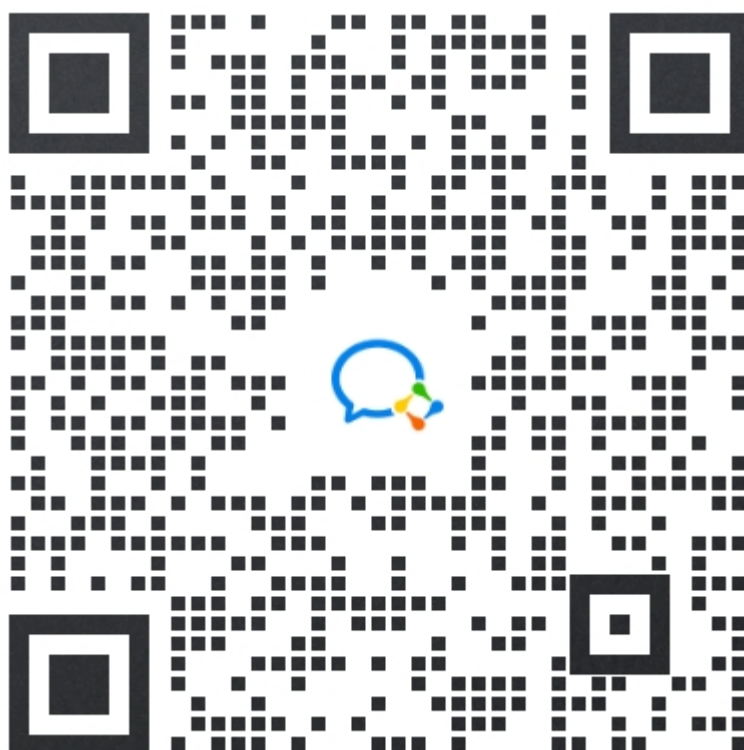


贪心算法精品课

版权说明

本“**比特就业课**”贪心算法精品课（以下简称“本精品课”）的所有内容，包括但不限于文字、图片、音频、视频、软件、程序、数据库、设计、布局、界面等，均由本精品课的开发者或授权方拥有版权。我们鼓励个人学习者使用本精品课进行学习和研究。在遵守相关法律法规的前提下，个人学习者可以下载、浏览、学习本精品课的内容，并为了个人学习、研究或教学目的而使用其中的材料。但请注意，**未经我们明确授权，个人学习者不得将本精品课的内容用于任何商业目的**，包括但不限于销售、转让、许可或以其他方式从中获利。此外，个人学习者也不得擅自修改、复制、传播、展示、表演或制作本精品课内容的衍生作品。任何未经授权的使用均属侵权行为，我们将依法追究法律责任。如果您希望以其他方式使用本精品课的内容，包括但不限于引用、转载、摘录、改编等，请事先与我们联系，获取书面授权。感谢您对“比特就业课”贪心算法精品课的关注与支持，我们将持续努力，为您提供更好的学习体验。特此说明。比特就业课版权所有方。

对比特算法感兴趣，可以联系这个微信。



板书链接

<https://gitee.com/wu-xiaozhe/algorithm>

精选题目

1. 柠檬水找零 (easy)

1. 题目链接: [860. 柠檬水找零](#)

2. 题目描述

在柠檬水摊上，每一杯柠檬水的售价为 5 美元。顾客排队购买你的产品，（按账单 `bills` 支付的顺序）一次购买一杯。

每位顾客只买一杯柠檬水，然后向你付 5 美元、10 美元或 20 美元。你必须给每个顾客正确找零，也就是说净交易是每位顾客向你支付 5 美元。

注意，一开始你手头没有任何零钱。

给你一个整数数组 `bills`，其中 `bills[i]` 是第 `i` 位顾客付的账。如果你能给每位顾客正确找零，返回 `true`，否则返回 `false`。

示例 1:

输入: `bills = [5,5,5,10,20]`

输出: `true`

解释:

前 3 位顾客那里，我们按顺序收取 3 张 5 美元的钞票。

第 4 位顾客那里，我们收取一张 10 美元的钞票，并返还 5 美元。

第 5 位顾客那里，我们找还一张 10 美元的钞票和一张 5 美元的钞票。

由于所有客户都得到了正确的找零，所以我们输出 `true`。

示例 2:

输入: `bills = [5,5,10,10,20]`

输出: `false`

解释:

前 2 位顾客那里，我们按顺序收取 2 张 5 美元的钞票。

对于接下来的 2 位顾客，我们收取一张 10 美元的钞票，然后返还 5 美元。

对于最后一位顾客，我们无法退回 15 美元，因为我们现在只有两张 10 美元的钞票。

由于不是每位顾客都得到了正确的找零，所以答案是 `false`。

提示:

- `1 <= bills.length <= 10(5)`
- `bills[i]` 不是 5 就是 10 或是 20

3. 解法（贪心）：

贪心策略：

分情况讨论：

- a. 遇到 5 元钱，直接收下；
- b. 遇到 10 元钱，找零 5 元钱之后，收下；
- c. 遇到 20 元钱：
 - i. 先尝试凑 10 + 5 的组合；
 - ii. 如果凑不出来，拼凑 5 + 5 + 5 的组合；

C++ 算法代码：

```
1 class Solution
2 {
3 public:
4     bool lemonadeChange(vector<int>& bills)
5     {
6         int five = 0, ten = 0;
7         for(auto x : bills)
8         {
9             if(x == 5) five++; // 5 元: 直接收下
10            else if(x == 10) // 10 元: 找零 5 元
11            {
12                if(five == 0) return false;
13                five--; ten++;
14            }
15            else // 20 元: 分情况讨论
16            {
17                if(ten && five) // 贪心
18                {
19                    ten--; five--;
20                }
21                else if(five >= 3)
22                {
23                    five -= 3;
24                }
25                else return false;
26            }
27        }
28        return true;
29    }
30 };
```

C++ 代码结果:

C++



Java 算法代码:

```
1 class Solution
2 {
3     public boolean lemonadeChange(int[] bills)
4     {
5         int five = 0, ten = 0;
6         for(int x : bills)
7         {
8             if(x == 5) // 5 元: 直接收下
9             {
10                 five++;
11             }
12             else if(x == 10) // 10 元: 找零 5 元
13             {
14                 if(five == 0) return false;
15                 five--; ten++;
16             }
17             else // 20 元: 分情况讨论
18             {
19                 if(five != 0 && ten != 0) // 贪心
20                 {
21                     five--; ten--;
22                 }
23                 else if(five >= 3)
24                 {
25                     five -= 3;
26                 }
27                 else return false;
28             }
29         }
30         return true;
31     }
32 }
```

Java 运行结果：

Java



2. 将数组和减半的最少操作次数 (medium)

1. 题目链接：2208. 将数组和减半的最少操作次数

2. 题目描述：

给你一个正整数数组 `nums` 。每一次操作中，你可以从 `nums` 中选择 **任意** 一个数并将它减小到 **恰好** 一半。（注意，在后续操作中你可以对减半过的数继续执行操作）

请你返回将 `nums` 数组和 **至少** 减少一半的 **最少** 操作数。

示例 1：

输入： `nums = [5,19,8,1]`

输出： 3

解释： 初始 `nums` 的和为 $5 + 19 + 8 + 1 = 33$ 。

以下是将数组和减少至少一半的一种方法：

选择数字 19 并减小为 9.5 。

选择数字 9.5 并减小为 4.75 。

选择数字 8 并减小为 4 。

最终数组为 `[5, 4.75, 4, 1]` ，和为 $5 + 4.75 + 4 + 1 = 14.75$ 。

`nums` 的和减小了 $33 - 14.75 = 18.25$ ，减小的部分超过了初始数组和的一半， $18.25 \geq 33/2 = 16.5$ 。

我们需要 3 个操作实现题目要求，所以返回 3 。

可以证明，无法通过少于 3 个操作使数组和减少至少一半。

示例 2：

输入： `nums = [3,8,20]`

输出： 3

解释： 初始 `nums` 的和为 $3 + 8 + 20 = 31$ 。

以下是将数组和减少至少一半的一种方法：

选择数字 20 并减小为 10 。

选择数字 10 并减小为 5 。

选择数字 3 并减小为 1.5 。

最终数组为 `[1.5, 8, 5]` ，和为 $1.5 + 8 + 5 = 14.5$ 。

nums 的和减小了 $31 - 14.5 = 16.5$ ，减小的部分超过了初始数组和的一半， $16.5 \geq 31/2 = 16.5$

。

我们需要 3 个操作实现题目要求，所以返回 3。

可以证明，无法通过少于 3 个操作使数组和减少至少一半。

提示：

- `1 <= nums.length <= 10^5`
- `1 <= nums[i] <= 10^7`

3. 解法（贪心）：

贪心策略：

- 每次挑选出「当前」数组中「最大」的数，然后「减半」；
- 直到数组和减少到至少一半为止。

为了「快速」挑选出数组中最大的数，我们可以利用「堆」这个数据结构。

C++ 算法代码：

```
1 class Solution
2 {
3 public:
4     int halveArray(vector<int>& nums)
5     {
6         priority_queue<double> heap; // 创建一个大根堆
7         double sum = 0.0;
8         for(int x : nums) // 把元素都丢进堆中，并求出累加和
9         {
10             heap.push(x);
11             sum += x;
12         }
13         sum /= 2.0; // 先算出目标和
14
15         int count = 0;
16         while(sum > 0) // 依次取出堆顶元素减半，直到减到之前的一半以下
17         {
18             double t = heap.top() / 2.0;
19             heap.pop();
20             sum -= t;
21             count++;
22             heap.push(t);
23         }
```

```

23     }
24     return count;
25 }
26 };

```

C++ 代码结果：

C++



Java 算法代码：

```

1 class Solution
2 {
3     public int halveArray(int[] nums)
4     {
5         // 创建一个大根堆
6         PriorityQueue<Double> heap = new PriorityQueue<>((a, b) ->
7             b.compareTo(a));
8         double sum = 0.0;
9         for(int x : nums) // 把元素都丢进堆中，并求出累加和
10        {
11            heap.offer((double)x);
12            sum += x;
13        }
14        sum /= 2.0; // 先算出目标和
15
16        int count = 0;
17        while(sum > 0) // 依次取出堆顶元素减半，直到减到之前的一半以下
18        {
19            double t = heap.poll() / 2.0;
20            sum -= t;
21            count++;
22            heap.offer(t);
23        }
24        return count;
25 }

```

Java 运行结果：

时间 166 ms

击败 68.41%

内存 55.5 MB

击败 82.28%

3. 最大数 (medium)

1. 题目链接：179. 最大数

2. 题目描述

给定一组非负整数 `nums`，重新排列每个数的顺序（每个数不可拆分）使之组成一个最大的整数。

注意：输出结果可能非常大，所以你需要返回一个字符串而不是整数。

示例 1：

输入：nums = [10,2] 输出："210"

示例 2：

输入：nums = [3,30,34,5,9] 输出："9534330"

提示：

- 1 ≤ nums.length ≤ 100
- 0 ≤ nums[i] ≤ 10⁹

3. 解法（贪心）：

可以先优化：

将所有数字当成字符串处理，那么两个数字之间的拼接操作以及比较操作就会很方便。

贪心策略：

按照题目的要求，重新定义一个新的排序规则，然后排序即可。

排序规则：

- 「A 拼接 B」 大于 「B 拼接 A」，那么 A 在前，B 在后；
- 「A 拼接 B」 等于 「B 拼接 A」，那么 A B 的顺序无所谓；
- 「A 拼接 B」 小于 「B 拼接 A」，那么 B 在前，A 在后；

