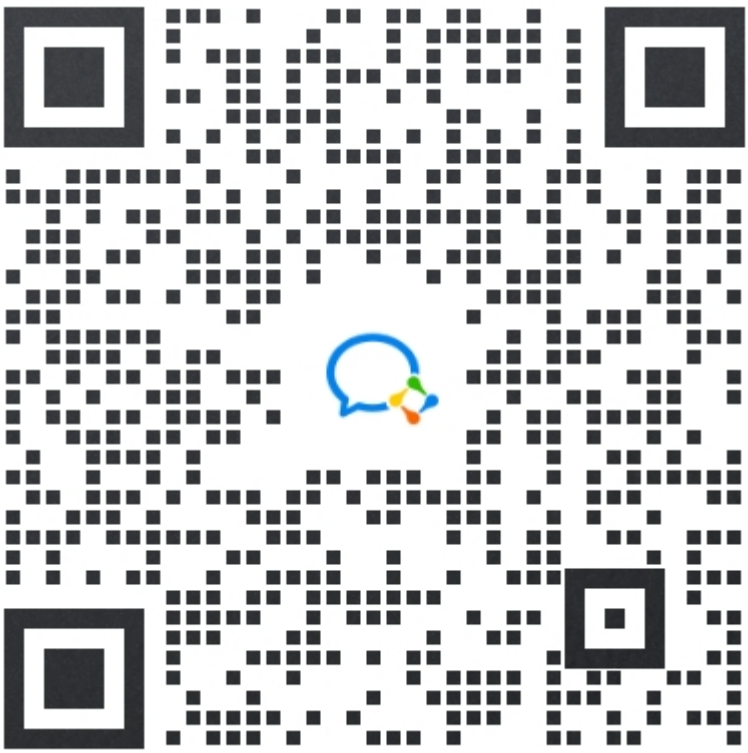
优选算法精品课(No.001~No.033)

版权说明

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

双指针

> 常见的双指针有两种形式，一种是对撞指针，一种是左右指针。
>
> **对撞指针：**一般用于顺序结构中，也称左右指针。
>
> - 对撞指针从两端向中间移动。一个指针从最左端开始，另一个从最右端开始，然后逐渐往中间逼近。
> - 对撞指针的终止条件一般是两个指针相遇或者错开（也可能在循环内部找到结果直接跳出循环），也就是：
>   - `left == right`（两个指针指向同一个位置）
>   - `left > right`（两个指针错开）
>
> **快慢指针：**又称为龟兔赛跑算法，其基本思想就是使用两个移动速度不同的指针在数组或链表等序列结构上移动。
>
> 这种方法对于处理环形链表或数组非常有用。
>
> 其实不单单是环形链表或者是数组，如果我们要研究的问题出现循环往复的情况时，均可考虑使用快慢指针的思想。
>
> 快慢指针的实现方式有很多种，最常用的一种就是：
>
> - 在一次循环中，每次让慢的指针向后移动一位，而快的指针往后移动两位，实现一快一慢。

1. 移动零（easy）

「数组分两块」是非常常见的一种题型，主要就是根据一种划分方式，将数组的内容分成左右两部分。这种类型的题，一般就是使用「双指针」来解决。

1. **题目链接：283. 移动零**
2. **题目描述：**

> 给定一个数组 `nums`，编写一个函数将所有 `0` 移动到数组的末尾，同时保持非零元素的相对顺序。
>
> 请注意 ，必须在不复制数组的情况下原地对数组进行操作。
>
> 示例 1:
>
> 输入: `nums = [0,1,0,3,12]`
>
> 输出: `[1,3,12,0,0]`
>
> 示例 2:
>
> 输入: `nums = [0]`

> 输出: `[0]`

3. **解法（快排的思想：数组划分区间 - 数组分两块）：**

算法思路：

在本题中，我们可以用一个 `cur` 指针来扫描整个数组，另一个 `dest` 指针用来记录非零数序列的最后一个位置。根据 `cur` 在扫描的过程中，遇到的不同情况，分类处理，实现数组的划分。

在 `cur` 遍历期间，使 `[0, dest]` 的元素全部都是非零元素，`[dest + 1, cur - 1]` 的元素全是零。

算法流程：

a. 初始化 `cur = 0`（用来遍历数组），`dest = -1`（指向非零元素序列的最后一个位置。因为刚开始我们不知道最后一个非零元素在什么位置，因此初始化为 `-1`）

b. `cur` 依次往后遍历每个元素，遍历到的元素会有下面两种情况：

i. 遇到的元素是 `0`，`cur` 直接 `++`。因为我们的目标是让 `[dest + 1, cur - 1]` 内的元素全都是零，因此当 `cur` 遇到 `0` 的时候，直接 `++`，就可以让 `0` 在 `cur - 1` 的位置上，从而在 `[dest + 1, cur - 1]` 内；

ii. 遇到的元素不是 `0`，`dest++`，并且交换 `cur` 位置和 `dest` 位置的元素，之后让 `cur++`，扫描下一个元素。

- 因为 `dest` 指向的位置是非零元素区间的最后一个位置，如果扫描到一个新的非零元素，那么它的位置应该在 `dest + 1` 的位置上，因此 `dest` 先自增 `1`；
- `dest++` 之后，指向的元素就是 `0` 元素（因为非零元素区间末尾的后一个元素就是 `0`），因此可以交换到 `cur` 所处的位置上，实现 `[0, dest]` 的元素全部都是非零元素，`[dest + 1, cur - 1]` 的元素全是零。

C++ 算法代码：

```
class Solution
{
public:
    void moveZeroes(vector<int>& nums)
    {
        for(int cur = 0, dest = -1; cur < nums.size(); cur++)
            if(nums[cur]) // 处理非零元素
                swap(nums[++dest], nums[cur]);
    }
};
```

C++ 代码结果：

Java 算法代码：

```java
class Solution
{
    public void moveZeroes(int[] nums)
    {
        for(int cur = 0, dest = -1; cur < nums.length; cur++)
            if(nums[cur] != 0) // 仅需处理非零元素
            {
                dest++; // dest 先向后移动一位
                // 交换
                int tmp = nums[cur];
                nums[cur] = nums[dest];
                nums[dest] = tmp;
            }
    }
}
```

Java 代码结果：

算法总结：

这个方法是往后我们学习「快排算法」的时候，「数据划分」过程的重要一步。如果将快排算法拆解的话，这一段小代码就是实现快排算法的「核心步骤」。

2. 复写零（easy）

1. **题目链接：1089. 复写零**

2. **题目描述：**

> 给你一个长度固定的整数数组 `arr` ，请你将该数组中出现的每个零都复写一遍，并将其余的元素向右平移。
>
> 注意：请不要在超过该数组长度的位置写入元素。请对输入的数组就地进行上述修改，不要从函数返回任何东西。
>
> 示例 1：
>
> 输入：`arr = [1,0,2,3,0,4,5,0]`
> 输出：`[1,0,0,2,3,0,0,4]`
>
> 解释：
>
> 调用函数后，输入的数组将被修改为：`[1,0,0,2,3,0,0,4]`

3. **解法（原地复写 - 双指针）：**

算法思路：

如果「从前向后」进行原地复写操作的话，由于 `0` 的出现会复写两次，导致没有复写的数「被覆盖掉」。因此我们选择「从后往前」的复写策略。

但是「从后向前」复写的时候，我们需要找到「最后一个复写的数」，因此我们的大体流程分两步：

i. 先找到最后一个复写的数；

ii. 然后从后向前进行复写操作。

算法流程：

a. 初始化两个指针 `cur = 0` ， `dest = 0` ；

b. 找到最后一个复写的数：

i. 当 `cur < n` 的时候，一直执行下面循环：

- 判断 `cur` 位置的元素：
  - 如果是 `0` 的话， `dest` 往后移动两位；
  - 否则， `dest` 往后移动一位。
- 判断 `dest` 时候已经到结束位置，如果结束就终止循环；
- 如果没有结束， `cur++` ，继续判断。

c. 判断 `dest` 是否越界到 `n` 的位置：

i. 如果越界，执行下面三步：

1. `n - 1` 位置的值修改成 `0` ；
2. `cur` 向移动一步；

3. `dest` 向前移动两步。

d. 从 `cur` 位置开始往前遍历原数组，依次还原出复写后的结果数组：

i. 判断 `cur` 位置的值：

1. 如果是 `0`：`dest` 以及 `dest - 1` 位置修改成 `0`，`dest -= 2`；
2. 如果非零：`dest` 位置修改成 `0`，`dest -= 1`；

ii. `cur--`，复写下一个位置。

C++ 算法代码：

```
class Solution
{
public:
    void duplicateZeros(vector<int>& arr)
    {
        // 1. 先找到最后一个数
        int cur = 0, dest = -1, n = arr.size();
        while(cur < n)
        {
            if(arr[cur]) dest++;
            else dest += 2;
            if(dest >= n - 1) break;
            cur++;
        }
        // 2. 处理一下边界情况
        if(dest == n)
        {
            arr[n - 1] = 0;
            cur--; dest -=2;
        }
        // 3. 从后向前完成复写操作
        while(cur >= 0)
        {
            if(arr[cur]) arr[dest--] = arr[cur--];
            else
            {
                arr[dest--] = 0;
                arr[dest--] = 0;
                cur--;
            }
        }
    }
};
```

C++ 代码结果：

Java 算法代码：

```java
class Solution
{
    public void duplicateZeros(int[] arr)
    {
        int cur = 0, dest = -1, n = arr.length;
        // 1. 先找到最后一个需要复写的数
        while(cur < n)
        {
            if(arr[cur] == 0) dest += 2;
            else dest += 1;
            if(dest >= n - 1) break;
            cur++;
        }
        // 2. 处理一下边界情况
        if(dest == n)
        {
            arr[n - 1] = 0;
            cur--; dest -= 2;
        }
        // 3. 从后向前完成复写操作
        while(cur >= 0)
        {
            if(arr[cur] != 0) arr[dest--] = arr[cur--];
            else
            {
                arr[dest--] = 0;
                arr[dest--] = 0;
                cur--;
            }
        }
    }
}
```

Java 代码结果：

3. 快乐数（medium）

1. **题目链接：202. 快乐数**

2. **题目描述：**

> 编写一个算法来判断一个数 `n` 是不是快乐数。
>
> 「快乐数」 定义为：
>
> - 对于一个正整数，每一次将该数替换为它每个位置上的数字的平方和。
> - 然后重复这个过程直到这个数变为 1，也可能是**无限循环**但始终变不到 `1`。
> - 如果这个过程 结果为 `1`，那么这个数就是快乐数。
> - 如果 `n` 是 快乐数 就返回 `true` ；不是，则返回 `false` 。
>
> 示例 1：
>
> 输入：`n = 19`
>
> 输出：`true`
>
> 解释：
>
> ```
> 19   ->   1 * 1 + 9 * 9 = 82
> 82   ->   8 * 8 + 2 * 2 = 68
> 68   ->   6 * 6 + 8 * 8 = 100
> 100 ->   1 * 1 + 0 * 0 + 0 * 0 = 1
> ```
>
> 示例 2：
>
> 输入：`n = 2`
>
> 输出：`false`
>
> 解释：（这里省去计算过程，只列出转换后的数）
>
> ```
> 2 -> 4 -> 16 -> 37 -> 58 -> 89 -> 145 -> 42 -> 20 -> 4 -> 16
> ```
>
> 往后就不必再计算了，因为出现了重复的数字，最后结果肯定不会是`1`

3. **题目分析：**

为了方便叙述，将「对于一个正整数，每一次将该数替换为它每个位置上的数字的平方和」这一个操作记为 `x` 操作；

题目告诉我们，当我们不断重复 `x` 操作的时候，计算一定会「死循环」，死的方式有两种：

- 情况一：一直在 `1` 中死循环，即 `1 -> 1 -> 1 -> 1......`
- 情况二：在历史的数据中死循环，但始终变不到 `1`

由于上述两种情况只会出现一种，因此，只要我们能确定循环是在「情况一」中进行，还是在「情况二」中进行，就能得到结果。

简单证明：

a. 经过一次变化之后的最大值 `9^2 * 10 = 810`（`2^31-1=2147483647`。选一个更大的最大 `9999999999`），也就是变化的区间在 `[1, 810]` 之间；

b. 根据「鸽巢原理」，一个数变化 `811` 次之后，必然会形成一个循环；

c. 因此，变化的过程最终会走到一个圈里面，因此可以用「快慢指针」来解决。

4. **解法（快慢指针）：**

算法思路：

根据上述的题目分析，我们可以知道，当重复执行 `x` 的时候，数据会陷入到一个「循环」之中。

而「快慢指针」有一个特性，就是在一个圆圈中，快指针总是会追上慢指针的，也就是说他们总会相遇在一个位置上。如果相遇位置的值是 `1`，那么这个数一定是快乐数；如果相遇位置不是 `1` 的话，那么就不是快乐数。

补充知识：如何求一个数 n 每个位置上的数字的平方和。

a. 把数 `n` 每一位的数提取出来：

循环迭代下面步骤：

i. `int t = n % 10` 提取个位；

ii. `n /= 10` 干掉个位；

直到 `n` 的值变为 `0`；

b. 提取每一位的时候，用一个变量 `tmp` 记录这一位的平方与之前提取位数的平方和

- `tmp = tmp + t * t`

C++ 算法代码：

```cpp
class Solution
{
public:
    int bitSum(int n) // 返回 n 这个数每一位上的平方和{
        int sum = 0;
        while(n)
        {
            int t = n % 10;
            sum += t * t;
            n /= 10;
        }
        return sum;
    }

    bool isHappy(int n)
    {
        int slow = n, fast = bitSum(n);
        while(slow != fast)
        {
            slow = bitSum(slow);
            fast = bitSum(bitSum(fast));
        }
        return slow == 1;
    }
};
```

C++ 运行结果：

Java 算法代码：

```java
class Solution
{
    public int bitSum(int n) // 返回 n 这个数每一位上的平方和
    {
        int sum = 0;
        while(n != 0)
        {
            int t = n % 10;
            sum += t * t;
```

```
            n /= 10;
        }
        return sum;
    }

    public boolean isHappy(int n)
    {
        int slow = n, fast = bitSum(n);
        while(slow != fast)
        {
            slow = bitSum(slow);
            fast = bitSum(bitSum(fast));
        }
        return slow == 1;
    }
}
```

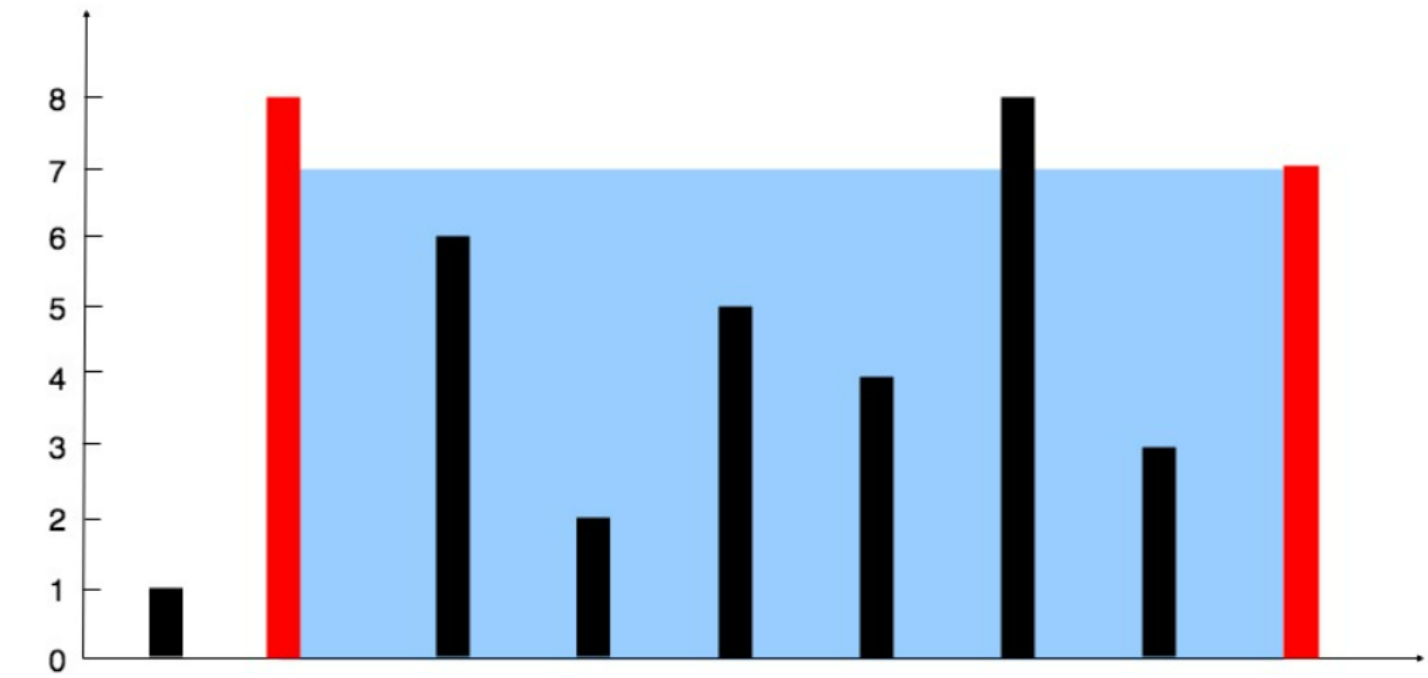
Java 运行结果：

4. 盛水最多的容器（medium）

1. **题目链接：11. 盛最多水的容器**

2. **题目描述：**

> 给定一个长度为 `n` 的整数数组 height。有 `n` 条垂线，第 i 条线的两个端点是 `(i, 0)` 和 `(i, height[i])` 。
>
> 找出其中的两条线，使得它们与 x 轴共同构成的容器可以容纳最多的水。
>
> 返回容器可以储存的最大水量。
>
> 说明：你不能倾斜容器。
>
> 示例 1：
>
> 输入：`[1,8,6,2,5,4,8,3,7]`
>
> 输出：`49`



