Maharaja Education Trust ®
MIT First Grade College
Department of Computer Applications
Industrial Suburb, Manandavadi Road, Mysuru -570008, www.mitfgc.in
Affiliated to University of Mysore, Accredited by NAAC with "A" Grade, ISO 9001:2015 Certified Institution

VISION OF THE INSTITUTE

Empower the individuals and society at large through educational excellence; sensitize them for a life dedicated to the service of fellow human beings and mother land.

MISSION OF THE INSTITUTE

To impact holistic education that enables the students to become socially responsive and useful, with roots firm on traditional and cultural values; and to hone their skills to accept challenges and respond to opportunities in a global scenario.

Design and Analysis of algorithm

Course Code: DSC501 - Theory

Total Contact Hours: 4 Hours/wk

Formative Assessment Marks: 25

Exam Marks: 25

Exam Duration: 03 Hrs

Prepared by:

ARVIND G

HOD Department of Computer Application

Program Name	**BCA**	Semester	**V**
Course Title	**Design and Analysis of Algorithm (Theory)**		
Course Code:	**DSC13**	No. of Credits	**04**
Contact hours	**52 Hours**	Duration of SEA/Exam	**2 hours**
Formative Assessment Marks	**40**	Summative Assessment Marks	**60**

Course Outcomes (COs): After the successful completion of the course, the student will be able to: CO1. Understand the fundamental concepts of algorithms and their complexity, including time and space complexity, worst-case and average-case analysis, and Big-O notation. BL (L1, L2)
CO2. Design algorithms for solving various types of problems, such as Sorting, Searching, Graph
traversal, Decrease-and-Conquer, Divide-and-Conquer and Greedy Techniques. BL (L1, L2, L3) CO3. Analyze and compare the time and space complexity of algorithms with other algorithmic techniques. BL (L1, L2,L3,L4)
CO4. Evaluate the performance of Sorting, Searching, Graph traversal, Decrease-and-Conquer, Divide-and-Conquer and Greedy Techniques using empirical testing and benchmarking, and identify their limitations and potential improvements. BL (L1, L2, L3, L4)
CO5. Apply various algorithm design to real-world problems and evaluate their effectiveness and efficiency in solving them. BL (L1, L2, L3)
Note: Blooms Level(BL): L1=Remember, L2=Understand, L3=Apply, L4=Analyze, L5= Evaluate, L6= Create

Contents	**52 Hrs**
Introduction: What is an Algorithm? Fundamentals of Algorithmic problem solving, Fundamentals of the Analysis of Algorithm Efficiency, Analysis Framework, Measuring the input size, Units for measuring Running time, Orders of Growth, Worst-case, Best-case and Average-case efficiencies.	10
Asymptotic Notations and Basic Efficiency classes, Informal Introduction, O-notation, Ω-notation, θ-notation, mathematical analysis of non-recursive algorithms, mathematical analysis of recursive algorithms.	10
Brute Force & Exhaustive Search: Introduction to Brute Force approach, Selection Sort and Bubble Sort, Sequential search, Exhaustive Search- Travelling Salesman Problem and Knapsack Problem, Depth First Search, Breadth First Search	11
Decrease-and-Conquer: Introduction, Insertion Sort, Topological Sorting **Divide-and-Conquer:** Introduction, Merge Sort, Quick Sort, Binary Search, Binary Tree traversals and related properties.	11
Greedy Technique: Introduction, Prim's Algorithm, Kruskal's Algorithm, Dijkstra's Algorithm, Lower-Bound Arguments, Decision Trees, P Problems, NP Problems, NP- Complete Problems, Challenges of Numerical Algorithms.	10

UNIT – 1

Introduction: What is an Algorithm? Fundamentals of Algorithmic problem solving, Fundamentals of the Analysis of Algorithm Efficiency, Analysis Framework, Measuring the input size, Units for measuring Running time, Orders of Growth, Worst-case, Best case and Average-case efficiencies.

INTRODUCTION

What is an algorithm?

Definition: An algorithm is defined as finite sequence of unambiguous instructions followed to accomplish a given task. It is also defined as unambiguous, step by step procedure (instructions) to solve a given problem in finite number of steps by accepting a set of inputs and producing the desired output. After producing the result, the algorithm should terminate. The notion of an algorithm is pictorially represented as shown below:

$$\begin{tikzcd}
 & & & \text{Input} \arrow[d] & \\
\text{Problem} \arrow[r] & \text{Algorithm} \arrow[r] & \text{Program} \arrow[r] & \text{Computer} \arrow[r] & \text{Output}
\end{tikzcd}$$

Observe the following activities to see how an algorithm is used to produce the desired result:

- The solution to a given problem is expressed in the form of an algorithm.
- The algorithm is converted into a program.
- The program when it is executed, accept the input and produces the desired output

The properties of an algorithm?" An algorithm must satisfy the

• **Input:** Each algorithm should have zero or more inputs. The range of inputs for which algorithm works should be satisfied

• **Output:** The algorithm should produce correct results. At least one output has to be produced.

• **Definiteness:** Each instruction should be clear and unambiguous.

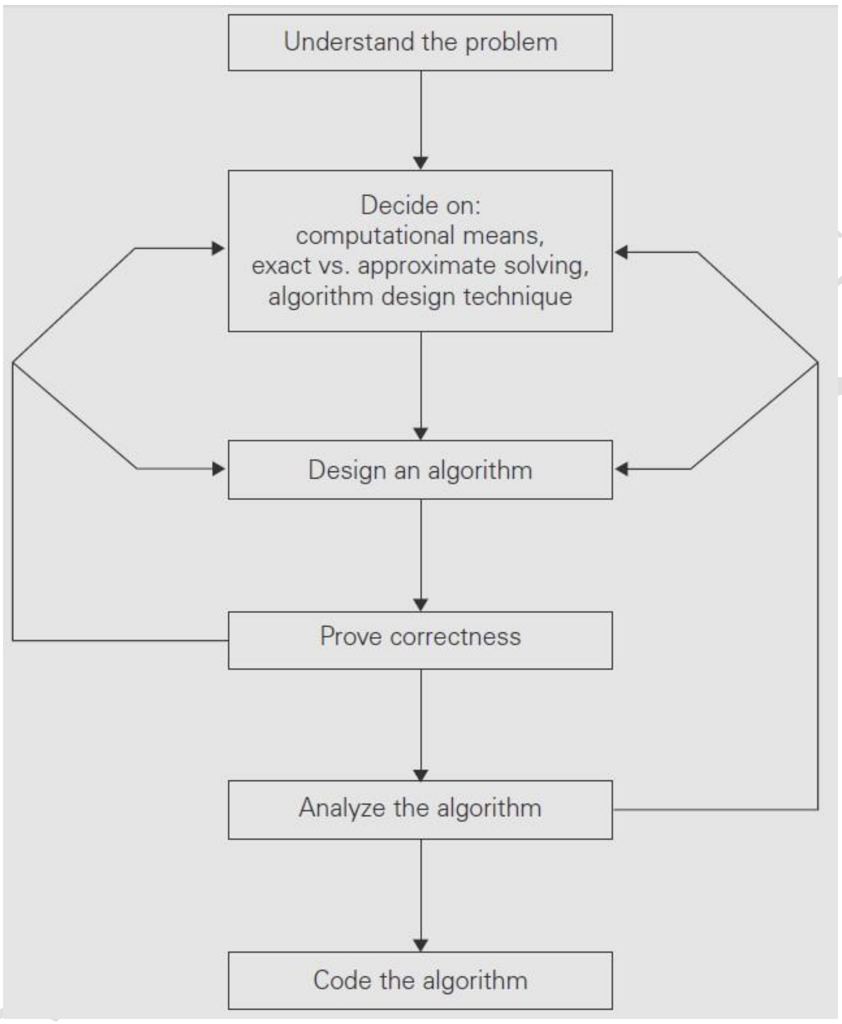
• **Effectiveness**: The instructions should be simple and should transform the given input to the desired output.

• **Finiteness:** The algorithm must terminate after a finite sequence of instructions.

Note: By looking at the algorithm, the programmer can write the program in C or C++ or any of the programming language. Before writing any program, the solution has to be expressed in the form of algorithms.

Algorithm design and analysis process

A sequence of steps involved in designing and analyzing an algorithm is shown inthe figure below.**:**



- Understand the problem
- Decide on Computational Device Exact Vs Approximate Algorithms
- Algorithm Design Techniques
- Design an algorithms
- Prove Correctness
- Analyze the Algorithm
- Code the Algorithm

Understanding the Problem:

Begin by clearly understanding the problem you are trying to solve. Identify the input, output, and constraints. This step is crucial for defining the scope of the algorithm.

Deciding on Computational Device:

Consider the computational device or model on which the algorithm will run. Different devices may have different resource constraints, and the algorithm should be designed to work efficiently within those constraints.