C++ 算法代码：


```

1 class Solution
2 {
3 public:
4     string largestNumber(vector<int>& nums)
5     {
6         // 优化: 把所有的数转化成字符串
7         vector<string> strs;
8         for(int x : nums) strs.push_back(to_string(x));
9
10        // 排序
11        sort(strs.begin(), strs.end(), [](const string& s1, const string& s2)
12        {
13            return s1 + s2 > s2 + s1;
14        });
15
16        // 提取结果
17        string ret;
18        for(auto& s : strs) ret += s;
19
20        if(ret[0] == '0') return "0";
21        return ret;
22    }
23 };

```

C++ 代码结果:

C++



Java 算法代码:

```

1 class Solution
2 {
3     public String largestNumber(int[] nums)
4     {
5         // 优化: 把所有的数转化成字符串
6         int n = nums.length;
7         String[] strs = new String[n];
8         for(int i = 0; i < n; i++) strs[i] = "" + nums[i];
9
10        // 排序

```

```

11     Arrays.sort(strs, (a, b) ->
12     {
13         return (b + a).compareTo(a + b);
14     });
15
16     // 提取结果
17     StringBuffer ret = new StringBuffer();
18     for(String s : strs) ret.append(s);
19
20     if(ret.charAt(0) == '0') return "0";
21     return ret.toString();
22 }
23 }

```

Java 运行结果：



4. 摆动序列 (medium)

1. 题目链接：376. 摆动序列

2. 题目描述

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**。

示例 1：

输入：`nums = [1,7,4,9,2,5]`

输出：6

解释：整个序列均为摆动序列，各元素之间的差值为 (6, -3, 5, -7, 3)。

示例 2:

输入: nums = [1,17,5,10,13,15,10,5,16,8]

输出: 7

解释：这个序列包含几个长度为 7 摆动序列。

其中一个为 [1, 17, 10, 13, 10, 16, 8]，各元素之间的差值为 (16, -7, 3, -3, 6, -8)。

示例 3:

输入: nums = [1,2,3,4,5,6,7,8,9]

输出: 2

提示:

- `1 <= nums.length <= 1000`
- `0 <= nums[i] <= 1000`

3. 解法（贪心）：

贪心策略：

对于某一个位置来说：

- 如果接下来呈现上升趋势的话，我们让其上升到波峰的位置；
- 如果接下来呈现下降趋势的话，我们让其下降到波谷的位置。

因此，如果把整个数组放在「折线图」中，我们统计出所有的波峰以及波谷的个数即可。

C++ 算法代码：

```
1 class Solution
2 {
3 public:
4     int wiggleMaxLength(vector<int>& nums)
5     {
6         int n = nums.size();
7         if(n < 2) return n;
8
9         int ret = 0, left = 0;
10        for(int i = 0; i < n - 1; i++)
11        {
12            int right = nums[i + 1] - nums[i]; // 计算接下来的趋势
13            if(right == 0) continue; // 如果水平，直接跳过
14            if(right * left <= 0) ret++; // 累加波峰或者波谷
15            left = right;
```

```

16     }
17     return ret + 1;
18 }
19 };

```

C++ 代码结果：

C++



Java 算法代码：

```

1 class Solution
2 {
3     public int wiggleMaxLength(int[] nums)
4     {
5         int n = nums.length;
6         if(n < 2) return n;
7
8         int ret = 0, left = 0;
9         for(int i = 0; i < n - 1; i++)
10        {
11            int right = nums[i + 1] - nums[i]; // 计算接下来的趋势
12            if(right == 0) continue; // 如果水平，直接跳过
13            if(left * right <= 0) ret++; // 累加波峰或者波谷
14            left = right;
15        }
16        return ret + 1;
17    }
18 }

```

Java 运行结果：

Java



5. 最长递增子序列 (medium)

1. 题目链接: 300. 最长递增子序列

2. 题目描述:

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。

示例 1:

输入: `nums = [10,9,2,5,3,7,101,18]`

输出: 4

解释: 最长递增子序列是 `[2,3,7,101]`，因此长度为 4。

示例 2:

输入: `nums = [0,1,0,3,2,3]`

输出: 4

示例 3:

输入: `nums = [7,7,7,7,7,7,7]`

输出: 1

提示:

- `1 <= nums.length <= 2500`
- `-10(4) <= nums[i] <= 10(4)`

3. 解法 (贪心) :

贪心策略:

我们在考虑最长递增子序列的长度的时候，其实并不关心这个序列长什么样子，我们只是关心最后一个元素是谁。这样新来一个元素之后，我们就可以判断是否可以拼接到它的后面。

因此，我们可以创建一个数组，统计长度为 `x` 的递增子序列中，最后一个元素是谁。为了尽可能的让这个序列更长，我们仅需统计长度为 `x` 的所有递增序列中最后一个元素的「最小值」。

统计的过程中发现，数组中的数呈现「递增」趋势，因此可以使用「二分」来查找插入位置。

C++ 算法代码:

```
1 class Solution
2 {
```

```

3 public:
4     int lengthOfLIS(vector<int>& nums)
5     {
6         int n = nums.size();
7         vector<int> ret;
8         ret.push_back(nums[0]);
9
10        for(int i = 1; i < n; i++)
11        {
12            if(nums[i] > ret.back()) // 如果能接在最后一个元素后面, 直接放
13            {
14                ret.push_back(nums[i]);
15            }
16            else
17            {
18                // 二分插入位置
19                int left = 0, right = ret.size() - 1;
20                while(left < right)
21                {
22                    int mid = (left + right) >> 1;
23                    if(ret[mid] < nums[i]) left = mid + 1;
24                    else right = mid;
25                }
26                ret[left] = nums[i]; // 放在 left 位置上
27            }
28        }
29        return ret.size();
30    }
31 };

```

C++ 代码结果:

C++



Java 算法代码:

```

1 class Solution
2 {
3     public int lengthOfLIS(int[] nums)
4     {

```

```

5      ArrayList<Integer> ret = new ArrayList<>();
6      int n = nums.length;
7      ret.add(nums[0]);
8
9      for(int i = 1; i < n; i++)
10     {
11         if(nums[i] > ret.get(ret.size() - 1)) // 如果能接在最后一个元素后面，
直接放
12         {
13             ret.add(nums[i]);
14         }
15         else
16         {
17             // 二分插入位置
18             int left = 0, right = ret.size() - 1;
19             while(left < right)
20             {
21                 int mid = (left + right) / 2;
22                 if(ret.get(mid) < nums[i]) left = mid + 1;
23                 else right = mid;
24             }
25             ret.set(left, nums[i]); // 放在 left 位置上
26         }
27     }
28     return ret.size();
29 }
30 }

```

Java 运行结果：

Java



6. 递增的三元子序列 (medium)

1. 题目链接：334. 递增的三元子序列

2. 题目描述

给你一个整数数组 `nums`，判断这个数组中是否存在长度为 3 的递增子序列。

如果存在这样的三元组下标 (i, j, k) 且满足 $i < j < k$ ，使得 $nums[i] < nums[j] < nums[k]$ ，返回 `true`；否则，返回 `false`。

示例 1:

输入: `nums = [1,2,3,4,5]`

输出: `true`

解释: 任何 $i < j < k$ 的三元组都满足题意

示例 2:

输入: `nums = [5,4,3,2,1]`

输出: `false`

解释: 不存在满足题意的三元组

示例 3:

输入: `nums = [2,1,5,0,4,6]`

输出: `true`

解释: 三元组 (3, 4, 5) 满足题意, 因为 `nums[3] == 0 < nums[4] == 4 < nums[5] == 6`

提示:

- `1 <= nums.length <= 5 * 10(5)`
- `-2(31) <= nums[i] <= 2(31) - 1`

3. 解法 (贪心) :

贪心策略:

最长递增子序列的简化版。

不用一个数组存数据, 仅需两个变量即可。也不用二分插入位置, 仅需两次比较就可以找到插入位置。

C++ 算法代码:

```
1 class Solution
2 {
3 public:
4     bool increasingTriplet(vector<int>& nums)
5     {
6         int a = nums[0], b = INT_MAX;
7         for(int i = 1; i < nums.size(); i++)
8         {
9             if(nums[i] > b) return true;
10            else if(nums[i] > a) b = nums[i];
11            else a = nums[i];
12        }
13    }
14 }
```



```
12     }
13     return false;
14 }
15 };
```

C++ 代码结果:

C++



Java 算法代码:

```
1 class Solution
2 {
3     public boolean increasingTriplet(int[] nums)
4     {
5         int a = nums[0], b = Integer.MAX_VALUE;
6         for(int i = 1; i < nums.length; i++)
7         {
8             if(nums[i] > b) return true;
9             else if(nums[i] > a) b = nums[i];
10            else a = nums[i];
11        }
12        return false;
13    }
14 }
```

Java 运行结果:

Java



7. 最长连续递增序列 (easy)

1. 题目链接: [674. 最长连续递增序列](#)

2. 题目描述：

给定一个未经排序的整数数组，找到最长且 **连续递增的子序列**，并返回该序列的长度。

连续递增的子序列 可以由两个下标 l 和 r ($l < r$) 确定，如果对于每个 $l \leq i < r$ ，都有 $nums[i] < nums[i + 1]$ ，那么子序列 $[nums[l], nums[l + 1], \dots, nums[r - 1], nums[r]]$ 就是连续递增子序列。

示例 1：

输入：nums = [1,3,5,4,7]

输出：3

解释：最长连续递增序列是 [1,3,5]，长度为3。

尽管 [1,3,5,7] 也是升序的子序列，但它不是连续的，因为 5 和 7 在原数组里被 4 隔开。

示例 2：

输入：nums = [2,2,2,2,2]

输出：1

解释：最长连续递增序列是 [2]，长度为1。

提示：

- $1 \leq \text{nums.length} \leq 10(4)$
- $-10(9) \leq \text{nums}[i] \leq 10(9)$

3. 解法（贪心）：

贪心策略：

找到以某个位置为起点的最长连续递增序列之后（设这个序列的末尾为 j 位置），接下来直接以 $j + 1$ 的位置为起点寻找下一个最长连续递增序列。

C++ 算法代码：

```
1 class Solution
2 {
3 public:
4     int findLengthOfLCIS(vector<int>& nums)
5     {
6         int ret = 0, n = nums.size();
7         for(int i = 0; i < n; )
8         {
9             int j = i + 1;
10            // 找到递增区间的末端
```

```

11         while(j < n && nums[j] > nums[j - 1]) j++;
12         ret = max(ret, j - i);
13         i = j; // 直接在循环中更新下一个位置的起点
14     }
15     return ret;
16 }
17 };

```

C++ 代码结果:

C++



Java 算法代码:

```

1 class Solution
2 {
3     public int findLengthOfLCIS(int[] nums)
4     {
5         int ret = 0, n = nums.length;
6         for(int i = 0; i < n; )
7         {
8             int j = i + 1;
9             // 找到递增区间的末端
10            while(j < n && nums[j] > nums[j - 1]) j++;
11            ret = Math.max(ret, j - i);
12            i = j; // 循环内部直接更新下一个位置的起点 - 贪心
13        }
14        return ret;
15    }
16 }

```

Java 运行结果:

Java



8. 买卖股票的最佳时机 (easy)

1. 题目链接: [121. 买卖股票的最佳时机](#)

2. 题目描述:

给定一个数组 `prices`，它的第 `i` 个元素 `prices[i]` 表示一支给定股票第 `i` 天的价格。

你只能选择 **某一天** 买入这只股票，并选择在 **未来的某一个不同的日子** 卖出该股票。设计一个算法来计算你能获取的最大利润。

返回你可以从这笔交易中获取的最大利润。如果你不能获取任何利润，返回 `0`。

示例 1:

输入: `[7,1,5,3,6,4]`

输出: `5`

解释: 在第 2 天 (股票价格 = 1) 的时候买入，在第 5 天 (股票价格 = 6) 的时候卖出，最大利润 = $6 - 1 = 5$ 。

注意利润不能是 $7 - 1 = 6$ ，因为卖出价格需要大于买入价格；同时，你不能在买入前卖出股票。

示例 2:

