

# CSC13: Algorithmics and Program Design



*Dr. S. Zeeshan Hussain*  
*Dept. of Computer Science*  
*Jamia Millia Islamia*  
*New Delhi*

# Unit-Wise Syllabus

- **Algorithmic Problem Solving:** Algorithms; Problem Solving Aspect: Algorithm Devising, Design and Top-down Design; Algorithm Implementation: Essential and Desirable Features of an Algorithm; Efficiency of an Algorithm, Analysis of Algorithms, Pseudo codes; Algorithm Efficiency, Analysis and Order; Importance of Developing Efficient Algorithms; Complexity Analysis of Algorithms: Every-Case Time Complexity, Worst-Case Time Complexity, Average-Case Time Complexity, Best-Case Time Complexity.
- **Basic Algorithms** – Exchanging the Values of Two Variables, Counting, Summation of a Set of Numbers, Factorial Computation, Sine Function Computation, Generation of the Fibonacci Sequence, Reversing the Digits of an Integer, Base Conversion, etc. Flowchart. Flowchart – Symbols and Conventions, Recursive Algorithms.
- **Factoring:** Finding the square root of number, Smallest Divisor of an integer, Greatest common divisor of two integers, generating prime numbers, computing prime factors of an integer, Generation of pseudo random numbers, Raising a number to a large power, Computing the  $n$ th Fibonacci number.
- **Arrays, Searching and Sorting:** Single and Multidimensional Arrays, Array Order Reversal, Array counting, Finding the maximum number in a set, partitioning an array, Monotones Subsequence; Searching: Linear and Binary Array Search; Sorting: Sorting by selection, Exchange and Insertion. Sorting by diminishing increment, Sorting by partitioning.
- **Programming:** Introduction, Game of Life, Programming Style: Names, Documentation and Format, Refinement and Modularity; Coding, Testing and Further Refinement: Stubs and Drivers; Program Tracing, Testing, Evaluation; Program Maintenance: Program Evaluation, Review, Revision and Redevelopment; and Problem Analysis, Requirements Specification, Coding and Programming Principles.
- REFERENCES
- **Dromy: How to Solve by Computer, PE (Unit 1-4)**
- **Kruse: Data Structures and Program Design, PHI (Unit-5)**
- **Robertson: Simple Program Design, A Step-by-Step Approach, Thomson**

# Introduction to Course

- Understanding Problem-solving aspects
- Illustrate algorithmic terminology and issues; and analyze the efficiency of algorithms.
- Develop iterative/recursive flowcharts and algorithms for basic computational problems.
- Devise factoring algorithms, analyze and develop their improved versions.
- Implement and analyze different array-based searching and sorting algorithms.
- Design well-styled programs, trace and test the same.

# Problem Solving

- *Intelligence is one of the key characteristics which differentiate a human being from other living creatures on the earth. Basic intelligence covers day to day problem solving and making strategies to handle different situations which keep arising in day to day life.*
- Can you think of a day in your life which goes without problem solving? In our life we are bound to solve problems. In our day to day activity such as purchasing something from a general store and making payments, depositing fee in school, or withdrawing money from bank account.
- Now, broadly we can say that problem is a kind of barrier to achieve something and problem solving is a process to get that barrier removed by performing some sequence of activities.
- Here it is necessary to mention that all the problems in the world can not be solved. There are some problems which have no solution and these problems are called ***Open Problems***.

# Algorithmic Problem Solving

In *algorithmic problem solving*, we deal with objects. Objects are data manipulated by the algorithm. To a cook, the objects are the various types of vegetables, meat and sauce. In algorithms, the data are numbers, words, lists, files, and so on. In solving a geometry problem, the data can be the length of a rectangle, the area of a circle, etc. Algorithm provides the logic; data provide the values. They go hand in hand.

Hence, we have this great truth:

***Program = Algorithm + Data Structures***

Data structures refer to the types of data used and how the data are organised in the program. Data come in different forms and types. Most programming languages provides simple data types such as integers, real numbers and characters, and more complex data structures such as arrays, records and files which are collections of data.

# Computer based Problem Solving

Computer based Problem solving involves:

1. Programs and Algorithms
2. Requirements for solving problems by computer

# Problem Solving Aspects

- Problem Solving is a creative process
- There is not any universal method of Problem solving
- Computer Problem solving is about undersanding.

Problem-solving involves the following aspects:

1. Problem definition phase
2. Getting started on a problem
3. The use of specific examples
4. Similarities among problems
5. Working backwards from the solution
6. General Problem-solving strategies- Divide-and-conquer, Dynamic Programming etc.



# Problem Solving 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



# Algorithm

- An *algorithm* is a well-defined computational procedure consisting of a set of instructions, that takes some value or set of values, as input, and produces some value or set of values, as output.
- In other words, an *algorithm* is a procedure that accepts data, manipulate them following the prescribed steps, so as to eventually fill the required unknown with the desired value(s).



# Importance of Algorithm

- An algorithm is a set of steps of operations to solve a problem performing calculation, data processing, and automated reasoning tasks. An algorithm is an efficient method that can be expressed within finite amount of time and space.
- *An algorithm is the best way to represent the solution of a particular problem in a very simple and efficient way. If we have an algorithm for a specific problem, then we can implement it in any programming language, meaning that the **algorithm** is independent from any programming languages.*

# Algorithmics

An algorithm, whose characteristics will be discussed later, is a form that embeds the complete logic of the solution. Its formal written version is called a *program*, or *code*. Thus, ***algorithmic problem solving*** actually comes in **two phases**:

- 1. Derivation of an algorithm that solves the problem, and***
- 2. Conversion of the algorithm into code.***

The latter, usually known as coding, is comparatively easier, since the logic is already present – it is just a matter of ensuring that the syntax rules of the programming language are adhered to.

The first phase is what that stumbles most people, for two main reasons. Firstly, it challenges the mental faculties to search for the right solution, and secondly, it requires the ability to articulate the solution concisely into step by-step instructions, a skill that is acquired only through lots of practice.

