975. Odd Even Jump **Dynamic Programming** Array Ordered Set **Monotonic Stack Leetcode Link** Hard

This problem asks us to count "good" starting indices in an array arr. A good starting index is defined as an index from which it is

Problem Description

possible to reach the end of the array by making a series of jumps following specific rules. These jumps are of two types: oddnumbered jumps and even-numbered jumps. For odd-numbered jumps, you look for the smallest value in the array that is greater than or equal to the value at the current index and jump to the first occurrence of this smallest value. For even-numbered jumps, you look for the largest value that is less than or equal to the value at the current index and jump to the first occurrence of this largest value. If there are no valid jumps possible from an index, you cannot proceed from that point. The goal is to determine the count of indices from which reaching the end of the array is possible.

Intuition

SortedDict from the sortedcontainers module in Python. This allows us to efficiently locate the next jump index using binary search operations (in O(log n) time complexity).

Since the jumps need us to look up the smallest larger (or largest smaller) value, we can leverage a sorted data structure like

The intuition behind this solution lies in dynamic programming and the use of efficient data structures to keep track of legal jumps.

With dynamic programming, we can avoid recalculating whether it's possible to reach the end from a given index by storing the results of our calculations. We define a recursive function dfs which tries to determine if it is possible to reach the end of the array starting at index i and making a specific type of jump (odd or even). We then apply memoization using the cache decorator to store the results and avoid redundant computations.

For the overall solution, we iterate backward through the array. At each index, we use the SortedDict structure to find the correct jump indices for both odd and even jumps. Then, we attempt to find out if it's possible to reach the end from each index making an odd jump first (since we must start with an odd-numbered jump). The total count of "good" starting indices is the sum of all indices from which the end can be reached by following the jumping rules.

And then, use a depth-first search (DFS) strategy to explore possible paths from each potential starting index to the last index, using memoization to speed up the process significantly. **Solution Approach**

The crux of the solution is to carefully build a graph-like structure that maps each index to the next index for an odd or even jump.

The given Python solution follows a bottom-up dynamic programming strategy with memoization. Here is a step-by-step breakdown of the implementation details:

1. Memoization Decorator: The @cache decorator is applied to the dfs function. This is Python's built-in way to store results of

expensive function calls and return the cached result when the same inputs occur again. It helps to avoid repeated computation

for the same starting index and jump type.

going from n - 1 to 0), this structure allows us to:

2. Helper Function dfs: This is the function that attempts to find if the current index (i) can reach the end of the array with a sequence of odd and even jumps. The parameter k indicates the jump type, 0 for even and 1 for odd.

- \circ Base Case: If the current index is the last in the array (i = n 1), the function returns True as we've successfully reached the end. ∘ Recursive Case: If a jump is possible (g[i][k] is not -1), the function recursively calls itself, toggling the jump type (k ^ 1) to simulate the alternation of jumps.
- \circ **Jump Not Possible**: If no jump is possible from the current index (g[i][k] is -1), the function returns False. 3. Graph g: A 2D list g is initialized to keep track of next indices for odd and even jumps from each index. The first dimension corresponds to the index in arr and the second dimension (0 or 1) corresponds to the type of jump.

4. SortedDict sd: A SortedDict is used to efficiently find the next indices for jumps. As we iterate backward through the array (i

arr[i]. • Even-numbered jumps: Use sd.bisect_right(arr[i]) - 1 to find the largest index where arr[j] is smaller than or equal to arr[i].

5. Building the Graph g: For each index i, we update g[i][1] for odd jumps and g[i][0] for even jumps, or assign -1 if no jump is

possible. Here, sd.values() returns the list of indices in the order of their corresponding values in sd.

• Odd-numbered jumps: Use sd.bisect_left(arr[i]) to find the smallest index where arr[j] is greater than or equal to

boolean results. The solution smartly combines memoized recursion for dynamic programming and binary search within a sorted dictionary to maintain a balance between time complexity and the practical execution speed of the algorithm.

6. Calculating Good Starting Indices: Finally, the sum of all indices from which the end of the array can be reached by starting

with an odd jump is computed. We do this by invoking dfs(i, 1) for all indices from 0 to n - 1, and then taking the sum of these

Let's consider a small array arr = [5, 1, 3, 4, 2] to illustrate the solution approach: 1. Initialization: We prepare a 2D list g where g[i][0] and g[i][1] will store the next index for an even and an odd jump from the index i, respectively. We also create a SortedDict named sd.

