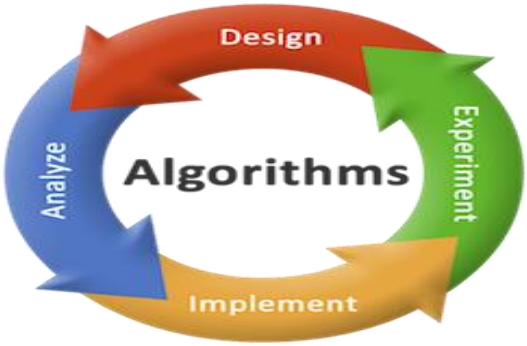
Course Code: CSE 235

Course Title: Design and Analysis of Algorithms

What is Algorithm?

Algorithm is a **step-by-step procedure**, which defines **a set of instructions to be executed in a certain order to get the desired output**. Algorithms are generally created **independent of underlying languages**, i.e., an algorithm can be implemented in more than one programming language.



From the **data structure point of view**, following are **some important categories of algorithms:**

- **Search** − Algorithm to search an item in a data structure.
- **Sort** − Algorithm to sort items in a certain order.
- **Insert** − Algorithm to insert item in a data structure.
- **Update** − Algorithm to update an existing item in a data structure.
- **Delete** − Algorithm to delete an existing item from a data structure.

Why Study Algorithm?

Algorithms **play a crucial role in computer science.** The **best algorithm ensures that the computer completes the task in the most efficient manner.** When it comes to efficiency, a good algorithm is really essential. An algorithm is **extremely important for optimizing a computer program.** The **reasons why we study algorithm** is given below:

(i) Time is precious

Suppose, Alice and Bob are trying to solve a simple problem of **finding the sum of the first 10^{11} natural numbers.** While **Bob was writing the algorithm, Alice implemented it.**

Algorithm (by Bob)

```
Initialize sum = 0
for every natural number n in range 1 to 10^11 (inclusive):
   add n to sum
sum is your answer
```

Code (by Alice)

```
int findSum() {

   int sum = 0;

   for (int v = 1; v <= 100000000000; v++) {

      sum += v;

   }

   return sum;

}
```

Alice and Bob are feeling euphoric of themselves that they could build something of their own in almost no time. Let's sneak into their workspace and listen to their conversation.

Alice: Let's run this code and find out the sum.

Bob: I ran this code a few minutes back but it's still not showing the output. What's wrong with it?

So, **something went wrong!** A computer is the most deterministic machine. Going back and trying to run it again won't help. So, let's **analyze what's wrong with this simple code.**

Two of the **most valuable resources** for a computer program are **time** and **memory**. The **time taken by the computer to run code** is:

Time to run code = Total number of instructions to be executed / Number of instructions executed per second (subject to unit)

The **number of instructions depends on the code** you used, and the **time taken to execute each code depends on your machine and compiler.**

In this case, the **total number of instructions executed** (let's say x) are:

$$x = 1 + (10^{11} + 1) + (10^{11}) + 1$$
$$= 2 * 10^{11} + 3$$

Let us **assume that a computer can execute** $y = 10^8$ **instructions in one second** (it can vary subject to machine configuration).

The **time taken to run above code** is:

Time taken to run code = x/y (greater than 16 minutes)

Is it possible to optimize the algorithm so that Alice and Bob do not have to wait for 16 minutes every time they run this code?

Now, you may already guess the right method. The **sum of first N natural numbers is given by the formula:**
Converting it into code will look something like this:

Sum = N * (N + 1) / 2

```
int sum(int N) {
    return N * (N + 1) / 2;
}
```

This **code executes in just one instruction** and **gets the task done no matter what the value is.** Let it be greater than the total number of atoms in the universe. It will find the result in no time.

The **time taken to solve the problem, in this case, is 1/y (which is 10 nanoseconds).** By the way, the fusion reaction of a hydrogen bomb takes 40-50 ns, which means your program will complete successfully even if someone throws a hydrogen bomb on your computer at the same time you ran your code.

