

LeetCode 2

— SetoWen

7. 双指针

X数之和/差

- 1. 两数之和：**经典简单题，解法很多，常见的是暴力和HashMap，但是也可以用双指针思路。建议面试者用双指针思路做完，自行分析下时间和空间复杂度。
- 167. 两数之和 II - 输入有序数组：**与上一题的区别是这题的输入是已按照升序排列的有序数组。
- 剑指 Offer 57. 和为s的两个数字：**与上一题的区别是，这题的返回值是这个数本身，而非下标。
- 面试题 16.24. 数对和：**与前面题目的区别是要输出数组中两数之和为指定值的所有整数对。
- 653. 两数之和 IV - 输入 BST：**给定一个二叉搜索树和一个目标结果，如果 BST 中存在两个元素且它们的和等于给定的目标结果，则返回 true。
- 15. 三数之和：**给你一个包含 n 个整数的数组 nums，判断 nums 中是否存在三个元素 a, b, c，使得 $a + b + c = 0$ ？请你找出所有和为 0 且不重复的三元组。
- 16. 最接近的三数之和：**给定一个包括 n 个整数的数组 nums 和一个目标值 target。找出 nums 中的三个整数，使得它们的和与 target 最接近。返回这三个数的和。假定每组输入只存在唯一答案。
- 18. 四数之和：**给定一个包含 n 个整数的数组 nums 和一个目标值 target，判断 nums 中是否存在四个元素 a, b, c 和 d，使得 $a + b + c + d$ 的值与 target 相等？找出所有满足条件且不重复的四元组。
- 面试题 16.06. 最小差：**给定两个整数数组 a 和 b，计算具有最小差绝对值的一对数值（每个数组中取一个值），并返回该对数值的差。

1. 给定一个整数数组 nums 和一个整数目标值 target，请你在该数组中找出 **和为目标值 target 的那两个** 整数，并返回它们的数组下标。

你可以假设每种输入只会对应一个答案。但是，数组中同一个元素在答案里不能重复出现。

你可以按任意顺序返回答案。

其实也可以用暴力、哈希表解法

2、3、4、与题1相似

```
vector<int> twoSum(vector<int>& nums, int target) {
    vector<int> nums_sort = nums;
    sort(nums_sort.begin(), nums_sort.end());
    int left = 0, right = nums_sort.size() - 1;
    while(left < right){
        if(nums_sort[left] + nums_sort[right] < target){
            left++;
        }
        else if(nums_sort[left] + nums_sort[right] > target){
            right--;
        }
        else {break;}
    }
    vector<int> ans;
    bool left_flag = false, right_flag = false;
    if(left == right) {return ans;}
    for(int k = 0; k < nums.size(); k++){
        if(nums[k] == nums_sort[left]) {ans.emplace_back(k); left_flag = true;}
        else if(nums[k] == nums_sort[right]) {ans.emplace_back(k); right_flag = true;}
        if(left_flag && right_flag) {return ans;}
    }
    return ans;
}
```

5. 双指针+中序遍历

```
bool findTarget(TreeNode* root, int k) {
    vector<int> tree_vec;
    inorderTraversal(root, tree_vec); →二叉搜索树中序遍历即可得到有序数组
    int left = 0, right = tree_vec.size() - 1;
    while(left < right){
        if(tree_vec[left] + tree_vec[right] < k) {left++;}
        else if(tree_vec[left] + tree_vec[right] > k) {right--;}
        else {return true;}
    }
    return false;
}
```

6.

```

vector<vector<int>> threeSum(vector<int>& nums) {
    vector<vector<int>> ans;
    sort(nums.begin(), nums.end());
    for(int i = 0; i < nums.size(); i++){
        if(nums[i] > 0) {break;}
        if(i > 0 && nums[i] == nums[i-1]) {continue;}
        twoSum(i, nums, ans);
    }
    return ans;
}

void twoSum(int cur, vector<int>& nums, vector<vector<int>> &ans){
    int target = -nums[cur];
    int left = cur+1, right = nums.size()-1;
    while(left < right){
        if(nums[left] + nums[right] < target) {left++;}
        else if(nums[left] + nums[right] > target) {right--;}
        else{
            vector<int> tmp = {nums[cur], nums[left], nums[right]};
            ans.emplace_back(tmp);
            while(left < right && nums[left] == nums[left+1]) {left++;}
            while(left < right && nums[right] == nums[right-1]) {right--;}
            left++; right--;
        }
    }
}

```

三数之和 遍历其左边的数组

两数之和

- 若 $nums[i] > 0$: 因为已经排序好, 所以后面不可能有三个数加和等于 0, 直接返回结果。
- 对于重复元素: 跳过, 避免出现重复解。
- 令左指针 $L = i + 1$, 右指针 $R = n - 1$, 当 $L < R$ 时, 执行循环:
 - 当 $nums[i] + nums[L] + nums[R] == 0$, 执行循环, 判断左界和右界是否和下一位置重复, 去除重复解。并同时将 L, R 移到下一位, 寻找新的解
 - 若和大于 0, 说明 $nums[R]$ 太大, R 左移
 - 若和小于 0, 说明 $nums[L]$ 太小, L 右移

7.

```

int threeSumClosest(vector<int>& nums, int target) {
    int mostclose = nums[0] + nums[1] + nums[2];
    sort(nums.begin(), nums.end());
    for(int i = 0; i < nums.size(); i++){
        if(i > 0 && nums[i] == nums[i-1]) {continue;}
        twoSum(i, nums, mostclose, target);
        if(mostclose == target) {return mostclose;}
    }
    return mostclose;
}

```

```

void twoSum(int cur, vector<int>& nums, int& mostclose, int target){
    int left = cur+1, right = nums.size()-1;
    while(left < right){
        int sum = nums[cur] + nums[left] + nums[right];
        if(abs(target - sum) < abs(target - mostclose)) {mostclose = sum;}
        if(target - sum > 0) {left++;}
        else if(target - sum < 0) {right--;}
        else {mostclose = target; return;}
    }
}

```

8.

```

vector<vector<int>> fourSum(vector<int>& nums, int target) {
    vector<vector<int>> ans;
    sort(nums.begin(), nums.end());
    for(int k = 0; k < nums.size(); k++){
        if(nums[k] >= 0 && nums[k] > target) {break;}
        if(k > 0 && nums[k] == nums[k-1]) {continue;}
        for(int i = k+1; i < nums.size(); i++){
            if(nums[i] + nums[k] >= 0 && nums[i] + nums[k] > target) {break;}
            if(i > k+1 && nums[i] == nums[i-1]) {continue;}
            int left = i+1, right = nums.size()-1;
            while(left < right){
                if((long)nums[k] + nums[i] + nums[left] + nums[right] < target) {left++;}
                else if((long)nums[k] + nums[i] + nums[left] + nums[right] > target) {right--;}
                else{
                    vector<int> tmp = {nums[k], nums[i], nums[left], nums[right]};
                    ans.emplace_back(tmp);
                    while(left < right && nums[left] == nums[left+1]) {left++;}
                    while(left < right && nums[right] == nums[right-1]) {right--;}
                    left++; right--;
                }
            }
        }
    }
    return ans;
}

```

```

9 int smallestDifference(vector<int>& a, vector<int>& b) {
    sort(a.begin(), a.end());
    sort(b.begin(), b.end());
    int i = 0, j = 0;
    long ans = LONG_MAX;
    while(i < a.size() && j < b.size()){
        if(a[i] != b[j]){
            long bias = abs((long)a[i]-(long)b[j]);
            ans = min(ans, bias);
            if(a[i] > b[j]) {j++;}
            else {i++;}
        }
        else{
            return 0;
        }
    }
    return ans;
}

```

柱状图装水

注明：此类题目解法较多，双指针只是其中一种解法。

1. **盛最多水的容器**：给你 n 个非负整数 a_1, a_2, \dots, a_n ，每个数代表坐标中的一个点 (i, a_i) 。在坐标内画 n 条垂直线，垂直线 i 的两个端点分别为 (i, a_i) 和 $(i, 0)$ 。找出其中的两条线，使得它们与 x 轴共同构成的容器可以容纳最多的水。
2. **接雨水和面试题 17.21. 直方图的水量**：给定 n 个非负整数表示每个宽度为 1 的柱子的高度图，计算按此排列的柱子，下雨之后能接多少雨水。这两题虽然题目叙述不太一样，但是代码是一样的。

PS：接雨水虽然是困难题，但是笔试乃至面试时出现的频率不低，建议大家都有掌握其中一种解法（双指针、动态规划或者单调栈），并熟知这种解法的时间和空间复杂度。接雨水还有一个3d接雨水[407. 接雨水 II](#)，此题较难，大家量力而行即可，面试遇到的概率不太大。

```

1 int maxArea(vector<int>& height) {
    int ans = INT_MIN;
    int left = 0, right = height.size()-1;
    while(left < right){
        int area = (right - left) * min(height[left], height[right]);
        ans = max(ans, area);
        if(height[left] < height[right]) {left++;}
        else if(height[left] > height[right]) {right--;}
        else {left++; right--;}
    }
    return ans;
}

```

我们先明确几个变量的意思：

$left_max$ ：左边的最大值，它是从左往右遍历找到的
 $right_max$ ：右边的最大值，它是从右往左遍历找到的
 $left$ ：从左往右处理的当前下标
 $right$ ：从右往左处理的当前下标

定理一：在某个位置上，它能存的水，取决于它左右两边的最大值中较小的一个。

定理二：当我们从左往右处理到 $left$ 下标时，左边的最大值 $left_max$ 对它而言是可信的，但 $right_max$ 对它而言是不可信的。（见下图，由于中间状况未求出 $right$ 下标而言， $right_max$ 必然是它右边最大的值）

定理三：当我们从右往左处理到 $right$ 下标时，右边的最大值 $right_max$ 对它而言是可信的，但 $left_max$ 对它而言是不可信的。



对于位置 $left$ 而言，它左边最大值一定就是 $left_max$ ，右边最大值“等于” $right_max$ 。这时候，如果 $left_max < right_max$ 成立，那么它就知道自己能存多少水了。无论右边将来会不会出现更大的 $right_max$ ，都不影响这个结果。所以当 $left_max < right_max$ 时，我们就不希望去处理 $left$ 下标；反之，我们希望去处理 $right$ 下标。

```

int trap(vector<int>& height) {
    int ans = 0;
    int left = 0, right = height.size()-1;
    int left_max = 0, right_max = 0;
    while(left <= right){
        if(left_max < right_max){
            ans += max(0, left_max - height[left]);
            left_max = max(left_max, height[left]);
            left++;
        }
        else{
            ans += max(0, right_max - height[right]);
            right_max = max(right_max, height[right]);
            right--;
        }
    }
    return ans;
}

```

排序

注明：当题目中指定原地修改、额外常数空间之类的字眼时，可以尝试用双指针思想。当题目输出和索引（下标）有关时，可以尝试用双指针思想。

1. 31. **下一个排列**：实现获取下一个排列的函数，算法需要将给定数字序列重新排列成字典序中下一个更大的排列。必须原地修改，只允许使用额外常数空间。
2. **面试题 16.16. 部分排序**：给定一个整数数组，编写一个函数，找出索引m和n，只要将索引区间[m,n]的元素排好序，整个数组就是有序的。注意：n-m尽量最小，也就是说，找出符合条件的最短序列。函数返回值为[m,n]，若不存在这样的m和n（例如整个数组是有序的），请返回[-1,-1]。
3. 26. **删除排序数组中的重复项**：给定一个排序数组，你需要在原地删除重复出现的元素，使得每个元素只出现一次，返回移除后数组的新长度。不要使用额外的数组空间，你必须在原地修改输入数组并在使用O(1)额外空间的条件下完成。
4. 88. **合并两个有序数组**：给你两个有序整数数组 nums1 和 nums2，请你将 nums2 合并到 nums1 中，使 nums1 成为一个有序数组。

1. 从后向前查找第一个相邻升序的元素对 (i, j) ，满足 $A[i] < A[j]$ 。此时 $[j, end]$ 必然是降序
2. 在 $[j, end]$ 从后向前查找第一个满足 $A[i] < A[k]$ 的 k 。 $A[i]$ 、 $A[k]$ 分别就是上文所说的「小数」、「大数」
3. 将 $A[i]$ 与 $A[k]$ 交换
4. 可以断定这时 $[j, end]$ 必然是降序，逆置 $[j, end]$ ，使其升序
5. 如果在步骤 1 找不到符合的相邻元素对，说明当前 $[begin, end]$ 为一个降序顺序，则直接跳到步骤 4

```
void nextPermutation(vector<int>& nums) {  
    int i = nums.size() - 2, j = nums.size() - 1, k = nums.size() - 1;  
    while(i >= 0 && nums[i] >= nums[j]) {i--; j--;}  
    if(i >= 0) //不是最后一个排列，否则直接整个数组升序排序  
        while(nums[i] >= nums[k]) {k--;}  
        swap(nums[i], nums[k]);  
    }  
    reverse(nums.begin() + j, nums.end());  
    return;  
}
```

2. ①暴力解

```
vector<int> subsort(vector<int>& array) {  
    vector<int> array_sort = array;  
    vector<int> ans = {-1, -1};  
    sort(array_sort.begin(), array_sort.end());  
    for(int i = 0; i < array.size(); i++){  
        if(array[i] != array_sort[i]) {ans[0] = i; break;}  
    }  
    for(int i = array.size() - 1; i >= 0; i--){  
        if(array[i] != array_sort[i]) {ans[1] = i; break;}  
    }  
    return ans;  
}
```

②双指针

```
vector<int> subsort(vector<int>& array) {  
    vector<int> ans = {-1, -1};  
    if(array.size() == 0) {return ans;}  
    int max = array[0], min = array[array.size() - 1];  
    //从左向右遍历，如果当前元素比它之前的最大的元素小，说明不是升序的，更新n为当前元素索引，继续遍历直到末尾  
    //从右向左遍历，如果当前元素比它之后的最小的元素大，说明不是降序的，更新m为当前元素索引，继续遍历直到开始  
    for(int right = 0, left = array.size() - 1; right < array.size(); right++, left--){  
        if(array[right] < max) {ans[1] = right;}  
        else {max = array[right];}  
        if(array[left] > min) {ans[0] = left;}  
        else {min = array[left];}  
    }  
    return ans;  
}
```

3. 原地最多保留1位



原地最多保留k位

① 双指针

```
int removeDuplicates(vector<int>& nums) {
    if(nums.size() <= 1) {return nums.size();}
    int slow, fast;
    for(slow = 0, fast = 1; fast < nums.size(); fast++){
        if(nums[slow] != nums[fast]) {
            slow++;
            nums[slow] = nums[fast];
        }
    }
    return slow+1;
}
```

② 通用方法

```
int removeDuplicates(vector<int>& nums) {
    return duplicates_process(nums, 1);
}

int duplicates_process(vector<int>& nums, int k){
    int left = 0;
    for(auto& c:nums){
        if(left < k || nums[left-k] != c){
            nums[left] = c;
            left++;
        }
    }
    return left;
}
```

4

```
void merge(vector<int>& nums1, int m, vector<int>& nums2, int n) {
    int i = nums1.size()-1;
    m--; n--;
    while(n >= 0){
        if(m >= 0 && nums1[m] >= nums2[n]) {swap(nums1[i], nums1[m]); m--;}
        else {swap(nums1[i], nums2[n]); n--;}
        i--;
    }
    return;
}
```

找值

1. 162. 寻找峰值：峰值元素是指其值大于左右相邻值的元素。给定一个输入数组 `nums`，其中 `nums[i] ≠ nums[i+1]`，找到峰值元素并返回其索引。

2. 剑指 Offer 59 - I. 滑动窗口的最大值：给定一个数组 `nums` 和滑动窗口的大小 `k`，请找出所有滑动窗口里的最大值。

1. 二分查找改

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

也许很多人答了，还是不懂。图解也只是把这题通俗地讲清楚而已，这里通俗地讲一下。这道题，最最重要的是条件、条件、条件，两边都是无穷，数组当中可能有很多波浪，也可能只有一个。如果尝试画图，就是跟股票是一样，没有规律。如果根据中点判断我们的二分方向该往何处取，这题目这只有是走最后一个函数。你这样想，中点所在地方，可能是家庭的山峰，山的下坡处，山的上坡处。如果是山峰，最后会二分终止，它也会识别。关键是我们二分的方向，并不知道山峰在我们左边还是右边。这几个字你都明白了，爬山（没错，就是带你去爬山），如果你往下爬方能走，也不可能走到新的山峰，但你也许是一个一直在爬坡，最后到达顶峰，但是你最后往上爬的方向走，就是最后一直上的山顶。由于最边界是无穷，所以就一定能找到山峰，总的一句话，往递增的方向上，二分，一定能找到。往递减的方向只是可能找到，也许没有。

2. ① 单调队列

假设我们当前处理到某个长度为 k 的窗口，此时窗口往右滑动一格，会导致后一个数（新窗口的右端点）添加进来，同时会导致前一个数（旧窗口的左端点）移出窗口。

随着窗口的不断平移，该过程会一直发生。若同一时刻存在两个数 $nums[j]$ 和 $nums[i]$ ($j < i$) 在同一个窗口内，下标更大的数会被更早移出窗口，此时如果有 $nums[j] <= nums[i]$ 的话，可以完全确定 $nums[j]$ 将不会成为后续任何一个窗口的最大值，此时可以将必然不会是答案的 $nums[j]$ 从候选集中进行移除。

不难发现，当我们所有必然不可能成为答案的元素（即所有满足的小于 $nums[i]$ ）移除后，候选集合满足「单调递减」特性，即集合元素为当前窗口中的最大值（为了满足窗口长度为 k 的要求，在从集合头部取答案时需要先将下标小于的等于的 $i - k$ 的元素移除）。

为方便从尾部添加元素，从头部获取答案，我们可以使用「双端队列」存储所有候选元素。

```
vector<int> maxSlidingWindow(vector<int>& nums, int k) {
    vector<int> ans;
    if(nums.size() < k) {return ans;}
    deque<int> deq; // 单调队列
    // 将第一个滑动窗口中的单调队列对应的坐标构建出来(单端队列一定是保持递增或递减的)
    // q中存的是对应窗口的nums坐标
    // 若新元素比 q 结尾坐标处的元素值大，从q尾开始移除坐标 (始终保持最大值在队头)，否则将新元素的坐标进入队列
    for(int i = 0; i < k; i++){
        while(!deq.empty() && nums[deq.back()] > nums[deq.front()]) {deq.pop_back();}
        deq.push_back(i);
    }
    ans.emplace_back(nums[deq.front()]);
    for(int i = k; i < nums.size(); i++){
        while(!deq.empty() && nums[i] > nums[deq.back()]) {deq.pop_back();} // 若q头坐标在滑动窗口左边界外侧
        deq.push_back(i);
        while(deq.front() <= i - k) {deq.pop_front();}
        ans.emplace_back(nums[deq.front()]); // q的头坐标即滑动窗口的最大值压入deq
    }
    return ans;
}
```

② 优先队列 (大根堆)

```
vector<int> maxSlidingWindow(vector<int>& nums, int k) {
    vector<int> ans;
    priority_queue<pair<int, int>, vector<pair<int, int>>, less<pair<int, int>> q; // 优先队列 (大根堆)
    for(int i = 0; i < k; i++){
        q.emplace(nums[i], i);
    }
    ans.emplace_back(q.top().first);
    for(int i = k; i < nums.size(); i++){
        q.emplace(nums[i], i);
        while(q.top().second <= i - k) {q.pop();}
        ans.emplace_back(q.top().first);
    }
    return ans;
}
```

调整位置

1. 283. 移动零：给定一个数组 $nums$ ，编写一个函数将所有 0 移动到数组的末尾，同时保持非零元素的相对顺序。

2. 剑指 Offer 21. 调整数组顺序使奇数位于偶数前面：输入一个整数数组，实现一个函数来调整该数组中数字的顺序，使得所有奇数位于数组的前半部分，所有偶数位于数组的后半部分。

1.

```
void moveZeroes(vector<int>& nums) {
    int left = 0, right = 0;
    for(; right < nums.size(); right++){
        if(nums[right] == 0) {zero_cnt++;}
        else{
            swap(nums[left], nums[right]);
            left++;
        }
    }
    return;
}
```

2.

```
vector<int> exchange(vector<int>& nums) {
    int left = 0, right = 0;
    for(; right < nums.size(); right++){
        if(nums[right] % 2 == 1){
            swap(nums[left], nums[right]);
            left++;
        }
    }
    return nums;
}
```

8. 哈希表

重复元素

1. 217. 存在重复元素：给定一个整数数组，判断是否存在重复元素。如果存在一值在数组中出现至少两次，函数返回true。如果数组中每个元素都不相同，则返回false。
2. 219. 存在重复元素 II：给定一个整数数组和一个整数 k，判断数组中是否存在两个不同的索引 i 和 j，使得 $nums[i] = nums[j]$ ，并且 $|i-j|$ 的差的绝对值至多为 k。
3. 220. 存在重复元素 III：在整数数组 nums 中，是否存在两个下标 i 和 j，使得 $nums[i] = nums[j]$ 的差的绝对值小于等于 t，且满足 $|i-j|$ 的差的绝对值也小于等于 k。如果存在则返回true，不存在则返回false。
4. 287. 寻找重复数：给定一个包含 n+1 个整数的数组 nums，其数字都在 1 到 n 之间（包括 1 和 n），可知至少存在一个重复的整数。假设 nums 只有一个重复的整数，找出这个重复的数。
5. 剑指 Offer 03. 数组中重复的数字：在一个长度为 n 的数组 nums 里的所有数字都在 0 ~ n-1 的范围内。数组中某些数字是重复的，但不知道有几个数字重复了，也不知道每个数字重复了几次。请找出数组中任意一个重复的数字。

1.

```
bool containsDuplicate(vector<int>& nums) {
    unordered_map<int, int> nums_hash;
    for(auto& n:nums){
        if(nums_hash.count(n)) {return true;}
        nums_hash[n]++;
    }
    return false;
}
```

2.

```
bool containsNearbyDuplicate(vector<int>& nums, int k) {
    unordered_map<int, int> nums_hash;
    for(int i = 0; i<nums.size(); i++){
        if(!nums_hash.count(nums[i])) {nums_hash[nums[i]] = i;}
        else{
            if(abs(i - nums_hash[nums[i]]) <= k) {return true;}
            else {nums_hash[nums[i]] = i;}
        }
    }
    return false;
}
```

3. 集合 set + 滑动窗口 + 二分查找

```
bool containsNearbyAlmostDuplicate(vector<int>& nums, int indexDiff, int valueDiff) {
    set<int> num_set;
    for(int i = 0; i<nums.size(); i++){
        auto lb = num_set.lower_bound(nums[i]-valueDiff); //二分查找大于等于此值的下界
        //查找到目标值，且确保在范围(nums[i]-valueDiff, nums[i]+valueDiff)内
        if(lb != num_set.end() && *lb <= nums[i]+valueDiff) {return true;}
        if(i >= indexDiff) {num_set.erase(nums[i-indexDiff]);}
        num_set.insert(nums[i]);
    }
    return false;
}
```

4.

①快慢指针(类似于142题的环形链表)

```
int findDuplicate(vector<int>& nums) {
    int slow = 0, fast = 0;
    slow = nums[slow]; fast = nums[nums[fast]];
    while(slow != fast){
        slow = nums[slow];
        fast = nums[nums[fast]];
    }
    int ans = 0;
    while(ans != slow){
        ans = nums[ans];
        slow = nums[slow];
    }
    return slow;
}
```

②哈希表

```
int findDuplicate(vector<int>& nums) {
    vector<bool> cnt(nums.size()-1, false);
    for(int i = 0; i<nums.size(); i++){
        if(cnt[nums[i]] == true) {return nums[i];}
        cnt[nums[i]] = true;
    }
    return -1;
}
```

5. 题十方法②

数组交集

注明：这两题除了哈希方法以外，都可以用双指针的方式。第二题的思考题可以思考下。

349. 两个数组的交集：输出结果中的每个元素一定是唯一的。
350. 两个数组的交集 II：输出结果中每个元素出现的次数，应与元素在两个数组中出现次数的最小值一致。思考题答案见解

1.

```
vector<int> intersection(vector<int>& nums1, vector<int>& nums2) {
    vector<int> ans;
    unordered_set<int> hashset;
    for(auto& n:nums1){
        hashset.emplace(n);
    }
    for(auto& n:nums2){
        if(hashset.count(n)) {
            ans.emplace_back(n);
            hashset.erase(n);
        }
    }
    return ans;
}
```

2.

```
vector<int> intersect(vector<int>& nums1, vector<int>& nums2) {
    unordered_map<int, int> hashmap;
    vector<int> ans;
    for(auto& n:nums1){
        hashmap[n]++;
    }
    for(auto& n:nums2){
        if(hashmap[n] > 0){
            ans.emplace_back(n);
            hashmap[n]--;
        }
    }
    return ans;
}
```

进阶的3个问题⇒

<https://leetcode.cn/problems/intersection-of-two-arrays-ii/solution/jin-jie-san-wen-by-user5707f/>

数字游戏

- 面试题 16.15. 珠玑妙算：给定一种颜色组合solution和一个猜测guess，编写一个方法，返回猜中和伪猜中的次数answer，其中answer[0]为猜中的次数，answer[1]为伪猜中的次数。
299. 猜数字游戏：请写出一个根据秘密数字和朋友的猜测数返回提示的函数，返回字符串的格式为xAyB，x和y都是数字，A表示公牛，用B表示奶牛。
36. 有效的数独：判断一个9×9的数独是否有效。只需要根据以下规则，验证已经填入的数字是否有效即可。
457. 分糖果：给定一个偶数长度的数组，其中不同的数字代表着不同种类的糖果，每一个数字代表一个糖果。你需要把这些糖果平均分给一个弟弟和一个妹妹。返回妹妹可以获得的最大糖果的种类数。
- 剑指 Offer 61. 扑克牌中的顺子：从扑克牌中随机抽5张牌，判断是不是一个顺子，即这5张牌是不是连续的。2~10为数字本身，A为1，J为11，Q为12，K为13，而大、小王为0，可以看成任意数字。A不能视为14。

1.

```
vector<int> masterMind(string solution, string guess) {
    vector<int> ans = {0,0};
    unordered_map<char, int> hash_m;
    int real_guess = 0, fake_guess = 0;
    for(auto& c:solution){
        hash_m[c]++;
    }
    for(int i = 0; i<4; i++){
        if(hash_m[guess[i]]){
            hash_m[guess[i]]--;
            fake_guess++;
        }
        if(solution[i] == guess[i]) {real_guess++;}
    }
    ans[0] = real_guess;
    ans[1] = fake_guess - real_guess;
    return ans;
}
```

2.

```
string getHint(string secret, string guess) {
    string ans = "";
    int real_guess = 0, fake_guess = 0;
    unordered_map<char, int> hash_m;
    for(auto& c:secret){
        hash_m[c]++;
    }
    for(int i = 0; i<guess.size(); i++){
        if(secret[i] == guess[i]) {real_guess++;}
        if(hash_m[guess[i]]){
            hash_m[guess[i]]--;
            fake_guess++;
        }
    }
    ans += to_string(real_guess);
    ans += 'A';
    ans += to_string(fake_guess-real_guess);
    ans += 'B';
    return ans;
}
```



```

bool isValidSudoku(vector<vector<char>>& board) {
    vector<vector<int>> row_hash(9, vector<int>(9, 0));
    vector<vector<int>> col_hash(9, vector<int>(9, 0));
    vector<vector<vector<int>>> box_hash(3, vector<vector<int>>(3, vector<int>(9, 0)));
    for(int i = 0; i < 9; i++){
        for(int j = 0; j < 9; j++){
            if(board[i][j] != '.'){
                int index = board[i][j] - '0' - 1;
                row_hash[i][index]++;
                col_hash[j][index]++;
                box_hash[i/3][j/3][index]++;
                if(row_hash[i][index] > 1 || col_hash[j][index] > 1 || box_hash[i/3][j/3][index] > 1){
                    return false;
                }
            }
        }
    }
    return true;
}

```

4.

```

int distributeCandies(vector<int>& candyType) {
    unordered_set<int> hash;
    for(auto& c:candyType) {hash.emplace(c);}
    return min(hash.size(), candyType.size()/2);
}

```

5、①哈希表

```

bool isStraight(vector<int>& nums) {
    vector<int> hash_num(14, 0);
    for(auto& c:nums) {hash_num[c]++;}
    int cnt = 0;
    for(int i = 1; i<14; i++){
        if(cnt == 5) {return true;}
        if(hash_num[i] > 0) {cnt++;}
        else{
            if(hash_num[0] > 0 && cnt > 0) {cnt++; hash_num[0]--;
            } else {cnt = 0;}
        }
    }
    return (cnt + hash_num[0]) == 5;
}

```

缓存机制

注明：第一题是常见面试题，通常不会让你写代码（或者简写一下），但是要你说出设计的思路，所以如何将思路清晰的说出来是面试前必备的重要能力。

1. 146. LRU 缓存机制和面试题 16.25. LRU 缓存：运用你所掌握的数据结构，设计和实现一个

LRU（最近最少使用）缓存机制。学java的同学可以试试基于LinkedHashMap的代码。

2. 460. LFU 缓存：请你为最不经常使用（LFU）缓存算法设计并实现数据结构。

6、

请你设计并实现一个满足 LRU（最近最少使用）缓存 约束的数据结构。

实现 LRUcache 类：

- LRUcache(int capacity) 以 正整数 作为容量 capacity 初始化 LRU 缓存
- int get(int key) 如果关键字 key 存在于缓存中，则返回关键字的值，否则返回 -1 。
- void put(int key, int value) 如果关键字 key 已经存在，则变更其数据值 value；如果不存在，则向缓存中插入该组 key-value 。如果插入操作导致关键字数量超过 capacity ，则应该 逐出 最久未使用的关键字。

函数 get 和 put 必须以 O(1) 的平均时间复杂度运行。

②链+遍历

```

bool isStraight(vector<int>& nums) {
    unordered_set<int> hash;
    int max_num = 0, min_num = 14;
    for(auto& c:nums){
        if(c == 0) {continue;}//跳过大小王
        max_num = max(max_num, c);
        min_num = min(min_num, c);
        if(hash.count(c)) {return false;}//当牌里一旦有重复的，则必不可能为顺子
        hash.emplace(c);
    }
    return (max_num - min_num) < 5;// 最大牌 - 最小牌 < 5 则可构成顺子
}

```

```

class LRUCache {
public:
    LRUCache(int capacity) {
        cap = capacity;
    }

    int get(int key) {
        auto it = hash_link.find(key);
        if(it != hash_link.end()){
            link.splice(link.begin(), link, it->second);
            return it->second->second;
        }
        return -1;
    }

    void put(int key, int value) {
        auto it = hash_link.find(key);
        if(it != hash_link.end()){
            link.splice(link.begin(), link, it->second);
            link.splice(link.begin(), link, it->second); //本来已存在此链
            it->second->second = value;
            it->second->first = value;
            return;
        }

        link.erase(link.begin()); //移除链表头部
        hash_link[key] = link.begin();
        if(link.size() > cap){
            hash_link.erase(link.begin().first);
            link.pop_back();
        }
        return;
    }

private:
    int cap;
    unordered_map<int, list<pair<int, int>>::iterator> hash_link; //哈希表各键在链表的位置
    list<pair<int, int>> link; //链表的键及其值，链表尾是久的键
};

