The ubiquitous binary search algorithm

Source: ***https://github.com/rusevrosen/codehub.github.io/tree/main/Algorithms***

1. *What is the ubiquitous binary search algorithm?*

Binary search is an efficient algorithm for finding an element within a sorted array. It works by repeatedly dividing the array in half until the element is found or it is clear that the element is not present in the array.

Here's how it works:

First, the algorithm determines the midpoint of the array.

If the element being searched for is less than the value at the midpoint, the algorithm repeats the process with the left half of the array.

If the element being searched for is greater than the value at the midpoint, the algorithm repeats the process with the right half of the array.

If the element being searched for is equal to the value at the midpoint, the search is successful and the algorithm returns the index of the element in the array.

Binary search is an efficient algorithm because it reduces the number of comparisons that need to be made by half with each iteration. This makes it much faster than linear search, which checks every element in the array until the desired element is found. Binary search is often used in computer programs to quickly find a specific value in a large dataset.

2. *Code example.*

```cpp
#include <iostream>

using namespace std;

// Function to perform binary search
int binarySearch(int array[], int size, int key)
{
    int left = 0;
    int right = size - 1;

    while (left <= right)
    {
        int mid = left + (right - left) / 2;

        if (array[mid] == key)
        {
            return mid;
        }
        else if (array[mid] < key)
        {
            left = mid + 1;
        }
        else
        {
            right = mid - 1;
        }
    }

    return -1;
}

int main()
{
    int array[] = {1, 2, 3, 4, 5};
```

```
    int size = sizeof(array) / sizeof(array[0]);
    int key = 4;

    int index = binarySearch(array, size, key);

    if (index != -1)
    {
        cout << "Key found at index: " << index << endl;
    }
    else
    {
        cout << "Key not found in array" << endl;
    }

    return 0;
}
```

```
Output: "Key found at index: 3"
```

This is because the value **4** is located at index **3** in the **array {1, 2, 3, 4, 5}.** The binary search function returns the index of the element if it is found, or **-1** if it is not found. In this case, the function returns **3**, indicating that the key was found at index 3.

3. Code explanation.

The binarySearch function takes in an array, the size of the array, and the key being searched for as arguments. It initializes two variables, left and right, to **0** and size **-** **1**, respectively. These variables represent the indices of the left and right ends of the portion of the array being searched. The function then enters a loop that continues as long as the left is less than or equal to the right.

Inside the loop, the midpoint of the array is calculated by adding left and right and dividing by **2**. Then, the value at the midpoint is compared to the key. If they are equal, the index of the midpoint is returned. If the key is less than the value at the midpoint, the search continues in the left half of the array by setting left to mid **+** **1**. If the key is greater than the value at the midpoint, the search continues in the right half of the array by setting right to mid **-** **1**.

If the loop completes without finding the key, the function returns -1 to indicate that the key was not found in the array.

In the main function, an **`array {1, 2, 3, 4, 5}`** is defined and the key to search for is set to **4**. The binarySearch function is called with these arguments and the result

4. Time complexity.

The time complexity of the binary search algorithm shown in the example above is **`O(log n),`** where n is the size of the array. This means that the time taken by the algorithm grows logarithmically with the size of the array.

In each iteration of the search loop, the size of the portion of the array being searched is halved. This means that the number of comparisons needed to find the key decreases by half with each iteration. As a result, the time complexity of the algorithm is logarithmic, rather than linear as in a simple linear search.

This makes binary search an efficient algorithm for searching through large datasets. It is much faster than linear search, which has a time complexity of **`O(n),`** because it does not need to check every element in the array.

It's important to note that binary search can only be used on sorted arrays. If the array is not sorted, the algorithm will not work correctly. The time complexity of sorting an array is typically **`O(n log n),`** so it is important to take this into consideration when deciding whether to use binary search or another search algorithm.

I wish you happy learning,

Your Rosen Rusev