∘ Insert (2, 4) into sd. \circ For i = 3 (value 4), the next odd jump goes to the smallest larger or equal value in sd (bisect_left(4)), but there is none, so g[3][1] = -1. For even jumps, the next jump is to the largest smaller or equal value in sd (bisect_right(4) - 1), which is

\circ For i = 2 (value 3), the next odd jump is to 4 and the next even jump is to 2. So g[2] = [4, 3]. Insert (3, 2) into sd.

Insert (4, 3) into sd.

the index of value 2, so g[3][0] = 4.

from sortedcontainers import SortedDict

i: current index

def oddEvenJumps(self, arr: List[int]) -> int:

from typing import List

class Solution:

8

9

10

16

18

19

20

21

22

23

24

25

30

31

32

33

34

35

36

35

36

37

38

39

40

41

42

43

44

45

47

48

49

50

51

52

53

54

56

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

26

27

28

29

30

31

32

33

34

35

36

37

38

39

40

41

42

43

44

50

51

56

58

63

64

68

69

70

57 }

55

C++ Solution

1 #include <vector>

2 #include <map>

6 class Solution {

public:

Example Walkthrough

Insert (1, 1) into sd.

 \circ For i = 1 (value 1), the next odd jump is to 3 and there is no even jump, so g[1] = [-1, 2].

example, the count is 1 (only index 1 leads to the end of the array by following jump rules).

Hence, for our array arr = [5, 1, 3, 4, 2], there is only one "good" starting index, which is the index 1.

2. Building the Graph: Start iterating from the end (i = 4) to the start (i = 0) of the array:

 \circ For i = 4 (value 2), sd is empty, so no jumps are possible. g[4] = [-1, -1].

 \circ For $i = \emptyset$ (value 5), there are no larger or smaller values than 5 in sd, so $g[\emptyset] = [-1, -1]$. 3. Calculating Good Starting Indices: We create a dfs function with memoization using the @cache decorator: \circ Start with index 0 and an odd jump (dfs(0, 1)). This function call returns False because g[0][1] = -1.

odfs(1, 1) returns True because g[1][1] = 2 and from index 2, an even jump leads to the end (dfs(2, 0) returns True).

4. Result: Count all True results from dfs(i, 1) for i in [0, 1, 2, 3, 4] to get the total number of "good" starting indices. In our

Python Solution

from functools import lru_cache # This is used for memoization in Python 3

Helper function to perform a depth-first search (dfs)

@lru_cache(maxsize=None) # use lru_cache for memoization

Perform the jump and toggle the jump_type (odd <-> even)

jump_graph[i][1] = sd.peekitem(next_jump_index)[1]

jump_graph[i][0] = sd.peekitem(next_jump_index)[1]

return dfs(jump_graph[index][jump_type], jump_type ^ 1)

jump_type: 1 for odd jumps, 0 for even jumps

n = len(arr) # Length of the input array

 $jump_graph = [[-1, -1] for _ in range(n)]$

Handle even jumps (descending order)

next_jump_index = sd.bisect_right(arr[i]) - 1

Build the graph in reverse

for i in range(n - 1, -1, -1):

if next_jump_index >= 0:

return goodStartingIndicesCount;

if (index == arrayLength - 1) {

return 1;

return 0;

#include <cstring> // For memset

#include <functional> // For std::function

int oddEvenJumps(std::vector<int>& arr) {

// Populate the next jump index array

for (int i = n - 1; i >= 0; --i) {

return 1;

return 0;

for (int i = 0; i < n; ++i) {

int n = arr.size(); // Size of the input array

return jumps[index][jumpType];

// Try to start a jump (odd) from each index

int jumps[n][2]; // Jump status (odd-0/even-1) array

int nextJump[n][2]; // Next jump index (odd-0/even-1) array

private int performJump(int index, int jumpType) {

if (nextIndices[index][jumpType] == -1) {

// Use memoization to avoid recomputation

if (dp[index][jumpType] != null) {

return dp[index][jumpType];

// Recursive function to determine if we can reach the end from index i

// If we've reached the end, return 1 (successful jump sequence)

