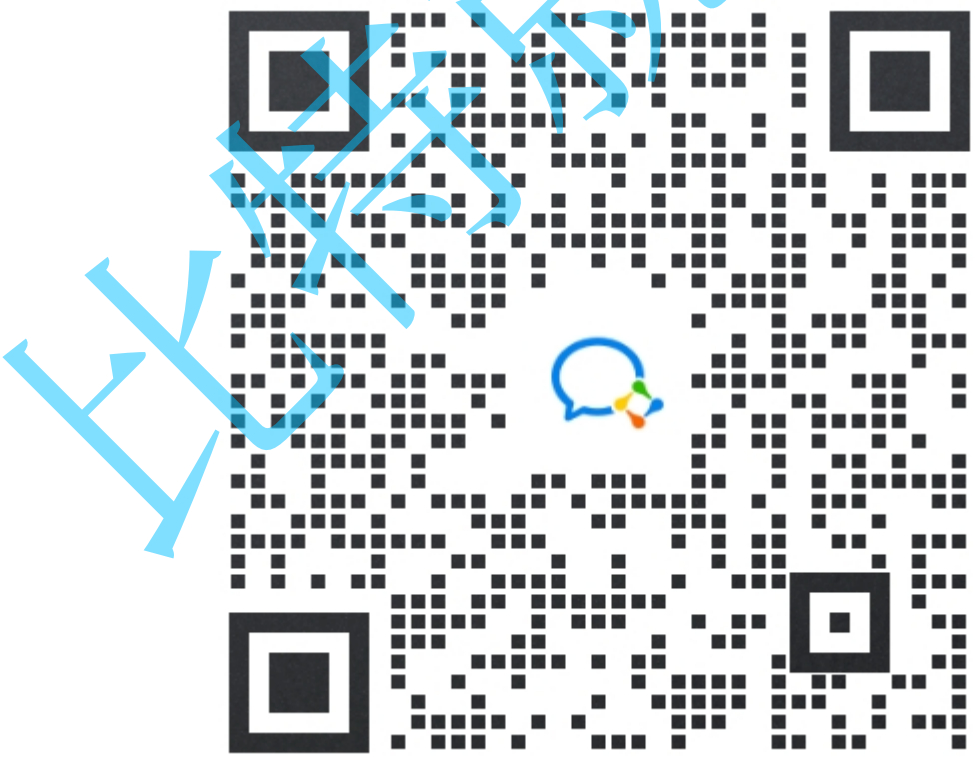
优选算法精品课(No.034~No.052)

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

位运算

33. 判断字符是否唯一（easy）

1. **题目链接：面试题 01.01. 判定字符是否唯一**

2. **题目描述：**

> 实现一个算法，确定一个字符串 s 的所有字符是否全都不同。
>
> 示例 1：
>
> 输入: s = "leetcode"
>
> 输出: false
>
> 示例 2：
>
> 输入: s = "abc"
>
> 输出: true
>
> 限制：
>
> 0 <= len(s) <= 100
>
> s[i]仅包含小写字母
>
> 如果你不使用额外的数据结构，会很加分。

3. **解法（位图的思想）：**

算法思路：

利用「位图」的思想，每一个「比特位」代表一个「字符，一个 `int` 类型的变量的 `32` 位足够表示所有的小写字母。比特位里面如果是 `0`，表示这个字符没有出现过。比特位里面的值是 `1`，表示该字符出现过。

那么我们就可以用一个「整数」来充当「哈希表」。

C++ 算法代码：

```cpp
class Solution
{
public:
    bool isUnique(string astr)
    {
        // 利用鸽巢原理来做的优化
        if(astr.size() > 26) return false;

        int bitMap = 0;
        for(auto ch : astr)
        {
            int i = ch - 'a';
            // 先判断字符是否已经出现过
            if(((bitMap >> i) & 1) == 1) return false;
            // 把当前字符加入到位图中
            bitMap |= 1 << i;
        }
        return true;
    }
};
```

C++ 运行结果：

Java 算法代码：

```java
class Solution {
    public boolean isUnique(String astr) {
        // 利用鸽巢原理来做优化
        if(astr.length() > 26) return false;

        int bitMap = 0;
        for(int i = 0; i < astr.length(); i++)
        {
            int x = astr.charAt(i) - 'a';
            // 先判断字符是否在位图中
            if(((bitMap >> x) & 1) == 1) return false;
            // 把当前字符加入到位图中
            bitMap |= 1 << x;
```

```
14          }
15          return true;
16      }
17  }
```

Java 运行结果：

34. 丢失的数字（easy）

1. **题目链接：268. 丢失的数字**

2. **题目描述：**

给定一个包含 [0, n] 中 n 个数的数组 nums ，找出 [0, n] 这个范围内没有出现在数组中的那个数。

示例 1：

输入：nums = [3,0,1]

输出：2

解释：n = 3，因为有 3 个数字，所以所有的数字都在范围 [0,3] 内。2 是丢失的数字，因为它没有出现在 nums 中。

示例 2：

输入：nums = [0,1]

输出：2

解释：n = 2，因为有 2 个数字，所以所有的数字都在范围 [0,2] 内。2 是丢失的数字，因为它没有出现在 nums 中。

示例 3：

输入：nums = [9,6,4,2,3,5,7,0,1]

输出：8

解释：n = 9，因为有 9 个数字，所以所有的数字都在范围 [0,9] 内。8 是丢失的数字，因为它没有出现在 nums 中。

示例 4：

> 输入：nums = [0]
>
> 输出：1
>
> 解释：n = 1，因为有 1 个数字，所以所有的数字都在范围 [0,1] 内。1 是丢失的数字，因为它没有出现在 nums 中。

提示：

> n == nums.length
>
> 1 <= n <= 10^4
>
> 0 <= nums[i] <= n
>
> nums 中的所有数字都 独一无二

进阶：你能否实现线性时间复杂度、仅使用额外常数空间的算法解决此问题?

3. **解法（位运算）：**

算法思路：

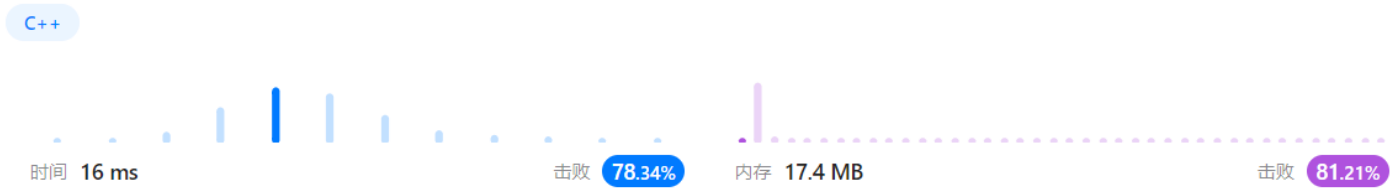
设数组的大小为 `n`，那么缺失之前的数就是 `[0, n]`，数组中是在 `[0, n]` 中缺失一个数形成的序列。

如果我们把数组中的所有数，以及 `[0, n]` 中的所有数全部「异或」在一起，那么根据「异或」运算的「消消乐」规律，最终的异或结果应该就是缺失的数~

C++ 算法代码：

```
class Solution
{
public:
    int missingNumber(vector<int>& nums)
    {
        int ret = 0;
        for(auto x : nums) ret ^= x;
        for(int i = 0; i <= nums.size(); i++) ret ^= i;
        return ret;
    }
};
```

C++ 代码结果：



Java 算法代码：

```java
class Solution {
    public int missingNumber(int[] nums) {
        int ret = 0;
        for(int x : nums) ret ^= x;
        for(int i = 0; i <= nums.length; i++) ret ^= i;
        return ret;
    }
}
```

Java 运行结果：

35. 两整数之和（medium）

1. **题目链接：371. 两整数之和**
2. **题目描述：**

> 给你两个整数 a 和 b，不使用 运算符 + 和 - ，计算并返回两整数之和。
>
> 示例 1：
>
> 输入：a = 1, b = 2
>
> 输出：3
>
> 示例 2：
>
> 输入：a = 2, b = 3
>
> 输出：5

> 提示：
>
> -1000 <= a, b <= 1000

3. 解法（位运算）：

算法思路：

- 异或 `^` 运算本质是「无进位加法」；
- 按位与 `&` 操作能够得到「进位」；
- 然后一直循环进行，直到「进位」变成 `0` 为止。

C++ 算法代码：

```cpp
class Solution
{
public:
    int getSum(int a, int b)
    {
        while(b != 0)
        {
            int x = a ^ b; // 先算出无进位相加的结果
            unsigned int carry = (unsigned int)(a & b) << 1; // 算出进位
            a = x;
            b = carry;
        }
        return a;
    }
};
```

C++ 运行结果：

Java 算法代码：

```java
class Solution {
```

```java
    public int getSum(int a, int b) {
        while(b != 0)
        {
            int x = a ^ b; // 先算出无进位相加的结果
            int carry = (a & b) << 1; // 计算进位
            a = x;
            b = carry;
        }
        return a;
    }
}
```

Java 运行结果：

36. 只出现一次的数字 II（medium）

1. **题目链接：137. 只出现一次的数字 II**

2. **题目描述：**

> 给你一个整数数组 nums ，除某个元素仅出现 一次 外，其余每个元素都恰出现 三次 。请你找出并返回那个只出现了一次的元素。
>
> 你必须设计并实现线性时间复杂度的算法且不使用额外空间来解决此问题。
>
> 示例 1：
>
> 输入：nums = [2,2,3,2]
>
> 输出：3
>
> 示例 2：
>
> 输入：nums = [0,1,0,1,0,1,99]
>
> 输出：99
>
> 提示：
>
> 1 <= nums.length <= 3 * 104

> -231 <= nums[i] <= 231 - 1
>
> nums 中，除某个元素仅出现 一次 外，其余每个元素都恰出现 三次

3. **解法（比特位计数）：**

算法思路：

设要找的数位 `ret`。

由于整个数组中，需要找的元素只出现了「一次」，其余的数都出现的「三次」，因此我们可以根据所有数的「某一个比特位」的总和 `%3` 的结果，快速定位到 `ret` 的「一个比特位上」的值是 `0` 还是 `1`。

这样，我们通过 `ret` 的每一个比特位上的值，就可以将 `ret` 给还原出来。

C++ 算法代码：

```cpp
class Solution
{
public:
    int singleNumber(vector<int>& nums)
    {
        int ret = 0;
        for(int i = 0; i < 32; i++) // 依次去修改 ret 中的每一位
        {
            int sum = 0;
            for(int x : nums) // 计算nums中所有的数的第 i 位的和
                if(((x >> i) & 1) == 1)
                    sum++;
            sum %= 3;
            if(sum == 1) ret |= 1 << i;
        }
        return ret;
    }
};
```

