1 订单编号格式校验

订单编号格式校验

题目描述：

在一个在线购物系统中，每个订单都有一个唯一的订单编号。订单编号由三部分组成：前缀、日期部分和序列号部分。前缀由大写字母组成，日期部分由 8 位数字组成，表示 YYYYMMDD 格式的日期，序列号部分由 4 位数字组成。例如："ORD202306151234" 现在需要你编写一个程序来判断这些订单编号是否符合以下规则：

1. 前缀部分必须是大写字母，长度为 3;
2. 日期部分必须是 8 位数字，且是一个有效的日期 (YYYYMMDD)。
3. 序列号部分必须是 4 位数字。
4. 编号必须以字母部分开头，日期部分中间，序列号部分结尾。

如果符合规则，输出"valid";
如果不符合规则，输出"invalid"。
提示: 日期这里需要判断闰年。闰年判断的条件是能被 4 整除，但不能被 100 整除; 或能被 400 整除。

输入描述：

第一行包含数据组数 n。
接下来的 n 行，每一行包含一个需要判断的订单编号字符串。

输出描述：

对于每一行，输出判断的结果。

样例输入 1：

```
6
ORD202306151234
ORD202313151234
ORD20230615123A
ORD20230615123
ORD2023061512345
ORD202306151234
```

样例输出 1：

```
valid
invalid
invalid
invalid
invalid
valid
```

思路分析：

1. 读取输入：
   - 使用 Scanner 类读取输入，首先获取输入的行数 n，然后依次读取每一行的订单编号。
2. 验证订单编号：
   - 对于每个订单编号，首先检查其长度是否为 15。如果长度不为 15，则直接返回”invalid”。
   - 如果长度为 15，则继续从字符串中提取订单编号的三部分：前缀、日期和序列号。
   - 分别对这三部分进行验证，确保其格式正确。

- 如果所有部分都合法，则返回”valid”，否则返回”invalid”。

3. 验证逻辑：
   - 前缀验证：提取订单编号的前 3 个字符，使用正则表达式 [A-Z]3 来确保前缀是 3 个大写字母。如果不匹配，返回”invalid”。
   - 日期验证：提取订单编号的第 4 到第 11 个字符，使用正则表达式 d8 确保日期部分是 8 位数字。如果不匹配或者日期无效，则返回”invalid”。具体的日期验证通过检查年、月、日是否合法。
   - 序列号验证：提取订单编号的最后 4 个字符，使用正则表达式 d4 确保序列号部分为 4 位数字。如果不匹配，返回”invalid”。
4. 日期合法性检查：
   - 年份：提取日期的前 4 位作为年份，判断是否为闰年（闰年的条件是能被 4 整除且不能被 100 整除，或者能被 400 整除）。
   - 月份：提取日期的中间两位作为月份，判断月份是否在 1 到 12 之间。
   - 日期：提取日期的最后两位作为日，检查这个日是否符合该月份的天数。比如 2 月最多 28 天，但如果是闰年则为 29 天。
5. 判断闰年：
   - 使用 `isLeapYear` 函数来判断一个年份是否是闰年。闰年的规则为：如果年份能被 4 整除但不能被 100 整除，或者能被 400 整除，则该年为闰年。

