COMPLEXICITY

An Introduction to Asymptotic Notations

The performance of an algorithm can change with a change in the input size. That is where Asymptotic Notations like Big O Notation comes into play. Asymptotic Notations can describe an algorithm's run time when the input tends toward a specific or limiting value. Asymptotic analysis helps to analyze the algorithm performance change in the order of input size.

What is Big O Notation in Data Structure?

Big O Notation in Data Structure is used to express algorithmic complexity using algebraic terms. It describes the upper bound of an algorithm's runtime and calculates the time and amount of memory needed to execute the algorithm for an input value.

Mathematical Definition

Consider the functions f(n) and g(n), where functions f and g are defined on an unbounded set of positive real numbers. g(n) is strictly positive for every large value of n.

The function f is said to be O(g) (read as big- oh of g), if, for a constant c>0 and a natural number n0, f (n) $\leq$ CG(n) for all n >=

n0

This can be written as:

f(n) = O(g(n)), where n tends to infinity (n $\rightarrow \infty$)

We can simply write the above expression as:

f(n) = O(g(n))

Properties of Big O Notation

The most important properties of Big O Notation in Data Structure are:

- **Constant Multiplication:**

If f(n) = CG(n), then O(f(n)) = O(g(n)) for a constant c > 0

- **Summation Function:**

If f(n) = f1(n) + f2(n) + -- + FM(n) and fi(n)$\leq$ fi+1(n) $\forall$ i=1, 2, --, m,

then O(f(n)) = O(max(f1(n), f2(n), --, fm(n)))

- **Logarithmic Function:**

If f(n) = log an and g(n)=log bn, then

O(f(n)) = O(g(n))

- **Polynomial Function:**

If f(n) = a0 + a1.n + a2.n2 + -- + am.nm, then
O(f(n)) = O(nm)

How Does Big O Notation Make a Runtime Analysis of an Algorithm?

In order to analyze and calculate an algorithm's performance, we must calculate and compare the worst-case runtime complexities of the algorithm. The order of O(1) - known as the Constant Running Time - is the fastest running time for an algorithm, with the time taken by the algorithm being equal for different input sizes. Although the Constant Running Time is the ideal runtime for an algorithm, it can be rarely achieved because the runtime depends on the size of n inputted.
For example, runtime analysis of an algorithm for a size of n = 20:
n=20,
log (20) = 2.996
20 = 20
20 log (20) = 59.9
20^2 = 400
2^20 = 1084576
20! = 2.432902 + 1818

- Runtime complexity of some common algorithmic examples:
- Runtime Complexity for Linear Search – O(n)
- Runtime Complexity for Binary Search – O(log n)
- Runtime Complexity for Bubble Sort, Insertion Sort, Selection Sort, Bucket Sort - O(n^c).
- Runtime Complexity for Exponential algorithms like Tower of Hanoi - O(c^n).
- Runtime Complexity for Heap Sort, Merge Sort - O(n log n).

How Does Big O Notation Analyze Space Complexity?

It is also essential to determine the space complexity of an algorithm. This is because space complexity indicates how much memory space the algorithm occupies. We compare the worst-case space complexities of the algorithm.
Before the Big O notation analyzes the Space complexity, the following tasks need to be implemented:

1. Implementation of the program for a particular algorithm.
2. The size of input n needs to be known to calculate the memory each item will hold.

Space Complexities of some common algorithms:
Linear Search, Binary Search, Bubble sort, Selection sort, Heap sort, Insertion sort - Space Complexity is O(1).

- Radix sort - Space complexity is O(n+k).
- Quick Sort - Space complexity is O(n).
- Merge sort - Space complexity is O(log n).