解释：图中垂直线代表输入数组 `[1,8,6,2,5,4,8,3,7]`。在此情况下，容器能够容纳水（表示为蓝色部分）的最大值为 `49`。

3. 解法一（暴力求解）（会超时）：

算法思路：

枚举出能构成的所有容器，找出其中容积最大的值。

- 容器容积的计算方式：

  设两指针 `i`，`j`，分别指向水槽板的最左端以及最右端，此时容器的宽度为 `j - i`。由于容器的高度由两板中的短板决定，因此可得容积公式：`v = (j - i) * min(height[i], height[j])`

算法代码：

```
class Solution {
public:
    int maxArea(vector<int>& height) {
        int n = height.size();
        int ret = 0;
        // 两层 for 枚举出所有可能出现的情况
        for (int i = 0; i < n; i++) {
            for (int j = i + 1; j < n; j++) {
                // 计算容积，找出最大的那一个
                ret = max(ret, min(height[i], height[j]) * (j - i));
            }
        }
```

```
        return ret;
    }
};
```

4. **解法二（对撞指针）：**

算法思路：

设两个指针 `left`， `right` 分别指向容器的左右两个端点，此时容器的容积：

`v = (right - left) * min( height[right], height[left])`

容器的左边界为 `height[left]`，右边界为 `height[right]`。

为了方便叙述，我们假设「左边边界」小于「右边边界」。

如果此时我们固定一个边界，改变另一个边界，水的容积会有如下变化形式：

- 容器的宽度一定变小。
- 由于左边界较小，决定了水的高度。如果改变左边界，新的水面高度不确定，但是一定不会超过右边的柱子高度，因此容器的容积可能会增大。
- 如果改变右边界，无论右边界移动到哪里，新的水面的高度一定不会超过左边界，也就是不会超过现在的水面高度，但是由于容器的宽度减小，因此容器的容积一定会变小的。

由此可见，左边界和其余边界的组合情况都可以舍去。所以我们可以 `left++` 跳过这个边界，继续去判断下一个左右边界。

当我们不断重复上述过程，每次都可以舍去大量不必要的枚举过程，直到 `left` 与 `right` 相遇。期间产生的所有的容积里面的最大值，就是最终答案。

C++ 算法代码：

```
class Solution
{
public:
    int maxArea(vector<int>& height)
    {
        int left = 0, right = height.size() - 1, ret = 0;
        while(left < right)
        {
            int v = min(height[left], height[right]) * (right - left);
            ret = max(ret, v);
            // 移动指针
```

```
            if(height[left] < height[right]) left++;
            else right--;
        }
        return ret;
    }
};
```

C++ 代码结果：

Java 算法代码：

```java
class Solution
{
    public int maxArea(int[] height)
    {
        int left = 0, right = height.length - 1, ret = 0;
        while(left < right)
        {
            int v = Math.min(height[left], height[right]) * (right - left);
            ret = Math.max(ret, v);
            if(height[left] < height[right]) left++;
            else right--;
        }
        return ret;
    }
}
```

Java 运行结果：

5. 有效三角形的个数（medium）

1. **题目链接：611. 有效三角形的个数**

2. **题目描述：**

> 给定一个包含非负整数的数组 `nums` ，返回其中可以组成三角形三条边的三元组个数。
>
> 示例 1:
>
> 输入: `nums = [2,2,3,4]`
>
> 输出: `3`
>
> 解释:有效的组合是:
>
> `2,3,4` (使用第一个 2)
>
> `2,3,4` (使用第二个 2)
>
> `2,2,3`
>
> 示例 2:
>
> 输入: `nums = [4,2,3,4]`
>
> 输出: `4`
>
> 解释:
>
> `4,2,3`
>
> `4,2,4`
>
> `4,3,4`
>
> `2,3,4`

3. **解法一（暴力求解）（会超时）：**

算法思路：

三层 `for` 循环枚举出所有的三元组，并且判断是否能构成三角形。

虽然说是暴力求解，但是还是想优化一下：

判断三角形的优化：

- 如果能构成三角形，需要满足任意两边之和要大于第三边。但是实际上只需让较小的两条边之和大于第三边即可。
- 因此我们可以先将原数组排序，然后从小到大枚举三元组，一方面省去枚举的数量，另一方面方便判断是否能构成三角形。

算法代码：

```cpp
class Solution {
public:
    int triangleNumber(vector<int>& nums) {
        // 1. 排序
        sort(nums.begin(), nums.end());
        int n = nums.size(), ret = 0;
        // 2. 从小到大枚举所有的三元组
        for (int i = 0; i < n; i++) {
            for (int j = i + 1; j < n; j++) {
                for (int k = j + 1; k < n; k++) {
                    // 当最小的两个边之和大于第三边的时候，统计答案
                    if (nums[i] + nums[j] > nums[k])
                        ret++;
                }
            }
        }
        return ret;
    }
};
```

4. **解法二（排序 + 双指针）：**

算法思路：

先将数组排序。

根据「解法一」中的优化思想，我们可以固定一个「最长边」，然后在比这条边小的有序数组中找出一个二元组，使这个二元组之和大于这个最长边。由于数组是有序的，我们可以利用「对撞指针」来优化。

设最长边枚举到 `i` 位置，区间 `[left, right]` 是 `i` 位置左边的区间（也就是比它小的区间）：

- 如果 `nums[left] + nums[right] > nums[i]`：
  - 说明 `[left, right - 1]` 区间上的所有元素均可以与 `nums[right]` 构成比 `nums[i]` 大的二元组
  - 满足条件的有 `right - left` 种
  - 此时 `right` 位置的元素的所有情况相当于全部考虑完毕，`right--` ，进入下一轮判断
- 如果 `nums[left] + nums[right] <= nums[i]`：
  - 说明 `left` 位置的元素是不可能与 `[left + 1, right]` 位置上的元素构成满足条件的二元组
  - `left` 位置的元素可以舍去，`left++` 进入下轮循环

C++ 算法代码：

```cpp
class Solution
{
public:
    int triangleNumber(vector<int>& nums)
    {
        // 1. 优化
        sort(nums.begin(), nums.end());

        // 2. 利用双指针解决问题
        int ret = 0, n = nums.size();
        for(int i = n - 1; i >= 2; i--) // 先固定最大的数
        {
            // 利用双指针快速统计符合要求的三元组的个数
            int left = 0, right = i - 1;
            while(left < right)
            {
                if(nums[left] + nums[right] > nums[i])
                {
                    ret += right - left;
                    right--;
                }
                else
                {
                    left++;
                }
            }
        }
        return ret;
    }
};
```

C++ 代码结果：

Java 算法代码：

```java
class Solution
{
    public int triangleNumber(int[] nums)
    {
        // 1. 优化：排序
        Arrays.sort(nums);

        // 2. 利用双指针解决问题
        int ret = 0, n = nums.length;
        for(int i = n - 1; i >= 2; i--) // 先固定最大的数
        {
            // 利用双指针快速统计出符合要求的三元组的个数
            int left = 0, right = i - 1;
            while(left < right)
            {
                if(nums[left] + nums[right] > nums[i])
                {
                    ret += right - left;
                    right--;
                }
                else
                {
                    left++;
                }
            }
        }
        return ret;
    }
}
```

Java 运行结果：

6. 和为 s 的两个数字（easy）

1. **题目链接：** **剑指 Offer 57. 和为s的两个数字**

2. **题目描述：**

> 输入一个递增排序的数组和一个数字 `s` ，在数组中查找两个数，使得它们的和正好是 `s` 。如果有多对数字的和等于 `s` ，则输出任意一对即可。

> 示例 1：
>
> 输入：`nums = [2,7,11,15], target = 9`
>
> 输出：`[2,7]` 或者 `[7,2]`

3. 解法一（暴力解法，会超时）：

算法思路：

两层 `for` 循环列出所有两个数字的组合，判断是否等于目标值。

算法流程：

两层 `for` 循环：

- 外层 `for` 循环依次枚举第一个数 `a`；
- 内层 `for` 循环依次枚举第二个数 `b`，让它与 `a` 匹配；

  `ps`：这里有个魔鬼细节：我们挑选第二个数的时候，可以不从第一个数开始选，因为 `a` 前面的数我们都已经在之前考虑过了；因此，我们可以从 `a` 往后的数开始列举。
- 然后将挑选的两个数相加，判断是否符合目标值。

算法代码：

```
class Solution {
public:
    vector<int> twoSum(vector<int>& nums, int target) {
        int n = nums.size();
        for (int i = 0; i < n; i++) { // 第一层循环从前往后列举第一个数
            for (int j = i + 1; j < n; j++) { // 第二层循环从 i 位置之后列举第二个数
                if (nums[i] + nums[j] == target) // 两个数的和等于目标值，说明我们已经找到结果了
                    return {nums[i], nums[j]};
            }
        }
        return {-1, -1};
    }
};
```

4. 解法二（双指针 - 对撞指针）：

算法思路：

注意到本题是升序的数组，因此可以用「对撞指针」优化时间复杂度。

算法流程（附带算法分析，为什么可以使用对撞指针）：

a. 初始化 `left`， `right` 分别指向数组的左右两端（这里不是我们理解的指针，而是数组的下标）

b. 当 `left < right` 的时候，一直循环

i. 当 `nums[left] + nums[right] == target` 时，说明找到结果，记录结果，并且返回；

ii. 当 `nums[left] + nums[right] < target` 时：

- 对于 `nums[left]` 而言，此时 `nums[right]` 相当于是 `nums[left]` 能碰到的最大值（别忘了，这里是升序数组哈~）。如果此时不符合要求，说明在这个数组里面，没有别的数符合 `nums[left]` 的要求了（最大的数都满足不了你，你已经没救了）。因此，我们可以大胆舍去这个数，让 `left++`，去比较下一组数据；
- 那对于 `nums[right]` 而言，由于此时两数之和是小于目标值的， `nums[right]` 还可以选择比 `nums[left]` 大的值继续努力达到目标值，因此 `right` 指针我们按兵不动；

iii. 当 `nums[left] + nums[right] > target` 时，同理我们可以舍去 `nums[right]` （最小的数都满足不了你，你也没救了）。让 `right--`，继续比较下一组数据，而 `left` 指针不变（因为他还是可以去匹配比 `nums[right]` 更小的数的）。

C++ 算法代码：

```
class Solution
{
public:
    vector<int> twoSum(vector<int>& nums, int target)
    {
        int left = 0, right = nums.size() - 1;
        while(left < right)
        {
            int sum = nums[left] + nums[right];
            if(sum > target) right--;
            else if(sum < target) left++;
            else return {nums[left], nums[right]};
        }
        // 照顾编译器
        return {-4941, -1};
```

```
    }
};
```

C++ 运行结果：

Java 算法代码：

```java
class Solution
{
    public int[] twoSum(int[] nums, int target)
    {
        int left = 0, right = nums.length - 1;
        while(left < right)
        {
            int sum = nums[left] + nums[right];
            if(sum > target) right--;
            else if(sum < target) left++;
            else return new int[] {nums[left], nums[right]};
        }
        // 照顾编译器
        return new int[]{0};
    }
}
```

Java 运行结果：

7. 三数之和（medium）

1. **题目链接：15. 三数之和**
2. **题目描述：**

给你一个整数数组 nums ，判断是否存在三元组 [nums[i], nums[j], nums[k]] 满足 i != j、i != k 且 j != k ，同时还满足 nums[i] + nums[j] + nums[k] == 0 。请你返回所有和为 0 且不重复的三元组。

注意：答案中不可以包含重复的三元组。

示例 1：

输入：nums = [-1,0,1,2,-1,-4]

输出：[[-1,-1,2],[-1,0,1]]

解释：

nums[0] + nums[1] + nums[2] = (-1) + 0 + 1 = 0 。

nums[1] + nums[2] + nums[4] = 0 + 1 + (-1) = 0 。

nums[0] + nums[3] + nums[4] = (-1) + 2 + (-1) = 0 。

不同的三元组是 [-1,0,1] 和 [-1,-1,2] 。

注意，输出的顺序和三元组的顺序并不重要。

示例 2：

输入：nums = [0,1,1]

输出：[]

解释：唯一可能的三元组和不为 0 。

示例 3：

输入：nums = [0,0,0]

输出：[[0,0,0]]

解释：唯一可能的三元组和为 0 。

提示：

3 <= nums.length <= 3000

-10^5 <= nums[i] <= 10^5

3. **解法（排序+双指针）：**

算法思路：

本题与两数之和类似，是非常经典的面试题。

与两数之和稍微不同的是，题目中要求找到所有「不重复」的三元组。那我们可以利用在两数之和那里用的双指针思想，来对我们的暴力枚举做优化：

i. 先排序；

ii. 然后固定一个数 `a`：

iii. 在这个数后面的区间内，使用「双指针算法」快速找到两个数之和等于 `-a` 即可。

但是要注意的是，这道题里面需要有「去重」操作~

i. 找到一个结果之后，`left` 和 `right` 指针要「跳过重复」的元素；

ii. 当使用完一次双指针算法之后，固定的 `a` 也要「跳过重复」的元素。

C++ 算法代码：

```
class Solution
{
public:
    vector<vector<int>> threeSum(vector<int>& nums)
    {
        vector<vector<int>> ret;

        // 1. 排序
        sort(nums.begin(), nums.end());

        // 2. 利用双指针解决问题
        int n = nums.size();
        for(int i = 0; i < n; ) // 固定数 a
        {
            if(nums[i] > 0) break; // 小优化
            int left = i + 1, right = n - 1, target = -nums[i];
            while(left < right)
            {
                int sum = nums[left] + nums[right];
                if(sum > target) right--;
                else if(sum < target) left++;
                else
                {
                    ret.push_back({nums[i], nums[left], nums[right]});
                    left++, right--;
                    // 去重操作 left 和 right
                    while(left < right && nums[left] == nums[left - 1]) left++;
                    while(left < right && nums[right] == nums[right + 1])
right--;
```

```
                }
            }
            // 去重 i
            i++;
            while(i < n && nums[i] == nums[i - 1]) i++;
        }
        return ret;
    }
};
```

C++ 代码结果：

Java 算法代码：

```
class Solution
{
    public List<List<Integer>> threeSum(int[] nums)
    {
        List<List<Integer>> ret = new ArrayList<>();

        // 1. 排序
        Arrays.sort(nums);

        // 2. 利用双指针解决问题
        int n = nums.length;
        for(int i = 0; i < n; ) // 固定数 a
        {
            if(nums[i] > 0) break; // 小优化
            int left = i + 1, right = n - 1, target = -nums[i];
            while(left < right)
            {
                int sum = nums[left] + nums[right];
                if(sum > target) right--;
                else if(sum < target) left++;
                else
                {
                    // nums[i]  nums[left]  num[right]
```

```
                    ret.add(new ArrayList<Integer>(Arrays.asList(nums[i], nums[left], nums[right])));
                    left++; right--; // 缩小区间继续寻找
                    // 去重: left right
                    while(left < right && nums[left] == nums[left - 1]) left++;
                    while(left < right && nums[right] == nums[right + 1]) right--;
                }
            }
            // 去重: i
            i++;
            while(i < n && nums[i] == nums[i - 1]) i++;
        }
        return ret;
    }
}
```

Java 运行结果：

8. 四数之和（medium）

1. **题目链接：18. 四数之和**

2. **题目描述：**

> 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
>
> 你可以按 任意顺序 返回答案 。
>
> 示例 1：
>
> 输入：nums = [1,0,-1,0,-2,2], target = 0
>
> 输出：[[-2,-1,1,2],[-2,0,0,2],[-1,0,0,1]]

> 示例 2：
>
> 输入：nums = [2,2,2,2,2], target = 8
>
> 输出：[[2,2,2,2]]
>
> 提示：
>
> 1 <= nums.length <= 200
>
> -109 <= nums[i] <= 109
>
> -109 <= target <= 109

3. **解法（排序 + 双指针）**

算法思路：

a. 依次固定一个数 `a`；

b. 在这个数 `a` 的后面区间上，利用「三数之和」找到三个数，使这三个数的和等于 `target - a` 即可。

C++ 算法代码：

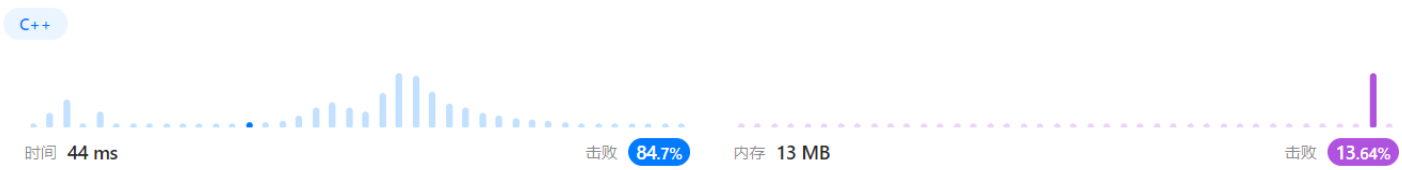
```
class Solution
{
public:
    vector<vector<int>> fourSum(vector<int>& nums, int target)
    {
        vector<vector<int>> ret;

        // 1. 排序
        sort(nums.begin(), nums.end());

        // 2. 利用双指针解决问题
        int n = nums.size();
        for(int i = 0; i < n; ) // 固定数 a
        {
            // 利用 三数之和
            for(int j = i + 1; j < n; ) // 固定数 b
            {
                // 双指针
                int left = j + 1, right = n - 1;
                long long aim = (long long)target - nums[i] - nums[j];
                while(left < right)
                {
                    int sum = nums[left] + nums[right];
```

```
                    if(sum < aim) left++;
                    else if(sum > aim) right--;
                    else
                    {
                        ret.push_back({nums[i], nums[j], nums[left++],
nums[right--]});
                        // 去重一
                        while(left < right && nums[left] == nums[left - 1])
left++;
                        while(left < right && nums[right] == nums[right + 1])
right--;
                    }
                }
                // 去重二
                j++;
                while(j < n && nums[j] == nums[j - 1]) j++;
            }
            // 去重三
            i++;
            while(i < n && nums[i] == nums[i - 1]) i++;
        }
        return ret;
    }
};
```

