Leetcode做题笔记

哈希

1. [49. 字母异位词分组 - 力扣（LeetCode）](#)

   题目简述：将属于同一单词的不同字母排序结果整理到一起，输出一个`List<List<String>>`。

   解题思路1：对每个输入的字符串按字母排序（sort）之后，再比对已有的Hash表，来判断是否属于同一个单词。这里排序是关键，可以省去一个个字母比对的时间。**（在字符串和数组当中，当每个元素的顺序不重要时，可以使用排序后的字符串或数组作为键）**

   > - **注意，sort的关键是用`Arrays.sort(char[])`**

   解题代码：

```java
import java.util.*;

class Solution {
    public List<List<String>> groupAnagrams(String[] strs) {
        List<List<String>> result = new ArrayList<>();
        Map<String,Integer> hashmap = new HashMap<>();
        for(String str:strs){
            char[] temp = str.toCharArray();
            Arrays.sort(temp);
            String tempStr = String.valueOf(temp);
            if(hashmap.containsKey(tempStr)){
                List<String> list = result.get(hashmap.get(tempStr));
                list.add(str);
            }else {
                hashmap.put(tempStr,hashmap.size());
                List<String > newList = new ArrayList<>();
                newList.add(str);
                result.add(newList);
            }
        }

        return result;
    }
}

```

   解题思路2（未尝试）：用质数表示26个字母，把字符串的各个字母相乘，这样可保证字母异位词的乘积必定是相等的。其余步骤就是用map存储。

   解题思路3（未尝试）：

```
public List<List<String>> groupAnagrams(String[] strs) {
       return new ArrayList<>
(Arrays.stream(strs).collect(Collectors.groupingBy(s ->
s.chars().sorted().collect(StringBuilder::new,
StringBuilder::appendCodePoint,
StringBuilder::append).toString())).values());
   }
```

2. 数组去重

思路：利用set（可以set去重之后再转为数组，也可以直接用set来遍历）

```
// int[] nums
Set<Integer> set = new HashSet<>();
for(int num:nums){
    set.add(num);
}
int[] newList = new int[set.size()];
int i=0;
for(Integer n:set){
    newList[i++]=n;
}
```

3. 1. 两数之和 - 力扣（LeetCode）

题目简述：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。

解题思路（进阶版，时间复杂度为O(n)）：使用哈希表，对于每一个 `x`，我们首先查询哈希表中是否存在 `target - x`，然后将 `x` 插入到哈希表中，即可保证不会让 `x` 和自己匹配。

解题代码：

```
class Solution {
    public int[] twoSum(int[] nums, int target) {
        Map<Integer, Integer> hashtable = new HashMap<Integer, Integer>
();
        for (int i = 0; i < nums.length; ++i) {
            if (hashtable.containsKey(target - nums[i])) {
                return new int[]{hashtable.get(target - nums[i]), i};
            }
            hashtable.put(nums[i], i);
        }
        return new int[0];
    }
}
```

4. 454. 四数相加 II - 力扣（LeetCode）

题目简述：给你四个整数数组 `nums1`、`nums2`、`nums3` 和 `nums4`，数组长度都是 `n`，请你计算有多少个元组 `(i, j, k, l)` 能满足：

- `0 <= i, j, k, l < n`
- `nums1[i] + nums2[j] + nums3[k] + nums4[l] == 0`

解题思路（分组 + 哈希表，O（n^2））：

我们可以将四个数组分成两部分，A 和 B 为一组，C 和 D 为另外一组。

对于 A 和 B，我们使用二重循环对它们进行遍历，得到所有 $A[i] + B[j]$ 的值并存入哈希映射中。对于哈希映射中的每个键值对，每个键表示一种 $A[i] + B[j]$，对应的值为 $A[i] + B[j]$ 出现的次数。

对于 C 和 D，我们同样使用二重循环对它们进行遍历。当遍历到 $C[k] + D[l]$ 时，如果 $-(C[k] + D[l])$ 出现在哈希映射中，那么将 $-(C[k] + D[l])$ 对应的值累加进答案中。

最终即可得到满足 $A[i] + B[j] + C[k] + D[l] = 0$ 的四元组数目。

解题代码：

```
class Solution {
    public int fourSumCount(int[] nums1, int[] nums2, int[] nums3, int[] nums4) {
        Map<Integer,Integer>map=new HashMap<>();
        int n= nums1.length;
        int count=0;
        for(int i=0;i<n;i++){
            for(int j=0;j<n;j++){
                int sum=nums1[i]+nums2[j];
                map.put(sum,map.getOrDefault(sum,0)+1);
            }
        }
        for(int i=0;i<n;i++){
            for(int j=0;j<n;j++){
                int sum=nums3[i]+nums4[j];
                count+=map.getOrDefault(-sum,0);
            }
        }
        return count;
    }
}
```

双指针

1. 11. 盛最多水的容器 - 力扣（LeetCode）

题目简述：在横坐标上找出两条高度，使其围成的容器能够盛放最多的水（短板效应，短边为高）

解题思路：`O(N)`算法，一开始两个指针一个指向开头一个指向结尾，此时容器的底是最大的，接下来随着指针向内移动，会造成容器的底变小，在这种情况下想要让容器盛水变多，就只有在容器的高上下功夫。 那我们该如何决策哪个指针移动呢？我们能够发现不管是左指针向右移动一位，还是右指针向左移动一位，容器的底都是一样的，都比原来减少了 1。这种情况下我们想要让指针移动后的容器面积增大，就要使移动后的容器的高尽量大，所以我们选择指针所指的高较小的那个指针进行移动，这样我们就保留了容器较高的那条边，放弃了较小的那条边，以获得有更高的边的机会。

解题代码：

```
class Solution {
    public int maxArea(int[] height) {
```

```
        int maxV=0;
        // 初始情况下容器底最大
        int left=0, right=height.length-1;
        // 左右指针往里缩的时候底变小，所以要期望高变大
        while(left<right){
            int minHeight = Math.min(height[left], height[right]);
            maxV=Math.max(maxV,minHeight*(right-left));
            // 总是移动高度较小那一边的指针，以期获得更大的高
            if(height[left]<height[right]){
                left++;
            }else{
                right--;
            }
        }
        return maxV;
    }
}
```

2. 15. 三数之和 - 力扣（LeetCode）

题目简述：从一个整数数组里面找出三个数，要求这三个数加起来为0且结果不能有重复（对运行时间卡的比较死，要求做详尽的剪枝操作）

解题思路：

- 首先对数组进行排序，排序后固定一个数 $nums[i]$，再使用左右指针指向 $nums[i]$ 后面的两端，数字分别为 $nums[L]$ 和 $nums[R]$，计算三个数的和 sum 判断是否满足为 0，满足则添加进结果集
- 如果 $nums[i]$ 大于 0，则三数之和必然无法等于 0，结束循环
- 如果 $nums[i] == nums[i-1]$，则说明该数字重复，会导致结果重复，所以应该跳过
- 当 $sum == 0$ 时，$nums[L] == nums[L+1]$ 则会导致结果重复，应该跳过，$L++$
- 当 $sum == 0$ 时，$nums[R] == nums[R-1]$ 则会导致结果重复，应该跳过，$R--$
- 时间复杂度：$O(n^2)$，n 为数组长度

解题代码：不多说了，直接放上别人的优秀题解

```
class Solution {
    public static List<List<Integer>> threeSum(int[] nums) {
        List<List<Integer>> ans = new ArrayList();
        int len = nums.length;
        if(nums == null || len < 3) return ans;
        Arrays.sort(nums); // 排序
        for (int i = 0; i < len ; i++) {
            if(nums[i] > 0) break; // 如果当前数字大于0，则三数之和一定大于0，所以结束循环
            if(i > 0 && nums[i] == nums[i-1]) continue; // 去重
            int L = i+1;
            int R = len-1;
            while(L < R){
                int sum = nums[i] + nums[L] + nums[R];
                if(sum == 0){
                    ans.add(Arrays.asList(nums[i],nums[L],nums[R]));
                    while (L<R && nums[L] == nums[L+1]) L++; // 去重
                    while (L<R && nums[R] == nums[R-1]) R--; // 去重
                    L++;
```

```
                    R--;
                }
                else if (sum < 0) L++;
                else if (sum > 0) R--;
            }
        }
        return ans;
    }
}

作者：画手大鹏
链接：https://leetcode.cn/problems/3sum/solutions/12307/hua-jie-suan-fa-15-san-shu-zhi-he-by-guanpengchn/
来源：力扣（LeetCode）
著作权归作者所有。商业转载请联系作者获得授权，非商业转载请注明出处。
```

3. 42. 接雨水 - 力扣（LeetCode）

题目描述：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。

解题思路：先从左往右循环，找到高度递增的柱子，并计算相邻两根柱子之间的接雨量，此时循环完之后即可得知最高柱子的下标。然后再从右往左循环（循环到最高柱子的下标即可），按照从左往右循环相反的逻辑，找到从右往左递增的柱子，并计算相邻两根柱子之间的接雨量。如此汇总即可得出总的接雨量。

示例 1：

> **输入：** height = [0,1,0,2,1,0,1,3,2,1,2,1]
> **输出：** 6
> **解释：** 上面是由数组 [0,1,0,2,1,0,1,3,2,1,2,1] 表示的高度图，在这种情况下，可以接 6 个单位的雨水（蓝色部分表示雨水）。

示例 2：

> **输入：** height = [4,2,0,3,2,5]
> **输出：** 9

解题代码：

```
// 个人题解
class Solution {
    public int trap(int[] height) {
        int total=0;
        int len = height.length;
        int left=-1,right = len;
        if(len<3){
            return 0;
        }
```

```
        // 从左往右循环，边计算左边能接的雨水量（相对于最高柱子而言），边记录最高的柱子下标
        for(int i=0;i<len;i++){
            if(left==-1 && i<len-1 && height[i]>height[i+1]){
                left=i;
                continue;
            }
            // 从左往右递增，找到后面比现在高的柱子，计算接雨量
            if(left>=0 && height[i]>=height[left]){
                for(int j=left+1;j<i;j++){
                    total+=height[left]-height[j];
                }
                left=i;
            }
        }
        if(left<0){
            return 0;
        }
        // 此时 left 即最高的下标(只需循环到最高柱子的下标即可)，从右往左计算右边的接雨量
        for(int i=len-1;i>=left;i--){
            if(right==len && i>0 && height[i]>height[i-1]){
                right=i;
                continue;
            }
            // 右边和左边的逻辑刚好相反，从右往左递增，找到前面比现在高的柱子，计算接雨量
            if(right<len&&height[i]>=height[right]){
                for(int j=i+1;j<right;j++){
                    total+=height[right]-height[j];
                }
                right=i;
            }
        }
        return total;
    }
}
```

4. 18. 四数之和 - 力扣（LeetCode）

   题目简述：给你一个由 `n` 个整数组成的数组 `nums` ，和一个目标值 `target` 。请你找出并返回满足下述全部条件且**不重复**的四元组 `[nums[a], nums[b], nums[c], nums[d]]` （若两个四元组元素一一对应，则认为两个四元组重复）：

   - `0 <= a, b, c, d < n`
   - `a`、`b`、`c` 和 `d` **互不相同**
   - `nums[a] + nums[b] + nums[c] + nums[d] == target`

   解题思路：

为了避免枚举到重复四元组，则需要保证每一重循环枚举到的元素不小于其上一重循环枚举到的元素，且在同一重循环中不能多次枚举到相同的元素。

为了实现上述要求，可以对数组进行排序，并且在循环过程中遵循以下两点：

- 每一种循环枚举到的下标必须大于上一重循环枚举到的下标；
- 同一重循环中，如果当前元素与上一个元素相同，则跳过当前元素。

使用上述方法，可以避免枚举到重复四元组，但是由于仍使用四重循环，时间复杂度仍是 $O(n^4)$。注意到数组已经被排序，因此可以使用双指针的方法去掉一重循环。

使用两重循环分别枚举前两个数，然后在两重循环枚举到的数之后使用双指针枚举剩下的两个数。假设两重循环枚举到的前两个数分别位于下标 i 和 j，其中 $i < j$。初始时，左右指针分别指向下标 $j+1$ 和下标 $n-1$。每次计算四个数的和，并进行如下操作：

- 如果和等于 $target$，则将枚举到的四个数加到答案中，然后将左指针右移直到遇到不同的数，将右指针左移直到遇到不同的数；
- 如果和小于 $target$，则将左指针右移一位；
- 如果和大于 $target$，则将右指针左移一位。

使用双指针枚举剩下的两个数的时间复杂度是 $O(n)$，因此总时间复杂度是 $O(n^3)$，低于 $O(n^4)$。

具体实现时，还可以进行一些剪枝操作：

- 在确定第一个数之后，如果 $nums[i]+nums[i+1]+nums[i+2]+nums[i+3] > target$，说明此时剩下的三个数无论取什么值，四数之和一定大于 $target$，因此退出第一重循环；
- 在确定第一个数之后，如果 $nums[i]+nums[n-3]+nums[n-2]+nums[n-1] < target$，说明此时剩下的三个数无论取什么值，四数之和一定小于 $target$，因此第一重循环直接进入下一轮，枚举 $nums[i+1]$；
- 在确定前两个数之后，如果 $nums[i]+nums[j]+nums[j+1]+nums[j+2] > target$，说明此时剩下的两个数无论取什么值，四数之和一定大于 $target$，因此退出第二重循环；
- 在确定前两个数之后，如果 $nums[i]+nums[j]+nums[n-2]+nums[n-1] < target$，说明此时剩下的两个数无论取什么值，四数之和一定小于 $target$，因此第二重循环直接进入下一轮，枚举 $nums[j+1]$。

解题代码：

```java
// 注意一些剪枝条件，有些测试用例卡的很死
// 例如：[1000000000,1000000000,1000000000,1000000000]     -294967296
class Solution {
    public List<List<Integer>> fourSum(int[] nums, int target) {
        List<List<Integer>> quadruplets = new ArrayList<List<Integer>>
();
        if (nums == null || nums.length < 4) {
            return quadruplets;
        }
        Arrays.sort(nums);
        int length = nums.length;
        for (int i = 0; i < length - 3; i++) {
            if (i > 0 && nums[i] == nums[i - 1]) {
                continue;
            }
            if ((long) nums[i] + nums[i + 1] + nums[i + 2] + nums[i + 3]
> target) {
```

```
                break;
            }
            if ((long) nums[i] + nums[length - 3] + nums[length - 2] +
nums[length - 1] < target) {
                continue;
            }
            for (int j = i + 1; j < length - 2; j++) {
                if (j > i + 1 && nums[j] == nums[j - 1]) {
                    continue;
                }
                if ((long) nums[i] + nums[j] + nums[j + 1] + nums[j + 2]
> target) {
                    break;
                }
                if ((long) nums[i] + nums[j] + nums[length - 2] +
nums[length - 1] < target) {
                    continue;
                }
                int left = j + 1, right = length - 1;
                while (left < right) {
                    long sum = (long) nums[i] + nums[j] + nums[left] +
nums[right];
                    if (sum == target) {
                        quadruplets.add(Arrays.asList(nums[i], nums[j],
nums[left], nums[right]));
                        while (left < right && nums[left] == nums[left +
1]) {
                            left++;
                        }
                        left++;
                        while (left < right && nums[right] == nums[right
- 1]) {
                            right--;
                        }
                        right--;
                    } else if (sum < target) {
                        left++;
                    } else {
                        right--;
                    }
                }
            }
        }
        return quadruplets;
    }
}
```

5. [1574. 删除最短的子数组使剩余数组有序 - 力扣（LeetCode）]

题目简述：给你一个整数数组 `arr` ，请你删除一个子数组（可以为空），使得 `arr` 中剩下的元素是 **非递减** 的。

一个子数组指的是原数组中连续的一个子序列。请你返回满足题目要求的最短子数组的长度。

解题思路： `O(N)` 双指针

考虑删掉一个子数组后的结果：

- 假如只剩下一段（从开头或结尾删除），就要求这一段有序
- 对于一般情况，剩下两段，则要求前一段有序，后一段有序，而且分点有序(左分点<=右分点)

我们可以枚举左分点，并尝试找到离它最近的右分点。由于前后都必须有序，也就是说左分点之前必须有序，右分点之后必须有序。
可以先找到左边的有序段最远处，设为`i`，左分点只可能在0~i中产生。当左分点向前移动时，由于右段的单调性，最近右分点也一定是不会向后的。所以可以从后往前枚举左分点，同时从后往前枚举右分点。

解题代码：

```java
class Solution {
    public static int findLengthOfShortestSubarray(int[] arr) {
        int left = 0;
        int right = arr.length - 1;
        while (left < arr.length - 1 && arr[left] <= arr[left + 1]) {
            left++;
        }
        if (left == arr.length - 1) {
            return 0;
        }

        while (right >= 1 && arr[right] >= arr[right - 1]) {
            right--;
        }

        int max = Math.max(left + 1, arr.length - right);
        int rightIndex = arr.length;
        for (int i = left; i >= 0; i--) {
            while (rightIndex > right && arr[rightIndex - 1] >= arr[i])
{
                rightIndex--;
            }
            int tempMax = i + 1 + arr.length - rightIndex;
            if (tempMax > max) {
                max = tempMax;
            }
        }

        return arr.length - max;
    }
}
```

子串

1. 560. 和为 K 的子数组 - 力扣（LeetCode）

题目简述：给你一个整数数组 `nums` 和一个整数 `k` ，请你统计并返回 *该数组中和为 `k` 的子数组的个数* 。子数组是数组中元素的连续非空序列。（O(N^2)算法较易想到，下面主要介绍优化算法）

解题思路：主要介绍`O(N)`算法（**前缀和 + 哈希表优化**）

思路和算法

我们可以基于方法一利用数据结构进行进一步的优化，我们知道方法一的瓶颈在于对每个 i，我们需要枚举所有的 j 来判断是否符合条件，这一步是否可以优化呢？答案是可以的。

我们定义 $pre[i]$ 为 $[0..i]$ 里所有数的和，则 $pre[i]$ 可以由 $pre[i-1]$ 递推而来，即：

$$pre[i] = pre[i-1] + nums[i]$$

那么「$[j..i]$ 这个子数组和为 k 」这个条件我们可以转化为

$$pre[i] - pre[j-1] == k$$

简单移项可得符合条件的下标 j 需要满足

$$pre[j-1] == pre[i] - k$$

所以我们考虑以 i 结尾的和为 k 的连续子数组个数时只要统计有多少个前缀和为 $pre[i]-k$ 的 $pre[j]$ 即可。我们建立哈希表 mp，以和为键，出现次数为对应的值，记录 $pre[i]$ 出现的次数，从左往右边更新 mp 边计算答案，那么以 i 结尾的答案 $mp[pre[i]-k]$ 即可在 $O(1)$ 时间内得到。最后的答案即为所有下标结尾的和为 k 的子数组个数之和。

```java
class Solution {
    public int subarraySum(int[] nums, int k) {
        int total = 0,pre=0;
        Map<Integer,Integer> map = new HashMap<>();
        map.put(0,1);
        for(int i=0;i< nums.length;i++){
            pre += nums[i];
            if(map.containsKey(pre-k)){
                total+=map.get(pre-k);
            }
            map.put(pre, map.getOrDefault(pre,0)+1);
        }

        return  total;
    }
}
```

2. [239. 滑动窗口最大值 - 力扣（LeetCode）]

题目简述：给定一个整数数组 `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。求滑动过程中，每个窗口中的最大值

解题思路（单调队列）：

当滑动窗口向右移动时，**只要 i 还在窗口中，那么 j 一定也还在窗口中**，这是 i 在 j 的左侧所保证的。因此，由于 $nums[j]$ 的存在，$nums[i]$ **一定不会是滑动窗口中的最大值了**，我们可以将 $nums[i]$ 永久地移除。

因此我们可以使用一个队列存储所有还没有被移除的下标。在队列中，这些下标按照从小到大的顺序被存储，并且它们在数组 $nums$ 中对应的值是严格单调递减的。因为如果队列中有两个相邻的下标，它们对应的值相等或者递增，那么令前者为 i，后者为 j，就对应了上面所说的情况，即 $nums[i]$ 会被移除，这就产生了矛盾。

当滑动窗口向右移动时，我们需要把一个新的元素放入队列中。为了保持队列的性质，我们会不断地将新的元素与队尾的元素相比较，如果前者大于等于后者，那么队尾的元素就可以被永久地移除，我们将其弹出队列。我们需要不断地进行此项操作，直到队列为空或者新的元素小于队尾的元素。

由于队列中下标对应的元素是严格单调递减的，因此此时队首下标对应的元素就是滑动窗口中的最大值。但与方法一中相同的是，此时的最大值可能在滑动窗口左边界的左侧，并且随着窗口向右移动，它永远不可能出现在滑动窗口中了。因此我们还需要不断从队首弹出元素，直到队首元素在窗口中为止。

为了可以同时弹出队首和队尾的元素，我们需要使用双端队列。满足这种单调性的双端队列一般称作「单调队列」。

解题代码：

```java
import java.util.Deque;
import java.util.LinkedList;

class Solution {
    public int[] maxSlidingWindow(int[] nums, int k) {
        int[] res = new int[nums.length-k+1];
        Deque<Integer> deque = new LinkedList<>(); // 单调队列，存储下标
        for(int i=0;i<k;i++){
            while(!deque.isEmpty()&&nums[i]>nums[deque.getLast()]){
                deque.removeLast();
            }
            deque.addLast(i);
        }
        res[0]=nums[deque.getFirst()];
        for (int i=k;i< nums.length;i++){
            // 清理队尾
            while(!deque.isEmpty()&&nums[i]>nums[deque.getLast()]){
                deque.removeLast();
            }
            deque.addLast(i);
            // 判断队首是否还在滑动窗口内
            while (deque.getFirst()<=i-k){
                deque.removeFirst();
            }
            res[i-k+1]=nums[deque.getFirst()];
        }
        return res;
    }
}
```