(ii) More on Scalability

Scalability is **scale plus ability**, which means the **quality of an algorithm/system to handle the problem of larger size.**

Consider the problem of **setting up a classroom of 50 students.** One of the simplest solutions is to **book a room, get a whiteboard, a few marker pens, and the problem is solved.**

But what if the size of the problem increases? What if the **number of students increased to 200?** The solution still holds but it **needs more resources.** In this case, you will probably **need a much larger room (probably a theatre), a projector screen and a digital pen.** Again, **what if the number of students increased to 1000?** The **solution fails or uses a lot of resources** when the size of the problem increases.

This means, your **solution wasn't scalable. What is a scalable solution then?**

Consider a site like **Khanacademy**, **millions of students can see videos, read answers at the same time and no more resources are required.** So, the **solution can solve the problems of larger size under resource crunch.**

If you see our **first solution** to find the sum of first N natural numbers, it **wasn't scalable.** It's because it **required linear growth in time with the linear growth in the size of the problem.** Such algorithms are also known as **linearly scalable algorithms.**

Our **second solution was very scalable** and didn't require the use of any more time to solve a problem of larger size. These are known as **constant-time algorithms.**

(iii) Memory is Expensive

Memory is **not always available in abundance.** While dealing with **code/system** which **requires you to store or produce a lot of data**, it is **critical** for your algorithm to **save the usage of memory** wherever possible. **For example:** While **storing data about people**, you **can save memory by storing only their date of birth, not their age.** You can always calculate it on the fly using their date of birth and current date.

Use of the Algorithms

Algorithms play a crucial role in various fields and have many applications. Some of the key areas where algorithms are used include:

1. **Computer Science:** Algorithms form the basis of computer programming and are used to solve problems ranging from simple sorting and searching to complex tasks such as artificial intelligence and machine learning.

2. **Mathematics:** Algorithms are used to solve mathematical problems, such as finding the optimal solution to a system of linear equations or finding the shortest path in a graph.

3. **Operations Research:** Algorithms are used to optimize and make decisions in fields such as transportation, logistics, and resource allocation.

4. **Data Science:** Algorithms are used to analyze, process, and extract insights from large amounts of data in fields such as marketing, finance, and healthcare.

Where are Algorithms Used in Computer Science?

Algorithms are **used in every part of computer science.** They form the field's backbone. In computer science, an **algorithm gives the computer a specific set of instructions,** which **allows the computer to do everything, be it running a calculator or running a rocket. Computer programs** are, at **their core, algorithms written** in programming languages that the computer can understand. Computer algorithms **play a big role in how social media works:** which **posts show up, which ads are seen,** and so on. These decisions are all made by algorithms. **Google's programmers use algorithms to optimize searches, predict what users are going to type, and more.**

Characteristics of an Algorithm

Not all procedures can be called an **algorithm.** An algorithm **should have the following characteristics:**

