DEPARTMENT OF BCA

ANALYSIS AND DESIGN OF ALGORITHMS (16BCA5D11)

Module 1 : Role of Algorithms in Computing

Prepared by: Prof. Bhavana Gowda D M

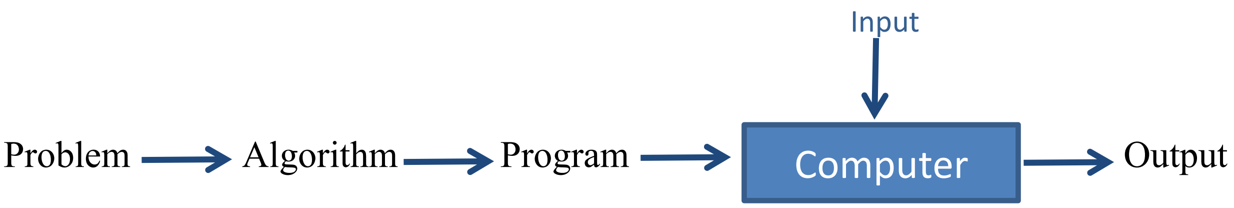
UNIT 1: ROLE OF ALGORITHMS IN COMPUTING

1. Introduction

What is an Algorithm?

An **algorithm** is defined as a sequence of finite unambiguous instructions to be followed to accomplish a given task in a finite number of steps by accepting a set of inputs and producing the desired output.

Notions of an Algorithms : Pictorial representation of an algorithm.



Need for studying algorithms:

- The study of algorithms is the cornerstone of computer science.
- It can be recognized as the core of computer science.
- Computer programs would not exist without algorithms.
- With computers becoming an essential part of our professional & personal life‘s, studying algorithms becomes a necessity, more so for computer science engineers.
- Another reason for studying algorithms is that if we know a standard set of important algorithms, they further our analytical skills & help us in developing new algorithms for required applications.

The Essential Characteristics of an Algorithm

The essential characteristics of an algorithm are;

1) Every step of an algorithm should perform a single task.
2) Ambiguity (Confusion) should not be included at any stage in an algorithm.
3) An algorithm should involve a finite number of steps to arrive at a solution.
4) Simple statement and structures should be used in the development of the algorithm.
5) Every algorithm should lead to a unique solution of the problem.
6) An algorithm should have the capability to handle some unexpected situations, which may arise during the solution of a problem (For example, division by zero).

Properties of an Algorithm

All algorithms must satisfy the following criteria:

1) *Input*. Zero or more quantities are externally supplied.
2) *Output.* At least one Output is produced.
3) *Definiteness.* Each instruction should be simple and clear.
4) *Finiteness.* If we trace out the instruction of an algorithm, then for all cases, the algorithm terminates after a finite number of steps.
5) *Effectiveness.* Every instruction must be very basic so that it can be carried out, in principal, by a person using only pencil and paper.

Advantages of Algorithm

1. It is a step-by-step representation of a solution to a given problem, which is very easy to understand.
2. It has got a definite procedure, which can be executed within a set period of time.

3. It is easy to first develop an algorithm, and then convert it into a flowchart and then into a computer program.
4. It is independent of programming language.
5. It is easy to debug as every step has got its own logical sequence.

Disadvantages of Algorithm

It is time consuming and cumbersome as an algorithm is developed first which is converted into a flowchart and then into a computer program.

Example : Computing GCD (Greatest Common Divisor) :

- GCD of two numbers m and n denoted by GCD(m,n)
- Defined as the largest integer that divides both m and n such that the remainder is Zero and applicable only for positive integers.

Different Ways of Computing GCD of two nubers:

1. Euclid's Algorithm
2. Repetative Subtraction
3. Consecutive Integer Checking Algorithm
4. Middle School Procedure using prime factors

1. Euclid's Algorithm : Named after the mathematician - Euclid, Alexandria

Procedure :

Step1 : Compute the remainder using the statement R <- M % N

Step 2: Assign N to M i.e., M <- N

Step 3: Assign remainder R to N i.e., N <- R

		2	3	
	M	N	R ← M % N	
Given {	6	10	6 ← 6 % 10	1
	10	6	4 ← 10 % 6	
	6	4	2 ← 6 % 4	
	4	2	0 ← 4 % 2	
	2	0	stop when n is zero	

Iterative Algorithm :

Algorithm GCD(M,N)
//Description : Computes GCD(M,N)
//Input : M and N should be positive integers
//Output: GCD of M and N

While $N \neq 0$ do
    R ← M%N
    M ← N
    N ← R
End While
Return M

Recursive Algorithm :

Algorithm GCD(M,N)

//Description : Computes GCD(M,N)

//Input : M and N should be positive integers

//Output: GCD of M and N

$$GCD(M,N) = \begin{cases} M & \text{if } N=0 \\ GCD(N,\ M\%N) & \text{Otherwise} \end{cases}$$

Consecutive Integer Checking :

WKT GCD of 2 numbers M and N cannot be more than the smaller of these two numbers.

