Binary Search

Binary Search can only use with sorted array

Algorithm

> Binary_Search(a, lower_bound, upper_bound, val) // 'a' is the given array, 'lower_bound' is the index of the first array element, 'upper_bound' is the index of the last array element, 'val' is the value to search

- Step 1: set beg = lower_bound, end = upper_bound, - pos = - 1
- Step 2: repeat steps 3 and 4 while beg <=end
- Step 3: set mid = (beg + end)/2
- Step 4: if a[mid] = val
- set pos = mid
- print pos
- go to step 6
- else if a[mid] > val
- set end = mid - 1
- else
- set beg = mid + 1
- [end of if]
- [end of loop]
- Step 5: if pos = -1
- print "value is not present in the array"
- [end of if]
- Step 6: exit

Code

Function

```
int binarySearch(int a[], int beg, int end, int val)
{
    int mid;
    if(end >= beg)
    {
        mid = (beg + end)/2;
/* if the item to be searched is present at middle */
        if(a[mid] == val)
        {
            return mid+1;
        }
            /* if the item to be searched is smaller than middle, then it
can only be in left subarray */
        else if(a[mid] < val)
        {
            return binarySearch(a, mid+1, end, val);
        }
        /* if the item to be searched is greater than middle, then it can
only be in right subarray */
    else
        {
            return binarySearch(a, beg, mid-1, val);
        }
    }
    return -1;
}
```

Inside main

```
    int a[] = {10, 12, 24, 29, 39, 40, 51, 56, 70}; // given array
    int val = 51; // value to be searched
    int n = sizeof(a) / sizeof(a[0]); // size of array
    int res = binarySearch(a, 0, n-1, val); // Store result
    cout<<"The elements of the array are - ";
    for (int i = 0; i < n; i++)
    cout<<a[i]<<" ";
    cout<<"\nElement to be searched is - "<<val;
    if (res == -1)
    cout<<"\nElement is not present in the array";
    else
    cout<<"\nElement is present at "<<res<<" position of array";
```

Explaination

What Binary Search Does in General

Imagine you have a sorted phone book and you want to find someone's number. You wouldn't start at the first page and flip through one by one, right? You'd open the book roughly in the middle. If the name you're looking for comes before the middle name, you'd focus on the first half of the book. If it comes after, you'd focus on the second half. You'd repeat this process until you find the name or determine it's not in the book.

That's exactly what binary search does, but for a sorted list of numbers (or other comparable items). It's a very efficient way to search, especially for large lists.

Code Explanation

Let's look at the code piece by piece:

1. **Includes and Namespace:**

   ```
   #include <iostream>
   using namespace std;
   ```

   - `#include <iostream>`: This line includes the iostream library, which allows us to perform input and output operations (like printing to the console using `cout`).

  - `using namespace std;`: This line tells the compiler to use the standard namespace, so you don't have to write `std::` before elements like `cout` and `cin`.

2. **The `binarySearch` Function:**

```
int binarySearch(int a[], int beg, int end, int val)
{
    int mid;
    if(end >= beg)
    {
        mid = (beg + end)/2;
        /* if the item to be searched is present at middle */
        if(a[mid] == val)
        {
            return mid+1;
        }
            /* if the item to be searched is smaller than middle, then
it can only be in left subarray */
        else if(a[mid] < val)
        {
            return binarySearch(a, mid+1, end, val);
        }
        /* if the item to be searched is greater than middle, then it
can only be in right subarray */
    else
        {
            return binarySearch(a, beg, mid-1, val);
        }
    }
    return -1;
}
```

  - **Function Signature:**
    - `int binarySearch(int a[], int beg, int end, int val)`:
      - `int`: The function returns an integer. This will be the position (index + 1) of the found value in the array, or -1 if the value isn't found.
      - `int a[]`: The array of integers that we are searching through. It's assumed to be sorted in ascending order.
      - `int beg`: The index of the beginning of the current search range.
      - `int end`: The index of the end of the current search range.
      - `int val`: The value that we are trying to find within the array.
  - **Base Case (When to Stop):**

```
if(end >= beg)
```

This `if` statement checks if the search range is still valid. If `end` is less than `beg`, it means we've narrowed down the search so much that the value can't possibly be in the array (because we've crossed over each other in our narrowing down).

