Big-O Notation

It is a simplified analysis of an Algorithm’s efficiency.

1. Complexity of the Algorithm is measured in terms of input size **N**
2. The Specifications of the Machine are irrelevant i.e. notation is machine-independent
3. Big-O can be used to analyze both Time and Space

Types of Measurements

1. Best Case Scenario
2. Worst Case Scenario
3. Average Case Scenario

When we look at Big-O for an Algorithm, we typically look at the **Worst Case** Scenario.

General Rules

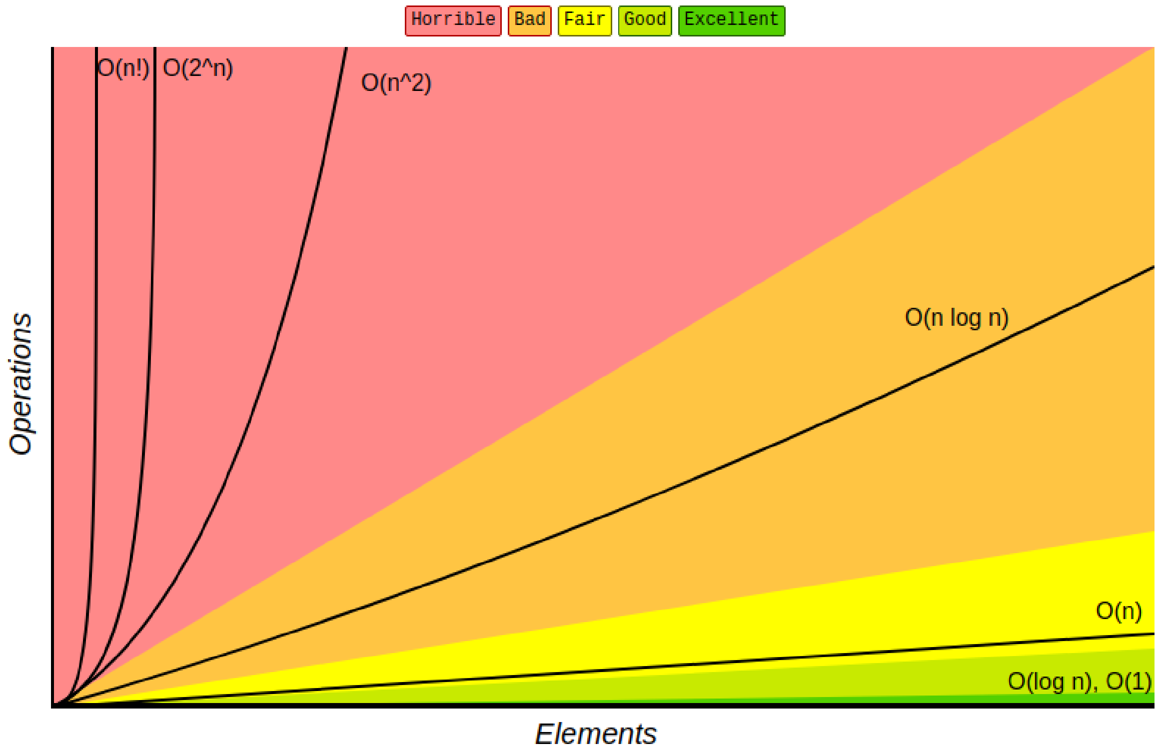
1. The Notation ignores Constants. As the input size **n** increases, the Constant **5** no longer matters

```
5n -> O(n)
```

2. Certain Terms Dominate others. We ignore low-order terms when higher order terms are present

```
O(1) < O(logn) < O(n) < O(nlogn) < O(n^2) < O(2^n) < O(n!)
```

Big-O Complexity Chart



Examples

Constant Time

```
// O(1) :: doesn't depend on Input Size N
int x = 300 + (50 * 10);
```

```
int x = 300 + (50 * 10);                         // O(1)
int y = 1000 / 2;                                // O(1)
std::cout << (x + y);                            // O(1)
// O(1) + O(1) + O(1) == 3 * O(1) == O(1)
```

Note: Constants are Ignored by the Notation.

Linear Time

```
for(int i = 0; i < n; i++)
      std::cout << i << std::endl;               // O(1)
// n * O(1) == O(N)
```

```
int x = 300 + (10 * 37);                               // O(1)
for(int i = 0; i < n; i++)                             //
      std::cout << i << std::endl;                     // O(n)
// O(1) + O(n) == O(n)
```

Note: A Higher order term **O(n)** was present which dominated the lower order term **O(1)**

Quadratic Time

```
for(int i = 0; i < n; i++) {
      for(int j = 0; j < n; j++) {
            std::cout << i * j << std::endl;           // O(1)
      }
}
// O(n) * O(n) == O(n^2)
```

```
if(x > o) {
      // O(1)
}
else if (x < 0) {
      // O(logn)
}
else {
      // O(n^2)
}
// worst-case scenario :: O(n^2)
```