3. [76. 最小覆盖子串 - 力扣（LeetCode）]

题目简述：给你一个字符串 `s` 、一个字符串 `t` 。返回 `s` 中涵盖 `t` 所有字符的最小子串。如果 `s` 中不存在涵盖 `t` 所有字符的子串，则返回空字符串 `""` 。

解题思路：

滑动窗口的思想：

用 `i` , `j` 表示滑动窗口的左边界和右边界，通过改变 `i` , `j` 来扩展和收缩滑动窗口，可以想象成一个窗口在字符串上游走，当这个窗口包含的元素满足条件，即包含字符串T的所有元素，记录下这个滑动窗口的长度 `j-i+1` ，这些长度中的最小值就是要求的结果。

步骤一

不断增加 `j` 使滑动窗口增大，直到窗口包含了T的所有元素

步骤二

不断增加 `i` 使滑动窗口缩小，因为是要求最小字串，所以将不必要的元素排除在外，使长度减小，直到碰到一个**必须包含的元素**，这个时候不能再扔了，再扔就不满足条件了，记录此时滑动窗口的长度，并保存最小值

步骤三

让 `i` 再增加一个位置，这个时候滑动窗口肯定不满足条件了，那么继续从**步骤一**开始执行，寻找新的满足条件的滑动窗口，如此反复，直到 `j` 超出了字符串S范围。

解题代码：

```java
class Solution {
    public String minWindow(String s, String t) {
        boolean[] exit = new boolean[52];
        int[] count = new int[52];
        for(int i=0;i<t.length();i++){
            char c=t.charAt(i);
            if(c>='a'&&c<='z'){
                exit[c-'a']=true;
                count[c-'a']++;
            }else{
                exit[c-'A'+26]=true;
                count[c-'A'+26]++;
            }
        }
        int left=0,right=0,num=0;
        int[] temp=new int[52];
        int min=Integer.MAX_VALUE;
        int begin=-1;
        while (right<s.length()){
            char c=s.charAt(right);
            if(c>='a'&&c<='z'&&exit[c-'a']){
                temp[c-'a']++;
                if(temp[c-'a']<=count[c-'a']){
                    num++;
                }
            }else if(c>='A'&&c<='Z'&&exit[c-'A'+26]){
                temp[c-'A'+26]++;
                if(temp[c-'A'+26]<=count[c-'A'+26]){
```

```
                    num++;
                }
            }
            while (num==t.length()){
                if(right-left+1<min){
                    min=right-left+1;
                    begin=left;
                }
                char op =s.charAt(left);
                if(op>='a'&&op<='z'&&exit[op-'a']){
                    temp[op-'a']--;
                    if(temp[op-'a']<count[op-'a']){
                        num--;
                    }
                }else if(op>='A'&&op<='Z'&&exit[op-'A'+26]){
                    temp[op-'A'+26]--;
                    if(temp[op-'A'+26]<count[op-'A'+26]){
                        num--;
                    }
                }
                left++;
            }
            right++;
        }
        if(min<Integer.MAX_VALUE){
            return s.substring(begin,begin+min);
        }else {
            return "";
        }
    }
}
```

数组

1. 238. 除自身以外数组的乘积 - 力扣（LeetCode）

   题目简述： 一个整数数组 `nums`，返回 *数组* `answer`，*其中* `answer[i]` *等于* `nums` *中除* `nums[i]` *之外其余各元素的乘积*。**不能使用除法，且在** `O(n)` **时间复杂度内完成此题。**

   题目思路： `answer[i]` = 其前缀积 * 后缀积

   解题代码：

```
class Solution {
    public int[] productExceptSelf(int[] nums) {
        int[] ans = new int[nums.length];
        Arrays.fill(ans,1);             // 给数组赋初值，不然全为0
        int left=1,right=1;                 // left：从左边累乘，right：从右边累乘
        // 将计算前缀积和后缀积放在一个for循环里面
        for(int i=0;i< nums.length;i++){    //最终每个元素其左右乘积进行相乘得出结果
            ans[i]*=left;                   //乘以其左边的乘积
            left*=nums[i];
```

```
            ans[nums.length-i-1]*=right;     //乘以其右边的乘积
            right*=nums[nums.length-i-1];
        }
        return ans;
    }
}
```

2. [41. 缺失的第一个正数 - 力扣（LeetCode）]

题目简述：给定一个未排序的整数数组 `nums` ，请你找出其中没有出现的最小的正整数。**（要求 `O(N)` 的时间复杂度和 `O(1)` 的空间复杂度）**

解题思路（标志法，原地算法）：先将所有的负数都表示成一个大于N的数，然后遍历数组，把所有在 [1,*N*] 范围内的数对应下标（N-1）的元素置为负数。然后再次遍历处理后的数组，得到的第一个大于0的元素其对应的整数（下标+1）即为缺失的第一个正数。（若数组元素都小于0，则说明N+1 是缺失的第一个正数）

> 我们对数组进行遍历，对于遍历到的数 x，如果它在 $[1, N]$ 的范围内，那么就将数组中的第 $x-1$ 个位置（注意：数组下标从 0 开始）打上「标记」。在遍历结束之后，如果所有的位置都被打上了标记，那么答案是 $N+1$，否则答案是最小的没有打上标记的位置加 1。

那么如何设计这个「标记」呢？由于数组中的数没有任何限制，因此这并不是一件容易的事情。但我们可以继续利用上面的提到的性质：由于我们只在意 $[1, N]$ 中的数，因此我们可以先对数组进行遍历，把不在 $[1, N]$ 范围内的数修改成任意一个大于 N 的数（例如 $N+1$）。这样一来，**数组中的所有数就都是正数了**，因此我们就可以将「标记」表示为「负号」。算法的流程如下：

- 我们将数组中所有小于等于 0 的数修改为 $N+1$；
- 我们遍历数组中的每一个数 x，它可能已经被打了标记，因此原本对应的数为 $|x|$，其中 $||$ 为绝对值符号。如果 $|x| \in [1, N]$，那么我们给数组中的第 $|x|-1$ 个位置的数添加一个负号。注意如果它已经有负号，不需要重复添加；
- 在遍历完成之后，如果数组中的每一个数都是负数，那么答案是 $N+1$，否则答案是第一个正数的位置加 1。

解题代码：

```
class Solution {
    public int firstMissingPositive(int[] nums) {
        int n = nums.length;
        for(int i=0;i<n;i++){
            if(nums[i]<=0){
                nums[i]=n+1;
            }
        }
        for(int num:nums){
            int temp = Math.abs(num);
            if(temp<=n){
                nums[temp-1]=-Math.abs(nums[temp-1]);
            }
        }
        for(int i=0;i<n;i++){
            if(nums[i]>0){
                return i+1;
            }
```

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

3. [380. O(1) 时间插入、删除和获取随机元素 - 力扣（LeetCode）](#)

题目简述：实现 `RandomizedSet` 类：

- `RandomizedSet()` 初始化 `RandomizedSet` 对象
- `bool insert(int val)` 当元素 `val` 不存在时，向集合中插入该项，并返回 `true` ；否则，返回 `false` 。
- `bool remove(int val)` 当元素 `val` 存在时，从集合中移除该项，并返回 `true` ；否则，返回 `false` 。
- `int getRandom()` 随机返回现有集合中的一项（测试用例保证调用此方法时集合中至少存在一个元素）。每个元素应该有 **相同的概率** 被返回。

你必须实现类的所有函数，并满足每个函数的 **平均** 时间复杂度为 `O(1)` 。

解题思路（变长数组 + 哈希表）：

变长数组可以在 $O(1)$ 的时间内完成获取随机元素操作，但是由于无法在 $O(1)$ 的时间内判断元素是否存在，因此不能在 $O(1)$ 的时间内完成插入和删除操作。哈希表可以在 $O(1)$ 的时间内完成插入和删除操作，但是由于无法根据下标定位到特定元素，因此不能在 $O(1)$ 的时间内完成获取随机元素操作。为了满足插入、删除和获取随机元素操作的时间复杂度都是 $O(1)$，需要将变长数组和哈希表结合，变长数组中存储元素，哈希表中存储每个元素在变长数组中的下标。

插入操作时，首先判断 val 是否在哈希表中，如果已经存在则返回 false，如果不存在则插入 val，操作如下：

1. 在变长数组的末尾添加 val；
2. 在添加 val 之前的变长数组长度为 val 所在下标 $index$，将 val 和下标 $index$ 存入哈希表；
3. 返回 true。

删除操作时，首先判断 val 是否在哈希表中，如果不存在则返回 false，如果存在则删除 val，操作如下：

1. 从哈希表中获得 val 的下标 $index$；
2. 将变长数组的最后一个元素 $last$ 移动到下标 $index$ 处，在哈希表中将 $last$ 的下标更新为 $index$；
3. 在变长数组中删除最后一个元素，在哈希表中删除 val；
4. 返回 true。

删除操作的重点在于将变长数组的最后一个元素移动到待删除元素的下标处，然后删除变长数组的最后一个元素。该操作的时间复杂度是 $O(1)$，且可以保证在删除操作之后变长数组中的所有元素的下标都连续，方便插入操作和获取随机元素操作。

获取随机元素操作时，由于变长数组中的所有元素的下标都连续，因此随机选取一个下标，返回变长数组中该下标处的元素。

解题代码：

```
class RandomizedSet {
    Map<Integer,Integer> map;
```

```
    List<Integer>list;
    Random rand;

    public RandomizedSet() {
        map=new HashMap<>();
        list=new ArrayList<>();
        rand=new Random();
    }

    public boolean insert(int val) {
        if(!map.containsKey(val)){
            map.put(val,list.size());
            list.add(val);
            return true;
        }else {
            return false;
        }
    }

    public boolean remove(int val) {
        if(map.containsKey(val)){
            int index=map.get(val);
            list.set(index, list.getLast());
            map.put(list.getLast(),index);
            map.remove(val);
            list.removeLast();
            return true;
        }else {
            return false;
        }
    }

    public int getRandom() {
        return list.get(rand.nextInt(list.size()));
    }
}
```

4. 134. 加油站 - 力扣（LeetCode）

题目简述：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。你从其中的一个加油站出发，开始时油箱为空。

给定两个整数数组 `gas` 和 `cost` ，如果你可以按顺序绕环路行驶一周，则返回出发时加油站的编号，否则返回 `-1` 。如果存在解，则 **保证** 它是 **唯一** 的。

解题思路：如果从起始加油站开始，无法走到下一个加油站，那么起始加油站到下一个加油站的所有加油站都不可能是起始加油站。因此，我们可以跳过这些加油站，将下一个加油站作为新的起始加油站，继续判断。

解题代码：

```
class Solution {
    public int canCompleteCircuit(int[] gas, int[] cost) {
        int n = gas.length;
        int totalTank = 0;
```

```
        int currTank = 0;
        int startingStation = 0;

        for (int i = 0; i < n; i++) {
            totalTank += gas[i] - cost[i];
            currTank += gas[i] - cost[i];

            // 如果当前油箱为空，则无法从当前加油站出发
            // 将下一个加油站作为新的起始加油站，并重置油箱
            if (currTank < 0) {
                startingStation = i + 1;
                currTank = 0;
            }
        }

        // 如果总油量小于总消耗量，则无法环绕一圈
        if (totalTank < 0) {
            return -1;
        }

        return startingStation;
    }
}
```

滑动窗口

1. 209. 长度最小的子数组 - 力扣（LeetCode）

   题目简述：给定一个含有 `n` 个正整数的数组和一个正整数 `target` 。

   找出该数组中满足其总和大于等于 `target` 的长度最小的 **连续子数组** `[numsl, numsl+1, ..., numsr-1, numsr]` ，并返回其长度。如果不存在符合条件的子数组，返回 `0` 。

   解题思路（前缀和 + 二分查找）：

   方法一的时间复杂度是 $O(n^2)$，因为在确定每个子数组的开始下标后，找到长度最小的子数组需要 $O(n)$ 的时间。如果使用二分查找，则可以将时间优化到 $O(\log n)$。

   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 $\text{sums}[i]$ 表示从 $\text{nums}[0]$ 到 $\text{nums}[i-1]$ 的元素和。得到前缀和之后，对于每个开始下标 i，可通过二分查找得到大于或等于 i 的最小下标 $bound$，使得 $\text{sums}[bound] - \text{sums}[i-1] \geq s$，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。

   解题代码：

```
class Solution {
    public int minSubArrayLen(int s, int[] nums) {
        int n = nums.length;
        if (n == 0) {
            return 0;
        }
```

```
        int ans = Integer.MAX_VALUE;
        int[] sums = new int[n + 1];
        // 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 target = s + sums[i - 1];
            int bound = Arrays.binarySearch(sums, target);
            if (bound < 0) {
                bound = -bound - 1;
            }
            if (bound <= n) {
                ans = Math.min(ans, bound - (i - 1));
            }
        }
        return ans == Integer.MAX_VALUE ? 0 : ans;
    }
}
```

解题思路（滑动窗口）：

在方法一和方法二中，都是每次确定子数组的开始下标，然后得到长度最小的子数组，因此时间复杂度较高。为了降低时间复杂度，可以使用滑动窗口的方法。

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 $\text{nums}[start]$ 到 $\text{nums}[end]$ 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 $\text{nums}[end]$ 加到 sum，如果 $sum \geq s$，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 $\text{nums}[start]$ 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。

解题代码：

```
class Solution {
    public int minSubArrayLen(int s, int[] nums) {
        int n = nums.length;
        if (n == 0) {
            return 0;
        }
        int ans = Integer.MAX_VALUE;
        int start = 0, end = 0;
        int sum = 0;
        while (end < n) {
            sum += nums[end];
            while (sum >= s) {
                ans = Math.min(ans, end - start + 1);
                sum -= nums[start];
                start++;
            }
            end++;
```

```
        }
        return ans == Integer.MAX_VALUE ? 0 : ans;
    }
}
```

2. [30. 串联所有单词的子串 - 力扣（LeetCode）]

题目简述：给定一个字符串 `s` 和一个字符串数组 `words`。 `words` 中所有字符串 **长度相同。**

`s` 中的 **串联子串** 是指一个包含 `words` 中所有字符串以任意顺序排列连接起来的子串。

- 例如，如果 `words = ["ab","cd","ef"]`， 那么 `"abcdef"`， `"abefcd"`，`"cdabef"`，`"cdefab"`，`"efabcd"`， 和 `"efcdab"` 都是串联子串。 `"acdbef"` 不是串联子串，因为他不是任何 `words` 排列的连接。

返回所有串联子串在 `s` 中的开始索引。你可以以 **任意顺序** 返回答案。

解题思路：

记 $words$ 的长度为 m，$words$ 中每个单词的长度为 n，s 的长度为 ls。首先需要将 s 划分为单词组，每个单词的大小均为 n（首尾除外）。这样的划分方法有 n 种，即先删去前 i（$i = 0 \sim n-1$）个字母后，将剩下的字母进行划分，如果末尾有不到 n 个字母也删去。对这 n 种划分得到的单词数组分别使用滑动窗口对 $words$ 进行类似于「字母异位词」的搜寻。

划分成单词组后，一个窗口包含 s 中前 m 个单词，用一个哈希表 $differ$ 表示窗口中单词频次和 $words$ 中单词频次之差。初始化 $differ$ 时，出现在窗口中的单词，每出现一次，相应的值增加 1，出现在 $words$ 中的单词，每出现一次，相应的值减少 1。然后将窗口右移，右侧会加入一个单词，左侧会移出一个单词，并对 $differ$ 做相应的更新。窗口移动时，若出现 $differ$ 中值不为 0 的键的数量为 0，则表示这个窗口中的单词频次和 $words$ 中单词频次相同，窗口的左端点是一个待求的起始位置。划分的方法有 n 种，做 n 次滑动窗口后，即可找到所有的起始位置。

解题代码（官方题解）：

```
class Solution {
    public List<Integer> findSubstring(String s, String[] words) {
        List<Integer> res = new ArrayList<>();
        // 所有单词的个数
        int num = words.length;
        // 每个单词的长度（是相同的）
        int wordLen = words[0].length();
        // 字符串长度
        int stringLen = s.length();

        for (int i = 0; i < wordLen; i++) {
            // 遍历的长度超过了整个字符串的长度，退出循环
            if (i + num * wordLen > stringLen) {
                break;
            }
            // differ表示窗口中的单词频次和words中的单词频次之差
            Map<String, Integer> differ = new HashMap<>();
            // 初始化窗口，窗口长度为num * wordLen,依次计算窗口里每个切分的单词的频次
            for (int j = 0; j < num; j++) {
                String word = s.substring(i + j * wordLen, i + (j + 1) * wordLen);
                differ.put(word, differ.getOrDefault(word, 0) + 1);
            }
            // 遍历words中的word，对窗口里每个单词计算差值
```

```
            for (String word : words) {
                differ.put(word, differ.getOrDefault(word, 0) - 1);
                // 差值为0时，移除掉这个word
                if (differ.get(word) == 0) {
                    differ.remove(word);
                }
            }
            // 开始滑动窗口
            for (int start = i; start < stringLen - num * wordLen + 1; start += wordLen) {
                if (start != i) {
                    // 右边的单词滑进来
                    String word = s.substring(start + (num - 1) * wordLen, start + num * wordLen);
                    differ.put(word, differ.getOrDefault(word, 0) + 1);
                    if (differ.get(word) == 0) {
                        differ.remove(word);
                    }
                    // 左边的单词滑出去
                    word = s.substring(start - wordLen, start);
                    differ.put(word, differ.getOrDefault(word, 0) - 1);
                    if (differ.get(word) == 0) {
                        differ.remove(word);
                    }
                    word = s.substring(start - wordLen, start);
                }
                // 窗口匹配的单词数等于words中对应的单词数
                if (differ.isEmpty()) {
                    res.add(start);
                }
            }
        }
        return res;
    }
}
```

解题代码（个人版，用时长）：

```
class Solution {
    Map<String,Integer>base=new HashMap<>();
    public List<Integer> findSubstring(String s, String[] words) {
        List<Integer>res=new ArrayList<>();
        for(String str:words){
            base.put(str,base.getOrDefault(str,0)+1);
        }
        int cnt= words.length;
        int len=words[0].length();
        int totalLen=cnt*len;
        if(s.length()<totalLen){
            return res;
        }
        for(int i=0;i+totalLen-1<s.length();i++){
            if(check(s.substring(i,i+totalLen),len,cnt)){
                res.add(i);
            }
```

```
        }
        return res;
    }

    private boolean check(String s,int len,int cnt){
        Map<String,Integer>map=new HashMap<>();
        int total=len*cnt;
        int sum=0;
        for(int i=0;i<total;i+=len){
            String str=s.substring(i,i+len);
            if(base.containsKey(str)){
                map.put(str,map.getOrDefault(str,0)+1);
                if(map.get(str)<=base.get(str)){
                    sum++;
                }else{
                    return false;
                }
            }else {
                return false;
            }
        }
        if(sum==cnt){
            return true;
        }else{
            return false;
        }
    }
}
```

矩阵

1. 240. 搜索二维矩阵 II - 力扣（LeetCode）

   题目简述：编写一个高效的算法来搜索 `m x 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。

   解题思路：这题不是很难，下面提供一种我没想到的巧妙思路（**Z 字形查找**，时间复杂度 `O(m+n)`，空间复杂度 `O(1)`）

方法三：Z 字形查找

思路与算法

我们可以从矩阵 $matrix$ 的右上角 $(0, n-1)$ 进行搜索。在每一步的搜索过程中，如果我们位于位置 (x, y)，那么我们希望在以 $matrix$ 的左下角为左下角、以 (x, y) 为右上角的矩阵中进行搜索，即行的范围为 $[x, m-1]$，列的范围为 $[0, y]$：

- 如果 $matrix[x, y] = target$，说明搜索完成；
- 如果 $matrix[x, y] > target$，由于每一列的元素都是升序排列的，那么在当前的搜索矩阵中，所有位于第 y 列的元素都是严格大于 $target$ 的，因此我们可以将它们全部忽略，即将 y 减少 1；
- 如果 $matrix[x, y] < target$，由于每一行的元素都是升序排列的，那么在当前的搜索矩阵中，所有位于第 x 行的元素都是严格小于 $target$ 的，因此我们可以将它们全部忽略，即将 x 增加 1。

在搜索的过程中，如果我们超出了矩阵的边界，那么说明矩阵中不存在 $target$。

解题代码：

```java
class Solution {
    public boolean searchMatrix(int[][] matrix, int target) {
        int m= matrix.length,n=matrix[0].length;
        int i=0,j=n-1;
        if(m==0||n==0) {
            return false;
        }
        // 从矩阵的右上角开始搜索
        while(i<m && j>=0){
            if(target==matrix[i][j]){
                return true;
            }
            if(target>matrix[i][j]){
                i++;
            }else{
                j--;
            }
        }

        return false;
    }
}
```

链表

- 146. LRU 缓存 - 力扣（LeetCode）

  题目简述：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。要求实现 `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。

解题思路：哈希表 + 双向链表。双向链表按照被使用的顺序存储了这些键值对，靠近头部的键值对是最近使用的，而靠近尾部的键值对是最久未使用的。哈希表即为普通的哈希映射（HashMap），通过缓存数据的键映射到其在双向链表中的位置。