To start with : small = min(m,n)

Divide m and n by small.

if remainder in both the case is zero, then small will be the GCD.

Otherwise, decrement small by 1 and repeat the process till both m and n are divisible by small.

small	m % small	n %small	small is GCD ?
6	10 % 6 = 4	6 % 6 = 0	6 is not GCD
5	10 % 5 = 0	6 % 5 = 1	5 is not GCD
4	10 % 4 = 2	6 % 4 = 2	4 is not GCD
3	10 % 3 = 1	6 % 3 = 0	3 is not GCD
2	**10 % 2 = 0**	**6 % 2 = 0**	**2 is GCD** (remainder is 0)

```
Algorithm GCD(M,N)
//Description : Computes GCD(M,N)
//Input : M and N should be positive integers
//Output: GCD of M and N
Small      min(m,n)
while(1)
        if (m mod small=0)
          if (n mod small =0)
                return small
          end if
        end if
small      small - 1
end while
return small
```

Note : This algorithm will not work if one of the input is Zero.

2. Repetative Subtraction :

Procedure :

- If m is greater than n, perform m-n and store in m

- If n is greater than m, perform n-m and store in n
- Repeat the above 2 steps as long as m and n are not equal
- if m and n are same, return m and n as GCD.

M	N	Description
10	6	M = 10 – 6 = 4 (Since M > N, subtract N from M and store in M)
4	6	N = 6 – 4 = 2 (Since N > M, subtract M from N and store in N)
4	2	M = 4 – 2 = 2 (Since M >N, subtract N from M and store in M)
2	2	Since M and N are same, the GCD will be either M or N which is 2.

GCD = 2.

<u>Algorithm GCD(M,N)</u>

```
//Description : Computes GCD(M,N)
//Input : M and N should be positive integers
//Output: GCD of M and N
while m ≠ n do
        if (m>n)
                m ← m-n
          else
                n ← n-m
          end if
end while
return m
```

Note : If one of two numbers is a Zero. Return non-negative number as the GCD

3. Middle School Procedure:

Procedure :

Step 1: Find the prime factors of m

Step 2: Find the prime factors of n

Step 3: Identify the common prime factors obtained in step 1 and step 2

Step 4: Find the product of all common factors and return the result as gcd of two given numbers.

<u>Disadvantages of Middle School Procedure:</u>

- More Complex
- This algorithm does not specify how to generate the prime factors and hence it is not a legitimate algorithm.
- This algorithm does not specify how to find common factors and hence it is not possible to write the program.

Fundamentals of Algorithmic Problem Solving

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

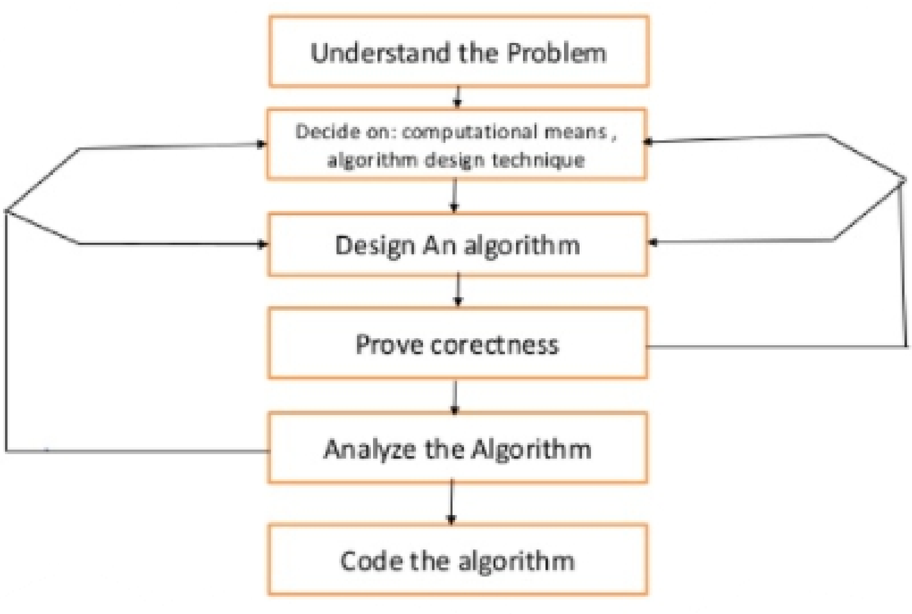


Figure: Flow diagram of designing and analyzing an algorithm

(i) Understanding the Problem

- This is the first step in designing of algorithm.
- Read the problem's description carefully to understand the problem statement completely.
- Ask questions for clarifying the doubts about the problem.
- Identify the problem types and use existing algorithm to find solution.
- Input (instance) to the problem and range of the input get fixed.

