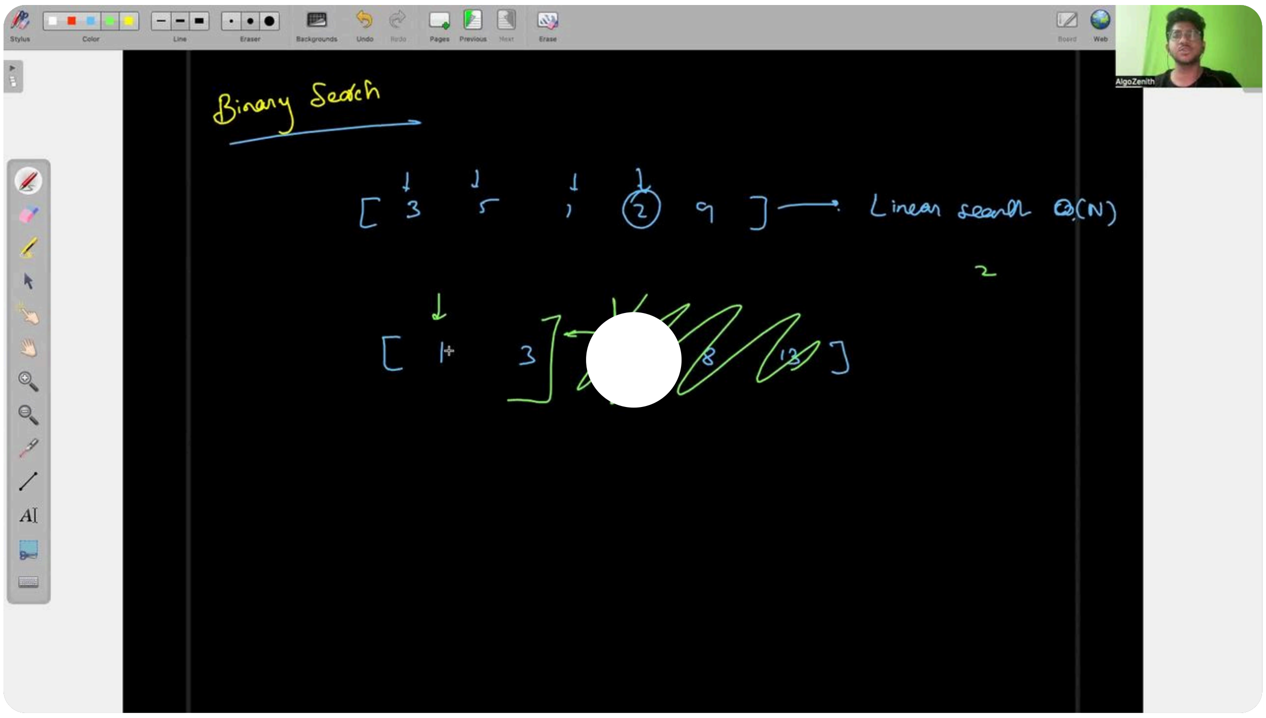
Binary Search Introduction



Exploring Precision: Binary Search in Action

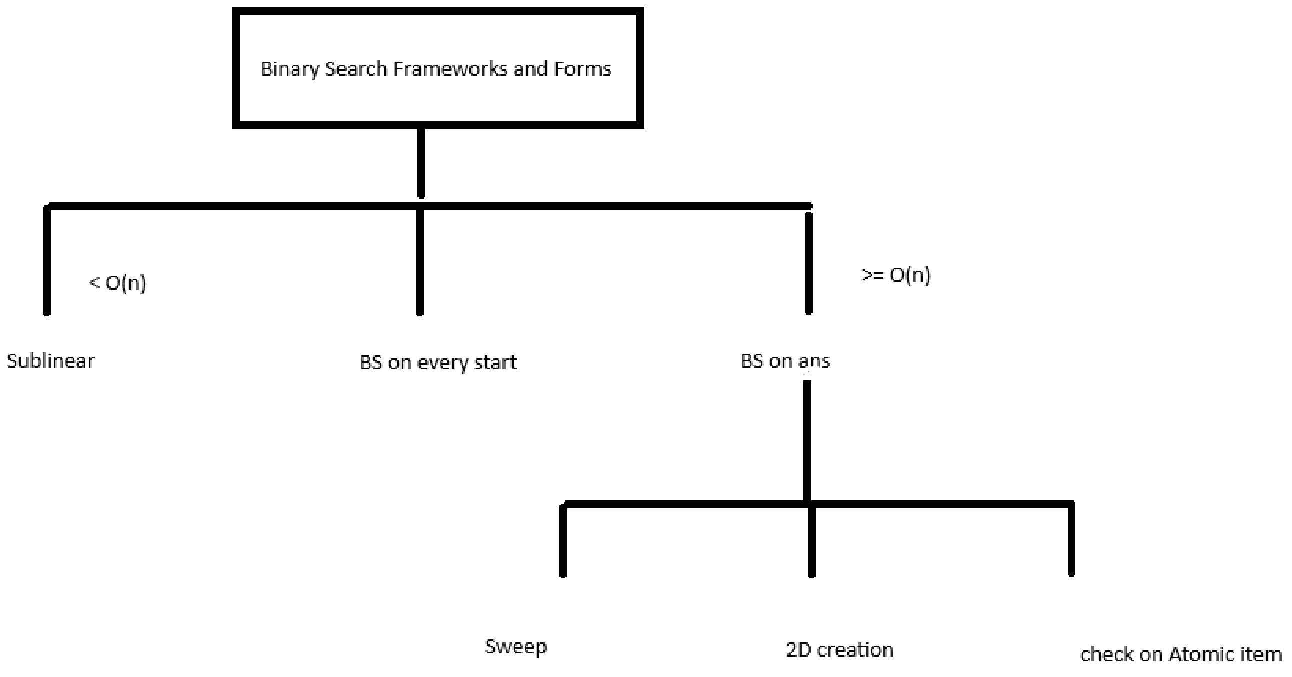
Binary search is a powerful algorithm used to efficiently locate a specific value within a sorted collection of data. This search technique significantly reduces the number of comparisons needed compared to linear search, making it particularly useful for large datasets. Let's delve into binary search through a real-life example and explore a simple implementation in C++.

Real-Life Example: Phone Book

Imagine you have a phone book sorted alphabetically by last name. If you were searching for a person named "John Doe," a binary search would allow you to quickly pinpoint the correct entry. Here's how it works:

1. **Start in the middle of the phone book.**
2. **Compare the last name of the person in the middle with "Doe."**
3. **If the last name matches, you've found the entry. If "Doe" comes before the middle entry alphabetically, repeat the process in the left half; otherwise, search the right half.**
4. **Continue narrowing down the search until you find "John Doe" or determine that he's not in the phone book.**

We will study several frameworks and forms in Binary Search.



Sublinear

Before delving into the intricacies of binary search, it's crucial to grasp three fundamental terms used in our Binary Search Template. These terms lay the groundwork for a clearer understanding of the algorithm:

1. **Search Space:** The search space represents the realm of potential answers to the problem at hand. Denoted by [l, h], where 'l' signifies the low index, and 'h' signifies the high index. The search space progressively narrows down through iterations, leading to the identification of the desired solution.

2. **Mid index:** In each iteration, the algorithm calculates the mid index of the current search space. The formula employed is `mid = l + (h - l) / 2`. This formula is preferred over the direct use of `mid = (l + h) / 2` to mitigate the risk of integer overflow. The mid index serves as a pivotal point, guiding the search towards the optimal solution.

3. **ans:** The 'ans' variable plays a crucial role in storing the best answer obtained thus far during the binary search process. As the algorithm progresses, 'ans' captures and retains the most viable solution, ensuring that the final outcome is both accurate and efficient. Initially, the value can vary problem to problem but generally is -1, meaning no answer found yet.

