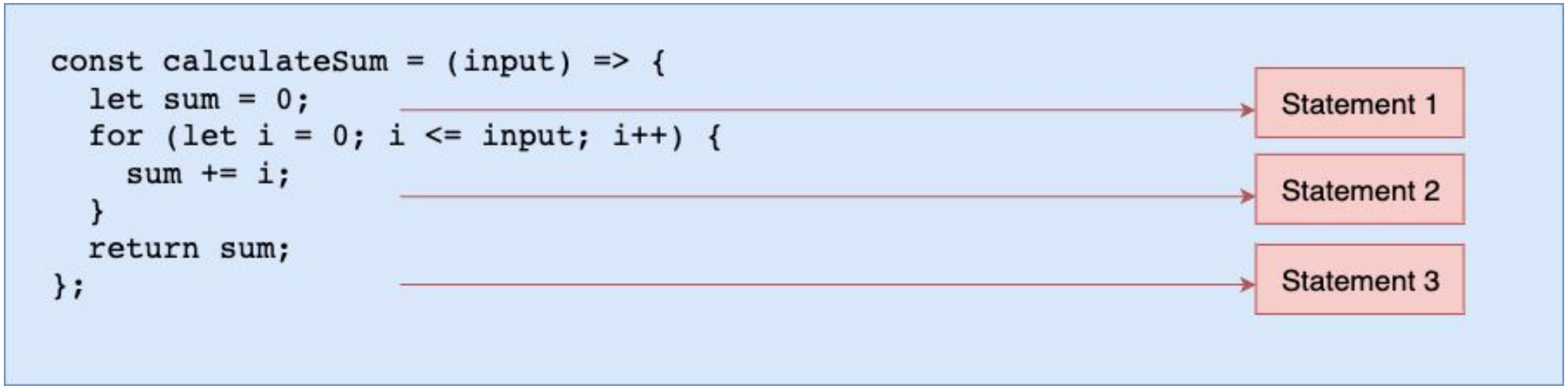
Time Complexity of **LINEAR SEARCH**

- Suppose you have an algorithm that computes the sum of numbers based on your input.
- If your input is 4, it will add 1+2+3+4 to output 10;
- if your input is 5, it will output 15 (meaning 1+2+3+4+5).

```
Fun calculateSum(input)
Sum=0
For I = 0 to input
Sum=sum+I
Return sum
```



```
const calculateSum = (input) => {
  let sum = 0;
  for (let i = 0; i <= input; i++) {
    sum += i;
  }
  return sum;
};
```

Explanation of algorithm

- Looking at the image above, we only have three statements.
- because there is a loop, the second statement will be executed based on the input size, so if the input is **four**, the second statement (statement 2) will be executed four times, meaning the entire algorithm will run **six (4 + 2)** times.

In simple words, the algorithm will run

input + 2 times,

where input can be any number.

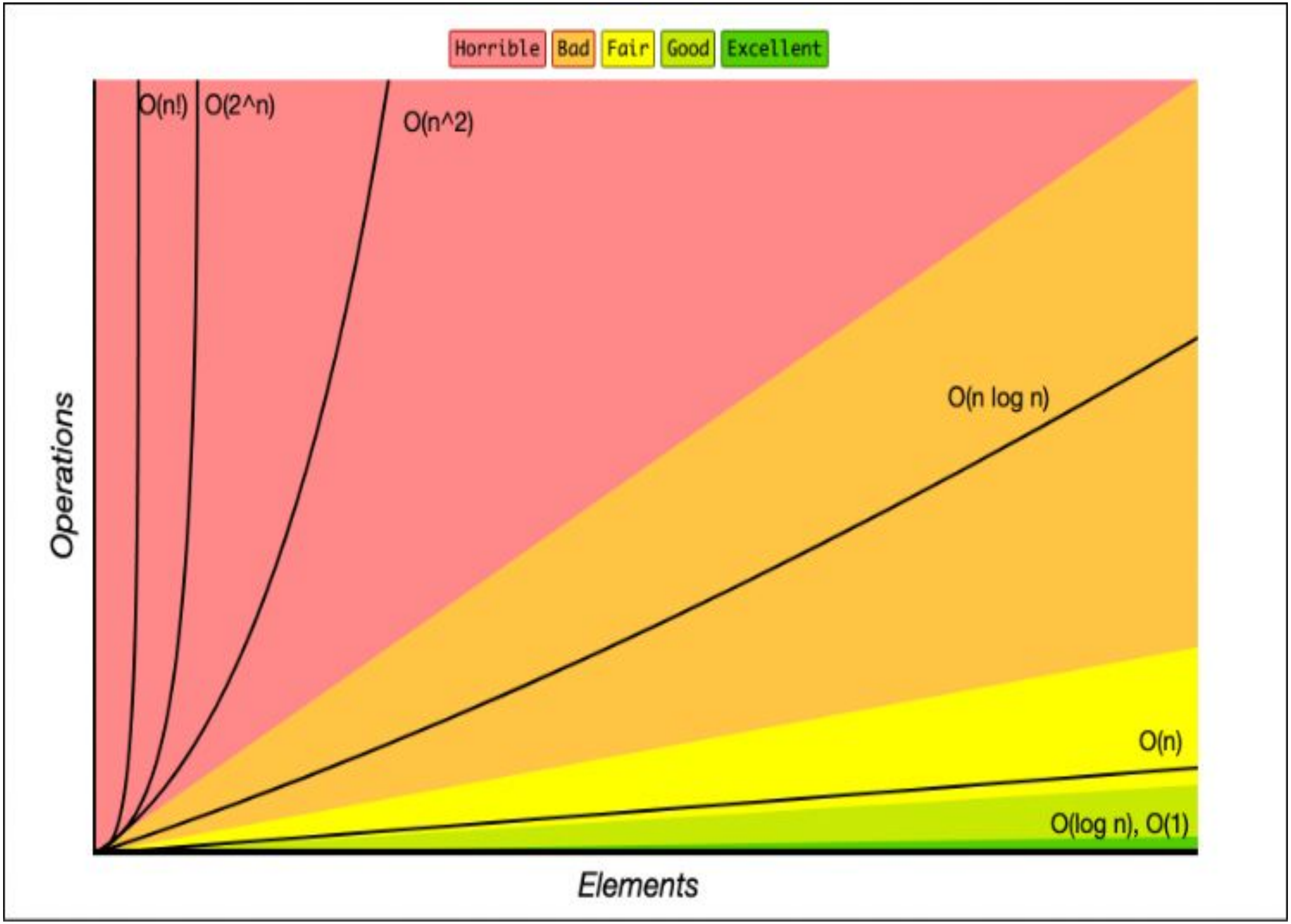
This shows that **it's expressed in terms of the input. In other words, it is a function of the input size**.