C++ 代码结果：

Java 算法代码：

```java
class Solution
{
    public int singleNumber(int[] nums)
    {
        int ret = 0;
        for(int i = 0; i < 32; i++) // 依次修改 ret 中的每一个比特位
        {
            int sum = 0;
            for(int x : nums) // 统计 nums 中所有的数的第 i 位的和
                if(((x >> i) & 1) == 1)
                    sum++;
            sum %= 3;
            if(sum == 1) ret |= 1 << i;
        }
        return ret;
    }
}
```

Java 运行结果：

37. 消失的两个数字（hard）

1. **题目链接：面试题 17.19. 消失的两个数字**

2. **题目描述：**

> 给定一个数组，包含从 1 到 N 所有的整数，但其中缺了两个数字。你能在 O(N) 时间内只用 O(1) 的空间找到它们吗？
>
> 以任意顺序返回这两个数字均可。
>
> 示例 1:
>
> 输入: [1]
>
> 输出: [2,3]

> 示例 2:
>
> 输入: [2,3]
>
> 输出: [1,4]
>
> 提示：
>
> nums.length <= 30000

3. **解法（位运算）：**

算法思路：

本题就是 **268. 丢失的数字** + **260. 只出现一次的数字 III** 组合起来的题。

先将数组中的数和 `[1, n + 2]` 区间内的所有数「异或」在一起，问题就变成了：有两个数出现了「一次」，其余所有的数出现了「两次」。进而变成了 **260. 只出现一次的数字 III** 这道题。

C++ 算法代码：

```
class Solution
{
public:
    vector<int> missingTwo(vector<int>& nums)
    {
        // 1. 将所有的数异或在一起
        int tmp = 0;
        for(auto x : nums) tmp ^= x;
        for(int i = 1; i <= nums.size() + 2; i++) tmp ^= i;

        // 2. 找出 a，b 中比特位不同的那一位
        int diff = 0;
        while(1)
        {
            if(((tmp >> diff) & 1) == 1) break;
            else diff++;
        }

        // 3. 根据 diff 位的不同，将所有的数划分为两类来异或
        int a = 0, b = 0;
        for(int x : nums)
            if(((x >> diff) & 1) == 1) b ^= x;
            else a ^= x;
        for(int i = 1; i <= nums.size() + 2; i++)
            if(((i >> diff) & 1) == 1) b ^= i;
            else a ^= i;
```

```
        return {a, b};
    }
};
```

C++ 代码结果：

Java 算法代码：

```java
class Solution
{
    public int[] missingTwo(int[] nums)
    {
        // 1. 先把所有的数异或在一起
        int tmp = 0;
        for(int x : nums) tmp ^= x;
        for(int i = 1; i <= nums.length + 2; i++) tmp ^= i;

        // 2. 找出 a, b 两个数比特位不同的那一位
        int diff = 0;
        while(true)
        {
            if(((tmp >> diff) & 1) == 1) break;
            else diff++;
        }

        // 3. 将所有的数按照 diff 位不同，分两类异或
        int[] ret = new int[2];
        for(int x : nums)
            if(((x >> diff) & 1) == 1) ret[1] ^= x;
            else ret[0] ^= x;
        for(int i = 1; i <= nums.length + 2; i++)
            if(((i >> diff) & 1) == 1) ret[1] ^= i;
            else ret[0] ^= i;
        return ret;
    }
}
```

Java 运行结果：

模拟

38. 替换所有的问号（easy）

1. **题目链接：1576. 替换所有的问号**

2. **题目描述：**

> 给你一个仅包含小写英文字母和 '?' 字符的字符串 s，请你将所有的 '?' 转换为若干小写字母，使最终的字符串不包含任何 连续重复 的字符。
>
> 注意：你 不能 修改非 '?' 字符。
>
> 题目测试用例保证 除 '?' 字符 之外，不存在连续重复的字符。
>
> 在完成所有转换（可能无需转换）后返回最终的字符串。如果有多个解决方案，请返回其中任何一个。可以证明，在给定的约束条件下，答案总是存在的。
>
> 示例 1：
>
> 输入：s = "?zs"
>
> 输出："azs"
>
> 解释：该示例共有 25 种解决方案，从 "azs" 到 "yzs" 都是符合题目要求的。只有 "z" 是无效的修改，因为字符串 "zzs" 中有连续重复的两个 'z' 。
>
> 示例 2：
>
> 输入：s = "ubv?w"
>
> 输出："ubvaw"
>
> 解释：该示例共有 24 种解决方案，只有替换成 "v" 和 "w" 不符合题目要求。因为 "ubvvw" 和 "ubvww" 都包含连续重复的字符。
>
> 提示：
>
> 1 <= s.length <= 100
>
> s 仅包含小写英文字母和 '?' 字符

3. **解法（模拟）：**

算法思路：

纯模拟。从前往后遍历整个字符串，找到问号之后，就用 `a ~ z` 的每一个字符去尝试替换即可。

C++ 算法代码：

```cpp
class Solution
{
public:
    string modifyString(string s)
    {
        int n = s.size();
        for(int i = 0; i < n; i++)
        {
            if(s[i] == '?') // 替换
            {
                for(char ch = 'a'; ch <= 'z'; ch++)
                {
                    if((i == 0 || ch != s[i - 1]) && (i == n - 1 || ch != s[i + 1]))
                    {
                        s[i] = ch;
                        break;
                    }
                }
            }
        }
        return s;
    }
};
```

C++ 代码结果：

Java 算法代码：

```java
class Solution
{
    public String modifyString(String ss)
    {
        char[] s = ss.toCharArray();
        int n = s.length;
        for(int i = 0; i < n; i++)
        {
            if(s[i] == '?') // 替换
            {
                for(char ch = 'a'; ch <= 'z'; ch++)
                {
                    if((i == 0 || ch != s[i - 1]) && (i == n - 1 || ch != s[i + 1]))
                    {
                        s[i] = ch;
                        break;
                    }
                }
            }
        }
        return String.valueOf(s);
    }
}
```

Java 运行结果：

39. 提莫攻击（easy）

1. **题目链接：495. 提莫攻击**

2. **题目描述：**

> 在《英雄联盟》的世界中，有一个叫 “提莫” 的英雄。他的攻击可以让敌方英雄艾希（编者注：寒冰射手）进入中毒状态。
>
> 当提莫攻击艾希，艾希的中毒状态正好持续 duration 秒。
>
> 正式地讲，提莫在 t 发起攻击意味着艾希在时间区间 [t, t + duration - 1]（含 t 和 t + duration - 1）处于中毒状态。如果提莫在中毒影响结束 前 再次攻击，中毒状态计时器将会 重置 ，在新的攻击之

后，中毒影响将会在 duration 秒后结束。

给你一个 非递减 的整数数组 timeSeries ，其中 timeSeries[i] 表示提莫在 timeSeries[i] 秒时对艾希发起攻击，以及一个表示中毒持续时间的整数 duration 。

返回艾希处于中毒状态的 总 秒数。

示例 1：

输入：timeSeries = [1,4], duration = 2

输出：4

解释：提莫攻击对艾希的影响如下：

- 第 1 秒，提莫攻击艾希并使其立即中毒。中毒状态会维持 2 秒，即第 1 秒和第 2 秒。
- 第 4 秒，提莫再次攻击艾希，艾希中毒状态又持续 2 秒，即第 4 秒和第 5 秒。

艾希在第 1、2、4、5 秒处于中毒状态，所以总中毒秒数是 4 。

示例 2：

输入：timeSeries = [1,2], duration = 2

输出：3

解释：提莫攻击对艾希的影响如下：

- 第 1 秒，提莫攻击艾希并使其立即中毒。中毒状态会维持 2 秒，即第 1 秒和第 2 秒。
- 第 2 秒，提莫再次攻击艾希，并重置中毒计时器，艾希中毒状态需要持续 2 秒，即第 2 秒和第 3 秒。

艾希在第 1、2、3 秒处于中毒状态，所以总中毒秒数是 3 。

提示：

1 <= timeSeries.length <= 10^4

0 <= timeSeries[i], duration <= 10^7

timeSeries 按 非递减 顺序排列

3. **解法（模拟 + 分情况讨论）：**

算法思路：

模拟 + 分情况讨论。

计算相邻两个时间点的差值：

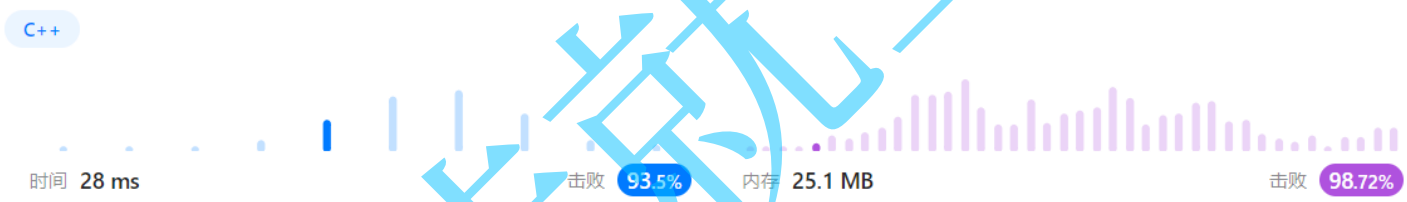
i. 如果差值大于等于中毒时间，说明上次中毒可以持续 `duration` 秒；

ii. 如果差值小于中毒时间，那么上次的中毒只能持续两者的差值。

C++ 算法代码：

```cpp
class Solution {
public:
    int findPoisonedDuration(vector<int>& timeSeries, int duration) {
        int ret = 0;
        for(int i = 1; i < timeSeries.size(); i++)
        {
            int tmp = timeSeries[i] - timeSeries[i - 1];
            if(tmp >= duration) ret += duration;
            else ret += tmp;
        }
        return ret + duration;
    }
};
```

C++ 代码结果：



Java 算法代码：

```java
class Solution {
    public int findPoisonedDuration(int[] timeSeries, int duration) {
        int ret = 0;
        for(int i = 1; i < timeSeries.length; i++) {
            int x = timeSeries[i] - timeSeries[i - 1];
            if(x >= duration) ret += duration;
            else ret += x;
        }
        return ret + duration;
    }
}
```

Java 运行结果：

40. N 字形变换（medium）

1. **题目链接：6. N 字形变换**

2. **题目描述：**

将一个给定字符串 s 根据给定的行数 numRows ，以从上往下、从左到右进行 Z 字形排列。

比如输入字符串为 "PAYPALISHIRING" 行数为 3 时，排列如下：

P A H N

A P L S I I G

Y I R

之后，你的输出需要从左往右逐行读取，产生出一个新的字符串，比如："PAHNAPLSIIGYIR"。

请你实现这个将字符串进行指定行数变换的函数：

string convert(string s, int numRows);

示例 1：

输入：s = "PAYPALISHIRING", numRows = 3

输出："PAHNAPLSIIGYIR"

