编程之美解题报告

姚大海 (acprimer.yao@gmail.com)

https://github.com/acprimer/Beauty-of-Programming

最后更新 2014-12-30

版权声明

内容简介

编程之美解题报告

数学符号

符号	意义
$lg(x)$	以 2 为底的对数 $log_2(x)$
$log_b(x)$	以 b 为底的对数 $log_b(x)$
$\lfloor x \rfloor$	下取整函数
$\lceil x \rceil$	上取整函数

GitHub 地址

本书是开源的，GitHub 地址：https://github.com/acprimer/Beauty-of-Programming

目录

第 1 章　游戏之乐　1

第 2 章　数学之魅　2

2.1　求二进制数中 1 的个数 2
2.2　不要被阶乘吓倒 4
2.3　寻找发帖“水王” 6
2.4　1 的数目 8
2.5　寻找最大的 K 个数 11
2.6　精确表达浮点数 11
2.7　最大公约数问题 13
2.8　找符合条件的整数 14
2.9　斐波纳契数列 14
2.10　寻找数组中的最大值和最小值　16
2.11　寻找最近点对 17
2.12　快速寻找满足条件的两个数 . 21
2.13　子数组的最大乘积 22

第 1 章
游戏之乐

暂无

第 2 章
数学之魅

2.1 求二进制数中 1 的个数

问题

对于一个字节（8bit）的无符号整型变量，求其二进制表示中“1”的个数，要求算法的执行效率尽可能高。

解法一

利用模运算求取 x 的每个二进制位上是否为 1。

```
// 时间复杂度 O(lg(x)), 空间复杂度 O(1)
int count_1(int x) {
    int ans = 0;
    while (x != 0) {
        ans += (x % 2);
        x /= 2;
    }
    return ans;
}
```

解法二

利用位运算求取 x 的每个二进制位上是否为 1。位运算比模运算效率要高。

```
// 时间复杂度 O(lg(x)), 空间复杂度 O(1)。
int count_2(int x) {
    int ans = 0;
    while (x != 0) {
        ans += (x & 0x01);
        x >>= 1;
    }
    return ans;
}
```

解法三

利用按位与 x&(x-1) 每次消除一个 1，直到 x 为 0。

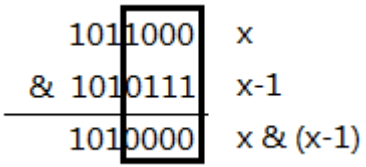

图 2-1 x & (x-1)

```
// 时间复杂度 O(M)，空间复杂度 O(1)，其中 M 为二进制中 1 的个数。
int count_3(int x) {
    int ans = 0;
    while (x != 0) {
        x &= (x - 1);
        ans++;
    }
    return ans;
}
```

扩展问题 1

如果变量是 32 位的 DWORD，你会使用上述的哪一种算法，或者改进哪一个算法？

解法

DWORD 是一个 32 位无符号整数，因此该问题与原问题唯一的区别就是数据的范围不同。上述的三种解法都适用，而且都不要做任何改进。另外对于原书中的解法四使用分支操作和解法五查表法是不适用的。时间复杂度和空间复杂度和原解法相同。

扩展问题 2

给定两个正整数（二进制形式表示）A 和 B，问把 A 变为 B 需要改变多少位（bit）？也就是说，整数 A 和 B 的二进制表示中有多少位是不同的？

解法

拿到这个问题，我们首先想到的就是按照从低到高依次检查 A 和 B 的每一位是否相同，然后记录个数，时间复杂度是 $O(log_2(Max(A,B)))$。接着，我们会想能不能把复杂度降到只和位不同的个数相关。自然的就会想到通过一种位运算把不同的 1 取出来，这种位

运算就是按位异或 ^。计算 C=A^B，再计算 C 中 1 的个数就行了，可以采用原题中的解法三。

```
// 时间复杂度 O(M)，空间复杂度 O(1)，其中 M 为 A 和 B 二进制中不同的个数。
int exercise_2(int A, int B) {
    return count(A ^ B);        // 调用上面的 count 函数，计算 A^B 中 1 的个数。
}
```

扩展问题 3

给定整数 N，判断它是否为 2 的幂，比如 $0, 2, 4, 8, \cdots$ 都是 2 的幂。这是原书中 2.2 的扩展问题，我感觉放在这里比较合适。

解法

```
boolean isPowerOf2(int N) {
    if (N >= 0 && ((N & (N - 1)) == 0)) return true;
    return false;
}
```

