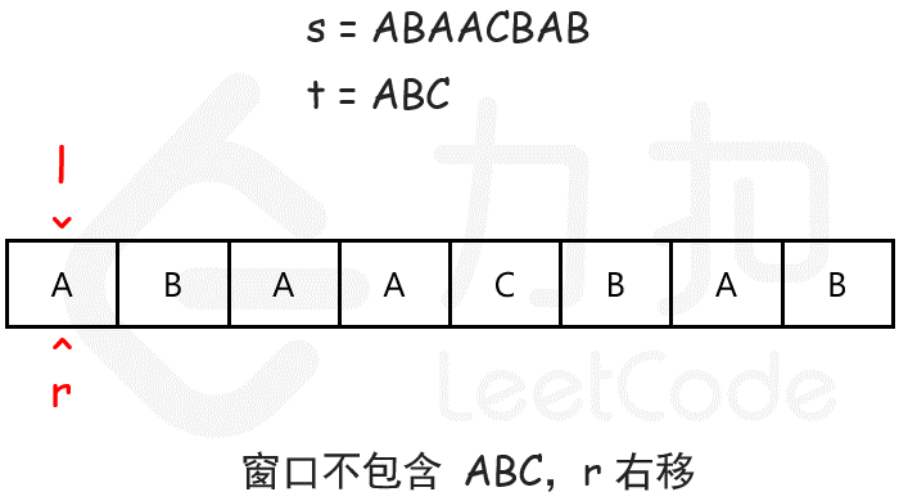
LeetCode 刷题记录

滑动窗口问题 (7)

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. Longest Substring Without Repeating Characters](#) 不会

```
 // 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)
{
    vector<int> m(128, -1);
    //  res 用来记录最长无重复子串的长度，left 指向该无重复子串左边的起始位置的前一个
    int res = 0, left = -1;
    // [left...i] 维护了一个窗口
    for (int i = 0; i < s.size(); ++i)
    {
        left = max(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. Minimum Window Substring

```
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. Sliding Window Maximum

```
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. Longest Repeating Character Replacement

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. Find All Anagrams in a String

```
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. Permutation in String (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 (7)

todo: 11和42的区别

11. Container With Most Water

```
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. Trapping Rain Water

```
// 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 (5)

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. Maximal Rectangle

```
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;
    }
};
```

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 (6)

33. Search in Rotated Sorted Array #todo

```
int search(vector<int>& nums, int target)
{
    //可以得出出规律，如果中间的数小于最右边的数，则右半段是有序的，若中间数大于最右边数，则左半段是有序的，我们只要在有序的半段里用首尾两个数组来判断目标值是否在这一区域内，这样就可以确定保留哪半边了
    int l = 0, r = nums.size()-1;
    while (l <= r) {
        int mid = (l+r) / 2;
        if (target == nums[mid])
            return mid;
        // there exists rotation; the middle element is in the left part of the array
        if (nums[mid] > nums[r])
        {
            if (target < nums[mid] && target >= nums[l])
                r = mid - 1;
            else
                l = mid + 1;
        }
        // there exists rotation; the middle element is in the right part of the array
        else if (nums[mid] < nums[l])
        {
            if (target > nums[mid] && target <= nums[r])
                l = mid + 1;
            else
                r = mid - 1;
        }
        // there is no rotation; just like normal binary search
        else
        {
            if (target < nums[mid])
                r = mid - 1;
```

```
            else
                l = mid + 1;
        }
    }
    return -1;
}
```

34. Find First and Last Position of Element in Sorted Array

```
class Solution
{
    // 找第一个大于等于target的值得位置
    int getFirstK(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; // 这个地方 right和left都可以
    }

    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; // 这个地方 right和left都可以
    }

    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; // 这个地方 right和left都可以
    }

public:
    vector<int> searchRange(vector<int> &nums, int target)
    {
        int start = getFirstK(nums, target);
        if (start == nums.size() || nums[start] != target)
            return {-1, -1};
        return {start, getFirstK(nums, target + 1) - 1};
    }
};
```

