For:	STRINGCODE LIMITED
Date	17/06/22
prepared by:	Justice Isreal Agbonma
Subject:	Data Structure and Algorithm (Big O notation, Time complexity and Space complexity)

Data Structure

Data structure refers to the pattern in which a set of given codes or function to a set of operation is written.

This plays a great role in the mode of execution of the given lines of code. A well-structured programming lines of code are expected to be highly readable and each steps leading to the final execution clearly defined and readable.

Algorithm

algorithm can be referred to a given problem that require solutions either mathematically or logically.

While using algorithms to prefer solutions to a given problem it is also important to consider the structure in which the solution codes are written. Some of the factors to consider while carrying out a data structure in you solution to an algorithm problem.

Consider:

1. Big O notation
2. Time complexity
3. Space complexity

Big O notation.

This is a symbolism used in complexity theory, this gives a gives a precise numeric and objective way to judging the performance of your code .the big O notation gives a better and clearer understanding on the importance of code simplicity even as the input grows or declines. It was invented by Paul Bachmann, Edmund Landau, and others, collectively called Bachmann–Landau notation or asymptotic notation.

We express complexity using big-O notation it describes the complexity of your code using algebraic terms.to make a clearer illustration , we can take a look at a typical example, O(1), which is usually pronounced “Big O 1”. O(n), which is usually pronounced “Big O n”. O(n^2), which is usually pronounced “Big O squared”. The letter “n” here represents the input size, and the function

“g(n) = n^2” inside the “O()” gives us an idea of how complex the algorithm is with respect to the input size.

This algorithm first iterates through the list with a for loop. Then for every element it uses for loop to find the smallest element in the remaining part of the list.

For a problem of size constant-time algorithm is O(1) . a linear-time algorithm is "order N" O(n) a quadratic-time algorithm is "order N squared": O(n2)

.For example, when analyzing some algorithm, it is important to note that the time taken to process or the number of steps run by the computer for an operation such as:

T= 4 n - 2 n + 2 would be less compared to running and operation like a loop function.

```
for ( let i=0; i<n;i++)
{
arr[i]
functions()
}
```

Incases of or going further to run a nested loop function

```
for ( let i=0; i<n;i++)
{
```

```
arr[i]


 for ( let j=0; j<n;j++)
{
arr[j]


 functions()
}
}.
```

for the first example T = 4 n - 2 n + 2 the processor takes an equal amount of time to run this equation irrespective of the value of n. taking for instance .

eg.

If n=5

The equation would be T = 4 (5) - 2 (5) + 2 =8.

And

If n=5000

The equation would be T = 4 (5000) - 2 (5000) + 2 =10,002.

This expression can still computer in a loop with a constant n value.

```
const answer =(n)=>{

for ( let i=0; i<n++){

arr[i]

}

}
```

Answer (4)

eg.

If n=5

The equation would be T = 4 (5) - 2 (5) + 2 =8.

And

If n=5000

The equation would be T = 4 (5000) - 2 (5000) + 2 =10,002.

This expression can still computer in a loop with a constant n value.

```
const answer =(n)=>{

for ( let i=0; i<n++){
```

arr[i]

}

}

answer (4)

This expression would be operated in constant time if O-of-4 being that the function runs at a constant time complexity but conversional it would be written as O(1).

The loop is directly proportional to the constant value of n being 4.

This means the function runs just one line of code only once.

The processor runs a simple mathematical calculation and arrive at the answer in same time and processor speed as though the value of n=5000.such equations are referred to as O-of-1 conversional written as O(1) and referred to as constant time for complexity .

For the second illustration for (let i=0; i<n;i++){ functions}the processor takes a longer time and this time then is largely dependent on the value of n where an is an array of an

unknown length , the solution of a for loop runs through the total length of the array to likely sort the required function. The time taken for such operation is largely dependent on the value of n being the length of the given array.

