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 2015-1-11

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
$\|S\|$	集合中元素的个数，或者字符串的长度

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

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。

$O(lg(x)) + O(1)$

```
1  int count_1(int x) {
2      int ans = 0;
3      while (x != 0) {
4          ans += (x % 2);
5          x /= 2;
6      }
7      return ans;
8  }
```

Chap02_01_NumberOfOnes.java

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。

$O(lg(x)) + O(1)$

```
1  int count_2(int x) {
2      int ans = 0;
3      while (x != 0) {
4          ans += (x & 0x01);
5          x >>= 1;
6      }
7      return ans;
8  }
```

Chap02_01_NumberOfOnes.java

解法三

利用按位与 x&(x-1) 每次消除一个 1，直到 x 为 0。时间复杂度 O(M)，，其中 M 为二进制中 1 的个数。

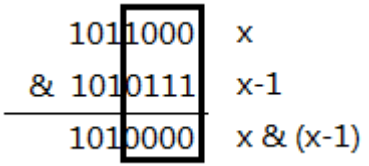

图 2-1 x & (x-1)

$O(M) + O(1)$

```
1  int count_3(int x) {
2      int ans = 0;
3      while (x != 0) {
4          x &= (x - 1);
5          ans++;
6      }
7      return ans;
8  }
```

Chap02_01_NumberOfOnes.java

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。

$O(M)+O(1)$

```
1  int exercise_2(int A, int B) {
2      return count(A ^ B);    // 调用上面的 count 函数，计算 A^B 中 1 的个数。
3  }
```

Chap02_01_NumberOfOnes.java

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

$O(1)+O(1)$

```
1  boolean isPowerOf2(int N) {
2      if (N >= 0 && ((N & (N - 1)) == 0)) return true;
3      return false;
4  }
```

Chap02_01_NumberOfOnes.java

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 $\lfloor \frac{N}{5} \rfloor + \lfloor \frac{N}{5^2} \rfloor + \lfloor \frac{N}{5^3} \rfloor + \cdots$.

$O(log5(N))+O(1)$

```
1  int count_1(int N) {
2      int ans = 0;
3      while (N != 0) {
4          N /= 5;
5          ans += N;
```

```
6       }
7       return ans;
8   }
```
Chap02_02_Factorial.java

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 $\lfloor \frac{N}{2} \rfloor + \lfloor \frac{N}{2^2} \rfloor + \lfloor \frac{N}{2^3} \rfloor + \cdots$.

$O(lg(N)) + O(1)$

```
1   int count_base_2(int N) {
2       int ans = 0;
3       while (N != 0) {
4           N >>= 1;
5           ans += N;
6       }
7       return ans;
8   }
```
Chap02_02_Factorial.java

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: $f(N, B) = min\{\frac{k_i}{t_i}\}$.

$O(lgB(N)) + O(1)$

```
1   int countZeros(int n, int b) {
2       int ans = Integer.MAX_VALUE;
3       for (int i = 2; i <= b; i++) {
4           if (b % i != 0) continue;
5           int cnt = 0;
```

```
6           while (b % i == 0) {
7               b /= i;
8               cnt++;
9           }
10          int tmp = 0, tn = n;
11          while (tn != 0) {
12              tn /= i;
13              tmp += tn;
14          }
15          ans = Math.min(ans, tmp / cnt);
16      }
17      return ans;
18  }
```

Chap02_02_Factorial.java

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。

$O(N)+O(N)$

```
1   int findMostWithStack(int[] ids) {
2       Stack<Integer> stack = new Stack<Integer>();
3       for(int i=0;i<ids.length;i++) {
4           if(stack.isEmpty()) {
5               stack.push(ids[i]);
6           } else {
7               if(stack.peek()==ids[i]) stack.push(ids[i]);
8               else stack.pop();
9           }
```

```
10      }
11      return stack.peek();
12  }
```

Chap02_03_WaterKing.java

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。

$O(N)+O(1)$

```
1   int findMost(int[] ids) {
2       int candidate = -1, count = 0;
3       for (int i = 0; i < ids.length; i++) {
4           if (count == 0) {
5               candidate = ids[i];
6               count = 1;
7           } else {
8               if (ids[i] == candidate) count++;
9               else count--;
10          }
11      }
12      return candidate;
13  }
```

Chap02_03_WaterKing.java

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。

$O(N)+O(1)$

```
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;
}
```

Chap02_03_WaterKing.java

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.

$$\underbrace{0, 1, 2 \cdots, 8, 9, \underline{10}, \underline{11}, \underline{12}, \underline{13}, \underline{14}, \underline{15}, \underline{16}, \underline{17}, \underline{18}, \underline{19}, 20, 21, \cdots, 98, 99}_{100}$$

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 $(higher + 1) * 10^K$。

$O(log10(N)) + O(1)$

```
1  int digitCounts(int n) {
2      int ans = 0;
3      int power = 1;
4      int higher = 0, lower = 0;
5      while (n != 0) {
6          int digit = n % 10;
7          higher = n / 10;
```

```
8           ans += (higher + (digit > 1 ? 1 : 0)) * power;
9           if (digit == 1) {
10              ans += lower + 1;
11          }
12          lower += power * digit;
13          power *= 10;
14          n /= 10;
15      }
16      return ans;
17  }
```

Chap02_04_Ones.java

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。

$O(lg(N)) + O(1)$

```
1   int digitCountsBinary(int n) {
2       int ans = 0;
3       int wei = 0;
```

```
4       int higher = 0, lower = 0;
5       while (n != 0) {
6           int digit = n & 0x01;
7           higher = n >> 1;
8           if (digit == 0) {
9               ans += (higher << wei);
10          } else {
11          ans += ((higher << wei) | lower) + 1;
12          }
13          lower += (digit << wei);
14          wei++;
15          n >>= 1;
16      }
17      return ans;
18  }
```

Chap02_04_Ones.java

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。

$O(N * lg(K)) + O(K)$

```
1   int[] findMostKthNumber(int[] nums, int k) {
2       PriorityQueue<Integer> queue = new PriorityQueue<Integer>(k + 1);
3       for (int num : nums) {
4           queue.offer(num);
5           if (queue.size() > k) {
6               queue.poll();
7           }
8       }
9       int[] ans = new int[k];
10      for (int i = 0; i < k; i++) {
11          ans[i] = queue.poll();
```

```
12      }
13      return ans;
14  }
```

