1533. Find the Index of the Large Integer

Binary Search Medium <u>Array</u> Interactive

Problem Description

In this problem, we are given a unique integer array where all elements are equal except for one that is larger than the others. Instead of direct access to the array, we are provided an API ArrayReader with two functions: compareSub and length. Our objective is to find the index of the element that is larger than the rest using the ArrayReader API, with the constraint of calling compareSub a maximum of 20 times. The compareSub function compares the sum of two sub-arrays within given indices and returns 1 if the first is larger, -1 if the

second is larger, and 0 if they are equal. Length returns the size of the array. It's important to note that both API functions are considered to operate in constant time, 0(1).

Given that all elements of the array are equal except for one, we can use a divide and conquer strategy to narrow down the search space efficiently. To do this, we can partition the array into three equal parts and use the compareSub method to compare

Intuition

the sums of these parts. The intuition is that the unique element, which is larger, will cause the sum of the part containing it to be greater than the others. Based on the comparison results, we can eliminate two-thirds of the array in each step, as follows:

• If the sum of the first and the second third is the same (compareSub returns 0), the element must be in the last third. • If the sum of the first third is greater than the sum of the second (compareSub returns 1), the element must be in the first third. • If the sum of the second third is greater than the sum of the first (compareSub returns -1), the element must be in the second third.

3. t3 is left + 2 * ((right - left) // 3) + 1, which is two-thirds into the current search space.

- By repeating this process and shrinking the search interval, we eventually isolate the unique element's index. Our approach guarantees that we will find the index with a minimal number of calls to the compareSub function, well within the 20 call limit.

The implementation of the solution follows a ternary search approach to identify the segment containing the unique element that

The algorithm initializes two pointers, left and right, representing the search boundaries, which at the beginning correspond to

the start and end indices of the array, respectively.

Solution Approach

is larger than the others.

In each iteration of the while loop, which continues as long as left is less than right, the search space is divided into thirds by calculating two indices, t2 and t3:

1. t1 is set to left, the beginning of the current search space. 2. t2 is calculated as left + (right - left) // 3, which is one-third into the current search space.

adjusts the left and right pointers: • If compareSub returns 0, this means the sums of the first and second thirds are equal, implying that the unique element is not in either of the

The solution then uses these indices to call compareSub(t1, t2, t2 + 1, t3). This comparison effectively evaluates the sum of

the first third against the sum of the second third of the search space. Based on the return value of compareSub, the algorithm

latter two-thirds of the search space. • If compareSub returns -1, this shows that the unique element is in the second third, and the search space is updated by setting left to t2 + 1 and right to t3.

• If compareSub returns 1, this indicates that the first third contains the unique element, so the right pointer is adjusted to t2 to discard the

first two thirds. Thus, the algorithm eliminates these two segments from consideration by setting left to t3 + 1.

index of the element that is larger than the rest using the solution approach described above.

• t3 = left + 2 * ((right - left) // 3) + 1 = 0 + 2 * ((7 - 0) // 3) + 1 = 5.

• t3 = left + 2 * ((right - left) // 3) + 1 = 6 + 2 * ((7 - 6) // 3) + 1 = 7.

The loop continues until left equals right, which means the search space has been narrowed down to a single element—the unique element that is larger than the others. At this point, the algorithm returns left, the index of the unique element.

This approach ensures that the search space is halved at each step, making it possible to find the element within a logarithmic

number of comparisons, specifically log3(n), where n is the length of the array. Using ternary search instead of binary search

allows us to reduce the number of necessary comparisons, capitalizing on the specific nature of the problem. **Example Walkthrough**

We have a total of 8 elements in the array, so our initial left is 0 and our right is 7. • Initially, t1 = left = 0. • t2 = left + (right - left) // 3 = 0 + (7 - 0) // 3 = 2.

Let's consider that we have an ArrayReader that represents the following array: [1, 1, 1, 2, 1, 1, 1, 1]. We need to find the

We call compareSub(0, 2, 3, 5) and suppose it returns 0, meaning the sum of elements from index 0 to 2 is the same as the sum of elements from index 3 to 5. Therefore, our unique element must be in the last third of the array.

On the next iteration:

Python

class Solution:

t1 = left = 6. Since right remains the same, we calculate t2 and t3 within the new bounds. • t2 = left + (right - left) // 3 = 6 + (7 - 6) // 3 = 6.

We call compareSub(6, 6, 7, 7) and it can only return 1, because the unique larger element must be at index 7 (since 6 and 6

Now that left and right have converged (both are 7), we found our unique larger value at the index 7. We return 7.

are the same index, and the only remaining index is 7).

def getIndex(self. reader: 'ArrayReader') -> int:

left, right = 0, reader.length() - 1

Initialize two pointers for binary search

second part end = left + two thirds

if comparison result == 0:

elif comparison result == 1:

left = third part start

third_part_start = second_part_end + 1

Compare the sum of the first third with the sum of the second third

If both sums are equal, the pivot index must be in the third part

When 'left' is equal to 'right', we have found the pivot index

// Assuming the array contains a peak element (an element that is greater

int left = 0; // Begin search at the start of the array

// Divide the current range into three equal parts

// discard the parts outside of this range

// The narrowed down range will eventually converge to a single element.

// Compare sum of elements in the first and second parts

int part1End = left + (right - left) / 3;

int part2End = left + (right - left) / 3 * 2;

int part2Start = part1End + 1;

int part3Start = part2End + 1;

if (comparisonResult == 0) {

} else if (comparisonResult == 1) {

left = part3Start;

right = part1End;

left = part2Start;

right = part2End;

} else {

return left;

// than its neighbours), this function finds the index of one such peak element.