(ii) Decision making

The Decision making is done on the following:

(a) Ascertaining the Capabilities of the Computational Device

- In random-access machine (RAM), instructions are executed one after another (The central assumption is that one operation at a time). Accordingly, algorithms designed to be executed on such machines are called sequential algorithms.
- In some newer computers, operations are executed concurrently, i.e., in parallel. Algorithms that take advantage of this capability are called parallel algorithms.
- Choice of computational devices like Processor and memory is mainly based on space and time efficiency

(b) Choosing between Exact and Approximate Problem Solving

- The next principal decision is to choose between solving the problem exactly and solving it approximately.
- An algorithm used to solve the problem exactly and produce correct result is called an exact algorithm.
- If the problem is so complex and not able to get exact solution, then we have to choose an algorithm called an approximation algorithm. i.e., produces an approximate answer. E.g., extracting square roots, solving nonlinear equations, and evaluating definite integrals.

(c) Algorithm Design Techniques

- An algorithm design technique (or “strategy” or “paradigm”) is a general approach to solving problems algorithmically that is applicable to a variety of problems from different areas of computing.

- Algorithms+ Data Structures = Programs
- Though Algorithms and Data Structures are independent, but they are combined together to develop program. Hence the choice of proper data structure is required before designing the algorithm.
- Implementation of algorithm is possible only with the help of Algorithms and Data Structures
- Algorithmic strategy / technique / paradigm are a general approach by which many problems can be solved algorithmically. E.g., Brute Force, Divide and Conquer, Dynamic Programming, Greedy Technique and so on

(iii) Methods of Specifying an Algorithm

There are three ways to specify an algorithm. They are:

a. Natural language

b. Pseudo code

c. Flowchart

(iv) Proving an Algorithm's Correctness

a. Once an algorithm has been specified then its correctness must be proved.

b. An algorithm must yields a required result for every legitimate input in a finite amount of time. For example, the correctness of Euclid's algorithm for computing the greatest common divisor stems from the correctness of the equality gcd(m, n) = gcd(n, m mod n).

c. A common technique for proving correctness is to use mathematical induction because an algorithm's iterations provide a natural sequence of steps needed for such proofs.

d. The notion of correctness for approximation algorithms is less straightforward than it is for exact algorithms. The error produced by the algorithm should not exceed a predefined limit

(v) Analyzing an Algorithm

a. For an algorithm the most important is efficiency. In fact, there are two kinds of algorithm efficiency. They are:
b. Time efficiency, indicating how fast the algorithm runs, and
c. Space efficiency, indicating how much extra memory it uses.
d. The efficiency of an algorithm is determined by measuring both time efficiency and space efficiency.
e. So factors to analyze an algorithm are:
    - ♣ Time efficiency of an algorithm
    - ♣ Space efficiency of an algorithm
    - ♣ Simplicity of an algorithm
    - ♣ Generality of an algorithm

(vi) Coding an Algorithm

a. The coding / implementation of an algorithm is done by a suitable programming language like C, C++, JAVA.
b. The transition from an algorithm to a program can be done either incorrectly or very inefficiently. Implementing an algorithm correctly is necessary. The Algorithm power should not reduce by inefficient implementation.
c. Standard tricks like computing a loop's invariant (an expression that does not change its value) outside the loop, collecting common sub-expressions, replacing expensive operations by cheap ones, selection of programming language and so on should be known to the programmer.
d. Typically, such improvements can speed up a program only by a constant factor, whereas a better algorithm can make a difference in

running time by orders of magnitude.

e. It is very essential to write an optimized code (efficient code) to reduce the burden of compiler.

Role of algorithms in computing

The most important problem types related to computing are:

1) *Sorting*: The sorting problem is to rearrange the items of a given list in non-decreasing (ascending) order.

2) *Searching:* The searching problem deals with finding a given value, called a search key, in a given set.