Example:

You are given an array of size n containing {0,1} where elements are in non-decreasing order. Implement a C++ program to find the index of the first occurrence of the value '1' in the array.

Test Case:

```
n = 9
arr = [0, 0, 1, 1, 1, 1, 1, 1, 1]
```

1. Initial Values:

- `l` is the starting index of the array, initially set to 0.
- `h` is the ending index of the array, initially set to n - 1 (where n is the size of the array).
- `ans` is initialized to -1.

2. Iteration 1:

- `mid` is calculated as the middle index between l and h.
- `arr[mid]` is checked. If it's equal to 1, we update ans to the current mid and move the search to the left half by setting `h = mid - 1`. If it's not equal to 1, we move to the right half by setting `l = mid + 1`. Remember, we will exclude the mid from our search space because we already checked whether it's a better answer or not.

3. Iteration 2:

- The search space is narrowed down based on the result from Iteration 1.
- The process repeats until `l` becomes greater than `h`, indicating the end of the search.

4. Final Result:

- The value of `ans` represents the index of the first occurrence of '1' in the array.

stl

Code:

```cpp
#include <bits/stdc++.h>
using namespace std;

int n;
int arr[100100];

int main() {
    cin >> n;
    for (int i = 0; i < n; i++) {
        cin >> arr[i];
    }

    int lo = 0, hi = n - 1, ans = -1;

    while (lo <= hi) {
        int mid = lo + (hi - lo) / 2;

        if (arr[mid] == 1) {
            ans = mid;
            hi = mid - 1;
        } else {
            lo = mid + 1;
        }
    }

    cout << "Final result: " << ans << endl;
    return 0;
}
```

Time complexity is O(log N), where N is the size of the array. This is because in each iteration, the search space is effectively halved.

Now, we keep the template the same, just add a check function and manipulate it for a given problem and try to convert it into the above solved problem.

Let’s take an example of implementing `lower_bound(x)`.

We know the lower bound of x, which means finding the first index such that the element at that index is ">=x".

Test Case:

```
arr = [1, 5, 6, 8, 8, 10, 11, 11, 12] and x = 11
```

Copy

Here, the value at index 6 is the first element greater than or equal to x (11). Prior to that index, all elements are smaller; hence, they are assigned the value 0. After that index, all elements are greater than or equal to x, so they are assigned the value 1. Now, the task is to find the first index containing the element 1.

Note: We do not convert it into a 0-1 array; we simply assume that this assignment is taking place.

Code:

```
#include <bits/stdc++.h>
using namespace std;

int n;
int arr[100100];
int x;

int check(int mid) {
    if (arr[mid] >= x) {
        return 1;
    } else {
        return 0;
    }
}

int main() {
    cin >> n;
    for (int i = 0; i < n; i++) {
        cin >> arr[i];
    }

    int lo = 0, hi = n - 1, ans = -1;
    while (lo <= hi) {
        int mid = (lo + hi) / 2; // lo+(hi-lo)/2;
```

Copy

```
        if (check(mid) == 1) {
            ans = mid;
            hi = mid - 1;
        } else {
            lo = mid + 1;
        }
    }

    cout << ans << endl;
}
```

Here also, the time complexity is O(log N), where N is the size of the array. This is because in each iteration, the search space is effectively halved. check() function just taking O(1).

Explanation:

- The `check` function takes an index `mid` as an argument.
- It compares the element at index `mid` in the array (`arr[mid]`) with the target value `x`.
- If `arr[mid]` is greater than or equal to `x`, the function returns 1, indicating that the current middle element is a potential candidate for the lower bound.
- If `arr[mid]` is less than
- `x`, the function returns 0, indicating that the lower bound must be in the right half of the current search interval.

Similarly, we manipulate the check function and convert the problems into a 0-1 array problem.