81. Search in Rotated Sorted Array II #todo

```
bool search(vector<int> &nums, int target)
{
    // 如果中间的数小于最右边的数，则右半段是有序的，若中间数大于最右边数，则左半段是有序的。
    int n = nums.size(), left = 0, right = n - 1;
    while (left <= right)
    {
        int mid = (left + right) / 2;
        if (nums[mid] == target)
            return true;
        if (nums[mid] < nums[right])
        {
            if (nums[mid] < target && nums[right] >= target)
                left = mid + 1;
            else
                right = mid - 1;
        }
        else if (nums[mid] > nums[right])
        {
            if (nums[left] <= target && nums[mid] > target)
                right = mid - 1;
            else
                left = mid + 1;
        }
        else
```

```
            --right;
    }
    return false;
}
```

153. Find Minimum in Rotated Sorted Array while条件 懵逼了

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

154. Find Minimum in Rotated Sorted Array II while条件 懵逼了

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

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;
}
```

[378. Kth Smallest Element in a Sorted Matrix]

```
//整个二维数组中 matrix[0][0]matrix[0][0] 为最小值，matrix[n - 1][n - 1]matrix[n-1][n-1] 为最大值，现在我们将其分别记作 ll 和 rr。
```

可以发现一个性质：任取一个数 midmid 满足 l\leq mid \leq rl≤mid≤r，那么矩阵中不大于 midmid 的数，肯定全部分布在矩阵的左上角。初始位置在 matrix[n - 1][0]matrix[n-1][0]（即左下角；设当前位置为 matrix[i][j]matrix[i][j]。若 matrix[i][j] \leq midmatrix[i][j]≤mid，则将当前所在列的不大于 midmid 的数的数量（即 i + 1i+1）累加到答案中，并向右移动，否则向上移动；不断移动直到走出格子为止。

```
bool check(vector<vector<int>> &matrix, int mid, int k, int n)
{
    int i = n - 1;
```

```
    int j = 0;
    int num = 0;
    while (i >= 0 && j < n)
    {
        if (matrix[i][j] <= mid)
        {
            num += i + 1;
            j++;
        }
        else
        {
            i--;
        }
    }
    return num >= k;
}
int kthSmallest(vector<vector<int>> &matrix, int k)
{
    int n = matrix.size();
    int left = matrix[0][0];
    int right = matrix[n - 1][n - 1];
    while (left < right)
    {
        int mid = left + (right - left) / 2;
        if (check(matrix, mid, k, n))
        {
            right = mid;
        }
        else
        {
            left = mid + 1;
        }
    }
    return left;
}
```

排序 (3)

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);
    }
};
```

快速排序

[75. Sort Colors](#) 快排partion

```
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--]);
        }
    }
};
```

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

```
// 利用快速排序原理
void wiggleSort(vector<int> &nums)
{
    auto midptr = nums.begin() + nums.size() / 2;
    nth_element(nums.begin(), midptr, nums.end());
    int mid = *midptr;

    int i = 0, j = 0, k = nums.size() - 1;
    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 (14)

K路归并

21. Merge Two Sorted Lists

```
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;

        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. Merge k Sorted Lists

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

25. Reverse Nodes in k-Group (hard 这题还不会) #todo

```
class Solution {
public:
     ListNode *reverseKGroup(ListNode *head, int k)
    {
        ListNode *dummy = new ListNode(-1), *pre = dummy, *cur = pre;
        dummy->next = head;
        int num = 0;
        while (cur = cur->next)
            ++num;
        while (num >= k)
        {
            cur = pre->next;
            for (int i = 1; i < k; ++i)
            {
                ListNode *t = cur->next;
                cur->next = t->next;
                t->next = pre->next;
                pre->next = t;
            }
            pre = cur;
            num -= k;
        }
        return dummy->next;
    }
};
```

快慢指针

61. Rotate List

```
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

```
// 解法一   (https://github.com/grandyang/leetcode/issues/143)
void reorderList(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;
    ListNode *last = mid, *pre = NULL;
    while (last)
    {
        ListNode *next = last->next;
        last->next = pre;
        pre = last;
        last = next;
    }
    while (head && pre)
    {
        ListNode *next = head->next;
        head->next = pre;
```

```
        pre = pre->next;
        head->next->next = next;
        head = next;
    }
}

