Big O Notation: Used to analyze the performance of the application.

Constant Time Complexity: If an algorithm's time complexity is constant, it means that it will always run in the same amount of time, no matter the input size.

Exp:

```
function timesTwo(num) {
  return 2 * num
}
let result = timesTwo(5) // 10
let result2 = timesTwo(2000) // 4000
```

Time complexity: Analysing how the runtime of an algorithm changes as the input increases.

Space complexity: The space required by the algorithm, not including inputs.

Linear time complexity (O(n)): When the running time of an algorithm increases linearly with the size of the input.

Exp:

```
function reverseArray(arr) {
  let newArr = []
  for (let i = arr.length - 1; i >= 0; i--) {
          newArr.push(arr[i])
  }
  return newArr
}
const reversedArray1 = reverseArray([1, 2, 3]) // [3, 2, 1]
const reversedArray2 = reverseArray([1, 2, 3, 4, 5, 6]) // [6, 5, 4, 3, 2, 1]
```

Quadratic time complexity(O(n2)): The runtime of the algorithm is directly proportional to the square of the size of the input.

Exp:

```
function multiplyAll(arr1, arr2) {
  if (arr1.length !== arr2.length) return undefined
  let total = 0
  for (let i of arr1) {
          for (let j of arr2) {
          total += i * j
          }
  }
  return total
}
let result1 = multiplyAll([1, 2], [5, 6]) // 33
let result2 = multiplyAll([1, 2, 3, 4], [5, 3, 1, 8]) // 170
```

Logarithmic time complexity(O log(n)): The input size grows, the number of operations grows very slowly.

Exp:

```
function logTime(arr) {
  let numberOfLoops = 0
  for (let i = 1; i < arr.length; i *= 2) {
          numberOfLoops++
  }
  return numberOfLoops
}
let loopsA = logTime([1]) // 0 loops
let loopsB = logTime([1, 2]) // 1 loop
let loopsC = logTime([1, 2, 3, 4]) // 2 loops
let loopsD = logTime([1, 2, 3, 4, 5, 6, 7, 8]) // 3 loops
let loopsE = logTime(Array(16)) // 4 loops
```

Linearithmic time complexity(n log(n)): Observed when there is a nested loop structure where the outer loop runs in linear time and the inner loop exhibits a time complexity of O(log n).

Exp:

```
function linearithmic(n) {
  for (let i = 0; i < n; i++) {
          for (let j = 1; j < n; j = j * 2) {
          console.log("Hello")
          }
  }
}
```

Exponential time complexity(O(2^n)): The growth rate doubles with each addition to the input (n), often iterating through all subsets of the input elements.

Exp:

```
function fibonacci(num) {
  // Base cases
  if (num === 0) return 0
  else if (num === 1) return 1
  // Recursive part
  return fibonacci(num - 1) + fibonacci(num - 2)
}
fibonacci(1) // 1
fibonacci(2) // 1
fibonacci(3) // 2
fibonacci(4) // 3
```

fibonacci(5) // 5

Factorial time complexity(O(n!)): When it grows in a factorial way based on the size of the input data.
Exp:

```
function factorial(n) {
  let num = n
  if (n === 0) return 1
  for (let i = 0; i < n; i++) {
          num = n * factorial(n - 1)
  }
  return num
}
```