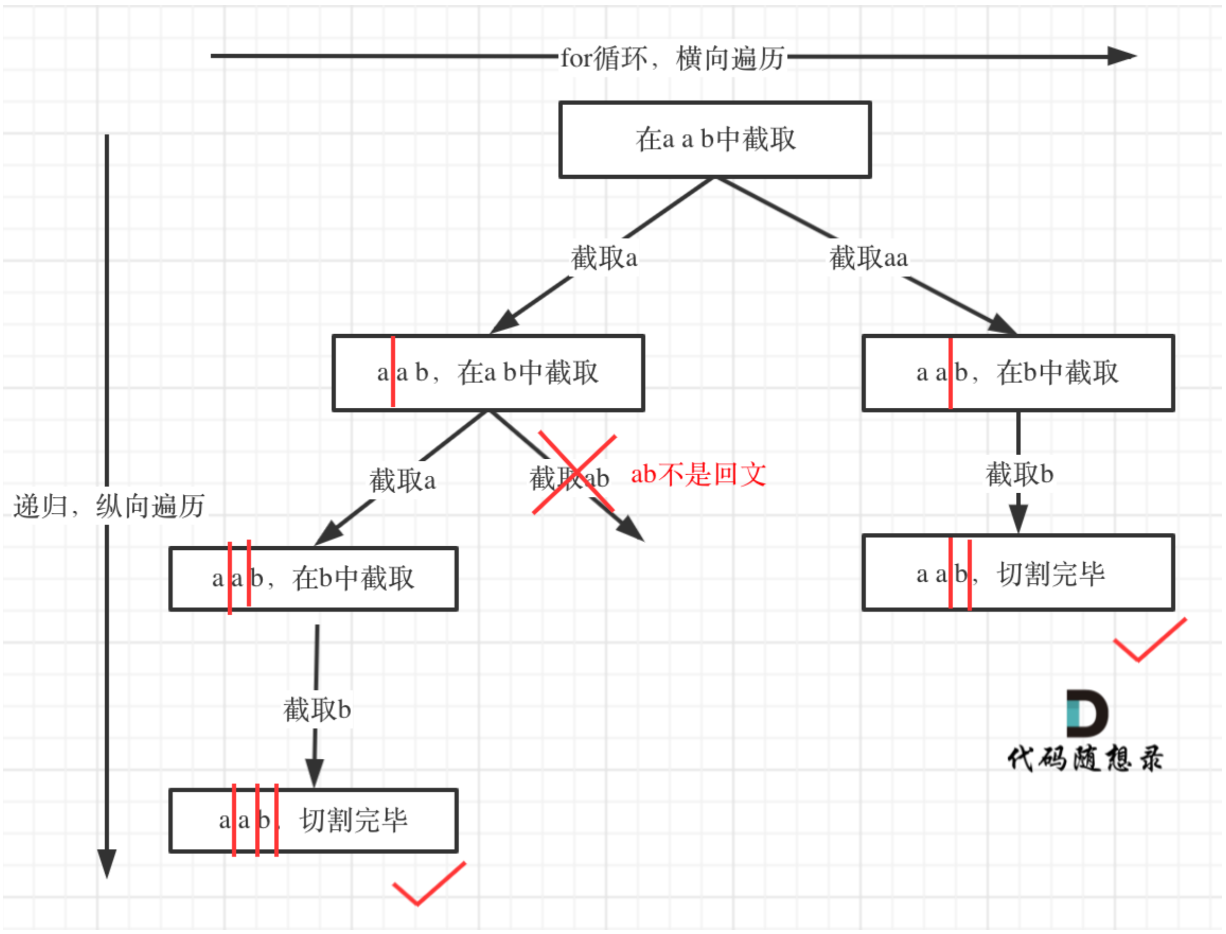
90. 子集 II

```
class Solution
{
    void dfs(vector<int>& nums, int start, vector<int>& temp, vector<vector<int>>& res)
    {
        res.push_back(temp);
        for(int i = start; i < nums.size(); i++)
        {
            if (i == start || nums[i] != nums[i-1])
            {

                temp.push_back(nums[i]);
                dfs(nums, i+1, temp, res);
                temp.pop_back();
            }
        }
    }
public:
    vector<vector<int>> subsetsWithDup(vector<int>& nums)
    {
        vector<int> temp;
        vector<vector<int>> res;
        sort(nums.begin(), nums.end());
        dfs(nums, 0, temp, res);
        return res;
    }
};
```

93. 复原IP地址

```
class Solution
{
public:
    vector<string> restoreIpAddresses(string s)
    {
        vector<string> result;
        string ip;
        dfs(s, 0, 0, ip, result); //paras:string s,start index of s,step(from0-
3),intermediate ip,final result
        return result;
    }
    void dfs(string s, int start, int step, string ip, vector<string> &result)
    {
        if (start == s.size() && step == 4)
        {
            ip.erase(ip.end() - 1); //remove the last '.' from the last decimal number
            result.push_back(ip);
            return;
        }
        if (s.size() - start > (4 - step) * 3)
            return;
        if (s.size() - start < (4 - step))
            return;
        int num = 0;
        for (int i = start; i < start + 3; i++)
        {
            num = num * 10 + (s[i] - '0');
            if (num <= 255)
            {
                ip += s[i];
                dfs(s, i + 1, step + 1, ip + '.', result);
            }
            if (num == 0)
                break;
        }
    }
};
```

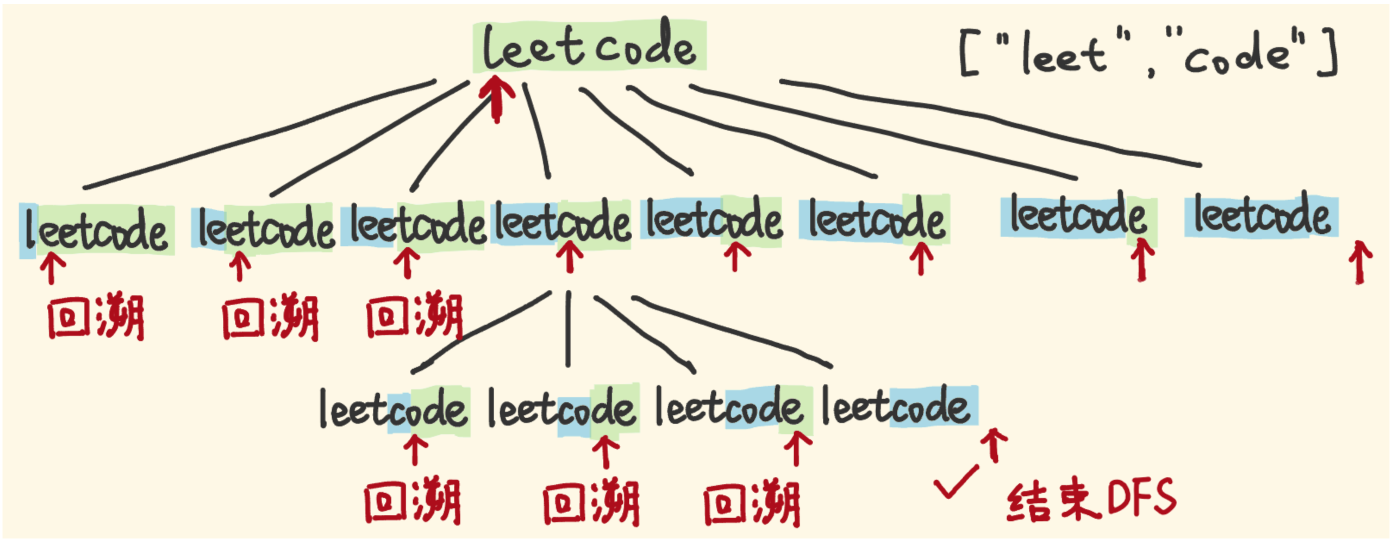
131. Palindrome Partitioning

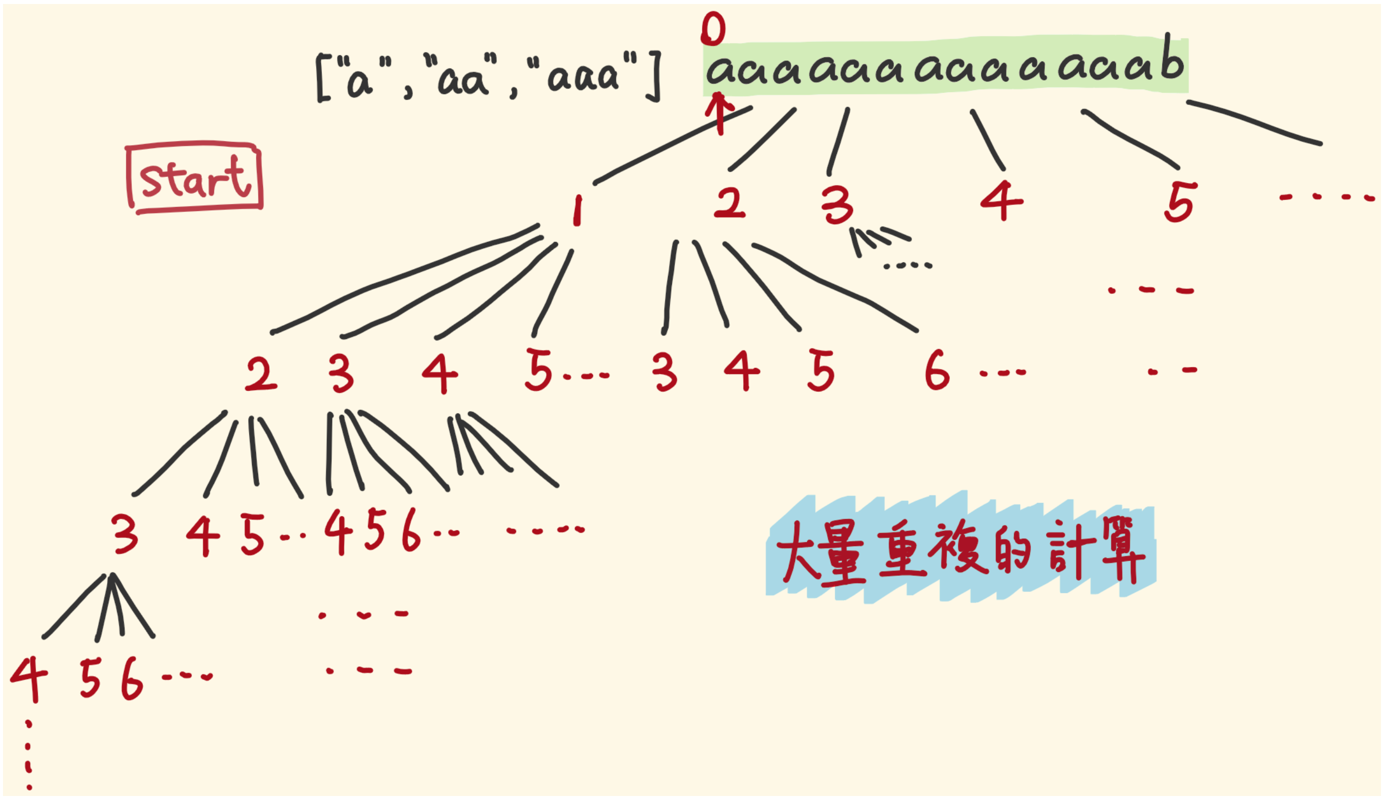


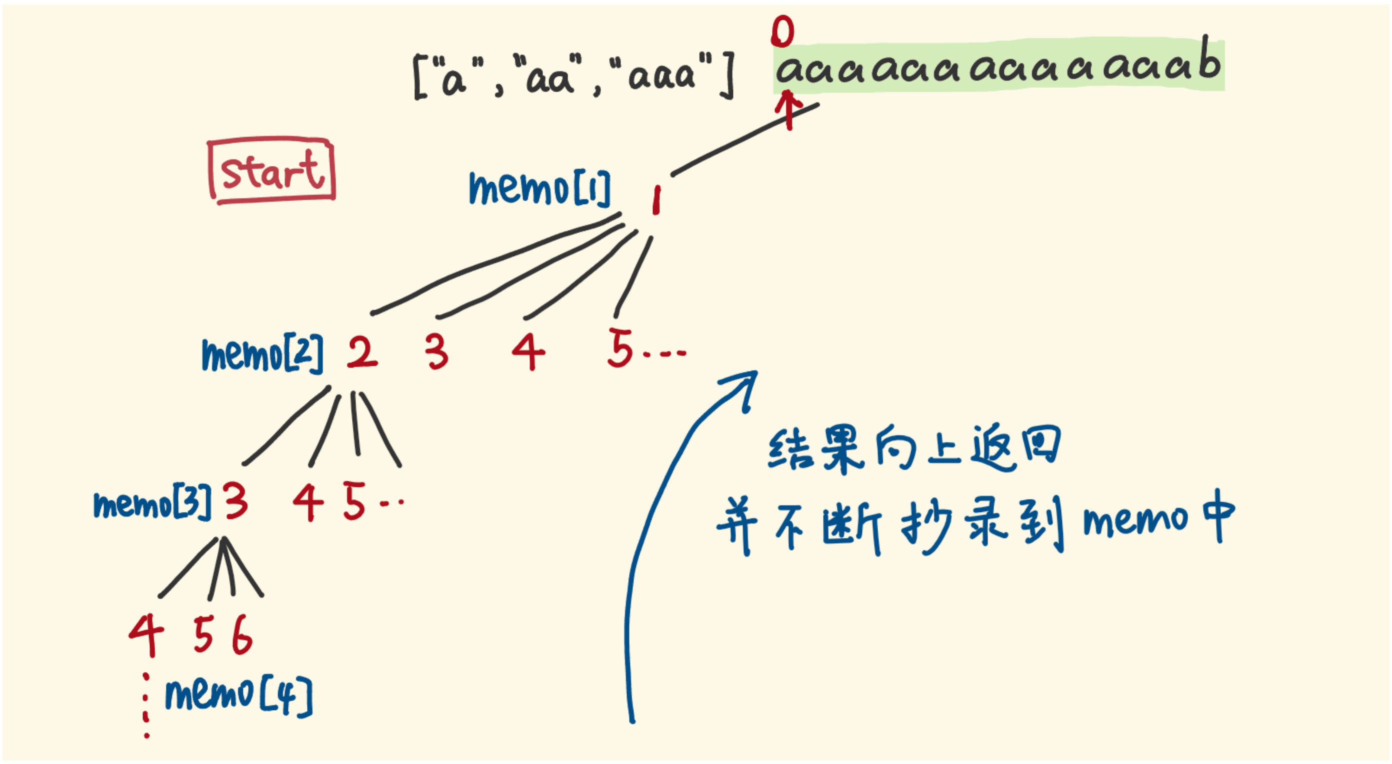
```
class Solution
{
    bool isPalindrome(const string& s, int start, int end)
    {
        while(start <= end) {
            if(s[start++] != s[end--])
                return false;
        }
        return true;
    }
    // dfs含义是
    void dfs(string &s, int index, vector<string> &temp, vector<vector<string>> &res)
    {
        if (index == s.size())
        {
            res.push_back(temp);
```

```
                return;
            }
            for(int i = index; i < s.size(); i++)  // a, aa, aab for循环横向遍历
            {
                if (isPalindrome(s, index, i)) // 这个地方可以用动态规划去优化
                {
                    temp.push_back(s.substr(index, i - index + 1)); // 获取[index,i]在s中的子
串
                    dfs(s, i + 1, temp, res);
                    temp.pop_back();
                }
            }
        }
public:
    vector<vector<string>> partition(string s)
    {
        vector<vector<string> > res;
        if(s.empty()) return res;
        vector<string> temp;
        dfs(s, 0, temp, res);
        return res;
    }
};
```