Chap02_05_FindKthNumber.java

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。

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

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

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

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

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

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

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

$O(N)+O(1)$

```
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);
}
```

Chap02_06_FloatNumber.java

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。

$O(lg(Max(x,y))) + O(1)$

```
1  int gcd(int x, int y) {
2      if(x==0) return y;
3      return gcd(y, x % y);
4  }
5  // 非递归形式
6  int gcd(int x, int y) {
7      while(y != 0) {
8          int r = x % y;
9          m = n;
10         n = r;
11     }
12     return x;
13 }
```

Chap02_07_GCD.java

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，$gcd(x,y) = gcd(x, \frac{y}{2})$。

$O(lg(Max(x,y))) + O(1)$

```
1  BigInteger gcd(BigInteger x, BigInteger y) {
2      if (x.compareTo(y) < 0) {
3          return gcd(y, x);
4      }
5      if (y.equals(BigInteger.ZERO)) {
6          return x;
7      }
8      if (isEven(x)) {
9          if (isEven(y)) { // x is even, y is even
10             return gcd(x.divide(BIG_TWO), y.divide(BIG_TWO)).multiply(BIG_TWO);
11         } else { // x is even, y is odd
12             return gcd(x.divide(BIG_TWO), y);
```

```
13          }
14      } else {
15          if (isEven(y)) { // x is odd, y is even
16              return gcd(x, y.divide(BIG_TWO));
17          } else { // x is odd, y is odd
18              return gcd(y, x.subtract(y));
19          }
20      }
21  }
22
23  public boolean isEven(BigInteger x) {
24      return x.mod(BIG_TWO).equals(BigInteger.ZERO);
25  }
```

Chap02_07_GCD.java

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 $F(n)$。

$O(N) + O(N)$

```
1   int fib(int n) {
2       int[] f = new int[n + 1];
3       f[0] = 0;
4       f[1] = 1;
```

```
5       for (int i = 2; i <= n; i++) {
6           f[i] = f[i - 1] + f[i - 2];
7       }
8       return f[n];
9   }
```

Chap02_09_Fibonacci.java

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。

$O(1) + O(N)$

```
1   int fib(int n) {
2       double root = Math.sqrt(5.0);
3       double ans = (Math.pow((1.0 + root) / 2.0, n)
4                   - Math.pow((1.0 - root) / 2.0, n))
5                    / root;
6       return (int) ans;
7   }
```

Chap02_09_Fibonacci.java

解法 3

根据公式

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

$O(lg(n)) + O(N)$

```
1   int fib(int n) {
2       int[] power = new int[]{1, 1, 1, 0};
3       int[] fn = new int[]{1, 0, 1, 0};
4       while (n != 0) {
5           if (n % 2 != 0) {
6               fn = multify(fn, power);
7           }
8           power = multify(power, power);
9           n /= 2;
10      }
11      return fn[1];
12  }
13  int[] multify(int[] x, int[] y) {
14      int[] ans = new int[4];
```

```
15      ans[0] = x[0] * y[0] + x[1] * y[2];
16      ans[1] = x[0] * y[1] + x[1] * y[3];
17      ans[2] = x[2] * y[0] + x[3] * y[2];
18      ans[3] = x[2] * y[1] + x[3] * y[3];
19      return ans;
```

Chap02_09_Fibonacci.java

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。

$O(n) + O(1)$

```
1   int[] findMaxMin(int[] nums) {
2       int[] maxmin = new int[]{Integer.MIN_VALUE, Integer.MAX_VALUE};
3       if (nums.length % 2 != 0) {
4           maxmin[0] = maxmin[1] = nums[0];
5       }
6       for (int i = nums.length % 2; i < nums.length; i += 2) {
7       int twoMax = nums[i];
8       int twoMin = nums[i + 1];
9       if (twoMax < twoMin) {
10          twoMax = nums[i + 1];
11          twoMin = nums[i];
12      }
13      maxmin[0] = Math.max(maxmin[0], twoMax);
14      maxmin[1] = Math.min(maxmin[1], twoMin);
15      }
16      return maxmin;
17  }
```

Chap02_10_FindMaxMin.java

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 $\frac{3}{2}N$。

$O(n) + O(1)$

```
1  int findSecondMax(int[] nums) {
2      int[] max = new int[]{Integer.MIN_VALUE, Integer.MIN_VALUE};
3      if (nums.length % 2 != 0) {
4          max[0] = max[1] = nums[0];
5      }
6      for (int i = nums.length % 2; i < nums.length; i += 2) {
7          int mmax = nums[i];
8          int smax = nums[i + 1];
9          if (mmax < smax) {
10             mmax = nums[i + 1];
11             smax = nums[i];
12         }
13         if (mmax > max[0]) {
14             max[1] = max[0];
15             max[0] = mmax;
16             if (smax > max[1]) {
17                 max[1] = smax;
18             }
19         } else {
20             if (mmax > max[1]) {
21                 max[1] = mmax;
22             }
23         }
24     }
25     return max[1];
26 }
```