Exact Vs. Approximate Algorithms:

Decide whether an exact solution is necessary or if an approximate solution would suffice. Exact algorithms guarantee optimal solutions, while approximate algorithms provide solutions that are close to optimal but might be obtained faster.

Algorithm Design Techniques:

Choose appropriate algorithm design techniques based on the problem at hand. Common techniques include:

- **Divide and Conquer**: Break the problem into smaller subproblems and solve them independently.
- **Dynamic Programming:** Solve subproblems and store their solutions to avoid redundant computations.
- **Greedy Algorithms:** Make locally optimal choices at each step with the hope of finding a global optimum.

Design an Algorithm:

Develop the step-by-step procedure to solve the problem based on the chosen design technique. Clearly define the algorithm's input, output, and the sequence of operations to achieve the desired result.

Prove Correctness:

Provide a mathematical or logical proof that the algorithm produces the correct output for any valid input. This step ensures the algorithm reliably solves the problem it was designed for.

Analyze the Algorithm:

Assess the algorithm's efficiency, primarily in terms of time and space complexity. Perform a worst-case, best-case, and average-case analysis to understand how the algorithm performs under different scenarios.

Code the Algorithm:

Implement the algorithm in a programming language of choice. Pay attention to details, and ensure that the code accurately reflects the designed algorithm. Test the algorithm with various inputs to validate its correctness and efficiency.

Computing GCD

The notion of an algorithm can be explained by computing the GCD of two numbers. Now, let us see "What is GCD of two numbers?"

Definition: The GCD (short form for Greatest Common Divisor) of two numbers m and n denoted by GCD(m, n) is defined as the largest integer that divides both m and n such that the remainder is zero. GCD of two numbers is defined only for positive integers but, not defined for negative integers and floating point numbers. For example, GCD(10, 30) can be obtained as shown below:

Step 1: The numbers 1, 2, 5, 10 divide 10

Step 2: The numbers 1, 2, 5, 6, 10 and 30 divide 30

Step 3: Observe from step I and step 2 that the numbers 1, 2, 5 and 10 are common divisors of both 10 and 30 and 10 is the greatest number which is common and hence it is called Greatest Common Divisor

So, GCD(10,30) = 10.

Now, let us see "What are the different ways of computing GCD of two numbers?" The GCD of two numbers can be computed using various methods as shown below:

Different ways of computing GCD

- Euclid's algorithm (Using modulus)
- Repetitive subtraction (Euclid's algorithm)
- Consecutive inter checking algorithm
- Middle school procedure using prime factors

Step 2: [Eliminate the multiples of p between 2 to n]

```
for p = 21 sqrt(n)
    if(a * [p]! = 0)                    // Is a[p] is prime? If so proceed
        i← p* p                 // p is the next prime number obtained
        while (i<= n)       // Obtain position of multiples of p
            a[i] ←0         // Multiples of p may exist
            i←i + p                     // Eliminate a[i] which is multiple of p
        End while               // Obtain the position of next multiple of p
    End if
End for
```

Step 3: [Obtain the prime numbers by copying the non-zero elements]

```
j←0
for i←2 to n do
    if(a * [i]! = 0)
        b[j] ← a[i] ;
        j← j + 1
    End if
End for

Step 4: [Output the prime numbers between 2 to n]
for i←0 to j– 1
    Write b[i]
End for

Step 5: [Finished]Exit
```

Units for measuring Running time

The running time of algorithms is often measured using different units depending on the context and the granularity of analysis. Here are common units for measuring running time:

Seconds (s):

- The most straightforward unit, representing the actual time in seconds that an algorithm takes to run on a specific machine.
- Suitable for measuring real-world performance but can be influenced by external factors like the machine's load.

Milliseconds (ms):

- A smaller unit than seconds, measuring time in thousandths of a second.
- Useful for more precise measurement when algorithms have relatively fast execution times.

Microseconds (µs):

- An even smaller unit than milliseconds, measuring time in millionths of a second.
- Appropriate for very fast algorithms or when extremely precise timing is required.

Nanoseconds (ns):

- A unit smaller than microseconds, measuring time in billionths of a second.

- Common in the context of measuring operations at the hardware level or very low-level algorithmic optimizations.

Operations or Basic Steps:

- Instead of measuring time, algorithms can be analyzed based on the number of basic steps or operations they perform.
- This unit provides an abstract measure of algorithmic complexity, independent of the specific hardware or software environment.

Big O Notation (O()):

- Represents the upper bound of an algorithm's time complexity in terms of a mathematical function, typically based on the input size.
- Provides a theoretical measure of efficiency, ignoring constant factors and lower-order terms.

Instruction Count:

- The number of machine instructions executed by an algorithm.
- Useful for low-level analysis and optimization, considering the specific machine architecture.

Comparisons or Swaps:

- For sorting algorithms, measuring the number of element comparisons or swaps provides insight into their efficiency.
- Relevant for algorithms where the primary operations involve comparisons or data rearrangement.

Analysis of algorithms

The main purpose of algorithm analysis is to design most efficient algorithms. Let use "On what factors efficiency of algorithm depends?" The efficiency of analgorithm depends on two factors:

1. Space efficiency
2. Time efficiency

"What is space efficiency?"

Definition: The space efficiency of an algorithm is the amount of memory required to run the program completely and efficiently. If the efficiency is measured with respect to the space (memory required), the word space complexity is often used. The space complexity of an algorithm depends on following factors:

Components that affect space

1. Program space
2. Data space

3. Stack space

- **Program space:** The space required for storing the machine program generated by the compiler or assembler is called program space.
- **Data space**: The space required to store the constants, variables etc., is called data
- **Stack space:** The space required to store the return address along with parameters that are passed to the function, local variables etc., is called stack space.

Note: The new technological innovations have improved the computer's speed and memory size by many orders of magnitude. Now a days, space requirement for an algorithm is not a concern and hence, we are not concentrating on space efficiency. Let us concentrate only on time efficiency..

"What is time efficiency?"

Definition: The time efficiency of an algorithm is measured purely on how fast a given algorithm is executed. Since the efficiency of an algorithm is measured using time, the word time complexity is often associated with an algorithm.

The time efficiency of the algorithm depends on various factors that are shown below:

Components that affect time efficiency

- Speed of the computer
- Choice of the programming language
- Compiler used
- Choice of the algorithm
- Number (Size) of inputs/Outputs
- Since we do not have any control over speed of the computer, programming language and compiler, let us concentrate only on next two factors such as:n
    - Choice of an algorithm
    - Number (size) of inputs

"What is basic operation?"

Definition: The operation that contributes most towards the running time of the algorithm is called basic operation. A statement that executes maximum number of times in a function is also called basic operation. The number of times basic operation is executed depends on size of the input. The basic operation is the most time consuming operation in the algorithm. For example:

- a statement present in the innermost loop in the algorithm
- addition operation while adding two matrices, since it is present in innermost loop
- multiplication operation in matrix multiplication since it is present in innermost loop

Now, let us see "How to compute the running time of an algorithm using basic operation or How time efficiency is analyzed?" The time efficiency is analyzed by determining the number of times the basic operation is executed. The running time T(n) is given by:

$T(n) \sim b*C(n)$

- T is the running time of the algorithm
- n is the size of the input
- b execution time for basic operation.
- C represent number of times the basic operation is executed

Order of growth

"What is order of growth? For what values of n we find the order of growth?"

Definition: We expect the algorithms to work faster for all values of n. Some algorithms execute faster for smaller values of n. But, as the value of n increases, they tend to be very slow. So, the behaviour of some algorithm changes with increase in value of n. This change in behaviour of the algorithm and algorithm's efficiency can be analyzed byconsidering the highest order of n. The order of growth is normally determined for largervalues of n for the following reasons:

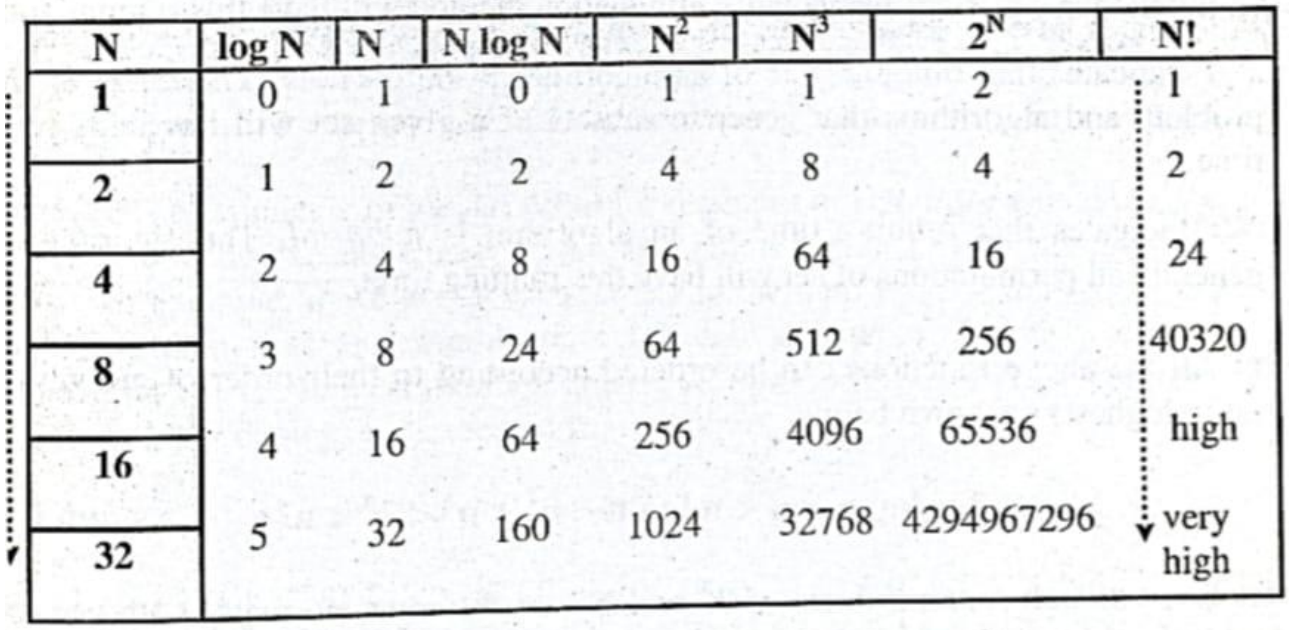
- The behavior of algorithm changes as the value of n increases
- In real time applications we normally encounter large values of n

For example, the order of growth with respect to two running times is shown below:

- ➢ Suppose T(n) approx c* C(n) . Observe that T(n) varies linearly with increase or decrease in the value of n. In this case, the order of growth is linear.
- ➢ Suppose T(n) approx c* C(n ^ 2) . In this context, the order of growth is quadratic.

Note: If the order of growth of one algorithm is linear and the order of growth of second algorithm to solve the same problem is quadratic, then it clearly indicates that running time of first algorithm is less and it is more efficient. So, while analyzing the time efficiency of an algorithm, the order of growth of n is important. Let us discuss orders of growth in the next section.

The concept of order of growth can be clearly understood by considering the common computing time functions shown in table 1.1.



N	log N	N	N log N	N^2	N^3	2^N	N!
1	0	1	0	1	1	2	1
2	1	2	2	4	8	4	2
4	2	4	8	16	64	16	24
8	3	8	24	64	512	256	40320
16	4	16	64	256	4096	65536	high
32	5	32	160	1024	32768	4294967296	very high

Fig 1.1 Values of some of the functions

Note: By comparing N (which is linear) and 2"(which is exponential) it is observed from the above table that exponential function grows very fast even for small variation of N. So, an algorithm with linear running time is preferred over an algorithm with exponential running time. The basic efficiency of asymptotic classes are shown below:

1 or any constant: Indicates that running time of a program is constant.

log N: Indicates that running time of a program is logarithmic. This running time occurs in programs that solve larger problems by reducing the problem size by a constant factor at each iteration of the loop (For example, binary search).

N: Indicates that running time of a program is linear. So, when N is 1000, the running time is 1000 units. When N is doubled, so does the running time. (For example linear search)

N log N: Indicates that running time of a program is N log N (For lack adjective, it is used as it is instead of linear, quadratic etc). The algorithm elements in ascending order such as quick sort, merge sort and heap sort wave this running time. (These sorting techniques are discussed in later chapters)

N^2 : Indicates that running time of a program is quadratic. The algorithms normallywill have two loops. For example, sorting algorithms such as bubble sort, selectionsort, addition and subtraction of two matrices have this running time.

N^3 : Indicates that running time of a program is cubic. The algorithms with runningtime will have three loops. For example, matrix multiplication, algorithm to solvesimultaneous equations using gauss-elimination method will have this running time.

2^N: Indicates that running time of an algorithm is exponential. The tower of Hanoiproblem and algorithms that generate subsets of a given set will have this runningtime

N! : Indicates that running time of an algorithm is factorial. The algongenerate all permutations of set will have this running time.

Note: All the above functions can be ordered according to their order of growth (fromlowest to highest) as shown below:

$$1 < \log(n) < n < n * \log(n) < n^2 < n^3 < 2^n < n!$$

Note: Even though running times of 2^N and N! are different, normally both classes are considered as exponential. The algorithms with this running time can be solved practically for smaller values of n.

Note: For a very large value of n, the exponential function 2" and factorial function n!generate a very high value such that even the fastest computer can take years to executean algorithm.

For example, a computer which executes 10^{12} instructions per second takes $4*10^{10}$ years to execute 2^{100} operations. The factorial function n! is much more than the value specified for 2^n.

Worst-case, Best-case and average case efficiencies

Now the question is "Will the algorithm efficiency depends on the size of algorithm's input alone?" For some of the problems, the time complexity will not depend on the number of inputs alone. For example, while searching for a specific item in an array of n elements using linear search, we have following three situations:

- An item we are searching for may be present in the very first location itself. In this case only one item is compared and this is the ***best case.***
- The item may be present somewhere in the middle which definitely takes some time. Running time is more when compared to the previous case for the same value of n. Since we do not know where the item is we have to consider the average number of cases and hence this situation is an ***average case.***
- The item we are searching for may not be present in the array requiring n number of comparisons and running time is more than the previous two cases. This may be considered as the ***worst case***.

So, knowing n alone itself is not enough to estimate the run time of an algorithm or a function. In such cases, we may have to find the worst-case efficiency, best case efficiency and average case efficiency. Now, let us see "What is worst case efficiency?"

"What is worst case efficiency?"

Definition: The efficiency of an algorithm for the input of size n for which the algorithm takes longest time to execute among all possible inputs is called worst case efficiency For example, if we use linear search and the item to be searched is no present in the array, then it is an example of ***worst case efficiency***.

In the worst case, the algorithm runs for the longest duration among all possible inputs for a given size. Here, maximum number of steps are executed for that input.

"What is best case efficiency?"

Definition: The efficiency of an algorithm for the input of size n for which the algorithm takes least time during execution among all possible inputs of that size is called best case efficiency.

In the best case, the algorithm runs fastest for the input specified. For example, in linear search, if the item to be searched is present in the beginning, then it is an example of the ***best case efficiency.***

Note: In the average case efficiency, the average number of basic operations executed will be considered. This is required only for the randomized input.

Example: Searching for an Element in an Array

Suppose you have an array of numbers, and you want to find a specific element in that array.

```
def search_element(arr, target):
    for num in arr:
        if num == target:
            return True
    return False
```

Now, let's analyze the efficiency of this algorithm in terms of worst-case, best-case, and average-case scenarios.

Worst-case efficiency:

- This is the scenario where the algorithm takes the maximum amount of time to complete.
- In our example, the worst-case scenario occurs when the target element is at the end of the array or is not present in the array.
- The worst-case time complexity is O(n), where n is the number of elements in the # Worst-case scenario
  arr = [1, 2, 3, 4, 5]
  target = 5
  result = search_element(arr, target)
  # In this case, the algorithm has to go through the entire array to find the target element.

Best-case efficiency:

- This is the scenario where the algorithm takes the minimum amount of time to complete.
- In our example, the best-case scenario occurs when the target element is at the beginning of the array.
- The best-case time complexity is O(1), as the algorithm may find the target element in the first iteration.
  # Best-case scenario
  arr = [1, 2, 3, 4, 5]
  target = 1
  result = search_element(arr, target)

# In this case, the algorithm finds the target element in the first iteration.

Average-case efficiency:

- This is the scenario where the algorithm takes an average amount of time to complete, considering all possible inputs.
- The average-case time complexity is often more challenging to analyze and may involve statistical analysis.
- In our example, if the target element is equally likely to be anywhere in the array, the average-case time complexity is O(n/2), which simplifies to O(n).

```
# Average-case
 scenarioarr = [1, 2, 3, 4, 5]
target = 3
result = search_element(arr, target)
# On average, the algorithm might need to check half of the array element
```

UNIT 2

Asymptotic Notations and Basic Efficiency classes, Informal Introduction, O-notation, **Ω**-notation, θ-notation, mathematical analysis of non-recursive algorithms, mathematical analysis of recursive algorithms.

Asymptotic notations

The efficiency of the algorithm is normally expressed using asymptotic notations. The order of growth can be expressed using two methods:

- Order of growth using asymptotic notations
- Order of growth using limits.

Before proceeding further, let us see "What do you mean by asymptotic behavior of a function?"

Definition: The value of the function may increase or decrease as the value of n increases. Based on the order of growth of n, the behavior of the function varies. Asymptotic notations are the notations using which two algorithms can be compared with respect to efficiency based on the order of growth of an algorithm's basic operation.

