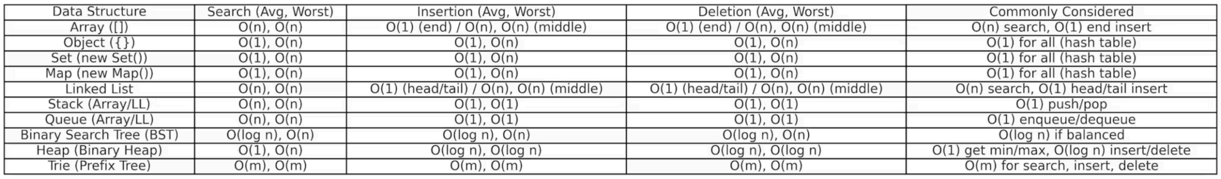
bigO



Data Structure	Search (Avg, Worst)	Insertion (Avg, Worst)	Deletion (Avg, Worst)	Commonly Considered
Array ([])	O(n), O(n)	O(1) (end) / O(n), O(n) (middle)	O(1) (end) / O(n), O(n) (middle)	O(n) search, O(1) end insert
Object ({})	O(1), O(n)	O(1), O(n)	O(1), O(n)	O(1) for all (hash table)
Set (new Set())	O(1), O(n)	O(1), O(n)	O(1), O(n)	O(1) for all (hash table)
Map (new Map())	O(1), O(n)	O(1), O(n)	O(1), O(n)	O(1) for all (hash table)
Linked List	O(n), O(n)	O(1) (head/tail) / O(n), O(n) (middle)	O(1) (head/tail) / O(n), O(n) (middle)	O(n) search, O(1) head/tail insert
Stack (Array/LL)	O(n), O(n)	O(1), O(1)	O(1), O(1)	O(1) push/pop
Queue (Array/LL)	O(n), O(n)	O(1), O(1)	O(1), O(1)	O(1) enqueue/dequeue
Binary Search Tree (BST)	O(log n), O(n)	O(log n), O(n)	O(log n), O(n)	O(log n) if balanced
Heap (Binary Heap)	O(1), O(n)	O(log n), O(log n)	O(log n), O(log n)	O(1) get min/max, O(log n) insert/delete
Trie (Prefix Tree)	O(m), O(m)	O(m), O(m)	O(m), O(m)	O(m) for search, insert, delete

Notes for Quick Revision:

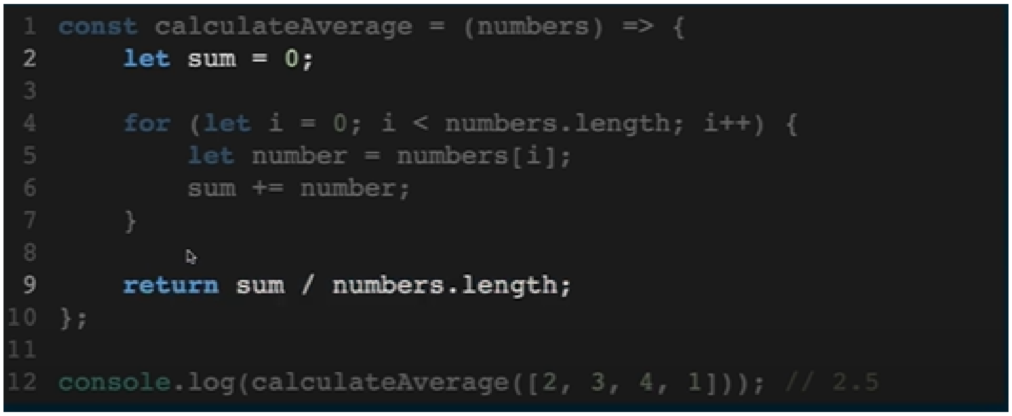
✅ **Hash Tables (`Object`, `Set`, `Map`)** → O(1) average, O(n) worst (rare)