实现代码：

```java
// 主类，入口方法
public class Main {
    public static void main(String[] args) {
        // 使用Scanner从控制台获取输入
        Scanner sc = new Scanner(System.in);
        int n = sc.nextInt();  // 读取订单编号的数量
        sc.nextLine();  // 消耗换行符

        // 循环处理每一个订单编号
        for (int i = 0; i < n; i++) {
            String code = sc.nextLine();  // 读取一行订单编号
            System.out.println(validate(code));  // 验证该订单编号是否合法并输出结果
        }

        sc.close();  // 关闭Scanner
    }

    // 验证订单编号是否合法的主方法
    private static String validate(String code) {
        // 如果长度不为15，直接返回"invalid"
        if (code.length() != 15) {
            return "invalid";
        }

        // 提取订单编号的前缀、日期部分和序列号部分
        String pre = code.substring(0, 3);  // 前缀部分
        String date = code.substring(3, 11);  // 日期部分
        String serial = code.substring(11, 15);  // 序列号部分

        // 验证前缀部分是否为三个大写字母
        if (!pre.matches("[A-Z]{3}")) {
```

```
            return "invalid";
        }

        // 验证日期部分是否合法
        if (!isValidDate(date)) {
            return "invalid";
        }

        // 验证序列号部分是否为四位数字
        if (!serial.matches("\\d{4}")) {
            return "invalid";
        }

        // 如果所有部分都合法，返回"valid"
        return "valid";
    }

    // 验证日期是否合法的辅助方法
    private static boolean isValidDate(String date) {
        // 如果日期部分长度不是8位，或者不全是数字，返回false
        if (date.length() != 8 || !date.matches("\\d{8}")) {
            return false;
        }

        // 提取年份、月份和日期
        int y = Integer.parseInt(date.substring(0, 4));  // 年份
        int m = Integer.parseInt(date.substring(4, 6));  // 月份
        int d = Integer.parseInt(date.substring(6, 8));  // 日期

        // 检查月份是否在1到12之间
        if (m < 1 || m > 12) {
            return false;
        }

        // 检查日期是否大于0
        if (d < 1) {
            return false;
        }

        // 每个月的最大天数
        int[] days = {31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31};
        // 如果是闰年，2月的天数应为29天
        if (isLeapYear(y)) {
            days[1] = 29;
        }

        // 判断日期是否在该月的最大天数之内
        return d <= days[m - 1];
    }

    // 判断是否为闰年的辅助方法
    private static boolean isLeapYear(int y) {
        // 闰年的条件：能被4整除且不能被100整除，或者能被400整除
        return (y % 4 == 0 && y % 100 != 0) || (y % 400 == 0);
    }
}
```

2 统计以大写字母开头的单词个数

统计以大写字母开头的单词个数

题目描述：

小美写单词喜欢横着写，她记录了若干个人的名字，但是不小心加进去了一些无关的单词。

一个名字单词以大写字母开头，请你帮助她统计共有多少个人的名字。

输入描述：

在一行上输入一个长度为 $n(1 \leq n \leq 10^5)$、且由大小写字母和空格混合构成的字符串 s 代表小美的全部单词，每个单词之间使用空格间隔。

除此之外，保证字符串的开头与结尾字符不为空格。

输出描述：

在一行上输出一个整数，代表人名的个数。

样例输入 1：

```
ABC abc Abc
```

样例输出 1：

```
2
```

样例输入 2：

```
A A c
```

样例输出 2：

```
2
```

思路分析：

1. 字符串拆分：

   我们需要把输入字符串按空格拆分成若干个单词，这里可以使用 Java 语言自带的 `split` 函数，这个函数会把字符串根据空格分隔成一个单词数组。

2. 判断条件：

   - 遍历每个单词，检查它的首字符是否是大写字母。可以通过调用 `Character.isUpperCase()`（Java）或者直接比较字符的 `ASCII` 值来判断。
   - 如果某个单词的首字符是大写字母，则这个单词计数为一个名字。

3. 输出结果：最后将计数器的值输出，表示以大写字母开头的单词数量。

实现代码：

```java
import java.util.Scanner;

public class Main {
    public static void main(String[] args) {
        // 创建Scanner对象用于读取用户输入
        Scanner sc = new Scanner(System.in);

        // 读取一整行的输入，作为字符串存入变量s
        String s = sc.nextLine();

```

```
        // 将字符串按空格拆分成单词数组，存入数组w
        String[] w = s.split(" ");

        // 初始化计数器，记录以大写字母开头的单词个数
        int cnt = 0;

        // 遍历数组中的每一个单词
        for (String x : w) {
            // 如果单词的首字母是大写字母，计数器加1
            if (Character.isUpperCase(x.charAt(0))) {
                cnt++;
            }
        }

        // 输出计数器的最终值，即以大写字母开头的单词个数
        System.out.println(cnt);
    }
}
```

3 最小种树区间长度

最小种树区间长度

题目描述：

长度无限长的公路上，小美雇佣了 n 位工人来种树，每个点最多种一棵树。

从左向右数，工人所站的位置为 $a_1, a_2, \cdots, a_n$。已知每位工人都会将自己所在位置的右侧一段长度的区间种满树，且每位工人的种树区间长度相同。