- In Big O, there are six major types of complexities (time and space):
- Constant: O(1)
- Linear time: O(n)
- Logarithmic time: O(n log n)
- Quadratic time: O(n^2)
- Exponential time: O(2^n)
- Factorial time: O(n!)

Before we look at examples for each time complexity, let's understand the Big O time complexity chart.

Big O Complexity Chart

- The Big O chart, also known as the Big O graph, is an asymptotic notation used to express the complexity of an algorithm or its performance as a function of input size.
- This helps programmers identify and fully understand the worst-case scenario and the execution time or memory required by an algorithm.


Horrible
Bad
Fair
Good
Excellent
O(n!)
O(2^n)
O(n^2)
O(n log n)
O(n)
O(log n), O(1)
Operations
Elements

- The Big O chart above shows that O(1), which stands for constant time complexity, is the best. This implies that your algorithm processes only one statement without any iteration.
- Then there's O(log n), which is good, and others like it, as shown below:

- **O(1)** - Excellent/Best
- **O(log n)** - Good
- **O(n)** - Fair
- **O(n log n)** - Bad
- **O(n^2)**, **O(2^n)** and **O(n!)** - Horrible/Worst

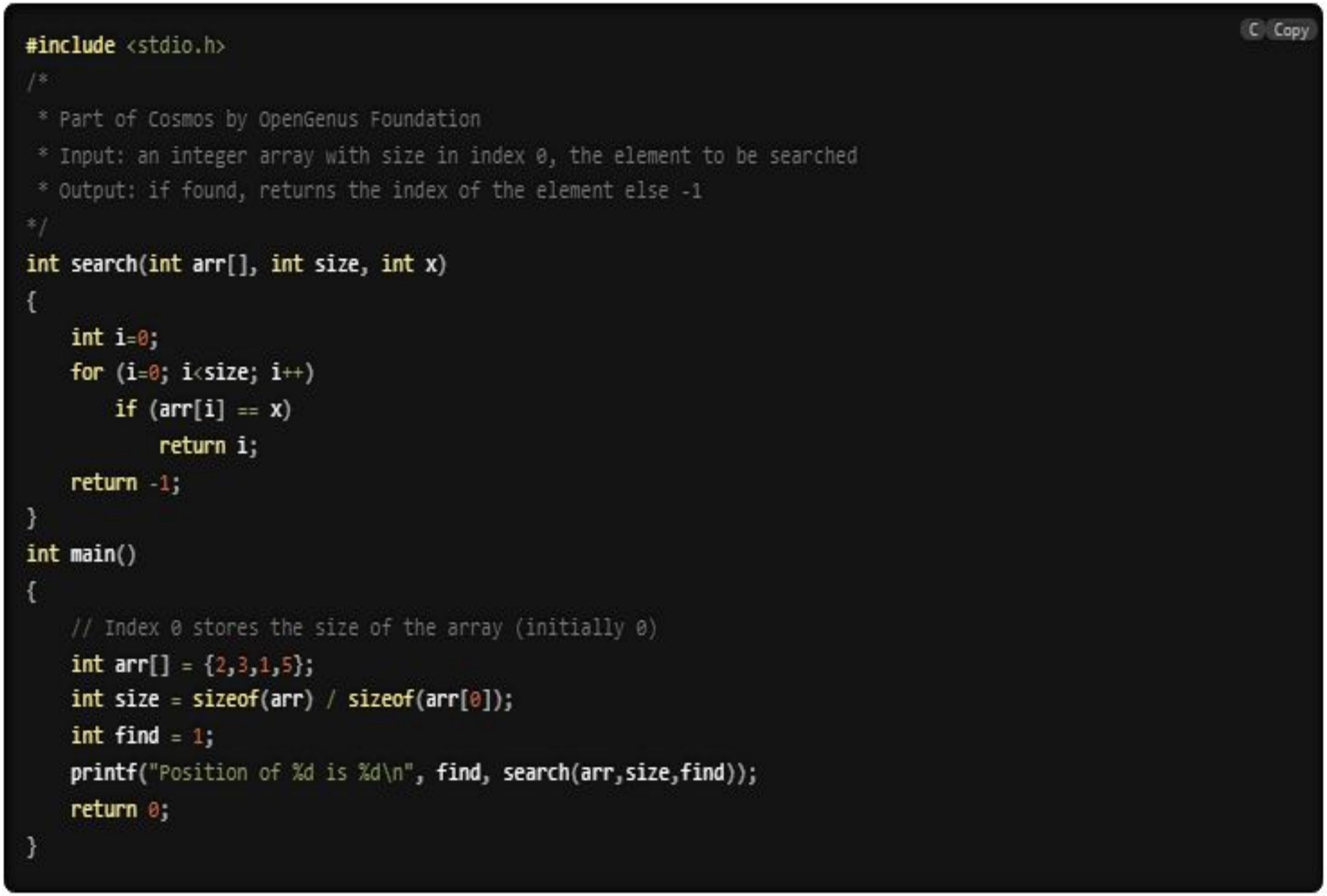
- We now understand the various time complexities, and you can recognize the best, good, and fair ones, as well as the bad and worst ones (always avoid the bad and worst time complexity).