示例 2：

输入：s = "PAYPALISHIRING", numRows = 4

输出："PINALSIGYAHRPI"

解释：

P I N

A L S I G

Y A H R

P I

> 示例 3：
>
> 输入：s = "A", numRows = 1
>
> 输出："A"
>
> 提示：
>
> 1 <= s.length <= 1000
>
> s 由英文字母（小写和大写）、',' 和 '.' 组成
>
> 1 <= numRows <= 1000

3. **解法（模拟 + 找规律）：**

算法思路：

找规律，用 row 代替行数，row = 4 时画出的 N 字形如下：

0			2row - 2			4row - 4
1		2row - 3		2row - 1		4row - 5	4row - 3
2	2row-4			2row		4row - 6			4row - 2
3			2row + 1			4row - 1

不难发现，数据是以 2row - 2 为一个周期进行规律变换的。将所有数替换成用周期来表示的变量：

第一行的数是：0, 2row - 2, 4row - 4；

第二行的数是：1, (2row - 2) - 1, (2row - 2) + 1, (4row - 4) - 1, (4row - 4) + 1；

第三行的数是：2, (2row - 2) - 2, (2row - 2) + 2, (4row - 4) - 2, (4row - 4) + 2；

第四行的数是：3, (2row - 2) + 3, (4row - 4) + 3。

可以观察到，第一行、第四行为差为 2row - 2 的等差数列；第二行、第三行除了第一个数取值为行数，每组下标为(2n - 1, 2n)的数围绕（2row - 2）的倍数左右取值。

以此规律，我们可以写出迭代算法。

C++ 算法代码：

```
class Solution
{
public:
    string convert(string s, int numRows)
```

```
    {
        // 处理边界情况
        if(numRows == 1) return s;

        string ret;
        int d = 2 * numRows - 2, n = s.size();

        // 1. 先处理第一行
        for(int i = 0; i < n; i += d)
            ret += s[i];

        // 2. 处理中间行
        for(int k = 1; k < numRows - 1; k++) // 枚举每一行
        {
            for(int i = k, j = d - k; i < n || j < n; i += d, j += d)
            {
                if(i < n) ret += s[i];
                if(j < n) ret += s[j];
            }
        }

        // 3. 处理最后一行
        for(int i = numRows - 1; i < n; i += d)
            ret += s[i];
        return ret;
    }
};
```

C++ 运行结果：

Java 算法代码：

```
class Solution
{
    public String convert(String s, int numRows)
    {
        // 处理一下边界情况
        if(numRows == 1) return s;
```

```
7
8           int d = 2 * numRows - 2, n = s.length();
9           StringBuilder ret = new StringBuilder();
10          // 1. 处理第一行
11          for(int i = 0; i < n; i += d)
12              ret.append(s.charAt(i));
13
14          // 2. 处理中间行
15          for(int k = 1; k < numRows - 1; k++) // 依次枚举中间行
16          {
17              for(int i = k, j = d - i; i < n || j < n; i += d, j += d)
18              {
19                  if(i < n) ret.append(s.charAt(i));
20                  if(j < n) ret.append(s.charAt(j));
21              }
22          }
23
24          // 3. 处理最后一行
25          for(int i = numRows - 1; i < n; i += d)
26              ret.append(s.charAt(i));
27          return ret.toString();
28      }
29  }
```

Java 运行结果：

41. 外观数列 (medium)

1. **题目链接：38. 外观数列**

2. **题目描述：**

> 给定一个正整数 n ，输出外观数列的第 n 项。
>
> 「外观数列」是一个整数序列，从数字 1 开始，序列中的每一项都是对前一项的描述。
>
> 你可以将其视作是由递归公式定义的数字字符串序列：
>
> countAndSay(1) = "1"
>
> countAndSay(n) 是对 countAndSay(n-1) 的描述，然后转换成另一个数字字符串。

前五项如下：

1. 1
2. 11
3. 21
4. 1211
5. 111221

第一项是数字 1

描述前一项，这个数是 1 即 “一个 1”，记作 "11"

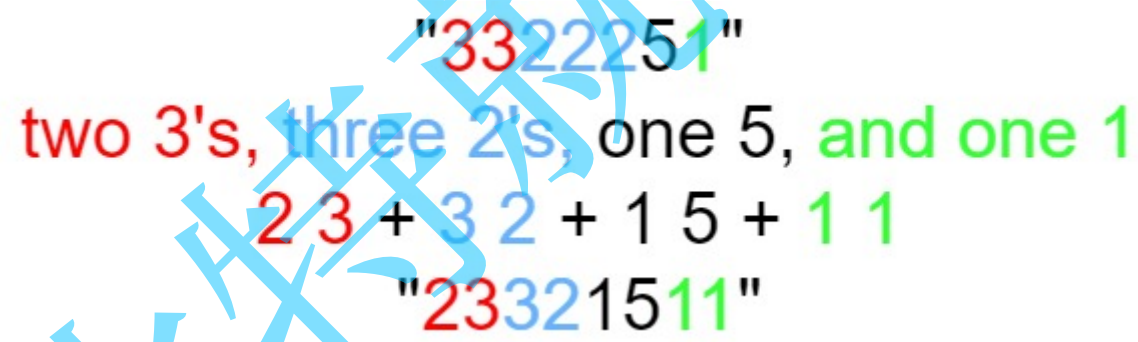
描述前一项，这个数是 11 即 “二个 1”，记作 "21"

描述前一项，这个数是 21 即 “一个 2 + 一个 1”，记作 "1211"

描述前一项，这个数是 1211 即 “一个 1 + 一个 2 + 二个 1”，记作 "111221"

要 描述 一个数字字符串，首先要将字符串分割为 最小 数量的组，每个组都由连续的最多 相同字符 组成。然后对于每个组，先描述字符的数量，然后描述字符，形成一个描述组。要将描述转换为数字字符串，先将每组中的字符数量用数字替换，再将所有描述组连接起来。

例如，数字字符串 "3322251" 的描述如下图：



示例 1：

输入：n = 1

输出："1"

解释：这是一个基本样例。

示例 2：

输入：n = 4

输出："1211"

解释：

countAndSay(1) = "1"

> countAndSay(2) = 读 "1" = 一 个 1 = "11"
>
> countAndSay(3) = 读 "11" = 二 个 1 = "21"
>
> countAndSay(4) = 读 "21" = 一 个 2 + 一 个 1 = "12" + "11" = "1211"
>
> 提示：
>
> 1 <= n <= 30

3. **解法（模拟）：**

算法思路：

所谓「外观数列」，其实只是依次统计字符串中连续且相同的字符的个数。依照题意，依次模拟即可。

C++ 算法代码：

```
class Solution
{
public:
    string countAndSay(int n)
    {
        string ret = "1";
        for(int i = 1; i < n; i++) // 解释 n - 1 次 ret 即可
        {
            string tmp;
            int len = ret.size();
            for(int left = 0, right = 0; right < len; )
            {
                while(right < len && ret[left] == ret[right]) right++;
                tmp += to_string(right - left) + ret[left];
                left = right;
            }
            ret = tmp;
        }
        return ret;
    }
};
```

C++ 运行结果：

Java 算法代码：

```java
class Solution
{
    public String countAndSay(int n)
    {
        String ret = "1";
        for(int i = 1; i < n; i++) // 解释 n - 1 次 ret 即可
        {
            StringBuilder tmp = new StringBuilder();
            int len = ret.length();
            for(int left = 0, right = 0; right < len; )
            {
                while(right < len && ret.charAt(left) == ret.charAt(right))
right++;
                tmp.append(Integer.toString(right - left));
                tmp.append(ret.charAt(left));
                left = right;
            }
            ret = tmp.toString();
        }
        return ret;
    }
}
```

Java 运行结果：

42. 数青蛙（medium）

1. **题目链接：1419. 数青蛙**

2. **题目描述：**

> 给你一个字符串 croakOfFrogs，它表示不同青蛙发出的蛙鸣声（字符串 "croak" ）的组合。由于同一时间可以有多只青蛙呱呱作响，所以 croakOfFrogs 中会混合多个 “croak” 。
>
> 请你返回模拟字符串中所有蛙鸣所需不同青蛙的最少数目。
>
> 要想发出蛙鸣 "croak"，青蛙必须 依序 输出 ‘c’ ,’ r’ ,’ o’ ,’ a’ ,’ k’ 这 5 个字母。如果没有输出全部五个字母，那么它就不会发出声音。如果字符串 croakOfFrogs 不是由若干有效的 "croak" 字符混合而成，请返回 -1 。
>
> 示例 1：
>
> 输入：croakOfFrogs = "croakcroak"
>
> 输出：1
>
> 解释：一只青蛙 “呱呱” 两次
>
> 示例 2：
>
> 输入：croakOfFrogs = "crcoakroak"
>
> 输出：2
>
> 解释：最少需要两只青蛙，“呱呱” 声用黑体标注
>
> 第一只青蛙 "crcoakroak"
>
> 第二只青蛙 "crcoakroak"
>
> 示例 3：
>
> 输入：croakOfFrogs = "croakcrook"
>
> 输出：-1
>
> 解释：给出的字符串不是 "croak" 的有效组合。
>
> 提示：
>
> 1 <= croakOfFrogs.length <= 105
>
> 字符串中的字符只有 'c', 'r', 'o', 'a' 或者 'k'

3. **解法（模拟 + 分情况讨论）**

算法思路：

模拟青蛙的叫声。

- 当遇到 `'r' 'o' 'a' 'k'` 这四个字符的时候，我们要去看看每一个字符对应的前驱字符，有没有青蛙叫出来。如果有青蛙叫出来，那就让这个青蛙接下来喊出来这个字符；如果没有，直接返回 `-1`；

- 当遇到 `'c'` 这个字符的时候，我们去看看 `'k'` 这个字符有没有青蛙叫出来。如果有，就让这个青蛙继续去喊 `'c'` 这个字符；如果没有的话，就重新搞一个青蛙。

C++ 算法代码：

```cpp
class Solution
{
public:
    int minNumberOfFrogs(string croakOfFrogs)
    {
        string t = "croak";
        int n = t.size();
        vector<int> hash(n); // 用数组来模拟哈希表

        unordered_map<char, int> index; //[x, x这个字符对应的下标]
        for(int i = 0; i < n; i++)
            index[t[i]] = i;

        for(auto ch : croakOfFrogs)
        {
            if(ch == 'c')
            {
                if(hash[n - 1] != 0) hash[n - 1]--;
                hash[0]++;
            }
            else
            {
                int i = index[ch];
                if(hash[i - 1] == 0) return -1;
                hash[i - 1]--; hash[i]++;
            }
        }

        for(int i = 0; i < n - 1; i++)
            if(hash[i] != 0)
                return -1;
        return hash[n - 1];
    }
};
```