For an array containing just four elements

e.g.

arr =[“boy’ ,’girl’ , ‘man’, woman’]

const answer =(n)=>{

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

arr[i]

}

}

answer(arr)

Where n is the number of input in the given array

Since the array contains just 4 elements it would take a certain amount of time to loop through all elements of this given array and if the array contains more elements

eg

arr =[“boy’ ,’gitl’ , ‘man’, woman’ ….]

```
const answer =(n)=>{
for ( let i=0; i<arr.length;i++){
arr[i]
}
}
answer(arr)
```

Where n is the number of input in the given array

This function would take the whole time to loop through the whole set of elements in the array, the time taken is larky dependent on the number of elements present in the array.

Such equations are referred to as O-of-n conversional written as O(n) and referred to as linear time for complexity .

There are situations where an algorithm would require a double loop operations to effectively carry out a function.

Eg.

```
const answer =(n)=>{
```

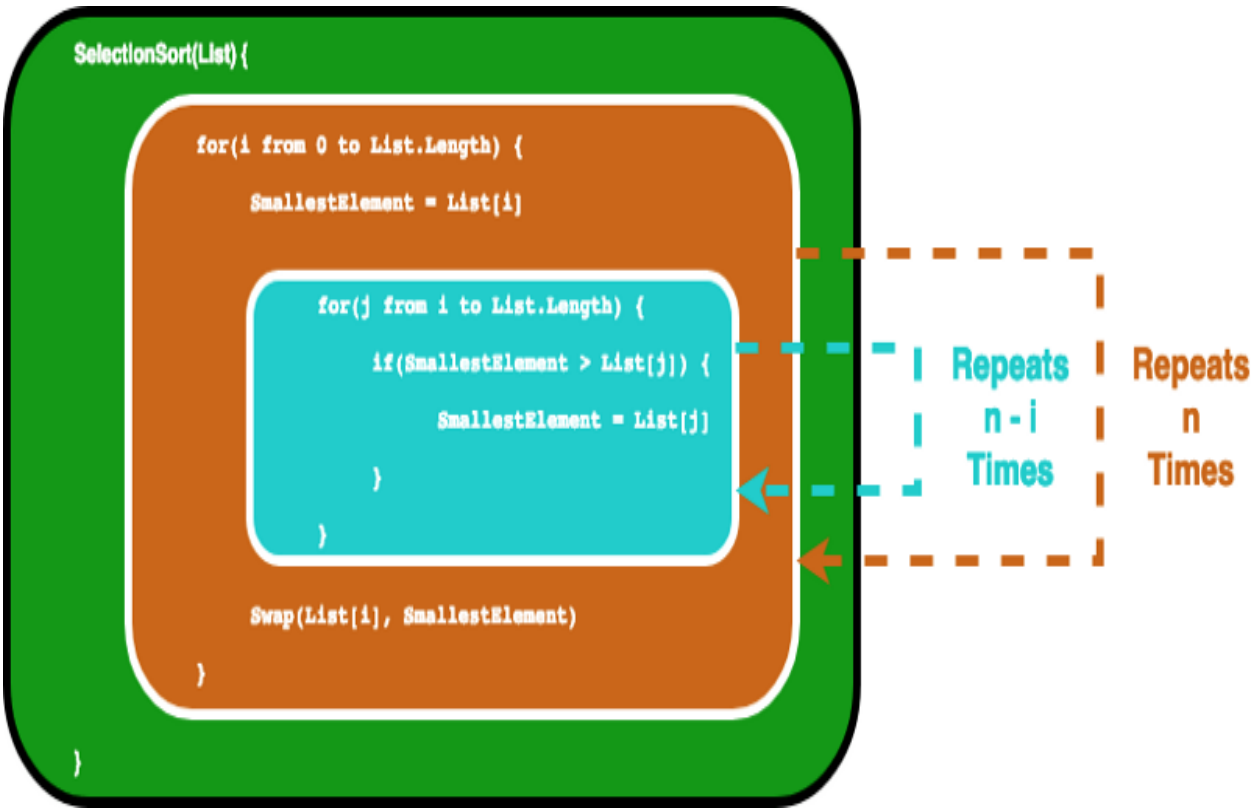
```
for ( let i=0; i<arr.length;i++){

arr[i]

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

arr[j]

}

}
```

answer (n)

O(n2)

This expression takes a longer time to execute in O(2n) time complexity.