C++ 运行结果：



Java 算法代码：

```
class Solution
{
    public List<List<Integer>> fourSum(int[] nums, int target)
    {
        List<List<Integer>> ret = new ArrayList<>();

        // 1. 排序
        Arrays.sort(nums);

        // 2. 利用双指针解决问题
```

```java
        int n = nums.length;
        for(int i = 0; i < n; ) // 固定数 a
        {
            // 三数之和
            for(int j = i + 1; j < n; ) // 固定数 b
            {
                // 双指针
                int left = j + 1, right = n - 1;
                long aim = (long)target - nums[i] - nums[j];
                while(left < right)
                {
                    int sum = nums[left] + nums[right];
                    if(sum > aim) right--;
                    else if(sum < aim) left++;
                    else
                    {
                        ret.add(Arrays.asList(nums[i], nums[j], nums[left++], nums[right--]));
                        // 去重一
                        while(left < right && nums[left] == nums[left - 1]) left++;
                        while(left < right && nums[right] == nums[right + 1]) right--;
                    }
                }
                // 去重二
                j++;
                while(j < n && nums[j] == nums[j - 1]) j++;
            }
            // 去重三
            i++;
            while(i < n && nums[i] == nums[i - 1]) i++;
        }
        return ret;
    }
}
```

Java 运行结果：

滑动窗口

9. 长度最小的子数组（medium）

1. **题目链接：209. 长度最小的子数组**

2. **题目描述：**

> 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和 `≥ 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 。
>
> 示例 1：
>
> 输入：`target = 7, nums = [2,3,1,2,4,3]`
>
> 输出：`2`
>
> 解释：
>
> 子数组 `[4,3]` 是该条件下的长度最小的子数组。
>
> 示例 2：
>
> 输入：`target = 4, nums = [1,4,4]`
>
> 输出：`1`
>
> 示例 3：
>
> 输入：`target = 11, nums = [1,1,1,1,1,1,1,1]`
>
> 输出：`0`

3. **解法一（暴力求解）（会超时）：**

算法思路：

「从前往后」枚举数组中的任意一个元素，把它当成起始位置。然后从这个「起始位置」开始，然后寻找一段最短的区间，使得这段区间的和「大于等于」目标值。

将所有元素作为起始位置所得的结果中，找到「最小值」即可。

算法代码：

```
1 class Solution {
2 public:
3     int minSubArrayLen(int target, vector<int>& nums) {
4         // 记录结果
```

```
        int ret = INT_MAX;
        int n = nums.size();

        // 枚举出所有满足和大于等于 target 的子数组[start, end]
        // 由于是取到最小，因此枚举的过程中要尽量让数组的长度最小
        // 枚举开始位置
        for (int start = 0; start < n; start++)
        {
            int sum = 0; // 记录从这个位置开始的连续数组的和
            // 寻找结束位置
            for (int end = start; end < n; end++)
            {
                sum += nums[end]; // 将当前位置加上

                if (sum >= target) // 当这段区间内的和满足条件时
                {
                    // 更新结果，start 开头的最短区间已经找到
                    ret = min(ret, end - start + 1);
                    break;
                }
            }
        }
        // 返回最后结果
        return ret == INT_MAX ? 0 : ret;
    }
};
```

4. 解法二（滑动窗口）：

算法思路：

由于此问题分析的对象是「一段连续的区间」，因此可以考虑「滑动窗口」的思想来解决这道题。

让滑动窗口满足：从 `i` 位置开始，窗口内所有元素的和小于 `target`（那么当窗口内元素之和第一次大于等于目标值的时候，就是 `i` 位置开始，满足条件的最小长度）。

做法：将右端元素划入窗口中，统计出此时窗口内元素的和：

- 如果窗口内元素之和大于等于 `target`：更新结果，并且将左端元素划出去的同时继续判断是否满足条件并更新结果（因为左端元素可能很小，划出去之后依旧满足条件）
- 如果窗口内元素之和不满足条件：`right++`，另下一个元素进入窗口。

相信科学（这也是很多题解以及帖子没告诉你的事情：只给你说怎么做，没给你解释为什么这么做）：

为何滑动窗口可以解决问题，并且时间复杂度更低？

- 这个窗口寻找的是：以当前窗口最左侧元素（记为 `left1`）为基准，符合条件的情况。也就是在这道题中，从 `left1` 开始，满足区间和 `sum >= target` 时的最右侧（记为 `right1`）能到哪里。
- 我们既然已经找到从 `left1` 开始的最优的区间，那么就可以大胆舍去 `left1`。但是如果继续像方法一一样，重新开始统计第二个元素（`left2`）往后的和，势必会有大量重复的计算（因为我们在求第一段区间的时候，已经算出很多元素的和了，这些和是可以在计算下次区间和的时候用上的）。
- 此时，`rigth1` 的作用就体现出来了，我们只需将 `left1` 这个值从 `sum` 中剔除。从 `right1` 这个元素开始，往后找满足 `left2` 元素的区间（此时 `right1` 也有可能是满足的，因为 `left1` 可能很小。`sum` 剔除掉 `left1` 之后，依旧满足大于等于 `target`）。这样我们就能省掉大量重复的计算。
- 这样我们不仅能解决问题，而且效率也会大大提升。

时间复杂度：虽然代码是两层循环，但是我们的 `left` 指针和 `right` 指针都是不回退的，两者最多都往后移动 `n` 次。因此时间复杂度是 `O(N)`。

C++ 算法代码：

```
class Solution
{
public:
    int minSubArrayLen(int target, vector<int>& nums)
    {
        int n = nums.size(), sum = 0, len = INT_MAX;
        for(int left = 0, right = 0; right < n; right++)
        {
            sum += nums[right]; // 进窗口
            while(sum >= target) // 判断
            {
                len = min(len, right - left + 1); // 更新结果
                sum -= nums[left++]; // 出窗口
            }
        }
        return len == INT_MAX ? 0 : len;
    }
};
```

C++ 代码结果：

Java 算法代码：

```java
class Solution
{
    public int minSubArrayLen(int target, int[] nums)
    {
        int n = nums.length, sum = 0, len = Integer.MAX_VALUE;
        for(int left = 0, right = 0; right < n; right++)
        {
            sum += nums[right]; // 进窗口
            while(sum >= target) // 判断
            {
                len = Math.min(len, right - left + 1); // 更新结果
                sum -= nums[left++]; // 出窗口
            }
        }
        return len == Integer.MAX_VALUE ? 0 : len;
    }
}
```

Java 运行结果：

10. 无重复字符的最长子串（medium）

1. **题目链接：3. 无重复字符的最长子串**

2. **题目描述：**

> 给定一个字符串 s ，请你找出其中不含有重复字符的最长子串的长度。
>
> 示例 1:

输入: `s = "abcabcbb"`
输出: `3`
解释: 因为无重复字符的最长子串是 `"abc"`，所以其长度为 `3`。

示例 2:

输入: `s = "bbbbb"`
输出: `1`
解释: 因为无重复字符的最长子串是 `"b"`，所以其长度为 `1`。

示例 3:

输入: `s = "pwwkew"`
输出: `3`
解释: 因为无重复字符的最长子串是 `"wke"`，所以其长度为 `3`。

请注意，你的答案必须是 子串 的长度，`"pwke"` 是一个*子序列*，不是子串。

提示：

- `0 <= s.length <= 5 * 10^4`
- `s` 由英文字母、数字、符号和空格组成 1.

3. 解法一（暴力求解）（不会超时，可以通过）：

算法思路：

枚举「从每一个位置」开始往后，无重复字符的子串可以到达什么位置。找出其中长度最大的即可。

在往后寻找无重复子串能到达的位置时，可以利用「哈希表」统计出字符出现的频次，来判断什么时候子串出现了重复元素。

算法代码：

```
class Solution {
public:
    int lengthOfLongestSubstring(string s) {
        int ret = 0; // 记录结果
        int n = s.length();

        // 1. 枚举从不同位置开始的最长重复子串
        // 枚举起始位置
        for (int i = 0; i < n; i++)
        {
```

```
            // 创建一个哈希表，统计频次
            int hash[128] = { 0 };

            // 寻找结束为止
            for (int j = i; j < n; j++)
            {
                hash[s[j]]++; // 统计字符出现的频次

                if (hash[s[j]] > 1) // 如果出现重复的
                    break;

                // 如果没有重复，就更新 ret
                ret = max(ret, j - i + 1);
            }
        }
        // 2. 返回结果
        return ret;
    }
};
```

4. 解法二（滑动窗口）：

算法思路：

研究的对象依旧是一段连续的区间，因此继续使用「滑动窗口」思想来优化。

让滑动窗口满足：窗口内所有元素都是不重复的。

做法：右端元素 `ch` 进入窗口的时候，哈希表统计这个字符的频次：

- 如果这个字符出现的频次超过 `1`，说明窗口内有重复元素，那么就从左侧开始划出窗口，直到 `ch` 这个元素的频次变为 `1`，然后再更新结果。
- 如果没有超过 `1`，说明当前窗口没有重复元素，可以直接更新结果

C++ 算法代码：

```
class Solution
{
public:
    int lengthOfLongestSubstring(string s)
    {
        int hash[128] = { 0 }; // 使用数组来模拟哈希表
        int left = 0, right = 0, n = s.size();
        int ret = 0;
```

```
        while(right < n)
        {
            hash[s[right]]++; // 进入窗口
            while(hash[s[right]] > 1) // 判断
                hash[s[left++]]--; // 出窗口
            ret = max(ret, right - left + 1); // 更新结果
            right++; // 让下一个元素进入窗口
        }
        return ret;
    }
};
```

C++ 代码结果：

Java 算法代码：

```
class Solution
{
    public int lengthOfLongestSubstring(String ss)
    {
        char[] s = ss.toCharArray();

        int[] hash = new int[128]; // 用数组模拟哈希表
        int left = 0, right = 0, n = ss.length();
        int ret = 0;
        while(right < n)
        {
            hash[s[right]]++; // 进入窗口
            while(hash[s[right]] > 1) // 判断
                hash[s[left++]]--; // 出窗口
            ret = Math.max(ret, right - left + 1); // 更新结果
            right++; // 让下一个字符进入窗口
        }
        return ret;
    }
}
```

Java 运行结果：

11. 最大连续 1 的个数 III（medium）

1. 题目链接：1004. 最大连续 1 的个数 III

2. 题目描述：

> 给定一个二进制数组 `nums` 和一个整数 `k`，如果可以翻转最多 `k` 个 `0` ，则返回 数组中连续 `1` 的最大个数。
>
> 示例 1：
>
> 输入：`nums = [1,1,1,0,0,0,1,1,1,1,0], K = 2`
>
> 输出：`6`
>
> 解释：
>
> `[1,1,1,0,0,1,1,1,1,1,1]`
>
> 红色数字从 `0` 翻转到 `1`，最长的子数组长度为 `6`。
>
> 示例 2：
>
> 输入：`nums = [0,0,1,1,0,0,1,1,1,0,1,1,0,0,0,1,1,1,1], K = 3`
>
> 输出：`10`
>
> 解释：
>
> `[0,0,1,1,1,1,1,1,1,1,1,1,0,0,0,1,1,1,1]`
>
> 红色数字从 `0` 翻转到 `1`，最长的子数组长度为 `10`。

3. 解法（滑动窗口）：

算法思路：

不要去想怎么翻转，不要把问题想的很复杂，这道题的结果无非就是一段连续的 `1` 中间塞了 `k` 个 `0` 嘛。

因此，我们可以把问题转化成：求数组中一段最长的连续区间，要求这段区间内 `0` 的个数不超过 `k` 个。

既然是连续区间，可以考虑使用「滑动窗口」来解决问题。

算法流程：

a. 初始化一个大小为 `2` 的数组就可以当做哈希表 `hash` 了；初始化一些变量 `left = 0`，`right = 0`，`ret = 0`；

b. 当 `right` 小于数组大小的时候，一直下列循环：

i. 让当前元素进入窗口，顺便统计到哈希表中；

ii. 检查 `0` 的个数是否超标：

- 如果超标，依次让左侧元素滑出窗口，顺便更新哈希表的值，直到 `0` 的个数恢复正常；

iii. 程序到这里，说明窗口内元素是符合要求的，更新结果；

iv. `right++`，处理下一个元素；

c. 循环结束后，`ret` 存的就是最终结果。

C++ 算法代码：

```
class Solution
{
public:
    int longestOnes(vector<int>& nums, int k)
    {
        int ret = 0;
        for(int left = 0, right = 0, zero = 0; right < nums.size(); right++)
        {
            if(nums[right] == 0) zero++; // 进窗口
            while(zero > k) // 判断
                if(nums[left++] == 0) zero--; // 出窗口
            ret = max(ret, right - left + 1); // 更新结果
        }
        return ret;
    }
};
```

C++ 代码结果：

Java 算法代码：

```java
class Solution
{
    public int longestOnes(int[] nums, int k)
    {
        int ret = 0;
        for(int left = 0, right = 0, zero = 0; right < nums.length; right++)
        {
            if(nums[right] == 0) zero++; // 进窗口
            while(zero > k) // 判断
                if(nums[left++] == 0) zero--; // 出窗口
            ret = Math.max(ret, right - left + 1); // 更新结果
        }
        return ret;
    }
}
```

Java 运行结果：

12. 将 x 减到 0 的最小操作数 (medium)

1. **题目链接：1658. 将 x 减到 0 的最小操作数**

2. **题目描述：**

> 给你一个整数数组 nums 和一个整数 x 。每一次操作时，你应当移除数组 nums 最左边或最右边的元素，然后从 x 中减去该元素的值。请注意，需要 修改 数组以供接下来的操作使用。
>
> 如果可以将 x 恰好 减到 0 ，返回 最小操作数 ；否则，返回 -1 。

> 示例 1：
>
> 输入：nums = [1,1,4,2,3], x = 5
>
> 输出：2
>
> 解释：最佳解决方案是移除后两个元素，将 x 减到 0 。
>
> 示例 2：
>
> 输入：nums = [5,6,7,8,9], x = 4
>
> 输出：-1
>
> 示例 3：
>
> 输入：nums = [3,2,20,1,1,3], x = 10
>
> 输出：5
>
> 解释：最佳解决方案是移除后三个元素和前两个元素（总共 5 次操作），将 x 减到 0 。
>
> 提示：
>
> 1 <= nums.length <= 10^5
>
> 1 <= nums[i] <= 10^4
>
> 1 <= x <= 10^9

3. **解法（滑动窗口）：**

算法思路：

题目要求的是数组「左端+右端」两段连续的、和为 `x` 的最短数组，信息量稍微多一些，不易理清思路；我们可以转化成求数组内一段连续的、和为 `sum(nums) - x` 的最长数组。此时，就是熟悉的「滑动窗口」问题了。

算法流程：

a. 转化问题：求 `target = sum(nums) - x`。如果 `target < 0`，问题无解；

b. 初始化左右指针 `l = 0`，`r = 0`（滑动窗口区间表示为 `[l, r)`，左右区间是否开闭很重要，必须设定与代码一致），记录当前滑动窗口内数组和的变量 `sum = 0`，记录当前满足条件数组的最大区间长度 `maxLen = -1`；

c. 当 `r` 小于等于数组长度时，一直循环：

i. 如果 `sum < target`，右移右指针，直至变量和大于等于 `target`，或右指针已经移到头；

ii. 如果 `sum > target`，右移左指针，直至变量和小于等于 `target`，或左指针已经移到头；

iii. 如果经过前两步的左右移动使得 `sum == target`，维护满足条件数组的最大长度，并让下个元素进入窗口；

d. 循环结束后，如果 `maxLen` 的值有意义，则计算结果返回；否则，返回 `-1`。

C++ 算法代码：

```
class Solution
{
public:
    int minOperations(vector<int>& nums, int x)
    {
        int sum = 0;
        for(int a : nums) sum += a;
        int target = sum - x;
        // 细节问题
        if(target < 0) return -1;

        int ret = -1;
        for(int left = 0, right = 0, tmp = 0; right < nums.size(); right++)
        {
            tmp += nums[right]; // 进窗口
            while(tmp > target) // 判断
                tmp -= nums[left++]; // 出窗口
            if(tmp == target) // 更新结果
                ret = max(ret, right - left + 1);
        }
        if(ret == -1) return ret;
        else return nums.size() - ret;
    }
};
```

C++ 代码结果：

Java 算法代码：

```java
class Solution
{
    public int minOperations(int[] nums, int x)
    {
        int sum = 0;
        for(int a : nums) sum += a;
        int target = sum - x;
        // 处理细节
        if(target < 0) return -1;

        int ret = -1;
        for(int left = 0, right = 0, tmp = 0; right < nums.length; right++)
        {
            tmp += nums[right]; // 进窗口
            while(tmp > target) // 判断
                tmp -= nums[left++]; // 出窗口
            if(tmp == target)
                ret = Math.max(ret, right - left + 1); // 更新结果
        }
        if(ret == -1) return ret;
        else return nums.length - ret;
    }
}
```

Java 运行结果：

13. 水果成篮（medium）

1. **题目链接：904. 水果成篮**

2. **题目描述：**

> 你正在探访一家农场，农场从左到右种植了一排果树。这些树用一个整数数组 fruits 表示，其中 fruits[i] 是第 i 棵树上的水果 种类 。
>
> 你想要尽可能多地收集水果。然而，农场的主人设定了一些严格的规矩，你必须按照要求采摘水果：
>
> - 你只有两个篮子，并且每个篮子只能装单一类型的水果。每个篮子能够装的水果总量没有限制。
> - 你可以选择任意一棵树开始采摘，你必须从每棵树（包括开始采摘的树）上恰好摘一个水果 。采摘的水果应当符合篮子中的水果类型。每采摘一次，你将会向右移动到下一棵树，并继续采摘。
> - 一旦你走到某棵树前，但水果不符合篮子的水果类型，那么就必须停止采摘。
>
> 给你一个整数数组 fruits ，返回你可以收集的水果的最大数目。
>
> 示例 1：
>
> 输入：fruits = [1,2,1]
>
> 输出：3
>
> 解释：
>
> 可以采摘全部 3 棵树。
>
> 示例 2：
>
> 输入：fruits = [0,1,2,2]
>
> 输出：3
>
> 解释：
>
> 可以采摘 [1,2,2] 这三棵树。
>
> 如果从第一棵树开始采摘，则只能采摘 [0,1] 这两棵树。
>
> 示例 3：
>
> 输入：fruits = [3,3,3,1,2,1,1,2,3,3,4]
>
> 输出：5
>
> 解释：
>
> 可以采摘 [1,2,1,1,2] 这五棵树。