C++ 运行结果：

C++

时间 24 ms　击败 56.29%　内存 9 MB　击败 56.64%

Java 算法代码：

```java
class Solution
{
    public int minNumberOfFrogs(String c)
    {
        char[] croakOfFrogs = c.toCharArray();
        String t = "croak";
        int n = t.length();
        int[] hash = new int[n]; // 数组模拟哈希表

        Map<Character, Integer> index = new HashMap<>(); // [x, x这个字符对应的下标]
        for(int i = 0; i < n; i++)
            index.put(t.charAt(i), i);

        for(char ch : croakOfFrogs)
        {
            if(ch == t.charAt(0))
            {
                if(hash[n - 1] != 0) hash[n - 1]--;
                hash[0]++;
            }
            else
            {
                int i = index.get(ch);
                if(hash[i - 1] == 0) return -1;
                hash[i - 1]--; hash[i]++;
            }
        }

        for(int i = 0; i < n - 1; i++)
            if(hash[i] != 0)
                return -1;

        return hash[n - 1];
    }
}
```

Java 运行结果：

分治 - 快速排序

43. 颜色分类（medium）

1. **题目链接：75. 颜色分类**

2. **题目描述：**

> 给定一个包含红色、白色和蓝色、共 n 个元素的数组 nums ，原地对它们进行排序，使得相同颜色的元素相邻，并按照红色、白色、蓝色顺序排列。
>
> 我们使用整数 0、 1 和 2 分别表示红色、白色和蓝色。
>
> 必须在不使用库的 sort 函数的情况下解决这个问题。
>
> 示例 1：
>
> 输入：nums = [2,0,2,1,1,0]
>
> 输出：[0,0,1,1,2,2]

3. **解法（快排思想 - 三指针法使数组分三块）：**

算法思路：

类比数组分两块的算法思想，这里是将数组分成三块，那么我们可以再添加一个指针，实现数组分三块。

设数组大小为 `n` ，定义三个指针 `left, cur, right`：

- `left`：用来标记 `0` 序列的末尾，因此初始化为 `-1`；
- `cur`：用来扫描数组，初始化为 `0`；
- `right`：用来标记 `2` 序列的起始位置，因此初始化为 `n`。

在 `cur` 往后扫描的过程中，保证：

- `[0, left]` 内的元素都是 `0`；
- `[left + 1, cur - 1]` 内的元素都是 `1`；

- `[cur, right - 1]` 内的元素是待定元素；
- `[right, n]` 内的元素都是 `2` 。

算法流程：

a. 初始化 `cur = 0, left = -1, right = numsSize` ；

b. 当 `cur < right` 的时候（因为 `right` 表示的是 `2` 序列的左边界，因此当 `cur` 碰到 `right` 的时候，说明已经将所有数据扫描完毕了），一直进行下面循环：

根据 `nums[cur]` 的值，可以分为下面三种情况：

i. `nums[cur] == 0` ；说明此时这个位置的元素需要在 `left + 1` 的位置上，因此交换 `left + 1` 与 `cur` 位置的元素，并且让 `left++` （指向 `0` 序列的右边界），`cur++` （为什么可以 `++` 呢，是因为 `left + 1` 位置要么是 `0` ，要么是 `cur` ，交换完毕之后，这个位置的值已经符合我们的要求，因此 `cur++` ）；

ii. `nums[cur] == 1` ；说明这个位置应该在 `left` 和 `cur` 之间，此时无需交换，直接让 `cur++` ，判断下一个元素即可；

iii. `nums[cur] == 2` ；说明这个位置的元素应该在 `right - 1` 的位置，因此交换 `right - 1` 与 `cur` 位置的元素，并且让 `right--` （指向 `2` 序列的左边界），`cur` 不变（因为交换过来的数是没有被判断过的，因此需要在下轮循环中判断）

c. 当循环结束之后：

`[0, left]` 表示 `0` 序列；

`[left + 1, right - 1]` 表示 `1` 序列；

`[right, numsSize - 1]` 表示 `2` 序列。

C++ 算法代码：

```
class Solution
{
public:
    void sortColors(vector<int>& nums)
    {
        int n = nums.size();
        int left = -1, right = n, i = 0;
        while(i < right)
        {
            if(nums[i] == 0) swap(nums[++left], nums[i++]);
            else if(nums[i] == 1) i++;
            else swap(nums[--right], nums[i]);
```

```
        }
    }
};
```

C++ 运行结果：

Java 算法代码：

```java
class Solution {
    public void swap(int[] nums, int i, int j)
    {
        int t = nums[i];
        nums[i] = nums[j];
        nums[j] = t;
    }

    public void sortColors(int[] nums) {
        int left = -1, right = nums.length, i = 0;
        while(i < right)
        {
            if(nums[i] == 0) swap(nums, ++left, i++);
            else if(nums[i] == 1) i++;
            else swap(nums, --right, i);
        }
    }
}
```

Java 运行结果：

44. 快速排序（medium）

1. **题目链接：912. 排序数组**

> 由于力扣的测试用例在不断加强，所以这里的**数组划分三块**的思想搭配**随机选择基准元素**的方法是比较优秀的。
>
> 顺便说个有趣的事：官方题解的快排代码提交后会超时~~~ **2022/12/07**

2. **题目描述：**

> 给你一个整数数组 nums，请你将该数组升序排列。
>
> 示例 1：
>
> 输入：nums = [5,2,3,1]
>
> 输出：[1,2,3,5]
>
> 示例 2：
>
> 输入：nums = [5,1,1,2,0,0]
>
> 输出：[0,0,1,1,2,5]

3. **解法（数组分三块思想 + 随机选择基准元素的快速排序）：**

算法思路：

我们在数据结构阶段学习的快速排序的思想可以知道，快排最核心的一步就是 **Partition** (分割数据)：将数据按照一个标准，分成左右两部分。

如果我们使用荷兰国旗问题的思想，将数组划分为 **左 中 右** 三部分：左边是比基准元素小的数据，中间是与基准元素相同的数据，右边是比基准元素大的数据。然后再去递归的排序左边部分和右边部分即可（可以舍去大量的中间部分）。

在处理数据量有很多重复的情况下，效率会大大提升。

算法流程：

随机选择基准算法流程：

函数设计：**int randomKey(vector<int>& nums, int left, int right)**

a. 在主函数那里种一颗随机数种子；

b. 在随机选择基准函数这里生成一个随机数；

c. 由于我们要随机产生一个基准，因此可以将随机数转换成随机下标：让随机数 % 上区间大小，然后加上区间的左边界即可。

快速排序算法主要流程：

a. 定义递归出口；

b. 利用随机选择基准函数生成一个基准元素；

c. 利用荷兰国旗思想将数组划分成三个区域；

d. 递归处理左边区域和右边区域。

C++ 算法代码：

```cpp
class Solution
{
public:
    vector<int> sortArray(vector<int>& nums)
    {
        srand(time(NULL)); // 种下一个随机数种子
        qsort(nums, 0, nums.size() - 1);
        return nums;
    }

    // 快排
    void qsort(vector<int>& nums, int l, int r)
    {
        if(l >= r) return;

        // 数组分三块
        int key = getRandom(nums, l, r);
        int i = l, left = l - 1, right = r + 1;
        while(i < right)
        {
            if(nums[i] < key) swap(nums[++left], nums[i++]);
            else if(nums[i] == key) i++;
            else swap(nums[--right], nums[i]);
        }

        // [l, left] [left + 1, right - 1] [right, r]
        qsort(nums, l, left);
        qsort(nums, right, r);
    }

    int getRandom(vector<int>& nums, int left, int right)
    {
        int r = rand();
        return nums[r % (right - left + 1) + left];
    }
};
```

C++ 运行结果：

Java 算法代码：

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

    public void qsort(int[] nums, int l, int r)
    {
        if(l >= r) return;

        // 数组分三块
        int key = nums[new Random().nextInt(r - l + 1) + l];
        int left = l - 1, right = r + 1, i = l;
        while(i < right)
        {
            if(nums[i] < key) swap(nums, ++left, i++);
            else if(nums[i] == key) i++;
            else swap(nums, --right, i);
        }

        // [l, left] [left + 1, right - 1] [rigth, r]
        qsort(nums, l, left);
        qsort(nums, right, r);
    }

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

Java 运行结果：

45. 快速选择算法（medium）

1. **题目链接：215. 数组中的第K个最大元素**

2. **题目描述：**

> 给定整数数组 nums 和整数 k，请返回数组中第 k 个最大的元素。
>
> 请注意，你需要找的是数组排序后的第 k 个最大的元素，而不是第 k 个不同的元素。
>
> 你必须设计并实现时间复杂度为 O(n) 的算法解决此问题。
>
> 示例 1:
>
> 输入: [3,2,1,5,6,4], k = 2
>
> 输出: 5
>
> 示例 2:
>
> 输入: [3,2,3,1,2,4,5,5,6], k = 4
>
> 输出: 4
>
> 提示：
>
> 1 <= k <= nums.length <= 10^5
>
> -10^4 <= nums[i] <= 10^4

3. **解法（快速选择算法）：**

算法思路：

在快排中，当我们把数组「分成三块」之后：`[l, left] [left + 1, right - 1] [right, r]` ，我们可以通过计算每一个区间内元素的「个数」，进而推断出我们要找的元素是在「哪一个区间」里面。

那么我们可以直接去「相应的区间」去寻找最终结果就好了。

C++ 算法代码：

```cpp
class Solution
{
public:
    int findKthLargest(vector<int>& nums, int k)
    {
        srand(time(NULL));
        return qsort(nums, 0, nums.size() - 1, k);
    }

    int qsort(vector<int>& nums, int l, int r, int k)
    {
        if(l == r) return nums[l];

        // 1. 随机选择基准元素
        int key = getRandom(nums, l, r);
        // 2. 根据基准元素将数组分三块
        int left = l - 1, right = r + 1, i = l;
        while(i < right)
        {
            if(nums[i] < key) swap(nums[++left], nums[i++]);
            else if(nums[i] == key) i++;
            else swap(nums[--right], nums[i]);
        }

        // 3. 分情况讨论
        int c = r - right + 1, b = right - left - 1;
        if(c >= k) return qsort(nums, right, r, k);
        else if(b + c >= k) return key;
        else return qsort(nums, l, left, k - b - c);
    }

    int getRandom(vector<int>& nums, int left, int right)
    {
        return nums[rand() % (right - left + 1) + left];
    }
};
```

C++ 代码结果：

Java 算法代码：

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

    public int qsort(int[] nums, int l, int r, int k)
    {
        if(l == r)
        {
            return nums[l];
        }

        // 1. 按照随机选择的基准元素，将数组分三块
        int key = nums[new Random().nextInt(r - l + 1) + l];
        int left = l - 1, right = r + 1, i = l;
        while(i < right)
        {
            if(nums[i] < key) swap(nums, ++left, i++);
            else if(nums[i] == key) i++;
            else swap(nums, --right, i);
        }

