Date:

Data Structures

- Array → Data Structure
  - ↳ Way to structure Data or Organize Data
- → Algorithms → Divide & Conquer method
  - ↳ Recursion Based.

Types :-

- Array ---> Linked (Spotify)
- Linked List → Insertion & Deletion
- Hashing → Mapping of key value Pair
- Graph → Networking (BFS & DFS)
- Tree → Non-linear (Networking)
- Stack → LIFO
- Queue → FIFO
  - ↳ First In First Out.
- ~~Free~~
- Heap → Maxima & Minima
  - ↳ k-closest points from origin

Searching Operation → Array → Binary Search

- ↳ Gives random Access
- ↳ call any element using index

Eg: LinkedIn → Networking

Date:

1) Linear
2) Non-Linear

Array is Linear DS

↳ Collection of items in a 'contiguous' manner
memory → RAM

Eg: Student Marks (100)

1000	1004	1008	1012	1016				
85	75	70	65	98	99	50	33	47
0	1	2	3	4	5	6	7	8

↳ Index

$n = 9$

Last index = $(n-1)$ always

Storing elements of Integer data

int → 4 bytes

Base Address → some address where first element is stored at. say 1000

Address of 9th index

1000 * (9 - 0) * 4 = 1036

→ Base Address + (i - Lower Bound) * Size of each element

Date:

Memory is stored as Hexadecimal value
↳ (0-9, A-F)
16 Digits

How to access any element in your Array?

array[7] => 33
↳ Random Access

O(1) → Constant time complexity

Applications :- Searching

n=9

[20, 45, 27, 47, 55, 67, 75, 88, 90]
0 1 2 3 4 5 6 7 8

x = 67 → our target element.

Linear Search

```
for i in range(0, n):
    if arr[i] == x:
        return i
return -1
```

T(n) = O(n)
↳ Worst case.

Date:

Best case → O(1)

↳ Element present near to the first index

Average Case :- worst Best

$$\frac{1}{2} \cdot O(n) + \frac{1}{2} O(1)$$

$$\Rightarrow O(n)$$

↳ Bigger Parts here

↳ Write code for linear search.

↳ Merge Sort → O(n) since we are using another array to store

↳ When you completly get extra/new array

↳ It is extra space.

⇒ Binary Search :- Applicable only for Sorted Array

Either it should be in Ascending or Descending Order.

↳ Lesser Time complexity than linear search.

Date:

0	1	2	3	4	5	6	7
20	30	40	50	60	70	80	90

Eg:- [20 30 40 50 60 70 80 90]

Target $x = 80$

Binary => There are 2 parts to Search for an element.

↳ 1) It will try to find out middle index

=>1) $mid = \frac{0+7}{2} = 3$ (Lower Bound)

2) arr(mid) == x:
return mid

3) arr(mid) < x:
left = mid + 1

4) else
arr(mid) > x
right = mid - 1
return -1

binary(arr, left, right):
while left < right:
->

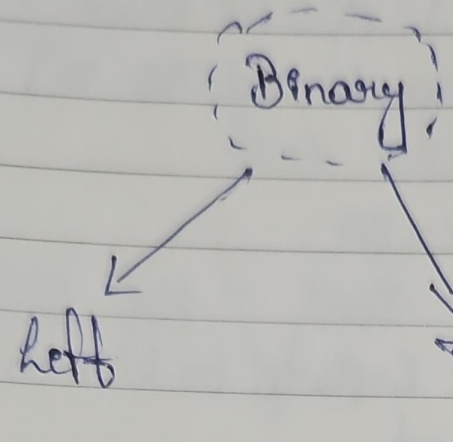
↳ Recursion is used:

binarySearch(arr, mid+1, right)

or

binarySearch(arr, left, mid-1)

Date:



↳ Your value will be either on right or left.

Eg:- binarySearch (arr, 0, 4)

↓

binarySearch (arr, 3, 4)

mid = 3

result → 3

[20, 30, 40, 50, 70]

0 1 2 3 4

x = 50

Always use mid = $\frac{(i+j)}{2}$.

i → start index , j → ending index.

Date:

Preferred method: ✗

↳ mid = i + (j - i) / 2

i -> start index

j -> end index

It can handle large numbers also perfectly fine without throwing minute issues.

Binary Search Algorithm Time Complexity:

```
binarySearch(arr, i, j, x):
    while (i <= j):
        mid = i + (j-i)/2 -> Constant
    if (arr(mid) == x):
        return mid -> Constant
    else if arr(mid) < x:
T(n/2) <- binarySearch(arr, mid+1, j, x)
    else:
T(n/2) <- binarySearch(arr, i, mid-1, x)
    return -1
```

$$T(n) = T\left(\frac{n}{2}\right) + C$$

Date:

Solving Recurrence Relation using Masters Theorem:-

$$T(n) = T(n/2) + c$$

$a = 1 \quad k = 0$

$b = 2 \quad p = 0$

↳ $\log_b a = \log_2 1 = 0$

It belongs to case 2:

$\Rightarrow \log_b a = k$

↳ $p > -1 \Rightarrow O(n^k \log^{p+1} n)$

$$\boxed{O(\log n)}$$

If it is having 2 occurances of search then first index value is returned