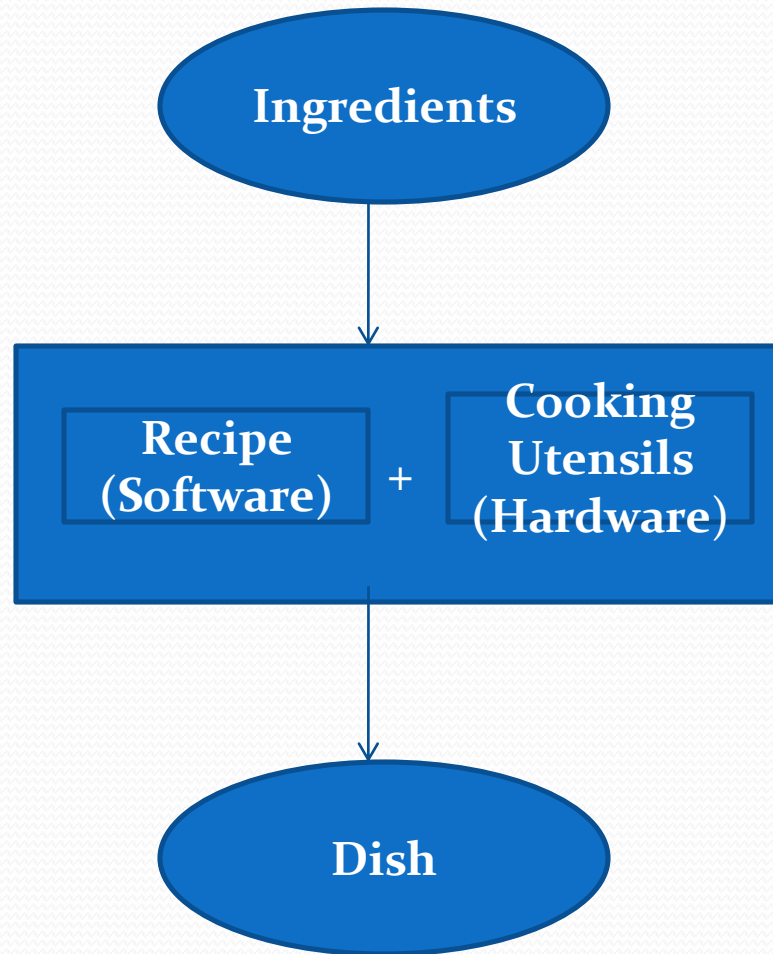
Many people are able to make claims like “oh yes, I know how to solve it”, but fall short when it comes to transferring the solution in their head onto paper.

Algorithms and their alter ego, programs, are the ***software***. The machine that runs the programs is the ***hardware***. Referring to the cook in our analogy again, his view can be depicted as follows:

# Algorithmics

- Human beings are quite good at executing algorithms. For example, children are taught at an early age how to execute long division in order to evaluate, say, 123456 divided by 78 and its remainder, and most soon become quite good at it. However, human beings are liable to make mistakes, and computers are much better than us at executing algorithms, at least so long as the algorithm is formulated precisely and correctly. The use of a computer to perform routine calculations is very effective.
- Formulating algorithms is a different matter. This is a task that few human beings practise and so we cannot claim to be good at it. But human beings are creative; computers, on the other hand, are incapable of formulating algorithms and even so-called “intelligent” systems rely on a human being to formulate the algorithm that is then executed by the computer.
- Algorithmic problem solving is about formulating the algorithms to solve a collection of problems. Improving our skills in algorithmic problem solving is a major challenge of the computer age.
- Formulating an algorithm makes problem solving decidedly harder, because it is necessary to formulate very clearly and precisely the procedure for solving the problem. The more general the problem, the harder it gets.

# Example



# Desirable Characteristics/Features of Algorithms

The main characteristics of algorithms are as follows –

- Algorithms must have a unique name
- Algorithms should have explicitly defined set of inputs and outputs
- Algorithms are well-ordered(in proper sequence) with unambiguous operations
- Algorithms halt in a finite amount of time. Algorithms should not run for infinity, i.e., an algorithm must end at some point
- Algorithms should be executable.

# Algorithm/Program constructs

**Sequential constructs-** Input/output statements, assignments, expressions

**Selection constructs-** if-else, switch

**Iterative constructs-** while, do while, for



# Examples of Algorithm

***Problem 1: Find the area of a Circle of radius  $r$ .***

Inputs to the algorithm: Radius  $r$  of the Circle.

Expected output: Area of the Circle

Algorithm:

Step1: Read\input the Radius  $r$  of the Circle

Step2: Area  $\leftarrow \text{PI} * r * r$  // calculation of area

Step3: Print Area

***Problem 2: Write an algorithm to read two numbers and find their sum.***

Inputs to the algorithm: First num1. Second num2.

Expected output: Sum of the two numbers.

Algorithm:

Step1: Start

Step2: Read\input the first num1.

Step3: Read\input the second num2.

Step4: Sum  $\text{num1} + \text{num2}$  // calculation of sum

Step5: Print Sum

Step6: End

## Examples of Algorithm contd ...

**Problem3:** write algorithm to find the result of equation:

$$f(x) = \begin{cases} -x, & x < 0 \end{cases}$$

$$x, & x \geq 0$$

- Step1: Start
- Step2: Read/input x
- Step3: If X Less than zero then  $F = -X$
- Step4: if X greater than or equal zero then  $F = X$
- Step5: Print F
- Step6: End

**Problem4:** An algorithm to find the largest value of any three numbers.

- Step1: Start
- Step2: Read/input A,B and C
- Step3: If  $(A \geq B)$  and  $(A \geq C)$  then  $Max = A$
- Step4: If  $(B \geq A)$  and  $(B \geq C)$  then  $Max = B$
- Step5: If  $(C \geq A)$  and  $(C \geq B)$  then  $Max = C$
- Step6: Print Max
- Step7: End