Now, let us see "What are the different types of asymptotic notations?" The different types of asymptotic notations are shown below:

Asymptotic notations

- O(Big Oh)
- Ω(Big Omega)
- ΘBigTheta

Now, let us see the informal definition and formal definition of above asymptotic notations

Informal Definitions of Asymptotic notations

In this section, let us "Give informal definitions of asymptotic notations"

Informal Definitions of Big-Oh(0)

Definition: Assuming n indicates the size of input and g(n) is a function, informally O(g(n)) is defined as set of functions with a small or same order of growth as g(n)) as n goes to infinity.

Ex 1: Let g(n) =n. Since n and I have smaller order of growth and n' has same order of growth when compared to n^2, we say.

Formal Definitions of Asymptotic notations

O (Big-Oh)

Now, let us see "What is Big Oh (0) notation?"

Definition: Let f(n) be the time efficiency of an algorithm. The function f(n) is said to be O(g(n)) [read as big-oh of g(n)), denoted by

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

or

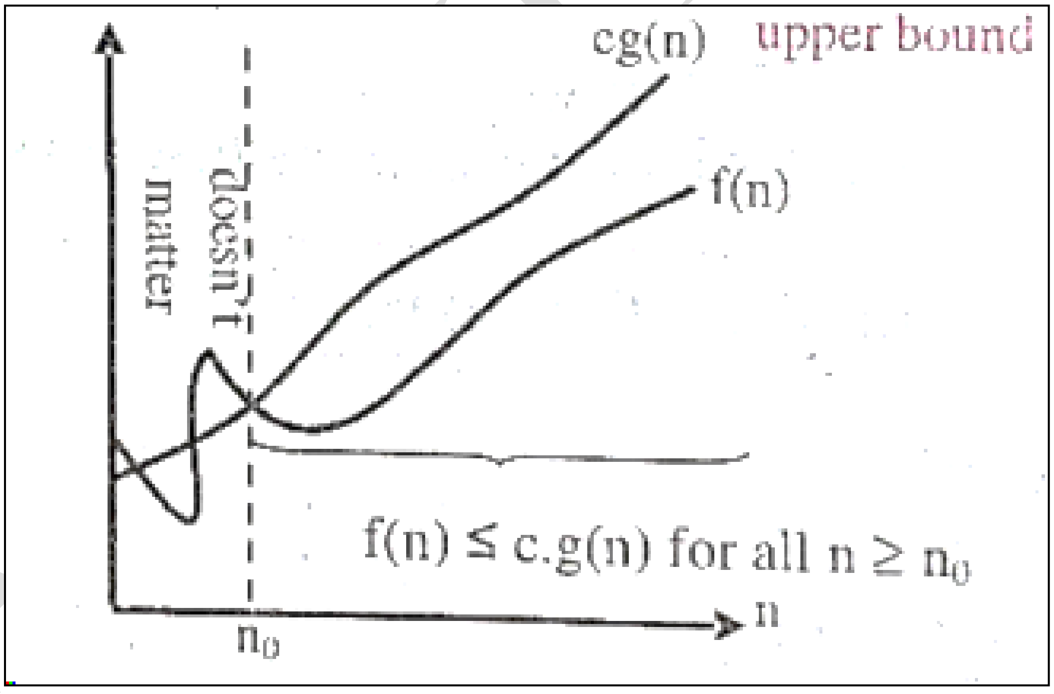
$$f(n) = O(g(n))$$

If and only if there exists a positive constant c and positive integer $n_0$satisfying the constraint

$f(n) <= c*g(n)$ for all $n \geq n_0$

So, if we draw the graph f(n) and c*g(n) verses n, the graph of the function f(n) lies

Below the graph of c g(n) for sufficiently large value of n as shown below:



Here, c .g(n) is the upper bound. The upper bound on f(n) indicates that function f(n) will not consume more than the specified time c* g(n) i.e., running time of function f(n) maybe equal to c .g(n), but it will never be worse than the upper bound. So, we can say that f(n) is generally faster than g(n)

Note: Big-O is the formal method of expressing the upper bound of an algorithm's running time. It is a measure of the longest amount of time it could possibly take for theAlgorithm to complete.

Note: The function g(n) is normally expressed using higher order terms of f(n) This is achieved using the following steps:

- Take the lower order term of f(n) replace the constant with next higher order variable. Repeat this step, till we get the higher order term and call it as c .g(n)
- Once the constraint " f(n) <= c.g(n) for n >= $n_0$is obtained we say f(n) ∈ O(g(n))

Example: Let f(n) = 100n + 5 Express f(n) using big-oh

Solution: It is given that f(n) = 100n + 5 Replacing 5 with n (so that next higher order term is obtained), we get 100n + n and call it c.g(n)

i.e..c. g(n) = 100n + n for n = 5

= 101nforn =5;

Now, the following constrain is satisfied:

f(n) <=c * g(n) for n ≥ n_0

I .e.,100n+5<=101 nfor n >= 5

It is clear from the above relations that c = 101 g(n) = n and n_0=5. So , by definition

f(n) ∈ O(g(n)) * I .e.., f(n) ∈ O(n)

Ω(Big-Omega)

Definition: Let f(n) be the time complexity of an algorithm. The function f(n) is said tobe Ω(g(n)) [read as big-omega of g(n)] which is denoted by

$$f(n) = \Omega g((n))$$

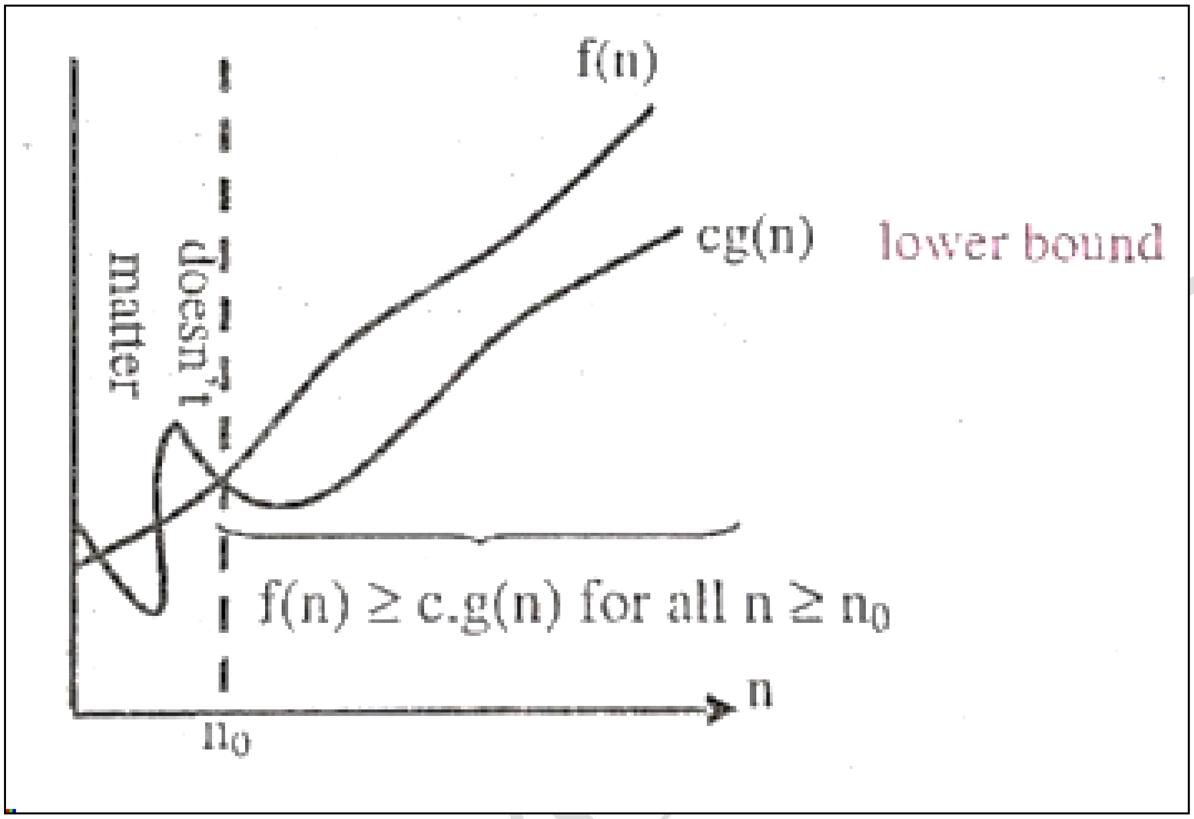
or

$$f(n) \text{ or } = \Omega(a(n))$$

If and only if there exists a positive constant c and non-negative integer no satisfying the constraint

$f(n) >= c*g(n)$ for all $n \geq n_0$.