3) String processing: A string is a sequence of characters from an alphabet. Strings comprise letters, numbers, and special characters; bit strings, which comprise zeros and ones; and gene sequences, which can be modeled by strings of characters from the four character alphabet {A, C, G, T}. It is very useful in bioinformatics. Searching for a given word in a text is called string matching.

4) *Graph problems*: A graph is a collection of points called vertices, some of which are connected by line segments called edges. Some of the graph problems are graph traversal, shortest path algorithm, topological sort, traveling salesman problem and the graph-coloring problem and so on.

5) *Combinatorial problems*: These are problems that ask, explicitly or implicitly, to find a combinatorial object such as a permutation, a combination, or a subset that satisfies certain constraints.

6) *Geometric problems:* Geometric algorithms deal with geometric objects such as points, lines, and polygons. Geometric algorithms are used in

computer graphics, robotics, and tomography.

7) Numerical problems are problems that involve mathematical equations, systems of equations, computing definite integrals, evaluating functions, and so on. The majority of such mathematical problems can be solved only approximately.

Algorithms as a Technology.

There a huge number of real life applications where, algorithms play the major and vital role. Few of such applications are discussed below.

- The ***Internet*** without which it is difficult to imagine a day is the result of *clever and efficient algorithms*. With the aid of these algorithms, various sites on the Internet are able to manage and manipulate this large volume of data. Finding good routes on which the data will travel and using search engine to find pages on which particular information is present.

- Another great milestone is the Human Genome Project which has great progress towards the goal of identification of the 100000 genes in human DNA, determining the sequences of the 3 billion chemical base pairs that make up the human DNA, storing this *huge amount of information* in databases, and developing tools for data analysis. Each of these steps required *sophisticated and efficient algorithms.*

- The day-to-day ***electronic commerce*** activities is hugely dependent on our personal information such as credit/debit card numbers, passwords, bank statements, OTPs and so on. The core technologies used include public-key crypto currency and digital signatures which are based on numerical algorithms and number theory.

- The approach of **linear programming** is also one such technique which is widely used like
  - In *manufacturing* and other commercial enterprises where resources need to be allocated scarcely in the most beneficial way.
  - Or a institution may want to determine where to spend money buying advertising in order to *maximize* the chances of their institution to grow.

- **Shortest path algorithm** also has an extensive use as
  - In a *transportation* firm such as a trucking or railroad company, may have financial interest in finding shortest path through a road or rail network because taking shortest path result in lower labour or fuel costs.
  - Or a *routing node* on the Internet may need to find the shortest path through the network in order to route a message quickly.

- Even an application that does not require algorithm content at the application level relies heavily on algorithms as the application depends on *hardware, GUI, networking or object orientation* and all of these make an extensive use of algorithms.

Getting Started

Fundamentals of the Analysis of Algorithm Efficiency

The efficiency of an algorithm can be in terms of time and space. The algorithm efficiency can be analyzed by the following ways.

a. Analysis Framework.

b. Asymptotic Notations and its properties.

c. Mathematical analysis for Recursive algorithms.
d. Mathematical analysis for Non-recursive algorithms.

Algorithm analysis is an important part of computational complexity theory, which provides theoretical estimation for the required resources of an algorithm to solve a specific computational problem. Most algorithms are designed to work with inputs of arbitrary length. Analysis of algorithms is the determination of the amount of time and space resources required to execute it.

The complexity of an algorithm describes the efficiency of the algorithm in terms of the amount of the memory required to process the data and the processing time.

Complexity of an algorithm is analyzed in two perspectives: **Time** and **Space**.

(a) Time Complexity

It's a function describing the amount of time required to run an algorithm in terms of the size of the input. "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.

Components that affect Time Efficiency :

A. Speed of the Computer
B. Choice of the Programming Language
C. Compiler Used
D. Choice of the Algorithm
E. Size of Inputs / Outputs

(b) Space Complexity

It's a function describing the amount of memory an algorithm takes in terms of the size 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.

Space complexity is sometimes ignored because the space used is minimal and/or obvious, however sometimes it becomes as important an issue as time.

Components that Affects Space Efficiency :

A. Program Space

B. Data Space

C. Stack Space

The Need for Analysis

In this section, we will discuss the need for analysis of algorithms and how to choose a better algorithm for a particular problem as one computational problem can be solved by different algorithms.

Algorithms are often quite different from one another, though the objective of these algorithms are the same. For example, we know that a set of numbers can be sorted using different algorithms. Number of comparisons performed by one algorithm may vary with others for the same input. Hence, time complexity of those algorithms may differ. At the same time, we need to calculate the memory space required by each algorithm.