解题代码：

```java
public class LRUCache {
    class DLinkedNode {
        int key;
        int value;
        DLinkedNode prev;
        DLinkedNode next;
        public DLinkedNode() {}
        public DLinkedNode(int _key, int _value) {key = _key; value = _value;}
    }

    private Map<Integer, DLinkedNode> cache = new HashMap<Integer, DLinkedNode>();
    private int size;
    private int capacity;
    private DLinkedNode head, tail;

    public LRUCache(int capacity) {
        this.size = 0;
        this.capacity = capacity;
        // 使用伪头部和伪尾部节点
        head = new DLinkedNode();
        tail = new DLinkedNode();
        head.next = tail;
        tail.prev = head;
    }

    public int get(int key) {
        DLinkedNode node = cache.get(key);
        if (node == null) {
            return -1;
        }
        // 如果 key 存在，先通过哈希表定位，再移到头部
        moveToHead(node);
        return node.value;
    }

    public void put(int key, int value) {
        DLinkedNode node = cache.get(key);
        if (node == null) {
            // 如果 key 不存在，创建一个新的节点
            DLinkedNode newNode = new DLinkedNode(key, value);
            // 添加进哈希表
            cache.put(key, newNode);
```

```
            // 添加至双向链表的头部
            addToHead(newNode);
            ++size;
            if (size > capacity) {
                // 如果超出容量，删除双向链表的尾部节点
                DLinkedNode tail = removeTail();
                // 删除哈希表中对应的项
                cache.remove(tail.key);
                --size;
            }
        }
        else {
            // 如果 key 存在，先通过哈希表定位，再修改 value，并移到头部
            node.value = value;
            moveToHead(node);
        }
    }

    private void addToHead(DLinkedNode node) {
        node.prev = head;
        node.next = head.next;
        head.next.prev = node;
        head.next = node;
    }

    private void removeNode(DLinkedNode node) {
        node.prev.next = node.next;
        node.next.prev = node.prev;
    }

    private void moveToHead(DLinkedNode node) {
        removeNode(node);
        addToHead(node);
    }

    private DLinkedNode removeTail() {
        DLinkedNode res = tail.prev;
        removeNode(res);
        return res;
    }
}
```

二叉树

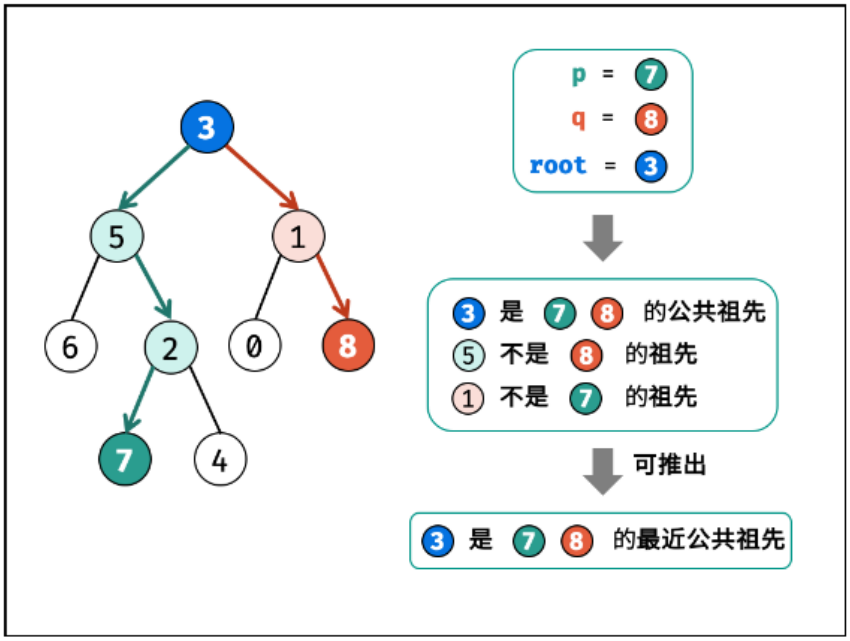
1. 236. 二叉树的最近公共祖先 - 力扣（LeetCode）

   题目简述：找到二叉树两个结点的最近公共父节点

   解题思路（递归）：

根据以上定义，若 $root$ 是 p,q 的 **最近公共祖先** ，则只可能为以下情况之一：

1. p 和 q 在 $root$ 的子树中，且分列 $root$ 的 **异侧**（即分别在左、右子树中）；
2. $p = root$ ，且 q 在 $root$ 的左或右子树中；
3. $q = root$ ，且 p 在 $root$ 的左或右子树中；



解题代码（递归）：

```java
// 未简化版
class Solution {
    public TreeNode lowestCommonAncestor(TreeNode root, TreeNode p, TreeNode q) {
        if (root == null || root == p || root == q) {
            //只要当前根节点是p和q中的任意一个，就返回（因为不能比这个更深了，再深p和q中的一个就没了）
            return root;
        }
        //根节点不是p和q中的任意一个，那么就继续分别往左子树和右子树找p和q
        TreeNode left = lowestCommonAncestor(root.left, p, q);
        TreeNode right = lowestCommonAncestor(root.right, p, q);
        //p和q都没找到，那就没有
        if(left == null && right == null) {
            return null;
        }
        //左子树没有p也没有q，就返回右子树的结果
        if (left == null) {
            return right;
        }
        //右子树没有p也没有q就返回左子树的结果
        if (right == null) {
            return left;
        }
        //左右子树都找到p和q了，那就说明p和q分别在左右两个子树上，所以此时的最近公共祖先就是root
        return root;
```

```
    }
}


// 简化版!
class Solution {
    public TreeNode lowestCommonAncestor(TreeNode root, TreeNode p, TreeNode q) {
        if (root == null || p == root || q == root) return root;
        TreeNode left = lowestCommonAncestor(root.left, p, q);
        TreeNode right = lowestCommonAncestor(root.right, p, q);
        if (left != null && right != null) return root;
        return left != null ? left : right;
    }
}
```

解题思路（存储父节点）： 哈希表存储所有节点的父节点，然后我们就可以利用节点的父节点信息从 p 结点开始不断往上跳，并记录已经访问过的节点，再从 q 节点开始不断往上跳，如果碰到已经访问过的节点，那么这个节点就是我们要找的最近公共祖先。

算法：

1. 从根节点开始遍历整棵二叉树，用哈希表记录每个节点的父节点指针。
2. 从 p 节点开始不断往它的祖先移动，并用数据结构记录已经访问过的祖先节点。
3. 同样，我们再从 q 节点开始不断往它的祖先移动，如果有祖先已经被访问过，即意味着这是 p 和 q 的深度最深的公共祖先，即 LCA 节点。

解题代码（存储父节点）：

```
class Solution {
    Map<Integer, TreeNode> parent = new HashMap<Integer, TreeNode>();
    Set<Integer> visited = new HashSet<Integer>();

    public void dfs(TreeNode root) {
        if (root.left != null) {
            parent.put(root.left.val, root);
            dfs(root.left);
        }
        if (root.right != null) {
            parent.put(root.right.val, root);
            dfs(root.right);
        }
    }

    public TreeNode lowestCommonAncestor(TreeNode root, TreeNode p, TreeNode q) {
        dfs(root);
        while (p != null) {
            visited.add(p.val);
            p = parent.get(p.val);
        }
        while (q != null) {
            if (visited.contains(q.val)) {
                return q;
```

```
            }
            q = parent.get(q.val);
        }
        return null;
    }
}
```

2. [124. 二叉树中的最大路径和 - 力扣（LeetCode）]

题目简述：二叉树中的 **路径** 被定义为一条节点序列，序列中每对相邻节点之间都存在一条边。同一个节点在一条路径序列中 **至多出现一次** 。该路径 **至少包含一个** 节点，且不一定经过根节点。**路径和** 是路径中各节点值的总和。给你一个二叉树的根节点 `root` ，返回其 **最大路径和** 。

解题思路：

1、那么，首先我们可以假设走到了某个节点，现在要面临的问题是路径的最大值问题，显然对于这种问题，每遍历到一个节点，我们都要求出包含该节点在内的此时的最大路径，并且在之后的遍历中更新这个最大值。对于该节点来说，它的最大路径currpath就等于左右子树的最大路径加上本身的值，也就是currpath = left+right+node.val，但是有一个前提，我们要求的是最大路径，所以若是left或者right小于等于0了，那么我们就没有必要把这些值加上了，因为加上一个负数，会使得最大路径变小。这里的最大路径中的最其实就是一个限定条件，也就是我们常说的贪心算法，只取最大，最好，其余的直接丢弃。

2、好了，1中的主体我们已经明确了，但是还存在一个问题，那就是left和right具体应该怎么求，也就是left和right的递归形式。显然我们要把node.left和node.right再次传输到递归函数中，重复上述的操作。但如果到达了叶子节点，是不是需要往上一层返回了呢？那么返回值又是多少呢？ 我们要明确left和right的基本含义，它们表示的是最大贡献，那么一个节点的最大贡献就等于node.val+max(left,right)，这个节点本身选上，然后从它的左右子树中选择最大的那个加上。 对于叶子节点也是这样，但是叶子节点的左右子树都为空，所以加上0，哎，注意看，此时是不是边界条件也出来了，但节点为空时，返回0 。 好了，至此循环的主体，返回值，边界条件都定义好了，那么整个递归的代码是不是就水到渠成了。这样一看递归也没什么了不起的！！！

ps:除了向下思考，其实也可以向上思考，比如还是遍历到了某个节点，那么这个节点向上一层走，是不是要有一个返回值呢，那么返回值是什么呢？是不是和自己原来需要的right(or left)相同，只不过现在轮到自己了，自己原来需要最大贡献，那么此时返回时就返回最大贡献，自己的最大贡献不就是node.val+max(left,right)。就像是老板一层一层的压榨员工一样。

不用看官方题解，那么复杂。 所有树的题目，都想成一颗只有根、左节点、右节点 的小树。然后一颗颗小树构成整棵大树，所以只需要考虑这颗小树即可。接下来分情况， 按照题意：一颗三个节点的小树的结果只可能有如下6种情况：

1. 根 + 左 + 右
2. 根 + 左
3. 根 + 右
4. 根
5. 左
6. 右

好了，分析上述6种情况， 只有 2,3,4 可以向上累加，而1,5,6不可以累加（这个很好想，情况1向上累加的话，必然出现分叉，情况5和6直接就跟上面的树枝断开的，没法累加），所以我们找一个全局变量存储 1,5,6这三种不可累加的最大值， 另一方面咱们用遍历树的方法求2,3,4这三种可以累加的情况。 最后把两类情况得到的最大值再取一个最大值即可。

解题代码：

```
class Solution {
    int maxSum = Integer.MIN_VALUE;

```

```
    public int maxPathSum(TreeNode root) {
        if (root == null) {
            return 0;
        }
        maxGain(root);
        return maxSum;
    }

    /**
     * 返回经过root的单边分支最大和， 即Math.max(root, root+left, root+right)
     * @param root
     * @return
     */
    public int maxGain(TreeNode node) {
        if (node == null) {
            return 0;
        }
        //计算左边分支最大值，左边分支如果为负数还不如不选择
        int leftMax = Math.max(0, maxGain(root.left));
        //计算右边分支最大值，右边分支如果为负数还不如不选择
        int rightMax = Math.max(0, maxGain(root.right));
        //left->root->right 作为路径与已经计算过历史最大值做比较
        maxSum = Math.max(maxSum, root.val + leftMax + rightMax);
        // 返回经过root的单边最大分支给当前root的父节点计算使用
        return root.val + Math.max(leftMax, rightMax);
    }
}
```

图论

1. 208. 实现 Trie (前缀树) - 力扣（LeetCode）

   题目简述：实现字典树这一数据结构：

   - `Trie()` 初始化前缀树对象。
   - `void insert(String word)` 向前缀树中插入字符串 `word` 。
   - `boolean search(String word)` 如果字符串 `word` 在前缀树中，返回 `true` （即，在检索之前已经插入）；否则，返回 `false` 。
   - `boolean startsWith(String prefix)` 如果之前已经插入的字符串 `word` 的前缀之一为 `prefix` ，返回 `true` ；否则，返回 `false` 。

   解题思路：208. 实现 Trie (前缀树) - 力扣（LeetCode）

   解题代码：

```
class Trie {
        boolean isEnd;
        Trie[] next;

        public Trie() {
            next=new Trie[26];
            isEnd = false;
        }
```

```
9
10          public void insert(String word) {
11              Trie node = this;
12              for(char c:word.toCharArray()){
13                  if(node.next[c-'a']==null){
14                      node.next[c-'a']=new Trie();
15                  }
16                  node=node.next[c-'a'];
17              }
18              node.isEnd=true;
19          }
20
21          public boolean search(String word) {
22              Trie node = this;
23              for(char c:word.toCharArray()){
24                  if(node.next[c-'a']==null){
25                      return false;
26                  }
27                  node=node.next[c-'a'];
28              }
29              return node.isEnd;
30          }
31
32          public boolean startsWith(String prefix) {
33              Trie node = this;
34              for(char c:prefix.toCharArray()){
35                  if(node.next[c-'a']==null){
36                      return false;
37                  }
38                  node=node.next[c-'a'];
39              }
40              return true;
41          }
42      }
43
44  /**
45   * Your Trie object will be instantiated and called as such:
46   * Trie obj = new Trie();
47   * obj.insert(word);
48   * boolean param_2 = obj.search(word);
49   * boolean param_3 = obj.startsWith(prefix);
50   */
```

回溯

回溯算法模版：

```
void backtracking(参数) {
    if (终止条件) {
        存放结果;
        return;
    }

    for (选择：本层集合中元素（树中节点孩子的数量就是集合的大小）) {
        处理节点;
        backtracking(路径，选择列表); // 递归
        回溯，撤销处理结果
    }
}
```

1. 78. 子集 - 力扣（LeetCode）

   题目简述：给定一个整数数组 `nums` ，数组中的元素 **互不相同** 。返回该数组所有可能的子集（幂集）。解集 **不能** 包含重复的子集。

   解法一（迭代）：可以把数组中的每一位与二进制串的一位相对应，用0和1来表示对应的整数是否出现在子集中，通过遍历0到2^n-1即可得出所有子集。

   ```
   class Solution {
       List<Integer> t = new ArrayList<Integer>();
       List<List<Integer>> ans = new ArrayList<List<Integer>>();

       public List<List<Integer>> subsets(int[] nums) {
           int n = nums.length;
           for (int mask = 0; mask < (1 << n); ++mask) {
               t.clear();
               for (int i = 0; i < n; ++i) {
                   if ((mask & (1 << i)) != 0) {
                       t.add(nums[i]);
                   }
               }
               // 要新建一个List来存储结果
               ans.add(new ArrayList<Integer>(t));
           }
           return ans;
       }
   }
   ```

   解法二（递归官方版）：采用深度优先搜索来递归，`dfs(cur,nums)`中的`cur`表示当前位置，通过根据当前位置对应的整数是否出现在子集中，可以分为两条支路，从而进行`dfs`递归，直到`cur==nums.length`，此时表示数组中的所有整数是否出现都已经被确认，然后将数组记录下来。

   ```
   class Solution {
       List<Integer> t = new ArrayList<Integer>();
       List<List<Integer>> ans = new ArrayList<List<Integer>>();

       public List<List<Integer>> subsets(int[] nums) {
           dfs(0, nums);
           return ans;
       }

   ```

```
    public void dfs(int cur, int[] nums) {
        // 注意是nums.length而不是nums.length-1
        if (cur == nums.length) {
            // 记录答案（注意要新建一个List来存储结果，不然后面的操作会修改之前存入的答案）
            ans.add(new ArrayList<Integer>(t));
            return;
        }
        // 考虑选择当前位置
        t.add(nums[cur]);
        dfs(cur + 1, nums);
        // 考虑不选择当前位置
        t.remove(t.size() - 1);
        dfs(cur + 1, nums);
    }
}
```

解法三（自己写的递归）：为了避免添加重复的子集，规定子集中的整数要满足递增的顺序，否则就不添加到结果中。（在递归的每一层都可能产生新的子集添加到结果中，因为子集的大小从0增加到数组的大小）

```
class Solution {
    List<List<Integer>> res = new ArrayList<>();
    public List<List<Integer>> subsets(int[] nums) {
        // 记录数组中的整数是否已经被添加到子集中
        boolean[] visited = new boolean[nums.length];
        Arrays.fill(visited,false);
        res.add(new ArrayList<>());
        // 先排序
        Arrays.sort(nums);
        backtrack(nums,new ArrayList<>(),visited);
        return res;
    }

    private void backtrack(int[] nums,List<Integer> list,boolean[] visited){
        // 先判断子集的大小是否已经最大
        if(list.size()== nums.length){
            return;
        }
        for(int i=0;i< nums.length;i++){
            if(!visited[i]){
                if(list.size()==0||nums[i]> list.get(list.size()-1)){// 确保子集中的元素递增
                    // 新创建一个List，作为存储新子集的容器
                    List<Integer> newList = new ArrayList<>(list);
                    newList.add(nums[i]);
                    visited[i]=true;
                    res.add(newList);
                    backtrack(nums,newList,visited);
                    // 回溯完撤销之前的操作
                    visited[i]=false;
                }
            }
        }
```

```
33
34      }
35  }
```

2. 491. 非递减子序列 - 力扣（LeetCode）

题目简述：给你一个整数数组 `nums` ，找出并返回所有该数组中不同的递增子序列，递增子序列中 **至少有两个元素** 。你可以按 **任意顺序** 返回答案。数组中可能含有重复元素，如出现两个整数相等，也可以视作递增序列的一种特殊情况。

解法注意点：回溯过程中同层节点不能重复，所以需要用set来记录已遍历过的的节点来去重（因为没有排序，所以可能出现相同的元素分散遍历，此时通过常规比较前后元素是否相同无法达到去重的效果，因此需要用set来记录）

详解链接：代码随想录 (programmercarl.com)

解题代码：

```java
class Solution {
    List<List<Integer>> list = new ArrayList<>();
    // List<Integer> tempList = new ArrayList<>();

    public List<List<Integer>> findSubsequences(int[] nums) {
        backTracking(nums, new ArrayList<>(), 0);
        return list;
    }

    void backTracking(int[] nums, List<Integer> tempList, int start) {
        if (tempList.size() >= 2) {
            list.add(new ArrayList<>(tempList));
        }
        Set<Integer> set = new HashSet<>(); // 记录同一层已经遍历过的元素
        for (int i = start; i < nums.length; i++) {
            set.add(nums[i]);
            if (tempList.size() == 0 || (tempList.size() > 0 && nums[i]
>= tempList.get(tempList.size() - 1))) {
                tempList.add(nums[i]);
                backTracking(nums, tempList, i + 1);
                tempList.remove(tempList.size() - 1);
            }
            while (i < nums.length - 1 && set.contains(nums[i+1])) {
                i++; // 同一层已经遍历过的元素需要跳过
            }
        }
    }
}
```

3. 332. 重新安排行程 - 力扣（LeetCode）

题目简述：

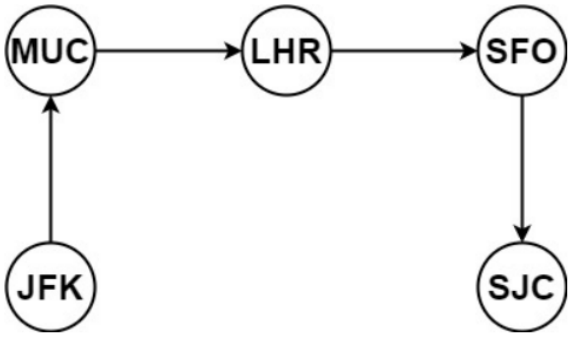
给你一份航线列表 `tickets` ，其中 `tickets[i] = [from_i, to_i]` 表示飞机出发和降落的机场地点。请你对该行程进行重新规划排序。

所有这些机票都属于一个从 `JFK` （肯尼迪国际机场）出发的先生，所以该行程必须从 `JFK` 开始。如果存在多种有效的行程，请你按字典排序返回最小的行程组合。

- 例如，行程 `["JFK", "LGA"]` 与 `["JFK", "LGB"]` 相比就更小，排序更靠前。

假定所有机票至少存在一种合理的行程。且所有的机票 必须都用一次 且 只能用一次。

示例 1：



> **输入：** `tickets = [["MUC","LHR"],["JFK","MUC"],["SFO","SJC"],["LHR","SFO"]]`
> **输出：** `["JFK","MUC","LHR","SFO","SJC"]`

解题思路：这题类似于dfs（深度优先搜索），我的思路是用Map存储所有机票信息（设为Map<String, Map<String, Integer>>，要注意可能会出现多张相同的机票，所以要用Integer而不是Boolean），然后走常规回溯思路。直接看代码：

代码随想录详解链接：代码随想录 (programmercarl.com)

解题代码一（我的常规代码，会超时）：

```
public class Solution{
    List<String> res = new ArrayList<>();

    Map<String, List<String>> map = new HashMap<>();
    Map<String, Map<String, Integer>> visitMap = new HashMap<>();

    public List<String> findItinerary(List<List<String>> tickets) {
        for (List<String> ticket : tickets) {
            String key = ticket.get(0);
            String val = ticket.get(1);
            if (map.containsKey(key)) {
                map.get(key).add(val);
                Map<String, Integer> countMap = visitMap.get(key);
                visitMap.get(key).put(val, countMap.getOrDefault(val, 0)
+ 1);
            } else {
                List<String> list = new ArrayList<>();
                list.add(val);
                Map<String, Integer> flagMap = new HashMap<>();
                flagMap.put(val, 1);
                map.put(key, list);
                visitMap.put(key, flagMap);
            }
        }
```

```
        for (Map.Entry<String, List<String>> entry : map.entrySet()) {
            Collections.sort(entry.getValue());
        }
        res.add("JFK");
        backTracking(tickets.size() + 1, "JFK");
        return res;
    }

    boolean backTracking(int count, String key) {
        if (res.size() == count) {
            return true;
        }
        if (!map.containsKey(key)) {
            return false;
        }
        for (String val : map.get(key)) {
            if (visitMap.get(key).get(val) == 0) {
                continue;
            }
            res.add(val);
            visitMap.get(key).put(val, visitMap.get(key).get(val) - 1);
            if (backTracking(count, val)) {
                return true;
            } else {
                res.remove(res.size() - 1);
                visitMap.get(key).put(val, visitMap.get(key).get(val) +
1);
            }
        }
        return false;
    }
}
```