139. 单词拆分

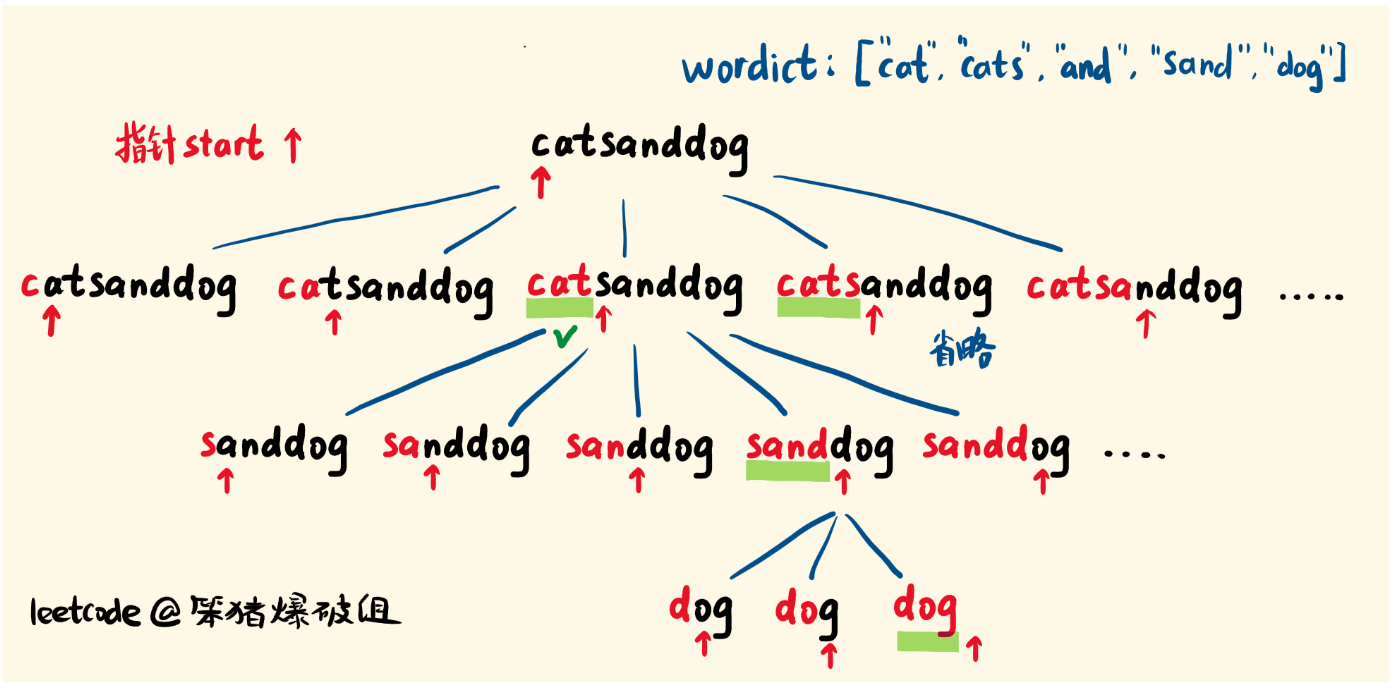






```
class Solution
{
public:
    /**
     * 解法一：
     * memo[i] 定义为范围为 [i, n] 的子字符串是否可以拆分，初始化为 -1，表示没有计算过，如果可以拆
分，则赋值为1，反之为0
     */
    bool dfs(string &s, vector<string>& wordDict, int index,  vector<int> &memo)
    {
        if (index == s.size())  // 指针越界，s都划分成单词了才走到这步的，没有剩余子串了，返回真，结束递归
            return true;
        if (memo[index] != -1)
            return memo[index];

        for(int i = index; i < s.size(); i++)
        {
            // 先判断字典中是否能找到子串 [index...i] 然后递归调用[i+1...s.size()]
            if (find(wordDict.begin(),wordDict.end(), s.substr(index, i - index + 1)) != wordDict.end() && dfs(s, wordDict, i +1, memo))
            {
                memo[index] = true;
                return true;
            }
        }
        memo[index]=false;
        return false;
    }



    bool wordBreak(string s, vector<string>& wordDict)
    {
        vector<int> memo(s.size(), -1);
        return dfs(s, wordDict, 0, memo);
    }

    /**
     * 解法二：
     * 其中 dp[i] 表示范围 [0, i) 内的子串是否可以拆分，注意这里 dp 数组的长度比s串的长度大1，是因为我们要 handle 空串的情况，我们初始化 dp[0] 为 true，然后开始遍历
     */
    bool wordBreak(string s, vector<string> &wordDict)
    {
        unordered_set<string> wordSet(wordDict.begin(), wordDict.end());
        vector<bool> dp(s.size() + 1);
        dp[0] = true;
        for (int i = 0; i < dp.size(); ++i)
        {
            for (int j = 0; j < i; ++j)
            {
                if (dp[j] && wordSet.count(s.substr(j, i - j)))
                {
                    dp[i] = true;
                    break;
                }
            }
        }
        return dp.back();
    }
};
```

140. Word Break II 注意和139的区别



wordict: ["cat", "cats", "and", "sand", "dog"]
指针start ↑
catsanddog
catsanddog catsanddog catsanddog catsanddog catsanddog .....
省略
sanddog sanddog sanddog sanddog sanddog ....
dog dog dog
leetcode @笨猪爆破组

```
// 解法一 暴力递归超时了  想改成不带返回值得形式
vector<string> dfs(string s, unordered_set<string> &wordSet, int start)
{
    if (start == s.size())
    {
        return {""};
    }
    vector<string> res;
    for (int i = start; i < s.size(); ++i)
    {
        if (wordSet.count(s.substr(start, i - start+1)) > 0)
        {
            vector<string> tmp = dfs(s, wordSet, i+1);
            for (auto str : tmp)
            {
                res.push_back(s.substr(start, i - start + 1) + (str.empty() ? "" : " ")
+ str);
            }
        }
    }
    return res;
}

vector<string> wordBreak(string s, vector<string> &wordDict)
```

```
{
    unordered_set<string> wordSet(wordDict.begin(), wordDict.end());
    return dfs(s, wordSet, 0);
}

// 解法二：增加记忆化搜索
vector<string> dfs(string &s, vector<string> &wordDict, unordered_map<int,
vector<string>> &memo, int index)
{
    if (index == s.size())
        return {""};

    if (memo.count(index) > 0)
        return memo[index];

    vector<string> res;
    for (int i = index; i < s.size();i++)  // 横向切割字符串 [index...i] 和
[i+1...s.size()]
    {
        // 先判断字典中是否能找到子串 [index...i] 然后递归调用[i+1...s.size()]
        if (find(wordDict.begin(),wordDict.end(), s.substr(index, i - index + 1)) !=
wordDict.end())
        {
            vector<string> temp = dfs(s, wordDict, memo, i+1);
            for(auto str : temp)
            {
                res.push_back(s.substr(index, i - index + 1) +  (str.empty() ? "" : "
") + str);
            }
        }
    }
    memo[index] = res;
    return res;
}

vector<string> wordBreak(string s, vector<string>& wordDict)
{
    unordered_map<int, vector<string>> memo;
    return dfs(s, wordDict,  memo, 0);
}
```

二叉树遍历相关

97. 中序遍历二叉树

```
// 二叉树中序非递归遍历
vector<int> inorderTraversal(TreeNode* root)
{
    vector<int> res;
    if (root == nullptr)
    {
        return {};
    }
    stack<TreeNode *> s;
    while(!s.empty() || root)
    {
        while (root != nullptr)
        {
            s.push(root);
            root = root->left;
        }
        root = s.top();
        res.push_back(root->val);
        s.pop();
        root = root->right;

    }
    return res;
}
```

98. 验证二叉搜索树

```
bool isValidBST(TreeNode* root)
{
    if (root == nullptr)
        return true;
    stack<TreeNode *> s;
    TreeNode *cur = root;
    TreeNode *pre = nullptr;
    while(!s.empty() || cur)
    {
        while(cur)
        {
            s.push(cur);
            cur =cur->left;
        }
        cur = s.top();
        s.pop();
        if (pre != nullptr && pre->val >= cur->val)
            return false;
        pre = cur;
        cur = cur->right;
    }
    return true;
}
```

98_2 判断是否完全二叉树

```
bool judgeTotal(TreeNode *root)
{
    if (root == nullptr)
        return true;
    queue<TreeNode *> q;
    q.push(root);
    while(!q.empty())
    {
        root = q.front();
        q.pop();
        if (root->left && root->right)
        {
            q.push(root->left);
            q.push(root->right);
        }

        else if (root->left == nullptr && root->right)
        {
            return false;
        }
        else
        {
            while(!q.empty())
            {
                root = q.front();
                q.pop();
                if (root->left || root->right)
                    return false;
            }
        }
    }
    return true;
}
```

114. 二叉树展开为链表 二叉树前序遍历非递归

```
void flatten(TreeNode* root)
{
    if(root == nullptr)
        return;
    stack<TreeNode *> s;
    vector<TreeNode *> node_list;
    s.push(root);
    while(!s.empty())
    {
        root = s.top();
        s.pop();
        node_list.push_back(root);
        // 因为需要先访问左子树，所以左子树后压栈
        if (root->right)
            s.push(root->right);
        if (root->left)
            s.push(root->left);
    }
    for(int i = 1; i < node_list.size(); i++)
    {
        auto prev = node_list[i - 1], curr = node_list[i];
        prev->left = nullptr;
        prev->right = curr;
    }
}
```

144. 二叉树的前序遍历

```
// 二叉树前序遍历非递归
vector<int> preorderTraversal(TreeNode* root)
{
    vector<int> res;
    if (root == nullptr)
        return res;

    stack<TreeNode *> s;
    s.push(root);

    while(!s.empty())
    {
        root = s.top();
        s.pop();
        res.push_back(root->val);

        if (root->right)
            s.push(root->right);
        if (root->left)
            s.push(root->left);
    }
    return res;
}
```

145. 二叉树的后序遍历

```
vector<int> postorderTraversal(TreeNode* root)
{
    vector<int> res;
    if (root == nullptr)
```

```
        return res;

    stack<TreeNode *> s1;
    stack<TreeNode *> s2;
    s1.push(root);
    while(!s1.empty())
    {
        root = s1.top();
        s1.pop();
        s2.push(root);

        if (root->left)
            s1.push(root->left);
        if (root->right)
            s1.push(root->right);
    }
    while(!s2.empty())
    {
        res.push_back(s2.top()->val);
        s2.pop();
    }
    return res;
}
```

230. 二叉搜索树中第K小的元素

```
int kthSmallest(TreeNode* root, int k)
{
    if (!root || k < 0)
        return 0;

    stack<TreeNode *> s;
    TreeNode *cur = root;
    while(!s.empty() || cur)
    {
        if (cur)
        {
            s.push(cur);
            cur = cur->left;
        }
        else
        {
            TreeNode *temp = s.top();
            s.pop();
            k--;
            if (k == 0)
                return temp->val;
            cur = temp->right;
        }
    }
    return 0;
}
```