// 解法二
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;
    }
    st.top()->next = NULL;
}
```

链表排序

86. Partition List # 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 *dummy = new ListNode(-1);
        ListNode *newDummy = new ListNode(-1);
        dummy->next = head;
        ListNode *cur = dummy, *p = newDummy;
        while (cur->next)
```

```
        {
            if (cur->next->val < x)
            {
                p->next = cur->next;
                p = p->next;
                cur->next = cur->next->next;
                p->next = NULL;
            }
            else
            {
                cur = cur->next;
            }
        }
        p->next = dummy->next;
        return newDummy->next;
    }
};
```

[147. Insertion Sort List](#) 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;

    ListNode *fast = head->next, *slow = head;
    // 找到中间结点
    while (fast && fast->next)
    {
```

```
        slow = slow->next;
        fast = fast->next->next;
    }

    fast = slow->next;
    slow->next = nullptr;
    return merge(sortList(head), sortList(fast));
}
```

原地链表翻转

25. Reverse Nodes in k-Group

```
// 每k组翻转
ListNode *reverseKGroup(ListNode *head, int k)
{
    ListNode *dummy = new ListNode(-1), *pre = dummy, *cur = pre;
    dummy->next = head;
    int num = -1;
    // 统计链表长度,因为是从假的头结点开始所以num从-1开始
    while (cur)
    {
        ++num;
        cur = cur->next;
    }
    while (num >= k)
    {
        cur = pre->next;
        for (int i = 1; i < k; ++i)
        {
            // 这个地方相当于头插法
            ListNode *t = cur->next;
            cur->next = t->next;
            t->next = pre->next;
            pre->next = t;
        }
        pre = cur;
        num -= k;
    }
    return dummy->next;
}
```

92. Reverse Linked List 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. Reverse Linked List

```
// 非递归
ListNode *reverseList(ListNode *head)
{
    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;
```

```
}
```

234. Palindrome Linked List

```
bool isPalindrome(ListNode *head)
{
    if (head == nullptr || head->next == nullptr)
        return true;
    ListNode *slow = head;
    ListNode *fast = head;
    while (fast->next != nullptr && fast->next->next != nullptr)
    {
        slow = slow->next;
        fast = fast->next->next;
    }
    slow->next = reverseList(slow->next);
    slow = slow->next;
    fast = head;
    while (slow)
    {
        if (fast->val != slow->val)
            return false;
        fast = fast->next;
        slow = slow->next;
    }
    return true;
}
ListNode *reverseList(ListNode *head)
{
    ListNode *pre = nullptr, *cur=head;
    while (cur)
    {
        ListNode *temp = cur->next;
        cur->next = pre;
        pre = cur;
        cur = temp;
    }
    return pre;
}
```

动态规划 (29)

1.坐标型动态规划 (5)

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. Unique Paths

```
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. Unique Paths II

```
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. Minimum Path Sum

```
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 (7)

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. Longest Valid Parentheses

```
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

```
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. Jump Game](#)

```
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;
}
```

132. Palindrome Partitioning 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. Word Break

```
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. House Robber

```
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];
}
```

300. Longest Increasing Subsequence

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

    int n = nums.size();
    // dp[i] 表示以nums[i]结尾的最长公共子序列
    vector<int> dp(n, 0);
    int max2 = INT_MIN;
    for (int i = 0; i < nums.size(); i++)
    {
        dp[i] = 1;
        for (int j = 0; j < i; j++)
        {
            if (nums[i] > nums[j])
            {
                dp[i] = max(dp[i], dp[j] + 1);
            }
```

```
        }
        max2 = max(dp[i], max2);
    }
    return max2;
}
```

3.双序列动态规划 (7)

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. Regular Expression Matching]

```
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
            {
                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. Wildcard Matching

```
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
    // 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. Edit Distance #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 = 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;
// 分别对应的增加，删除，修改操作
            }
        }
    }
    return dp[m][n];
}
```

97. Interleaving String

```
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. Distinct Subsequences

```
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. Minimum ASCII Delete Sum for Two Strings

```
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. Longest Common Subsequence

```
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 （1）

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 （5）

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]

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);
    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);
            }
        }
    }
    return dp[amount] > amount ? -1 : dp[amount];
}
```

518. Coin Change 2 注意和322的区别

todo: **还不懂后面的这个去掉当前硬币的这种情况**　还有空间优化需要看

```
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)
    {
        dp[i][0] = 1;
        for (int j = 1; 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

