Monday, 12 September 2022 8:25 PM

Searching

Q. Given an array of integers [3], check if 7 is present or not. If so, at 7?

→ go to each & every element of the array & check if it is equal to 7

Linear search

Problem: Input (arr, n, x) &
for i = 0 ⟶ n - 1
if (arr[i] == x)
return true;
return false;

→ O(n)

Given: array has some ordering

→ sorting order

Sorted array

→ O(n)

Binary Search

Q

Input: sorted array
Find: 7
Take: middle element
Use: mid value

Given: sorted integer array (ascending)
[1 3 5 6 8 9 10 12]

0 1 2 3 4 5 6 7

mid = (l + r) / 2 | mid = l + (r - l) / 2

mid = 3 | mid = 3

l = 0 | r = 7

l = 4 | r = 7

mid = 5

l = 5 | r = 7

mid = 6

l = 6 | r = 7

mid = 7

l = 7 | r = 7

x = 10

l = 7 | r = 6

→ l > r

mid = (l + r) / 2 = 13 / 2 = 6.5 = 6

(A ≤ x)

if A[mid] == x) {
return mid;
}
else if (A[mid] < x) {
l = mid + 1;
}
else {
r = mid - 1;
}

Pseudocode (Iterative Implementation)

l = 0
r = arr.length - 1
while (l ≤ r) {
int mid = (l + r) / 2 | mid = l + (r - l) / 2
if (arr[mid] == x) {
return mid;
}
if (arr[mid] < x) {
l = mid + 1;
}
else {
r = mid - 1;
}
}
return -1

Time Complexity

T(n) = T(n/2)

T(n) = T(n/2) + c ⟶ (1)

T(n/2) = T(n/4) + c ⟶ (2)

T(n/4) = T(n/8) + c ⟶ (3)

T(n/8) = T(n/16) + c ⟶ (4)

T(n) = T(n/2) + c ⟶ (1)

T(n/2) = T(n/4) + c ⟶ (2)

T(n) = T(n/4) + 2c ⟶ (3)

T(n/4) = T(n/8) + c ⟶ (4)

T(n) = T(n/8) + 3c ⟶ (5)

T(n) = T(n/16) + 4c ⟶ (6)

Generalize, as $\frac{n}{2^k}$ diminishes, we reach only 1 element.

$T\left(\frac{n}{2^k}\right) = T(1)$

$\frac{n}{2^k} = 1$

$T(n) = T\left(\frac{n}{2^k}\right) + kc$

Taking log on both sides,

$\log_2 n = k$

$T(n) = T\left(\frac{n}{2^{\log_2 n}}\right) + kc$ → $\frac{n}{2^{\log_2 n}} = 1$

$T(n) = T\left(\frac{n}{n}\right) + \log_2 n \cdot c$

$T(n) = T(1) + \log_2 n \cdot c$

$T(n) = \log_2 n$

Time complexity → $O(\log_2 n)$
Space Complexity → $O(1)$

→ Recursive Implementation

binary search (int[] A, int low, int high, int x){
if (low < high) return -1;
int mid = (low + high) / 2;
if (x == A[mid]) return mid;
else if (x < A[mid]
return binary search(A, low, mid-1, x);
else
return binary search(A, mid+1, high, x);
}

TC → $O(\log_2 n)$
SC → $O(\log_2 n)$

Conditions for applying binary search

→ Array / list should be monotonic function.

Monotonicity

The monotonicity of a function refers to a function is increasing or decreasing

Monotonic function

→ functions is increasing or the entire domain or decreasing or the entire domain.

sorted (asc) | desc

Non-monotonic functions

A non-monotonic function is a function that is increasing or decreasing in different intervals of the domain.

Q. Given a sorted array, find the first value > x

2 3 5 6 8 10 12

x = 5

ans = 6

mid = (l + r) / 2

ans = mid

l = mid + 1

Q. 1 2 3 4 5 6 7 8 9 10

x = 7

l = 0
r = n - 1
ans = -1
while (l ≤ r) {
int mid = l + (r - l) / 2;
if (a[mid] > x) {
ans = mid;
r = mid - 1;
}
else
l = mid + 1;
}
if (ans == -1) {
print ("not found");
}
else return ans;

Q. Given a sorted array, find the first value ≥ x

2 3 5 6 8 10 12

x = 11

ans = 12

l = 0
r = n - 1
ans = -1
while (l ≤ r)

if (a[mid] ≥ x)
ans = mid;
r = mid - 1;

else
l = mid + 1;

return ans;

→ Upper bound and lower bound of a binary search

→ bound is different positions where the value of just the bounded continued crossing the mentioned

Eg: 1 2 3 3 3 5 6 9 9

lb → 3
ub → 9

Eg: 1 2 3 3 5 6 9 9

lb → 3
ub → 5

Eg: 1 2 3 3 5 6 9 9

lb → 6
ub → 9

Frequency of element

→ (upper bound - lower bound)

Search

→ primitive data types

① Arrays: binary search(int arr, key)

→ if element present → index
if element ∉ → (-(insertion) - 1)

② Collections: binary search(),

point at which you can insert element to keep array sorted