Note that the big-O expressions do not have constants or low-order terms. This is because, when N gets large enough, constants and low-order terms don't matter (a constant-time algorithm will be faster than a linear-time algorithm, which will be faster than a quadratic-time algorithm).



This actually ends up giving us a geometric sum, and with some high-school math we would find that the inner loop will repeat for 1+2 … + n times, which equals n(n-1)/2 times. If we multiply this out, we will end up getting $n^2/2-n/2$.

When we calculate big O notation, we only care about the dominant terms, and we do not care about the coefficients. Thus we take the n^2 as our final big O. We write it as $O(n^2)$, which again is pronounced *“Big O squared”*.

TIME COMPLEXITY

While carrying out an algorithm expression time should be considered a priority , this is the amount of time taken to run efficiently a given line of code function as the input increases .
In simple terms time complexity is the amount of time taken for the processor to run a given line of code .
Time complexity can be further broken into different categories :

1. Constant complexity
2. Linear complexity
3. Quadratic complexity

1. Constant complexity **O(1)**

This has the least complexity Often called ***"constant time"***, putting time into serious consideration , if you can create an algorithm to solve the problem in O(1) time complexity,this would be considered as best practice. In some scenarios, the complexity may go beyond O(1), then we can analyze them by finding its O(1/g(n)) counterpart. For example, O(1/n) is more complex than $O(1/n^2)$.

O(log(n)) is more simplified than O(1),

As complexity is often related to divide and conquer algorithms, O(log(n)) is generally a good complexity you can reach for sorting algorithms. O(log(n)) .

eg.

If n=5

The equation would be T = 4 (5) - 2 (5) + 2 =8.

And

If n=5000

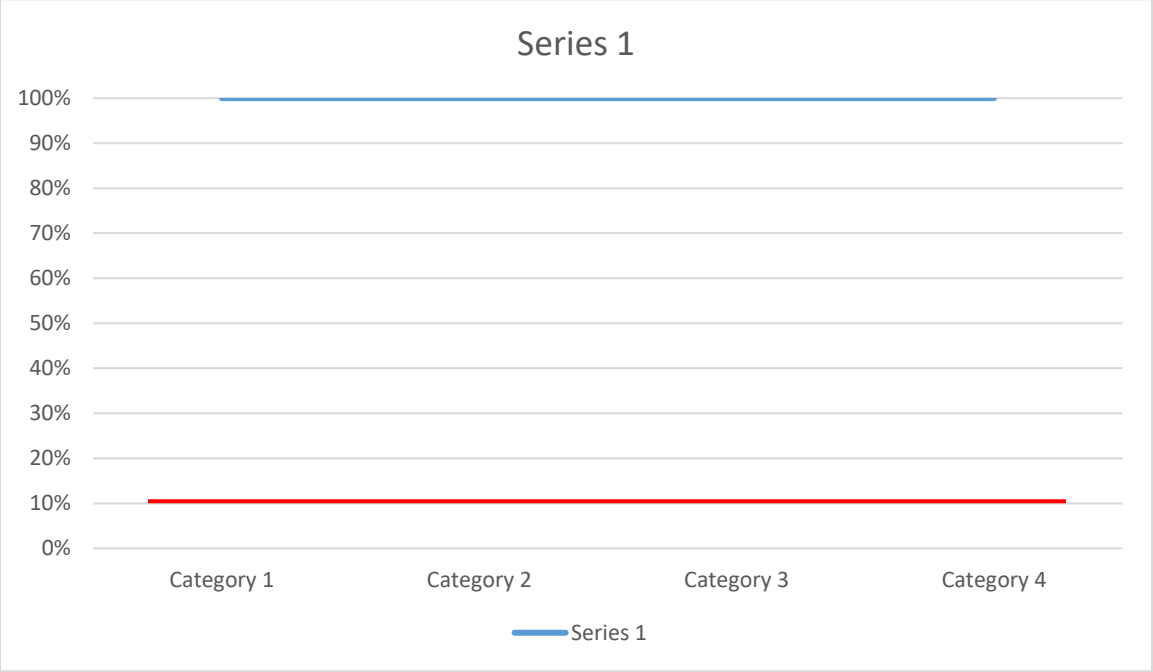
The equation would be T = 4 (5000) - 2 (5000) + 2 =10,002.