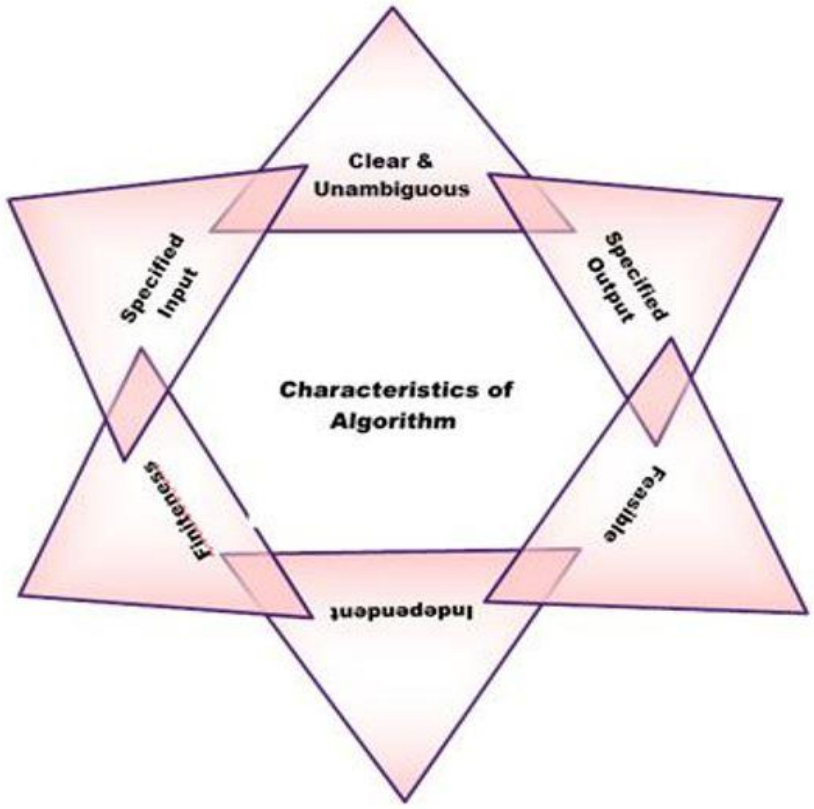


Figure: Characteristics of an Algorithm

- **Clear and Unambiguous**: Algorithm should be clear and unambiguous. Each of its steps (or phases), and their inputs/outputs should be clear and must lead to only one meaning.
- **Specified Input**: An algorithm should have 0 or more well-defined inputs.
- **Specified Output**: An algorithm should have 1 or more well-defined outputs, and should match the desired output.
- **Finiteness**: Algorithms must terminate after a finite number of steps.
- **Feasible**: The algorithm must be simple, general, and practical, and it must be able to be run with the resources available. It should not incorporate any futuristic technology.
- **Independent**: An algorithm should have step-by-step directions, which should be independent of any programming code.

How to Write an Algorithm?

There are **no well-defined standards for writing algorithms.** Rather, it is **problem and resource dependent.** Algorithms are **never written to support a particular programming code.**

As we know that **all programming languages share basic code constructs like loops (do, for, while), flow-control (if-else),** etc. **These common constructs can be used to write an algorithm.**

We write algorithms in a **step-by-step manner,** but it is not always the case. Algorithm writing is a process and is **executed after the problem domain is well-defined.** That is, we **should know the problem domain, for which we are designing a solution.**

Example

Let's try to learn algorithm-writing by using an example.

Problem − Design an **algorithm to add two numbers and display the result.**

Step 1 − START
Step 2 − declare three integers **a**, **b** & **c**
Step 3 − define values of **a** & **b**
Step 4 − add values of **a** & **b**
Step 5 − store output of step 4 to **c**
Step 6 − print **c**
Step 7 − STOP

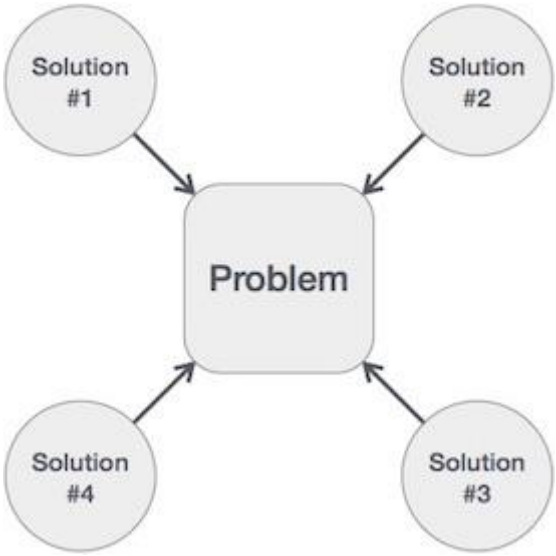
Algorithms tell the programmers how to code the program. **Alternatively, the algorithm can be written as:**

Step 1 − START ADD
Step 2 − get values of **a** & **b**
Step 3 − c ← a + b
Step 4 − display c
Step 5 − STOP

In design and analysis of algorithms, **usually the second method is used to describe an algorithm.** It makes it **easy for the analyst to analyze the algorithm ignoring all unwanted definitions.** He can **observe what operations are being used and how the process is flowing.**

Writing **step numbers**, is **optional.**

We design an algorithm to **get a solution of a given problem.** A **problem can be solved in more than one way.**



Hence, **many solution algorithms can be derived for a given problem.** The **next step is to analyze those proposed solution algorithms and implement the best suitable solution.**