// If there is no valid next jump, return 0 (failed to reach the end)

std::map<int, int> indicesMap; // Map to hold value-to-index mappings

memset(jumps, 0, sizeof(jumps)); // Initialize the jump status array to 0

auto it = indicesMap.lower_bound(arr[i]); // Iterator for the 'odd' jump

// Set next jump for 'even', using previous iterator if not the beginning

nextJump[i][0] = it == indicesMap.begin() ? -1 : std::prev(it)->second;

it = indicesMap.upper_bound(arr[i]); // Iterator for the 'even' jump

indicesMap[arr[i]] = i; // Update the map with the current index

std::function<int(int, int)> dfs = [&](int index, int jumpType) -> int {

// Perform the jump and store the result in the jump status array

// Depth-first search function to perform the jump calculation

int answer = 0; // Initialize answer to count successful jumps

52 // Mock implementation, replace 'some-library-to-handle-arrays' with the actual library

// This should find the first element in the map that is not less than 'value'

// This should find the first element in the map that is greater than 'value'

// Mock implementation for the prev function, replace with actual implementation

are performed on SortedDict to find the next indices for odd and even jumps.

// Implement the lowerBound logic specific to your requirements

// Implement the upperBound logic specific to your requirements

return { begin: () => false, value: 0 }; // Placeholder return

function prev(iterator: { value: number }): { value: number } {

// Implement the prev logic specific to your requirements

return { value: iterator.value }; // Placeholder return

return { end: () => false, value: 0 }; // Placeholder return

function lowerBound(map: Map<number, number>, value: number): { end: () => boolean, value: number } {

function upperBound(map: Map<number, number>, value: number): { begin: () => boolean, value: number } {

nextJump[i][1] = it == indicesMap.end() ? -1 : it->second; // Set next jump for 'odd'

if (index == n - 1) { // If we've reached the last index, the jump is successful

if (jumps[index][jumpType] != 0) { // If the jump status is already calculated, return it

if (nextJump[index][jumpType] == -1) { // If there's no next jump, return 0

return jumps[index][jumpType] = dfs(nextJump[index][jumpType], jumpType ^ 1);

// Perform the next jump with the alternating jump type (odd -> even, even -> odd)

return dp[index][jumpType] = performJump(nextIndices[index][jumpType], jumpType ^ 1);

o dfs(3, 1) returns False since no odd jump is possible from 3.

o dfs(4, 1) also returns False since no jump is possible from 4.

def dfs(index: int, jump_type: int) -> bool: 11 if index == n - 1: # If we've reached the last element, return True 12 13 if $jump_graph[index][jump_type] == -1: # If there's no valid <math>jump$, return False 14 return False 15

26 27 # Handle odd jumps (ascending order) 28 next_jump_index = sd.bisect_left(arr[i]) 29 if next_jump_index < len(sd):</pre>

Initialize a 2D array with an additional dimension representing the jump type

sd = SortedDict() # Use SortedDict to maintain a sorted order of elements

```
# Update the SortedDict with the current element
 37
 38
                 sd[arr[i]] = i
 39
             # We can start with an odd jump (1), so we check for each index if we can reach the end
 40
             return sum(dfs(i, 1) for i in range(n))
 41
 42
    # Example usage of the solution — uncomment to test:
 44 # sol = Solution()
 45 # print(sol.oddEvenJumps([10, 13, 12, 14, 15])) # Expected output: 2
 46
Java Solution
  1 class Solution {
         private int arrayLength;
         private Integer[][] dp;
         private int[][] nextIndices;
         // Main function to compute the number of good starting indices
         public int oddEvenJumps(int[] arr) {
             TreeMap<Integer, Integer> valueToIndexMap = new TreeMap<>();
  8
             arrayLength = arr.length;
  9
 10
             dp = new Integer[arrayLength][2];
             nextIndices = new int[arrayLength][2];
 11
 12
 13
             // Preprocessing: Populate the next indices for odd and even jumps for each element
 14
             for (int i = arrayLength - 1; i >= 0; --i) {
 15
                 // Find the smallest number greater than or equal to current number
 16
                 var higher = valueToIndexMap.ceilingEntry(arr[i]);
 17
                 // Update the next index for an odd jump
 18
                 nextIndices[i][1] = higher == null ? -1 : higher.getValue();
 19
 20
                 // Find the greatest number less than or equal to the current number
 21
                 var lower = valueToIndexMap.floorEntry(arr[i]);
 22
                 // Update the next index for an even jump
 23
                 nextIndices[i][0] = lower == null ? -1 : lower.getValue();
 24
 25
                 // Map the current number to its index
 26
                 valueToIndexMap.put(arr[i], i);
 27
 28
 29
             int goodStartingIndicesCount = 0;
             // Check each index as a starting point
 30
 31
             for (int i = 0; i < arrayLength; ++i) {</pre>
 32
                 // If the current index leads to the end by a sequence of jumps, include it in the count
 33
                 goodStartingIndicesCount += performJump(i, 1);
 34
```

