- 层序遍历 #102
- 翻转二叉树 #226
- 对称二叉树 #101
- 相同的树 #100
- 二叉树的最大深度 #104
- 二叉树的最小深度 #111
- 完全二叉树的节点个数 #222
- 判断平衡二叉树 #110
- 二叉树的所有路径 #257
- 左子叶之和 #404
- 树左下角的值 #513
- 路经总和 #112
- 路经总和II #113
- 中序和后序构造树 #106
- 前序和中序构造树 #105
- 最大二叉树 #654
- 合并二叉树 #617
- BST的搜索 #700
- 验证BST #98
- BST最小绝对差 #530
- BST的众数 #501
- 二叉树的最近公共祖先 #236
- 二叉搜索树的最近公共祖先 #235
- BST插入 #701
- BST的删除 #450
- 修剪BST #669
- 有序数组转为平衡BST #108
- BST转为累加树 #538

☆回溯

- 递归要素
- 二叉树的最大路径和 #124
- 组合 #77
- 组合总和III #216
- 电话号码的字母组合 #17
- 组合总和 #39
- 组合总和II #40
- 分割回文串 #131
- 复原IP地址 #93
- 子集 #78
- 子集II #90
- 递增子序列 #491
- 全排列 #46
- 全排列II #47
- 重新安排行程 #332
- N皇后 #51
- 解数独 #37

☆贪心

- 分发饼干 #455
- 摆动序列 #376
- 最大子数组和 #53
- 买卖股票的最佳时机II #122
- 跳跃游戏 #55
- 跳跃游戏II #45
- K次取反后最大化的数组和 #1005
- 加油站 #134
- 分发糖果 #135
- 柠檬水找零 #860
- 根据身高重建队列 #406

☆目录

> 二分查找：704, 33
>
> 双指针：27, 26, 977, O-5, 209, 15, 18
>
> 链表：203, 707, 206, 24, 19, 160, 142
>
> 哈希表：242, 349, 202, 1, 454, 383
>
> 字符串：344, 541, 151, 58, 28, 459
>
> 栈与队列：232, 225, 20, 1047, 150, 347, 239
>
> 排序：O-51, 315, 215
>
> 二叉树：144, 94, 145, 102, 226, 101, 100, 104, 111, 222, 110, 257, 404, 513,
>
> 112, 113, 106, 105, 654, 617, 700, 98, 530, 501, 236, 235, 701, 450, 669, 108, 538
>
> 回溯：124, 77, 216, 17, 39, 40, 131, 93, 78, 90, 491, 46, 47, 332, 51, 37
>
> 贪心：455, 376, 53, 122, 55, 45, 1005, 134, 135, 860, 406, 452, 435, 763, 56, 738, 968
>
> 动态规划：509, 70, 62, 63, 343, 96, 416, 1049, 494, 474, 518, 377, 70, 322, 279, 139, 198, 213,
>
> 337, 121, 122, 123, 188, 309, 714, 300, 674, 718, 1143, 1035, 53, 392, 115, 49, 72, 647, 516
>
> 其他：59

☆二分查找

模板 #704

```cpp
class Solution {
public:
    int search(vector<int>& nums, int target) {
        if (nums.size() == 0) return -1;

        // 1. 左开右闭
        // 2. [left, right)
        // 3. while( < )
        // 4. right = mid
        int left = 0, right = nums.size();
        while (left < right) {
            int mid = left + (right - left) / 2;
            if (target < nums[mid]) {  // target 比较都是 < 或者 > , 没有 =
                right = mid;
            } else if (target > nums[mid]) {
                left = mid + 1;
            } else {
                return mid;
            }
        }

        return -1;
    }
};
```

搜索旋转排序数组 #33

```
class Solution {
public:
    int search(vector<int>& nums, int target) {
        if (nums.size() == 0) return -1;

        int left = 0, right = nums.size();
        while(left < right) {
            int mid = left + (right - left) / 2;
            // 旋转后的数组有一个很重要的特点是，随便以一个节点为界，两边至少有一个是有序的
            // 跟普通二分查找不同的是，如果 target < nums[mid] ，不能直接对 right 赋值
            // 不先判断 target 与 nums[mid] 的大小，而是先判断哪一边是有序的，然后再判断
target 和 nums[mid] 的大小关系
            if (target == nums[mid]) {
                return mid;
            } else if (nums[left] < nums[mid]) {
                if (nums[left] <= target && target < nums[mid]) {
                    right = mid;
                } else {
                    left = mid + 1;
                }
            } else{
                if (nums[mid] <= target && target <= nums[right - 1]) {
                    left  = mid + 1;
                } else {
                    right = mid;
                }

            }
        }

        return -1;
    }
};
```

搜索旋转排序数组II #81

```
class Solution {
public:
    bool search(vector<int>& nums, int target) {
        if (nums.size() == 0) return -1;

        int left = 0, right = nums.size();
        while (left < right) {
            int mid = left + (right - left) / 2;

            if (nums[mid] == target) return true;
            // 中点跟左端点相同时，因为有重复元素，无法判断左右哪一个是有序的
            // Z比如 nums=[3,1,2,3,3,3,3], target=2
            // 让左端点右移一个，减少一个重复项的干扰
            if (nums[left] == nums[mid]) {
                left++;
            } else if (nums[left] < nums[mid]) { // 左边有序，而且这里一定要 else
if，否则上边 left++ 后有可能越界
                if (nums[left] <= target && target < nums[mid]) {
```

```
                    right = mid;
                } else {
                    left = mid + 1;
                }
            } else {
                if (nums[mid] <= target && target <= nums[right - 1]) {
                    left = mid + 1;
                } else {
                    right = mid;
                }
            }
        }

        return false;
    }
};
```

☆双指针

移除元素 #27

```
// 时间复杂度：O(n)
// 空间复杂度：O(1)
class Solution {
public:
    int removeElement(vector<int>& nums, int val) {
        if (nums.size() == 0) return 0;

        int slow = 0;
        for (int fast = 0; fast < nums.size(); fast++) {
            if (nums[fast] != val) {
                nums[slow++] = nums[fast];
            }
        }

        return slow;
    }
};
```

删除有序数组重复项 #26

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

        // slow 在外边，fast 用 for 遍历，这是隐藏的双层循环遍历方式
        int slow = 0;
        for (int fast = 0; fast < nums.size(); fast++) {
            // 不判断的，即是遇到重复的了，要跳过
            // 注意一定是 ++slow，而不是 slow++，否则会多覆盖前一个元素
            // nums[++slow] = nums[fast] 是把每个重复的第一个元素进行往前复制，后边的都跳过了
            if (nums[slow] != nums[fast]) {
                nums[++slow] = nums[fast];
```

```
            }
        }

        return slow + 1;
    }
};
```

有序数组的平方 #977

```
class Solution {
public:
    vector<int> sortedSquares(vector<int>& nums) {
        if (nums.size() == 0) return {};
        if (nums.size() == 1) return {nums[0] * nums[0]};

        vector<int> res(nums.size(), 0);
        int index = nums.size() - 1;
        int left = 0, right = nums.size() - 1;
        while (left <= right && index >= 0) {
            if (abs(nums[left]) < abs(nums[right])) {
                res[index--] = nums[right] * nums[right];
                right--;
            } else {
                res[index--] = nums[left] * nums[left];
                left++;
            }
        }

        return res;
    }
};
```

替换空格 #O-5

```
class Solution {
public:
    string replaceSpace(string s) {
        if (s.size() == 0) return "";
        int cnt = 0;
        for (char ch : s) {
            if (ch == ' ') cnt++;
        }
        int old_size = s.size();
        int new_size = old_size + 2 * cnt;
        s.resize(new_size); // 注意是 2* 不是 3*

        // string res(n, ' '); // 可以这样定义某个长度的字符串
        // 下边用空间O(1)的方法，也就是原数组基础上去做
        int i = old_size - 1, j = new_size - 1;
        while (i >= 0 && j >= 0) {
            if (s[i] != ' ') {
                s[j--] = s[i--];
            } else {
                s[j] = '0';
                s[j - 1] = '2';
                s[j - 2] = '%';
```

```
                j = j - 3;
                i--;
            }
        }

        return s;
    }
};
```

长度最小的子数组 #209

滑动窗口

```
class Solution {
public:
    int minSubArrayLen(int target, vector<int>& nums) {
        if (nums.size() == 1) return nums[0] >= target ? 1 : 0;

        int res = INT_MAX;
        int i = 0;
        int sum = 0;
        for (int j = 0; j < nums.size(); j++) {
            sum += nums[j];
            // 一定是 while，否则比如 [2,3,1,2,4,3] 会越过 [1,2,4] 这一部分
            while (sum >= target) {
                int length = j - i + 1;
                if (res > length) res = length;
                sum -= nums[i]; // 一定要记得 sum 减去 nums[i]，而且要在 i++ 之前
                i++;
            }
        }

        return res == INT_MAX ? 0 : res; // 记得还要判断一下是否是初始值
    }
};
```

三数之和 #15

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

        // 1. 一定要排序
        // 2. nums[i] = a, nums[left] = b, nums[right] = c
        // 3. i,left,right 都要分别去重
        sort(nums.begin(), nums.end());
        for (int i = 0; i < nums.size(); i++) {
            if (nums[i] > 0) return res;
            if (i > 0 && nums[i] == nums[i - 1]) continue;  // nums[i] == nums[i
+ 1] 是错误的去重

            int left = i + 1, right = nums.size() - 1;  // left = i + 1
            while (left < right) {
                if (nums[i] + nums[left] + nums[right] > 0) {
```

```
                    right--;
                    while (left < right && nums[right] == nums[right + 1])
right--;
                } else if (nums[i] + nums[left] + nums[right] < 0) {
                    left++;
                    while (left < right && nums[left] == nums[left - 1]) left++;
                } else {
                    res.push_back(vector<int>{nums[i], nums[left],
nums[right]});
                    // // 去重逻辑应该放在找到一个三元组之后 ??
                    // while (right > left && nums[right] == nums[right - 1])
right--;
                    // while (right > left && nums[left] == nums[left + 1])
left++;
                    right--;
                    left++;
                    while (left < right && nums[right] == nums[right + 1])
right--;
                    while (left < right && nums[left] == nums[left - 1]) left++;
                }
            }
        }

        return res;
    }
};
```

四数之和 #18

```
class Solution {
public:
    vector<vector<int>> fourSum(vector<int>& nums, int target) {
        vector<vector<int>> res;
        if (nums.size() < 4) return res;

        sort(nums.begin(), nums.end());
        // 千万不要忘记去重，四个都要去重，尤其是 i,j 容易忘记
        for (int i = 0; i < nums.size(); i++) {
            if (i > 0 && nums[i] == nums[i - 1]) continue;
            for (int j = i + 1; j < nums.size(); j++) {
                if (j > i + 1 && nums[j] == nums[j - 1]) continue;
                int left = j + 1, right = nums.size() - 1;
                while (left < right) {
                    if ((long) nums[i] + nums[j] + nums[left] + nums[right] >
target) { // 注意整型溢出
                        right--;
                        while (left < right && nums[right] == nums[right + 1])
right--;
                    } else if((long) nums[i] + nums[j] - target < - nums[left] -
nums[right]) {
                        left++;
                        while (left < right && nums[left] == nums[left - 1])
left++;
                    } else {
                        res.push_back(vector<int>{nums[i], nums[j], nums[left],
nums[right]});
                        right--;
```

```
                        left++;
                        while (left < right && nums[right] == nums[right + 1])
right--;
                        while (left < right && nums[left] == nums[left - 1])
left++;
                    }
                }
            }
        }

        return res;
    }
};
```

☆链表

定义

```
struct ListNode {
    int val;
    ListNode *next;
    ListNode(int x) : val(x), next(NULL) {}
};
```

删除元素 #203

```
/**
 * Definition for singly-linked list.
 * struct ListNode {
 *     int val;
 *     ListNode *next;
 *     ListNode() : val(0), next(nullptr) {}
 *     ListNode(int x) : val(x), next(nullptr) {}
 *     ListNode(int x, ListNode *next) : val(x), next(next) {}
 * };
 */
class Solution {
public:
    ListNode* removeElements(ListNode* head, int val) {
        ListNode* dummyHead = new ListNode(0);
        dummyHead->next = head;
        ListNode* cur = dummyHead;

        // 一定是去判断 cur->next
        while (cur->next != nullptr) {
            if (cur->next->val == val) {
                ListNode* tmp = cur->next;
                cur->next = cur->next->next;
                delete tmp;
            } else {
                cur = cur->next;
            }
        }
```

```
        // 别忘了删除虚拟头结点，而且要返回新的头结点也就是 dummyHead->next，因为之前的
head 有可能已经被删除了
        head = dummyHead->next;
        delete dummyHead;
        return head;
    }
};
```

插入元素 #707

```
void addAtIndex(int index, int val) {
    if(index > _size) return;

    ListNode* newNode = new ListNode(val);
    ListNode* cur = _dummyHead;
    while(index--) {
        cur = cur->next;
    }
    newNode->next = cur->next;
    cur->next = newNode;
}
```

翻转链表 #206

```
class Solution {
public:
    ListNode* reverseList(ListNode* head) {
        if (!head) return nullptr;

        ListNode* pre = nullptr;  // 可以定义为 nullptr
        ListNode* cur = head;
        ListNode* tmp;  // 提到循环前边定义
        while (cur != nullptr) {  // 判断cur, 而不是 cur->next
            tmp = cur->next;
            cur->next = pre;
            pre = cur;
            cur = tmp;
        }

        return pre;
    }
};
```

两两交换链表中的节点 #24

```
class Solution {
public:
    ListNode* swapPairs(ListNode* head) {
        ListNode* dummyHead = new ListNode(0);
        dummyHead->next = head;

        // 不用定义两个 pre 和 cur，只要一个cur
        // 定义一个虚拟头结点，让 cur 指向它，方便后续操作
        // 注意 while 循环里要判断 cur->next 以及 cur->next->next 都不为空
```

```
        ListNode* cur = dummyHead;
        while (cur->next != nullptr && cur->next->next != nullptr) {
            ListNode* tmp1 = cur->next;
            ListNode* tmp2 = cur->next->next->next;

            cur->next = cur->next->next;
            cur->next->next = tmp1;
            cur->next->next->next = tmp2;
            cur = cur->next->next;
            // delete tmp1;  // 不要删除啊，这不是删除元素，tmp是有具体指向某个元素的
        }

        return dummyHead->next;

    }
};
```

删除链表倒数第N个数 #19

```
class Solution {
public:
    ListNode* removeNthFromEnd(ListNode* head, int n) {
        ListNode* dummyNode = new ListNode(0);
        dummyNode->next = head;
        // 快慢指针，也是双指针的一种
        ListNode* slow = dummyNode;
        ListNode* fast = dummyNode;

        while (fast != nullptr && n--) {
            fast = fast->next;
        }
        while (fast->next != nullptr) {
            slow = slow->next;
            fast = fast->next;
        }

        ListNode* tmp = slow->next;
        slow->next = slow->next->next;
        delete tmp;

        return dummyNode->next;
    }
};
```

链表相交 #160

```
class Solution {
public:
    ListNode *getIntersectionNode(ListNode *headA, ListNode *headB) {
        int lenA = 0, lenB = 0;
        ListNode* curA = headA;
        ListNode* curB = headB;

        while (curA != nullptr) {
            lenA++;
            curA = curA->next;
```

```
        }
        while (curB != nullptr) {
            lenB++;
            curB = curB->next;
        }

        // 千万不要忘了 curA,curB 要重新指向，因为此时已经指向末尾了
        curA = headA;
        curB = headB;
        if (lenB > lenA) {
            swap(lenA, lenB);
            swap(curA, curB);
        }

        int cnt = lenA - lenB;
        while (cnt--) {
            curA = curA->next;
        }
        while (curA != nullptr) {
            if (curA == curB) {
                return curA;
            } else {
                curA = curA->next;
                curB = curB->next;
            }
        }

        return nullptr;
    }
};
```

环形链表II #142

```
class Solution {
public:
    ListNode *detectCycle(ListNode *head) {
        ListNode* slow = head;
        ListNode* fast = head;
        // 1. 快慢指针，快指针两步两步的走，慢指针一步一步的走
        //    快指针相当于每一次都靠近慢指针一步，如果有环的话，一定可以相遇
        // 2. 如果判断有环，那么找到环的入口，是比较难的
        //    假设head到入环口距离x个节点，入环口到相遇节点距离y，环的距离减去y的距离为z
        //    那么slow走了 x+y，fast走了 x+n*(y+z)+y = 2*(x+y)，得 x = (n-1)*(z+y)
+ z
        //    n=1的时候，x=z，让一个节点index1指向头结点，一个节点index2指向相遇节点，两个
节点同时走，相遇节点就是入环口
        //    n>1的时候，与 n=1 的情况相同，只不过 index2 多走了 n-1 圈而已
        // 3. 一定注意，while 里的判断一定要有 fast，因为你判断 fast->next 不为空之前，当
然要先判断 fast 不为空
        while (fast && fast->next) {
            slow = slow->next;
            fast = fast->next->next;
            if (slow == fast) {
                ListNode* index1 = head;
                ListNode* index2 = slow;
                while (index1 != index2) {
                    index1 = index1->next;
```

```
                    index2 = index2->next;
                }
                return index1;
            }
        }

        return nullptr;
    }
};
```