2.2 不要被阶乘吓倒

问题

1. 给定一个整数 N，那么 N 的阶乘 $N!$ 末尾有多少个 0 呢？例如，$N = 10$，$N! = 3628800$，$N!$ 的末尾有两个 0。

2. 求 $N!$ 的二进制表示中最低位 1 的位置。

问题 1 解法

对 $N!$ 进行质因数分解得到 $N! = \prod {p_i}^{k_i}$，由于 10=2×5，所以 $N!$ 末尾 0 的个数只和 5 的指数有关。计算 $N!$ 中有多少个 5，可以利用公式 $\lfloor \frac{N}{5} \rfloor + \lfloor \frac{N}{5^2} \rfloor + \lfloor \frac{N}{5^3} \rfloor + \cdots$.

```
// 时间复杂度 O(log5(N))，空间复杂度 O(1)
int count_1(int N) {
    int ans = 0;
        while (N != 0) {
        N /= 5;
        ans += N;
    }
    return ans;
}
```

问题 2 解法一

和问题 1 基本相同，只是现在二进制表示中末尾 0 的个数是和 $N!$ 中 2 的指数有关。利用公式 $\lfloor \frac{N}{2} \rfloor + \lfloor \frac{N}{2^2} \rfloor + \lfloor \frac{N}{2^3} \rfloor + \cdots$.

```
// 时间复杂度 O(lg(N)), 空间复杂度 O(1)
int count_base_2(int N) {
    int ans = 0;
    while (N != 0) {
        N >>= 1;
        ans += N;
    }
    return ans;
}
```

问题 2 解法二

$N!$ 中质因数 2 的个数等于 $N - N$ 的二进制表示中 1 的个数。时间复杂度 O(M)，M 是 N 二进制表示中 1 的个数。这个解法技巧性太强了，不看解答完全想不到，而且扩展性差。

代码略。

扩展问题

求整数 N 的 B 进制表示中末尾 0 的个数。题目来源 UVA 10061.

解法

对 $N!$ 进行质因数分解得到 $N! = \prod {p_i}^{k_i}$，对 B 进行质因数分解得到 $N! = \prod {p_i}^{t_i}$，我们用 $f(N, B)$ 来表示 N 的 B 进制表示中末尾 0 的个数，可以得出下面的等式: $f(N, B) = min\{\frac{k_i}{t_i}\}$.

```
// 时间复杂度 O(lgB(N)), 空间复杂度 O(1)
int countZeros(int n, int b) {
    int ans = Integer.MAX_VALUE;
    for (int i = 2; i <= b; i++) {
        if (b % i != 0) continue;
        int cnt = 0;
        while (b % i == 0) {
            b /= i;
            cnt++;
        }
        int tmp = 0, tn = n;
        while (tn != 0) {
            tn /= i;
            tmp += tn;
```

```
            }
            ans = Math.min(ans, tmp / cnt);
        }
        return ans;
    }
```

这里我们可以考虑一个问题，可不可以直接求出最大的 $p_i{}^{t_i}$，然后计算 $N!$ 中有多少个 p_i。提示：N=4，B=40.

2.3 寻找发帖“水王”

问题

给定一个整型数组，每个数组元素表示一个 ID，数组长度为 N，数组中有一个 ID 出现的次数超过了数组长度的一半，求出这个 ID。

解法 1

最简单的解法就是暴力枚举每一个数组元素，判断该元素是否超过一半。时间复杂度是 $O(N^2)$。

然后我们考虑对数组进行排序（很多问题一旦排序之后就会有思路了），这时候数组的中位数一定就是我们要求的 ID。时间复杂度是 $O(Nlg(N))$。

最后，我们考虑这样一个场景，让不同的 ID 进行 PK，最后剩下的一定就是超过一半的 ID。可以用一个栈来模拟数组中的元素 PK 的过程。

```
// 时间复杂度 O(N)，空间复杂度 O(N)
int findMostWithStack(int[] ids) {
    Stack<Integer> stack = new Stack<Integer>();
    for(int i=0;i<ids.length;i++) {
        if(stack.isEmpty()) {
            stack.push(ids[i]);
        } else {
            if(stack.peek()==ids[i]) stack.push(ids[i]);
            else stack.pop();
        }
    }
    return stack.peek();
}
```

解法 2

下面我们就来考虑一下是否可以降低空间复杂度。其实上面的做法中我们对于栈中的元素并不关心，我们只关心当前栈顶元素和栈的大小，所以我们就可以不用栈来处理了，用两个变量记录栈的大小和栈顶元素即可。

```
// 时间复杂度 O(N), 空间复杂度 O(1)
int findMost(int[] ids) {
    int candidate = -1, count = 0;
    for (int i = 0; i < ids.length; i++) {
        if (count == 0) {
            candidate = ids[i];
            count = 1;
        } else {
            if (ids[i] == candidate) count++;
            else count--;
        }
    }
    return candidate;
}
```

解法 3

利用哈希表。

扩展问题 1

原问题中保证一定存在一个元素超过一半，那么如果去掉这个条件呢？求出这个元素的下标（数组下标从 0 开始），如果不存在则返回 -1.