# Examples of Algorithm contd

**Problem6: An algorithm to calculate even numbers between 0 and 99**

- Step1. Start
- Step2.  $I \leftarrow 0$
- Step3. Write I in standard output
- Step4.  $I \leftarrow I+2$
- Step5. If ( $I \leq 98$ ) then go to line 3
- Step6. End

**Problem 7: Design an algorithm which gets a natural value, n as its input and calculates odd numbers equal or less than n. Then write them in the standard output.**

- Step1. Start
- Step2. Read n
- Step3.  $I \leftarrow 1$
- Step4. Write I
- Step5.  $I \leftarrow I + 2$
- Step6. If ( $I \leq n$ ) then go to line 4
- Step7. End








# Flowchart

The flowchart is a diagram which visually presents the flow of data through processing systems. This means by seeing a flow chart one can know the operations performed and the sequence of these operations in a system. Algorithms are nothing but sequence of steps for solving problems. So a flow chart can be used for representing an algorithm. A flowchart, will describe the operations (and in what sequence) are required to solve a given problem. You can see a flow chart as a blueprint of a design you have made for solving a problem.

For example suppose you are going for a picnic with your friends then you plan for the activities you will do there. If you have a plan of activities then you know clearly when you will do what activity. Similarly when you have a problem to solve using computer or in other word you need to write a computer program for a problem then it will be good to draw a flowchart prior to writing a computer program. Flowchart is drawn according to defined rules.

# Flowchart Symbols

- There are 6 basic symbols commonly used in flowcharting of assembly language Programs: Terminal, Process, input/output, Decision, Connector and Predefined Process.

Symbol	Name	Function
	Process	Indicates any type of internal operation inside the Processor or Memory
	Input/output	Used for any Input / Output (I/O) operation. Indicates that the computer is to obtain data or output results
	Decision	Used to ask a question that can be answered in a binary format (Yes/No, True/False)
	Connector	Allows the flowchart to be drawn without intersecting lines or without a reverse flow.
	Predefined Process	Used to invoke a subroutine or an Interrupt program.
	Terminal	Indicates the starting or ending of the program or process
	Flow Lines	Shows direction of flow.

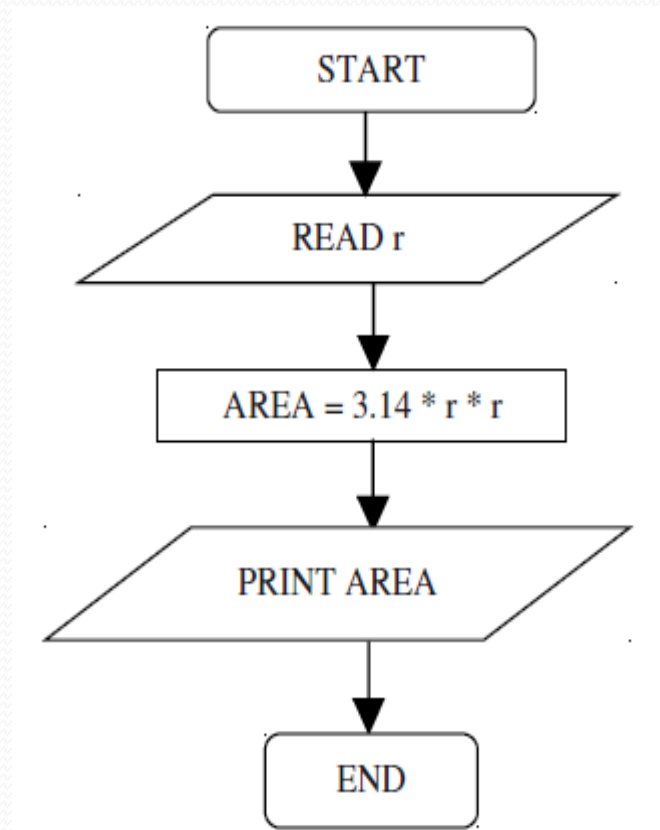
# General Rules for flowcharting

1. All boxes of the flowchart are connected with Arrows. (Not lines)
2. Flowchart symbols have an entry point on the top of the symbol with no other entry points.  
The exit point for all flowchart symbols is on the bottom except for the Decision symbol.
3. The Decision symbol has two exit points; these can be on the sides or the bottom and one side.
4. Generally a flowchart will flow from top to bottom. However, an upward flow can be shown as long as it does not exceed 3 symbols.
5. Connectors are used to connect breaks in the flowchart. Examples are:
  - From one page to another page.
  - From the bottom of the page to the top of the same page.
  - An upward flow of more than 3 symbols
6. Subroutines and Interrupt programs have their own and independent flowcharts.
7. All flow charts start with a Terminal or Predefined Process (for interrupt programs or subroutines) symbol.
8. All flowcharts end with a terminal or a contentious loop.

Flowcharting uses symbols that have been in use for a number of years to represent the type of operations and/or processes being performed. The standardised format provides a common method for people to visualise problems together in the same manner. The use of standardised symbols makes the flow charts easier to interpret, however, standardizing symbols is not as important as the sequence of activities that make up the process.