Now, we keep the template the same, just add a check function and manipulate it for a given problem and try to convert it into the above solved problem.

Let's take an example of implementing `lower_bound(x)`.

We know the lower bound of x, which means finding the first index such that the element at that index is ">=x".

Test Case:

```
arr = [1, 5, 6, 8, 8, 10, 11, 11, 12] and x = 11
```

Here, the value at index 6 is the first element greater than or equal to x (11). Prior to that index, all elements are smaller; hence, they are assigned the value 0. After that index, all elements are greater than or equal to x, so they are assigned the value 1. Now, the task is to find the first index containing the element 1.

Note: We do not convert it into a 0-1 array; we simply assume that this assignment is taking place.

Code:

```
#include <bits/stdc++.h>
using namespace std;

int n;
int arr[100100];
int x;

int check(int mid) {
    if (arr[mid] >= x) {
        return 1;
    } else {
        return 0;
    }
}

int main() {
    cin >> n;
    for (int i = 0; i < n; i++) {
        cin >> arr[i];
    }

    int lo = 0, hi = n - 1, ans = -1;
    while (lo <= hi) {
        int mid = (lo + hi) / 2; // lo+(hi-lo)/2;
        if (check(mid) == 1) {
            ans = mid;
            hi = mid - 1;
        } else {
            lo = mid + 1;
        }
    }

    cout << ans << endl;
}
```

Here also, the time complexity is O(log N), where N is the size of the array. This is because in each iteration, the search space is effectively halved. check() function just taking O(1).

Explanation:

- The `check` function takes an index `mid` as an argument.
- It compares the element at index `mid` in the array (`arr[mid]`) with the target value `x`.
- If `arr[mid]` is greater than or equal to `x`, the function returns 1, indicating that the current middle element is a potential candidate for the lower bound.
- If `arr[mid]` is less than
- `x`, the function returns 0, indicating that the lower bound must be in the right half of the current search interval.

Similarly, we manipulate the check function and convert the problems into a 0-1 array problem.

Problem Statement: Given a rotated sorted array with distinct elements, find the index of the minimum element in the array.

Code:

```
#include <bits/stdc++.h>
using namespace std;
#define ll long long
#define endl "\n"

#define IOS                  \
    ios::sync_with_stdio(0); \
    cin.tie(0);              \
    cout.tie(0);

int check(int mid, int a[], int n)
{
    if (a[mid] <= a[n - 1])
    {
        return 1;
    }
    else
    {
        return 0;
    }
}

int main()
{
    IOS
    int t;
    cin >> t;
    while (t--)
    {
        int n;
        cin >> n;
        int a[n];
        for (int i = 0; i < n; i++)
        {
            cin >> a[i];
        }
        int l = 0, h = n - 1, ans = -1;
        while (l <= h)
        {
            int mid = (l + h) / 2;

            if (check(mid, a, n) == 1)
            {
                ans = mid;
                h = mid - 1;
            }
            else
            {
                l = mid + 1;
            }
        }
        cout << ans << endl;
    }
}
```

Algorithm:

1. Define a function `check` which takes in the index `mid`, the array `a[]`, and the size `n` of the array. This function checks if the element at index `mid` is less than or equal to the last element of the array. If true, it returns 1, else 0.
2. In the main function: a. Input the number of test cases `t`. b. For each test case: i. Input the size `n` of the array. ii. Input the elements of the array `a[]`. iii. Initialize variables `l` and `h` as the lower and upper bounds of the array respectively, and `ans` as -1. iv. Use a while loop with the condition `l <= h`: - Calculate `mid` as `(l + h) / 2`. - If `check(mid, a, n) == 1`, update `ans` to `mid` and adjust `h` to `mid - 1`. - Else, adjust `l` to `mid + 1`. v. Output the value of `ans`.

Time Complexity: The time complexity of the binary search algorithm is O(log n), where n is the size of the array. Therefore, the time complexity of this code is also O(log n) for each test case.