235. 二叉搜索树的最近公共祖先

```
TreeNode *lowestCommonAncestor(TreeNode *root, TreeNode *p, TreeNode *q)
{
    //如果跟节点的值 比pq都大,那么最近公共祖先只能在左子树上，否则在右子树上，只有当root->val 在pq之间的话，root才是最近公共祖先
    while (true)
    {
        if (root->val > p->val && root->val > q->val)
            root = root->left;
        else if (root->val < p->val && root->val < q->val)
            root = root->right;
        else
            break;
    }
    return root;
}

   // 如果根节点的值大于p和q之间的较大值，说明p和q都在左子树中，那么此时我们就进入根节点的左子节点继续递归，如果根节点小于p和q之间的较小值，说明p和q都在右子树中，那么此时我们就进入根节点的右子节点继续递归，如果都不是，则说明当前根节点就是最小共同父节点，直接返回即可  递归版本
    TreeNode *lowestCommonAncestor_2(TreeNode *root, TreeNode *p, TreeNode *q)
    {
        if (!root)
            return NULL;
        if (root->val > max(p->val, q->val))
            return lowestCommonAncestor(root->left, p, q);
        else if (root->val < min(p->val, q->val))
            return lowestCommonAncestor(root->right, p, q);
        else
            return root;
    }
```

236. 二叉树的最近公共祖先

```
TreeNode *lowestCommonAncestor(TreeNode *root, TreeNode *p, TreeNode *q)
{
    // 看当前结点是否为空，若为空则直接返回空，若为p或q中的任意一个，也直接返回当前结点。否则的话就对其左右子结点分别调用递归函数，由于这道题限制了p和q一定都在二叉树中存在，那么如果当前结点不等于p或q，p和q要么分别位于左右子树中，要么同时位于左子树，或者同时位于右子树
    if (root == nullptr || root == p || root == q)
        return root;
    TreeNode *left = lowestCommonAncestor(root->left, p, q);
    TreeNode *right = lowestCommonAncestor(root->right, p, q);

    if (left && right)
        return root;
    else
        return left != nullptr ? left : right;
}
```

538. 把二叉搜索树转换为累加树

```
int sum = 0;
TreeNode* convertBST_1(TreeNode* root) {
    if (!root) return NULL;
    convertBST(root->right);
    root->val += sum;
    sum = root->val;
    convertBST(root->left);
    return root;
}

TreeNode* convertBST(TreeNode* root) {
    if (!root) return NULL;
    int sum = 0;
    stack<TreeNode*> st;
    TreeNode *p = root;
    while (p || !st.empty()) {
        while (p) {
            st.push(p);
            p = p->right;
        }
        p = st.top(); st.pop();
        p->val += sum;
        sum = p->val;
        p = p->left;
    }
    return root;
}
```

543. 二叉树的直径

```
int diameterOfBinaryTree(TreeNode* root)
{
    int res = 0;
    maxDepth(root, res);
    return res;
}
int maxDepth(TreeNode* node, int& res
{
    if (!node) return 0;
    int left = maxDepth(node->left, res);
    int right = maxDepth(node->right, res);
    res = max(res, left + right);
    return max(left, right) + 1;
}
```

99. 恢复二叉搜索树

```
/**
 * 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 {
    void inOrderTraverse(TreeNode* &root, TreeNode* &pre,TreeNode* &first,TreeNode*
&second)
    {
        if (root == nullptr)
            return;

        stack<TreeNode*> s;

        while(!s.empty() || root)
        {
            if(root)
            {
                s.push(root);
                root = root->left;
            }
            else
            {
                root = s.top();
                s.pop();
                if (pre != nullptr && pre->val > root->val)
                {
                    first = first == nullptr ? pre : first;
                    second = root;
                }
                pre = root;
                root = root->right;
            }
        }
    }

public:
    void recoverTree(TreeNode* root)
    {

        TreeNode *first = nullptr;
        TreeNode *second = nullptr;
        TreeNode *pre = nullptr;
        inOrderTraverse(root, pre, first, second);

        int temp = first->val;
        first->val = second->val;
        second->val = temp;

    }
};
```

99_2找到搜索二叉树中两个错误的节点

```
class Solution {
public:
    /**
     *
     * @param root TreeNode类 the root
     * @return int整型vector
     */
    void inOrderTraverse(TreeNode* &root, TreeNode* &pre, TreeNode* &first,TreeNode*
&second)
    {
        if (root == nullptr)
            return;

        stack<TreeNode*> s;

        while(!s.empty() || root)
        {
            if(root)
            {
                s.push(root);
                root = root->left;
            }
            else
            {
                root = s.top();
                s.pop();

                if (pre != nullptr && pre->val > root->val)
                {
                    first = first == nullptr ? pre : first;
                    second = root;
                }
                pre = root;
                root = root->right;
            }
        }
    }


    vector<int> findError(TreeNode* root) {
        // write code here
        TreeNode* first=nullptr;
        TreeNode* second=nullptr;
        TreeNode* pre= nullptr;
        vector<int> res;
        inOrderTraverse(root, pre, first, second);
        res.push_back(first->val);
        res.push_back(second->val);
        sort(res.begin(), res.end());
        return res;

    }
```

二叉树的DFS

通常采用递归

100. 相同的树

```
bool isSameTree(TreeNode *p, TreeNode *q)
{
    if (p == nullptr && q == nullptr)
        return true;
    if ((p == nullptr && q != nullptr) || (p != nullptr && q == nullptr))
        return false;

    if (!p && !q && p->val != q->val)
        return false;
    else
    {
        return p->val == q->val && isSameTree(p->left, q->left) && isSameTree(p->right,
q->right);
    }
}
```

101. 对称二叉树

```
class Solution
{
    bool isSymmetricTree(TreeNode *root1, TreeNode *root2)
    {
        if (!root1 && !root2)
            return true;
        if (!root1 || !root2)
            return false;
        if (root1 && root2)
            return root1->val == root2->val && isSymmetricTree(root1->left, root2-
>right) && isSymmetricTree(root1->right, root2->left);
        else
            return false;
    }

public:
    bool isSymmetric(TreeNode *root)
    {
        if (!root)
            return true;
        return isSymmetricTree(root->left, root->right);
    }
};
```

104. 二叉树的最大深度

```
int maxDepth(TreeNode* root)
{
    if (root == NULL)
        return 0;
    else return max(maxDepth(root->left), maxDepth(root->right)) + 1;
}
```

110. 平衡二叉树

```
int height(TreeNode *root)
{
    if(root == NULL)return 0;
        return max(height(root->left), height(root->right)) + 1;
}
bool isBalanced(TreeNode* root)
{
    if(root == NULL)
        return true;
    else return isBalanced(root->right) && isBalanced(root->left) && abs(height(root-
>left) - height(root->right)) <= 1;
}
```

572. 另一个树的子树

```
class Solution {
    bool isSame(TreeNode* s, TreeNode* t)
    {
        if (s == nullptr && t == nullptr)
            return true;
        if (s == nullptr || t == nullptr)
            return false;

        if (s->val != t->val)
            return false;
        else
            return isSame(s->left, t->left) && isSame(s->right, t->right);

    }


public:
    bool isSubtree(TreeNode* s, TreeNode* t)
    {
        if (s == nullptr && t == nullptr)
            return true;
        if (s == nullptr || t == nullptr)
            return false;
        if (isSame(s, t))
            return true;
        return isSubtree(s->left, t) || isSubtree(s->right, t);
    }
};
```

112. 路径总和

```
bool hasPathSum(TreeNode *root, int sum)
{
    if (root == NULL)
        return false;
    if (root->left == NULL && root->right == NULL && sum == root->val) // 叶子节点
        return true;
    else
        return hasPathSum(root->right, sum - root->val) || hasPathSum(root->left, sum -
root->val);
}
```

113. 路径总和 II（和剑指 Offer 34. 二叉树中和为某一值的路径一样）

```
// **本质上还是回溯**
class Solution
{
public:
    void dfs(TreeNode *node, int sum, vector<int> &temp, vector<vector<int>> &res)
    {
        if (node == nullptr)
            return;
        temp.push_back(node->val);
        // 叶子节点
        if (sum == node->val && node->left == nullptr  && node->right == nullptr)
        {
            res.push_back(temp);
        }
        dfs(node->left, sum - node->val, temp, res);
        dfs(node->right, sum - node->val, temp, res);
        temp.pop_back();
    }
    vector<vector<int>> pathSum(TreeNode *root, int sum)
    {
        vector<vector<int>> res;
        vector<int> temp;
        dfs(root, sum, temp, res);
        return res;
    }
};
```

437. 路径总和 III

```
// 本质上还是回溯
// 每一个节点都有记录了一条从根节点到当前节点到路径 path
// 用一个变量 curSum 记录路径节点总和，然后看 curSum 和 sum 是否相等，相等的话结果 res 加1，
// 不等的话继续查看子路径和有没有满足题意的，做法就是每次去掉一个节点，看路径和是否等于给定值
```

```
int pathSum(TreeNode* root, int sum)
{
    int res = 0;
    vector<TreeNode*> temp;
    dfs(root, sum, 0, temp, res);
    return res;
}
void dfs(TreeNode* node, int sum, int curSum, vector<TreeNode*>& temp, int& res) {
    if (!node) return;
    curSum += node->val;
    temp.push_back(node);
    if (curSum == sum) ++res;
    int t = curSum;
    for (int i = 0; i < temp.size() - 1; ++i) {
        t -= temp[i]->val;
        if (t == sum) ++res;
    }
    dfs(node->left, sum, curSum, temp, res);
    dfs(node->right, sum, curSum, temp, res);
    temp.pop_back();
}
```

124. 二叉树中的最大路径和todo: 还不是很懂

```
class Solution
{
public:
    // 递归函数表示 从node出发能得到的最大路径和
    // 1) Max path sum lies only in the right half.
    // 2) Max path sum lies only in the left half.
    // 3) Max path passes from left to right half (or vice versa) through the root
node.
    int dfs(TreeNode *node, int &res)
    {
        if (node == nullptr)
            return 0;

        // 分别求出左右子树上的最大路径和
        int left = max(dfs(node->left, res), 0);
        int right = max(dfs(node->right, res), 0);

        // todo: 这个地方不理解
        res = max(left + right + node->val, res);

        // 对当前node来说, 它的最大路径和就是当前节点值加上 其左右子树上的最大值
        return max(left, right) + node->val;
    }

    int maxPathSum(TreeNode *root)
    {
        int res = INT_MIN;
        dfs(root, res);
        return res;
    }
};
```

129. 求根节点到叶节点数字之和

```
class Solution
{
public:
    int sumNumbers(TreeNode *root)
    {
        if(!root)
            return 0;
        int res = 0;
        dfs(root, res, 0);
        return res;
    }

    void dfs(TreeNode *root, int &sum, int cur_sum)
    {
        cur_sum = cur_sum * 10 + root->val;
        if (!root->left && !root->right)
        {
            sum += cur_sum;
        }
        if (root->left)
            dfs(root->left, sum, cur_sum);
        if (root->right)
            dfs(root->right, sum, cur_sum);
    }
};
```

226. 翻转二叉树

```
TreeNode* invertTree(TreeNode* root) {
    if (root == nullptr)
        return root;
    TreeNode *temp = root->left;
    root->left = root->right;
    root->right = temp;

    root->left = invertTree(root->left);
    root->right = invertTree(root->right);

    return root;

}
```

617. 合并二叉树

```
TreeNode* mergeTrees(TreeNode* t1, TreeNode* t2) {
    if (!t1) return t2;
    if (!t2) return t1;
    TreeNode *t = new TreeNode(t1->val + t2->val);
    t->left = mergeTrees(t1->left, t2->left);
    t->right = mergeTrees(t1->right, t2->right);
    return t;
}
```

剑指 Offer 26. 树的子结构

剑指 Offer 33. 二叉搜索树的后序遍历序列
剑指 Offer 54. 二叉搜索树的第k大节点(inorder)

树和链表结合

剑指 Offer 36. 二叉搜索树与双向链表 没看懂 感觉非递归方式可能好理解点 424 收费题

```
class Solution {
public:
    Node* treeToDoublyList(Node* root)
    {
        if(root == NULL)    return NULL;
        inorder(root);
        head->left = pre;   //链表头的前驱指向链表尾
        pre->right = head;  //链表尾的后继指向链表头
        return head;
    }
private:
    Node* pre = NULL;   //前驱节点
    Node* head = NULL;  //双向链表的头节点
    void inorder(Node* cur)
    {
        if(cur == NULL)    return;
        inorder(cur->left);
        //当前前驱节点为空，说明这是双向链表的头节点（树中最左节点）
        if(pre == NULL)
            head = cur;
        //此时已有前驱，说明这是链表中的某个中间节点，将前驱的右指针指向cur
        else
            pre->right = cur;
        //把cur（当前节点）的左指针指向其前驱
        cur->left = pre;
        //当前节点成为前驱
        pre = cur;
        //递归结束后，pre指向链表的尾节点
        inorder(cur->right);
    }
};

//
Node* treeToDoublyList(Node* root)
{
    if (!root) return NULL;
    Node *head = NULL, *pre = NULL;
    stack<Node*> st;
    while (root || !st.empty()) {
        while (root) {
            st.push(root);
            root = root->left;
        }
        root = st.top(); st.pop();
        if (!head) head = root;
        if (pre) {
            pre->right = root;
            root->left = pre;
        }
        pre = root;
        root = root->right;
    }
    head->left = pre;
    pre->right = head;
    return head;
}
```

109. 有序链表转换二叉搜索树

```
class Solution
{
    TreeNode *sortedListToBST(ListNode *head, ListNode *tail)
    {
       if (head == tail) return nullptr;
       if( head->next == tail )
        {
            TreeNode *root = new TreeNode( head->val );
            return root;
        }
        ListNode *mid = head;
        ListNode *fast = head;
        // 寻找中间结点
        while(fast->next != tail && fast->next->next != tail )
        {
            mid = mid->next;
            fast = fast->next->next;
        }
        TreeNode *root = new TreeNode(mid->val);
        root->left = sortedListToBST(head, mid);
        root->right = sortedListToBST(mid->next, tail);
        return root;
    }

public:
    TreeNode *sortedListToBST(ListNode *head)
    {
        return sortedListToBST(head, nullptr);
    }
};
```