        // 2. 分情况讨论
        int c = r - right + 1, b = right - left - 1;
        if(c >= k) return qsort(nums, right, r, k);
        else if(c + b >= k) return key;
        else return qsort(nums, l, left, k - b - c);
    }

    public void swap(int[] nums, int i, int j)
    {
        int t = nums[i];
        nums[i] = nums[j];
        nums[j] = t;
    }

}
```

Java 运行结果：

46. 最小的 k 个数（medium）

1. **题目链接：剑指 Offer 40. 最小的k个数**

2. **题目描述：**

> 输入整数数组 arr ，找出其中最小的 k 个数。例如，输入4、5、1、6、2、7、3、8这8个数字，则最小的4个数字是1、2、3、4。
>
> 示例 1：
>
> 输入：arr = [3,2,1], k = 2
>
> 输出：[1,2] 或者 [2,1]
>
> 示例 2：
>
> 输入：arr = [0,1,2,1], k = 1
>
> 输出：[0]
>
> 限制：
>
> 0 <= k <= arr.length <= 10000
>
> 0 <= arr[i] <= 10000

3. **解法（快速选择算法）：**

算法思路：

在快排中，当我们把数组「分成三块」之后：`[l, left] [left + 1, right - 1] [right, r]` ，我们可以通过计算每一个区间内元素的「个数」，进而推断出最小的 k 个数在哪些区间里面。

那么我们可以直接去「相应的区间」继续划分数组即可。

C++ 算法代码：

```
1 class Solution
2 {
```

```cpp
public:
    vector<int> getLeastNumbers(vector<int>& nums, int k)
    {
        srand(time(NULL));
        qsort(nums, 0, nums.size() - 1, k);
        return {nums.begin(), nums.begin() + k};
    }

    void qsort(vector<int>& nums, int l, int r, int k)
    {
        if(l >= r) return;

        // 1. 随机选择一个基准元素 + 数组分三块
        int key = getRandom(nums, l, r);
        int left = l - 1, right = r + 1, i = l;
        while(i < right)
        {
            if(nums[i] < key) swap(nums[++left], nums[i++]);
            else if(nums[i] == key) i++;
            else swap(nums[--right], nums[i]);
        }
        // [l, left][left + 1, right - 1] [right, r]
        // 2. 分情况讨论
        int a = left - l + 1, b = right - left - 1;
        if(a > k) qsort(nums, l, left, k);
        else if(a + b >= k) return;
        else qsort(nums, right, r, k - a - b);
    }

    int getRandom(vector<int>& nums, int l, int r)
    {
        return nums[rand() % (r - l + 1) + l];
    }
};
```

C++ 代码结果：

Java 算法代码：

```java
class Solution
{
    public int[] getLeastNumbers(int[] nums, int k)
    {
        qsort(nums, 0, nums.length - 1, k);

        int[] ret = new int[k];
        for(int i = 0; i < k; i++)
            ret[i] = nums[i];
        return ret;
    }

    public void qsort(int[] nums, int l, int r, int k)
    {
        if(l >= r) return;

        // 1. 随机选择一个基准元素 + 数组分三块
        int key = nums[new Random().nextInt(r - l + 1) + l];
        int left = l - 1, right = r + 1, i = l;
        while(i < right)
        {
            if(nums[i] < key) swap(nums, ++left, i++);
            else if(nums[i] == key) i++;
            else swap(nums, --right, i);
        }

        // 2. 分类讨论
        int a = left - l + 1, b = right - left - 1;
        if(a > k) qsort(nums, l, left, k);
        else if(a + b >= k) return;
        else qsort(nums, right, r, k - a - b);
    }

    public void swap(int[] nums, int i, int j)
    {
        int t = nums[i];
        nums[i] = nums[j];
        nums[j] = t;
    }
}

```

Java 运行结果：

分治 - 归并排序

47. 归并排序（medium）

1. **题目链接：912. 排序数组**

2. **题目描述：**

> 给你一个整数数组 nums，请你将该数组升序排列。
>
> 示例 1：
>
> 输入：nums = [5,2,3,1]
>
> 输出：[1,2,3,5]
>
> 示例 2：
>
> 输入：nums = [5,1,1,2,0,0]
>
> 输出：[0,0,1,1,2,5]

3. **解法（归并排序）：**

算法思路：

归并排序的流程充分的体现了「分而治之」的思想，大体过程分为两步：

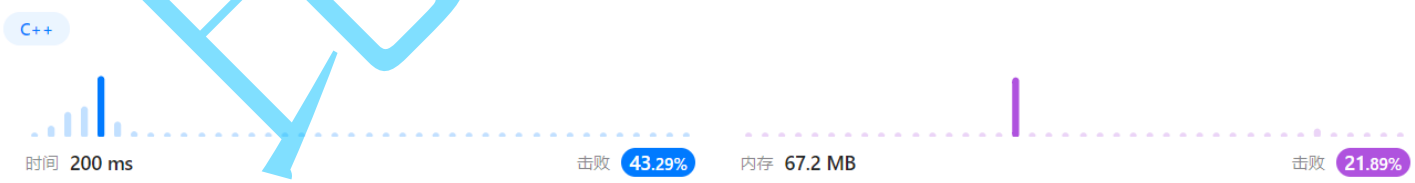
- 分：将数组一分为二为两部分，一直分解到数组的长度为 `1`，使整个数组的排序过程被分为「左半部分排序」+「右半部分排序」；
- 治：将两个较短的「有序数组合并成一个长的有序数组」，一直合并到最初的长度。

C++ 算法代码：

```
class Solution
{
    vector<int> tmp;

public:
    vector<int> sortArray(vector<int>& nums)
    {
        tmp.resize(nums.size());
        mergeSort(nums, 0, nums.size() - 1);
```

```
        return nums;
    }

    void mergeSort(vector<int>& nums, int left, int right)
    {
        if(left >= right) return;

        // 1. 选择中间点划分区间
        int mid = (left + right) >> 1;
        // [left, mid] [mid + 1, right]

        // 2. 把左右区间排序
        mergeSort(nums, left, mid);
        mergeSort(nums, mid + 1, right);

        // 3. 合并两个有序数组
        int cur1 = left, cur2 = mid + 1, i = 0;
        while(cur1 <= mid && cur2 <= right)
            tmp[i++] = nums[cur1] <= nums[cur2] ? nums[cur1++] :
nums[cur2++];
        // 处理没有遍历完的数组
        while(cur1 <= mid) tmp[i++] = nums[cur1++];
        while(cur2 <= right) tmp[i++] = nums[cur2++];

        // 4. 还原
        for(int i = left; i <= right; i++)
            nums[i] = tmp[i - left];
    }
};
```

C++ 运行结果：



Java 算法代码：

```
class Solution
{
    int[] tmp;
    public int[] sortArray(int[] nums)
    {
```

```java
        tmp = new int[nums.length];
        mergeSort(nums, 0, nums.length - 1);
        return nums;
    }

    public void mergeSort(int[] nums, int left, int right)
    {
        if(left >= right) return;

        // 1. 根据中间点划分区间
        int mid = (left + right) / 2;
        // [left, mid] [mid + 1, right]

        // 2. 先把左右区间排个序
        mergeSort(nums, left, mid);
        mergeSort(nums, mid + 1, right);

        // 3. 合并两个有序数组
        int cur1 = left, cur2 = mid + 1, i = 0;
        while(cur1 <= mid && cur2 <= right)
            tmp[i++] = nums[cur1] <= nums[cur2] ? nums[cur1++] : nums[cur2++];
        // 处理没有遍历完的数组
        while(cur1 <= mid) tmp[i++] = nums[cur1++];
        while(cur2 <= right) tmp[i++] = nums[cur2++];