```

2

请你为最不经常使用 (LFU) 缓存算法设计并实现数据结构。

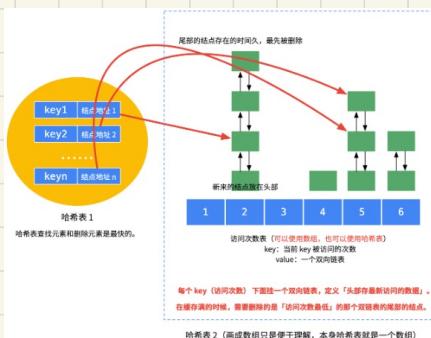
实现 LFUCache 类:

- `LFUCache(int capacity)` - 用数据结构的容量 `capacity` 初始化对象
- `int get(int key)` - 如果键 `key` 存在于缓存中，则获取键的值，否则返回 `-1`。
- `void put(int key, int value)` - 如果键 `key` 已存在，则变更其值；如果键不存在，请插入键值对。当缓存达到其容量 `capacity` 时，则应该在插入新项之前，移除最近最不经常使用的项。在此问题中，当存在平局（即两个或更多个键具有相同使用频率）时，应该去除最近最久未使用的键。

为了确定最不常使用的键，可以为缓存中的每个键维护一个 使用计数器。使用计数最小的键是最久未使用的键。

当一个键首次插入到缓存中时，它的使用计数器被设置为 1 (由于 `put` 操作)。对缓存中的键执行 `get` 或 `put` 操作，使用计数器的值将递增。

函数 `get` 和 `put` 必须以 $O(1)$ 的平均时间复杂度运行。



难，以后再看

LRU (Least Recently Used) 缓存机制 (看时间)

「力扣」第 146 题：[LRU缓存机制](#)

- 在缓存满的时候，删除缓存里最久未使用的数据，然后再放入新元素；
- 数据的访问时间很重要，访问时间距离现在越近，就越不容易被删除；

就是喜新厌旧，淘汰在缓存里呆的时间最久的元素。在删除元素的时候，只看「时间」这一个维度。

LFU (Least Frequently Used) 缓存机制 (看访问次数)

- 在缓存满的时候，删除缓存里使用次数最少的元素，然后在缓存中放入新元素；
- 数据的访问次数很重要，访问次数越多，就越不容易被删除；
- 根据题意，「当存在平局（即两个或更多个键具有相同使用频率）时，最近最少使用的键将被去除」，即在「访问次数」相同的情况下，按照时间顺序，先删除在缓存里时间最久的数据。

说明：本题其实就是在「力扣」第 146 题：[LRU缓存机制](#) 的基础上，在删除策略里多考虑了一个维度（「访问次数」）的信息。

核心思想：先考虑访问次数，在访问次数相同的情况下，再考虑缓存的时间。

9. 位运算

异或

1. 面试题 16.01. 交换数字：编写一个函数，不用临时变量，直接交换numbers = [a, b]中a与b的值。
提升：这题除了使用异或以外，还可以使用加减法。比如：

方法1 (+--)

```
numbers[0] = numbers[0] + numbers[1];
numbers[1] = numbers[0] - numbers[1];
numbers[0] = numbers[0] - numbers[1];
```

方法2 (-+)

```
numbers[0] = numbers[0] - numbers[1];
numbers[1] = numbers[0] + numbers[1];
numbers[0] = numbers[1] - numbers[0];
```

2. 268. 丢失的数字、剑指 Offer 53 - II. 0 ~ n-1中缺失的数字和面试题 17.04. 消失的数字：给定一个包含 [0, n] 中 n 个数的数组 nums，找出 [0, n] 这个范围内没有出现在数组中的那个数。三题可以用同一个代码。本题除了位运算解法，还可以用置換法、二分法和数学法，有兴趣的可以去题解区找，有我笔记的可以直接看思路（含有原题解链接，有疑问可以直接与题主沟通）。如果面试时候出了这题，记得与面试官沟通要求，可能会被要求用某种固定的时间或者空间复杂度。

3. 136. 只出现一次的数字：给定一个非空整数数组，除了某个元素只出现一次以外，其余每个元素均出现两次。找出那个只出现了一次的元素。
4. 137. 只出现一次的数字 II 和剑指 Offer 56 - II. 数组中数字出现的次数 II：给定一个非空整数数组，除了某个元素只出现一次以外，其余每个元素均出现了三次。找出那个只出现了一次的元素。

异或三大定律 |
 交换律 $\rightarrow a \wedge b = b \wedge a$
结合律 $\rightarrow (a \wedge b) \wedge c = a \wedge (b \wedge c)$
自反性 $\rightarrow x \wedge x = 0, x \wedge 0 = x, x \wedge 1 = x$
 $\Rightarrow A \wedge B \wedge B = A \wedge (B \wedge B) = A \wedge 0 = A$

- 异或交换两个值，不可以用临时变量

思路：利用异或的交换律

代码

```
a = a ^ b;
b = a ^ b;
a = a ^ b;
```

- 异或找出重复的元素：1~1000放在含有1001个元素的数组中，只有唯一的一个元素值重复，其它均只出现一次

思路

$1^2 \wedge \dots \wedge n \wedge \dots \wedge 1000$ ，无论这两个n出现在什么位置，都可以转换成为 $1^2 \wedge \dots \wedge 1000 \wedge (n \wedge n)$ 的形式。

其次，对于任何数x，都有 $x \wedge x = 0$ ， $x \wedge 0 = x$ 。

$1^2 \wedge \dots \wedge n \wedge \dots \wedge 1000 = 1^2 \wedge \dots \wedge 1000 \wedge (n \wedge n) = 1^2 \wedge \dots \wedge 1000 \wedge 0 = 1^2 \wedge \dots \wedge 1000$ （即序列中除了n的所有数的异或）。

令， $1^2 \wedge \dots \wedge 1000$ （序列中不包含n）的结果为T

则 $1^2 \wedge \dots \wedge 1000$ （序列中包含n）的结果就是 $T \wedge n$ 。

$T \wedge (T \wedge n) = n$ 。

所以，将 $1^2 \wedge \dots \wedge 1000$ 的结果和数组中所有的数异或，得到的结果就是重复数。

代码

```
int result = 0;
for(int i = 1; i < length; i++) {  $\Rightarrow T$ 
    result ^= i;
}
for(int j = 0; j < length; j++) {  $\Rightarrow T \wedge T \wedge n$ 
    result ^= arr[j];
}
```

5. 260. 只出现一次的数字 III 和剑指 Offer 56 - I. 数组中数字出现的次数：给定一个整数数组 nums，其中恰好有两个元素只出现一次，其余所有元素均出现两次。找出只出现一次的那两个元素。
6. 面试题 17.19. 消失的两个数字：给定一个数组，包含从 1 到 N 所有的整数，但其中缺了两个数字。在 O(N) 时间内只用 O(1) 的空间找到它们。
7. 645. 错误的集合：集合 s 包含从 1 到 n 的整数。不幸的是，因为数据错误，导致集合里面某一个数字复制成了集合里面的另外一个数字的值，导致集合丢失了一个数字并且还有一个数字重复。给定一个数组 nums 代表了集合 S 发生错误后的结果。请你找出重复出现的整数，再找到丢失的整数，将它们以数组的形式返回。
8. 89. 格雷编码：格雷编码是一个二进制数字系统，在该系统中，两个连续的数值仅有 1 位数的差异。给定一个代表编码总位数的非负整数 n，打印其格雷编码序列。即便有多个不同答案，你也只需要返回其中一种。
9. 371. 两整数之和、剑指 Offer 65. 不用加减乘除做加法和面试题 17.01. 不用加号的加法：写一个函数，求两个整数之和，要求在函数体内不得使用“+”、“-”、“*”、“/”四则运算符号。

```

vector<int> swapNumbers(vector<int>& numbers) {
    numbers[0] = numbers[0]^numbers[1];
    numbers[1] = numbers[0]^numbers[1];
    numbers[0] = numbers[0]^numbers[1];
    return numbers;
}

```

2.① 异或

```

int missingNumber(vector<int>& nums) {
    int xor1 = 0;
    for(int i = 0; i<nums.size(); i++){
        xor1 ^= nums[i];
    }
    for(int i = 0; i<=nums.size(); i++){
        xor1 ^= i;
    }
    return xor1;
}

```

3.

```

int singleNumber(vector<int>& nums) {
    int ans = 0;
    for(auto& n:nums){
        ans ^= n;
    }
    return ans;
}

```

② 哈希表

```

int missingNumber(vector<int>& nums) {
    vector<bool> hash_num(nums.size()+1, false);
    for(int i = 0; i<nums.size(); i++){
        hash_num[nums[i]] = true;
    }
    for(int i = 0; i<hash_num.size(); i++){
        if(hash_num[i] == false) {return i;}
    }
    return -1;
}

```

4、排序+判断

```

int singleNumber(vector<int>& nums) {
    sort(nums.begin(), nums.end());
    for(int i = 0; i<nums.size()-2; i+=3){
        if(nums[i] != nums[i+2]) {return nums[i];}
    }
    return nums.back();
}

```

5. <https://leetcode.cn/problems/single-number-iii/solutions/1967822/by-zhong-xia-w-ueqg/>

```

vector<int> singleNumber(vector<int>& nums) {
    int xor1 = 0;
    for(auto& n:nums){
        xor1 ^= n; //算出只出现一次的两个元素的异或
    }
    int index = 1;
    while((xor1 & index) == 0){ //从低位到高位检测两个元素第一个不相等的位
        index <= 1;
    }
    vector<int> ans = {0,0};
    for(auto& n:nums){ //将两个元素分开
        if(n & index){ //若在index对应位是1，则归为元素1
            ans[0] ^= n;
        }
        else{ //若index对应位是0，则归为元素2
            ans[1] ^= n;
        }
    }
    return ans;
}

```

6. 与题解相似

```
vector<int> missingTwo(vector<int>& nums) {
    int xor1 = 0;
    int cnt = 1;
    while(cnt <= nums.size() + 2) {
        xor1 ^= cnt;
        cnt++;
    }
    for(auto& n:nums){
        xor1 ^= n;
    }
    int index = 1;
    while((xor1 & index) == 0) {index <= 1;};
    vector<int> ans = {0,0};
    cnt = 1;
    while(cnt <= nums.size() + 2) {
        if(index & cnt) {ans[0] ^= cnt;};
        else {ans[1] ^= cnt;};
        cnt++;
    }
    for(auto& n:nums){
        if(index & n) {ans[0] ^= n;};
        else {ans[1] ^= n;};
    }
    return ans;
}
```

不考虑

7. 1 2 2 4 $\Rightarrow 0^1 2^2 2^2 3^0$
 1 2 3 4 $\Rightarrow 0^0 1^2 3^0$
 $\Rightarrow 2^3$

```
vector<int> findErrorNums(vector<int>& nums) {
    int xor1 = 0;
    int cnt = 1;
    while(cnt <= nums.size()) {
        xor1 ^= cnt;
        cnt++;
    }
    for(auto& n:nums){
        xor1 ^= n;
    }
    vector<int> ans = {0,0};
    int index = 1;
    while((xor1 & index) == 0) {index <= 1;};
    cnt = 1;
    while(cnt <= nums.size()){
        if(index & cnt) {ans[0] ^= cnt;};
        else {ans[1] ^= cnt;};
        cnt++;
    }
    for(auto& n:nums){
        if(index & n) {ans[0] ^= n;};
        else {ans[1] ^= n;};
    }
    //返回要求重复的数字在前，缺失的数字在后
    for(auto& n:nums){
        if(n == ans[1]){
            return {ans[1], ans[0]};
        }
    }
    return ans;
}
```

8.

n位格雷码序列 是一个由 2^n 个整数组成的序列，其中：

- 每个整数都在范围 $[0, 2^n - 1]$ 内 (含 0 和 $2^n - 1$)
- 第一个整数是 0
- 一个整数在序列中出现 不超过一次
- 每对 相邻 整数的二进制表示 恰好一位不同
- 第一个 和 最后一个 整数的二进制表示 恰好一位不同

给你一个整数 n ，返回任一有效的 n 位格雷码序列。

格雷编码公式：第 i 位格雷编码 = $i \wedge (i \gg 1)$

0 0 0 0 0	←	0 0 0 0 0
0 0 0 0 1	←	0 0 0 0 1
0 0 0 1 1	←	0 0 0 1 0
0 0 0 1 0	←	0 0 0 1 1
0 0 1 1 0	←	0 0 1 0 0

```
vector<int> grayCode(int n) {
    vector<int> ans(pow(2,n));
    for(int i = 0; i < ans.size(); i++){
        ans[i] = i ^ (i>>1);
    }
    return ans;
}
```

9.

<https://leetcode.cn/problems/sum-of-two-integers/solutions/1017617/liang-zheng-shu-zhi-he-by-leetcode-solut/>

```
int getSum(int a, int b) {
    while(b != 0){
        unsigned int carry = (unsigned int)(a & b) << 1;
        a = a ^ b;
        b = carry;
    }
    return a;
}
```

移位

- 面试题 05.07. 配对交换：**编写程序，交换某个整数的奇数位和偶数位，尽量使用较少的指令（也就是说，位0与位1交换，位2与位3交换，以此类推）。
- 面试题 05.01. 插入：**给定两个整型数字 N 与 M，以及表示比特位置的 i 与 j（ $i \leq j$ ，且从 0 位开始计算）。编写一种方法，使 M 对应的二进制数字插入 N 对应的二进制数字的第 $i \sim j$ 位区域，不足之处用 0 补齐。具体插入过程如图所示。
- 190. 颠倒二进制：**颠倒给定的 32 位无符号整数的二进制位。
- 面试题 05.03. 翻转数位：**给定一个32位整数 num，你可以将一个数位从0变为1。请编写一个程序，找出你能够获得的最长的一串1的长度。
- 面试题 05.04. 下一个数：**下一个数。给定一个正整数，找出与其二进制表达式中1的个数相同且大小最接近的那两个数（一个略大，一个略小）。

1.

```
int exchangeBits(int num) {
    int odd = num & 0b01010101010101010101010101010101;
    int even = num & 0b10101010101010101010101010101010;
    odd = odd << 1;
    even = even >> 1;
    return odd | even;
}
```

<https://leetcode.cn/problems/exchange-lcci/solutions/98026/weiyun-suan-jie-jue-by-teleg-xing/>

2.

```
int insertBits(int N, int M, int i, int j) {
    //把N的i到j位置为0
    for(int k = i; k<=j; k++){
        if(N & (1<<k)){
            N ^= (1<<k);
        }
    }
    //把M的数值左移i位
    M <<= i;
    return N | M;
}
```

3.

```
uint32_t reverseBits(uint32_t n) {
    uint32_t ans = 0;
    for(int i = 0; i<32; i++){
        ans |= (1 & n);
        n >>= 1;
        if(i != 31) {ans <<= 1;}
    }
    return ans;
}
```

4. 双指针 01 和 10

```
int reverseBits(int num) {
    if(~num == 0) {return 32;} // -1有32位连续，但按照下面的算法会算出33位
    int max_cnt = 0, cur_cnt = 0, pre_cnt = 0;
    for(int i = 0; i<32; i++){
        if(num & 1){
            cur_cnt++;
        } else{
            if(num & 2){
                pre_cnt = cur_cnt;
            } else{
                pre_cnt = 0;
            }
            cur_cnt = 0;
        }
        max_cnt = max(max_cnt, pre_cnt+cur_cnt+1);
        num >>= 1;
    }
    return max_cnt;
}
```

5.

找到稍微大的数：找到第一个01变成10

- 最简单情况...00000111从低位到高位，找到第一个01变成10 0000 0111->0000 1011即可
- 复杂情况...0011 1000 找到第一个01变成10以后，把低位的1右移 0011 1000->01011000->0100 0011第三个数符合题意

找到稍微小的数：找到第一个10变成01

- 最简单情况...1110 0000 从低位到高位，找到第一个10变成01 1110 0000->1101 0000即可
- 复杂情况...1110 0011 找到第一个10变成01以后，把低位的1左移 1110 0011->1110 0011->1101 0011->1101 1100 第三个数符合题意

```
int getBig(int num){
    int cnt = 0;
    //去掉低位开始的0
    if((num & (1<<cnt)) == 0){
        while(cnt < 31 && (num & (1<<cnt)) == 0){
            cnt++;
        }
        if(cnt == 31) {return -1;}
        num += (1<<cnt); //01111000->01111000
        cnt--;
        num -= (1<<cnt); //01111000->01111000
        cnt--;/cnt指向要右移的第一个1
        int count = 0;
        //c1大于0才需要右移
        while(c1 > 0 && cnt > 0 && (num & (1<<cnt))>0){
            num += (1<<cnt);
            count++; //count代表几个1
            cnt--;
        }
        //把右边清0
        while(count > 0){
            count--;
            num += (1<<cnt);
        }
        return num;
    }
}

int getSmall(int num){
    int cnt = 0;
    //去掉开始的1
    if((num & (1<<cnt)) > 0){
        while(cnt < 31 && (num & (1<<cnt)) > 0){
            cnt++;
        }
        if(cnt == 31) {return -1;}
        num -= (1<<cnt); //1000111->0000111
        cnt--;
        num += (1<<cnt); //0000111->0000111
        //cnt指向从低位到高位第一个0的位置
        int c0 = cnt;
        while(c0 < 31 && (num & (1<<c0)) == 0){
            c0++;
        }
        if(c0 == 31) {return -1;}
        num -= (1<<c0); //0000111->0000111
        cnt--;
        num += (1<<cnt); //0000111->0000111
        //把右边清0
        while(ch0 > 0){
            ch0--;
            num -= (1<<ch0);
        }
        return num;
    }
}
```

```
vector<int> findClosedNumbers(int num) {
    if(num == INT_MAX) {return {-1,-1};}
    vector<int> ans(2,-1);
    ans[0] = getBig(num);
    ans[1] = getSmall(num);
    return ans;
}
```

布赖恩·克尼根 (Brian Kernighan) 算法

推荐阅读链接: 汉明距离官解

1. 461. 汉明距离: 两个整数之间的汉明距离指的是这两个数字对应二进制位不同的位置的数目。

给出两个整数 x 和 y , 计算它们之间的汉明距离。

2. 面试题 05.06. 整数转换: 整数转换。编写一个函数, 确定需要改变几个位才能将整数A转成整数B。

3. 201. 数字范围按位与: 给定范围 $[m, n]$, 其中 $0 \leq m \leq n \leq 2147483647$, 返回此范围内所有数字的按位与 (包含 m, n 两端点)。

1. ①普通位运算

```
int hammingDistance(int x, int y) {
    int cnt = 0;
    for(int i = 0; i<32; i++){
        if((1 & x) != (1 & y)) {cnt++;}
        x >>= 1;
        y >>= 1;
    }
    return cnt;
}
```

② 布赖恩·克尼根算法

在方法 1 中, 对于 $s = (10001100)_2$ 的情况, 我们需要循环右移 8 次才能得到答案, 而实际上如果我们可以跳过两个 1 之间的 0, 直接对 1 进行计数, 那么只需要循环 3 次即可。

我们可以使用 Brian Kernighan 算法进行优化, 具体地, 该算法可以被描述为这样一个结论: 记 $f(x)$ 表示 x 和 $x - 1$ 进行与运算所得的结果 (即 $f(x) = x \& (x - 1)$) , 那么 $f(x)$ 恰为 x 剔去其二进制表示中最右侧的 1 的结果。

Brian Kernighan

$x =$	1	0	0	0	1	0	0	0
$x - 1 =$	1	0	0	0	0	1	1	1
$x \& (x - 1) =$	1	0	0	0	0	0	0	0

基于该算法, 当我们计算出 $s = x \oplus y$, 只需要不断让 $s = f(s)$, 直到 $s = 0$ 即可。这样每循环一次, s 都会剔去其二进制表示中最右侧的 1, 最终循环的次数即为 x 的二进制表示中最右侧的 1 的数量。

```
int hammingDistance(int x, int y) {
    int cnt = 0;
    int s = x ^ y;
    while(s){
        s &= s-1;
        cnt++;
    }
    return cnt;
}
```

2.

```
int convertInteger(int A, int B) {
    unsigned int s = A ^ B;
    int ans = 0; 若用 int 可能会碰到 INT_MIN 溢出
    while(s){
        s &= s-1;
        ans++;
    }
    return ans;
}
```

3.

我们观察按位与运算的性质。对于一系列的位, 例如 $[1, 1, 0, 1, 1]$, 只要有一个零值的位, 那么这一系列位的按位与运算结果都将为零。

因此, 最终我们可以将问题重新表述为: 给定两个整数, 我们要找到它们对应的二进制字符串的公共前缀。

基于上述技巧, 我们可以用它来计算两个二进制字符串的公共前缀。

其思想是, 对于给定的范围 $[m, n]$ ($m < n$), 我们可以对数字 n 迭代地应用上述技巧, 清除最右边的 1, 直到它小于或等于 m , 此时非公共前缀部分的 1 均被消去。因此最后我们返回 n 即可。

清除 n 最右边的 1, 直到 $n \leq m$

输入	m =	9	0	0	0	0	1	0	0	1
	n =	12	0	0	0	0	1	1	0	0

第 1 步	m =	9	0	0	0	0	1	0	0	1
	n =	8	0	0	0	0	1	0	0	0

```
int rangeBitwiseAnd(int left, int right) {
    while(left < right){
        right &= right-1;
    }
    return right;
}
```

10. 数组

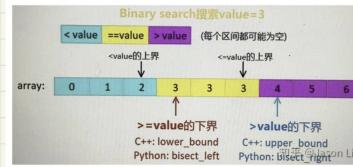
二分法 =>

```
/* 二分查找-找出<x<value>的下界
A[9] = {0,1,2,3,3,3,4,5,6};
lower_bound(A, 0, 8, 3);
*/
int lower_bound(int array[], int first, int last, int value)
{
    while(first < last)
    {
        int mid = first + (last-first)/2;
        if(array[mid] < value) {first = mid + 1;}
        else {last = mid;}
    }
    return first;
}
```

```
/* 二分查找-找出<x<value>的下界
A[9] = {0,1,2,3,3,3,4,5,6};
upper_bound(A, 0, 8, 3);
*/
int upper_bound(int array[], int first, int last, int value)
{
    while(first < last)
    {
        int mid = first + (last-first)/2;
        if(array[mid] <= value) {first = mid + 1;}
        else {last = mid;}
    }
    return first;
}
```

lower_bound(value)-1 找出 $x < value$ 的上界

upper_bound(value)-1 找出 $x \leq value$ 上界



二分 - 查找元素或者索引

704. 二分查找：给定一个 n 个元素有序的（升序）整型数组 nums 和一个目标值 target，写一个函数搜索 nums 中的 target，如果目标值存在返回下标，否则返回 -1。
- 剑指 Offer 53 - I. 在排序数组中查找数字 I：统计一个数字在排序数组中出现的次数。
35. 搜索插入位置：给定一个排序数组和一个目标值，在数组中找到目标值，并返回其索引。如果目标值不存在于数组中，返回它将会被按顺序插入的位置。
34. 在排序数组中查找元素的第一个和最后一个位置：给定一个按照升序排列的整数数组 nums，和一个目标值 target。找出给定目标值在数组中的开始位置和结束位置。如果数组中不存在目标值 target，返回 [-1, -1]。
- 面试题 08.03. 魔术索引：魔术索引。在数组 A[0..n-1] 中，有所谓的魔术索引，满足条件 A[i] = i。给定一个有序整数数组，编写一种方法找出魔术索引，若有的话，在数组 A 中找出一个魔术索引，如果没有，则返回-1。若有多个魔术索引，返回索引值最小的一个。

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

2. ①一次查找

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

②两次查找

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

→ lower_bound
→ upper_bound
为防止 upper_bound 的边界问题，如 [2,3,4]
当 right == nums.size()-1 且 nums[right] == target 时，返回 right-left+1；
当 right == nums.size()-1 且 nums[right] != target 时，返回 right-left；
当 right < nums.size()-1 且 nums[right] == target 时，返回 right-left+1；
当 right < nums.size()-1 且 nums[right] != target 时，返回 right-left；

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

注意边界，如 [1,2,3] 中要插入 1，二分查找
输出位置 2

4.

```

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

return ans;
}

```

二分 - 旋转排序数组

1. 33. 搜索旋转排序数组：升序排列的整数数组 nums 在预先未知的某个点上进行了旋转（例如，[0,1,2,4,5,6,7] 经旋转后可能变为 [4,5,6,7,0,1,2]）。请你在数组中搜索 target，如果数组中存在这个目标值，则返回它的索引，否则返回 -1。**无重复元素**
2. 81. 搜索旋转排序数组 II：假设按照升序排序的数组在预先未知的某个点上进行了旋转。（例如，数组 [0,1,2,2,5,6] 可能变为 [2,5,6,0,0,1,2]）。编写一个函数来判断给定的目标值是否存在于数组中。若存在返回 true，否则返回 false。**有重复元素**
3. 面试题 10.03. 搜索旋转数组：搜索旋转数组。给定一个排序后的数组，包含 n 个整数，但这个数组已被旋转过很多次了，次数不详。请编写代码找出数组中的某个元素，假设数组元素原来是按升序排列的。若有多个相同元素，返回索引值最小的一个。
4. 153. 寻找旋转排序数组中的最小值：假设按照升序排序的数组在预先未知的某个点上进行了旋转。例如，数组 [0,1,2,4,5,6,7] 可能变为 [4,5,6,7,0,1,2]。请找出其中最小的元素。**无重复元素**
5. 154. 寻找旋转排序数组中的最小值 II 和剑指 Offer 11. 旋转数组的最小关键字：与上一题的不同点在于数组中可能存在重复的元素。PS：虽然 154 标注是困难题，但是剑指 offer 11 标注是简单题。个人认为 153 做完必做 154，拓展思维能力。面试时候遇到的题目都比较简单，但是样例丰富，需要自行摸索隐藏条件。**有重复元素**

1.

将数组一分为二，其中一定有一个是有序的，另一个可能是有序，也能是部分有序。此时有序部分用二分法查找。无序部分再一分为二，其中一个一定有序，另一个可能有序，可能无序。就这样循环。

```

int search(vector<int>& nums, int target) {
    int first = 0, last = nums.size()-1, mid = 0;
    while(first <= last){
        mid = first + (last - first)/2;
        if(nums[mid] == target) {return mid;}
        if (nums[first] < nums[mid]) { // 左边部分是有序的
            if(target >= nums[first] && target < nums[mid]) {last = mid - 1;}
            else {first = mid + 1;}
        }
        else if (nums[first] > nums[mid]) { // 右边部分是有序的
            if(target <= nums[last] && target > nums[mid]) {first = mid + 1;}
            else {last = mid - 1;}
        }
        // 恢复数组的二段性，只有数组具有二段性（注意不是单调性）才可以使用二分查找
        else {first++;}
    }
    return -1;
}

```

5. 分治，带 DFS 味道的二分

```

int findMagicIndex(vector<int>& nums) {
    return dfs_binsearch(nums, 0, nums.size()-1);
}

int dfs_binsearch(vector<int>& nums, int left, int right){
    if(left > right) {return -1;}
    int mid = left + (right - left)/2;
    int left_ans = dfs_binsearch(nums, left, mid-1);
    if(left_ans != -1) {return left_ans;}
    else if(nums[mid] == mid) {return mid;}
    return dfs_binsearch(nums, mid+1, right);
}

```

边界问题，如 [2,2] 中找 2，
↑ 使用 upper_bound 会输出 2！

1. 第二种解法

① 找左边最下标为 0 的元素
② 使用 lower_bound 查找

```

int search(vector<int>& nums, int target) {
    if(nums.size() == 0) {return -1;}
    // 找到旋转点
    int first = 0, last = nums.size()-1, mid = 0;
    while(first <= last){
        mid = first + (last - first)/2;
        if(nums[mid] > nums[last]) {first = mid + 1;}
        else if (nums[mid] < nums[last]) {last = mid;}
        else {last--;}
    }
    int realo = first; // 旋转点 → 负相结果元 0
    // 使用 lower_bound 二分查找
    if(target > nums[nums.size()-1]) {first = 0; last = realo - 1;}
    else {first = realo; last = nums.size()-1;} // 判断选择哪一边找

    while(first < last){
        int mid = first + (last - first)/2;
        if(nums[mid] < target) {first = mid + 1;}
        else {last = mid;}
    }
    return nums[first] == target ? first : -1;
}

```

2. 二段性 → 某一段符合某个性质（如大于等于 3），而另外一段不符合这个性质

```

bool search(vector<int>& nums, int target) {
    int first = 0, last = nums.size()-1, mid = 0;
    while(first <= last){
        mid = first + (last - first)/2;
        if(nums[mid] == target) {return true;}
        if (nums[first] < nums[mid]) { // 左边部分是有序的
            if(target >= nums[first] && target < nums[mid]) {last = mid - 1;}
            else {first = mid + 1;}
        }
        else if (nums[first] > nums[mid]) { // 右边部分是有序的
            if(target <= nums[last] && target > nums[mid]) {first = mid + 1;}
            else {last = mid - 1;}
        }
        // 恢复数组的二段性，只有数组具有二段性（注意不是单调性）才可以使用二分查找
        else {first++;}
    }
    return false;
}

```

↑
所执行的与相同

3.

```

int search(vector<int>& arr, int target) {
    if(arr[0] == target) {return 0;} //注意
    int first = 0, last = arr.size()-1, mid = 0;
    while(first <= last){
        mid = first + (last - first)/2;
        if(arr[mid] == target) {
            while(mid > 1 && arr[mid - 1] == arr[mid]) {mid--;} //找索引最小的
            return mid;
        }
        if (arr[first] < arr[mid]){ //左边部分是有序的
            if(target >= arr[first] && target < arr[mid]) {last = mid - 1;}
            else {first = mid + 1;}
        }
        else if(arr[first] > arr[mid]){ // 右边部分是有序的
            if(target <= arr[last] && target > arr[mid]) {first = mid + 1;}
            else {last = mid - 1;}
        }
    }
    //恢复数组的二段性，只有数组具有二段性（注意不是单调性）才可以使用二分查找
    else {first++;}
}
return -1;
}

```

4.5、相当于旋转前的下标为0的元素（旋转点）

```

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

```

二分 - 山脉数组

941. 有效的山脉数组：给定一个整数数组 A，如果它是有效的山脉数组就返回 true，否则返回 false。
852. 山脉数组的峰顶索引：给你由整数组成的山脉数组 arr，返回任何满足 $arr[0] < arr[1] < \dots < arr[i-1] < arr[i] > arr[i+1] > \dots > arr[arr.length-1]$ 的下标 i。
提升：返回峰顶对应的值，即山脉数组的最大值。
1095. 山脉数组中查找目标值：给你一个 山脉数组 mountainArr，请你返回能够使得 mountainArr.get(index) 等于 target 最小的下标 index 值。PS：较难，建议量力而行。

1.

```

bool validMountainArray(vector<int>& arr) {
    if(arr.size() < 3) {return false;}
    int first = 0, last = arr.size()-1;
    while(first < arr.size()-1 && arr[first] < arr[first+1]) {first++;}
    while(last > 0 && arr[last] < arr[last-1]) {last--;}
    if(first == last && first > 0 && last < arr.size()-1) {return true;}
    else {return false;}
}

```

2.

```

int peakIndexInMountainArray(vector<int>& arr) {
    //first设为1是因为while循环中mid-1可能会超出索引，且索引是0的元素必不是山峰
    int first = 1, last = arr.size()-1, mid = 0;
    while(first < last){
        mid = first + (last - first)/2;
        if(arr[mid] > arr[mid-1] && arr[mid] > arr[mid+1]) {return mid;}
        else if(arr[mid] < arr[mid-1]) {last = mid - 1;}
        else {first = mid + 1;}
    }
    return first;
}