Chap02_10_FindMaxMin.java

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，

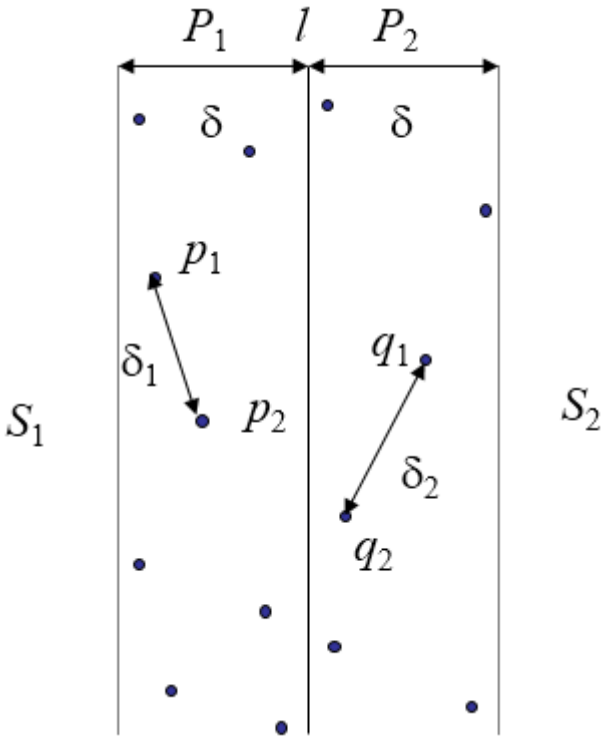

图 2-2 最近点对的三种情况

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：

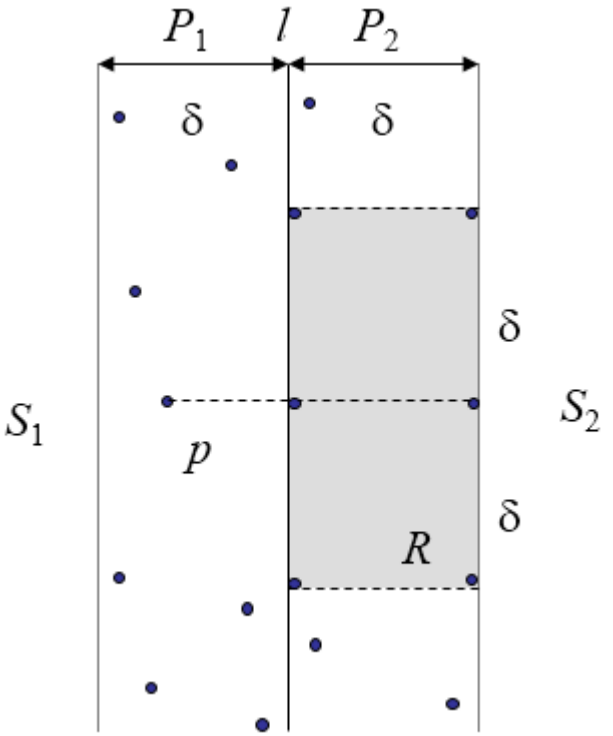

图 2-3 最小距离的区域

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 $f(n) = nlg(n)$。

$O(nlgn) + O(n)$

```
// 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;
        }
    });
```

```
17      Point2D[] pointsByY = new Point2D[points.length];
18      for (int i = 0; i < points.length; i++) {
19          pointsByY[i] = points[i];
20      }
21
22      Point2D[] aux = new Point2D[points.length];
23
24      return closest(points, pointsByY, aux, 0, points.length - 1);
25  }
26
27  private double closest(Point2D[] pointsByX, Point2D[] pointsByY, Point2D[] aux, int start, int
28      if (end <= start) {
29          return Double.POSITIVE_INFINITY;
30      }
31
32      int mid = start + (end - start) / 2;
33      int median = pointsByX[mid].x;
34      double dLeft = closest(pointsByX, pointsByY, aux, start, mid);
35      double dRight = closest(pointsByX, pointsByY, aux, mid + 1, end);
36      double delta = Math.min(dLeft, dRight);
37
38      // O(n)
39      merge(pointsByY, aux, start, mid, end);
40
41      int count = 0;
42      for (int i = start; i <= end; i++) {
43          if (Math.abs(pointsByY[i].x - median) < delta) {
44              aux[count++] = pointsByY[i];
45          }
46      }
47      // O(n)
48      for (int i = 0; i < count; i++) {
49          for (int j = i + 1; j < count && aux[j].y - aux[i].y < delta; j++) {
50              double distance = dist(aux[i], aux[j]);
51              delta = Math.min(delta, distance);
52          }
53      }
54      return delta;
55  }
56
57  private double dist(Point2D p, Point2D q) {
58      double dx = p.x - q.x;
59      double dy = p.y - q.y;
60      return Math.sqrt(dx * dx + dy * dy);
61  }
62
63  private void merge(Point2D[] pointsByY, Point2D[] aux, int start, int mid, int end) {
64      for (int i = start; i <= end; i++) {
```

```
65          aux[i] = pointsByY[i];
66      }
67      int i = start, j = mid + 1, k = start;
68      while (i <= mid || j <= end) {
69          if (j > end || (i <= mid && aux[i].y < aux[j].y)) {
70              pointsByY[k++] = aux[i++];
71          } else {
72              pointsByY[k++] = aux[j++];
73          }
74      }
75  }
```

Chap02_11_ClosestPairOfPoints.java

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 $O(n)$。

$O(n) + O(n)$

```
1   int[] twoSum(int[] numbers, int target) {
2       HashMap<Integer, Integer> map = new HashMap<Integer, Integer>();
3       for (int i = 0; i < numbers.length; i++) {
4           Integer other = map.get(target - numbers[i]);
5           if (other != null) {
6               return new int[]{other, i};
7           }
8           map.put(numbers[i], i);
9       }
10      return new int[]{-1, -1};
11  }
```

Chap02_12_TwoSum.java

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++。

$O(n) + O(1)$

```
// 这里假设数组是升序排列的
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};
}
```

Chap02_12_TwoSum.java

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)。

$O(n) + O(n)$

```
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];
    }
```

```
14      for (int i = len - 2; i >= 0; i--) {
15          rightProduct[i] = rightProduct[i + 1] * numbers[i + 1];
16      }
17      for (int i = 0; i < numbers.length; i++) {
18          ans = Math.max(ans, leftProduct[i] * rightProduct[i]);
19      }
20      return ans;
21  }
```

Chap02_13_MaxProduct.java

2.14 求数组的子数组之和的最大值

问题

给定一个数组，求出子数组之和的最大值

解法

$O(n) + O(1)$

```
1   int maxSubArray(int[] A) {
2       int ans = A[0];
3       int sum = A[0];
4       for (int i = 1; i < A.length; i++) {
5           sum = Math.max(A[i], A[i] + sum);
6           ans = Math.max(ans, sum);
7       }
8       return ans;
9   }
```