3. **解法（滑动窗口）：**

算法思路：

研究的对象是一段连续的区间，可以使用「滑动窗口」思想来解决问题。

让滑动窗口满足：窗口内水果的种类只有两种。

做法：右端水果进入窗口的时候，用哈希表统计这个水果的频次。这个水果进来后，判断哈希表的大小：

- 如果大小超过 **2**：说明窗口内水果种类超过了两种。那么就从左侧开始依次将水果划出窗口，直到哈希表的大小小于等于 **2**，然后更新结果；
- 如果没有超过 **2**，说明当前窗口内水果的种类不超过两种，直接更新结果 **ret**。

算法流程：

a. 初始化哈希表 **hash** 来统计窗口内水果的种类和数量；

b. 初始化变量：左右指针 **left = 0**，**right = 0**，记录结果的变量 **ret = 0**；

c. 当 **right** 小于数组大小的时候，一直执行下列循环：

i. 将当前水果放入哈希表中；

ii. 判断当前水果进来后，哈希表的大小：

- 如果超过 **2**：
  - 将左侧元素滑出窗口，并且在哈希表中将该元素的频次减一；
  - 如果这个元素的频次减一之后变成了 **0**，就把该元素从哈希表中删除；
  - 重复上述两个过程，直到哈希表中的大小不超过 **2**；

iii. 更新结果 **ret**；

iv. **right++**，让下一个元素进入窗口；

d. 循环结束后，**ret** 存的就是最终结果。

C++ 算法代码（使用容器）：

```
class Solution
{
public:
    int totalFruit(vector<int>& f)
    {
        unordered_map<int, int> hash; // 统计窗口内出现了多少种水果

        int ret = 0;
        for(int left = 0, right = 0; right < f.size(); right++)
        {
            hash[f[right]]++; // 进窗口
            while(hash.size() > 2) // 判断
            {
                // 出窗口
```

```
                hash[f[left]]--;
                if(hash[f[left]] == 0)
                    hash.erase(f[left]);
                left++;
            }
            ret = max(ret, right - left + 1);
        }
        return ret;
    }
};
```

C++ 算法代码（用数组模拟哈希表）：

```
class Solution
{
public:
    int totalFruit(vector<int>& f)
    {
        int hash[100001] = { 0 }; // 统计窗口内出现了多少种水果

        int ret = 0;
        for(int left = 0, right = 0, kinds = 0; right < f.size(); right++)
        {
            if(hash[f[right]] == 0) kinds++; // 维护水果的种类
            hash[f[right]]++; // 进窗口

            while(kinds > 2) // 判断
            {
                // 出窗口
                hash[f[left]]--;
                if(hash[f[left]] == 0) kinds--;
                left++;
            }
            ret = max(ret, right - left + 1);
        }
        return ret;
    }
};
```

C++ 代码结果：

Java 算法代码（使用容器）：

```java
class Solution
{
    public int totalFruit(int[] f)
    {
        Map<Integer, Integer> hash = new HashMap<Integer, Integer>(); // 统计窗口内水果的种类

        int ret = 0;
        for(int left = 0, right = 0; right < f.length; right++)
        {
            int in = f[right];
            hash.put(in, hash.getOrDefault(in, 0) + 1); // 进窗口
            while(hash.size() > 2)
            {
                int out = f[left];
                hash.put(out, hash.get(out) - 1); // 出窗口
                if(hash.get(out) == 0)
                    hash.remove(out);
                left++;
            }
            // 更新结果
            ret = Math.max(ret, right - left + 1);
        }
        return ret;
    }
}
```

Java 算法代码（用数组模拟哈希表）：

```java
class Solution
{
    public int totalFruit(int[] f)
    {
        int n = f.length;
        int[] hash = new int[n + 1]; // 统计窗口内水果的种类

```

```
        int ret = 0;
        for(int left = 0, right = 0, kinds = 0; right < n; right++)
        {
            int in = f[right];
            if(hash[in] == 0) kinds++; // 维护水果种类
            hash[in]++; // 进窗口

            while(kinds > 2) // 判断
            {
                int out = f[left];
                hash[out]--; // 出窗口
                if(hash[out] == 0) kinds--;
                left++;
            }
            // 更新结果
            ret = Math.max(ret, right - left + 1);
        }
        return ret;
    }
}
```

Java 运行结果：

14. 找到字符串中所有字母异位词（medium）

1. **题目链接：438. 找到字符串中所有字母异位词**
2. **题目描述：**

> 给定两个字符串 s 和 p，找到 s 中所有 p 的 异位词 的子串，返回这些子串的起始索引。不考虑答案输出的顺序。
>
> 异位词 指由相同字母重排列形成的字符串（包括相同的字符串）。
>
> 示例 1:
>
> 输入:
>
> s = "cbaebabacd", p = "abc"

> 输出:
>
> [0,6]
>
> 解释:
>
> 起始索引等于 0 的子串是 "cba", 它是 "abc" 的异位词。
>
> 起始索引等于 6 的子串是 "bac", 它是 "abc" 的异位词。
>
> 示例 2:
>
> 输入:
>
> s = "abab", p = "ab"
>
> 输出:
>
> [0,1,2]
>
> 解释:
>
> 起始索引等于 0 的子串是 "ab", 它是 "ab" 的异位词。
>
> 起始索引等于 1 的子串是 "ba", 它是 "ab" 的异位词。
>
> 起始索引等于 2 的子串是 "ab", 它是 "ab" 的异位词。
>
> 提示:
>
> 1 <= s.length, p.length <= 3 * 10^4
>
> s 和 p 仅包含小写字母

3. **解法（滑动窗口 + 哈希表）：**

算法思路：

- 因为字符串 `p` 的异位词的长度一定与字符串 `p` 的长度相同，所以我们可以在字符串 `s` 中构造一个长度为与字符串 `p` 的长度相同的滑动窗口，并在滑动中维护窗口中每种字母的数量；
- 当窗口中每种字母的数量与字符串 `p` 中每种字母的数量相同时，则说明当前窗口为字符串 `p` 的异位词；
- 因此可以用两个大小为 `26` 的数组来模拟哈希表，一个来保存 `s` 中的子串每个字符出现的个数，另一个来保存 `p` 中每一个字符出现的个数。这样就能判断两个串是否是异位词。

C++ 算法代码：

```
```

```cpp
class Solution
{
public:
    vector<int> findAnagrams(string s, string p)
    {
        vector<int> ret;
        int hash1[26] = { 0 }; // 统计字符串 p 中每个字符出现的个数
        for(auto ch : p) hash1[ch - 'a']++;

        int hash2[26] = { 0 }; // 统计窗口里面的每一个字符出现的个数
        int m = p.size();
        for(int left = 0, right = 0, count = 0; right < s.size(); right++)
        {
            char in = s[right];
            // 进窗口 + 维护 count
            if(++hash2[in - 'a'] <= hash1[in - 'a']) count++;
            if(right - left + 1 > m) // 判断
            {
                char out = s[left++];
                // 出窗口 + 维护 count
                if(hash2[out - 'a']-- <= hash1[out - 'a']) count--;
            }
            // 更新结果
            if(count == m) ret.push_back(left);
        }
        return ret;
    }
};
```

C++ 代码结果：

Java 算法代码：

```java
class Solution
{
    public List<Integer> findAnagrams(String ss, String pp)
    {
        List<Integer> ret = new ArrayList<Integer>();
        char[] s = ss.toCharArray();
```

```java
        char[] p = pp.toCharArray();

        int[] hash1 = new int[26]; // 统计字符串 p 中每一个字符出现的个数
        for(char ch : p) hash1[ch - 'a']++;

        int[] hash2 = new int[26]; // 统计窗口中每一个字符出现的个数
        int m = p.length;
        for(int left = 0, right = 0, count = 0; right < s.length; right++)
        {
            char in = s[right];
            // 进窗口 + 维护 count
            if(++hash2[in - 'a'] <= hash1[in - 'a']) count++;
            if(right - left + 1 > m) // 判断
            {
                char out = s[left++];
                // 出窗口 + 维护 count
                if(hash2[out - 'a']-- <= hash1[out - 'a']) count--;
            }
            // 更新结果
            if(count == m) ret.add(left);
        }
        return ret;
    }
}
```

Java 运行结果：

15. 串联所有单词的子串（hard）

1. **题目链接：30. 串联所有单词的子串**

2. **题目描述：**

> 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。

返回所有串联字串在 s 中的开始索引。你可以以 任意顺序 返回答案。

示例 1：

输入：s = "barfoothefoobarman", words = ["foo","bar"]

输出：[0,9]

解释：因为 words.length == 2 同时 words[i].length == 3，连接的子字符串的长度必须为 6。

子串 "barfoo" 开始位置是 0。它是 words 中以 ["bar","foo"] 顺序排列的连接。

子串 "foobar" 开始位置是 9。它是 words 中以 ["foo","bar"] 顺序排列的连接。

输出顺序无关紧要。返回 [9,0] 也是可以的。

示例 2：

输入：s = "wordgoodgoodgoodbestword", words = ["word","good","best","word"]

输出：[]

解释：因为 words.length == 4 并且 words[i].length == 4，所以串联子串的长度必须为 16。

s 中没有子串长度为 16 并且等于 words 的任何顺序排列的连接。

所以我们返回一个空数组。

示例 3：

输入：s = "barfoofoobarthefoobarman", words = ["bar","foo","the"]

输出：[6,9,12]

解释：因为 words.length == 3 并且 words[i].length == 3，所以串联子串的长度必须为 9。

子串 "foobarthe" 开始位置是 6。它是 words 中以 ["foo","bar","the"] 顺序排列的连接。

子串 "barthefoo" 开始位置是 9。它是 words 中以 ["bar","the","foo"] 顺序排列的连接。

子串 "thefoobar" 开始位置是 12。它是 words 中以 ["the","foo","bar"] 顺序排列的连接。

提示：

1 <= s.length <= 104

1 <= words.length <= 5000

1 <= words[i].length <= 30

words[i] 和 s 由小写英文字母组成

3. **解法一（暴力解法）：**

算法思路：

如果我们把每一个单词看成一个一个字母，问题就变成了找到「字符串中所有的字母异位词」。无非就是之前处理的对象是一个一个的字符，我们这里处理的对象是一个一个的单词。

C++ 算法代码：

```
class Solution
{
public:
    vector<int> findSubstring(string s, vector<string>& words)
    {
        vector<int> ret;
        unordered_map<string, int> hash1; // 保存 words 里面所有单词的频次
        for(auto& s : words) hash1[s]++;

        int len = words[0].size(), m = words.size();
        for(int i = 0; i < len; i++) // 执行 len 次
        {
            unordered_map<string, int> hash2; // 维护窗口内单词的频次
            for(int left = i, right = i, count = 0; right + len <= s.size(); right += len)
            {
                // 进窗口 + 维护 count
                string in = s.substr(right, len);
                hash2[in]++;
                if(hash1.count(in) && hash2[in] <= hash1[in]) count++;
                // 判断
                if(right - left + 1 > len * m)
                {
                    // 出窗口 + 维护 count
                    string out = s.substr(left, len);
                    if(hash1.count(out) && hash2[out] <= hash1[out]) count--;
                    hash2[out]--;
                    left += len;
                }
                // 更新结果
                if(count == m) ret.push_back(left);
            }
        }
        return ret;
    }
};
```

C++ 代码结果：

Java 算法代码：

```java
class Solution
{
    public List<Integer> findSubstring(String s, String[] words)
    {
        List<Integer> ret = new ArrayList<Integer>();
        // 保存字典中所有单词的频次
        Map<String, Integer> hash1 = new HashMap<String, Integer>();
        for(String str : words) hash1.put(str, hash1.getOrDefault(str, 0) + 1);

        int len = words[0].length(), m = words.length;
        for(int i = 0; i < len; i++) // 执行次数
        {
            // 保存窗口内所有单词的频次
            Map<String, Integer> hash2 = new HashMap<String, Integer>();
            for(int left = i, right = i, count = 0; right + len <= s.length(); right += len)
            {
                // 进窗口 + 维护 count
                String in = s.substring(right, right + len);
                hash2.put(in, hash2.getOrDefault(in, 0) + 1);
                if(hash2.get(in) <= hash1.getOrDefault(in, 0)) count++;
                // 判断
                if(right - left + 1 > len * m)
                {
                    // 出窗口 + 维护 count
                    String out = s.substring(left, left + len);
                    if(hash2.get(out) <= hash1.getOrDefault(out, 0)) count--;
                    hash2.put(out, hash2.get(out) - 1);
                    left += len;
                }
                // 更新结果
                if(count == m) ret.add(left);
            }
        }
        return ret;
```

```
35      }
36  }
```

Java 运行结果：

16. 最小覆盖子串（hard）

1. **题目链接：76. 最小覆盖子串**

2. **题目描述：**

> 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 "" 。
>
> 注意：
>
> 对于 t 中重复字符，我们寻找的子字符串中该字符数量必须不少于 t 中该字符数量。
>
> 如果 s 中存在这样的子串，我们保证它是唯一的答案。
>
> 示例 1：
>
> 输入：s = "ADOBECODEBANC", t = "ABC"
>
> 输出："BANC"
>
> 解释：
>
> 最小覆盖子串 "BANC" 包含来自字符串 t 的 'A'、'B' 和 'C'。
>
> 示例 2：
>
> 输入：s = "a", t = "a"
>
> 输出："a"
>
> 解释：
>
> 整个字符串 s 是最小覆盖子串。
>
> 示例 3:
>
> 输入: s = "a", t = "aa"
>
> 输出: ""
>
> 解释:

> t 中两个字符 'a' 均应包含在 s 的子串中，
>
> 因此没有符合条件的子字符串，返回空字符串。

3. **解法（滑动窗口 + 哈希表）：**

算法思路：

- 研究对象是连续的区间，因此可以尝试使用滑动窗口的思想来解决。
- **如何判断当前窗口内的所有字符是符合要求的呢？**

  我们可以使用两个哈希表，其中一个将目标串的信息统计起来，另一个哈希表动态的维护窗口内字符串的信息。

  当动态哈希表中包含目标串中所有的字符，并且对应的个数都不小于目标串的哈希表中各个字符的个数，那么当前的窗口就是一种可行的方案。

算法流程：

a. 定义两个全局的哈希表：`1` 号哈希表 `hash1` 用来记录子串的信息，`2` 号哈希表 `hash2` 用来记录目标串 `t` 的信息；

b. 实现一个接口函数，判断当前窗口是否满足要求：

   i. 遍历两个哈希表中对应位置的元素：

      - 如果 `t` 中某个字符的数量大于窗口中字符的数量，也就是 `2` 号哈希表某个位置大于 `1` 号哈希表。说明不匹配，返回 `false`；
      - 如果全都匹配，返回 `true`。

主函数中：

a. 先将 `t` 的信息放入 `2` 号哈希表中；

b. 初始化一些变量：左右指针：`left = 0, right = 0`；目标子串的长度：`len = INT_MAX`；目标子串的起始位置：`retleft`；（通过目标子串的起始位置和长度，我们就能找到结果）

c. 当 `right` 小于字符串 `s` 的长度时，一直下列循环：

   i. 将当前遍历到的元素扔进 `1` 号哈希表中；

   ii. 检测当前窗口是否满足条件：

      - 如果满足条件：
        - 判断当前窗口是否变小。如果变小：更新长度 `len`，以及字符串的起始位置 `retleft`；
        - 判断完毕后，将左侧元素滑出窗口，顺便更新 `1` 号哈希表；

- 重复上面两个过程，直到窗口不满足条件；

iii. `right++`，遍历下一个元素；

d. 判断 `len` 的长度是否等于 `INT_MAX`：

i. 如果相等，说明没有匹配，返回空串；

ii. 如果不想等，说明匹配，返回 `s` 中从 `retleft` 位置往后 `len` 长度的字符串。

C++ 算法代码：

```
class Solution
{
public:
    string minWindow(string s, string t)
    {
        int hash1[128] = { 0 }; // 统计字符串 t 中每一个字符的频次
        int kinds = 0; // 统计有效字符有多少种
        for(auto ch : t)
            if(hash1[ch]++ == 0) kinds++;
        int hash2[128] = { 0 }; // 统计窗口内每个字符的频次

        int minlen = INT_MAX, begin = -1;
        for(int left = 0, right = 0, count = 0; right < s.size(); right++)
        {
            char in = s[right];
            if(++hash2[in] == hash1[in]) count++; // 进窗口 + 维护 count
            while(count == kinds) // 判断条件
            {
                if(right - left + 1 < minlen) // 更新结果
                {
                    minlen = right - left + 1;
                    begin = left;
                }
                char out = s[left++];
                if(hash2[out]-- == hash1[out]) count--; // 出窗口 + 维护 count
            }
        }
        if(begin == -1) return "";
        else return s.substr(begin, minlen);
    }
};

```

C++ 代码结果：

Java 算法代码：

```java
class Solution {
    public String minWindow(String ss, String tt) {
        char[] s = ss.toCharArray();
        char[] t = tt.toCharArray();

        int[] hash1 = new int[128]; // 统计字符串 t 中每一个字符的频次
        int kinds = 0; // 统计有效字符有多少种
        for(char ch : t)
            if(hash1[ch]++ == 0) kinds++;
        int[] hash2 = new int[128]; // 统计窗口内每个字符的频次

        int minlen = Integer.MAX_VALUE, begin = -1;
        for(int left = 0, right = 0, count = 0; right < s.length; right++)
        {
            char in = s[right];
            if(++hash2[in] == hash1[in]) count++; // 进窗口 + 维护 count
            while(count == kinds) // 判断条件
            {
                if(right - left + 1 < minlen) // 更新结果
                {
                    minlen = right - left + 1;
                    begin = left;
                }
                char out = s[left++];
                if(hash2[out]-- == hash1[out]) count--; // 出窗口 + 维护 count
            }
        }
        if(begin == -1) return new String();
        else return ss.substring(begin, begin + minlen);
    }
}
```

