

## Number Range (medium)

### We'll cover the following ^

- Problem Statement
- Try it yourself
- Solution
- Code
  - Time complexity
  - Space complexity

### 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++

```
1 def find_range(arr, key):
2     result = [-1, -1]
3     # TODO: Write your code here
4     return result
5
6 def main():
7     print(find_range([4, 6, 6, 6, 9], 6))
8     print(find_range([1, 3, 8, 10, 15], 10))
9     print(find_range([1, 3, 8, 10, 15], 12))
10
11
12 main()
13
```

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

```
1 def find_range(arr, key):
2     result = [-1, -1]
3     result[0] = binary_search(arr, key, False)
4     if result[0] != -1: # no need to search, if 'key' is not present in the input array
5         result[1] = binary_search(arr, key, True)
6     return result
7
8
9 # modified Binary Search
10 def binary_search(arr, key, findMaxIndex):
11     keyIndex = -1
12     start, end = 0, len(arr) - 1
13     while start <= end:
14         mid = start + (end - start) // 2
15         if key < arr[mid]:
16             end = mid - 1
17         elif key > arr[mid]:
18             start = mid + 1
19         else: # key == arr[mid]
20             keyIndex = mid
21             if findMaxIndex:
22                 start = mid + 1 # search ahead to find the last index of 'key'
23             else:
24                 end = mid - 1 # search behind to find the first index of 'key'
25     return keyIndex
26
27
28
29 def main():
30     print(find_range([4, 6, 6, 6, 9], 6))
31     print(find_range([1, 3, 8, 10, 15], 10))
32     print(find_range([1, 3, 8, 10, 15], 12))
33
34
35 main()
36
```

Run Save Reset

## 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  $O(\log N)$  where 'N' is the total elements in the given array.

## Space complexity #

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

.

Next Letter (medium)

.

Search in a Sorted Infinite Array (medi...

✓ Completed

⚠ Report an Issue    🗉 Ask a Question