This expression can still computer in a loop with a constant n value.

```
const answer =(n)=>{
for ( let i=0; i<n++){
arr[i]
}
}
answer (4)
```

This expression would be operated in constant time if O-of-4 being that the function runs at a constant time complexity but conversional it would be written as O(1).

The loop is directly proportional to the constant value of n being 4.



Constant complexity **O(1)**

2. Linear complexity

$O(2^n)$ is more complex than $O(n^{99})$, but $O(2^n)$ is actually less complex than O(1). for (let i=0; i<n;i++){ functions}the processor takes a longer time and this time then is largely dependent on the value of n where an is an array of an unknown length , the solution of a for loop runs through the total length of the array to likely sort the required function. The time taken for such operation is largely dependent on the value of n being the length of the given array.

For an array containing just four elements

e.g.

```
arr =[“boy’ ,’girl’ , ‘man’, woman’]

const answer =(n)=>{

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

arr[i]

}

}

answer(arr)
```

Where n is the number of input in the given array

Since the array contains just 4 elements it would take a certain amount of time to loop through all elements of this given array and if the array contains more elements

eg

arr =[“boy’ ,’gitl’ , ‘man’, woman’ ….]

const answer =(n)=>{

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

arr[i]

}

}

answer(arr)

Where n is the number of input in the given array

Eg 2

Functions like:

split(“”)

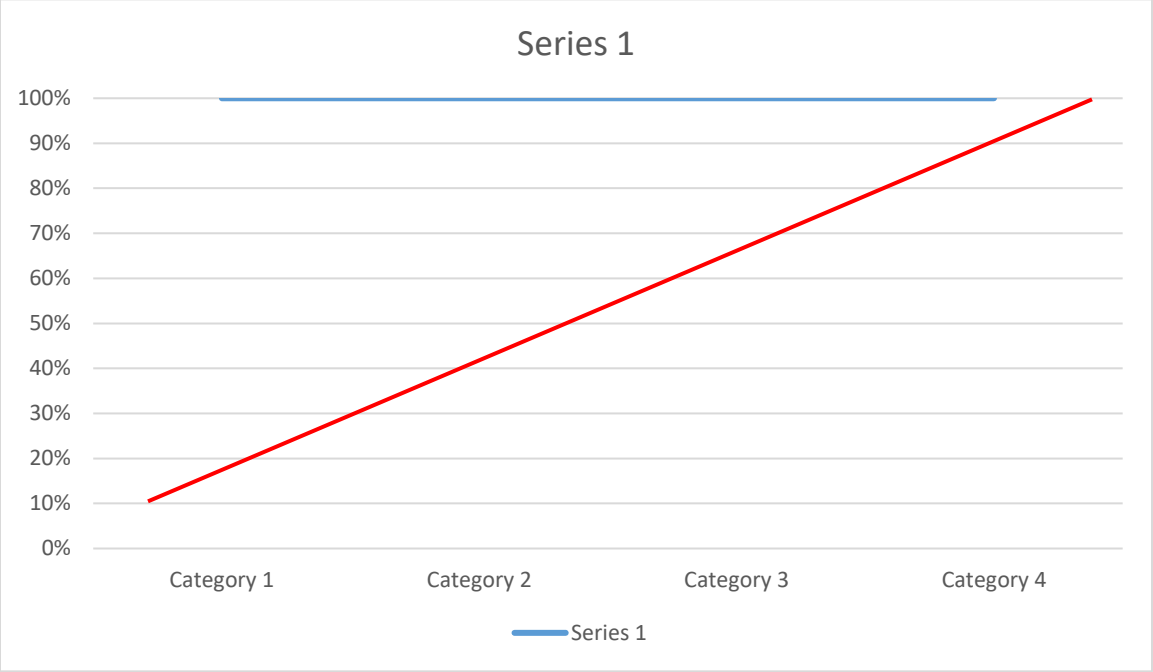
reduce()

include()

sort()

map()

all fall under the category of linear time complexity as such functions would need to indirectly loop through the target array or object to fetch , push or attract the choice target or result.