> 超时原因是因为不仅遍历了tickets把数据存到map中，还在最后进行了排序，不仅空间复杂度大，时间复杂度也提高了。优化思路，把两个map合起来，直接用 Map<String, Map<String, Integer>> （或直接用Map<String, TreeMap<String, Integer>>）存储，其中value 的类型为TreeMap，这样在存储的时候会自动按字典序排序，下面是优化的代码

解题代码二（优化版）：

```
class Solution {
    List<String> res = new ArrayList<>();

    Map<String, Map<String, Integer>> map = new HashMap<>();

    public List<String> findItinerary(List<List<String>> tickets) {
        for (List<String> ticket : tickets) {
            String key = ticket.get(0);
            String val = ticket.get(1);
            if (map.containsKey(key)) {
                Map<String, Integer> treeMap = map.get(key);
                map.get(key).put(val, treeMap.getOrDefault(val, 0) + 1);
            } else {
                // 用TreeMap来存储，这样会根据key来排序（达到了自动排序的效果）
                TreeMap<String, Integer> treeMap = new TreeMap<>();
```

```
                treeMap.put(val, 1);
                map.put(key, treeMap);
            }
        }
        res.add("JFK");
        backTracking(tickets.size() + 1, "JFK");
        return res;
    }

    boolean backTracking(int size, String key) {
        if (res.size() == size) {
            return true;
        }
        if (!map.containsKey(key)) {
            return false;
        }
        for (Map.Entry<String, Integer> entry : map.get(key).entrySet())
{
            int count = entry.getValue();
            if (count == 0) {
                continue;
            }
            res.add(entry.getKey());
            entry.setValue(count-1);
            if (backTracking(size, entry.getKey())) {
                return true;
            } else {
                res.remove(res.size() - 1);
                entry.setValue(count);
            }
        }
        return false;
    }
}
```

BFS

1. LCP 09. 最小跳跃次数 - 力扣（LeetCode）

   题目简述：游戏机由 `N` 个特殊弹簧排成一排，编号为 `0` 到 `N-1`。初始有一个小球在编号 `0` 的弹簧处。若小球在编号为 `i` 的弹簧处，通过按动弹簧，可以选择把小球向右弹射 `jump[i]` 的距离，或者向左弹射到任意左侧弹簧的位置。也就是说，在编号为 `i` 弹簧处按动弹簧，小球可以弹向 `0` 到 `i-1` 中任意弹簧或者 `i+jump[i]` 的弹簧（若 `i+jump[i]>=N` ，则表示小球弹出了机器）。小球位于编号 0 处的弹簧时不能再向左弹。为了获得奖励，你需要将小球弹出机器。请求出最少需要按动多少次弹簧，可以将小球从编号 `0` 弹簧弹出整个机器，即向右越过编号 `N-1` 的弹簧。

   解题思路一（BFS）：采用广度优先搜索的算法思想，初始化step为0，然后逐层递增step，并把当前step能够跳到的并且之前没有遍历过的下标放进队列中，为了优化往前跳的遍历，记录下每次已经遍历过的最大的下标max。在每一层中取出队列中的下标，做如下判断：1、如果当前下标加上jump[i]能够跳出机器，则直接返回step+1　2、判断当前下标加上jump[i]的下标是否被访问过，若没有，则加入队列中　3、如果当前下标比max大，则把所有max下标之后的未被访问的下标加入队列中，然后更新max

解题代码：

```java
class Solution {
    public static int minJump(int[] jump) {
        Deque<Integer> queue = new ArrayDeque<>(); // 用于记录每一层（给定跳跃步数）所能跳到的位置
        int max = 0; // 记录下每一层（给定跳跃步数）所能跳到的最大下标
        int step = 0; // 记录跳跃次数
        boolean[] flag = new boolean[jump.length]; // 记录每个位置是否已经被遍历过
        queue.offer(0);
        flag[0] = true;
        while (!queue.isEmpty()) {
            int size = queue.size();
            for (int i = 0; i < size; i++) {
                int x = queue.poll();
                if (x + jump[x] >= jump.length) {
                    return step + 1;
                }
                if (!flag[x + jump[x]]) {
                    queue.offer(x + jump[x]);
                    flag[x + jump[x]] = true;
                }
                if (x > max) {
                    for (int j = max + 1; j < x; j++) {
                        queue.offer(j);
                        flag[j] = true;
                    }
                    max = x;
                }

            }
            step++;
        }
        return -1;
    }
}
```

二分查找

1. 4. 寻找两个正序数组的中位数 - 力扣（LeetCode）

   题目简述：给定两个大小分别为 `m` 和 `n` 的正序（从小到大）数组 `nums1` 和 `nums2`。请你找出并返回这两个正序数组的 **中位数** 。

   算法的时间复杂度应该为 `O(log (m+n))` 。

   解题思路：看官方题解——https://leetcode.cn/problems/median-of-two-sorted-arrays/solutions/258842/xun-zhao-liang-ge-you-xu-shu-zu-de-zhong-wei-s-114/?envType=study-plan-v2&envId=top-100-liked

   解题代码（二分查找）：

```java
class Solution {
    public double findMedianSortedArrays(int[] nums1, int[] nums2) {
        int length1 = nums1.length, length2 = nums2.length;
        int totalLength = length1 + length2;
        if (totalLength % 2 == 1) {
            int midIndex = totalLength / 2;
            double median = getKthElement(nums1, nums2, midIndex + 1);
            return median;
        } else {
            int midIndex1 = totalLength / 2 - 1, midIndex2 = totalLength / 2;
            double median = (getKthElement(nums1, nums2, midIndex1 + 1) + getKthElement(nums1, nums2, midIndex2 + 1)) / 2.0;
            return median;
        }
    }

    public int getKthElement(int[] nums1, int[] nums2, int k) {
        /* 主要思路：要找到第 k (k>1) 小的元素，那么就取 pivot1 = nums1[k/2-1] 和 pivot2 = nums2[k/2-1] 进行比较
         * 这里的 "/" 表示整除
         * nums1 中小于等于 pivot1 的元素有 nums1[0 .. k/2-2] 共计 k/2-1 个
         * nums2 中小于等于 pivot2 的元素有 nums2[0 .. k/2-2] 共计 k/2-1 个
         * 取 pivot = min(pivot1, pivot2)，两个数组中小于等于 pivot 的元素共计不会超过 (k/2-1) + (k/2-1) <= k-2 个
         * 这样 pivot 本身最大也只能是第 k-1 小的元素
         * 如果 pivot = pivot1，那么 nums1[0 .. k/2-1] 都不可能是第 k 小的元素。把这些元素全部 "删除"，剩下的作为新的 nums1 数组
         * 如果 pivot = pivot2，那么 nums2[0 .. k/2-1] 都不可能是第 k 小的元素。把这些元素全部 "删除"，剩下的作为新的 nums2 数组
         * 由于我们 "删除" 了一些元素（这些元素都比第 k 小的元素要小），因此需要修改 k 的值，减去删除的数的个数
         */

        int length1 = nums1.length, length2 = nums2.length;
        int index1 = 0, index2 = 0;
        int kthElement = 0;

        while (true) {
            // 边界情况
            if (index1 == length1) {
                return nums2[index2 + k - 1];
            }
            if (index2 == length2) {
                return nums1[index1 + k - 1];
            }
            if (k == 1) {
                return Math.min(nums1[index1], nums2[index2]);
            }

            // 正常情况
            int half = k / 2;
            int newIndex1 = Math.min(index1 + half, length1) - 1;
            int newIndex2 = Math.min(index2 + half, length2) - 1;
            int pivot1 = nums1[newIndex1], pivot2 = nums2[newIndex2];
```

```
            if (pivot1 <= pivot2) {
                k -= (newIndex1 - index1 + 1);
                index1 = newIndex1 + 1;
            } else {
                k -= (newIndex2 - index2 + 1);
                index2 = newIndex2 + 1;
            }
        }
    }
}
```

解题代码（划分数组）：

```
class Solution {
    public double findMedianSortedArrays(int[] nums1, int[] nums2) {
        if (nums1.length > nums2.length) {
            return findMedianSortedArrays(nums2, nums1);
        }

        int m = nums1.length;
        int n = nums2.length;
        int left = 0, right = m;
        // median1：前一部分的最大值
        // median2：后一部分的最小值
        int median1 = 0, median2 = 0;

        while (left <= right) {
            // 前一部分包含 nums1[0 .. i-1] 和 nums2[0 .. j-1]
            // 后一部分包含 nums1[i .. m-1] 和 nums2[j .. n-1]
            int i = (left + right) / 2;
            int j = (m + n + 1) / 2 - i;

            // nums_im1, nums_i, nums_jm1, nums_j 分别表示 nums1[i-1], nums1[i], nums2[j-1], nums2[j]
            int nums_im1 = (i == 0 ? Integer.MIN_VALUE : nums1[i - 1]);
            int nums_i = (i == m ? Integer.MAX_VALUE : nums1[i]);
            int nums_jm1 = (j == 0 ? Integer.MIN_VALUE : nums2[j - 1]);
            int nums_j = (j == n ? Integer.MAX_VALUE : nums2[j]);

            if (nums_im1 <= nums_j) {
                median1 = Math.max(nums_im1, nums_jm1);
                median2 = Math.min(nums_i, nums_j);
                left = i + 1;
            } else {
                right = i - 1;
            }
        }

        return (m + n) % 2 == 0 ? (median1 + median2) / 2.0 : median1;
    }
}
```

贪心

1. 45. 跳跃游戏 II - 力扣（LeetCode）

   题目简述：给定一个长度为 `n` 的 **0 索引**整数数组 `nums`，初始位置为 `nums[0]`。每个元素 `nums[i]` 表示从索引 `i` 向前跳转的最大长度。换句话说，如果你在 `nums[i]` 处，你可以跳转到任意 `nums[i + j]` 处。返回到达 `nums[n - 1]` 的最小跳跃次数。

   解题思路：每次跳跃前先计算跳跃范围，然后依次计算这个范围内的各个坐标所能到达的最远位置，选取能到达最远位置的坐标作为这次跳跃的目的地，以此类推，直至到达终点。

   解题代码：

```java
class Solution {
    public int jump(int[] nums) {
        if(nums.length==1){
            return 0;
        }
        int max=0,maxIndex=0;
        int count=1;
        int begin=0,end=nums[0];
        while (end< nums.length-1){
            max=end;
            maxIndex=begin;
            for(int i=begin+1;i<=end;i++){
                if(i+nums[i]>max){
                    max=i+nums[i];
                    maxIndex=i;
                }
            }
            begin=maxIndex;
            end=max;
            count++;
        }
        return count;
    }
}
```

动态规划

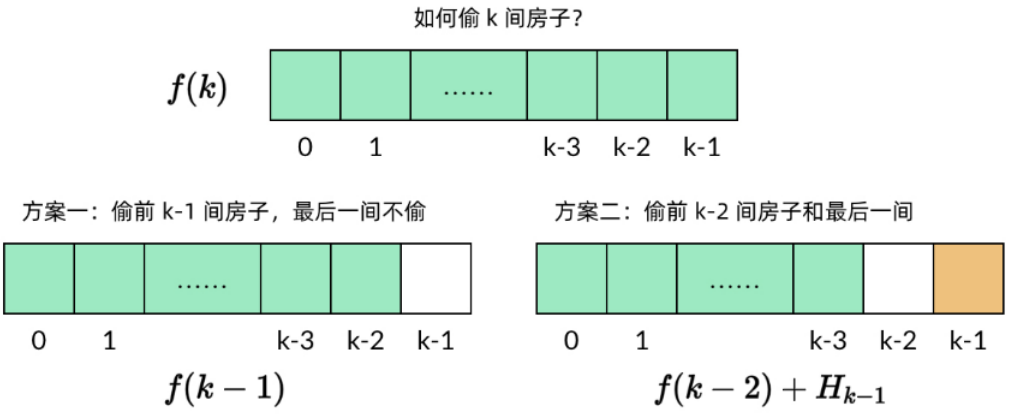
1. 198. 打家劫舍 - 力扣（LeetCode）

   题目简述：给定一个代表每个房屋存放金额的非负整数数组，在不能偷相邻房子的前提条件下，计算能够偷到的最大金额

   题解（有助于理解动态规划）：198. 打家劫舍 - 力扣（LeetCode）

解题思路：

假设一共有 n 个房子，每个房子的金额分别是 $H_0, H_1, \ldots, H_{n-1}$，子问题 $f(k)$ 表示从前 k 个房子（即 $H_0, H_1, \ldots, H_{k-1}$）中能偷到的最大金额。那么，偷 k 个房子有两种偷法：



k 个房子中最后一个房子是 H_{k-1}。如果不偷这个房子，那么问题就变成在前 $k-1$ 个房子中偷到最大的金额，也就是子问题 $f(k-1)$。如果偷这个房子，那么前一个房子 H_{k-2} 显然不能偷，其他房子不受影响。那么问题就变成在前 $k-2$ 个房子中偷到的最大的金额。两种情况中，选择金额较大的一种结果。

$$f(k) = \max\{f(k-1), H_{k-1} + f(k-2)\}$$

解题代码：

- 常规版

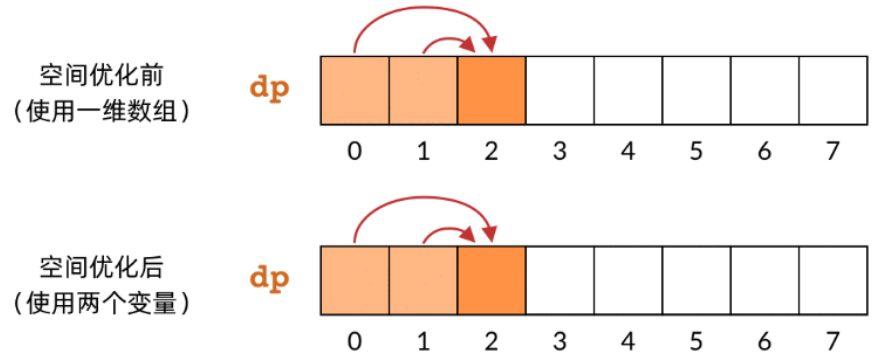
```java
public int rob(int[] nums) {
    if (nums.length == 0) {
        return 0;
    }
    // 子问题:
    // f(k) = 偷 [0..k) 房间中的最大金额

    // f(0) = 0
    // f(1) = nums[0]
    // f(k) = max{ rob(k-1), nums[k-1] + rob(k-2) }

    int N = nums.length;
    int[] dp = new int[N+1];
    dp[0] = 0;
    dp[1] = nums[0];
    for (int k = 2; k <= N; k++) {
        dp[k] = Math.max(dp[k-1], nums[k-1] + dp[k-2]);
    }
    return dp[N];
}
```

- 空间优化版

空间优化的基本原理是，很多时候我们并不需要始终持有全部的 DP 数组。对于小偷问题，我们发现，最后一步计算 $f(n)$ 的时候，实际上只用到了 $f(n-1)$ 和 $f(n-2)$ 的结果。$n-3$ 之前的子问题，实际上早就已经用不到了。那么，我们可以只用两个变量保存两个子问题的结果，就可以依次计算出所有的子问题。下面的动图比较了空间优化前和优化后的对比关系：




```java
public int rob(int[] nums) {
    int prev = 0;
    int curr = 0;

    // 每次循环，计算“偷到当前房子为止的最大金额”
    for (int i : nums) {
        // 循环开始时，curr 表示 dp[k-1]，prev 表示 dp[k-2]
        // dp[k] = max{ dp[k-1], dp[k-2] + i }
        int temp = Math.max(curr, prev + i);
        prev = curr;
        curr = temp;
        // 循环结束时，curr 表示 dp[k]，prev 表示 dp[k-1]
    }

    return curr;
}
```

2. 279. 完全平方数 - 力扣（LeetCode）

题目简述：给你一个整数 `n` ，返回 *和为 `n` 的完全平方数的最少数量* 。

解题思路：

方法一：动态规划

思路及算法

我们可以依据题目的要求写出状态表达式：$f[i]$ 表示最少需要多少个数的平方来表示整数 i。

这些数必然落在区间 $[1, \sqrt{n}]$。我们可以枚举这些数，假设当前枚举到 j，那么我们还需要取若干数的平方，构成 $i - j^2$。此时我们发现该子问题和原问题类似，只是规模变小了。这符合了动态规划的要求，于是我们可以写出状态转移方程。

$$f[i] = 1 + \min_{j=1}^{\lfloor \sqrt{i} \rfloor} f[i - j^2]$$

其中 $f[0] = 0$ 为边界条件，实际上我们无法表示数字 0，只是为了保证状态转移过程中遇到 j 恰为 $\sqrt{i}$ 的情况合法。

同时因为计算 $f[i]$ 时所需要用到的状态仅有 $f[i - j^2]$，必然小于 i，因此我们只需要从小到大地枚举 i 来计算 $f[i]$ 即可。

解题代码：

```java
class Solution {
    public int numSquares(int n) {
        int[] dp=new int[n+1];
        dp[0]=0;
        for(int i=1;i<=n;i++){
            int minnum=Integer.MAX_VALUE;
            for(int j=1;j*j<=i;j++){
                minnum=Math.min(minnum,dp[i-j*j]);
            }
            dp[i]=1+minnum;
        }
        return dp[n];
    }
}
```

3. 139. 单词拆分 - 力扣（LeetCode）

题目简述：给你一个字符串 `s` 和一个字符串列表 `wordDict` 作为字典。如果可以利用字典中出现的一个或多个单词拼接出 `s` 则返回 `true`。

解题思路：

思路和算法

我们定义 $dp[i]$ 表示字符串 s 前 i 个字符组成的字符串 $s[0..i-1]$ 是否能被空格拆分成若干个字典中出现的单词。从前往后计算考虑转移方程，每次转移的时候我们需要枚举包含位置 $i-1$ 的最后一个单词，看它是否出现在字典中以及除去这部分的字符串是否合法即可。公式化来说，我们需要枚举 $s[0..i-1]$ 中的分割点 j ，看 $s[0..j-1]$ 组成的字符串 s_1（**默认** $j=0$ **时** s_1 **为空串**）和 $s[j..i-1]$ 组成的字符串 s_2 是否都合法，如果两个字符串均合法，那么按照定义 s_1 和 s_2 拼接成的字符串也同样合法。由于计算到 $dp[i]$ 时我们已经计算出了 $dp[0..i-1]$ 的值，因此字符串 s_1 是否合法可以直接由 $dp[j]$ 得知，剩下的我们只需要看 s_2 是否合法即可，因此我们可以得出如下转移方程：

$$dp[i] = dp[j] \ \&\& \ check(s[j..i-1])$$

其中 $check(s[j..i-1])$ 表示子串 $s[j..i-1]$ 是否出现在字典中。

对于检查一个字符串是否出现在给定的字符串列表里一般可以考虑哈希表来快速判断，同时也可以做一些简单的剪枝，枚举分割点的时候倒着枚举，如果分割点 j 到 i 的长度已经大于字典列表里最长的单词的长度，那么就结束枚举，但是需要注意的是下面的代码给出的是不带剪枝的写法。

对于边界条件，我们定义 $dp[0] = true$ 表示空串且合法。

解题代码：

```java
class Solution {
    public boolean wordBreak(String s, List<String> wordDict) {
        Set<String> set = new HashSet<>(wordDict);
        boolean[] dp=new boolean[s.length()+1];
        int maxLen=0;
        // 找到字典列表最长单词的长度
        for(String str:set){
            maxLen=Math.max(maxLen,str.length());
        }
        dp[0]=true;
```

```
        for(int i=1;i<=s.length();i++){
            for(int j=0;j<i;j++){
                // 剪枝操作
                if(i-j>maxLen){
                    continue;
                }
                if(dp[j]&&set.contains(s.substring(j,i))){
                    dp[i]=true;
                    break;
                }
            }
        }
        return dp[s.length()];
    }
}
```

4. 300. 最长递增子序列 - 力扣（LeetCode）

题目简述：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。**子序列** 是由数组派生而来的序列，删除（或不删除）数组中的元素而不改变其余元素的顺序。例如，`[3,6,2,7]` 是数组 `[0,3,1,6,2,2,7]` 的子序列。

解题思路（解法一）：

思路与算法

定义 $dp[i]$ 为考虑前 i 个元素，以第 i 个数字结尾的最长上升子序列的长度，**注意** $nums[i]$ **必须被选取。**

我们从小到大计算 dp 数组的值，在计算 $dp[i]$ 之前，我们已经计算出 $dp[0 \ldots i-1]$ 的值，则状态转移方程为：

$$dp[i] = \max(dp[j]) + 1, \text{其中} \, 0 \le j < i \, \text{且} \, num[j] < num[i]$$

即考虑往 $dp[0 \ldots i-1]$ 中最长的上升子序列后面再加一个 $nums[i]$。由于 $dp[j]$ 代表 $nums[0 \ldots j]$ 中以 $nums[j]$ 结尾的最长上升子序列，所以如果能从 $dp[j]$ 这个状态转移过来，那么 $nums[i]$ 必然要大于 $nums[j]$，才能将 $nums[i]$ 放在 $nums[j]$ 后面以形成更长的上升子序列。

最后，整个数组的最长上升子序列即所有 $dp[i]$ 中的最大值。

$$\text{LIS}_{length} = \max(dp[i]), \text{其中} \, 0 \le i < n$$

解题代码（解法一：动态规划）：

```
class Solution {
    public int lengthOfLIS(int[] nums) {
        if (nums.length == 0) {
            return 0;
        }
        int[] dp = new int[nums.length];
        dp[0] = 1;
        int maxans = 1;
        for (int i = 1; i < nums.length; i++) {
            dp[i] = 1;
            for (int j = 0; j < i; j++) {
                if (nums[i] > nums[j]) {
                    dp[i] = Math.max(dp[i], dp[j] + 1);
                }
```

```
            }
            maxans = Math.max(maxans, dp[i]);
        }
        return maxans;
    }
}
```