```

```

3、
int findInMountainArray(int target, MountainArray &mountainArr) {
    int Topindex = findTop(mountainArr);
    int Topvalue = mountainArr.get(Topindex);
    if(target == Topvalue) {return Topindex;}
    if(target > Topvalue) {return -1;}
    int ans = binsearch1(0, Topindex-1, target, mountainArr);
    if(ans != -1) {return ans;}
    else {return binsearch2(Topindex+1, mountainArr.length()-1, target, mountainArr);}
}

int binsearch1(int left, int right, int target, MountainArray &mountainArr){
    int first = left, last = right, mid = 0;
    while(first < last){
        mid = first + (last - first)/2;
        if(mountainArr.get(mid) < target) {first = mid + 1;}
        else {last = mid;}
    }
    if(mountainArr.get(first) == target) {return first;}
    else {return -1;}
}

int binsearch2(int left, int right, int target, MountainArray &mountainArr){
    int first = left, last = right, mid = 0;
    while(first < last){
        mid = first + (last - first)/2;
        if(mountainArr.get(mid) > target) {first = mid + 1;}
        else {last = mid;}
    }
    if(mountainArr.get(first) == target) {return first;}
    else {return -1;}
}

int findTop(MountainArray &mountainArr) {
    int first = 1, last = mountainArr.length()-1, mid = 0;
    while(first < last){
        mid = first + (last - first)/2;
        if(mountainArr.get(mid) > mountainArr.get(mid-1) && mountainArr.get(mid) > mountainArr.get(mid+1)) {return mid;}
        else if(mountainArr.get(mid) < mountainArr.get(mid-1)) {last = mid - 1;}
        else {first = mid + 1;}
    }
    return first;
}

```

二维矩阵

1. 剑指 Offer 29. 顺时针打印矩阵：输入一个矩阵，按照从外向里以顺时针的顺序依次打印出每一个数字。

提升：顺时针打印矩阵的最后一个点的坐标（假设坐标从0开始）

2. 54. 螺旋矩阵：给定一个包含 $m \times n$ 个元素的矩阵（ m 行, n 列），请按照顺时针螺旋顺序，返回矩阵中的所有元素。PS：与上一题只有返回值的数据类型不同。

3. 55. 螺旋矩阵 II：给定一个正整数 n ，生成一个包含 1 到 n^2 所有元素，且元素按顺时针顺序螺旋排列的正方形矩阵。

4. 48. 旋转图像和面试题 01.07. 旋转矩阵：给定一个 $n \times n$ 的二维矩阵表示一个图像。将图像顺时针旋转90度。

提升：逆时针90度，顺时针或逆时针180度（翻转）。

5. 378. 有序矩阵中第k小的元素：给定一个 $n \times n$ 矩阵，其中每行和每列元素均按升序排序，找到矩阵中第 k 小的元素。

提升：有序矩阵中第 k 大的元素

6. 74. 搜索二维矩阵：编写一个高效的算法来判断 $m \times n$ 矩阵中，是否存在一个目标值。该矩阵具有如下特性：每行的整数从左到右按升序排列。每行的第一个整数大于前一行的最后一个整数。

7. 240. 搜索二维矩阵 II、剑指 Offer 04. 二维数组中的查找和面试题 10.09. 排序矩阵查找：编写一个高效的算法来搜索 $m \times n$ 矩阵 matrix 中的一个目标值 target。该矩阵具有以下特性：每行的元素从左到右升序排列。每列的元素从上到下升序排列。

0	1	2	3	4
0	0	0	0	0
1	0	1	2	3
2	0	4	5	6
3	0	7	8	9
4	0	0	0	0

⇒ 对递增数组二分查找

⇒ 对递减数组二分查找

⇒ 找山峰

1.2 在原矩阵外面包围一圈

```

vector<int> spiralOrder(vector<vector<int>>& matrix) {
    if(matrix.size() == 0) {return {};}
    vector<vector<bool>> visited(matrix.size() + 2, vector<bool>(matrix[0].size() + 2, false));
    for(int j = 0; j < visited[0].size(); j++){
        visited[0][j] = true;
        visited[visited.size() - 1][j] = true;
    }
    for(int i = 0; i < visited.size(); i++){
        visited[i][0] = true;
        visited[i][visited[0].size() - 1] = true;
    }
    vector<int> ans;
    vector<vector<int>> direction(4, vector<int>(2));
    direction[0][0] = 0; direction[0][1] = 1;
    direction[1][0] = 1; direction[1][1] = 0;
    direction[2][0] = 0; direction[2][1] = -1;
    direction[3][0] = -1; direction[3][1] = 0;
    int di = 0;
    int x = 1, y = 1;
    while(1){
        if(!visited[x][y]){
            visited[x][y] = true;
            ans.emplace_back(matrix[x-1][y-1]);
            int new_x = x + direction[di%4][0], new_y = y + direction[di%4][1];
            if(!visited[new_x][new_y]) {x = new_x; y = new_y;}
            else {break;}
        }
        else{
            int new_x = x + direction[di%4][0], new_y = y + direction[di%4][1];
            if(!visited[new_x][new_y]) {x = new_x; y = new_y;}
            else {break;}
        }
    }
    return ans;
}

```



3. 与题1相同

```

vector<vector<int>> generateMatrix(int n) {
    if(n == 0) {return {};}
    vector<vector<int>> matrix(n, vector<int>(n, 0));
    vector<vector<bool>> visited(n*2, vector<bool>(n*2, false));
    for(int j = 0; j < visited.size(); j++) {
        visited[0][j] = true;
        visited[visited.size()-1][j] = true;
        visited[j][0] = true;
        visited[j][visited.size()-1] = true;
    }
    vector<vector<int>> direction(4, vector<int>(2));
    direction[0][0] = 0; direction[0][1] = 1;
    direction[1][0] = 1; direction[1][1] = 0;
    direction[2][0] = 0; direction[2][1] = -1;
    direction[3][0] = -1; direction[3][1] = 0;
    int di = 0;
    int x = 1, y = 1;
    int cnt = 0;
    while(1){
        if(!visited[x][y]){
            visited[x][y] = true;
            matrix[x-1][y-1] = (+cnt);
            int new_x = x + direction[di%4][0], new_y = y + direction[di%4][1];
            if(!visited[new_x][new_y]) {x = new_x; y = new_y;}
            else {break;}
        }
        else{
            int new_x = x + direction[di%4][0], new_y = y + direction[di%4][1];
            if(!visited[new_x][new_y]) {x = new_x; y = new_y;}
            else {break;}
        }
    }
    return matrix;
}

```

5. 手链十脸帝十队列

```

int kthSmallest(vector<vector<int>>& matrix, int k) {
    int n = matrix.size();
    priority_queue<int, vector<int>, greater<int> heap1;//小根堆
    unordered_map<int, queue<pair<int, int>> hash1;//记录数值所对应的坐标
    for(int i = 0; i < n; i++){
        heap1.emplace(matrix[i][0]);
        hash1[matrix[i][0]].push(make_pair(i, 0));
    }
    for(int j = 0; j < k-1; j++){
        int temp = heap1.top();
        heap1.pop();
        int x = hash1[temp].front().first, y = hash1[temp].front().second;
        hash1[temp].pop();
        if(y != n-1){
            heap1.emplace(matrix[x][y+1]);
            hash1[matrix[x][y+1]].push(make_pair(x, y+1));
        }
    }
    return heap1.top();
}

```

其实这个题的难点在于，用堆+哈希表的插入删除：你需要保证小堆中的候选插入到堆中。于是每次从堆中取出候选人的时候，将堆中第一个放入堆中。这样就可以保证堆中只有一行。因为每次取出候选人的时候，将堆中第一个放入堆中。在堆中只有一行。

1	2	3	4	5
3	5	8	10	12
4	7	10	12	17
6	9	11	15	20
8	10	12	16	25

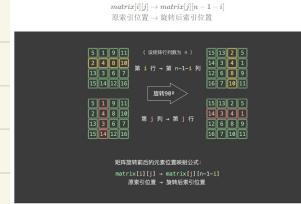
第一次选出的候选人，将左上角的数标出即可。选出之后，我们如何找到下一个候选人呢？

2	3	4	5	
3	5	8	10	12
4	7	10	12	17
6	9	11	15	20
8	10	12	16	25

4. ①辅助矩阵

如图所示，矩阵顺时针转 90° 后，可找到以下规律：

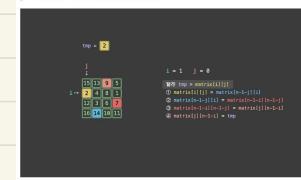
- 第 i 行元素旋转到第 i-1 列 i 元素；
 - 第 i 列元素旋转到第 i 行 i 元素；
- 因此，对于矩阵任意第 i 行、第 j 列元素 $matrix[i][j]$ ，则旋转 90° 后「元素位置旋转公式」为：



② 原地旋转

具体来看，当矩阵大小 n 为偶数时，取第 $\frac{n}{2}$ 行、前 $\frac{n}{2}$ 列的元素为起始点；当矩阵大小 n 为奇数时，取第 $\frac{n-1}{2}$ 行、前 $\frac{n-1}{2}$ 列的元素为起始点。

令 $matrix[i][j] = A$ ，根据数组开头的元素加总公式，可推导适用于任意起始点的元素旋转逻辑：



```

void rotate(vector<vector<int>>& matrix) {
    int n = matrix.size();
    for(int i = 0; i < n/2; i++){
        for(int j = 0; j < (n+1)/2; j++){
            int tmp = matrix[i][j];
            matrix[i][j] = matrix[n-1-j][i];
            matrix[n-1-j][i] = matrix[n-1-i][n-1-j];
            matrix[n-1-i][n-1-j] = matrix[j][n-1-i];
            matrix[j][n-1-i] = tmp;
        }
    }
}

```

<https://leetcode.cn/problems/rotate-image/solutions/1228078/48-xuan-zhuan-tu-xiang-fu-zhu-ju-zhen-yu-jobi/>

其实非常简单，刚才弹出的位置右移一格就行了。这样还是能保证候选入列表中每一个数字每一行的最小值。那全局最小值必然在其中！我们首次选出候选人，当中的最小值，然后把上次选出候选人的右边一个补进来，就能一直保证全局最小值在候选人列表中产生。示例：（穿黄色衣服的为候选人）（顺序是每一行都是从左向右看）（当某一行弹掉东西，候选人列表的长度就会少1）

1	2	3	4	5
5	8	10	12	12
4	7	10	12	17
6	9	11	15	20
8	10	12	16	25

4	5			
5	8	10	12	12
4	7	10	12	17
6	9	11	15	20
8	10	12	16	25

8	10	12		
7	10	12	17	
6	9	11	15	20
8	10	12	16	25

<https://leetcode.cn/problems/kth-smallest-element-in-a-sorted-matrix/solutions/312485/shi-yong-dui-heapde-si-lu-xiang-jie-ling-fu-python/>

6. 坐标转换后进行二分查找

```
bool searchMatrix(vector<vector<int>>& matrix, int target) {
    int m = matrix.size(), n = matrix[0].size();
    int first = 0, last = m*n - 1, mid = 0;
    int mid_x = 0, mid_y = 0;
    while(first < last){
        mid = first + (last - first)/2;
        mid_x = mid / n; mid_y = mid % n;
        if(matrix[mid_x][mid_y] < target) {first = mid + 1;}
        else {last = mid;}
    }
    if(matrix[first/n][first%n] == target) {return true;}
    else {return false;}
}
```

7. 从左上角开始查找

```
bool searchMatrix(vector<vector<int>>& matrix, int target) {
    int m = matrix.size(), n = matrix[0].size();
    int x = 0, y = n - 1;
    while(x < m && y >= 0){
        if(matrix[x][y] > target) {y--;}
        else if(matrix[x][y] < target) {x++;}
        else {return true;}
    }
    return false;
}
```

滑动窗口

该类题目量力而行，普通求职者遇上的概率不大，面试官为你或者对你期望比较高的时候很有可能会出这种常见的困难题。

1. 239. 滑动窗口最大值和剑指 Offer 59 - I. 滑动窗口的最大值：给你一个整数数组 nums，有一个大小为 k 的滑动窗口从数组的最左侧移动到数组的最右侧。你只可以看到在滑动窗口内的 k 个数字。滑动窗口每次只向右移动一位。返回滑动窗口中的最大值。

提升1：修改输出的数据类型，比如：java 中将输出改成 List<Integer>

提升2：输入的数组或者滑动窗口为空

2. 480. 滑动窗口中位数：给你一个数组 nums，有一个大小为 k 的窗口从最左端滑动到最右端。

窗口中有 k 个数，每次窗口向右移动 1 位。你的任务是找出每次窗口移动后得到的新窗口中元素的中位数，并输出由它们组成的数组。

提升：如果数组个数是偶数，则在该窗口排序数字后，返回第 N/2 个数字。

1. ★看双指针章节 (P.7)

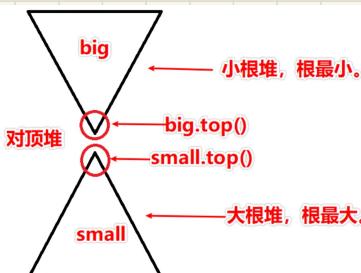
{ 双向队列
优先队列 }

2. ★

```
priority_queue<int, vector<int>, less<int> small; // 小根堆
priority_queue<int, vector<int>, greater<int> big; // 大根堆
unordered_map<int, int> mp;
```

```
double get_median(int k){
    if(k % 2) {return small.top();}
    else {return ((long long)small.top() + big.top()) * 0.5;}
}
```

```
vector<double> medianSlidingWindow(vector<int>& nums, int k) {
    for(int i = 0; i < k; i++) {small.emplace(nums[i]);}
    for(int i = 0; i < k; i++) {big.emplace(small.top()); small.pop();}
    vector<double> ans;
    ans.emplace_back(get_median(k));
    for(int i = k; i < nums.size(); i++){
        if(balance == 0, l = nums[i-k];
        mp[l]++;
        if(!small.empty() && l <= small.top()) {balance--;}
        else {balance++;}
        if(!small.empty() && nums[i] <= small.top()){
            small.emplace(nums[i]);
            balance++;
        }
        else{
            big.emplace(nums[i]);
            balance--;
        }
        if(balance > 0){
            big.emplace(small.top());
            small.pop();
        }
        else if(balance < 0){
            small.emplace(big.top());
            big.pop();
        }
        while(!small.empty() && mp[small.top()] > 0){
            mp[small.top()]--;
            small.pop();
        }
        while(!big.empty() && mp[big.top()] > 0){
            mp[big.top()]--;
            big.pop();
        }
        ans.emplace_back(get_median(k));
    }
    return ans;
}
```



<https://leetcode.cn/problems/sliding-window-median/solutions/589213/feng-xian-dui-chong-shuang-dui-dui-ding-hq1dt/>

中位数

与滑动窗口一样，属于常见的困难题，有一定可能遇到，建议量力而行。

1. 295. 数据流的中位数、剑指 Offer 41. 数据流中的中位数和面试题 17.20. 连续中值：如何得到一个数据流中的中位数？如果从数据流中读出奇数个数值，那么中位数就是所有数值排序之后位于中间的数值。如果从数据流中读出偶数个数值，那么中位数就是所有数值排序之后中间两个数的平均值。
提升：假设数字是不断进入数组的，在每次添加一个新的数进入数组的同时返回当前新数组的中位数。例如输入[1,2,3,4,5]，返回[1,1.2,2,3]。

```
class MedianFinder {
public:
    /** initialize your data structure here. */
    priority_queue<int, vector<int>, less<int>> small;
    priority_queue<int, vector<int>, greater<int>> big;
    MedianFinder(){}
    void addNum(int num) {
        if(small.empty() && big.empty()) {small.emplace(num); return;}
        if(!small.empty() && num <= small.top()){
            small.emplace(num);
            if(big.size() + 1 < small.size()){
                big.emplace(small.top());
                small.pop();
            }
        } else{
            big.emplace(num);
            if(big.size() > small.size()){
                small.emplace(big.top());
                big.pop();
            }
        }
    }
    double findMedian() {
        if(small.size() > big.size()) {return small.top();}
        else {return (long long)(small.top() + big.top()) * 0.5;}
    }
};
```

与上一题相似
优先队列实现双堆

摩尔投票法 ==> 求众数

1. 169. 多数元素、面试题 17.10. 主要元素和剑指 Offer 39. 数组中出现次数超过一半的数字：给定一个大小为 n 的数组，找到其中的多数元素。多数元素是指在数组中出现次数大于 $\lfloor n/2 \rfloor$ 的元素。你可以假设数组是非空的，并且给定的数组总是存在多数元素。

思考：如果不保证给定数组总是存在多数元素，还能用摩尔投票法么？如果不能的话，思考下用什么方法做。→ 内联验证

2. 229. 求众数 II：给定一个大小为 n 的整数数组，找出其中所有出现超过 $\lfloor n/3 \rfloor$ 次的元素。

```
int majorityElement(vector<int>& nums) {
    int ans = 0, votes = 0;
    for(int num:nums){
        if(votes == 0) {ans = num;}
        votes += (num == ans ? 1 : -1);
    }
    return ans;
}
```

```
int cnt = 0;
for(int num:nums){
    if(num == ans) {cnt++;}
}
return cnt > nums.size()/2 ? ans : -1;
```

<https://leetcode.cn/problems/shu-zu-zhong-zhong-shu-jie-shu-chao-guo-yi-ban-de-shu-zu-lcof/solutions/138691/mian-shi-ti-39-shu-zu-zhong-zhong-ci-shu-chao-3/>

输入数组 $nums$ 的众数为 x ，数组长度为 n 。

推论一：若记 $众数$ 的票数为 $+1$ ， $非众数$ 的票数为 -1 ，则一定有所有数字的票数和 > 0 。

推论二：若数组的前 $n - k$ 个数字的票数和 $= 0$ ，则数组剩余 $(n - k)$ 个数字的票数和一定仍 > 0 ，即后 $(n - k)$ 个数字的众数仍为 x 。



根据以上推论，记数组首个元素为 n_1 ，众数为 x ，遍历并统计票数。当发生 $票数 = 0$ 时，新的众数的众数一定不变，这是由于：

- 当 $n_1 = x$ ：抵消的所有数字中，有一半是众数 x 。

- 当 $n_1 \neq x$ ：抵消的所有数字中，众数 x 的数量至少为 0 个，最多为一半。

利用此特性，每轮假设发生 $票数 = 0$ 都可以缩小剩余数组区间。当遍历完成时，最后一轮假设的数字即为众数。

2.

```

vector<int> majorityElement(vector<int>& nums) {
    int ans1 = 0, ans2 = 0, votes1 = 0, votes2 = 0;
    for(int num:nums){
        if(votes1 > 0 && num == ans1) {votes1++}; //如果该元素为第一个元素，则计数加1
        else if(votes2 > 0 && num == ans2) {votes2++}; //如果该元素为第二个元素，则计数加1
        else if(votes1 == 0) {ans1 = num; votes1++}; //选择第一个元素
        else if(votes2 == 0) {ans2 = num; votes2++}; //选择第二个元素
        else {votes1--; votes2--}; //如果三个元素均不相同，则相互抵消1次
    }

    //验证
    int cnt1 = 0, cnt2 = 0;
    for(int num:nums){
        if(num == ans1) {cnt1++};
        else if(num == ans2) {cnt2++};
    }

    vector<int> ans;
    if(votes1 > 0 && cnt1 > nums.size()/3) {ans.emplace_back(ans1)};
    if(votes2 > 0 && cnt2 > nums.size()/3) {ans.emplace_back(ans2)};
    return ans;
}

```

- 与超过 $\lceil n/2 \rceil$ 次 2个不同被抵消相似，超过 $\lceil n/3 \rceil$ 次是3个不同的元素被抵消
- 且答案最多只有 $k-1$ 个，在这里 $k=3$ ，故最多有2个答案

- 我们每次检测当前元素是否为第一个选中的元素或者第二个选中的元素。
- 每次我们发现当前元素与已经选中的两个元素都不相同，则进行抵消一次。
- 如果存在最终选票大于0的元素，我们还需要再次统计已选中元素的次数，检查元素的次数是否大于 $\lceil \frac{n}{3} \rceil$ 。

子序列/子数组

子序列可以不连续，子数组连续

1. 491. 递增子序列：给定一个整型数组，你的任务是找到所有该数组的递增子序列，递增子序列的长度至少是2。

提升：输出最长的递增子序列 / 输出最长的非递减子序列 / 输出最长的递减子序列 / 输出最长的非递增子序列

2. 300. 最长递增子序列：给你一个整数数组 nums，找到其中最长严格递增子序列的长度。

提升1：输出最长的非递减子序列的长度 / 输出最长的递减子序列的长度 / 输出最长的非递增子序列的长度 → 输出具体的动态规划方法有利

提升2：将上述题目中的子序列改成子数组（必须连续）。输出最长递增子数组长度，最长递增子数组具体数组，最长非递减子数组长度，最长非递减子数组具体数组，最长递减子数组长度，最长递减子数组具体数组，最长非递增子数组长度，最长非递增子数组具体数组。PS：量力而行，做到其中一个以后再改其他的。

3. 53. 最大子序和面试题 16.17. 连续数列：给定一个整数数组 nums，找到一个具有最大和的连续子数组（子数组最少包含一个元素），返回其最大和。

提升：输出组成最大子序和的具体数组，如有多个，尝试输出任意一个/所有可以构成最大子序和的组合 → 输出具体的动态规划方法有利

4. 209. 长度最小的子数组：给定一个含有 n 个正整数的数组和一个正整数 s，找出该数组中满足其和 ≥ s 的长度最小的 连续 子数组，并返回其长度。如果不存在符合条件的子数组，返回 0。

5. 581. 最短无序连续子数组：给你一个整数数组 nums，你需要找出一个连续子数组，如果对这个子数组进行升序排序，那么整个数组都会变为升序排序。请你找出符合题意的最短子数组，并输出它的长度。

6. 718. 最长重复子数组：给两个整数数组 A 和 B，返回两个数组中公共的、长度最长的子数组的长度。

7. 907. 子数组的最小值之和：给定一个整数数组 A，找到 min(B) 的总和，其中 B 的范围为 A 的每个（连续）子数组。

8. 560. 和为K的子数组：给定一个整数数组和一个整数 k，你需要找到该数组中和为 k 的连续的子数组的个数。

9. 152. 乘积最大子数组：给你一个整数数组 nums，请你找出数组中乘积最大的连续子数组（该子数组中至少包含一个数字），并返回该子数组所对应的乘积。

1. DFS回溯 + unordered_set去重

```

vector<vector<int>> findSubsequences(vector<int>& nums) {
    vector<int> path;
    vector<vector<int>> ans;
    dfs(nums, 0, path, ans);
    return ans;
}

void dfs(vector<int>& nums, int index, vector<int>& path, vector<vector<int>> &ans){
    if(path.size() > 1){
        ans.emplace_back(path);
    }

    unordered_set<int> s;
    for(int i=index+1 < nums.size(); i++){
        if(s.count(nums[i])) {continue};
        else {s.emplace(nums[i])};
        if(path.size() == 0 || nums[index-1] <= nums[i]){
            path.emplace_back(nums[i]);
            dfs(nums, i+1, path, ans);
            path.pop_back();
        }
    }
    return;
}

```

2. ① 动态规划

① dp[i]的定义

dp[i]表示i之前包括i的最长上升子序列。

② 状态转移方程

位置的最长升序子序列等于从0到i-1各个位置的最长升序子序列 + 1的最大值。

所以：if (nums[i] > nums[j]) dp[i] = max(dp[i], dp[j] + 1);

注意这里不是要dp[i]与dp[j] + 1进行比较，而是我们要取dp[j] + 1的最大值。

③ dp[i]的初始化

每一个i，对应的dp[i]（即最长上升子序列）起始大小至少都是1。

④ 确定遍历顺序

dp[i]是有0到i-1各个位置的最长升序子序列推导而来，那么遍历一定是从前向后遍历。

```

int lengthOfLIS(vector<int>& nums) {
    if(nums.size() <= 1) {return nums.size();}

    vector<int> dp(nums.size(), 1);
    int ans = 0;

    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);}
        }
        if(dp[i] > ans) {ans = dp[i];}
    }

    return ans;
}

```

② DFS 回溯(会超时)

```

int lengthOfLIS(vector<int>& nums) {
    vector<int> path;
    int ans = 0;
    dfs(nums, 0, path, ans);
    return ans;
}

void dfs(vector<int>& nums, int index, vector<int>& path, int& ans){
    int n = path.size();
    if(n > 0) {ans = max(ans, n);}
    unordered_set<int> s;
    for(int i = index; i < nums.size(); i++){
        if(s.count(nums[i])) {continue;}
        else {s.emplace(nums[i]);}
        if(path.size() == 0 || nums[index-1] < nums[i]){
            path.emplace_back(nums[i]);
            dfs(nums, i+1, path, ans);
            path.pop_back();
        }
    }
}
return;
}

```

子数组情况

```

int lengthLIA(vector<int>& nums){
    if(nums.size() <= 1) {return nums.size();}
    vector<int> ans;
    int i = 1, ans_cnt = 1, cnt = 1;
    while(i < nums.size()){
        if(nums[i] > nums[i-1]) {cnt++;}
        else{
            ans_cnt = max(ans_cnt, cnt);
            cnt = 1;
        }
        i++;
    }
    ans_cnt = max(ans_cnt, cnt);
    return ans_cnt;
}

```

3. ① 直接解

```

int maxSubArray(vector<int>& nums) {
    int sum = 0, max_sum = INT_MIN;
    for(int i = 0; i < nums.size(); i++){
        sum += nums[i];
        max_sum = max(max_sum, sum);
        if(sum < 0) {sum = 0;}
    }
    return max_sum;
}

```

② 动态规划

① 定义状态 (定义子问题)

$dp[i]$: 表示以 $nums[i]$ 结尾的连续子数组的最大和。

② 状态转移方程 (描述子问题之间的联系)

根据状态的定义，由于 $nums[i]$ 一定会被选取，并且以 $nums[i]$ 结尾的连续子数组与以 $nums[i-1]$ 结尾的连续子数组只相差一个元素 $nums[i]$ 。

假设数组 $nums$ 的值全部严格大于 0，那么一定有 $dp[i] = dp[i-1] + nums[i]$ 。

可是 $dp[i-1]$ 可能是负数，于是分类讨论：

- 如果 $dp[i-1] > 0$ ，那么可以把 $nums[i]$ 直接接在 $dp[i-1]$ 表示的那个数组的后面，得到更大的连续子数组；
- 如果 $dp[i-1] \leq 0$ ，那么 $nums[i]$ 加上前面的数 $dp[i-1]$ 后值不会变大。于是 $dp[i]$ 「另立门户」，此时单独的一个 $nums[i]$ 的值，就是 $dp[i]$ 。

以上两种情况的最大值就是 $dp[i]$ 的值，写出如下状态转移方程：

$$dp[i] = \begin{cases} dp[i-1] + nums[i], & \text{if } dp[i-1] > 0 \\ nums[i], & \text{if } dp[i-1] \leq 0 \end{cases}$$

③ 思考初始值

$dp[0]$ ：根据定义，只有 1 个数，一定以 $nums[0]$ 结尾，因此 $dp[0] = nums[0]$ 。

```

int maxSubArray(vector<int>& nums) {
    vector<int> dp(nums.size());
    dp[0] = nums[0];
    int ans = dp[0];
    for(int i = 1; i < nums.size(); i++){
        if(dp[i-1] < 0) {dp[i] = nums[i];}
        else {dp[i] = dp[i-1] + nums[i];}
        ans = max(ans, dp[i]);
    }
    return ans;
}

```

最后再谈谈「无后效性」

「无后效性」是我多次提到的一个「动态规划」中非常重要的概念，在我看来，理解这个概念无比重要。很遗憾，《算法导论》上没有讲到「无后效性」。我找了一本在《豆瓣》目前豆瓣上评分 9.2 的书《算法竞赛进阶指南》，这本书和《算法导论》《算法 4》和 liuyubobobo 老师的算法课程一样，在我学习算法与数据结构的路上，都发挥了巨大的作用。

李煜东著《算法竞赛进阶指南》，摘录如下：

为了保证计算子问题能够按照顺序、不重复地进行，动态规划要求已经求解的子问题不受后续阶段的影响。这个条件也被叫做「无后效性」。换言之，动态规划对状态空间的遍历构成一张有向无环图，遍历就是该有向无环图的一个拓扑序。有向无环图中的节点对应问题中的「状态」，图中的边则对应状态之间的「转移」，转移的选取就是动态规划中的「决策」。

即 $dp[i]$ 不受 $dp[i-1]$ 影响，且 $dp[i]$ 后必可求出 $dp[i]$

4. ① 滑动窗口

定义两个指针 `start` 和 `end` 分别表示子数组（滑动窗口窗口）的开始位置和结束位置，维护变量 `sum` 存储子数组中的元素和（即从 `nums[start]` 到 `nums[end]` 的元素和）。

初始状态下，`start` 和 `end` 都指向下标 0，`sum` 的值为 0。

每一轮迭代，将 `nums[end]` 加到 `sum`，如果 `sum ≥ s`，则更新子数组的最小长度（此时子数组的长度是 `end - start + 1`），然后将 `nums[start]` 从 `sum` 中减去并将 `start` 右移，直到 `sum < s`，在此过程中同样更新子数组的最小长度。在每一轮迭代的最后，将 `end` 右移。

```
int minSubArrayLen(int target, vector<int>& nums) {
    int first = 0, last = 0, sum = 0, min_cnt = INT_MAX;
    for(; last < nums.size(); last++){
        sum += nums[last];
        while(sum >= target){
            min_cnt = min(min_cnt, last - first + 1);
            sum -= nums[first];
            first++;
        }
    }
    return min_cnt == INT_MAX ? 0 : min_cnt;
}
```

6. ① 动态规划

1. 状态 $\Rightarrow dp[i][j]$ 表示 $nums1[i:j]$ 和 $nums2[i:j]$ 的最长公共前缀

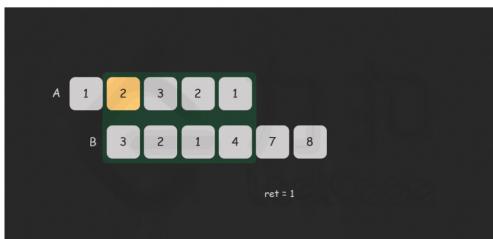
2. 状态转移方程 $\Rightarrow dp[i][j] = \begin{cases} dp[i+1][j+1] + 1, & nums1[i:j] = nums2[i:j] \\ 0, & nums1[i:j] \neq nums2[i:j] \end{cases}$

3. 初始值 $\Rightarrow dp[0:nums1.size()][0:nums2.size()] = 0$

```
int findLength(vector<int>& nums1, vector<int>& nums2) {
    int n1 = nums1.size(), n2 = nums2.size(), ans = INT_MIN;
    vector<vector<int>> dp(n1 + 1, vector<int>(n2 + 1));
    dp[n1][n2] = 0;
    for(int i = n1 - 1; i >= 0; i--){
        for(int j = n2 - 1; j >= 0; j--){
            if(nums1[i] == nums2[j]) {dp[i][j] = dp[i+1][j+1] + 1;}
            else {dp[i][j] = 0;}
            ans = max(ans, dp[i][j]);
        }
    }
    return ans;
}
```