Algorithm Analysis

Efficiency of an algorithm can be **analyzed at two different stages**, **before implementation and after implementation.** They are the following:

- ***A Priori* Analysis**: This is a **theoretical analysis** of an algorithm. Efficiency of an algorithm is **measured by assuming that all other factors, for example, processor speed, memory etc. are constant and have no effect on the implementation.**
- ***A Posterior* Analysis**: This is an **empirical analysis** of an algorithm. The **selected algorithm is implemented using programming language.** This is **then executed on target computer machine.** In this analysis, **actual statistics like running time and space required, are collected.**

Algorithm Complexity

Suppose **X** is an **algorithm** and **n** are the **size of input data**. The **time and space** used by the algorithm X are the **two main factors**, which **decide the efficiency of X.**

- **Time Factor**: Time is **measured by counting the number of key operations** such as comparisons in the sorting algorithm.
- **Space Factor**: Space is **measured by counting the maximum memory space required** by the algorithm.

The **complexity of an algorithm f(n)** gives the **running time and the storage space required by the algorithm** in terms of **n** as the size of input data.

(i) Space Complexity

Space complexity of an algorithm **represents the amount of memory space required by the algorithm in its life cycle.** The **space required by an algorithm** is **equal to the sum of the following two components**:

- A **fixed part** that is a space **required to store certain data and variables** that are **independent of the size of the problem. For example,** simple variables and **constants used, program size**, etc.
- A **variable part** is a space **required by variables, whose size depends on the size of the problem. For example,** dynamic memory allocation, recursion stack space, etc.

Space complexity S(P) of any algorithm P is:

S(P) = C + SP(I), where **C is the fixed part** and **SP(I) is the variable part** of the algorithm, which depends on instance characteristic I.

Following is a **simple example** that tries to explain the concept.

```
Algorithm: SUM (A, B)
Step 1 - START
Step 2 - C ← A + B
Step 3 - Stop
```

Here we have **three variables A, B, and C and one constant.** Hence,

S(P) = 1 + 3

Now, **space depends** on **data types** of given variables and **constant types** and it will be **multiplied accordingly.**

Language C compiler takes the following space:

Type	Size
bool, char, unsigned char, signed char	1 byte
int, unsigned int, signed int, unsigned short int, signed short int	2 bytes
float, unsigned long int, signed long int	4 bytes
double	8 bytes
long double	10 bytes

Example 1: Addition of Numbers

```
{
  float a = x + y + z;
  return (a);
}
```

Answer: In the above example, there are **4 float variables** those are **a, x, y, z.** So, they will take **4 bytes** (as given in the table above) space for each variable. **An extra 4-byte** space will also be added to the total space complexity for the **return value** that is a.

Hence, the **total space complexity = 4*4 + 4 = 20 bytes**

(ii) Time Complexity

Time complexity of an algorithm **represents the amount of time required by the algorithm to run to completion. Time requirements** can be defined as a numerical function **T(n)**, where T(n) can be **measured as the number of steps, provided each step consumes constant time.**

For example, addition of two **n-bit integers** takes **n** steps. Consequently, the total **computational time** is:

T(n) = c ∗ n, where **c is the time taken for the addition of two bits.**

Here, we observe that **T(n) grows linearly as the input size increases.**

Typical Complexities of an Algorithm

- Constant Complexity: It imposes a complexity of O(1). It undergoes an execution of a constant number of steps like 1, 5, 10, etc. for solving a given problem. The count of operations is independent of the input data size.
- Logarithmic Complexity: It imposes a complexity of O(log(N)). It undergoes the execution of the order of log(N) steps. To perform operations on N elements, it often takes the logarithmic base as 2. For N = 1,000,000, an algorithm that has a complexity of O(log(N)) would undergo 20 steps (with a constant precision).