So, if we draw the graph f(n) and c*g(n) verses n, the graph of f(n) lies above the graph of g(n) for sufficiently large value of n as shown below:



This notation gives the lower bound on a function f(n) within a constant factor. The lower bound on f(n) indicates that function f(n) will consume at least the specified time c*g(n)i.e., the algorithm has a running time that is always greater than c*g(n). In general, the lower bound implies that below this time the algorithm cannot perform better.

Note: $f(n) \geq c*g(n)$ indicates that g(n) is a lower bound and the running time of an algorithm is always greater than g(n). So, big-omega notation is used for finding best case time efficiency.

Example: Let f(n) = 100n+ 5. Express f(n) using big-omega

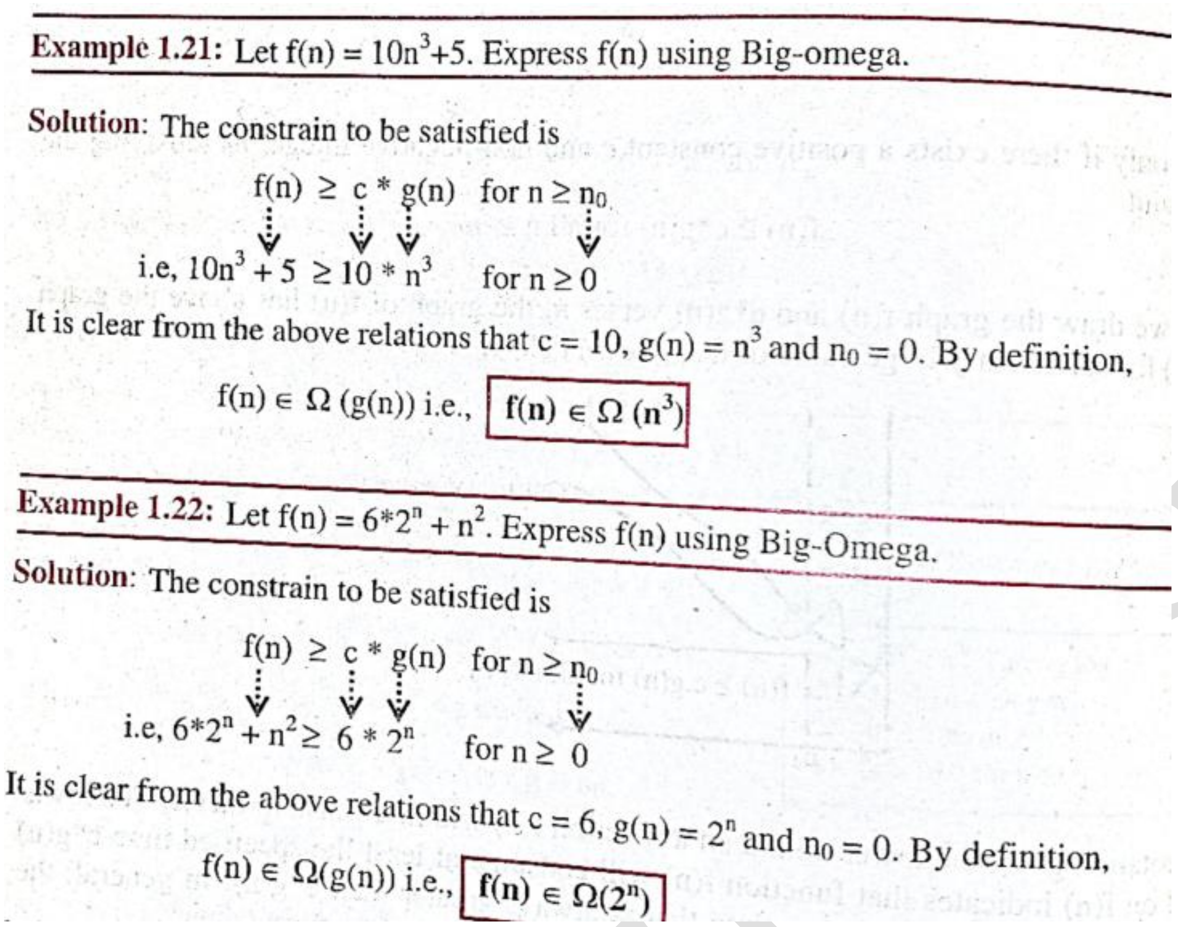
Solution: The constrain to be satisfied is

$f(n) >= c* g(n)$ for $n >= n_0$

i.e., $100n+5 \geq 100 n$ for $n>=0$

It is clear from the above relations that c = 100. g(n) = n and $n_0 = 0$. So, by definition

$$f(n) = \Omega(g(n)) \text{i.e.,} f(n) \in \Omega(n)$$



Example 1.21: Let $f(n) = 10n^3+5$. Express f(n) using Big-omega.

Solution: The constrain to be satisfied is

$$f(n) \geq c * g(n) \quad \text{for } n \geq n_0$$

$$\text{i.e, } 10n^3 + 5 \geq 10 * n^3 \quad \text{for } n \geq 0$$

It is clear from the above relations that $c = 10$, $g(n) = n^3$ and $n_0 = 0$. By definition,

$$f(n) \in \Omega(g(n)) \text{ i.e., } \boxed{f(n) \in \Omega(n^3)}$$

Example 1.22: Let $f(n) = 6*2^n + n^2$. Express f(n) using Big-Omega.

Solution: The constrain to be satisfied is

$$f(n) \geq c * g(n) \quad \text{for } n \geq n_0$$

$$\text{i.e, } 6*2^n + n^2 \geq 6 * 2^n \quad \text{for } n \geq 0$$

It is clear from the above relations that $c = 6$, $g(n) = 2^n$ and $n_0 = 0$. By definition,

$$f(n) \in \Omega(g(n)) \text{ i.e., } \boxed{f(n) \in \Omega(2^n)}$$

Big-Theta

Definition: Let f(n) be the time complexity of an algorithm. The function f(n) is said to be big-theta of g(n), denoted

$$f(n) = \Theta(g(n))$$

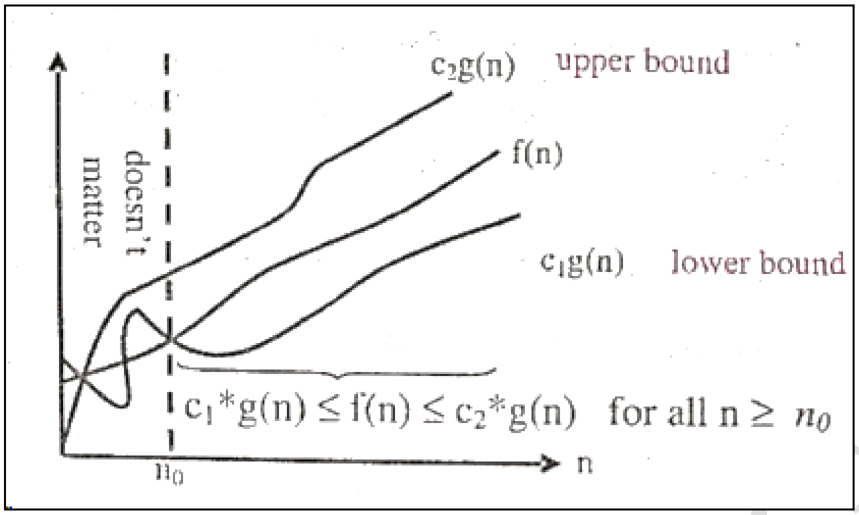
or

$$f(n)= \Theta(g(n))$$

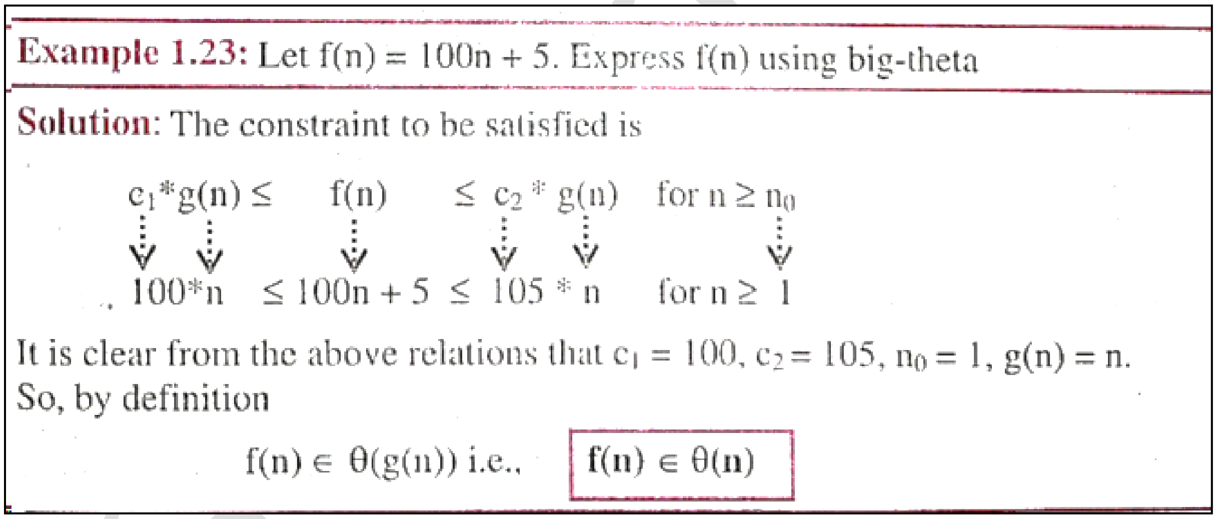
if and only if there exists some positive constants c_1, c_2 and non-negative integer n_o satisfying the constraint

$c_{1*}g(n)<=f(n)<=c_2* g(n)$ for all $n\geq n_0$.