☆哈希表

字母异位词 #242

```
class Solution {
public:
    bool isAnagram(string s, string t) {
        if (s.size() != t.size()) return false;

        // 必须要初始化，int a[n] = {0}，是可以全部初始化为0的
        // 但如果 int a[n] = {1}，只会初始化为 1,0,0,0...
        int cnt[26] = {0};
        for (int i = 0; i < s.size(); i++) {
            cnt[s[i] - 'a']++;
            cnt[t[i] - 'a']--;
        }
        for (int c : cnt) {
            if (c != 0) return false;
        }

        return true;
    }
};
```

两个数组的交集 #349

```
class Solution {
public:
    vector<int> intersection(vector<int>& nums1, vector<int>& nums2) {
        unordered_set<int> set(nums1.begin(), nums1.end());
        unordered_set<int> res;

        for (int num :nums2) {
            if (set.find(num) != set.end()) {
                res.insert(num);
            }
        }

        return vector<int>(res.begin(), res.end());
    }
};
```

快乐数 #202

```cpp
class Solution {
public:
    bool isHappy(int n) {
        unordered_set<int> uset;
        // 让 n 不停的变化，定义一个 sum，让 sum 保存中间的结果
        // 定义一个 set，保存中间所有出现过的结果，一旦再次出现即将出现无限循环
        while (1) {

            int sum = 0;
            while (n) {
                sum += (n % 10) * (n % 10);
                n = n / 10;
            }

            if (sum == 1) return true;
            if (uset.find(sum) != uset.end()) {
                return false;
            } else {
                uset.insert(sum);
            }

            n = sum;  //  千万别忘了最后要把 n 重新赋值为中间处理的结果
        }
        return false;
    }
};
```

两数之和 #1

```cpp
class Solution {
public:
    vector<int> twoSum(vector<int>& nums, int target) {
        // 如果不需要下标，只需要找到两个数，那么可以用 set，而且只能遍历一遍
        // 此题目需要下标，所以必须用 map 去做
        unordered_map<int, int> map;

        for (int i = 0; i < nums.size(); i++) {
            // 技巧，定义 auto iter，因为后边要用 iter->second
            auto iter = map.find(target - nums[i]);
            if (iter != map.end()) {
                return {i, iter->second};
            } else {
                map[nums[i]] = i;
            }
        }

        return {};
    }
};
```

四数相加II #454

```
class Solution {
public:
    int fourSumCount(vector<int>& nums1, vector<int>& nums2, vector<int>& nums3,
vector<int>& nums4) {
        // 默认初始化为0 ??
        unordered_map<int, int> umap;

        for (int a : nums1) {
            for (int b : nums2) {
                umap[a + b]++;
            }
        }

        int cnt = 0;
        for (int c : nums3) {
            for (int d : nums4) {
                if (umap.find(0 - (c + d)) != umap.end()) {
                    cnt += umap[0 - (c + d)];
                }
            }
        }

        return cnt;
    }
};
```

赎金信 #383

```
// 数组
class Solution {
public:
    bool canConstruct(string ransomNote, string magazine) {
        // map 还是比较占空间，可以使用 数组 去保存
        int record[26] = {0};

        for (char ch : magazine) {
            record[ch - 'a']++;
        }
        for (char ch : ransomNote) {
            record[ch - 'a']--;
            // 最好是判断在一个循环里，更方便
            // 只有第一个是对于 magazine 遍历，才可以判断 <0，反过来得重新循环 record 才可
以判断
            if (record[ch - 'a'] < 0) return false;
        }

        return true;
    }
};
// map
class Solution {
public:
    bool canConstruct(string ransomNote, string magazine) {
        unordered_map<char, int> umap;
```

```
        for (char ch : ransomNote) {
            umap[ch]++;
        }
        for (char ch :magazine) {
            if (umap.find(ch) != umap.end()) {
                umap[ch]--;
            }
            // 注意这里 else 是不可以直接判断为 false，因为magazine可能有很多用不到的字符
        }
        for (auto iter = umap.begin(); iter != umap.end(); iter++) {
            // 这里一定是判断 >0 ， 而不是 ==0，因为有可能magazine很多相匹配的字符，会使它的值 <0
            if (iter->second > 0) return false;
        }

        return true;
    }
};
```

☆字符串

反转字符串 #344

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

反转字符串II #541

```
class Solution {
public:
    void reverse(string& s, int begin, int end) {
        if (begin >= end) return;
        // 当下标用 start 和 end 计算的时候，要千万注意边界怎么表示的
        // 这里 i < (end - begin) / 2 就是错的，一定要在前边加上 begin! ! !
        for (int i = begin, j = end -  1; i < begin + (end - begin) / 2; i++, j--) {
            swap(s[i], s[j]);
        }
    }
    string reverseStr(string s, int k) {
        for (int i = 0; i < s.size(); i += 2 * k) {
            if (i + k - 1 < s.size() - 1) {
                reverse(s, i, i + k);
            } else {
                reverse(s, i, s.size());
            }
```

```
        }
        return s;
    }
};
```

翻转字符串里的单词 #151

左旋转字符串 #O-58

```
// 1.
class Solution {
public:
    string reverseLeftWords(string s, int n) {
        // 不开辟空间，在原字符串上改
        // s.beigin()指向第一个元素，s.end()指向最后一个元素的下一个
        reverse(s.begin(), s.begin() + n);  // [begin, begin + n) 左开右闭
        reverse(s.begin() + n, s.end());
        reverse(s.begin(), s.end());
        return s;
    }
};
// 2.
class Solution {
public:
    string reverseLeftWords(string s, int n) {
        string res = s; // res复制s，对res改动不会对s改动
        int j = 0;
        for (int i = n; i < s.size(); i++) {
            res[j++] = s[i];
        }
        for (int i = 0; i < n; i++) {
            res[j++] = s[i];
        }
        return res;
    }
};
```

KMP #28

```
class Solution {
public:
    void getNext(int* next, const string& s) {
        // 1. 计算next数组，next[i] 代表的是在s这个字符串上，下标 i 以及之前的字符串中的最长相等前后缀的长度
        // 2. next[i] 也指示着当 s[i]和s[j]不匹配的时候，要回退哪个下标上
        //    回退到最长长度这个数的下标上，正好之前的匹配好的长度就是这个最长长度，因为下标从0开始
        // 3. 计算next数组的本质，是把模式字符串既作为主串，也作为模式串，进行模式匹配
        // 4. 前缀不包括首元素，后缀不包括结尾元素，所以主串从 i=1 开始遍历
        int j = 0;
        next[0] = 0;
        for (int i = 1; i < s.size(); i++) {
            while (j > 0 && s[i] != s[j]) { // 不相等的时候要一直回退，直到相等或者回退到模式串的首元素
                j = next[j - 1];
```

```
            }
            if (s[i] == s[j]) {
                j++;
            }
            // j++ 完之后，j的大小就代表的是已经匹配好的字符串的长度
            next[i] = j;
        }
    }
    int strStr(string haystack, string needle) {
        // next数组长度跟模式串相等
        int next[needle.size()];
        getNext(next, needle);

        int j = 0;
        for (int i = 0; i < haystack.size(); i++) {
            while (j > 0 && haystack[i] != needle[j]) {
                j = next[j - 1];
            }
            if (haystack[i] == needle[j]) {
                j++;
            }
            if (j == needle.size()) {  // 是双等号 == ！！！！！！
                return i - j + 1;  // [i-j+1, ..., i] 长度刚好是 j，也就是
needle.size()
            }
        }

        return -1;
    }
};
```

重复的子字符串 #459

暴力

```
class Solution {
public:
    bool repeatedSubstringPattern(string s) {
        if (s.size() == 1) return false;

        // 如果字符串s存在子串s'重复出现的情况，那么这个子串s'一定是s的前缀，并且s的长度是
s'长度的整倍数
        // 每次选择 [0,i) 这个范围的子串，然后从 j=i 开始遍历主串s，必须满足 s[j]=s[j-i]，
否则[0,i)的子串不可以构成主串s
        for (int i = 1; i <= s.size() / 2; i++) {
            if (s.size() % i == 0) {  // 整倍数
                bool flag = true;
                for (int j = i; j < s.size(); j++) {
                    if (s[j] != s[j - i]) {
                        flag = false;
                        break;
                    }
                }
                if (flag == true) return true;
            }
        }
        return false;
```

```
    }
};
```

KMP

```
bool repeatedSubstringPattern (string s) {
    if (s.size() == 0) {
        return false;
    }
    int next[s.size()];
    getNext(next, s);
    int len = s.size();
    if (next[len - 1] != 0 && len % (len - (next[len - 1] )) == 0) {
        return true;
    }
    return false;
}
```

☆栈与队列

栈实现队列 #232

```
class MyQueue {
public:
    stack<int> s1;
    stack<int> s2;
    MyQueue() {

    }

    void push(int x) {
        s1.push(x);
    }

    int pop() {
        if (s1.empty() && s2.empty()) return -1;
        if (s2.empty()) {
            while (!s1.empty()) {
                s2.push(s1.top());
                s1.pop();
            }
        }
        int top = s2.top();
        s2.pop();
        return top;
    }

    int peek() {
        int top = this->pop();
        s2.push(top);  // 一定是 s2.push(), 因为pop操作里，一定是在s2上进行pop的
        return top;
    }

    bool empty() {
        return s1.empty() && s2.empty();
    }
```

```
};
```

队列实现栈 #225

```
class MyStack {
public:
    queue<int> que;
    MyStack() {

    }

    void push(int x) {
        que.push(x);
    }

    int pop() {
        // 用一个队列模拟栈，依次把队首的元素push到队尾，留下最后一个元素即可
        int size = que.size();
        size--;
        while (size--) {
            que.push(que.front());
            que.pop();
        }
        int top = que.front();
        que.pop();
        return top;
    }

    int top() {
        return que.back();
    }

    bool empty() {
        return que.empty();
    }
};
```

有效的括号 #20

```
class Solution {
public:
    bool isValid(string s) {
        stack<char> st;
        for (char ch : s) {
            if (ch == '(') st.push(')');
            else if (ch == '[') st.push(']');
            else if (ch == '{') st.push('}');
            else {
                // 一定注意，因为要用到 st.top 或者 st.pop，那么一定要记着先判断是不是为空
                if (!st.empty() && st.top() == ch) st.pop();
                else return false;
            }
        }

        return st.empty();
    }
```

```
};
```

删除字符串相邻重复项 #1047

```
// 1. stack
class Solution {
public:
    string removeDuplicates(string s) {
        stack<char> st;
        for (char ch : s) {
            if (!st.empty() && ch == st.top()) {  // 一定要注意判断是否为空！！！
                st.pop();
            } else {
                st.push(ch);
            }
        }

        string res = "";
        while (!st.empty()) {
            res.push_back(st.top());  // res += st.top()
            st.pop();
        }
        reverse(res.begin(), res.end());

        return res;
    }
};
// 2. string
class Solution {
public:
    string removeDuplicates(string s) {
        string res = "";
        for(char ch : s) {
            if(res.empty() || ch != res.back()) {
                res.push_back(ch);
            } else {
                res.pop_back();
            }
        }
        return res;

    }
};
```

逆波兰 #150

```
class Solution {
public:
    int evalRPN(vector<string>& tokens) {
        // 逆波兰表达式是一种后缀表达式，即运算符写在后面，去掉括号后无歧义，其实相当于是二叉树中的后序遍历
        // 栈操作：遇到数字入栈；遇到算符则取出栈顶两个数字进行计算，并将结果压入栈中
        stack<int> st;
        for (string str : tokens) {
            if (str == "+" || str == "-" || str == "*" || str == "/") {
                int a = st.top();
```

```
                st.pop();
                int b = st.top();
                st.pop();

                int res;
                if (str == "+") res = b + a;
                else if (str == "-") res = b - a;
                else if (str == "*") res = b * a;
                else if (str == "/") res = b / a;

                st.push(res); // 最后不要忘了把运算的结果 push 进去
            } else {
                st.push(stoi(str));
            }
        }

        return st.top();
    }
};
```

前k个高频元素 #347

```
// priority_queue
class Solution {
public:
    // 1. struct {
    //        bool operator()(int a, int b){
    //             return a > b;
    //        }
    //    }
    struct cmp {
        bool operator()(pair<int, int>& p1, pair<int, int>& p2) {
            return p1.second > p2.second;  // 小顶堆，与 vector 排序顺序相反
        }
    };
    vector<int> topKFrequent(vector<int>& nums, int k) {
        unordered_map<int, int> umap;
        for (int num : nums) {
            umap[num]++;
        }

        // 2. priority_queue<Type, Container, Functional>
        // 3. 优先队列默认大顶堆，堆顶元素是最大的，每次pop就是pop堆顶元素，插入在队尾插入
        // 4. 因为要维护前k个最大的，不能把大的pop出去，所以定义一个小顶堆
        priority_queue<pair<int, int>, vector<pair<int, int>>, cmp> pri_que;
        for (auto iter = umap.begin(); iter != umap.end(); iter++) { //
unordered_map<int, int>::iterator
            pri_que.push(*iter);  // *iter
            if (pri_que.size() > k) {
                pri_que.pop();
            }
        }

        vector<int> res(k);
        for (int i = k - 1; i >= 0; i--) {  // 小顶堆每次pop的都是最小的，所以倒序遍历
            res[i] = pri_que.top().first;
            pri_que.pop();
```

```
        }

        return res;
    }
};
// map
class Solution {
public:
    // 要加 static，但是不知道为什么
    // return a > b，意味着 a > b 的时候，a排在b的前面，也就是从大到小排序
    static bool cmp(const pair<int, int>& a, const pair<int, int>& b) {
        return a.second > b.second;
    }
    vector<int> topKFrequent(vector<int>& nums, int k) {
        unordered_map<int, int> umap;
        for (int num : nums) {
            umap[num]++;
        }

        // sort 无法对 map 进行排序，所以可以先转为 vector 然后再排序
        // pair<int, int> 用 .first .second 去访问元素
        // 注意是 vector<pair<int,int>>，没有 vector<int, int> 的写法
        vector<pair<int, int>> vec(umap.begin(), umap.end());
        sort(vec.begin(), vec.end(), cmp);
        vector<int> res(k);
        for (int i = 0; i < k; i++) {  // 从小到大排序也可以，就是最后要倒序遍历
            res[i] = vec[i].first;
        }
        return res;
    }
};
```

滑动窗口最大值 #239

单调队列

```
// 1.
class Solution {
public:
    class Myque {
    public:
        // c++ 的 stack, queue 底层实现其实默认都是双向队列 deque
        // 这里用 deque 实现符合本题目条件的单调队列，维持队列元素从队首到队尾是单调的，本题目是单调递减
        deque<int> que;
        // pop: 如果队首元素等于滑动窗口即将要滑走的元素，那么直接pop_front就可以。如果不相等，就不操作。
        void pop(int val) {
            if (!que.empty() && que.front() == val) {
                que.pop_front();
            }
        }
        // push: 如果要push进来的元素大于队尾元素，那么需要pop_back掉队尾的元素，一直到push进来的元素小于等于队尾元素为止，因为要保证单调性。
        void push (int val) {
            while (!que.empty() && val > que.back()) {
                que.pop_back();
```

```
            }
            que.push_back(val);
        }
        int front () {
            return que.front();
        }
    };
    vector<int> maxSlidingWindow(vector<int>& nums, int k) {
        Myque my_que;
        vector<int> res;
        for (int i = 0; i < k; i++) {  // 先把第一个滑动窗口push进队列
            my_que.push(nums[i]);
        }
        res.push_back(my_que.front());
        for (int i = k; i < nums.size(); i++) {
            my_que.pop(nums[i - k]);
            my_que.push(nums[i]);
            res.push_back(my_que.front());
        }

        return res;
    }
};
// 2.
class Solution {
public:
    vector<int> maxSlidingWindow(vector<int>& nums, int k) {
        deque<int> q;  //双向列表，保存下标，实现单调(递减)队列
        for(int i = 0; i < k; i++) {
            while(!q.empty() && nums[i] >= nums[q.back()]) {
                q.pop_back();
            }
            q.push_back(i);
        }

        vector<int> res = {nums[q.front()]};  // initiate
        for(int i = k; i < nums.size(); i++) {
            while(!q.empty() && nums[i] >= nums[q.back()]) {  // 1.保持单调递减
                q.pop_back();
            }
            q.push_back(i);

            while(q.front() <= i - k) {  // 2.保证队首元素下标在窗口内
                q.pop_front();
            }
            res.push_back(nums[q.front()]); // 队首永远是最大元素的下标
        }
        return res;
    }
};
// 3.
class Solution {
public:
    vector<int> maxSlidingWindow(vector<int>& nums, int k) {
        deque<int> q;  //双向列表，保存下标，实现单调(递减)队列
        vector<int> res;  // initiate
        for(int i = 0; i < nums.size(); i++) {
            while(!q.empty() && nums[i] >= nums[q.back()]) {  // 1.保持单调递减
```

```
                q.pop_back();
            }
            q.push_back(i);

            while(q.front() <= i - k) {  // 2.保证队首元素下标在窗口内
                q.pop_front();
            }
            if(i >= k - 1) {
                res.push_back(nums[q.front()]); // 队首永远是最大元素的下标
            }
        }
        return res;
    }
};
```