        // 4. 还原
        for(int j = left; j <= right; j++)
            nums[j] = tmp[j - left];
    }
}
```

Java 运行结果：

48. 数组中的逆序对（hard）

1. **题目链接：剑指 Offer 51. 数组中的逆序对**
2. **题目描述：**

> 在数组中的两个数字，如果前面一个数字大于后面的数字，则这两个数字组成一个逆序对。输入一个数组，求出这个数组中的逆序对的总数。
>
> 示例 1:
>
> 输入: [7,5,6,4]
>
> 输出: 5

3. **解法（利用归并排序的过程 --- 分治）：**

算法思路：

用归并排序求逆序数是很经典的方法，主要就是在归并排序的合并过程中统计出逆序对的数量，也就是在合并两个有序序列的过程中，能够快速求出逆序对的数量。

我们将这个问题分解成几个小问题，逐一破解这道题。

注意：默认都是升序，如果掌握升序的话，降序的归并过程也是可以解决问题的。

- 先解决第一个问题，为什么可以利用归并排序？

如果我们将数组从中间划分成两个部分，那么我们可以将逆序对产生的方式划分成三组：

- **逆序对中两个元素：全部从左数组中选择**
- **逆序对中两个元素：全部从右数组中选择**
- **逆序对中两个元素：一个选左数组另一个选右数组**

根据**排列组合的分类相加原理**，三种种情况下产生的逆序对的总和，正好等于总的逆序对数量。

而这个思路正好匹配归并排序的过程：

- **先排序左数组；**
- **再排序右数组；**
- **左数组和右数组合二为一。**

因此，我们可以利用归并排序的过程，**先求出左半数组中逆序对的数量，再求出右半数组中逆序对的数量，最后求出一个选择左边，另一个选择右边情况下逆序对的数量**，三者相加即可。

- 解决第二个问题，为什么要这么做？

在归并排序合并的过程中，我们得到的是两个**有序的**数组。我们是可以利用数组的**有序性**，快速统计出逆序对的数量，而不是将所有情况都枚举出来。

- 最核心的问题，如何在合并两个有序数组的过程中，统计出逆序对的数量?

合并两个有序序列时求逆序对的方法有两种：

1. 快速统计出某个数前面有多少个数比它大；
2. 快速统计出某个数后面有多少个数比它小；

方法一：快速统计出某个数前面有多少个数比它大

通过一个示例来演示方法一：

假定已经有两个已经有序的序列以及辅助数组 **left = [5, 7, 9] right = [4, 5, 8] help = [ ]**，通过合并两个有序数组的过程，来求得逆序对的数量：

规定如下定义来叙述过程:

cur1 遍历 **left** 数组，**cur2** 遍历 **right** 数组

ret 记录逆序对的数量

第一轮循环：

left[cur1] > right[cur2]，由于两个数组都是升序的，那么我们可以断定，此刻 **left** 数组中 **[cur1, 2]** 区间内的 **3** 个元素均可与 **right[cur2]** 的元素构成逆序对，因此可以累加逆序对的数量 **ret += 3**，并且将 **right[cur2]** 加入到辅助数组中，**cur2++** 遍历下一个元素。

第一轮循环结束后：**left = [5, 7, 9] right = [x, 5, 8] help = [4] ret = 3 cur1 = 0 cur2 = 1**

第二轮循环：

left[cur1] == right[cur2]，因为 **right[cur2]** 可能与 **left** 数组中往后的元素构成逆序对，因此我们需要将 **left[cur1]** 加入到辅助数组中去，此时没有产生逆序对，不更新 **ret**。

第二轮循环结束后：**left = [x, 7, 9] right = [x, 5, 8] help = [4, 5] ret = 3 cur1 = 1 cur2 = 1**

第三轮循环：

left[cur1] > right[cur2]，与第一轮循环相同，此刻 **left** 数组中**[cur1, 2]** 区间内的 **2** 个元素均可与 **right[cur2]** 的元素构成逆序对，更新 **ret** 的值为 **ret += 2**，并且将 **right[cur2]** 加入到辅助数组中去，**cur2++** 遍历下一个元素。

第三轮循环结束后：**left = [x, 7, 9] right = [x, x, 8] help = [4, 5, 5] ret = 5 cur1 = 1 cur2 = 2**

第四轮循环：

left[cur1] < right[cur2]，由于两个数组都是升序的，因此我们可以确定 **left[cur1]** 比 **right** 数组中的所有元素都要小。**left[cur1]** 这个元素是不可能与 **right** 数组中的元素构成逆序对。因此，大胆

的将 **left[cur1]** 这个元素加入到辅助数组中去，不更细 **ret** 的值。

第四轮循环结束后：**left = [x, x, 9] right = [x, x, 8] help = [4, 5, 5, 7] ret = 5 cur1 = 2 cur2 = 2**

第五轮循环：

left[cur1] > right[cur2]，与第一、第三轮循环相同。此时 left 数组内的 1 个元素能与 right[cur2] 构成逆序对，更新 ret 的值，并且将 right[cur2] 加入到辅助数组中去。

第五轮循环结束后：**left = [x, x, 9] right = [x, x, x] help = [4, 5, 5, 7, 8] ret = 6 cur1 = 2 cur2 = 2**

处理剩余元素：

- 如果是左边出现剩余，说明左边剩下的所有元素都是比右边元素大的，但是它们都是已经被计算过的（我们以右边的元素为基准的），因此不会产生逆序对，仅需归并排序即可。
- 如果是右边出现剩余，说明右边剩下的元素都是比左边大的，不符合逆序对的定义，因此也不需要处理，仅需归并排序即可。

整个过程只需将两个数组遍历一遍即可，时间复杂度为 **O(N)**。

由上述过程我们可以得出方法一统计逆序对的关键点：

在**合并有序数组**的时候，遇到**左数组当前元素 > 右数组当前元素**时，我们可以通过计算**左数组中剩余元素**的长度，就可快速求出**右数组当前元素前面有多少个数比它大**，对比解法一中一个一个枚举逆序对效率快了许多。

方法二：快速统计出某个数后面有多少个数比它小

依旧通过一个示例来演示方法二：

假定已经有两个已经有序的序列以及辅助数组 **left = [5, 7, 9] right = [4, 5, 8] help = [ ]**，通过合并两个有序数组的过程，来求得逆序对的数量：

规定如下定义来叙述过程:

cur1 遍历 **left** 数组，**cur2** 遍历 **right** 数组

ret 记录逆序对的数量

第一轮循环：

left[cur1] > right[cur2]，先不要着急统计，因为我们要找的是当前元素后面有多少比它小的，这里虽然出现了一个，但是 **right** 数组中依旧还可能有其余比它小的。因此此时仅将 **right[cur2]** 加入到辅助数组中去，并且将 **cur2++**。

第一轮循环结束后：**left = [5, 7, 9] right = [x, 5, 8] help = [4] ret = 0 cur1 = 0 cur2 = 1**

第二轮循环：

left[cur1] == right[cur2]，由于两个数组都是升序，这个时候对于元素 **left[cur1]** 来说，我们已经可以断定 **right** 数组中 **[0, cur2)** 左闭右开区间上的元素都是比它小的。因此此时可以统计逆序对的数量 **ret += cur2 - 0**，并且将 **left[cur1]** 放入到辅助数组中去，**cur1++** 遍历下一个元素。

第二轮循环结束后：**left = [x, 7, 9] right = [x, 5, 8] help = [4, 5] ret = 1 cur1 = 1 cur2 = 1**

第三轮循环：

left[cur1] > right[cur2]，与第一轮循环相同，直接将 **right[cur2]** 加入到辅助数组中去，**cur2++** 遍历下一个元素。

第三轮循环结束后：**left = [x, 7, 9] right = [x, x, 8] help = [4, 5, 5] ret = 1 cur1 = 1 cur2 = 2**

第四轮循环：

left[cur1] < right[cur2]，由于两个数组都是升序的，这个时候对于元素 **left[cur1]** 来说，我们依旧已经可以断定 **right** 数组中 **[0, cur2)** 左闭右开区间上的元素都是比它小的。因此此时可以统计逆序对的数量 **ret += cur2 - 0**，并且将 **left[cur1]** 放入到辅助数组中去，**cur1++** 遍历下一个元素。

第四轮循环结束后：**left = [9] right = [8] help = [4, 5, 5, 7] ret = 3 cur1 = 2 cur2 = 2**

第五轮循环：

left[cur1] > right[cur2]，与第一、第三轮循环相同。直接将 **right[cur2]** 加入到辅助数组中去，**cur2++** 遍历下一个元素。

第五轮循环结束后：**left = [x, x, 9] right = [x, x, x] help = [4, 5, 5, 7, 8] ret = 3 cur1 = 2 cur2 = 2**

处理剩余元素：

- 如果是左边出现剩余，说明左边剩下的所有元素都是比右边元素大的，但是相比较于方法一，逆序对的数量是没有统计过的。因此，我们需要统计 ret 的值：
  - 设左边数组剩余元素的个数为 **leave**
  - **ret += leave * （cur2 - 0）**

  对于本题来说，处理剩余元素的时候， **left** 数组剩余 **1** 个元素，**cur2 - 0 = 3**，因此 **ret** 需要类加上 **3**，结果为 **6**。与**方法一**求得的结果相同。
- 如果是右边出现剩余，说明右边剩下的元素都是比左边大的，不符合逆序对的定义，因此也不需要处理，仅需归并排序即可。

整个过程只需将两个数组遍历一遍即可，时间复杂度依旧为 **O(N)**。

由上述过程我们可以得出**方法二**统计逆序对的关键点：

在**合并有序数组**的时候，遇到**左数组当前元素 <= 右数组当前元素**时，我们可以通过计算**右数组已经遍历过的元素**的长度，快速求出**左数组当前元素后面有多少个数比它大**。

但是需要注意的是，在处理剩余元素的时候，方法二还需要统计逆序对的数量。

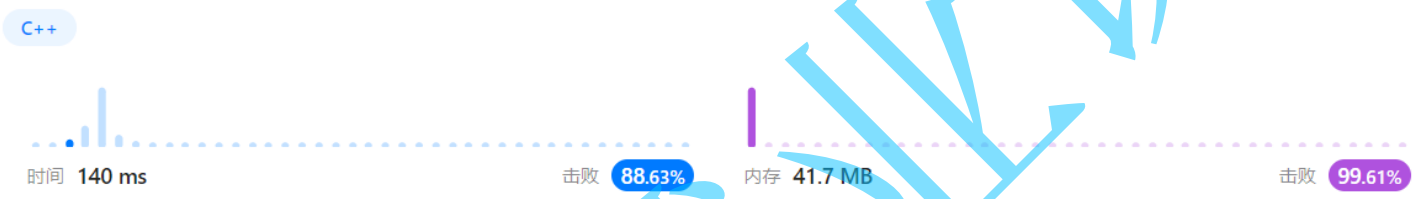
升序的版本

C++ 算法代码：

```
class Solution
{
    int tmp[50010];
public:
    int reversePairs(vector<int>& nums)
    {
        return mergeSort(nums, 0, nums.size() - 1);
    }

    int mergeSort(vector<int>& nums, int left, int right)
    {
        if(left >= right) return 0;

        int ret = 0;
        // 1. 找中间点，将数组分成两部分
        int mid = (left + right) >> 1;
        // [left, mid][mid + 1, right]

        // 2. 左边的个数 + 排序 + 右边的个数 + 排序
        ret += mergeSort(nums, left, mid);
        ret += mergeSort(nums, mid + 1, right);

        // 3. 一左一右的个数
        int cur1 = left, cur2 = mid + 1, i = 0;
        while(cur1 <= mid && cur2 <= right) // 升序的时候
        {
            if(nums[cur1] <= nums[cur2])
            {
                tmp[i++] = nums[cur1++];
            }
            else
            {
                ret += mid - cur1 + 1;
                tmp[i++] = nums[cur2++];
```

```
            }
        }

        // 4. 处理一下排序
        while(cur1 <= mid) tmp[i++] = nums[cur1++];
        while(cur2 <= right) tmp[i++] = nums[cur2++];
        for(int j = left; j <= right; j++)
            nums[j] = tmp[j - left];

        return ret;
    }
};
```

C++ 运行结果：



Java 算法代码：

```java
class Solution
{
    int[] tmp;
    public int reversePairs(int[] nums)
    {
        int n = nums.length;
        tmp = new int[n];
        return mergeSort(nums, 0, n - 1);
    }

    public int mergeSort(int[] nums, int left, int right)
    {
        if(left >= right) return 0;

        int ret = 0;
        // 1. 选择一个中间点，将数组划分成两部分
        int mid = (left + right) / 2;
        // [left, mid] [mid + 1, right]

        // 2. 左半部分的个数 + 排序 + 右半部分的个数 + 排序
        ret += mergeSort(nums, left, mid);
```

```
        ret += mergeSort(nums, mid + 1, right);

        // 3. 一左一右的个数
        int cur1 = left, cur2 = mid + 1, i = 0;
        while(cur1 <= mid && cur2 <= right) // 升序版本
        {
            if(nums[cur1] <= nums[cur2])
            {
                tmp[i++] = nums[cur1++];
            }
            else
            {
                ret += mid - cur1 + 1;
                tmp[i++] = nums[cur2++];
            }
        }

        // 4. 处理一下排序
        while(cur1 <= mid) tmp[i++] = nums[cur1++];
        while(cur2 <= right) tmp[i++] = nums[cur2++];
        for(int j = left; j <= right; j++)
            nums[j] = tmp[j - left];

        return ret;
    }
}
```

Java 运行结果：

降序的版本

C++ 算法代码：

```
class Solution
{
    int tmp[50010];
public:
    int reversePairs(vector<int>& nums)
    {
```