The algorithm analysis framework consists of the following:

- Measuring an Input's Size
- Units for Measuring Running Time
- Orders of Growth
- Worst-Case, Best-Case, and Average-Case Efficiencies

Measuring an Input's Size:

An algorithm's efficiency is defined as a function of some parameter n indicating the algorithm's input size. In most cases, selecting such a parameter is quite straightforward. For example, it will be the size of the list for problems of sorting, searching.

Units for Measuring Running Time:

Some standard unit of time measurement such as a second, or millisecond, and so on can be used to measure the running time of a program after implementing the algorithm.

Drawbacks

- Dependence on the speed of a particular computer.
- Dependence on the quality of a program implementing the algorithm.
- The compiler used in generating the machine code.
- The difficulty of clocking the actual running time of the program.

So, we need metric to measure an algorithm's efficiency that does not depend on these extraneous factors. One possible approach is **to count the number of times each of the algorithm's operations is executed**. This approach is excessively difficult.

Orders of Growth

A difference in running times on small inputs is not what really distinguishes efficient algorithms from inefficient ones. Some common orders of growth seen often in complexity analysis are:

$O(1)$	constant
$O(\log n)$	logarithmic
$O(n)$	linear
$O(n \log n)$	"n log n"
$O(n^2)$	quadratic
$O(n^3)$	cubic
$n^{O(1)}$	polynomial
$2^{O(n)}$	exponential

Table: Values (approximate) of several functions important for analysis of algorithms

n	$\sqrt{n}$	$log_2 n$	n	$n\ log_2 n$	n^2	n^3	2^n	$n!$
1	1	0	1	0	1	1	2	1
2	1.4	1	2	2	4	4	4	2
4	2	2	4	8	16	64	16	24
8	2.8	3	8	$2.4 \cdot 10^1$	64	$5.1 \cdot 10^2$	$2.6 \cdot 10^2$	$4.0 \cdot 10^4$
10	3.2	3.3	10	$3.3 \cdot 10^1$	10^2	10^3	10^3	$3.6 \cdot 10^6$
16	4	4	16	$6.4 \cdot 10^1$	$2.6 \cdot 10^2$	$4.1 \cdot 10^3$	$6.5 \cdot 10^4$	$2.1 \cdot 10^{13}$
10^2	10	6.6	10^2	$6.6 \cdot 10^2$	10^4	10^6	$1.3 \cdot 10^{30}$	$9.3 \cdot 10^{157}$
10^3	31	10	10^3	$1.0 \cdot 10^4$	10^6	10^9	Very big computation	
10^4	10^2	13	10^4	$1.3 \cdot 10^5$	10^8	10^{12}		
10^5	$3.2 \cdot 10^2$	17	10^5	$1.7 \cdot 10^6$	10^{10}	10^{15}		
10^6	10^3	20	10^6	$2.0 \cdot 10^7$	10^{12}	10^{18}		

Worst-Case, Best-Case, and Average-Case Efficiencies

Analysis of algorithm is the process of analyzing the problem-solving capability of the algorithm in terms of the time and size required (the size of memory for storage while implementation). However, the main concern of analysis of algorithms is the required time or performance. Generally, we perform the following types of analysis −

- **Worst-case** − The maximum number of steps taken on any instance of size **a**.
- **Best-case** − The minimum number of steps taken on any instance of size **a**.
- **Average case** − An average number of steps taken on any instance of size **a**.

EXAMPLE:

Consider the Sequential Search algorithm

SequentialSearch(A[0..n - 1], K)

//Searches for a given value in a given array by sequential search

//Input: An array A[0..n - 1] and a search key K

//Output: The index of the first element in A that matches K or -1 if there

is no match i ←0

while i < n and A[i] ≠

K do i ←i + 1

if i < n

return i else

return -1

Clearly, the running time of this algorithm can be quite different for the same list size n. In the worst case, there is no matching of elements or the first matching element can found at last on the list. In the best case, there is matching of elements at first on the list.

The **worst-case efficiency** of an algorithm is its efficiency for the worst case input of size n. The algorithm runs the longest among all possible inputs of that size.

The **best-case efficiency** of an algorithm is its efficiency for the best case input of size n. The algorithm runs the fastest among all possible inputs of that size n. In sequential search, If we search a first element in list of size n. (i.e. first element equal to a search key).

The **Average case efficiency** lies between best case and worst case. To analyze the algorithm's average case efficiency, we must make some assumptions about possible inputs of size n.