解法

如果不存在超过一半的元素，上面的两个解法其实并不能正确返回。比如 [1,2,3]，返回的就是 3. 其实这里要做的就是对这个得到的结果进行判断出现次数是否超过一半，时间和空间复杂度都不改变。

代码略。

扩展问题 2

有 3 个元素出现的次数超过了数组长度的 $\frac{1}{4}$，找出这 3 个元素。

解法

先排序再查找是可以找出的，时间复杂度是 $O(Nlg(N))$，这里就不给出具体代码了。我们同样也可以采用 PK 的方式来决出次数最多的 3 个数字，这样就需要保存 3 个候选元素和它们对应的次数，在遇到新的元素时，先跟这个 3 个候选元素比较，如果次数为 0 就把当前元素作为候选元素，如果当前元素与候选元素相等那么把它的次数加 1，如果当前

元素与 3 个候选元素都不相等就把 3 个候选元素的次数都减去 1，最后得到的 3 个候选元素就是答案了。

我们还可以将问题抽象成一个通用的情况：有 k-1 个元素出现了超过 $\frac{1}{k}$ 次，求出这 k-1 个元素。

```
// 时间复杂度 O(N), 空间复杂度 O(1)
int[] majorityNumber(ArrayList<Integer> nums, int k) {
    int[] candidates = new int[k];
    int[] count = new int[k];
    for (int i = 0; i < nums.size(); i++) {
        int cur = nums.get(i), j;
        for (j = 0; j < k; j++) {
            if (count[j] == 0) {
                candidates[j] = cur;
                count[j]++;
                break;
            } else if (candidates[j] == cur) {
                count[j]++;
                break;
            }
        }
        if (j >= k) {
            for(int idx=0;idx<k;idx++) {
                count[idx]--;
            }
        }
    }
    return candidates;
}
```

2.4 1 的数目

问题

从 1 到 N, 顺序写下这 N 个数字, 比如 1,2,3,4,..., 数一下这其中出现了多少次数字"1"。

1. 定义函数 $f(N)$ 返回 1 到 N 之间出现的 1 的个数，比如 $f(12) = 5$。

2. 满足条件 $f(N) = N$ 的最大的 N 是多少?

问题 1 解法

如果从 1 到 N 遍历每一个数字并且求出每个数字中包含了几个 1，这种做法的时间复杂度是 $O(N * log_{10}(N))$。

我们可以这样考虑，我们首先观察个位数字是循环出现的，循环节是 10，每 10 个数

字出现一个 1.

$$\underbrace{0,\underline{1},2\cdots,8,9}_{10}$$

再来观察一下十位数，这次每 100 个数字就会出现 10 个 1.

$$\underbrace{0,1,2\cdots,8,9,\underline{1}0,\underline{1}1,\underline{1}2,\underline{1}3,\underline{1}4,\underline{1}5,\underline{1}6,\underline{1}7,\underline{1}8,\underline{1}9,20,21,\cdots,98,99}_{100}$$

同理，百位上每 1000 个数字出现 100 个 1 $\cdots$

下面，我们举个例子说明一下怎么求 1 出现的次数。N=1231，先看个位，总共会循环 123+1 次，每个循环有 1 个 1，再看十位，总共循环 12+1 次，每个循环有 10 个 1，再看百位，总共循环 1+1 次，每个循环有 100 个 1，最后是千位，总共循环 0.232 次，每个循环有 1000 个 1.

关键问题就是如何求循环次数，我们考虑第 K 位（个位是第 0 位），digit 为当前位上的数字，higher 为高位的数字，lower 为低位的数字。digit 和 1 的关系有三种:

1. digit=0，循环次数为 higher，出现次数为 $higher * 10^K$。
2. digit=1，循环次数为 higher，出现次数为 $higher * 10^K + lower + 1$。
3. digit>1，循环次数为 higher+1，出现次数为 $(higher + 1) * 10^K$。

```
// 时间复杂度 O(log10(N)), 空间复杂度 O(1)
int digitCounts(int n) {
    int ans = 0;
    int power = 1;
    int higher = 0, lower = 0;
    while (n != 0) {
        int digit = n % 10;
        higher = n / 10;
        ans += (higher + (digit > 1 ? 1 : 0)) * power;
        if (digit == 1) {
            ans += lower + 1;
        }
        lower += power * digit;
        power *= 10;
        n /= 10;
    }
    return ans;
}
```

