UNIVERSITY OF ENGINEERING & MANAGEMENT, KOLKATA

Course Name : Data Structure using C
Topic: Time & Space Complexity

Kaustuv Bhattacharjee

Introduction

- **Time Complexity** of an algorithm is the running time of a program (number of machine instructions a program executes) as a function of the input size. It depends on the algorithm itself and the input size.

- **Space complexity** of an algorithm is the amount of computer memory that is required during the program execution as a function of the input size. Depends on two factors:
  - **Fixed part:** Space needed for storing instructions, constants, variables, and structured variables (like arrays and structures)
  - **Variable part:** Space needed for recursion stack, and for structured variables that are allocated space dynamically during the runtime of a program.

Some Concepts

- **Worst Case Running Time:** This is the upper bound on the running time for any input. This means the algorithm will never go beyond the upper bound.
- **Best Case Running Time:** This is the lower bound on the running time for any input. This means the algorithm can never improve beyond the lower bound irrespective of any input.
- **Average Case Running Time:** An algorithm is an estimate of the running time for an 'average' input. It assumes that all inputs of a given size are equally likely.
- **Amortized Case Running Time:** It refers to the time required to perform a sequence of (related) operations averaged over all the operations performed. Amortized analysis guarantees the average performance of each operation in the worst case.

Time-Space Trade Off

- Best algorithm to solve a problem is that requires **less memory space** and takes **less time** to complete its execution.
- This is practically not always possible. Hence require **time-space trade-off.**
- If space is a big constraint, then one might choose a program that takes less space at the cost of more CPU time.
- If time is a major constraint, then one might choose a program that takes minimum time to execute at the cost of more space.

Expressing Time and Space Complexity

- Expressed using a function f(n) where n is the input size for a given instance of the problem

Objective:

- To predict the rate of growth of complexity as the input size of the problem increases
- Find an optimal solution to a given problem that is most efficient

Algorithm Efficiency-Linear Loops

- ***The loop updation statements either adds or subtracts the loop-controlling variable.***

- ***Example 1:***

```
for(i=0;i<100;i++)
        statement block;
```

Here 100 is the loop factor. Hence the formula is **f(n)=n**

- ***Example 2:***

```
for(i=0;i<100;i+=2)
        statement block;
```

Here the number of iterations is half the number of the loop factor. Hence the formula is **f(n)=n/2**

Algorithm Efficiency- Logarithmic Loops

- ***The loop-controlling variable is either multiplied or divided during each iteration of the loop.***

- ***Example 1:***

```
for(i=0;i<1000;i*=2)
        statement block;
```

Here the loop will be executed 10 times, which is approximately $\log_2 1000$.

Same is the scenario for the following example:

```
for(i=1000;i>0;i/=2)
        statement block;
```

Hence the general formula is **$f(n) = \log_2 n$**

Algorithm Efficiency- Nested Loops

- ***To determine the number of iterations each loop completes. The total is then obtained as the product of the number of iterations in the inner loop and the number of iterations in the outer loop.***

- ***Example 1 (Linear logarithmic loop):***

```
for(i=0;i<10;i++)
    for(j=1; j<10;j*=2)
        statement block;
```

Here the outer loop will be executed 10 times and the inner loop will be executed $\log_2 10$ times. Hence, the number of iterations for this code can be given as $10 \log_2 10$.

The general formula is **$f(n) = n\log_2 n$**

Algorithm Efficiency- Nested Loops Contd…

- ***Example 2 (Quadratic loop -number of iterations in the inner loop is equal to the number of iterations in the outer loop):***

```
for(i=0;i<10;i++)
    for(j=0; j<10;j++)
        statement block;
```

Here the outer loop will be executed 10 times and the inner loop will also be executed 10 times. Hence, the number of iterations for this code can be given as 10*10=100.

The general formula is $f(n)= \mathbf{n^2}$

Algorithm Efficiency- Nested Loops Contd…

- ***Example 3 (Dependent Quadratic loop -number of iterations in the inner loop is dependent on the outer loop):***

```
for(i=0;i<10;i++)
    for(j=0; j<=i; j++)
        statement block;
```

Here the outer loop will be executed 10 times. Inner loop will execute just once in the first iteration, twice in the second iteration, thrice in the third iteration, so on and so forth. Hence, the number of iterations can be calculated as

$1 + 2 + 3 + \ldots + 9 + 10 = 55$

If we calculate the average of this loop (55/10 = 5.5), we will observe that it is equal to the number of iterations in the outer loop (10) plus 1 divided by 2.

