Minimum Difference Element (medium)

We'll cover the following

Problem Statement

Given an array of numbers sorted in ascending order, find the element in the array that has the minimum difference with the given 'key'.

Example 1:

```
Input: [4, 6, 10], key = 7
Output: 6
Explanation: The difference between the key '7' and '6' is minimum than any other number in th
e array
```

Example 2:

```
Input: [4, 6, 10], key = 4
Output: 4
```

Example 3:

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

Example 4:

```
Input: [4, 6, 10], key = 17
Output: 10
```

Try it yourself

Try solving this question here:

Java | Python3 | JS | C++

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


console.log(search_min_diff_element([4, 6, 10], 7))
console.log(search_min_diff_element([4, 6, 10], 4))
console.log(search_min_diff_element([1, 3, 8, 10, 15], 12))
console.log(search_min_diff_element([4, 6, 10], 17))

```

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 number that has the minimum difference with the given 'key'.

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 we find the 'key' we will return it as the minimum difference number. If we can't find the 'key', (at the end of the loop) we can find the differences between the 'key' and the numbers pointed out by indices `start` and `end`, as these two numbers will be closest to the 'key'. The number that gives minimum difference will be our required number.

Code

Here is what our algorithm will look like:

Java | Python3 | C++ | JS

```
function search_min_diff_element(arr, key) {
  if (key < arr[0]) {
    return arr[0];
```

```
}
const n = arr.length;
if (key > arr[n - 1]) {
  return arr[n - 1];
}

let start = 0;
let end = n - 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 {
    return arr[mid];
  }
}

// at the end of the while loop, 'start === end+1'
// we are not able to find the element in the given array
// return the element which is closest to the 'key'
if ((arr[start] - key) < (key - arr[end])) {
  return arr[start];
}
```

RUN SAVE RESET

Close

Output 2.000s

```
6
4
10
10
```

Time complexity

Since, we are reducing the search range by half at every step, this means 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 $O(1)$.