问题 2 解法

无

扩展问题 1

现在给你你一个数字 $K, K \in [0, 1, 2, \cdots, 8, 9]$。求 1 到 N 中出现 K 的次数。题目来源 Lint Code Digit Count。

解法

和原问题基本一致，只不过要特殊考虑一下 0，因为 0 不能作为数字开头。

扩展问题 2

对于其他进制表示方法，也可以试试，看看什么规律。例如二进制：

$f(1) = 1$

$f(10) = 10$ （01，10 两个 1）

$f(11) = 100$ （01，10，11 四个 1）

解法

和十进制的做法基本一致，为了方便计算我们采用十进制来表示数字，比如上面第三个式子我们用十进制表示为 $f(3) = 4$，也就是输入输出都是十进制数，但是计算 1 的个数是采用二进制数。这里给出的代码采用的很多位运算。

```
// 时间复杂度 O(lg(N)), 空间复杂度 O(1)
int digitCountsBinary(int n) {
    int ans = 0;
    int wei = 0;
    int higher = 0, lower = 0;
    while (n != 0) {
        int digit = n & 0x01;
        higher = n >> 1;
        if (digit == 0) {
            ans += (higher << wei);
        } else {
        ans += ((higher << wei) | lower) + 1;
        }
        lower += (digit << wei);
        wei++;
        n >>= 1;
    }
    return ans;
}
```

2.5 寻找最大的 K 个数

问题

给你个很大的数组，求出这个数组最大的前 K 个数。

解法

拿到问题看到前 K 个，首先就想到了二叉堆，我们维护一个小根堆的二叉堆，用来保存最大的 K 个数。然后不断的遍历数组，每遇到一个新的元素就让它进堆，如果堆的大小大于 K 了，就把最小的元素删除，继续保持小根堆的性质不变。这样做既可以处理很大的数组，而且可以处理动态到来的元素，是一个在线算法。维护一个二叉堆的时间复杂度是 $O(lg(K))$，总的复杂度是 $O(N * lg(K))$，N 为数组长度。我们在具体的实现的时候，可以直接用 PriorityQueue 类来维护二叉堆。

```
// 时间复杂度 O(N*lg(K))，空间复杂度 O(K)
int[] findMostKthNumber(int[] nums, int k) {
    PriorityQueue<Integer> queue = new PriorityQueue<Integer>(k + 1);
    for (int num : nums) {
        queue.offer(num);
        if (queue.size() > k) {
            queue.poll();
        }
    }
    int[] ans = new int[k];
    for (int i = 0; i < k; i++) {
        ans[i] = queue.poll();
    }
    return ans;
}
```

2.6 精确表达浮点数

问题

用分数表示小数。

$$0.9 = \frac{9}{10}$$

$$0.333(3) = \frac{1}{3}$$

$$0.1(2) = \frac{11}{90}$$

左边小数中括号部分为循环节。

解法

我们用 $0.\overbrace{a_1\cdots a_k}^{k}\overbrace{(b_1\cdots b_c)}^{c}$ 表示小数，其中非循环节部分长度为 k，数值表示为 $\overline{a_1\cdots a_k}$，循环节部分长度为 c，数值表示为 $\overline{b_1\cdots b_c}$。我们将最终要求的分数表示成 $\frac{A}{B}$

下面我们用公式推导出 A 和 B。

$$x = 0.\overbrace{a_1\cdots a_k}^{k}\overbrace{(b_1\cdots b_c)}^{c}$$

$$10^k * x = \overline{a_1\cdots a_k}.(\overline{b_1\cdots b_c})$$

$$10^{k+c} * x = \overline{a_1\cdots a_k b_1\cdots b_c}.(\overline{b_1\cdots b_c})$$

$$(10^{k+c} - 10^k) * x = \overline{a_1\cdots a_k b_1\cdots b_c} - \overline{a_1\cdots a_k}$$

$$x = \frac{A}{B} = \frac{\overline{a_1\cdots a_k b_1\cdots b_c} - \overline{a_1\cdots a_k}}{10^k * (10^c - 1)} = \frac{\overline{a_1\cdots a_k} * (10^c - 1) + \overline{b_1\cdots b_c}}{10^k * (10^c - 1)}$$

