Sorted Search

Learn two different ways to complete the search of a sorted array. We'll go over the brute force method and a more elegant method.

Sorted Search

Instructions

Write a function that accepts a sorted array of integers and a number. Return the index of that number if present. The function should return `-1` for target values not in the array.

Input: Array of Integers, Integer

Output: An integer from `-1` onwards.

(/learn)

Examples:

```
search([1, 3, 6, 13, 17], 13); // -> 3
search([1, 3, 6, 13, 17], 12); // -> -1
```

```
function search(numbers, target) {
    // Your code here
}
```

Solution 1

```
function search(numbers, target) {
    for(let i = 0; i < numbers.length; i++) {
        if(numbers[i] === target) {
            return i;
        }
    }

    return -1;
}
```

This solution is very simple. We go through the array and try to find our target.

Time

Since we're going through the whole array, the time complexity is:

O(n).

Space

Since we store a set number of variables, space complexity is:

O(1).

Binary Search

We can come up with a better solution. Since the array is sorted, we can essentially jump around the array until we find the index we're looking for.

Finding a Word

Imagine looking for a word in a dictionary. Would it be efficient to go through every single word until we find the one we want? No, that would be horribly inefficient.

A better approach would be to open the dictionary halfway. If our word is alphabetically before the words in the middle page, we know our word is in the first half of the book.

We can then flip to ~1/4 of the way through the dictionary. Again, repeating the above process, we can eliminate another half of the remaining pages.

We can repeat the above steps again and again until we find our word. This ensures that even if the dictionary is huge, we can find our word much faster than if we were to go through each word individually.

Scaling

In fact, if we double the size of the dictionary by adding in more words, we would only have to repeat this process one more time. That's not much more work even though the dictionary is much thicker.

Let's turn this approach into code.

Solution 2

```
function binarySearch(numbers, target) {
    let startIndex = 0;
    let endIndex = numbers.length - 1;

    if(target < numbers[startIndex] || target > numbers[endIndex]) {
        return -1;
    }
    
    while(true) {
        if(numbers[startIndex] === target) {
            return startIndex;
        }
        
        if(numbers[endIndex] === target) {
            return endIndex;
        }
        
        if(endIndex - startIndex <= 1) {
            // indicates the number isn't present
            return -1;
        }
        
        const middleIndex = Math.floor((startIndex + endIndex) / 2);
        
        if(target > numbers[middleIndex]) {
            startIndex = middleIndex + 1;
        } else if(target < numbers[middleIndex]) {
            endIndex = middleIndex - 1;
        } else {
            return middleIndex;
        }
```

How it Works

We start by ensuring that the target is within range of the array, returning `false` otherwise.

In the loop, we first check if the target is equal to the values at the beginning and end of our array. Then we check if the target is larger or smaller than the value at the center.

If it's smaller than the value at the center, we know that it's going to be somewhere in the 1st half of the array. We can focus on that.

If it's larger, we know that it can't possibly be in the first half, so we can focus on searching the 2nd half.

Over and over, we cut the amount we have to search by half until we close in on its location. If we find it along the way, we can stop and return its index.

If not found, we'll eventually get to a point where `startIndex` is equal to `endIndex - 1`. Here, we know that the target is in between two values and we can return `-1`.

Time

We're eliminating half of the array in every iteration of our loop, so time complexity is O(log$_2$(n)), which simplifies to:

O(log(n)).

Space

The space complexity is again:

O(1).

Conclusion

Dealing with sorted data allows us to make many optimizations. We know the relationship of an item to the items before it and after it, allowing us to jump around the data and hone in on the value we're looking for.

If data comes in unsorted, sometimes it would be wise to sort it ourselves. If we have a solution with `O(n^2)` time complexity, it's worth it to see if we can sort the data.

Sorting it would be an `O(n * log(n))` process, which is better than `O(n^2)`. Once it's sorted, the next steps are often linear or logarithmic, meaning they are insignificant compared to the sorting itself.

← Back Next →

Rotate Matrix, Advanced

Balanced Brackets

✓ Mark as Completed

Report an Issue