Problem Statement: Given a bitonic array A consisting of N integers and an integer Q, in each query, you will be given an integer K. Find the positions of K in A. Integer K exists in A.

Code:

```
#include<bits/stdc++.h>
using namespace std;

int n,q;
vector<int> arr;

bool check(int i){
   if(arr[i]>arr[i-1]) return 1;
   else return 0;
}

void solve(){
   cin>>n>>q;
   arr.resize(n);
   for(int i=0;i<n;i++){
       cin>>arr[i];
   }

   int lo = 1;
   int hi = n-1;
   int peak = 0;
   while(lo<=hi){
       int mid = (lo+hi)/2;
       if(check(mid)){
           peak = mid;
           lo = mid + 1;
       }else{
           hi = mid - 1;
       }
   }

   while(q--){
       int k;
       cin>>k;

       vector<int> final;
       lo = 0;
       hi = peak-1;
       while(lo<=hi){
           int mid = (lo+hi)/2;
           if(arr[mid]==k){
                final.push_back(mid+1);
                break;
            }else if(arr[mid]>k){
                hi=mid-1;
            }else{
                lo=mid+1;
            }
        }

        lo = peak;
        hi = n-1;
        while(lo<=hi){
            int mid = (lo+hi)/2;
            if(arr[mid]==k){
                final.push_back(mid+1);
                break;
            }else if(arr[mid]>k){
                lo=mid+1;
            }else{
                hi=mid-1;
            }
        }

        for(auto v:final){
            cout<<v<<" ";
        }
        cout<<endl;
    }
}

signed main(){
   ios_base::sync_with_stdio(0);cin.tie(0);cout.tie(0);
   int _t;cin>>_t;while(_t--)
   solve();
}
```

Algorithm:

1. Define a function `check` which takes an index `i` and checks if `arr[i]` is greater than `arr[i-1]`. If true, returns 1, else returns 0.

2. In the `solve` function: a. Input the number of queries `q` and the size `n` of the array. b. Input the elements of the array `arr[]`. c. Find the peak index of the bitonic array using binary search. Start with `lo` at index 1 and `hi` at `n-1`. Iterate until `lo <= hi`. Within the loop:

   - Calculate `mid` as `(lo + hi) / 2`.
   - If `check(mid)` is true, update `peak` to `mid` and adjust `lo` to `mid + 1`.
   - Else, adjust `hi` to `mid - 1`. d. For each query: i. Input the value of `k`. ii. Initialize an empty vector `final`. iii. Perform binary search on the left half of the array (0 to peak-1) to find positions of `k`. Add positions to `final`. iv. Perform binary search on the right half of the array (peak to n-1) to find positions of `k`. Add positions to `final`. v. Output the positions stored in `final`.

3. In the `main` function:

   - Input the number of test cases `_t`.
   - Call the `solve` function `_t` times.

Time Complexity:

- Preprocessing to find the peak of the bitonic array: O(log n)
- Binary search for each query: O(log n)
- Therefore, the overall time complexity for each test case is O(q * log n), where q is the number of queries and n is the size of the array.

Mark as Read

Write a comment

B I U

• Before you post...

I Understand

- **No Cheating or Violation of Honor Code** : Users are usually prohibited from discussing or sharing solutions for active contests. This is to prevent cheating and to maintain the integrity of the platform.
- **Avoid Posting Full Solutions** : Users are encouraged not to post entire solutions to problems, especially for active contests. Instead, it's common to share hints, tips, or discuss general approaches without giving away the entire solution.
- **Be Respectful** : Users are expected to be respectful and considerate of others. Avoid using offensive language, making personal attacks, or engaging in any form of harassment.
- **Stay On Topic** : Discussions should stay relevant to the problem at hand. Off-topic discussions or spamming are generally discouraged.
- **No Advertising or Self-Promotion** : Users should not use the platform for advertising or self-promotion. This includes promoting external services, products, or personal projects.

All Comments