Chap02_14_MaxSubarray.java

2.15 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 $O(n^2 * m)$。

$O(n^2 * m) + O(1)$

```
1   int maxSubmatrix(int[][] A) {
2       int rows = A.length;
```

```
3       int columns = A[0].length;
4       int ans = Integer.MIN_VALUE;
5       for (int i = 1; i < rows; i++) {
6           for (int j = 0; j < columns; j++) {
7               A[i][j] += A[i - 1][j];
8           }
9       }
10      // 枚举起始行和结束行
11      for (int i = 0; i < rows; i++) {
12          for (int j = i; j < rows; j++) {
13              int sum = 0;
14              for (int k = 0; k < columns; k++) {
15                  int x = A[j][k] - (i == 0 ? 0 : A[i - 1][k]);
16                  sum = Math.max(x, x + sum);
17                  ans = Math.max(ans, sum);
18              }
19          }
20      }
21      return ans;
22  }
```

Chap02_15_MaxSubmatrix.java

2.16 求数组中最长递增子序列

问题

求一个一维数组中最长递增子序列的长度。例如，在序列 [1,7,3,6,4,5,2] 中，最长递增子序列为 [1,3,4,5]，长度为 4。

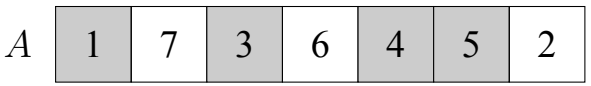

图 2-4 最长递增子序列

解法 1

用 $f[i]$ 表示序列 $[A_0, A_1, \ldots, A_i]$ 中以 A_i 为结尾的最长递增子序列的长度。利用动态规划，求出状态转移方程为：

$$f[i] = \begin{cases} 1 & n = 0; \\ max_{A_j < A_i}(f[j] + 1) & n > 0. \end{cases}$$

因此只要遍历每一个元素，计算出 $f[i]$ 并记录 $f[i]$ 的最大值即为最长递增子序列的长度。

A	1	7	3	6	4	5	2
f	1	2	2	3	3	4	2

图 2-5 最长递增子序列求解

$O(n^2) + O(n)$

```
int LIS(int[] numbers) {
    int ans = 1;
    int n = numbers.length;
    int[] len = new int[n];
    for (int i = 0; i < n; i++) {
        len[i] = 1;
        for (int j = 0; j < i; j++) {
            if (numbers[i] > numbers[j] && len[j] + 1 > len[i]) {
                len[i] = len[j] + 1;
            }
        }
        ans = Math.max(ans, len[i]);
    }
    return ans;
}
```

Chap02_16_LongestIncreasingSubsequence.java

2.17 数组循环移位

问题

把一个含有 N 个元素的数组循环右移 K 位，要求时间复杂度为 $O(N)$，且只允许使用两个附加变量。

解法

比如数组序列为 $abcd1234$，要求循环右移 4 位变成 $1234abcd$，我们发现前半段和后半段的相对顺序变了，但是内部顺序没变，因此可以通过整体反转数组然后再局部反转。

1. 反转数组，$abcd1234 \Longrightarrow 4321dcba$。
2. 反转前半部分，$4321dcba \Longrightarrow 1234dcba$。
3. 反转后半部分，$1234dcba \Longrightarrow 1234abcd$。

$O(n) + O(1)$

```
1  void cyclicShift(int[] numbers, int k) {
2      int n = numbers.length;
3      k %= n;
4      reverse(numbers, 0, n - 1);
5      reverse(numbers, 0, n - k - 1);
6      reverse(numbers, n - k, n - 1);
7  }
8
9  void reverse(int[] numbers, int start, int end) {
10     for (; start < end; start++, end--) {
11         int tmp = numbers[start];
12         numbers[start] = numbers[end];
13         numbers[end] = tmp;
14     }
15 }
```

Chap02_17_ArrayCyclicShift.java

2.18 数组分割

问题

有一个无序、元素个数为 2n 的正整数数组，要求：如何能把这个数组分割为元素个数为 n 的两个数组，并使两个子数组的和最接近。

解法

2.19 区间重合判断

问题

给定一个源区间 $[x, y](y \geq x)$ 和 N 个无序的目标区间 $[x_1, y_1][x_2, y_2][x_3, y_3] \ldots [x_n, y_n]$，判断源区间 $[x, y]$ 是不是在目标区间内，也即 $[x, y] \in \bigcup_{i=1}^{n} [x_i, y_i]$?

解法

先将目标区间合并再判断源区间是否在合并之后的目标区间。合并操作可以通过排序进行优化，将目标区间按照 x 坐标升序排序，然后利用贪心算法从前到后遍历目标区间，如果当前区间和前一个区间有交集，即 $x_i <= y_{i-1}$ 就合并两个区间。查找源区间是否在目标区间可以词啊用二分查找，只需要查找最后一个 x 坐标小于源区间 x 坐标的区间然后判断即可。合并操作需要 $O(nlgn)$，查找操作需要 O(lgn)，总的时间复杂度为 $O(nlgn)$。

$O(nlgn) + O(1)$

```
class Interval {
    int start, end;

    Interval(int s, int e) {
        start = s;
        end = e;
    }
}

boolean isIntervalsOverlapping(Interval[] target, Interval source) {
    if (target.length <= 0) {
        return false;
    }
    int n = merge(target);
    return search(target, source, n);
}

int merge(Interval[] target) {
    Arrays.sort(target, new Comparator<Interval>() {
        @Override
        public int compare(Interval a, Interval b) {
            return a.start > b.start ? 1 : -1;
        }
    });
    int len = 0;
    for (int i = 1; i < target.length; i++) {
        if (target[i].start > target[len].end) {
            target[++len] = target[i];
        } else if (target[i].end > target[len].end) {
        target[len].end = target[i].end;
    }
}
return len + 1;
}

private boolean search(Interval[] target, Interval source, int n) {
    int left = 0, right = n - 1;
    while (left <= right) {
        int mid = left + (right - left) / 2;
        if (source.start >= target[mid].start && source.end <= target[mid].end) {
            return true;
        }
        if (source.start >= target[mid].start) {
            left = mid + 1;
        } else {
        right = mid - 1;
    }
```

```
48  }
49  return false;
50  }
```
Chap02_19_IntervalsOverlapping.java