Java 运行结果：

二分

17. 二分查找（easy）

1. **题目链接：leetcode 704.二分查找**

2. **题目描述：**

> 给定一个 n 个元素有序的（升序）整型数组 nums 和一个目标值 target ，写一个函数搜索 nums 中的 target，如果目标值存在返回下标，否则返回 -1。
>
> 示例 1:
>
> 输入: nums = [-1,0,3,5,9,12], target = 9
>
> 输出: 4
>
> 解释: 9 出现在 nums 中并且下标为 4
>
> 示例 2:
>
> 输入: nums = [-1,0,3,5,9,12], target = 2
>
> 输出: -1
>
> 解释: 2 不存在 nums 中因此返回 -1
>
> 提示：
>
> 你可以假设 nums 中的所有元素是不重复的。
>
> n 将在 [1, 10000]之间。
>
> nums 的每个元素都将在 [-9999, 9999]之间。

3. **算法流程：**

   a. 定义 `left`， `right` 指针，分别指向数组的左右区间。

   b. 找到待查找区间的中间点 `mid` ，找到之后分三种情况讨论：

      i. `arr[mid] == target` 说明正好找到，返回 `mid` 的值；

ii. `arr[mid] > target` 说明 `[mid, right]` 这段区间都是大于 `target` 的，因此舍去右边区间，在左边 `[left, mid -1]` 的区间继续查找，即让 `right = mid - 1`，然后重复 `2` 过程；

iii. `arr[mid] < target` 说明 `[left, mid]` 这段区间的值都是小于 `target` 的，因此舍去左边区间，在右边 `[mid + 1, right]` 区间继续查找，即让 `left = mid + 1`，然后重复 `2` 过程；

c. 当 `left` 与 `right` 错开时，说明整个区间都没有这个数，返回 `-1`。

算法代码：

```c
int search(int* nums, int numsSize, int target)
{
    // 初始化 left 与 right 指针
    int left = 0, right = numsSize - 1;
    // 由于两个指针相交时，当前元素还未判断，因此需要取等号
    while (left <= right)
    {
        // 先找到区间的中间元素
        int mid = left + (right - left) / 2;
        // 分三种情况讨论
        if (nums[mid] == target) return mid;
        else if (nums[mid] > target) right = mid - 1;
        else left = mid + 1;
    }
    // 如果程序走到这里，说明没有找到目标值，返回 -1
    return -1;
}
```

18. 在排序数组中查找元素的第一个和最后一个位置（medium）

1. **题目链接：** **Leetcode 34.在排序数组中查找元素的第一个和最后一个位置**

2. **题目描述：**

> 给你一个按照非递减顺序排列的整数数组 nums，和一个目标值 target。请你找出给定目标值在数组中的开始位置和结束位置。
>
> 如果数组中不存在目标值 target，返回 [-1, -1]。
>
> 你必须设计并实现时间复杂度为 O(log n) 的算法解决此问题。

> 示例 1：
>
> 输入：nums = [5,7,7,8,8,10], target = 8
>
> 输出：[3,4]
>
> 示例 2：
>
> 输入：nums = [5,7,7,8,8,10], target = 6
>
> 输出：[-1,-1]
>
> 示例 3：
>
> 输入：nums = [], target = 0
>
> 输出：[-1,-1]
>
> 提示：
>
> 0 <= nums.length <= 105
>
> -109 <= nums[i] <= 109
>
> nums 是一个非递减数组
>
> -109 <= target <= 109

3. **算法思路：**

   用的还是二分思想，就是根据数据的性质，在某种判断条件下将区间一分为二，然后舍去其中一个区间，然后再另一个区间内查找；

   方便叙述，用 `x` 表示该元素， `resLeft` 表示左边界， `resRight` 表示右边界。

寻找左边界思路：

- 寻找左边界：
  - 我们注意到以左边界划分的两个区间的特点：
    - 左边区间 `[left, resLeft - 1]` 都是小于 `x` 的；
    - 右边区间（包括左边界） `[resLeft, right]` 都是大于等于 `x` 的；
- 因此，关于 `mid` 的落点，我们可以分为下面两种情况：
  - 当我们的 `mid` 落在 `[left, resLeft - 1]` 区间的时候，也就是 `arr[mid] < target` 。说明 `[left, mid]` 都是可以舍去的，此时更新 `left` 到 `mid + 1` 的位置，继续在 `[mid + 1, right]` 上寻找左边界；
  - 当 `mid` 落在 `[resLeft, right]` 的区间的时候，也就是 `arr[mid] >= target` 。说明 `[mid + 1, right]` （因为 `mid` 可能是最终结果，不能舍去）是可以舍去的，此时更新 `right` 到 `mid` 的位置，继续在 `[left, mid]` 上寻找左边界；

- 由此，就可以通过二分，来快速寻找左边界；

注意：这里找中间元素需要**向下取整**。

因为后续移动左右指针的时候：

- 左指针：`left = mid + 1`，是会向后移动的，因此区间是会缩小的；
- 右指针：`right = mid`，可能会原地踏步（比如：如果向上取整的话，如果剩下 `1,2` 两个元素，`left == 1`，`right == 2`，`mid == 2`。更新区间之后，`left，right，mid` 的值没有改变，就会陷入死循环）。

因此一定要注意，当 `right = mid` 的时候，要向下取整。

寻找右边界思路：

- 寻右左边界：
  - 用 `resRight` 表示右边界；
  - 我们注意到右边界的特点：
    - 左边区间（包括右边界）`[left, resRight]` 都是小于等于 `x` 的；
    - 右边区间 `[resRight+ 1, right]` 都是大于 `x` 的；
- 因此，关于 `mid` 的落点，我们可以分为下面两种情况：
  - 当我们的 `mid` 落在 `[left, resRight]` 区间的时候，说明 `[left, mid - 1]`（`mid` 不可以舍去，因为有可能是最终结果）都是可以舍去的，此时更新 `left` 到 `mid` 的位置；
  - 当 `mid` 落在 `[resRight+ 1, right]` 的区间的时候，说明 `[mid, right]` 内的元素是可以舍去的，此时更新 `right` 到 `mid - 1` 的位置；
- 由此，就可以通过二分，来快速寻找右边界；

注意：这里找中间元素需要向上取整。

因为后续移动左右指针的时候：

- 左指针：`left = mid`，可能会原地踏步（比如：如果向下取整的话，如果剩下 `1,2` 两个元素，`left == 1， right == 2，mid == 1`。更新区间之后，`left，right，mid` 的值没有改变，就会陷入死循环）。
- 右指针：`right = mid - 1`，是会向前移动的，因此区间是会缩小的；

因此一定要注意，当 `right = mid` 的时候，要向下取整。

二分查找算法总结：

请大家一定不要觉得背下模板就能解决所有二分问题。二分问题最重要的就是要分析题意，然后确定要搜索的区间，根据分析问题来写出二分查找算法的代码。

> 要分析题意，确定搜索区间，不要死记模板，不要看左闭右开什么乱七八糟的题解
>
> 要分析题意，确定搜索区间，不要死记模板，不要看左闭右开什么乱七八糟的题解
>
> 要分析题意，确定搜索区间，不要死记模板，不要看左闭右开什么乱七八糟的题解

重要的事情说三遍。

模板记忆技巧：

1. 关于什么时候用三段式，还是二段式中的某一个，一定不要强行去用，而是通过具体的问题分析情况，根据查找区间的变化确定指针的转移过程，从而选择一个模板。
2. 当选择两段式的模板时：
   - 在求 `mid` 的时候，只有 `right - 1` 的情况下，才会向上取整（也就是 `+1` 取中间数）

C++ 算法代码：

```cpp
class Solution
{
public:
    vector<int> searchRange(vector<int>& nums, int target)
    {
        // 处理边界情况
        if(nums.size() == 0) return {-1, -1};

        int begin = 0;
        // 1. 二分左端点
        int left = 0, right = nums.size() - 1;
        while(left < right)
        {
            int mid = left + (right - left) / 2;
            if(nums[mid] < target) left = mid + 1;
            else right = mid;
        }
        // 判断是否有结果
        if(nums[left] != target) return {-1, -1};
        else begin = left; // 标记一下左端点

        // 2. 二分右端点
        left = 0, right = nums.size() - 1;
        while(left < right)
        {
            int mid = left + (right - left + 1) / 2;
```

```
            if(nums[mid] <= target) left = mid;
            else right = mid - 1;
        }
        return {begin, right};
    }
};
```

C++ 代码结果：

Java 算法代码：

```java
class Solution
{
    public int[] searchRange(int[] nums, int target)
    {
        int[] ret = new int[2];
        ret[0] = ret[1] = -1;
        // 处理边界情况
        if(nums.length == 0) return ret;

        // 1. 二分左端点
        int left = 0, right = nums.length - 1;
        while(left < right)
        {
            int mid = left + (right - left) / 2;
            if(nums[mid] < target) left = mid + 1;
            else right = mid;
        }
        // 判断是否有结果
        if(nums[left] != target) return ret;
        else ret[0] = right;

        // 2. 二分右端点
        left = 0; right = nums.length - 1;
        while(left < right)
        {
            int mid = left + (right - left + 1) / 2;
            if(nums[mid] <= target) left = mid;
```

```
28                 else right = mid - 1;
29             }
30             ret[1] = left;
31             return ret;
32         }
33 }
34
```

Java 运行结果：

19. 搜索插入位置（easy）

1. **题目链接：35. 搜索插入位置**

2. **题目描述：**

> 给定一个排序数组和一个目标值，在数组中找到目标值，并返回其索引。如果目标值不存在于数组中，返回它将会被按顺序插入的位置。
>
> 请必须使用时间复杂度为 `O(log n)` 的算法。
>
> 示例 1:
>
> 输入: `nums = [1,3,5,6], target = 5`
>
> 输出: `2`
>
> 示例 2:
>
> 输入: `nums = [1,3,5,6], target = 2`
>
> 输出: `1`
>
> 示例 3:
>
> 输入: `nums = [1,3,5,6], target = 7`
>
> 输出: `4`

3. **解法（二分查找算法）：**

算法思路：

a. 分析插入位置左右两侧区间上元素的特点：

设插入位置的坐标为 `index`，根据插入位置的特点可以知道：

- `[left, index - 1]` 内的所有元素均是小于 `target` 的；
- `[index, right]` 内的所有元素均是大于等于 `target` 的。

b. 设 `left` 为本轮查询的左边界，`right` 为本轮查询的右边界。根据 `mid` 位置元素的信息，分析下一轮查询的区间：

- 当 `nums[mid] >= target` 时，说明 `mid` 落在了 `[index, right]` 区间上，`mid` 左边包括 `mid` 本身，可能是最终结果，所以我们接下来查找的区间在 `[left, mid]` 上。因此，更新 `right` 到 `mid` 位置，继续查找。
- 当 `nums[mid] < target` 时，说明 `mid` 落在了 `[left, index - 1]` 区间上，`mid` 右边但不包括 `mid` 本身，可能是最终结果，所以我们接下来查找的区间在 `[mid + 1, right]` 上。因此，更新 `left` 到 `mid + 1` 的位置，继续查找。

c. 直到我们的查找区间的长度变为 `1`，也就是 `left == right` 的时候，`left` 或者 `right` 所在的位置就是我们要找的结果。

C++ 算法代码：

```cpp
class Solution
{
public:
    int searchInsert(vector<int>& nums, int target)
    {
        int left = 0, right = nums.size() - 1;
        while(left < right)
        {
            int mid = left + (right - left) / 2;
            if(nums[mid] < target) left = mid + 1;
            else right = mid;
        }
        if(nums[left] < target) return right + 1;
        return right;
    }
};
```

C++ 代码结果：

Java 算法代码：

```java
class Solution
{
    public int searchInsert(int[] nums, int target)
    {
        int left = 0, right = nums.length - 1;
        while(left < right)
        {
            int mid = left + (right - left) / 2;
            if(nums[mid] < target) left = mid + 1;
            else right = mid;
        }
        // 特判一下第三种情况
        if(nums[right] < target) return right + 1;
        return right;
    }
}
```

Java 运行结果：

20. x 的平方根（easy）

1. **题目链接：69. x 的平方根**

2. **题目描述：**

> 给你一个非负整数 `x` ，计算并返回 `x` 的 算术平方根 。
>
> 由于返回类型是整数，结果只保留 整数部分 ，小数部分将被 舍去 。
>
> 注意：不允许使用任何内置指数函数和算符，例如 `pow(x, 0.5)` 或者 `x ** 0.5` 。
>
> 示例 1：
>
> 输入：`x = 4`
>
> 输出：`2`
>
> 示例 2：

> 输入：`x = 8`
>
> 输出：`2`
>
> 解释：
>
> `8` 的算术平方根是 `2.82842...`, 由于返回类型是整数，小数部分将被舍去。

3. 解法一（暴力查找）：

算法思路：

依次枚举 `[0, x]` 之间的所有数 `i`：

（这里没有必要研究是否枚举到 `x / 2` 还是 `x / 2 + 1`。因为我们找到结果之后直接就返回了，往后的情况就不会再判断。反而研究枚举区间，既耽误时间，又可能出错）

- 如果 `i * i == x`，直接返回 `x`；
- 如果 `i * i > x`，说明之前的一个数是结果，返回 `i - 1`。

由于 `i * i` 可能超过 `int` 的最大值，因此使用 `long long` 类型。

算法代码：

```cpp
class Solution {
public:
    int mySqrt(int x) {
        // 由于两个较大的数相乘可能会超过 int 最大范围
        // 因此用 long long
        long long i = 0;
        for (i = 0; i <= x; i++)
        {
            // 如果两个数相乘正好等于 x，直接返回 i
            if (i * i == x) return i;
            // 如果第一次出现两个数相乘大于 x，说明结果是前一个数
            if (i * i > x) return i - 1;
        }

        // 为了处理oj题需要控制所有路径都有返回值
        return -1;
    }
};
```

4. 解法二（二分查找算法）：

算法思路：

设 `x` 的平方根的最终结果为 `index`：

a. 分析 `index` 左右两次数据的特点：

- `[0, index]` 之间的元素，平方之后都是小于等于 `x` 的；
- `[index + 1, x]` 之间的元素，平方之后都是大于 `x` 的。

因此可以使用二分查找算法。

C++ 算法代码：

```cpp
class Solution
{
public:
    int mySqrt(int x)
    {
        if(x < 1) return 0; // 处理边界情况
        int left = 1, right = x;
        while(left < right)
        {
            long long mid = left + (right - left + 1) / 2; // 防溢出
            if(mid * mid <= x) left = mid;
            else right = mid - 1;
        }
        return left;
    }
};
```

C++ 代码结果：

Java 算法代码：

```java
class Solution
{
    public int mySqrt(int x)
```

```
4       {
5           // 细节
6           if(x < 1) return 0;
7           long left = 1, right = x;
8           while(left < right)
9           {
10              long mid = left + (right - left + 1) / 2;
11              if(mid * mid <= x) left = mid;
12              else right = mid - 1;
13          }
14          return (int)left;
15      }
16  }
```

Java 运行结果：

21. 山峰数组的峰顶（easy）

1. **题目链接：852. 山脉数组的峰顶索引**

2. **题目描述：**

> 符合下列属性的数组 `arr` 称为 山脉数组：
>
> - `arr.length >= 3`
> - 存在 `i (0 < i < arr.length - 1)` 使得：
>   - `arr[0] < arr[1] < ... arr[i-1] < arr[i]`
>   - `arr[i] > arr[i+1] > ... > arr[arr.length - 1]`
>
> 给你由整数组成的山脉数组 `arr` ，返回任何满足 `arr[0] < arr[1] < ... arr[i - 1] < arr[i] > arr[i + 1] > ... > arr[arr.length - 1]` 的下标 `i` 。
>
> 示例 1：
>
> 输入：`arr = [0,1,0]`
>
> 输出：`1`
>
> 示例 2：
>
> 输入：`arr = [0,2,1,0]`

> 输出：`1`
>
> 示例 3：
>
> 输入：`arr = [24,69,100,99,79,78,67,36,26,19]`
> 输出：`2`

3. 解法一（暴力查找）：

算法思路：

峰顶的特点：比两侧的元素都要大。

因此，我们可以遍历数组内的每一个元素，找到某一个元素比两边的元素大即可。

算法代码：

```
class Solution {
public:
    int peakIndexInMountainArray(vector<int>& arr) {
        int n = arr.size();

        // 遍历数组内每一个元素，直到找到峰顶
        for (int i = 1; i < n - 1; i++)
            // 峰顶满足的条件
            if (arr[i] > arr[i - 1] && arr[i] > arr[i + 1])
                return i;

        // 为了处理 oj 需要控制所有路径都有返回值
        return -1;
    }
};
```

4. 解法二（二分查找）：

算法思路：

1. 分析峰顶位置的数据特点，以及山峰两旁的数据的特点：
   - 峰顶数据特点：`arr[i] > arr[i - 1] && arr[i] > arr[i + 1]`；
   - 峰顶左边的数据特点：`arr[i] > arr[i - 1] && arr[i] < arr[i + 1]`，也就是呈现上升趋势；

- 峰顶右边数据的特点：`arr[i] < arr[i - 1] && arr[i] > arr[i + 1]`，也就是呈现下降趋势。

2. 因此，根据 `mid` 位置的信息，我们可以分为下面三种情况：
   - 如果 `mid` 位置呈现上升趋势，说明我们接下来要在 `[mid + 1, right]` 区间继续搜索；
   - 如果 `mid` 位置呈现下降趋势，说明我们接下来要在 `[left, mid - 1]` 区间搜索；
   - 如果 `mid` 位置就是山峰，直接返回结果。

C++ 算法代码：

```cpp
class Solution
{
public:
    int peakIndexInMountainArray(vector<int>& arr)
    {
        int left = 1, right = arr.size() - 2;
        while(left < right)
        {
            int mid = left + (right - left + 1) / 2;
            if(arr[mid] > arr[mid - 1]) left = mid;
            else right = mid - 1;
        }
        return left;
    }
};
```