解题思路（解法二：贪心 + 二分查找）：

考虑一个简单的贪心，如果我们要使上升子序列尽可能的长，则我们需要让序列上升得尽可能慢，因此我们希望每次在上升子序列最后加上的那个数尽可能的小。

基于上面的贪心思路，我们维护一个数组 $d[i]$ ，表示长度为 i 的最长上升子序列的末尾元素的最小值，用 len 记录目前最长上升子序列的长度，起始时 len 为 1，$d[1] = nums[0]$。

同时我们可以注意到 $d[i]$ 是关于 i 单调递增的。因为如果 $d[j] \geq d[i]$ 且 $j < i$，我们考虑从长度为 i 的最长上升子序列的末尾删除 $i - j$ 个元素，那么这个序列长度变为 j ，且第 j 个元素 x（末尾元素）必然小于 $d[i]$，也就小于 $d[j]$。那么我们就找到了一个长度为 j 的最长上升子序列，并且末尾元素比 $d[j]$ 小，从而产生了矛盾。因此数组 d 的单调性得证。

我们依次遍历数组 $nums$ 中的每个元素，并更新数组 d 和 len 的值。如果 $nums[i] > d[len]$ 则更新 $len = len + 1$，否则在 $d[1 \ldots len]$ 中找满足 $d[i - 1] < nums[j] < d[i]$ 的下标 i，并更新 $d[i] = nums[j]$。

根据 d 数组的单调性，我们可以使用二分查找寻找下标 i，优化时间复杂度。

最后整个算法流程为：

- 设当前已求出的最长上升子序列的长度为 len（初始时为 1），从前往后遍历数组 $nums$，在遍历到 $nums[i]$ 时：
    - 如果 $nums[i] > d[len]$ ，则直接加入到 d 数组末尾，并更新 $len = len + 1$；
    - 否则，在 d 数组中二分查找，找到第一个比 $nums[i]$ 小的数 $d[k]$ ，并更新 $d[k + 1] = nums[i]$。

解题代码（解法二）：

```
class Solution {
    public int lengthOfLIS(int[] nums) {
        if (nums.length == 0) {
            return 0;
        }
        int len=1,n= nums.length;
        int[] dp=new int[n+1];
        dp[len]=nums[0];
        for(int i=1;i< nums.length;i++){
            if(nums[i]>dp[len]){
                dp[++len]=nums[i];
            }else{
                int left=1,right=len-1;
                while (left<=right){
                    int middle=(left+right)/2;
                    if(dp[middle]==nums[i]){
                        left=middle;
                        break;
                    } else if (dp[middle]>nums[i]) {
                        right=middle-1;
```

```
                    }else{
                        left=middle+1;
                    }
                }
                dp[left]=nums[i];
            }
        }
        return len;
    }

}
```

5. 152. 乘积最大子数组 - 力扣（LeetCode）

题目简述：给你一个整数数组 `nums` ，请你找出数组中乘积最大的非空连续子数组（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。

解题思路（动态规划法）：

我们可以根据正负性进行分类讨论。

考虑当前位置如果是一个负数的话，那么我们希望以它前一个位置结尾的某个段的积也是个负数，这样就可以负负得正，并且我们希望这个积尽可能「负得更多」，即尽可能小。如果当前位置是一个正数的话，我们更希望以它前一个位置结尾的某个段的积也是个正数，并且希望它尽可能地大。

于是这里我们可以再维护一个 $f_{\min}(i)$，它表示以第 i 个元素结尾的乘积最小子数组的乘积，那么我们可以得到这样的动态规划转移方程：

$$f_{\max}(i) = \max_{i=1}^{n}\{f_{\max}(i-1) \times a_i, f_{\min}(i-1) \times a_i, a_i\}$$
$$f_{\min}(i) = \min_{i=1}^{n}\{f_{\max}(i-1) \times a_i, f_{\min}(i-1) \times a_i, a_i\}$$

它代表第 i 个元素结尾的乘积最大子数组的乘积 $f_{\max}(i)$，可以考虑把 a_i 加入第 $i-1$ 个元素结尾的乘积最大或最小的子数组的乘积中，二者加上 a_i，三者取大，就是第 i 个元素结尾的乘积最大子数组的乘积。第 i 个元素结尾的乘积最小子数组的乘积 $f_{\min}(i)$ 同理。

解题代码：

```
class Solution {
    public int maxProduct(int[] nums) {
        if(nums.length==0){
            return 0;
        }
        int[] min=new int[nums.length];
        int[] max=new int[nums.length];
        for(int i=0;i< nums.length;i++){
            min[i]=nums[i];
            max[i]=nums[i];
        }
        int maxn=max[0];
        for(int i=1;i< nums.length;i++){
            max[i]=Math.max(Math.max(max[i-1]*nums[i],min[i-1]*nums[i]),nums[i]);
            min[i]=Math.min(Math.min(max[i-1]*nums[i],min[i-1]*nums[i]),nums[i]);
            maxn=Math.max(maxn,max[i]);
        }
        return maxn;
```

```
    }
}
```

6. 416. 分割等和子集 - 力扣（LeetCode）

题目简述：给你一个 **只包含正整数** 的 **非空** 数组 `nums` 。请你判断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。

解题思路：

方法一：动态规划

思路与算法

这道题可以换一种表述：给定一个只包含正整数的非空数组 $nums[0]$，判断是否可以从数组中选出一些数字，使得这些数字的和等于整个数组的元素和的一半。因此这个问题可以转换成「$0-1$ 背包问题」。这道题与传统的「$0-1$ 背包问题」的区别在于，传统的「$0-1$ 背包问题」要求选取的物品的重量之和**不能超过**背包的总容量，这道题则要求选取的数字的和**恰好等于**整个数组的元素和的一半。类似于传统的「$0-1$ 背包问题」，可以使用动态规划求解。

在使用动态规划求解之前，首先需要进行以下判断。

- 根据数组的长度 n 判断数组是否可以被划分。如果 $n<2$，则不可能将数组分割成元素和相等的两个子集，因此直接返回 false。
- 计算整个数组的元素和 sum 以及最大元素 $maxNum$。如果 sum 是奇数，则不可能将数组分割成元素和相等的两个子集，因此直接返回 false。如果 sum 是偶数，则令 $target=\frac{sum}{2}$，需要判断是否可以从数组中选出一些数字，使得这些数字的和等于 $target$。如果 $maxNum>target$，则除了 $maxNum$ 以外的所有元素之和一定小于 $target$，因此不可能将数组分割成元素和相等的两个子集，直接返回 false。

创建二维数组 dp，包含 n 行 $target+1$ 列，其中 $dp[i][j]$ 表示从数组的 $[0,i]$ 下标范围内选取若干个正整数（可以是 0 个），是否存在一种选取方案使得被选取的正整数的和等于 j。初始时，dp 中的全部元素都是 false。

在定义状态之后，需要考虑边界情况。以下两种情况都属于边界情况。

- 如果不选取任何正整数，则被选取的正整数等于 0。因此对于所有 $0\le i<n$，都有 $dp[i][0]=\text{true}$。
- 当 $i==0$ 时，只有一个正整数 $nums[0]$ 可以被选取，因此 $dp[0][nums[0]]=\text{true}$。

对于 $i>0$ 且 $j>0$ 的情况，如何确定 $dp[i][j]$ 的值？需要分别考虑以下两种情况。

- 如果 $j\ge nums[i]$，则对于当前的数字 $nums[i]$，可以选取也可以不选取，两种情况只要有一个为 true，就有 $dp[i][j]=\text{true}$。
  - 如果不选取 $nums[i]$，则 $dp[i][j]=dp[i-1][j]$；
  - 如果选取 $nums[i]$，则 $dp[i][j]=dp[i-1][j-nums[i]]$。
- 如果 $j<nums[i]$，则在选取的数字的和等于 j 的情况下无法选取当前的数字 $nums[i]$，因此有 $dp[i][j]=dp[i-1][j]$。

状态转移方程如下：

$$dp[i][j]=\begin{cases}dp[i-1][j]\mid dp[i-1][j-nums[i]], & j\ge nums[i]\\ dp[i-1][j], & j<nums[i]\end{cases}$$

最终得到 $dp[n-1][target]$ 即为答案。

解题代码（未优化空间复杂度版）：

```java
class Solution {
    public boolean canPartition(int[] nums) {
        int n = nums.length;
        if (n < 2) {
            return false;
        }
        int sum = 0, maxNum = 0;
        for (int num : nums) {
            sum += num;
            maxNum = Math.max(maxNum, num);
        }
        if (sum % 2 != 0) {
            return false;
        }
        int target = sum / 2;
        if (maxNum > target) {
            return false;
        }
        boolean[][] dp = new boolean[n][target + 1];
        for (int i = 0; i < n; i++) {
            dp[i][0] = true;
        }
        dp[0][nums[0]] = true;
        for (int i = 1; i < n; i++) {
            int num = nums[i];
            for (int j = 1; j <= target; j++) {
                if (j >= num) {
                    dp[i][j] = dp[i - 1][j] | dp[i - 1][j - num];
                } else {
                    dp[i][j] = dp[i - 1][j];
                }
            }
        }
        return dp[n - 1][target];
    }
}
```

解题代码（优化版）：

上述代码的空间复杂度是 $O(n \times target)$。但是可以发现在计算 dp 的过程中，每一行的 dp 值都只与上一行的 dp 值有关，因此只需要一个一维数组即可将空间复杂度降到 $O(target)$。此时的转移方程为：

$$dp[j] = dp[j] \mid dp[j - nums[i]]$$

且需要注意的是第二层的循环我们需要**从大到小计算**，因为如果我们从小到大更新 dp 值，那么在计算 $dp[j]$ 值的时候，$dp[j - nums[i]]$ 已经是被更新过的状态，不再是上一行的 dp 值。

```java
class Solution {
    public boolean canPartition(int[] nums) {
        int n = nums.length;
        if (n < 2) {
            return false;
        }
        int sum = 0, maxNum = 0;
```

```
        for (int num : nums) {
            sum += num;
            maxNum = Math.max(maxNum, num);
        }
        if (sum % 2 != 0) {
            return false;
        }
        int target = sum / 2;
        if (maxNum > target) {
            return false;
        }
        boolean[] dp = new boolean[target + 1];
        dp[0] = true;
        for (int i = 0; i < n; i++) {
            int num = nums[i];
            for (int j = target; j >= num; --j) {
                dp[j] |= dp[j - num];
            }
        }
        return dp[target];
    }
}
```

7. 32. 最长有效括号 - 力扣（LeetCode）

题目简述：给你一个只包含 `'('` 和 `')'` 的字符串，找出最长有效（格式正确且连续）括号子串的长度。

解题思路（动态规划）：

我们定义 $dp[i]$ 表示以下标 i 字符结尾的最长有效括号的长度。我们将 dp 数组全部初始化为 0 。显然有效的子串一定以 ‘)’ 结尾，因此我们可以知道以 ‘(’ 结尾的子串对应的 dp 值必定为 0 ，我们只需要求解 ‘)’ 在 dp 数组中对应位置的值。

我们从前往后遍历字符串求解 dp 值，我们每两个字符检查一次：

1. $s[i] = $ ‘)’ 且 $s[i-1] = $ ‘(’，也就是字符串形如 “……()”，我们可以推出：

$$dp[i] = dp[i-2] + 2$$

   我们可以进行这样的转移，是因为结束部分的 "()" 是一个有效子字符串，并且将之前有效子字符串的长度增加了 2 。

2. $s[i] = $ ‘)’ 且 $s[i-1] = $ ‘)’，也就是字符串形如 “……))”，我们可以推出：
   如果 $s[i - dp[i-1] - 1] = $ ‘(’，那么

$$dp[i] = dp[i-1] + dp[i - dp[i-1] - 2] + 2$$

我们考虑如果倒数第二个 ‘)’ 是一个有效子字符串的一部分（记作 sub_s），对于最后一个 ‘)’，如果它是一个更长子字符串的一部分，那么它一定有一个对应的 ‘(’，且它的位置在倒数第二个 ‘)’ 所在的有效子字符串的前面（也就是 sub_s 的前面）。因此，如果子字符串 sub_s 的前面恰好是 ‘(’，那么我们就用 2 加上 sub_s 的长度（$dp[i-1]$）去更新 $dp[i]$。同时，我们也会把有效子串 “(sub_s)” 之前的有效子串的长度也加上，也就是再加上 $dp[i - dp[i-1] - 2]$。

最后的答案即为 dp 数组中的最大值。

解题代码：

```java
class Solution {
    public int longestValidParentheses(String s) {
        int maxans = 0;
        int[] dp = new int[s.length()];
        for (int i = 1; i < s.length(); i++) {
            if (s.charAt(i) == ')') {
                if (s.charAt(i - 1) == '(') {
                    dp[i] = (i >= 2 ? dp[i - 2] : 0) + 2;
                } else if (i - dp[i - 1] > 0 && s.charAt(i - dp[i - 1] - 1) == '(') {
                    dp[i] = dp[i - 1] + ((i - dp[i - 1]) >= 2 ? dp[i - dp[i - 1] - 2] : 0) + 2;
                }
                maxans = Math.max(maxans, dp[i]);
            }
        }
        return maxans;
    }
}
```

解题思路（栈）：

思路和算法

撇开方法一提及的动态规划方法，相信大多数人对于这题的第一直觉是找到每个可能的子串后判断它的有效性，但这样的时间复杂度会达到 $O(n^3)$，无法通过所有测试用例。但是通过栈，我们可以在遍历给定字符串的过程中去判断到目前为止扫描的子串的有效性，同时能得到最长有效括号的长度。

具体做法是我们始终保持栈底元素为当前已经遍历过的元素中「最后一个没有被匹配的右括号的下标」，这样的做法主要是考虑了边界条件的处理，栈里其他元素维护左括号的下标：

- 对于遇到的每个 ‘(’ ，我们将它的下标放入栈中
- 对于遇到的每个 ‘)’ ，我们先弹出栈顶元素表示匹配了当前右括号：
  - 如果栈为空，说明当前的右括号为没有被匹配的右括号，我们将其下标放入栈中来更新我们之前提到的「最后一个没有被匹配的右括号的下标」
  - 如果栈不为空，当前右括号的下标减去栈顶元素即为「以该右括号为结尾的最长有效括号的长度」

我们从前往后遍历字符串并更新答案即可。

需要注意的是，如果一开始栈为空，第一个字符为左括号的时候我们会将其放入栈中，这样就不满足提及的「最后一个没有被匹配的右括号的下标」，为了保持统一，我们在一开始的时候往栈中放入一个值为 -1 的元素。

解题代码：

```java
class Solution {
    public int longestValidParentheses(String s) {
        int maxans = 0;
        Deque<Integer> stack = new LinkedList<Integer>();
        stack.push(-1);
        for (int i = 0; i < s.length(); i++) {
            if (s.charAt(i) == '(') {
                stack.push(i);
            } else {
```

```
                stack.pop();
                if (stack.isEmpty()) {
                    stack.push(i);
                } else {
                    maxans = Math.max(maxans, i - stack.peek());
                }
            }
        }
        return maxans;
    }
}
```

解题思路（贪心，左右指针法）：

[32. 最长有效括号 - 力扣（LeetCode）]

思路和算法

在此方法中，我们利用两个计数器 $left$ 和 $right$ 。首先，我们从左到右遍历字符串，对于遇到的每个 '(' ，我们增加 $left$ 计数器，对于遇到的每个 ')' ，我们增加 $right$ 计数器。每当 $left$ 计数器与 $right$ 计数器相等时，我们计算当前有效字符串的长度，并且记录目前为止找到的最长子字符串。当 $right$ 计数器比 $left$ 计数器大时，我们将 $left$ 和 $right$ 计数器同时变回 0。

这样的做法贪心地考虑了以当前字符下标结尾的有效括号长度，每次当右括号数量多于左括号数量的时候之前的字符我们都扔掉不再考虑，重新从下一个字符开始计算，但这样会漏掉一种情况，就是遍历的时候左括号的数量始终大于右括号的数量，即 `(()` ，这种时候最长有效括号是求不出来的。

解决的方法也很简单，我们只需要从右往左遍历用类似的方法计算即可，只是这个时候判断条件反了过来：

- 当 $left$ 计数器比 $right$ 计数器大时，我们将 $left$ 和 $right$ 计数器同时变回 0
- 当 $left$ 计数器与 $right$ 计数器相等时，我们计算当前有效字符串的长度，并且记录目前为止找到的最长子字符串

这样我们就能涵盖所有情况从而求解出答案。

解题代码：

```
class Solution {
    public int longestValidParentheses(String s) {
        int left = 0, right = 0, maxlength = 0;
        for (int i = 0; i < s.length(); i++) {
            if (s.charAt(i) == '(') {
                left++;
            } else {
                right++;
            }
            if (left == right) {
                maxlength = Math.max(maxlength, 2 * right);
            } else if (right > left) {
                left = right = 0;
            }
        }
        left = right = 0;
        for (int i = s.length() - 1; i >= 0; i--) {
            if (s.charAt(i) == '(') {
```

```
                left++;
            } else {
                right++;
            }
            if (left == right) {
                maxlength = Math.max(maxlength, 2 * left);
            } else if (left > right) {
                left = right = 0;
            }
        }
        return maxlength;
    }
}
```

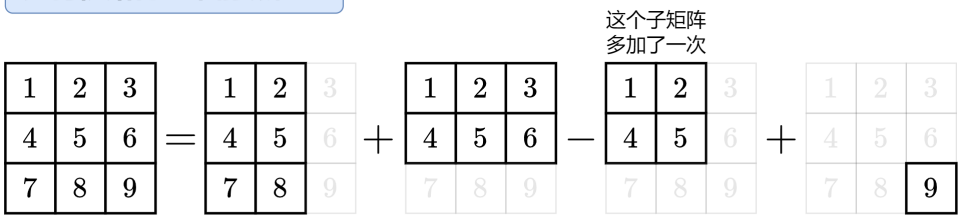
8. 3148. 矩阵中的最大得分 - 力扣（LeetCode）

题目简述：给你一个由 **正整数** 组成、大小为 `m x n` 的矩阵 `grid`。你可以从矩阵中的任一单元格移动到另一个位于正下方或正右侧的任意单元格（不必相邻）。从值为 `c1` 的单元格移动到值为 `c2` 的单元格的得分为 `c2 - c1`。你可以从 **任一** 单元格开始，并且必须至少移动一次。返回你能得到的 **最大** 总得分。

解题思路（二维前缀最小值）：

先了解解决二位前缀和的思路：3148. 矩阵中的最大得分 - 力扣（LeetCode）

如何初始化二维前缀和?

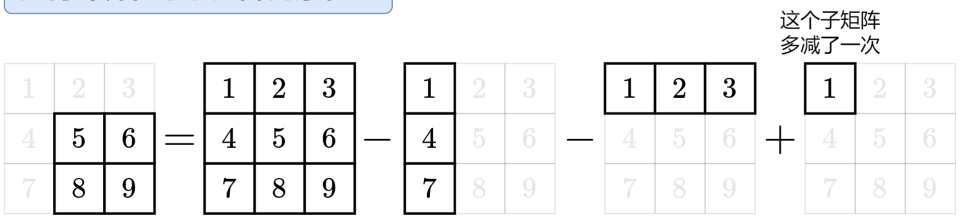


定义 $sum[i+1][j+1]$ 表示左上角为 $a[0][0]$，右下角为 $a[i][j]$ 的子矩阵元素和。

采用这种定义方式，无需单独处理第一行/第一列的元素和。

$sum[i+1][j+1] = sum[i+1][j] + sum[i][j+1] - sum[i][j] + a[i][j]$。

如何计算任意子矩阵元素和?