The general formula is $f(n) = \mathbf{n(n+1)/2}$

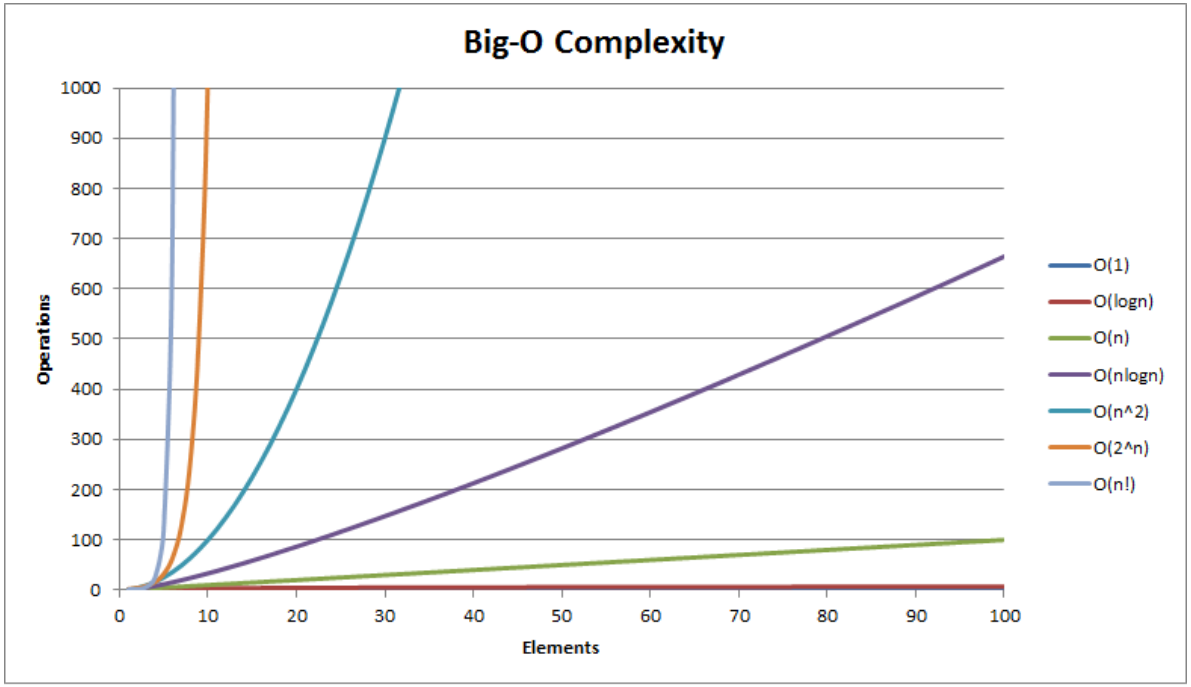
Time Complexity – Polynomial vs Exponential

- ***Polynomial Function:*** A function with expressions where an input variable is raised to a constant power, say x^a where x is a variable and a is a constant.
- Example: $x^3 + 3x + 3$

- ***Exponential Function:*** An exponential function is defined by the formula $f(x) = a^x$, where the input variable x occurs as an exponent.
- Example: $2^x + 3x + 3$

Time Complexity – Polynomial vs Exponential behavior

- Any exponential function will grow significantly faster (long term) than any polynomial function, so the distinction is relevant to the efficiency of an algorithm, especially for large values of n.

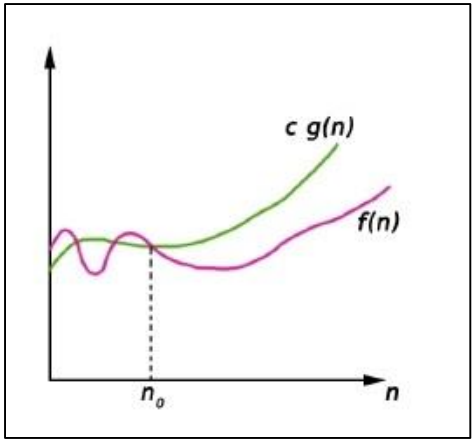


Big O Notation

- Big O notation (O stands for *Order of*) expresses complexity of a function for a large value of input variable **n** [written as **O(n)**].
- It represents the **upper bound** of the runtime of an algorithm. That means, to calculate the longest time an algorithm can take for its execution.
- It is used for calculating the **worst-case time** complexity of an algorithm
- It ignores the constant multipliers [so **O(n)** is equivalent to **O(4n)**].
- **Example:** If a sorting algorithm performs $\mathbf{n^2}$ operations to sort just **n** elements, then that algorithm would be described as an $\mathbf{O(n^2)}$ algorithm.