So, if we draw the graph f(n), c_1,*g(n) and c_2 g*(n) verses n, the graph of function f(n) lies above the graph of c_1*g(n) and lies below the graph of c_2* g(n) for sufficiently large value of n as shown below:



This notation is used to denote both lower bound and upper bound on a function f(n) within a constant factor. The upper bound on f(n) indicates that function f(n) will not consume more than the specified time cog(n).The lower bound on f(n) indicates that function f(n) in the best case will consume at least the specified time $c_1*g(n)$.



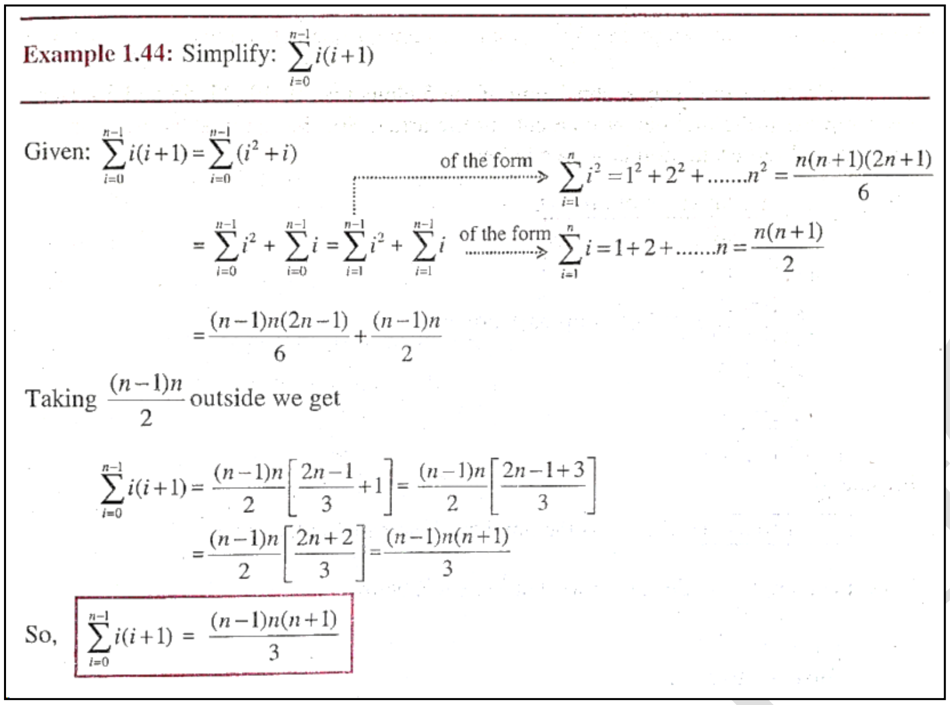
Example 1.23: Let f(n) = 100n + 5. Express f(n) using big-theta

Solution: The constraint to be satisfied is

$$\begin{array}{ccccc} c_1*g(n) \le & f(n) & \le c_2*g(n) & \text{for } n \ge n_0 \\ \Downarrow & \Downarrow & \Downarrow & \Downarrow \\ 100*n & \le 100n+5 & \le 105*n & \text{for } n \ge 1 \end{array}$$

It is clear from the above relations that $c_1 = 100$, $c_2 = 105$, $n_0 = 1$, $g(n) = n$.
So, by definition

$$f(n) \in \theta(g(n)) \text{ i.e., } \boxed{f(n) \in \theta(n)}$$



Example 1.44: Simplify: $\sum_{i=0}^{n-1} i(i+1)$

Given: $\sum_{i=0}^{n-1} i(i+1) = \sum_{i=0}^{n-1} (i^2+i)$ $\xrightarrow{\text{of the form}}$ $\sum_{i=1}^{n} i^2 = 1^2+2^2+\ldots\ldots n^2 = \frac{n(n+1)(2n+1)}{6}$

$$= \sum_{i=0}^{n-1} i^2 + \sum_{i=0}^{n-1} i = \sum_{i=1}^{n-1} i^2 + \sum_{i=1}^{n-1} i \xrightarrow{\text{of the form}} \sum_{i=1}^{n} i = 1+2+\ldots\ldots n = \frac{n(n+1)}{2}$$

$$= \frac{(n-1)n(2n-1)}{6} + \frac{(n-1)n}{2}$$

Taking $\frac{(n-1)n}{2}$ outside we get

$$\sum_{i=0}^{n-1} i(i+1) = \frac{(n-1)n}{2}\left[\frac{2n-1}{3}+1\right] = \frac{(n-1)n}{2}\left[\frac{2n-1+3}{3}\right]$$

$$= \frac{(n-1)n}{2}\left[\frac{2n+2}{3}\right] = \frac{(n-1)n(n+1)}{3}$$

So, $\boxed{\sum_{i=0}^{n-1} i(i+1) = \frac{(n-1)n(n+1)}{3}}$

Mathematical analysis of non - recursive algorithms

The general plan of energy non recursive algorithm is shown below:

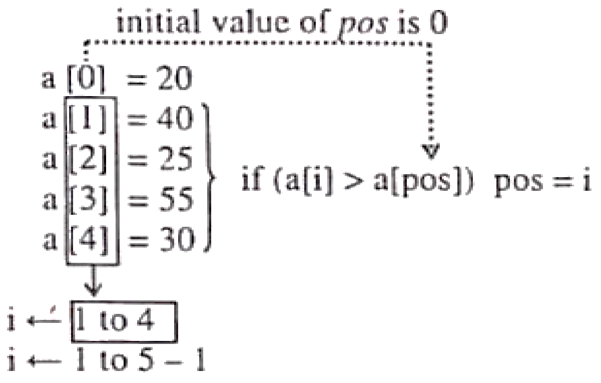
- Based on the size of input data in the number of parameters to be considered.
- Identify the basic operation in the algorithm
- Check whether the number of times the basic operation is executed depends only on the size of the input, if the basic operation to be executed depends on the some other conditions, then it is necessary to obtain the worst case, best case and average case separately.
- Open the total number of times a basic operation is executed
- Simplify using standard formula and obtain the order of growth.

Now let us consider some of the algorithms and see how to analyze these algorithms.

1.5.1 Maximum of n elements

Let us "Design the algorithm to find largest of *n* numbers and obtain the time efficiency"

Design: Consider the array *a* consisting of the 5 elements 20, 40, 25, 55 and 30. Here *n* = 5 represent the number of elements in the array. So, the parameters are *a* and *n*. The pictorial representation is shown below:



In general,

$\boxed{i \leftarrow 1 \text{ to } n-1}$ where *n* is the number of elements in the array

Now, the complete code can be written as shown below:

```
pos ← 0
for i ←1 to n – 1
        if (a[i] > a[pos])  pos ← i
end for
```

Now, the complete algorithm to find the largest of N elements can be written as shown below:

Example 1.45: Algorithm to find maximum of *n* elements

ALGORITHM Maximum(a[], n)
//Purpose : Find the largest of *n* numbers

```
//Inputs      : n – the number of items present in the table
                a – the table consisting of n elements
//Output      : pos – contains the position of largest element

        pos ← 0                               // 0 is assumed to be the position of  largest
                                              // element

        for i ← 1 to n-1 do                   // Find the position of largest element in the
                if ( a[i] > a[pos]) ) pos ← i // remaining elements of array from 1 to n-1
        end for

        return pos                            // return the position of largest element
```

End of algorithm Largest

Analysis: The time efficiency can be calculated as shown below:

Step 1: The parameter to be considered is *n* which represent the size of the input

Step 2: The element comparison i.e., **"if (a[i] > a[pos])"** is the basic operation

Step 3: The total number of times the basic operation is executed can be calculated as shown below:

```
for i ← 1 to n-1 do
    if ( a[i] > a[pos] ) pos ← i
    ......
```