② 滑动窗口

我们可以枚举 A 和 B 所有的对齐方式。对齐的方式有两类：第一类为 A 不变，B 的首元素与 A 中的某个元素对齐；第二类为 B 不变，A 的首元素与 B 中的某个元素对齐。对于每一种对齐方式，我们计算它们相对位置相同的重复子数组即可。



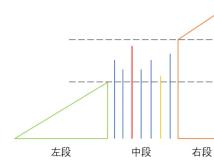
5. ① 力扣

```
int findUnsortedSubarray(vector<int>& nums) {
    int begin = 0, end = -1;
    vector<int> nums_s = nums;
    sort(nums_s.begin(), nums_s.end());
    for(int i = 0; i < nums.size(); i++){
        if(nums[i] != nums_s[i]) {begin = i; break;}
    }
    for(int i = nums.size() - 1; i >= 0; i--){
        if(nums[i] != nums_s[i]) {end = i; break;}
    }
    return end - begin + 1;
}
```

② 一次遍历

```
int findUnsortedSubarray(vector<int>& nums) {
    int n = nums.size();
    int begin = 0, end = -1;
    int max = nums[0], min = nums[n-1];
    for(int i = 0; i < n; i++){
        if(nums[i] < max) {end = i;}
        else {max = nums[i];}
        if(nums[n-i-1] > min) {begin = n-i-1;}
        else {min = nums[n-i-1];}
    }
    return end - begin + 1;
}
```

我们可以将这个数组分成三段：左段、右段和中间段。左段是标准的升序数组，中间数组是无序的，但满足左小大于右，右大小于左的最大值。最大值小于右的最小值。



那么我们目标就很明确了，找中段的左右边界，我们分别定义为 `begin` 和 `end`：分两头开始遍历。

- 从左到右维护一个最大值 `max`，在进入右段之前，那么遍历到的 `max[i]` 都是小于 `max` 的。我们要找的 `end` 就是遍历到第一个小于 `max` 元素的位置。
- 同理，从右到左维护一个最小值 `min`，在进入左段之前，那么遍历到的 `min[i]` 也都是大于 `min` 的。我们要找的 `begin` 就是最后一个大于 `min` 元素的位置。

```
int findLength(vector<int>& nums1, vector<int>& nums2) {
    int n1 = nums1.size(), n2 = nums2.size();
    int ans = INT_MIN, maxlen = 0;
    for(int i = 0; i < n1; i++){
        maxlen = findmax(nums1, nums2, i, 0, min(n1 - i, n2));
        ans = max(ans, maxlen);
    }
    for(int i = 0; i < n2; i++){
        maxlen = findmax(nums1, nums2, 0, i, min(n1, n2 - i));
        ans = max(ans, maxlen);
    }
    return ans;
}

int findmax(vector<int>& nums1, vector<int>& nums2, int add1, int add2, int len){
    int length = INT_MIN, cnt = 0;
    for(int i = 0; i < len; i++){
        if(nums1[add1 + i] == nums2[add2 + i]) {cnt++;}
        else {cnt = 0;}
        length = max(length, cnt);
    }
    return length;
}
```

7. ① 动态规划+栈(单调递增栈)

```
int sumSubarrayMins(vector<int>& arr) {
    long long ans = 0;
    long long mod = 1e9 + 7;
    stack<int> st;
    vector<int> dp(arr.size());
    for(int i = 0; i < arr.size(); i++){
        while(!st.empty() && arr[st.top()] > arr[i]) {st.pop();}
        int k = st.empty() ? (i+1) : (i-st.top());
        dp[i] = (st.empty() ? 0 : dp[i-k] + k * arr[i];
        ans = (ans + dp[i]) % mod;
        st.emplace(i);
    }
    return ans;
}
```

② 暴力解(会超时)

```
int sumSubarrayMins(vector<int>& arr) {
    long long ans = 0;
    long long mod = 1e9 + 7;
    for(int i; i < arr.size(); i++){
        int minval = arr[i];
        for(int j = i; j < arr.size(); j++){
            minval = min(minval, arr[j]);
            ans = (ans + minval) % mod;
        }
    }
    return ans;
}
```

设以 $arr[i]$ 为最右最小的值的子数组长度为 k :

- 当 $j - i - k + 1$ 时：连续子序列 $[arr[j], arr[j+1], \dots, arr[i]]$ 的最小值为 $arr[i]$ ，即 $arr[j] = arr[i]$ 。
- 当 $j - i - k + 1$ 时：连续子序列 $[arr[j], arr[j+1], \dots, arr[i]]$ 的最小值为 $arr[i]$ ，即 $arr[j] < arr[i]$ ，更小。通过分析可以得出它的最小值 $arr[j][i] = \min(arr[j], arr[i]) = arr[i] - k + 1$ 。

则可以知道递推公式如下：

$$\begin{aligned} \sum_{j=i}^k arr[j] &= \sum_{j=i}^{i+k-1} arr[j] + \sum_{j=i+k}^k arr[j] \\ &= \sum_{j=i}^{i+k-1} arr[j] + k \times arr[i] \\ &= \sum_{j=i}^{i+k-1} arr[j] - k + k \times arr[i] \end{aligned}$$

我们令 $dphi[i] = \sum_{j=i}^{i+k-1} arr[j]$ ，则上述式子简化为：

$$dphi[i] - dphi[i - k + 1] + k \times arr[i]$$

我们维护一个单调栈，很容易求出元素 x 在左边第一个比它小的元素，即求出以 x 为最右且最小的子数组的最大长度，子数组的最小值之和为 $\sum_{j=i}^{i+k-1} arr[j]$ 。

具体解法如下：

- 从左向右遍历数组并维护一个单调递增栈，如果栈顶的元素大于等于当前元素 $arr[i]$ ，则弹出栈顶元素，同时将元素 $arr[i]$ 放在左边第一个比它小的子数组的头。
- 我们求出当前值为最右且最小的子数组的边长 k ，根据上面递推公式求出 $dphi[i]$ ，最终的返回值为 $\sum_{i=0}^{n-1} dphi[i]$ 。

解题入口 → 子数组 $[arr[1], arr[2], \dots, arr[n]]$ 的最小值

<https://leetcode/problems/sum-of-subarray-minimums/solutions/1929461/zhi-shu-zu-de-zui-xiao-zhi-zhi-he-by-leetcod...>

4. ② 前缀和+二分查找

为了使用二分查找，需要额外创建一个数组 $sums$ 用于存储数组 $nums$ 的前缀和，其中 $sums[i]$ 表示从 $nums[0]$ 到 $nums[i-1]$ 的元素和。得到前缀和之后，对于每个开始下标 i ，可通过二分查找找到大于或等于 i 的最小下标 $bound$ ，使得 $sums[bound] - sums[i-1] \geq s$ ，并更新子数组的最小长度（此时子数组的长度是 $bound - (i-1)$ ）。

因为这道题保证了数组中每个元素都为正，所以前缀和一定是递增的，这一点保证了二分的正确性。如果题目没有说明数组中每个元素都为正，这里就不能使用二分来查找这个位置了。

```
int minSubArrayLen(int target, vector<int>& nums) {
    int n = nums.size();
    if (n == 0) {return 0;}
    int ans = INT_MAX;
    vector<int> sums(n + 1, 0);
    // 为了方便计算，令 size = n + 1
    // sums[0] = 0 意味着前 0 个元素的前缀和为 0
    // sums[1] = A[0] 前 1 个元素的前缀和为 A[0]
    // 以此类推
    for (int i = 1; i <= n; i++) {sums[i] = sums[i - 1] + nums[i - 1];}
    for (int i = 1; i <= n; i++) {
        int presum = target + sums[i - 1];
        auto bound = lower_bound(sums.begin(), sums.end(), presum);
        if (bound != sums.end()) {
            int bias = (bound - sums.begin()) - i + 1;
            ans = min(ans, bias);
        }
    }
    return ans == INT_MAX ? 0 : ans;
}
```

8. ① 参考 7.① 方法 → 前缀和，由于 $nums$ 中元素不保证非负，故不可用二分查找，改用哈希表

```

int subarraySum(vector<int>& nums, int k) {
    int ans = 0;
    vector<int> sums(nums.size() + 1, 0);
    unordered_map<int, vector<int>> hash;
    for(int i = 1; i <= nums.size(); i++){
        sums[i] = sums[i-1] + nums[i-1];
        hash[sums[i]].emplace_back(i);
    }
    for(int i = 1; i <= nums.size(); i++){
        int presum = k + sums[i-1];
        if(hash.count(presum)){
            for(auto j:hash[presum]){
                if(j >= i) {ans++;}
            }
        }
    }
    return ans;
}

```

② 前缀和 + 哈希表 (优化)

```

int subarraySum(vector<int>& nums, int k) {
    unordered_map<int, int> hash;
    hash[0] = 1;
    int presum = 0, ans = 0;
    for(int num:nums){
        presum += num;
        if(hash.count(presum - k)) {ans += hash[presum - k];}
        hash[presum]++;
    }
    return ans;
}

```

1. 定义状态： $dp_{max}[i]$, $dp_{min}[i]$ → 以下标为 i 的元素作为结尾的最大/最小数相乘
 2. 状态转移方程：
 $dp_{max}[i] = \max(nums[i], dp_{max}[i-1] \times nums[i], dp_{min}[i-1] \times nums[i])$
 $dp_{min}[i] = \min(nums[i], dp_{max}[i-1] \times nums[i], dp_{min}[i-1] \times nums[i])$
 3. 初始值： $dp_{max}[0] = nums[0]$, $dp_{min}[0] = nums[0]$

```

int maxProduct(vector<int>& nums) {
    vector<int> dp_max(nums.size(), 0);
    vector<int> dp_min(nums.size(), 0);
    dp_max[0] = nums[0]; dp_min[0] = nums[0];
    int ans = INT_MIN;
    ans = max(ans, dp_max[0]);
    for(int i = 1; i < nums.size(); i++){
        int mul_max = dp_max[i-1] * nums[i];
        int mul_min = dp_min[i-1] * nums[i];
        dp_max[i] = max(nums[i], max(mul_max, mul_min));
        dp_min[i] = min(nums[i], min(mul_max, mul_min));
        ans = max(ans, dp_max[i]);
    }
    return ans;
}

```

二分+贪心

笔试常考题，用动规可能超出时间限制。

1. 1011. 在 D 天内送达包裹的能力：传送带上的包裹必须在 D 天内从一个港口运送到另一个港口。传送带上的第 i 个包裹的重量为 weights[i]。每一天，我们都会按给出重量的顺序往传送带上装载包裹。我们装载的重量不会超过船的最大运载重量。返回能在 D 天内将传送带上的所有包裹送达的船的最低运载能力。
2. LCP 12. 小张刷题计划：为了提高自己的代码能力，小张制定了 LeetCode 刷题计划，他选中了 LeetCode 题库中的 n 道题，编号从 0 到 n-1，并计划在 m 天内按照题目编号顺序刷完所有的题目（注意，小张不能用多天完成同一题）。在小张刷题计划中，小张需要用 time[i] 的时间完成编号 i 的题目。此外，小张还可以使用场外求助功能，通过询问他的好朋友小杨题目的解法，可以省去该题的做题时间。为了防止“小张刷题计划”变成“小杨刷题计划”，小张每天最多使用一次求助。我们定义 m 天中做题时间最多的一天耗时为 T（小杨完成的题目不计入做题总时间）。请你帮小张求出最小的 T 是多少。
3. 875. 爱吃香蕉的珂珂：珂珂喜欢吃香蕉。这里有 N 堆香蕉，第 i 堆中有 piles[i] 根香蕉。警卫已经离开了，将在 H 小时后回来。珂珂可以决定她吃香蕉的速度 K（单位：根/小时）。一个小时，她会选择一堆香蕉，从中吃掉 K 根。如果这堆香蕉少于 K 根，她将吃掉这堆的所有香蕉，然后这一小时内不会再吃更多的香蕉。珂珂喜欢慢慢吃，但仍然想在警卫回来前吃掉所有的香蕉。返回她可以在 H 小时内吃掉所有香蕉的最小速度 K（K 为整数）。
4. 1482. 制作 m 束花所需的最少天数：给你一个整数数组 bloomDay，以及两个整数 m 和 k。现需要制作 m 束花。制作花束时，需要使用花园中相邻的 k 朵花。花园中有 n 朵花，第 i 朵花会在 bloomDay[i] 时盛开，恰好可以用于一束花中。请你返回从花园中摘 m 束花需要等待的最少的天数。如果不能摘到 m 束花则返回 -1。

1. 贪心体现在对所有的范围为 [max_num, sum] 或重叠 考虑一次

假定「 D 天内运送完所有包裹的最低运力」为 ans ，那么在以 ans 为分割点的数据上具有「二段性」：

- 数值范围在 $(-\infty, ans]$ 的运力必然「不满足」 D 天内运送完所有包裹的要求
- 数值范围在 $[ans, +\infty)$ 的运力必然「满足」 D 天内运送完所有包裹的要求

即我们可以通过「二分」来找到恰好满足 D 天内运送完所有包裹的分割点 ans 。

接下来我们要确定二分的范围，由于不存在包裹拆分的情况，考虑如下两种边界情况：

- 理论最低运力：只确保所有包裹能够被运送，自然也包括重量最大的包裹，此时理论最低运力为 $\max_{i=1}^n$ 为数组 weights 中的最大值
- 理论最高运力：使得所有包裹在最短时间（一天）内运送完成，此时理论最高运力为 $\sum_{i=1}^n$ 为数组 weights 的总和

由此，我们可以确定二分的范围为 $[max, sum]$ 。

```
int shipWithinDays(vector<int>& weights, int days) {
    int max_num = INT_MIN, sum = 0;
    for(int& w:weights){
        max_num = max(max_num, w);
        sum += w;
    }
    int first = max_num, last = sum, mid = 0;
    while(first < last){
        mid = first + (last - first)/2;
        if(!check(weights, mid, days)) {first = mid + 1;} →此重  
无脑调大  
do...while  
码，使着重量大
        else {last = mid;}
    }
    return first;
}

bool check(vector<int>& weights, int load, int days){
    int size = weights.size();
    int cnt = 0;
    int i = 0, sum = 0;
    while(i < size){
        while(i < size && sum + weights[i] <= load){
            sum += weights[i];
            i++;
        }
        sum = 0; cnt++;
    }
    return (cnt <= days);
}
```

题意概述

给定一个数组，将其划分成 M 份，使得每份元素之和最大值最小，每份可以任意减去其中一个元素。

题解

如果不考虑每份可以任意减去一个元素，就是一个经典的二分问题，具有单调最优的性质：如果最大值为 i 可以满足条件划分，那么最大值为 $i+1$ 也可以。所以就直接二分最大值 i ，找到最小满足条件的 i 即可。

本题加了一个条件：每份可以删除任意一个数组。为了能够让最大值最小，显然每份中删去的一定是最大元素，所以在二分判定可行性时，维护当前序列的最大值，然后记录删除最大值的结果，也就是说二分的判定是：是否可以每组删除最大值之后，总和都小于等于 i 。

```
int minTime(vector<int>& time, int m) {
    int sum = 0;
    for(int& t:time) {sum += t;}
    int first = 0, last = sum, mid = 0;
    while(first < last){
        mid = first + (last - first)/2;
        if(!check(time, mid, m)) {first = mid + 1;}
        else {last = mid;}
    }
    return first;
}

bool check(vector<int>& time, int load, int m){
    int cnt = 1;
    int sum = 0;
    int max_time = time[0];
    for(int i = 1; i < time.size(); i++){
        if(sum + min(max_time, time[i]) <= load){
            sum += min(max_time, time[i]);
            max_time = max(max_time, time[i]);
        }
        else{
            cnt++;
            sum = 0;
            max_time = time[i];
        }
    }
    return (cnt <= m);
}
```

3.

```

int minEatingSpeed(vector<int>& piles, int h) {
    int max_speed = INT_MIN;
    for(int& p:piles) {max_speed = max(max_speed, p);}
    int first = 1, last = max_speed, mid = 0;
    while(first < last){
        mid = first + (last - first)/2;
        if(!check(piles, mid, h)) {first = mid + 1;}
        else {last = mid;}
    }
    return first;
}

bool check(vector<int>& piles, int speed, int h){
    int cnt = 0;
    for(int i = 0; i < piles.size(); i++){
        if(piles[i] % speed) {cnt++;}
        cnt += piles[i]/speed;
    }
    return (cnt <= h);
}

```

4.

```

int minDays(vector<int>& bloomDay, int m, int k) {
    if((long long)m * k > bloomDay.size()) {return -1;}
    int min_wait = INT_MAX, max_wait = INT_MIN;
    for(int& b:bloomDay){
        min_wait = min(min_wait, b);
        max_wait = max(max_wait, b);
    }
    int first = min_wait, last = max_wait, mid = 0;
    while(first < last){
        mid = first + (last - first)/2;
        if(!check(bloomDay, mid, m, k)) {first = mid + 1;}
        else {last = mid;}
    }
    return first;
}

bool check(vector<int>& bloomDay, int load, int m, int k){
    int cnt = 0, cnt_k = 0;
    for(int i = 0; i < bloomDay.size(); i++){
        if(bloomDay[i] <= load){
            cnt_k++;
            if(cnt_k == k) {cnt++; cnt_k = 0;}
        }
        else {cnt_k = 0;}
    }
    return (cnt >= m);
}

```

11. 动态规划 \Rightarrow 不断迭代地决策求最优解的过程

背包问题

一共有 9 例

https://blog.csdn.net/weixin_42638946/article/details/114028588

① 0-1 背包问题

有 N 件物品和一个容量是 V 的背包。每件物品只能使用一次。

第 i 件物品的体积是 v_i , 价值是 w_i 。

求解将哪些物品装入背包，可使这些物品的总体积不超过背包容量，且总价值最大。
输出最大价值。

输入格式

第一行两个整数, N , V , 用空格隔开, 分别表示物品数量和背包容积。

接下来有 N 行, 每行两个整数 v_i, w_i , 用空格隔开, 分别表示第 i 件物品的体积和价值。

输出格式

输出一个整数, 表示最大价值。

题解与原题:

<https://www.acwing.com/solution/content/1374/>

只有选和不选某物品的考虑, 故为 0-1

2.1 版本 1 二维

(1) 状态 $f[i][j]$ 定义: 前 i 个物品, 背包容量 j 下的最优解 (最大价值) :

• 前的状态依赖于之前的状态, 可以理解为从初始状态 $f[0][0] = 0$ 开始决策, 有 N 件物品, 则需要 N 次决策, 每次对第 i 件物品的决策, 状态 $f[i][j]$ 不断由之前的状念更新而来。

(2) 当前背包容量不够 ($j < v[i]$), 没得选, 因此前 i 个物品最优解即为前 $i - 1$ 个物品最优解:

• 对应代码: $f[i][j] = f[i - 1][j]$.

(3) 当前背包容量够, 可以选, 因此需要决策选与不选第 i 个物品:

• 选: $f[i][j] = f[i - 1][j - v[i]] + w[i]$.

• 不选: $f[i][j] = f[i - 1][j]$.

• 我们的决策该如何取最大值, 因此以上两种情况取 $\max()$.

```
#include<bits/stdc++.h>
using namespace std;

int main(){
    int n, m;
    cin>>n>>m;
    vector<int> v(n+1); //体积
    vector<int> w(n+1); //价值
    vector<vector<int>> dp(n+1, vector<int>(m+1)); //dp数组
    for(int i = 1; i <= n; i++){
        for(int j = 0; j <= m; j++){
            if(j < v[i]) dp[i][j] = dp[i-1][j]; //当前背包容量装不进第i个物品
            else dp[i][j] = max(dp[i-1][j], dp[i-1][j-v[i]]+w[i]); //能装, 需进行决策是否选择第i个物品
        }
    }
    cout<<dp[n][m]<<endl;
    return 0;
}
```

2.2 版本 2 一维

将状态 $f[i][j]$ 优化到一维 $f[j]$, 实际上只需要做一个等价变形。

为什么可以这样变形呢? 我们定义的状态 $f[i][j]$ 可以求得任意合法的 i 与 j 最优解, 但题目只要求得最终状态 $f[n][m]$, 因此我们只需要一维的空间来更新状态。

(1) 状态 $f[j]$ 定义: N 件物品, 背包容量 j 下的最优解。

(2) 注意枚举背包容量 j 必须从 0 开始。

(3) 为什么一维情况下枚举背包容量需要逆序? 在二维情况下, 状态 $f[i][j]$ 是由上一轮 $i - 1$ 的状态得来的, $f[i][j]$ 与 $f[i - 1][j]$ 是独立的, 而优化到一维后, 如果我们还是正序, 则有 $f[\text{较小体积}]$ 更新到 $f[\text{较大体积}]$, 则有可能本应该用第 $i - 1$ 轮的状态却用的是第 i 轮的状态。

(4) 例如, 一维状态第 i 轮对体积为 3 的物品进行决策, 则 $f[7]$ 由 $f[4]$ 更新而来, 这里的 $f[4]$ 正确应该是 $f[1 - 1][4]$, 但小到大枚举, 这里的 $f[4]$ 在第 i 轮计算却变成了 $f[1][4]$, 当逆序枚举背包容量 j 时, 我们从 $f[7]$ 同样由 $f[4]$ 更新, 但由于是逆序, 这里的 $f[4]$ 还没有在第 i 轮计算, 所以此时实际计算的 $f[4]$ 仍然是 $f[1 - 1][4]$ 。

(5) 简单来说, 一维情况正序更新状态 $f[j]$ 需要用到前面计算的状态已经被「污染」, 逆序则不会有这样的问题。

状态转移方程为: $f[j] = \max(f[j], f[j - v[i]] + w[i])$.

```
for(int i = 1; i <= n; i++)
    for(int j = m; j >= 0; j--){
        {
            if(j < v[i])
                f[i][j] = f[i - 1][j]; // 优化前
            f[j] = f[j]; // 优化后, 该行自动成立, 可省略。
        }
        else
            f[i][j] = max(f[i - 1][j], f[i - 1][j - v[i]] + w[i]); // 优化前
            f[j] = max(f[j], f[j - v[i]] + w[i]); // 优化后
    }
}
```

实际上, 只有当枚举的背包容量 $>= v[i]$ 时才会更新状态, 因此我们可以修改循环终止条件进一步优化。

```
#include<bits/stdc++.h>
using namespace std;

int main(){
    int n, m;
    cin>>n>>m;
    vector<int> v(n+1); //体积
    vector<int> w(n+1); //价值
    vector<int> dp(m+1); //dp数组
    for(int i = 1; i <= n; i++){
        for(int j = m; j >= v[i]; j--){
            dp[j] = max(dp[j], dp[j-v[i]] + w[i]);
        }
    }
    cout<<dp[m]<<endl;
    return 0;
}
```

力扣中的变型题

1. 416. 分割等和子集：给定一个只包含正整数的非空数组。是否可以将这个数组分割成两个子集，使得两个子集的元素和相等。本题推荐题解是：[@liweiwei1419的动态规划（转换为 0-1 背包问题）](#)
2. 494. 目标和：给定一个非负整数数组， a_1, a_2, \dots, a_n ，和一个目标数， S 。现在你有两个符号 $+$ 和 $-$ 。对于数组中的任意一个整数，你都可以从 $+$ 或 $-$ 中选择一个符号添加在前面。
3. 1049. 最后一块石头的重量 II：有一堆石头，每块石头的重量都是正整数。每一回合，从中选出任意两块石头，然后将它们一起粉碎。假设石头的重量分别为 x 和 y ，且 $x \leq y$ 。那么粉碎的可能结果如下：如果 $x == y$ ，那么两块石头都会被完全粉碎；如果 $x != y$ ，那么重量为 x 的石头将会完全粉碎，而重量为 y 的石头新重量为 $y-x$ 。最后，最多只会剩下一块石头。返回此石头最小的可能重量。如果没有石头剩下，就返回0。

1.

转换为「0 - 1」背包问题

这道题目是我学习「背包」问题的入门问题，做这道题需要做一个等价转换：是否可以从输入数组中挑选出一些正整数，使得这些数的和等于整个数组元素的和的一半。很坦白地说，如果是我我的老师告诉我可以这样想，我很难想出来。容易知道：数组的和一定是偶数。

本题与0-1背包问题有一个很大的不同，即：

- 0-1背包问题选取的物品的容积总量不能超过规定的总量；
- 本题选取的数字之和需要恰好等于规定的和的一半。

dp为int型

二维

一维

```
bool canPartition(vector<int>& nums) {
    int sum = 0;
    for(int& num:nums) {sum += num;}
    if(sum % 2) {return false;}
    int n = nums.size(), m = sum/2;
    vector<vector<int>> dp(n+1, vector<int>(m+1, 0));
    for(int i = 1; i <= n; i++){
        for(int j = 1; j <= m; j++){
            if(j < nums[i-1]) {dp[i][j] = dp[i-1][j];}
            else {dp[i][j] = max(dp[i-1][j], dp[i-1][j-nums[i-1]]+nums[i-1]);}
        }
    }
    if(dp[n][m] == m) {return true;}
    else {return false;}
}
```

```
bool canPartition(vector<int>& nums) {
    int sum = 0;
    for(int& num:nums) {sum += num;}
    if(sum % 2) {return false;}
    int n = nums.size(), m = sum/2;
    vector<int> dp(m+1, 0);
    for(int i = 1; i <= n; i++){
        for(int j = m; j >= nums[i-1]; j--){
            dp[j] = max(dp[j], dp[j-nums[i-1]]+nums[i-1]);
        }
    }
    if(dp[m] == m) {return true;}
    else {return false;}
}
```

dp为bool型

二维

一维

此题
V就是W
别为Nums

```
bool canPartition(vector<int>& nums) {
    int sum = 0;
    for(int& num:nums) {sum += num;}
    if(sum % 2) {return false;}
    int n = nums.size(), m = sum/2;
    vector<vector<bool>> dp(n+1, vector<bool>(m+1, false));
    for(int i = 1; i <= n; i++){
        for(int j = 1; j <= m; j++){
            if(j < nums[i-1]) {dp[i][j] = dp[i-1][j];}
            else if(j == nums[i-1]) {dp[i][j] = true; continue;}
            else {dp[i][j] = dp[i-1][j] || dp[i-1][j - nums[i-1]];}
        }
    }
    return dp[n][m];
}
```

```
bool canPartition(vector<int>& nums) {
    int sum = 0;
    for(int& num:nums) {sum += num;}
    if(sum % 2) {return false;}
    int n = nums.size(), m = sum/2;
    vector<bool> dp(m+1, false);
    dp[0] = true;
    for(int i = 1; i <= n; i++){
        for(int j = m; j >= nums[i-1]; j--){
            dp[j] = dp[j] || dp[j - nums[i-1]];
        }
    }
    return dp[m];
}
```

2.

考虑整下target，二维

$f[i][j]$ 代表考虑前 i 个数，当前计算结果为 j 的方案数，令 nums 下标从1开始。

那么 $f[n][target]$ 为最终答案， $f[0][0] = 1$ 为初始条件：代表不考虑任何数，凑出计算结果为0的方案数为1种。

根据每个数值只能搭配 $+/ -$ 使用，可得状态转移方程：

$$f[i][j] = f[i-1][j - \text{nums}[i-1]] + f[i-1][j + \text{nums}[i-1]]$$

到这里，既有了「状态定义」和「转移方程」，又有了可以滚动下去的「有效值」（起始条件）。

距离我们完成所有分析还差最后一步。

当使用递推形式时，我们通常会使用「静态数组」来存储动规值，因此还需要考虑维度范围的：

- 第一维为物品数量：范围为 nums 数组长度
- 第二维为中间结果：令 s 为所有 nums 元素的总和（题目给定了 $\text{nums}[i]$ 为非负数的条件，否则需要对 $\text{nums}[1]$ 取绝对值再累加），那么中间结果的范围为 $[-s, s]$

```
int findTargetSumWays(vector<int>& nums, int target) {
    int sum = 0;
    for(int& num:nums) {sum += num;}
    if(abs(target) > sum) {return 0;}
    int n = nums.size(), m = sum;
    vector<vector<int>> dp(n+1, vector<int>(2*m+1, 0));
    dp[0][0] = 1;
    for(int i = 1; i <= n; i++){
        for(int j = -m; j <= m; j++){
            if(j - nums[i-1] >= -m) {dp[i][j+m] += dp[i-1][(j-nums[i-1])+m];}
            if(j + nums[i-1] <= m) {dp[i][j+m] += dp[i-1][(j+nums[i-1])+m];}
        }
    }
    return dp[n][target+m];
}
```

只考虑 target 的非负值部分，二维

```
int findTargetSumWays(vector<int>& nums, int target) {
    int sum = 0;
    for(int& num:nums) {sum += num;}
    if(abs(target) > sum) {return 0;}
    if((sum + target) % 2) {return 0;}
    int n = nums.size(), m = (sum + target)/2;
    vector<vector<int>> dp(n+1, vector<int>(m+1, 0));
    dp[0][0] = 1;
    for(int i = 1; i <= n; i++){
        for(int j = 0; j <= m; j++){
            dp[i][j] += dp[i-1][j];
            if(j >= nums[i-1]) {dp[i][j] += dp[i-1][j-nums[i-1]];}
        }
    }
    return dp[n][m];
}
```

设 target 非负值部分为 Pos, 负值部分为 Neg, 则

$$\text{target} = \text{Pos} + \text{Neg}$$

$$\text{sum} = \text{Pos} - \text{Neg}$$

$$\text{Pos} = \frac{\text{target} + \text{sum}}{2}$$

只考虑 target 的非负值部分，一维

```
int findTargetSumWays(vector<int>& nums, int target) {
    int sum = 0;
    for(int& num:nums) {sum += num;}
    if(abs(target) > sum) {return 0;}
    if((sum + target) % 2) {return 0;}
    int n = nums.size(), m = (sum + target)/2;
    vector<int> dp(m+1, 0);
    dp[0] = 1;
    for(int i = 1; i <= n; i++){
        for(int j = m; j >= nums[i-1]; j--){
            dp[j] += dp[j-nums[i-1]];
        }
    }
    return dp[m];
}
```

3、 $\text{dp}[j] \rightarrow$ 容许最大重量为 target 时, 考虑前 i-1 块石头可达到的最大重量

```
int lastStoneWeightIII(vector<int>& stones) {
    int sum = 0;
    for(int& s:stones) {sum += s;}
    int n = stones.size(), m = sum/2;
    vector<int> dp(m+1);
    for(int i = 0; i < n; i++){
        for(int j = m; j >= stones[i]; j--){
            dp[j] = max(dp[j], dp[j-stones[i]]+stones[i]);
        }
    }
    return sum - dp[m] - dp[m];
}
```

②完全背包问题

有 N 种物品和一个容量是 V 的背包，每种物品都有无限件可用。

第 i 种物品的体积是 v_i ，价值是 w_i 。

求解将哪些物品装入背包，可使这些物品的总体积不超过背包容量，且总价值最大。

输出最大价值。

输入格式

第一行两个整数， N ， V ，用空格隔开，分别表示物品种数和背包容积。

接下来有 N 行，每行两个整数 v_i, w_i ，用空格隔开，分别表示第 i 种物品的体积和价值。

输出格式

输出一个整数，表示最大价值。

原始思路（会超时）

```
#include<bits/stdc++.h>
using namespace std;

int main(){
    int n, m;
    cin>>n>>m;
    vector<int> v(n+1); //体积
    vector<int> w(n+1); //价值
    vector<vector<int>> dp(n+1, vector<int>(m+1)); //dp数组
    for(int i = 1; i <= n; i++){
        for(int j = 1; j <= m; j++){
            for(int k = 0; k <= v[i]; k++){
                dp[i][j] = max(dp[i][j], dp[i-1][j-k+v[i]]+k*w[i]);
            }
        }
    }
    cout<<dp[n][m]<<endl;
    return 0;
}
```