输入: `prices = [7,6,4,3,1]`

输出: `0`

解释: 在这种情况下，没有交易完成，所以最大利润为 0。

提示:

- `1 <= prices.length <= 10(5)`
- `0 <= prices[i] <= 10(4)`

3. 解法 (贪心) :

贪心策略:

由于只能交易一次，所以对于某一个位置 `i`，要想获得最大利润，仅需知道前面所有元素的最小值。然后在最小值的位置「买入」股票，在当前位置「卖出」股票即可。

C++ 算法代码:

```
1 class Solution
2 {
3 public:
```

```

4     int maxProfit(vector<int>& prices)
5     {
6         int ret = 0; // 记录最终结果
7         for(int i = 0, prevMin = INT_MAX; i < prices.size(); i++)
8         {
9             ret = max(ret, prices[i] - prevMin); // 先更新结果
10            prevMin = min(prevMin, prices[i]); // 再更新最小值
11        }
12        return ret;
13    }
14 };

```

C++ 代码结果：

C++



Java 算法代码：

```

1 class Solution
2 {
3     public int maxProfit(int[] prices)
4     {
5         int ret = 0; // 记录最终结果
6         for(int i = 0, prevMin = Integer.MAX_VALUE; i < prices.length; i++)
7         {
8             ret = Math.max(ret, prices[i] - prevMin); // 先更新结果
9             prevMin = Math.min(prevMin, prices[i]); // 再更新最小值
10        }
11        return ret;
12    }
13 }

```

Java 运行结果：

Java



9. 买卖股票的最佳时机 II (medium)

1. 题目链接: [122. 买卖股票的最佳时机 II](#)

2. 题目描述:

给你一个整数数组 `prices`，其中 `prices[i]` 表示某支股票第 `i` 天的价格。

在每一天，你可以决定是否购买和/或出售股票。你在任何时候 **最多** 只能持有一股股票。你也可以先购买，然后在 **同一天** 出售。

返回 **你能获得的最大利润**。

示例 1:

输入: `prices = [7,1,5,3,6,4]`

输出: 7

解释: 在第 2 天 (股票价格 = 1) 的时候买入, 在第 3 天 (股票价格 = 5) 的时候卖出, 这笔交易所能获得利润 = $5 - 1 = 4$ 。

随后, 在第 4 天 (股票价格 = 3) 的时候买入, 在第 5 天 (股票价格 = 6) 的时候卖出, 这笔交易所能获得利润 = $6 - 3 = 3$ 。

总利润为 $4 + 3 = 7$ 。

示例 2:

输入: `prices = [1,2,3,4,5]`

输出: 4

解释: 在第 1 天 (股票价格 = 1) 的时候买入, 在第 5 天 (股票价格 = 5) 的时候卖出, 这笔交易所能获得利润 = $5 - 1 = 4$ 。

总利润为 4。

示例 3:

输入: `prices = [7,6,4,3,1]`

输出: 0

解释: 在这种情况下, 交易无法获得正利润, 所以不参与交易可以获得最大利润, 最大利润为 0。

提示:

- `1 <= prices.length <= 3 * 104`
- `0 <= prices[i] <= 104`

3. 解法 (贪心) :

贪心策略：

由于可以进行无限次交易，所以只要是一个「上升区域」，我们就把利润拿到手就好了。

C++ 算法代码：

```
1 class Solution
2 {
3 public:
4     int maxProfit(vector<int>& p)
5     {
6         // 实现方式一：双指针
7         int ret = 0, n = p.size();
8         for(int i = 0; i < n; i++)
9         {
10             int j = i;
11             while(j + 1 < n && p[j + 1] > p[j]) j++; // 找上升的末端
12             ret += p[j] - p[i];
13             i = j;
14         }
15         return ret;
16     }
17 };
18
19 class Solution
20 {
21 public:
22     int maxProfit(vector<int>& prices)
23     {
24         // 实现方式二：拆分成一天一天
25         int ret = 0;
26         for(int i = 1; i < prices.size(); i++)
27         {
28             if(prices[i] > prices[i - 1])
29                 ret += prices[i] - prices[i - 1];
30         }
31         return ret;
32     }
33 };
```

C++ 代码结果：

C++

时间 8 ms

击败 56.61%

内存 12.7 MB

击败 66.98%

Java 算法代码：

```
1 class Solution
2 {
3     public int maxProfit(int[] prices)
4     {
5         // 实现方式一：双指针
6         int ret = 0, n = prices.length;
7         for(int i = 0; i < n; i++)
8         {
9             int j = i;
10            while(j + 1 < n && prices[j] < prices[j + 1]) j++; // 向后寻找上升的
            末端
11            ret += prices[j] - prices[i];
12            i = j;
13        }
14        return ret;
15    }
16 }
17
18 class Solution
19 {
20     public int maxProfit(int[] prices)
21     {
22         // 实现方式二：拆分成一天一天的形式
23         int ret = 0;
24         for(int i = 1; i < prices.length; i++)
25         {
26             if(prices[i] > prices[i - 1])
27             {
28                 ret += prices[i] - prices[i - 1];
29             }
30         }
31         return ret;
32     }
33 }
```


Java 运行结果：

Java

时间 1 ms

击败 69.84%

内存 42.9 MB

击败 93.20%

10. K 次取反后最大化的数组和（easy）

1. 题目链接：1005. K 次取反后最大化的数组和

2. 题目描述：

给你一个整数数组 `nums` 和一个整数 `k`，按以下方法修改该数组：

- 选择某个下标 `i` 并将 `nums[i]` 替换为 `-nums[i]`。

重复这个过程恰好 `k` 次。可以多次选择同一个下标 `i`。

以这种方式修改数组后，返回数组 **可能的最大和**。

示例 1：

输入：nums = [4,2,3], k = 1

输出：5

解释：选择下标 1，nums 变为 [4,-2,3]。

示例 2：

输入：nums = [3,-1,0,2], k = 3

输出：6

解释：选择下标 (1, 2, 2)，nums 变为 [3,1,0,2]。

示例 3：

输入：nums = [2,-3,-1,5,-4], k = 2

输出：13

解释：选择下标 (1, 4)，nums 变为 [2,3,-1,5,4]。

提示：

- `1 <= nums.length <= 10(4)`
- `-100 <= nums[i] <= 100`
- `1 <= k <= 10(4)`

3. 解法（贪心）：

贪心策略：

分情况讨论，设整个数组中负数的个数为 m 个：

- a. $m > k$ ：把前 k 小负数，全部变成正数；
- b. $m == k$ ：把所有的负数全部转化成正数；
- c. $m < k$ ：
 - i. 先把所有的负数变成正数；
 - ii. 然后根据 $k - m$ 的奇偶分情况讨论：
 1. 如果是偶数，直接忽略；
 2. 如果是奇数，挑选当前数组中最小的数，变成负数

C++ 算法代码：

```
1 class Solution
2 {
3 public:
4     int largestSumAfterKNegations(vector<int>& nums, int k)
5     {
6         int m = 0, minElem = INT_MAX, n = nums.size();
7         for(auto x : nums)
8         {
9             if(x < 0) m++;
10            minElem = min(minElem, abs(x)); // 求绝对值最小的那个数
11        }
12        // 分类讨论
13        int ret = 0;
14        if(m > k)
15        {
16            sort(nums.begin(), nums.end());
17            for(int i = 0; i < k; i++) // 前 k 小个负数，变成正数
18            {
19                ret += -nums[i];
20            }
21            for(int i = k; i < n; i++) // 后面的数不变
22            {
23                ret += nums[i];
24            }
25        }
26        else
27        {
28            // 把所有的负数变成正数
29            for(auto x : nums) ret += abs(x);
```

```

30         if((k - m) % 2) // 判断是否处理最小的正数
31         {
32             ret -= minElem * 2;
33         }
34     }
35     return ret;
36 }
37 };

```

C++ 代码结果:

C++



Java 算法代码:

```

1 class Solution
2 {
3     public int largestSumAfterKNegations(int[] nums, int k)
4     {
5         int m = 0, minElem = Integer.MAX_VALUE, n = nums.length;
6         for(int x : nums)
7         {
8             if(x < 0) m++;
9             minElem = Math.min(minElem, Math.abs(x));
10        }
11        // 分类讨论
12        int ret = 0;
13        if(m > k)
14        {
15            Arrays.sort(nums);
16            for(int i = 0; i < k; i++) // 前 k 小个负数, 变成正数
17            {
18                ret += -nums[i];
19            }
20            for(int i = k; i < n; i++) // 后面的数不变
21            {
22                ret += nums[i];
23            }
24        }
25        else

```

```

26      {
27          // 把负数全部变成正数
28          for(int x : nums) ret += Math.abs(x);
29          if((k - m) % 2 != 0) // 判断是否处理最小的正数
30          {
31              ret -= minElem * 2;
32          }
33      }
34      return ret;
35  }
36 }

```

Java 运行结果：

Java



11. 按身高排序 (easy)

1. 题目链接：2418. 按身高排序

2. 题目描述：

给你一个字符串数组 `names`，和一个由 **互不相同** 的正整数组成的数组 `heights`。两个数组的长度均为 `n`。

对于每个下标 `i`，`names[i]` 和 `heights[i]` 表示第 `i` 个人的名字和身高。

请按身高 **降序** 顺序返回对应的名字数组 `names`。

示例 1：

输入： `names = ["Mary","John","Emma"], heights = [180,165,170]`

输出： `["Mary","Emma","John"]`

解释： Mary 最高，接着是 Emma 和 John。

示例 2：

输入： `names = ["Alice","Bob","Bob"], heights = [155,185,150]`

输出： `["Bob","Alice","Bob"]`

解释： 第一个 Bob 最高，然后是 Alice 和第二个 Bob。

提示：

- `n == names.length == heights.length`
- `1 <= n <= 10(3)`
- `1 <= names[i].length <= 20`
- `1 <= heights[i] <= 10(5)`
- `names[i]` 由大小写英文字母组成
- `heights` 中的所有值互不相同

3. 解法（通过排序 "索引" 的方式）：

算法思路：

我们不能直接按照 `i` 位置对应的 `heights` 来排序，因为排序过程是会移动元素的，但是 `names` 内的元素是不会移动的。

由题意可知，`names` 数组和 `heights` 数组的下标是一一对应的，因此我们可以重新创建出来一个下标数组，将这个下标数组按照 `heights[i]` 的大小排序。

那么，当下标数组排完序之后，里面的顺序就相当于 `heights` 这个数组排完序之后的下标。之后通过排序后的下标，依次找到原来的 `name`，完成对名字的排序。

C++ 算法代码：

```
1 class Solution
2 {
3 public:
4     vector<string> sortPeople(vector<string>& names, vector<int>& heights)
5     {
6         // 1. 创建一个下标数组
7         int n = names.size();
8         vector<int> index(n);
9         for(int i = 0; i < n; i++) index[i] = i;
10
11         // 2. 对下标进行排序
12         sort(index.begin(), index.end(), [&](int i, int j)
13         {
14             return heights[i] > heights[j];
15         });
16
17         // 3. 提取结果
18         vector<string> ret;
19         for(int i : index)
20         {
```

```

21         ret.push_back(names[i]);
22     }
23     return ret;
24 }
25 };

```

C++ 运行结果：



Java 算法代码：

```

1 class Solution
2 {
3     public String[] sortPeople(String[] names, int[] heights)
4     {
5         // 1. 创建一个下标数组
6         int n = names.length;
7         Integer[] index = new Integer[n];
8         for(int i = 0; i < n; i++) index[i] = i;
9
10        // 2. 对下标数组排序
11        Arrays.sort(index, (i, j) ->
12        {
13            return heights[j] - heights[i];
14        });
15
16        // 3. 提取结果
17        String[] ret = new String[n];
18        for(int i = 0; i < n; i++)
19        {
20            ret[i] = names[index[i]];
21        }
22        return ret;
23    }
24 }