# Some examples of Flowcharts

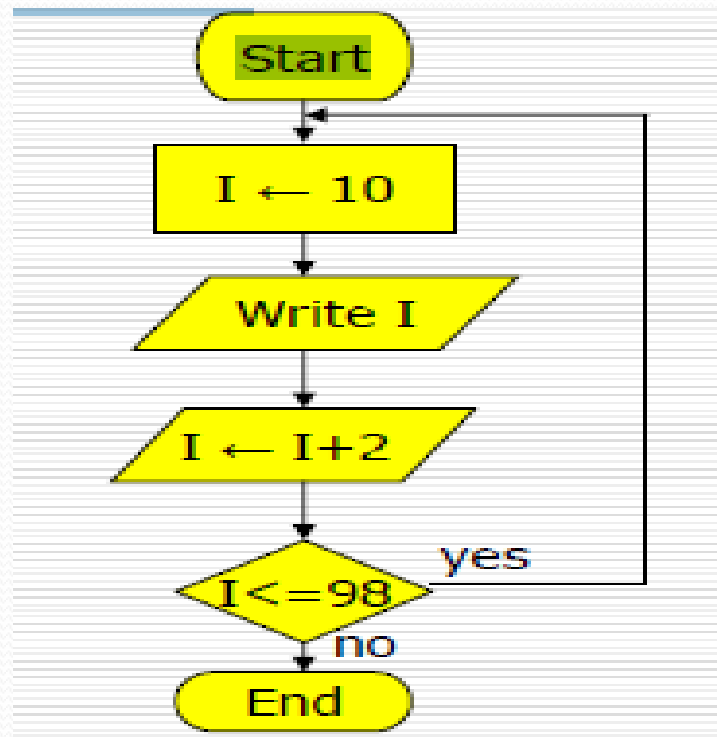
*To find the area of a circle of radius  $r$ .*





# Some examples of Flowcharts

*Printing even numbers between 9 and 100*



# 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.
- Our goal here is to study in depth the process of algorithm design with particular emphasis on the problem solving aspects of the task.

# Problem

- To generate Fibonacci Series : 1,1,2,3,5,8,13,21----

Step1:

$$x = 1^*$$

$$y = 1^*$$

$$z = x + y = 2^*$$

Step2:

$$x = y = 1$$

$$y = z = 2$$

$$z = x + y = 3^*$$

and so on.

# Factorial of a number

$$n! = n * n-1!$$

$$= n * (n-1) * n-2!$$

-

$$= n * (n-1) * (n-2) * n-3!$$

-----

$$0! = 1$$

- Iterative Algorithm
- Recursive Algorithm

$$\text{Fact}(n) = n * \text{Fact}(n-1)$$

$$= n * (n-1) * \text{Fact}(n-2)$$

-----

# Algorithm and Program

- The computer solution to a problem is a set of explicit and unambiguous instructions expressed in a programming language. This set of instructions is called a program.
- A program may also be thought of an algorithm expressed in a programming language.
- An algorithm therefore corresponds to a problem that is independent of any programming language.
- An algorithm consists of a set of explicit and unambiguous finite steps which when carried out for a given set of initial conditions produce the corresponding desired output and terminate in a finite time.

# Top-Down Design

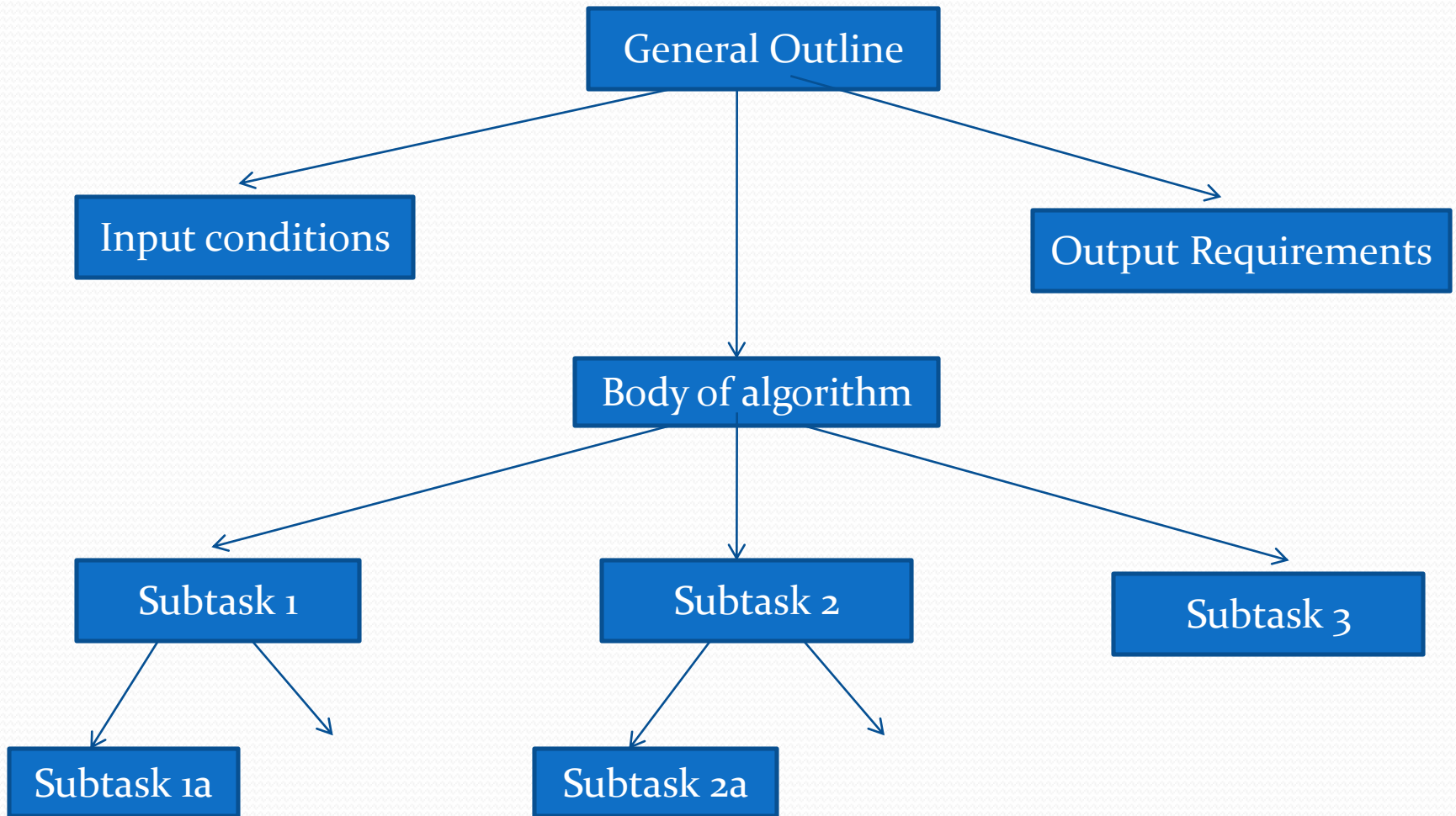
- The primary goal in computer based problem solving is an algorithm which is capable of being implemented as a correct and efficient computer program.
- We should consider those aspects of problem solving and algorithm design which are closer to the algorithm implementation.
- Once the problem to be solved is defined then question arises-how to solve it.
- The key to algorithm design is the ability to manage the inherent complexity of most problems that require computer solution.
- Two techniques used for algorithm design are:

## ***Top-down design and Bottom-up design***

Top-down design involves the following steps to handle the inherent logical complexity of computer algorithms.