$$A = \overline{a_1\cdots a_k} * (10^c - 1) + \overline{b_1\cdots b_c}$$

$$B = 10^k * (10^c - 1)$$

```
// 时间复杂度 O(N), 空间复杂度 O(1)
int[] floatNubmer(String number) {
    int[] fraction = new int[2];
    int pre = 0, cycle = 0;
    int powerPre =1, powerCycle=1;
    boolean flag = false;
    for(int i=2;i<number.length() && number.charAt(i)!=')';i++) {
        if(number.charAt(i)=='(') {
            flag = true;
            continue;
        }
        if(flag) {
            cycle = cycle * 10 + number.charAt(i)-'0';
            powerCycle*=10;
         } else {
            pre = pre * 10 + number.charAt(i) - '0';
            powerPre*=10;
        }
    }
    if(flag) {
        fraction[0] = pre * (powerCycle - 1) + cycle;
        fraction[1] = powerPre * (powerCycle - 1);
    } else {
        fraction[0] = pre;
        fraction[1] = powerPre;
    }
    int d = gcd(fraction[0], fraction[1]);
    fraction[0] /= d;
```

```
    fraction[1] /= d;
    return fraction;
}

int gcd(int a, int b) {
    if(a==0) return b;
    return gcd(b%a, a);
}
```

2.7 最大公约数问题

问题

求两个正整数的最大公约数（GCD），如果两个正整数很大，有简单的算法吗？

解法 1

根据欧几里德的辗转相除法可知 $gcd(x,y) = gcd(y, x\%y)$。因此可以采用简单的递归来求解最大公约数。

```
// 时间复杂度 O(lg(Max(x,y))), 空间复杂度 O(1)
int gcd(int x, int y) {
        if(x==0) return y;
        return gcd(y, x \% y);
}
// 非递归形式
int gcd(int x, int y) {
        while(y != 0) {
                int r = x \% y;
                m = n;
                n = r;
        }
        return x;
}
```

解法 2

当正整数很大的时候，取模运算的开销很大，因此要避免使用取模运算。下面给出一种利用奇偶性来计算最大公约数的方法。

1. 当 x, y 均为偶数时，$gcd(x,y) = 2 * gcd(\frac{x}{2}, \frac{y}{2})$。
2. 当 x, y 均为奇数时，$gcd(x,y) = 2 * gcd(y, x-y)$。
3. 当 x 为偶数，y 为奇数时，$gcd(x,y) = gcd(\frac{y}{2}, y)$。

4. 当 x 为奇数，y 为偶数时，$gcd(x,y) = gcd(x, \frac{y}{2})$。

```
// 时间复杂度 O(lg(Max(x,y))), 空间复杂度 O(1)
BigInteger gcd(BigInteger x, BigInteger y) {
    if (x.compareTo(y) < 0) {
        return gcd(y, x);
    }
    if (y.equals(BigInteger.ZERO)) {
        return x;
    }
    if (isEven(x)) {
        if (isEven(y)) { // x is even, y is even
                return gcd(x.divide(BIG_TWO), y.divide(BIG_TWO)).multiply(BIG_TWO);
        } else { // x is even, y is odd
            return gcd(x.divide(BIG_TWO), y);
        }
    } else {
        if (isEven(y)) { // x is odd, y is even
            return gcd(x, y.divide(BIG_TWO));
        } else { // x is odd, y is odd
            return gcd(y, x.subtract(y));
        }
    }
}

public boolean isEven(BigInteger x) {
        return x.mod(BIG_TWO).equals(BigInteger.ZERO);
}
```

2.8 找符合条件的整数

问题

任意给定一个正整数 N，求一个最小的正整数 M（M>1），使得 N×M 的十进制表示形式里只含有 0，1.

2.9 斐波纳契数列

问题

斐波纳契数列的递推公式为：

$$F(n) = \begin{cases} 0 & \text{if} \quad n = 0; \\ 1 & \text{if} \quad n = 1; \\ F(n-1) + F(n-2) & \text{if} \quad n > 1. \end{cases}$$

求 $F(n)$。

解法 1

记录每一个计算过的 $F(n)$。

```
// 时间复杂度 O(N), 空间复杂度 O(N)
int fib(int n) {
    int[] f = new int[n + 1];
    f[0] = 0;
    f[1] = 1;
    for (int i = 2; i <= n; i++) {
            f[i] = f[i - 1] + f[i - 2];
    }
    return f[n];
}
```

解法 2

利用特征公式 $x^2 - x - 1 = 0$ 求出通向公式为

$$F(n) = \frac{1}{\sqrt{5}} \times \left[(\frac{1+\sqrt{5}}{2})^n - (\frac{1-\sqrt{5}}{2})^n \right]$$

不过因为引入了无理数，所以不能保证结果的精度。

```
// 时间复杂度 O(1), 空间复杂度 O(1)
int fib(int n) {
    double root = Math.sqrt(5.0);
    double ans = (Math.pow((1.0 + root) / 2.0, n) - Math.pow((1.0 - root) / 2.0, n)) / root;
    return (int) ans;
}
```