```
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];
}
```

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:
    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 ] ]


    }
};
```

6.区间型动态规划 （4）

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. Longest Palindromic Substring

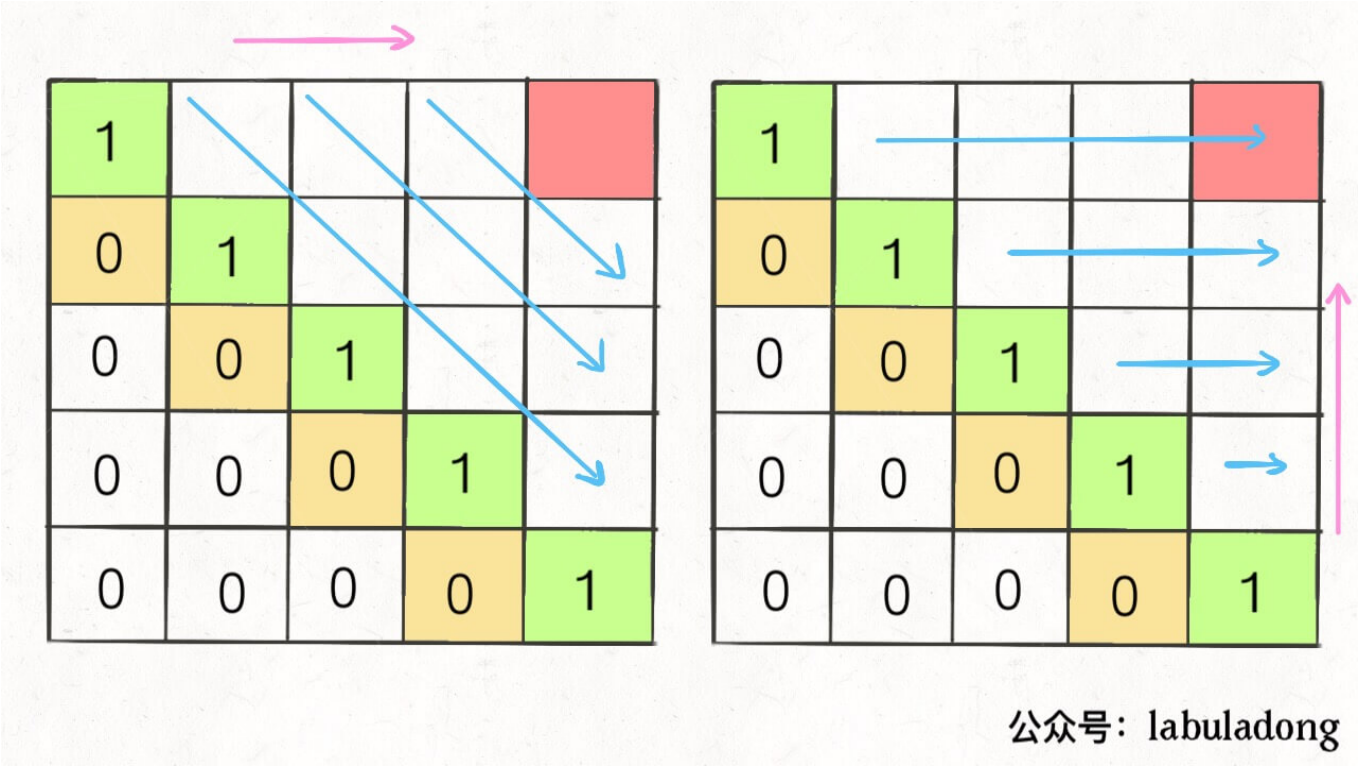
```
string longestPalindrome(string s) // todo: 时间上还得优化
{
    if (s.empty())
        return "";
    int len = 0;  // 记录最长回文子串的长度
    int left = 0, right = 0; // 记录最长回文子串的左右边界
    // dp[i][j] 表示 s[i...j]上是否为回文子串
    vector<vector<bool>> dp(s.size(), vector<bool>(s.size(), false));、
    for (int i = 0;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

132. Palindrome Partitioning 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. Longest Palindromic Subsequence 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. Palindromic Substrings]

