Algorithms & Time Complexity

Algorithm:

Sequence of steps to solve a clearly defined problem.

Which Algorithm is the best:

For solving a particular problem, there can be multiple ways of approaching it. But how do we know what approach is the best? Well, in case of algorithms, the most performant one is the one which takes less time, and less space in memory.

Comparing Performance of Algorithms:

Time Complexity of Algorithms is displayed using asymptotic notations. Some most common ones are Omega, Theta and Big O. Generally, Big O notation is used for describing the time complexity of any algorithm as it measures it for the worst case i.e one input is too big.

Commonly Used Time Complexities:

Most commonly used time complexities are:

- Constant Time Complexity O (1)
- Linear Time Complexity O (n)
- Quadratic Time Complexity O (n^2)
- Logarithmic Time Complexity O (log n)
- Exponential Time Complexity O (2^n)

Let's see each of them in details

Constant Time Complexity O (1):

Constant time complexity means that the algorithm will take constant time to run regardless of the input. It will take the same amount of time for 10 values and the same amount for time for hundreds and thousands of values as well.

Example:

Below is the function which takes an array as an input and returns the last element of the array:

```
function getLastElement( array ){
        return array[ array.length - 1 ]
}
```

Explanation:

Let's say we have an array of size 10. The *getLastElement* function will receive that array and simply returns the last element using its index, no calculations required. So, for an

array of size 100 or even 1000, the procedure will be the same and it won't affect the runtime of the algorithm at all.

Linear Time Complexity O (n):

Linear time complexity means that the algorithm's runtime will increase in a linear manner as the input grows. The algorithm will be fast for a smaller number of *n* but it will keep getting slower as the *n* grows.

Example:

```
function sumArray( array ){
        let result = 0;
        for( let i = 0; i < array.length; i++ ){
                result  += array[i]
        }
        return result
}
```

Explanation:

In the above function, the *for* loop iterates over every element of the array. As the size of the array increases, the number of iterations increases and thus it results in increased runtime of the function.

Quadratic Time Complexity O (n^2):

Quadratic time complexity means that the algorithm's runtime will grow proportionally to the square of the size of input data. This typically happens when there are two nested loops, iterating over the input size.

Example:

```
function bubbleSort( array ){
        let n = array.length;
        for ( let i = 0; i < n; i++ ){
                for ( let j = 0; j <  n - i - 1; j++ ){
                        if ( array[j] > array[i] + 1 ){
                        let temp = array[j];
                        array[j] = array[j+1];
                        array[j+1] = temp;
                        }
                }
        }
        return array;
}
```

Explanation:

In the above example, there is a function which takes an array as input and runs two nested loops on it, the outer loop runs once and with each iteration of outer loop, inner loop executes from 0 to *n*, making the time increasing proportional to the square of the size of array.

Logarithmic Time Complexity O (log n):

Logarithmic time complexity means that the algorithm's runtime will be cut to half with every iteration of the loop. These algorithms are much faster than linear time algorithms, especially for large datasets.

Example:

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

Explanation:

In the above algorithm, we search for the *target* element in the *array.* With every iteration, we are cutting the array to half to shorten the search area which makes this algorithm more efficient than linear algorithms which run on the whole size of the array.

Exponential Time Complexity O (2^n):

Exponential time complexity means that the algorithm's runtime doubles with each additional input element. These algorithms are extremely slow for large inputs because the number of operations grows exponentially.

Example:

```
function fibonacci( n ){
        if( n <= 1 ){
                return n;
```

```
        }
        return fibonacci( n - 1 ) + fibonacci( n - 2 );
    }
```

Explanation:
In this algorithm, we calculate the *nth* Fibonacci number recursively. For each *n*, the function calls itself twice: once for *fibonacci(n - 1)* and once for *fibonacci(n - 2)*. Each of these recursive calls further calls itself, generating a binary tree of function calls. This doubling of recursive calls with each level is what gives the algorithm its exponential time complexity.