Topics Covered:

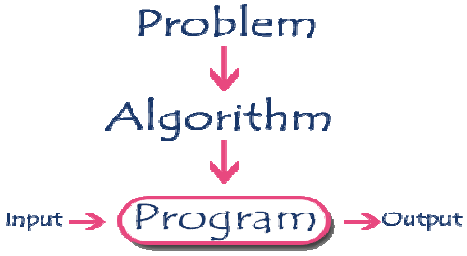
Algorithm, Efficiency of an Algorithm, Time and Space Complexity, Asymptotic notations: Big Oh, Big Theta and Big Omega, Time-Space trade-off. Abstract Data Types (ADT).

Algorithm:

An algorithm is a step by step procedure to solve a problem. In normal language, the algorithm is defined as a sequence of statements which are used to perform a task. In computer science, an algorithm can be defined as follows...

An algorithm is a sequence of unambiguous instructions used for solving a problem, which can be implemented (as a program) on a computer.

Algorithms are used to convert our problem solution into step by step statements. These statements can be converted into computer programming instructions which form a program. This program is executed by a computer to produce a solution. Here, the program takes required data as input, processes data according to the program instructions and finally produces a result as shown in the following picture.



Specifications of Algorithms

Every algorithm must satisfy the following specifications...

1. **Input -** Every algorithm must take zero or more number of input values from external.
2. **Output -** Every algorithm must produce an output as a result.
3. **Definiteness -** Every statement/instruction in an algorithm must be clear and unambiguous (only one interpretation).
4. **Finiteness -** For all different cases, the algorithm must produce a result within a finite number of steps.
5. **Effectiveness -** Every instruction must be basic enough to be carried out and it also must be feasible.

Example of an Algorithm

Let us consider the following problem for finding the largest value in a given list of values. **Problem Statement:** Find the largest number in the given list of numbers? **Input:** A list of positive integer numbers. (List must contain at least one number). **Output:** The largest number in the given list of positive integer numbers.

Consider the given list of numbers as 'L' (input) and the largest number as 'max' (Output).

Algorithm

1. **Step 1:** Define a variable 'max' and initialize with '0'.
2. **Step 2:** Compare first number (say 'x') in the list 'L' with 'max', if 'x' is larger than 'max', set 'max' to 'x'.
3. **Step 3:** Repeat step 2 for all numbers in the list 'L'.
4. **Step 4:** Display the value of 'max' as a result.

Code using C Programming Language

```
int findMax(L)
{
   int max = 0,i;
   for(i=0; i < listSize; i++)
   {
      if(L[i] > max)
          max = L[i];
   }
   return max;
}
```

Efficiency of an Algorithm:

A measure of the average execution time necessary for an **algorithm** to complete work on a set of data. **Algorithm efficiency** is characterized by its order. Typically a bubble sort **algorithm** will have **efficiency** in sorting N items proportional to and of the order of N^2, usually written $O(N^2)$

Time and Space Complexity:

There are two main complexity measures of the efficiency of an algorithm:

- ***Time complexity*** is a function describing the amount of time an algorithm takes in terms of the amount of input to the algorithm. "Time" can mean the number of memory accesses performed, the number of comparisons between integers, the number of times some inner loop is executed, or some other natural unit related to the amount of real-time the algorithm will take. We try to keep this idea of time separate from "wall clock" time since many factors unrelated to the algorithm itself can affect the real-time (like the language used, type of computing hardware, proficiency of the

program, optimization in the compiler, etc.). It turns out that, if we chose the units wisely, all of the other stuff doesn't matter and we can get an independent measure of the efficiency of the algorithm.

- ***Space complexity*** is a function describing the amount of memory (space) an algorithm takes in terms of the amount of input to the algorithm. We often speak of "extra" memory needed, not counting the memory needed to store the input itself. Again, we use natural (but fixed-length) units to measure this. We can use bytes, but it's easier to use, say, number of integers used, number of fixed-sized structures, etc. In the end, the function we come up with will be independent of the actual number of bytes needed to represent the unit. Space complexity is sometimes ignored because the space used is minimal and/or obvious, but sometimes it becomes as important an issue as time.