- 1. Breaking a problem into sub problems**
- 2. Choice of a suitable data structure**
- 3. Construction of loops**
- 4. Establishing initial conditions for loops.**
- 5. Finding the iterative constructs**
- 6. Termination of loops**

# Top-Down Design





# Programming Languages

- The purpose of **programming languages** is to **formulate methods at a level of detail** where a computer can execute them. While we in textual descriptions of methods are often satisfied with describing what we wish to do, programming languages require considerably more constructive descriptions.
- Computers are quite basic creatures compared to us humans. They only understand a very limited set of instructions such as adding numbers, multiplying numbers, or moving data around within its memory. The syntax of programming languages often seems a bit arcane at first, but it grows on you with coding experience.
- e.g; Machine Language, Assembly Language, High-level Languages- C/C++, Java etc

# Pseudo Code

Somewhere in between describing algorithms in English text and in a programming language we find something called **pseudo code**. As **hinted by its** name it is not quite real code. The instructions we write are not part of the programming language of any particular computer.

The point of pseudo code is to be independent of what computer it is implemented on. Instead, it tries to convey the main points of an algorithm in a detailed manner so that it can easily be translated into any particular programming language. Secondly, we sometimes fall back to the liberties of the English language. At some point, we may decide that “choose the smallest number of a sequence” is clear enough for our audience.

With an explanation of this distinction in hand, let us look at a concrete example of what is meant by pseudo code.

# Pseudo code Example

```
Sub fizzbuzz()  
  For i = 1 to 100  
    print_number = True  
    If i is divisible by 3 Then  
      Print "Fizz"  
      print_number = False  
    End If  
    If i is divisible by 5 Then  
      Print "Buzz"  
      print_number = False  
    End If  
    If print_number = True Then  
      print i  
      Print a newline  
    End If  
  Next i  
End Sub
```

## Pseudo Code

We usually present algorithms in the form of some *pseudo-code*, which is normally a mixture of English statements, some mathematical notations, and selected keywords from a programming language. There is no standard convention for writing pseudo-code; each author may have his own style, as long as clarity is ensured.

Below are two versions of the same algorithm, one is written mainly in English, and the other in pseudo-code. The problem concerned is to find the minimum, maximum, and average of a list of numbers. Make a comparison.

# Algorithm and Pseudo code

- **Algorithm:**

1. First, you initialise *sum* to zero, *min* to a very Big number, and *max* to a very small number.
2. Then, you enter the numbers, one by one.
3. For each number that you have entered, assign it to *num* and add it to the *sum*.
4. At the same time, you compare *num* with *Min*, if *num* is smaller than *min*, let *min* be *num* instead.
5. Similarly, you compare *num* with *max*, if *Num* is larger than *max*, let *max* be *num* instead.
6. After all the numbers have been entered, You divide *sum* by the numbers of items entered, and let *ave* be the result.
7. End of algorithm.

- **Pseudo Code:**

*sum* <- 0   *count* <- 0 { *sum* = sum of numbers;  
*count* = how many numbers are entered? }