Linear complexity

3. Quadratic complexity $O(n^2)$

$O(n * \log(n))$ is more complex than $O(n)$ but less complex than $O(n^2)$, because $O(n^2) = O(n * n)$ and n is more complex than $\log(n)$.

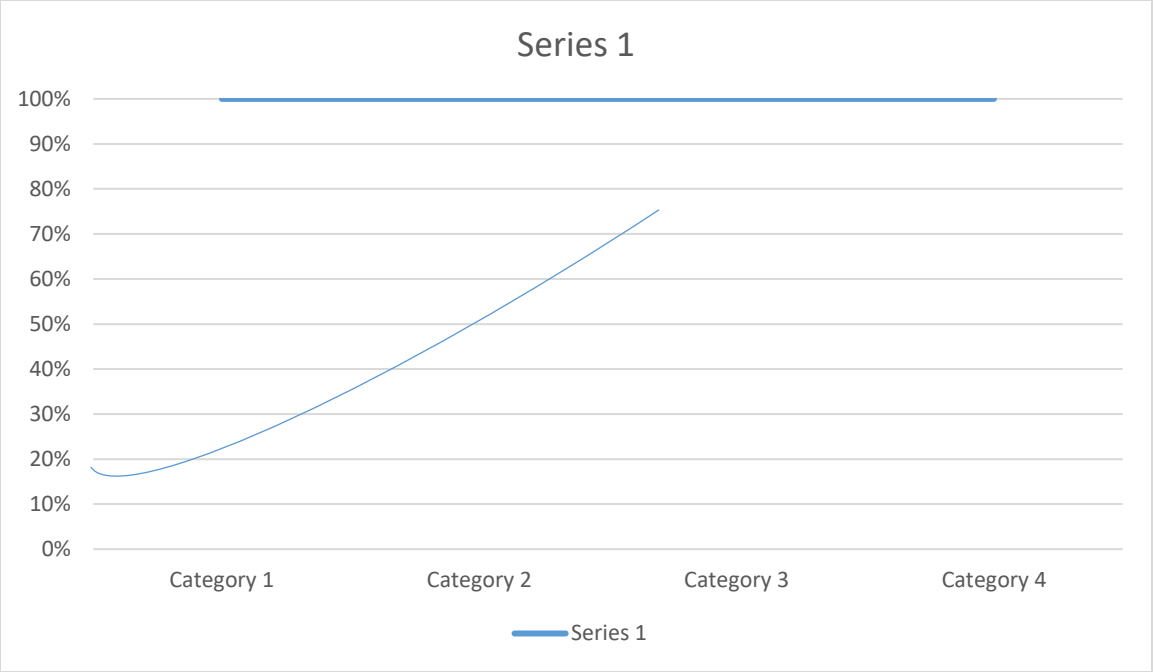
Eg.

```
const answer =(n)=>{

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

arr[i]

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

arr[j]

}

}

answer (n)
```