二维

```
#include<bits/stdc++.h>
using namespace std;

int main(){
    int n, m;
    cin>>n>>m;
    vector<int> v(n+1); //体积
    vector<int> w(n+1); //价值
    vector<vector<int>> dp(n+1, vector<int>(m+1)); //dp数组
    for(int i = 1; i <= n; i++) {cin>>v[i]>>w[i];
    for(int i = 1; i <= n; i++){
        for(int j = 1; j <= m; j++){
            if(j < v[i]) {dp[i][j] = dp[i-1][j];
            else {dp[i][j] = max(dp[i-1][j], dp[i][j-v[i]]+w[i]);}
        }
    }
    cout<<dp[n][m]<<endl;
    return 0;
}
```

一维

```
#include<bits/stdc++.h>
using namespace std;

int main(){
    int n, m;
    cin>>n>>m;
    vector<int> v(n+1); //体积
    vector<int> w(n+1); //价值
    vector<int> dp(m+1); //dp数组
    for(int i = 1; i <= n; i++) {cin>>v[i]>>w[i];
    for(int i = 1; i <= n; i++){
        for(int j = v[i]; j <= m; j++){
            dp[j] = max(dp[j], dp[j-v[i]]+w[i]);
        }
    }
    cout<<dp[m]<<endl;
    return 0;
}
```

原题与题解：

<https://www.acwing.com/solution/content/5345/>

⇒ 每种物品可选择无限次

$dp[i][j]$ → 前 i 种物品，背包容量为 j 下的最优解（最大值）

我们列举一下更新次序的内部关系：

$\rightarrow f[i][j-v[i]] + w[i]$

$f[i][j] = \max(f[i-1][j], f[i-1][j-v[i]] + w[i], f[i-1][j-2*v[i]] + 2*w[i], f[i-1][j-3*v[i]] + 3*w[i], \dots)$
 $f[i][j-v[i]] = \max(f[i-1][j-v[i]], f[i-1][j-2*v[i]] + w[i], f[i-1][j-3*v[i]] + 2*w[i], \dots)$

由上两式，可得出如下递推关系：

$$f[i][j] = \max(f[i][j-v[i]] + w[i], f[i-1][j])$$

$dp[j]$ → n 种物品，背包容量为 j 下的最优解

你会发现，这个代码与 01 背包的代码只有 01 的循环次序不同而已。为什么这样改就可行呢？首先想想为什么 01 背包中要按照 $v = V..0$ 的逆序来循环。这是因为要保证第 i 次循环中的状态 $f[i][v]$ 是由状态 $f[i-1][v - c[i]]$ 递推而来。换句话说，这正是为了保证每件物品只选一次，保证在考虑“选入第 i 件物品”这件策略时，依据的是一个绝无已经选入第 i 件物品的子结果 $f[i-1][v - c[i]]$ 。而现在完全背包的特点恰是每种物品可选无限件，所以在考虑“加选一件第 i 种物品”这种策略时，却正需要一个可能已选入第 i 件物品的子结果 $f[i][v - c[i]]$ ，所以就可以，并且必须采用 $v = 0..V$ 的顺序循环。这就是这个简单的程序为何成立的道理。

力扣中的变型题

322. 零钱兑换：给定不同面额的硬币 coins 和一个总金额 amount。编写一个函数来计算可以凑成总金额所需的最少的硬币个数。如果没有任何一种硬币组合能组成总金额，返回 -1。
518. 零钱兑换 II：给定不同面额的硬币和一个总金额。写出函数来计算可以凑成总金额的硬币组合数。假设每一种面额的硬币有无限个。
377. 组合总和 IV：给定一个由正整数组成且不存在重复数字的数组，找出和为给定目标正整数的组合的个数。
- 面试题 08.11. 硬币：硬币。给定数量不限的硬币，币值为25分、10分、5分和1分，编写代码计算n分有几种表示法。(结果可能会很大，你需要将结果模上1000000007)

1. 二维

```
int coinChange(vector<int>& coins, int amount) {
    int n = coins.size(), m = amount;
    vector<vector<int>> dp(n+1, vector<int>(m+1, 1000000000)); //不用INT_MAX是因为下面操作+1会溢出
    for(int i = 0; i <= n; i++) {dp[i][0] = 0;} → 这里初始化
    for(int i = 1; i <= n; i++){
        for(int j = 1; j <= m; j++){
            if(j < coins[i-1]) {dp[i][j] = dp[i-1][j];}
            else {dp[i][j] = min(dp[i-1][j], dp[i][j-coins[i-1]] + 1);}
        }
    }
    return dp[n][m] == 1000000000 ? -1 : dp[n][m];
}
```

使用 INT_MAX

```
int coinChange(vector<int>& coins, int amount) {
    int n = coins.size(), m = amount;
    vector<vector<int>> dp(n+1, vector<int>(m+1, INT_MAX));
    for(int i = 0; i <= n; i++) {dp[i][0] = 1;}
    for(int i = 1; i <= n; i++){
        for(int j = 1; j <= m; j++){
            if(j < coins[i-1] || dp[i-1][j-coins[i-1]] == INT_MAX) {dp[i][j] = dp[i-1][j];}
            else {dp[i][j] = min(dp[i-1][j], dp[i][j-coins[i-1]] + 1);}
        }
    }
    return dp[n][m] == INT_MAX ? -1 : dp[n][m];
}
```

-维

```
int coinChange(vector<int>& coins, int amount) {
    int n = coins.size(), m = amount;
    vector<int> dp(m+1, 1000000000); //不用INT_MAX是因为下面操作+1会溢出
    dp[0] = 0;
    for(int i = 1; i <= n; i++){
        for(int j = coins[i-1]; j <= m; j++){
            dp[j] = min(dp[j], dp[j-coins[i-1]] + 1);
        }
    }
    return dp[m] == 1000000000 ? -1 : dp[m];
}
```

```
int coinChange(vector<int>& coins, int amount) {
    int n = coins.size(), m = amount;
    vector<int> dp(m+1, INT_MAX);
    dp[0] = 0;
    for(int i = 1; i <= n; i++){
        for(int j = coins[i-1]; j <= m; j++){
            if(dp[j-coins[i-1]] != INT_MAX) {dp[j] = min(dp[j], dp[j-coins[i-1]] + 1);}
        }
    }
    return dp[m] == INT_MAX ? -1 : dp[m];
}
```

2. 二维 dp[i][j] → 前 i 种硬币下，金额恰好为 j 的旗数

```
int change(int amount, vector<int>& coins) {
    int n = coins.size(), m = amount;
    vector<vector<int>> dp(n+1, vector<int>(m+1, 0));
    for(int i = 0; i <= n; i++) {dp[i][0] = 1;} ←
    for(int i = 1; i <= n; i++){
        for(int j = 1; j <= m; j++){
            dp[i][j] ← dp[i-1][j];
            if(j >= coins[i-1]) {dp[i][j] ← dp[i][j] + dp[i][j-coins[i-1]];}
        }
    }
    return dp[n][m];
}
```

一维

```
int change(int amount, vector<int>& coins) {
    int n = coins.size(), m = amount;
    vector<int> dp(m+1);
    dp[0] = 1;
    for(int i = 1; i <= n; i++){
        for(int j = coins[i-1]; j <= m; j++){
            dp[j] ← dp[j] + dp[j-coins[i-1]];
        }
    }
    return dp[m];
}
```

题目要求

dp初值

dp首端值

for中操作

最大值

0

dp[0]=0

max(-,-)

最次数/旗数

0

dp[0]=1

+=

最少需要物品

MAX_INT/1000000000

dp[0]=0

min(-,-)

判断题可行

false

dp[0]=true

||

3. 如果求组合数就是外层for循环遍历物品，内层for遍历背包。{ }

如果求排列数就是外层for遍历背包，内层for循环遍历物品。

如果把遍历nums（物品）放在外循环，遍历target的作用为内循环的话，举一个例子：计算dp[4]的时候，结果集只有{1,3}这样的集合，不会有{3,1}这样的集合，因为nums遍历放在外层，3只能出现在1后面！

所以本题遍历顺序最终遍历顺序：target（背包）放在外循环，将nums（物品）放在内循环，内循环从前到后遍历。

二维

```
int combinationSum4(vector<int>& nums, int target) {  
    int n = nums.size(), m = target;  
    vector<vector<int>> dp(n+1, vector<int>(m+1));  
    for(int i = 0; i <= n; i++) {dp[i][0] = 1;}  
    for(int j = 1; j <= m; j++){  
        for(int i = 1; i <= n; i++){  
            dp[i][j] += dp[i-1][j];  
            if(j >= nums[i-1] && dp[i][j] < INT_MAX - dp[n][j-nums[i-1]]) {  
                dp[i][j] += dp[n][j-nums[i-1]];  
            }  
        }  
    }  
    return dp[n][m];  
}
```

dp[i][j] → 前i种数字下，和恰为j的方案数

一维

```
int combinationSum4(vector<int>& nums, int target) {  
    int n = nums.size(), m = target;  
    vector<int> dp(m+1, 0);  
    dp[0] = 1;  
    for (int i = 1; i <= m; i++) { // 遍历背包  
        for (int j = 1; j <= n; j++) { // 遍历物品  
            // 测试用例有两个数相加超过1int的数据，所以在if里加上dp[j]<INT_MAX-dp[j-num]  
            if (j >= nums[i-1] && dp[j] < INT_MAX - dp[j-nums[i-1]]) {  
                dp[i] += dp[j-nums[i-1]];  
            }  
        }  
    }  
    return dp[m];  
}
```

4. 与题2相同

二维 dp[i][j] → 前i种货币下，金额恰为j的方案数

```
int waysToChange(int n) {  
    vector<int> coins = {25, 10, 5, 1};  
    int m = n;  
    vector<vector<int>> dp(coins.size()+1, vector<int>(m+1));  
    for(int i = 0; i <= coins.size(); i++) {dp[i][0] = 1;}  
    for(int i = 1; i <= coins.size(); i++){  
        for(int j = 1; j <= m; j++){  
            dp[i][j] += dp[i-1][j];  
            if(j >= coins[i-1] && dp[i][j] < INT_MAX - dp[i][j-coins[i-1]]){  
                dp[i][j] = (dp[i][j] + dp[i][j-coins[i-1]]) % 1000000007;  
            }  
        }  
    }  
    return dp[coins.size()][m];  
}
```

题目要求

一维

```
int waysToChange(int n) {  
    vector<int> coins = {25, 10, 5, 1};  
    int m = n;  
    vector<int> dp(m+1);  
    dp[0] = 1;  
    for(int i = 1; i <= coins.size(); i++){  
        for(int j = coins[i-1]; j <= m; j++){  
            dp[j] = (dp[j] + dp[j-coins[i-1]]) % 1000000007;  
        }  
    }  
    return dp[m];  
}
```

③ 多重背包问题

有 N 种物品和一个容量是 V 的背包。

第 i 种物品最多有 s_i 件，每件体积是 v_i ，价值是 w_i 。

求解将哪些物品装入背包，可使物品体积总和不超过背包容量，且价值总和最大。

输出最大价值。

输入格式

第一行两个整数， N, V ，用空格隔开，分别表示物品种类数和背包容量。

接下来有 N 行，每行三个整数 v_i, w_i, s_i ，用空格隔开，分别表示第 i 种物品的体积、价值和数量。

输出格式

输出一个整数，表示最大价值。

数据范围

$0 < N, V \leq 100$

$0 < v_i, w_i, s_i \leq 100$

原题与题解：

<https://www.acwing.com/solution/content/17554/>

⇒ 每种物品的选择有数量限制

二维(看作是有数量限制的完全背包问题)

```
#include <bits/stdc++.h>
using namespace std;

int main(){
    int n, m;
    cin >> n >> m;
    vector<int> v(n+1); //体积
    vector<int> w(n+1); //价值
    vector<int> s(n+1); //数量
    vector<vector<int>> dp(n+1, vector<int>(m+1)); //dp数组
    for(int i = 1; i <= n; i++) {cin >> v[i] >> w[i] >> s[i];}
    for(int i = 1; i <= n; i++){
        for(int j = 1; j <= m; j++){
            for(int k = 0; k <= s[i] && k*v[i] <= j; k++){
                dp[i][j] = max(dp[i][j], dp[i-1][j-k*v[i]]+k*w[i]);
            }
        }
    }
    cout << dp[n][m] << endl;
    return 0;
}
```

⇒ $dp[i][j]$: 前 i 种物品下，容量为 j 时的最大价值

一维(看作是把同种物品拆分为一个整体的01背包问题)

```
#include <bits/stdc++.h>
using namespace std;

int main(){
    int n, m;
    cin >> n >> m;
    vector<int> v(n+1); //体积
    vector<int> w(n+1); //价值
    vector<int> s(n+1); //数量
    vector<int> dp(m+1); //dp数组
    for(int i = 1; i <= n; i++) {cin >> v[i] >> w[i] >> s[i];}
    for(int i = 1; i <= n; i++){
        for(int j = m; j >= v[i]; j--){
            for(int k = 1; k <= s[i] && k*v[i] <= j; k++){
                dp[j] = max(dp[j], dp[j-k*v[i]]+k*w[i]);
            }
        }
    }
    cout << dp[m] << endl;
    return 0;
}
```

⇒ $dp[j]$: 带数量限制的 n 种物品下，容量为 j 的最大价值

二进制优化法(先拆分每种物品，后看作01背包问题) \Rightarrow 适用于数据量较大的问题

<https://www.acwing.com/solution/content/20115/>

$f[i, j] = \max\{f[i-1, j], f[i-1, j-v] + w, f[i-1, j-2v] + 2w, f[i-1, j-3v] + 3w, \dots\}$
 $f[i, j-v] = \max\{f[i-1, j], f[i-1, j-2v] + w, f[i-1, j-3v] + 2w, \dots\}$
 通过上述比例，可以得到 $f[i][j] = \max\{f[i-1][j], f[i][j-v] + w\}$

再来看下多重背包。

$f[i, j] = \max\{f[i-1, j], f[i-1, j-v] + w, f[i-1, j-2v] + 2w, \dots, f[i-1, j-Sv] + Sw\}$
 $f[i, j-v] = \max\{f[i-1, j-v], f[i-1, j-2v] + w, \dots, f[i-1, j-Sv] + (S-1)w\}$
 $f[i-1, j-(S+1)v] + Sw$

怎么比完全背包问题比较就多出了一项？

其实，一般从实际意义出发来考虑即可。这是在分析 $f[i, j-w]$ 这个状态的表达式，首先这个状态的含义是：从前 i 个物品中选，且总体积不超过 w 的最大价值。我们现在最多只能选 i 个物品，因此如果我们选 i 个第 j 物品，那么体积上就要减去 $s * v$ ，价值上就要加上 $s * w$ ，那更新到状态中去就是 $f[i-1, j-v] + s * w$

那为什么完全背包不会有多重最后一项？

完全背包由于对每种物品没有选择个数的限制，所以只要体积够用就可以一直选，没有最后一项。

二维

```
#include <bits/stdc++.h>
using namespace std;

int main(){
    int n, m;
    cin >> n >> m;
    vector<int> v; //体积
    vector<int> w; //价值
    v.emplace_back(0); w.emplace_back(0); //首项初始化
    for(int i = 1; i <= n; i++){
        int v_raw, w_raw, s_raw;
        cin >> v_raw >> w_raw >> s_raw;
        int k = 1;
        while(k <= s_raw){ //二进制优化拆分
            v.emplace_back(v_raw * k);
            w.emplace_back(w_raw * k);
            s_raw -= k;
            k *= 2;
        }
        if(s_raw > 0){ //剩余的一组
            v.emplace_back(v_raw * s_raw);
            w.emplace_back(w_raw * s_raw);
        }
    }
    v = v.size() - 1;
    vector<vector<int>> dp(m+1, vector<int>(m+1)); //dp数组
    for(int i = 1; i <= n; i++){
        for(int j = 1; j <= m; j++){
            if(j < v[i]) {dp[i][j] = dp[i-1][j];}
            else {dp[i][j] = max(dp[i-1][j], dp[i-1][j-v[i]]+w[i]);}
        }
    }
    cout << dp[n][m] << endl;
    return 0;
}
```

\Rightarrow 故何像先拆分
并优先转为方
程

二进制优化思维就是：现在给出一堆苹果和10个箱子，选出n个苹果。将这一堆苹果分别按照1, 2, 4, 8, 16, ..., 512分到10个箱子里面。那么由于任何一个数字 $x \in [0, 1023]$ (第11个箱子才能取到1024)，评论区有讨论这个都可以从这10个箱子里的苹果数量表示出来，但是这样选择的次数就是 ≤ 10 次。

比如：

- 如果要拿1001个苹果，传统就是要拿1001次；二进制的思维，就是拿7个箱子就行（分别是装有512、256、128、64、32、8、1个苹果的这7个箱子）。这样一来，1001次操作就变成7次操作就行了。

这样利用二进制优化，时间复杂度就从 $O(n^3)$ 降到 $O(n^2 \log S)$ ，从 $4 * 10^9$ 到 $2 * 10^7$ 。

一维

```
#include <bits/stdc++.h>
using namespace std;

int main(){
    int n, m;
    cin >> n >> m;
    vector<int> v; //体积
    vector<int> w; //价值
    v.emplace_back(0); w.emplace_back(0); //首项初始化
    for(int i = 1; i <= n; i++){
        int v_raw, w_raw, s_raw;
        cin >> v_raw >> w_raw >> s_raw;
        int k = 1;
        while(k <= s_raw){ //二进制优化拆分
            v.emplace_back(v_raw * k);
            w.emplace_back(w_raw * k);
            s_raw -= k;
            k *= 2;
        }
        if(s_raw > 0){ //剩余的一组
            v.emplace_back(v_raw * s_raw);
            w.emplace_back(w_raw * s_raw);
        }
    }
    n = v.size() - 1;
    vector<int> dp(m+1); //dp数组
    for(int i = 1; i <= n; i++){
        for(int j = m; j >= v[i]; j--){
            dp[j] = max(dp[j], dp[j-v[i]]+w[i]);
        }
    }
    cout << dp[m] << endl;
    return 0;
}
```

拆分后
 \Downarrow
化为01背包问题

④ 混合背包问题

有 N 种物品和一个容量是 V 的背包。

物品一共有三类：

- 第一类物品只能用 1 次（01 背包）；
- 第二类物品可以用无限次（完全背包）；
- 第三类物品最多使用 s_i 次（多重背包）；

每种体积是 v_i ，价值是 w_i 。

求解将哪些物品装入背包，可使物品体积总和不超过背包容量，且价值总和最大。

输出最大值。

输入格式

第一行两个整数 N, V ，用空格隔开，分别表示物品种类数和背包容量。

接下来有 N 行，每行三个整数 v_i, w_i, s_i ，用空格隔开，分别表示第 i 种物品的体积、价值和数量。

- $s_i = -1$ 表示第 i 种物品只能用 1 次；
- $s_i = 0$ 表示第 i 种物品可以用无限次；
- $s_i > 0$ 表示第 i 种物品可以使用 s_i 次；

输出格式

输出一个整数，表示最大价值。

→ 上述三种类型的物品都有

原题与题解：

<https://www.acwing.com/solution/content/23633/>

三种类型的物品都转为确切数量： $|01 \Rightarrow s[i] = 1|$



$|完全 \Rightarrow s[i] = m / v[i]|$

$|多重 \Rightarrow s[i]|$

转化为多重背包问题

01. 完全背包问题本质都可以转化为多重背包问题

二进制优化

```
#include <bits/stdc++.h>
using namespace std;

int main(){
    int n, m;
    cin>>n>>m;
    vector<int> v(n+1); //体积
    vector<int> w(n+1); //价值
    vector<int> s(n+1); //数量
    vector<int> dp(m+1); //dp数组
    for(int i = 1; i <= n; i++){
        cin>>v[i]>>w[i]>>s[i];
        if(s[i] == -1) {s[i] = 1;} //只能用一次的物品
        else if(s[i] == 0) {s[i] = m / v[i];} //能用无限次的物品
    }
    for(int i = 1; i <= n; i++){
        for(int j = m; j >= v[i]; j--){
            for(int k = 1; k <= s[i] && k*v[i] <= j; k++){
                dp[j] = max(dp[j], dp[j-k*v[i]]+k*w[i]);
            }
        }
    }
    cout<<dp[m]<<endl;
    return 0;
}
```

```
#include <bits/stdc++.h>
using namespace std;

int main(){
    int n, m;
    cin>>n>>m;
    vector<int> v; //体积
    vector<int> w; //价值
    v.emplace_back(0); w.emplace_back(0);
    for(int i = 1; i <= n; i++){
        int v_raw, w_raw, s_raw;
        cin>>v_raw>>w_raw>>s_raw;
        if(s_raw == -1) {s_raw = 1;} //只能用一次的物品
        else if(s_raw == 0) {s_raw = m / v_raw;} //能用无限次的物品
        int k = 1;
        while(k <= s_raw){
            v.emplace_back(v_raw*k);
            w.emplace_back(w_raw*k);
            s_raw -= k;
            k *= 2;
        }
        if(s_raw > 0){
            v.emplace_back(v_raw*s_raw);
            w.emplace_back(w_raw*s_raw);
        }
    }
    n = v.size() - 1;
    vector<int> dp(m+1); //dp数组
    for(int i = 1; i <= n; i++){
        for(int j = m; j >= v[i]; j--){
            dp[j] = max(dp[j], dp[j-v[i]]+w[i]);
        }
    }
    cout<<dp[m]<<endl;
    return 0;
}
```

⑤ 二维费用背包问题

有 N 件物品和一个容量是 V 的背包，背包能承受的最大重量是 M 。

每件物品只能用一次。体积是 v_i ，重量是 m_i ，价值是 w_i 。

求解将哪些物品装入背包，可使物品总体积不超过背包容量，总重量不超过背包可承受的最大重量，且价值总和最大。

输出最大价值。

输入格式

第一行三个整数， N, V, M ，用空格隔开，分别表示物品件数、背包容积和背包可承受的最大重量。

接下来有 N 行，每行三个整数 v_i, m_i, w_i ，用空格隔开，分别表示第 i 件物品的体积、重量和价值。

输出格式

输出一个整数，表示最大价值。

三维

```
#include <bits/stdc++.h>
using namespace std;

int main(){
    int n, m, o;
    cin >> n >> o;
    vector<int> v(n+1); //体积
    vector<int> g(n+1); //重量
    vector<int> w(n+1); //价值
    vector<vector<vector<int>>> dp(m+1, vector<vector<int>>(m+1, vector<int>(o+1))); //dp数组
    for(int i = 1; i <= n; i++) {cin >> v[i] >> g[i] >> w[i];}
    for(int i = 1; i <= m; i++) {
        for(int j = 1; j <= o; j++) {
            for(int k = 1; k <= n; k++) {
                if(j < v[i] || k < g[i]) {dp[i][j][k] = dp[i-1][j][k];}
                else {dp[i][j][k] = max(dp[i-1][j][k], dp[i-1][j-v[i]][k-g[i]] + w[i]);}
            }
        }
    }
    cout << dp[n][m][o] << endl;
    return 0;
}
```

⑥ 分组背包问题

有 N 组物品和一个容量是 V 的背包。

每组物品有若干个，同一组内的物品最多只能选一个。

每件物品的体积是 v_{ij} ，价值是 w_{ij} ，其中 i 是组号， j 是组内编号。

求解将哪些物品装入背包，可使物品总体积不超过背包容量，且总价值最大。

输出最大价值。

输入格式

第一行有两个整数 N, V ，用空格隔开，分别表示物品组数和背包容量。

接下来有 N 组数据：

- 每组数据第一行有一个整数 S_i ，表示第 i 组物品的组内数量；
- 每组数据接下来有 S_i 行，每行有两个整数 v_{ij}, w_{ij} ，用空格隔开，分别表示第 i 个物品组的第 j 个物品的体积和价值；

输出格式

输出一个整数，表示最大价值。

二维 $dp[i][j] \rightarrow$ 考虑前*i*组下量为j的最大价值

```
#include <bits/stdc++.h>
using namespace std;

int main(){
    int n, m;
    cin >> n >> m;
    vector<vector<int>> v(n+1, vector<int>()); //体积
    vector<vector<int>> w(n+1, vector<int>()); //价值
    vector<int> s(n+1); //每组中物品的个数
    vector<vector<int>> dp(m+1, vector<int>(m+1)); //dp数组
    for(int i = 1; i <= n; i++) {
        cin >> s[i];
        v[i].resize(s[i]);
        w[i].resize(s[i]);
        for(int j = 0; j < s[i]; j++) {
            cin >> v[i][j] >> w[i][j];
        }
    }
    for(int i = 1; i <= n; i++) {
        for(int j = 1; j <= m; j++) {
            dp[i][j] = dp[i-1][j]; //初始化，或是第i组中哪个都不选
            for(int k = 0; k < s[i]; k++) {
                if(v[i][k] <= j) { //此判断不能并在k的for循环中
                    dp[i][j] = max(dp[i][j], dp[i-1][j-v[i][k]]+w[i][k]);
                }
            }
        }
    }
    cout << dp[n][m] << endl;
    return 0;
}
```

题解与题解：

<https://www.acwing.com/solution/content/23631/>

这道题目跟 01 背包很像，只不过实在 01 背包的基础上加上了一个重量限制。

01 背包的动态转移方程是

$$f_{i,j} = \max(f_{i-1,j}, f_{i-1,j-v_i} + w_i)$$

那么这个多了个重量，那么可以再开一维，变成三维

$$f_{i,j,k} = \max(f_{i-1,j,k}, f_{i-1,j-v_i,k-m_i} + w_i)$$

同 01 背包，它也可以压缩亿下空间。于是变成了

$$f_{j,k} = \max(f_{j-v_i,k-m_i}, f_{j,k})$$

二维

```
#include <bits/stdc++.h>
using namespace std;

int main(){
    int n, m, o;
    cin >> n >> o;
    vector<int> v(n+1); //体积
    vector<int> g(n+1); //重量
    vector<int> w(n+1); //价值
    vector<vector<int>> dp(m+1, vector<int>(o+1)); //dp数组
    for(int i = 1; i <= n; i++) {cin >> v[i] >> g[i] >> w[i];}
    for(int i = 1; i <= m; i++) {
        for(int j = 1; j <= o; j++) {
            for(int k = 1; k <= n; k++) {
                for(int l = m; l >= i; l--) {
                    for(int m = o; m >= v[i]; m--) {
                        if(g[k] >= l) {dp[j][k] = max(dp[j][k], dp[j-v[i]][l-g[i]] + w[i]);}
                    }
                }
            }
        }
    }
    cout << dp[m][o] << endl;
    return 0;
}
```

题解与题解：

<https://www.acwing.com/solution/content/3483/>

可与多重背包对比着看

一维 $dp[j] \rightarrow$ 考虑所有组下量为j的最大价值

```
#include <bits/stdc++.h>
using namespace std;

int main(){
    int n, m;
    cin >> n >> m;
    vector<vector<int>> v(n+1, vector<int>()); //体积
    vector<vector<int>> w(n+1, vector<int>()); //价值
    vector<int> s(n+1); //每组中物品的个数
    vector<vector<int>> dp(m+1); //dp数组
    for(int i = 1; i <= n; i++) {
        cin >> s[i];
        v[i].resize(s[i]);
        w[i].resize(s[i]);
        for(int j = 0; j < s[i]; j++) {
            cin >> v[i][j] >> w[i][j];
        }
    }
    for(int i = 1; i <= n; i++) {
        for(int j = m; j >= 0; j--) {
            for(int k = 0; k < s[i]; k++) {
                if(v[i][k] <= j) { //此判断不能并在k的for循环中
                    dp[j] = max(dp[j], dp[j-v[i][k]] + w[i][k]);
                }
            }
        }
    }
    cout << dp[m] << endl;
    return 0;
}
```

⑦ 背包问题与方案数

有 N 件物品和一个容量是 V 的背包。每件物品只能使用一次。

第 i 件物品的体积是 v_i , 价值是 w_i 。

求解将哪些物品装入背包，可使这些物品的总体积不超过背包容量，且总价值最大。

输出 最优选法的方案数。注意答案可能很大，请输出答案模 $10^9 + 7$ 的结果。

输入格式

第一行两个整数, N, V , 用空格隔开, 分别表示物品数量和背包容积。

接下来有 N 行, 每行两个整数 v_i, w_i , 用空格隔开, 分别表示第 i 件物品的体积和价值。

输出格式

输出一个整数, 表示 方案数 模 $10^9 + 7$ 的结果。

二维

$dp[i][j] \rightarrow$ 前 i 件物品体积为 j 的最大价值
 $g[i][j] \rightarrow$ 前 i 件物品体积为 j 的最大价值对应的方案数

为了记录方案数, 我们还需要另外开辟一个数组 $g(i, j)$, 代表的含义是: $f(i, j)$ 取最大值时的方案数。 $g(i, j)$ 的求解存在三种情况:

- (1) 如果 $f(i-1, j) > f(i-1, j-v) + w$, 则 $g(i, j) = g(i-1, j)$;
- (2) 如果 $f(i-1, j) < f(i-1, j-v) + w$, 则 $g(i, j) = g(i-1, j-v)$;
- (3) 如果 $f(i-1, j) = f(i-1, j-v) + w$, 则 $g(i, j) = g(i-1, j) + g(i-1, j-v)$;

```
#include <bits/stdc++.h>
using namespace std;

int main(){
    int n, m;
    cin >> n >> m;
    vector<int> v(n+1); //体积
    vector<int> w(n+1); //价值
    vector<vector<int>> dp(n+1, vector<int>(m+1)); //dp数组
    vector<vector<int>> g(n+1, vector<int>(m+1)); //记录方案数
    for(int i = 1; i <= n; i++) {cin >> v[i] >> w[i];}
    for(int i = 1; i <= n; i++) {dp[i][0] = 1; g[i][0] = 1;}
    for(int i = 1; i <= n; i++) {
        for(int j = 1; j <= m; j++) {
            if(j - v[i] <= 0) {dp[i][j] = dp[i-1][j]; g[i][j] = g[i-1][j];}
            else {dp[i][j] = max(dp[i-1][j], dp[i-1][j-v[i]]+w[i]);}
            //处理方案数
            if(dp[i][j] == dp[i-1][j]) {g[i][j] = g[i-1][j];}
            if(dp[i][j] == dp[i-1][j-v[i]]+w[i]) {g[i][j] = g[i-1][j-v[i]]+g[i-1][j];}
            if(dp[i][j] == dp[i-1][j-v[i]]+w[i]) {g[i][j] = g[i-1][j-v[i]]+g[i-1][j];}
        }
    }
    int ans = 0; //记录最大价值的方案数
    for(int j = 0; j <= m; j++) {
        if(dp[n][j] == dp[n][m]) {ans = (ans + g[n][j]) % 1000000007;}
    }
    cout << ans << endl;
    return 0;
}
```

原题与题解:

<https://www.acwing.com/solution/content/54273/>

与前面的变型题 方案数不同, 此题多出了
价值 w 的考虑, 因此不可用前面方案数的方法

一维

$dp[j] \rightarrow$ 所有物品体积为 j 的最大价值
 $g[j] \rightarrow$ 所有物品体积为 j 的最大价值对应的方案数

```
#include <bits/stdc++.h>
using namespace std;

int main(){
    int n, m;
    cin >> n >> m;
    vector<int> v(n+1); //体积
    vector<int> w(n+1); //价值
    vector<int> dp(m+1); //dp数组
    vector<int> g(m+1); //记录方案数
    for(int i = 1; i <= n; i++) {cin >> v[i] >> w[i];}
    dp[0] = 1; g[0] = 1;
    for(int i = 1; i <= n; i++) {
        for(int j = m; j >= v[i]; j--) {
            int temp = max(dp[j], dp[j-v[i]]+w[i]);
            int cnc = 0;
            //处理方案数
            if(temp == dp[j]) {cnc = (cnt + g[j]) % 1000000007;}
            if(temp == dp[j-v[i]]+w[i]) {cnt = (cnt + g[j-v[i]]) % 1000000007;}
            dp[j] = temp;
            g[j] = cnc;
        }
    }
    int max_val = 0; //获取最大价值, 因为取到最大价值消耗的体积不一定恰好等于背包容量
    for(int j = 0; j <= m; j++) {
        if(max_val == dp[j]) {max_val = max(max_val, dp[j]);}
    }
    int ans = 0; //记录最大价值时的方案数
    for(int j = 0; j <= m; j++) {
        if(max_val == dp[j]) {ans = (ans + g[j]) % 1000000007;}
    }
    cout << ans << endl;
    return 0;
}
```

⑧ 背包问题具体方案

有 N 件物品和一个容量是 V 的背包。每件物品只能使用一次。

第 i 件物品的体积是 v_i , 价值是 w_i 。

求解将哪些物品装入背包, 可使这些物品的总体积不超过背包容量, 且总价值最大。

输出 字典序最小的方案。这里的字典序是指: 所选物品的编号所构成的序列。物品的编号范围是 $1 \dots N$ 。

输入格式

第一行两个整数, N, V , 用空格隔开, 分别表示物品数量和背包容积。

接下来有 N 行, 每行两个整数 v_i, w_i , 用空格隔开, 分别表示第 i 件物品的体积和价值。

输出格式

输出一行, 包含若干个用空格隔开的整数, 表示最优解中所选物品的编号序列, 且该编号序列的字典序最小。

物品编号范围是 $1 \dots N$ 。

题目要求输出字典序最小的解, 假设存在一个包含第1个物品的最优解, 为了确保字典序最小那么我们必然要选第一个。那么问题就转化成从 $2 \dots N$ 这些物品中找到最优解。之前的 $f(i, j)$ 记录的都是前 i 个物品总容量为 j 的最优解, 那么我们现在将 $f(i, j)$ 定义为从第 i 个元素到最后一个元素总容量为 j 的最优解。接下来考虑状态转移:

$$f(i, j) = \max(f(i+1, j), f(i+1, j-v[i]) + w[i])$$

两种情况, 第一种是不选第 i 个物品, 那么最优解等同于从第 $i+1$ 个物品到最后一个元素总容量为 j 的最优解; 第二种是选了第 i 个物品, 那么最优解等同于当前物品的价值 $w[i]$ 加上从第 $i+1$ 个物品到最后一个元素总容量为 $j - v[i]$ 的最优解。

计算完状态表示后, 考虑如何得到最小字典序的解。首先 $f(1, m)$ 肯定是最大价值, 那么我们便开始考虑能否选取第一个物品呢。

如果 $f(1, m) = f(2, m - v[1]) + w[1]$, 说明选取了第1个物品可以得到最优解。

如果 $f(1, m) = f(2, m)$, 说明不选取第一个物品才能得到最优解。

如果 $f(1, m) = f(2, m) = f(2, m - v[1]) + w[1]$, 说明选不选都可以得到最优解, 但是为了考虑字典序最小, 我们也需要选取该物品。

原题与题解:

<https://www.acwing.com/solution/content/2687/>

注意: 此时不能将二维数组压缩为一维数组, 这是因为数组 dp 中间的状态还需要被使用

```
#include <bits/stdc++.h>
using namespace std;

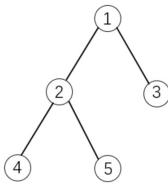
int main(){
    int n, m;
    cin >> n >> m;
    vector<int> v(n+2); //体积
    vector<int> w(n+2); //价值
    vector<vector<int>> dp(n+2, vector<int>(m+2)); //dp数组
    for(int i = 1; i <= n; i++) {cin >> v[i] >> w[i];}
    for(int i = 1; i <= n; i++) {dp[i][0] = 1; v[i] = v[i+1];}
    for(int i = 1; i <= n; i++) {
        for(int j = 1; j <= m; j++) {
            if(j < v[i]) {dp[i][j] = dp[i+1][j];}
            else {dp[i][j] = max(dp[i+1][j], dp[i+1][j-v[i]]+w[i]);}
        }
    }
    int j = m;
    for(int i = 1; i <= n; i++) {
        if(j >= v[i] && dp[i][j] == dp[i+1][j-v[i]]+w[i]) {
            cout << i << " ";
            j -= v[i];
        }
    }
    cout << endl;
    return 0;
}
```

⑨有依赖的背包问题

有 N 个物品和一个容量是 V 的背包。

物品之间具有依赖关系，自依赖关系组成一棵树的形状。如果选择一个物品，则必须选择它的父节点。

如下图所示：



如果选择物品 5，则必须选择物品 1 和 2。这是因为 2 是 5 的父节点，1 是 2 的父节点。

每件物品的编号是 i , 体积是 v_i , 价值是 w_i , 依赖的父节点编号是 p_i , 物品的下标范围是 $1 \dots N$ 。

求解将哪些物品装入背包，可使物品总体积不超过背包容量，且总价值最大。

输出最大价值。

输入格式

第一行有两个整数 N, V , 用空格隔开, 分别表示物品个数和背包容量。

接下来有 N 行数据, 每行数据表示一个物品。

第 i 行有三个整数 v_i, w_i, p_i , 用空格隔开, 分别表示物品的体积、价值和依赖的物品编号。
如果 $p_i = -1$, 表示根节点。数据保证所有物品构成一棵树。

输出格式

输出一个整数, 表示最大价值。

```
#include <bits/stdc++.h>
using namespace std;

void dfs(int node, vector<int>& v, vector<int>& w, vector<vector<int>>& tree, vector<vector<int>>& dp, int& m){
    // 点node必须选, 所以初始化dp[node][v[node] ~ m] = w[node]
    for(int i = v[node]; i <= m; i++) {dp[node][i] = w[node];}
    for(int i = 0; i < tree[node].size(); i++){
        int son = tree[node][i];
        dfs(son, v, w, tree, dp, m);
        // j的范围为v[node]~m, 小于v[node]无法选择node的子节点
        for(int j = m; j >= v[son]; j--){
            // 分给son的空间不能大于j-v[node], 不然都无法选父节点node
            for(int k = 0; k <= j-v[son]; k++){
                dp[node][j] = max(dp[node][j], dp[node][j-k]+dp[son][k]);
            }
        }
    }
}

int main(){
    int n, m, root;
    cin>>n>>m;
    vector<int> v(n+1); // 体积
    vector<int> w(n+1); // 价值
    vector<vector<int>> tree(n+1, vector<int>()); // 节点的子节点
    vector<vector<int>> dp(n+1, vector<int>(m+1)); // dp数组: 考虑第i个节点及其子节点, 在容量不超过j时所获得的最大价值
    for(int i = 1; i <= n; i++){
        int pa;
        cin>>v[i]>>w[i]>>pa;
        // 构建dp树
        if(pa == -1) {root = i;}
        else {tree[pa].emplace_back(i);}
    }
    dfs(root, v, w, tree, dp, m);
    cout<<dp[root][m]<<endl;
    return 0;
}
```

原题与题解:

<https://www.acwing.com/solution/content/8316/>

树形DP

dfs在遍历到 x 结点时, 先考虑一定选上根节点 x , 因此初始化 $f[x][v[x] \sim m] = w[x]$

在分组背包部分:

j 的范围 $[m, v[x]]$ 小于 $v[x]$ 则没有意义因为连根结点都放不下;

k 的范围 $[0, j-v[x]]$, 当大于 $j-v[x]$ 时分给该子树的容量过多, 剩余的容量连根节点的物品都放不下了;

斐波那契数列

509. 斐波那契数和剑指 Offer 10-I. 斐波那契数列：写一个函数，输入 n，求斐波那契 (Fibonacci) 数列的第 n 项 (即 $F(n)$)。
70. 爬楼梯和剑指 Offer 10-II. 青蛙跳台阶问题：假设你正在爬楼梯。需要 n 阶你才能到达楼顶。每次你可以爬 1 或 2 个台阶。你有多少种不同的方法可以爬到楼顶呢？
- 面试题 08.01. 三步问题：三步问题。有个小孩正在上楼梯，楼梯有n阶台阶，小孩一次可以上1阶、2阶或3阶。实现一种方法，计算小孩有多少种上楼梯的方式。
- 提升题（变态跳台阶）：一只青蛙一次可以跳上1级台阶，也可以跳上2级……它也可以跳上n级。求该青蛙跳上一个n级的台阶总共有多少种跳法。java版本代码为

①

```
int fib(int n) {
    vector<int> F(n+2); 防止越界
    F[0] = 0; F[1] = 1;
    for(int i = 2; i <= n; i++){
        F[i] = (F[i-1] + F[i-2]) % 1000000007;
    }
    return F[n];
}
```

② $dp[i] \rightarrow$ 跳 i 级台阶共有多少种跳法

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

③

```
int waysToStep(int n) {
    vector<int> dp(n+3);
    dp[0] = 1; dp[1] = 1; dp[2] = 2;
    for(int i = 3; i <= n; i++){
        dp[i] = ((dp[i-1] + dp[i-2]) % 1000000007 + dp[i-3]) % 1000000007;
    }
    return dp[n];
}
```

④ 由 $dp[n-1] = dp[0] + dp[1] + \dots + dp[n-1]$
 $= dp[0] + dp[1] + \dots + dp[n-2]$ $\Rightarrow dp[n] = 2 \times dp[n-1]$

$$dp[n] = dp[0] + dp[1] + \dots + dp[n-2] + dp[n-1]$$

```
int jumpFloorII(int number) {
    if(!number) return 0;
    return (number == 1) ? 1 : 2 * jumpFloorII(number-1);
}
```

递归

```
int jumpFloorII(int number) {
    return (!number) ? 0 : (1 << number - 1); 相当于 pow(2, number-1)
}
```

股票系列

经典必做系列，具体题目不放了，只写区别。

121. 买卖股票的最佳时机 和 剑指 Offer 63. 股票的最大利润：最多1笔交易
122. 买卖股票的最佳时机 II：尽可能更多次的交易
123. 买卖股票的最佳时机 III：2笔交易
188. 买卖股票的最佳时机 IV：最多k笔交易
309. 最佳买卖股票时机含冷冻期：与122题的区别是卖出股票后，你无法在第二天买入股票（即冷冻期为1天）。
714. 买卖股票的最佳时机含手续费：与122题的区别是每笔交易都需要付手续费。

题解：<https://leetcode.cn/problems/best-time-to-buy-and-sell-stock/solutions/38477/bao-li-mei-ju-dong-tai-gui-hua-chai-fen-si-xiang-b/>

状态定义：

$\text{dp}[i][j]$ ：下标为 i 这一天结束的时候，手上持股状态为 j 时，我们持有的现金数。换种说法： $\text{dp}[i][j]$ 表示天数 $[0, i]$ 区间里，下标 i 这一天状态为 j 的时候能够获得的最大利润。其中：

- $j = 0$ ，表示当前不持股；
- $j = 1$ ，表示当前持股。

注意：下标为 i 的这一天的计算结果包含了区间 $[0, i]$ 所有的信息，因此最后输出 $\text{dp}[len - 1][0]$ 。

推导状态转移方程：

$\text{dp}[i][0]$ ：规定了今天不持股，有以下两种情况：

- 昨天不持股，今天什么都不做；
- 昨天持股，今天卖出股票（现金数增加），

$\text{dp}[i][1]$ ：规定了今天持股，有以下两种情况：

- 昨天持股，今天什么都不做（现金数与昨天一样）；
- 昨天不持股，今天买入股票（注意：只允许交易一次，因此手上的现金数就是当天的股价的相反数）。

妙

二维

```
int maxProfit(vector<int>& prices) {
    int len = prices.size();
    if(len < 2) {return 0;}
    // dp[i][0] 下标为 i 这天结束的时候，不持股，手上拥有的现金数
    // dp[i][1] 下标为 i 这天结束的时候，持股，手上拥有的现金数
    vector<vector<int>> dp(len, vector<int>(2));
    // 初始化：不持股显然为 0，持股就需要减去第 1 天（下标为 0）的股价
    dp[0][0] = 0;
    dp[0][1] = -prices[0];
    // 从第 2 天开始遍历
    for(int i = 1; i < len; i++){
        dp[i][0] = max(dp[i-1][0], dp[i-1][1] + prices[i]); }互换顺序也可
        dp[i][1] = max(dp[i-1][1], -prices[i]);
    }
    return dp[len-1][0];
}
```

一维

```
int maxProfit(vector<int>& prices) {
    int len = prices.size();
    if(len < 2) {return 0;}
    // dp[0] 遍历到的那一天结束后，不持股，手上拥有的现金数
    // dp[1] 遍历到的那一天结束后，持股，手上拥有的现金数
    vector<int> dp(2);
    // 初始化：不持股显然为 0，持股就需要减去第 1 天（下标为 0）的股价
    dp[0] = 0;
    dp[1] = -prices[0];
    // 从第 2 天开始遍历
    for(int i = 1; i < len; i++){
        dp[0] = max(dp[0], dp[1] + prices[i]); }互换顺序也可
        dp[1] = max(dp[1], -prices[i]);
    }
    return dp[0];
}
```

一维

```
int maxProfit(vector<int>& prices) {
    int len = prices.size();
    if(len < 2) {return 0;}
    // dp[0] 遍历到的那一天结束后，不持股，手上拥有的最大现金数
    // dp[1] 遍历到的那一天结束后，持股，手上拥有的最大现金数
    vector<int> dp(2);
    // 初始化：不持股显然为 0，持股就需要减去第 1 天（下标为 0）的股价
    dp[0] = 0;
    dp[1] = -prices[0];
    // 从第 2 天开始遍历
    for(int i = 1; i < len; i++){
        dp[0] = max(dp[0], dp[1] + prices[i]); }互换顺序也可
        dp[1] = max(dp[1], dp[0] - prices[i]);
    }
    return dp[0];
}
```

二维

```
int maxProfit(vector<int>& prices) {
    int len = prices.size();
    if(len < 2) {return 0;}
    // d[i][j][k] 下标为 i 这天结束的时候，买入股份次数为 j，且持股(1)/不持股(0)时，手上拥有的最大现金数
    vector<vector<vector<int>> dp(len, vector<vector<int>>(3, vector<int>(2)));
    // 初始化
    dp[0][1][1] = -prices[0];
    dp[0][2][1] = -prices[0];
    // 从第二天开始遍历
    for(int i = 1; i < len; i++){
        // 转移顺序先持股，再卖出
        dp[i][1][1] = max(dp[i-1][1][1], -prices[i]);
        dp[i][1][0] = max(dp[i-1][1][0], dp[i-1][1][1] + prices[i]);
        dp[i][2][1] = max(dp[i-1][2][1], dp[i-1][1][0] - prices[i]);
        dp[i][2][0] = max(dp[i-1][2][0], dp[i-1][2][1] + prices[i]);
    }
    return dp[len-1][2][0];
}
```

可

② 二维

二维

```
int maxProfit(vector<int>& prices) {
    int len = prices.size();
    if(len < 2) {return 0;}
    // dp[i][j][k] 表示在 [0, i] 区间里（状态具有后缀性质），交易进行了 j 次，并且状态为 k 时我们拥有的现金数。其中 j 和 k 的含义如下：
    // j = 0 表示没有交易发生；j = 1 表示此时已经发生了 1 次买入股票的行为；j = 2 表示此时已经发生了 2 次买入股票的行为。
    // 即我们一般规定 记录一次交易发生在 买入股票 的时候。
    // k = 0 表示当前不持股；k = 1 表示当前持股。
    // 考虑初始化：
    // 下标为 0 这一天，交易次数为 0、1、2 并且状态为 0 和 1 的初值应该如下设置：
```

- $\text{dp}[0][0][0] = 0$ ：这是必然的；
- $\text{dp}[0][0][1] = 0$ ：表示一次交易都没有发生，但是持股，这是不可能的，也不会有后续的决策用这个状态值，所以不用填；
- $\text{dp}[0][1][0] = 0$ ：表示发生了 1 次交易，但是不持股，这是不可能的，**虽然没有意义，但是设成 0 不会影响最终结果**；
- $\text{dp}[0][1][1] = -prices[0]$ ：表示发生了第一次交易，并且持股，所以我们持有的现金数就是当天股价的相反数；
- $\text{dp}[0][2][0] = 0$ ：表示发生了 2 次交易，但是不持股，这是不可能的，**虽然没有意义，但是设成 0 不会影响最终结果**；
- $\text{dp}[0][2][1] = -prices[0]$ ：表示发生了 2 次交易，并且持股，这是不可能的，注意：**虽然没有意义，但是设成 0 是错误的**，这是因为交易还没有发生，必须规定当天 k 状态为 1（持股），需要参考以往的状态转移，一种很有可能的情况就是没有交易是最好的情况。**→初始化 j-prices[i]**

说明： $\text{dp}[i][2][1] = 0$ 设置成为负无穷这件事情我可能没有说清楚，大家可以尝试特殊测试用例 $[1, 2, 3, 4, 5]$ ，对 $\text{dp}[i][2][1] = 0$ 与 $\text{dp}[i][2][1] = -infinity$ 的状态转移的差异去理解。

注意：只有**之前**的状态有被赋值的时候，才可能有当前状态。

一维

```

int maxProfit(vector<int>& prices) {
    int len = prices.size();
    if(len < 2) {return 0;}
    // d[i][j][k] 下标为i这一天结束的时候，买入股份数次为j，且持股(1)/不持股(0)时，手上拥有的最大现金数
    vector<vector<vector<int>> dp(3, vector<int>(2));
    // 初始化
    dp[1][1][0] = -prices[0];
    dp[2][1][0] = -prices[0];
    // 从第2天开始遍历
    for(int i = 1; i < len; i++){
        // 转移顺序先持股，再卖出
        dp[1][1][1] = max(dp[1][1], -prices[i]);
        dp[1][0][0] = max(dp[1][0], dp[1][1] + prices[i]);
        dp[2][1][1] = max(dp[2][1], dp[1][0] - prices[i]);
        dp[2][0][0] = max(dp[2][0], dp[2][1] + prices[i]);
    }
    return dp[2][0];
}

```

④ 二维

```

int maxProfit(int k, vector<int>& prices) {
    int len = prices.size();
    if(k == 0 || len < 2) {return 0;}
    // d[i][j][k]: 下标为i这一天结束的时候，买入股份数次为j，且持股(1)/不持股(0)时，手上拥有的最大现金数
    vector<vector<vector<int>> dp(len+1, vector<vector<int>(k+1, vector<int>(2)));
    // 初始化：把持股的部分都设置为一个较小的负值
    for(int i = 0; i < len; i++){
        for(int j = 0; j < k; j++){
            dp[i][j][1] = INT_MIN;
        }
    }
    // 开始遍历
    for(int i = 1; i < len; i++){
        for(int j = 1; j <= k; j++){
            // 先持股后卖出
            dp[i][j][1] = max(dp[i-1][j][1], dp[i-1][j-1][0] - prices[i-1]);
            dp[i][j][0] = max(dp[i-1][j][0], dp[i-1][j][1] + prices[i-1]);
        }
    }
    return dp[len][k][0];
}

```

一维

```

int maxProfit(int k, vector<int>& prices) {
    int len = prices.size();
    if(k == 0 || len < 2) {return 0;}
    // d[i][k]: 遍历到的那一天结束后，买入股份数次为j，且持股(1)/不持股(0)时，手上拥有的最大现金数
    vector<vector<int>> dp(k+1, vector<int>(2));
    // 初始化：把持股的部分都设置为一个较小的负值
    for(int j = 0; j <= k; j++){
        dp[j][1] = INT_MIN;
    }
    // 开始遍历
    for(int i = 1; i < len; i++){
        for(int j = 1; j <= k; j++){
            // 先持股后卖出
            dp[j][1] = max(dp[j][1], dp[j-1][0] - prices[i-1]);
            dp[j][0] = max(dp[j][0], dp[j][1] + prices[i-1]);
        }
    }
    return dp[k][0];
}

```

二维

```

int maxProfit(vector<int>& prices) {
    int len = prices.size();
    if(len < 2) {return 0;}
    vector<vector<int>> dp(len, vector<int>(3));
    dp[0][0] = 0;
    dp[0][1] = -prices[0];
    dp[0][2] = 0;
    // dp[i][0]: 下标为i这一天结束后，手上不持有股票，并且今天不是冷冻期，拥有的现金数
    // dp[i][1]: 下标为i这一天结束后，手上持有股票时，拥有的现金数
    // dp[i][2]: 下标为i这一天结束后，手上不持有股票，并且今天是冷冻期，拥有的现金数
    for(int i = 1; i < len; i++){
        dp[i][0] = max(dp[i-1][0], dp[i-1][2]);
        dp[i][1] = max(dp[i-1][1], dp[i-1][0]-prices[i]);
        dp[i][2] = dp[i-1][1] + prices[i];
    }
    return max(dp[len-1][0], dp[len-1][2]);
}

```

一维

```

int maxProfit(vector<int>& prices) {
    int len = prices.size();
    if(len < 2) {return 0;}
    vector<int> dp(3);
    dp[0] = 0;
    dp[1] = -prices[0];
    dp[2] = 0;
    int pre0 = dp[0];
    int pre2 = dp[2];
    // dp[0]: 手上不持有股票，并且今天不是冷冻期，拥有的现金数
    // dp[1]: 手上持有股票时，拥有的现金数
    // dp[2]: 手上不持有股票，并且今天是冷冻期，拥有的现金数
    // 这里要用pre来记录i-1时候的现金数，不然max里的是更新后的i的现金数，显然不对
    // 由于之前的题目是按照先持股后卖出的顺序，所以max里的是没更新的现金数，故不需要记录
    for(int i = 1; i < len; i++){
        dp[0] = max(dp[0], pre2);
        dp[1] = max(dp[1], pre0-prices[i]);
        dp[2] = dp[1] + prices[i];
        pre0 = dp[0];
        pre2 = dp[2];
    }
    return max(dp[0], dp[2]);
}

```

⑤

思路：这道题增加了冷冻期这个限制条件。根据题意：卖出股票后的第 2 天为冷冻期。即：

- 卖出股票的当天：不持股；
- 卖出股票的第 2 天：冷冻期（不能买入股票，当然也不能卖出股票）；
- 卖出股票的第 3 天：可以买入股票，也可以什么都不操作。

并没有限制多少笔交易，因此需要增加一个状态。

把“冷冻期”定义成一个状态，不太好推导状态转移方程，由于事实上就只有「持股」和「不持股」这两种情况。因此可以为不持股增加一种情况：

第 1 步：状态定义

dp[i][j] 表示 [i..j] 区间内，在下标为 i 这一天状态为 j 时，我们手上拥有的金钱数。

这里 j 可以取 3 个值（下面的定义非常需要）：

- 0 表示：今天 不是 卖出了股票的不持股状态；
- 1 表示：持股；
- 2 表示：今天由于卖出了股票的不持股状态；
- 0 的转移

昨天	今天	分析是否可以转移，可以转移的情况下今天的操作
0	0	可以转移，今天什么都不做。
0	1	可以转移，今天买入股票。
0	2	不可以转移，不持股的情况下，不能卖出股票。

• 1 的转移

昨天	今天	分析是否可以转移，可以转移的情况下今天的操作
1	0	不可以转移，根据题意，只能转移到 2。
1	1	可以转移，今天什么都不操作。
1	2	根据题意可以转移。

• 2 的转移

昨天	今天	分析是否可以转移，可以转移的情况下今天的操作
2	0	可以转移，根据题意，今天就是冷冻期，什么都不能操作，进入状态 0。
2	1	不可以转移，根据题意，昨天刚刚买入股票，今天不能执行买入操作。
2	2	不可以转移，不持股的情况下，不能卖出股票。

6

二驻 (在题②基础上增加fee即可)

```
int maxProfit(vector<int>& prices, int fee) {
    int len = prices.size();
    if(len < 2) {return 0;}
    // dp[i][0] 下标为 i 这天结束的时候, 不持股, 手上拥有的最大现金数
    // dp[i][1] 下标为 i 这天结束的时候, 持股, 手上拥有的最大现金数
    vector<vector<int>> dp(len, vector<int>(2));
    // 初始化: 不持股显然为 0, 持股就需要减去第 1 天 (下标为 0) 的股价
    dp[0][0] = 0;
    dp[0][1] = -prices[0] - fee;
    // 从第 2 天开始遍历
    for(int i = 1; i < len; i++){
        dp[i][0] = max(dp[i-1][0], dp[i-1][1] + prices[i]);
        dp[i][1] = max(dp[i-1][1], dp[i-1][0] - prices[i] - fee);
    }
    return dp[len-1][0];
}
```

一维

```
int maxProfit(vector<int>& prices, int fee) {
    int len = prices.size();
    if(len < 2) {return 0;}
    // dp[0] 遍历到的那一天结束后, 不持股, 手上拥有的最大现金数
    // dp[1] 遍历到的那一天结束后, 持股, 手上拥有的最大现金数
    vector<int> dp(2);
    // 初始化: 不持股显然为 0, 持股就需要减去第 1 天 (下标为 0) 的股价
    dp[0] = 0;
    dp[1] = -prices[0] - fee;
    // 从第 2 天开始遍历
    for(int i = 1; i < len; i++){
        dp[0] = max(dp[0], dp[1] + prices[i]);
        dp[1] = max(dp[1], dp[0] - prices[i] - fee);
    }
    return dp[0];
}
```

鸡蛋或者其他物品掉落

高频改造题，把鸡蛋改成任意物品（比如手机），就是道新的笔试题了。

1. 887. 鸡蛋掉落：你将获得 K 个鸡蛋，并可以使用一栋从 1 到 N 共有 N 层楼的建筑。每个蛋的功能都是一样的，如果一个蛋碎了，你就不能再把它掉下去。你知道存在楼层 F，满足 $0 \leq F \leq N$ 任何从高于 F 的楼层落下的鸡蛋都会碎，从 F 楼层或比它低的楼层落下的鸡蛋都不会破。每次移动，你可以取一个鸡蛋（如果你有完整的鸡蛋）并把它从任一楼层 X 扔下（满足 $1 \leq X \leq N$ ）。你的目标是确切地知道 F 的值是多少。无论 F 的初始值如何，你确定 F 的值的最小移动次数是多少？

类型题：只给2个鸡蛋时候怎么办

动态规划

```
int superEggDrop(int k, int n) {
    // dp[i][j]：一共有 k 层楼梯的情况下，使用 j 个鸡蛋的最少实验的次数
    vector<vector<int>> dp(n+1, vector<int>(k+1, INT_MAX));
    // 初始化
    // 楼层为 0 的时候，不管鸡蛋个数多少，都测试不出鸡蛋的 F 值，故全为 0
    for(int j = 0; k <= k; j++) {dp[0][j] = 0;}
    // 楼层为 1 的时候，0 个鸡蛋的时候，扔 0 次
    dp[1][0] = 0;
    // 楼层为 1 的时候，1 个以及 1 个鸡蛋以上只需要扔 1 次
    for(int j = 1; j <= k; j++) {dp[1][j] = 1;}
    // 鸡蛋个数为 0 的时候，不管楼层为多少，也测试不出鸡蛋的 F 值，故全为 0
    // 鸡蛋个数为 1 的时候，这是一种极端情况，要试出 F 值，最少次数就等于楼层高度
    for(int i = 0; i < n; i++) {dp[i][0] = 0; dp[i][1] = i;}
    // 从第 2 行，第 2 行开始遍历
    for(int i = 2; i <= n; i++){
        for(int j = 2; j <= k; j++){
            for(int k = 1; k <= i; k++){
                // 碎了，就需要往底层继续扔，层数少 1，鸡蛋也少 1
                // 不碎，就需要往高层继续扔，层数是当前层到最高层的距离差，鸡蛋数量不少
                // 两种情况都做了一次尝试，所以加 1
                dp[i][j] = min(dp[i-1][j], max(dp[k-1][j-1], dp[i-k][j]) + 1);
            }
        }
    }
    return dp[n][k];
}
```

二维 (另一种写法)

```
int maxProfit(vector<int>& prices, int fee) {
    int len = prices.size();
    if(len < 2) {return 0;}
    // dp[i][0] 下标为 i 这天结束的时候, 不持股, 手上拥有的最大现金数
    // dp[i][1] 下标为 i 这天结束的时候, 持股, 手上拥有的最大现金数
    vector<vector<int>> dp(len, vector<int>(2));
    // 初始化: 不持股显然为 0, 持股就需要减去第 1 天 (下标为 0) 的股价
    dp[0][0] = 0;
    dp[0][1] = -prices[0];
    // 从第 2 天开始遍历
    for(int i = 1; i < len; i++){
        dp[i][0] = max(dp[i-1][0], dp[i-1][1] + prices[i]);
        dp[i][1] = max(dp[i-1][1], dp[i-1][0] - prices[i] - fee);
    }
    return dp[len-1][0];
}
```

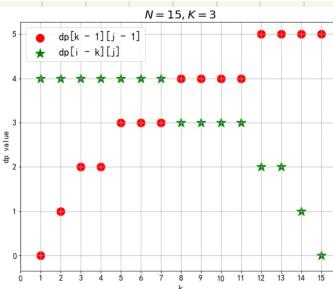
题解：

<https://leetcode.cn/problems/super-egg-drop/solutions/197012/dong-tai-gui-hua-zhi-jie-shi-quan-fang-ti-jie-fang/>

动态规划 + 二分查找

- $dp[k-1][j-1]$ ：根据语义， k 增大的时候，楼层大小越大，它的值就越大；
- $dp[i-k][j]$ ：根据语义， k 增大的时候，楼层大小越小，它的值就越小。

可以得出一个是单调不减的（ $dp[k-1][j-1]$ ，下图红点），一个是单调不增的（ $dp[i-k][j]$ ，下图绿星），并且它们的值都是整数。



(生成图表的代码在本题解末尾。)

从图上可以看出：二者的较大值的最小点在它们交汇的地方。那么有没有可能不交汇，当然有可能（上面第 3 张图），二者较大值的最小者一定出现在画成曲线段交点的两侧，并且二者差值不会超过 1，也就是如果没有重合的点，两边的最大值是一样的（从图上看出来的，没有严格证明）。因此取左端和右端两点中的一点都可以。不失一般性，可以取左边的那个点的 k。