C++ 代码结果：

Java 算法代码：

```java
class Solution
{
    public int peakIndexInMountainArray(int[] arr)
```

```
    {
        int left = 1, right = arr.length - 2;
        while(left < right)
        {
            int mid = left + (right - left + 1) / 2;
            if(arr[mid] > arr[mid - 1]) left = mid;
            else right = mid - 1;
        }
        return left;
    }
}
```

Java 运行结果：

22. 寻找峰值（medium）

1. **题目链接：162. 寻找峰值**

2. **题目描述：**

> 峰值元素是指其值严格大于左右相邻值的元素。
>
> 给你一个整数数组 nums，找到峰值元素并返回其索引。数组可能包含多个峰值，在这种情况下，返回 任何一个峰值 所在位置即可。
>
> 你可以假设 nums[-1] = nums[n] = -∞ 。
>
> 你必须实现时间复杂度为 O(log n) 的算法来解决此问题。
>
> 示例 1：
>
> 输入：nums = [1,2,3,1]
>
> 输出：2
>
> 解释：3 是峰值元素，你的函数应该返回其索引 2。
>
> 示例 2：
>
> 输入：nums = [1,2,1,3,5,6,4]
>
> 输出：1 或 5

> 解释：你的函数可以返回索引 1，其峰值元素为 2；
>
> 或者返回索引 5， 其峰值元素为 6。
>
> 提示：
>
> 1 <= nums.length <= 1000
>
> -231 <= nums[i] <= 231 - 1
>
> 对于所有有效的 i 都有 nums[i] != nums[i + 1]

3. **解法二（二分查找算法）：**

算法思路：

寻找二段性：

任取一个点 `i`，与下一个点 `i + 1`，会有如下两种情况：

- `arr[i] > arr[i + 1]`：此时「左侧区域」一定会存在山峰（因为最左侧是负无穷），那么我们可以去左侧去寻找结果；
- `arr[i] < arr[i + 1]`：此时「右侧区域」一定会存在山峰（因为最右侧是负无穷），那么我们可以去右侧去寻找结果。

当我们找到「二段性」的时候，就可以尝试用「二分查找」算法来解决问题。

C++ 算法代码：

```cpp
class Solution
{
public:
    int peakIndexInMountainArray(vector<int>& arr)
    {
        int left = 1, right = arr.size() - 2;
        while(left < right)
        {
            int mid = left + (right - left + 1) / 2;
            if(arr[mid] > arr[mid - 1]) left = mid;
            else right = mid - 1;
        }
        return left;
    }
};
```

C++ 运行结果：

Java 算法代码：

```java
class Solution
{
    public int peakIndexInMountainArray(int[] arr)
    {
        int left = 1, right = arr.length - 2;
        while(left < right)
        {
            int mid = left + (right - left + 1) / 2;
            if(arr[mid] > arr[mid - 1]) left = mid;
            else right = mid - 1;
        }
        return left;
    }
}
```

Java 运行结果：

23. 搜索旋转排序数组中的最小值（medium）

1. **题目链接：153. 寻找旋转排序数组中的最小值**

2. **题目描述：**

> 整数数组 `nums` 按升序排列，数组中的值 互不相同 。
>
> 在传递给函数之前， `nums` 在预先未知的某个下标 `k（0 <= k < nums.length）` 上进行了 旋转，使数组变为 `[nums[k], nums[k+1], ..., nums[n-1], nums[0], nums[1], ..., nums[k-1]]` （下标 从 `0` 开始 计数）。例如， `[0,1,2,4,5,6,7]` 在下标 `3` 处经旋转后可能变为 `[4,5,6,7,0,1,2]` 。

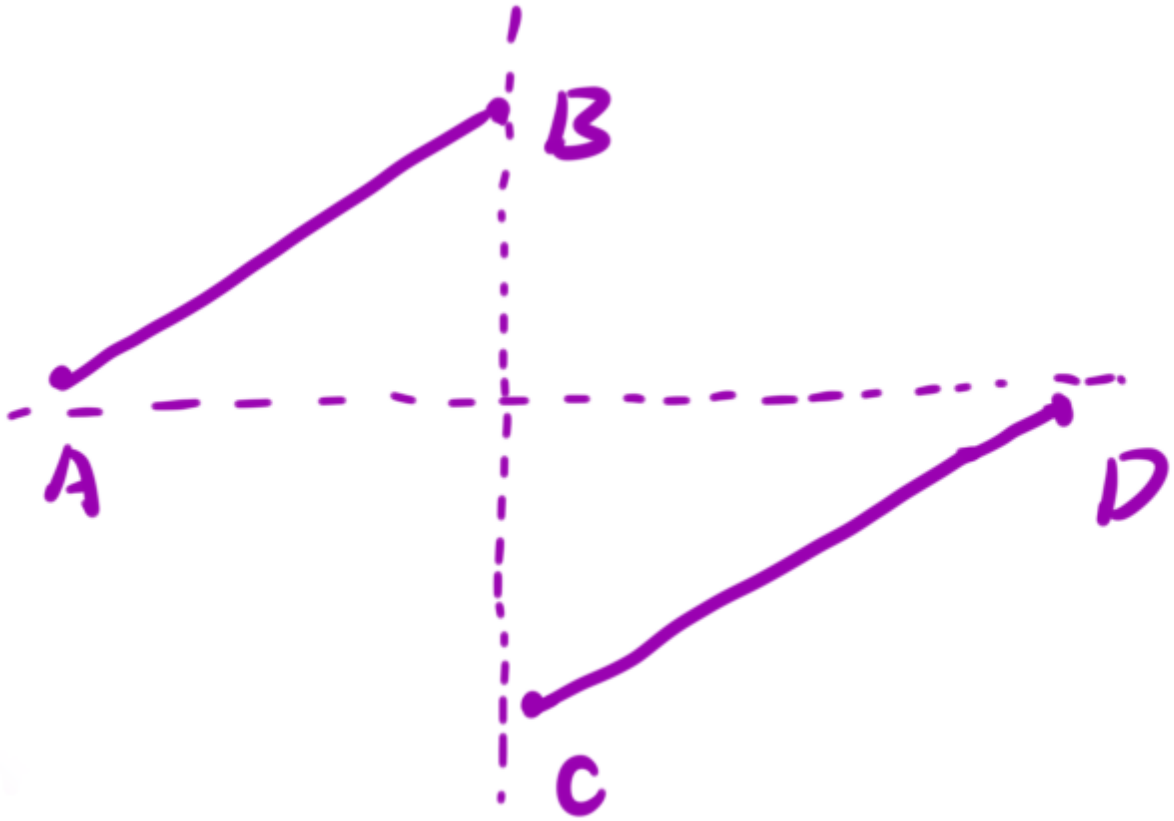
> 给你 旋转后 的数组 `nums` 和一个整数 `target` ，如果 `nums` 中存在这个目标值 `target` ，则返回它的下标，否则返回 `-1` 。
>
> 你必须设计一个时间复杂度为 `O(log n)` 的算法解决此问题。
>
> 示例 1：
>
> 输入：`nums = [4,5,6,7,0,1,2], target = 0`
>
> 输出：`4`
>
> 示例 2：
>
> 输入：`nums = [4,5,6,7,0,1,2], target = 3`
>
> 输出：`-1`
>
> 示例 3：
>
> 输入：`nums = [1], target = 0`
>
> 输出：`-1`

关于暴力查找，只需遍历一遍数组，这里不再赘述。

3. **解法（二分查找）：**

算法思路：

题目中的数组规则如下图所示：



其中 `C` 点就是我们要求的点。

二分的本质：找到一个判断标准，使得查找区间能够一分为二。

通过图像我们可以发现，`[A，B]` 区间内的点都是严格大于 `D` 点的值的，`C` 点的值是严格小于 `D` 点的值的。但是当 `[C，D]` 区间只有一个元素的时候，`C` 点的值是可能等于 `D` 点的值的。

因此，初始化左右两个指针 `left`，`right`：

然后根据 `mid` 的落点，我们可以这样划分下一次查询的区间：

- 当 `mid` 在 `[A，B]` 区间的时候，也就是 `mid` 位置的值严格大于 `D` 点的值，下一次查询区间在 `[mid + 1，right]` 上；
- 当 `mid` 在 `[C，D]` 区间的时候，也就是 `mid` 位置的值严格小于等于 `D` 点的值，下次查询区间在 `[left，mid]` 上。

当区间长度变成 `1` 的时候，就是我们要找的结果。

C++ 算法代码：

```cpp
class Solution
{
public:
    int findMin(vector<int>& nums)
    {
        int left = 0, right = nums.size() - 1;
        int x = nums[right]; // 标记一下最后一个位置的值
        while(left < right)
        {
            int mid = left + (right - left) / 2;
            if(nums[mid] > x) left = mid + 1;
            else right = mid;
        }
        return nums[left];
    }
};
```

C++ 运行结果：

Java 算法代码：

```java
class Solution {
    public int findMin(int[] nums) {
        int left = 0, right = nums.length - 1;
        int x = nums[right]; // 标记一下最后一个位置的值
        while(left < right)
        {
            int mid = left + (right - left) / 2;
            if(nums[mid] > x) left = mid + 1;
            else right = mid;
        }
        return nums[left];
    }
}
```

Java 运行结果：

24. 0～n-1 中缺失的数字（easy）

1. **题目链接：剑指 Offer 53 - II. 0～n-1中缺失的数字**

2. **题目描述：**

> 一个长度为n-1的递增排序数组中的所有数字都是唯一的，并且每个数字都在范围0～n-1之内。在范围0～n-1内的n个数字中有且只有一个数字不在该数组中，请找出这个数字。
>
> 示例 1:
>
> 输入: [0,1,3]
>
> 输出: 2
>
> 示例 2:
>
> 输入: [0,1,2,3,4,5,6,7,9]
>
> 输出: 8
>
> 限制：
>
> 1 <= 数组长度 <= 10000

3. **解法（二分查找算法）：**

算法思路：

关于这道题中，时间复杂度为 `O(N)` 的解法有很多种，而且也是比较好想的，这里就不再赘述。本题只讲解一个最优的二分法，来解决这个问题。

在这个升序的数组中，我们发现：

- 在第一个缺失位置的左边，数组内的元素都是与数组的下标相等的；
- 在第一个缺失位置的右边，数组内的元素与数组下标是不相等的。

因此，我们可以利用这个「二段性」，来使用「二分查找」算法。

C++ 算法代码：

```
1 class Solution
```

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

C++ 代码结果：

Java 算法代码：

```java
class Solution {
    public int missingNumber(int[] nums) {
        int left = 0, right = nums.length - 1;
        while(left < right) {
            int mid = left + (right - left) / 2;
            if(nums[mid] == mid) left = mid + 1;
            else right = mid;
        }
        return left == nums[left] ? left + 1 : left;
    }
}
```

Java 运行结果：

前缀和

25. 【模板】一维前缀和（easy）

1. **题目链接：** **【模板】前缀和**
2. **题目描述：**

描述

给定一个长度为n的数组$a_1, a_2, ...a_n$.
接下来有q次查询, 每次查询有两个参数l, r.
对于每个询问, 请输出$a_l + a_{l+1} + ... + a_r$

输入描述:

第一行包含两个整数n和q.
第二行包含n个整数, 表示$a_1, a_2, ...a_n$.
接下来q行,每行包含两个整数 l和r.
$1 \leq n, q \leq 10^5$
$-10^9 \leq a[i] \leq 10^9$
$1 \leq l \leq r \leq n$

输出描述:

输出q行,每行代表一次查询的结果.

示例1

输入:
```
3 2
1 2 4
1 2
2 3
```
复制

输出:
```
3
6
```
复制

3. **解法（前缀和）：**

算法思路：

a. 先预处理出来一个「前缀和」数组：

用 `dp[i]` 表示：`[1, i]` 区间内所有元素的和，那么 `dp[i - 1]` 里面存的就是 `[1, i - 1]` 区间内所有元素的和，那么：可得递推公式：`dp[i] = dp[i - 1] + arr[i]`；

b. 使用前缀和数组，「快速」求出「某一个区间内」所有元素的和：

当询问的区间是 `[l, r]` 时：区间内所有元素的和为：`dp[r] - dp[l - 1]`。

C++ 算法代码：

```cpp
#include <iostream>
using namespace std;

const int N = 100010;
long long arr[N], dp[N];
int n, q;

int main()
{
    cin >> n >> q;
    // 读取数据
    for(int i = 1; i <= n; i++) cin >> arr[i];
    // 处理前缀和数组
    for(int i = 1; i <= n; i++) dp[i] = dp[i - 1] + arr[i];
    while(q--)
    {
        int l, r;
        cin >> l >> r;
        // 计算区间和
        cout << dp[r] - dp[l - 1] << endl;
    }
    return 0;
}
```

C++ 代码结果：

Java 算法代码：

```java
import java.util.Scanner;

// 注意类名必须为 Main, 不要有任何 package xxx 信息
public class Main {
    public static void main(String[] args) {
        Scanner scan = new Scanner(System.in);
        int n = scan.nextInt();
        int q = scan.nextInt();
        // 为了防止溢出，用 long 类型的数组
        int[] arr = new int[n + 1];
        long[] dp = new long[n + 1];

        for (int i = 1; i <= n; i++) { // 读数据
            arr[i] = scan.nextInt();
        }
        for (int i = 1; i <= n; i++) { // 处理前缀和数组
            dp[i] = dp[i - 1] + arr[i];
        }
        while (q > 0) {
            int l = scan.nextInt();
            int r = scan.nextInt();
            System.out.println(dp[r] - dp[l - 1]); // 使用前缀和数组
            q--;
        }
    }
}
```

Java 运行结果：

26. 【模板】二维前缀和（medium）

1. **题目链接：** 【模板】二维前缀和
2. **题目描述：**

描述

给你一个 n 行 m 列的矩阵 A， 下标从1开始。

接下来有 q 次查询，每次查询输入 4 个参数 x1 , y1 , x2 , y2

请输出以 (x1, y1) 为左上角 , (x2,y2) 为右下角的子矩阵的和，

输入描述：

第一行包含三个整数n,m,q.

接下来n行，每行m个整数，代表矩阵的元素

接下来q行，每行4个整数x1, y1, x2, y2，分别代表这次查询的参数

$1 \leq n, m \leq 1000$
$1 \leq q \leq 10^5$

$-10^9 \leq a[i][j] \leq 10^9$
$1 \leq x_1 \leq x_2 \leq n$
$1 \leq y_1 \leq y_2 \leq m$

输出描述：

输出q行，每行表示查询结果。

示例1

输入：
```
3 4 3
1 2 3 4
3 2 1 0
1 5 7 8
1 1 2 2
1 1 3 3
1 2 3 4
```
复制

输出：
```
8
25
32
```
复制

TOP

备注：

读入数据可能很大，请注意读写时间。

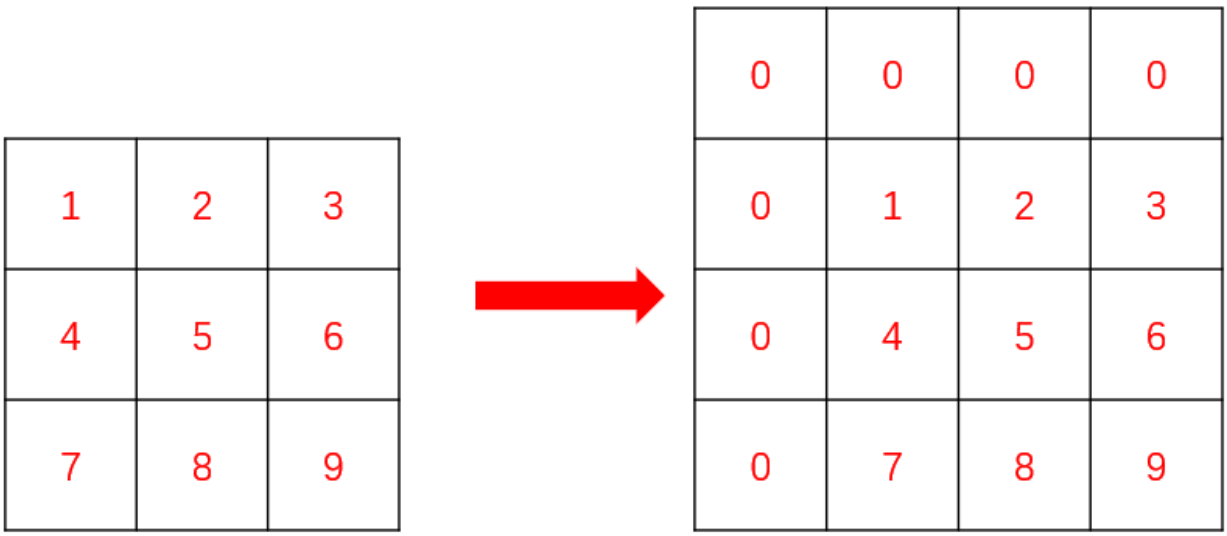
3. **解法：**

算法思路：

类比于一维数组的形式，如果我们能处理出来从 `[0, 0]` 位置到 `[i, j]` 位置这片区域内所有元素的累加和，就可以在 `O(1)` 的时间内，搞定矩阵内任意区域内所有元素的累加和。因此我们接下来仅需完成两步即可：

- 第一步：搞出来前缀和矩阵

  这里就要用到一维数组里面的拓展知识，我们要在矩阵的最上面和最左边添加上一行和一列0，这样我们就可以省去非常多的边界条件的处理（同学们可以自行尝试直接搞出来前缀和矩阵，边界条件的处理会让你崩溃的）。处理后的矩阵就像这样：