- Linear Complexity: It imposes a complexity of O(N). It encompasses the same number of steps as that of the total number of elements to implement an operation on N elements.
- For example, if there exist 500 elements, then it will take about 500 steps.
- Quadratic Complexity: It imposes a complexity of O(n^2). For N input data size, it undergoes the order of N^2 count of operations on N number of elements for solving a given problem.
- If N = 100, it will endure 10,000 steps.
- Cubic Complexity: It imposes a complexity of O(n^3). For N input data size, it executes the order of N^3 steps on N elements to solve a given problem. For example, if there exist 100 elements, it is going to execute 1,000,000 steps.
- Exponential Complexity: It imposes a complexity of O(2^n), O(N!), O(n^k), …. For N elements, it will execute the order of count of operations that is exponentially dependable on the input data size.
- For example, if N = 10, then the exponential function 2^N will result in 1024. Similarly, if N = 20, it will result in 1048 576, and if N = 100, it will result in a number having 30 digits. The exponential function N! grows even faster; for example, if N = 5 will result in 120. Likewise, if N = 10, it will result in 3,628,800 and so on.

How to Approximate the Time Taken by the Algorithm?

So, to find it out, we shall **first understand the types of the algorithm** we have. There are **two types of algorithms**. These are:

1. **Iterative Algorithm:** In the iterative approach, the **function repeatedly runs until the condition is met or it fails**. It involves the looping construct.
2. **Recursive Algorithm:** In the recursive approach, the **function calls itself until the condition is met**. It integrates the branching structure.

However, it is worth noting that **any program that is written in iteration could be written as recursion.** Likewise, a **recursive program can be converted to iteration**, making both of these **algorithm's equivalent to each other.**

But to **analyze the iterative program**, we have to **count the number of times the loop is going to execute**, whereas in the **recursive program**, we **use recursive equations**.

Suppose the **program is neither iterative nor recursive**. In that case, it can be concluded that there is **no dependency of the running time on the input data size**, i.e., whatever is the input size, the **running time is going to be a constant value**. Thus, for such programs, the **complexity will be O(1)**.

Consider the following programs that are written in simple English and does not correspond to any syntax.

Example1:

In the first example, we have an **integer i** and a **for loop running from i equals 1 to n.** Now the question arises, **how many times does the name get printed?**

```
1. A()
2. {
3. int i;
4. for (i=1 to n)
5. printf("Edward");
6. }
```

Since i equals 1 to n, so the above **program will print Edward, n number of times.** Thus, the **complexity will be O(n)**.

Example2:

```
1. A()
2. {
3. int i, j:
4. for (i=1 to n)
5. for (j=1 to n)
6. printf("Edward");
7. }
```

In this case, firstly, the **outer loop will run n times,** such that for each time, the **inner loop will also run n times**. Thus, the time **complexity will be O(n^2)**.

Algorithm Design Techniques

The following is a list of several popular design approaches:

1. Divide and Conquer Approach: It is a top-down approach. The algorithms which follow the divide & conquer techniques **involve three steps:**

- Divide the original problem into a set of subproblems.

- Solve every subproblem individually, recursively.
- Combine the solution of the subproblems (top level) into a solution of the whole original problem.

2. Greedy Technique: Greedy method is **used to solve the optimization problem.** An **optimization problem** is one in which we are **given a set of input values,** which are **required either to be maximized or minimized** (known as objective), i.e., some constraints or conditions.

- Greedy Algorithm **always makes the choice (greedy criteria) looks best at the moment,** to optimize a given objective.
- The greedy algorithm **doesn't always guarantee the optimal solution** however it generally produces a solution that is very close in value to the optimal.

3. Dynamic Programming: Dynamic Programming is a bottom-up approach which **solve all possible small problems and then combine them to obtain solutions for bigger problems.** This is particularly **helpful when the number of copying subproblems is exponentially large.** Dynamic Programming is frequently related to **Optimization Problems**.

4. Backtracking Algorithm: Backtracking Algorithm **tries each possibility until they find the right one.** It is a depth-first search of the set of possible solution. During the search, **if an alternative doesn't work, then backtrack to the choice point, the place which presented different alternatives, and tries the next alternative.**

5. Randomized Algorithm: A randomized algorithm **uses a random number at least once during the computation make a decision. For example,** in **Quick Sort,** using **a random number to choose a pivot.**

Asymptotic Analysis

Asymptotic analysis of an algorithm refers to **defining the mathematical foundation/framing of its run-time performance. Using asymptotic analysis,** we can very well **conclude the best case, average case, and worst case scenario of an algorithm.** Usually, the **time required** by an algorithm **falls under three types:**

- **Best Case**: **Minimum time** required for program execution.
- **Average Case**: **Average time** required for program execution.