也就是找到使得 $dp[i-k][j] \leq dp[k-1][j-1]$ 最大的那个 k 值即可，这里使用二分查找法。关键在于 $dp[i-k][j] > dp[k-1][j-1]$ 的时候， k 一定不是我们要找的，根据这一点写出二分的代码。

```

int superEggDrop(int k, int n) {
    // dp[i][j]：一共有 i 层楼梯的情况下，使用 j 个鸡蛋的最少实验的次数
    vector<vector<int>> dp(n+1, vector<int>(k+1, INT_MAX));
    // 初始化
    // 楼层为 0 的时候，不管鸡蛋个数多少，都测试不出鸡蛋的 F 值，故全为 0
    for(int j = 0; j <= k; j++) {dp[0][j] = 0;}
    // 楼层为 1 的时候，0 个鸡蛋的时候，扔 0 次
    dp[1][0] = 0;
    // 楼层为 1 的时候，1 个以及 1 个鸡蛋以上只需要扔 1 次
    for(int j = 1; j <= k; j++) {dp[1][j] = 1;}
    // 鸡蛋个数为 0 的时候，不管楼层为多少，也测试不出鸡蛋的 F 值，故全为 0
    // 鸡蛋个数为 1 的时候，这是一种极端情况，要测试出 F 值，最少次数就等于楼层高度
    for(int i = 0; i < n; i++) {dp[i][0] = 0; dp[i][1] = i;}
    // 从第 2 行，第 2 列开始遍历
    for(int i = 2; i < n; i++) {
        for(int j = 2; j <= k; j++){
            int first = 1, last = i;
            while(first < last){
                // 找 dp[k - 1][i - 1] <= dp[i - mid][j] 的最大值 k
                int mid = first + (last - first)/2;
                int EggBreak = dp[mid-1][i-1];
                int EggnotBreak = dp[i-mid][j];
                if(EggBreak < EggnotBreak) {first = mid + 1;}
                else {last = mid;}
            }
            dp[i][j] = max(dp[first-1][j-1], dp[i-first][j]) + 1;
        }
    }
    return dp[n][k];
}

```

lower_bound

打家劫舍

高频改造题，搞不懂有这智商为啥去打家劫舍。

198. 打家劫舍：你是一个专业的小偷，计划偷窃沿街的房屋。每间房内都藏有一定的现金，影响你偷窃的唯一制约因素就是相邻的房屋装有相互连通的防盗系统，如果两间相邻的房屋在同一晚上被小偷闯入，系统会自动报警。给定一个代表每个房屋存放金额的非负整数数组，计算你 不触动警报装置的情况下，一夜之内能够偷窃到的最高金额。
213. 打家劫舍 II：你是一个专业的小偷，计划偷窃沿街的房屋，每间房内都藏有一定的现金。这个地方所有的房屋都围成一圈，这意味着第一个房屋和最后一个房屋是紧挨着的。同时，相邻的房屋装有相互连通的防盗系统，如果两间相邻的房屋在同一晚上被小偷闯入，系统会自动报警。给定一个代表每个房屋存放金额的非负整数数组，计算你 在不触动警报装置的情况下，能够偷窃到的最高金额。
337. 打家劫舍 III：在上次打劫完一条街道之后和一圈房屋后，小偷又发现了一个新的可行窃的地区。这个地区只有一个入口，我们称之为“根”。除了“根”之外，每栋房子有且只有一个“父”房子与之相连。一番侦察之后，聪明的小偷意识到“这个地方的所有房屋的排列类似于一棵二叉树”。如果两个直接相连的房子在同一天晚上被打劫，房屋将自动报警。计算在不触动警报的情况下，小偷一晚能够盗取的最高金额。

解题：

<https://leetcode.cn/problems/house-robber/solutions/994214/dai-ma-sui-xiang-lu-dai-ni-xue-tou-dong-ezyt3/>

1. 确定dp数组（dp table）以及下标的含义

dp[i]: 考虑下标i（包括i）以内的房屋，最多可以偷窃的金额为dp[i]。

2. 确定递推公式

决定dp[i]的因素就是第i房间偷还是不偷。

如果偷第i房间，那么dp[i] = dp[i - 2] + nums[i]，即：第i-1房一定是不考虑的，找出下标i-2（包括i-2）以内的房屋，最多可以偷窃的金额为dp[i-2]加上第i房间偷到的钱。

如果不偷第i房间，那么dp[i] = dp[i - 1]，即考虑i-1房。（注意这里是考虑，并不是一定要偷1房，这是很多同学容易混淆的点）

然后dp[i]取最大值，即dp[i] = max(dp[i - 2] + nums[i], dp[i - 1]);

3. dp数组如何初始化

从递推公式dp[i] = max(dp[i - 2] + nums[i], dp[i - 1]);可以看出，递推公式的基础就是dp[0]和dp[1]

从dp[i]的定义上来讲，dp[0]一定是nums[0]，dp[1]就是nums[0]和nums[1]的最大值即：dp[1] = max(nums[0], nums[1]);

```

int rob(vector<int>& nums) {
    int n = nums.size();
    if(n == 0) {return 0;}
    if(n == 1) {return nums[0];}
    vector<int> dp(n);
    // dp[i]: 考虑下标为i以内且包括i的房屋，可以偷窃的最高金额
    dp[0] = nums[0];
    dp[1] = max(nums[0], nums[1]);
    for(int i = 2; i < n; i++){
        dp[i] = max(dp[i-1], dp[i-2] + nums[i]);
    }
    return dp[n-1];
}

```

2.

```

int rob(vector<int>& nums) {
    int n = nums.size();
    if(n == 0) {return 0;}
    if(n == 1) {return nums[0];}
    int rob1 = DPprocessing(nums, 0, n-2); //不考虑最后一个房屋
    int rob2 = DPprocessing(nums, 1, n-1); //不考虑第一个房屋
    return max(rob1, rob2);
}

int DPprocessing(vector<int>& nums, int start, int end){
    if(start == end) {return nums[start];}
    vector<int> dp(nums.size());
    dp[start] = nums[start];
    dp[start + 1] = max(nums[start], nums[start + 1]);
    for(int i = start + 2; i <= end; i++){
        dp[i] = max(dp[i-1], dp[i-2] + nums[i]);
    }
    return dp[end];
}

```

3. 所以dp数组 (dp table) 以及下标的含义：下标为0记录不偷该节点所得到的最大金钱，下标为1记录偷该节点所得到的最大金钱。

所以本题dp数组就是一个长度为2的数组！

那么有同学可能疑惑，长度为2的数组怎么标记树中每个节点的状态呢？

别忘了在递归的过程中，系统栈会保存每一层递归的参数。

首先明确的是使用后序遍历。因为通过递归函数的返回值来做下一步计算。

通过递归左节点，得到左节点偷与不偷的金钱。

通过递归右节点，得到右节点偷与不偷的金钱。

如果是偷当前节点，那么左右孩子就不能偷， $val1 = cur->val + left[0] + right[0]$; (如果对下标含义理解错误在回顾一下dp数组的含义)

如果不偷当前节点，那么左右孩子就可以偷，至于到底偷不偷一定是选一个最大的，所以： $val2 = max(left[0], left[1]) + max(right[0], right[1])$;

最后当前节点的状态就是 $(val2, val1)$ 即：{不偷当前节点得到的最大金钱，偷当前节点得到的最大金钱}

```

int rob(TreeNode* root) {
    vector<int> result = robTree(root);
    return max(result[0], result[1]);
}

```

// 长度为2的dp数组，下标为0是不偷当前节点下的最高金额，下标为1是偷当前节点下的最高金额
vector<int> robTree(TreeNode* cur){
 //若此节点为空，则偷与不偷该节点最高金额都是0，其实此句也相当于初始化dp数组
 if(cur == nullptr) {return vector<int>{0, 0};}
 // 后序遍历，保证该节点的子节点都已经遍历完
 vector<int> dp_left = robTree(cur->left);
 vector<int> dp_right = robTree(cur->right);
 // 偷当前节点，故不可以偷其左右节点
 int val1 = cur->val + dp_left[0] + dp_right[0];
 int val2 = max(dp_left[0], dp_left[1]) + max(dp_right[0], dp_right[1]);
 return {val2, val1};
}

路径

- 62. 不同路径：**一个机器人位于一个 $m \times n$ 网格的左上角（起始点在下图中标记为 "Start"）。机器人每次只能向下或者向右移动一步。机器人试图达到网格的右下角（在下图中标记为 "Finish"）。问总共有多少条不同的路径？
- 63. 不同路径 II：**与上一题的区别是网格中有障碍物。
改造题：面试题 08.02：迷路的机器人：是不同路径2的求路径题目（用回溯）
- 64. 最小路径和：**给定一个包含非负整数的 $m \times n$ 网格 grid，找出一条从左上角到右下角的路径，使得路径上的数字总和为最小。每次只能向下或者向右移动一步。
- 4. 剑指 Offer 47. 礼物的最大价值：**在一个 $m \times n$ 的棋盘的每一格都放有一个礼物，每个礼物都有一定的价值（价值大于 0）。你可以从棋盘的左上角开始拿格子里的礼物，并每次向右或者向下移动一格、直到到达棋盘的右下角。给定一个棋盘及其上面的礼物的价值，请计算你最多能拿到多少价值的礼物？
- 5. 120. 三角形最小路径和：**给定一个三角形 triangle，找出自顶向下的最小路径和。每一步只能移动到下一行中相邻的结点上。相邻的结点在这里指的是下标与上一层结点下标相同或者等于上一层结点下标 + 1 的两个结点。也就是说，如果正位于当前行的下标 i，那么下一步可以移动到下一行的下标 i 或 i + 1。

1. ① 动态规划

$dp[i][j] \rightarrow$ 到下标为(i,j)的树的路径数

```

int uniquePaths(int m, int n) {
    vector<vector<int>> dp(m, vector<int>(n));
    dp[0][0] = 1;
    for(int i = 0; i < m; i++){
        for(int j = 0; j < n; j++){
            if(j > 0) {dp[i][j] += dp[i][j-1];}
            if(i > 0) {dp[i][j] += dp[i-1][j];}
        }
    }
    return dp[m-1][n-1];
}

```

② DFS (会超时)

```

int uniquePaths(int m, int n) {
    int cnt = 0;
    dfs(1, 1, m, n, cnt);
    return cnt;
}

void dfs(int x, int y, int& m, int& n, int& cnt){
    if(x == m && y == n) {cnt++; return;}
    if(x > m || y > n) {return;}
    dfs(x, y+1, m, n, cnt);
    dfs(x+1, y, m, n, cnt);
}

```

```

2. int uniquePathsWithObstacles(vector<vector<int>>& obstacleGrid) {
    int m = obstacleGrid.size(), n = obstacleGrid[0].size();
    vector<vector<int>> dp(m, vector<int>(n));
    dp[0][0] = 1;
    for(int i = 0; i < m; i++){
        for(int j = 0; j < n; j++){
            if(obstacleGrid[i][j] == 1) {dp[i][j] = 0;}
            else{
                if(j > 0) {dp[i][j] += dp[i][j-1];}
                if(i > 0) {dp[i][j] += dp[i-1][j];}
            }
        }
    }
    return dp[m-1][n-1];
}

```

改造题、由于是具体路径，故无法用 DP，只能用 DFS

```

vector<vector<int>> pathWithObstacles(vector<vector<int>>& obstacleGrid) {
    int m = obstacleGrid.size(), n = obstacleGrid[0].size();
    if(obstacleGrid[0][0] || obstacleGrid[m - 1][n - 1]) return {};
    vector<vector<bool>> visited(m, vector<bool>(n, false));
    vector<vector<int>> path;
    vector<vector<int>> direction = {{0, 1}, {1, 0}};
    bool isFind = false;
    visited[0][0] = true;
    path.push_back({0, 0});
    dfs(0, 0, m, n, path, visited, isFind, obstacleGrid, direction);
    if(isFind) {return path;}
    else {return {};}
}

void dfs(int x, int y, int& m, int& n, vector<vector<int>>& path, vector<vector<bool>>& visited, bool& isFind, vector<vector<int>>& obstacleGrid,
vector<vector<int>>& direction){
    if(isFind) {return;}
    if(x == m-1 && y == n-1){
        isFind = true;
        return;
    }
    for(int i = 0; i < 2 && !isFind; i++){
        int newx = x + direction[i][0];
        int newy = y + direction[i][1];
        if(newx > 0 && newy >= 0 && newx < m && newy < n){
            if(!visited[newx][newy] && !obstacleGrid[newx][newy]){
                visited[newx][newy] = true;
                path.push_back({newx, newy});
                dfs(newx, newy, m, n, path, visited, isFind, obstacleGrid, direction);
                if(!isFind) {path.pop_back();}
            }
        }
    }
}

```

由于只用某一条路径，故用 isFind 来
提前结束 DFS

3. $dp[i][j] \rightarrow$ 从 $(0,0)$ 到 (i,j) 路径上的数字总和最小值

```

int minPathSum(vector<vector<int>>& grid) {
    int m = grid.size(), n = grid[0].size();
    vector<vector<int>> dp(m, vector<int>(n));
    dp[0][0] = grid[0][0];
    for(int i = 1; i < m; i++) {dp[i][0] = dp[i-1][0] + grid[i][0];}
    for(int j = 1; j < n; j++) {dp[0][j] = dp[0][j-1] + grid[0][j];}
    for(int i = 1; i < m; i++){
        for(int j = 1; j < n; j++){
            dp[i][j] = min(dp[i-1][j], dp[i][j-1]) + grid[i][j];
        }
    }
    return dp[m-1][n-1];
}

```

4. 将第3题的 min 改为 max 即可

5. 自下向上 DP

这里的 DP 数组就是 triangle，其实第3、4题也可以这样处理

```

int minimumTotal(vector<vector<int>>& triangle) {
    for(int i = triangle.size()-2; i >= 0; i--){
        for(int j = 0; j < triangle[i].size(); j++){
            triangle[i][j] += min(triangle[i+1][j], triangle[i+1][j+1]);
        }
    }
    return triangle[0][0];
}

```

石子游戏

很多乱七八糟的，只列举前两题，注意找规律。

- 1. 877. 石子游戏：**亚历克斯和李用几堆石子在做游戏。偶数堆石子排成一行，每堆都有正整数颗石子 $piles[i]$ 。游戏以谁手中的石子最多来决出胜负。石子的总数是奇数，所以没有平局。亚历克斯和李轮流进行，亚历克斯先开始。每回合，玩家从行的开始或结束处取走整堆石头。这种情况一直持续到没有更多的石子堆为止，此时手中石子最多的玩家获胜。假设亚历克斯和李都发挥出最佳水平，当亚历克斯赢得比赛时返回 `true`，当李赢得比赛时返回 `false`。**PS：**此题选手必胜，想清楚为什么。
- 2. 1140. 石子游戏 II：**亚历克斯和李继续他们的石子游戏。许多堆石子 排成一行，每堆都有正整数颗石子 $piles[i]$ 。游戏以谁手中的石子最多来决出胜负。亚历克斯和李轮流进行，亚历克斯先开始。最初， $M = 1$ 。在每个玩家的回合中，该玩家可以拿走剩下的 前 X 堆的所有石子，其中 $1 \leq X \leq 2M$ 。然后，令 $M = \max(M, X)$ 。游戏一直持续到所有石子都被拿走。假设亚历克斯和李都发挥出最佳水平，返回亚历克斯可以得到的最大数量的石头。

1. 定义二维数组 dp ，其行数和列数都等于石子的堆数， $dp[i][j]$ 表示当剩下的石子堆为下标*i*到下标 *j*时，即在下标范围 $[i, j]$ 中，当前玩家与另一个玩家的石子数量之差的最大值，注意当前玩家不一定是先手 Alice。

只有当 $i \leq j$ 时，剩下的石子堆才有意义，因此当 $i > j$ 时， $dp[i][j] = 0$ 。

当 $i = j$ 时，只剩下一堆石子，当前玩家只能取走这堆石子，因此对于所有 $0 \leq i < piles.length$ ，都有 $dp[i][i] = piles[i]$ 。

当 $i < j$ 时，当前玩家可以选择取走 $piles[i]$ 或 $piles[j]$ ，然后轮到另一个玩家在剩下的石子堆中取走石子。在两种方案中，当前玩家会选择最优的方案，使得自己的石子数量最大化。因此可以得到如下状态转移方程：

$$dp[i][j] = \max(piles[i] - dp[i+1][j], piles[j] - dp[i][j-1])$$

最后判断 $dp[0][piles.length - 1]$ 的值，如果大于 0，则 Alice 的石子数量大于 Bob 的石子数量，因此 Alice 赢得比赛，否则 Bob 赢得比赛。

```

bool stoneGame(vector<int>& piles) {
    int n = piles.size();
    // dp[i][j]: 在下标为[i,j]的石子堆下，当前玩家与另一个玩家的石子数量之差的最大值
    vector<vector<int>> dp(n, vector<int>(n));
    for(int i = 0; i < n; i++) {dp[i][i] = piles[i];}
    for(int i = n-2; i >= 0; i--){
        for(int j = i+1; j < n; j++){
            dp[i][j] = max(piles[i] - dp[i+1][j], piles[j] - dp[i][j-1]);
        }
    }
    return dp[0][n-1] > 0;
}

```

2. 看不懂 =.=

字符串编辑

虽然是困难，但笔面试出现频率较高

- 面试题 01.05. 一次编辑：**字符串有三种编辑操作:插入一个字符、删除一个字符或者替换一个字符。给定两个字符串，编写一个函数判定它们是否只需要一次(或者零次)编辑。
- 72. 编辑距离：**字符串有三种编辑操作:插入一个字符、删除一个字符或者替换一个字符。给你两个单词 word1 和 word2，请你计算出将 word1 转换成 word2 所使用的最少操作数。

预备题

1143. 最长公共子序列

中等 🌱 1.2k ★ ⚡

相关企业

给定两个字符串 text1 和 text2 , 返回这两个字符串的最长 公共子序列 的长度。如果不存 公共子序列 , 返回 0。

一个字符串的 子序列 是指这样一个新的字符串: 它是由原字符串在不改变字符的相对顺序的情况下删除某些字符 (也可以不删除任何字符) 后组成的新字符串。

• 例如, “ace” 是 “abcde”的子序列, 但 “aec” 不是 “abcde”的子序列。

两个字符串的 公共子序列 是这两个字符串所共同拥有的子序列。

```
int longestCommonSubsequence(string text1, string text2) {
    int s1 = text1.size(), s2 = text2.size();
    // dp[i][j]:text1的[0,i-1]与text2的[0,j-1]的最长公共子序列
    vector<vector<int>> dp(s1+1, vector<int>(s2+1, 0));
    for(int i = 1; i <= s1; i++) {
        for(int j = 1; j <= s2; 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[s1][s2];
}
```

输入: $\text{text1} = \text{"abcde"}, \text{text2} = \text{"ace"}$

dp[i][j]	a	c	e
a	0	0	0
b	0	1	1
c	0	1	2
d	0	1	2
e	0	1	2

718. 最长重复子数组 (与上题不同的是, 这一题 的重复子数组是连续的)

相关企业

给两个整数数组 nums1 和 nums2 , 返回 两个数组中 公共的、长度最长的子数组的长度。

```
int findLength(vector<int>& nums1, vector<int>& nums2) {
    int s1 = nums1.size(), s2 = nums2.size(), result = 0;
    // dp[i][j]:nums1的[0,i-1]与nums2的[0,j-1]的最长重复子数组, 且此重复子数组一定包括i-1和j-1的元素
    vector<vector<int>> dp(s1+1, vector<int>(s2+1, 0));
    for(int i = 1; i <= s1; i++) {
        for(int j = 1; j <= s2; j++){
            if(nums1[i-1] == nums2[j-1]) {dp[i][j] = dp[i-1][j-1] + 1;}
            if(dp[i][j] > result) {result = dp[i][j];}
        }
    }
    return result;
}
```

B:	3	2	1	4	7
1	0	0	0	0	0
2	0	0	0	0	0
3	0	0	0	0	0
2	0	0	0	0	0
1	0	0	0	0	0

最大长度为3 公众号: 代码教练

1. 将第2题的 return 改为 $dp[\text{len1}][\text{len2}] \leq 1$ 即可

```
int minDistance(string word1, string word2) {
    int len1 = word1.size(), len2 = word2.size();
    // dp[i][j]: 将word1[0,i-1]转换为word2[0,j-1]的方案数
    vector<vector<int>> dp(len1+1, vector<int>(len2+1));
    // 初始化
    for(int i = 1; i <= len1; i++) {dp[i][0] = i;}
    for(int j = 1; j <= len2; j++) {dp[0][j] = j;}
    for(int i = 1; i <= len1; i++) {
        for(int j = 1; j <= len2; j++){
            if(word1[i-1] == word2[j-1]){
                dp[i][j] = dp[i-1][j-1];
            }
            else{
                int Sinsert = dp[i][j-1] + 1; //插入
                int Sdelete = dp[i-1][j] + 1; //删除
                int Sreplace = dp[i-1][j-1] + 1; //替换
                dp[i][j] = min(min(Sinsert, Sdelete), Sreplace);
            }
        }
    }
    return dp[len1][len2];
}
```

题解:

<https://leetcode.cn/problems/edit-distance/solutions/188814/dong-tai-gui-hua-java-by-liweiwei1419/>

会议室

笔试经常遇到会议室题目的改造, 但很尴尬的是它们都是会员题, 会员题不放题目了。思路是前缀和, 动态规划精讲 (一) 中有这个专题。

1. 252. 会议室

2. 253. 会议室2

3. 1353. 最多可以参加的会议数目: 给你一个数组 events , 其中 $\text{events}[i] = [\text{startDay}_i, \text{endDay}_i]$, 表示会议 i 开始于 startDay_i , 结束于 endDay_i 。你可以在满足 $\text{startDay}_i \leq d \leq \text{endDay}_i$ 中的任意一天 d 参加会议 i 。注意, 一天只能参加一个会议。请你返回你可以参加的最大会议数目。

3.

```

int maxEvents(vector<vector<int>>& events) {
    int maxDay = 0; // 最晚的结束时间
    unordered_map<int, vector<int>> hash; // 记录具有相同开始时间的会议
    for(auto& event:events){
        if(event[1] > maxDay) {maxDay = event[1];}
        hash[event[0]].emplace_back(event[1]);
    }
    int ans = 0;
    priority_queue<int, vector<int>, greater<int>> que; // 结束时间由小到大排序
    for(int i = 1; i <= maxDay; i++){
        // 先将已经结束了的会议排除掉
        while(!que.empty() && que.top() < i) {que.pop();}
        // 把此刻开始召开的会议放入队列
        for(int h:hash[i]){
            que.push(h);
        }
        // 参加一个会议
        if(!que.empty()){
            ans++;
            que.pop();
        }
    }
    return ans;
}

```

状态压缩DP

- 1. 464. 我能赢吗:** 给定一个整数 `maxChoosableInteger` (整数池中可选择的最大数) 和另一个整数 `desiredTotal` (累计和)，判断先出手的玩家是否能稳赢 (假设两位玩家游戏时都表现最佳) ?
- 2. 526. 优美的排列:** 假设有从 1 到 N 的 N 个整数，如果从这 N 个数字中成功构造出一个数组，使得数组的第 i 位 ($1 \leq i \leq N$) 满足如下两个条件中的一个，我们就称这个数组为一个优美的排列。条件：第 i 位的数字能被 i 整除； i 能被第 i 位上的数字整除。现在给定一个整数 N ，请问可以构造多少个优美的排列？
- 3. 1349. 参加考试的最大学生数:** 给你一个 $m * n$ 的矩阵 `seats` 表示教室中的座位分布。如果座位是坏的 (不可用)，就用 '#' 表示；否则，用 '.' 表示。学生可以看到左侧、右侧、左上、右上这四个方向上紧邻他的学生的答卷，但是看不到直接坐在他前面或者后面的学生的答卷。请你计算并返回该考场可以容纳的一起参加考试且无法作弊的最大学生人数。学生必须坐在状况良好的座位上。

2.

利用数据范围不超过 15，我们可以使用「状态压缩 DP」进行求解。

使用一个二进制数表示当前哪些数已被选，哪些数未被选，目的是为了可以使用位运算进行加速。

我们可以通过一个具体的样例，来感受下「状态压缩」是什么意思：

例如 `(000...0101)_2` 代表值为 1 和值为 3 的数字已经被使用了，而值为 2 的节点尚未被使用。

然后再来看看使用「状态压缩」的话，一些基本的操作该如何进行：

假设变量 `state` 存放了「当前数的使用情况」，当我们需要检查值为 k 的数是否被使用时，可以使用位运算 `a = (state >> k) & 1`，来获取 `state` 中第 k 位的二进制表示，如果 `a` 为 1 代表值为 k 的数字已被使用，如果为 0 则未被访问。

二维

```

int countArrangement(int n) {
    int mask = 1<<n;
    // dp[i][state]:考虑数组前 i 个，在选择某个整数后选择整数情况为 state 下的所有方案数量
    vector<vector<int>> dp(n+1, vector<int>(mask));
    dp[0][0] = 1;
    for(int i = 1; i <= n; i++){
        for(int state = 0; state < mask; state++){
            for(int k = 1; k <= n; k++){
                // k 对应于 state 中为 1
                if(((state >> (k-1)) & 1) == 0) {continue;}
                // k 与 i 要满足整除关系
                if(k % i == 0 && i % k == 0) {continue;}
                // state & (~1 << (k-1)) 代表将 state 中数值 k 的位置置零
                dp[i][state] += dp[i-1][state & (~1 << (k-1))];
            }
        }
    }
    return dp[n][mask-1];
}

```

1.

Don't Understand ✕

3.

一维

通过对朴素的状压 DP 的分析，我们发现，在决策第 i 位的时候，理论上我们应该使用的数字数量也应该为 i 个。

但这一点在朴素状压 DP 中并没有体现，这就导致了我们在决策第 i 位的时候，仍然需要对所有的 `state` 进行计数检查（即使那些二进制表示中 1 的出现次数不为 i 个的状态）。

因此我们可以换个思路进行枚举（使用新的状态定义并优化转移方程）。

定义 `f[state]` 为当前选择数值情况为 `state` 时的所有方案的数量。

这样仍然有 `f[0] = 1` 的初始化条件，最终答案为 `f[(1 << n) - 1]`。

不失一般性考虑 `f[state]` 如何计算：

从当前状态 `state` 进行出发，检查 `state` 中的每一位 1 作为最后一个被选择的数值，这样仍然可以确保方案数「不重不漏」的被统计，同时由于我们「从小到大」对 `state` 进行枚举，因此计算 `f[state]` 所依赖的其他状态值必然都已经被计算完成。

同样的，我们仍然需要确保 `state` 中的那一位作为最后一个的 1 需要与所放的位置整除关系。

因此我们需要一个计算 `state` 的 1 的个数的方法，这里使用 `lowbit` 实现即可。

```

int countArrangement(int n) {
    int mask = 1<<n;
    // dp[state]:选择整数情况为 state 下的所有方案数量
    vector<int> dp(mask);
    dp[0] = 1;
    for(int state = 0; state < mask; state++){
        // 计算有多少个位是 1，则可以知道当前已经选择了几个整数
        int cnt = getcnt(state);
        for(int i = 0; i < n; i++){
            if(((state >> i) & 1) == 0) {continue;}
            if((i+1) % cnt != 0 && cnt % (i+1) != 0) {continue;}
            dp[state] += dp[state & (~1 << i)];
        }
    }
    return dp[mask-1];
}

int getcnt(int x){
    int ans = 0;
    while(x != 0){
        int tmp = x & -x; // 获得 x 的最低位 1 所表示的值，相当于函数 lowbit
        x -= tmp;
        ans++;
    }
    return ans;
}

```

数位DP

1. 面试题 17.06. 2 出现的次数：编写一个方法，计算从 0 到 n (含 n) 中数字 2 出现的次数。

2. 902. 最大为 N 的数字组合：我们有一组排序的数字 D，它是 {1', '2', '3', '4', '5', '6', '7', '8', '9'} 的非空子集。（请注意，'0' 不包括在内。）现在，我们用这些数字进行组合写数字，想用多少次就用多少次。例如 D = {1', '3', '5'}，我们可以写出像 '13', '551', '1351315' 这样的数字。返回可以使用 D 中的数字写出的小于或等于 N 的正整数的数目。

2. 将 n 转换成字符串 s，定义 $f(i, isLimit, isNum)$ 表示构造从左往右第 i 位及其之后位数的合法方案数，其中：

- $isLimit$ 表示当前是否受到了 n 的约束。若为真，则第 i 位填入的数字至多为 $s[i]$ ，否则至多为 9。例如 $n = 234$ ，如果前面填了 23，那么最后一位至多填 4；如果前面填的不是 23，那么最后一位至多填 9。如果在受到约束的情况下填了 $s[i]$ ，那么后续填入的数字仍会受到 n 的约束。
- $isNum$ 表示 i 前面的位数是否填了数字。若为假，则当前位置可以跳过（不填数字），或者要填入的数字至少为 1；若为真，则必须填数字，且要填入的数字从 0 开始。这样我们可以控制构造出的是一位数/两位数/三位数等等。对于本题而言，要填入的数字可直接从 digits 中选择。

题解 + 模板

<https://leetcode.cn/problems/numbers-at-most-n-given-digit-set/solutions/1900101/shu-wei-dp-tong-yong-mo-ban-xiang-xi-zhu-q5dg/>

将 n 转换成字符串 s，定义 $f(i, cnt_2, isLimit, isNum)$ 表示构造从左往右第 i 位及其之后位数中的 2 的个数。其余参数的含义：

- cnt_2 表示前面填了多少个 2。
- $isLimit$ 表示当前是否受到了 n 的约束。若为真，则第 i 位填入的数字至多为 $s[i]$ ，否则可以是 9。如果在受到约束的情况下填了 $s[i]$ ，那么后续填入的数字仍会受到 n 的约束。
- $isNum$ 表示 i 前面的位数是否填了数字。若为假，则当前位置可以跳过（不填数字），或者要填入的数字至少为 1；若为真，则要填入的数字可以从 0 开始。

后面两个参数可适用于其它数位 DP 题目。

枚举要填入的数字，具体实现逻辑见代码。对于本题来说，由于前导零对答案无影响， $isNum$ 可以省略。

题解 + 模板

<https://leetcode.cn/problems/number-of-2s-in-range-lcci/solutions/1750395/by-endlesscheng-x4mf/>

可以用贪心的高频面试题

笔试经常用这些题目进行改造。如果面试时候想用贪心的话，要说清楚为什么这题可以用贪心，说不清楚的话还是其他方法吧。这边的题目绝对纠结难度了，对初学者来说都不简单。

1. 354. 俄罗斯套娃信封问题：给定一些标有宽度和高度的信封，宽度和高度以整数对形式 (w, h) 出现。当另一个信封的宽度和高度都比这个信封大的时候，这个信封就可以放进另一个信封里，如同俄罗斯套娃一样。请计算最多能有多少个信封能组成一个“俄罗斯套娃”信封（即可以把一个信封放到另一个信封里面）。PS：本题的构造很常在笔试中出现。

2. 剑指 Offer 14 - I. 剪绳子 I：给你一根长度为 n 的绳子，请把绳子剪成整数长度的 m 段 (m, n 都是整数， $m \geq 1$ 且 $n > 1$)，每段绳子的长度记为 $k[0], k[1], \dots, k[m-1]$ 。请问 $k[0]k[1]\cdots k[m-1]$ 可能的最大乘积是多少？例如，当绳子的长度是 8 时，我们把它剪成长度分别为 2、3、3 的三段，此时得到的最大乘积是 18，答案需要取模 $10^9 + 7$ (1000000007)，如计算初始结果为 10000000000 ，请返回 1。