这样，我们填写前缀和矩阵数组的时候，下标直接从 `1` 开始，能大胆使用 `i - 1`，`j - 1` 位置的值。

注意 `dp` 表与原数组 `matrix` 内的元素的映射关系：

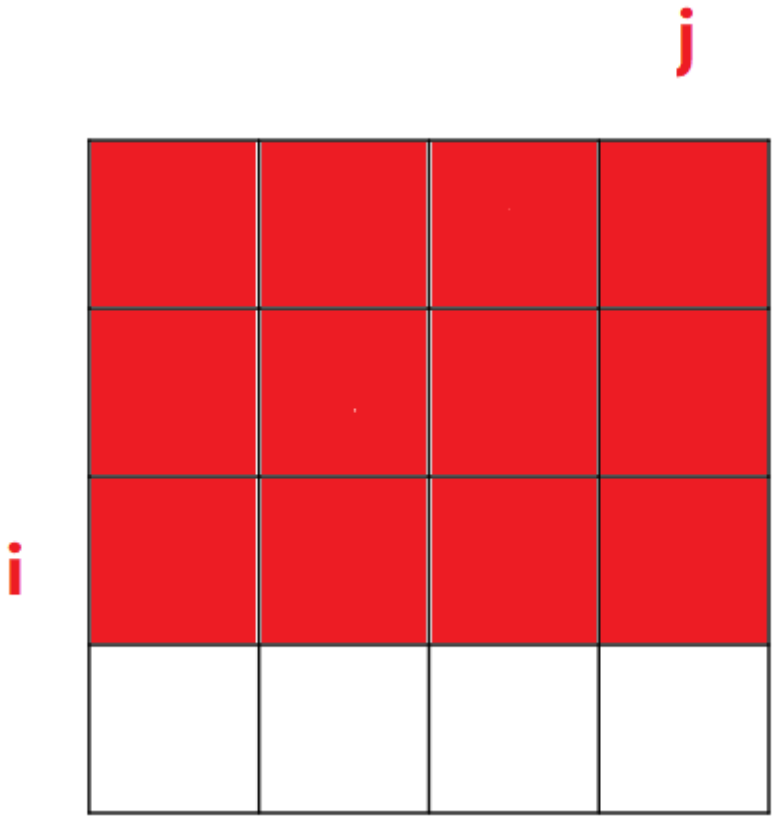
i. 从 `dp` 表到 `matrix` 矩阵，横纵坐标减一；

ii. 从 `matrix` 矩阵到 `dp` 表，横纵坐标加一。

前缀和矩阵中 `sum[i][j]` 的含义，以及如何递推二维前缀和方程

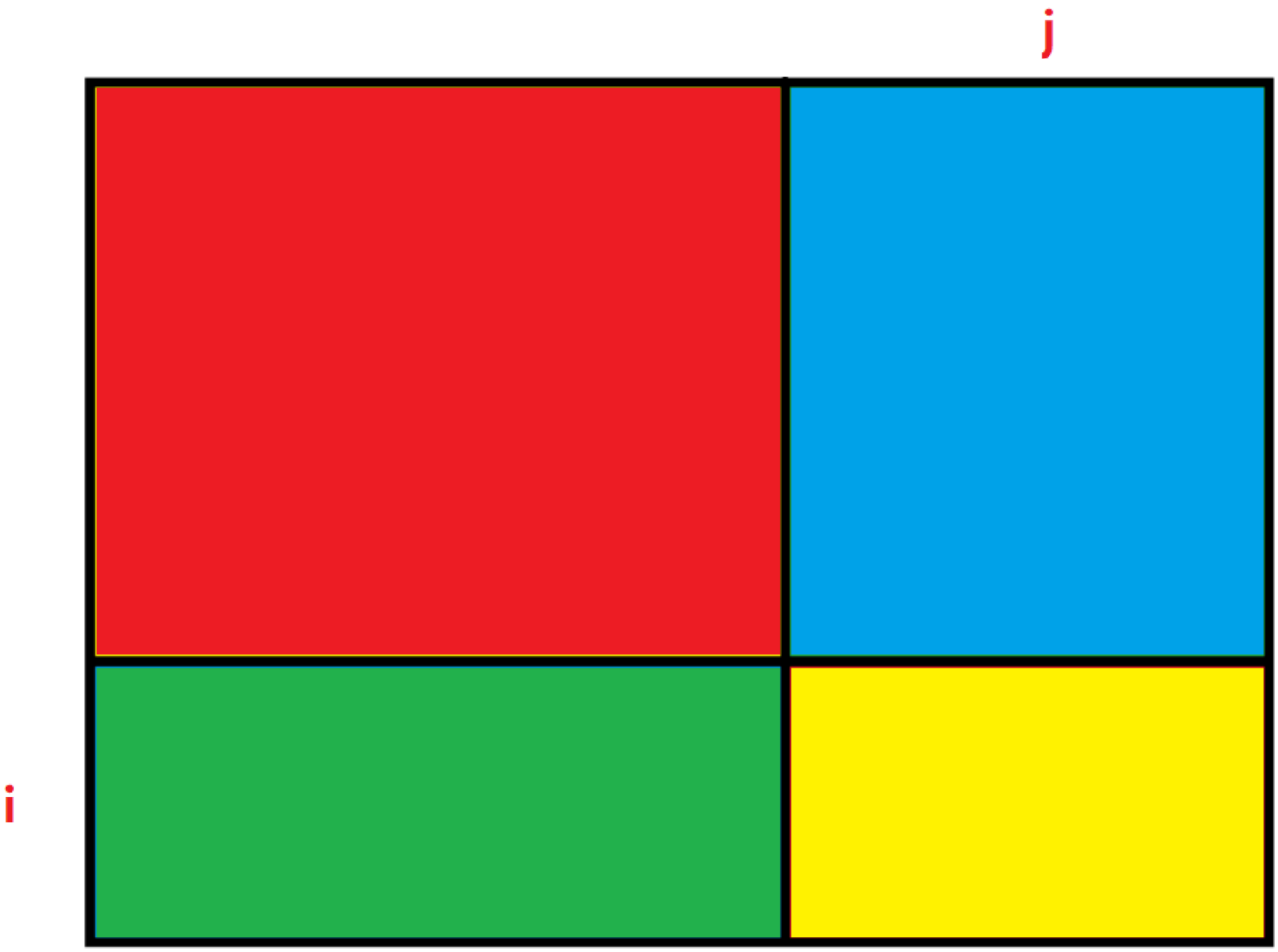
a. `sum[i][j]` 的含义：

`sum[i][j]` 表示，从 `[0, 0]` 位置到 `[i, j]` 位置这段区域内，所有元素的累加和。对应下图的红色区域：



a. 递推方程：

其实这个递推方程非常像我们小学做过求图形面积的题，我们可以将 `[0, 0]` 位置到 `[i, j]` 位置这段区域分解成下面的部分：



`sum[i][j]` = 红 + 蓝 + 绿 + 黄，分析一下这四块区域：

i. 黄色部分最简单，它就是数组中的 `matrix[i - 1][j - 1]` （注意坐标的映射关系）

ii. 单独的蓝不好求，因为它不是我们定义的状态表示中的区域，同理，单独的绿也是；

iii. 但是如果是红 + 蓝，正好是我们 `dp` 数组中 `sum[i - 1][j]` 的值，美滋滋；

iv. 同理，如果是红 + 绿，正好是我们 `dp` 数组中 `sum[i][j - 1]` 的值；

v. 如果把上面求的三个值加起来，那就是黄 + 红 + 蓝 + 红 + 绿，发现多算了一部分红的面积，因此再单独减去红的面积即可；

vi. 红的面积正好也是符合 `dp` 数组的定义的，即 `sum[i - 1][j - 1]`

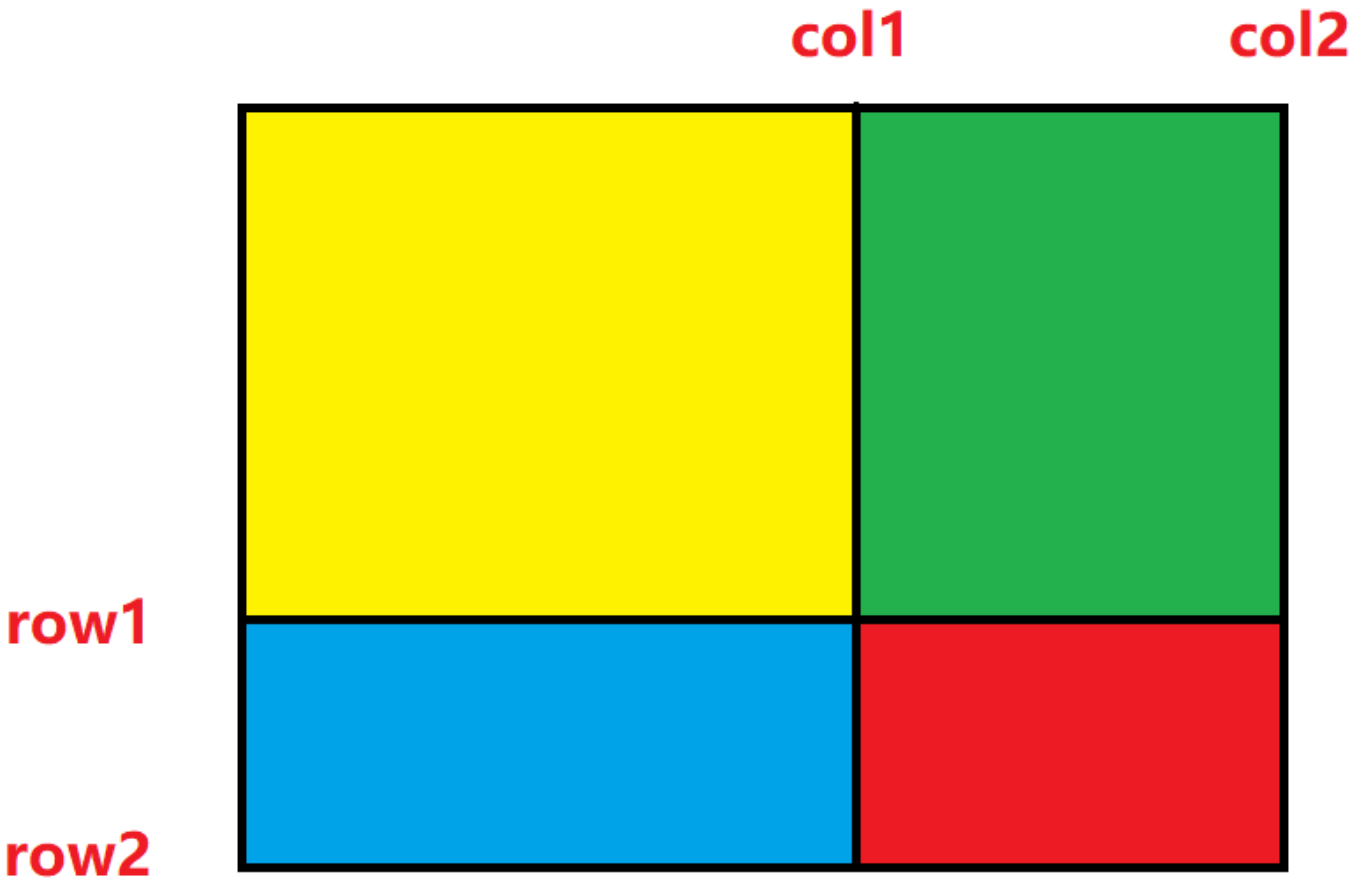
综上所述，我们的递推方程就是：

```
sum[i][j]=sum[i - 1][j] + sum[i][j - 1] - sum[i - 1][j - 1]+matrix[i - 1][j - 1]
```

- 第二步：使用前缀和矩阵

题目的接口中提供的参数是原始矩阵的下标，为了避免下标映射错误，这里直接先把下标映射成 `dp` 表里面对应的下标：`row1++, col1++, row2++, col2++`

接下来分析如何使用这个前缀和矩阵，如下图（注意这里的 `row` 和 `col` 都处理过了，对应的正是 `sum` 矩阵中的下标）：



对于左上角 `(row1, col1)` 、右下角 `(row2, col2)` 围成的区域，正好是红色的部分。因此我们要求的就是红色部分的面积，继续分析几个区域：

i. 黄色，能直接求出来，就是 `sum[row1 - 1, col1 - 1]` （为什么减一？因为要剔除掉 `row` 这一行和 `col` 这一列）

ii. 绿色，直接求不好求，但是和黄色拼起来，正好是 `sum` 表内 `sum[row1 - 1][col2]` 的数据；

iii. 同理，蓝色不好求，但是 蓝 + 黄 = `sum[row2][col1 - 1]` ；

iv. 再看看整个面积，好求嘛？非常好求，正好是 `sum[row2][col2]` ；

v. 那么，红色就 = 整个面积 - 黄 - 绿 - 蓝，但是绿蓝不好求，我们可以这样减：整个面积 -（绿 + 黄 ）-（蓝 + 黄），这样相当于多减去了一个黄，再加上即可

综上所述：红 = 整个面积 -（绿 + 黄）-（蓝 + 黄）+ 黄，从而可得红色区域内的元素总和为：

`sum[row2][col2]-sum[row2][col1 - 1]-sum[row1 - 1][col2]+sum[row1 - 1][col1 - 1]`

C++ 算法代码：

```cpp
#include <iostream>
using namespace std;

const int N = 1010;
int arr[N][N];
long long dp[N][N];
int n, m, q;

int main()
{
    cin >> n >> m >> q;
    // 读入数据
    for(int i = 1; i <= n; i++)
        for(int j = 1; j <= m; j++)
            cin >> arr[i][j];
    // 处理前缀和矩阵
    for(int i = 1; i <= n; i++)
        for(int j = 1; j <= m; j++)
            dp[i][j] = dp[i - 1][j] + dp[i][j - 1] + arr[i][j] - dp[i - 1][j - 1];
    // 使用前缀和矩阵
    int x1, y1, x2, y2;
    while(q--)
    {
        cin >> x1 >> y1 >> x2 >> y2;
        cout << dp[x2][y2] - dp[x1 - 1][y2] - dp[x2][y1 - 1] + dp[x1 - 1][y1 - 1] << endl;
    }
}
```

C++ 代码结果：

Java 算法代码：

```java
import java.util.Scanner;

// 注意类名必须为 Main, 不要有任何 package xxx 信息
public class Main {
    public static void main(String[] args) {
        Scanner in = new Scanner(System.in);
        int n = in.nextInt();
        int m = in.nextInt();
        int q = in.nextInt();
        int[][] arr = new int[n + 1][m + 1];
        long[][] dp = new long[n + 1][m + 1];

        // 读入数据
        for(int i = 1; i <= n; i++)
            for(int j = 1; j <= m; j++)
                arr[i][j] = in.nextInt();

        // 处理前缀和矩阵
        for(int i = 1; i <= n; i++)
            for(int j = 1; j <= m; j++)
                dp[i][j] = dp[i - 1][j] + dp[i][j - 1] - dp[i - 1][j - 1] +
arr[i][j];
```

```
22
23          // 使用前缀和矩阵
24          while(q > 0)
25          {
26              int x1 = in.nextInt(), y1 = in.nextInt(), x2 = in.nextInt(), y2 =
    in.nextInt();
27              System.out.println(dp[x2][y2] - dp[x1 - 1][y2] - dp[x2][y1 - 1] +
    dp[x1 - 1][y1 - 1]);
28              q--;
29          }
30      }
31  }
```

Java 运行结果：

27. 寻找数组的中心下标（easy）

1. **题目链接：724. 寻找数组的中心下标**

2. **题目描述：**

> 给你一个整数数组 nums ，请计算数组的 中心下标 。
>
> 数组 中心下标 是数组的一个下标，其左侧所有元素相加的和等于右侧所有元素相加的和。

如果中心下标位于数组最左端，那么左侧数之和视为 0 ，因为在下标的左侧不存在元素。这一点对于中心下标位于数组最右端同样适用。

如果数组有多个中心下标，应该返回 最靠近左边 的那一个。如果数组不存在中心下标，返回 -1 。

示例 1：

输入：nums = [1, 7, 3, 6, 5, 6]

输出：3

解释：

中心下标是 3 。

左侧数之和 sum = nums[0] + nums[1] + nums[2] = 1 + 7 + 3 = 11 ，

右侧数之和 sum = nums[4] + nums[5] = 5 + 6 = 11 ，二者相等。

示例 2：

输入：nums = [1, 2, 3]

输出：-1

解释：

数组中不存在满足此条件的中心下标。

示例 3：

输入：nums = [2, 1, -1]

输出：0

解释：

中心下标是 0 。

左侧数之和 sum = 0 ，（下标 0 左侧不存在元素），

右侧数之和 sum = nums[1] + nums[2] = 1 + -1 = 0 。

提示：

1 <= nums.length <= 10^4

-1000 <= nums[i] <= 1000

3. **解法（前缀和）：**

算法思路：

从中心下标的定义可知，除中心下标的元素外，该元素左边的「前缀和」等于该元素右边的「后缀和」。

- 因此，我们可以先预处理出来两个数组，一个表示前缀和，另一个表示后缀和。
- 然后，我们可以用一个 `for` 循环枚举可能的中心下标，判断每一个位置的「前缀和」以及「后缀和」，如果二者相等，就返回当前下标。

C++ 算法代码：

```cpp
class Solution {
public:
    int pivotIndex(vector<int>& nums) {
        // lsum[i] 表示：[0, i - 1] 区间所有元素的和
        // rsum[i] 表示：[i + 1, n - 1] 区间所有元素的和
        int n = nums.size();
        vector<int> lsum(n), rsum(n);
        // 预处理前缀和后缀和数组
        for(int i = 1; i < n; i++)
            lsum[i] = lsum[i - 1] + nums[i - 1];
        for(int i = n - 2; i >= 0; i--)
            rsum[i] = rsum[i + 1] + nums[i + 1];

        // 判断
        for(int i = 0; i < n; i++)
            if(lsum[i] == rsum[i])
                return i;
        return -1;
    }
};
```

C++ 运行结果：

Java 算法代码：

```java
class Solution
{
    public int pivotIndex(int[] nums)
    {
        // lsum[i] 表示：[0, i - 1] 区间所有元素的和
        // rsum[i] 表示：[i + 1, n - 1] 区间所有元素的和
        int n = nums.length;
        int[] lsum = new int[n];
        int[] rsum = new int[n];
        // 预处理前缀和后缀和数组
        for(int i = 1; i < n; i++)
            lsum[i] = lsum[i - 1] + nums[i - 1];
        for(int i = n - 2; i >= 0; i--)
            rsum[i] = rsum[i + 1] + nums[i + 1];

        // 判断
        for(int i = 0; i < n; i++)
            if(lsum[i] == rsum[i])
                return i;
        return -1;
    }
}
```