*Min* <- ? { *min* to hold the smallest value  
eventually }

*Max* <- ? { *max* to hold the largest value  
eventually }

for each *num* entered, increment *count*

*sum* <- *sum* + *num*

if *num* < *min* then *min* <- *num*

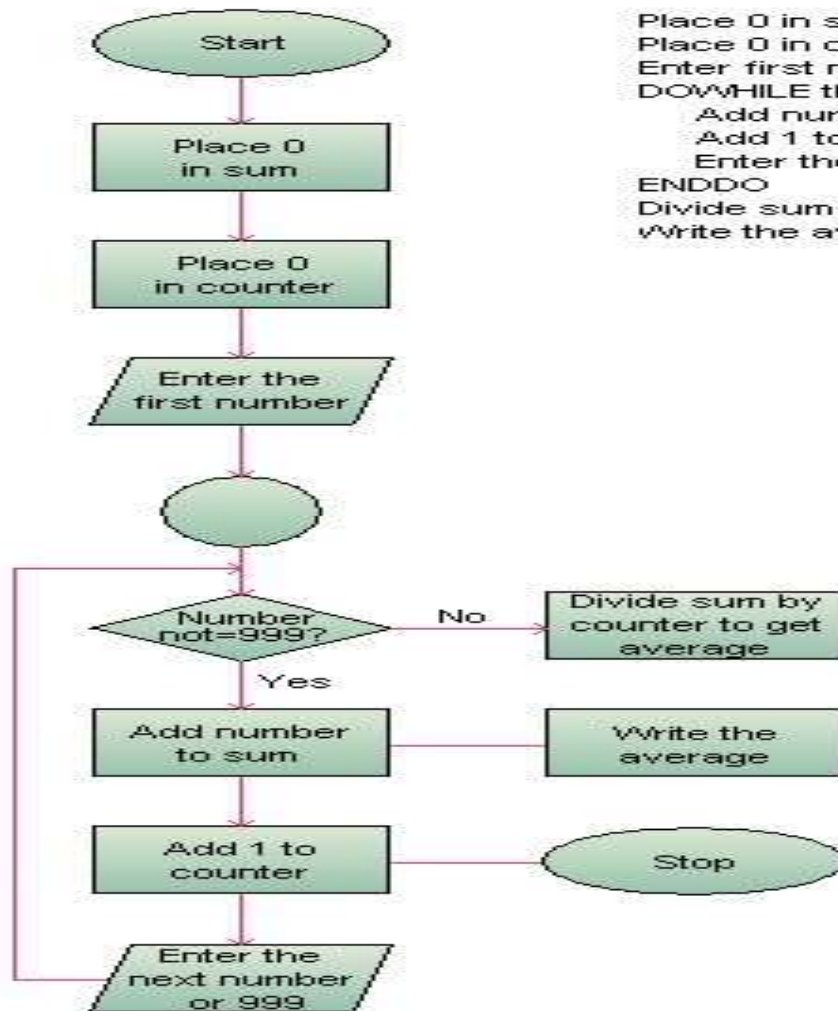
if *num* > *max* then *max* <- *num*

*Ave* <- *sum*/*count*

Note the use of indentation and symbols in the second version. What should *min* and *max* be initialised with?

# Flow Chart and Pseudo Code

**(a) Flowchart**



**(b) Pseudocode**

```
Place 0 in sum
Place 0 in counter
Enter first number
DOWHILE the number is not equal to 999
    Add number to sum
    Add 1 to counter
    Enter the next number or 999
ENDDO
Divide sum by counter to get ave
Write the average
```

# Implementation of Algorithms

- The implementation of an algorithm that has been properly designed in a top-down fashion, requires following considerations:
  1. Use of procedures to emphasize modularity
  2. Choice of variable names
  3. Documentation of programs
  4. Debugging programs
  5. Program testing



# Efficiency of Algorithms

- Efficiency considerations for algorithms are inherently related to *design, implementation and analysis* of algorithms.
- Every algorithm uses up some of computer's resources to complete its task. The resources related to efficiency are CPU time and internal memory.
- Because of the high cost of computing resources it is always desirable to design algorithms that are economical in the use of CPU time and memory. But a good design of algorithm is sometimes not possible due to the *complexity* of the problem.

# Designing Efficient Algorithms

- Designing algorithm to solve the problem efficiently is not so easy. It requires following useful points for consideration:
  1. Redundant computations
  2. Referencing array elements
  3. Inefficiency due to late termination
  4. Early detection of desired output conditions
  5. Trading storage for efficiency gains.

# Redundant computations

- Most of the inefficiencies that creep into the implementation of algorithms because redundant computations are made or unnecessary storage is used.

## Example:

### Version 1:

```
x = 0;
for (i=1;i<=n;i++) { // This loop does twice the no. of multiplication necessary.
    x = x + 0.01;
    y= (a*a*a + c) *x*x+ b*b*x;
    printf("x = %d, y= %dn",x,y);
}
```

### Version 2:

```
a3c = a*a*a + c;
b2= b*b;
x = 0;
for (i=1;i<=n;i++) {
    x = x + 0.01;y= a3c *x*x+ b2*x;
    printf("x = %d, y= %dn",x,y);
}
```

# Referencing array elements

- *Array processing sometime requires redundant computations.*

**Example**: Finding the maximum value and its position in an array.

Version 1:

```
p= 0;  
for (i=1;i<n;i++)  
if(a[i] > a[p]) p=i; // Use of a[p] requires two memory references and an  
addition operation to locate the correct value
```

```
max = a[p];
```

Version 1:

```
p= 0;  
max = a[0];  
for (i=1;i<n;i++)  
if(a[i] > max) { // Use of variable max requires only one memory reference  
max = a[i];  
p=i;  
}
```

# Inefficiency due to late termination

- *Linear search in array of names.*

Version1:

while name\_sought<> current\_name and no\_end\_of  
\_file do get next name from list

Version2:

while name\_sought> current\_name and no\_end\_of  
\_file do get next name from list

test if current\_name is equal to name\_sought then break

# Inefficiency due to late termination

## Example: Bubble sort

### Version 1:

```
for (i=0; i<n; i++)  
    for (j=0; j < n ; j++) // inner loop goes the full length of array  
        if (a[j]>a[j+1]) then exchange a[j] with a[j+1].
```

### Version 2:

```
for (i=0; i<n; i++)  
    for (j=0; j < n-i ; j++)  
        if (a[j]>a[j+1]) then exchange a[j] with a[j+1].
```

# Analysis of Algorithms

- What do you mean by a ‘good’ algorithm design?
- It has both qualitative and quantitative aspects.
- We are interested in a solution which is economical in the use of computing and human resources.
- We characterize an algorithm’s performance in terms of size ( $n$ ) of the problem being solved.
- Computation cost as a function of problem size ( $n$ ) is calculated as logarithmic dependence on  $n$  (better) at lower end to exponential dependence on  $n$  at higher end.
- We can analyze the computational complexities of an algorithm to solve a problem in terms of computing time and space (internal memory).

# Qualitative Aspects

**Good Algorithms usually possess the following qualities and capabilities:**

- They are simple but powerful.
- They can be easily understood that is the implementation is clear and concise
- They can be easily modified if necessary.
- They are correct for clearly defined situations
- They are economical in the use of computer time, computer storage and peripherals.
- They are documented well enough to be used by others
- They are not dependent on a particular machine.
- They are able to be used as subprocedure for other problems



# Quantitative Aspects (Measures)

- The qualitative aspects of a good algorithm are very important but it is also necessary to provide some qualitative measures to compute the evaluation of 'goodness' of an algorithm.
- Quantitative measures help in predicting the performance of an algorithm and to compare the relative performance of two or more algorithms.
- Use of an algorithm that is more efficient means a saving in computing resources which results into saving in time and money.

# Computational Complexity

- An algorithm's performance is measured in terms of  $\text{size}(n)$  of the problem being solved.
- More computing resources are needed to solve larger problems.
- How does the cost of solving the problem vary as  $n$  increases?
- Is it linearly dependent(cost vs size)?
- Linear dependence on  $n$  is true for some simple algorithms.
- At the lower end of the scale we have algorithms with logarithmic (or better) dependence on  $n$ , while at the higher end of the scale we have algorithms with an exponential dependence on  $n$ . With increasing  $n$  the relative difference in the cost of a computation is enormous for these two extremes.
- A Computational model is required for the evaluation of algorithms.
- The no. of operations, comparisons, exchanges, moves etc increase as  $n$  grows.

