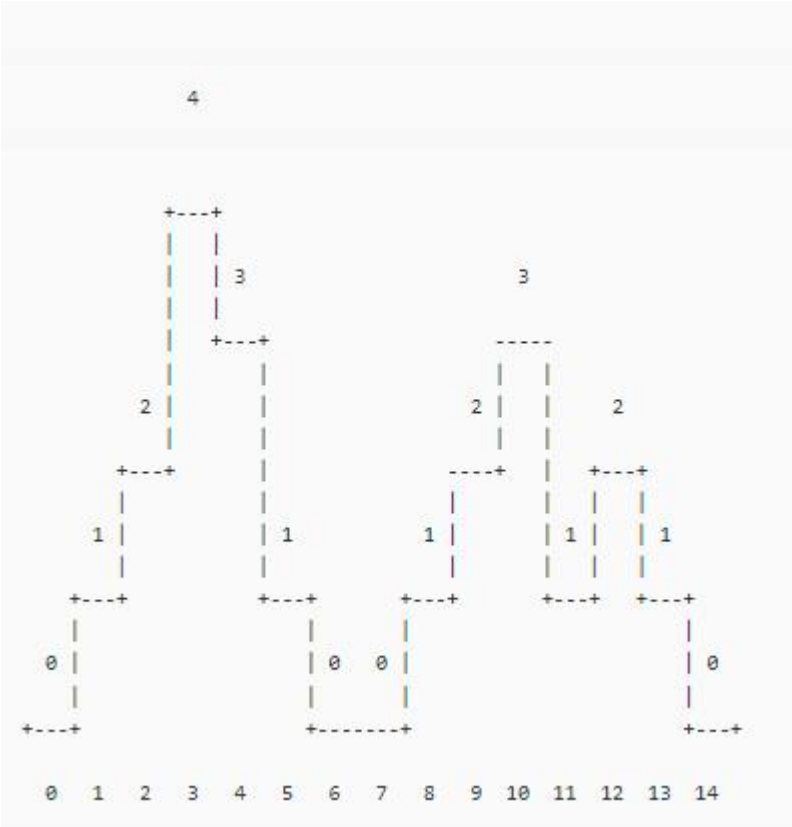
题目描述：

攀登者喜欢寻找各种地图，并且尝试攀登到最高的山峰。

地图表示为一维数组，数组的索引代表水平位置，数组的高度代表相对海拔高度。其中数组元素 0 代表地面。

例如[0,1,4,3,1,0,0,1,2,3,1,2,1,0]，代表如下图所示的地图，地图中有两个山脉位置分别为 1,2,3,4,5 和 8,9,10,11,12,13，最高峰高度分别为 4,3。最高峰位置分别为 3,10。

一个山脉可能有多座山峰(高度大于两边，或者在地图边界)。



2．登山时会消耗登山者的体力(整数)，上山时，消耗相邻高度差两倍的体力，下坡时消耗相邻高度差一倍的体力，平地不消耗体力，登山者体力消耗到零时会有生命危险。

例如，上图所示的山峰，从索引 0，走到索引 1，高度差为 1，需要消耗 2X1 = 2 的体力，从索引 2 高度 2 走到高度 4 索引 3 需要消耗 2X2 = 4 的体力。如果是从索引 3 走到索引 4 则消耗 1X1 的体力。

3．登山者体力上限为 999。

4．登山时的起点和终点可以是地图中任何高度为 0 的地面例如上图中的 0,6,7,14 都可以作为登山的起点和终点

攀登者想要评估一张地图内有多少座山峰可以进行攀登，且可以安全返回到地面，且无生命危险。

例如上图中的数组，有 3 个不同的山峰，登上位置在 3 的山可以从位置 0 或者位置 6 开始，从位置 0 登到山顶需要消耗体力 1X2+1X2+2X2 = 8，从山顶返回到地面 0 需要消耗体力 2X1 + 1X1 + 1X1 = 4 的体力，按照登山路线 0->3->0 需要消耗体力 12。攀登者至少需要 12 以上的体力（大于 12）才能安全返回。