```

Java 运行结果：



12. 优势洗牌（田忌赛马）（medium）

1. 题目链接：870. 优势洗牌

注意注意注意！！

做这道题之前建议先把 [2418. 按身高排序](#) 里面，通过排序数组下标，进而不改变原数组顺序的排序技巧好好吸收消化掉。不然里面变量众多，很容易犯迷。

2. 题目描述：

给定两个长度相等的数组 `nums1` 和 `nums2`，`nums1` 相对于 `nums2` 的优势可以用满足 `nums1[i] > nums2[i]` 的索引 `i` 的数目来描述。

返回 `nums1` 的任意排列，使其相对于 `nums2` 的优势最大化。

示例 1：

输入：`nums1 = [2,7,11,15]`, `nums2 = [1,10,4,11]`

输出：[2,11,7,15]

示例 2：

输入：`nums1 = [12,24,8,32]`, `nums2 = [13,25,32,11]`

输出：[24,32,8,12]

提示：

- `1 <= nums1.length <= 10(5)`
- `nums2.length == nums1.length`
- `0 <= nums1[i], nums2[i] <= 10(9)`

3. 解法（贪心）：

讲一下田忌赛马背后包含的博弈论和贪心策略：

田忌赛马没听过的自行百度，这里讲一下田忌赛马背后的博弈决策，从三匹马拓展到 `n` 匹马之间博弈的最优策略。

田忌：下等马 中等马 上等马

齐王：下等马 中等马 上等马

- a. 田忌的下等马 `pk` 不过齐王的下等马，因此把这匹马丢去消耗一个齐王的最强战马！
- b. 接下来选择中等马 `pk` 齐王的下等马，勉强获胜；
- c. 最后用上等马 `pk` 齐王的中等马，勉强获胜。

由此，我们可以得出一个最优的决策方式：

- a. 当己方此时最差的比不过对面最差的时候，让我方最差的去处理掉对面最好的（反正要输，不如去拖掉对面一个最强的）；
- b. 当己方此时
- c. 最差的能比得上对面最差的时候，就让两者比下去（最差的都能获胜，为什么要输呢）。

每次决策，都会使我方处于优势。

C++ 算法代码：

```
1 class Solution
2 {
3 public:
4     vector<int> advantageCount(vector<int>& nums1, vector<int>& nums2)
5     {
6         int n = nums1.size();
7         // 1. 排序
8         sort(nums1.begin(), nums1.end());
9         vector<int> index2(n);
10        for(int i = 0; i < n; i++) index2[i] = i;
11        sort(index2.begin(), index2.end(), [&](int i, int j)
12        {
13            return nums2[i] < nums2[j];
14        });
15
16        // 2. 田忌赛马
17        vector<int> ret(n);
18        int left = 0, right = n - 1;
19        for(auto x : nums1)
20        {
21            if(x > nums2[index2[right]]) ret[index2[right--]] = x;
22            else ret[index2[left++]] = x;
23        }
24        return ret;
25    }
26 };
```


C++ 运行结果:

C++



Java 算法代码:

```
1 class Solution
2 {
3     public int[] advantageCount(int[] nums1, int[] nums2)
4     {
5         int n = nums1.length;
6         // 1. 排序
7         Arrays.sort(nums1);
8         Integer[] index2 = new Integer[n];
9         for(int i = 0; i < n; i++) index2[i] = i;
10        Arrays.sort(index2, (i, j) ->
11        {
12            return nums2[i] - nums2[j];
13        });
14
15        // 2. 田忌赛马
16        int[] ret = new int[n];
17        int left = 0, right = n - 1;
18        for(int x : nums1)
19        {
20            if(x > nums2[index2[left]])
21            {
22                ret[index2[left++]] = x;
23            }
24            else
25            {
26                ret[index2[right--]] = x;
27            }
28        }
29        return ret;
30    }
31 }
```

Java 运行结果:



13. 最长回文串 (easy)

1. 题目链接: [409. 最长回文串](#)

2. 题目描述:

给定一个包含大写字母和小写字母的字符串 `s`，返回 通过这些字母构造的 **最长的回文串**。
在构造过程中，请注意 **区分大小写**。比如 `"Aa"` 不能当做一个回文字符串。

示例 1:

输入:`s = "abcccccdd"`

输出:7

解释:

我们可以构造的最长的回文串是"dccaccd", 它的长度是 7。

示例 2:

输入:`s = "a"`

输出:1

示例 3:

输入:`s = "aaaaacccc"`

输出:7

提示:

- `1 <= s.length <= 2000`
- `s` 只由小写 和/或 大写英文字母组成

3. 解法 (贪心) :

贪心策略:

用尽可能多的字符去构造回文串:

- 如果字符出现偶数个，那么全部都可以用来构造回文串；
- 如果字符出现奇数个，减去一个之后，剩下的字符能够全部用来构造回文串；
- 最后再判断一下，如果有字符出现奇数个，就把它单独拿出来放在中间。

C++ 算法代码：

```
1 class Solution
2 {
3 public:
4     int longestPalindrome(string s)
5     {
6         // 1. 计数 - 用数组模拟哈希表
7         int hash[127] = { 0 };
8         for(char ch : s) hash[ch]++;
9
10        // 2. 统计结果
11        int ret = 0;
12        for(int x : hash)
13        {
14            ret += x / 2 * 2;
15        }
16        return ret < s.size() ? ret + 1 : ret;
17    }
18 };
```

C++ 代码结果：

C++



Java 算法代码：

```
1 class Solution
2 {
3     public int longestPalindrome(String s)
4     {
5         // 1. 计数 - 用数组模拟哈希表
6         int[] hash = new int[127];
7         for(int i = 0; i < s.length(); i++)
8         {
9             hash[s.charAt(i)]++;
10        }
11    }
```

```

11
12     // 2. 统计结果
13     int ret = 0;
14     for(int x : hash)
15     {
16         ret += x / 2 * 2;
17     }
18     return ret < s.length() ? ret + 1 : ret;
19 }
20 }

```

Java 运行结果:

Java



14. 增减字符串匹配 (easy)

1. 题目链接: 942. 增减字符串匹配

2. 题目描述:

由范围 $[0, n]$ 内所有整数组成的 $n + 1$ 个整数的排列序列可以表示为长度为 n 的字符串 s ，其中:

- 如果 $\text{perm}[i] < \text{perm}[i + 1]$ ，那么 $s[i] == 'I'$
- 如果 $\text{perm}[i] > \text{perm}[i + 1]$ ，那么 $s[i] == 'D'$

给定一个字符串 s ，重构排列 perm 并返回它。如果有多个有效排列 perm ，则返回其中 **任何一个**。

示例 1:

输入: $s = "IDID"$

输出: $[0,4,1,3,2]$

示例 2:

输入: $s = "III"$

输出: $[0,1,2,3]$

示例 3:

输入: `s = "DDI"`

输出: `[3,2,0,1]`

提示:

- `1 <= s.length <= 10(5)`
- `s` 只包含字符 `"I"` 或 `"D"`

3. 解法 (贪心) :

贪心策略:

- 当遇到 `'I'` 的时候, 为了让下一个上升的数可选择的「范围更多」, 当前选择「最小」的那个数;
- 当遇到 `'D'` 的时候, 为了让下一个下降的数可选择的「范围更多」, 选择当前「最大」的那个数。

C++ 算法代码:

```
1 class Solution
2 {
3 public:
4     vector<int> diStringMatch(string s)
5     {
6         int left = 0, right = s.size(); // 用 left, right 标记最小值和最大值
7         vector<int> ret;
8         for(auto ch : s)
9         {
10             if(ch == 'I') ret.push_back(left++);
11             else ret.push_back(right--);
12         }
13         ret.push_back(left); // 把最后一个数放进去
14         return ret;
15     }
16 };
```

C++ 代码结果:

C++

时间 8 ms

击败 41.69%

内存 9.2 MB

击败 5.25%

Java 算法代码：

```
1 class Solution
2 {
3     public int[] diStringMatch(String s)
4     {
5         int n = s.length();
6         int left = 0, right = n; // 用 left, right 标记最小值和最大值
7         int[] ret = new int[n + 1];
8         for(int i = 0; i < n; i++)
9         {
10             if(s.charAt(i) == 'I')
11             {
12                 ret[i] = left++;
13             }
14             else
15             {
16                 ret[i] = right--;
17             }
18         }
19         ret[n] = left; // 把最后一个数放进去
20         return ret;
21     }
22 }
```

Java 运行结果：



15. 分发饼干 (easy)

1. 题目链接：455. 分发饼干

2. 题目描述：

假设你是一位很棒的家长，想要给你的孩子们一些小饼干。但是，每个孩子最多只能给一块饼干。

对每个孩子 i ，都有一个胃口值 $g[i]$ (i 的索引)，这是能让孩子们满足胃口的饼干的最小尺寸；并且每块饼干 j ，都有一个尺寸 $s[j]$ (j 的索引)。如果 $s[j] \geq g[i]$ ，我们可以将这个饼干 j 分配给孩子

i，这个孩子会得到满足。你的目标是尽可能满足越多数量的孩子，并输出这个最大数值。

示例 1:

输入: $g = [1,2,3]$, $s = [1,1]$

输出: 1

解释:

你有三个孩子和两块小饼干，3个孩子的胃口值分别是：1,2,3。

虽然你有两块小饼干，由于他们的尺寸都是1，你只能让胃口值是1的孩子满足。

所以你应该输出1。

示例 2:

输入: $g = [1,2]$, $s = [1,2,3]$

输出: 2

解释:

你有两个孩子和三块小饼干，2个孩子的胃口值分别是1,2。

你拥有的饼干数量和尺寸都足以让所有孩子满足。

所以你应该输出2。

提示:

- 1 <= g.length <= 3 * 10⁴
- 0 <= s.length <= 3 * 10⁴
- 1 <= g[i], s[j] <= 2³¹ - 1

3. 解法（贪心）：

（既然是很棒的家长，为什么不多买一些饼干呢（#狗头））

贪心策略：

先将两个数组排序。

针对胃口较小的孩子，从小到大挑选饼干：

- 如果当前饼干能满足，直接喂（最小的饼干都能满足，不要浪费大饼干）；
- 如果当前饼干不能满足，放弃这个饼干，去检测下一个饼干（这个饼干连最小胃口的孩子都无法满足，更别提那些胃口大的孩子了）。

C++ 算法代码：

```
1 class Solution
2 {
```

```

3 public:
4     int findContentChildren(vector<int>& g, vector<int>& s)
5     {
6         // 先排序
7         sort(g.begin(), g.end());
8         sort(s.begin(), s.end());
9
10        // 利用双指针找答案
11        int ret = 0, n = s.size();
12        for(int i = 0, j = 0; i < g.size() && j < n; i++, j++)
13        {
14            while(j < n && s[j] < g[i]) j++; // 找饼干
15            if(j < n) ret++;
16        }
17        return ret;
18    }
19 };

```

C++ 代码结果:

C++



Java 算法代码:

```

1 class Solution
2 {
3     public int findContentChildren(int[] g, int[] s)
4     {
5         // 排序
6         Arrays.sort(g);
7         Arrays.sort(s);
8
9         // 利用双指针找答案
10        int ret = 0, m = g.length, n = s.length;
11        for(int i = 0, j = 0; i < m && j < n; i++, j++)
12        {
13            while(j < n && s[j] < g[i]) j++; // 找饼干
14            if(j < n) ret++;
15        }
16        return ret;
17    }
18 }