二叉树的重新构建

105. 从前序与中序遍历序列构造二叉树

```
TreeNode *buildTree(vector<int> &preorder, int preStart, int preEnd, vector<int>
&inorder, int inStart, int inEnd)
{
    if (preStart > preEnd ||inStart > inEnd )
        return nullptr;

    // 先建立根节点
    TreeNode *root = new TreeNode(preorder[preStart]);
    // 在中序遍历中找到根节点所在位置，然后就可以确定左右子树的节点数目
    int i = find(inorder.begin(), inorder.end(), preorder[preStart]) - inorder.begin();
    root->left = buildTree(preorder, preStart + 1, preStart + i - inStart, inorder,
inStart, i - 1);
    root->right = buildTree(preorder, preStart + i - inStart + 1, preEnd, inorder, i +
1, inEnd);

    return root;
}

TreeNode *buildTree(vector<int> &preorder, vector<int> &inorder)
{
    return buildTree(preorder, 0, preorder.size() - 1, inorder, 0, inorder.size() - 1);
}
```

106. 从中序与后序遍历序列构造二叉树

```
TreeNode *buildTree(vector<int> &inorder, int inStart, int inEnd, vector<int>
&postorder, int postStart, int postEnd)
{
    if (postStart > postEnd || inStart > inEnd)
        return nullptr;

    TreeNode *root = new TreeNode(postorder[postEnd]);

    int i = find(inorder.begin(), inorder.end(), postorder[postEnd]) - inorder.begin();
    // 注意推导一下下标公式就👌
    root->left = buildTree(inorder, inStart, i - 1, postorder, postStart, i+ postStart-
inStart-1 ); // 左子树
    root->right = buildTree(inorder, i+1 ,inEnd, postorder, i+postStart-inStart,
postEnd-1);  // 右子树

    return root;
}

TreeNode *buildTree(vector<int> &inorder, vector<int> &postorder)
{
    return buildTree(inorder,0, inorder.size()-1, postorder, 0, postorder.size()-1);
}
```

297. 二叉树的序列化与反序列化

```
class Codec {
    TreeNode* de(istringstream& iss)
    {
        TreeNode* root = NULL;
        string word;
        if (iss >> word && word != "null") {
            root = new TreeNode(stoi(word));
            root->left = de(iss);
            root->right = de(iss);
        }
        return root;
    }
public:

    // Encodes a tree to a single string.
    string serialize(TreeNode* root) {
        if (root == NULL)
        {
            return "null";
        }
        else
            return to_string(root->val) + " " + serialize(root->left) + " " +
serialize(root->right);



    }

    // Decodes your encoded data to tree.
    TreeNode* deserialize(string data) {
        istringstream iss(data);
        return de(iss);
    }
};
```

606. Construct String from Binary Tree

1008. Construct Binary Search Tree from Preorder Traversal

889.Construct Binary Tree from Preorder and Postorder Traversal

区间合并

56. 合并区间

```
vector<vector<int>> merge(vector<vector<int>>& intervals) {
    if (intervals.size() == 0)
    {
        return {};
    }
    // 首先将列表中的区间按左端点排序，然后将第一个区间加入到merged数组中
    // 1: 如果当前区间的左端点在merged数组中最后一个区间的右端点之后,那么他们不会重合,则直接将该区间
加入数组merged中
    // 2: 如果当前区间的左端点在merged数组中最后一个区间的右端点之前, 需要更新当前区间的右端点更新数
组中merged中最后一个区间的右端点, 取二者的最大值
    sort(intervals.begin(), intervals.end());
    vector<vector<int>> merged;
    for(int i = 0; i < intervals.size(); i++)
    {
        int left = intervals[i][0];
        int right = intervals[i][1];

        if(!merged.size() || merged.back()[1] < left) // 条件1
            merged.push_back({left, right});
        else // 条件2
            merged.back()[1] = max(merged.back()[1], right);
    }
    return merged;

}
```

57. 插入区间

```
//用一个变量 cur 来遍历区间，如果当前 cur 区间的结束位置小于要插入的区间的起始位置的话，说明没有重叠，则将 cur 区间加入结果 res 中，然后 cur 自增1。
// 每次用取两个区间起始位置的较小值，和结束位置的较大值来更新要插入的区间，然后 cur 自增1。直到 cur 越界或者没有重叠时 while 循环退出。之后将更新好的新区间加入结果 res，然后将 cur 之后的区间再加入结果 res 中即可
vector<vector<int>> insert(vector<vector<int>> &intervals, vector<int> &newInterval)
{
    vector<vector<int>> res;
    int n = intervals.size(), cur = 0;
    for (int i = 0; i < n; ++i)
    {
        if (intervals[i][1] < newInterval[0])  // 没有重叠的情况一 待插入区间在右边
        {
            res.push_back(intervals[i]);
            ++cur;
        }
        else if (intervals[i][0] > newInterval[1])  // 没有重叠的情况二 待插入区间在左边边
        {
            res.push_back(intervals[i]);
        }
        else // 有重叠 左边界去较小值 有边界取最大值
        {
            newInterval[0] = min(newInterval[0], intervals[i][0]);
            newInterval[1] = max(newInterval[1], intervals[i][1]);
        }
    }
    res.insert(res.begin() + cur, newInterval);
    return res;
}
```

986. Interval List Intersections

双堆模式

155. 最小栈

```
class MinStack {
public:
    stack<int> s1, s2; // s2 辅助栈 用来保存最小元素
    /** initialize your data structure here. */
    MinStack() {

    }

    void push(int x)
    {
        s1.push(x);
        if (s2.empty() || x <= s2.top()) s2.push(x);
    }

    void pop() {
        if (s1.top() == s2.top()) s2.pop();
        s1.pop();
    }

    int top()
    {
        return s1.top();
    }

    int getMin()
    {
         return s2.top();
    }
};

/**
 * Your MinStack object will be instantiated and called as such:
 * MinStack* obj = new MinStack();
 * obj->push(x);
 * obj->pop();
 * int param_3 = obj->top();
 * int param_4 = obj->getMin();
 */
```

295. 数据流的中位数

```
class MedianFinder {
public:
    priority_queue<long> small, large;
    /** initialize your data structure here. */
    MedianFinder() {

    }

    void addNum(int num) {
        small.push(num);
        large.push(-small.top());
        small.pop();
        if (small.size() < large.size()) {
            small.push(-large.top());
            large.pop();
        }
    }

    double findMedian() {
        return small.size() > large.size() ? small.top() : 0.5 *(small.top() - 
large.top());
    }
};
```

480. Sliding Window Median

剑指 Offer 09. 用两个栈实现队列

前K大的数模式HEAP

采用priority queue 或者 说在python 中的heapq
求top k 采用最小堆（默认）
采用最大堆的时候可以采用push 负的value

215. 数组中的第K个最大元素

难度中等1093

```
class Solution {
private:
    void heapInsert(vector<int> &arr, int index, int value)
    {
        arr[index] = value;
        while(index != 0)
        {
            int parent = (index-1) / 2; // 获取父节点
            if(arr[parent] > arr[index])
            {
                int temp = arr[parent];
                arr[parent] = arr[index];
                arr[index] = temp;

                index = parent;
            }
            else
            {
                break;
            }
        }
    }

void heapify(vector<int> &arr, int index, int size)
{
    int left = 2 * index + 1;
    while(left < size)
    {
        int smallest = left + 1 < size && arr[left+1] < arr[left] ? left +1: left ; //
smallest记录左右子树中较小的那个
        if (arr[smallest] > arr[index])
            break;

        int temp = arr[smallest];
        arr[smallest] = arr[index];
        arr[index] = temp;

        index = smallest;
        left = 2 * index + 1;
    }
}

int partition(vector<int>& nums, int left, int right)
{
    auto pivot = nums[left];
    int i = left;
    int j = right;
    while (i < j) {
        while (j > i && nums[j] >= pivot) {
            j--; // find first smaller than pivot from right
        }
        nums[i] = nums[j];
        while (i < j && nums[i] <= pivot) {
            i++; // find first larger than pivot from left
        }
        nums[j] = nums[i];
    }
    nums[i] = pivot;

    return i;
}


public:
 // 解法一：利用堆排序
    int findKthLargest(vector<int>& nums, int k) {
        if (nums.empty() || nums.size() < k)
            return 0;

        // int *heap = (int *)malloc(sizeof(int) * k);
        vector<int> heap(k , 0);

        for(int i = 0; i < k; i++)
        {
            heapInsert(heap, i, nums[i]);
        }

        for(int j = k; j < nums.size(); j++)
        {
            if (nums[j] > heap[0])
            {
                heap[0] = nums[j];
                heapify(heap, 0, k);
            }
        }
        return heap[0];
    }
    // 解法二：利用快速排序来做
    int findKthLargest(vector<int>& nums, int k) {
        int left = 0;
        int right = nums.size() - 1;
        while (left <= right) {
            int mid = partition(nums, left, right);
            int target = (nums.size() - k);
            if (mid == target) {
                return nums[mid];
            } else if (mid < target) {
                left = mid + 1;
            } else {
                right = mid - 1;
            }
        }
        return -1;
    }
};
```

```
class Solution {
public:
    int findKthLargest(vector<int>& nums, int k) {
        int left = 0, right = nums.size() - 1;
        while (true) {
            int pos = partition(nums, left, right);
            if (pos == k - 1) return nums[pos];
            if (pos > k - 1) right = pos - 1;
            else left = pos + 1;
        }
    }
    int partition(vector<int> &nums, int left, int right)
    {
        int small = left -1;
        for(int i = left; i < right; i++)
        {
            if (nums[i] > nums[right]) // 从大到小
                swap(nums[i], nums[++small]);
        }
        swap(nums[++small], nums[right]);
        return small;
    }
};
```

347. Top K Frequent Elements

```
vector<int> topKFrequent(vector<int>& nums, int k)
{
    unordered_map<int, int> m;
    priority_queue<pair<int, int>> q;
    vector<int> res;
    // 先统计每个数字出现的次数
    for (auto a : nums)
         ++m[a];
    // 然后将次数 和数字 放到大根堆里面去，然后从大根堆里面一次弹出k个数字
    for (auto it : m) q.push({it.second, it.first});
    for (int i = 0; i < k; ++i)
    {
        res.push_back(q.top().second);
        q.pop();
    }
    return res;
}
```

373. Find K Pairs with Smallest Sums

数组

未排序数组中累加和为给定值的最长子数组长度

```
int maxlenEqualK(vector<int>& arr, int k) {
    // write code here
    if(arr.size()==0)
        return 0;
    vector<int> sum(arr.size(),0);//累积数组
    sum[0]=arr[0];//初始化第一个
    unordered_map<int,int> mp;//存放前累积和 对应的下标
    mp[sum[0]]=0;
    for(int i=1;i<arr.size();i++)
    {

        sum[i]=sum[i-1]+arr[i];  //累积和
        mp[sum[i]]=i;//该累积和的下标（由于i迭代增加 所以i存放的是最大的i）

    }
    int ans=0;
    for(int i=0;i<arr.size()-1;i++)
    {
        if(mp.find(sum[i]+k)!=mp.end())  //查找当前累积和相差k的值是否存在
        {
            ans=max(ans,mp[sum[i]+k]-i);

        }

    }
    //存在直接从0项开始累积和为k的
    if(mp.find(k)!=mp.end())
        ans=max(ans,mp[k]+1);
    return ans;
}
```

1. 两数之和

```
vector<int> twoSum(vector<int>& nums, int target)
{
    unordered_map<int, int> hash; // 记录每个出现的数字的位置
    vector<int> res;

    for(int i = 0; i < nums.size(); i++)
    {
        if (hash.find(target-nums[i]) != hash.end())
        {
            res.push_back(i);
            res.push_back(hash[target-nums[i]]);
        }
        else
        {
            hash[nums[i]] = i;
        }
    }
    return res;
}
```

53. 最大子序和

```
int maxSubArray(vector<int>& nums)
{
    int res = INT_MIN;
    int cur_sum = 0;
    for(int i = 0; i < nums.size(); i++)
    {
        cur_sum += nums[i];
        res = max(res, cur_sum);
        cur_sum = cur_sum > 0 ? cur_sum:0;
    }

    return res;
}
```

88. 合并两个有序数组

```
void merge(vector<int> &nums1, int m, vector<int> &nums2, int n)
{
    int a = m - 1;
    int b = n - 1;
    int i = m + n - 1; // calculate the index of the last element of the merged array

    // go from the back by A and B and compare and put to the A element which is larger
    while (a >= 0 && b >= 0)
    {
        if (nums1[a] > nums2[b])
            nums1[i--] = nums1[a--];
        else
            nums1[i--] = nums2[b--];
    }

    // if B is longer than A just copy the rest of B to A location, otherwise no need
to do anything
    while (b >= 0)
        nums1[i--] = nums2[b--];
}
```

15. 三数之和 #Todo

```
vector<vector<int>> threeSum(vector<int>& nums)
{
    vector<vector<int>> res;
    sort(nums.begin(), nums.end());

    if (nums.empty() || nums.size() < 3)
        return res;

    for(int i = 0; i < nums.size()-2; i++)
    {
        if (i == 0 || nums[i] != nums[i-1])
        {
            //twoSum(nums, i, i+1, nums.size()-1, 0 - nums[i], res);
            int left = i + 1;
            int right = nums.size() -1;
            int target = 0 - nums[i];
            while(left < right)
            {
                if (nums[left] + nums[right] < target)
                {
                    left++;
                }
                else if (nums[left] + nums[right] > target)
                {
                    right--;
                }
                else
                {
                    if (left == i+1 || nums[left] != nums[left-1])
                    {
                        res.push_back({nums[i], nums[left], nums[right]});
                    }
                    left++;
                    right--;
                }
            }

        }
    }
    return res;
}
```

