华为OD机试 - 乘坐保密电梯（Java & JS & Python）

题目描述

有一座保密大楼，你从0楼到达指定楼层m，必须这样的规则乘坐 电梯 ：

给定一个数字序列，每次根据序列中的数字n，上升n层或者下降n层，前后两次的方向必须相反，规定首次的方向向上，自行组织序列的顺序按规定操作到达指定楼层。

求解到达楼层的序列组合，如果不能到达楼层，给出小于该楼层的最近序列组合。

输入描述

第一行：期望的楼层，取值范围[1,50]; 序列总个数，取值范围[1,23]

第二行：序列，每个值取值范围[1,50]

输出描述

能够达到楼层或者小于该楼层最近的序列

备注

- 操作电梯时不限定楼层范围。
- 必须对序列中的每个项进行操作，不能只使用一部分。

用例

输入	5 3 1 2 6
输出	6 2 1
说明	1 2 6，6 2 1均为可行解，按先处理大值的原则结果为6 2 1

题目解析

我的解题思路如下：

由于题目说

> 给定一个数字序列，每次根据序列中的数字n，上升n层或者下降n层，前后两次的方向必须相反，规定首次的方向向上，自行组织序列的顺序按规定操作到达指定楼层

因此，我们可以认为要从数字序列中选出两组数，一组是上升，一组是下降

假设数字序列总个数n，期望楼层是t

- 如果n是偶数，则上升组要选n/2个，其余的都是下降组
- 如果n是奇数，则上升组要选n/2+1个，其余的都是下降组

假设数字序列所有元素之和是 sum，而上升组的和upSum，那么下降组的和就是sum - upSum，

如果 upSum - (sum - upSum) <= t，那么当前分组就是一个可能解，而其中sum，t都是已知的，因此可以转化为如下：

upSum <= (sum + t) / 2

即，我们要从给定的数字序列中选择固定数量（n/2或者n/2+1）个，让他的和小于且最接近或等于 (sum + t) / 2。

本题数量级不大，可以当成组合问题求解。

组合求解时，有个难点，那就是我们求解的组合是上升序列，而最终题目要求返回的是整体序列。因此如果求组合，我们只记录上升序列的话，则后期还要过滤剩余的下降序列，然后再将二者进行交叉合并，这就非常麻烦了。

我的思路是，定义一个boolean数组，名字是path，长度和nums相同，如果对应的nums[i]是上升序列的元素，则将path[i] = true。

最后，我们只要根据path即可快速分类出nums中哪些元素是上升序列的，哪些元素是下降序列的。

另外，题目自带用例的说明中提到：

> 1 2 6，6 2 1均为可行解，按先处理大值的原则结果为6 2 1

貌似是一个组合对应多个排列的情况，这个解决方案是，我们可以在求组合之前，就将nums进行降序，这样先被选进组合的元素必然是大值，因此最后交叉合并的目标序列肯定只会是6 2 1，而不会是1 2 6。