```



```
17     }  
18 }
```

Java 运行结果：



16. 最优除法 (medium)

1. 题目链接：553. 最优除法

2. 题目描述：

给定一正整数数组 `nums`，`nums` 中的相邻整数将进行浮点除法。例如， $[2,3,4] \rightarrow 2 / 3 / 4$ 。

- 例如，`nums = [2,3,4]`，我们将求表达式的值 `"2/3/4"`。

但是，你可以在任意位置添加任意数目的括号，来改变算数的优先级。你需要找出怎么添加括号，以便计算后的表达式的值为最大值。

以字符串格式返回具有最大值的对应表达式。

注意：你的表达式不应该包含多余的括号。

示例 1：

输入：`[1000,100,10,2]`

输出：`"1000/(100/10/2)"`

解释： $1000/(100/10/2) = 1000/((100/10)/2) = 200$

但是，以下加粗的括号 `"1000/((100/10)/2)"` 是冗余的，因为他们并不影响操作的优先级，所以你需要返回 `"1000/(100/10/2)"`。

其他用例：

$1000/(100/10)/2 = 50$

$1000/(100/(10/2)) = 50$

$1000/100/10/2 = 0.5$

$1000/100/(10/2) = 2$

示例 2：

输入: nums = [2,3,4]

输出: "2/(3/4)"

解释: $(2/(3/4)) = 8/3 = 2.667$

可以看出，在尝试了所有的可能性之后，我们无法得到一个结果大于 2.667 的表达式。

说明:

- `1 <= nums.length <= 10`
- `2 <= nums[i] <= 1000`
- 对于给定的输入只有一种最优除法。

3. 解法（贪心）：

贪心策略：

在最终的结果中，前两个数的位置是无法改变的。

因为每一个数的都是大于等于 2 的，为了让结果更大，我们应该尽可能的把剩下的数全都放在「分子」上。

C++ 算法代码：

```
1 class Solution
2 {
3 public:
4     string optimalDivision(vector<int>& nums)
5     {
6         int n = nums.size();
7         // 先处理两个边界情况
8         if(n == 1)
9         {
10             return to_string(nums[0]);
11         }
12         if(n == 2)
13         {
14             return to_string(nums[0]) + "/" + to_string(nums[1]);
15         }
16         string ret = to_string(nums[0]) + "/" + to_string(nums[1]);
17         for(int i = 2; i < n; i++)
18         {
19             ret += "/" + to_string(nums[i]);
20         }
21         ret += ")";
22         return ret;
```

```
23     }  
24 };
```

C++ 代码结果:

C++



Java 算法代码:

```
1 class Solution  
2 {  
3     public String optimalDivision(int[] nums)  
4     {  
5         int n = nums.length;  
6         StringBuffer ret = new StringBuffer();  
7         // 先处理两个边界情况  
8         if(n == 1)  
9         {  
10            return ret.append(nums[0]).toString();  
11        }  
12        if(n == 2)  
13        {  
14            return ret.append(nums[0]).append("/").append(nums[1]).toString();  
15        }  
16        ret.append(nums[0]).append("/(").append(nums[1]);  
17        for(int i = 2; i < n; i++)  
18        {  
19            ret.append("/").append(nums[i]);  
20        }  
21        ret.append(")");  
22        return ret.toString();  
23    }  
24 }
```

Java 运行结果:



17. 跳跃游戏 II (medium)

1. 题目链接：45. 跳跃游戏 II

2. 题目描述：

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：

- $0 \leq j \leq \text{nums}[i]$
- $i + j < n$

返回到达 `nums[n - 1]` 的最小跳跃次数。生成的测试用例可以到达 `nums[n - 1]`。

示例 1:

输入: `nums = [2,3,1,1,4]`

输出: 2

解释: 跳到最后一个位置的最小跳跃数是 2。

从下标为 0 跳到下标为 1 的位置，跳 1 步，然后跳 3 步到达数组的最后一个位置。

示例 2:

输入: `nums = [2,3,0,1,4]`

输出: 2

提示:

- $1 \leq \text{nums.length} \leq 10^4$
- $0 \leq \text{nums}[i] \leq 1000$
- 题目保证可以到达 `nums[n-1]`

3. 解法（动态规划 + 类似层序遍历）：

动态规划：

a. 状态表示：

`dp[i]` 表示从 0 位置开始，到达 i 位置时候的最小跳跃次数

b. 状态转移方程：

对于 `dp[i]`，我们遍历 `0 ~ i - 1` 区间（用指针 `j` 表示），只要能够从 `j` 位置跳到 `i` 位置（`nums[j] + j >= i`），我们就用 `dp[j] + 1` 更新 `dp[i]` 里面的值，找到所有情况下的最小值即可。

类似层序遍历的过程：

用类似层序遍历的过程，将第 `i` 次跳跃的「起始位置」和「结束位置」找出来，用这次跳跃的情况，更新出下一次跳跃的「起始位置」和「终止位置」。

这样「循环往复」，就能更新出到达 `n - 1` 位置的最小跳跃步数。

C++ 算法代码：

```
1 class Solution
2 {
3 public:
4     int jump(vector<int>& nums)
5     {
6         int left = 0, right = 0, maxPos = 0, ret = 0, n = nums.size();
7         while(left <= right) // 保险的写法，以防跳不到 n - 1 的位置
8         {
9             if(maxPos >= n - 1) // 先判断一下是否已经能跳到最后一个位置
10            {
11                return ret;
12            }
13            // 遍历当成层，更新下一层的最右端点
14            for(int i = left; i <= right; i++)
15            {
16                maxPos = max(maxPos, nums[i] + i);
17            }
18            left = right + 1;
19            right = maxPos;
20            ret++;
21        }
22        return -1; // 跳不到的情况
23    }
24 };
```

C++ 代码结果：

C++

时间 4 ms

击败 99.62%

内存 16.6 MB

击败 19.34%

Java 算法代码:

```
1 class Solution
2 {
3     public int jump(int[] nums)
4     {
5         int left = 0, right = 0, ret = 0, maxPos = 0, n = nums.length;
6         while(left <= right) // 以防跳不到 n - 1 的位置
7         {
8             if(maxPos >= n - 1) // 判断是否已经能跳到最后一个位置
9             {
10                 return ret;
11             }
12             for(int i = left; i <= right; i++)
13             {
14                 // 更新下一层的最右端点
15                 maxPos = Math.max(maxPos, nums[i] + i);
16             }
17             left = right + 1;
18             right = maxPos;
19             ret++;
20         }
21         return -1;
22     }
23 }
```

Java 运行结果:

Java

时间 1 ms

击败 99.16%

内存 43.5 MB

击败 59.43%

18. 跳跃游戏 (medium)

1. 题目链接：55. 跳跃游戏

2. 题目描述

给你一个非负整数数组 `nums`，你最初位于数组的 **第一个下标**。数组中的每个元素代表你在该位置可以跳跃的最大长度。

判断你是否能够到达最后一个下标，如果可以，返回 `true`；否则，返回 `false`。

示例 1:

输入: `nums = [2,3,1,1,4]`

输出: `true`

解释: 可以先跳 1 步，从下标 0 到达下标 1, 然后再从下标 1 跳 3 步到达最后一个下标。

示例 2:

输入: `nums = [3,2,1,0,4]`

输出: `false`

解释: 无论如何，总会到达下标为 3 的位置。但该下标的最大跳跃长度是 0，所以永远不可能到达最后一个下标。

提示:

- `1 <= nums.length <= 10(4)`
- `0 <= nums[i] <= 10(5)`

3. 解法:

和 跳跃游戏II 一样，仅需修改一下返回值即可。

C++ 算法代码:

```
1 class Solution
2 {
3 public:
4     bool canJump(vector<int>& nums)
5     {
6         int left = 0, right = 0, maxPos = 0, n = nums.size();
7         while(left <= right)
8         {
9             if(maxPos >= n - 1)
10             {
11                 return true;
12             }
```

```

13         for(int i = left; i <= right; i++)
14         {
15             maxPos = max(maxPos, nums[i] + i);
16         }
17         left = right + 1;
18         right = maxPos;
19     }
20     return false;
21 }
22 };

```

C++ 代码结果:

C++



Java 算法代码:

```

1 class Solution
2 {
3     public boolean canJump(int[] nums)
4     {
5         int left = 0, right = 0, maxPos = 0, n = nums.length;
6         while(left <= right)
7         {
8             if(maxPos >= n - 1)
9             {
10                 return true;
11             }
12             for(int i = left; i <= right; i++)
13             {
14                 maxPos = Math.max(maxPos, nums[i] + i);
15             }
16             left = right + 1;
17             right = maxPos;
18         }
19         return false;
20     }
21 }

```


Java 运行结果：

Java

时间 2 ms

击败 89.11%

内存 42.3 MB

击败 98.77%

19. 加油站 (medium)

1. 题目链接：134. 加油站

2. 题目描述：

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。

示例 1:

输入: $gas = [1,2,3,4,5]$, $cost = [3,4,5,1,2]$

输出: 3

解释:

从 3 号加油站(索引为 3 处)出发，可获得 4 升汽油。此时油箱有 $= 0 + 4 = 4$ 升汽油

开往 4 号加油站，此时油箱有 $4 - 1 + 5 = 8$ 升汽油

开往 0 号加油站，此时油箱有 $8 - 2 + 1 = 7$ 升汽油

开往 1 号加油站，此时油箱有 $7 - 3 + 2 = 6$ 升汽油

开往 2 号加油站，此时油箱有 $6 - 4 + 3 = 5$ 升汽油

开往 3 号加油站，你需要消耗 5 升汽油，正好足够你返回到 3 号加油站。

因此，3 可为起始索引。

示例 2:

输入: $gas = [2,3,4]$, $cost = [3,4,3]$

输出: -1

解释:

你不能从 0 号或 1 号加油站出发，因为没有足够的汽油可以让你行驶到下一个加油站。

我们从 2 号加油站出发，可以获得 4 升汽油。此时油箱有 $= 0 + 4 = 4$ 升汽油

开往 0 号加油站，此时油箱有 $4 - 3 + 2 = 3$ 升汽油

开往 1 号加油站，此时油箱有 $3 - 3 + 3 = 3$ 升汽油

你无法返回 2 号加油站，因为返程需要消耗 4 升汽油，但是你的油箱只有 3 升汽油。
因此，无论如何，你都不可能绕环路行驶一周。

提示：

- `gas.length == n`
- `cost.length == n`
- `1 <= n <= 10(5)`
- `0 <= gas[i], cost[i] <= 10(4)`

3. 解法（暴力解法 -> 贪心）：

暴力解法：

- 依次枚举所有的起点；
- 从起点开始，模拟一遍加油的流程

贪心优化：

我们发现，当从 `i` 位置出发，走了 `step` 步之后，如果失败了。那么 `[i, i + step]` 这个区间内任意一个位置作为起点，都不可能环绕一圈。

因此我们枚举的下一个起点，应该是 `i + step + 1`。

C++ 算法代码：

```
1 class Solution
2 {
3 public:
4     int canCompleteCircuit(vector<int>& gas, vector<int>& cost)
5     {
6         int n = gas.size();
7         for(int i = 0; i < n; i++) // 依次枚举所有的起点
8         {
9             int rest = 0; // 标记一下净收益
10            int step = 0;
11            for( ; step < n; step++) // 枚举向后走的步数
12            {
13                int index = (i + step) % n; // 求出走 step 步之后的下标
14                rest = rest + gas[index] - cost[index];
15                if(rest < 0) break;
16            }
17            if(rest >= 0) return i;
```

```

18         i = i + step; // 优化
19     }
20     return -1;
21 }
22 };