# Computational Complexity

- In designing of Algorithm, complexity analysis of an algorithm is an essential aspect. Mainly, algorithmic complexity is concerned about its performance, how fast or slow it works.
- 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**.



# Time and Space Complexities

## Time Complexity

- It is 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.

## Space Complexity

- It is 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.

# Asymptotic Analysis

- The asymptotic behavior of a function  $f(n)$  refers to the growth of  $f(n)$  as  $n$  gets large.
- We typically ignore small values of  $n$ , since we are usually interested in estimating how slow the program will be on large inputs.
- A good rule of thumb is that the slower the asymptotic growth rate, the better the algorithm. Though it's not always true.
- For example, a linear algorithm  $f(n)=d * n + k$  is always asymptotically better than a quadratic one,  $f(n)=c * n^2 + q$ .

# The Order Notation (Big O Notation)

## Big-O Analysis of Algorithms

- The Big O notation defines an upper bound of an algorithm, it bounds a function only from above. For example, consider the case of Insertion Sort. It takes linear time in best case and quadratic time in worst case. We can safely say that the time complexity of Insertion sort is  $O(n^2)$ . Note that  $O(n^2)$  also covers linear time.
- The Big-O Asymptotic Notation gives us the Upper Bound Idea, mathematically described below:

*$f(n) = O(g(n))$  if there exists a positive integer  $n_0$  and a positive constant  $c$ , such that  $f(n) \leq c \cdot g(n) \forall n \geq n_0$ .*

•

# Asymptotic Complexity

- The asymptotic complexity of an algorithm can be used to determine the size of problems it can solve using a conventional sequential computer.
- We can write the above relationships as,  
$$\lim_{n \rightarrow \infty} f(n)/g(n) = c$$
, where  $c$  is not equal to zero

**Example:** If an algorithm requires  $(3n^2 + 6n + 3)$  comparisons to complete its task then we have

$$f(n) = 3n^2 + 6n + 3 \text{ and } \lim_{n \rightarrow \infty} (3n^2 + 6n + 3)/n^2 = 3$$

It means the algorithm has an asymptotic complexity of  $O(n^2)$ .

*Constants of proportionality is important. It can happen that an algorithm with a higher asymptotic complexity and very small Constant of proportionality performs better for some particular range of  $n$  than algorithm with lower asymptotic complexity and a higher Constant of proportionality*



# Big-O Run-Time Analysis

The general step wise procedure for Big-O runtime analysis is as follows:

- Figure out what the input is and what  $n$  represents.
- Express the maximum number of operations, the algorithm performs in terms of  $n$ .
- Eliminate all excluding the highest order terms.
- Remove all the constant factors.

**Some of the useful properties on Big-O notation analysis are as follow:**

## **Constant Multiplication:**

*If  $f(n) = c.g(n)$ , then  $O(f(n)) = O(g(n))$  ; where  $c$  is a nonzero constant.*

## **Polynomial Function:**

*If  $f(n) = a_0 + a_1.n + a_2.n^2 + \dots + a_m.n^m$ , then  $O(f(n)) = O(n^m)$ .*

## **Summation Function:**

*If  $f(n) = f_1(n) + f_2(n) + \dots + f_m(n)$  and  $f_i(n) \leq f_{i+1}(n) \forall i=1, 2, \dots, m$ , then  $O(f(n)) = O(\max(f_1(n), f_2(n), \dots, f_m(n)))$ .*

## **Logarithmic Function:**

*If  $f(n) = \log_a n$  and  $g(n) = \log_b n$ , then  $O(f(n)) = O(g(n))$   
; all log functions grow in the same manner in terms of Big-O.*

Basically, this asymptotic notation is used to measure and compare the worst-case scenarios of algorithms theoretically. For any algorithm, the Big-O analysis should be straightforward as long as we correctly identify the operations that are dependent on  $n$ , the input size.



# Runtime Analysis of Algorithms

- In general cases, we mainly used to measure and compare the worst-case theoretical running time complexities of algorithms for the performance analysis.
- The fastest possible running time for any algorithm is  $O(1)$ , commonly referred to as *Constant Running Time*. In this case, the algorithm always takes the same amount of time to execute, regardless of the input size. This is the ideal runtime for an algorithm, but it's rarely achievable. In actual cases, the performance (Runtime) of an algorithm depends on  $n$ , that is the size of the input or the number of operations is required for each input item.

# Classification of Algorithms

The algorithms can be classified as follows from the best-to worst performance (Running Time Complexity):