扩展问题 1

如何判断两个矩形是否相交？这里给出的矩形的两条边都是平行于坐标轴的。

解答

因为这里讨论的矩形都是平行于坐标轴的，所以可以用矩形的左下角和右上角坐标唯一表示一个矩形。

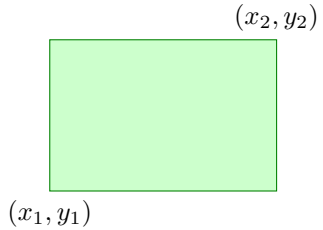

图 2-6 矩形窗口

这里先给出矩形的定义：

Rectangle.java
```
1   class Rectangle {
2       int x1, y1, x2, y2;
3
4       Rectangle(int x1, int y1, int x2, int y2) {
5           this.x1 = x1;
6           this.y1 = y1;
7           this.x2 = x2;
8           this.y2 = y2;
9       }
10  }
```
Rectangle.java

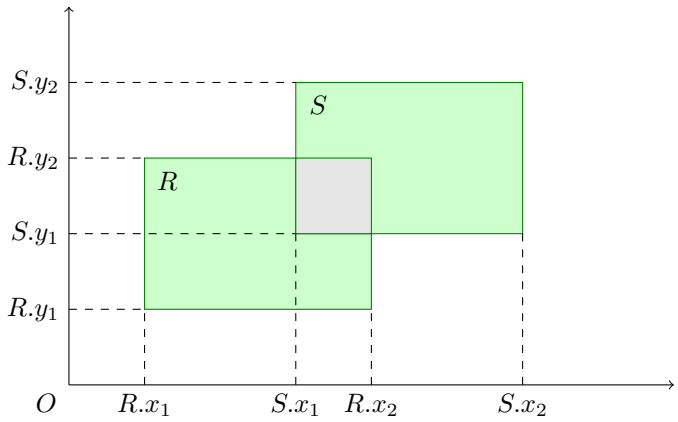

图 2-7 矩形窗口相交

对于平行于坐标轴的两个矩形，我们可以将它们分别投影到 X 坐标轴和 Y 坐标轴。矩形 R(x1,y1,x2,y2) 在 X 轴上的投影就是线段 $[x1, x2]$，在 Y 轴上的投影就是线段 $[y1, y2]$。此时，如果两个矩形在 X 轴和 Y 轴上的投影都相交则说明两个矩形相交。

$O(1) + O(1)$

```
1  boolean isRectangleIntersect(Rectangle R, Rectangle S) {
2      return R.x1 <= S.x2 && R.x2 >= S.x1
3          && R.y1 <= S.y2 && R.y2 >= S.y1;
4  }
```

Chap02_19_IntervalsOverlapping.java

扩展问题 2

如何处理二维空间的覆盖问题？例如在 Windows 桌面上有若干窗口，如何判断某一个窗口是否完全被其他窗口覆盖？

解答

$O(n^3) + O(n)$

```
1  private List<Integer> xCoordinate;
2  private List<Integer> yCoordinate;
3
4  boolean isRectangleOverlapping(Rectangle[] rects, Rectangle target) {
5      int n = rects.length;
6      xCoordinate = new ArrayList<Integer>(2 * (n + 1));
7      yCoordinate = new ArrayList<Integer>(2 * (n + 1));
8      boolean[][] flag = new boolean[2 * n][2 * n];
9
10     for (int i = 0; i < n; i++) {
```

```
            addToList(rects[i]);
        }
        addToList(target);

        Collections.sort(xCoordinate);
        Collections.sort(yCoordinate);

        for (int k = 0; k <= n; k++) {
            Rectangle rect = target;
            if (k < n) {
                rect = rects[k];
            }
            for (int i = 0; i < xCoordinate.size() && xCoordinate.get(i) < rect.x2; i++) {
                if (xCoordinate.get(i) < rect.x1) {
                    continue;
                }
                for (int j = 0; j < yCoordinate.size() && yCoordinate.get(j) < rect.y2; j++) {
                    if (yCoordinate.get(j) < rect.y1) {
                        continue;
                    }
                    if (k == n && !flag[i][j]) {
                        return false;
                    }
                    flag[i][j] = true;
                }
            }
        }
        return true;
    }

    private void addToList(Rectangle rect) {
        if (!xCoordinate.contains(rect.x1)) {
            xCoordinate.add(rect.x1);
        }
        if (!xCoordinate.contains(rect.x2)) {
            xCoordinate.add(rect.x2);
        }
        if (!yCoordinate.contains(rect.y1)) {
            yCoordinate.add(rect.y1);
        }
        if (!yCoordinate.contains(rect.y2)) {
            yCoordinate.add(rect.y2);
        }
    }
```

Chap02_19_IntervalsOverlapping.java

2.20 程序理解和时间分析

问题

阅读程序，回答一下问题：

Chap02_19_IntervalsOverlapping.java

```
public static void main(String[] args) {
    long startTime = System.currentTimeMillis();
    int[] rg = new int[]{2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16,
        17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31};
    for (long i = 1; i < Long.MAX_VALUE; i++) {
        int hit = 0;
        int hit1 = -1;
        int hit2 = -1;
        for (int j = 0; j < rg.length && hit <= 2; j++) {
            if (i % rg[j] != 0) {
                hit++;
                if (hit == 1) {
                    hit1 = j;
                } else if (hit == 2) {
                    hit2 = j;
                } else {
                    break;
                }
            }
        }
        if (hit == 2 && hit1 + 1 == hit2) {
            System.out.println(i);
            break;
        }
    }
    long endTime = System.currentTimeMillis();
    System.out.println(endTime-startTime);
}
```

Chap02_19_IntervalsOverlapping.java

1. 这个程序要找的是符合什么条件的数？
2. 这样的数存在吗？符合这一条件的最小数是什么？
3. 在电脑上运行这一程序，你估计多长时间才能输出第一个结果？时间精确到分钟。

解法

根据 if (hit == 2 && hit1 + 1 == hit2) 这一行代码可以看出程序要找的数字 N 满足：

1. 在 $[2,3,\ldots,31]$ 中，只有两个数 K_1 和 K_2 不能整除 N。

2. 而且这两个不能整除 N 的数相差为 1，即 $K_1+1=K_2$。

我们可以先假设满足条件的 $K_1=2,K_2=3$，那么其他整数都能整除 N，但是很明显只要能被 4 整除就能被 2 整除。因此，我们可以想到只有能被 16，17 整除，但是不被其他整除的数字。所以 $N=2^3*3^2*5^2*7*11*13*19*23*29*31=2123581660200$。

2.21 只考加法的面试题

问题

一个整数可以表示成一系列连续整数的和，比如 $9=4+5=2+3+4$。

1. 写一个程序，对于一个 64 位正整数 N，输出 N 所有可能的连续自然数之和的算式。

2. 有一些数字不能表示成连续自然数的和，这些数有什么规律，能否证明。

3. 64 位正整数范围内，子序列数目最多的数是哪个数?