```

C++ 代码结果：

C++



Java 算法代码：

```

1 class Solution
2 {
3     public int canCompleteCircuit(int[] gas, int[] cost)
4     {
5         int n = gas.length;
6         for(int i = 0; i < n; i++) // 依次枚举所有的起点
7         {
8             int rest = 0; // 统计净收益
9             int step = 0;
10            for( ; step < n; step++) // 枚举向后走的步数
11            {
12                int index = (i + step) % n; // 走 step 步之后的下标
13                rest = rest + gas[index] - cost[index];
14                if(rest < 0)
15                {
16                    break;
17                }
18            }
19            if(rest >= 0)
20            {
21                return i;
22            }
23            i = i + step; // 优化
24        }
25        return -1;
26    }
27 }

```

Java 运行结果：



20. 单调递增的数字 (medium)

1. 题目链接：738. 单调递增的数字

2. 题目描述：

当且仅当每个相邻位数上的数字 x 和 y 满足 $x \leq y$ 时，我们称这个整数是单调递增的。

给定一个整数 n ，返回 小于或等于 n 的最大数字，且数字呈单调递增。

• 示例 1:

输入: $n = 10$

输出: 9

• 示例 2:

输入: $n = 1234$

输出: 1234

• 示例 3:

输入: $n = 332$

输出: 299

• 提示:

$0 \leq n \leq 10^9$

3. 解法（贪心）：

- 为了方便处理数中的每一位数字，可以先讲整数转换成字符串；
- 从左往右扫描，找到第一个递减的位置；
- 从这个位置向前推，推到相同区域的最左端；
- 该点的值 `-1`，后面的所有数统一变成 `9`。

C++ 算法代码：

```

1 class Solution
2 {
3 public:
4     int monotoneIncreasingDigits(int n)
5     {
6         string s = to_string(n); // 把数字转化成字符串
7
8         int i = 0, m = s.size();
9         // 找第一个递减的位置
10        while(i + 1 < m && s[i] <= s[i + 1]) i++;
11
12        if(i + 1 == m) return n; // 判断一下特殊情况
13
14        // 回推
15        while(i - 1 >= 0 && s[i] == s[i - 1]) i--;
16
17        s[i]--;
18        for(int j = i + 1; j < m; j++) s[j] = '9';
19        return stoi(s);
20    }
21 };

```

C++ 代码结果：



Java 算法代码：

```

1 class Solution
2 {
3     public int monotoneIncreasingDigits(int n)
4     {
5         // 把数字转化成字符串
6         char[] s = Integer.toString(n).toCharArray();
7
8         int i = 0, m = s.length;
9         // 找第一个递减的位置
10        while(i + 1 < m && s[i] <= s[i + 1]) i++;
11        if(i == m - 1) return n; // 特判一下特殊情况
12

```

```

13      // 回退
14      while(i - 1 >= 0 && s[i] == s[i - 1]) i--;
15
16      s[i]--;
17      for(int j = i + 1; j < m; j++) s[j] = '9';
18
19      return Integer.parseInt(new String(s));
20  }
21 }

```

Java 运行结果：

Java

时间 1 ms

击败 94.91%

内存 38.4 MB

击败 51.39%

21. 坏了的计算器 (medium)

1. 题目链接：991. 坏了的计算器

2. 题目描述：

在显示着数字 `startValue` 的坏计算器上，我们可以执行以下两种操作：

- **双倍 (Double)**：将显示屏上的数字乘 2；
- **递减 (Decrement)**：将显示屏上的数字减 1。

给定两个整数 `startValue` 和 `target`。返回显示数字 `target` 所需的最小操作数。

示例 1：

输入：startValue = 2, target = 3

输出：2

解释：先进行双倍运算，然后再进行递减运算 {2 -> 4 -> 3}。

示例 2：

输入：startValue = 5, target = 8

输出：2

解释：先递减，再双倍 {5 -> 4 -> 8}。

示例 3：

输入：startValue = 3, target = 10

输出：3

解释：先双倍，然后递减，再双倍 {3 -> 6 -> 5 -> 10}.

提示：

- `1 <= startValue, target <= 10^9`

3. 解法（贪心）：

贪心策略：

正难则反：

当「反着」来思考的时候，我们发现：

- 当 `end <= begin` 的时候，只能执行「加法」操作；
- 当 `end > begin` 的时候，对于「奇数」来说，只能执行「加法」操作；对于「偶数」来说，最好的方式就是执行「除法」操作

这样的话，每次的操作都是「固定唯一」的。

C++ 算法代码：

```
1 class Solution
2 {
3 public:
4     int brokenCalc(int startValue, int target)
5     {
6         // 正难则反 + 贪心
7         int ret = 0;
8         while(target > startValue)
9         {
10             if(target % 2 == 0) target /= 2;
11             else target += 1;
12             ret++;
13         }
14         return ret + startValue - target;
15     }
16 };
```

C++ 代码结果：

C++

时间 0 ms

击败 100%

内存 6.2 MB

击败 9.75%

Java 算法代码：

```
1 class Solution
2 {
3     public int brokenCalc(int startValue, int target)
4     {
5         // 正难则反 + 贪心
6         int ret = 0;
7         while(target > startValue)
8         {
9             if(target % 2 == 0) target /= 2;
10            else target += 1;
11            ret++;
12        }
13        return ret + startValue - target;
14    }
15 }
```

Java 运行结果：

Java

时间 0 ms

击败 100%

内存 37.9 MB

击败 88.89%

22. 合并区间 (medium)

1. 题目链接：56. 合并区间

2. 题目描述：

以数组 `intervals` 表示若干个区间的集合，其中单个区间为 `intervals[i] = [start(i), end(i)]`。请你合并所有重叠的区间，并返回 一个不重叠的区间数组，该数组需恰好覆盖输入中的所有区间。

示例 1:

输入: intervals = [[1,3],[2,6],[8,10],[15,18]]

输出: [[1,6],[8,10],[15,18]]

解释: 区间 [1,3] 和 [2,6] 重叠, 将它们合并为 [1,6].

示例 2:

输入: intervals = [[1,4],[4,5]]

输出: [[1,5]]

解释: 区间 [1,4] 和 [4,5] 可被视为重叠区间。

提示:

- `1 <= intervals.length <= 10^4`
- `intervals[i].length == 2`
- `0 <= start(i) <= end(i) <= 10^4`

3. 解法 (排序 + 贪心) :

贪心策略:

- 先按照区间的「左端点」排序: 此时我们会发现, 能够合并的区间都是连续的;
- 然后从左往后, 按照求「并集」的方式, 合并区间。

如何求并集:

由于区间已经按照「左端点」排过序了, 因此当两个区间「合并」的时候, 合并后的区间:

- 左端点就是「前一个区间」的左端点;
- 右端点就是两者「右端点的最大值」。

C++ 算法代码:

```
1 class Solution
2 {
3 public:
4     vector<vector<int>> merge(vector<vector<int>>& intervals)
5     {
6         // 1. 先按照左端点排序
7         sort(intervals.begin(), intervals.end());
8     }
```

```

9      // 2. 合并区间
10     int left = intervals[0][0], right = intervals[0][1];
11     vector<vector<int>> ret;
12     for(int i = 1; i < intervals.size(); i++)
13     {
14         int a = intervals[i][0], b = intervals[i][1];
15         if(a <= right) // 有重叠部分
16         {
17             // 合并 - 求并集
18             right = max(right, b);
19         }
20         else // 没有重叠部分
21         {
22             ret.push_back({left, right}); // 加入到结果中
23             left = a;
24             right = b;
25         }
26     }
27     // 别忘了最后一个区间
28     ret.push_back({left, right});
29     return ret;
30 }
31 };

```

C++ 代码结果:

C++



Java 算法代码:

```

1  class Solution
2  {
3      public int[][] merge(int[][] intervals)
4      {
5          // 1. 按照左端点排序
6          Arrays.sort(intervals, (v1, v2) ->
7              {
8                  return v1[0] - v2[0];
9              });
10

```

```

11      // 2. 合并区间 - 求并集
12      int left = intervals[0][0], right = intervals[0][1];
13      List<int[]> ret = new ArrayList<>();
14      for(int i = 1; i < intervals.length; i++)
15      {
16          int a = intervals[i][0], b = intervals[i][1];
17          if(a <= right) // 有重叠部分
18          {
19              // 合并 - 求并集
20              right = Math.max(right, b);
21          }
22          else // 不能合并
23          {
24              ret.add(new int[]{left, right});
25              left = a;
26              right = b;
27          }
28      }
29      // 别忘了最后一个区间
30      ret.add(new int[]{left, right});
31      return ret.toArray(new int[0][]);
32  }
33 }

```

Java 运行结果:

Java



23. 无重叠区间 (medium)

1. 题目链接: [435. 无重叠区间](#)

2. 题目描述

给定一个区间的集合 `intervals`，其中 `intervals[i] = [start(i), end(i)]`。返回需要移除区间的最小数量，使剩余区间互不重叠。

示例 1:

输入: `intervals = [[1,2],[2,3],[3,4],[1,3]]`

输出:1

解释:移除 [1,3] 后,剩下的区间没有重叠。

示例 2:

输入:intervals = [[1,2], [1,2], [1,2]]

输出:2

解释:你需要移除两个 [1,2] 来使剩下的区间没有重叠。

示例 3:

输入:intervals = [[1,2], [2,3]]

输出:0

解释:你不需要移除任何区间,因为它们已经是无重叠的了。

提示:

- `1 <= intervals.length <= 10^5`
- `intervals[i].length == 2`
- `-5 * 10^4 <= start(i) < end(i) <= 5 * 10^4`

3. 解法 (贪心) :

贪心策略:

- 按照「左端点」排序;
- 当两个区间「重叠」的时候,为了能够「在移除某个区间后,保留更多的区间」,我们应该把「区间范围较大」的区间移除。

如何移除区间范围较大的区间:

由于已经按照「左端点」排序了,因此两个区间重叠的时候,我们应该移除「右端点较大」的区间,

C++ 算法代码:

```
1 class Solution
2 {
3 public:
4     int eraseOverlapIntervals(vector<vector<int>>& intervals)
5     {
6         // 1. 按照左端点排序
```

```

7         sort(intervals.begin(), intervals.end());
8
9         // 2. 移除区间
10        int ret = 0;
11        int left = intervals[0][0], right = intervals[0][1];
12        for(int i = 1; i < intervals.size(); i++)
13        {
14            int a = intervals[i][0], b = intervals[i][1];
15            if(a < right) // 有重叠部分
16            {
17                ret++; // 删掉一个区间
18                right = min(right, b);
19            }
20            else // 没有重叠部分
21            {
22                right = b;
23            }
24        }
25        return ret;
26    }
27 };

```

C++ 代码结果:



Java 算法代码:

```

1 class Solution
2 {
3     public int eraseOverlapIntervals(int[][] intervals)
4     {
5         // 1. 按照左端点排序
6         Arrays.sort(intervals, (v1, v2) ->
7         {
8             return v1[0] - v2[0];
9         });
10
11        // 2. 移除区间
12        int ret = 0;

```

```

13     int left = intervals[0][0], right = intervals[0][1];
14     for(int i = 1; i < intervals.length; i++)
15     {
16         int a = intervals[i][0], b = intervals[i][1];
17         if(a < right) // 有重叠区间
18         {
19             ret++;
20             right = Math.min(right, b); // 贪心：删除右端点较大的区间
21         }
22         else // 没有重叠区间
23         {
24             right = b;
25         }
26     }
27     return ret;
28 }
29 }

```

Java 运行结果：



24. 用最少数量的箭引爆气球 (medium)

1. 题目链接：452. 用最少数量的箭引爆气球

2. 题目描述

有一些球形气球贴在一堵用 XY 平面表示的墙面上。墙面上的气球记录在整数数组 `points`，其中 `points[i] = [x(start), x(end)]` 表示水平直径在 `x(start)` 和 `x(end)` 之间的气球。你不知道气球的确切 y 坐标。