设子矩阵左上角为 $a[r_1][c_1]$，右下角为 $a[r_2-1][c_2-1]$。

子矩阵元素和 $= sum[r_2][c_2] - sum[r_2][c_1] - sum[r_1][c_2] + sum[r_1][c_1]$。

再以此类比来求二维前缀最小值：

把 $grid[i][j]$ 视作**海拔高度**，把得分视作**重力势能的变化量**。从高度 c_1 移动到高度 c_2，重力势能增加了 $c_2 - c_1$。注意 $c_2 - c_1$ 可能是负数。

题目相当于计算重力势能的变化量之和，也就是**终点与起点的海拔高度之差**。

枚举终点位置 (i, j)，那么起点的海拔高度越小越好。由于我们只能向右和向下走，所以起点只能在 (i, j) 的左上方向（可以是 (i, j) 的正左方向或正上方向）。

按照【图解】一张图秒懂二维前缀和 的思路，定义 $f[i+1][j+1]$ 表示左上角在 $(0, 0)$，右下角在 (i, j) 的子矩阵的**最小值**。

类似二维前缀和，$f[i+1][j+1]$ 可以递推计算：

$$f[i+1][j+1] = \min(f[i+1][j], f[i][j+1], grid[i][j])$$

注意题目要求至少移动一次，也就是起点与终点不能重合。如果终点在 (i, j)，那么起点的海拔高度最小值为

$$\min(f[i+1][j], f[i][j+1])$$

终点与起点的海拔高度之差为

$$grid[i][j] - \min(f[i+1][j], f[i][j+1])$$

上式的最大值即为答案。

解题代码：

```java
class Solution {
    public int maxScore(List<List<Integer>> grid) {
        int y = grid.get(0).size();
        int x = grid.size();
        int MAX = Integer.MIN_VALUE;
        int[][] f = new int[x + 1][y + 1];
        Arrays.fill(f[0], Integer.MAX_VALUE);
        for (int i = 0; i < x; i++) {
            f[i + 1][0] = Integer.MAX_VALUE;
            for (int j = 0; j < y; j++) {
                int temp = Math.min(f[i][j + 1], f[i + 1][j]);
                MAX = Math.max(MAX, grid.get(i).get(j) - temp);
                f[i+1][j+1] = Math.min(temp,grid.get(i).get(j));
            }
        }
        return MAX;
    }
}
```

优化：维护每列的最小值 *colMin*，这样空间复杂度更小

```java
class Solution {
    public int maxScore(List<List<Integer>> grid) {
        int ans = Integer.MIN_VALUE;
        int n = grid.get(0).size();
        int[] colMin = new int[n];
        Arrays.fill(colMin, Integer.MAX_VALUE);
        for (List<Integer> row : grid) {
            int preMin = Integer.MAX_VALUE; // colMin[0..j] 的最小值
            for (int j = 0; j < n; j++) {
                int x = row.get(j);
```

```
                ans = Math.max(ans, x - Math.min(preMin, colMin[j]));
                colMin[j] = Math.min(colMin[j], x);
                preMin = Math.min(preMin, colMin[j]);
            }
        }
        return ans;
    }
}

作者：灵茶山艾府
链接：https://leetcode.cn/problems/maximum-difference-score-in-a-grid/solutions/2774823/nao-jin-ji-zhuan-wan-dppythonjavacgo-by-swux7/
来源：力扣（LeetCode）
著作权归作者所有。商业转载请联系作者获得授权，非商业转载请注明出处。
```

9. 376. 摆动序列 - 力扣（LeetCode）

题目简述：如果连续数字之间的差严格地在正数和负数之间交替，则数字序列称为 **摆动序列** 。第一个差（如果存在的话）可能是正数或负数。仅有一个元素或者含两个不等元素的序列也视作摆动序列。

- 例如，`[1, 7, 4, 9, 2, 5]` 是一个 **摆动序列** ，因为差值 `(6, -3, 5, -7, 3)` 是正负交替出现的。
- 相反，`[1, 4, 7, 2, 5]` 和 `[1, 7, 4, 5, 5]` 不是摆动序列，第一个序列是因为它的前两个差值都是正数，第二个序列是因为它的最后一个差值为零。

子序列 可以通过从原始序列中删除一些（也可以不删除）元素来获得，剩下的元素保持其原始顺序。

给你一个整数数组 `nums` ，返回 `nums` 中作为 **摆动序列** 的 **最长子序列的长度** 。

解题思路一（动态规划，未优化）：376. 摆动序列 - 力扣（LeetCode）

每当我们选择一个元素作为摆动序列的一部分时，这个元素要么是上升的，要么是下降的，这取决于前一个元素的大小。那么列出状态表达式为：

1. $up[i]$ 表示以前 i 个元素中的某一个为结尾的最长的「上升摆动序列」的长度。
2. $down[i]$ 表示以前 i 个元素中的某一个为结尾的最长的「下降摆动序列」的长度。

这样我们可以用同样的方法说明 $down[i]$ 的状态转移规则，最终的状态转移方程为：

$$up[i] = \begin{cases} up[i-1], & nums[i] \le nums[i-1] \\ \max(up[i-1], down[i-1]+1), & nums[i] > nums[i-1] \end{cases}$$

$$down[i] = \begin{cases} down[i-1], & nums[i] \ge nums[i-1] \\ \max(up[i-1]+1, down[i-1]), & nums[i] < nums[i-1] \end{cases}$$

解题代码：

```
class Solution {
    public int wiggleMaxLength(int[] nums) {
        if (nums.length < 2) {
            return nums.length;
        }
        int[] up = new int[nums.length];
        int[] down = new int[nums.length];
```

```
        up[0]=down[0]=1;

        for(int i=1;i< nums.length;i++){
            if(nums[i]>nums[i-1]){
                up[i]=Math.max(up[i-1],down[i-1]+1);
                down[i]=down[i-1];
            } else if (nums[i]<nums[i-1]) {
                down[i]=Math.max(down[i-1],up[i-1]+1);
                up[i]=up[i-1];
            } else {
                up[i]=up[i-1];
                down[i]=down[i-1];
            }
        }

        return Math.max(up[nums.length-1],down[nums.length-1]);
    }
}
```

解题思路二（优化版动态规划）：

针对思路一，由于最后结果只涉及up[n-1]和down[n-1]，所以只需要维护两个变量up和down来根据前一个状态来进行转移即可。

注意到每有一个「峰」到「谷」的下降趋势，$down$ 值才会增加，每有一个「谷」到「峰」的上升趋势，up 值才会增加。且过程中 $down$ 与 up 的差的绝对值值恒不大于 1，即 $up \leq down + 1$ 且 $down \leq up + 1$，于是有 $\max(up, down + 1) = down + 1$ 且 $\max(up + 1, down) = up + 1$。这样我们可以省去不必要的比较大小的过程。

解题代码：

```
class Solution {
    public int wiggleMaxLength(int[] nums) {
        if (nums.length < 2) {
            return nums.length;
        }
        int up = 1, down = 1;

        for (int i = 1; i < nums.length; i++) {
            if (nums[i] > nums[i - 1]) {
                up = down + 1;
            } else if (nums[i] < nums[i - 1]) {
                down = up + 1;
            }
        }

        return Math.max(up, down);
    }
}
```

解题思路三（计算全局递增/递减改变次数，也可以理解为贪心算法）： 只要统计递增/递减的变化次数即可（即为答案）

力扣官方贪心思路如下（代码有所不同，个人觉得自己贴在下面的代码更简洁优雅）：

思路及解法

观察这个序列可以发现，我们不断地交错选择「峰」与「谷」，可以使得该序列尽可能长。证明非常简单：如果我们选择了一个「过渡元素」，那么在原序列中，这个「过渡元素」的两侧有一个「峰」和一个「谷」。不失一般性，我们假设在原序列中的出现顺序为「峰」「过渡元素」「谷」。如果「过渡元素」在选择的序列中小于其两侧的元素，那么「谷」一定没有在选择的序列中出现，我们可以将「过渡元素」替换成「谷」；同理，如果「过渡元素」在选择的序列中大于其两侧的元素，那么「峰」一定没有在选择的序列中出现，我们可以将「过渡元素」替换成「峰」。这样一来，我们总可以将任意满足要求的序列中的所有「过渡元素」替换成「峰」或「谷」。并且由于我们不断地交错选择「峰」与「谷」的方法就可以满足要求，因此这种选择方法就一定可以达到可选元素数量的最大值。

这样，我们只需要统计该序列中「峰」与「谷」的数量即可（注意序列两端的数也是「峰」或「谷」），但需要注意处理相邻的相同元素。

在实际代码中，我们记录当前序列的上升下降趋势。每次加入一个新元素时，用新的上升下降趋势与之前对比，如果出现了「峰」或「谷」，答案加一，并更新当前序列的上升下降趋势。

解题代码：

```java
public class Solutions {

    // 只要统计递增/递减的变化次数即可（即为答案）
    public int wiggleMaxLength(int[] nums) {
        if (nums.length < 2) {
            return nums.length;
        }
        int up = 0; // 判断之前序列的末尾是否递增（1：递增，0：未赋值，-1：递减）
        int res = 1;

        for (int i = 1; i < nums.length; i++) {
            if (nums[i] > nums[i - 1]) { // 当前是递增
                // 若先前是递减则增加改变次数并更新状态（递增or递减）
                if (up <= 0) {
                    up = 1;
                    res++;
                }
            } else if (nums[i] < nums[i - 1]) { // 当前是递减
                // 若先前是递增则增加改变次数并更新状态（递增or递减）
                if (up >= 0) {
                    up = -1;
                    res++;
                }
            }
        }

        return res;
    }
}
```

多维动态规划

1. 5. 最长回文子串 - 力扣（LeetCode）

   题目简述：给你一个字符串 `s`，找到 `s` 中最长的回文子串。

   解题思路（动态规划）：

   根据这样的思路，我们就可以用动态规划的方法解决本题。我们用 $P(i,j)$ 表示字符串 s 的第 i 到 j 个字母组成的串（下文表示成 $s[i:j]$）是否为回文串：

   $$P(i,j)=\begin{cases}\text{true}, & \text{如果子串 } S_i \dots S_j \text{ 是回文串} \\ \text{false}, & \text{其它情况}\end{cases}$$

   这里的「其它情况」包含两种可能性：

   - $s[i,j]$ 本身不是一个回文串；
   - $i>j$，此时 $s[i,j]$ 本身不合法。

   那么我们就可以写出动态规划的状态转移方程：

   $$P(i,j)=P(i+1,j-1)\wedge(S_i==S_j)$$

   也就是说，只有 $s[i+1:j-1]$ 是回文串，并且 s 的第 i 和 j 个字母相同时，$s[i:j]$ 才会是回文串。

   上文的所有讨论是建立在子串长度大于 2 的前提之上的，我们还需要考虑动态规划中的边界条件，即子串的长度为 1 或 2。对于长度为 1 的子串，它显然是个回文串；对于长度为 2 的子串，只要它的两个字母相同，它就是一个回文串。因此我们就可以写出动态规划的边界条件：

   $$\begin{cases}P(i,i)=\text{true} \\ P(i,i+1)=(S_i==S_{i+1})\end{cases}$$

   根据这个思路，我们就可以完成动态规划了，最终的答案即为所有 $P(i,j)=\text{true}$ 中 $j-i+1$（即子串长度）的最大值。**注意：在状态转移方程中，我们是从长度较短的字符串向长度较长的字符串进行转移的，因此一定要注意动态规划的循环顺序。**

   解题代码：

```java
public class Solution {

    public String longestPalindrome(String s) {
        int len = s.length();
        if (len < 2) {
            return s;
        }

        int maxLen = 1;
        int begin = 0;
        // dp[i][j] 表示 s[i..j] 是否是回文串
        boolean[][] dp = new boolean[len][len];
        // 初始化：所有长度为 1 的子串都是回文串
        for (int i = 0; i < len; i++) {
            dp[i][i] = true;
        }

        char[] charArray = s.toCharArray();
        // 递推开始
```

```
        // 先枚举子串长度
        for (int L = 2; L <= len; L++) {
            // 枚举左边界，左边界的上限设置可以宽松一些
            for (int i = 0; i < len; i++) {
                // 由 L 和 i 可以确定右边界，即 j - i + 1 = L 得
                int j = L + i - 1;
                // 如果右边界越界，就可以退出当前循环
                if (j >= len) {
                    break;
                }

                if (charArray[i] != charArray[j]) {
                    dp[i][j] = false;
                } else {
                    if (j - i < 3) {
                        dp[i][j] = true;
                    } else {
                        dp[i][j] = dp[i + 1][j - 1];
                    }
                }

                // 只要 dp[i][L] == true 成立，就表示子串 s[i..L] 是回文，此时记录回文长度和起始位置
                if (dp[i][j] && j - i + 1 > maxLen) {
                    maxLen = j - i + 1;
                    begin = i;
                }
            }
        }
        return s.substring(begin, begin + maxLen);
    }
}
```

解题思路（中心扩展算法）：

思路与算法

我们仔细观察一下方法一中的状态转移方程：

$$\begin{cases} P(i,i) & = & \text{true} \\ P(i,i+1) & = & (S_i == S_{i+1}) \\ P(i,j) & = & P(i+1,j-1) \wedge (S_i == S_j) \end{cases}$$

找出其中的状态转移链：

$$P(i,j) \leftarrow P(i+1,j-1) \leftarrow P(i+2,j-2) \leftarrow \cdots \leftarrow \text{某一边界情况}$$

可以发现，**所有的状态在转移的时候的可能性都是唯一的。**也就是说，我们可以从每一种边界情况开始「扩展」，也可以得出所有的状态对应的答案。

边界情况即为子串长度为 1 或 2 的情况。我们枚举每一种边界情况，并从对应的子串开始不断地向两边扩展。如果两边的字母相同，我们就可以继续扩展，例如从 $P(i+1,j-1)$ 扩展到 $P(i,j)$；如果两边的字母不同，我们就可以停止扩展，因为在这之后的子串都不能是回文串了。

聪明的读者此时应该可以发现，「边界情况」对应的子串实际上就是我们「扩展」出的回文串的「回文中心」。方法二的本质即为：我们枚举所有的「回文中心」并尝试「扩展」，直到无法扩展为止，此时的回文串长度即为此「回文中心」下的最长回文串长度。我们对所有的长度求出最大值，即可得到最终的答案。

解题代码：

```java
class Solution {
    public String longestPalindrome(String s) {
        if(s.length()<=1){
            return s;
        }
        int begin=0,maxLen=1;
        for(int i=0;i<s.length();i++){
            int[] len1=expand(i,i,s);
            int[] len2=expand(i,i+1,s);
            if(len1[1]>=len2[1]&&len1[1]>maxLen){
                maxLen=len1[1];
                begin=len1[0];
            } else if (len2[1]>len1[1]&&len2[1]>maxLen) {
                maxLen=len2[1];
                begin=len2[0];
            }
        }
        return s.substring(begin,begin+maxLen);
    }

    // 返回以[left,right]为边界子串进行扩展的最长回文子串的信息，int[0]存储回文串的起始下标，int[1]存储回文串的长度
    private int[] expand(int left,int right,String s){
        int[] res=new int[2];
        while (left>=0&&right<s.length()&&s.charAt(left)==s.charAt(right)){
            left--;
            right++;
        }
        res[0]=left+1;
        res[1]=right-left-1;
        return res;
    }
}
```

2. 1143. 最长公共子序列 - 力扣（LeetCode）

题目简述：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。如果不存在 **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。

解题思路：用`DP[i][j]` 表示 `text1[0 ... i]` 和 `text2[0 ... j]` 的最长公共子序列。如果 text1[i] 等于 text2[j]，则 `DP[i][j] = DP[i - 1][j - 1] + 1`；否则，`DP[i][j] = max(DP[i - 1][j], DP[i][j - 1])`。边界条件：当 i=0时，text1[0:i]为空，空字符串和任何字符串的最长公共子序列的长度都是0，因此对于任意 0≤j≤n，有 `dp[0][`j`]=0`。当 j=0 时，text2[0:j]为空，同理可得，对任意 0≤i≤m，有 `dp[`i`][0]=0`。

解题代码：

```java
class Solution {
    public int longestCommonSubsequence(String text1, String text2) {
        int m = text1.length(), n = text2.length();
        int[][] dp = new int[m + 1][n + 1];
        for (int i = 1; i <= m; i++) {
            char c1 = text1.charAt(i - 1);
            for (int j = 1; j <= n; j++) {
                char c2 = text2.charAt(j - 1);
                if (c1 == c2) {
                    dp[i][j] = dp[i - 1][j - 1] + 1;
                } else {
                    dp[i][j] = Math.max(dp[i - 1][j], dp[i][j - 1]);
                }
            }
        }
        return dp[m][n];
    }
}
```

3. 72. 编辑距离 - 力扣（LeetCode）

题目简述：给你两个单词 `word1` 和 `word2`， *请返回将 `word1` 转换成 `word2` 所使用的最少操作数*。

你可以对一个单词进行如下三种操作：

- 插入一个字符
- 删除一个字符
- 替换一个字符

解题思路：动态规划：

`dp[i][j]` 代表 `word1` 到 `i` 位置转换成 `word2` 到 `j` 位置需要最少步数

所以，

当 `word1[i] == word2[j]`， `dp[i][j] = dp[i-1][j-1]`；

当 `word1[i] != word2[j]`， `dp[i][j] = min(dp[i-1][j-1], dp[i-1][j], dp[i][j-1]) + 1`

其中，`dp[i-1][j-1]` 表示替换操作，`dp[i-1][j]` 表示删除操作，`dp[i][j-1]` 表示插入操作。

注意，针对第一行，第一列要单独考虑，我们引入 `''` 下图所示：

	‘ ’	r	o	s
‘ ’	0	1	2	3
h	1			
o	2			
r	3			
s	4			
e	5			

第一行，是 `word1` 为空变成 `word2` 最少步数，就是插入操作

第一列，是 `word2` 为空，需要的最少步数，就是删除操作

解题代码：

```java
// 多维数组动态规划
class Solution {
    public int minDistance(String word1, String word2) {
        int[][] dp=new int[word1.length()+1][word2.length()+1];
        dp[0][0]=0;
        for(int j=1;j<=word2.length();j++){
            dp[0][j]=dp[0][j-1]+1;
        }
        for(int i=1;i<=word1.length();i++){
            dp[i][0]=dp[i-1][0]+1;
        }
        for(int i=1;i<=word1.length();i++){
            for(int j=1;j<=word2.length();j++){
                if(word1.charAt(i-1)==word2.charAt(j-1)){
                    dp[i][j]=dp[i-1][j-1];
                }else{
                    dp[i][j]=Math.min(Math.min(dp[i-1][j-1],dp[i-1]
[j]),dp[i][j-1])+1;
                }
            }
        }
        return dp[word1.length()][word2.length()];
    }
}
```

```java
// 优化之后的一维数组动态规划
class Solution {
    public int minDistance(String word1, String word2) {
        char[] s=word1.toCharArray();
        char[] t=word2.toCharArray();
        int[] dp = new int[word2.length()+1];   // dp[i]表示从word1转换到
word2前i个字符所需的最小次数
        for(int j=1;j<=word2.length();j++){    // 当word1为空时，转换成
word2前i个字符所需的最小次数
            dp[j]=j;
        }
        for(int i=1;i<=word1.length();i++){ // 这里的i可以理解为二维dp[i]
[j]中的i
            int pre = dp[0]++;  // pre的实质是二维动归中的dp[i-1][j-1],
dp[0]++表示 dp[i+1][0]=dp[i][0]+1
            for(int j=1;j<=word2.length();j++){
                int temp=dp[j];
                if(s[i-1]==t[j-1]){
                    dp[j]=pre;
                }else{
                    dp[j]=Math.min((Math.min(pre,dp[j])),dp[j-1])+1;
                }
                pre=temp;
            }
        }
        return dp[word2.length()];
    }
}
// 优化成一维的实质是只存储二维中的一行，然后在外层遍历中慢慢更新。
```

4. [516. 最长回文子序列 - 力扣（LeetCode）]

题目简述：给你一个字符串 `s` ，找出其中最长的回文子序列，并返回该序列的长度。

子序列定义为：不改变剩余字符顺序的情况下，删除某些字符或者不删除任何字符形成的一个序列。

解题思路：https://programmercarl.com/0516.%E6%9C%80%E9%95%BF%E5%9B%9E%E6%96%87%E5%AD%90%E5%BA%8F%E5%88%97.html （代码随想录）

https://leetcode.cn/problems/longest-palindromic-subsequence/solutions/15118/dong-tai-gui-hua-si-yao-su-by-a380922457-3/ （力扣精选题解）

动规五部曲分析如下：

1. 确定dp数组（dp table）以及下标的含义

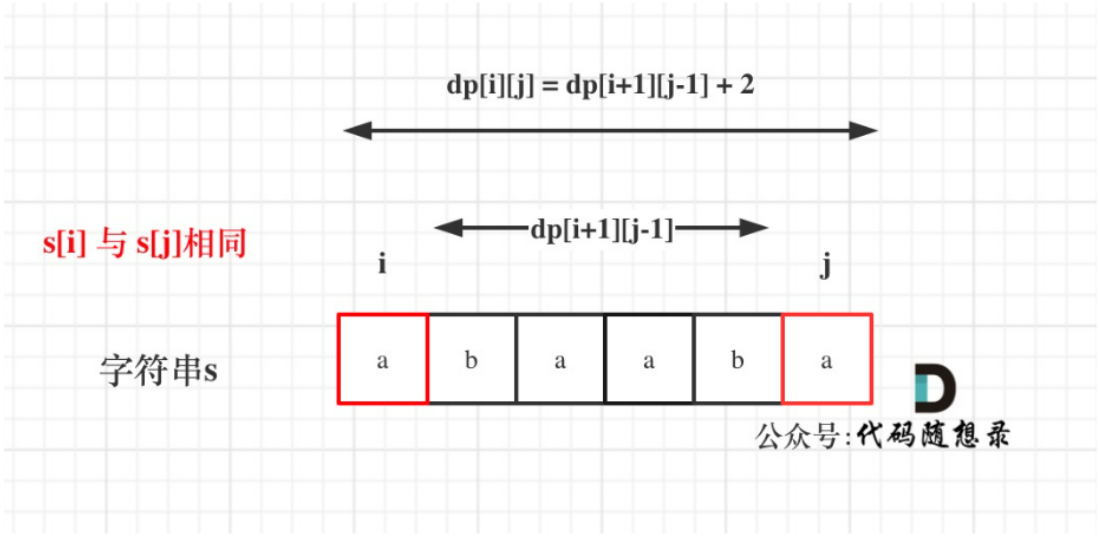
dp[i][j]：字符串s在[i, j]范围内最长的回文子序列的长度为dp[i][j]。

2. 确定递推公式

在判断回文子串的题目中，关键逻辑就是看s[i]与s[j]是否相同。

如果s[i]与s[j]相同，那么dp[i][j] = dp[i + 1][j - 1] + 2;