解法 3

根据公式

$$\begin{bmatrix} F(n) & F(n-1) \\ F(n-1) & F(n-2) \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}^n$$

```
// 时间复杂度 O(lg(n)), 空间复杂度 O(1)
int fib(int n) {
    int[] power = new int[]{1, 1, 1, 0};
    int[] fn = new int[]{1, 0, 1, 0};
    while (n != 0) {
            if (n % 2 != 0) {
                fn = multify(fn, power);
        }
        power = multify(power, power);
        n /= 2;
    }
    return fn[1];
```

```
}
int[] multify(int[] x, int[] y) {
    int[] ans = new int[4];
    ans[0] = x[0] * y[0] + x[1] * y[2];
    ans[1] = x[0] * y[1] + x[1] * y[3];
    ans[2] = x[2] * y[0] + x[3] * y[2];
    ans[3] = x[2] * y[1] + x[3] * y[3];
    return ans;
```

2.10 寻找数组中的最大值和最小值

问题

在数组中找出最大值和最小值，最少需要比较多少次。

解法 1

扫描一遍数组，将每个数都和最大值最小值比较，需要 $2 * N$ 次比较。代码略。

解法 2

我们每次处理两个数，先比较这两个数，找出比较大的数和最大值比较，找出比较小的数和最小值比较，处理 2 个数要比较 3 次，总共需要比较 $\frac{3}{2}N$ 次。

```
// 时间复杂度 O(n)，空间复杂度 O(1)
int[] findMaxMin(int[] nums) {
    int[] maxmin = new int[]{Integer.MIN_VALUE, Integer.MAX_VALUE};
    if (nums.length % 2 != 0) {
        maxmin[0] = maxmin[1] = nums[0];
    }
    for (int i = nums.length % 2; i < nums.length; i += 2) {
    int twoMax = nums[i];
    int twoMin = nums[i + 1];
    if (twoMax < twoMin) {
        twoMax = nums[i + 1];
        twoMin = nums[i];
    }
    maxmin[0] = Math.max(maxmin[0], twoMax);
    maxmin[1] = Math.min(maxmin[1], twoMin);
    }
    return maxmin;
}
```

扩展问题

如果需要找出数组中的次大元素，需要比较多少次？

解法

和找出最大最小值一样，同时记录最大值和次大值，每次处理两个元素，如果较大的数比最大值大，就更新最大值和次大值，再把较小的数和次大值比较。否则，就把较大的值和次大值比较。比较次数为 $\frac{3}{2}N$

```
// 时间复杂度 O(n)，空间复杂度 O(1)
int findSecondMax(int[] nums) {
    int[] max = new int[]{Integer.MIN_VALUE, Integer.MIN_VALUE};
    if (nums.length % 2 != 0) {
        max[0] = max[1] = nums[0];
    }
    for (int i = nums.length % 2; i < nums.length; i += 2) {
        int mmax = nums[i];
        int smax = nums[i + 1];
        if (mmax < smax) {
            mmax = nums[i + 1];
            smax = nums[i];
        }
        if (mmax > max[0]) {
            max[1] = max[0];
            max[0] = mmax;
            if (smax > max[1]) {
                max[1] = smax;
            }
        } else {
            if (mmax > max[1]) {
                max[1] = mmax;
            }
        }
    }
    return max[1];
}
```

2.11 寻找最近点对

问题

给定平面上 N 个点的坐标，找出距离最近的两个点。

解法

最直接的方法就是求出每两个点之间的距离，然后找出最小值，这种做法的时间复杂度是 $O(n^2)$。我们可以采用分治的方法将时间复杂度降到 $O(nlg(n))$。