```
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 （1）

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;
    }
};
```

分治 （还不会 先总结着）

395. Longest Substring with At Least K Repeating Characters

深度优先搜索 (8)

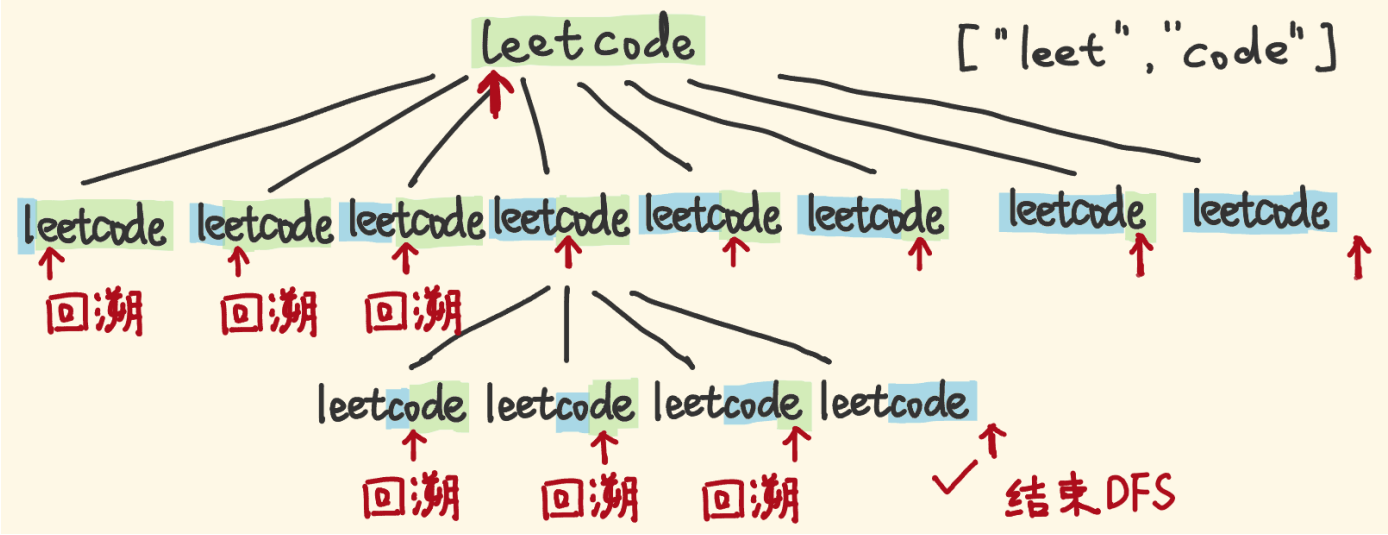
经典DFS

```
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. Word Search

```
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;
    }
};
```

139. Word Break




```
 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;
}
```

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;
}
```

[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 (6)

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. Minimum Depth of Binary Tree

```
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;
}
```

[127. Word Ladder I](#) 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)
{
    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;
}
```

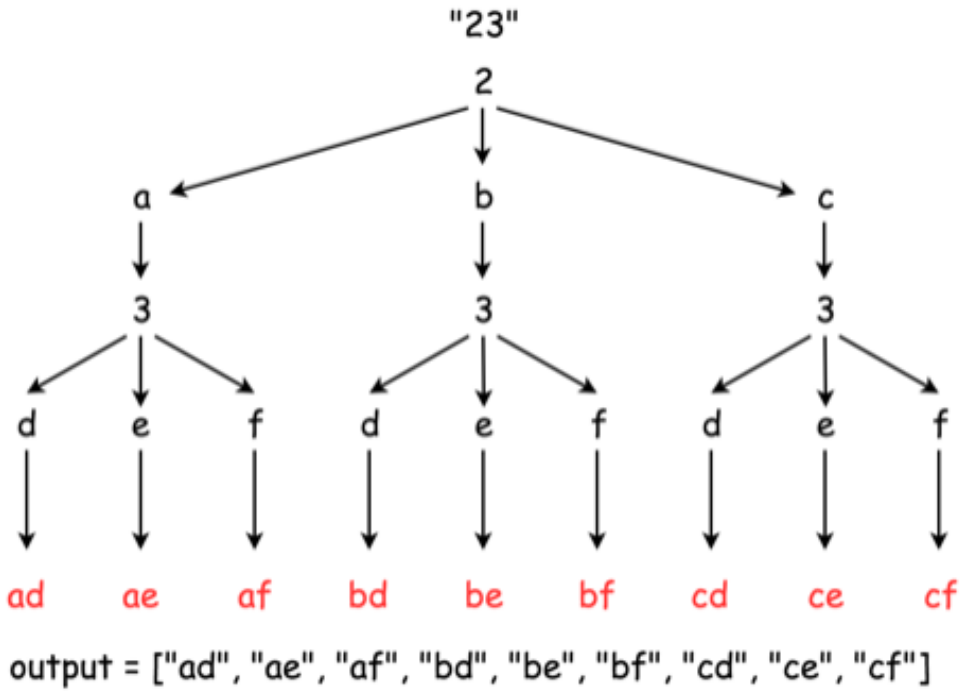
207. Course Schedule【拓扑排序】

```
 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;
}
```