$f(n) = \sum_{i=1}^{n-1} 1$ **Note:** Upper bound = n-1, Lower bound = 1

$= (n-1)-1+1$ // Result = [Upper bound – lower bound +1]

$= n-1$

i.e., $\mathbf{f(n) = n-1 \approx n}$ **// By neglecting lower order terms and constants**

Step 4: Express f(n) using asymptotic notation. So, time complexity is given by:

$$\boxed{f(n) \in O(n)}$$

Mathematical analysis of recursive algorithms

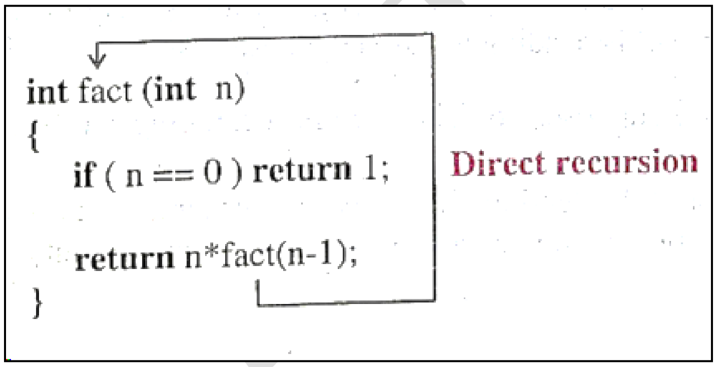
Recursion is a powerful tool but least understood by most novice students. Programming languages such as Pascal, C, C++ etc support recursion. Now, let us see "What is recursion? What are the various types of recursion?"

Definition: A recursion is a method of solving the problem where the solution to a problem depends on solutions to smaller instances of the same problem. Thus, a recursive function is a function that calls itself during execution. This enables the function to repeat itself several times to solve a given problem.

The various types of recursion are shown below:

1. Direct recursion
2. Indirect recursion

Direct recursion: A recursive function that invokes itself is said to have direct recursion. For example, the factorial function calls itself (detailed explanation is given later) and hence the function is said to have direct recursion.

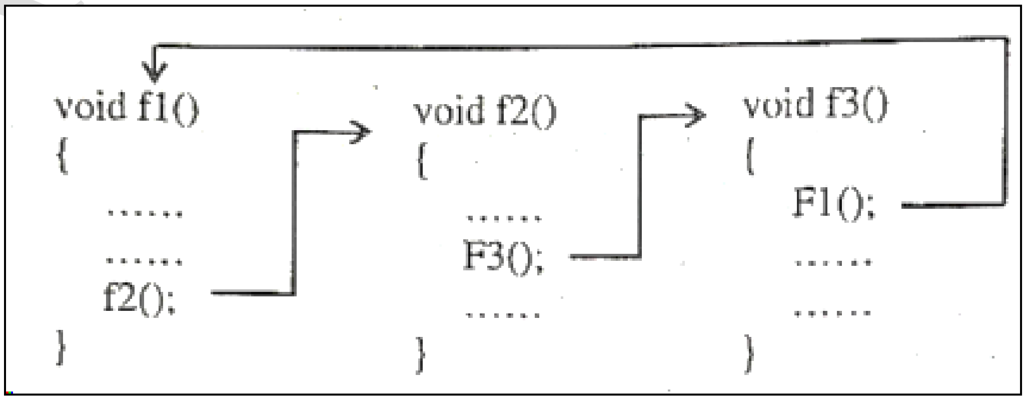


```
int fact (int  n)
{
    if ( n == 0 ) return 1;

    return n*fact(n-1);
}
```

Indirect recursion: A function which contains a call to another function which in turn calls another function which in turn calls another function and so on and eventually calls the first function is called indirect recursion.

It is very difficult to read, understand and find any logical errors in a function that has indirect recursion. For example, a function fl invokes f2 which in turn invokes f3 which in turn invokes fl is said to have indirect recursion. This is pictorially represented as shown below:



```
void f1()        void f2()        void f3()
{                {                {
   ......           ......           F1();
   ......           F3();            ......
   f2();            ......           ......
}                }                }
```

Now, the question is ***"How to design recursive functions?"*** Every recursive call must solve one part of the problem using base case or reduce the size (or instance) of the problem using general case. Now, let us see ***"What is base case? What is a general case?"***

Definition: A ***base case*** is a special case where solution can be obtained without using recursion. This is also called base/terminal condition. Each recursive function must have a base case. A base case serves two purposes:

- It acts as terminating condition.
- The recursive function obtains the solution from the base case it reaches.

For example, in the function factorial O! is I is the base case or terminal condition.

Definition: In any recursive function, the part of the function except base case is called general case. This portion of the code contains the logic required to reduce the size (or instance) of the problem so as to move towards the base case or terminal condition. Here, each time the function is called, the size (or instance) of the problem is reduced.

For example, in the function fact, n*fact(n-1) is general case. By decreasing the value of n by 1, the function fact is heading towards the base case.

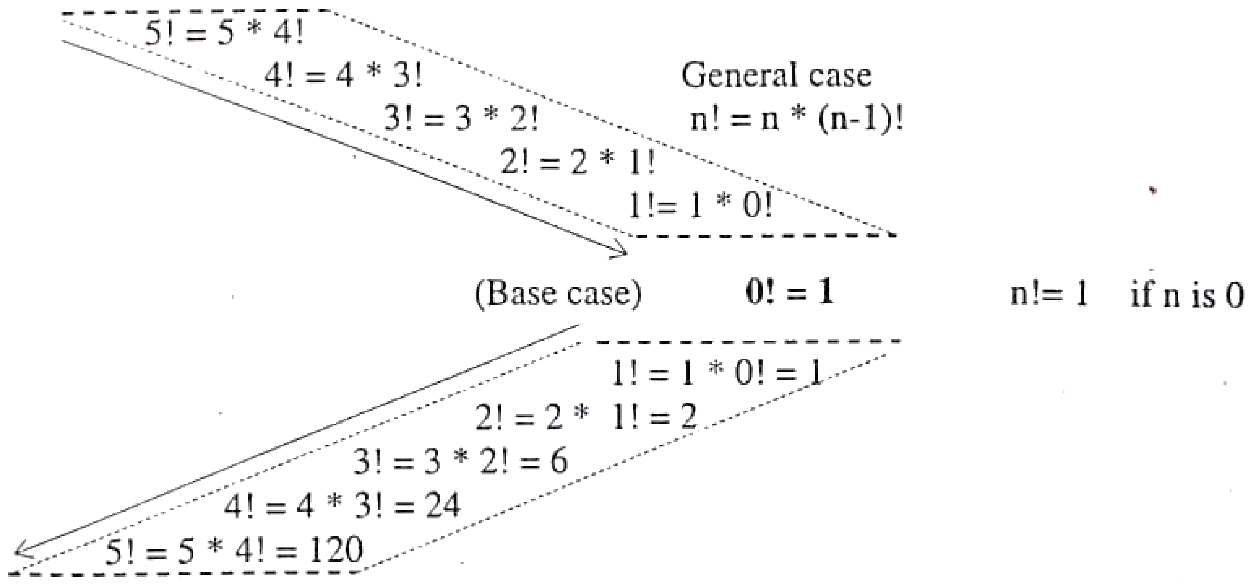
So, the general rules that we are supposed to follow while designing any recursive algorithm are:

- **Determine the base case**. Careful attention should be given here, because: when base case is reached, the function must execute a return statement without a call to recursive function.
- **Determine the general case.** Here also careful attention should be given and see that each call must reduce the size of the problem and moves towards base case.
- **Combine the base case and general case** into a function.

Note: A recursive function should never generate infinite sequence of calls on itself. An algorithm exhibiting this sequence of calls will never terminate and hence it is called infinite recursion. If a base case does not exist, no recursive function can ever be computed.

1.6.1 Factorial of a number

Now, let us see "How to compute factorial of 5 using recursion?" We can compute 5! as shown below:



Thus the recursive definition can be written as shown below:

$$\boxed{\begin{array}{ll} n! = 1 & \text{if } n == 0 \\ n! = n * (n-1)! & \text{otherwise} \end{array}} \quad \text{or} \quad \boxed{n! = \begin{cases} 1 & \text{if } n == 0 \\ n * (n-1)! & \text{otherwise} \end{cases}}$$

The above definition can also be written as shown below:

$$\boxed{F(n) = \begin{cases} 1 & \text{if } n == 0 \\ n * F(n-1) & \text{otherwise} \end{cases}}$$

Using the above recursive definition, the recursive algorithm can be written as shown below:

Example 1.53: Recursive algorithm to find the factorial of N

```
Algorithm fact(n)
// Purpose      : This function computes factorial of n
// Input        : n: represent a positive integer
//Output        : factorial of n

        if ( n == 0 )    return 1              // n! = 1 if n is 0
        return           n*fact(n-1)           // n! = n*(n-1)! otherwise
```

End of algorithm factorial

The C function for the above algorithm can be written as shown below:

Example 1.54: C function to find the factorial of N

```
int fact(int n)
{
        if ( n == 0 )    return 1;          /* factorial of n when n = 0 */
        return           n*fact(n-1);       /* factorial of n when n > 0 */
}
```

Now, let us see "What is the general plan to analyze the efficiency of recursive algorithms?" The general plan to analyze the recursive algorithms is shown below:

- Identify the parameters based on the size of the input
- Identify the basic operation in the algorithm
- Obtain the number of times the basic operation is executed on different inputs of the same size. If it varies, then it is necessary to obtain the worst case, best case and average case separately.
- Obtain a recurrence relation with an appropriate initial condition
- Solve the recurrence relation and obtain the order of growth and express using asymptotic notations.