- **Finding the Middle:**

  ```
  mid = (beg + end)/2;
  ```

  Calculates the middle index of the current search range.

- **Checking the Middle Value:**

  ```
  if(a[mid] == val)
  {
      return mid+1;
  }
  ```

  If the value at the middle index (`a[mid]`) is equal to the value we're searching for (`val`), then we've found it! The function returns `mid + 1`. It's `mid + 1` because the problem description wants the *position* which is usually 1-based indexed, not 0-based index.

- **Recursive Calls (Narrowing the Search):**

  ```
  else if(a[mid] < val)
  {
      return binarySearch(a, mid+1, end, val);
  }
  else
  {
      return binarySearch(a, beg, mid-1, val);
  }
  ```

  These `else if` and `else` blocks handle the cases where the value we're searching for is *not* at the middle index. This is the "divide" part of "divide and conquer."

  - If `a[mid] < val` (the middle value is less than the value we're looking for), then the value must be in the *right half* of the current search range. So, we recursively call `binarySearch` again, but this time the `beg` index is `mid + 1` (we're starting the search from the element after the middle), and the `end` remains the same.
  - If `a[mid] > val` (the middle value is greater than the value we're looking for), then the value must be in the *left half* of the current search range. We recursively call `binarySearch` with the `beg` remaining the same, but the `end` index is now `mid - 1` (we're ending the search at the element before the middle).

- **Value Not Found:**

```
return -1;
```

If the `if (end >= beg)` condition becomes false, it means we've exhausted the search range without finding the value. In this case, the function returns `-1` to indicate that the value is not present in the array.

3. **The `main` Function:**

```
int main() {
    int a[] = {10, 12, 24, 29, 39, 40, 51, 56, 70}; // given array
    int val = 51; // value to be searched
    int n = sizeof(a) / sizeof(a[0]); // size of array
    int res = binarySearch(a, 0, n-1, val); // Store result
    cout<<"The elements of the array are - ";
    for (int i = 0; i < n; i++)
    cout<<a[i]<<" ";
    cout<<"\nElement to be searched is - "<<val;
    if (res == -1)
    cout<<"\nElement is not present in the array";
    else
    cout<<"\nElement is present at "<<res<<" position of array";
    return 0;
}
```

- **Initialization:**
  - `int a[] = {10, 12, 24, 29, 39, 40, 51, 56, 70};`: This creates a sorted array of integers. **Binary search *requires* the array to be sorted.**
  - `int val = 51;`: This sets the value that we're searching for.
  - `int n = sizeof(a) / sizeof(a[0]);`: This calculates the number of elements in the array. `sizeof(a)` gives the total size of the array in bytes, and `sizeof(a[0])` gives the size of one element (an `int`) in bytes. Dividing them gives the number of elements.
  - `int res = binarySearch(a, 0, n-1, val);`: This calls the `binarySearch` function to perform the search.
    - `a`: The array.
    - `0`: The starting index of the search (the beginning of the array).
    - `n - 1`: The ending index of the search (the end of the array). Remember that arrays are 0-indexed, so the last element is at index `n - 1`.
    - `val`: The value to search for.
  - The result of `binarySearch` (either the index of the found element or -1) is stored in the `res` variable.
- **Output:**
  - The code then prints the array elements, the value being searched, and a message indicating whether the value was found and, if so, at what position in the array.

In Summary

The code implements the binary search algorithm to find a value within a sorted array. It repeatedly divides the search range in half until the value is found or the search range is empty. This approach provides a significant performance improvement over a linear search (checking each element one by one), especially for large arrays. The function returns the 1-based position of the element if found, and -1 if not found.