Number Range (medium)

We'll cover the following

Problem Statement

Given an array of numbers sorted in ascending order, find the range of a given number 'key'. The range of the 'key' will be the first and last position of the 'key' in the array.

Write a function to return the range of the 'key'. If the 'key' is not present return [-1, -1].

Example 1:

```
Input: [4, 6, 6, 6, 9], key = 6
Output: [1, 3]
```

Example 2:

```
Input: [1, 3, 8, 10, 15], key = 10
Output: [3, 3]
```

Example 3:

```
Input: [1, 3, 8, 10, 15], key = 12
Output: [-1, -1]
```

Try it yourself

Try solving this question here:

Java | Python3 | JS | C++

```
const find_range = function(arr, key) {
  result = [- 1, -1];
  // TODO: Write your code here
  return result;
};


console.log(find_range([4, 6, 6, 6, 9], 6))
console.log(find_range([1, 3, 8, 10, 15], 10))
console.log(find_range([1, 3, 8, 10, 15], 12))

```

RUN SAVE RESET

Solution

The problem follows the **Binary Search** pattern. Since Binary Search helps us find a number in a sorted array efficiently, we can use a modified version of the Binary Search to find the first and the last position of a number.

We can use a similar approach as discussed in Order-agnostic Binary Search. We will try to search for the 'key' in the given array; if the 'key' is found (i.e. `key == arr[middle]`) we have two options:

1. When trying to find the first position of the 'key', we can update `end = middle - 1` to see if the key is present before `middle`.
2. When trying to find the last position of the 'key', we can update `start = middle + 1` to see if the key is present after `middle`.

In both cases, we will keep track of the last position where we found the 'key'. These positions will be the required range.

Code

Here is what our algorithm will look like:

Java | Python3 | C++ | JS

```
function find_range(arr, key) {
  result = [-1, -1];
  result[0] = binary_search(arr, key, false);
  if (result[0] !== -1) { // no need to search, if 'key' is not present in the input array
```

```
  result[1] = binary_search(arr, key, true);
 }

 return result;
}

// modified Binary Search
function binary_search(arr, key, findMaxIndex) {
 let keyIndex = -1;
 let start = 0;
 let end = arr.length - 1;
 while (start <= end) {
  mid = Math.floor(start + (end - start) / 2);
  if (key < arr[mid]) {
   end = mid - 1;
  } else if (key > arr[mid]) {
   start = mid + 1;
  } else { // key === arr[mid]
   keyIndex = mid;
   if (findMaxIndex) {
    start = mid + 1; // search ahead to find the last index of 'key'
   } else {
    end = mid - 1; // search behind to find the first index of 'key'
   }
```

RUN SAVE RESET

Close

Output 1.801s

```
[ 1, 3 ]
[ 3, 3 ]
[ -1, -1 ]
```

Time complexity

Since, we are reducing the search range by half at every step, this means that the time complexity of our algorithm will be $O(logN)$ where 'N' is the total elements in the given array.

Space complexity

The algorithm runs in constant space $O(1)$.

Back
Next Letter (medium)

Next
Search in a Sorted Infinite Array (medi...

Mark as Completed

Report an Issue Ask a Question