示例 1

输入：

```
[0,1,4,3,1,0,0,1,2,3,1,2,1,0],13
```

输出：

```
3
```

说明：

登山者只能够登上位置 10 和 12 的山峰，7->10->7，14->12>14

示例 2

输入：

```
[1,4,3],999
```

输出：

```
0
```

说明：

没有合适的起点和终点

```
import java.util.*;


public class Solution {
    public static LinkedHashSet<Integer> getHillSet(int[] hill_map) {
        // write code here

        LinkedHashSet<Integer> hillSet = new LinkedHashSet<>();
        for (int i = 1; i < hill_map.length - 1; i++) {
            int i1Pre = hill_map[i - 1];
            int i1After = hill_map[i + 1];
            int i1 = hill_map[i];
            if (i1 > i1Pre && i1After < i1) {
                hillSet.add(i);
            }
        }
        if (hill_map.length > 1) {
            if (hill_map[0] > hill_map[1]) {
                hillSet.add(0);
            }
            if (hill_map[hill_map.length - 1] > hill_map[hill_map.length - 2]) {
                hillSet.add(hill_map.length - 1);
            }

        }
        return hillSet;

    }
    /**
     * 获取可以攀登的山峰数量
     * @param hill_map int 整型一维数组 地图
     * @param strength int 整型 登山者的体力值
     * @return int 整型
     */
    public     int count_climbable(int[] hill_map, int strength) {
```

```
// write code here

TreeSet<Integer> pSet = new TreeSet<>();
for (int i = 0; i < hill_map.length; i++) {
	int i1 = hill_map[i];
	if (i1 == 0) {
		pSet.add(i);
	}

}
if (pSet.size() == 0) {
	return 0;
}
LinkedHashSet<Integer> hillSet = getHillSet(hill_map);

HashMap<Integer, Integer> upMin = new HashMap<>();
HashMap<Integer, Integer> downMin = new HashMap<>();

for (Integer i : hillSet) {
	Integer leftStart = null;
	try {
		leftStart = pSet.floor(i);
	} catch (Exception e) {

	}
	if (leftStart != null) {
		int upSum = 0;
		int downSum = 0;
		for (int j = leftStart; j < i; j++) {
			if (hill_map[j + 1] > hill_map[j]) {
				upSum += (hill_map[j + 1] - hill_map[j]) * 2;
				downSum += (hill_map[j + 1] - hill_map[j]) * 1;
			} else if (hill_map[j + 1] < hill_map[j]) {
				upSum += (hill_map[j] - hill_map[j + 1]) * 1;
				downSum += (hill_map[j] - hill_map[j + 1]) * 2;
			}
		}
		if (upSum != 0) {
			if (upMin.get(i) == null) {
				upMin.put(i, upSum);
			} else if (upSum < upMin.get(i)) {
				upMin.put(i, upSum);
			}
		}
```

```
        if (downSum != 0) {
            if (downMin.get(i) == null) {
                downMin.put(i, downSum);
            } else if (downSum < downMin.get(i)) {
                downMin.put(i, downSum);
            }
        }

    }


    Integer rightStart = null;
    try {
        rightStart = pSet.ceiling(i);
    } catch (Exception e) {

    }
    if (rightStart != null) {
        int upSum = 0;
        int downSum = 0;
        for (int j = rightStart; j > i; j--) {
            if (hill_map[j - 1] > hill_map[j]) {
                upSum += (hill_map[j - 1] - hill_map[j]) * 2;
                downSum += (hill_map[j - 1] - hill_map[j]) * 1;
            } else if (hill_map[j - 1] < hill_map[j]) {
                upSum += (hill_map[j] - hill_map[j - 1]) * 1;
                downSum += (hill_map[j] - hill_map[j - 1]) * 2;
            }
        }
        if (upSum != 0) {
            if (upMin.get(i) == null) {
                upMin.put(i, upSum);
            } else if (upSum < upMin.get(i)) {
                upMin.put(i, upSum);
            }
        }
        if (downSum != 0) {
            if (downMin.get(i) == null) {
                downMin.put(i, downSum);
            } else if (downSum < downMin.get(i)) {
                downMin.put(i, downSum);
            }
        }
    }
```

```
            }

            List<Integer> upDownSumList = new ArrayList<>();

            for (Map.Entry<Integer, Integer> integerIntegerEntry : upMin.entrySet()) {
                Integer upKey = integerIntegerEntry.getKey();
                Integer upVal = integerIntegerEntry.getValue();
                Integer downMinVal = downMin.get(upKey);
                if (downMinVal != null) {
                    upDownSumList.add(upVal + downMinVal);
                }
            }

            Collections.sort(upDownSumList);

//              int strengthSum = 0;
            int count = 0;
            for (Integer i : upDownSumList) {
//                  strengthSum += i;
                if (i < strength) {
                    count++;
                } else {
                    break;
                }

            }

            return count;


        }
    }
```