26. 删除有序数组中的重复项

```
int removeDuplicates(vector<int>& nums)
{
    if (nums.empty())
        return 0;
    int k = 1;
    for (int i = 1; i < nums.size(); i++)
    {
        if (nums[i] == nums[i-1])
        {
            continue;
        }
        else
        {
            nums[k] = nums[i];
            k++;
        }
    }
    return k;
}
```

41. 缺失的第一个正数 #todo

```
int firstMissingPositive(vector<int> &nums)
{
    //思路是把1放在数组第一个位置 nums[0]，2放在第二个位置 nums[1]，即需要把 nums[i] 放在
nums[nums[i] - 1]上，\
    // 遍历整个数组，如果 nums[i] != i + 1, 而 nums[i] 为整数且不大于n，另外 nums[i] 不等于
nums[nums[i] - 1] 的话，
    // 将两者位置调换，如果不满足上述条件直接跳过，最后再遍历一遍数组，如果对应位置上的数不正确则返回正
确的数
    int n = nums.size();
    for (int i = 0; i < n; ++i)
    {
        while (nums[i] > 0 && nums[i] <= n && nums[nums[i] - 1] != nums[i])
        {
            swap(nums[i], nums[nums[i] - 1]);
        }
    }
    for (int i = 0; i < n; ++i)
    {
        if (nums[i] != i + 1)
```

```
            return i + 1;
    }
    return n + 1;
}
```

48. 旋转图像 #todo 需要扣一下边界

```
class Solution
{
    void rotateEdge(vector<vector<int>>& matrix, int left_x, int left_y, int right_x,
int right_y)
    {
        int times = right_x - left_x;
        int temp = 0;
        for(int i = 0; i < times; i++)
        {
            temp = matrix[left_x][left_y + i];
            matrix[left_x][left_y + i] = matrix[right_x-i][left_y];
            matrix[right_x-i][left_y] = matrix[right_x][right_y-i];
            matrix[right_x][right_y-i] = matrix[left_x+i][right_y];
            matrix[left_x+i][right_y] = temp;
        }
    }

public:
    void rotate(vector<vector<int>>& matrix) {

        int left_x = 0;
        int left_y = 0;
        int right_x = matrix.size()-1;
        int right_y = matrix[0].size()- 1;
        while(left_x < right_x)
        {
            rotateEdge(matrix, left_x++, left_y++, right_x--, right_y--);
        }
    }
};
```

54. 螺旋矩阵#todo 同48 注意边界

```
vector<int> spiralOrder(vector<vector<int>>& matrix)
{
    if (matrix.empty() || matrix[0].empty())
        return {};
    int m = matrix.size(), n = matrix[0].size();
    vector<int> res;
    int up = 0, down = m - 1, left = 0, right = n - 1;
    while (true)
    {
        for (int j = left; j <= right; ++j)
            res.push_back(matrix[up][j]);
        if (++up > down)
            break;
        for (int i = up; i <= down; ++i)
            res.push_back(matrix[i][right]);
        if (--right < left)
            break;
        for (int j = right; j >= left; --j)
            res.push_back(matrix[down][j]);
        if (--down < up)
            break;
        for (int i = down; i >= up; --i)
            res.push_back(matrix[i][left]);
        if (++left > right)
            break;
    }
    return res;
}
```

59. 螺旋矩阵 II

```
vector<vector<int>> generateMatrix(int n) {
    vector<vector<int>> res(n, vector<int>(n));
    int up = 0, down = n - 1, left = 0, right = n - 1, val = 1;
    while (true)
    {
        for (int j = left; j <= right; ++j)
            res[up][j] = val++;
        if (++up > down)
            break;
        for (int i = up; i <= down; ++i)
            res[i][right] = val++;
        if (--right < left)
            break;
        for (int j = right; j >= left; --j)
            res[down][j] = val++;
        if (--down < up)
            break;
        for (int i = down; i >= up; --i)
            res[i][left] = val++;
        if (++left > right)
            break;
    }
    return res;
}
```

66. 加一

```
vector<int> plusOne(vector<int> &digits)
{
    int n = digits.size();
    for (int i = n - 1; i >= 0; --i)
    {
        if (digits[i] == 9) // 针对 多个9的情况
        {
            digits[i] = 0;
        }
        else
        {
            digits[i]++;
            return digits;
        }
    }
    digits[0] = 1;
    digits.push_back(0);
    return digits;
}
```

73. 矩阵置零

```
class Solution {
public
    // 用一个长度为m的一维数组记录各行中是否有0，用一个长度为n的一维数组记录各列中是否有0，最后直接更新matrix数组即可。这道题的要求是用O(1)的空间，那么我们就不能新建数组，我们考虑就用原数组的第一行第一列来记录各行各列是否有0.

//- 先扫描第一行第一列，如果有0，则将各自的flag设置为true
//- 然后扫描除去第一行第一列的整个数组，如果有0，则将对应的第一行和第一列的数字赋0
//- 再次遍历除去第一行第一列的整个数组，如果对应的第一行和第一列的数字有一个为0，则将当前值赋0
//- 最后根据第一行第一列的flag来更新第一行第一列
    void setZeroes(vector<vector<int> > &matrix) {
        if (matrix.empty() || matrix[0].empty()) return;
        int m = matrix.size(), n = matrix[0].size();
        bool rowZero = false, colZero = false;
        for (int i = 0; i < m; ++i) {
            if (matrix[i][0] == 0) colZero = true;
        }
        for (int i = 0; i < n; ++i) {
            if (matrix[0][i] == 0) rowZero = true;
        }
        for (int i = 1; i < m; ++i) {
            for (int j = 1; j < n; ++j) {
                if (matrix[i][j] == 0) {
                    matrix[0][j] = 0;
                    matrix[i][0] = 0;
                }
            }
        }
        for (int i = 1; i < m; ++i) {
            for (int j = 1; j < n; ++j) {
                if (matrix[0][j] == 0 || matrix[i][0] == 0) {
                    matrix[i][j] = 0;
                }
            }
        }
        if (rowZero) {
            for (int i = 0; i < n; ++i) matrix[0][i] = 0;
        }
        if (colZero) {
            for (int i = 0; i < m; ++i) matrix[i][0] = 0;
        }
    }

};
```

88. 合并两个有序数组

```
void merge(vector<int> &nums1, int m, vector<int> &nums2, int n)
{
    int a = m - 1;
    int b = n - 1;
    int i = m + n - 1; // calculate the index of the last element of the merged array

    // go from the back by A and B and compare and put to the A element which is larger
    while (a >= 0 && b >= 0)
    {
        if (nums1[a] > nums2[b])
            nums1[i--] = nums1[a--];
        else
            nums1[i--] = nums2[b--];
    }

    // if B is longer than A just copy the rest of B to A location, otherwise no need
to do anything
    while (b >= 0)
        nums1[i--] = nums2[b--];
}
```

128. 最长连续序列 #TODO

```
//使用一个集合HashSet存入所有的数字，然后遍历数组中的每个数字，如果其在集合中存在，那么将其移除，然后分别用两个变量pre和next算出其前一个数跟后一个数，然后在集合中循环查找，如果pre在集合中，那么将pre移除集合，然后pre再自减1，直至pre不在集合之中，对next采用同样的方法，那么next-pre-1就是当前数字的最长连续序列，更新res即
int longestConsecutive(vector<int>& nums)
{
    int res = 0;
    unordered_set<int> s(nums.begin(), nums.end());
    for (int val : nums) {
        if (!s.count(val)) continue;
        s.erase(val);
        int pre = val - 1, next = val + 1;
        while (s.count(pre)) s.erase(pre--);
        while (s.count(next)) s.erase(next++);
        res = max(res, next - pre - 1);
    }
    return res;
}
```

136. 只出现一次的数字#todo 位运算 还不会

```
int singleNumber(vector<int> &nums)
{
    // 如果我们把两个相同的数字异或，0与0 '异或' 是0，1与1 '异或' 也是0，那么我们会得到0。根据这个特点，我们把数组中所有的数字都 '异或' 起来，则每对相同的数字都会得0，然后最后剩下来的数字就是那个只有1次的数字
    if (nums.empty())
        return 0;
    int first = nums[0];
    for (int i = 1; i < nums.size(); i++)
    {
        first = first ^ nums[i];
    }
    return first;
}
```

150. 逆波兰表达式求值#Todo 需要敲一遍

```
// 逆波兰表达式
int evalRPN(vector<string> &tokens)
{
    if (tokens.size() == 1)
        return stoi(tokens[0]);
    stack<int> st;
    for (int i = 0; i < tokens.size(); ++i)
    {
        if (tokens[i] != "+" && tokens[i] != "-" && tokens[i] != "*" && tokens[i] !=
"/")
        {
            st.push(stoi(tokens[i]));
        }
        else
        {
            int num1 = st.top();
            st.pop();
            int num2 = st.top();
            st.pop();
            if (tokens[i] == "+")
                st.push(num2 + num1);
            if (tokens[i] == "-")
                st.push(num2 - num1);
            if (tokens[i] == "*")
                st.push(num2 * num1);
            if (tokens[i] == "/")
                st.push(num2 / num1);
        }
    }
    return st.top();
}
```

152. 乘积最大子数组

```
int maxProduct(vector<int>& nums)
{
    // 最大值 可能来自三种可能,
    // 分别记录当前的最大值和最小值
    int curMin, curMax;
    int preMax = nums[0], preMin = nums[0];
    int res = nums[0];

    for(int i = 1; i < nums.size(); i++)
    {
        curMin = min(min(preMax * nums[i], preMin * nums[i]) , nums[i]);
        curMax = max(max(preMax * nums[i], preMin * nums[i]) , nums[i]);
        preMin = curMin;
        preMax = curMax;
        res = max(res, curMax);
    }
    return res;
}
```

169. 多数元素

```
int majorityElement(vector<int>& nums)
{
    int n = nums.size();
    if (n <= 0)
        return 0;

    int time = 0;
    int cand = 0;
    for(int i = 0; i < n; i++)
    {
        if (time == 0)
        {
            cand = nums[i];
            time =1;
        }
        else if (nums[i] == cand)
        {
            time++;
        }
        else
        {
            time--;
        }
    }
    return cand;
}
```

179. 最大数

```
string largestNumber(vector<int>& nums)
{
    //对于两个数字a和b来说，如果将其都转为字符串，如果 ab > ba，则a排在前面，比如9和34，由于
934>349，所以9排在前面，
    // 再比如说 30 和3，由于 303<330，所以3排在 30 的前面。按照这种规则对原数组进行排序后，将每个数
字转化为字符串再连接起来就是最终结果
    string res;
    sort(nums.begin(), nums.end(), [](int a, int b) {
       return to_string(a) + to_string(b) > to_string(b) + to_string(a);
    });
    for (int i = 0; i < nums.size(); ++i) {
        res += to_string(nums[i]);
    }
    return res[0] == '0' ? "0" : res;
}
```

189. 旋转数组 #todo

```
class Solution {
public:
    void rotate(vector<int>& nums, int k)
    {
        if (nums.empty() || (k %= nums.size()) == 0)
            return;
        int n = nums.size();
        // todo: 记住reverse是左闭右开的
        reverse(nums.begin(), nums.begin() + n - k);
        reverse(nums.begin() + n - k, nums.end());
        reverse(nums.begin(), nums.end());
    }
};
```

215. 数组中的第K个最大元素

```
class Solution
{
private:
    void heapInsert(vector<int> &arr, int index, int value)
    {
        arr[index] = value;
        while(index != 0)
        {
            int parent = (index-1) / 2; // 获取父节点

            if(arr[parent] > arr[index])
            {
                int temp = arr[parent];
                arr[parent] = arr[index];
                arr[index] = temp;

                index = parent;
            }
            else
            {
                break;
            }
        }
    }

void heapify(vector<int> &arr, int index, int size)
{
    int left = 2 * index + 1;
    while(left < size)
    {
        int smallest = left + 1 < size && arr[left+1] < arr[left] ? left +1: left ; //
smallest记录左右子树中较小的那个
        if (arr[smallest] > arr[index])
            break;

        int temp = arr[smallest];
        arr[smallest] = arr[index];
        arr[index] = temp;

        index = smallest;
        left = 2 * index + 1;
    }
}
public:
    int findKthLargest(vector<int>& nums, int k) {
        if (nums.empty() || nums.size() < k)
            return 0;

        // int *heap = (int *)malloc(sizeof(int) * k);
        vector<int> heap(k , 0);

        for(int i = 0; i < k; i++)
        {
            heapInsert(heap, i, nums[i]);
        }

        for(int j = k; j < nums.size(); j++)
        {
            if (nums[j] > heap[0])
            {
                heap[0] = nums[j];
                heapify(heap, 0, k);
            }
        }
        return heap[0];
    }
};

class Solution {
public:
    int findKthLargest(vector<int>& nums, int k) {
        int left = 0, right = nums.size() - 1;
        while (true) {
            int pos = partition(nums, left, right);
            if (pos == k - 1) return nums[pos];
            if (pos > k - 1) right = pos - 1;
            else left = pos + 1;
        }
    }
    int partition(vector<int>& nums, int left, int right) {
        int pivot = nums[left], l = left + 1, r = right;
        while (l <= r) {
            if (nums[l] < pivot && nums[r] > pivot) {
                swap(nums[l++], nums[r--]);
            }
            if (nums[l] >= pivot) ++l;
            if (nums[r] <= pivot) --r;
        }
        swap(nums[left], nums[r]);
        return r;
    }
};
```