☆排序

归并排序

```
const int N = 1e6;
int n;
int q[N], temp[N];
void merge_sort(int* q, int l, int r) {
    // 0. 千万别忘了终止判断！！！merge和quick都需要
    if (l >= r) return;
    // 1. [l, mid] && [mid + 1, r]
    //    归并排序采用分治思路，先分再治(合)
    int mid = l + r >> 1;
    merge_sort(q, l, mid);
    merge_sort(q, mid + 1, r);

    // 2. 上边执行完，则认为 [l,mid] 和 [mid+1,r] 分别完成了排序，那么下边要做就是归并起来
    int i = l, j = mid + 1;
    int k = 0;
    while (i <= mid && j <= r) {
        if (q[i] <= q[j]) temp[k++] = q[i++];
        else temp[k++] = q[j++];
    }
    // 3. 一定别忘了扫尾
    while (i <= mid) temp[k++] = q[i++];
    while (j <= r) temp[k++] = q[j++];
    for (int i = 0; j < k; i++) {
        q[l + i] = temp[i];
    }
}
```

逆序对 #O-51

```
class Solution {
public:
    int merge_sort(vector<int>& nums, vector<int>& temp, int l, int r) {
        if (l >= r) return 0;
        int mid = l + r >> 1;
        int res = merge_sort(nums, temp, l, mid) + merge_sort(nums, temp, mid +
1, r);
```

```
        // 1. [l, mid] && [mid + 1, r]
        // 2. 当 nums[i] > nums[j] 时，统计逆序对的个数
        //    因为左右两边都是排序好了的，因此 nums[i, mid] 都是大于 nums[j] 的，因此逆序对个数 mid-i+1.
        //    虽然之前排序好的数，已经打乱了原来的顺序，但是在每一个合并的阶段，都把逆序对的个数加上了
        //    因为每次合并的时候计算逆序对个数，只考虑左边大于右边的情况，所以不会计算重复
        // 3. 也可以当 nums[i] <= nums[j] 时候，统计逆序对的个数
        //    此时，nums[mid+1, j-1] 一定都是自小于 nums[i] 的，而且已经归并回temp辅助数组了，逆序对个数 j-1-(mid+1)+1 = j-mid-1
        //    这里是 <= 有等号的时候，因为如果相等也是第一次碰到，不会说 j 前边还有跟 i 相等的
        //    这个方法要注意，当最后收尾的时候，如果是对 nums[i], 依然要进行计算逆序对
        // 4. 统计逆序对个数的时机选择哪一个都可以，选择一个就始终选择这一个，就不会出现漏或者重复的情况
        //    一定要明白，一定是谁归并回（落回）辅助数组的时候，就统计谁的逆序对数量，因为归并回去了，所以不会重复
        int i = l, j = mid + 1;
        int k = 0;
       while(i <= mid && j <= r) {
            if (nums[i] <= nums[j]) {
                temp[k++] = nums[i++];
                res += j - mid - 1;
            } else {
                temp[k++] = nums[j++];
                // res += mid - i + 1;
            }
        }
        while (i <= mid) {
            temp[k++] = nums[i++];
            res += j - mid - 1;
        }
        while (j <= r) {
            temp[k++] = nums[j++];
        }
        for (int i = 0; i < k; i++) {
            nums[l + i] = temp[i];
        }
        return res;
    }
    int reversePairs(vector<int>& nums) {
        if (nums.size() <= 1) return 0;
        vector<int> temp(nums.size());
        return merge_sort(nums, temp, 0, nums.size() - 1);
    }
};
```

元素右侧的逆序数 #315

```
class Solution {
public:
    vector<pair<int, int>> temp;
    vector<int> res;
    void merge_sort(vector<pair<int, int>>& nums_index, int l, int r) {
        if (l >= r) return;
        int mid = l + r >> 1;
```

```
        merge_sort(nums_index, l, mid);
        merge_sort(nums_index, mid + 1, r);

        // 3. 这个题只可以当 nums[i] <= nums[j] 时候，统计逆序对的个数
        //    因为我们要计算的是某个元素右侧比它小的元素，如果当 nums[i]>nums[j] 的时候计算，就把好多元素混在一起了
        //    此时，nums[mid+1, j-1] 一定都是自小于 nums[i] 的，而且已经归并回temp辅助数组了，逆序对个数 j-1-(mid+1)+1 = j-mid-1
        //    这里是 <= 有等号的时候，因为如果相等也是第一次碰到，不会说 j 前边还有跟 i 相等的
        //    这个方法要注意，当最后收尾的时候，如果是对 nums[i], 依然要进行计算逆序对
        // 4. temp辅助数组一定也是跟 nums_index 一样的，同样需要记录坐标
        //    坐标和数值一直绑定在一起，如果只归并了数值，那么后边复制回 nums_index 数组的时候，就跟数值匹配不上了
        int i = l, j = mid + 1;  // j = mid + 1  不是 r !!!!
        int k = l;
        while (i <= mid && j <= r) {
            if (nums_index[i].first <= nums_index[j].first) {
                res[nums_index[i].second] += j - mid -1;  // res累加一定要在temp之前，否则i已经变了！！
                temp[k++] = nums_index[i++];
            } else {
                temp[k++] = nums_index[j++];
            }
        }
        while (i <= mid) {
            res[nums_index[i].second] += j - mid -1;
            temp[k++] = nums_index[i++];
        }
        while (j <= r) {
            temp[k++] = nums_index[j++];
        }

        for (int i = l; i <= r; i++) {
            nums_index[i] = temp[i];
        }
    }
    vector<int> countSmaller(vector<int>& nums) {
        // 1. 与之前单纯的求逆序对总和不一样的是，这个题要非常准确的知道哪一个元素的具体逆序对数
        //    而在归并排序的过程中，元素都打乱了顺序，所以我们要记录 nums[i] 在原始数组中的下标
        //    定义索引数组 nums_index, .first 是数的大小，用来比较进行归并排序， .second 是在原始数组里的下标
        // 2. 定义的时候千万别给一个初始大小啊，否则后边的push_back全都加到后边了，前边全是0
        vector<pair<int, int>> nums_index;
        for (int i = 0; i < nums.size(); i++) {
            nums_index.push_back(pair<int, int>(nums[i], i));
        }
        temp = vector<pair<int, int>>(nums.size());
        res = vector<int>(nums.size(), 0);

        merge_sort(nums_index, 0, nums.size() - 1);
        return res;
    }
};
```

快速排序

```
void quick_sort(int* q, int l, int r) {
    // 0. 千万别忘了终止判断！！！merge和quick都需要
    if (l >= r) return;
    // 1. 下标 l-1 和 r+1
    // 2. 3个 while 都是 <
    // 3. ++i, --j
    // 4. 一定要判断 i < j
    // 5. 跟merge_sort相反，qucik_sort的递归处理在后边
    //    而且是 [l,j] && [j+1,r] 注意是 j ！！ 这里 i可不可以？
    int x = q[l + r >> 1];
    int i = l - 1, j = r + 1;
    while (i < j) {
        while (q[++i] < x);
        while (q[--j] > x);
        if (i < j) {
            swap(q[i], q[j]);
        }
    }

    quick_sort(q, l, j);
    quick_sort(q, j + 1, r);
}
```

第K个最大的数 #215

```
class Solution {
public:
    int quick_sort_k(vector<int>& nums, int l, int r, int k) {
        if (l > r) return -1;
        if (l == r) return nums[l];

        int x = nums[l + r >> 1];
        int i = l - 1, j = r + 1;
        while (i < j) {
            while (nums[++i] < x);
            while (nums[--j] > x);
            if (i < j) swap(nums[i], nums[j]);
        }

        // 1. 注意上边的快速排序是按照从小到大的顺序排序的
        // 2. [l,j] && [j+1,r] 两个范围，右边的一定大于左边的所有元素
        //    [l,i-1] && [i,r] 这个区间也可以，快排这个区间应该可以，但是这个题超出时间限制不知道为啥
        //    跳出循环的时候，一定是 i>=j, q[i-1]<x, q[i]>=x, q[j+1]>x, q[j]<=x, 所以决定了上边两个区间是对的
        // 3. 所以，如果求第K个最大的数，当K<=right右边长度的时候，那么目标一定就在右边，因为右边是大的数
        // 4. 为什么要先++i,--j, 是为了防止陷入死循环，因为如果i<j而且q[i]=x=q[j], 那么就会陷入死循环
        int right = r - j;
        if (k <= right) return quick_sort_k(nums, j + 1, r, k);
        else return quick_sort_k(nums, l, j, k - right);
    }
    int findKthLargest(vector<int>& nums, int k) {
```

```
        return quick_sort_k(nums, 0, nums.size() - 1, k);
    }
};
        // int left = j - l + 1;  // 第K个最小的数
        // if(k <= left) return quick_sort(nums, l, j, k);
        // else return quick_sort(nums, j + 1, r, k - left);
```

拓扑排序 #210

```
class Solution {
public:
    vector<int> findOrder(int numCourses, vector<vector<int>>& prerequisites) {
        // 1. （有向图）拓扑排序：BFS+贪心。（DFS也可以，但是BFS最经典）
        // 2. 入度为0的节点加入队列，pop出去的节点的相邻节点的入度减一，当入度为0时加入队列
        unordered_map<int, int> inDegree;
        unordered_map<int, vector<int>> adjust;
        for (auto item : prerequisites) {
            int course = item[0];
            int pre = item[1];
            inDegree[course]++;
            adjust[pre].push_back(course);
        }

        queue<int> que;
        for (int i = 0; i < numCourses; i++) {
            if (inDegree.find(i) == inDegree.end()) {
                que.push(i);
            }
        }

        vector<int> res;
        while (!que.empty()) {
            int pre = que.front();
            res.push_back(pre);
            que.pop();

            for (int course : adjust[pre]) {
                inDegree[course]--;
                if (inDegree[course] == 0) {
                    que.push(course);
                    inDegree.erase(course);
                }
            }
        }
        // 3. 拓扑排序还可以判断有向图是否有环，如果最后inDegree不为空，说明有没有遍历完的节点，即存在环。
        //    为了实现这个，在每次入度数减为0要加入队列的时候，都要记得inDegree要清除该节点才行
        if (!inDegree.empty()) return {};
        return res;
    }
};
```

☆二叉树

定义

```
struct TreeNode {
    int val;
    TreeNode *left;
    TreeNode *right;
    TreeNode(int x) : val(x), left(NULL), right(NULL) {}
};
```

深度遍历 #144,94,145

前中后序 #144 #94 #145

```
class Solution {
public:
    void traverse(TreeNode *cur, vector<int>& vec) {
        if(cur == nullptr) return;
        vec.push_back(cur->val);
        traverse(cur->left, vec);
        traverse(cur->right, vec);
    }

    vector<int> preorderTraversal(TreeNode* root) {
        vector<int> res;
        traverse(root, res);
        return res;
    }
};

void traverse(TreeNode *cur, vector<int>& vec) {
    if(cur == nullptr) return;
    traverse(cur->left, vec);
    vec.push_back(cur->val);
    traverse(cur->right, vec);
}
void traverse(TreeNode *cur, vector<int>& vec) {
    if(cur == nullptr) return;
    traverse(cur->left, vec);
    traverse(cur->right, vec);
    vec.push_back(cur->val);
}
```

☆迭代遍历（栈）

层序遍历 #102

```
class Solution {
public:
    vector<vector<int>> levelOrder(TreeNode* root) {
        vector<vector<int>> res;
        if(!root) return res;

        queue<TreeNode*> que;
        que.push(root);
```

```
        while(!que.empty()) {
            vector<int> vec;
            int size = que.size();
            // 一定要提前计算que.size(),因为会变
            // 也不要写 while(que.empty()),也是因为que.size()会变
            for(int i = 0; i < size; i++) {
                TreeNode* node = que.front();
                vec.push_back(node->val);
                que.pop();

                if(node->left) que.push(node->left);
                if(node->right) que.push(node->right);
                // delete tmp; // 不可以写删除，否则会段错
误
            }
            res.push_back(vec);
        }
        return res;
    }
};
```

翻转二叉树 #226

```
class Solution {
public:
    TreeNode* invertTree(TreeNode* root) {
        if(!root) return nullptr;
        // if(root->left == nullptr && root->right == nullptr) return root;

        swap(root->left, root->right);      //前序遍历
        invertTree(root->left);             //中序遍历不可以，会翻转两次
        invertTree(root->right);
        // swap(root->left, root->right);  //后序遍历

        return root;
    }
};
```

对称二叉树 #101

```
class Solution {
public:
    bool compare_sym(TreeNode* left,TreeNode* right) {
        if(!left && !right) return true;
        else if((!left && right) || (left && !right)) return false;
        else if(left->val != right->val) return false;

        // 左右节点val相等，传入“左左，右右”和“左右，右左”两对节点，递归遍历外侧和内侧的树是
否对称
        return compare_sym(left->left, right->right) && compare_sym(left->right,
right->left);
    }
    bool isSymmetric(TreeNode* root) {
        if(!root) return true;
        return compare_sym(root->left, root->right);
    }
```

```
};
```

相同的树 #100

```
class Solution {
public:
    bool isSameTree(TreeNode* p, TreeNode* q) {
        if(!p && !q) return true;
        else if(!p && q || p && !q) return false;
        else if(p->val != q->val) return false;
        return isSameTree(p->left, q->left) && isSameTree(p->right, q->right);
    }
};
```

二叉树的最大深度 #104

```
class Solution {
public:
    int maxDepth(TreeNode* root) {
        if(!root) return 0;
        // if(!root->left && !root->right) return 1;
        return 1 + max(maxDepth(root->left), maxDepth(root->right));
    }
};
// n叉树最大深度
class Solution {
public:
    int maxDepth(Node* root) {
        if(!root) return 0;
        int depth = 0;
        for(Node* node : root->children) {
            int tmp = maxDepth(node);
            depth = depth > tmp ? depth : tmp;
        }
        return depth + 1;
    }
};
```

二叉树的最小深度 #111

```
class Solution {
public:
    int minDepth(TreeNode* root) {
        if(!root) return 0;
        if(!root->left && !root->right) return 1;
        int depth = INT32_MAX;
        if(root->left) depth = min(depth, minDepth(root->left));
        if(root->right) depth = min(depth, minDepth(root->right));
        return 1 + depth;
    }
};
```

完全二叉树的节点个数 #222

```
// 1. 递归遍历 O(n) O(logn)
class Solution {
public:
    int countNodes(TreeNode* root) {
        if(!root) return 0;
        return 1 + countNodes(root->left) + countNodes(root->right);
    }
};
// 2. 分别递归左孩子，和右孩子，递归到某一深度一定会有左孩子或者右孩子为满二叉树 O(logn *
logn)?? O(logn)
class Solution {
public:
    int countNodes(TreeNode* root) {
        if(!root) return 0;
        TreeNode* left = root->left;
        TreeNode* right = root->right;
        int leftDepth = 0, rightDepth = 0;
        while(left) {
            leftDepth++;
            left = left->left;
        }
        while(right) {
            rightDepth++;
            right = right->right;
        }
        if(leftDepth == rightDepth) {
            return pow(2, leftDepth + 1) - 1;
        }
        return 1 + countNodes(root->left) + countNodes(root->right);
    }
};
```

判断平衡二叉树 #110

```
class Solution {
public:
    int getDepth(TreeNode* root) {
        if(!root) return 0;
        int leftDepth = getDepth(root->left);
        int rightDepth = getDepth(root->right);
        if(leftDepth == -1 || rightDepth == -1) return -1;
        if(abs(leftDepth - rightDepth) <= 1) return 1 + max(leftDepth,
rightDepth);
        return -1;
    }
    bool isBalanced(TreeNode* root) {
        int depth = getDepth(root);
        return depth == -1 ? false : true;
    }
};
```

