Codility_

CodeCheck Report: trainingS6ZEX6-4CV

Test Name:

Check out Codility training tasks

Summary Timeline

Tasks summary

Task	Time spent	Score
SlalomSkiing ⚠ Java 8	1 min	100%

Total score

Tasks Details

1. **SlalomSkiing**

Hard

Given a sequence, find the longest subsequence that can be decomposed into at most three monotonic parts.

Task Score	Correctness	Performance
100%	100%	100%

Task description

You are a skier participating in a giant slalom. The slalom track is located on a ski slope, goes downhill and is fenced by barriers on both sides. The barriers are perpendicular to the starting line located at the top of the slope. There are N slalom gates on the track. Each gate is placed at a distinct distance from the starting line and from the barrier on the right-hand side (looking downhill).

You start from any place on the starting line, ski down the track passing as many gates as possible, and finish the slalom at the bottom of the slope. *Passing* a gate means skiing through the position of the gate.

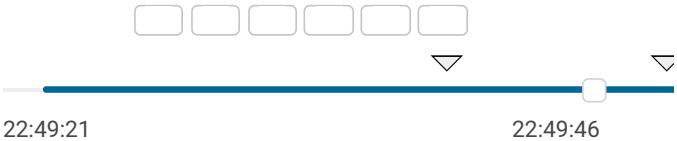
You can ski downhill in either of two directions: to the left or to the right. When you ski to the left, you pass gates of increasing distances from the right barrier, and when you ski to the right, you pass gates of decreasing distances from the right barrier. You want to ski to the left at the beginning.

Unfortunately, changing direction (left to right or vice versa) is exhausting, so you have decided to change direction *at most two* times during your ride. Because of this, you have allowed yourself to miss some of the gates on the way down the slope. You would

Solution

Programming language used:	Java 8
Total time used:	1 minutes
Effective time used:	1 minutes
Notes:	*not defined yet*

Task timeline



22:49:21 22:49:46

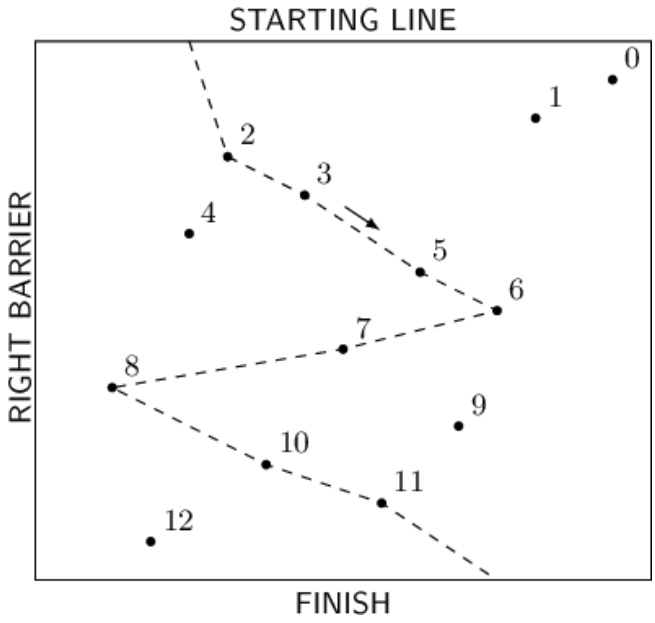
Code: 22:49:46 UTC, java, final, score: **100** show code in pop-up

like to know the maximum number of gates that you can pass with at most two changes of direction.

The arrangement of the gates is given as an array A consisting of N integers, whose elements specify the positions of the gates: gate K (for 0 ≤ K < N) is at a distance of K+1 from the starting line, and at a distance of A[K] from the right barrier.

For example, consider array A such that:

```
A[0] = 15
A[1] = 13
A[2] = 5
A[3] = 7
A[4] = 4
A[5] = 10
A[6] = 12
A[7] = 8
A[8] = 2
A[9] = 11
A[10] = 6
A[11] = 9
A[12] = 3
```



The picture above illustrates the example track with N = 13 gates and a course that passes eight gates. After starting, you ski to the left (from your own perspective). You pass gates 2, 3, 5, 6 and then change direction to the right. After that you pass gates 7, 8 and then change direction to the left. Finally, you pass gates 10, 11 and finish the slalom. There is no possible way of passing more gates using at most two changes of direction.