238. 除自身以外数组的乘积 #todo

```
//由于最终的结果都是要乘到结果 res 中，所以可以不用单独的数组来保存乘积，而是直接累积到结果 res 中，
//我们先从前面遍历一遍，将乘积的累积存入结果 res 中，然后从后面开始遍历，用到一个临时变量 right，初始化为1，然后每次不断累积，最终得到正确结果，
vector<int> productExceptSelf(vector<int>& nums)
{
    vector<int> res(nums.size());
    int k = 1;
    for(int i = 0; i < nums.size(); i++)
    {
        res[i] = k;
        k = k * nums[i]; // 此时数组存储的是除去当前元素左边的元素乘积
    }

    k = 1;
    for(int i = res.size() - 1; i >= 0; i--)
    {
        res[i] *= k;// k为该数右边的乘积。
        k = k * nums[i]; // 此时数组等于左边的 * 该数右边的。
    }
    return res;
}
```

283. 移动零

```
void moveZeroes(vector<int>& nums)
{
    if (nums.empty())
        return;
    int len = 0;
    for(int i = 0; i < nums.size(); i++)
    {
        if (nums[i] != 0)
        {
            nums[len++] = nums[i];
        }
    }
    for(int i = len; i < nums.size(); i++)
    {
        nums[i] = 0;
    }
}
```

[334. 递增的三元子序列]

```
// you are fucking genius
bool increasingTriplet(vector<int> &nums)
{
    int c1 = INT_MAX, c2 = INT_MAX;
    for (int x : nums)
    {
        if (x <= c1)
        {
            c1 = x; // c1 is min seen so far (it's a candidate for 1st element)
        }
        else if (x <= c2)  //  c1 < x <= c2
        {           // here when x > c1, i.e. x might be either c2 or c3
            c2 = x; // x is better than the current c2, store it
        }
        else  // x > c2
        {                // here when we have/had c1 < c2 already and x > c2
            return true; // the increasing subsequence of 3 elements exists
        }
    }
    return false;
}
```

[349. 两个数组的交集]

```
vector<int> intersection(vector<int>& nums1, vector<int>& nums2)
{
       vector<int> res;
        int i = 0, j = 0;
        sort(nums1.begin(), nums1.end());
        sort(nums2.begin(), nums2.end());
        while (i < nums1.size() && j < nums2.size()) {
            if (nums1[i] < nums2[j]) ++i;
            else if (nums1[i] > nums2[j]) ++j;
            else {
                if (res.empty() || res.back() != nums1[i]) {
                    res.push_back(nums1[i]);
                }
                ++i; ++j;
            }
        }
        return res;
    }
```

350. 两个数组的交集 II

```
vector<int> intersect(vector<int> &nums1, vector<int> &nums2)
{
    vector<int> res;
    int i = 0, j = 0;
    sort(nums1.begin(), nums1.end());
    sort(nums2.begin(), nums2.end());
    while (i < nums1.size() && j < nums2.size())
    {
        if (nums1[i] < nums2[j])
            ++i;
        else if (nums1[i] > nums2[j])
            ++j;
        else
        {

            res.push_back(nums1[i]);

            ++i;
            ++j;
        }
    }
    return res;
}
```

384. 打乱数组 #Todo: 洗牌算法

```
class Solution {
public:
    Solution(vector<int> nums): v(nums) {}

    /** Resets the array to its original configuration and return it. */
    vector<int> reset() {
        return v;
    }
    /** Returns a random shuffling of the array. */
    vector<int> shuffle()
    {
        vector<int> res = v;
        for (int i = 0; i < res.size(); ++i)
        {
            int t = i + rand() % (res.size() - i);
            swap(res[i], res[t]);
        }
        return res;
    }

private:
    vector<int> v;
};
```

448. 找到所有数组中消失的数字

```
//第二种方法是将nums[i]置换到其对应的位置nums[nums[i]-1]上去，比如对于没有缺失项的正确的顺序应该是[1, 2, 3, 4, 5, 6, 7, 8]，而我们现在却是[4,3,2,7,8,2,3,1]，我们需要把数字移动到正确的位置上去，比如第一个4就应该和7先交换个位置，以此类推，最后得到的顺序应该是[1, 2, 3, 4, 3, 2, 7, 8]，我们最后在对应位置检验，如果nums[i]和i+1不等，那么我们将i+1存入结果res中即可，
vector<int> findDisappearedNumbers(vector<int>& nums) {
    vector<int> res;
    for (int i = 0; i < nums.size(); ++i) {
        if (nums[i] != nums[nums[i] - 1]) {
            swap(nums[i], nums[nums[i] - 1]);
            --i;
        }
    }
    for (int i = 0; i < nums.size(); ++i) {
        if (nums[i] != i + 1) {
            res.push_back(i + 1);
        }
    }
    return res;
}
```

454. 四数相加 II

```
int fourSumCount(vector<int>& A, vector<int>& B, vector<int>& C, vector<int>& D)
{
    // 把A和B的两两之和都求出来，在 HashMap 中建立两数之和跟其出现次数之间的映射
    // 再遍历C和D中任意两个数之和，只要看哈希表存不存在这两数之和的相反数就行了
    int res = 0;
    unordered_map<int, int> hasTwoSum;
    for (int i = 0; i < A.size(); ++i)
    {
        for (int j = 0; j < B.size(); ++j)
            hasTwoSum[A[i] + B[j]] ++;
    }
    for (int i = 0; i < C.size(); ++i)
    {
        for (int j = 0; j < D.size(); ++j)
        {
            int target = -1 * (C[i] + D[j]);
            if (hasTwoSum.find(target) != hasTwoSum.end())
                res += hasTwoSum[target];
        }
    }
    return res;
}
```

523. 连续的子数组和

```
bool checkSubarraySum(vector<int>& nums, int k) {
    int n = nums.size(), sum = 0;
    // 余数和当前位置之间的映射
    unordered_map<int, int> m{{0,-1}};
    for (int i = 0; i < n; ++i) {
        sum += nums[i];
        int t = (k == 0) ? sum : (sum % k);
        if (m.count(t)) {
            if (i - m[t] > 1) return true;
        } else m[t] = i;
    }
    return false;
}
```

560. 和为K的子数组

```
int subarraySum(vector<int>& nums, int k)
{
    unordered_map<int, int> hasSum;
    int sum = 0;
    hasSum[0]=1;
    int res = 0;
    // 如果能在hasSum中能找到 说明在i 位置至少存在一个位置j 使得[0...j]的累加和为sum-k, 那么从
[j+1...i]的累加和就是k了.
    // 所以有可能不只一个j满足条件,用hasSum记录在i位置之前出现多少j满足条件
    for(int i = 0; i < nums.size(); i++)
    {
        sum += nums[i];
        if (hasSum.find(sum-k) != hasSum.end())
            res += hasSum[sum-k];
        hasSum[sum] += 1;
    }
    return res;
}
```

718. 最长重复子数组

```
int findLength(vector<int>& A, vector<int>& B) {
    int res = 0, m = A.size(), n = B.size();
    // dp[i][j] 表示数组A的前i个数字和数组B的前j个数字的最长子数组的长度
    vector<vector<int>> dp(m + 1, vector<int>(n + 1, 0));
    for (int i = 1; i <= m; ++i) {
        for (int j = 1; j <= n; ++j) {
            dp[i][j] = (A[i - 1] == B[j - 1]) ? dp[i - 1][j - 1] + 1 : 0;
            res = max(res, dp[i][j]);
        }
    }
    return res;
}
```

845. 数组中的最长山脉

难度中等190

```
int longestMountain(vector<int>& A) {
    //首先来找山峰, peak 的范围是 [1, n-1], 因为首尾两个数字都不能做山峰, 能做山峰的位置上的数必须大于其左右两边的数字, 然后分别向左右两个方向遍历, 这样就可以找到完整的山, 用 right-left+1 来更新结果 res
    int res = 0, n = A.size();
    for (int i = 1; i < n - 1; ++i) {
        if (A[i - 1] < A[i] && A[i + 1] < A[i]) {
            int left = i - 1, right = i + 1;
            while (left > 0 && A[left - 1] < A[left]) --left;
            while (right < n - 1 && A[right] > A[right + 1]) ++right;
            res = max(res, right - left + 1);
        }
    }
    return res;
}
```

915. 分割数组

```
int partitionDisjoint(vector<int>& A)
    {
        // 里使用三个变量，partitionIdx 表示分割点的位置，preMax 表示 left 中的最大值，curMax 表示当前的最大值。思路是遍历每个数字，更新当前最大值 curMax，并且判断若当前数字 A[i] 小于 preMax，说明这个数字也一定是属于 left 数组的，此时整个遍历到的区域应该都是属于 left 的，所以 preMax 要更新为 curMax，并且当前位置也就是潜在的分割点，所以 partitionIdx 更新为i。由于题目中限定了一定会有分割点，所以这种方法是可以得到正确结果的
        int partitionIdx = 0, preMax = A[0], curMax = preMax;
    for (int i = 1; i < A.size(); ++i) {
            curMax = max(curMax, A[i]);
      if (A[i] < preMax) {
        preMax = curMax;
        partitionIdx = i;
      }
    }
    return partitionIdx + 1;
    }
```

字符串(6)

7. 整数反转

```
int reverse(int x)
{
    int y = 0;
    int n;
    while (x != 0)
    {
        n = x % 10;
        if (y > INT_MAX / 10 || y < INT_MIN / 10)
        {
            return 0;
        }
        y = y * 10 + n;
        x = x / 10;
    }
    return y;
}
```

8. 字符串转换整数 (atoi)

```
int myAtoi(string str)
{
    if (str.empty()) return 0;
    int sign = 1, base = 0, i = 0, n = str.size();
    while (i < n && str[i] == ' ') ++i;
    if (i < n && (str[i] == '+' || str[i] == '-')) {
        sign = (str[i++] == '+') ? 1 : -1;
    }
    while (i < n && str[i] >= '0' && str[i] <= '9') {
        if (base > INT_MAX / 10 || (base == INT_MAX / 10 && str[i] - '0' > 7)) {
            return (sign == 1) ? INT_MAX : INT_MIN;
        }
        base = 10 * base + (str[i++] - '0');
    }
    return base * sign;
}
```

14. 最长公共前缀

```
string longestCommonPrefix(vector<string>& strs)
{
    // write code here
    int n = strs.size();
    if (n == 0)
    {
        return "";
    }
    // 如果有共同前缀的话，一定会出现在首尾两端的字符串中，所以只需要找首尾字母串的共同前缀即可
    string res = "";
    sort(strs.begin(), strs.end()); // sort the array
    string first = strs[0];         // first word
    string last = strs[n - 1];      // last word
    int limit = min(first.length(), last.length());
    for (int i = 0; i < limit; i++)
    {   // find out the longest common prefix between first and last word
        if (first[i] == last[i])
        {
            res += first[i];
        }
        else
        {
            break;
        }
    }
    return res;
}
```

28. 实现 strStr() KMP

```
class Solution {
    vector<int> getNext(string s)
    {
        int m = s.length();

        vector<int> next(m);
        next[0] = -1;
        if (m == 1)
            return next;

        next[1] = 0;
        int pos = 2;
        int cn = 0;

        while(pos < m)
        {
            if (s[pos-1] == s[cn])
            {
                next[pos++] = ++cn;
            }
            else if (cn > 0)
            {
                cn = next[cn];
            }
            else
            {
                next[pos++] =  0;
            }
        }
       return next;
    }
public:
    int strStr(string s, string m)
    {
        if (m.length() == 0)
            return 0;
        if (m.length() < 1 || s.length() < m.length())
            return -1;

        int i = 0;
        int j = 0;
        vector<int> next = getNext(m);

        while(i < s.length() && j < m.length())
        {
            if(s[i] == m[j])  // 如果当前字符相等,说明匹配上了,那么两个指针都向右走
            {
                i++;
                j++;
            } // 下面两种情况是 当前字符不相等 该如何匹配
            else if (next[j] == -1)
            {
                i++;
            }
            else
            {
                j = next[j];
            }
        }
        return j == m.length() ? i - j : -1;
    }
};
```

224. 基本计算器

```
int calculate(string s) {
    int res = 0, sign = 1, n = s.size();
    stack<int> st;
    for (int i = 0; i < n; ++i) {
        char c = s[i];
        if (c >= '0') {
            int num = 0;
            while (i < n && s[i] >= '0') {
                num = 10 * num + (s[i++] - '0');
            }
            res += sign * num;
            --i;
        } else if (c == '+') {
            sign = 1;
        } else if (c == '-') {
            sign = -1;
        } else if (c == '(') {
            st.push(res);
            st.push(sign);
            res = 0;
            sign = 1;
        } else if (c == ')') {
            res *= st.top(); st.pop();
            res += st.top(); st.pop();
        }
    }
    return res;
}
```

227. 基本计算器 II

```
int calculate(string s) {
    long res = 0, num = 0, n = s.size();
    char op = '+';
    stack<int> st;
    for (int i = 0; i < n; ++i) {
        if (s[i] >= '0') {
            num = num * 10 + s[i] - '0';
        }
        if ((s[i] < '0' && s[i] != ' ') || i == n - 1) {
            if (op == '+') st.push(num);
            if (op == '-') st.push(-num);
            if (op == '*' || op == '/') {
                int tmp = (op == '*') ? st.top() * num : st.top() / num;
                st.pop();
                st.push(tmp);
            }
            op = s[i];
            num = 0;
        }
    }
    while (!st.empty()) {
        res += st.top();
        st.pop();
    }
    return res;
}
```