二叉树的所有路径 #257

```
class Solution {
public:
    void traverse(TreeNode* node, vector<int>& path, vector<string>& res) {
        path.push_back(node->val);
        if(!node->left && !node->right) {  // 1. 前序遍历
            string str;
            int i = 0;
            for(; i < path.size() - 1; i++) {
                str += to_string(path[i]);
                str += "->";
            }
            str += to_string(path[i]);
            res.push_back(str);
        }

        if(node->left) {
            traverse(node->left, path, res);
            path.pop_back();
        }
        if(node->right) {
            traverse(node->right, path, res);
            path.pop_back();
        }
    }
    vector<string> binaryTreePaths(TreeNode* root) {
        if(!root) return {};
        vector<int> path;
        vector<string> res;
        traverse(root, path, res);
        return res;
    }
};
```

左子叶之和 #404

```
// 1. 不能通过当前节点判断是否为左子叶，要通过父节点去判断
class Solution {
public:
    int sum;
    void traverse(TreeNode* root) {
        if(!root) return;
        if(root->left && !root->left->left && !root->left->right) {
            sum += root->left->val;
        }
        traverse(root->left);
        traverse(root->right);
    }
    int sumOfLeftLeaves(TreeNode* root) {
        if(!root) return 0;
        sum = 0;
        traverse(root);
        return sum;
    }
};
```

```
// 2. 后序遍历
class Solution {
public:
    int sumOfLeftLeaves(TreeNode* root) {
        if (root == NULL) return 0;

        int leftValue = sumOfLeftLeaves(root->left);    // 左
        int rightValue = sumOfLeftLeaves(root->right);  // 右
                                                        // 中
        int midValue = 0;
        if (root->left && !root->left->left && !root->left->right) { // 中
            midValue = root->left->val;
        }
        int sum = midValue + leftValue + rightValue;
        return sum;
    }
};
```

树左下角的值 #513

```
// 1. 递归遍历（回溯）
class Solution {
public:
    int maxDepth = -1;
    int maxVal;
    void traverse(TreeNode* root, int leftDepth) {
        if(!root->left && !root->right) {
            if(leftDepth > maxDepth) {  //保证首先遍历最后一行的最左边元素，而且只进入循环这一次
                maxDepth = leftDepth;
                maxVal = root->val;
            }
        }
        if(root->left) {
            leftDepth++;
            traverse(root->left, leftDepth);
            leftDepth--;
        }
        if(root->right) {
            leftDepth++;
            traverse(root->right, leftDepth);
            leftDepth--;
        }
    }
    int findBottomLeftValue(TreeNode* root) {
        if(!root) return -1;
        int leftDepth = 0;
        traverse(root, leftDepth);
        return maxVal;
    }
};
// 2. 层序遍历
class Solution {
public:
    int findBottomLeftValue(TreeNode* root) {
        if(!root) return -1;
```

```
        queue<TreeNode*> que;
        que.push(root);
        int val = 0;
        while(!que.empty()) {
            int size = que.size();
            for(int i = 0; i < size; i++) {
                TreeNode* node = que.front();
                que.pop();
                if(i == 0) val = node->val;  //只保存每一行的第一个元素，最后结束代表的就是最后一行最左边元素

                if(node->left) que.push(node->left);
                if(node->right) que.push(node->right);
            }
        }
        return val;

    }
};
```

路经总和 #112

```
class Solution {
public:
    bool traverse(TreeNode* root, int sum) {
        if(!root->left && !root->right) {
            if(sum == 0) return true;
        }
        if(root->left) {
            sum -= root->left->val;  // 回溯的时候，前后都要写好，因为不是全局变量，返回时在里边函数做的操作相当于没做
            if(traverse(root->left, sum)) return true;  // 一定要判断一下，否则后边的false会覆盖叶子节点返回的true
            sum += root->left->val;
        }
        if(root->right) {
            sum -= root->right->val;
            if(traverse(root->right, sum)) return true;
            sum += root->right->val;
        }
        return false;
    }
    bool hasPathSum(TreeNode* root, int targetSum) {
        if(!root) return false;
        return traverse(root, targetSum - root->val);   // 要提前减去root->val，traverse里不用先减了
    }
};
// 也可以定义全局变量和全局标志flag
```

路经总和II #113

```
class Solution {
public:
    vector<vector<int>> res;
    vector<int> path;
    void traverse(TreeNode* root, int sum) {
        if(!root->left && !root->right) {
            if(sum == 0) {
                res.push_back(path);
                return;
            }
        }
        if(root->left) {
            sum -= root->left->val;
            path.push_back(root->left->val);
            traverse(root->left, sum);
            sum += root->left->val;
            path.pop_back();
        }
        if(root->right) {
            sum -= root->right->val;
            path.push_back(root->right->val);
            traverse(root->right, sum);
            sum += root->right->val;
            path.pop_back();
        }
    }
    vector<vector<int>> pathSum(TreeNode* root, int targetSum) {
        res.clear();
        path.clear();
        if(!root) return res;

        path.push_back(root->val);
        traverse(root, targetSum - root->val);
        return res;
    }
};
```

中序和后序构造树 #106

```
class Solution {
public:
    TreeNode* traverse(vector<int>& inorder, int in_begin, int in_end,
vector<int>& postorder, int post_begin, int post_end) {
        // 注意不要使用 x.size(),下标不要用0或者x.size()-1 之类的
        if(post_begin == post_end) return nullptr;  // 一定要对空数组的判断

        int rootVal = postorder[post_end - 1];
        TreeNode* root = new TreeNode(rootVal);

        if(post_end - post_begin == 1) return root;

        int delimiterIndex;
        for(delimiterIndex = in_begin; delimiterIndex < in_end;
delimiterIndex++) {
```

```
            if(inorder[delimiterIndex] == rootVal) break;
        }

        root->left = traverse(inorder, in_begin, delimiterIndex, postorder,
post_begin, post_begin + delimiterIndex - in_begin);
        root->right = traverse(inorder, delimiterIndex + 1, in_end, postorder,
post_begin + delimiterIndex - in_begin, post_end - 1);
        return root;
    }
    TreeNode* buildTree(vector<int>& inorder, vector<int>& postorder) {
        if(inorder.size() == 0 || postorder.size() == 0) return nullptr;
        // 坚持左闭右开的区间
        return traverse(inorder, 0, inorder.size(), postorder, 0,
postorder.size());
    }
};
```

前序和中序构造树 #105

```
class Solution {
public:
    TreeNode* traverse(vector<int>& preorder, int pre_begin, int pre_end,
vector<int>& inorder, int in_begin, int in_end) {
        // 注意不要使用 x.size(),下标不要用0或者x.size()-1 之类的
        if(pre_begin == pre_end) return nullptr;  // 不要使用 preorder.size()

        int rootVal = preorder[pre_begin];  // 不要使用下标[0]
        TreeNode* root = new TreeNode(rootVal);

        if(pre_end - pre_begin == 1) return root;

        int delimiterIndex;
        for(delimiterIndex = in_begin; delimiterIndex < in_end;
delimiterIndex++) {
            if(inorder[delimiterIndex] == rootVal) break;
        }

        root->left = traverse(preorder, pre_begin + 1, pre_begin + 1 +
delimiterIndex - in_begin, inorder, in_begin, delimiterIndex);  // 起始位置不要用1,
而是pre_begin+1
        root->right = traverse(preorder, pre_begin + 1 + delimiterIndex -
in_begin, pre_end, inorder, delimiterIndex + 1, in_end);
        return root;
    }
    TreeNode* buildTree(vector<int>& preorder, vector<int>& inorder) {
        if(preorder.size() == 0 || inorder.size() == 0) return nullptr;
        return traverse(preorder, 0, preorder.size(), inorder, 0,
inorder.size());
    }
};
```

最大二叉树 #654

```
class Solution {
public:
    TreeNode* traverse(vector<int>& nums, int begin, int end) {
        int size = end - begin;
        if(size == 0) return nullptr;
        else if(size == 1) return new TreeNode(nums[begin]);

        int maxIndex = begin;
        for(int i = begin; i < end; i++) {
            if(nums[i] > nums[maxIndex]) maxIndex = i;
        }
        TreeNode* root = new TreeNode(nums[maxIndex]);
        root->left = traverse(nums, begin, maxIndex);
        root->right = traverse(nums, maxIndex + 1, end);

        return root;
    }
    TreeNode* constructMaximumBinaryTree(vector<int>& nums) {
        return traverse(nums, 0, nums.size());
    }
};
```

合并二叉树 #617

```
class Solution {
public:
    TreeNode* mergeTrees(TreeNode* root1, TreeNode* root2) {
        if(!root1 && !root2) return nullptr;
        else if(root1 && !root2) return root1;
        else if(!root1 && root2) return root2;

        root1->val = root1->val + root2->val;  // 重复利用root1
        root1->left = mergeTrees(root1->left, root2->left);  // 传入两个数的参数
        root1->right = mergeTrees(root1->right, root2->right);

        return root1;
    }
};
```

BST的搜索 #700

BST-二叉搜索树

```
// 1.递归
class Solution {
public:
    TreeNode* searchBST(TreeNode* root, int val) {
        if(!root) return nullptr;
        if(root->val == val) return root;
        else if(root->val > val) return searchBST(root->left, val);
        return searchBST(root->right, val);
    }
};
```

```
// 2.迭代
class Solution {
public:
    TreeNode* searchBST(TreeNode* root, int val) {
        while(root) {
            if(root->val > val) root = root->left;
            else if(root->val < val) root = root->right;
            else return root;
        }
        return nullptr;
    }
};
```

验证BST #98

```
class Solution {
public:
    TreeNode* pre = nullptr;  // 记录前一个节点，方便比较，否则有INT_MIN值，不好比较
    bool isValidBST(TreeNode* root) {
        if(!root) return true;

        bool left = isValidBST(root->left);

        if((pre != nullptr) && (pre->val >= root->val)) return false;  // 一定是
>=,因为二叉搜索树不能有重复节点
        pre = root;

        bool right = isValidBST(root->right);

        return left && right;
    }
};
// 也可以定义 long long maxVal = LONG_MIN 方便比较，或者定义一个vector最后判断是否有序。
```

BST最小绝对差 #530

```
// 1. 保存前一个节点pre
class Solution {
public:
    TreeNode* pre = nullptr;
    int res = INT_MAX;
    void traverse(TreeNode* root) {
        if(!root) return;
        traverse(root->left);

        if((pre != nullptr) && (root->val - pre->val < res)) res = root->val -
pre->val;
        pre = root;

        traverse(root->right);
    }

    int getMinimumDifference(TreeNode* root) {
        traverse(root);
        return res;
```

```
    }
};

class Solution {
public:
    vector<int> vec;
    void traverse(TreeNode* root) {
        if(!root) return;
        traverse(root->left);
        vec.push_back(root->val);
        traverse(root->right);
    }
    int getMinimumDifference(TreeNode* root) {
        traverse(root);
        int min = INT_MAX;
        for(int i = 0; i < vec.size() - 1; i++) {
            int tmp = vec[i + 1] - vec[i];
            if(tmp < min) min = tmp;
        }
        return min;
    }
};
```

BST的众数 #501

```
class Solution {
public:
    vector<int> res;
    int max = 0;
    int cnt = 0;
    TreeNode* pre = nullptr;

    void traverse(TreeNode* root) {
        if(!root) return;
        traverse(root->left);

        if(pre == nullptr) {
            cnt = 1;
        } else if(pre->val == root->val) {
            cnt++;
        } else {
            cnt = 1;
        }
        pre = root;  // 一定记得更新

        if(cnt == max) {
            res.push_back(root->val);
        } else if(cnt > max) {
            res.clear();
            max = cnt;
            res.push_back(root->val);
        }

        traverse(root->right);
    }
    vector<int> findMode(TreeNode* root) {
        traverse(root);
```

```
        return res;
    }
};
```

二叉树的最近公共祖先 #236

```
// 后序遍历，回溯
class Solution {
public:
    TreeNode* lowestCommonAncestor(TreeNode* root, TreeNode* p, TreeNode* q) {
        if(root == p || root == q || root == nullptr) return root;

        TreeNode* left = lowestCommonAncestor(root->left, p, q);
        TreeNode* right = lowestCommonAncestor(root->right, p, q);

        // 四种情况
        if(left && right) return root;
        if(left && !right) return left;
        if(!left && right) return right;
        else return nullptr;
    }
};
```

二叉搜索树的最近公共祖先 #235

```
// 如果两个节点值都小于根节点，说明他们都在根节点的左子树上
// 如果两个节点值都大于根节点，说明他们都在根节点的右子树上
// 如果一个节点值大于根节点，一个节点值小于根节点，说明他们他们在根节点的异侧，那么根节点就是他们的最近公共祖先节点
class Solution {
public:
    TreeNode* lowestCommonAncestor(TreeNode* root, TreeNode* p, TreeNode* q) {
        if(!root) return nullptr;

        if((p->val < root->val) && (q->val < root->val)) return lowestCommonAncestor(root->left, p, q);
        else if((p->val > root->val) && (q->val > root->val)) return lowestCommonAncestor(root->right, p, q);
        else return root;
    }
};
```

BST插入 #701

```
// 1.
class Solution {
public:
    TreeNode* insertIntoBST(TreeNode* root, int val) {
        TreeNode* node = new TreeNode(val);
        if(!root) return node;

        TreeNode* cur = root;
        TreeNode* pre;
        while(cur) {
```

```
            pre = cur;
            if(val < cur->val) cur = cur->left;
            else cur = cur->right;
        }
        if(val < pre->val) pre->left = node;
        else pre->right = node;
        return root;
    }
};
// 2. 通过递归函数返回值完成新加入节点的父子关系赋值操作
class Solution {
public:
    TreeNode* insertIntoBST(TreeNode* root, int val) {
        if(!root) {
            TreeNode* node = new TreeNode(val);
            return node;
        }

        if(val < root->val) root->left = insertIntoBST(root->left, val);  // 串联起来了
        else root->right = insertIntoBST(root->right, val);

        return root;
    }
};
```

BST的删除 #450

```
// 由于BST的性质，最多只删除一个节点，因为没有重复的节点
class Solution {
public:
    TreeNode* deleteNode(TreeNode* root, int key) {
        if(!root) return nullptr;

        if(root->val == key) {
            if(!root->left && !root->right) {
                delete root;
                return nullptr;
            } else if(!root->left) {
                TreeNode* tmp = root->right;
                delete root;
                return tmp;
            } else if(!root->right) {
                TreeNode* tmp = root->left;
                delete root;
                return tmp;
            } else {
                 // 都不空，因为是删除一个节点，不能舍弃其他节点，所以比较麻烦，跟下一个题目进行对比
                TreeNode* cur = root->right;
                while(cur->left) {
                    cur = cur->left;
                }
                cur->left = root->left;
                TreeNode* tmp = root;
                root = root->right;
                delete tmp;
```

```
                return root;
            }
        } else if(root->val > key) {
            root->left = deleteNode(root->left, key);
        } else {
            root->right = deleteNode(root->right, key);
        }
        return root;
    }
};
```

修剪BST #669

```
class Solution {
public:
    TreeNode* trimBST(TreeNode* root, int low, int high) {
        if(!root) return nullptr;

        // 跟删除特定的值不同，删除不在某个区间的值，根据值大小的判断可以舍弃某一半的子树，返回符合条件的另一半子树
        if(root->val < low) {
            TreeNode* right = trimBST(root->right, low, high);
            return right;
        } else if(root->val > high) {
            TreeNode*  left = trimBST(root->left, low, high);
            return left;
        }

        root->left = trimBST(root->left, low, high);  // root->left接入符合条件的左孩子
        root->right = trimBST(root->right, low, high);  // root->right接入符合条件的右孩子
        return root;
    }
};
```

有序数组转为平衡BST #108

```
// 每次以数组中间位置的元素为分割点，自然就是平衡二叉搜索树
class Solution {
public:
    TreeNode* traverse(vector<int>& nums, int begin, int end) {
        int size = end - begin;
        if(size == 0) return nullptr;
        if(size == 1) return new TreeNode(nums[begin]);

        int mid = begin + size / 2;
        TreeNode* root = new TreeNode(nums[mid]);
        root->left = traverse(nums, begin, mid);
        root->right = traverse(nums, mid + 1, end);

        return root;
    }
    TreeNode* sortedArrayToBST(vector<int>& nums) {
        return traverse(nums, 0, nums.size());
    }
```

```
};
```

BST转为累加树 #538

```
// 逻辑是这样的
// 首先明确遍历的顺序是【逆中序】遍历
// 其次，BST的中序遍历是有序的，我们要做的中间结点的逻辑就是加上前一个结点的值，定义一个pre保存前一个结点的值
// 然后因为是逆中序遍历，所以相当于是倒着往前加
class Solution {
public:
    int pre = 0;
    void traverse(TreeNode* root) {
        if(!root) return;

        traverse(root->right);
        root->val += pre;
        pre = root->val;
        traverse(root->left);
    }
    TreeNode* convertBST(TreeNode* root) {
        traverse(root);
        return root;
    }
};
```

☆回溯

递归要素

- 确定参数和返回值
- 确定终止条件
- 确定单层递归逻辑

二叉树的最大路径和 #124

```
class Solution {
private:
    int maxSum = INT_MIN;
public:
    int dfs(TreeNode *node){
        if(node == nullptr) return 0;

        int leftMax = max(dfs(node->left), 0);
        int rightMax = max(dfs(node->right), 0);

        int newPath = node->val + leftMax + rightMax;
        maxSum = max(maxSum, newPath);
        return node->val + max(leftMax, rightMax);
    }
    int maxPathSum(TreeNode* root) {
        dfs(root);
        return maxSum;
    }
};
```

组合 #77

```
//  回溯精神：for循环横向遍历，递归纵向遍历，回溯不断调整结果集
class Solution {

public:
    vector<vector<int>> res;
    vector<int> path;
    void backtracking(int n, int k, int begin) {
        if(path.size() == k) {
            res.push_back(path);
            return;
        }

        // for(int i = begin; i <= n; i++) {
        for(int i = begin; i <= n - (k - path.size()) + 1; i++) { // 优化，剪枝
            path.push_back(i);
            backtracking(n, k, i + 1);
            path.pop_back();
        }
        return;
    }
    vector<vector<int>> combine(int n, int k) {
        res.clear();
        path.clear();
        backtracking(n, k, 1);
        return res;
    }
};
```

组合总和III #216

找出所有相加之和为 n 的 k 个数的组合。组合中只允许含有 1 - 9 的正整数.