如图：

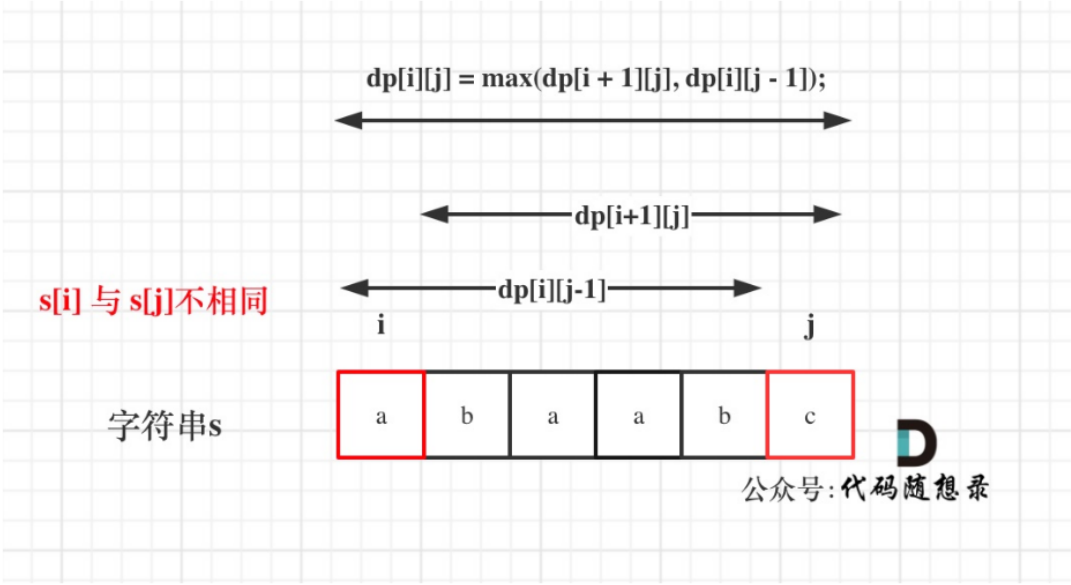


如果s[i]与s[j]不相同，说明s[i]和s[j]的同时加入 并不能增加[i,j]区间回文子序列的长度，那么分别加入s[i]、s[j]看看哪一个可以组成最长的回文子序列。

加入s[j]的回文子序列长度为dp[i + 1][j]。

加入s[i]的回文子序列长度为dp[i][j - 1]。

那么dp[i][j]一定是取最大的，即：dp[i][j] = max(dp[i + 1][j], dp[i][j - 1]);



解题代码：

```
class Solution {
    public static int longestPalindromeSubseq(String s) {
        if (s.length() <= 1) {
            return s.length();
        }
        int[][] dp = new int[s.length()][s.length()];
        for (int i = 0; i < s.length(); i++) {
            dp[i][i] = 1;
        }
        for (int i = 0; i < s.length() - 1; i++) {
            if (s.charAt(i) == s.charAt(i + 1)) {
                dp[i][i + 1] = 2;
            } else {
                dp[i][i + 1] = 1;
            }
        }
        for (int len = 3; len <= s.length(); len++) {
            for (int i = 0; i < s.length() - len + 1; i++) {
                if (s.charAt(i) == s.charAt(i + len - 1)) {
                    dp[i][i + len - 1] = dp[i + 1][i + len - 2] + 2;
                } else {
                    dp[i][i + len - 1] = Math.max(dp[i + 1][i + len - 1], dp[i][i + len - 2]);
                }
            }
        }

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

```
29 }
```

栈

> java官方文档推荐用deque实现栈（stack）。Deque是double ended queue，将其理解成双端结束的队列，双端队列，可以在首尾插入或删除元素（注意与Queue的区别，Queue是FIFO的单端队列，Deque是双端队列）。
>
> **接口分析：**
>
> - addFirst(): 向队头插入元素，如果元素为空，则发生NPE(空指针异常)
> - addLast(): 向队尾插入元素，如果为空，则发生NPE
> - offerFirst(): 向队头插入元素，如果插入成功返回true，否则返回false
> - offerLast(): 向队尾插入元素，如果插入成功返回true，否则返回false
> - removeFirst(): 返回并移除队头元素，如果该元素是null，则发生NoSuchElementException
> - removeLast(): 返回并移除队尾元素，如果该元素是null，则发生NoSuchElementException
> - pollFirst(): 返回并移除队头元素，如果队列无元素，则返回null
> - pollLast(): 返回并移除队尾元素，如果队列无元素，则返回null
> - getFirst(): 获取队头元素但不移除，如果队列无元素，则发生NoSuchElementException
> - getLast(): 获取队尾元素但不移除，如果队列无元素，则发生NoSuchElementException
> - peekFirst(): 获取队头元素但不移除，如果队列无元素，则返回null
> - peekLast(): 获取队尾元素但不移除，如果队列无元素，则返回null
> - peek():获取队头元素但不移除，如果队列无元素，则返回null（用作栈时相当于top()操作）。
> - pop(): 弹出栈中元素，也就是返回并移除队头元素，等价于removeFirst()，如果队列无元素，则发生NoSuchElementException
> - push(): 向栈中压入元素，也就是向队头增加元素，等价于addFirst()，如果元素为null，则发生NPE，如果栈空间受到限制，则发生IllegalStateException
>
> **实现**
>
> - ArrayDeque: 基于数组实现的线性双向队列，通常作为栈或队列使用，但是栈的效率不如LinkedList高。
> - LinkedList: 基于链表实现的链式双向队列，通常作为栈或队列使用，但是队列的效率不如ArrayQueue高。
>
> **代码示例**
>
> ```
> 1 Deque<Integer> stack = new LinkedList<>();
> 2 stack.push(-1); // 入栈
> 3 stack.pop(); // 出栈
> 4 stack.peek() // 相当于top()操作，获取队头（栈顶）元素
> ```

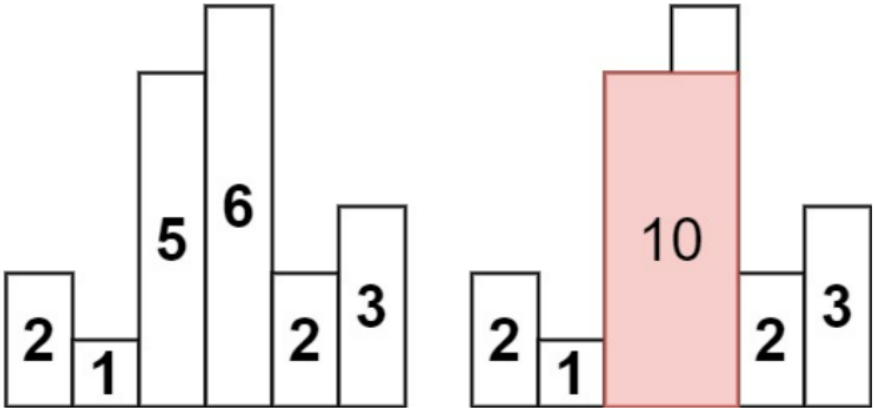
1. 84. 柱状图中最大的矩形 - 力扣（LeetCode）

题目简述：

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。

示例 1:



> **输入:** heights = [2,1,5,6,2,3]
> **输出:** 10
> **解释:** 最大的矩形为图中红色区域，面积为 10

解题思路1（单调栈）： 84. 柱状图中最大的矩形 - 力扣（LeetCode）

单调栈

1. 单调栈分为单调递增栈和单调递减栈
   11. 单调递增栈即栈内元素保持单调递增的栈

      l. 同理单调递减栈即栈内元素保持单调递减的栈
2. 操作规则（下面都以单调递增栈为例）
   21. 如果新的元素比栈顶元素大，就入栈

      v. 如果新的元素较小，那就一直把栈内元素弹出来，直到栈顶比新元素小
3. 加入这样一个规则之后，会有什么效果
   31. 栈内的元素是递增的

      af. 当元素出栈时，说明这个**新元素**是出栈元素**向后**找第一个比其小的元素

      > 举个例子，配合下图，现在索引在 6 ，栈里是 1 5 6 。
      > 接下来新元素是 2 ，那么 6 需要出栈。
      > 当 6 出栈时，右边 2 代表是 6 右边第一个比 6 小的元素。

      ag. 当元素出栈后，说明新**栈顶元素**是出栈元素**向前**找第一个比其小的元素

      > 当 6 出栈时，5 成为新的栈顶，那么 5 就是 6 左边第一个比 6 小的元素。

4. 代码模板

C++

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

解题代码（两个他人的题解&一个自己的题解）：

```
// 他人题解
class Solution {
    public int largestRectangleArea(int[] heights) {
        /*
        只做单调栈思路:参考"编程狂想曲"思路比较好理解
        1.核心思想:求每条柱子可以向左右延伸的长度->矩形最大宽度;矩形的高->柱子的高度
            计算以每一根柱子高度为高的矩形面积,维护面积最大值
        2.朴素的想法:遍历每一根柱子的高度然后向两边进行扩散找到最大宽度
        3.单调栈优化:因为最终的目的是寻找对应柱子height[i]右边首个严格小于height[i]的柱子height[r]
            左边同理找到首个严格小于height[i]的柱子height[l]
            维护一个单调递增栈(栈底->栈顶),那么每当遇到新加入的元素<栈顶便可以确定栈顶柱子右边界
            而栈顶柱子左边界就是栈顶柱子下面的柱子(<栈顶柱子)
            左右边界确定以后就可以进行面积计算与维护最大面积
        时间复杂度:O(N),空间复杂度:O(N)
        */
        // 引入哨兵
        // 哨兵的作用是 将最后的元素出栈计算面积 以及 将开头的元素顺利入栈
        // len为引入哨兵后的数组长度
        int len = heights.length + 2;
        int[] newHeight = new int[len];
        newHeight[0] = newHeight[len - 1] = 0;
        // [1,2,3]->[0,1,2,3,0]
        for(int i = 1; i < len - 1; i++) {
            newHeight[i] = heights[i - 1];
        }
        // 单调递增栈:存储每个柱子的索引,使得这些索引对应的柱子高度单调递增
        Stack<Integer> stack = new Stack<>();
        // 最大矩形面积
        int res = 0;
        // 遍历哨兵数组
        for(int i = 0; i < len; i++) {
            // 栈不为空且当前柱子高度<栈顶索引对应的柱子高度
            // 说明栈顶元素的右边界已经确定,就是索引为i的柱子(不含)
            // 此时将栈顶元素出栈,栈顶矩形左边界为栈顶元素下面的索引(首个小于栈顶)
            while(!stack.empty() && newHeight[i] < newHeight[stack.peek()]) {
```

```
                // 栈顶索引出栈并记录
                int pop = stack.pop();
                // 计算出栈顶元素矩形的宽度如(0,1,2)->[1,2,1],两边都不包含
                // 因此右索引-左索引-1=矩形宽度
                int w = i - stack.peek() - 1;
                // 栈顶索引对应的柱子高度就是矩形的高度
                int h = newHeight[pop];
                // 计算矩形面积
                int area = w * h;
                // 维护矩形面积最大值
                res = Math.max(res, area);
            }
            // 每当弹出一个索引就计算一个矩形面积
            // 直到当前元素>=栈顶元素(或者栈为空)时,栈顶柱子的右边界还没确定
            // 因此当前元素索引入栈即可
            stack.push(i);
        }
        return res;
    }
}
```

```
// 自己的题解（用了数组模拟单调栈，运行速度会更快）
class Solution {
    public int largestRectangleArea(int[] heights) {
        int max = 0;
        int[] stack = new int[heights.length + 2]; // 模拟单调栈（存储索引）
        int topIndex = -1; // 栈顶索引
        stack[++topIndex] = -1; // 先把最左侧的索引放进去（即0的左侧，-1）
        for (int i = 0; i < heights.length; i++) {
            if (topIndex == 0) {
                stack[++topIndex] = i;
            } else {
                while (topIndex > 0 && heights[i] < heights[stack[topIndex]]) {
                    int height = heights[stack[topIndex--]];
                    int width = i - stack[topIndex] - 1;
                    max = Math.max(max, height * width);
                }
                stack[++topIndex] = i;
            }
        }
        // 最后还有把0放进去模拟一遍，防止出现漏算的情况（0即代表最后一根柱子右侧的高度，即0））
        while (topIndex > 0 && 0 < heights[stack[topIndex]]) {
            int height = heights[stack[topIndex--]];
            int width = heights.length - stack[topIndex] - 1;
            max = Math.max(max, height * width);
        }
        return max;
    }

}
```

```java
// 代码随想录题解（有优化空间，可以用数组模拟单调栈，速度会更快）
class Solution {
    public int largestRectangleArea(int[] heights) {
        int[] newHeight = new int[heights.length + 2];
        System.arraycopy(heights, 0, newHeight, 1, heights.length);
        newHeight[heights.length+1] = 0;
        newHeight[0] = 0;

        Stack<Integer> stack = new Stack<>();
        stack.push(0);

        int res = 0;
        for (int i = 1; i < newHeight.length; i++) {
            while (newHeight[i] < newHeight[stack.peek()]) {
                int mid = stack.pop();
                int w = i - stack.peek() - 1;
                int h = newHeight[mid];
                res = Math.max(res, w * h);
            }
            stack.push(i);

        }
        return res;
    }
}
```

解题思路2（双指针）： 代码随想录 (programmercarl.com)，本质就是求每一个高度作为矩形高的情况下的最大宽度，然后遍历求出最大值

解题代码：

```java
class Solution {
    public int largestRectangleArea(int[] heights) {
        int max = 0;
        int[] minLeftIndex = new int[heights.length];
        int[] minRightIndex = new int[heights.length];

        // 记录每个柱子 左边第一个小于该柱子的下标
        minLeftIndex[0] = -1; // 注意这里初始化，防止下面while死循环
        for (int i = 1; i < heights.length; i++) {
            int j = i - 1;
            // 这里不是用if，而是不断向左寻找的过程
            while (j >= 0 && heights[j] >= heights[i]) {
                j = minLeftIndex[j];
            }
            minLeftIndex[i] = j;
        }

        // 记录每个柱子 右边第一个小于该柱子的下标
        minRightIndex[heights.length - 1] = heights.length; // 注意这里初始化，防止下面while死循环
        for (int i = heights.length - 2; i >= 0; i--) {
            int j = i + 1;
```

```
            // 这里不是用if，而是不断向右寻找的过程
            while (j < heights.length && heights[j] >= heights[i]) {
                j = minRightIndex[j];
            }
            minRightIndex[i] = j;
        }

        // 遍历求最大
        for (int i = 0; i < heights.length; i++) {
            int width = minRightIndex[i] - minLeftIndex[i] - 1;
            int tempMax = width * heights[i];
            max = Math.max(max, tempMax);
        }

        return max;
    }
}
```

堆

1. 215. 数组中的第K个最大元素 - 力扣（LeetCode）

题目简述：给定整数数组 `nums` 和整数 `k`，请返回数组中第 `k` 个最大的元素。请注意，你需要找的是数组排序后的第 `k` 个最大的元素，而不是第 `k` 个不同的元素。必须设计并实现时间复杂度为 `O(n)` 的算法解决此问题。

解题思路一（基于快速排序的选择方法）：

我们可以用快速排序来解决这个问题，先对原数组排序，再返回倒数第 k 个位置，这样平均时间复杂度是 $O(n \log n)$，但其实我们可以做的更快。

首先我们来回顾一下快速排序，这是一个典型的分治算法。我们对数组 $a[l \cdots r]$ 做快速排序的过程是（参考《算法导论》）：

- **分解：** 将数组 $a[l \cdots r]$ 「划分」成两个子数组 $a[l \cdots q-1]$、$a[q+1 \cdots r]$，使得 $a[l \cdots q-1]$ 中的每个元素小于等于 $a[q]$，且 $a[q]$ 小于等于 $a[q+1 \cdots r]$ 中的每个元素。其中，计算下标 q 也是「划分」过程的一部分。
- **解决：** 通过递归调用快速排序，对子数组 $a[l \cdots q-1]$ 和 $a[q+1 \cdots r]$ 进行排序。
- **合并：** 因为子数组都是原址排序的，所以不需要进行合并操作，$a[l \cdots r]$ 已经有序。
- 上文中提到的 **「划分」** 过程是：从子数组 $a[l \cdots r]$ 中选择任意一个元素 x 作为主元，**调整子数组的元素使得左边的元素都小于等于它，右边的元素都大于等于它，** x 的最终位置就是 q。

由此可以发现每次经过「划分」操作后，我们一定可以确定一个元素的最终位置，即 x 的最终位置为 q，并且保证 $a[l \cdots q-1]$ 中的每个元素小于等于 $a[q]$，且 $a[q]$ 小于等于 $a[q+1 \cdots r]$ 中的每个元素。**所以只要某次划分的 q 为倒数第 k 个下标的时候，我们就已经找到了答案。** 我们只关心这一点，至于 $a[l \cdots q-1]$ 和 $a[q+1 \cdots r]$ 是否是有序的，我们不关心。

因此我们可以改进快速排序算法来解决这个问题：在分解的过程当中，我们会对子数组进行划分，如果划分得到的 q 正好就是我们需要的下标，就直接返回 $a[q]$；否则，如果 q 比目标下标小，就递归右子区间，否则递归左子区间。这样就可以把原来递归两个区间变成只递归一个区间，提高了时间效率。这就是「快速选择」算法。

解题代码：

```java
class Solution {
    public int findKthLargest(int[] nums, int k) {
        return quickSelect(nums, 0, nums.length - 1, nums.length - k);
    }

    private int quickSelect(int[] nums, int left, int right, int target)
{
        int index = partition(nums, left, right);
        if (index == target) {
            return nums[index];
        } else {
            return index > target ?
                quickSelect(nums, left, index - 1, target) :
                quickSelect(nums, index + 1, right, target);
        }
    }

    private int partition(int[] nums, int left, int right) {
        swap(nums, left, left + new Random().nextInt(right - left + 1));
        int pivot = nums[left];
        while (left < right) {
            while (left < right && nums[right] > pivot) {
                right--;
            }
            if (left < right) {
                nums[left++] = nums[right];
            }
            while (left < right && nums[left] < pivot) {
                left++;
            }
            if (left < right) {
                nums[right--] = nums[left];
            }
        }
        nums[left] = pivot;
        return left;
    }

    private void swap(int[] nums, int i, int j) {
        int swap = nums[i];
        nums[i] = nums[j];
        nums[j] = swap;
    }
}
```

解题思路二（计数排序）：根据题目的约束条件，`-10000 <= nums[i] <= 10000`，可以构造一个大小为20001的桶，用来存放对应数在数组中的个数，然后从大往小依次遍历每个桶，输出第K大的元素。

解题代码：

```java
class Solution {
    public int findKthLargest(int[] nums, int k) {
        int[] buckets = new int[20001];
        for(int i=0;i< nums.length;i++){
```

```
            buckets[nums[i]+10000]++;
        }
        for (int i=20000;i>=0;i--){
            k-=buckets[i];
            if(k<=0){
                return i-10000;
            }
        }
        return 0;
    }
}
```

解题思路三（基于堆排序的选择方法）：

也可以使用堆排序来解决这个问题——建立一个大根堆，做 k-1次删除操作后堆顶元素就是我们要找的答案。在这道题中尤其要搞懂「建堆」、「调整」和「删除」的过程（自己维护一个堆）。时间复杂度为`O(nlogn)`，空间复杂度为`O(logn)`

解题代码：

```
// 使用PriorityQueue来实现
import java.util.Comparator;
import java.util.PriorityQueue;

public class Solution {

    public int findKthLargest(int[] nums, int k) {
        int len = nums.length;
        // 使用一个含有 k 个元素的最小堆，PriorityQueue 底层是动态数组，为了防止数组扩容产生消耗，可以先指定数组的长度
        PriorityQueue<Integer> minHeap = new PriorityQueue<>(k, Comparator.comparingInt(a -> a));
        // Java 里没有 heapify ，因此我们逐个将前 k 个元素添加到 minHeap 里
        for (int i = 0; i < k; i++) {
            minHeap.offer(nums[i]);
        }

        for (int i = k; i < len; i++) {
            // 看一眼，不拿出，因为有可能没有必要替换
            Integer topElement = minHeap.peek();
            // 只要当前遍历的元素比堆顶元素大，堆顶弹出，遍历的元素进去
            if (nums[i] > topElement) {
                // Java 没有 replace()，所以得先 poll() 出来，然后再放回去
                minHeap.poll();
                minHeap.offer(nums[i]);
            }
        }
        return minHeap.peek();
    }
}
```

```java
// 使用PriorityQueue来实现的解法2
class Solution {
    public int findKthLargest(int[] nums, int k) {
        PriorityQueue<Integer> heap = new PriorityQueue<>();
        for (int num : nums) {
            heap.add(num);
            if (heap.size() > k) {
                heap.poll();
            }
        }
        return heap.peek();
    }
}
```

```java
// 手动构造一个堆
class Solution {
    public int findKthLargest(int[] nums, int k) {
        buildHeap(nums,nums.length);
        int size=nums.length;
        for(int i=1;i<k;i++){
            swap(nums,0,size-1);
            size--;
            maxHeapify(nums,0,size);
        }
        return nums[0];
    }

    private void buildHeap(int[] nums,int size){
        for(int i=size/2-1;i>=0;i--){
            maxHeapify(nums,i,size);
        }
    }

    private void maxHeapify(int[] nums,int index, int size){
        int child=2*index+1;
        while (child<size){
            if(child<size-1&&nums[child]<nums[child+1]){
                child++;
            }
            if(nums[index]>=nums[child]){
                break;
            }
            swap(nums,index,child);
            index=child;
            child=2*child+1;
        }
    }

    private void swap(int[] nums,int i,int j){
        int temp=nums[i];
        nums[i]=nums[j];
        nums[j]=temp;
    }
}
```

2. [347. 前 K 个高频元素 - 力扣（LeetCode）]

题目简述：给你一个整数数组 `nums` 和一个整数 `k` ，请你返回其中出现频率前 `k` 高的元素。你可以按 **任意顺序** 返回答案。

解题思路：首先遍历整个数组，并使用哈希表记录每个数字出现的次数，并形成一个「出现次数数组」。然后维护一个大小为K的小顶堆，然后遍历「出现次数数组」：

- 如果堆的元素个数小于 k，就可以直接插入堆中。
- 如果堆的元素个数等于 k，则检查堆顶与当前出现次数的大小。如果堆顶更大，说明至少有 k 个数字的出现次数比当前值大，故舍弃当前值；否则，就弹出堆顶，并将当前值插入堆中。

解题代码（个人简化版）：