Asymptotic notations: Big Oh, Big Theta and Big Omega

Whenever we want to perform an analysis of an algorithm, we need to calculate the complexity of that algorithm. But when we calculate the complexity of an algorithm it does not provide the exact amount of resource required. So instead of taking the exact amount of resource, we represent that complexity in a general form (Notation) which produces the basic nature of that algorithm. We use that general form (Notation) for the analysis process.

Asymptotic notation of an algorithm is a mathematical representation of its complexity.

Majorly, we use three types of asymptotic notations and those are as follows:

1. **Big - Oh (O)**
2. **Big - Omega (Ω)**
3. **Big - Theta (Θ)**

Big - Oh Notation (O)

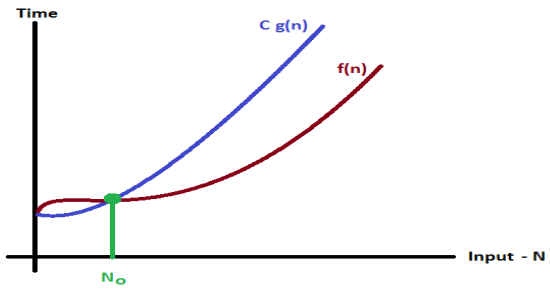
Big - Oh notation is used to define the **upper bound** of an algorithm in terms of Time Complexity.
That means Big - Oh notation always indicates the maximum time required by an algorithm for all input values.

That means Big - Oh notation describes the worst case of an algorithm time complexity. Big - Oh Notation can be defined as follows.

Consider function f(n) as time complexity of an algorithm and g(n) is the most significant term. If f(n) <= C g(n) for all n >= n_0, C > 0 and n_0 >= 1. Then we can represent f(n) as O(g(n)).

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

Consider the following graph drawn for the values of f(n) and C g(n) for input (n) value on X-Axis and time required is on Y-Axis



In above graph after a particular input value n_0, always C g(n) is greater than f(n) which indicates the algorithm's upper bound.

Example

Consider the following f(n) and g(n).
f(n) = 3n + 2
g(n) = n
If we want to represent **f(n)** as **O(g(n))** then it must satisfy **f(n) <= C g(n)** for all values of **C > 0** and **n_0>= 1**
f(n) <= C g(n)
⇒3n + 2 <= C n
Above condition is always TRUE for all values of **C = 4** and **n >= 2**.
By using Big - Oh notation we can represent the time complexity as follows...
3n + 2 = O(n)

Big - Omega Notation (Ω)

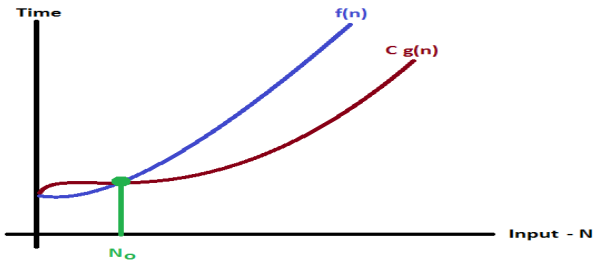
Big - Omega notation is used to define the **lower bound** of an algorithm in terms of Time Complexity.
That means Big-Omega notation always indicates the minimum time required by an algorithm for all input values.

That means Big-Omega notation describes the best case of an algorithm time complexity. Big - Omega Notation can be defined as follows:

Consider function f(n) as time complexity of an algorithm and g(n) is the most significant term. If f(n) >= C g(n) for all n >= n0, C > 0 and n0 >= 1. Then we can represent f(n) as Ω(g(n)).

f(n) = Ω(g(n))

Consider the following graph drawn for the values of f(n) and C g(n) for input (n) value on X-Axis and time required is on Y-Axis



In above graph after a particular input value n_0, always C g(n) is less than f(n) which indicates the algorithm's lower bound.

Example

Consider the following f(n) and g(n)...

f(n) = 3n + 2

g(n) = n

If we want to represent **f(n)** as **Ω(g(n))** then it must satisfy **f(n) >= C g(n)** for all values of **C > 0** and **n_0>= 1**

f(n) >= C g(n)

⇒3n + 2 >= C n

Above condition is always TRUE for all values of **C = 1** and **n >= 1**.

By using Big-Omega notation we can represent the time complexity as follows...