3. 剑指 Offer 14 - II. 剪绳子 II：给你一根长度为 n 的绳子，请把绳子剪成整数长度的 m 段 (m, n 都是整数， $n \geq 1$ 并且 $m > 1$)，每段绳子的长度记为 $k[0], k[1], \dots, k[m-1]$ 。请问 $k[0]k[1]\cdots k[m-1]$ 可能的最大乘积是多少？例如，当绳子的长度是 8 时，我们把它剪成长度分别为 2、3、3 的三段，此时得到的最大乘积是 18，答案需要取模 $10^9 + 7$ (1000000007)，如计算初始结果为 10000000000 ，请返回 1。

4. 55. 跳跃游戏：给定一个非负整数数组，你最初位于数组的第一个位置。数组中的每个元素代表你在该位置可以跳跃的最大长度。判断你是否能够到达最后一个位置。

5. 45. 跳跃游戏 II：给定一个非负整数数组，你最初位于数组的第一个位置。数组中的每个元素代表你在该位置可以跳跃的最大长度。你的目标是使用最少的跳跃次数到达数组的最后一个位置。

6. 134. 加油站：在一条环路上有 N 个加油站，其中第 i 个加油站有汽油 $gas[i]$ 升。你有一辆油箱容量无限的汽车，从第 i 个加油站开往第 $i+1$ 个加油站需要消耗汽油 $cost[i]$ 升。从你其中的一个加油站出发，开始时油箱为空，如果你可以绕环路行驶一周，则返回出发时加油站的编号，否则返回 -1。

```
int atMostNGivenDigitSet(vector<string>& digits, int n) {
    string s = to_string(n);
    int m = s.size();
    // dp[i]: 从左向右从第 i 位到最后一位生成的整数的方案数
    vector<int> dp(m, -1);
    return dfs(0, true, false, s, m, dp, digits);
}

int dfs(int i, bool is_limit, bool is_num, string& s, int& m, vector<int> dp, vector<string>& digits) {
    if(i == m) {return is_num ? 1 : 0;} // 如果填了数字，则为 1 种合法方案
    if(!is_limit && is_num && dp[i] > 0) {return dp[i];} // 在不受任何约束的情况下，返回记录的结果，避免重复运算
    int res = 0;
    if(is_num) // 前面填了数字，那么可以跳过当前位置，也不填数字
        // is_limit 改为 false，因为没填数字，位数都比 n 要短，自然不会受到 n 的约束
        // is_num 仍然为 false，因为没填任何数字
        res += ddfs(i+1, false, false, s, m, dp, digits);
    else { // 填了数字
        for(auto& d: digits) {
            if(d > up) {break;} // 超过上限，由于 digits 是有序的，后面的 d 都会超过上限，故退出循环
            // is_limit: 如果当前受到 n 的约束，且填的数字等于上限，那么后面仍然会受到 n 的约束
            // is_num: 为 true，因为填了数字
            res += ddfs(i+1, is_limit && d == up, true, s, m, dp, digits);
        }
    }
    if(is_limit && is_num) {dp[i] = res;} // 在不受任何约束的情况下，记录结果
    return res;
}
```

```
int numberOf2sInRange(int n) {
    string s = to_string(n);
    int m = s.size();
    // dp[i][cnt2]: 从左到右第 i 位到最后一位，2 出现次数为 cnt2 时的方案数
    vector<vector<int>> dp(m, vector<int>(m, -1));
    return dfs(0, 0, true, s, m, dp);
}

int dfs(int i, int cnt2, bool is_limit, string& s, int& m, vector<vector<int>>& dp) {
    if(i == m) {return cnt2;}
    if(!is_limit && dp[i][cnt2] >= 0) {return dp[i][cnt2];}
    int res = 0;
    int up = is_limit ? s[i] - '0' : 9;
    for(int d = 0; d < up; d++) {
        res += ddfs(i+1, cnt2 + (d == 2 ? 1 : 0), is_limit && d == up, s, m, dp);
    }
    if(!is_limit) {dp[i][cnt2] = res;}
    return res;
}
```

1. 贪心算法（又称 [贪心算法](#)）是指，在对问题求解时，总是做出在当前看来是最好的选择。也就是说，[不从整体最优上加以考虑，他所做出的是在某种意义上的局部最优解](#)。

2. 贪心选择是针对问题的**整体最优解**可以通过**通过一系列局部最优的选择**，即贪心选择来达到。这是贪心算法的第一个基本要素。

3. 当一个问题的最优解包含其子问题的最优解时，称此问题具有**最优子结构性质**，运用贪心策略在每一次转化后都取得了最优解。问题的**最优子结构性质**是该问题可用**动态规划法**求解的关键特征。贪心算法的每一次操作都对结果产生直接影响。贪心算法对每个子问题的解决方案都做选择，不能回退。

4. 贪心算法的基本思想是**从问题的某一个初始解出发一步一步地进行**，根据某个优化度量，**每一步都要确保能获得局部最优解**。每一步只考虑一个数据，他的选择应该满足局部优化的条件。若下一个数据和部分最优解连在一起不再是可行解时，就不把这个数据添加到部分解中。直直到所有数据收集完，或者不能再添加算法停止。

联系：

动态规划与贪心算法类似，都是将问题实例归纳为更小的、相似的子问题，并通过求解子问题产生一个全局最优解。

区别：

[贪心法](#) 选择当前最优解，而动态规划通过求解局部子问题的最优解来达到全局最优解。

7. 5.6. 合并区间：给出一个区间的集合，求合并所有重叠的区间。

8. 605. 种花问题：假设有一个很长的花坛，一部分地块种植了花，另一部分却没。可是，花不能种植在相邻的地块上，它们会争夺水源，两者都会死去。给你一个整数数组 flowerbed 表示花坛，由若干 0 和 1 组成，其中 0 表示没种花植，1 表示种植了花。另有一个数 n，能否在不打破种植规则的情况下种入 n 朵花？则返回 true，不能则返回 false。

9. 406. 根据身高重建队列：假设有一打乱顺序的一群人站成一个队列，数组 people 表示队列中一些人的属性（不一定按顺序）。每个人 $people[i] = [hi, ki]$ 表示第 i 个人的身高为 hi ，前面正好有 ki 个人身高大于或等于 hi 的人。请你重新排造并返回输入数组 people 所表示的队列。返回的队列应该格式化为数组 queue，其中 $queue[i] = [hi, ki]$ 是队列中第 i 个人的属性 ($queue[0]$ 是排在队列前面的人)。

2. 创造数组 $dp[i]$, 其中 $dp[i]$ 表示将正整数 i 拆分成至少两个正整数的和之后, 这些正整数的最大乘积。特别地, 0 不是正整数, 1 是最小的正整数, 0 和 1 都不能拆分, 因此 $dp[0] = dp[1] = 0$ 。

当 $i \geq 2$ 时, 假设对正整数 i 拆分出的第一个正整数是 j ($1 \leq j < i$) , 则有以下两种方案:

- 将 i 拆分成 j 和 $i-j$ 的和, 且 $i-j$ 不再拆分成多个正整数, 此时的乘积是 $j \times (i-j)$;
- 将 i 拆分成 j 和 $i-j$ 的和, 且 $i-j$ 继续拆分成多个正整数, 此时的乘积是 $j \times dp[i-j]$;

因此, 当 j 固定时, 有 $dp[i] = \max(j \times (i-j), j \times dp[i-j])$, 由于 j 的取值范围是 1 到 $i-1$, 需要遍历所有的 j 得到 $dp[i]$ 的最大值, 因此可以得到状态转移方程如下:

$$dp[i] = \max_{1 \leq j < i} \{ \max(j \times (i-j), j \times dp[i-j]) \}$$

最终得到 $dp[n]$ 的值即为将正整数 n 拆分成至少两个正整数的和之后, 这些正整数的最大乘积。

```
int cuttingRope(int n) {
    // dp[i]: 将整数i拆分为至少两个整数后这些整数的最大乘积
    vector<int> dp(n+1);
    dp[0] = 0; dp[1] = 1;
    for(int i = 2; i <= n; i++){
        for(int j = 1; j < i; j++){
            dp[i] = max(dp[i], max(j * (i-j), j * dp[i-j]));
        }
    }
    return dp[n];
}
```

3. 数值变得巨大, 不可用 dp, 故用贪心 由数学证明知等于3的绳子段越多, 乘积越大

```
int cuttingRope(int n) {
    if(n < 4) {return n-1;}
    long res = 1;
    while(n > 4){
        res = (res * 3) % 1000000007;
        n -= 3;
    }
    return res * n % 1000000007;
}
```

- 4.
1. 如果某一个作为 **起跳点** 的格子可以跳跃的距离是 3, 那么表示后面 3 个格子都可以作为 **起跳点**
 2. 可以对每一个能作为 **起跳点** 的格子都尝试跳一次, 把 **能跳到最远的距离** 不断更新
 3. 如果可以一直跳到最后, 就成功了

```
bool canJump(vector<int>& nums) {
    int k = 0;
    for(int i = 0; i < nums.size(); i++){
        if(i > k) {return false;}
        k = max(k, i + nums[i]);
    }
    return true;
}
```

5.

```
int jump(vector<int>& nums) {
    int ans = 0, start = 0, end = 1;
    while(end < nums.size()){
        int maxPos = 0;
        for(int i = start; i < end; i++){
            maxPos = max(maxPos, i + nums[i]); // 能跳到最远的距离
        }
        start = end; // 下一次起跳点范围开始的格子
        end = maxPos + 1; // 下一次起跳点范围结束的格子
        ans++; // 跳跃次数
    }
    return ans;
}
```

6. 实现步骤: 贪心的本质是选择每一阶段的局部最优, 从而达到全局最优。

- 遍历数组 i 从 0 开始累加 $rest[i]$, 和记为 $curSum$;
- 计算每个加油站的剩余量 $curSum + gas[i] - cost[i]$;
- 若 $curSum$ 小于零, 说明 $[0, i]$ 区间都不能作为起始位置, 起始位置从 $i+1$ 算起, $curSum$ 清零重新计算。

```
int canCompleteCircuit(vector<int>& gas, vector<int>& cost) {
    int cursum = 0, sum = 0, start = 0;
    for(int i = 0; i < gas.size(); i++){
        cursum += gas[i] - cost[i];
        sum += gas[i] - cost[i];
        if(cursum < 0){
            start = i + 1; // 更改起点
            cursum = 0;
        }
    }
    if(sum < 0) {return -1;} // 无法环绕一周
    else {return start;}
}
```

7.

```
vector<vector<int>> merge(vector<vector<int>>& intervals) {
    sort(intervals.begin(), intervals.end());
    int L = intervals[0][0], R = intervals[0][1];
    vector<vector<int>> ans;
    for(int i = 1; i < intervals.size(); i++){
        if(R >= intervals[i][0]){
            L = min(L, intervals[i][0]);
            R = max(R, intervals[i][1]);
        }
        else{
            ans.push_back({L, R});
            L = intervals[i][0];
            R = intervals[i][1];
        }
    }
    ans.push_back({L, R});
    return ans;
}
```

8.

```
bool canPlaceFlowers(vector<int>& flowerbed, int n) {
    //头尾都加上0，易于后面的判断
    flowerbed.insert(flowerbed.begin(), 0);
    flowerbed.emplace_back(0);
    for(int i = 1; i < flowerbed.size() - 1; i++){
        if(flowerbed[i] == 0 && flowerbed[i-1] == 0 && flowerbed[i+1] == 0){
            n--;
            flowerbed[i] = 1;
        }
    }
    return n <= 0 ? true : false;
}
```

9. 身高从大到小排（身高相同k小的站前面）

{7 0} {7 1} {6 1} {5 0} {5 2} {4 4}

{5 2}前面一定都比{5 2}高，那么{5 2}可以放心插入下标为2的位置。这样就确定了{5 2}前面一定有两个比它高的元素

```
vector<vector<int>> reconstructQueue(vector<vector<int>>& people) {
    sort(people.begin(), people.end(), cmp);
    vector<vector<int>> que;
    for(int i = 0; i < people.size(); i++){
        int position = people[i][1];
        que.insert(que.begin() + position, people[i]);
    }
    return que;
}

static bool cmp(const vector<int> a, const vector<int> b){
    if(a[0] == b[0]) {return a[1] < b[1];} //按下标为1的从小到大排序
    else {return a[0] > b[0];} //按下标为0的从大到小排序
}
```

掌握 cmp 的用法

12. 栈

栈的转化

232. 用栈实现队列、剑指 Offer 09. 用两个栈实现队列和面试题 03.04. 化栈为队：请你仅使用两个栈实现先入先出队列。队列应当支持一般队列的所有操作 (push、pop、peek、empty)。
225. 用队列实现栈：使用队列实现栈的下列操作：push(x) -- 元素 x 入栈；pop() -- 移除栈顶元素；top() -- 获取栈顶元素；empty() -- 返回栈是否为空。
- 面试题 03.01. 三合一：描述如何只用一个数组来实现三个栈。实现push(stackNum, value)、pop(stackNum)、isEmpty(stackNum)、peek(stackNum)方法。stackNum表示栈下标，value 表示压入的值。

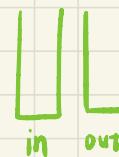
1.

```
stack<int> s_in;
stack<int> s_out;

CQueue() {
}

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

int deleteHead() {
    int tmp = 0;
    if(s_out.empty()){
        if(s_in.empty()) {return -1;}
        while(!s_in.empty()){
            tmp = s_in.top();
            s_in.pop();
            s_out.push(tmp);
        }
    }
    tmp = s_out.top();
    s_out.pop();
    return tmp;
}
```



2.

```
queue<int> q1; //主队列
queue<int> q2; //辅助队列

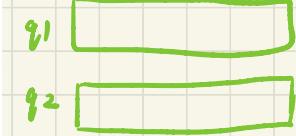
MyStack() {
}

void push(int x) {
    q2.push(x);
    while(!q1.empty()){
        q2.push(q1.front());
        q1.pop();
    }
    swap(q1, q2);
}

int pop() {
    int tmp = q1.front();
    q1.pop();
    return tmp;
}

int top() {
    return q1.front();
}

bool empty() {
    return q1.empty();
}
```



3.

```
vector<int> stack3; //前三个元素分别代表三个栈现有的元素个数
int size = 0;

TripleInOne(int stackSize) {
    size = stackSize;
    stack3.resize(3*(stackSize+1), 0);
}

void push(int stackNum, int value) {
    if(stack3[stackNum] == size) {return;} //栈已满
    stack3[3 + stackNum*size + stack3[stackNum]] = value;
    stack3[stackNum]++;
}

int pop(int stackNum) {
    if(stack3[stackNum] == 0) {return -1;}
    int tmp = stack3[3 + stackNum*size + stack3[stackNum] - 1];
    stack3[stackNum]--;
    return tmp;
}

int peek(int stackNum) {
    if(stack3[stackNum] == 0) {return -1;}
    return stack3[3 + stackNum*size + stack3[stackNum] - 1];
}

bool isEmpty(int stackNum) {
    return stack3[stackNum] == 0 ? true : false;
}
```

最小/大栈

1. 155. 最小栈、剑指 Offer 30. 包含min函数的栈和面试题 03.02. 栈的最小值：定义栈的数据结构，请在该类型中实现一个能够得到栈的最小元素的 min 函数在该栈中，调用 min、push 及 pop 的时间复杂度都是 O(1)。
2. 716. 最大栈：会员题放不了题目，感兴趣的可以看看。

1. 使用辅助栈存储最小值

```
stack<int> s;
stack<int> s_min;
MinStack() {
}

void push(int val) {
    s.push(val);
    //若s_min为空或者val小于等于s_min栈顶元素时，则s_min可以进栈
    if(s_min.empty() || val <= s_min.top()){
        s_min.push(val);
    }
}

void pop() {
    int tmp = s.top();
    s.pop();
    //如果s的栈顶元素等于s_min的栈顶元素，则s_min也得出栈
    if(tmp == s_min.top()) {s_min.pop();}
}

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

int getMin() {
    return s_min.top();
}
```

2. 与题思路一样

但题目要求多实现一个函数 `popMax()`，用于删除栈的最大值

```
int popMax() {
    int tmp=s_max.top();
    st_max.pop(); //先获得当前的最大值
    stack<int> st_tmp;
    while(st.top()!=tmp){//将最大值前面的元素弹出，保存到临时的栈中
        st_tmp.push(st.top());
        st.pop();
    }
    st.pop(); //弹出最大元素
    while(st_tmp.empty()){ //将临时栈中的元素压回到栈中
        int top_num=st_tmp.top();
        st.push(top_num);
        st.pop();
        //因为之前的最大值已经弹出，则最大值st_max也可同步的更新
        if(st_max.empty()){
            st_max.push(top_num);
        } else if(top_num>st_max.top()){
            st_max.push(top_num);
        }
    }
    return tmp; //返回最大值
}
```

注意 `st_max` 也要注意更新

验证栈序列

1. 946. 验证栈序列和剑指Offer31.栈的压入、弹出序列：给定 pushed 和 popped 两个序列，每个序列中的值都不重复，只有当它们可能是在最初空栈上进行的推入 push 和弹出 pop 操作序列的结果时，返回 true；否则，返回 false。

考虑借用一个辅助栈 `stack`，模拟压入 / 弹出操作的排列。根据是否模拟成功，即可得到结果。

- 入栈操作：按照压栈序列的顺序执行。
- 出栈操作：每次入栈后，循环判断“栈顶元素 = 弹出序列的当前元素”是否成立，将符合弹出序列顺序的栈顶元素全部弹出。

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

栈排序

1. 面试题 03.05. 栈排序：编写程序，对栈进行排序使最小元素位于栈顶。最多只能使用一个其他的临时栈存放数据，但不得将元素复制到别的数据结构（如数组）中。该栈支持如下操作：push、pop、peek 和 isEmpty。当栈为空时，peek 返回 -1。
2. 面试题 03.03. 堆盘子：设想有一堆盘子，堆太高可能会倒下来。因此，在现实生活中，盘子堆到一定高度时，我们就会另外堆一堆盘子。请实现数据结构 SetOfStacks，模拟这种行为。SetOfStacks 应该由多个栈组成，并且在前一个栈填满时新建一个栈。此外，SetOfStacks.push() 和 SetOfStacks.pop() 应该与普通栈的操作方法相同（也就是说，pop() 返回的值，应该跟只有一个栈时的情况一样）。进阶：实现一个 popAt(int index) 方法，根据指定的子栈，执行 pop 操作。

```
1 stack<int> st1;
stack<int> st2;
SortedStack() {
}

void push(int val) {
    while(!st1.empty() && st1.top() < val){
        st2.push(st1.top());
        st1.pop();
    }
    st1.push(val);
    while(!st2.empty()){
        st1.push(st2.top());
        st2.pop();
    }
}

void pop() {
    if(!st1.empty()) {st1.pop();}
}

int peek() {
    if(!st1.empty()) {return st1.top();}
    else {return -1;}
}

bool isEmpty() {
    return st1.empty();
}
```

```
2 int capacity = 0;
vector<stack<int>> starray;

StackOfPlates(int cap) {
    capacity = cap;
}

void push(int val) {
    if(capacity == 0) {return;}
    if(starray.empty() || starray.back().size() == capacity){
        starray.emplace_back(stack<int>());
    }
    starray.back().push(val);
}

int pop() {
    if(capacity == 0 || starray.empty()) {return -1;}
    int ans = starray.back().top();
    starray.back().pop();
    if(starray.back().empty()) {starray.pop_back();}
    return ans;
}

int popAt(int index) {
    if(capacity == 0 || index >= starray.size() || starray[index].empty()){
        return -1;
    }
    int ans = starray[index].top();
    starray[index].pop();
    if(starray[index].empty()) {starray.erase(starray.begin() + index);}
    return ans;
}
```

单调栈

仅列举可以用单调栈求解的题目，并不意味着最优解是单调栈，也不意味着其他解法不可行。

1. 84. 柱状图中最大的矩形：给定 n 个非负整数，用来表示柱状图中各个柱子的高度。每个柱子彼此相邻，且宽度为 1。求在该柱状图中，能够勾勒出来的矩形的最大面积。
2. 85. 最大矩形：给定一个仅包含 0 和 1、大小为 rows x cols 的二维二进制矩阵，找出只包含 1 的最大矩形，并返回其面积。
3. 221. 最大正方形：在一个由 '0' 和 '1' 组成的二维矩阵内，找到只包含 '1' 的最大正方形，并返回其面积。
4. 739. 每日温度：请根据每日气温列表，重新生成一个列表。对应位置的输出为：要想观测到更高的气温，至少需要等待的天数。如果气温在这之后都不会升高，请在该位置用 0 来代替。
5. 901. 股票价格跨度：编写一个 StockSpanner 类，它收集某些股票的每日报价，并返回该股票当日价格的跨度。
6. 496. 下一个更大元素 I：给你两个没有重复元素的数组 nums1 和 nums2，其中 nums1 是 nums2 的子集。请你找出 nums1 中每个元素在 nums2 中的下一个比其大的值。
7. 316. 去除重复字母 和 1081. 不同字符的最小子序列：给你一个字符串 s，请你去除字符串中重复的字母，使得每个字母只出现一次。需保证返回结果的字典序最小（要求不能打乱其他字符的相对位置）。
8. 402. 移掉 k 位数字：给定一个以字符串表示的非负整数 num，移除这个数中的 k 位数字，使得剩下的数字最小。
9. 907. 子数组的最小值之和：给定一个整数数组 A，找到 min(B) 的总和，其中 B 的范围为 A 的每个（连续）子数组。

1. 单调递增栈的实现

```

stack<int> st;
for(int i = 0; i < nums.size(); i++){
{
    while(!st.empty() && st.top() > nums[i])
    {
        st.pop();
    }
    st.push(nums[i]);
}
}

```

本题利用单调递增栈的特性，当遇到比栈顶元素小的，即处理栈顶元素对应的矩形高度

题解：

<https://leetcode.cn/problems/largest-rectangle-in-histogram/solutions/626295/dong-hua-yan-shi-dan-qiao-zhao-84zhu-zhu-03w3/>

```

int largestRectangleArea(vector<int> heights) {
    int ans = 0;
    stack<int> st;
    // 哨兵：前后补0
    // 1.为什么要在heights最前面加0？
    // 因为没有在heights前加0，不能保证stack不为空，所以left的值就需要赋初始值0
    // 2.为什么要在heights最后面加0？
    // 在最后加0可以保证矩形高度都是递增的特殊情况下ans也能进行计算
    heights.insert(heights.begin(), 0);
    heights.emplace_back(0);
    for(int i = 0; i < heights.size(); i++){
        //如果栈不为空且当前考察的元素值小于栈顶元素值，则表示以栈顶元素值为高的矩形面积可以确定
        while(!st.empty() && heights[i] < heights[st.top()]){
            int cur = st.top();
            st.pop();
            int curheight = heights[cur];//矩形高度
            int left = st.top();//左边界
            int right = i;//右边界
            int curwidth = right - left - 1;//矩形宽度
            ans = max(ans, curwidth * curheight);
        }
        st.push(i);
    }
    return ans;
}

```

矩形高度范围为 left 和 right 之间（不含 left 和 right）的数

2. 由第1题衍生而来

前缀和 + 单调栈

为了方便，我们令 $matrix$ 为 mat ，矩阵的行高为 n ，矩阵的列宽为 m 。

定义坐标系：左上角坐标为 $(1, 1)$ ，右下角坐标为 (n, m) 。

我们将 mat 的每一行作为基准，以基准线为起点，往上连续 1 的个数为高度。

以题目样例进行说明：

1	0	1	0	0	
1	0	1	1	1	
1	1	1	1	1	
1	0	0	1	0	

1. 红框部分代表「以第一行为基准线，统计每一列中基准线及以上连续 1 的个数」，此时只有第一行，可得：[1, 0, 1, 0, 0]

2. 黄框部分代表「以第二行作为基准线，统计每一列中基准线及以上连续 1 的个数」，此时有第一行和第二行，可得：[2, 0, 2, 1, 1]

3. 蓝框部分代表「以第三行作为基准线，统计每一列中基准线及以上连续 1 的个数」，此时有三行，可得：[3, 1, 3, 2, 2] –

将矩阵两行进行这样的转换好处理。对于原矩阵中面积最大的矩形，其下边缘必然对应了某一条基准线，从而问题转换为（题解）84. 柱状图中最大的矩形。

预处理基准线数据可以通过前缀和思想来做，构建一个二维数组 sum （为了方便，我们令二维数组下标从 1 开始）并从上往下地构造：

$$sum[i][j] = \begin{cases} 0 & mat[i][j] = 0 \\ sum[i-1][j] + 1 & mat[i][j] = 1 \end{cases}$$

当有了 sum 之后，则是和（题解）84. 柱状图中最大的矩形一样的做法：枚举高度 + 单调栈。

int maximalRectangle(vector<vector<char>>& matrix) {

```

int n = matrix.size(), m = matrix[0].size(), ans = 0;
vector<vector<int>> sum(n+1, vector<int>(m+2));
for(int i = 1; i <= n; i++){
    for(int j = 1; j < m; j++){
        sum[i][j] = (matrix[i-1][j-1] == '0' ? 0 : sum[i-1][j] + 1); => 前缀和
    }
}
for(int i = 1; i <= n; i++){
    vector<int> curline = sum[i];
    largestRectangleArea(curline, ans);
}
return ans;
}

```

int largestRectangleArea(vector<int> heights, int& ans) {

```

stack<int> st;
// heights.insert(heights.begin(), 0);
// heights.emplace_back(0);
for(int i = 0; i < heights.size(); i++){
    while(!st.empty() && heights[i] < heights[st.top()]){
        int cur = st.top();
        st.pop();
        int curheight = heights[cur];//矩形高度
        int left = st.top();//左边界
        int right = i;//右边界
        int curwidth = right - left - 1;//矩形宽度
        ans = max(ans, curwidth * curheight);
    }
    st.push(i);
}
return ans;
}

```

第 2 题代码

3. ① 单调栈 (递增)

```

int maximalSquare(vector<vector<char>>& matrix) {
    int n = matrix.size(), m = matrix[0].size(), ans = 0;
    vector<vector<int>> sum(n+1, vector<int>(m+2));
    for(int i = 1; i <= n; i++){
        for(int j = 1; j <= m; j++){
            sum[i][j] = (matrix[i-1][j-1] == '0' ? 0 : sum[i-1][j] + 1);
        }
    }
    for(int i = 1; i <= n; i++){
        vector<int> curline = sum[i];
        largestRectangleArea(curline, ans);
    }
    return ans;
}

int largestRectangleArea(vector<int>& heights, int& ans) {
    stack<int> st;
    for(int i = 0; i < heights.size(); i++){
        while(!st.empty() && heights[i] < heights[st.top()]){
            int cur = st.top();
            st.pop();
            int curheight = heights[cur]; // 矩形高度
            int left = st.top(); // 左边界
            int right = i; // 右边界
            int curwidth = right - left - 1; // 矩形宽度
            int side = min(curheight, curwidth); // 正方形边长
            ans = max(ans, side * side);
        }
        st.push(i);
    }
    return ans;
}

```

② 动态规划 (快很多)

<https://leetcode.cn/problems/maximal-square/solutions/44586/li-jie-san-zhe-qu-zui-xiao-1-by-lzhlyle/>

```

int maximalSquare(vector<vector<char>>& matrix) {
    if(matrix.size() < 1 || matrix[0].size() < 1) {return 0;}
    int height = matrix.size(), width = matrix[0].size();
    int ans = 0;
    // dp[i][j]: 以(i-1, j-1)为右下角的正方形的最大边长
    vector<vector<int>> dp(height+1, vector<int>(width+1));
    for(int row = 0; row < height; row++){
        for(int col = 0; col < width; col++){
            if(matrix[row][col] == '1'){
                dp[row+1][col+1] = min(min(dp[row][col], dp[row][col+1]), dp[row+1][col]) + 1;
                ans = max(ans, dp[row+1][col+1]);
            }
        }
    }
    return ans * ans;
}

```

4. 单调递减栈

```

vector<int> dailyTemperatures(vector<int>& temperatures) {
    vector<int> answer(temperatures.size(), 0);
    stack<int> st; // 单调递减栈
    for(int i = 0; i < temperatures.size(); i++){
        while(!st.empty() && temperatures[i] > temperatures[st.top()]){
            answer[st.top()] = i - st.top();
            st.pop();
        }
        st.push(i);
    }
    return answer;
}

```

5. 单调递减栈

```

vector<int> prices;
stack<int> st;

StockSpanner() {
    prices.emplace_back(INT_MAX); } 哨兵
    st.push(0);
}

int next(int price) {
    prices.emplace_back(price);
    while(price >= prices[st.top()]){
        st.pop();
    }
    int ans = (prices.size()-1) - st.top();
    st.push(prices.size()-1);
    return ans;
}

```

6. 我们可以先预处理 $nums_2$, 使查询 $nums_1$ 中的每个元素在 $nums_2$ 中对应位置的右边的第一个更大的元素值时不需要再遍历 $nums_2$ 。于是, 我们将题目分解为两个子问题:

- 第 1 个子问题: 如何更高效地计算 $nums_2$ 中每个元素右边的第一个更大的值;
- 第 2 个子问题: 如何存储第 1 个子问题的结果。

单调递减栈

```
vector<int> nextGreaterElement(vector<int>& nums1, vector<int>& nums2) {  
    unordered_map<int, int> hash;  
    stack<int> st;  
    for(int i = nums2.size()-1; i >= 0; i--){  
        while(!st.empty() && nums2[i] >= st.top()) {st.pop();}  
        hash[nums2[i]] = st.empty() ? -1 : st.top();  
        st.push(nums2[i]);  
    }  
    vector<int> ans;  
    for(int i = 0; i < nums1.size(); i++){  
        ans.emplace_back(hash[nums1[i]]);  
    }  
    return ans;  
}
```

8. 单调递增栈

```
string removeKdigits(string num, int k) {  
    vector<char> st; // 使用vector模拟单调栈, 方便之后答案由栈底向栈顶输出  
    for(int i = 0; i < num.size(); i++){  
        while(!st.empty() && num[i] < st.back() && k > 0){  
            st.pop_back();  
            k--;  
        }  
        st.emplace_back(num[i]);  
    }  
    // 还没删够数字  
    while(k){  
        st.pop_back();  
        k--;  
    }  
    string ans = "";  
    bool zero = true;  
    for(int i = 0; i < st.size(); i++){  
        if(zero && st[i] == '0') {continue;} // 数字开头不可以是0  
        zero = false;  
        ans += st[i];  
    }  
    return ans == "" ? "0" : ans;  
}
```

<https://leetcode.cn/problems/remove-k-digits/solutions/484949/yi-diao-kwei-shu-zl-by-leetcode-solution/>

7. 单调递增栈

```
string removeDuplicateLetters(string s) {  
    unordered_map<char, int> hash_cnt; // 记录每个数字出现次数  
    unordered_map<char, bool> hash_visited; // 记录每个数字是否可以访问  
    for(char c : s){  
        hash_cnt[c]++;  
        hash_visited[c] = false;  
    }  
    string st; // 使用字符串模拟单调栈  
    for(char c : s){  
        if(!hash_visited[c]){// 如果此字符还没有存在于st中, 则可以加入进去  
            while(!st.empty() && c < st.back()){  
                if(hash_cnt[st.back()] > 0){// 若为0则不可以弹出  
                    hash_visited[st.back()] = false;  
                    st.pop_back();  
                }  
                else{  
                    break;  
                }  
            }  
            hash_visited[c] = true;  
            st.push_back(c);  
        }  
        hash_cnt[c]--;  
    }  
    return st;  
}
```

<https://leetcode.cn/problems/remove-duplicate-letters/solutions/527359/qu-chu-zhong-fu-zi-mu-by-leetcode-soluti-yuso/>

9. 单调递增栈 四刷题