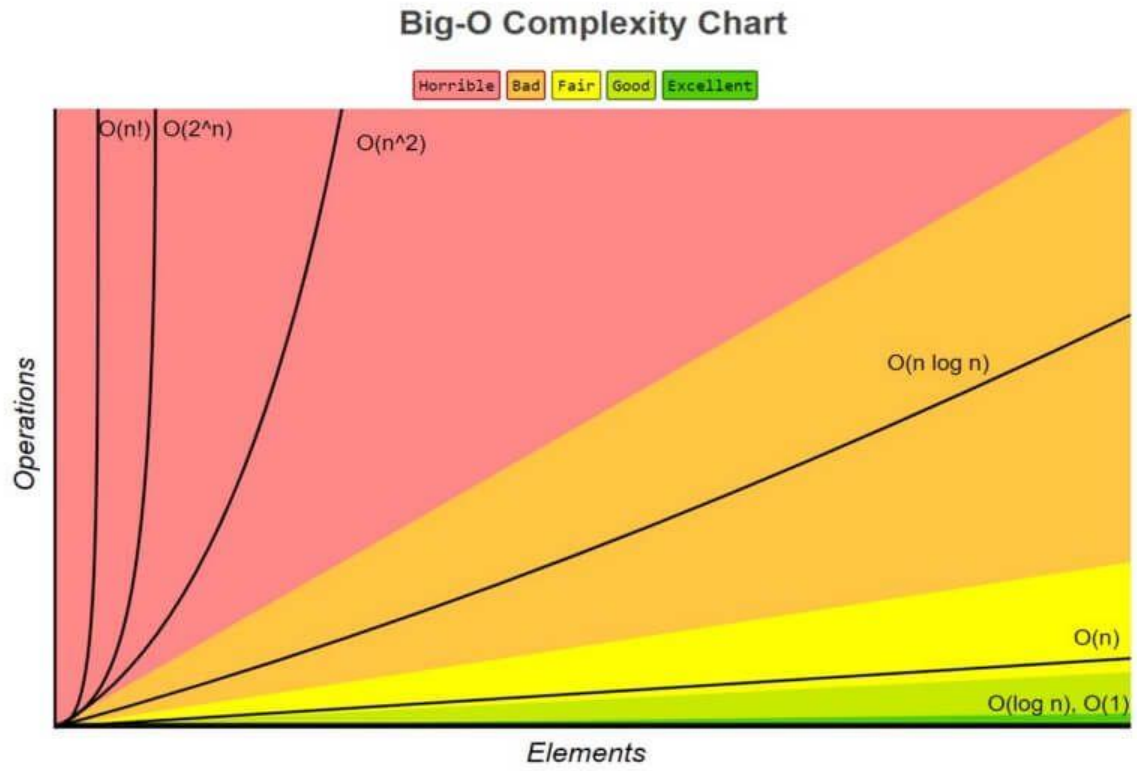
O(n2)

This kind of coding structure an algorithm would require a double loop operations to effectively carry out a function.

It takes a longer time to execute because for every round of loop for i another loop is run arr.length of the child array



Quadratic complexity $O(n^2)$



Complexity Growth Illustration from Big O Cheatsheet

Time complexity should be considered a priority over space priority as it’s important to have a faster operating app or web service than a light low byte package.

3. Space Complexity

This is how much memory used (RAM) do we need as the inputs provided to the code gets larger it is related to how much memory the program will use, and therefore is also an important factor to analyze.

The space complexity works similarly to time complexity. For example, selection sort has a space complexity of O(1), because it only stores one minimum value and its index for comparison, the maximum space used does not increase with the input size.

Most primitive (boleon and numbers) take up o(1) constant apace in the computer RAM.

Eg.

Let x = 100 and let y = 50000

Bothe variables above takes up the same amount of space in the computer RAM memory space .

Storing arrays and objects takes up O(n) linear space .

Eg.

Array1 = [“boy’, “girls’ ,”man”]

Array2 = [“boy’, “girls’ ,”man”, “cakes”, “house”]

In the array variable able ;

Array2 takes more space than array1. O(n)

Some algorithms, such as bucket sort, have a space complexity of O(n), but are able to chop down the time complexity to O(1). Bucket sort sorts the array by creating a sorted list of all the possible elements in the array, then increments the count whenever the element is encountered. In the end the sorted array will be the sorted list elements repeated by their counts.

Bucket Sort Visualization

The complexity can also be analyzed as best case, worst case, average case and expected case.

Let’s take **insertion sort,** for example. Insertion sort iterates through all the elements in the list. If the element is larger than

its previous element, it inserts the element backwards until it is larger than the previous element.