// Continue searching while the range is not narrowed down to one element

int right = reader.length() - 1; // End search at the last element of the array

int comparisonResult = reader.compareSub(left, part1End, part2Start, part2End);

// If sums are equal, the peak must be in the third part, discard first two parts

// If sum of first part is greater, the peak is in the first part, discard the rest

// If sum of second part is greater, the peak is between part2Start and part2End,

// This element is a peak (a candidate index where the element is not smaller than its neighbors).

If the sum of the first third is greater, the pivot index is in the first part

Divide the range into three equal parts

So now, we update left to t3 + 1 = 6.

constraints. Solution Implementation

This example demonstrates that by narrowing down the search space with each comparison and adjusting the pointers

accordingly, we can find the index of the unique larger element with a minimal number of calls to the API, satisfying our

and determine their endpoints one third = (right - left) // 3 two_thirds = 2 * one_third first part end = left + one third second part start = first part end + 1

comparison result = reader.compareSub(left, first part end, second part start, second part end)

```
right = first part end
# If the sum of the second third is greater, the pivot index is in the second part
else:
    left, right = second_part_start, second_part_end
```

return left

while left < right:</pre>

```
Java
class Solution {
    public int getIndex(ArrayReader reader) {
        // Initialize pointers for left and right boundaries
        int left = 0;
        int right = reader.length() - 1;
        // Use a modified version of binary search to find the specific index
        while (left < right) {</pre>
            // Divide the range into three equal parts
            int mid1 = left + (right - left) / 3;
            int mid2 = left + (right - left) / 3 * 2 + 1;
            // Compare the sum of the first two-thirds with the sum of the last two-thirds
            int comparisonResult = reader.compareSub(left, mid1, mid1 + 1, mid2);
            // If the sums are equal, the desired index is in the last third
            if (comparisonResult == 0) {
                left = mid2 + 1;
            // If the sum of the first third is greater, the desired index is in the first third
            } else if (comparisonResult == 1) {
                right = mid1;
            // Otherwise, the desired index is in the middle third
            } else {
                left = mid1 + 1;
                right = mid2;
        // Left should now be the desired index as the range has been narrowed down to a single element
        return left;
```

```
TypeScript
```

};

C++

public:

class Solution {

int getIndex(ArrayReader& reader) {

while (left < right) {</pre>

```
// Function to find the index of one peak element.
// A peak element is greater than its neighbors.
function getIndex(reader: { length: () => number, compareSub: (start1: number, end1: number, start2: number, end2: number) => number
   let left = 0; // Start of the search range
   let right = reader.length() - 1; // End of the search range
   // Continue search while there is more than one element in the range
   while (left < right) {</pre>
       // Divide the current range into three equal parts
        const part1End = left + Math.floor((right - left) / 3);
        const part2Start = part1End + 1;
        const part2End = left + Math.floor((right - left) / 3 * 2);
        const part3Start = part2End + 1;
       // Compare the sum of elements in the first and second parts
        const comparisonResult = reader.compareSub(left, part1End, part2Start, part2End);
        if (comparisonResult === 0) {
            // Sums are equal, peak must be in the third part
            left = part3Start:
        } else if (comparisonResult === 1) {
            // Sum of first part is greater, peak is in the first part
            right = part1End;
        } else {
            // Sum of second part is greater, peak is in between part2Start and part2End
            left = part2Start;
            right = part2End;
   // Once the range is narrowed down to one element, return it
   // This element is not quaranteed to be a peak, but based on the problem
   // statement it is considered as a correct answer.
   return left;
class Solution:
   def getIndex(self, reader: 'ArrayReader') -> int:
       # Initialize two pointers for binary search
        left, right = 0, reader.length() - 1
       while left < right:</pre>
           # Divide the range into three equal parts
           # and determine their endpoints
           one third = (right - left) // 3
            two_thirds = 2 * one_third
```

When 'left' is equal to 'right', we have found the pivot index return left

Time and Space Complexity

else:

first part end = left + one third

if comparison result == 0:

elif comparison result == 1:

right = first part end

left = third part start

second part end = left + two thirds

second part start = first part end + 1

third_part_start = second_part_end + 1

Compare the sum of the first third with the sum of the second third

If both sums are equal, the pivot index must be in the third part

left, right = second_part_start, second_part_end

If the sum of the first third is greater, the pivot index is in the first part

If the sum of the second third is greater, the pivot index is in the second part

comparison_result = reader.compareSub(left, first_part_end, second_part_start, second_part_end)

The given code uses a ternary search approach to find an index in a simulated array using the ArrayReader API. The while loop repeatedly narrows down the search range by dividing it into thirds and comparing the sums of different segments of the array. **Time Complexity:**

The time complexity is 0(log3(n)), where n is the length of the array. In each iteration of the loop, the size of the current search range is reduced by 1/3. Since we are making two calls to compareSub per iteration, these calls do not significantly increase the

size of the array.

times we can divide the range length by 3 until we reach a range of length 1. **Space Complexity:** The space complexity is 0(1) because no additional space is allocated proportional to the input size. The algorithm only uses a fixed number of variables to store the indices (left, right, t1, t2, and t3) and the comparison result cmp regardless of the

time complexity, and the dominating factor remains the division by three in each step. The complexity is determined by how many