Asymptotic notation and Basic Efficiency Classes

Asymptotic Notation:

Asymptotic notations are the mathematical notations used to describe the running time of an algorithm when the input tends towards a particular value or a limiting value. Execution time of an algorithm depends on the instruction set, processor speed, disk I/O speed, etc. Hence, we estimate the efficiency of an algorithm asymptotically.

Let t(n) and g(n) can be any nonnegative functions defined on the set of natural numbers. The algorithm's running time t(n) usually indicated by its basic operation count C(n), and g(n), some simple function to compare with the count.

Different types of asymptotic notations are used to represent the complexity of an algorithm. Following asymptotic notations are used to calculate the running time complexity of an algorithm.

- **O** − Big Oh
- **Ω** − Big omega
- **θ** − Big theta

Big Oh (O): Asymptotic Upper Bound

'O' (Big Oh) is the most commonly used notation. A function ***f(n)*** can be represented is the order of ***g(n)*** that is ***O(g(n))***, if there exists a value of positive integer **n** as $\mathbf{n_0}$ and a positive constant **c** such that −

$$f(n) \leqslant c.g(n) \quad \text{for} \quad n > n_0 \quad \text{in all case}$$

Hence, function ***g(n)*** is an upper bound for function ***f(n)***, as ***g(n)*** grows faster than ***f(n)***.

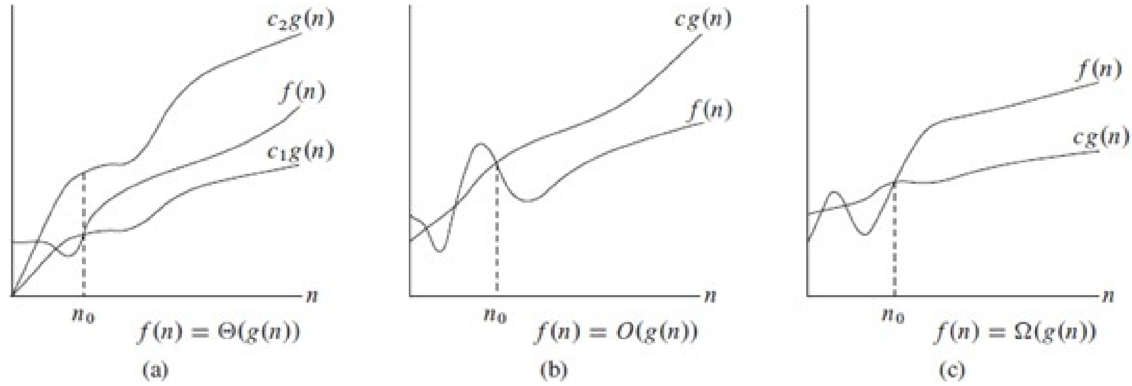
Example

Let us consider a given function, $f(n)=4.n^3+10.4.n^2+5.n+1$

Considering g(n)=n3,

$$f(n) \leqslant 5.g(n) \text{ for all the values of } n>2$$

Hence, the complexity of ***f(n)*** can be represented as O(g(n)), i.e. $O(4.n^3)$



Omega (Ω): Asymptotic Lower Bound

We say that f(n)=Ω(g(n)) when there exists constant **c** that f(n)⩾ c.g(n) for all sufficiently large value of **n**. Here **n** is a positive integer. It means function ***g*** is a lower bound for function ***f***; after a certain value of ***n, f*** will never go below ***g***.

Example

Let us consider a given function, f(n)= 4.n^3+10. 4.n^2+5.n+1

Considering g(n)= ^{3} , f(n)⩾ 4.g(n) for all the values of n>0

Hence, the complexity of ***f(n)*** can be represented as Ω(g(n)), i.e. Ω(^{3})

Theta (θ): Asymptotic Tight Bound

We say that f(n)=θ(g(n)) when there exist constants **c1** and **c2** that $_1$.g(n)⩽ f(n)⩽ $_2$.g(n) for all sufficiently large value of **n**. Here **n** is a positive integer. This means function **g** is a tight bound for function **f**.

Example

Let us consider a given function, f(n)= 4.n^3+10. 4.n^2+5.n+1

Considering g(n)= ^{3}, 4.g(n)⩽ f(n)⩽ 5.g(n) for all the large values of **n**.

Hence, the complexity of ***f(n)*** can be represented as θ(g(n)), i.e. θ(n^3).

Asymptotic Analysis

Solving Recurrence Equations

A recurrence is an equation or inequality that describes a function in terms of its value on smaller inputs. Recurrences are generally used in divide-and-conquer paradigm.