Analysis The time efficiency of the algorithm to find the factorial of a number can be obtained as shown below:

Step 1: The parameter to be considered is n, which is a measure of input's size

Step 2: The basic operation is the multiplication statement.

Step 3: The total number of multiplications can be obtained using the recurrence relation as shown below:

$$\boxed{t(n) = \begin{cases} 0 & \text{if } n = 0 \\ 1 + t(n-1) & \text{otherwise} \end{cases}} \quad \text{OR} \quad \begin{array}{l} t(0) = 0 \\ t(n) = 1 + t(n-1) \end{array} \quad \text{------------(1)}$$

The above recurrence relation can be solved using repeated substitution as shown below:

$t(n) = 1 + t(n-1)$

$= 1 + 1 + t(n-2)$

$= 2 + t(n-2)$

$= 2 + 1 + t(n-3)$

$= 3 + t(n-3)$

$= 4 + t(n-4)$

....

$t(n) = 1 + t(n-1)$ ---------------------------------(2)

$t(n-1) = 1 + t(n-2)$ replacing n by n-1 in equation (2)

$t(n-2) = 1 + t(n-3)$ replacing n by n-2 in equation (2)

Note: Looking at the previous expression it is written

....

....

$= i + t(n-i)$

Finally, to get initial condition t(0), let $i = n$

$= n + t(n-n)$

$= n + t(0)$ **Note:** $t(0) = 0$ from equation (1)

$= n + 0 = n$

So, the time complexity of factorial of N is given by $t(n) \in \Theta(n)$

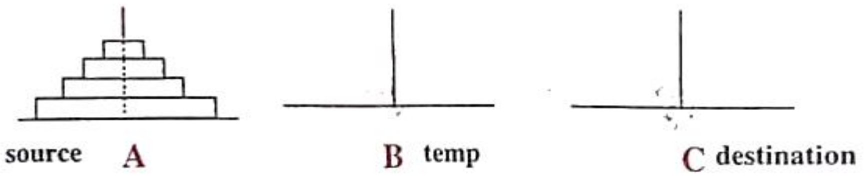
1.6.2 Tower of Hanoi

Let us see, "What is tower of Hanoi problem?" In this problem, there are three needles say **A**, **B** and **C**. The different diameters of **n** discs are placed one above the other through the needle **A** and the discs are placed such that always a smaller disc is placed above the larger disc. The two needles B and C are empty. All the discs from needle A are to be transferred to needle C using needle B as temporary storage.

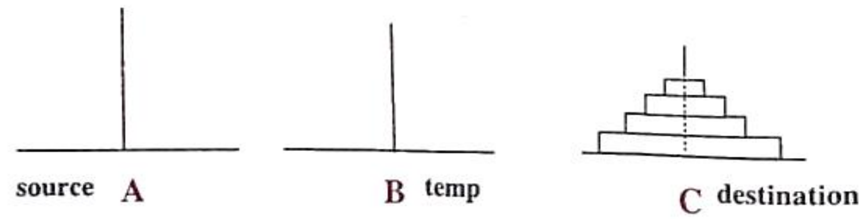
The rules to be followed while transferring the discs are

- Only one disc is moved at a time from one needle to another needle
- Smaller disc is on top of the larger disc at any time.
- Only one needle can be used to for storing intermediate discs

The initial set up of the problem is shown below.



After transferring all discs from **A** to **C** we get the following setup.



Base Case: Now, let us see "What is the base case?" This case occurs when there are no discs. In such situation we simply return the control to the calling function using the statement **return**. The base case can be written as:

Action	Condition
return	**if n = 0**

General case: Now, let us see "What is the general case?" This case occurs if one or more discs have to be transferred from source to destination. If there are *n* discs, then all *n* discs can be transferred recursively using following three steps:

- Move **n-1** discs recursively from **source** to **temp**.
- Move **n**th disc from **source** to **destination**.
- Move **n-1** discs recursively from **temp** to **destination.**

The algorithm to implement tower of Hanoi problem is shown below:

Example 1.56: Algorithm for tower of Hanoi problem

```
Algorithm TowerOfHanoi(n, source, temp, destination)
//Purpose: To move n discs from source to destination and see that only one disc is
           moved and always the smaller disc should be placed above the larger disc
           using one of the needle as temporary holder for the disc being moved
// Inputs :
//      n: total number of disks to be moved
//Output:
//      all n disks should be available on the destination needle.
Step 1: [ Check for base case ]
        if ( n = 0 ) return
Step 2: [Recursively move n-1 disks from source to temp]
        TowerOfHanoi(n-1, source, destination, temp)
Step 3: [Move nth disk from source to destination]
        write("Move disk", n, "from", source, "to", destination)
Step 4: [Recursively move n-1 disks from temp to destination]
        TowerOfHanoi(n-1, temp, source, destination)
```

End of tower of Hanoi algorithm

Analysis The time efficiency can be calculated as shown below:

Step 1: The parameter to be considered is *n*, which represent number of disks

Step 2: The basic operation is the movement of disk

Step 3: The total number of disk movements can be obtained using the recurrence relation as shown below:

$$\boxed{t(n) = \begin{cases} 1 & \text{if } n = 1 \\ \underbrace{t(n-1)}_{\text{Number of disk movements from source to temp}} + \underbrace{1}_{\text{Number of disk movements from source to destination}} + \underbrace{t(n-1)}_{\text{Number of disk movements from temp to destination}} & \text{otherwise} \end{cases}}$$

The above recurrence relation can also be written as shown below:

$$f(1) = 1 \qquad \text{--(1)}$$
$$f(n) = 2f(n-1) + 1$$

The above recurrence relation can be solved using repeated substitution as shown below:

$$f(n) = f(n-1) + 1 + f(n-1)$$

$f(n) = 2f(n-1) + 1$ ----------------(2)
$f(n-1) = 2f(n-2) + 1$ by replacing n by n-1 in eq. (2)

$$= 2f(n-1) + 1$$
$$= 2\,[2f(n-2) + 1\,] + 1$$

$f(n-2) = 2f(n-3) + 1$ by replacing n by n-2 in eq. (2)

$$= 2^2 f(n-2) + 2 + 1$$
$$= 2^2\,[2f(n-3) + 1] + 2 + 1$$
$$= 2^3 f(n-3) + 2^2 + 2 + 1$$
$$= 2^4 f(n-4) + 2^3 + 2^2 + 2 + 1$$

Note: Written by looking at the previous expression

....
....

In general,

$$= 2^i f(n-i) + 2^{i-1} + 2^{i-2} \ldots\ldots 2^3 + 2^2 + 2 + 1$$

$$= 2^i f(n-i) + \underbrace{2^{i-1} + 2^{i-2} \ldots\ldots 2^3 + 2^2 + 2^1 + 2^0}_{} \quad \text{------------------ (3)}$$

This geometric series can be solved using $S = \dfrac{a(r^n - 1)}{r - 1}$

where a = 1, r = 2, n = i (**since number of terms from 0 to i-1 = i**)

So, $S = \dfrac{1(2^i - 1)}{2 - 1} = 2^i - 1$. Substituting this value in equation (3) we have,

$= 2^i f(n-i) + 2^i - 1$

$\vdots$ ......> Finally, to get initial condition f(1), let i = n-1

$= 2^{n-1} f(n-[n-1]) + 2^{n-1} - 1$

$= 2^{n-1} f(n - n + 1) + 2^{n-1} - 1$

$= 2^{n-1} f(1) + 2^{n-1} - 1$

$= 2^{n-1} * 1 + 2^{n-1} - 1$ **Note:** f(1) = 1 from equation (1)

$= 2^{n-1} + 2^{n-1} - 1$

$= 2*2^{n-1} - 1 = 2* \frac{2^n}{2} - 1 = 2^n - 1$

So, **$f(n) = 2^n - 1$**

Step 4: Since number of disk movements in best case is same as worst case, we express f(n) using θ-notation as shown below:

So, the time complexity for Tower of Hanoi problem is given by $f(n) \in \theta(2^n - 1) \in \theta(2^n)$