Introduction Arrays in Context of DSA

Array is linear data structure that holds element at continuous memory locations.
Accessing any element using its index
arr[7] complexity is O(1)

Big O notation:
it is a mathematical concept used in computer science to describe time complexity or space complexity of an algorithm in terms of how it grows in the size of n.
We ca understand how an algorithm performs when the data is very large.

key concepts:
Time Complexity: How the runtime of an algorithm increase as input grows.
Space Complexity: How memory usage increases with input size.
Worst-case scenario: Big O notation is usually describes worst case performance.

Big O notations:

Big O	Name	Example
O(1)	Constant time	Accessing an array element by its index, sum of natural numbers using sum formula
O(log n)	Logarithmic time	Binary Search
O(n)	Linear time	linear search, (loop through an array)
O(n log n)	Linearithmic time	Merge Sort and quicksort
$O(n^2)$	Quadratic time	Nested Loops (bubble sort)
$O(2^n)$	Exponential time	Recursive (fibonacci)
O(n!)	Factorial time	solving travelling salesman brute-force

Understanding Complexity of Array operations

```
const arr=[1,2,3,4,5,6];
console.log(arr[5]); //access using index O(1)
arr[2]=9; //update using index O(1)
//remove element from last index
console.log(arr.pop()); //complexity O(1)
//add at last
arr.push(10); //complexity O(1)
//insert or remove from first index
arr.unshift(15); //complexity O(n)
arr.shift(); //complexity O(n)
```

Array Traversal

```
const arr=[1,2,3,4,5,6];

//Array Traversal
for (let i = 0; i < arr.length; i++)
console.log(arr[i]);
//foreach
for(let i of arr)
console.log(i);
```

Common Algorithms

```
const arr=[1,2,3,4,5,6];

// find Min and Max
let max= Math.max(...arr);
let min= Math.min(...arr);

//Reverse an Arrray
let reversed = arr.reverse();

//Linear Search
function linearSearch(arr,target){
for(let i=0;i<arr.length;i++){
if(arr[i]===target)
return i;
}
return -1;
```

```
}
```

let's understand Some more..

```
function binarySearch(arr, target) {
arr.sort((a,b)=>a-b);
let left = 0, right = arr.length - 1;
while (left <= right) {
let mid = Math.floor((left + right) / 2);
if (arr[mid] === target)
return mid;
else if (arr[mid] < target)
left = mid + 1;
else
right = mid - 1
}
return -1;
}
```

Sorting Algorithms

Bubble Sort algorithms works in pass
if you have n number of elements in your array then it runs for n-1 pass.

so, in each pass it will put the last element in its correct position.
when you compare the numbers if the number is greater that the next number then we need swap log we can use below swapping logics.
// using temp variable

```
//swap 2 numbers logic

let temp=arr[i];
arr[i]=arr[i+1];
arr[i+1]=temp
//another way to swap
[arr[i],arr[i+1]]=[arr[i+1],arr[i]]
```

Simple bubble Sort logic:

```
function bubblesort(arr){
for(let pass=1;pass<arr.length;pass++){
swapped=false;
```

```
for(let i=0;i<arr.length-pass;i++){
if(arr[i]>arr[i+1]){
[arr[i],arr[i+1]]=[arr[i+1],arr[i]]
}
}
}
}
const array= [1,2,3,7,5,6];
bubblesort(array);
console.log('Result',array)
```

Efficient Bubble Sorting Logic Incase if the Array is already Sorted no need to iterate each pass

```
function bubblesort(arr){
let swapped;
for(let pass=1;pass<arr.length;pass++){
swapped=false;
for(let i=0;i<arr.length-pass;i++){
if(arr[i]>arr[i+1]){
[arr[i],arr[i+1]]=[arr[i+1],arr[i]]
swapped=true;
}
}
if(!swapped) break;
}
}
const array= [1,2,3,7,5,6];
bubblesort(array);
console.log('Result',array)
```

Time Complexity:

Best Case O(n)

when the array is already sorted. it detects no swap in first pass and exit early

Average Case: $O(n^2)$

still it compare every pair till the mid way

Worst Case: $O(n^2)$

when the array is in reverse order and each element has to bubble to reach to its place.

Space Complexity:

there is no extra space used complexity O(1)