297. Serialize and Deserialize Binary Tree

回溯 (13)

[17. Letter Combinations of a Phone Number]




```
//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. Generate Parentheses

```
class Solution
{
public:
    void backtrack(vector<string> &res, string &cur, int open, int close, int n)
    {
        if (cur.size() == 2 * n)
        {
            res.push_back(cur);
            return;
        }
        if (open < n)
        {
            cur.push_back('(');
            backtrack(res, cur, open + 1, close, n);
            cur.pop_back();
        }

        if (close < open)
        {
            cur.push_back(')');
            backtrack(res, cur, open, close + 1, n);
            cur.pop_back();
        }
    }

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

39. Combination Sum

```
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++)
        {
            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()); // 为什么要先排序
        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. Combination Sum II

```
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;
    }
};
```

46. Permutations

```
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. Permutations II

```
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. Subsets

```
class Solution
{
    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]);
        dfs(nums, i+1, temp, res);
        temp.pop_back();
```

```
        dfs(nums, i+1, temp, res);
    }
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;
    }
};
```

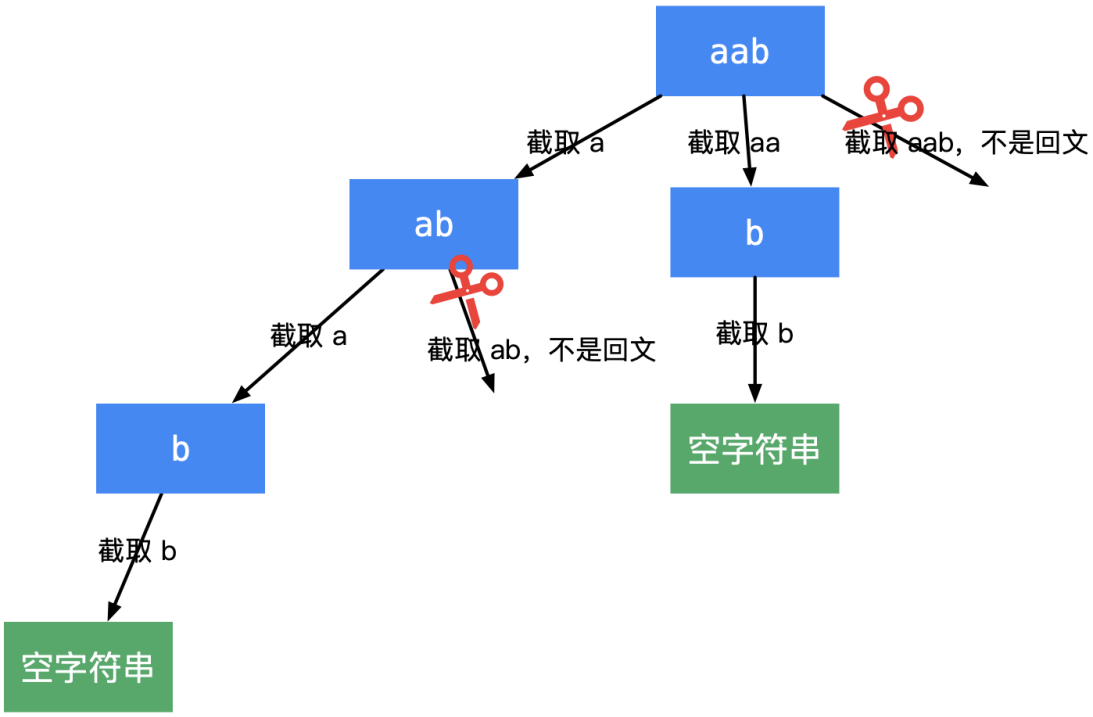
90. Subsets II