```
// 哈希表 + 优先队列
class Solution {
    public int[] topKFrequent(int[] nums, int k) {
        int[] ans=new int[k];
        Map<Integer,Integer> map =new HashMap<>();
        for(int num:nums){
            map.put(num,map.getOrDefault(num,0)+1);
        }
        // 构造一个降序排序的优先队列
        PriorityQueue<Map.Entry<Integer,Integer>> queue=new
PriorityQueue<>((o1, o2) -> o2.getValue()-o1.getValue());
        queue.addAll(map.entrySet());
        for(int i=0;i<k;i++){
            ans[i]=queue.poll().getKey();
        }
        return ans;
    }
}
```

解题代码（官方版）：

```
// 官方的代码，把哈希表转换成一个大小为2的数组，第一个元素代表数组的值，第二个元素代表
了该值出现的次数，然后再利用
// PriorityQueue来优化
class Solution {
    public int[] topKFrequent(int[] nums, int k) {
        Map<Integer, Integer> occurrences = new HashMap<Integer,
Integer>();
        for (int num : nums) {
            occurrences.put(num, occurrences.getOrDefault(num, 0) + 1);
        }

        // int[] 的第一个元素代表数组的值，第二个元素代表了该值出现的次数
        PriorityQueue<int[]> queue = new PriorityQueue<int[]>(new
Comparator<int[]>() {
            public int compare(int[] m, int[] n) {
                return m[1] - n[1];
            }
        });
        for (Map.Entry<Integer, Integer> entry : occurrences.entrySet())
{
            int num = entry.getKey(), count = entry.getValue();
```

```
            if (queue.size() == k) {
                if (queue.peek()[1] < count) {
                    queue.poll();
                    queue.offer(new int[]{num, count});
                }
            } else {
                queue.offer(new int[]{num, count});
            }
        }
        int[] ret = new int[k];
        for (int i = 0; i < k; ++i) {
            ret[i] = queue.poll()[0];
        }
        return ret;
    }
}
```

3. 295. 数据流的中位数 - 力扣（LeetCode）

题目简述：实现 MedianFinder 类:

- `MedianFinder()` 初始化 `MedianFinder` 对象。
- `void addNum(int num)` 将数据流中的整数 `num` 添加到数据结构中。
- `double findMedian()` 返回到目前为止所有元素的中位数。与实际答案相差 `10-5` 以内的答案将被接受。

解题思路一（优先队列）：

思路和算法

我们用两个优先队列 $queMax$ 和 $queMin$ 分别记录大于中位数的数和小于等于中位数的数。当累计添加的数的数量为奇数时，$queMin$ 中的数的数量比 $queMax$ 多一个，此时中位数为 $queMin$ 的队头。当累计添加的数的数量为偶数时，两个优先队列中的数的数量相同，此时中位数为它们的队头的平均值。

当我们尝试添加一个数 num 到数据结构中，我们需要分情况讨论：

1. $num \le \max\{queMin\}$

   此时 num 小于等于中位数，我们需要将该数添加到 $queMin$ 中。新的中位数将小于等于原来的中位数，因此我们可能需要将 $queMin$ 中最大的数移动到 $queMax$ 中。

2. $num > \max\{queMin\}$

   此时 num 大于中位数，我们需要将该数添加到 $queMin$ 中。新的中位数将大于等于原来的中位数，因此我们可能需要将 $queMax$ 中最小的数移动到 $queMin$ 中。

特别地，当累计添加的数的数量为 0 时，我们将 num 添加到 $queMin$ 中。

解题代码：

```
class MedianFinder {
    PriorityQueue<Integer> queMin;
    PriorityQueue<Integer> queMax;

    public MedianFinder() {
        queMin = new PriorityQueue<Integer>((a, b) -> (b - a));
        queMax = new PriorityQueue<Integer>((a, b) -> (a - b));
    }
```

```java

    public void addNum(int num) {
        if (queMin.isEmpty() || num <= queMin.peek()) {
            queMin.offer(num);
            if (queMax.size() + 1 < queMin.size()) {
                queMax.offer(queMin.poll());
            }
        } else {
            queMax.offer(num);
            if (queMax.size() > queMin.size()) {
                queMin.offer(queMax.poll());
            }
        }
    }

    public double findMedian() {
        if (queMin.size() > queMax.size()) {
            return queMin.peek();
        }
        return (queMin.peek() + queMax.peek()) / 2.0;
    }
}
```

解题思路二（有序集合 + 双指针）：

思路和算法

我们也可以使用有序集合维护这些数。我们把有序集合看作自动排序的数组，使用双指针指向有序集合中的中位数元素即可。当累计添加的数的数量为奇数时，双指针指向同一个元素。当累计添加的数的数量为偶数时，双指针分别指向构成中位数的两个数。

当我们尝试添加一个数 num 到数据结构中，我们需要分情况讨论：

1. 初始有序集合为空时，我们直接让左右指针指向 num 所在的位置。
2. 有序集合为中元素为奇数时，$left$ 和 $right$ 同时指向中位数。如果 num 大于等于中位数，那么只要让 $left$ 左移，否则让 $right$ 右移即可。
3. 有序集合为中元素为偶数时，$left$ 和 $right$ 分别指向构成中位数的两个数。
    - 当 num 成为新的唯一的中位数，那么我们让 $left$ 右移，$right$ 左移，这样它们即可指向 num 所在的位置；
    - 当 num 大于等于 $right$，那么我们让 $left$ 右移即可；
    - 当 num 小于 $right$ 指向的值，那么我们让 $right$ 左移，注意到如果 num 恰等于 $left$ 指向的值，那么 num 将被插入到 $left$ 右侧，使得 $left$ 和 $right$ 间距增大，所以我们还需要额外让 $left$ 指向移动后的 $right$。

解题代码：

```java
// 略.....
```

技巧

1. 136. 只出现一次的数字 - 力扣（LeetCode）

   题目简述：给你一个 **非空** 整数数组 `nums` ，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。必须设计并实现线性时间复杂度的算法来解决此问题，且该算法只使用常量额外空间。

   解题思路：用异或运算：

   1. 交换律：a ^ b ^ c <=> a ^ c ^ b
   2. 任何数于0异或为任何数 0 ^ n => n
   3. 相同的数异或为0: n ^ n => 0

   解题代码：

```java
class Solution {
    public int singleNumber(int[] nums) {
        int res=0;
        for(int num:nums){
            res^=num;
        }
        return res;
    }
}
```

2. 169. 多数元素 - 力扣（LeetCode）

   题目简述：给定一个大小为 `n` 的数组 `nums` ，返回其中的多数元素。多数元素是指在数组中出现次数 **大于** `⌊ n/2 ⌋` 的元素。

   解题思路：该题目的解法众多，这里只介绍最为巧妙的时间复杂度为 O(n)、空间复杂度为 O(1) 的算法(**Boyer-Moore 投票算法**)。

   Boyer-Moore 算法的本质和方法四中的分治十分类似。我们首先给出 Boyer-Moore 算法的详细步骤：

   - 我们维护一个候选众数 `candidate` 和它出现的次数 `count` 。初始时 `candidate` 可以为任意值， `count` 为 `0` ；
   - 我们遍历数组 `nums` 中的所有元素，对于每个元素 `x` ，在判断 `x` 之前，如果 `count` 的值为 `0` ，我们先将 `x` 的值赋予 `candidate` ，随后我们判断 `x` ：
     - 如果 `x` 与 `candidate` 相等，那么计数器 `count` 的值增加 `1` ；
     - 如果 `x` 与 `candidate` 不等，那么计数器 `count` 的值减少 `1` 。
   - 在遍历完成后， `candidate` 即为整个数组的众数。

   解题代码：

```java
class Solution {
    public int majorityElement(int[] nums) {
        int count = 0;
        Integer candidate = null;

        for (int num : nums) {
            if (count == 0) {
                candidate = num;
```

```
            }
            count += (num == candidate) ? 1 : -1;
        }

        return candidate;
    }
}
```

3. 287. 寻找重复数 - 力扣（LeetCode）

题目简述：给定一个包含 `n + 1` 个整数的数组 `nums`，其数字都在 `[1, n]` 范围内（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**，返回 **这个重复的数**。你设计的解决方案必须 **不修改** 数组 `nums` 且只用常量级 `O(1)` 的额外空间

解题思路：此题较难，直接看官方题解，三种思路（最后一种线性时间复杂度）287. 寻找重复数 - 力扣（LeetCode）

解题代码：

```
// 二分查找法
class Solution {
    public int findDuplicate(int[] nums) {
        int n = nums.length;
        int l = 1, r = n - 1, ans = -1;
        while (l <= r) {
            int mid = (l + r) >> 1;
            int cnt = 0;
            for (int i = 0; i < n; ++i) {
                if (nums[i] <= mid) {
                    cnt++;
                }
            }
            if (cnt <= mid) {
                l = mid + 1;
            } else {
                r = mid - 1;
                ans = mid;
            }
        }
        return ans;
    }
}
```

```
// 二进制法
class Solution {
    public int findDuplicate(int[] nums) {
        int n = nums.length, ans = 0;
        int bit_max = 31;
        while (((n - 1) >> bit_max) == 0) {
            bit_max -= 1;
        }
        for (int bit = 0; bit <= bit_max; ++bit) {
            int x = 0, y = 0;
            for (int i = 0; i < n; ++i) {
                if ((nums[i] & (1 << bit)) != 0) {
```

```
                    x += 1;
                }
                if (i >= 1 && ((i & (1 << bit)) != 0)) {
                    y += 1;
                }
            }
            if (x > y) {
                ans |= 1 << bit;
            }
        }
        return ans;
    }
}
```

```
// 快慢指针法
class Solution {
    public int findDuplicate(int[] nums) {
        int slow = 0, fast = 0;
        do {
            slow = nums[slow];
            fast = nums[nums[fast]];
        } while (slow != fast);
        slow = 0;
        while (slow != fast) {
            slow = nums[slow];
            fast = nums[fast];
        }
        return slow;
    }
}
```

4. 31. 下一个排列 - 力扣（LeetCode）

题目简述：整数数组的 **下一个排列** 是指其整数的下一个字典序更大的排列。更正式地，如果数组的所有排列根据其字典顺序从小到大排列在一个容器中，那么数组的 **下一个排列** 就是在这个有序容器中排在它后面的那个排列。如果不存在下一个更大的排列，那么这个数组必须重排为字典序最小的排列（即，其元素按升序排列）。

- 例如，`arr = [1,2,3]` 的下一个排列是 `[1,3,2]` 。
- 类似地，`arr = [2,3,1]` 的下一个排列是 `[3,1,2]` 。
- 而 `arr = [3,2,1]` 的下一个排列是 `[1,2,3]` ，因为 `[3,2,1]` 不存在一个字典序更大的排列。

给你一个整数数组 `nums` ，找出 `nums` 的下一个排列。

必须 **原地** 修改，只允许使用额外常数空间

解题思路：31. 下一个排列 - 力扣（LeetCode）

算法推导

如何得到这样的排列顺序？这是本文的重点。我们可以这样来分析：

1. 我们希望下一个数 **比当前数大**，这样才满足 “下一个排列” 的定义。因此只需要 **将后面的「大数」与前面的「小数」交换**，就能得到一个更大的数。比如 `123456`，将 `5` 和 `6` 交换就能得到一个更大的数 `123465`。
2. 我们还希望下一个数 **增加的幅度尽可能的小**，这样才满足“下一个排列与当前排列紧邻“的要求。为了满足这个要求，我们需要：
    a. 在 **尽可能靠右的低位** 进行交换，需要 **从后向前** 查找
    b. 将一个 **尽可能小的「大数」** 与前面的「小数」交换。比如 `123465`，下一个排列应该把 `5` 和 `4` 交换而不是把 `6` 和 `4` 交换
    c. 将「大数」换到前面后，需要将「大数」后面的所有数 **重置为升序**，**升序排列就是最小的排列**。以 `123465` 为例：首先按照上一步，交换 `5` 和 `4`，得到 `123564`；然后需要将 `5` 之后的数重置为升序，得到 `123546`。显然 `123546` 比 `123564` 更小，`123546` 就是 `123465` 的下一个排列

以上就是求 “下一个排列” 的分析过程。

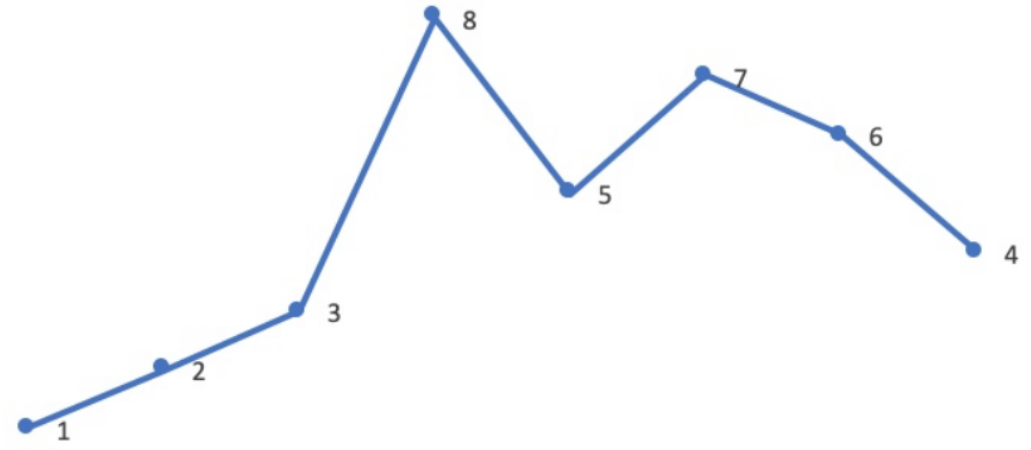
算法过程

标准的 “下一个排列” 算法可以描述为：

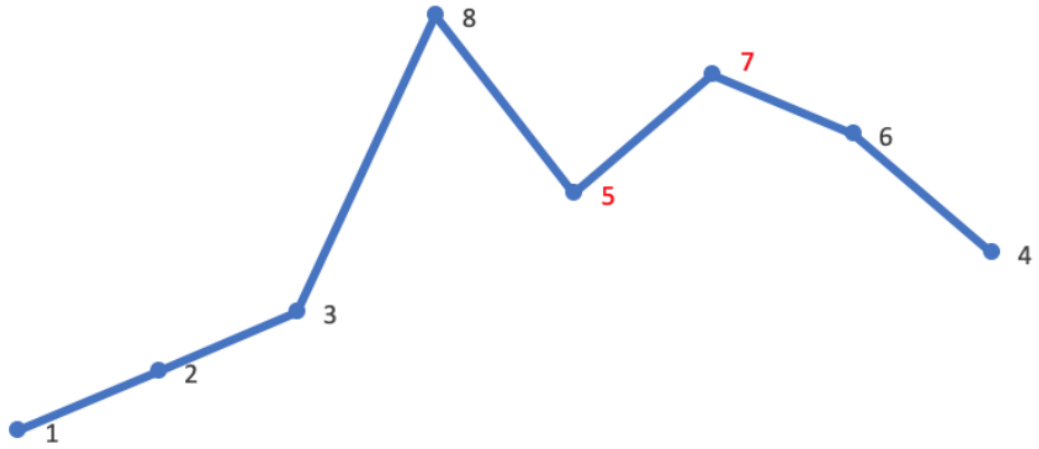
1. **从后向前** 查找第一个 **相邻升序** 的元素对 `(i,j)`，满足 `A[i] < A[j]`。此时 `[j,end)` 必然是降序
2. 在 `[j,end)` **从后向前** 查找第一个满足 `A[i] < A[k]` 的 `k`。`A[i]`、`A[k]` 分别就是上文所说的「小数」、「大数」
3. 将 `A[i]` 与 `A[k]` 交换
4. 可以断定这时 `[j,end)` 必然是降序，逆置 `[j,end)`，使其升序
5. 如果在步骤 1 找不到符合的相邻元素对，说明当前 `[begin,end)` 为一个降序顺序，则直接跳到步骤 4

可视化

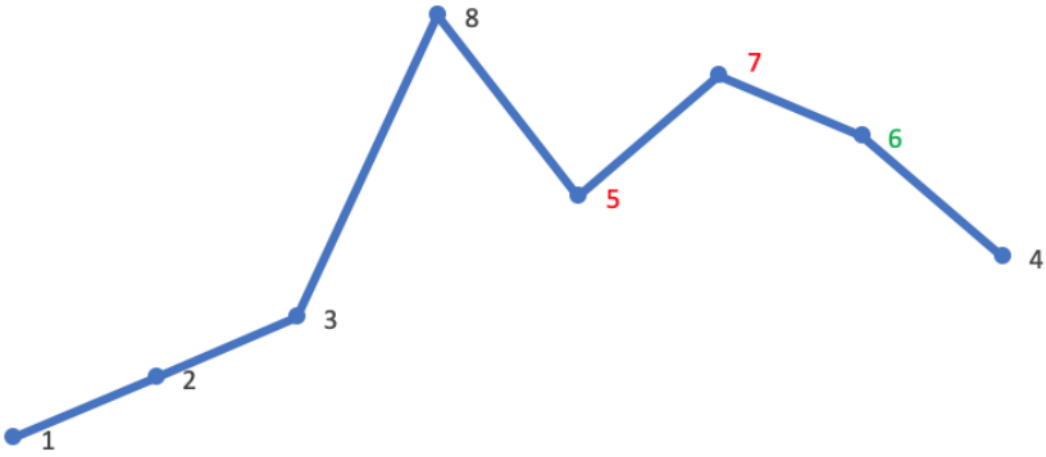
以求 `12385764` 的下一个排列为例：



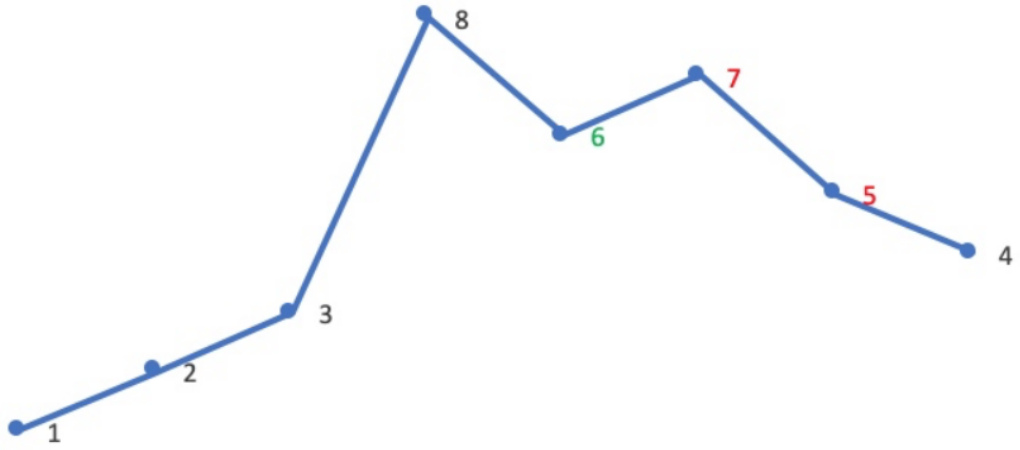
首先从后向前查找第一个相邻升序的元素对 `(i,j)`。这里 `i=4`，`j=5`，对应的值为 `5`，`7`：



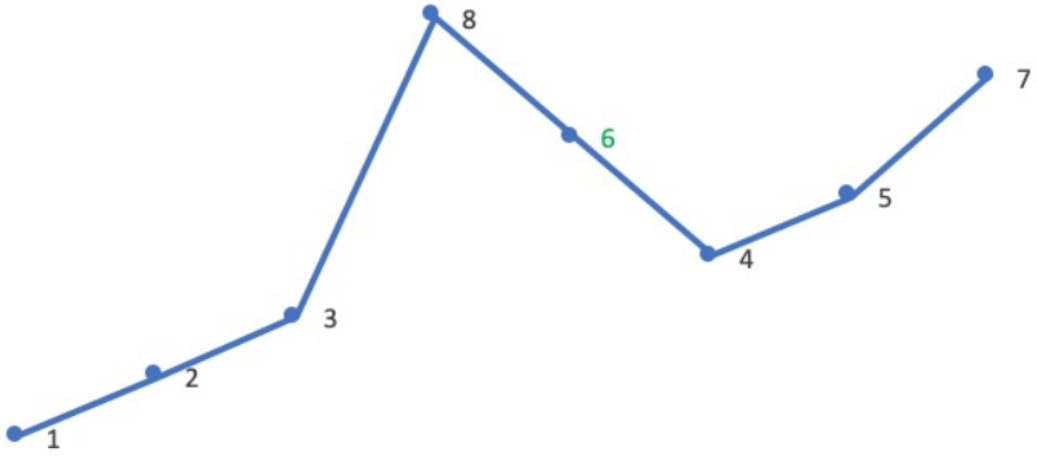
然后在 `[j,end)` 从后向前查找第一个大于 `A[i]` 的值 `A[k]`。这里 `A[i]` 是 `5`，故 `A[k]` 是 `6`：



将 `A[i]` 与 `A[k]` 交换。这里交换 `5` 、 `6` ：



这时 `[j,end)` 必然是降序，逆置 `[j,end)` ，使其升序。这里逆置 `[7,5,4]` ：



因此， `12385764` 的下一个排列就是 `12386457` 。

解题代码：

```
class Solution {
    public void nextPermutation(int[] nums) {
        if(nums.length<2){
```

```
            return;
        }
        int i= nums.length-2,j= nums.length-1;

        // find: A[i]<A[j]
        while (i>=0&&nums[i]>=nums[j]){
            i--;
            j--;
        }

        if(i>=0){ // 不是最后一个排列
            int k= nums.length-1;
            while (nums[i]>=nums[k]){
                k--;
            }
            // swap A[i], A[k]
            int temp=nums[k];
            nums[k]=nums[i];
            nums[i]=temp;
        }

        // reverse A[j:end]
        i=j;
        j= nums.length-1;
        while (i<j){
            int temp=nums[j];
            nums[j]=nums[i];
            nums[i]=temp;
            i++;
            j--;
        }
    }
}
```