解法

$O(nlgn)+O(1)$

```
List<List<Long>> output(long number) {
    long left = 1;
    List<List<Long>> ans = new ArrayList<List<Long>>();
    for (long i = 2; i < Math.sqrt(2) * Math.sqrt(number); i++) {
        if ((number - left) % i == 0) {
            List<Long> list = new ArrayList<Long>();
            for (long j = (number - left) / i; j < (number - left) / i + i; j++) {
                list.add(j);
            }
            ans.add(list);
        }
        left += i;
    }
    return ans;
}
```

Chap02_21_Add.java

$2^n(n\geq 0)$ 不能表示为连续自然数的和。

猜想：N 的子序列个数。

子序列数目最多的一个数是 $3^4*5^3*7*11*13*17*19*23*29*31*37*41*47$。

第 3 章
结构之法

这一章是关于字符串和链表的一些算法，也是面试中常考的数据结构。

这里我们先给出链表节点的定义。

ListNode.java
```
class ListNode {
    public int data;
    public ListNode next;

    public ListNode(int x) {
        data = x;
        next = null;
    }
}
```
ListNode.java

此外，本章还涉及到一些二叉树的算法，这里也先给出二叉树节点的定义。

TreeNode.java
```
class TreeNode {
    public int data;
    public ListNode left, right;

    public TreeNode(int x) {
        data = x;
        left = right = null;
    }
}
```
TreeNode.java

3.1 字符串移位包含的问题

问题

给定两个字符串 S_1 和 S_2，要求判定 S_2 是否能够被 S_1 做循环移位（rotate）得到的字符串包含。例如，给定 $S_1 = AABCD$ 和 $S_2 = CDAA$，返回 true，给定 $S_1 = ABCD$ 和 $S_2 = ACBD$，返回 false。

解法一

对 S_1 做循环移位的操作等同于字符串连接操作 S_1S_1，如果满足 $|S_1| \geq |S_2|$ 和 $S_2 \subset S_1S_1$，那么 S_2 可以由 S_1 循环移位所得。时间复杂度为 $O(N)$，其中 $N = |S_1|$。

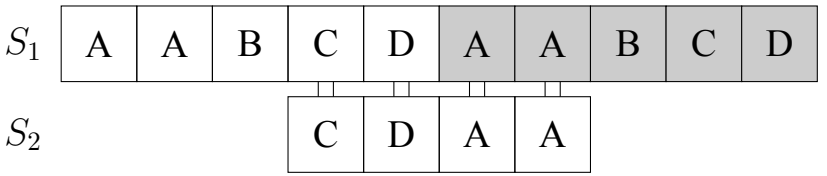

图 3-1 循环移位

$O(N) + O(N)$

```
1  boolean shiftSearch(String source, String target) {
2      if (source.length() < target.length()) {
3          return false;
4      }
5      source += source;
6      // 也可以直接调用 return source.contains(target);
7      return search(source, target);
8  }
9  // 这里也给出了字符串查找的代码，复杂度 O(m*n)。
10 boolean search(String source, String target) {
11     if (source.length() < target.length()) {
12         return false;
13     }
14     int j = 0;
15     for (int i = 0; i < source.length() && j < target.length(); i++) {
16         if (source.charAt(i) == target.charAt(j)) {
17             j++;
18         } else {
19             i -= j;
20             j = 0;
21         }
22     }
23     return j >= target.length();
24 }
```

Chap03_01_StringShiftSearch.java

解法二

解法一中申请了新的空间，其实是不必要的，我们可以通过直接操作 S_1 的下标 i，使得 $0 \leq i \leq 2 * |S_1| - 1$，这样问题就变成了简单的字符串查找问题了。

$O(N) + O(1)$

```
1  boolean shiftSearch_2(String source, String target) {
2      if (source.length() < target.length()) {
```

```
3           return false;
4       }
5       int sLen = source.length();
6       int tLen = target.length();
7       int j = 0;
8       for (int i = 0; i < sLen * 2 && j < tLen; i++) {
9           if (source.charAt(i % sLen) == target.charAt(j)) {
10              j++;
11          } else {
12              i -= j;
13              j = 0;
14          }
15      }
16      return j >= tLen;
17  }
```

Chap03_01_StringShiftSearch.java

3.2 电话号码对应英文单词

问题

电话的号码盘一般可以用于输入字母，比如用 2 可以输入 A,B,C, 用 3 可以是如 D,E,F，对于号码 5869872，可以依次输出其代表的所有字母组合，如 JTMWTPA……

解法一

利用递归，找出所有组合。

$O(3^N) + O(3^N)$

```
1   public static final String[] maps = {"", "", "ABC", "DEF",
2                                        "GHI", "JKL", "MNO",
3                                         "PQRS", "TUV", "WXYZ"};
4
5   List<String> findWords(String phoneNumber) {
6       List<String> words = new ArrayList<String>();
7       words.add("");
8       words = dfs(words, phoneNumber, 0);
9       return words;
10  }
11
12  private List<String> dfs(List<String> words, String phoneNumber, int start) {
13      if (start >= phoneNumber.length()) {
14          return words;
15      }
16      List<String> list = new ArrayList<String>();
17      int number = phoneNumber.charAt(start) - '0';
```

```
18      for (String word : words) {
19          for (int j = 0; j < maps[number].length(); j++) {
20              list.add(word + maps[number].charAt(j));
21          }
22      }
23      return dfs(list, phoneNumber, start + 1);
24  }
```