- **Worst Case**: **Maximum time** required for program execution.

.

Asymptotic analysis is **input bound** i.e., if **there's no input** to the algorithm, it is concluded to work in a **constant time.**

Asymptotic analysis **refers to computing the running time of any operation in mathematical units** of computation. For example, the **running time** of one operation is **computed as *f*(n)** and may be for another operation it is **computed as $g(n^2)$.** This means the **first operation running time will increase linearly** with the increase in **n** and the running time of the **second operation will increase exponentially** when **n** increases.

Asymptotic Notations

Following are the **commonly used asymptotic notations** to calculate the running time complexity of an algorithm.

- O Notation
- Ω Notation
- θ Notation

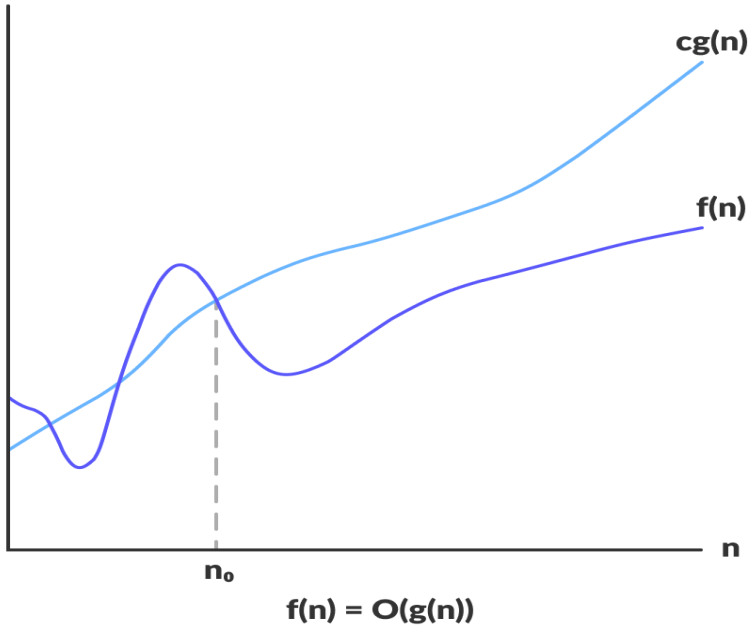
Big-O Notation (O-notation)

Big-O notation **represents the upper bound** of the running time of an algorithm. Thus, it gives the **worst-case complexity** of an algorithm.

Big-O gives the upper bound of a function.

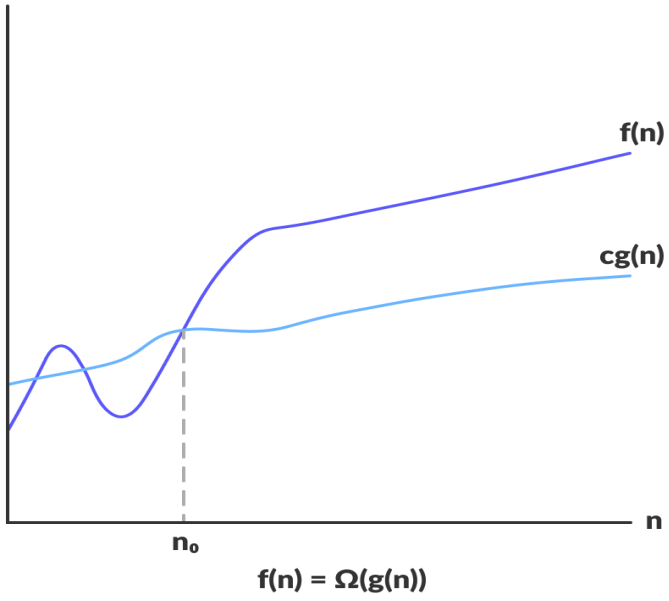
$$O(g(n)) = \{ f(n): \text{there exist positive constants } c \text{ and } n_0 \text{ such that } 0 \leq f(n) \leq cg(n) \text{ for all } n \geq n_0 \}$$

The above expression can be described as a **function f(n) belongs to the set O(g(n))** if there exists a **positive constant c such that it lies between 0 and cg(n),** for sufficiently large n.
For **any value of n,** the **running time** of an algorithm **does not cross the time provided by O(g(n)).**
Since it gives the worst-case running time of an algorithm, it is **widely used to analyze an algorithm** as we are always interested in the worst-case scenario.