415. 字符串相加_大数加法

```
string addStrings(string num1, string num2)
{
    string res = "";
        int add=0, i=num1.size()-1, j=num2.size()-1;
        while(i>=0 || j>=0 || add>0)
        {
            int cur = add;
            cur += (i >= 0 ? num1[i--] - '0' : 0);
            cur += (j >= 0 ? num2[j--] - '0' : 0);

            add = cur / 10;//用来判断是否有进位

            cur %= 10;

            res += (cur + '0');
        }
        reverse(res.begin(), res.end());//翻转字符串
        return res;
}

string addStrings(string num1, string num2) {
    string res = "";
    int m = num1.size(), n = num2.size(), i = m - 1, j = n - 1, carry = 0;
    while (i >= 0 || j >= 0) {
        int a = i >= 0 ? num1[i--] - '0' : 0;
        int b = j >= 0 ? num2[j--] - '0' : 0;
        int sum = a + b + carry;
        res.insert(res.begin(), sum % 10 + '0');
        carry = sum / 10;
    }
    return carry ? "1" + res : res;
}
```

415_2 大数减法

```
string sub(string s1, string s2) {
    int i = s1.length()-1;
    int j = s2.length()-1;
    int flag = 0;
    string ans = "";
    while (i>=0&&j >=0) {
        s1[i] = s1[i] - flag;
        if (s1[i] >= s2[j]) {
            flag = 0;
            int temp = s1[i] - s2[j];
            ans = ans + to_string(temp);
        }
        else {
            int temp = s1[i] - s2[j] + 10;
            ans = ans + to_string(temp);
            flag = 1;
        }
        i--, j--;
    }
    while (i>=0) {//处理剩余部分
        if (flag == 0)
            ans = ans + s1[i];
        else {
            int temp = s1[i] - '1';
            ans = ans + to_string(temp);
        }
        i--;
    }
    //翻转并去除前导0
    int len = ans.length();
    string ss = "";
    for (int i = len-1; i>=0; i--) {
        if (ans[i] == '0')
            continue;
        ss = ss + ans[i];
    }
    return ss;
}
```

43. 字符串相乘 大数相乘

```
string multiply(string num1, string num2)
{
    vector<int> res(num1.size()+num2.size(), 0);
    string      ans;

    for (int i = num1.size()-1; i >= 0; --i)
    {
        for (int j = num2.size()-1; j >= 0; --j)
        {
            res[i+j+1] += ((num1[i] - '0') * (num2[j] - '0'));
            res[i+j] += res[i+j+1]/10;
            res[i+j+1] %= 10;
        }
    }

    bool zero = false;
    for (auto n : res)
    {
        if (n || zero)
        {
            zero = true;
            ans += to_string(n);
        }
    }
    ans = (ans == "") ? "0" : ans;
    return ans;
}

string solve(string num1, string num2) {
    // write code here
    if (num1 == "0" || num2 == "0")
            return "0";

    vector<int> temp(num1.size() + num2.size());
    string res;
    for(int i = num1.size()-1; i >= 0; i--)
    {
        for(int j = num2.size()-1; j >= 0; j--)
        {
            temp[i+j+1] += ((num1[i] - '0') * (num2[j] - '0'));
            temp[i+j] += temp[i+j+1]/10;
            temp[i+j+1] %= 10;
        }
    }
    for(int i = 0; i < temp.size(); i++)
    {
        if (i == 0 && temp[0] == 0)
            continue;
        else
            res += to_string(temp[i]);
    }

    return res == "" ? "0" :res;
}
```

151. 翻转字符串里的单词

难度中等323

```
string reverseWords(string s)
{
    // storeIndex表示当前存储到的位置
    int storeIndex = 0, n = s.size();
    reverse(s.begin(), s.end());
    for (int i = 0; i < n; ++i) {
        if (s[i] != ' ')
        {
            if (storeIndex != 0)
                s[storeIndex++] = ' ';
            int j = i;
            while (j < n && s[j] != ' ')
                s[storeIndex++] = s[j++];
            reverse(s.begin() + storeIndex - (j - i), s.begin() + storeIndex);
            i = j;
        }
    }
    s.resize(storeIndex);
    return s;
}
```

168. Excel表列名称

```
class Solution {
public:
    string convertToTitle(int n) {
        string res = "";
        while (n) {
            if (n % 26 == 0) {
                res += 'Z';
                n -= 26;
            } else {
                res += n % 26 - 1 + 'A';
                n -= n % 26;
            }
            n /= 26;
        }
        reverse(res.begin(), res.end());
        return res;
    }
};
```

394. 字符串解码

```
string decodeString(string s)
{
    //我们也可以用迭代的方法写出来，当然需要用 stack 来辅助运算，我们用两个 stack，一个用来保存个数，一个用来保存字符串，我们遍历输入字符串，如果遇到数字，我们更新计数变量 cnt；如果遇到左括号，我们把当前 cnt 压入数字栈中，把当前t压入字符串栈中；如果遇到右括号时，我们取出数字栈中顶元素，存入变量k，然后给字符串栈的顶元素循环加上k个t字符串，然后取出顶元素存入字符串t中；如果遇到字母，我们直接加入字符串t中即可，参见代码如下：
    string t = "";
    stack<int> s_num;
    stack<string> s_str;
    int cnt = 0;
    for (int i = 0; i < s.size(); ++i) {
        if (s[i] >= '0' && s[i] <= '9') {
            cnt = 10 * cnt + s[i] - '0';
        } else if (s[i] == '[') {
            s_num.push(cnt);
            s_str.push(t);
            cnt = 0; t.clear();
        } else if (s[i] == ']') {
            int k = s_num.top(); s_num.pop();
            for (int j = 0; j < k; ++j) s_str.top() += t;
            t = s_str.top(); s_str.pop();
        } else {
            t += s[i];
        }
    }
    return s_str.empty() ? t : s_str.top();
}
```

49. 字母异位词分组

```
// 有点巧妙
vector<vector<string>> groupAnagrams(vector<string>& strs)
{
    unordered_map<string, vector<string>> mp; // 字典 排序
    // for (string s : strs)
    // {
    //     string t = s;
    //     sort(t.begin(), t.end());
    //     mp[t].push_back(s);
    // }

    for(int i = 0; i < strs.size(); i ++)
    {
        string t = strs[i];
        sort(t.begin(), t.end());
        mp[t].push_back(strs[i]);
    }
    vector<vector<string>> res;
    for (auto p : mp)
    {
        res.push_back(p.second);
    }
    return res;
}
```

125. 验证回文串

```
bool isPalindrome(string s)
{
    int n = s.size();
    int left = 0, right = n - 1;
    while (left < right)
    {
        while (left < right && !isalnum(s[left]))
            ++left;
        while (left < right && !isalnum(s[right]))
```

```
            --right;
        if (left < right)
        {
            if (tolower(s[left]) != tolower(s[right]))
                return false;
            ++left;
            --right;
        }
    }
    return true;
}
```

其他

135. 分发糖果

```
int candy(vector<int>& ratings)
{
    // 先来看看两遍遍历的解法，首先初始化每个人一个糖果，然后这个算法需要遍历两遍，第一遍从左向右遍历，如果右边的小盆友的等级高，等加一个糖果，这样保证了一个方向上高等级的糖果多。然后再从右向左遍历一遍，如果相邻两个左边的等级高，而左边的糖果又少的话，则左边糖果数为右边糖果数加一。最后再把所有小盆友的糖果数都加起来返回即可。
    int res = 0, n = ratings.size();
    vector<int> nums(n, 1);
    for (int i = 0; i < n - 1; ++i)
    {
        if (ratings[i + 1] > ratings[i])
            nums[i + 1] = nums[i] + 1;
    }
    for (int i = n - 1; i > 0; --i)
    {
        if (ratings[i - 1] > ratings[i])
            nums[i - 1] = max(nums[i - 1], nums[i] + 1);
    }
    for (int num : nums)
        res += num;
    return res;
}
```

208. 实现 Trie (前缀树)

```
class Trie {
public:

    struct TrieNode {
        unordered_map<char, TrieNode*> map;
        bool endOfWord = false;
    };

    TrieNode *root;
    /** Initialize your data structure here. */
    Trie() {
        root = new TrieNode();
    }

    /** Inserts a word into the trie. */
    void insert(string word) {
        TrieNode *curr = root;
        for (int i=0; i<word.size(); i++) {
            if (curr->map.find(word[i]) == curr->map.end()) {
                TrieNode *newNode = new TrieNode();
                curr->map[word[i]] = newNode;
                curr = newNode;
            } else {
                curr = curr->map[word[i]];
            }
        }
        curr->endOfWord = true;;
    }

    /** Returns if the word is in the trie. */
    bool search(string word) {
        TrieNode *curr = root;
        for (int i=0; i<word.size(); i++) {
            if (curr->map.find(word[i]) == curr->map.end()) {
                return false;
            } else {
                curr = curr->map[word[i]];
            }
        }
        return curr->endOfWord;
    }

    /** Returns if there is any word in the trie that starts with the given prefix. */
    bool startsWith(string prefix) {
        TrieNode *curr = root;
        for (int i=0; i<prefix.size(); i++) {
            if (curr->map.find(prefix[i]) == curr->map.end()) {
                return false;
            } else {
                curr = curr->map[prefix[i]];
            }
        }
        return true;
    }
};
```

146. LRU 缓存机制

```
class LRUCache{
private:
    int cap;
    list<pair<int, int>> l;
    unordered_map<int, list<pair<int, int>>::iterator> m;
public:
    LRUCache(int capacity) {
        cap = capacity;
    }

    // get 相对简单些，我们在 HashMap 中查找给定的 key，若不存在直接返回 -1。如果存在则将此项移到顶部，这里我们使用 C++ STL 中的函数 splice，专门移动链表中的一个或若干个结点到某个特定的位置，这里我们就只移动 key 对应的迭代器到列表的开头，然后返回 value
    int get(int key) {
        auto it = m.find(key);
        if (it == m.end()) return -1;
        l.splice(l.begin(), l, it->second);
        return it->second->second;
    }
    // 我们也是现在 HashMap 中查找给定的 key，如果存在就删掉原有项，并在顶部插入新来项，然后判断是否溢出，若溢出则删掉底部项(最不常用项)
    void put(int key, int value) {
        auto it = m.find(key);
        if (it != m.end()) l.erase(it->second);
        l.push_front(make_pair(key, value));
        m[key] = l.begin();
        if (m.size() > cap) {
            int k = l.rbegin()->first;
            l.pop_back();
            m.erase(k);
        }
    }
};
class Solution {
public:
    /**
     * lru design
     * @param operators int整型vector<vector<>> the ops
     * @param k int整型 the k
     * @return int整型vector
     */
    vector<int>keys;
    vector<int>values;
    vector<int>res;
    void set(int key,int value,int k)
    {
        if(keys.size()<k)
        {
            keys.push_back(key);
            values.push_back(value);
        }
        else
        {
            keys.erase(keys.begin());
            values.erase(values.begin());
            keys.push_back(key);
            values.push_back(value);
        }
    }
    void get(int key)
    {
        int i;
        for(i=0;i<keys.size();i++)
        {
            if(keys[i]==key)
            {
                break;
            }
        }
        if(i==keys.size())
        {
            res.push_back(-1);
            return ;
        }
        int value=values[i];
        keys.erase(keys.begin()+i);
        values.erase(values.begin()+i);
        keys.push_back(key);
        values.push_back(value);
        res.push_back(value);
    }
    vector<int> LRU(vector<vector<int> >& operators, int k) {
        // write code here
        for(int i=0;i<operators.size();i++)
        {
            if(operators[i][0]==1)
            {
                int key=operators[i][1];
                int v=operators[i][2];
                set(key,v,k);
            }
            else
            {
                get(operators[i][1]);
            }
        }
        return res;
    }
};
```

470. 用 Rand7() 实现 Rand10()

```
int rand10()
{
    while (true) {
        int num = (rand7() - 1) * 7 + rand7();
        if (num <= 40) return num % 10 + 1;
    }
}
```

丢旗子

```
//核心思路：每次扔的位置都是最佳的，i个棋子扔time次，第1次时，如果碎了，向下可以探测“i-1个棋子扔time-1次”层；如果没碎，向上可以探测“i个棋子扔time-1次”层。上下层数加当前1层即为i个棋子扔time次能探测的最大层数
class Solution {
public:
    int solve(int n, int k) {
        if(n <= 1 || k == 1) return n; //层数小于等于1和棋子数等于1的情况
        int best = log2(n) + 1; //棋子足够条件下扔的最小次数
        if(k >= best) return best; //如果棋子数足够则返回最小次数
        int dp[k + 1]; //用来记录扔1~k个棋子能探测的最大层数
        for(int &i: dp) i = 1; //无论有几个棋子扔1次都只能探测一层
        for(int time = 2;;time++) { //从扔第2次开始（前面初始化dp数组时扔了第1次）
            for(int i = k; i >= 2; i--) { //从k个棋子开始刷新dp数组（倒过来刷新省去记录临时值的步骤）
                dp[i] = dp[i] + dp[i-1] + 1; //关键一步
                if(dp[i] >= n) return time; //如果探测层数大于n，则返回扔的次数
            }
            dp[1] = time; //1个棋子扔time次最多探测time层
        }
    }
};
```