✅ **Arrays & Linked Lists** → O(n) search, O(1) insertion at ends

✅ **Stack & Queue** → O(1) push/pop/enqueue/dequeue

✅ **Trees (`BST`, `Heap`)** → O(log n) for balanced trees, O(n) worst case

✅ **Trie** → O(m) where m = word length



```
1  const calculateAverage = (numbers) => {
2      let sum = 0;
3
4      for (let i = 0; i < numbers.length; i++) {
5          let number = numbers[i];
6          sum += number;
7      }
8
9      return sum / numbers.length;
10 };
11
12 console.log(calculateAverage([2, 3, 4, 1])); // 2.5
```

line 2 and line 9 will run contant no. of time so we will igonre it,

for loop mainly contribute for time complexity here, which include 5 steps:

i = 0, which we ignore as it will happen only once

i< number.length , execute n time

i++, execute n times

let number = number[i]. execute n times

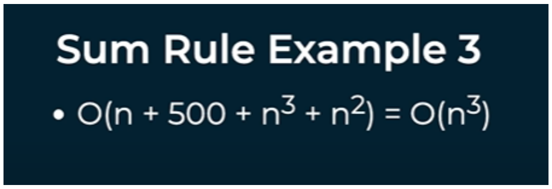
sum+=numbers, execute n times

total n*4 steps, but we ignore any multiplication in calcuting bigO (**PRODUCT RULE**)

bigO, where n is the lenght of input array

- **SUM RULE:**

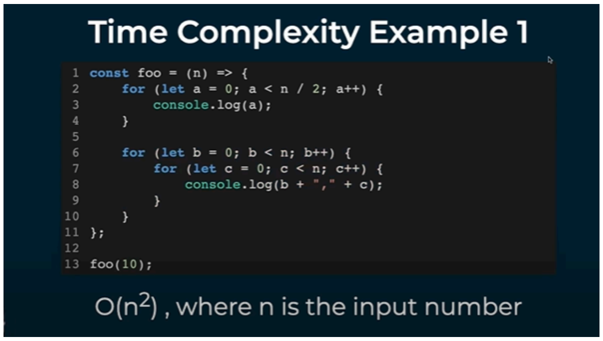
  If the bigO is the sum of multiple term , only keep the largest term, drop the rest.



Sum Rule Example 3

- $O(n + 500 + n^3 + n^2) = O(n^3)$

==



Time Complexity Example 1

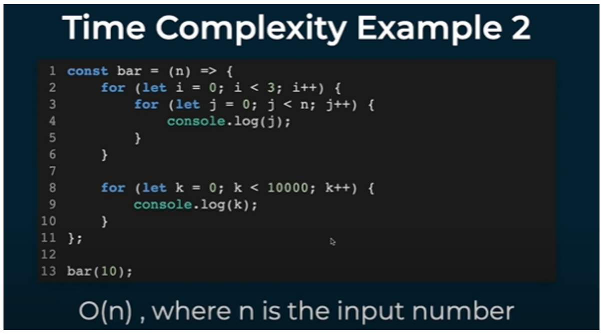
```
const foo = (n) => {
    for (let a = 0; a < n / 2; a++) {
        console.log(a);
    }

    for (let b = 0; b < n; b++) {
        for (let c = 0; c < n; c++) {
            console.log(b + "," + c);
        }
    }
};

foo(10);
```

$O(n^2)$, where n is the input number

n/2 = n

n*n = n*2

n+n*2 = n*2



Time Complexity Example 2

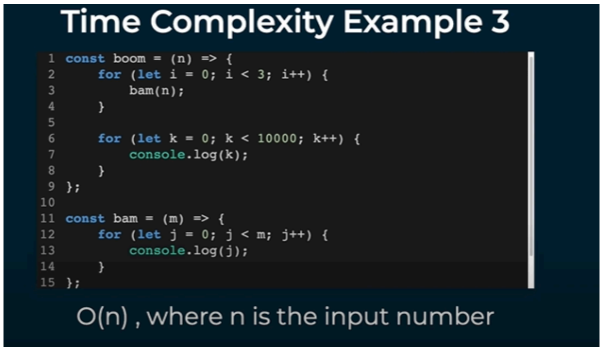
```
const bar = (n) => {
    for (let i = 0; i < 3; i++) {
        for (let j = 0; j < n; j++) {
            console.log(j);
        }
    }

    for (let k = 0; k < 10000; k++) {
        console.log(k);
    }
};

bar(10);
```