Omega Notation (Ω-notation)

Omega notation **represents the lower bound** of the running time of an algorithm. Thus, it **provides the best case complexity** of an algorithm.



Omega gives the lower bound of a function.

$\Omega(g(n)) = \{ f(n)$: there exist positive constants c and n_0 such that $0 \leq cg(n) \leq f(n)$ for all $n \geq n_0 \}$

The above expression can be described as a **function f(n) belongs to the set Ω(g(n))** if there exists a **positive constant c such that it lies above cg(n),** for sufficiently large n.
For **any value of n,** the **minimum time required** by the algorithm is given by Omega Ω(g(n)).

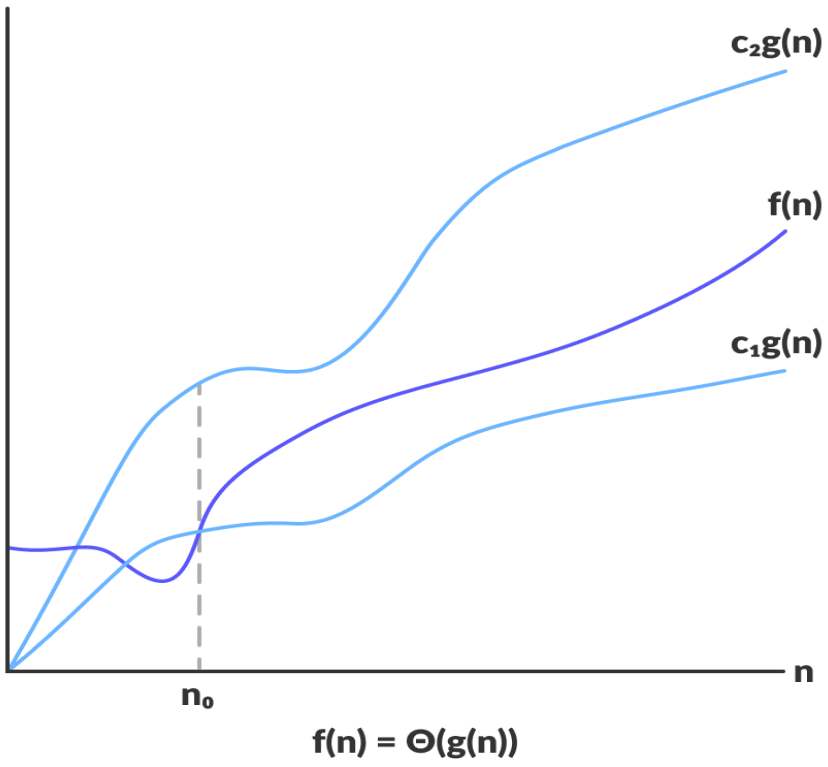
Theta Notation (Θ-notation)

Theta notation **encloses the function from above and below.** Since it **represents the upper and the lower bound** of the running time of an algorithm, it is used for **analyzing the average-case complexity** of an algorithm.

For a function g(n), Θ(g(n)) is given by the relation:

$$\Theta(g(n)) = \{ f(n): \text{there exist positive constants } c_1, c_2 \text{ and } n_0 \text{ such that } 0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n) \text{ for all } n \geq n_0 \}$$

The above expression can be described as a **function f(n) belongs to the set Θ(g(n))** if there exist **positive constants c_1 and c_2 such that it can be sandwiched between c_1g(n) and c_2g(n),** for sufficiently large n.
If a function f(n) lies anywhere in between c_1g(n) and c_2g(n) for all n ≥ n0, then f(n) is said to be asymptotically tight bound.



f(n) = Θ(g(n))