进制转换

```
string solve(int M, int N) {
    // write code here
    if(M == 0) return "0";//如果M=0就直接返回
    bool flag = false;//记录是不是负数
    if(M < 0){
        //如果是负数flag=true，M 取相反数
        flag = true;
        M = -M;
    }
    string res = "";//返回最终的结果
    string jz = "0123456789ABCDEF";//对应进制的某一位
    while(M != 0){//就对应转换为N进制的逆序样子
        res += jz[M % N];
        M /= N;
    }
    reverse(res.begin(),res.end());//逆序一下才是对应的N进制
    if(flag) res.insert(0,"-");//如果是负数就在头位置插入一个-号
    return res;
}
```

括号序列

```
bool isValid(string s) {
        // write code here
        stack<char> parentheses;
        for (int i = 0; i < s.size(); ++i) {
            if (s[i] == '(' || s[i] == '[' || s[i] == '{') parentheses.push(s[i]);
            else {
                if (parentheses.empty()) return false;
                if (s[i] == ')' && parentheses.top() != '(') return false;
                if (s[i] == ']' && parentheses.top() != '[') return false;
                if (s[i] == '}' && parentheses.top() != '{') return false;
                parentheses.pop();
            }
        }
        return parentheses.empty();
    }
```

剑指offer 刷题记录

剑指 Offer 03. 数组中重复的数字

```
int findRepeatNumber(vector<int>& nums)
{
    for(int i = 0; i < nums.size(); ++i)
    {
        while(nums[i] != i)      //当前元素不等于下标
        {
            if(nums[i] == nums[nums[i]])
                return nums[i];
            swap(nums[i],nums[nums[i]]);
        }
    }
    return -1;
}
```

剑指 Offer 04. 二维数组中的查找

```
bool findNumberIn2DArray(vector<vector<int>>& matrix, int target)
{
    if (matrix.empty())
        return false;

    int row = matrix.size() - 1;
    int col = 0;

    while (row >= 0 && col < matrix[0].size())
    {
        if (matrix[row][col] == target)
            return true;
        else if (matrix[row][col] < target)
            col++;
        else
            row--;
    }
    return false;
}
```

剑指 Offer 05. 替换空格

```
string replaceSpace(string s) {
    string res;   //存储结果

    for(auto &c : s){   //遍历原字符串
        if(c == ' '){
            res.push_back('%');
            res.push_back('2');
            res.push_back('0');
        }
        else{
            res.push_back(c);
        }
    }
    return res;
}
```

剑指 Offer 06. 从尾到头打印链表

```
vector<int> reversePrint(ListNode* head)
{
    vector<int> res;
    stack<int> s;
    //入栈
    while(head)
    {
        s.push(head->val);
        head = head->next;
    }
    //出栈
    while(!s.empty())
    {
        res.push_back(s.top());
        s.pop();
    }
    return res;

}
```

剑指 Offer 07. 重建二叉树

```
TreeNode *buildTree(vector<int> &preorder, int preStart, int preEnd, vector<int>
&inorder, int inStart, int inEnd)
{
    if (preStart > preEnd ||inStart > inEnd )
        return nullptr;

    // 先建立根节点
    TreeNode *root = new TreeNode(preorder[preStart]);
    // 在中序遍历中找到根节点所在位置，然后就可以确定左右子树的节点数目
    int i = find(inorder.begin(), inorder.end(), preorder[preStart]) - inorder.begin();
    root->left = buildTree(preorder, preStart + 1, preStart + i - inStart, inorder,
inStart, i - 1);
    root->right = buildTree(preorder, preStart + i - inStart + 1, preEnd, inorder, i +
1, inEnd);

    return root;
}

TreeNode* buildTree(vector<int>& preorder, vector<int>& inorder)
{
    return buildTree(preorder, 0, preorder.size() - 1, inorder, 0, inorder.size() - 1);
}
```

剑指 Offer 09. 用两个栈实现队列

```

class CQueue {
public:
    stack<int> stack1,stack2;
    CQueue() {

        while (!stack1.empty()) {
            stack1.pop();
        }
        while (!stack2.empty()) {
            stack2.pop();
        }



    }

    void appendTail(int value) {
        stack1.push(value);
    }

    int deleteHead()
    {
        // 如果第二个栈为空
        if (stack2.empty())
        {
            while (!stack1.empty())
            {
                stack2.push(stack1.top());
                stack1.pop();
            }
        }
        if (stack2.empty())
        {
            return -1;
        }
        else
        {
            int deleteItem = stack2.top();
            stack2.pop();
            return deleteItem;
        }

    }
};

/**
 * Your CQueue object will be instantiated and called as such:
 * CQueue* obj = new CQueue();
 * obj->appendTail(value);
 * int param_2 = obj->deleteHead();
 */
```

剑指 Offer 10- I. 斐波那契数列

```
int fib(int n) {
    if(n <= 1)  return n;
    int a = 0, b = 1, c = 0;
    for(int i = 2; i <= n; ++i)     //从f(2)开始计算到f(n)
    {
        c = (a + b) % 1000000007;
        a = b;
        b = c;
    }
    return c;
}
```

剑指 Offer 10- II. 青蛙跳台阶问题

```
int numWays(int n)
{
    if (n <= 1)
        return 1;
    vector<int> dp(n, 0);
    dp[0] = 1;
    dp[1] = 2;
    for (int i = 2; i < n; i++)
    {
        dp[i] = (dp[i - 1] + dp[i - 2]) % 1000000007;
    }
    return dp[n - 1];
}
```

剑指 Offer 11. 旋转数组的最小数字

```
int minArray(vector<int>& nums)
{
    //，若数组没有旋转或者旋转点在左半段的时候，中间值是一定小于右边界值的，所以要去左半边继续搜索
    int left = 0, right = nums.size() - 1;
    while (left < right)
    {
        int mid = left + (right - left) / 2;
        if (nums[mid] > nums[right])  // 如果中间值比右边的大，则在右半边寻找最小值
            left = mid + 1;
        else if (nums[mid] < nums[right])// 如果中间值比右边的小，则在左半边寻找最小值，注意这里
mid可能就是最小值，所right =mid;
            right = mid;
        else
            --right;
    }
    return nums[right];
}
```

剑指 Offer 12. 矩阵中的路径

```
bool dfs(vector<vector<char>>& board, string &word, int i, int j, int step)
{
    if (i < 0 || i >= board.size() || j < 0 || j >= board[0].size() || word[step] !=
board[i][j] || step >= word.size())
        return false;

    if (step == word.size()-1 && word[step] == board[i][j])
        return true;
    char temp = board[i][j];
    board[i][j] = '1';
    bool res =  dfs(board, word, i + 1, j, step+1) ||
                dfs(board, word, i - 1, j, step+1) ||
                dfs(board, word, i, j + 1, step+1) ||
                dfs(board, word, i, j - 1, step+1);
    board[i][j] = temp;
    return res;
}

bool exist(vector<vector<char>>& board, string word)
{
    if (board.empty())
        return false;

    // return dfs(board, word, 0, 0, 0);

    for(int i = 0; i < board.size(); i++)
    {
        for(int j = 0; j < board[0].size(); j++)
        {
            if (dfs(board, word, i, j, 0))
                return true;
        }
    }
    return false;

}
```

剑指 Offer 13. 机器人的运动范围

```
int getsum(int num)
{
    int sum = 0;
    while(num)
    {
        sum += num % 10;
        num /= 10;

    }
    return sum;
}
int dfs(int m, int n, int k, int i, int j, vector<int> &visted)
{
    if (i < 0 || i >= m || j < 0 || j >= n || visted[i * n +j] == 1 || getsum(i) +
getsum(j) > k)
```

```
        return 0;
    visted[i * n +j] = 1;
    int res = 1+ dfs(m, n, k, i + 1, j, visted) +
              dfs(m, n, k, i - 1, j, visted) +
              dfs(m, n, k, i, j + 1, visted) +
              dfs(m, n, k, i, j - 1, visted);
    return res;
}


int movingCount(int m, int n, int k)
{
    vector<int> visted(m * n, 0);
    return dfs(m, n, k, 0, 0, visted);
}
```

剑指 Offer 15. 二进制中1的个数

```
class Solution {
public:
   // 若 n \& 1 = 0n&1=0 , 则 nn 二进制 最右一位 为 00 ;
   // 若 n \& 1 = 1n&1=1 , 则 nn 二进制 最右一位 为 11 。
    //判断 nn 最右一位是否为 11 , 根据结果计数。
  // 将 nn 右移一位(本题要求把数字 nn 看作无符号数,因此使用 无符号右移 操作)。
    int hammingWeight(uint32_t n) {
        int count = 0;
        while (n) {
            n &= (n - 1);
            count++;
        }
        return count;
    }
};
```

剑指 Offer 26. 树的子结构

```
class Solution {
public:
    bool isSubStructure(TreeNode* A, TreeNode* B) {
        if(!A || !B) return false;
        if(issame(A, B)) return true;
        return isSubStructure(A->left, B) || isSubStructure(A->right, B);
    }
    bool issame(TreeNode* a, TreeNode* b)
    {
        if(!b) return true;
        if(!a || a->val != b->val) return false;
        return issame(a->left, b->left) && issame(a->right, b->right);
    }
};
```

剑指 Offer 27. 二叉树的镜像

```
class Solution
{
    void dfs(TreeNode* root)
    {
        if (root == nullptr)
            return ;
        TreeNode *temp = root->left;
        root->left = root->right;
        root->right = temp;

        dfs(root->left);
        dfs(root->right);



    }
public:
    TreeNode* mirrorTree(TreeNode* root)
    {
        dfs(root);
        return root;
    }
};
```

剑指 Offer 31. 栈的压入、弹出序列

```
bool validateStackSequences(vector<int>& pushed, vector<int>& popped) {
        stack<int> st;
        int i = 0;
        for (int num : pushed) {
          st.push(num);
          while (!st.empty() && st.top() == popped[i]) {
            st.pop();
            ++i;
          }
        }
        return st.empty();
    }
```

剑指 Offer 28. 对称的二叉树

```
class Solution {

    bool dfs(TreeNode *root1, TreeNode *root2)
    {

        if (root1 == nullptr && root2 == nullptr)
            return true;

        if (root1 == nullptr || root2 == nullptr)
            return false;

        if (root1 && root2)
            return root1->val == root2->val && dfs(root1->left, root2->right) &&
dfs(root1->right, root2->left);
        else
            return false;
    }

public:
    bool isSymmetric(TreeNode* root)
    {
        if (!root)
            return true;
        return dfs(root, root);
    }
};
```

剑指 Offer 33. 二叉搜索树的后序遍历序列

```
class Solution {
public:
    // 根据二叉树的定义来看 最后一个是根节点 前面的序列中必须存在一个拐点 拐点前后的值 要么均大于根 要么均小于根
    vector<int> res;
    bool verifyPostorder(vector<int>& postorder) {
        res = postorder;
        return dfs(0, postorder.size() - 1);
    }
    bool dfs(int l, int r)
    {
        if(l >= r) return true;//退出条件
        int root = res[r];//最后一个点是根结点
        int k = l;//从最左边开始
        while(k < r && res[k] < root) k++;//符合左子树的点
        for(int i = k; i < r; i++)//此时的k是右子树的第一个点
        {
            if(res[i] < root)//如果右子树小于根，说明不符合
            return false;
        }
        return dfs(l, k - 1) && dfs(k, r - 1);//逐步缩小范围
    }
};
```

剑指 Offer 30. 包含min函数的栈

```
class MinStack {
public:
    stack<int> s1;
    stack<int> s2;
    /** initialize your data structure here. */
    MinStack() {

    }

    void push(int x) {
        s1.push(x);
      if (s2.empty() || x <= min())  s2.push(x);
    }

    void pop() {
        if (s1.top() == min())  s2.pop();
      s1.pop();
    }

    int top() {
         return s1.top();
    }

    int min() {
        return s2.top();
    }
};
```

剑指 Offer 41. 数据流中的中位数

```
class MedianFinder {
public:
    //因为我们要 取得中位数，所以只要关注前一半数字的最小值和后一段数字的最大值，采用堆来进行维护时间复杂度为O(logn)O(logn)。
    // 那么B在构造时用小顶堆+元素取负的方法将其转换为大顶堆
    priority_queue<long> small, large;
    /** initialize your data structure here. */
    MedianFinder() {

    }

    void addNum(int num) {
        small.push(num);
        large.push(-small.top());
        small.pop();
        if (small.size() < large.size())
        {
            small.push(-large.top());
            large.pop();
        }
    }

    double findMedian() {
        return small.size() > large.size() ? small.top() : 0.5 * (small.top() -
large.top());
    }
};
```

剑指 Offer 51. 数组中的逆序对

```
// 对数组 nums[left...mid]  [mid+1...right]进行合并
int merge(vector<int> &nums, int left, int mid, int right)
{
    vector<int> help;
    int index = 0;
    int i = left;
    int j = mid + 1;
    int count = 0;

    while(i <= mid && j <= right)
    {
        if (nums[i] > nums[j])
        {
            count += mid -i +1;
            help.push_back(nums[j]);
            j++;
        }
        else
        {
            help.push_back(nums[i]);
            i++;
        }
    }

    while (i <= mid)
    {
        help.push_back(nums[i++]);
    }
    while (j <= right)
    {
        help.push_back(nums[j++]);
    }
    for (int index = 0; index < help.size(); index++)
    {
        nums[left + index] = help[index];
    }

    return count;
}

int mergeSort(vector<int> &nums, int left, int right)
{
    if (left == right)
    {
        return 0;
    }
    int mid = left + (right - left) /2 ;
    int leftCount = mergeSort(nums, left, mid);
    int rightCount = mergeSort(nums, mid + 1, right);
    int count = merge(nums, left, mid, right);
    return leftCount + rightCount + count;
}


int reversePairs(vector<int>& nums)
{
    if (nums.empty())
        return 0;

    return mergeSort(nums, 0, nums.size() - 1);
}
```