O(n) , where n is the input number

3n = n

10000

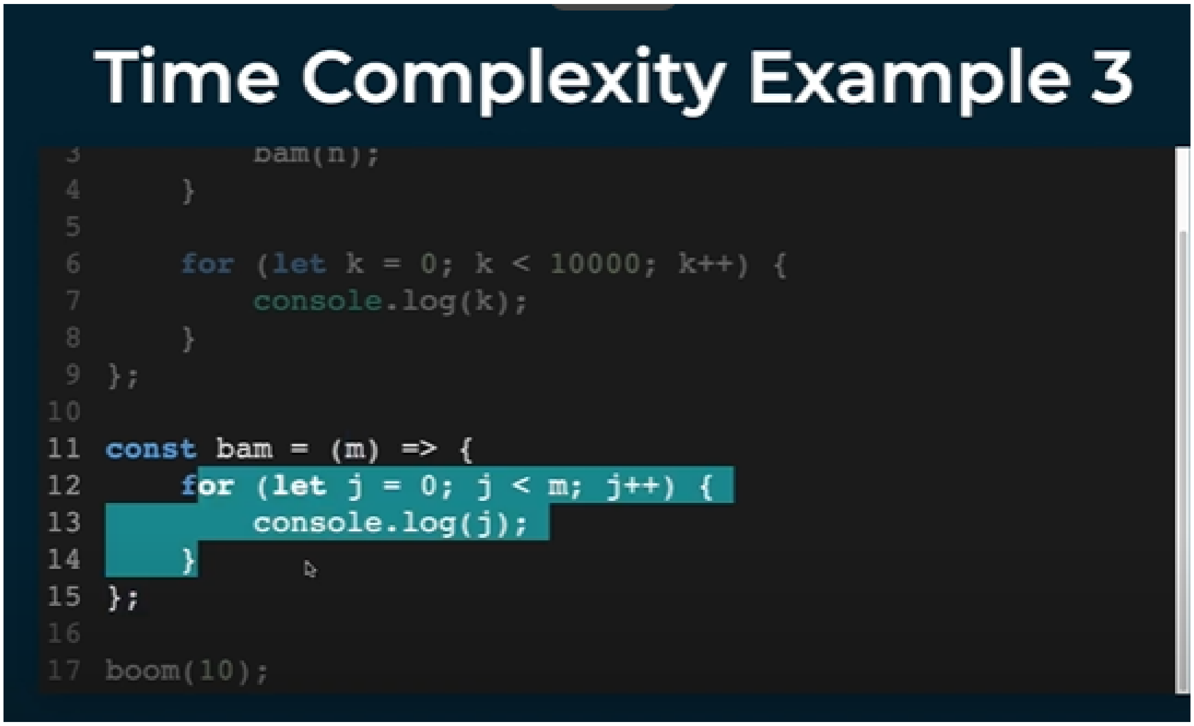
n + 10000 = n



Time Complexity Example 3

```
const boom = (n) => {
    for (let i = 0; i < 3; i++) {
        bam(n);
    }

    for (let k = 0; k < 10000; k++) {
        console.log(k);
    }
};

const bam = (m) => {
    for (let j = 0; j < m; j++) {
        console.log(j);
    }
};
```

O(n) , where n is the input number



Time Complexity Example 3

```
        bam(n);
    }

    for (let k = 0; k < 10000; k++) {
        console.log(k);
    }
};

const bam = (m) => {
    for (let j = 0; j < m; j++) {
        console.log(j);
    }
};

boom(10);
```

line 2 = 3

line 3 = n

2nd for loop = 10000

3n + 10000 = O(n)

==