每种组合中不存在重复的数字。

```
class Solution {
public:
    vector<vector<int>> res;
    vector<int> path;
    int curSum = 0;
    void backtracking(int k, int n, int begin) {
        if(curSum > n) {
            // 这里千万不要回溯，直接return就好，因为for循环会统一回溯，否则会多pop元素了
            // curSum -= path[path.size() - 1];
            // path.pop_back();
            return;
        }
        if(path.size() == k) {
            if(curSum == n) {
                res.push_back(path);
            }
            return;
        }
        // 这样写也对，但是path.size()会超过k，会进行下边无意义的多余的循环
        // if(path.size() == k && curSum == n) {
```

```
        //     res.push_back(path);
        //     return;
        // }

        for(int i = begin; i <= 9 - (k - path.size()) + 1; i++) {
            curSum += i;
            path.push_back(i);
            backtracking(k, n, i + 1);
            curSum -= i;
            path.pop_back();
        }
        return;
    }
    vector<vector<int>> combinationSum3(int k, int n) {
        backtracking(k, n, 1);
        return res;
    }
};
```

电话号码的字母组合 #17

```
// 这个是多个集合求组合，上一个题是一个集合求组合。
class Solution {
public:
    // 字母表
    const string letterMap[10] = {
        "",
        "",
        "abc",
        "def",
        "ghi",
        "jkl",
        "mno",
        "pqrs",
        "tuv",
        "wxyz"
    };
    vector<string> res;
    string str;
    // 没有begin，而是index代表遍历的是第几个集合，每个集合从0开始遍历
    void backtracking(const string& digits, int index) {
        if(index == digits.size()) {
            res.push_back(str);
            return;
        }

        int letter = digits[index] - '0';
        string letters = letterMap[letter];
        for(int i = 0; i < letters.size(); i++) {
            str.push_back(letters[i]);
            backtracking(digits, index + 1);
            str.pop_back();
        }
        return;
    }
    vector<string> letterCombinations(string digits) {
        if(digits.size() == 0) return res;
```

```
        backtracking(digits, 0);
        return res;
    }
};
```

组合总和 #39

无重复元素数组 candidates, 目标数 target, 找出 candidates 中所有可以使数字和为 target 的组合。

candidates 中的数字可以重复取。

```
class Solution {
public:
    vector<vector<int>> res;
    vector<int> path;
    int curSum = 0;
    void backtracking(vector<int>& candidates, int target, int index) {
        if(curSum > target) return;
        if(curSum == target) {
            res.push_back(path);
            return;
        }

        for(int i = index; i < candidates.size(); i++) {
            curSum += candidates[i];
            path.push_back(candidates[i]);
            backtracking(candidates, target, i); // 不是 index+1, 也不是 i+1, 而是
i, 代表可以重复选取
            curSum -= candidates[i];
            path.pop_back();
        }
    }
    vector<vector<int>> combinationSum(vector<int>& candidates, int target) {
        backtracking(candidates, target, 0);
        return res;
    }
};
```

组合总和II #40

有重复数组 candidates, 目标数 target, 找出 candidates 中所有可以使数字和为 target 的组合。

candidates 中的每个数字在每个组合中只能使用一次, 解集不能包含重复的组合。

```
class Solution {
public:
    vector<vector<int>> res;
    vector<int> path;
    int curSum = 0;
    void backtracking(vector<int>& candidates, int target, int index) {
        if(curSum > target) return;
        if(curSum == target) {
            res.push_back(path);
            return;
        }
```

```
        for(int i = index; i < candidates.size(); i++) {
            if(i > index && candidates[i] == candidates[i - 1]) continue;  // 同一个树层，不可以重复取
            curSum += candidates[i];
            path.push_back(candidates[i]);
            backtracking(candidates, target, i + 1);
            curSum -= candidates[i];
            path.pop_back();
        }
    }
    vector<vector<int>> combinationSum2(vector<int>& candidates, int target) {
        // 必须要先进行排序，后边去重才是有效的
        sort(candidates.begin(), candidates.end());
        backtracking(candidates, target, 0);
        return res;
    }
};
```

分割回文串 #131

```
class Solution {
public:
    vector<vector<string>> res;
    vector<string> path;
    bool isPalindrome(const string& s, int start, int end) {  // 要定义start，end
        for(int i = start, j = end; i < j; i++, j--) {
            if(s[i] != s[j]) return false;
        }
        return true;
    }
    // index是切割的位置，表示在s[index]这个元素后边进行切割，当index=s.size()则是切割到末尾了
    void backtracking(const string& s, int index) {  // 定义为 const string& 是引用，节省空间和时间
        if(index >= s.size()) {
            res.push_back(path);
            return;
        }

        for(int i = index; i < s.size(); i++) {
            if(isPalindrome(s, index, i)) {  // [index, i] 闭区间的子串
                path.push_back(s.substr(index, i - index + 1));  // string.substr(index,len) 第二个参数是长度
            } else {
                continue;
            }

            backtracking(s, i + 1);
            path.pop_back();
        }
    }
    vector<vector<string>> partition(string s) {
        backtracking(s, 0);
        return res;
    }
};
```

复原IP地址 #93

```
class Solution {
public:
    vector<string> res;
    vector<string> path;
    void backtracking(const string& s, int index) {
        if(path.size() == 4 && index >= s.size()) {  // 一定别忘了必须有且只有四段，而且两个条件必须同时满足
            string str;
            for(string ss : path) {
                str += ss + '.';
            }
            str.pop_back();
            res.push_back(str);
            return;
        }

        for(int i = index; i < s.size(); i++) {
            if(isValid(s, index, i)) {
                path.push_back(s.substr(index, i - index + 1));
            } else {
                break;  // 如果不合法，直接跳出循环即可，没必要continue了
            }

            backtracking(s, i + 1);
            path.pop_back();
        }
    }
    bool isValid(const string& s, int begin, int end) {
        if(begin > end) return false;  // 别忘了判断是否为空
        if(s[begin] == '0' && begin != end) return false;

        int num = 0;
        for(int i = begin; i <= end; i++) {
            if(s[i] > '9' || s[i] < '0') return false;
            num = (s[i] - '0') + num * 10;
            if(num > 255) return false;
        }
        return true;
    }

    vector<string> restoreIpAddresses(string s) {
        if(s.size() > 12) return res;  // 剪枝操作
        backtracking(s, 0);
        return res;
    }
};
```

子集 #78

无重复元素数组nums，返回数组所有子集

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

```
    vector<int> path;
    void backtracking(vector<int>& nums, int index) {
        res.push_back(path);  // 提前加
        if(index >= nums.size()) {
            return;
        }

        for(int i = index; i < nums.size(); i++) {
            path.push_back(nums[i]);
            backtracking(nums, i + 1);
            path.pop_back();
        }
    }
    vector<vector<int>> subsets(vector<int>& nums) {
        backtracking(nums, 0);
        return res;
    }
};
```

子集II #90

有重复元素的整数数组 nums，返回该数组所有可能的子集（幂集）。

```
class Solution {
public:
    vector<vector<int>> res;
    vector<int> path;
    void backtracking(vector<int>& nums, int index) {
        res.push_back(path);
        if(index >= nums.size()) {
            return;
        }

        for(ini = index; i < nums.size(); i++) {
            if(i > index && nums[i] == nums[i - 1]) continue;  // 去重
            path.push_back(nums[i]);
            backtracking(nums, i + 1);
            path.pop_back();
        }
    }
    vector<vector<int>> subsetsWithDup(vector<int>& nums) {
        if(nums.size() == 0) return res;
        sort(nums.begin(), nums.end());
        backtracking(nums, 0);
        return res;
    }
};
```

递增子序列 #491

```
class Solution {
public:
    vector<vector<int>> res;
    vector<int> path;
    void backtracking(vector<int>& nums, int index) {
        if(path.size() >= 2) res.push_back(path);
```

```
        if(index >= nums.size()) return;

        unordered_set<int> uset;  // 记录本层（树层）元素是否重用过
        for(int i = index; i < nums.size(); i++) {
            // 一定要判断 path.empty()，不要用 i>index 去判断
            // i>index 的时候:
            // (1)树退回到第一层的时候，i>index，但是path是空的，path.back()会报错或者返回一个很大的数
            // (2)树往下递归的时候，选取集合第一个元素，i==index 不会进入if()，但是如果元素小于path.back()会加到path里
            if((!path.empty() && nums[i] < path.back()) || (uset.find(nums[i]) != uset.end())) continue;
            uset.insert(nums[i]);
            path.push_back(nums[i]);
            backtracking(nums, i + 1);
            path.pop_back();
        }
    }
    vector<vector<int>> findSubsequences(vector<int>& nums) {
        backtracking(nums, 0);
        return res;
    }
};
```

全排列 #46

```
class Solution {
public:
    vector<vector<int>> res;
    vector<int> path;
    void backtracking(vector<int>& nums, vector<bool>& used) {
        if (path.size() == nums.size()) {
            res.push_back(path);
            return;
        }

        for (int i = 0; i < nums.size(); i++) {
            if (used[i] == true) continue;
            used[i] = true;  // 不要写成 ==
            path.push_back(nums[i]);
            backtracking(nums, used);
            used[i] = false;
            path.pop_back();
        }
    }
    vector<vector<int>> permute(vector<int>& nums) {
        vector<bool> used(nums.size(), false);
        backtracking(nums, used);
        return res;
    }
};
```

全排列II #47

```
class Solution {
public:
    vector<vector<int>> res;
    vector<int> path;
    void backtracking(vector<int>& nums, vector<bool>& used) {
        if (path.size() == nums.size()) {
            res.push_back(path);
            return;
        }

        // used[i - 1] == true 代表当前这个树枝用过i-1这个元素
        // used[i - 1] == false 代表当前这个树层用过i-1这个元素
        // 因为同一个树层，在"弹"回来的时候，前一个元素的used置为了false，而且只能判断前一个元素是否用过
        for (int i = 0; i < nums.size(); i++) {
            if(used[i] == true) continue;
            if (i > 0 && nums[i] == nums[i - 1] && used[i - 1] == false) continue;
            used[i] = true;
            path.push_back(nums[i]);
            backtracking(nums, used);
            path.pop_back();
            used[i] = false;
        }
    }
    vector<vector<int>> permuteUnique(vector<int>& nums) {
        vector<bool> used(nums.size(), false);
        sort(nums.begin(), nums.end());  // 结果集需要去重的都需要先排序
        backtracking(nums, used);
        return res;
    }
};
```

重新安排行程 #332

```
class Solution {
public:
    unordered_map<string, map<string, int>> targets;
    vector<string> res;
    bool backtracking(int ticketsNum) {
        if (res.size() == ticketsNum + 1) return true;  // 欧拉回路，res.size() = 顶点数 = 边数 + 1 = ticketsNum + 1

        // &引用，一定是遍历res中最末尾的节点对应的targets
        // const string 是因为 map 中的 key 是不能改变的
        for (pair<const string, int>& tar : targets[res.back()]) {
            if (tar.second > 0) {
                tar.second--;
                res.push_back(tar.first);
                if(backtracking(ticketsNum)) return true;  // 这个很关键，只需要找到一条合适的到达叶子节点的路径
                res.pop_back();
                tar.second++;  // 别忘了
            }
```

```
        }
        return false;
    }
    vector<string> findItinerary(vector<vector<string>>& tickets) {
        for(const vector<string>& ticket : tickets) {  // const
            targets[ticket[0]][ticket[1]]++;
        }
        res.push_back("JFK");
        backtracking(tickets.size());
        return res;
    }
};
```

N皇后 #51

```
class Solution {
public:
    vector<vector<string>> res;
    vector<string> chessboard;
    void backtracking(int n, int row) {
        if (row == n) {
            res.push_back(chessboard);
            return;
        }

        // row代表行，深度遍历；col代表列，for层遍历
        for (int col = 0; col < n; col++) {
            if (isValid(n, row, col)) {
                chessboard[row][col] = 'Q';
                backtracking(n, row + 1);
                chessboard[row][col] = '.';
            }
        }

    }
    bool isValid(int n, int row, int col) {
        // 上方的列的方向，下方不用考虑，因为会回溯回来
        for (int i = 0; i < row; i++) {
            if (chessboard[i][col] == 'Q') return false;
        }
       // 左斜上45°方向，右斜下45°方向不用考虑
        for (int i = row - 1, j = col - 1; i >= 0 && j >= 0; i--, j--) {
            if(chessboard[i][j] == 'Q') return false;
        }
        // 右斜上45°方向，左斜下45°方向不用考虑
        for (int i = row - 1, j = col + 1; i >= 0 && j < n; i--, j++) {
            if(chessboard[i][j] == 'Q') return false;
        }
        return true;

    }
    vector<vector<string>> solveNQueens(int n) {
        chessboard = vector<string>(n, string(n, '.'));
        backtracking(n, 0);
        return res;
    }
};
```

解数独 #37

```
class Solution {
public:
    // 符合条件的才返回，其他的丢弃，所以有bool返回值
    bool backtracking(vector<vector<char>>& board) {
        // N皇后:只需要一层for循环遍历一行，递归来来遍历列，因为每一行每一列只放一个皇后
        // 数独:二维递归，先遍历二维的结构，两个for循环嵌套一个递归，递归遍历数字1-9
        //       不需要返回值，因为每次递归都比之前多填了一个数，当数填满之后就自动停止了
        for(int row = 0; row < 9; row++) {
            for(int col = 0; col < 9; col++) {
                if(board[row][col] != '.') continue;
                for(char ch = '1'; ch <= '9'; ch++) {
                    if(isValid(row, col, ch, board)) {
                        board[row][col] = ch;
                        if(backtracking(board)) return true;
                        board[row][col] = '.';
                    }
                }
                return false;  // 尝试了1-9个数都不行，那么说明无解
            }
        }
        return true;
    }
    bool isValid(int row, int col,char ch, vector<vector<char>>& board) {
        // 行列
        for(int i = 0; i < 9; i++) {
            if(board[row][i] == ch) return false;
            if(board[i][col] == ch) return false;
        }

        // 3x3格子
        int startRow = row / 3 * 3;
        int startCol = col / 3 * 3;
        for(int i = startRow; i < startRow + 3; i++) {
            for(int j = startCol; j < startCol + 3; j++) {
                if(board[i][j] == ch) return false;
            }
        }
        return true;
    }
    void solveSudoku(vector<vector<char>>& board) {
        backtracking(board);
        return;
    }
};
```

☆贪心

分发饼干 #455

```
class Solution {
public:
    int findContentChildren(vector<int>& g, vector<int>& s) {
        sort(g.begin(), g.end());
        sort(s.begin(), s.end());

        int index_g = 0;
        // 遍历孩子，小饼干先喂饱胃口小的孩子
        // index_g 也是一个技巧，不要写两个循环
        for(int i = 0; i < s.size(); i++) {
            if(index_g < g.size() && g[index_g] <= s[i]) {
                index_g++;
            }
        }
        return index_g;
    }
};
```