根据 X 坐标将这些点从小到大排序，然后分成左右两个部分 S_1 和 S_2，分别求出它们的最小距离 δ_1 和 δ_2，那么最小距离有三种情况：

1. 最近点对都来自 S_1，那么最小距离为 δ_1。

2. 最近点对都来自 S_2，那么最小距离为 δ_2。

3. 最近点对分别来自 S_1 和 S_2，这种情况需要特殊讨论。。

下面我们来讨论第三种情况，

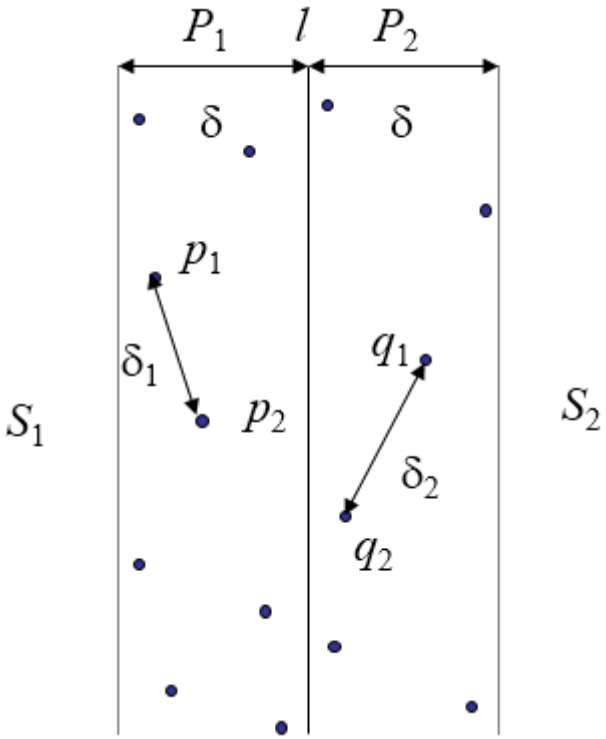

图 2-2 最近点对的三种情况

如果其中一个点 p 来自 S_1，另一个点来自 S_2，那么最多只有 6 个点和 p 的距离小于 $\delta = min(\delta_1, \delta_2)$。如下图所示：

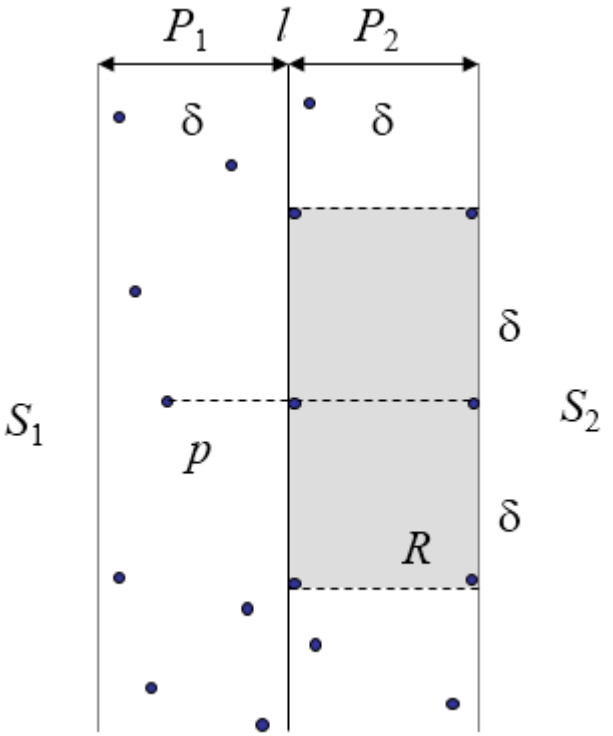

图 2-3 最小距离的区域

因此我们可以通过对 δ 范围内的点按照 Y 坐标排序，这样就能在 $O(n)$ 的时间内求出最小点对。根据时间复杂度 $f(n) = 2 * f(\frac{n}{2}) + O(n)$ 可以求出 $f(n) = nlg(n)$。

```
// 时间复杂度 O(nlgn)，空间复杂度 O(n)
// Point2D 的定义
class Point2D {
    int x, y;
    public Point2D(int x, int y) {
        this.x = x;
        this.y = y;
    }
}
double minDist(Point2D[] points) {
        // sort by x
    Arrays.sort(points, new Comparator<Point2D>() {
        @Override
        public int compare(Point2D p, Point2D q) {
            return p.x > q.x ? 1 : -1;
        }
    });
    Point2D[] pointsByY = new Point2D[points.length];
```

```
    for (int i = 0; i < points.length; i++) {
        pointsByY[i] = points[i];
    }

    Point2D[] aux = new Point2D[points.length];

    return closest(points, pointsByY, aux, 0, points.length - 1);
}

private double closest(Point2D[] pointsByX, Point2D[] pointsByY, Point2D[] aux, int start, int
    if (end <= start) {
        return Double.POSITIVE_INFINITY;
    }

    int mid = start + (end - start) / 2;
    int median = pointsByX[mid].x;
    double dLeft = closest(pointsByX, pointsByY, aux, start, mid);
    double dRight = closest(pointsByX, pointsByY, aux, mid + 1, end);
    double delta = Math.min(dLeft, dRight);

    // O(n)
    merge(pointsByY, aux, start, mid, end);

    int count = 0;
    for (int i = start; i <= end; i++) {
        if (Math.abs(pointsByY[i].x - median) < delta) {
            aux[count++] = pointsByY[i];
        }
    }
    // O(n)
    for (int i = 0; i < count; i++) {
        for (int j = i + 1; j < count && aux[j].y - aux[i].y < delta; j++) {
            double distance = dist(aux[i], aux[j]);
            delta = Math.min(delta, distance);
        }
    }
    return delta;
}

private double dist(Point2D p, Point2D q) {
    double dx = p.x - q.x;
    double dy = p.y - q.y;
    return Math.sqrt(dx * dx + dy * dy);
}

private void merge(Point2D[] pointsByY, Point2D[] aux, int start, int mid, int end) {
    for (int i = start; i <= end; i++) {
            aux[i] = pointsByY[i];
    }
    int i = start, j = mid + 1, k = start;
    while (i <= mid || j <= end) {
        if (j > end || (i <= mid && aux[i].y < aux[j].y)) {
            pointsByY[k++] = aux[i++];
```

```
            } else {
                pointsByY[k++] = aux[j++];
            }
        }
    }
```