一支弓箭可以沿着 x 轴从不同点 **完全垂直** 地射出。在坐标 `x` 处射出一支箭，若有一个气球的直径的开始和结束坐标为 `x(start)`，`x(end)` 且满足 `x(start) ≤ x ≤ x(end)`，则该气球会被 **引爆**。可以射出的弓箭的数量 **没有限制**。弓箭一旦被射出之后，可以无限地前进。

给你一个数组 `points`，返回引爆所有气球所必须射出的 **最小** 弓箭数。

示例 1：

输入：points = [[10,16],[2,8],[1,6],[7,12]]

输出：2

解释：气球可以用2支箭来爆破：

- 在 $x = 6$ 处射出箭，击破气球 $[2,8]$ 和 $[1,6]$ 。
- 在 $x = 11$ 处发射箭，击破气球 $[10,16]$ 和 $[7,12]$ 。

示例 2：

输入：points = $[[1,2],[3,4],[5,6],[7,8]]$

输出：4

解释：每个气球需要射出一支箭，总共需要4支箭。

示例 3：

输入：points = $[[1,2],[2,3],[3,4],[4,5]]$

输出：2

解释：气球可以用2支箭来爆破：

- 在 $x = 2$ 处发射箭，击破气球 $[1,2]$ 和 $[2,3]$ 。
- 在 $x = 4$ 处射出箭，击破气球 $[3,4]$ 和 $[4,5]$ 。

提示：

- `1 <= points.length <= 10^5`
- `points[i].length == 2`
- `-2^31 <= x(start) < x(end) <= 2^31 - 1`

3. 解法（贪心）：

贪心策略：

- 按照左端点排序，我们发现，排序后有这样一个性质：「互相重叠的区间都是连续的」；
- 这样，我们在射箭的时候，要发挥每一支箭「最大的作用」，应该把「互相重叠的区间」统一引爆。

如何求互相重叠区间？

由于我们是按照「左端点」排序的，因此对于两个区间，我们求的是它们的「交集」：

- 左端点为两个区间左端点的「最大值」（但是左端点不会影响我们的合并结果，所以可以忽略）；
- 右端点为两个区间右端点的「最小值」。

C++ 算法代码：

```

1 class Solution
2 {
3 public:
4     int findMinArrowShots(vector<vector<int>>& points)
5     {
6         // 1. 按照左端点排序
7         sort(points.begin(), points.end());
8
9         // 2. 求互相重叠区间的数量
10        int right = points[0][1];
11        int ret = 1;
12        for(int i = 1; i < points.size(); i++)
13        {
14            int a = points[i][0], b = points[i][1];
15            if(a <= right) // 有重叠部分
16            {
17                right = min(right, b);
18            }
19            else // 无重叠部分
20            {
21                ret++;
22                right = b;
23            }
24        }
25        return ret;
26    }
27 };

```

C++ 代码结果:



Java 算法代码:

```

1 class Solution
2 {
3     public int findMinArrowShots(int[][] points)
4     {
5         // 1. 按照左端点排序
6         Arrays.sort(points, (v1, v2) ->

```



```

7      {
8          return v1[0] > v2[0] ? 1 : -1;
9      });
10
11      // 2. 求出互相重叠的区间的数量
12      int right = points[0][1];
13      int ret = 1;
14      for(int i = 1; i < points.length; i++)
15      {
16          int a = points[i][0], b = points[i][1];
17          if(a <= right) // 有重叠
18          {
19              right = Math.min(right, b);
20          }
21          else // 没有重叠
22          {
23              ret++;
24              right = b;
25          }
26      }
27      return ret;
28  }
29  }

```

Java 运行结果：



25. 整数替换 (medium)

1. 题目链接：397. 整数替换

2. 题目描述

给定一个正整数 n ，你可以做如下操作：

- 如果 n 是偶数，则用 $n / 2$ 替换 n 。
- 如果 n 是奇数，则可以用 $n + 1$ 或 $n - 1$ 替换 n 。

返回 n 变为 1 所需的最小替换次数。

示例 1:

输入: $n = 8$

输出: 3

解释: $8 \rightarrow 4 \rightarrow 2 \rightarrow 1$

示例 2:

输入: $n = 7$

输出: 4

解释: $7 \rightarrow 8 \rightarrow 4 \rightarrow 2 \rightarrow 1$

或 $7 \rightarrow 6 \rightarrow 3 \rightarrow 2 \rightarrow 1$

示例 3:

输入: $n = 4$

输出: 2

提示:

- $1 \leq n \leq 2^{31} - 1$

3. 解法 (贪心) :

贪心策略:

我们的任何选择, 应该让这个数尽可能快的变成 1。

对于偶数: 只能执行除 2 操作, 没有什么分析的;

对于奇数:

- 当 $n == 1$ 的时候, 不用执行任何操作;
- 当 $n == 3$ 的时候, 变成 1 的最优操作数是 2;
- 当 $n > 1 \ \&\& \ n \% 3 == 1$ 的时候, 那么它的二进制表示是 $\dots\dots\dots01$, 最优的方式应该选择 -1 , 这样就可以把末尾的 1 干掉, 接下来执行除法操作, 能够更快的变成 1;
- 当 $n > 3 \ \&\& \ n \% 3 == 3$ 的时候, 那么它的二进制表示是 $\dots\dots\dots11$, 此时最优的策略应该是 $+1$, 这样可以把一堆连续的 1 转换成 0, 更快的变成 1。

C++ 算法代码:

```
1 class Solution
2 {
3 public:
4     int integerReplacement(int n)
```

```

5      {
6          int ret = 0;
7          while(n > 1)
8          {
9              // 分类讨论
10             if(n % 2 == 0)
11             {
12                 ret++;
13                 n /= 2;
14             }
15             else
16             {
17                 if(n == 3)
18                 {
19                     ret += 2;
20                     n = 1;
21                 }
22                 else if(n % 4 == 1)
23                 {
24                     ret += 2;
25                     n /= 2;
26                 }
27                 else
28                 {
29                     ret += 2;
30                     n = n / 2 + 1;
31                 }
32             }
33         }
34         return ret;
35     }
36 };

```

C++ 代码结果：

C++



Java 算法代码：

```
1 class Solution
```

```

2 {
3     public int integerReplacement(int n)
4     {
5         int ret = 0;
6         while(n > 1)
7         {
8             // 分类讨论
9             if(n % 2 == 0) // 如果是偶数
10            {
11                n /= 2;
12                ret++;
13            }
14            else
15            {
16                if(n == 3)
17                {
18                    ret += 2;
19                    n = 1;
20                }
21                else if(n % 4 == 1)
22                {
23                    n = n / 2;
24                    ret += 2;
25                }
26                else
27                {
28                    n = n / 2 + 1;
29                    ret += 2;
30                }
31            }
32        }
33        return ret;
34    }
35 }

```

Java 运行结果：



26. 俄罗斯套娃信封问题 (hard)

1. 题目链接：354. 俄罗斯套娃信封问题

2. 题目描述：

给你一个二维整数数组 `envelopes`，其中 `envelopes[i] = [w(i), h(i)]`，表示第 `i` 个信封的宽度和高度。

当另一个信封的宽度和高度都比这个信封大的时候，这个信封就可以放进另一个信封里，如同俄罗斯套娃一样。

请计算 **最多能有多少个** 信封能组成一组“俄罗斯套娃”信封（即可以把一个信封放到另一个信封里面）。

注意：不允许旋转信封。

示例 1：

输入：`envelopes = [[5,4],[6,4],[6,7],[2,3]]`

输出：3

解释：最多信封的个数为3, 组合为:[2,3] => [5,4] => [6,7]。

示例 2：

输入：`envelopes = [[1,1],[1,1],[1,1]]`

输出：1

提示：

- `1 <= envelopes.length <= 10^5`
- `envelopes[i].length == 2`
- `1 <= w(i), h(i) <= 10^5`

3. 解法一（动态规划）：

将数组按照左端点排序之后，问题就转化成了最长上升子序列模型，那接下来我们就可以用解决最长上升子序列的经验，来解决这个问题（虽然会超时，但是还是要好好写代码）。

1. 状态表示：

`dp[i]` 表示：以 `i` 位置的信封为结尾的所有套娃序列中，最长的套娃序列的长度；

2. 状态转移方程：

`dp[i] = max(dp[j] + 1)` 其中 `0 <= j < i && e[i][0] > e[j][0] && e[i][1] > e[j][1]`；

3. 初始化:

全部初始化为 1;

4. 填表顺序:

从左往右;

5. 返回值:

整个 dp 表中的最大值。

解法一：C++ 算法代码:

```
1 class Solution
2 {
3 public:
4     int maxEnvelopes(vector<vector<int>>& e)
5     {
6         // 解法一：动态规划
7         // 预处理
8         sort(e.begin(), e.end());
9
10        int n = e.size();
11        vector<int> dp(n, 1);
12        int ret = 1;
13
14        for(int i = 1; i < n; i++)
15        {
16            for(int j = 0; j < i; j++)
17            {
18                if(e[i][0] > e[j][0] && e[i][1] > e[j][1])
19                {
20                    dp[i] = max(dp[i], dp[j] + 1);
21                }
22            }
23            ret = max(ret, dp[i]);
24        }
25        return ret;
26    }
27 };
```

解法一：Java 算法代码：

```
1 class Solution
2 {
3     public int maxEnvelopes(int[][] e)
4     {
5         // 解法一：动态规划
6         Arrays.sort(e, (v1, v2) ->
7         {
8             return v1[0] - v2[0];
9         });
10
11         int n = e.length;
12         int[] dp = new int[n];
13         int ret = 1;
14
15         for(int i = 0; i < n; i++)
16         {
17             dp[i] = 1; // 初始化
18             for(int j = 0; j < i; j++)
19             {
20                 if(e[i][0] > e[j][0] && e[i][1] > e[j][1])
21                 {
22                     dp[i] = Math.max(dp[i], dp[j] + 1);
23                 }
24             }
25             ret = Math.max(ret, dp[i]);
26         }
27         return ret;
28     }
29 }
```

4. 解法二（重写排序 + 贪心 + 二分）：

当我们将整个信封按照「下面的规则」排序之后：

- i. 左端点不同的时候：按照「左端点从小到大」排序；
- ii. 左端点相同的时候：按照「右端点从大到小」排序

我们发现，问题就变成了仅考虑信封的「右端点」，完完全全的变成的「最长上升子序列」的模型。那么我们就可以用「贪心 + 二分」优化我们的算法。

解法二：C++ 算法代码：

```
1 class Solution
2 {
3 public:
4     int maxEnvelopes(vector<vector<int>>& e)
5     {
6         // 解法二：重写排序 + 贪心 + 二分
7         sort(e.begin(), e.end(), [&](const vector<int>& v1, const vector<int>&
v2)
8         {
9             return v1[0] != v2[0] ? v1[0] < v2[0] : v1[1] > v2[1];
10        });
11
12        // 贪心 + 二分
13        vector<int> ret;
14        ret.push_back(e[0][1]);
15
16        for(int i = 1; i < e.size(); i++)
17        {
18            int b = e[i][1];
19            if(b > ret.back())
20            {
21                ret.push_back(b);
22            }
23            else
24            {
25                int left = 0, right = ret.size() - 1;
26                while(left < right)
27                {
28                    int mid = (left + right) / 2;
29                    if(ret[mid] >= b) right = mid;
30                    else left = mid + 1;
31                }
32                ret[left] = b;
33            }
34        }
35        return ret.size();
36    }
37 };
```