摆动序列 #376

```
class Solution {
public:
    int wiggleMaxLength(vector<int>& nums) {
        int curDiff = 0;
        int preDiff = 0;  // 为了好比较
        int res = 1;  // 默认为1，因为长度为2的数组摆动序列长度为2

        for(int i = 0; i < nums.size() -  1; i++) {
            curDiff = nums[i + 1] - nums[i];
            // 这里没有 curDiff=0 的情况，说明自动忽略了平坡的情况
            if((curDiff > 0 && preDiff <= 0) || (curDiff < 0 && preDiff >= 0)) {
                res++;
                preDiff = curDiff;
            }
        }

        return res;
    }
};
```

最大子数组和 #53

```
class Solution {
public:
    int maxSubArray(vector<int>& nums) {
        if(nums.size() == 1) return nums[0];
        int sum = 0;
        int res = INT_MIN;
        for(int i = 0; i < nums.size(); i++) {
            sum += nums[i];
            if(sum > res) {
                res = sum;
            }
```

```
            if(sum < 0) sum = 0;  // 如果当前和小于0，一定重新开始计，因为加负数一定更小
        }

        return res;
    }
};
```

买卖股票的最佳时机II #122

```
class Solution {
public:
    int maxProfit(vector<int>& prices) {
        if(prices.size() == 1) return 0;
        int res = 0;
        for(int i = 1; i < prices.size(); i++) {
            res += max(0, prices[i] - prices[i - 1]);  // 只收集正利润
        }
        return res;
    }
};
```

跳跃游戏 #55

```
class Solution {
public:
    bool canJump(vector<int>& nums) {
        int index = 0;
        for(int i = 0; i < nums.size(); i++) {
            if(index < i) return false;
            index = max(index, i + nums[i]);  // 每次移动，更新跳跃的最大范围

        }
        return true;
    }
};
```

跳跃游戏II #45

```
class Solution {
public:
    int jump(vector<int>& nums) {
        if (nums.size() == 1) return 0;
        int curDistance = 0, nextDistance = 0;
        int step = 0;

        // 关键在于不知道当前要跳多少步，这个方法比较巧妙，不去考虑跳多少步，而是决定在什么时候加步数
        // 还是一个一个遍历，遍历的时候不断更新覆盖范围，当前坐标跟当前覆盖范围相等的时候，就要加步数了
        for (int i = 0; i < nums.size(); i++) {
            nextDistance = max(nums[i] + i, nextDistance);
            if (i == curDistance) {
                step++;
                curDistance = nextDistance;
```

```
            } else {
                continue;
            }
            if (nextDistance >= nums.size() - 1) break;
        }

        return step;
    }
};
```

K次取反后最大化的数组和 #1005

```
class Solution {
static bool cmp(int a, int b) {
    return abs(a) > abs(b);
}
public:
    int largestSumAfterKNegations(vector<int>& nums, int k) {
        int res = 0;
        sort(nums.begin(), nums.end(), cmp);  // 按照绝对值大小，从大到小排序
        for (int i = 0; i < nums.size(); i++) {
            if(nums[i] < 0 && k > 0) {  // 1. 局部最优，优先把绝对值大的负数转为正数
                nums[i] *= -1;
                k--;
            }
        }
        // while (k) {
        //     nums[nums.size() - 1] *= -1;
        //     k--;
        // }
        if(k % 2 == 1) nums[nums.size() - 1] *= -1;  // 2. 如果k仍然>0，说明负数都处理过了，那么在绝对值最小的数上进行重复翻转
        for(int a : nums) res += a;

        return res;
    }
};
```

加油站 #134

```
class Solution {
public:
    int canCompleteCircuit(vector<int>& gas, vector<int>& cost) {
        int sum = 0;
        int min_gas = INT_MAX;
        int min_index;

        // 想象折线图，找到折线图的最低点，将整个折线图移到x轴以上，x轴的交点即是最低点，最低点下一个点就是起始的点
        // 最低点的这个点的gas-cost肯定是你负的，否则不会是最低，所以从当前这个点开始肯定不行，要从下一个点开始
        // 无论最低点前边有多少个负数，从下一个节点开始会一点点弥补回来
        for (int i = 0; i < gas.size(); i++) {
            sum += gas[i] - cost[i];
            if (sum < min_gas) {
                min_gas = sum;
```

```
                min_index = i;
            }
        }

        return sum < 0 ? -1 : (min_index + 1) % gas.size();
    }
};
```

分发糖果 #135

```
class Solution {
public:
    int candy(vector<int>& ratings) {
        vector<int> candy(ratings.size(), 1);
        // 先从左往右遍历，如果右边比左边分数高，那么右边糖果数+1
        for (int i = 0; i < ratings.size() - 1; i++) {
            if (ratings[i + 1] > ratings[i]) {
                candy[i + 1] = candy[i] + 1;  // candy[i + 1]++ 这样写不对
            }
        }
        // 再从右往左遍历，而且要从后往前，如果左边比右边分数高
        // 那么左边糖果数等于max(之前更新的，右边+1),要考虑之前更新的值，就是为了保持之前的比较关系
        for (int i = ratings.size() - 1; i > 0; i--) {
            if (ratings[i - 1] > ratings[i]) {
                candy[i - 1] = max(candy[i - 1], candy[i] + 1);
            }
        }

        int res = 0;
        for (int a : candy) res += a;
        return res;
    }
};
```

柠檬水找零 #860

```
class Solution {
public:
    bool lemonadeChange(vector<int>& bills) {
        // 划分为三个情况去分析
        int five = 0, ten = 0, twenty = 0;
        for (int bill : bills) {
            if (bill == 5) {
                five++;
            } else if (bill == 10) {
                if (five <= 0) return false;
                five--;
                ten++;
            } else if (bill == 20) {
                // 付款20的时候，优先找5+10，其次5*3，因为5的作用更多，还可以应对付款10的情况
                if (five > 0 && ten > 0) {
                    five--;
                    ten--;
                    twenty++;
```

```
                } else if (five >= 3) {
                    five -= 3;
                    twenty++;
                } else return false;
            }
        }
        return true;
    }
};
```

根据身高重建队列 #406

```
class Solution {
public:
    static bool cmp(vector<int>& a, vector<int>& b) {
        if(a[0] == b[0]) return a[1] < b[1];  // 意思是a[1]如果小于b[1]的话，a[1]在前边
        return a[0] > b[0];  // [0]大的在前边
    }
    vector<vector<int>> reconstructQueue(vector<vector<int>>& people) {
        sort(people.begin(), people.end(), cmp);
        list<vector<int>> que;  // 底层实现是链表，插入速度更快

        for (int i = 0; i < people.size(); i++) {
            int k = people[i][1];
            list<vector<int>>::iterator it = que.begin();  // 注意语法
            while(k--) {
                it++;
            }
            que.insert(it, people[i]);
        }

        return vector<vector<int>>(que.begin(), que.end());
    }
};
```

用最少数量的箭引爆气球 #452

```
class Solution {
public:
    static bool cmp(vector<int>& a, vector<int>& b) {
        return a[0] < b[0];
    }
    int findMinArrowShots(vector<vector<int>>& points) {
        if (points.size() == 1) return 1;
        sort(points.begin(), points.end(), cmp);

        int res = 1;
        for (int i = 1; i < points.size(); i++) {
            if (points[i][0] > points[i - 1][1]) {  // 通过右边界判断，不挨着，需要箭数+1，注意不包含等于
                res++;
            } else {  // 如果挨着，右边界统一到右边气球的左坐标上，这样其实points[i]的区间就代表了重叠的区域
                points[i][1] = min(points[i - 1][1], points[i][1]);
            }
```

```
        }

        return res;
    }
};
```

无重叠区间 #435

```
class Solution {
public:
    static bool cmp(vector<int>& a, vector<int>& b) {
        return a[1] < b[1];  // 按照右边界，从小到大排序
    }
    int eraseOverlapIntervals(vector<vector<int>>& intervals) {
        if (intervals.size() == 1) return 0;

        sort(intervals.begin(), intervals.end(), cmp);
        int res = 1;
        int end = intervals[0][1];
        for (int i = 1; i < intervals.size(); i++) {  //  从左到右遍历
            if (intervals[i][0] >= end) {  // 计算非重叠区域的个数，重叠的自动跳过
                res++;
                end = intervals[i][1];
            }
        }
        return intervals.size() - res;
    }
};
```

划分字母区间 #763

```
class Solution {
public:
    vector<int> partitionLabels(string s) {
        int hash[26] = {0};
        for (int i = 0; i < s.size(); i++) {
            hash[s[i] - 'a'] = i;  // 记录当前字符出现的最远位置
        }

        int left = 0, right = 0;
        vector<int> res;
        for (int i = 0; i < s.size(); i++) {
            right = max(right, hash[s[i] - 'a']);  // 记录一段字符中最远的边界
            if (i == right) {  // 找到之前字符出现的最大出现位置和当前位置相等，即为一个片
段
                res.push_back(right - left + 1);
                left = i + 1;
            }
        }

        return res;
    }
};
```

合并区间 #56

```cpp
class Solution {
public:
    static bool cmp(vector<int>& a, vector<int>& b) {
        return a[0] < b[0];
    }
    vector<vector<int>> merge(vector<vector<int>>& intervals) {
        if (intervals.size() == 1) return intervals;
        sort(intervals.begin(), intervals.end(), cmp);

        vector<vector<int>> res;
        for (int i = 1; i < intervals.size(); i++) {
            if (intervals[i][0] <= intervals[i - 1][1]) {
                intervals[i][0] = min(intervals[i - 1][0], intervals[i][0]);
                intervals[i][1] = max(intervals[i - 1][1], intervals[i][1]);
            } else {
                res.push_back(intervals[i - 1]);
            }
        }
        res.push_back(intervals.back());

        return res;
    }
};
```

单调递增的数字 #738

```cpp
class Solution {
public:
    int monotoneIncreasingDigits(int n) {
        if(n < 10) return n;

        string s = to_string(n);  // to_string()
        int flag = s.size();  // 记录需要赋值为9的最初位置，后边都要赋值为9
        for (int i = s.size() - 1; i > 0; i--) {
            if (s[i - 1] > s[i]) {
                flag = i;
                s[i - 1]--;
            }
        }

        for (int i = flag; i < s.size(); i++) {
            s[i] = '9';
        }

        return stoi(s);  // stoi()
    }
};
```

监控二叉树 #968

```
/**
 * Definition for a binary tree node.
 * struct TreeNode {
 *     int val;
 *     TreeNode *left;
 *     TreeNode *right;
 *     TreeNode() : val(0), left(nullptr), right(nullptr) {}
 *     TreeNode(int x) : val(x), left(nullptr), right(nullptr) {}
 *     TreeNode(int x, TreeNode *left, TreeNode *right) : val(x), left(left),
right(right) {}
 * };
 */
class Solution {
public:
    int res = 0;
    int traverse(TreeNode* root) {
        // 0：无覆盖
        // 1：有覆盖
        // 2：有摄像头
        if (root == nullptr) return 1;
        int left = traverse(root->left);
        int right = traverse(root->right);
        if (left == 1 && right == 1) {  //  11 两个孩子都覆盖，说明父节点肯定无覆盖，这
里先不要急着加摄像头
            return 0;
        } else if (left == 0  || right == 0) {  // 00,01,02,10,20 只要有一个无覆盖，
父节点一定要加摄像头，否则跨过去就落下了
            res++;
            return 2;
        } else {  // 12,21,22
            return 1;
        }
    }
    int minCameraCover(TreeNode* root) {
        if (traverse(root) == 0) {  // 处理根节点无覆盖的情况
            res++;
        }
        return res;
    }
};
```

☆动态规划

动规要素

- 确定dp数组和下标的含义
- 确定递推公式
- dp数组初始化
- 确定遍历顺序
- 举例推导dp数组

斐波那契数 #509

```
class Solution {
public:
    int fib(int n) {
        if (n < 2) return n;

        int dp[n + 1];
        dp[0] = 0;
        dp[1] = 1;
        for (int i = 2; i <= n; i++) {
            dp[i] = dp[i - 1] + dp[i - 2];
        }

        return dp[n];
    }
};
```

爬楼梯 #70

```
class Solution {
public:
    int fib(int n) {
        if (n < 2) return n;

        int dp[n + 1];
        dp[0] = 0;
        dp[1] = 1;
        for (int i = 2; i <= n; i++) {
            dp[i] = dp[i - 1] + dp[i - 2];
        }

        return dp[n];
    }
};
```

不同路径 #62

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

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

```
};
```

不同路径II #63

```
class Solution {
public:
    int uniquePathsWithObstacles(vector<vector<int>>& obstacleGrid) {
        int m = obstacleGrid.size();
        int n = obstacleGrid[0].size();

        if (obstacleGrid[0][0] == 1 || obstacleGrid[m - 1][n - 1]) return 0;
        vector<vector<int>> dp(m, vector<int>(n, 0));
        for (int i = 0; i < m && obstacleGrid[i][0] == 0; i++) dp[i][0] = 1;
        for (int j = 0; j < n && obstacleGrid[0][j] == 0; j++) dp[0][j] = 1;

        for (int i = 1; i < m; i++) {
            for (int j = 1; j < n; j++) {
                if (obstacleGrid[i][j] == 1) continue;
                dp[i][j] = dp[i - 1][j] + dp[i][j - 1];
            }
        }

        return dp[m - 1][n - 1];
    }
};
```

整数拆分 #343

```
class Solution {
public:
    int integerBreak(int n) {
        vector<int> dp(n + 1, 0);
        dp[2] = 1;  // dp[0],dp[1]没有意义

        // i从3开始遍历，j代表从1-(i-1)进行拆分
        for (int i = 3; i <= n; i++) {
            for (int j = 1; j <= i - 1; j++) {
                // 1. 要比较dp[i]，因为这个循环可能随时变化
                // 2. 为什么要比较 (i-j)*j，因为 dp[i-j]里不包括拆分为 i-j 的情况
                // 3. 可以这样理解，(i-j)*j 是拆分为两个数的情况，dp[i-j]*j 是拆分为两个
数及以上的情况
                dp[i] = max(dp[i], max(dp[i - j] * j, (i - j) * j));
            }
        }

        return dp[n];
    }
};
```

不同的二叉搜索树 #96

```
class Solution {
public:
    int numTrees(int n) {
        vector<int> dp(n + 1);
        dp[0] = 1;
        dp[1] = 1;

        // 1...i 一共 i 个节点
        for (int i = 2; i <= n; i++) {
            // 以 j 为头节点， 根据二叉搜索树规则，左子树为 j-1 个节点， 右子树为 i-j 个节点
            for (int j = 1; j <= i; j++) {
                dp[i] += dp[j - 1] * dp[i - j];
            }
        }

        return dp[n];
    }
};
```

0-1背包模板

```
void bag_01() {
    vector<int> weight = {1, 3, 4};
    vector<int> value = {15, 20, 30};
    int bagweight = 4;

    // 二维数组-模板
    vector<vector<int>> dp(weight.size(), vector<int>(value.size(), 0));
    // 初始化
    for (int j = weight[0]; j <= bagweight; j++) {
        dp[0][j] = value[0];
    }
    // i,j 都从 1 开始遍历，因为下标为 0 的都初始化过了
    for (int i = 1; i < weight.size(); i++) {
        for (int j = 1; j <= bagweight; j++) {
            if (j < weight[i]) {  // 不选i
                dp[i][j] = dp[i - 1][j];
            } else {              // 选i
                dp[i][j] = max(dp[i - 1][j], dp[i - 1][j - weight[i]] + value[i]);
            }
        }
    }

    // 一维数组-模板
    vector<int> dp(bagWeight + 1, 0);  // 注意dp数组定义的大小是 bagWeight+1
    for (int i = 0; i < weight.size(); i++) {
         // 背包重量的【遍历必须在里边】。如果放在外边，那么背包只放了一个物品；放在里边，相当于考虑了 0...i 这些物品
         // 一维dp数组，用【倒序】，保证了物品不会重复放，而且本质还是对二维数组的遍历，右下角的值依赖上一层左上角的值
         // j >= nums[i] 防止下标溢出
         for (int j = bagweight; j >= weight[i]; j--) {
```

```
            dp[j] = max(dp[j], dp[j - weight[i]] + value[i]);
        }
    }

    cout << dp[weight.size() - 1][bagweight] << endl;
}

int main() {
    bag_01();
}
```

分割等和子集 #416

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

        // 每个元素最大是100，数组长度最大是200，所以一般和最大为10000
        int sum = 0;
        for (int i = 0; i < nums.size(); i++) {
            sum += nums[i];
        }
        if (sum % 2 == 1) return false;
        int target = sum / 2;
        vector<int> dp(target, 0);