Big O Notation- Interpretation

- **Big-O (O)** notation gives an upper bound for a function **f(n)** to within a constant factor.
- **f(n) = O(g(n)),** If there are positive constants n_0 and **c** such that, to the right of n_0 the **f(n)** always lies on or below **c*g(n).** c is a constant which depends on the following factors:
  - The programming language used
  - The quality of the compiler or interpreter
  - The CPU speed
  - The size of the main memory and the access time to it
  - The knowledge of the programmer
  - The algorithm itself



Big O Notation- Examples & Limitations

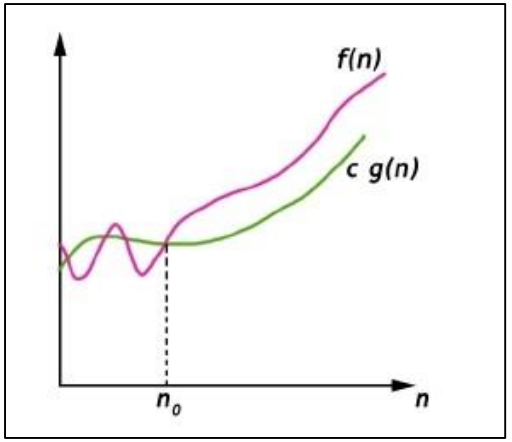
- Examples of functions in $O(n^3)$ include: $n^{2.9}$, n^3, $n^3 + n$, $540n^3 + 10$.
- **Limitations:**
- Many algorithms are too hard to analyze mathematically.
- There may not be sufficient information to calculate the behavior of the algorithm in the average case.
- Big O analysis only tells us how the algorithm grows with the size of the problem, not how efficient it is, as it does not consider the programming effort.
- It ignores important constants. For example, if one algorithm takes $O(n^2)$ time to execute and the other takes $O(100000n^2)$ time to execute, then as per Big O, both algorithm have equal time complexity. In real-time systems, this may be a serious consideration.

Omega (Ω) Notation

- It represents the **lower bound** of the runtime of an algorithm.
- It is used for calculating the **best time** an algorithm can take to complete its execution.
- Used for measuring the **best case time** complexity of an algorithm.
- It means that the function can never do better than the specified value but it may do worse.

Omega (Ω) Notation- Interpretation & Examples

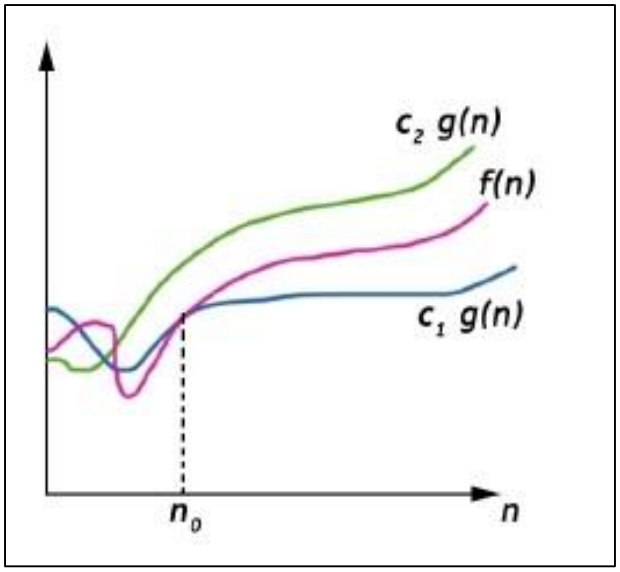
- **Big-Omega (Ω)** notation gives a lower bound for a function **f(n)** to within a constant factor.
- **f(n) = Ω(g(n)),** If there are positive constants n_0 and c such that, to the right of n_0 the **f(n)** always lies on or above **c*g(n).** **[0 ≤ c g(n) ≤ f(n), for all n >= n_0]**



- **Examples:** Ω(n^2) include: n^2, $n^{2.9}$, $n^3 + n^2$, n^3

Theta(θ) Notation

- **Theta(θ)** notation gives bound for a function **f(n)** to within a constant factor.
- We write **f(n) = θ(g(n)),** If there are positive constants n_0 ,c_1 and c_2 such that, to the right of n_0 the **f(n)** always lies between c_1*g(n) and c_2*g(n) inclusive.



Time Complexity Problems

1. In each of the below mentioned scenario, deduce the general formula f(n) where n is the input size for a given instance of the problem:

a.
```
for(i=0;i<100;i+=2)
        statement block;
```
b.
```
for(i=1000;i>0;i/=2)
        statement block;
```
c.
```
for(i=0;i<10;i++)
        for(j=1; j<10;j*=2)
                statement block;
```
d.
```
for(i=0;i<10;i++)
        for(j=0; j<10;j++)
                statement block;
```
e.
```
for(i=0;i<10;i++)
        for(j=0; j<=i; j++)
                statement block;
```

2. What is a polynomial function and an exponential function? Provide examples in each of the functions.

3. Let $f(n)=2^n$ and $g(n)=n^{2.}$ Compare the values of f(n) and g(n) for n=1 to n, and provide the conclusion on the rate of increase among them.

Thank You