3n + 2 = Ω(n)

Big - Theta Notation (Θ)

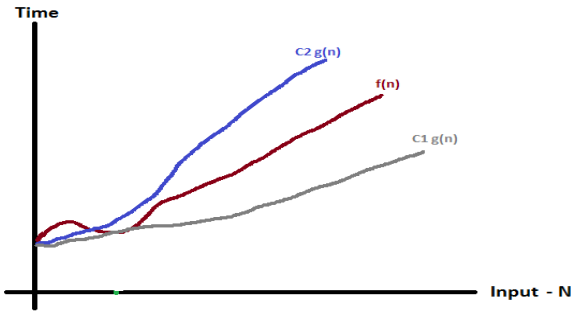
Big - Theta notation is used to define the **average bound** of an algorithm in terms of Time Complexity. That means Big - Theta notation always indicates the average time required by an algorithm for all input values.

That means Big - Theta notation describes the average case of an algorithm time complexity. Big - Theta Notation can be defined as follows:

Consider function f(n) as time complexity of an algorithm and g(n) is the most significant term. If C1 g(n) <= f(n) <= C2 g(n) for all n >= n0, C1 > 0, C2 > 0 and n0 >= 1. Then we can represent f(n) as Θ(g(n)).

f(n) = Θ(g(n))

Consider the following graph drawn for the values of f(n) and C g(n) for input (n) value on X-Axis and time required is on Y-Axis



In above graph after a particular input value n_0, always C_1 g(n) is less than f(n) and C_2 g(n) is greater than f(n) which indicates the algorithm's average bound.

Example

Consider the following f(n) and g(n)...

f(n) = 3n + 2

g(n) = n

If we want to represent **f(n)** as **Θ(g(n))** then it must satisfy **C_1 g(n) <= f(n) <= C_2 g(n)** for all values of **C_1 > 0, C_2 > 0** and **n_0>= 1**

C_1 g(n) <= f(n) <= C_2 g(n)

⇒C_1 n <= 3n + 2 <= C_2 n

Above condition is always TRUE for all values of **C_1 = 1, C_2 = 4** and **n >= 2**.

By using Big - Theta notation we can represent the time complexity as follows...

3n + 2 = Θ(n)

Common Asymptotic Notations

Following is a list of some common asymptotic notations:

Constant	–	O(1)
Logarithmic	–	O(log n)
Linear	–	O(n)
n log n	–	O(n log n)
Quadratic	–	$O(n^2)$
Cubic	–	$O(n^3)$
Polynomial	–	$n^{O(1)}$
Exponential	–	$2^{O(n)}$

Time-Space Trade-off:

The best algorithm to solve a given problem is one that requires less space in memory and takes less time to complete its execution. But in practice, it is not always possible to achieve both these objectives. As we know there may be more than one approach to solve a particular problem. One approach may take more space but takes less time to complete its execution while the other approach may take less space but takes more time to complete its execution. We may have to sacrifice one at the cost of the other. If space is our constraint, then we have to choose a program that requires less space at the cost of more execution time. On the other hand, if time is our constraint then we have to choose a program that takes less time to complete its execution at the cost of more space. That is what we can say that there exists a time-space tradeoff among algorithms.

Abstract Data Types (ADT):

Abstract data type (ADT) is a mathematical model with a collection of operations defined on that model. The abstract data type encapsulates a data type in the sense that the definition of the type and all operations can be localized and are not visible to the user of the ADT. To the user the declaration of the ADT and its operations are important. Implementation of abstract data type is the translation into statements of programming language which chooses the data structure to represent the abstract data type. For example class, structures, union, enumerated data types are abstract data types.

ADT (Abstract Data Types) are the data types which are made up of or composed of primitive data types as these ADT can be application or implementation specific, they could be created on a need basis.

Here are some examples.

- stack: operations are "push an item onto the stack", "pop an item from the stack", "ask if the stack is empty"; an implementation may be as an array, linked list, etc.
- queue: operations are "add to the end of the queue", "delete from the beginning of the queue", "ask if the queue is empty"; an implementation may be as an array or linked list or heap.
- search structure: operations are "insert an item", "ask if an item is in the structure", and "delete an item"; an implementation may be as an array, linked list, tree, a hash table.