        // dp[j]代表背包重量为 j 时，可以得到的最大的value值，这里即最大的子集的和
        // 背包重量 j 最大为 target， dp[j]最大值时是 j，j取最大时也就是 target
        for (int i = 0; i < nums.size(); i++) {
            // 背包重量的【遍历必须在里边】。如果放在外边，那么背包只放了一个物品；放在里边，
相当于考虑了 0...i 这些物品
            // 一维dp数组，用【倒序】，保证了物品不会重复放，而且本质还是对二维数组的遍历，右
下角的值依赖上一层左上角的值
            // j >= nums[i] 防止下标溢出
            for (int j = target; j >= nums[i]; j--) {
                dp[j] = max(dp[j], dp[j - nums[i]] + nums[i]);
            }
        }

        if (dp[target] == target) return true;
        return false;
    }
};
```

最后一块石头的重量II #1049

```
class Solution {
public:
    int lastStoneWeightII(vector<int>& stones) {
        int sum = 0;
        for (int i = 0; i < stones.size(); i++) {
            sum += stones[i];
        }
        int target = sum / 2;
        vector<int> dp(target + 1, 0);
```

```
        for (int i = 0; i < stones.size(); i++) {
            for (int j = target; j >= stones[i]; j--) {
                dp[j] = max(dp[j], dp[j - stones[i]] + stones[i]);
            }
        }

        return sum - dp[target] * 2;
    }
};
```

目标和 #494

```
class Solution {
public:
    int findTargetSumWays(vector<int>& nums, int target) {
        int sum = 0;
        for (int num : nums) sum += num;
        if (abs(target) > sum) return 0;            // S有可能是负数
        if ((target + sum) % 2 == 1) return 0;
        int bagSize = (target + sum) / 2;

        // 初始化
        // dp[0]=1，不能为0，因为后边的累加都要依赖于dp[0]，可以解释为装满容量为0的背包有1种方法，也就是装0件物品
        // 其他的要初始化为0，否则会对累加的结果有影响
        vector<int> dp(bagSize + 1, 0);
        dp[0] = 1;
        for (int i = 0; i < nums.size(); i++) {
            // 求装满背包有几种方法的情况下，递推公式一般为 dp[j] = dp[j - nums[i]]
            for (int j = bagSize; j >= nums[i]; j--) {
                dp[j] += dp[j - nums[i]];
            }
        }

        return dp[bagSize];
    }
};
```

一和零 #474

```
class Solution {
public:
    int findMaxForm(vector<string>& strs, int m, int n) {
        vector<vector<int>> dp(m + 1, vector<int>(n + 1, 0));

        // dp[i][j]中的 i 不是二维dp中的物品序号，这里也是重量，只不过这个是重量为两维的一维dp背包问题
        // 对于strs的遍历相当于是对物品的遍历
        for (string str : strs) {
            int zero_num = 0, one_num = 0;
            for (char ch : str) {
                if (ch == '0') {
                    zero_num++;
                } else {
                    one_num++;
                }
```

```
            }
            // 这里两层循环都是对重量的遍历，只不过这个是重量为两维的一维dp背包问题
            // 本质还是采用的一维dp，所以都采用倒序，前边初始化为0就行，嵌套顺序在里边在外边无所谓
            for (int i = m; i >= zero_num; i--) {
                for (int j = n; j >= one_num; j--) {
                    dp[i][j] = max(dp[i][j], dp[i - zero_num][j - one_num] + 1);
                }
            }
        }
        return dp[m][n];
    }
};
```

完全背包模板

```
// 0-1背包
for(int i = 0; i < weight.size(); i++) { // 遍历物品
    for(int j = bagWeight; j >= weight[i]; j--) { // 遍历背包容量，倒序
        dp[j] = max(dp[j], dp[j - weight[i]] + value[i]);
    }
}
// 完全背包
// 0-1背包内嵌循环是从大到小倒序遍历，是因为要保证每个物品仅被添加一次
// 完全背包的物品可以重复添加，所以从小到大正序遍历
// 两个for循环的顺序都可以，但是问装满背包有几种方式的时候， for循环的顺序就有区别了，要根据实际题目而变
for(int i = 0; i < weight.size(); i++) { // 遍历物品
    for(int j = weight[i]; j <= bagWeight; j++) { // 遍历背包容量，正序
        dp[j] = max(dp[j], dp[j - weight[i]] + value[i]);
    }
}
```

零钱兑换II #518

```
class Solution {
public:
    int change(int amount, vector<int>& coins) {
        // if (amount == 0) return 0; // 这一行不要加
        vector<int> dp(amount + 1, 0);

        // 注意此题目是要求装成背包重量的组合数
        // 1. 求总和的时候，推导公式一般为 dp[j] += dp[j - coins[i]]
        // 2. 求总和的时候，dp[0]=1是必须的，否则后边没法累加，但是非0下标的必须为0，防止多加
        // 3. 不仅求总和，而且是组合数，也就是{1,5}和{5,1}是一种情况，此时必须先遍历物品，先把1放进去计算，后把5放进去
        dp[0] = 1;
        for (int i = 0; i < coins.size(); i++) {
            for (int j = coins[i]; j <= amount; j++) {
                dp[j] += dp[j - coins[i]];
            }
        }

        return dp[amount];
    }
```

```
};
```

组合总和(排列) #377

```
class Solution {
public:
    int combinationSum4(vector<int>& nums, int target) {
        vector<int> dp(target + 1, 0);

        // 1. 一定要初始化dp[0]
        // 2. 组合：背包遍历在内
        //    排列：背包遍历在外
        // 3. 完全背包：内循环从小到大遍历
        dp[0] = 1;
        for (int j = 0; j <= target; j++) {
            // for (int i = 0; nums[i] <= j; i++) { 这样判断不行
            for (int i = 0; i < nums.size(); i++) {
                if (nums[i] <= j && dp[j] < INT_MAX - dp[j - nums[i]]) { // c++用例，有两个数相加会溢出，需要判断一下
                    dp[j] += dp[j - nums[i]];
                }
            }
        }

        return dp[target];
    }
};
```

爬楼梯 #70

```
class Solution {
public:
    int climbStairs(int n) {
        vector<int> dp(n + 1, 0);

        dp[0] = 1;
        // 相当于组合总和那个题，数组只有两个数 {1，2}，然后总和为 n
        // 把内循环改为 i<=m 就是相当于可以爬 1.2.3....m 阶台阶的答案
        for (int j = 0; j <=n; j++) {
            for (int i = 1; i <= 2; i++) {
                if (i <= j) {
                    dp[j] += dp[j - i];
                }
            }
        }

        return dp[n];
    }
};
```

零钱兑换 #322

```
class Solution {
public:
    int coinChange(vector<int>& coins, int amount) {
        vector<int> dp(amount + 1, INT_MAX);

        // dp[0]的初始化一定要好好想清楚，要模拟一下数组，这里是0否则后边比较的时候总是赋值为INT_MAX
        // 非零下标的初始化一般初始化值是一样的，因为这个题要比较最小值，所以非零下标初始化为INT_MAX
        dp[0] = 0;
        for (int i = 0; i < coins.size(); i++) {
            for (int j = coins[i]; j <= amount; j++) {
                if (dp[j - coins[i]] < INT_MAX) {  // 要加一个判断，否则如果dp[j-coins[i]]==INT_MAX，后边再加1就溢出了
                    dp[j] = min(dp[j], dp[j - coins[i]] + 1);  // 别忘了+1
                }
            }
        }

        return dp[amount] == INT_MAX ? -1 : dp[amount];
    }
};
```

完全平方数 #279

```
class Solution {
public:
    int numSquares(int n) {
        vector<int> dp(n + 1, INT_MAX);

        dp[0] = 0;
        int m = sqrt(n);
        for (int i = 1; i <= m; i++) {
            for (int j = i * i; j <= n; j++) {
                if (dp[j - i * i] < INT_MAX) {
                    dp[j] = min(dp[j], dp[j - i * i] + 1);
                }
            }
        }

        return dp[n];
    }
};
```

单词拆分 #139

```
class Solution {
public:
    bool wordBreak(string s, vector<string>& wordDict) {
        unordered_set wordSet(wordDict.begin(), wordDict.end());
        vector<int> dp(s.size() + 1, 0);

        // 1. 背包是字符串s，字典里的每一个字符串是物品
```

```
        // 2. 这个题特殊就特殊在dp[i]的含义上，dp[i]代表长度为i的子串，能不能由字典里的单词拼接成的结果，i 是长度！！
        // 3. dp[i]对应子串的下标是 [0,...i-1]，s.substr(j, i-j)对应下标 [j,...,i-1]
        // 4. dp[0]一定是true，否则永远进不到if那一句里，非零下标的初始化为false
        dp[0] = true;
        for (int i = 1; i <= s.size(); i++) {
            for (int j = 0; j < i; j++) {
                string substr = s.substr(j, i - j);  // substr(截取开始下标，截取长度)
                if (wordSet.find(substr) != wordSet.end() && dp[j] == true) {
                    dp[i] = true;
                }
            }
        }

        return dp[s.size()];
    }
};
```

多重背包模板

```
void test_multi_pack() {
    vector<int> weight = {1, 3, 4};
    vector<int> value = {15, 20, 30};
    vector<int> nums = {2, 3, 2};
    int bagWeight = 10;
    vector<int> dp(bagWeight + 1, 0);


    for(int i = 0; i < weight.size(); i++) { // 遍历物品
        for(int j = bagWeight; j >= weight[i]; j--) { // 遍历背包容量
            // 以上为01背包，然后加一个遍历个数
            for (int k = 1; k <= nums[i] && (j - k * weight[i]) >= 0; k++) { // 遍历个数
                dp[j] = max(dp[j], dp[j - k * weight[i]] + k * value[i]);
            }
        }
        // 打印一下dp数组
        for (int j = 0; j <= bagWeight; j++) {
            cout << dp[j] << " ";
        }
        cout << endl;
    }
    cout << dp[bagWeight] << endl;
}
int main() {
    test_multi_pack();
}
```

打家劫舍 #198

```
class Solution {
public:
    int rob(vector<int>& nums) {
        if (nums.size() == 0) return 0;
        if (nums.size() == 1) return nums[0];
```

```
        vector<int> dp(nums.size(), 0);
        dp[0] = nums[0];
        dp[1] = max(nums[0], nums[1]);
        // 1. dp[i]代表下标 [0,...i] 数组的偷窃最高金额
        // 2. i 偷的话, dp[i] = dp[i-2] + nums[i], 完全不考虑 i-1 的情况
        // 3. i 不偷的话, dp[i] = dp[i - 1], 考虑 i-1 的情况, 但是并不意味着 i-1 是要偷
的
        for (int i = 2; i < nums.size(); i++) {
            dp[i] = max(dp[i - 1], dp[i - 2] + nums[i]);
        }

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

打家劫舍II #213

```
class Solution {
public:
    int rob(vector<int>& nums) {
        if (nums.size() == 0) return 0;
        if (nums.size() == 1) return nums[0];

        // 有三种情况：不考虑首尾元素，不考虑首元素，不考虑尾元素。
        // 第二三种情况包含了第一种，所以考虑后两种即可。注意考虑，不等于就一定要选那个元素。
        int left = rob_range(nums, 0, nums.size() - 2);
        int right = rob_range(nums, 1, nums.size() - 1);
        return max(left, right);
    }
    int rob_range(vector<int>& nums, int begin, int end) {
        // [begin, end]
        if (end == begin) return nums[begin];

        // 1. 一定注意，所有标了 begin、end 下标的题目，都要切记不要从下标为 0，1 开始赋值
了
        // 2. dp 的长度还是声明为 nums.size() 个，但是只用 [begin,end] 这一部分，方便后
边赋值
        vector<int> dp(nums.size(), 0);  // dp[i] : 0,...,i => 实际只考虑
[begin,...,end] 范围内的数
        dp[begin] = nums[begin];
        dp[begin + 1] = max(nums[begin], nums[begin + 1]);

        for (int i = begin + 2; i <= end; i++) {
            dp[i] = max(dp[i - 1], dp[i - 2] + nums[i]);
        }

        return dp[end];
    }
};
```

打家劫舍III #337

```
/**
 * Definition for a binary tree node.
 * struct TreeNode {
 *     int val;
 *     TreeNode *left;
 *     TreeNode *right;
 *     TreeNode() : val(0), left(nullptr), right(nullptr) {}
 *     TreeNode(int x) : val(x), left(nullptr), right(nullptr) {}
 *     TreeNode(int x, TreeNode *left, TreeNode *right) : val(x), left(left),
right(right) {}
 * };
 */
class Solution {
public:
    int rob(TreeNode* root) {
        vector<int> res = rob_tree(root);
        return max(res[0], res[1]);
    }
    vector<int> rob_tree(TreeNode* root) {
        if (root == nullptr) return {0, 0};

        // 1. 某个节点偷不偷是两个状态，不太好用一个数表示，所以用两个数表示
        // 2. 树形dp的典型题，确定遍历顺序是最关键的
        //    偷不偷当前节点的最后结果值依赖于子节点的结果，所以采用后序遍历
        vector<int> left = rob_tree(root->left);
        vector<int> right = rob_tree(root->right);
        // 3. res1：不偷当前节点，那么就考虑两个子节点的各自两个状态，选最大的那个
        //    res2：偷的话，就当前节点值加上两个子节点不偷的值
        int res1 = max(left[0], left[1]) + max(right[0], right[1]);
        int res2 = root->val + left[0] + right[0];


        return {res1, res2};
    }
};
```

买卖股票的最佳时机 #121

```
// 贪心
class Solution {
public:
    int maxProfit(vector<int>& prices) {
        int low = INT_MAX;
        int res = 0;
        // 每次记录左边最小的值，并且每次都计算最大的间距
        // 因为不一定是减去最小的值才是最大的间距
        for (int i = 0; i < prices.size(); i++) {
            low = min(low, prices[i]);
            res = max(res, prices[i] - low);
        }
        return res;
    }
};
// 动规
```

```
class Solution {
public:
    int maxProfit(vector<int>& prices) {
        // dp[i]代表下标为 i 的这个股票是否持有的情况下的现金数量，这题难在怎么定义dp
        // 用两个状态去表示，dp[i][0] 代表不持有第i支股票的现金数，dp[i][1]代表持有的现金
        vector<vector<int>> dp(prices.size(), vector<int>(2, 0));
        dp[0][0] = 0;
        dp[0][1] = -prices[0];
        for (int i = 1; i < prices.size(); i++) {
            // 1. 不持有的话，由两个状态推出，dp[i-1]不持有那么继续保持，dp[i-1]持有的话那么就相当于以prices[i]的价格卖出去
            // 2. 持有的话，有两个状态推出，dp[i-1]持有的话继续保持，dp[i-1]不持有的话相当于以prices[i]的价格买入，现金为-prices[i]
            dp[i][0] = max(dp[i - 1][0], prices[i] + dp[i - 1][1]); // 因为买入的时候取的是负数 -prices[i]，所以这里是加号
            dp[i][1] = max(dp[i - 1][1], -prices[i]);
        }

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

买卖股票的最佳时机II #122

```
class Solution {
public:
    int maxProfit(vector<int>& prices) {
        vector<vector<int>> dp(prices.size(), vector<int>(2, 0));
        dp[0][0] = 0;
        dp[0][1] = -prices[0];

        for (int i = 1; i < prices.size(); i++) {
            dp[i][0] = max(dp[i - 1][0], prices[i] + dp[i - 1][1]);
            // 这里是跟121那个题唯一不同的地方，121题因为只会买入一次，但是这个题可以买入多次
            // 买的时候之前可能已经产生了利润，所以 -prices[i] 要加上 dp[i-1][1]
            dp[i][1] = max(dp[i - 1][1], -prices[i] + dp[i - 1][0]);
        }

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

买卖股票的最佳时机III #123

```
class Solution {
public:
    int maxProfit(vector<int>& prices) {
        // 一共五个状态：
        // 0. 没有操作  1. 第一次买入的持有  2. 第一次卖出的非持有
        //              3. 第二次买入的持有  4. 第二次卖出的非持有
        // dp[i][j] 表示第 i 天，j 为[0-4] 五个状态，dp[i][j]表示第i天状态j时的最大现金数
        vector<vector<int>> dp(prices.size(), vector<int>(5, 0));
        dp[0][0] = 0;
        dp[0][1] = -prices[0];
        dp[0][2] = 0;
        dp[0][3] = -prices[0];
```

```
        dp[0][4] = 0;
        for (int i = 1; i < prices.size(); i++) {
            dp[i][0] = dp[i - 1][0];
            dp[i][1] = max(dp[i - 1][1], -prices[i] + dp[i - 1][0]);
            dp[i][2] = max(dp[i - 1][2], prices[i] + dp[i - 1][1]);
            dp[i][3] = max(dp[i - 1][3], -prices[i] + dp[i - 1][2]);
            dp[i][4] = max(dp[i - 1][4], prices[i] + dp[i - 1][3]);
        }

        return dp[prices.size() - 1][4];
    }
};
```

