**Time complexity of:**

function print(){

console.log("Hello World")

}

is O(1) because it has only one console.log() and doesn’t contain any loops or depends on any input

function sumArray(arr) {

let sum = 0;

for (let i = 0; i < arr.length; i++) {

sum += arr[i];

}

return sum;

}

is O(1) for initialization i.e., sum = 0,

O(n) for loop which runs n times,

O(1) for return statement,

So, total time complexity is O(1) + O(n) + O(1) = O(n)

function findX(arr) {

let x = [];

for (let i = 0; i < arr.length; i++) {

for (let j = 0; j < arr.length; j++) {

if (arr[i] + arr[j] === 10) {

x.push([arr[i], arr[j]]);

}

}

}

return x;

}

is O(n) for initialization x = [],

O(n) × O(n) for nested loops,

O(1) for inner block of code,

O(1) for return statement

So, the total time complexity is O(1) + O(n2) + O(1) + O(1)

function getFirstTwoElements(arr) {

if (arr.length < 2) {

return null;

}

const first = arr[0];

const second = arr[1];

return [first, second];

}

is O(1) for conditional statement,

O(1) for each accessing arr[0] and arr[1],

O(1) for return statement,

So, the time complexity is O(1) + O(1) + O(1) + O(1) = O(1)

function processTwoArrays(arr1, arr2) {

let sum1 = 0;

for (const item of arr1) {

sum1 += item;

}

let sum2 = 0;

for (const item of arr2) {

sum2 += item;

}

return sum1 + sum2;

}

is O(1) for initialization, return statement,

O(n) for first loop,

O(m) for second loop,

So, the time complexity is O(n) + O(m) + O(1) = O(n + m)

function countF(n) {

let count = 0;

for (let i = 1; i < n; i = i \* 2) {

count++;

}

return count;

}

is O(1) for initialization and return statement,

the loop starts from 1 and runs until n, with the iteration of i\*2 (i.e., 1, 2, 4, 8, ...),

So, the time complexity is O(log n) + O(1) = O(log n)

**Worst, average and best case of:**

function findElement(sortedArr, target) {

for (let i = 0; i < sortedArr.length; i++) {

if (sortedArr[i] === target) {

return i;

}

}

return -1;

}

**Best case:** If the target is at first position, the loop runs 1 time, i.e., O(1)

**Average case:** If the target is at middle, the loop runs about n/2 times, i.e., O(n/2)

**Worst case:** If the target is at end or not available, the loop runs whole (n times), i.e., O(n)

function recursiveSum(n) {

if (n <= 0) {

return 0;

}

return n + recursiveSum(n - 1);

}

**Best case:** If n<=0, i.e., O(1)

**Average case:** If n is comparatively small, i.e., O(n)

**Worst case:** If n is comparatively large, i.e., O(n)

function dFunction(arr) {

const seen = {};

for (let i = 0; i < arr.length; i++) {

if (seen[arr[i]]) {

return true;

}

seen[arr[i]] = true;

}

return false;

}

**Best case:** If duplicates are first two elements, i.e., O(1)

**Average case:** If duplicates are not found early but found somewhere in middle,i.e., O(n)

**Worst case:** If duplicates are at the very last or not available, i.e., O(n)

function repeatLog(arr) {

for (let i = 0; i < arr.length; i++) {

let repetitions = arr[i];

for (let j = 0; j < repetitions; j++) {

console.log('hello');

}

}

}

**Best case:** If all arr[i] = 0, inner loop never runs, i.e., O(n)

**Average case:** If average arr[i] = k, then inner loop runs k times for each element, i.e., O(n×k)

**Worst case:** If arr[i] = m (maximum value), i.e., O(n×m)