```cpp
        return mergeSort(nums, 0, nums.size() - 1);
    }

    int mergeSort(vector<int>& nums, int left, int right)
    {
        if(left >= right) return 0;

        int ret = 0;
        // 1. 找中间点，将数组分成两部分
        int mid = (left + right) >> 1;
        // [left, mid][mid + 1, right]

        // 2. 左边的个数 + 排序 + 右边的个数 + 排序
        ret += mergeSort(nums, left, mid);
        ret += mergeSort(nums, mid + 1, right);

        // 3. 一左一右的个数
        int cur1 = left, cur2 = mid + 1, i = 0;
        while(cur1 <= mid && cur2 <= right) // 降序的版本
        {
            if(nums[cur1] <= nums[cur2])
            {
                tmp[i++] = nums[cur2++];
            }
            else
            {
                ret += right - cur2 + 1;
                tmp[i++] = nums[cur1++];
            }
        }

        // 4. 处理一下排序
        while(cur1 <= mid) tmp[i++] = nums[cur1++];
        while(cur2 <= right) tmp[i++] = nums[cur2++];
        for(int j = left; j <= right; j++)
            nums[j] = tmp[j - left];

        return ret;
    }
};
```

C++ 运行结果：

C++

时间 140 ms 击败 88.63%

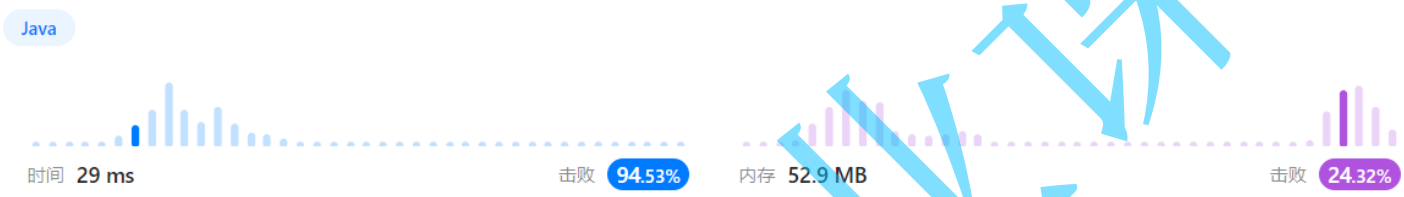
内存 41.7 MB 击败 99.61%

Java 算法代码：

```java
class Solution
{
    int[] tmp;
    public int reversePairs(int[] nums)
    {
        int n = nums.length;
        tmp = new int[n];
        return mergeSort(nums, 0, n - 1);
    }

    public int mergeSort(int[] nums, int left, int right)
    {
        if(left >= right) return 0;

        int ret = 0;
        // 1. 选择一个中间点，将数组划分成两部分
        int mid = (left + right) / 2;
        // [left, mid] [mid + 1, right]

        // 2. 左半部分的个数 + 排序 + 右半部分的个数 + 排序
        ret += mergeSort(nums, left, mid);
        ret += mergeSort(nums, mid + 1, right);

        // 3. 一左一右的个数
        int cur1 = left, cur2 = mid + 1, i = 0;
        while(cur1 <= mid && cur2 <= right) // 降序的版本
        {
            if(nums[cur1] <= nums[cur2])
            {
                tmp[i++] = nums[cur2++];
            }
            else
            {
                ret += right - cur2 + 1;
                tmp[i++] = nums[cur1++];
            }
        }
```

```
38
39          // 4. 处理一下排序
40          while(cur1 <= mid) tmp[i++] = nums[cur1++];
41          while(cur2 <= right) tmp[i++] = nums[cur2++];
42          for(int j = left; j <= right; j++)
43              nums[j] = tmp[j - left];
44
45          return ret;
46      }
47  }
```

Java 运行结果：



49. 计算右侧小于当前元素的个数（hard）

1. **题目链接：315. 计算右侧小于当前元素的个数**

2. **题目描述：**

> 给你一个整数数组 nums ，按要求返回一个新数组 counts 。数组 counts 有该性质： counts[i] 的值是 nums[i] 右侧小于 nums[i] 的元素的数量。
>
> 示例 1：
>
> 输入：nums = [5,2,6,1]
>
> 输出：[2,1,1,0]
>
> 解释：
>
> 5 的右侧有 2 个更小的元素 (2 和 1)
>
> 2 的右侧仅有 1 个更小的元素 (1)
>
> 6 的右侧有 1 个更小的元素 (1)
>
> 1 的右侧有 0 个更小的元素

3. **解法（归并排序）：**

算法思路：

这一道题的解法与 **求数组中的逆序对** 的解法是类似的，但是这一道题要求的不是求总的个数，而是要返回一个数组，记录**每一个元素**的右边有多少个元素比自己小。

但是在我们归并排序的过程中，元素的下标是会跟着变化的，因此我们需要一个辅助数组，来将**数组元素**和**对应的下标**绑定在一起归并，也就是再归并元素的时候，顺势将下标也转移到对应的位置上。

由于我们要快速统计出某一个元素后面有多少个比它小的，因此我们可以利用求逆序对的第二种方法。

算法流程：

- 创建两个全局的数组：

  vector<int> index：记录下标
  vector<int> ret：记录结果

  index 用来与原数组中对应位置的元素绑定，**ret** 用来记录每个位置统计出来的逆序对的个数。

- **countSmaller()** 主函数：
  a. 计算 **nums** 数组的大小为 **n**；
  b. 初始化定义的两个全局的数组；
    i. 为两个数组开辟大小为 **n** 的空间
    ii. **index** 初始化为数组下标；
    iii. **ret** 初始化为 **0**
  c. 调用 **mergeSort()** 函数，并且返回 **ret** 结果数组。

- **void mergeSort(vector<int>& nums, int left, int right)** 函数：

  函数设计：通过修改全局的数组 **ret**，统计出每一个位置对应的逆序对的数量，并且排序；

  无需返回值，因为直接对全局变量修改，当函数结束的时候，全局变量已经被修改成最后的结果。

- **mergeSort() 函数流程：**
  a. 定义递归出口：left >= right 时，直接返回；
  b. 划分区间：根据中点 **mid**，将区间划分为 **[left, mid]** 和 **[mid + 1, right]**；
  c. 统计左右两个区间逆序对的数量：
    i. 统计左边区间 **[left, mid]** 中每个元素对应的逆序对的数量到 **ret** 数组中，并排序；
    ii. 统计右边区间 **[mid + 1, right]** 中每个元素对应的逆序对的数量到 **ret** 数组中，并排序。

d. 合并左右两个有序区间，并且统计出逆序对的数量：

i. 创建两个大小为 **right - left + 1** 大小的辅助数组：

- **numsTmp**： 排序用的辅助数组；
- **indexTmp**：处理下标用的辅助数组。

ii. 初始化遍历数组的指针：**cur1 = left**（遍历左半部分数组）**cur2 = mid + 1**（遍历右半边数组）**dest = 0**（遍历辅助数组）**curRet**（记录合并时产生的逆序对的数量）；

iii. 循环合并区间：

- 当 **nums[cur1] <= nums[cur2]** 时：
  - 说明此时 **[mid + 1, cur2)** 之间的元素都是小于 **nums[cur1]** 的，需要累加到 **ret** 数组的 **indext[cur1]** 位置上（因为 **index** 存储的是元素对应位置在原数组中的下标）
  - 归并排序：不仅要将数据放在对应的位置上，也要将数据对应的坐标也放在对应的位置上，使数据与原始的下标绑定在一起移动。
- 当 **nums[cur1] > nums[cur2]** 时，无需统计，直接归并，注意 **index** 也要跟着归并。

iv. 处理归并排序中剩余的元素；

- 当左边有剩余的时候，**还需要统计逆序对的数量**；
- 当右边还有剩余的时候，无需统计，直接归并。

v. 将辅助数组的内容替换到原数组中去；

C++ 算法代码：

```cpp
class Solution
{
    vector<int> ret;
    vector<int> index; // 记录 nums 中当前元素的原始下标
    int tmpNums[500010];
    int tmpIndex[500010];
public:
    vector<int> countSmaller(vector<int>& nums)
    {
        int n = nums.size();
        ret.resize(n);
        index.resize(n);

        // 初始化一下 index 数组
        for(int i = 0; i < n; i++)
            index[i] = i;

```

```
18          mergeSort(nums, 0, n - 1);
19          return ret;
20      }
21
22      void mergeSort(vector<int>& nums, int left, int right)
23      {
24          if(left >= right) return;
25
26          // 1. 根据中间元素，划分区间
27          int mid = (left + right) >> 1;
28          // [left, mid]  [mid + 1, right]
29
30          // 2. 先处理左右两部分
31          mergeSort(nums, left, mid);
32          mergeSort(nums, mid + 1, right);
33
34          // 3. 处理一左一右的情况
35          int cur1 = left, cur2 = mid + 1, i = 0;
36          while(cur1 <= mid && cur2 <= right) // 降序
37          {
38              if(nums[cur1] <= nums[cur2])
39              {
40                  tmpNums[i] = nums[cur2];
41                  tmpIndex[i++] = index[cur2++];
42              }
43              else
44              {
45                  ret[index[cur1]] += right - cur2 + 1; // 重点
46                  tmpNums[i] = nums[cur1];
47                  tmpIndex[i++] = index[cur1++];
48              }
49          }
50
51          // 4. 处理剩下的排序过程
52          while(cur1 <= mid)
53          {
54              tmpNums[i] = nums[cur1];
55              tmpIndex[i++] = index[cur1++];
56          }
57          while(cur2 <= right)
58          {
59              tmpNums[i] = nums[cur2];
60              tmpIndex[i++] = index[cur2++];
61          }
62          for(int j = left; j <= right; j++)
63          {
64              nums[j] = tmpNums[j - left];
```

```
            index[j] = tmpIndex[j - left];
        }
    }
};
```

C++ 运行结果：

Java 算法代码：

```java
class Solution
{
    int[] ret;
    int[] index; // 标记 nums 中当前元素的原始下标
    int[] tmpIndex;
    int[] tmpNums;

    public List<Integer> countSmaller(int[] nums)
    {
        int n = nums.length;
        ret = new int[n];
        index = new int[n];
        tmpIndex = new int[n];
        tmpNums = new int[n];

        // 初始化 index 数组
        for(int i = 0; i < n; i++)
            index[i] = i;

        mergeSort(nums, 0, n - 1);
        List<Integer> l = new ArrayList<Integer>();
        for(int x : ret)
            l.add(x);
        return l;
    }

    public void mergeSort(int[] nums, int left, int right)
    {
        if(left >= right) return;
```