49 }; 50

};

```
answer += dfs(i, 1); // Count only the successful jumps
 45
 46
 47
             return answer; // Return the number of successful jumps
 48
Typescript Solution
  1 // Importing necessary functions for the code to work
  2 import { lowerBound, prev } from 'some-library-to-handle-arrays'; // Placeholder, replace with actual library
    const n: number; // Size of the input array
    const indicesMap: Map<number, number> = new Map(); // Map to hold value-to-index mappings
    const jumps: number[][] = Array.from(Array(n), () => Array(2).fill(0)); // Jump status array for odd (0) and even (1) jumps
    const nextJump: number[][] = Array.from(Array(n), () => Array(2).fill(-1)); // Next jump index array for odd (0) and even (1) jumps
  9 function populateNextJump(arr: number[]): void {
      // Reverse iterate through the input array to populate the next jump indices
 10
       for (let i = n - 1; i >= 0; --i) {
 11
        const it = lowerBound(indicesMap, arr[i]); // Lower bound for the odd jump
 12
         nextJump[i][1] = it === indicesMap.end() ? -1 : it.value; // Set next jump index for the odd jump
 13
 14
 15
         const itUpper = upperBound(indicesMap, arr[i]); // Upper bound for the even jump
        // Set next jump index for the even jump, using the previous element if not at the beginning
 16
 17
         nextJump[i][0] = itUpper === indicesMap.begin() ? -1 : prev(itUpper).value;
 18
 19
         indicesMap.set(arr[i], i); // Update the indices map with the current index
 20
 21 }
 22
    function depthFirstSearch(index: number, jumpType: number): number {
      // If we've reached the last index, the jump is successful
 24
 25
      if (index === n - 1) {
 26
         return 1;
 27
      // If there's no next jump, the jump is not successful
 28
      if (nextJump[index][jumpType] === -1) {
 29
 30
         return 0;
 31
 32
      // If the jump status is already computed, return it
 33
      if (jumps[index][jumpType] !== 0) {
 34
        return jumps[index][jumpType];
 35
      // Calculate the jump status by jumping to the next index, and alternate the jump type
 36
       jumps[index][jumpType] = depthFirstSearch(nextJump[index][jumpType], 1 - jumpType);
 38
       return jumps[index][jumpType];
 39 }
 40
     function oddEvenJumps(arr: number[]): number {
       populateNextJump(arr); // Populate next jump indices before computing jumps
 42
 43
 44
       let answer: number = 0; // Initialize answer to count successful odd jumps from each index
 45
      // Iterate through each index in the input array to start an odd jump
 46
      for (let i = 0; i < n; ++i) {
         answer += depthFirstSearch(i, 1); // Add to the answer if the odd jump is successful
 47
 48
      return answer; // Return the total count of successful odd jumps
```

Time Complexity

call, after the initial computation.

balanced binary search tree (BST) through SortedDict.

Time and Space Complexity

• The bisect_left and bisect_right operations on a BST have a time complexity of O(log m), where m is the number of

Space Complexity

elements in the BST at the time of the query. In the worst case, m can be as large as n. \circ Inserting an element into the SortedDict also has a time complexity of $O(\log n)$. • The DFS function dfs could potentially be called for every starting position i with every jump type k. As memoization is used,

each pair (i, k) will be computed at most once. Therefore, we can consider it to have a constant time complexity 0(1) for each

The function oddEvenJumps involves both a dynamic programming approach (through memoization with @cache) and the use of a

• The loop that constructs the graph g runs for n iterations, where n is the length of arr. In each iteration, two bisect operations

- The final sum calls dfs for each starting position i with the initial jump type 1 (odd jump), totaling n calls to dfs. However, since dfs is memoized, the actual complexity of all these calls altogether is O(n).
- Combining these parts, the overall time complexity is $O(n \log n)$ due to the combination of the loop and the operations on the SortedDict within the loop, which dominates over the linear time complexity of the final sum.

• The space used by the memoization in dfs function will require 0(n * 2) space since it stores results for each element and jump

type (odd or even). This simplifies to O(n). • The graph g is a 2D array with dimensions n * 2, also taking O(n) space.

• The SortedDict can hold up to n elements, so it also consumes O(n) space.

Overall, combining the space for memoization, the graph g, and the SortedDict, the total space complexity is O(n).