Write a function:

```
class Solution { public int solution(int[] A); }
```

that, given an array A consisting of N integers, describing the positions of the gates on the track, returns the maximum number of gates that you can pass during one ski run.

For example, given the above data, the function should return 8, as explained above.

For the following array A consisting of N = 2 elements:

```
A[0] = 1
A[1] = 5
```

the function should return 2.

```
1  // you can also use imports, for example:
2  import java.util.*;
3
4  // you can write to stdout for debugging purpo
5  // System.out.println("this is a debug message
6
7  class Solution {
8          /**
9           *
10          * @param A
11          * @return
12          * @see Original solution: https://bo
13          */
14         public int solution(int[] A) {
15                 int N = A.length;
16
17                 long[] dataWithDoubleMirror =
18                 long max = Arrays.stream(A).ma
19                 long mirror = 2 * max;
20
21                 for (int i = 0; i < N; i++) {
22                         dataWithDoubleMirror[
23                         dataWithDoubleMirror[
24                         dataWithDoubleMirror[
25                 }
26
27                 return findLongestSize(dataWit
28         }
29
30         private static int findLongestSize(lo
31                 long[] tempSequence = new long
32                 int maxLength = 0;
33
34                 for (long value : values) {
35                         int insertPosition = A
36                         if (insertPosition < 0
37                                 insertPositio
38                         }
39
40                         if (insertPosition ==
41                                 maxLength++;
42                         }
43
44                         tempSequence[insertPos
45                 }
46
47                 return maxLength;
48         }
49 }
```

Analysis summary

The solution obtained perfect score.

Analysis

Detected time complexity: **O(N * log(N))**

collapse all — Example tests

▼ example — ✔ OK

example test

1. 0.008 s OK

collapse all — Correctness tests

▼

Write an **efficient** algorithm for the following assumptions:

- N is an integer within the range [1..100,000];
- each element of array A is an integer within the range [1..1,000,000,000];
- the elements of A are all distinct.

simple ✔ OK
simple test, N <= 5

1. 0.012 s OK
2. 0.008 s OK
3. 0.012 s OK
4. 0.008 s OK
5. 0.008 s OK
6. 0.012 s OK
7. 0.012 s OK
8. 0.012 s OK
9. 0.008 s OK
10. 0.012 s OK
11. 0.008 s OK
12. 0.008 s OK

▼ small_permutation ✔ OK
small permutation

1. 0.008 s OK
2. 0.012 s OK
3. 0.008 s OK
4. 0.008 s OK
5. 0.008 s OK

▼ small_functional1 ✔ OK
small functional test

1. 0.012 s OK
2. 0.008 s OK
3. 0.008 s OK

▼ small_functional2 ✔ OK
small functional test

1. 0.012 s OK
2. 0.008 s OK
3. 0.008 s OK

▼ small_random ✔ OK
small random

1. 0.008 s OK
2. 0.008 s OK

▼ small_monotonic ✔ OK
small monotonic sequence with additional random elements

1. 0.012 s OK
2. 0.008 s OK
3. 0.008 s OK
4. 0.008 s OK
5. 0.012 s OK
6. 0.008 s OK

▼ small_shuffled_monotonic shuffled small monotonic sequences	✔ OK	
1. 0.008 s OK		
2. 0.008 s OK		
3. 0.008 s OK		
▼ small_concatenated_monotonic concatenated small monotonic sequences	✔ OK	
1. 0.012 s OK		
2. 0.008 s OK		

collapse all

Performance tests

▼ medium_random long subsequence hidden in random array, N <= 250	✔ OK	
1. 0.008 s OK		
2. 0.008 s OK		
▼ huge_random long subsequence hidden in random array, N <= 100,000	✔ OK	
1. 0.620 s OK		
2. 0.652 s OK		
▼ huge_monotonic huge monotonic sequence with additional random elements, N <= 100,000	✔ OK	
1. 0.636 s OK		
2. 0.544 s OK		
3. 0.068 s OK		
4. 0.068 s OK		
▼ huge_shuffled_monotonic shuffled huge monotonic sequences, N <= 100,000	✔ OK	
1. 0.468 s OK		
2. 0.036 s OK		
3. 0.056 s OK		
4. 0.060 s OK		
▼ huge_concatenated_monotonic concatenated huge monotonic sequences, N <= 100,000	✔ OK	
1. 0.544 s OK		
2. 0.532 s OK		
3. 0.072 s OK		
4. 0.048 s OK		