Chap03_02_PhoneNumber.java

解法二

非递归版本。

$O(3^N) + O(3^N)$

```
1   public static final String[] maps = {"", "", "ABC", "DEF",
2                                        "GHI", "JKL", "MNO",
3                                        "PQRS", "TUV", "WXYZ"};
4
5   List<String> findWords_2(String phoneNumber) {
6       List<String> words = new ArrayList<String>();
7       words.add("");
8       for (int i = 0; i < phoneNumber.length(); i++) {
9           List<String> list = new ArrayList<String>();
10          for (String word : words) {
11              int number = phoneNumber.charAt(i) - '0';
12              for (int j = 0; j < maps[number].length(); j++) {
13                  list.add(word + maps[number].charAt(j));
14              }
15          }
16          words = list;
17      }
18      return words;
19  }
```

Chap03_02_PhoneNumber.java

3.3 计算字符串的相似度

问题

字符串之间的相似度等于“距离 +1”的倒数，而字符串之间的距离定义为将一个字符串变换到另一个字符串的操作次数。这里我们定义了三种操作：

解法一

算法时间复杂度为 $O(N * M)$，其中 $N = |S|, M = |T|$。

$O(N * M) + O(N * M)$

```
int similarity(String s, String t) {
    s = "$" + s;
    t = "$" + t;
    int sLen = s.length();
    int tLen = t.length();
    int[][] dist = new int[sLen][tLen];
    for (int i = 0; i < tLen; i++) {
        dist[0][i] = i;
    }
    for (int i = 0; i < sLen; i++) {
        dist[i][0] = i;
    }
    for (int i = 1; i < sLen; i++) {
        for (int j = 1; j < tLen; j++) {
            dist[i][j] = dist[i - 1][j - 1]
                    + (s.charAt(i) == t.charAt(j) ? 0 : 1);
            dist[i][j] = Math.min(dist[i][j],
                    Math.min(dist[i - 1][j], dist[i][j - 1]) + 1);
        }
    }
    return dist[sLen - 1][tLen - 1];
}
```

Chap03_03_StringSimilarity.java

解法二

可以采用滚动数组来节省空间。

$O(N * M) + O(M)$

```
int similarity(String s, String t) {
    s = "$" + s;
    t = "$" + t;
    int sLen = s.length();
    int tLen = t.length();
    int[][] dist = new int[2][tLen];
    for (int i = 0; i < tLen; i++) {
        dist[0][i] = i;
    }
    dist[1][0] = 1;
    for (int i = 1; i < sLen; i++) {
        for (int j = 1; j < tLen; j++) {
            dist[i % 2][j] = dist[(i + 1) % 2][j - 1]
                    + (s.charAt(i) == t.charAt(j) ? 0 : 1);
            dist[i % 2][j] = Math.min(dist[i % 2][j],
                    Math.min(dist[(i + 1) % 2][j], dist[i % 2][j - 1]) + 1);
        }
    }
```

```
19      return dist[(sLen + 1) % 2][tLen - 1];
20  }
```
Chap03_03_StringSimilarity.java

3.4 从无头单链表中删除节点

问题

假设有一个没有头指针的单链表，一个指针指向此单链表中间的一个节点（不是第一个，也不是最后一个），请将该节点从单链表中删除。

解法

$O(1) + O(1)$

```
1   void deleteNode(ListNode p) {
2       if (p.next == null) {
3           return;
4       }
5       p.data = p.next.data;
6       p.next = p.next.next;
7   }
```
Chap03_04_DeleteLinkedListNode.java

3.5 编程判断两个链表是否相交

问题

给出两个单向链表的头指针，判断这两个链表是否相交。这里假设两个链表均不带环。

解法

如果两个链表相交，那么这两个链表的最后一个指针一定是相同的。因此我们只要记录下两个链表最后一个指针 A_{tail} 和 B_{tail}，判断它们是否相同即可。

$O(L) + O(0)$

```
1   boolean isIntersection(ListNode headA, ListNode headB) {
2       if (headA == null || headB == null) {
3           return false;
4       }
5       while (headA.next != null) {
6           headA = headA.next;
7       }
```

```
8       while (headB.next != null) {
9           headB = headB.next;
10      }
11      return headA == headB;
12  }
```

Chap03_06_LinkedListIntersection.java

扩展问题

如何求出这两个链表相交的第一个节点呢？

解法

既然相交，那么最后一个节点一定相同，然后就去判断倒数第二个节点是否相同，以此类推。现在的问题就是这是单向链表，所以只能从头向后遍历，因此我们可以先让两个链表的长度变成一样的，就是遍历那个长的链表直到长度和短链表相同。然后，这时候就可以同时遍历这两个链表并判断节点是否相同，若相同则返回。直到遍历到链表的结尾，一定会有一个节点返回（因为最后一个节点一定是相同的）。

$O(L) + O(1)$

```
1   ListNode findIntersection(ListNode headA, ListNode headB) {
2       int lenA = lengthOfLinkedList(headA);
3       int lenB = lengthOfLinkedList(headB);
4       while (lenA >= 0) {
5           if (headA == headB) {
6               return headA;
7           }
8           int diff = lenA - lenB;
9           if (diff >= 0) {
10              lenA--;
11              headA = headA.next;
12          }
13          if (diff <= 0) {
14              lenB--;
15              headB = headB.next;
16          }
17      }
18      return null;
19  }
20  // 计算链表长度
21  int lengthOfLinkedList(ListNode head) {
22      int length = 0;
23      while (head != null) {
24          length++;
25          head = head.next;
```

```
26      }
27      return length;
28  }
```

Chap03_06_LinkedListIntersection.java

3.6 队列中取最大值操作问题

问题

假设有这样一个拥有 3 个操作的队列：

1. EnQueue(v): 将 v 加入队列中。
2. DeQueue: 删除队首元素并返回。
3. MaxElement: 返回队列中的最大元素。

请设计一种数据结构和算法，让 MaxElement 操作的时间复杂度尽可能的降低。

解法

增加一个 max 数组表示队列中最大元素，对于入队操作，如果元素 v 比 max 数组最后一个元素大，就删除 max 最后一个元素，将 v 加入到 max 数组中。反之，如果 v 小于 max 最后一个元素，就直接将 v 加入到 max 数组中。对于出队操作，如果出队元素和 max 数组第一个元素相同，就删除 max 第一个元素。

