# Question

You are given an integer array A. From some starting index, you can make a series of jumps. The (1st, 3rd, 5th, ...) jumps in the series are called **odd-numbered jumps**, and the (2nd, 4th, 6th, ...) jumps in the series are called **even-numbered jumps**. Note that the **jumps** are numbered, not the indices.

You may jump forward from index i to index j (with i < j) in the following way:

* During **odd-numbered jumps** (i.e., jumps 1, 3, 5, ...), you jump to the index j such that A[i] <= A[j] and A[j] is the smallest possible value. If there are multiple such indices j, you can only jump to the **smallest** such index j.
* During **even-numbered jumps** (i.e., jumps 2, 4, 6, ...), you jump to the index j such that A[i] >= A[j] and A[j] is the largest possible value. If there are multiple such indices j, you can only jump to the **smallest** such index j.
* It may be the case that for some index i, there are no legal jumps.

A starting index is **good** if, starting from that index, you can reach the end of the array (index A.length - 1) by jumping some number of times (possibly 0 or more than once).

Return *the number of****good****starting indices*.

**Example 1:**

**Input:** A = [10,13,12,14,15]

**Output:** 2

**Explanation:**

From starting index i = 0, we can make our 1st jump to i = 2 (since A[2] is the smallest among A[1], A[2], A[3],

A[4] that is greater or equal to A[0]), then we cannot jump any more.

From starting index i = 1 and i = 2, we can make our 1st jump to i = 3, then we cannot jump any more.

From starting index i = 3, we can make our 1st jump to i = 4, so we have reached the end.

From starting index i = 4, we have reached the end already.

In total, there are 2 different starting indices i = 3 and i = 4, where we can reach the end with some number of

jumps.

**Example 2:**

**Input:** A = [2,3,1,1,4]

**Output:** 3

**Explanation:**

From starting index i = 0, we make jumps to i = 1, i = 2, i = 3:

During our 1st jump (odd-numbered), we first jump to i = 1 because A[1] is the smallest value in [A[1], A[2],

A[3], A[4]] that is greater than or equal to A[0].

During our 2nd jump (even-numbered), we jump from i = 1 to i = 2 because A[2] is the largest value in [A[2], A[3],

A[4]] that is less than or equal to A[1]. A[3] is also the largest value, but 2 is a smaller index, so we can

only jump to i = 2 and not i = 3

During our 3rd jump (odd-numbered), we jump from i = 2 to i = 3 because A[3] is the smallest value in [A[3], A[4]]

that is greater than or equal to A[2].

We can't jump from i = 3 to i = 4, so the starting index i = 0 is not good.

In a similar manner, we can deduce that:

From starting index i = 1, we jump to i = 4, so we reach the end.

From starting index i = 2, we jump to i = 3, and then we can't jump anymore.

From starting index i = 3, we jump to i = 4, so we reach the end.

From starting index i = 4, we are already at the end.

In total, there are 3 different starting indices i = 1, i = 3, and i = 4, where we can reach the end with some

number of jumps.

**Example 3:**

**Input:** A = [5,1,3,4,2]

**Output:** 3

**Explanation:**

We can reach the end from starting indices 1, 2, and 4.

**Constraints:**

* 1 <= A.length <= 2 \* 104
* 0 <= A[i] < 105

# Solution

#### **Approach 1: Monotonic Stack**

**Intuition**

First, we notice that where you jump to is determined only by the state of your current index and the jump number parity.

For each state, there is exactly one state you could jump to (or you can't jump.) If we somehow knew these jumps, we could solve the problem by a simple traversal.

So the problem reduces to solving this question: for some index i during an odd numbered jump, what index do we jump to (if any)? The question for even-numbered jumps is similar.

**Algorithm**

Let's figure out where index i jumps to, assuming this is an odd-numbered jump.

Let's consider each value of A in order from smallest to largest. When we consider a value A[j] = v, we search the values we have already processed (which are <= v) from largest to smallest. If we find that we have already processed some value v0 = A[i] with i < j, then we know i jumps to j.

Naively this is a little slow, but we can speed this up with a common trick for harder problems: a monotonic stack. (For another example of this technique, please see the solution to this problem: [(Article - Sum of Subarray Minimums)](https://leetcode.com/articles/sum-of-subarray-minimums/))

Let's store the indices i of the processed values v0 = A[i] in a stack, and maintain the invariant that this is monotone decreasing. When we add a new index j, we pop all the smaller indices i < j from the stack, which all jump to j.

Afterwards, we know oddnext[i], the index where i jumps to if this is an odd numbered jump. Similarly, we know evennext[i]. We can use this information to quickly build out all reachable states using dynamic programming.

Python:

|  |
| --- |
| class Solution(object):  def oddEvenJumps(self, A):  N = len(A)  def make(B):  ans = [None] \* N  stack = [] # invariant: stack is decreasing  for i in B:  while stack and i > stack[-1]:  ans[stack.pop()] = i  stack.append(i)  return ans  B = sorted(range(N), key = lambda i: A[i])  oddnext = make(B)  B.sort(key = lambda i: -A[i])  evennext = make(B)  odd = [False] \* N  even = [False] \* N  odd[N-1] = even[N-1] = True  for i in xrange(N-2, -1, -1):  if oddnext[i] is not None:  odd[i] = even[oddnext[i]]  if evennext[i] is not None:  even[i] = odd[evennext[i]]  return sum(odd) |

**Complexity Analysis**

* Time Complexity: O(N \log N)*O*(*N*log*N*), where N*N* is the length of A.
* Space Complexity: O(N)*O*(*N*).

#### **Approach 2: Tree Map**

**Intuition**

As in Approach 1, the problem reduces to solving this question: for some index i during an odd numbered jump, what index do we jump to (if any)?

**Algorithm**

We can use a TreeMap, which is an excellent structure for maintaining sorted data. Our map vals will map values v = A[i] to indices i.

Iterating from i = N-2 to i = 0, we have some value v = A[i] and we want to know what the next largest or next smallest value is. The TreeMap.lowerKey and TreeMap.higherKey functions do this for us.

With this in mind, the rest of the solution is straightforward: we use dynamic programming to maintain odd[i] and even[i]: whether the state of being at index i on an odd or even numbered jump is possible to reach.

Java

|  |
| --- |
| class Solution {  public int oddEvenJumps(int[] A) {  int N = A.length;  if (N <= 1) return N;  boolean[] odd = new boolean[N];  boolean[] even = new boolean[N];  odd[N-1] = even[N-1] = true;  TreeMap<Integer, Integer> vals = new TreeMap();  vals.put(A[N-1], N-1);  for (int i = N-2; i >= 0; --i) {  int v = A[i];  if (vals.containsKey(v)) {  odd[i] = even[vals.get(v)];  even[i] = odd[vals.get(v)];  } else {  Integer lower = vals.lowerKey(v);  Integer higher = vals.higherKey(v);  if (lower != null)  even[i] = odd[vals.get(lower)];  if (higher != null) {  odd[i] = even[vals.get(higher)];  }  }  vals.put(v, i);  }  int ans = 0;  for (boolean b: odd)  if (b) ans++;  return ans;  }  } |

**Complexity Analysis**

* Time Complexity: O(N \log N)*O*(*N*log*N*), where N*N* is the length of A.
* Space Complexity: O(N)*O*(*N*).