```
30
31          // 1. 根据中间元素划分区间
32          int mid = (left + right) / 2;
33          // [left, mid] [mid + 1, right]
34
35          // 2. 处理左右两个区间
36          mergeSort(nums, left, mid);
37          mergeSort(nums, mid + 1, right);
38
39          // 3. 处理一左一右的情况
40          int cur1 = left, cur2 = mid + 1, i = 0;
41          while(cur1 <= mid && cur2 <= right) // 降序排序
42          {
43              if(nums[cur1] <= nums[cur2])
44              {
45                  tmpNums[i] = nums[cur2];
46                  tmpIndex[i++] = index[cur2++];
47              }
48              else
49              {
50                  ret[index[cur1]] += right - cur2 + 1; // 重点
51                  tmpNums[i] = nums[cur1];
52                  tmpIndex[i++] = index[cur1++];
53              }
54          }
55
56          // 4. 处理剩余的排序工作
57          while(cur1 <= mid)
58          {
59              tmpNums[i] = nums[cur1];
60              tmpIndex[i++] = index[cur1++];
61          }
62          while(cur2 <= right)
63          {
64              tmpNums[i] = nums[cur2];
65              tmpIndex[i++] = index[cur2++];
66          }
67          for(int j = left; j <= right; j++)
68          {
69              nums[j] = tmpNums[j - left];
70              index[j] = tmpIndex[j - left];
71          }
72      }
73  }
74
```

Java 运行结果：

50. 翻转对（hard）

1. **题目链接：493. 翻转对**

2. **题目描述：**

> 给定一个数组 nums ，如果 i < j 且 nums[i] > 2*nums[j] 我们就将 (i, j) 称作一个重要翻转对。你需要返回给定数组中的重要翻转对的数量。
>
> 示例 1:
>
> 输入: [1,3,2,3,1]
>
> 输出: 2

题目解析：

翻转对和逆序对的定义大同小异，逆序对是前面的数要大于后面的数。而翻转对是前面的一个数要大于后面某个数的两倍。因此，我们依旧可以用归并排序的思想来解决这个问题。

3. **解法（归并排序）：**

算法思路：

大思路与求逆序对的思路一样，就是利用归并排序的思想，将求整个数组的翻转对的数量，转换成三部分：**左半区间翻转对的数量**，**右半区间翻转对的数量**，**一左一右选择时翻转对的数量**。重点就是在合并区间过程中，如何计算出翻转对的数量。

与上个问题不同的是，上一道题我们可以一边合并一遍计算，但是这道题要求的是左边元素大于右边元素的两倍，如果我们直接合并的话，是无法快速计算出翻转对的数量的。

例如 **left = [4, 5, 6] right = [3, 4, 5]** 时，如果是归并排序的话，我们需要计算 **left** 数组中有多少个能与 **3** 组成翻转对。但是我们要遍历到最后一个元素 **6** 才能确定，时间复杂度较高。

因此我们需要在**归并排序之前**完成翻转对的统计。

下面依旧以一个示例来模仿两个有序序列如何快速求出翻转对的过程：

假定已经有两个已经有序的序列 **left = [4, 5, 6] right = [1, 2, 3]** 。

用两个指针 **cur1 cur2** 遍历两个数组。

- 对于任意给定的 **left[cur1]** 而言，我们不断地向右移动 **cur2**，直到 **left[cur1] <= 2 * right[cur2]**。此时对于 **right** 数组而言，**cur2** 之前的元素全部都可以与 **left[cur1]** 构成翻转对。
- 随后，我们再将 **cur1** 向右移动一个单位，此时 **cur2** 指针并不需要回退（因为 **left** 数组是升序的）依旧往右移动直到 **left[cur1] <= 2 * right[cur2]**。不断重复这样的过程，就能够求出所有左右端点分别位于两个子数组的翻转对数目。

由于两个指针最后都是**不回退**的的扫描到数组的结尾，因此两个有序序列求出翻转对的时间复杂度是 **O(N)**。

综上所述，我们可以利用归并排序的过程，将求一个数组的翻转对转换成求 左数组的翻转对数量 + 右数组中翻转对的数量 + 左右数组合并时翻转对的数量。

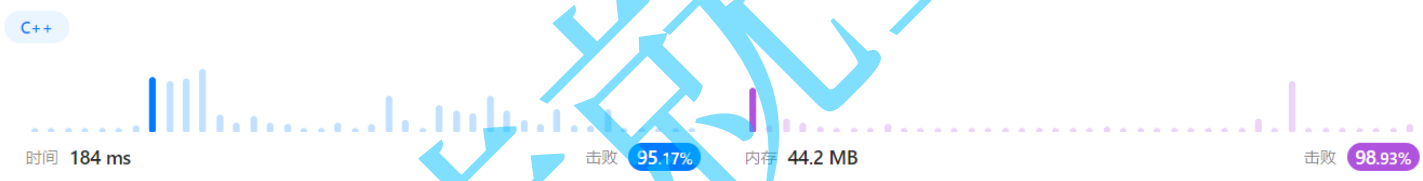
降序版本

C++ 算法代码：

```
class Solution
{
    int tmp[50010];
public:
    int reversePairs(vector<int>& nums)
    {
        return mergeSort(nums, 0, nums.size() - 1);
    }

    int mergeSort(vector<int>& nums, int left, int right)
    {
        if(left >= right) return 0;

        int ret = 0;
        // 1. 先根据中间元素划分区间
        int mid = (left + right) >> 1;
        // [left, mid] [mid + 1, right]

        // 2. 先计算左右两侧的翻转对
        ret += mergeSort(nums, left, mid);
        ret += mergeSort(nums, mid + 1, right);

        // 3. 先计算翻转对的数量
        int cur1 = left, cur2 = mid + 1, i = left;
        while(cur1 <= mid) // 降序的情况
        {
            while(cur2 <= right && nums[cur2] >= nums[cur1] / 2.0) cur2++;
            if(cur2 > right)
```

```
29                 break;
30             ret += right - cur2 + 1;
31             cur1++;
32         }
33 
34         // 4. 合并两个有序数组
35         cur1 = left, cur2 = mid + 1;
36         while(cur1 <= mid && cur2 <= right)
37             tmp[i++] = nums[cur1] <= nums[cur2] ? nums[cur2++] : nums[cur1++];
38         while(cur1 <= mid) tmp[i++] = nums[cur1++];
39         while(cur2 <= right) tmp[i++] = nums[cur2++];
40 
41         for(int j = left; j <= right; j++)
42             nums[j] = tmp[j];
43 
44         return ret;
45     }
46 };
```

C++ 运行结果：



Java 算法代码：

```
1  class Solution
2  {
3      int[] tmp;
4      public int reversePairs(int[] nums)
5      {
6          int n = nums.length;
7          tmp = new int[n];
8 
9          return mergeSort(nums, 0, n - 1);
10     }
11 
12     public int mergeSort(int[] nums, int left, int right)
13     {
14         if(left >= right) return 0;
15 
16         int ret = 0;
```

```java
        // 1. 根据中间元素，将区间分成两部分
        int mid = (left + right) / 2;
        // [left, mid] [mid + 1, right]

        // 2. 求出左右两个区间的翻转对
        ret += mergeSort(nums, left, mid);
        ret += mergeSort(nums, mid + 1, right);

        // 3. 处理一左一右 - 先计算翻转对
        int cur1 = left, cur2 = mid + 1, i = left;
        // 降序版本
        while(cur1 <= mid)
        {
            while(cur2 <= right && nums[cur2] >= nums[cur1] / 2.0) cur2++;
            if(cur2 > right)
                break;
            ret += right - cur2 + 1;
            cur1++;
        }

        // 4. 合并两个有序数组
        cur1 = left; cur2 = mid + 1;
        while(cur1 <= mid && cur2 <= right)
            tmp[i++] = nums[cur1] <= nums[cur2] ? nums[cur2++] : nums[cur1++];
        while(cur1 <= mid) tmp[i++] = nums[cur1++];
        while(cur2 <= right) tmp[i++] = nums[cur2++];

        for(int j = left; j <= right; j++)
            nums[j] = tmp[j];

        return ret;
    }
}
```

Java 运行结果：

升序版本

C++ 算法代码：

```cpp
class Solution
{
    int tmp[50010];
public:
    int reversePairs(vector<int>& nums)
    {
        return mergeSort(nums, 0, nums.size() - 1);
    }

    int mergeSort(vector<int>& nums, int left, int right)
    {
        if(left >= right) return 0;

        int ret = 0;
        // 1. 先根据中间元素划分区间
        int mid = (left + right) >> 1;
        // [left, mid] [mid + 1, right]

        // 2. 先计算左右两侧的翻转对
        ret += mergeSort(nums, left, mid);
        ret += mergeSort(nums, mid + 1, right);

        // 3. 先计算翻转对的数量
        int cur1 = left, cur2 = mid + 1, i = left;
        while(cur2 <= right) // 升序的情况
        {
            while(cur1 <= mid && nums[cur2] >= nums[cur1] / 2.0) cur1++;
            if(cur1 > mid)
                break;
            ret += mid - cur1 + 1;
            cur2++;
        }

        // 4. 合并两个有序数组
        cur1 = left, cur2 = mid + 1;
        while(cur1 <= mid && cur2 <= right)
            tmp[i++] = nums[cur1] <= nums[cur2] ? nums[cur1++] : nums[cur2++];
        while(cur1 <= mid) tmp[i++] = nums[cur1++];
        while(cur2 <= right) tmp[i++] = nums[cur2++];

        for(int j = left; j <= right; j++)
            nums[j] = tmp[j];

        return ret;
    }
};
```

C++ 运行结果：

Java 算法代码：

```java
class Solution
{
    int[] tmp;
    public int reversePairs(int[] nums)
    {
        int n = nums.length;
        tmp = new int[n];

        return mergeSort(nums, 0, n - 1);
    }

    public int mergeSort(int[] nums, int left, int right)
    {
        if(left >= right) return 0;

        int ret = 0;
        // 1. 根据中间元素，将区间分成两部分
        int mid = (left + right) / 2;
        // [left, mid] [mid + 1, right]

        // 2. 求出左右两个区间的翻转对
        ret += mergeSort(nums, left, mid);
        ret += mergeSort(nums, mid + 1, right);

        // 3. 处理一左一右 - 先计算翻转对
        int cur1 = left, cur2 = mid + 1, i = left;
        // 升序版本
        while(cur2 <= right)
        {
            while(cur1 <= mid && nums[cur2] >= nums[cur1] / 2.0) cur1++;
            if(cur1 > mid)
                break;
            ret += mid - cur1 + 1;
            cur2++;
        }

```

```
37          // 4. 合并两个有序数组 - 升序
38          cur1 = left; cur2 = mid + 1;
39          while(cur1 <= mid && cur2 <= right)
40              tmp[i++] = nums[cur1] <= nums[cur2] ? nums[cur1++] : nums[cur2++];
41          while(cur1 <= mid) tmp[i++] = nums[cur1++];
42          while(cur2 <= right) tmp[i++] = nums[cur2++];
43
44          for(int j = left; j <= right; j++)
45              nums[j] = tmp[j];
46
47          return ret;
48      }
49  }
```

Java 运行结果：