2.12 快速寻找满足条件的两个数

问题

快速找出一个数组中的两个数字，使得它们的和等于一个给定的值。

解法 1

最简单的做法就是穷举：从数组中任意取出两个数字计算它们的和是否为给定的值，时间复杂度为 $O(n^2)$。

更快的查找方法是利用哈希表直接查找，将每个数字映射到哈希表中，然后对每个数字在哈希表中查找是否存在另一个数字。时间复杂度是 $O(n)$。

```
// 时间复杂度 O(n)，空间复杂度 O(n)
int[] twoSum(int[] numbers, int target) {
    HashMap<Integer, Integer> map = new HashMap<Integer, Integer>();
    for (int i = 0; i < numbers.length; i++) {
        Integer other = map.get(target - numbers[i]);
        if (other != null) {
            return new int[]{other, i};
        }
        map.put(numbers[i], i);
    }
    return new int[]{-1, -1};
}
```

解法 2

如果对数组按照从小到大进行排序，那就可以利用二分查找来找出另一个数字是否在数组中，时间复杂度是 $O(nlgn)$。另外，我们还可以采用两个指针的想法，让两个指针 p,q 分别指向数组首部和尾部。如果这两个数的和大于目标值就移动 q–，如果大于目标值就移动 p++。

```
// 这里假设数组是升序排列的
// 时间复杂度 O(n)，空间复杂度 O(1)
int[] twoSum_2(int[] numbers, int target) {
    int i = 0, j = numbers.length - 1;
    while (i < j) {
        if (numbers[i] + numbers[j] > target) {
```

```
            j--;
        } else if (numbers[i] + numbers[j] < target) {
            i++;
        } else {
            return new int[]{i, j};
        }
    }
    return new int[]{-1, -1};
}
```

2.13 子数组的最大乘积

问题

给定一个长度为 N 的整数数组，只允许使用乘法，不能用除法，计算任意 N-1 个数的组合中乘积最大的一组。

解法

使用两个乘积数组，$left[i] = \prod_{k=0}^{i-1} numbers[k]$，$right[i] = \prod_{k=n-1}^{i+1} numbers[k]$，则 $ans = max_{i=0}^{n-1}(left[i] * right[i])$。算法的时间复杂度是 O(n)。

```
// 时间复杂度 O(n)，空间复杂度 O(n)
int maxProduct(int[] numbers) {
    int len = numbers.length;
    if (len <= 1) {
        return 0;
    }
    int ans = 0;
    int[] leftProduct = new int[len];
    int[] rightProduct = new int[len];
    leftProduct[0] = 1;
    rightProduct[len - 1] = 1;
    for (int i = 1; i < numbers.length; i++) {
        leftProduct[i] = leftProduct[i - 1] * numbers[i - 1];
    }
    for (int i = len - 2; i >= 0; i--) {
        rightProduct[i] = rightProduct[i + 1] * numbers[i + 1];
    }
    for (int i = 0; i < numbers.length; i++) {
        ans = Math.max(ans, leftProduct[i] * rightProduct[i]);
    }
    return ans;
}
```

2.14 子数组之和的最大值（二维）

问题

对于 2.14 的一维数组可以扩展到二维的数组，如何求出一个矩形区域使得和最大呢？

解法

直接将二维投影成一维的情况，然后利用最大子数组的解法求出最大矩形和。算法的时间复杂度是 $O(n^2 * m)$。

```
// 时间复杂度 O(n^2*m), 空间复杂度 O(1)
int maxSubmatrix(int[][] A) {
    int rows = A.length;
    int columns = A[0].length;
    int ans = Integer.MIN_VALUE;
    for (int i = 1; i < rows; i++) {
        for (int j = 0; j < columns; j++) {
            A[i][j] += A[i - 1][j];
        }
    }
    for (int i = 0; i < rows; i++) {
        for (int j = i; j < rows; j++) {
            int sum = 0;
            for (int k = 0; k < columns; k++) {
                int x = A[j][k] - (i == 0 ? 0 : A[i - 1][k]);
                sum = Math.max(x, x + sum);
                ans = Math.max(ans, sum);
            }
        }
    }
    return ans;
}
```