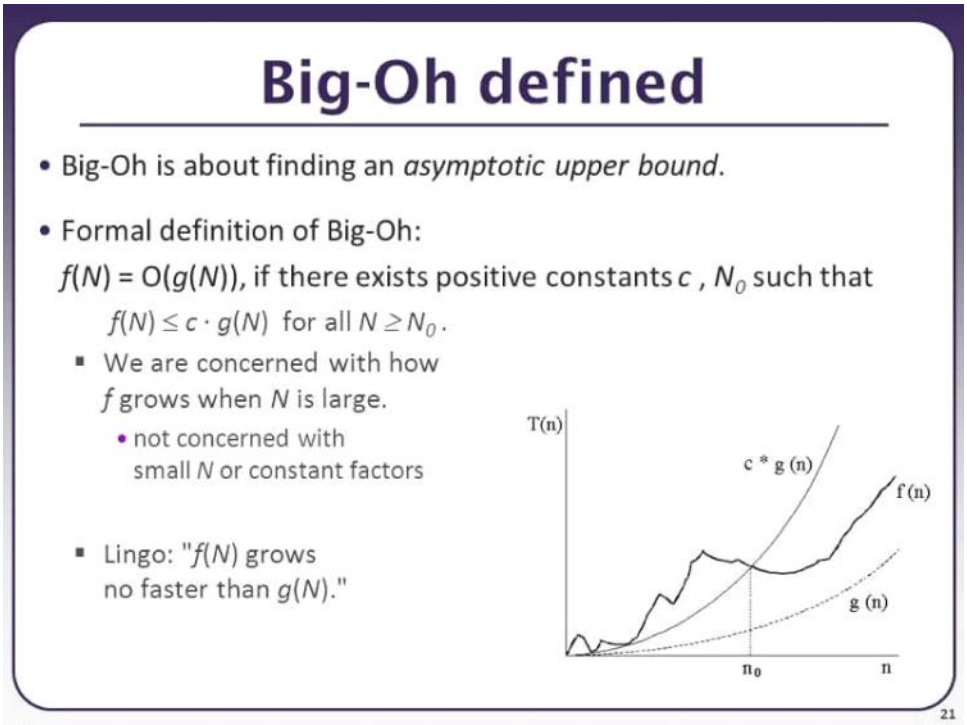
Example of Big O Notation in C

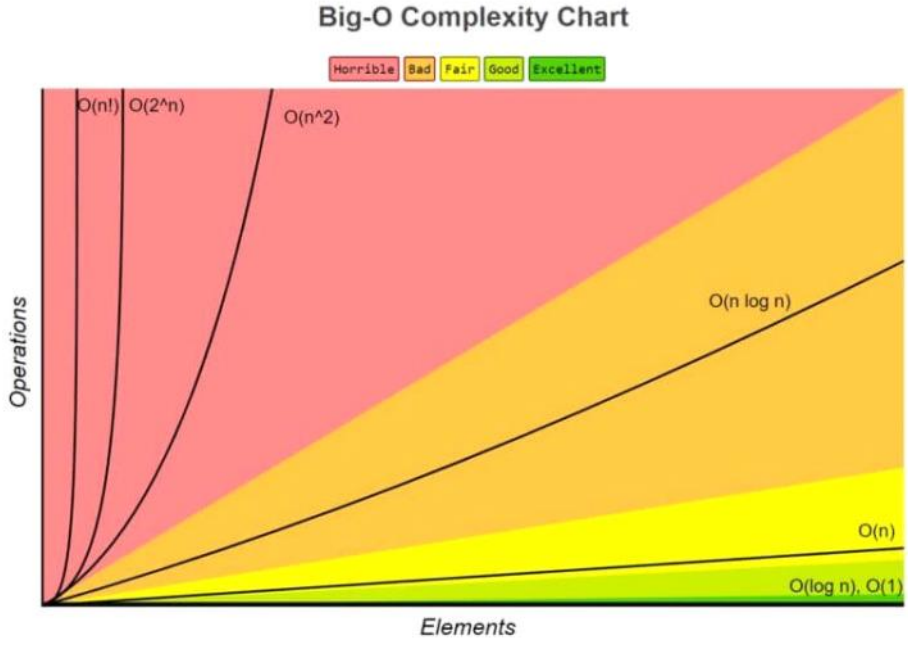
Implementation of Selection Sort algorithm in C to find worst-case complexity (Big O Notation) of the algorithm:

```
for(int i=0; i<n; i++)
{
  int min = i;
  for(int j=i; j<n; j++)
  {
    if(array[j]<array[min])
      min=j;
  }
  int temp = array[i];
  array[i] = array[min];
  array[min] = temp;
}
```

Explanation:

The range of the first (outer) for loop is i<n, meaning the order of the loop is O(n).
The range for the second (inner) for loop is j<n; so, the order of the loop is again O(n).
Average efficiency is calculated as n/2 for a constant c, but we ignore the constant.
Thus, the order comes to be O(n).
We get runtime complexity by multiplying the inner and outer loop order. It is O(n^2).
In this way, you can implement other algorithms in C, and analyze and determine the complexities.





1. O(1) has the least complexity

Often called ***"constant time"***, if you can create an algorithm to solve the problem in O(1), you are probably at your best. 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).

2. O(log(n)) is more complex than O(1), but less complex than polynomials

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)) is less complex than O($\sqrt{n}$), because the square root function can be considered a polynomial, where the

exponent is 0.5.

3. Complexity of polynomials increases as the exponent increases

For example, $O(n^5)$ is more complex than $O(n^4)$. Due to the simplicity of it, we actually went over quite many examples of polynomials in the previous sections.

4. Exponentials have greater complexity than polynomials as long as the coefficients are positive multiples of n

$O(2^n)$ is more complex than $O(n^{99})$, but $O(2^n)$ is actually less complex than $O(1)$. We generally take 2 as base for exponentials and logarithms because things tends to be binary in Computer Science, but exponents can be changed by changing the coefficients. If not specified, the base for logarithms is assumed to be 2.

5. Factorials have greater complexity than exponentials

If you are interested in the reasoning, look up the Gamma function, it is an analytic continuation of a factorial. A short proof is that both factorials and exponentials have the same number of multiplications, but the numbers that get multiplied grow for factorials, while remaining constant for exponentials.

6. Multiplying terms

When multiplying, the complexity will be greater than the original, but no more than the equivalence of multiplying something that is more complex. For example, $O(n * log(n))$ 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).