$O(1) + O(N)$

```
1   public class MaxQueue {
2       Queue<Integer> queue = new LinkedList<Integer>();
3       List<Integer> max = new LinkedList<Integer>();
4       void push(int x) {
5           queue.offer(x);
6           if(max.size()==0 || x<=max.get(max.size()-1)) {
7               max.add(x);
8           } else {
9               max.remove(max.size()-1);
10              max.add(x);
11          }
12      }
13      int pop() {
14          int x = queue.poll();
15          if(x==max.get(0)) {
16              max.remove(0);
17          }
18          return x;
```

```
19      }
20      int maxElement() {
21          return max.get(0);
22      }
23  }
```

Chap03_07_MaxQueue.java

3.7 求二叉树中节点的最大距离

问题

我们定义二叉树中两个节点的距离为两个节点之间边的个数。求一棵二叉树中相距最远的两个节点之间的距离。

解法

我们先来定义经过节点 R 的最大距离为 $f(R)$，那么整棵树的最大距离就是 $f(T) = max_{R \in T} f(R)$。所以我们只要求出每个节点的 $f(R)$ 即可，我们会发现经过节点 R 的最大距离的两个节点一定分别来自左右两个子树（或者树根 R），并且一定是距离 R 最远的两个叶子节点。

所以我们推出下面的公式：

$$f(R) = H(R_{left}) + H(R_{right}) + 1$$

其中 $H(R)$ 是以 R 为树根的子树的高度。

$O(N) + O(1)$

```
1   int ans = 1;
2
3   public int maxDist(TreeNode root) {
4       maxHeight(root);
5       return ans - 1;
6   }
7
8   public int maxHeight(TreeNode root) {
9       if (root == null) {
10          return 0;
11      }
12      int leftDist = maxHeight(root.left);
13      int rightDist = maxHeight(root.right);
14      ans = Math.max(ans, leftDist + rightDist + 1);
15      return Math.max(leftDist, rightDist) + 1;
16  }
```

Chap03_08_MaxDistInBinaryTree.java

3.8 重建二叉树

问题

给出二叉树的前序遍历和中序遍历的结果，求出原始的二叉树。

解法

前序遍历结果为:$[a, b, d, c, e, f]$，
中序遍历结果为:$[d, b, a, e, c, f]$。
构造出的二叉树如图所示:

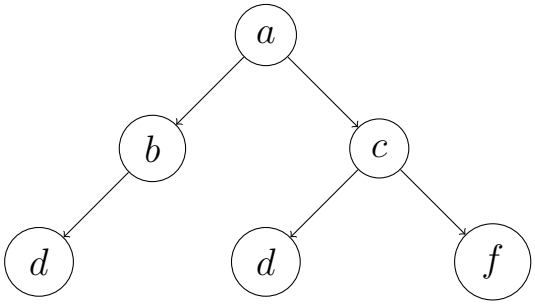

图 3-2 重建二叉树

对于前序遍历 $[\underline{a}, b, d, c, e, f]$ 中 $\underline{a}$ 一定是根节点，因此我们在中序遍历 $[d, b, \underline{a}, e, c, f]$ 中也找到 $\underline{a}$，根据中序遍历可以知道 $[d, b]$ 是左子树，$[e, c, f]$ 是右子树。从而可以递归构造左子树并返回左子树的根节点作为当前节点的左孩子，同理构造右子树。

$O(N) + O(1)$

```
1  public TreeNode buildTree(String preorder, String inorder) {
2      if (preorder == null) {
3          return null;
4      }
5      char val = preorder.charAt(0);
6      TreeNode node = new TreeNode(val);
7      int idx = inorder.indexOf(val);
8      if (idx > 0) {
9          node.left = buildTree(preorder.substring(1, idx + 1), inorder.substring(0, idx));
10     }
11     if (idx < inorder.length() - 1) {
12         node.right = buildTree(preorder.substring(idx + 1), inorder.substring(idx + 1));
13     }
14     return node;
15 }
```

Chap03_09_ConstructBinaryTree.java

3.9 分层遍历二叉树

问题

分层输出二叉树的节点。

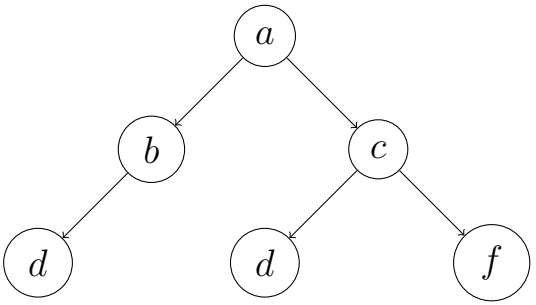

图 3-3 分层遍历二叉树

输出结果为:

a

b c

d e f

解法

利用队列。

$O(N) + O(N)$

```
1  public void traversal(TreeNode root) {
2      if (root == null) {
3          return;
4      }
5      Queue<TreeNode> queue = new LinkedList<TreeNode>();
6      queue.offer(root);
7      int curLayer = 1;
8      int nextLayer = 0;
9      while (!queue.isEmpty()) {
10         TreeNode node = queue.poll();
11         curLayer--;
12         System.out.printf("%d ", node.data);
13         if (node.left != null) {
14             queue.offer(node.left);
15             nextLayer++;
16         }
17         if (node.right != null) {
```

```
18              queue.offer(node.right);
19              nextLayer++;
20          }
21          if (curLayer == 0) {
22              System.out.println();
23              curLayer = nextLayer;
24              nextLayer = 0;
25          }
26      }
27  }
```

Chap03_10_TreeLevelOrderTraversal.java

3.10 程序改错

问题

找出一个有序数组 A 中等于目标值 targe 的元素的序号，如果有多个元素满足条件则返回序号最大的，如果没有则返回 -1。

解法

二分查找。

$O(lg(N)) + O(1)$

```
1   int bsearch3(int[] A, int target) {
2       int left = -1, right = A.length;
3       while (right - left > 1) {
4           int mid = (left + right) >>> 1;
5           if (A[mid] <= target) {
6               left = mid;
7           } else {
8               right = mid;
9           }
10      }
11      if (A[left] == target) {
12          return left;
13      }
14      return -1;
15  }
```

Chap03_11_DebugCode.java