现在小美希望公路上至少有 k 棵树，为了节约成本，他希望每位工人种树的区间长度尽可能短，请你帮他求出，工人们的种树区间至少多长，才能使得公路被种上至少 k 棵树。

输入描述：

第一行输入两个正整数 $n, k (1 \leq n, k \leq 2 \times 10^5)$，分别表示工人的数量，以及小美要求树的最少数量。

第二行输入 n 个正整数 $a_1, a_2, \cdots, a_n (1 \leq a_i \leq 2 \times 10^5)$，表示每名工人的位置。

输出描述：

在一行上输出一个整数，代表工人们最短的种树区间长度。

样例输入 1：

```
3 6
1 2 5
```

样例输出 1：

```
3
```

说明：

每位工人种树的区间长度至少为 3.

这样以来:

第一名工人种：1,2,3 点的树。

第二名工人种：2,3,4 点的树。

第三名工人种：5,6,7 点的树。

由于每个位置最多种一棵树，因此共有：1，2，3，4，5，6，7 这些点有树，满足至少 $k = 6$ 棵树。

可以证明，不存在比 3 更小的答案,

思路分析：

1. 排序工人位置：
   - 由于工人的位置是无序的，为了简化计算，我们首先对工人的位置进行排序。这样，种树的区间就可以方便地计算和遍历。
2. 二分查找确定最小区间长度：
   - 二分查找的本质是在一定的区间范围内查找最优解，这里我们需要查找最短的区间长度使得能种下至少 `k` 棵树。
   - 定义 `left` 为最短的可能区间（即 1），`right` 为最大的可能区间（即工人位置的最大值与最小值之差）。
3. 核心函数 `canPlantTrees`：
   - 在给定区间长度下，判断是否可以种至少 `k` 棵树。这个函数的核心是通过遍历工人的位置，计算每个工人能种的树的数量，并统计总的树数。如果总树数达到了 `k`，则返回 `true`。

实现代码：

```java
import java.util.Arrays;
import java.util.Scanner;

public class Main {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);

        // 读取输入值，n是工人数，k是需要种的树的数量
        int n = sc.nextInt();
        int k = sc.nextInt();
        int[] pos = new int[n];  // 存储每个工人的位置
        for (int i = 0; i < n; i++) {
            pos[i] = sc.nextInt();
        }

        // 对工人位置排序，方便后续计算
        Arrays.sort(pos);

        // 二分查找区间长度，left是最小区间长度，right是最大可能区间长度
        int left = 1;
        int right = pos[n - 1] - pos[0] + 1;
        int res = right;  // 保存最终结果

        // 开始二分查找
        while (left <= right) {
            int mid = (left + right) / 2;  // 中间的区间长度

            // 如果当前区间长度能种够k棵树，则尝试更小的区间长度
            if (canPlant(pos, n, k, mid)) {
                res = mid;
                right = mid - 1;
            } else {
                left = mid + 1;
            }
        }

        // 输出最小的区间长度
        System.out.println(res);
    }

    // 辅助函数，判断给定区间长度len时，能否种下至少k棵树
    private static boolean canPlant(int[] pos, int n, int k, int len) {
        int trees = 0;  // 已种的树的数量
        int lastPos = -1;  // 上一次种树的位置

        for (int i = 0; i < n; i++) {
            int start = pos[i];  // 当前工人的起始位置
            int end = start + len - 1;  // 工人可以种树的终止位置

            // 如果上一次种树的位置在当前工人的起始位置之前，直接种树
            if (lastPos < start) {
                trees += len;  // 可以种下整整一个区间长度的树
                lastPos = end;  // 更新最后种树的位置
            }
            // 如果上一次种树的位置在当前工人区域内，则只种剩下的部分
            else if (lastPos < end) {
                trees += end - lastPos;  // 计算剩下的区域长度并种树
                lastPos = end;  // 更新最后种树的位置
            }

            // 如果种下的树的数量已经达到或超过k棵，返回true
            if (trees >= k) {
```

```
                return true;
            }
        }

        // 最后检查是否种了足够的树
        return trees >= k;
    }
}
```