解法二：C++ 代码结果：

解法二：Java 算法代码：

```
1 class Solution
2 {
3     public int maxEnvelopes(int[][] e)
4     {
5         // 解法二：重写排序 + 贪心 + 二分
6         Arrays.sort(e, (v1, v2) ->
7         {
8             return v1[0] != v2[0] ? v1[0] - v2[0] : v2[1] - v1[1];
9         });
10
11         // 贪心 + 二分
12         ArrayList<Integer> ret = new ArrayList<>();
13         ret.add(e[0][1]);
14
15         for(int i = 1; i < e.length; i++)
16         {
17             int b = e[i][1];
18             if(b > ret.get(ret.size() - 1))
19             {
20                 ret.add(b);
21             }
22             else
23             {
24                 int left = 0, right = ret.size() - 1;
25                 while(left < right)
26                 {
27                     int mid = (left + right) / 2;
28                     if(ret.get(mid) >= b) right = mid;
29                     else left = mid + 1;
30                 }
31                 ret.set(left, b);
32             }
33         }
34         return ret.size();
35     }
36 }
```

解法二：Java 运行结果：



27. 可被三整除的最大和 (medium)

1. 题目链接：1262. 可被三整除的最大和

2. 题目描述

给你一个整数数组 `nums`，请你找出并返回能被三整除的元素最大和。

示例 1：

输入：nums = [3,6,5,1,8]

输出：18

解释：选出数字 3, 6, 1 和 8，它们的和是 18（可被 3 整除的最大和）。

示例 2：

输入：nums = [4]

输出：0

解释：4 不能被 3 整除，所以无法选出数字，返回 0。

示例 3：

输入：nums = [1,2,3,4,4]

输出：12

解释：选出数字 1, 3, 4 以及 4，它们的和是 12（可被 3 整除的最大和）。

提示：

- $1 \leq \text{nums.length} \leq 4 \times 10^4$
- $1 \leq \text{nums}[i] \leq 10^4$

3. 解法（正难则反 + 贪心 + 分类讨论）：

正难则反：

我们可以先把所有的数累加在一起，然后根据累加和的结果，贪心的删除一些数。

分类讨论：

设累加和为 `sum`，用 `x` 标记 `%3 == 1` 的数，用 `y` 标记 `% 3 == 2` 的数。

那么根据 `sum` 的余数，可以分为下面三种情况：

- a. `sum % 3 == 0`，此时所有元素的和就是满足要求的，那么我们一个也不用删除；
- b. `sum % 3 == 1`，此时数组中要么存在一个 `x`，要么存在两个 `y`。因为我们要的是最大值，所以应该选择 `x` 中最小的那么数，记为 `x1`，或者是 `y` 中最小以及次小的两个数，记为 `y1, y2`。

那么，我们应该选择两种情况下的最大值：`max(sum - x1, sum - y1 - y2)`；

- c. `sum % 3 == 2`，此时数组中要么存在一个 `y`，要么存在两个 `x`。因为我们要的是最大值，所以应该选择 `y` 中最小的那么数，记为 `y1`，或者是 `x` 中最小以及次小的两个数，记为 `x1, x2`。

那么，我们应该选择两种情况下的最大值：`max(sum - y1, sum - x1 - x2)`；

C++ 算法代码：

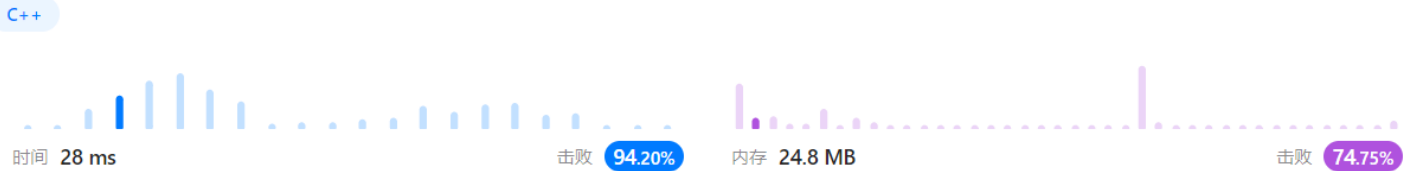
```
1 class Solution
2 {
3 public:
4     int maxSumDivThree(vector<int>& nums)
5     {
6         const int INF = 0x3f3f3f3f;
7
8         int sum = 0, x1 = INF, x2 = INF, y1 = INF, y2 = INF;
9         for(auto x : nums)
10        {
11            sum += x;
12            if(x % 3 == 1)
13            {
14                if(x < x1) x2 = x1, x1 = x;
15                else if(x < x2) x2 = x;
16            }
17            else if(x % 3 == 2)
18            {
19                if(x < y1) y2 = y1, y1 = x;
20                else if(x < y2) y2 = x;
21            }
22        }
23
24        // 分类讨论
```

```

25         if(sum % 3 == 0) return sum;
26     else if(sum % 3 == 1) return max(sum - x1, sum - y1 - y2);
27     else return max(sum - y1, sum - x1 - x2);
28     }
29 };

```

C++ 代码结果：



Java 算法代码：

```

1  class Solution
2  {
3      public int maxSumDivThree(int[] nums)
4      {
5          int INF = 0x3f3f3f3f;
6
7          int sum = 0, x1 = INF, x2 = INF, y1 = INF, y2 = INF;
8          for(int x : nums)
9          {
10             sum += x;
11             if(x % 3 == 1)
12             {
13                 if(x < x1)
14                 {
15                     x2 = x1;
16                     x1 = x;
17                 }
18                 else if(x < x2)
19                 {
20                     x2 = x;
21                 }
22             }
23             else if(x % 3 == 2)
24             {
25                 if(x < y1)
26                 {
27                     y2 = y1;
28                     y1 = x;

```

```

29         }
30         else if(x < y2)
31         {
32             y2 = x;
33         }
34     }
35 }
36
37 // 分类讨论
38 if(sum % 3 == 0) return sum;
39 else if(sum % 3 == 1) return Math.max(sum - x1, sum - y1 - y2);
40 else return Math.max(sum - y1, sum - x1 - x2);
41 }
42 }

```

Java 运行结果：

Java



28. 距离相等的条形码 (medium)

1. 题目链接：1054. 距离相等的条形码

2. 题目描述

在一个仓库里，有一排条形码，其中第 i 个条形码为 `barcodes[i]`。

请你重新排列这些条形码，使其中任意两个相邻的条形码不能相等。你可以返回任何满足该要求的答案，此题保证存在答案。

示例 1：

输入：barcodes = [1,1,1,2,2,2]

输出：[2,1,2,1,2,1]

示例 2：

输入：barcodes = [1,1,1,1,2,2,3,3]

输出：[1,3,1,3,2,1,2,1]

提示：

- `1 <= barcodes.length <= 10000`
- `1 <= barcodes[i] <= 10000`

3. 解法（贪心）：

贪心策略：

每次处理一批相同的数字，往 n 个空里面摆放；

每次摆放的时候，隔一个格子摆放一个数；

优先处理出现次数最多的那个数。

C++ 算法代码：

```
1 class Solution
2 {
3 public:
4     vector<int> rearrangeBarcodes(vector<int>& b)
5     {
6         unordered_map<int, int> hash; // 统计每个数出现的频次
7         int maxVal = 0, maxCount = 0;
8         for(auto x : b)
9         {
10             if(maxCount < ++hash[x])
11             {
12                 maxCount = hash[x];
13                 maxVal = x;
14             }
15         }
16
17         int n = b.size();
18         vector<int> ret(n);
19         int index = 0;
20         // 先处理出现次数最多的那个数
21         for(int i = 0; i < maxCount; i++)
22         {
23             ret[index] = maxVal;
24             index += 2;
25         }
26
27         // 处理剩下的数
28         hash.erase(maxVal);
29         for(auto& [x, y] : hash)
30         {
```

```

31         for(int i = 0; i < y; i++)
32         {
33             if(index >= n) index = 1;
34             ret[index] = x;
35             index += 2;
36         }
37     }
38     return ret;
39 }
40 };

```

C++ 代码结果:

C++



Java 算法代码:

```

1  class Solution
2  {
3      public int[] rearrangeBarcodes(int[] b)
4      {
5          Map<Integer, Integer> hash = new HashMap<>(); // 统计每个数出现了多少次
6          int maxVal = 0, maxCount = 0;
7          for(int x : b)
8          {
9              hash.put(x, hash.getOrDefault(x, 0) + 1);
10             if(maxCount < hash.get(x))
11             {
12                 maxVal = x;
13                 maxCount = hash.get(x);
14             }
15         }
16
17         int n = b.length;
18         int[] ret = new int[n];
19         int index = 0;
20         // 先处理出现次数最多的那个数
21         for(int i = 0; i < maxCount; i++)
22         {
23             ret[index] = maxVal;

```

```

24         index += 2;
25     }
26
27     hash.remove(maxVal);
28     for(int x : hash.keySet())
29     {
30         for(int i = 0; i < hash.get(x); i++)
31         {
32             if(index >= n) index = 1;
33             ret[index] = x;
34             index += 2;
35         }
36     }
37     return ret;
38 }
39 }

```

Java 运行结果:

Java



29. 重构字符串 (medium)

1. 题目链接: 767. 重构字符串

2. 题目描述

给定一个字符串 `s`，检查是否能重新排布其中的字母，使得两相邻的字符不同。

返回 `s` 的任意可能的重新排列。若不可行，返回空字符串 `""`。

示例 1:

输入: `s = "aab"`

输出: `"aba"`

示例 2:

输入: `s = "aaab"`

输出: `""`

提示:

- `1 <= s.length <= 500`
- `s` 只包含小写字母

3. 解法（贪心）：

贪心策略：

与上面的一道题解法一致~

C++ 算法代码：

```
1 class Solution
2 {
3 public:
4     string reorganizeString(string s)
5     {
6         int hash[26] = { 0 };
7         char maxChar = ' ';
8         int maxCount = 0;
9         for(auto ch : s)
10        {
11            if(maxCount < ++hash[ch - 'a'])
12            {
13                maxChar = ch;
14                maxCount = hash[ch - 'a'];
15            }
16        }
17
18        // 先判断一下
19        int n = s.size();
20        if(maxCount > (n + 1) / 2) return "";
21
22        string ret(n, ' ');
23        int index = 0;
24        // 先处理出现次数最多的那个字符
25        for(int i = 0; i < maxCount; i++)
26        {
27            ret[index] = maxChar;
28            index += 2;
29        }
30        hash[maxChar - 'a'] = 0;
31
32        for(int i = 0; i < 26; i++)
33        {
```

```

34         for(int j = 0; j < hash[i]; j++)
35         {
36             if(index >= n) index = 1;
37             ret[index] = 'a' + i;
38             index += 2;
39         }
40     }
41     return ret;
42 }
43 };

```

C++ 代码结果:



Java 算法代码:

```

1 class Solution
2 {
3     public String reorganizeString(String s)
4     {
5         int[] hash = new int[26];
6         char maxChar = ' ';
7         int maxCount = 0;
8         for(int i = 0; i < s.length(); i++)
9         {
10             char ch = s.charAt(i);
11             if(maxCount < ++hash[ch - 'a'])
12             {
13                 maxChar = ch;
14                 maxCount = hash[ch - 'a'];
15             }
16         }
17
18         int n = s.length();
19         // 先判断
20         if(maxCount > (n + 1) / 2) return "";
21
22         char[] ret = new char[n];
23         int index = 0;

```

```

24      // 先处理次数最多的字符
25      for(int i = 0; i < maxCount; i++)
26      {
27          ret[index] = maxChar;
28          index += 2;
29      }
30      hash[maxChar - 'a'] = 0;
31
32      for(int i = 0; i < 26; i++)
33      {
34          for(int j = 0; j < hash[i]; j++)
35          {
36              if(index >= n) index = 1;
37              ret[index] = (char)(i + 'a');
38              index += 2;
39          }
40      }
41      return new String(ret);
42  }
43 }

```

Java 运行结果：

Java