If the array is initially sorted, no swap will be made. The algorithm will just iterate through the array once, which results a time complexity of O(n). Therefore, we would say that the **best-**

5 2 6 8 7 1 2 5 7 8 9 6 2 5 4

case time complexity of insertion sort is O(n). A complexity of O(n) is also often called **linear complexity**.
Sometimes an algorithm just has bad luck. Quick sort, for example, will have to go through the list in O(n) time if the elements are sorted in the opposite order, but on average it sorts the array in O(n * log(n)) time. Generally, when we evaluate time complexity of an algorithm, we look at their **worst-case** performance. More on that and quick sort will be discussed in the next section as you read.
The average case complexity describes the expected performance of the algorithm. Sometimes involves calculating the probability of each scenarios. It can get complicated to go into the details and therefore not discussed in this article. Below

is a cheat-sheet on the time and space complexity of typical algorithms.

SEARCH ALGORITHM

This method scans through an array to find an element it can be done using .include(), .indexOf(), .find().

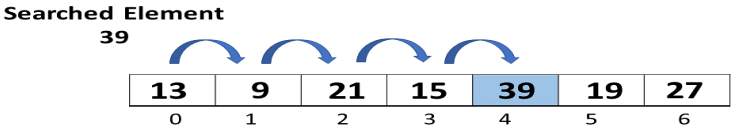
It can further be explained as a process of fetching a specific element in a collection of elements. The collection can be an array or a linked list. If you find the element in the list, the process is considered successful, and it returns the location of that element.

Search algorithm can be further broken into 2 patterns linear and binary search.

1. Linear search O(n)

This is a searching in algorithm that searches by manually checking everything in the data set to see if it is the value we are searching for, it is the most basic search technique. In this type of search, you go through the entire list and try to fetch a match for a single element. If you find a match, then the address of the matching target element is returned.

On the other hand, if the element is not found, then it returns a NULL value.



The procedures for implementing linear search are as follows:

Step 1: First, read the search element (Target element) in the array.

Step 2: In the second step compare the search element with the first element in the array.

Step 3: If both are matched, display "Target element is found" and terminate the Linear Search function.

Step 4: If both are not matched, compare the search element with the next element in the array.

Step 5: In this step, repeat steps 3 and 4 until the search (Target) element is compared with the last element of the array.

Step 6 - If the last element in the list does not match, the Linear Search Function will be terminated, and the message "Element is not found" will be displayed.

Linear Search (Array Arr, Value a) // Arr is the name of the array, and a is the searched element.

Step 1: Set i to 0 // i is the index of an array which starts from 0

Step 2: ifi > n then go to step 7 // n is the number of elements in array

Step 3: if Arr[i] = a then go to step 6

Step 4: Set i to i + 1

Step 5: Goto step 2

Step 6: Print element a found at index i and go to step 8

Step 7: Print element not found

Step 8: Exit

Start

linear_search (Array , value)

For each element in the array

If (searched element == value)

Return's the searched lament location

end if

end for

end

1. Linear search can be applied to both single-dimensional and multi-dimensional arrays.
2. Linear search is easy to implement and effective when the array contains only a few elements.
3. Linear Search is also efficient when the search is performed to fetch a single search in an unordered-List.

2. Binary search O(logn)

   This is a searching algorithm that can search in O(logn) time for assorted array , compared to a linear search for an unsorted array.

   Binary search only works on sorted arrays and is able to search that array much faster due to the presorting. It achieves this by eliminating half of the things to search for by each search.

Binary Search Algorithm can be implemented in the following two ways

1. Iterative Method
2. Recursive Method

1. Iteration Method

```
binarySearch(arr, x, low, high)
    repeat till low = high
        mid = (low + high)/2
            if (x == arr[mid])
            return mid


            else if (x > arr[mid]) // x is on the right side
                low = mid + 1


            else                // x is on the left side
                high = mid - 1
```

2. Recursive Method (The recursive method follows the divide and conquers approach)

```
binarySearch(arr, x, low, high)
      if low > high
          return False


      else
          mid = (low + high) / 2
              if x == arr[mid]
              return mid


          else if x > arr[mid]        // x is on the right side
              return binarySearch(arr, x, mid + 1, high)
```

```
else                    // x is on the right side
    return binarySearch(arr, x, low, mid - 1)
```