还有一个问题，那就是可能存在多个组合都是最优解的情况，此时我们依旧要按照先处理大值的原则，因此下面代码中我实现了compare方法来从多个最优解中选出先处理大值的。

```
import java.util.*;
import java.util.stream.Collectors;

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

    // 期望的楼层, 序列总个数
    int m = sc.nextInt(), n = sc.nextInt();

    // 序列
    Integer[] nums = new Integer[n];
```

```java
    for (int i = 0; i < n; i++) nums[i] = sc.nextInt();

    System.out.println(getResult(m, n, nums));
  }

  static int expectUpCount; // 上升序列组合的元素个数
  static int limitUpSum; // 上升序列组合的元素之和的上限
  static int maxUpSumBelowLimit = 0; // 记录不超过limitUpSum的最大的上升序列的和，该变量用于帮助ans剔除不是最优解的上升序列
  static ArrayList<boolean[]> paths = new ArrayList<>(); // paths记录求组合过程中发现的符合要求的序列

  public static String getResult(int m, int n, Integer[] nums) {
    // 计算expectUpCount、limitUpSum
    int sum = Arrays.stream(nums).reduce(Integer::sum).orElse(0);
    expectUpCount = n / 2 + n % 2;
    limitUpSum = (sum + m) / 2;

    // 将原始序列降序
    Arrays.sort(nums, (a, b) -> b - a);

    // 求组合
    dfs(nums, 0, new boolean[n], 0, 0);

    // 本题没有说明异常处理，我理解应该不会存在paths.size == 0的情况，但是为了保险，还是处理一下
    if (paths.size() == 0) {
      return "-1";
    }

    // 首先将paths转化为结果序列，然后对结果序列进行排序，排序规则是：按先处理大值的原则，最后取出最优解
    ArrayList<Integer> ans =
      paths.stream()
        .map(path -> getSeq(path, nums))
        .sorted((seq1, seq2) -> compare(seq1, seq2))
        .collect(Collectors.toList())
        .get(0);

    // 打印最优序列
    StringJoiner sj = new StringJoiner(" ");
```

```
50      for (Integer v : ans) sj.add(v + "");
51      return sj.toString();
52    }
53
54    /**
55     * 该方法用于选取出上升序列的组合
56     *
57     * @param nums 原始序列（降序排序后）
58     * @param index 当前层级能选取的元素范围的起始索引
59     * @param path
60     *      记录上升序列组合，注意path这里采用的是nums.length长度的boolean[]数组，如果path[i]为true，则代表nums[i]是上升，否则nums[i]就是下降
61     * @param sum 上升序列组合的元素和
62     * @param count 上升序列组合的元素个数
63     */
64    public static void dfs(Integer[] nums, int index, boolean[] path, int sum, int count) {
65      // 如果上升序列元素个数超过了expectUpCount, 则结束对应递归
66      if (count > expectUpCount) return;
67
68      if (count == expectUpCount) {
69        // 非更优解，直接剔除
70        if (sum < maxUpSumBelowLimit) return;
71
72        // 发现更优解，则清空paths
73        if (sum > maxUpSumBelowLimit) {
74          maxUpSumBelowLimit = sum;
75          paths.clear();
76        }
77
78        // 记录对应解
79        paths.add(Arrays.copyOf(path, path.length));
80        return;
81      }
82
83      for (int i = index; i < nums.length; i++) {
84        int num = nums[i];
85
86        if (sum + num > limitUpSum) continue;
87
```

```
88        path[i] = true;
89        dfs(nums, i + 1, path, sum + num, count + 1);
90        path[i] = false;
91      }
92    }
93
94    // 根据path解析出上升序列和下降序列，并对上升序列和下降序列进行交叉合并，形成目标序列
95    public static ArrayList<Integer> getSeq(boolean[] path, Integer[] nums) {
96      // 上升序列
97      LinkedList<Integer> up = new LinkedList<>();
98      // 下降序列
99      LinkedList<Integer> down = new LinkedList<>();
100
101     // 根据path解析出上升序列、下降序列
102     for (int i = 0; i < nums.length; i++) {
103       if (path[i]) up.add(nums[i]);
104       else down.add(nums[i]);
105     }
106
107     // 目标序列容器
108     ArrayList<Integer> seq = new ArrayList<>();
109
110     // 交叉合并
111     for (int i = 0; i < nums.length / 2; i++) {
112       seq.add(up.removeFirst());
113       seq.add(down.removeFirst());
114     }
115
116     if (up.size() > 0) {
117       seq.add(up.removeFirst());
118     }
119
120     return seq;
121   }
122
123   // 比较相同长度的两个集合的大小，按逐个元素比较
124   public static int compare(ArrayList<Integer> list1, ArrayList<Integer> list2) {
125     for (int i = 0; i < list1.size(); i++) {
126       int v1 = list1.get(i);
127       int v2 = list2.get(i);
```

```
128
129         if (v1 > v2) return -1;
130         if (v1 < v2) return 1;
131       }
132       return 0;
133     }
134 }
```