A recurrence relation can be solved using the following methods −

- **Substitution Method** − In this method, we guess a bound and using mathematical induction we prove that our assumption was correct.
- **Recursion Tree Method** − In this method, a recurrence tree is formed where each node represents the cost.
- **Master's Theorem** − This is another important technique to find the complexity of a recurrence relation.

General Plan for Analyzing the Time Efficiency of Recursive Algorithms

1. Decide on a parameter (or parameters) indicating an input's size.
2. Identify the algorithm's basic operation.
3. Check whether the number of times the basic operation is executed can vary on different inputs of the same size; if it can, the worst-case, average-case, and best-case efficiencies must be investigated separately.
4. Set up a recurrence relation, with an appropriate initial condition, for the number of times the basic operation is executed.
5. Solve the recurrence or, at least, ascertain the order of growth of its solution.

EXAMPLE 1: Compute the factorial function $F(n) = n!$ for an arbitrary non-negative integer n. Since $n! = 1 \bullet \ldots \bullet (n-1) \bullet n = (n-1)! \bullet n$, for $n \geq 1$ and $0! = 1$ by definition, we can compute $F(n) = F(n-1) \bullet n$ with the following recursive algorithm.

```
ALGORITHM Factorial(n)
//Computes n! Recursively
//Input: A nonnegative integer n
//Output: The value of n!
if n = 0 return 1
else return Factorial(n − 1) * n
```

Algorithm analysis

- For simplicity, we consider n itself as an indicator of this algorithm's input size. i.e. 1.
- The basic operation of the algorithm is multiplication, whose number of executions we denote M(n). Since the function F(n) is computed according to the formula $F(n) = F(n-1)*n$ for $n > 0$.
- The number of multiplications M(n) needed to compute it must satisfy the equality

$$M(n) = \underset{\substack{\uparrow \\ \text{To compute} \\ F(n-1)}}{M(n-1)} + \underset{\substack{\uparrow \\ \text{To multiply} \\ F(n-1) \text{ by } n}}{1} \quad \text{for } n > 0$$

  - $M(n-1)$ multiplications are spent to compute $F(n-1)$, and one more multiplication is needed to multiply the result by n.

Recurrence relations

The last equation defines the sequence M(n) that we need to find. This equation defines $M(n)$ not explicitly, i.e., as a function of n, but implicitly as a function of its value at another point, namely $n - 1$. Such equations are called recurrence relations or recurrences.

Solve the recurrence relation $M(n)=M(n-1)+1$, i.e., to find an explicit formula for $M(n)$ in terms of n only. To determine a solution uniquely, we need an initial condition that tells us the value with which the sequence starts. We can obtain this value by inspecting the condition that makes the algorithm stop its recursive calls:

if n = 0 return 1.

This tells us two things. First, since the calls stop when $n = 0$, the smallest value of n for which this algorithm is executed and hence $M(n)$ defined is 0. Second, by inspecting the pseudo code's exiting line, we can see that when $n = 0$, the algorithm performs no multiplications.

$M(0) = 0.$

the calls stop when $n = 0$ no multiplications when $n = 0$

Thus, the recurrence relation and initial condition for the algorithm's number of multiplications *M(n)*:

$M(n) = M(n - 1) + 1$ for $n > 0$,

$M(0) = 0$ for $n = 0$.

Method of backward substitutions

$$
\begin{aligned}
M(n) &= M(n-1) + 1 && \text{substitute } M(n-1) = M(n-2) + 1 \\
&= [M(n-2) + 1] + 1 \\
&= M(n-2) + 2 && \text{substitute } M(n-2) = M(n-3) + 1 \\
&= [M(n-3) + 1] + 2 \\
&= M(n-3) + 3 \\
&\ldots \\
&= M(n-i) + i \\
&\ldots \\
&= M(n-n) + n \\
&= n.
\end{aligned}
$$

Therefore $M(n)=n$

<u>General Plan for Analyzing the Time Efficiency of Non-recursive Algorithms</u>

1. Decide on a parameter (or parameters) indicating an input's size.
2. Identify the algorithm's basic operation (in the innermost loop).
3. Check whether the number of times the basic operation is executed depends only on the size of an input. If it also depends on some additional property, the worst-case, average-case, and, if necessary, best-case efficiencies have to be investigated separately.
4. Set up a sum expressing the number of times the algorithm's basic operation is executed.
5. Using standard formulas and rules of sum manipulation either find a closed form formula for the count or at the least, establish its order of growth.