```
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. Restore IP Addresses

```
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 表示s 从index开始的子串拆成回文的方式
    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++)
        {
            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));
                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. Word Break](#) 实际上是动态规划

```
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 wordBreak_1(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;
    }

    /**
```

```
     * 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

```
class Solution
{
public:

    /**
     * 解法一：
     * 先扫一遍wordDict数组，看有没有单词可以当s的开头，那么我们可以发现cat和cats都可以，比如我们先选了cat，那么此时s就变成了 "sanddog"
     *
     */
    vector<string> wordBreak(string s, vector<string> &wordDict)
    {
        unordered_map<string, vector<string>> m;
        return dfs(s, wordDict, m);
    }

    // dfs表示 s在wordDict中的拆分方式， 遍历wordDict中每一个字符串,是否是s的开头,如果是
    // 在s中去掉该字符串,剩下的继续递归
    vector<string> dfs(string s, vector<string> &wordDict, unordered_map<string, vector<string>> &m)
    {
```

```
        if (m.count(s))
            return m[s];
        if (s.empty())
            return {""};
        vector<string> res;
        for (string word : wordDict)
        {
            if (s.substr(0, word.size()) != word)
                continue;
            vector<string> rem = helper(s.substr(word.size()), wordDict, m);
            for (string str : rem)
            {
                res.push_back(word + (str.empty() ? "" : " ") + str);
            }
        }
        return m[s] = res;
    }
};
```

二叉树 (3)

[114. Flatten Binary Tree to Linked List](#) todo: 没看懂

```
void flatten(TreeNode* root)
{
  while(root)
  {
      if (root->left && root->right)
      {
          TreeNode *t = root->left;
          while(t->right)
          {
              t = t->right;
          }
          t->right = root->right;
      }

      if (root->left)
      {
          root->right = root->left;

      }
      root->left = nullptr;
      root = root->right;
  }
}
```

235. Lowest Common Ancestor of a Binary Search Tree

```
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. Lowest Common Ancestor of a Binary Tree

```
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;
}
```

树的DFS (8)

通常采用递归

100. Same Tree

```
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. Symmetric Tree

```
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. Maximum Depth of Binary Tree

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

110. Balanced Binary Tree

```
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;
}
```

112. Path Sum

```
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. Path Sum 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;
    }
};
```

124. Binary Tree Maximum Path Sum 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;
    }
};
```

437. Path Sum III

```
class Solution
{
public:
    // 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
    void dfs(TreeNode *node, int sum, int curSum, vector<TreeNode *>& path, int &res)
    {
        if (node == nullptr)
```

```
            return;

        curSum += node->val;
        path.push_back(node);
        if (curSum == sum)
            res++;
        int t = curSum;
        for (int i = 0; i < path.size() - 1; ++i) {
            t -= path[i]->val;
            if (t == sum) ++res;
        }
        dfs(node->left, sum, curSum, path, res);
        dfs(node->right, sum, curSum, path, res);
        path.pop_back();

    }

    int pathSum(TreeNode *root, int sum)
    {
        int res = 0;
        vector<TreeNode*> path;
        dfs(root, sum, 0, path, res);
        return res;
    }
};
```

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 (3)

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);
    }
};

// 二叉树中序遍历非递归还不是很熟悉
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. Convert Sorted List to Binary Search Tree

```
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, NULL);
    }
};
```

树的重新构建

105. Construct Binary Tree from Preorder and Inorder Traversal

```
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. Construct Binary Tree from Inorder and Postorder Traversal

```
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);
}
```

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 (3)

56. Merge Intervals

```
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. Insert Interval

```
//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. Min Stack]

[295 Find-Median-from-Data-Stream]

[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. Kth Largest Element in an Array]

[347. Top K Frequent Elements]

[373. Find K Pairs with Smallest Sums]

#