买卖股票的最佳时机IV #188

```
class Solution {
public:
    int maxProfit(int k, vector<int>& prices) {
        if (prices.size() == 0) return 0;
        vector<vector<int>> dp(prices.size(), vector<int>(1 + 2 * k, 0));
        for (int i = 1; i <= k; i++) {
            dp[0][2 * i - 1] = -prices[0];
            dp[0][2 * i] = 0;
        }

        for (int i = 1; i < prices.size(); i++) {
            for (int j = 1; j <= k; j++) {
                dp[i][2 * j - 1] = max(dp[i - 1][2 * j - 1], -prices[i] + dp[i -
1][2 * j - 2]);
                dp[i][2 * j] = max(dp[i - 1][2 * j], prices[i] + dp[i - 1][2 * j
- 1]);
            }
        }

        return dp[prices.size() - 1][2 * k];
    }
};
```

最佳买卖股票时机含冷冻期 # 309

```
class Solution {
public:
    int maxProfit(vector<int>& prices) {
        // 四个状态：
        // 0. 持有股票，可以是今天刚买的，也可以是之前买的一直没卖
        // 1. 不持有股票，今天卖的
        // 2. 不持有股票，而且是之前卖的，并且过了冷冻期，也就是说最晚是前天卖的
        // 3. 不持有股票，今天是冷冻期，也就是说是昨天卖的
        vector<vector<int>> dp(prices.size(), vector<int>(4, 0));
        dp[0][0] = -prices[0];

        for (int i = 1; i < prices.size(); i++) {
            dp[i][0] = max(dp[i - 1][0], max(-prices[i] + dp[i - 1][2], -
prices[i] + dp[i - 1][3]));
            dp[i][1] = prices[i] + dp[i - 1][0];
            dp[i][2] = max(dp[i - 1][2], dp[i - 1][3]);
```

```
            dp[i][3] = dp[i - 1][1];
        }

        return max(dp[prices.size() - 1][1], max(dp[prices.size() - 1][2],
dp[prices.size() - 1][3]));

    }
};
```

买卖股票的最佳时机含手续费 #714

```
class Solution {
public:
    int maxProfit(vector<int>& prices, int fee) {
        vector<vector<int>> dp(prices.size(), vector<int>(2, 0));
        dp[0][1] = -prices[0];

        for (int i = 1; i < prices.size(); i++) {
            dp[i][0] = max(dp[i - 1][0], prices[i] - fee + dp[i - 1][1]);
            dp[i][1] = max(dp[i - 1][1], -prices[i] + dp[i - 1][0]);
        }

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

最长递增子序列 #300

```
class Solution {
public:
    int lengthOfLIS(vector<int>& nums) {
        // 非连续
        // 1. dp的含义如何定义很难，定义的原则就是方便推导出转移方程，如果不好推倒那么要想是不
是换一种定义
        // 2. 此题目，定义 dp[i] 是 从下标 0 到以 nums[i] 这个元素为结尾的子序列的最长严格
递增子序列的长度
        // 3. 注意一定是以 nums[i] 为结尾，这样就容易推导出转移方程了，要在 [0,...,i-1] 这
个范围进行遍历
        // 4. 每一个元素至少本身是严格递增子序列，所以初始化为 1
        vector<int> dp(nums.size(), 1);
        int res = 1;
        // 这个两层循环，也非常重要
        for (int i = 1; i < nums.size(); i++) {
            for (int j = 0; j < i; j++) {
                if (nums[i] > nums[j]) {
                    dp[i] = max(dp[i], dp[j] + 1);
                }
                // 不大于的时候，直接跳过了，相当于只看了比nums[i]小的，所以要对中间结果进行
比较
            }
            // 虽然这个题不是求连续的，但是也要在中间进行比较
            // 因为dp[i]的定义是以nums[i]为结尾，这样的定义中间都要进行比较
            // 有疑问的时候，画图模拟一遍就明白了
            if (dp[i] > res) res = dp[i];
        }
```

```
        return res;
    }
};
```

最长连续递增序列 #674

```
class Solution {
public:
    int findLengthOfLCIS(vector<int>& nums) {
        // 连续
        vector<int> dp(nums.size(), 1);
        int res = 1;

        for (int i = 1; i < nums.size(); i++) {
            if (nums[i] > nums[i - 1]) {
                dp[i] = dp[i - 1] + 1;
            } else  {
                dp[i] = 1;  // continue
            }
            if (dp[i] > res) res = dp[i];
        }

        return res;
    }
};
```

最长重复子数组 #718

```
class Solution {
public:
    int findLength(vector<int>& nums1, vector<int>& nums2) {
        // 连续
        // 1. dp[i][j] 代表以 nums1[i-1] 和 nums2[j-1] 为结尾的两个数组中最长的公共子数组的长度，子数组注意是连续的
        // 2. 推导公式: 如果 nums1[i-1] == nums[j-1]，那么 dp[i][j] = dp[i-1][j-1] + 1
        // 3. 根据数组定义和推导公式，初始化 dp[0][j] 和 dp[i][0] 都是 0
        vector<vector<int>> dp(nums1.size() + 1, vector<int>(nums2.size() + 1, 0));

        int res = 0;
        // 要两层循环，非常关键
        for (int i = 1; i <= nums1.size(); i++) {
            for (int j = 1; j <= nums2.size(); j++) {
                if (nums1[i -  1] == nums2[j  - 1]) {
                    dp[i][j] = dp[i - 1][j - 1] + 1;
                }
                if (dp[i][j] > res) res = dp[i][j];
            }
        }

        return res;
    }
};
```

最长公共子序列 #1143

```cpp
class Solution {
public:
    int longestCommonSubsequence(string text1, string text2) {
        // 非连续
        vector<vector<int>> dp(text1.size() + 1, vector<int>(text2.size() + 1,
0));

        // 1. dp[i][j] 代表 text1[0,...,i-1] 和 text2[0,...,j-1] 两个字符串最长子序列
的长度
        // 2. 两种情况：text1[i-1] ==/!= text[j-1]
        for (int i = 1; i <= text1.size(); i++) {
            for (int j = 1; j <= text2.size(); j++) {
                if (text1[i - 1] == text2[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[text1.size()][text2.size()];
    }
};
```

不相交的线 #1035

```cpp
class Solution {
public:
    int maxUncrossedLines(vector<int>& nums1, vector<int>& nums2) {
        // 就是求最长公共子序列
        vector<vector<int>> dp(nums1.size() + 1, vector<int>(nums2.size() + 1,
0));

        for (int i = 1; i <= nums1.size(); i++) {
            for (int j = 1; j <= nums2.size(); j++) {
                if (nums1[i - 1] == nums2[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[nums1.size()][nums2.size()];
    }
};
```

最大子数组和 #53

```cpp
// 贪心
class Solution {
public:
    int maxSubArray(vector<int>& nums) {
        if(nums.size() == 1) return nums[0];
        int sum = 0;
        int res = INT_MIN;
        for(int i = 0; i < nums.size(); i++) {
            sum += nums[i];
            if(sum > res) {
                res = sum;
            }
            if(sum < 0) sum = 0;
        }

        return res;
    }
};
// 动规
class Solution {
public:
    int maxSubArray(vector<int>& nums) {
        // 连续
        vector<int> dp(nums.size(), 0);
        dp[0] = nums[0];

        // 1. dp[i] 代表以 nums[i] 为结尾的最大的连续子数组的和，需要连续的话都是这样定义
        // 2. 连续的时候，必须要在过程中进行比较最后的结果
        int res = nums[0];
        for (int i = 1; i < nums.size(); i ++) {
            dp[i] = max(dp[i - 1] + nums[i], nums[i]);
            if (dp[i] > res) res = dp[i];
        }

        return res;
    }
};
```

判断子序列 #392

```cpp
// 双指针
class Solution {
public:
    bool isSubsequence(string s, string t) {
        int i = 0, j = 0;
        for (int j = 0; j < t.size(); j++) {
            if (s[i] == t[j]) {
                i++;
            }
        }
        if (i == s.size()) return true;
        return false;
    }
};
```

```cpp
// 动规
class Solution {
public:
    bool isSubsequence(string s, string t) {
        // 非连续
        vector<vector<int>> dp(s.size() + 1, vector<int>(t.size() + 1, 0));

        for (int i = 1; i <= s.size(); i++) {
            for (int j = 1; j <= t.size(); j++) {
                if(s[i - 1] == t[j - 1]) {
                    dp[i][j] = dp[i - 1][j - 1] + 1;
                } else {  // 对 t 进行删除操作，继续匹配
                    dp[i][j] = dp[i][j - 1];
                }
            }
        }
        if (dp[s.size()][t.size()] == s.size()) return true;
        return false;
    }
};
```

不同的子序列 #115

```cpp
class Solution {
public:
    int numDistinct(string s, string t) {
        // 非连续
        // int,long,long long 都会溢出，要用 unsigned long long
        vector<vector<unsigned long long>> dp(s.size() + 1, vector<unsigned long
long>(t.size() + 1, 0));
        dp[0][0] = 1;
        for (int i = 1; i <= s.size(); i++) dp[i][0] = 1; // t为空串，在s中出现一次

        for (int i = 1; i <= s.size(); i++) {
            for (int j = 1; j <= t.size(); j++) {
                // 当s[i-1]==t[j-1]时，s不一定要选择s[i-1]去匹配，还可以选择之前的去匹配
                // 不相等的时候，就只有一种情况了
                if (s[i - 1] == t[j - 1]) {
                    dp[i][j] = dp[i - 1][j - 1] + dp[i - 1][j];
                } else {
                    dp[i][j] = dp[i - 1][j];
                }
            }
        }

        return dp[s.size()][t.size()];
    }
};
```

两个字符串的删除操作 # 49

```cpp
class Solution {
public:
    int minDistance(string word1, string word2) {
        vector<vector<int>> dp(word1.size() + 1, vector<int>(word2.size() + 1,
0));
```

```
        for (int i = 1; i <= word1.size(); i++) dp[i][0] = i;
        for (int j = 1; j <= word2.size(); j++) dp[0][j] = j;

        for (int i = 1; i <= word1.size(); i++) {
            for (int j = 1; j <= word2.size(); j++) {
                if (word1[i - 1] == word2[j - 1]) {
                    dp[i][j] = dp[i - 1][j - 1];
                } else {  // 不相等就删除，3种情况：各自删除、word1删除、word2删除。
                    dp[i][j] = min({dp[i - 1][j - 1] + 2, dp[i - 1][j] + 1,
dp[i][j - 1] + 1});
                }
            }
        }

        return dp[word1.size()][word2.size()];
    }
};
```

编辑距离 #72

```
class Solution {
public:
    int minDistance(string word1, string word2) {
        vector<vector<int>> dp(word1.size() + 1, vector<int>(word2.size() + 1,
0));

        // 1. dp[i][j] 定义为以 word1[i-1] 和 word2[j-1] 为结尾的最少操作数，跟连续子序
列的定义类似
        // 2. word1[i-1] == word2[j-1] 直接 dp[i][j] = dp[i-1][j-1]
        // 3. word1[i-1] != word2[j-1] 分为三种操作： 删，增，换。
        //    其中增和删的操作达到的效果是相同的，即只考虑 删 和 换 两种操作。
        // 4. 删 ： 1) word1删除结尾元素 2)word2删除结尾元素
        //    换 ：    word1[i-1]替换为word2[j-1]
        // 5. 注意初始化不为0了，dp[0][j] = j，dp[i][0] = i 即删除 i 个操作

        for (int i = 1; i <= word1.size(); i++) dp[i][0] = i;
        for (int j = 1; j <= word2.size(); j++) dp[0][j] = j;

        for (int i = 1; i <= word1.size(); i++) {
            for (int j = 1; j <= word2.size(); 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], dp[i][j - 1], dp[i - 1][j -
1]}) + 1;
                }
            }
        }

        return dp[word1.size()][word2.size()];
    }
};
```

回文子串 #647

```cpp
class Solution {
public:
    int countSubstrings(string s) {
        vector<vector<bool>> dp(s.size(), vector<bool>(s.size(), false));
        int cnt = 0;
        // 1. dp[i][j] 代表 [i,j] 闭区间的字串是否为回文子串，bool类型
        // 2. s[i]!=s[j] 肯定是false
        // 3. s[i]==s[j] 三种情况：
        //    1) i=j 单个字母，肯定是回文子串
        //    2) j-i=1 两个相邻的字母相同，是回文子串
        //    3) j-i > 1  要看 dp[i+1][j-1]
        //    情况三决定了遍历顺序，要从下到上，从左到右
        //    即只考虑对角线元素以及对角线以上的元素，且对角线元素肯定都是true
        for (int i = s.size() - 1; i >= 0; i--) {
            for (int j = i; j < s.size(); j++) {
                if (s[i] == s[j]) {
                    if (j - i <= 1) {
                        dp[i][j] = true;
                        cnt++;
                    } else if (dp[i + 1][j - 1]) {
                        dp[i][j] = true;
                        cnt++;
                    }
                }
            }
        }

        return cnt;
    }
};
```

最长回文子序列 #516

```cpp
class Solution {
public:
    int longestPalindromeSubseq(string s) {
        vector<vector<int>> dp(s.size(), vector<int>(s.size(), 0));
        // 1. dp[i][j] 代表 [i,j] 闭区间内的最长的回文子序列长度
        // 2. 子序列是非连续的，所以区间长度越大肯定越长，可以直接定义为长度
        //    字串是连续的，区间长度越大不一定越长，所以要单独计算cnt
        // 3. 根据推导公式，要初始化，对角线元素都是1，对角线以下是0
        for (int i = 0; i < s.size(); i++) dp[i][i] = 1;
        for (int i = s.size() - 1; i >= 0; i--) {
            for (int j = i + 1; j < s.size(); j++) {
                if (s[i] == s[j]) {
                    // // 单个字母和两个相邻的字母的时候可以直接赋值
                    // // 而且这两种情况无法用 dp[i+1][j-1]，这个元素是空的，在对角线以下
                    // if (j - i <= 1) {
                    //     dp[i][j] = j - i + 1;
                    // } else {
                    //     dp[i][j] = dp[i + 1][j - 1] + 2;
                    // }
                    dp[i][j] = dp[i + 1][j - 1] + 2;
                } else {
```

```
                    dp[i][j] = max(dp[i][j - 1], dp[i + 1][j]);
                }
            }
        }

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

☆其他

螺旋矩阵 #59

☆单调栈

☆Error

1. vector.size() 返回的是无符号数，vector.size()-1 会是一个很大的数，用的时候 int n = vector.size()-1 就可以了.

☆C++

STL

```
// vector
vector<int> vec;
vector<int> vec(n, val);
vector<int> vec(set.begin(), set.end()); // unordered_set<int> set;

vec.size();
vec.push_back(elem);
vec.pop_back();
vec.back();
vec.resize(n);
vec.empty();

vec.insert(pos, elem);
vec.erase(pos);
vec.erase(begin, end); // delete [begin, end);

// #include <algorithm>
reverse(vec.begin(), vec.end());
sort(vec.begin(), vec.end());
bool cmp(const int &a, const int &b) {
    return a > b;
}
sort(vec.begin(), vec.end(), cmp);

return {};
return {1, 2, 3};


// set
set<int> set;
```

```
set.find(key);  // set.find(key) != set.end();
set.count();
set.insert(val);


// map
map<int, string> map;

map.find(key);
map.count(key);
map.insert(pair<int, string>(1, "abc"));
map[1] = "abc";

auto iter_1 = map.find(val);
for (auto iter = umap.begin(); iter != umap.end(); iter++)
iter->first;
iter->second;

// stack
stack<int> st;
st.push(x);
st.pop();  // void
st.top();
st.empty();


// queue
queue<int> qu;
que.push(x);
que.pop();
que.front();
que.back();

// priority_queue 优先队列 （堆，默认是大顶堆）
priority_queue<Type, Container, Functional>  // container 容器一般为 vector

struct cmp {
    bool operator()(pair<int, int>& p1, pair<int, int>& p2) {
        return p1.second > p2.second;  // 小顶堆，与 vector 排序顺序相反
    }
};
priority_queue<pair<int, int>, vector<pair<int, int>>, cmp> pri_que;

p_que.push(x);
p_que.pop();
p_que.top(); // 堆顶元素


// deque 双端队列
deque.push_back();
deque.pop_back();
deque.push_front();
deque.pop_front();
deque.front();
deque.back();
```