EXAMPLE 1: Consider the problem of finding the value of the largest element in a list of n numbers. Assume that the list is implemented as an array for simplicity.

```
ALGORITHM MaxElement(A[0..n − 1])
//Determines the value of the largest element in a given array
//Input: An array A[0..n − 1] of real numbers
//Output: The value of the largest element in
A maxval ←A[0]
for i ←1 to n − 1 do
        if A[i]>maxval
        maxval←A[i]
End for
   return maxval
```

Algorithm Analysis

- The measure of an input's size here is the number of elements in the array, i.e., n.
- There are two operations in the for loop's body: o The comparison A[i]> maxval and o The assignment maxval←A[i].
- The comparison operation is considered as the algorithm's basic operation, because the comparison is executed on each repetition of the loop and not the assignment.
- The number of comparisons will be the same for all arrays of size n; therefore, there is no need to distinguish among the worst, average, and best cases here.
- Let C(n) denotes the number of times this comparison is executed. The algorithm makes one comparison on each execution of the loop, which is repeated for each value of the loop's variable i within the bounds 1 and n

$$c(n) = \sum_{i=1}^{n-1} 1$$

i.e., Sum up 1 in repeated n-1 times

$$c(n) = \sum_{i=1}^{n-1} 1 = n - 1 \in \Theta(n)$$

− 1, inclusive. Therefore, the sum for C(n) is calculated as follows:

EXAMPLE 2:

Consider the element uniqueness problem: check whether all the Elements in a given array of n elements are distinct.

```
ALGORITHM UniqueElements(A[0..n − 1])
//Determines whether all the elements in a given array are distinct
 //Input: An array A[0..n − 1]
//Output: Returns “true” if all the elements in A are distinct and
“false” otherwise
for i ←0 to n − 2 do
for j ←i + 1 to n − 1 do
        if A[i]= A[j ]
         return 0;
end for
end for
```

Algorithm Analysis:

- The measure of an input’s size here is the number of elements in the array, i.e., n.
- There are two operations in the for loop’s body: o The comparison A[i]> maxval and o The assignment maxval←A[i].
- The comparison operation is considered as the algorithm’s basic operation, because the comparison is executed on each repetition of the loop and not the assignment.
- The number of comparisons will be the same for all arrays of size n; therefore, there is no need to distinguish among the worst, average, and best cases here.
- Let C(n) denotes the number of times this comparison is executed. The algorithm makes one comparison on each execution of the loop, which is repeated for each value of the loop’s variable i within the bounds 1 and n

$$c(n) = \sum_{i=1}^{n-1} 1$$

i.e., Sum up 1 in repeated n-1 times

$$c(n) = \sum_{i=1}^{n-1} 1 = n - 1 \in \Theta(n)$$

− 1, inclusive. Therefore, the sum for C(n) is calculated as follows:

Algorithm Analysis:

The natural measure of the input's size here is again n (the number of elements in the array).

Since the innermost loop contains a single operation (the comparison of two elements), we should consider it as the algorithm's basic operation.

The number of element comparisons depends not only on n but also on whether there are equal elements in the array and, if there are, which array positions they occupy. We will limit our investigation to the worst case only.

One comparison is made for each repetition of the innermost loop, i.e., for each value of the loop variable j between its limits i + 1 and n − 1; this is repeated for each value of the outer loop, i.e., for each value of the loop variable i between its limits 0 and n −2

$$
\begin{aligned}
C_{worst}(n) &= \sum_{i=0}^{n-2} \sum_{j=i+1}^{n-1} 1 = \sum_{i=0}^{n-2}[(n-1)-(i+1)+1] = \sum_{i=0}^{n-2}(n-1-i) \\
&= \sum_{i=0}^{n-2}(n-1) - \sum_{i=0}^{n-2} i = (n-1)\sum_{i=0}^{n-2} 1 - \frac{(n-2)(n-1)}{2} \\
&= (n-1)^2 - \frac{(n-2)(n-1)}{2} = \frac{(n-1)n}{2} \approx \frac{1}{2}n^2 \in \Theta(n^2).
\end{aligned}
$$

Algorithm Design :

The important aspects of algorithm design include creating an efficient algorithm to solve a problem in an efficient way using minimum time and space.

To solve a problem, different approaches can be followed. Some of them can be efficient with respect to time consumption, whereas other approaches may be memory efficient. However, one has to keep in mind that both time consumption and memory usage cannot be optimized simultaneously. If we require an algorithm to run in lesser time, we have to invest in more memory and if we require an algorithm to run with lesser memory, we need to have more time.

Problem Development Steps

The following steps are involved in solving computational problems.

Problem definition

Development of a model

Specification of an Algorithm

Designing an Algorithm

Checking the correctness of an Algorithm

Analysis of an Algorithm

Implementation of an Algorithm

Program testing

Documentation