- how you know which algorithm has which time complexity, given that this is meant to be a cheatsheet

- When your calculation is not dependent on the input size, it is a constant time complexity (O(1)).
- When the input size is reduced by half, maybe when iterating, handling recursion, or whatsoever, it is a logarithmic time complexity (O(log n)).
- When you have a single loop within your algorithm, it is linear time complexity (O(n)).
- When you have nested loops within your algorithm, meaning a loop in a loop, it is quadratic time complexity (O(n^2)).
- When the growth rate doubles with each addition to the input, it is exponential time complexity (O2^n).

Basic of Linear Search

- Linear Search Algorithm is an algorithm which checks all elements in a given list sequentially and compares with element with a given element which is the element being searched. This algorithm is used to check if an element is present in a list.


C Copy

```c
#include <stdio.h>
/*
 * Part of Cosmos by OpenGenus Foundation
 * Input: an integer array with size in index 0, the element to be searched
 * Output: if found, returns the index of the element else -1
*/
int search(int arr[], int size, int x)
{
    int i=0;
    for (i=0; i<size; i++)
        if (arr[i] == x)
            return i;
    return -1;
}
int main()
{
    // Index 0 stores the size of the array (initially 0)
    int arr[] = {2,3,1,5};
    int size = sizeof(arr) / sizeof(arr[0]);
    int find = 1;
    printf("Position of %d is %d\n", find, search(arr,size,find));
    return 0;
}
```

Analysis of Best Case Time Complexity of Linear Search

- The Best Case will take place if:
- The element to be search is on the first index.
- The number of comparisons in this case is 1. Thereforce, Best Case Time Complexity of Linear Search is **O(1)**.

Analysis of Average Case Time Complexity of Linear Search

- Let there be N distinct numbers: a1, a2, ..., a(N-1), aN
- We need to find element P.
- There are two cases:
  - **Case 1:** The element P can be in N distinct indexes from 0 to N-1.
  - **Case 2:** There will be a case when the element P is not present in the list.

- There are N numbers case 1 and 1 case 2. So, there are N+1 distinct cases to consider in total.

Case 1

- if element P is in index K, then Linear Search will do K+1 comparisons.
- Number of comparisons for all cases in case 1
  = Comparisons if element is in index 0 +
  Comparisons if element is in index 1 + ... +
  Comparisons if element is in index N-1
  = 1 + 2 + ... + N
  = N * (N+1) / 2 comparisons

Case 2

- If element P is not in the list, then Linear Search will do N comparisons.
- **Number of comparisons for all cases in case 2 = N**

- Therefore, total number of comparisons for all N+1 cases = N * (N+1) / 2 + N
  = N * ((N+1)/2 + 1)
- Average number of comparisons = (N * ((N+1)/2 + 1)) / (N+1)
  = N/2 + N/(N+1)
- The dominant term in "Average number of comparisons" is N/2. So, the Average Case Time Complexity of Linear Search is O(N).

Analysis of Worst Case Time Complexity of Linear Search

- The worst case will take place if:
- The element to be search is in the last index
- The element to be search is not present in the list
- In both cases, the maximum number of comparisons take place in Linear Search which is equal to N comparisons.
- Hence, the Worst Case Time Complexity of Linear Search is O(N).
- Number of Comparisons in Worst Case: N