Java 运行结果：

28. 除自身以外数组的乘积（medium）

1. **题目链接：238. 除自身以外数组的乘积**

2. **题目描述：**

> 给你一个整数数组 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 。
>
> 题目数据 保证 数组 nums 之中任意元素的全部前缀元素和后缀的乘积都在 32 位 整数范围内。
>
> **请不要使用除法**，且在 O(n) 时间复杂度内完成此题。
>
> 示例 1:

> 输入: nums = [1,2,3,4]
>
> 输出: [24,12,8,6]
>
> 示例 2:
>
> 输入: nums = [-1,1,0,-3,3]
>
> 输出: [0,0,9,0,0]
>
> 提示：
>
> 2 <= nums.length <= 105
>
> -30 <= nums[i] <= 30
>
> 保证 数组 nums 之中任意元素的全部前缀元素和后缀的乘积都在 32 位 整数范围内
>
> 进阶：你可以在 O(1) 的额外空间复杂度内完成这个题目吗？（出于对空间复杂度分析的目的，输出数组不被视为额外空间。）

3. **解法（前缀和数组）：**

算法思路：

注意题目的要求，不能使用除法，并且要在 `O(N)` 的时间复杂度内完成该题。那么我们就不能使用暴力的解法，以及求出整个数组的乘积，然后除以单个元素的方法。

继续分析，根据题意，对于每一个位置的最终结果 `ret[i]`，它是由两部分组成的：

i. `nums[0] * nums[1] * nums[2] * ... * nums[i - 1]`

ii. `nums[i + 1] * nums[i + 2] * ... * nums[n - 1]`

于是，我们可以利用前缀和的思想，使用两个数组 post 和 suf，分别处理出来两个信息：

i. post 表示：i 位置之前的所有元素，即 `[0, i - 1]` 区间内所有元素的前缀乘积，

ii. suf 表示：i 位置之后的所有元素，即 `[i + 1, n - 1]` 区间内所有元素的后缀乘积

然后再处理最终结果。

C++ 算法代码：

```
1 class Solution
2 {
3 public:
4     vector<int> productExceptSelf(vector<int>& nums)
5     {
```

```cpp
        // lprod 表示：[0, i - 1] 区间内所有元素的乘积
        // rprod 表示：[i + 1, n - 1] 区间内所有元素的乘积
        int n = nums.size();
        vector<int> lprod(n + 1), rprod(n + 1);
        lprod[0] = 1, rprod[n - 1] = 1;

        // 预处理前缀积以及后缀积
        for(int i = 1; i < n; i++)
            lprod[i] = lprod[i - 1] * nums[i - 1];
        for(int i = n - 2; i >= 0; i--)
            rprod[i] = rprod[i + 1] * nums[i + 1];

        // 处理结果数组
        vector<int> ret(n);
        for(int i = 0; i < n; i++)
            ret[i] = lprod[i] * rprod[i];
        return ret;
    }
};
```

C++ 运行结果：

Java 算法代码：

```java
class Solution {
    public int[] productExceptSelf(int[] nums) {
        // lprod 表示：[0, i - 1] 区间内所有元素的乘积
        // rprod 表示：[i + 1, n - 1] 区间内所有元素的乘积
        int n = nums.length;
        int[] lprod = new int[n];
        int[] rprod = new int[n];
        lprod[0] = 1; rprod[n - 1] = 1;

        // 预处理前缀积以及后缀积
        for(int i = 1; i < n; i++)
            lprod[i] = lprod[i - 1] * nums[i - 1];
        for(int i = n - 2; i >= 0; i--)
            rprod[i] = rprod[i + 1] * nums[i + 1];
```

```
15
16          // 处理结果数组
17          int[] ret = new int[n];
18          for(int i = 0; i < n; i++)
19              ret[i] = lprod[i] * rprod[i];
20          return ret;
21      }
22  }
```

Java 运行结果：

29. 和为 k 的子数组（medium）

1. **题目链接：560. 和为 K 的子数组**

2. **题目描述：**

> 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 的连续子数组的个数 。
>
> 示例 1：
>
> 输入：nums = [1,1,1], k = 2
>
> 输出：2
>
> 示例 2：
>
> 输入：nums = [1,2,3], k = 3
>
> 输出：2
>
> 提示：
>
> 1 <= nums.length <= 2 * 10^4
>
> -1000 <= nums[i] <= 1000
>
> -10^7 <= k <= 10^7

4. **解法一（将前缀和存在哈希表中）：**

算法思路：

设 `i` 为数组中的任意位置，用 `sum[i]` 表示 `[0, i]` 区间内所有元素的和。

想知道有多少个「以 `i` 为结尾的和为 `k` 的子数组」，就要找到有多少个起始位置为 `x1, x2, x3...` 使得 `[x, i]` 区间内的所有元素的和为 `k`。那么 `[0, x]` 区间内的和是不是就是 `sum[i] - k` 了。于是问题就变成：

- 找到在 `[0, i - 1]` 区间内，有多少前缀和等于 `sum[i] - k` 的即可。

我们不用真的初始化一个前缀和数组，因为我们只关心在 `i` 位置之前，有多少个前缀和等于 `sum[i] - k`。因此，我们仅需用一个哈希表，一边求当前位置的前缀和，一边存下之前每一种前缀和出现的次数。

C++ 算法代码：

```
class Solution
{
public:
    int subarraySum(vector<int>& nums, int k)
    {
        unordered_map<int, int> hash; // 统计前缀和出现的次数
        hash[0] = 1;

        int sum = 0, ret = 0;
        for(auto x : nums)
        {
            sum += x; // 计算当前位置的前缀和
            if(hash.count(sum - k)) ret += hash[sum - k]; // 统计个数
            hash[sum]++;
```

```
        }
        return ret;
    }
};
```

C++ 运行结果：

Java 算法代码：

```java
class Solution {
    public int subarraySum(int[] nums, int k) {
        Map<Integer, Integer> hash = new HashMap<Integer, Integer>();
        hash.put(0, 1);

        int sum = 0, ret = 0;
        for(int x : nums)
        {
            sum += x; // 计算当前位置的前缀和
            ret += hash.getOrDefault(sum - k, 0); // 统计结果
            hash.put(sum, hash.getOrDefault(sum, 0) + 1); // 把当前的前缀和丢到哈希表里面
        }
        return ret;
    }
}
```

Java 运行结果：

30. 和可被 K 整除的子数组（medium）

（本题是某一年的蓝桥杯竞赛原题哈）

1. **题目链接：974. 和可被 K 整除的子数组**

2. **题目描述：**

> 给定一个整数数组 nums 和一个整数 k ，返回其中元素之和可被 k 整除的（连续、非空） 子数组 的数目。
>
> 子数组 是数组的 连续 部分。
>
> 示例 1：
>
> 输入：
>
> nums = [4,5,0,-2,-3,1], k = 5
>
> 输出：
>
> 7
>
> 解释：
>
> 有 7 个子数组满足其元素之和可被 k = 5 整除：
>
> [4, 5, 0, -2, -3, 1], [5], [5, 0], [5, 0, -2, -3], [0], [0, -2, -3], [-2, -3]
>
> 示例 2:
>
> 输入:
>
> nums = [5], k = 9
>
> 输出:
>
> 0
>
> 提示:
>
> 1 <= nums.length <= 3 * 104
>
> -104 <= nums[i] <= 104
>
> 2 <= k <= 104

3. **解法（前缀和在哈希表中）：**

（暴力解法就是枚举出所有的子数组的和，这里不再赘述。）

本题需要的前置知识：

- 同余定理

  如果 `(a - b) % n == 0` ，那么我们可以得到一个结论： `a % n == b % n` 。用文字叙述就是，如果两个数相减的差能被 n 整除，那么这两个数对 n 取模的结果相同。

例如：`(26 - 2) % 12 == 0`，那么 `26 % 12 == 2 % 12 == 2`。

- `c++` 中负数取模的结果，以及如何修正「负数取模」的结果

  a. `c++` 中关于负数的取模运算，结果是「把负数当成正数，取模之后的结果加上一个负号」。

  例如：`-1 % 3 = -(1 % 3) = -1`

  b. 因为有负数，为了防止发生「出现负数」的结果，以 `(a % n + n) % n` 的形式输出保证为正。

  例如：`-1 % 3 = (-1 % 3 + 3) % 3 = 2`

算法思路：

思路与 560. 和为 K 的子数组 这道题的思路相似。

设 `i` 为数组中的任意位置，用 `sum[i]` 表示 `[0, i]` 区间内所有元素的和。

- 想知道有多少个「以 `i` 为结尾的可被 `k` 整除的子数组」，就要找到有多少个起始位置为 `x1, x2, x3...` 使得 `[x, i]` 区间内的所有元素的和可被 `k` 整除。
- 设 `[0, x - 1]` 区间内所有元素之和等于 `a`，`[0, i]` 区间内所有元素的和等于 `b`，可得 `(b - a) % k == 0`。
- 由同余定理可得，`[0, x - 1]` 区间与 `[0, i]` 区间内的前缀和同余。于是问题就变成：
  - 找到在 `[0, i - 1]` 区间内，有多少前缀和的余数等于 `sum[i] % k` 的即可。

我们不用真的初始化一个前缀和数组，因为我们只关心在 `i` 位置之前，有多少个前缀和等于 `sum[i] - k`。因此，我们仅需用一个哈希表，一边求当前位置的前缀和，一边存下之前每一种前缀和出现的次数。

C++ 算法代码：

```
1 class Solution
```

```cpp
{
public:
    int subarraysDivByK(vector<int>& nums, int k)
    {
        unordered_map<int, int> hash;
        hash[0 % k] = 1; // 0 这个数的余数

        int sum = 0, ret = 0;
        for(auto x : nums)
        {
            sum += x; // 算出当前位置的前缀和
            int r = (sum % k + k) % k; // 修正后的余数
            if(hash.count(r)) ret += hash[r]; // 统计结果
            hash[r]++;
        }
        return ret;
    }
};
```

C++ 运行结果：

Java 算法代码：

```java
class Solution {
    public int subarraysDivByK(int[] nums, int k) {
        Map<Integer, Integer> hash = new HashMap<Integer, Integer>();
        hash.put(0 % k, 1);

        int sum = 0, ret = 0;
        for(int x : nums)
        {
            sum += x; // 计算当前位置的前缀和
            int r = (sum % k + k) % k;
            ret += hash.getOrDefault(r, 0); // 统计结果
            hash.put(r, hash.getOrDefault(r, 0) + 1);
        }
        return ret;
    }
```

```
16 }
```

Java 运行结果：

31. 连续数组（medium）

1. **题目链接：525. 连续数组**

2. **题目描述：**

> 给定一个二进制数组 nums , 找到含有相同数量的 0 和 1 的最长连续子数组，并返回该子数组的长度。
>
> 示例 1:
>
> 输入: nums = [0,1]
>
> 输出: 2
>
> 说明:
>
> [0, 1] 是具有相同数量 0 和 1 的最长连续子数组。
>
> 示例 2:
>
> 输入: nums = [0,1,0]
>
> 输出: 2
>
> 说明:
>
> [0, 1] (或 [1, 0]) 是具有相同数量 0 和 1 的最长连续子数组。
>
> 提示：
>
> 1 <= nums.length <= 10^5
>
> nums[i] 不是 0 就是 1

（暴力解法就是枚举所有的子数组，然后判断子数组是否满足要求，这里不再赘述。）

3. **解法（前缀和在哈希表中）：**

算法思路：

稍微转化一下题目，就会变成我们熟悉的题：

- 本题让我们找出一段连续的区间，`0` 和 `1` 出现的次数相同。
- 如果将 `0` 记为 `-1`，`1` 记为 `1`，问题就变成了找出一段区间，这段区间的和等于 `0`。
- 于是，就和 **560. 和为 K 的子数组** 这道题的思路一样

设 `i` 为数组中的任意位置，用 `sum[i]` 表示 `[0, i]` 区间内所有元素的和。

想知道最大的「以 `i` 为结尾的和为 `0` 的子数组」，就要找到从左往右第一个 `x1` 使得 `[x1, i]` 区间内的所有元素的和为 `0`。那么 `[0, x1 - 1]` 区间内的和是不是就是 `sum[i]` 了。于是问题就变成：

- 找到在 `[0, i - 1]` 区间内，第一次出现 `sum[i]` 的位置即可。

我们不用真的初始化一个前缀和数组，因为我们只关心在 `i` 位置之前，第一个前缀和等于 `sum[i]` 的位置。因此，我们仅需用一个哈希表，一边求当前位置的前缀和，一边记录第一次出现该前缀和的位置。

C++ 算法代码：

```
1  class Solution
2  {
3  public:
4      int findMaxLength(vector<int>& nums)
5      {
6          unordered_map<int, int> hash;
7          hash[0] = -1; // 默认有一个前缀和为 0 的情况
8
9          int sum = 0, ret = 0;
10         for(int i = 0; i < nums.size(); i++)
11         {
```

```
            sum += nums[i] == 0 ? -1 : 1; // 计算当前位置的前缀和
            if(hash.count(sum)) ret = max(ret, i - hash[sum]);
            else hash[sum] = i;
        }
        return ret;
    }
};
```

C++ 运行结果：

Java 算法代码：

```
class Solution {
    public int findMaxLength(int[] nums) {
        Map<Integer, Integer> hash = new HashMap<Integer, Integer>();
        hash.put(0, -1); // 默认存在一个前缀和为 0 的情况

        int sum = 0, ret = 0;
        for(int i = 0; i < nums.length; i++)
        {
            sum += (nums[i] == 0 ? -1 : 1); // 计算当前位置的前缀和
            if(hash.containsKey(sum)) ret = Math.max(ret, i - hash.get(sum));
            else hash.put(sum, i);
        }
        return ret;
    }
}
```

Java 运行结果：

32. 矩阵区域和（medium）

1. **题目链接：1314. 矩阵区域和**

2. **题目描述：**

> 给你一个 m x n 的矩阵 mat 和一个整数 k ，请你返回一个矩阵 answer ，其中每个 answer[i][j] 是所有满足下述条件的元素 mat[r][c] 的和：
>
> - i - k <= r <= i + k,
> - j - k <= c <= j + k 且
> - (r, c) 在矩阵内。
>
> 示例 1：
>
> 输入：mat = [[1,2,3],[4,5,6],[7,8,9]], k = 1
>
> 输出：[[12,21,16],[27,45,33],[24,39,28]]
>
> 示例 2：
>
> 输入：mat = [[1,2,3],[4,5,6],[7,8,9]], k = 2
>
> 输出：[[45,45,45],[45,45,45],[45,45,45]]
>
> 提示：
>
> m == mat.length
>
> n == mat[i].length
>
> 1 <= m, n, k <= 100
>
> 1 <= mat[i][j] <= 100

3. **解法：**

算法思路：

二维前缀和的简单应用题，关键就是我们在填写结果矩阵的时候，要找到原矩阵对应区域的「左上角」以及「右下角」的坐标（推荐大家画图）

左上角坐标：`x1 = i - k, y1 = j - k`，但是由于会「超过矩阵」的范围，因此需要对 `0` 取一个 `max`。因此修正后的坐标为：`x1 = max(0, i - k), y1 = max(0, j - k)`；

右下角坐标：`x1 = i + k, y1 = j + k`，但是由于会「超过矩阵」的范围，因此需要对 `m - 1`，以及 `n - 1` 取一个 `min`。因此修正后的坐标为：`x2 = min(m - 1, i + k), y2 = min(n - 1, j + k)`。

然后将求出来的坐标代入到「二维前缀和矩阵」的计算公式上即可~（但是要注意下标的映射关系）

C++ 算法代码：

```
class Solution {
public:
    vector<vector<int>> matrixBlockSum(vector<vector<int>>& mat, int k) {
        int m = mat.size(), n = mat[0].size();
        vector<vector<int>> dp(m + 1, vector<int>(n + 1));

        // 1. 预处理前缀和矩阵
        for(int i = 1; i <= m; i++)
            for(int j = 1; j <= n; j++)
                dp[i][j] = dp[i - 1][j] + dp[i][j - 1] - dp[i - 1][j - 1] + mat[i - 1][j - 1];

        // 2. 使用
        vector<vector<int>> ret(m, vector<int>(n));
        for(int i = 0; i < m; i++)
            for(int j = 0; j < n; j++)
            {
                int x1 = max(0, i - k) + 1, y1 = max(0, j - k) + 1;
                int x2 = min(m - 1, i + k) + 1, y2 = min(n - 1, j + k) + 1;
                ret[i][j] = dp[x2][y2] - dp[x1 - 1][y2] - dp[x2][y1 - 1] + dp[x1 - 1][y1 - 1];
            }

        return ret;
    }
};
```

C++ 代码结果：

Java 算法代码：

```java
class Solution {
    public int[][] matrixBlockSum(int[][] mat, int k) {
        int m = mat.length, n = mat[0].length;

        // 1. 预处理前缀和矩阵
        int[][] dp = new int[m + 1][n + 1];
        for(int i = 1; i <= m; i++)
            for(int j = 1; j <= n; j++)
                dp[i][j] = dp[i - 1][j] + dp[i][j - 1] - dp[i - 1][j - 1] + mat[i - 1][j - 1];

        // 2. 使用
        int[][] ret = new int[m][n];
        for(int i = 0; i < m; i++)
            for(int j = 0; j < n; j++)
            {
                int x1 = Math.max(0, i - k) + 1, y1 = Math.max(0, j - k) + 1;
                int x2 = Math.min(m - 1, i + k) + 1, y2 = Math.min(n - 1, j + k) + 1;
                ret[i][j] = dp[x2][y2] - dp[x1 - 1][y2] - dp[x2][y1 - 1] + dp[x1 - 1][y1 - 1];
            }
        return ret;
    }
}
```

Java 运行结果：