***A logarithmic algorithm –  $O(\log n)$***

*Runtime grows logarithmically in proportion to  $n$ .*

***A linear algorithm –  $O(n)$***

*Runtime grows directly in proportion to  $n$ .*

***A superlinear algorithm –  $O(n \log n)$***

*Runtime grows in proportion to  $n$ .*

***A polynomial algorithm –  $O(n^c)$***

*Runtime grows quicker than previous all based on  $n$ .*

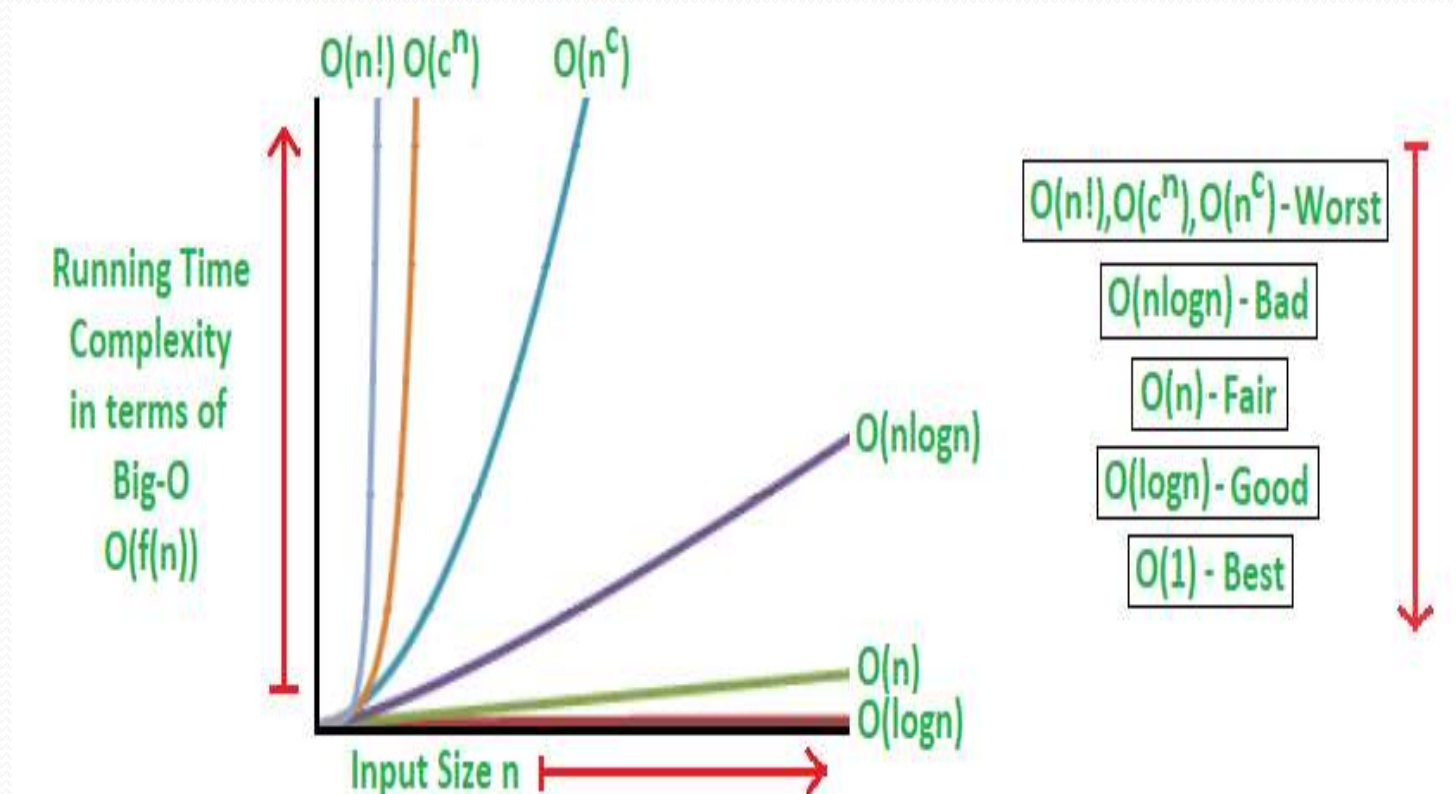
***An exponential algorithm –  $O(c^n)$***

*Runtime grows even faster than polynomial algorithm based on  $n$ .*

***A factorial algorithm –  $O(n!)$***

*Runtime grows the fastest and becomes quickly unusable for even small values of  $n$ . Where,  $n$  is the input size and  $c$  is a positive constant.*

# Running Time Complexity



# Cost Vs Size

Computational cost as a function of problem size(n) is shown in the table hereunder.

$\log_2 n$	N	$n \log_2 n$	$N^2$	$N^3$	$2^n$
1	2	2	4	8	4
3.323	10	33.22	$10^2$	$10^3$	$>10^3$
6.644	$10^2$	664.4	$10^4$	$10^6$	$>>10^{23}$
9.966	$10^3$	9966.0	$10^6$	$10^9$	$>> 10^{250}$
13.287	$10^4$	132,887	$10^8$	$10^{12}$	$>> 10^{2500}$

# Algorithmic Examples of Runtime Analysis

Some of the examples of all those types of algorithms (in worst-case scenarios) are mentioned below:

***Logarithmic algorithm*** –  $O(\log n)$  – Binary Search.

***Linear algorithm*** –  $O(n)$  – Linear Search.

***Superlinear algorithm*** –  $O(n \log n)$  – Heap Sort, Merge Sort.

***Polynomial algorithm*** –  $O(n^c)$  – Strassen's Matrix Multiplication, Bubble Sort, Selection Sort, Insertion Sort, Bucket Sort.

***Exponential algorithm*** –  $O(c^n)$  – Tower of Hanoi.

***Factorial algorithm*** –  $O(n!)$  – Determinant Expansion by Minors, Brute force Search algorithm for Traveling Salesman Problem.

# Mathematical Examples of Runtime Analysis

The performances (Runtimes) of different orders of algorithms vary rapidly as  $n$  (the input size) gets larger. Let's consider the mathematical example:

If  $n = 10$ ,

$\log(10) = 1$ ;

$10 = 10$ ;

$10\log(10)=10$ ;

$10^2=100$ ;

$2^{10}=1024$ ;

$10!=3628800$ ;

If  $n=20$ ,

$\log(20) = 2.996$ ;

$20 = 20$ ;

$20\log(20)=59.9$ ;

$20^2=400$ ;

$2^{20}=1048576$ ;

$20!=2.432902e+18^{18}$ ;

# Analysis Cases

- 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.
- 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  $n$ .
  - **Best-case** – The minimum number of steps taken on any instance of size  $n$ .
  - **Average case** – An average number of steps taken on any instance of size  $n$ .

Let us consider the following implementation of Linear Search.

# Linear Search Example

- C implementation

```
#include <stdio.h>
// Linearly search x in arr[].
// If x is present then return the index,
// otherwise return -1
int search(int arr[], int n, int x)
{
    int i;
    for (i = 0; i < n; i++) {
        if (arr[i] == x)
            return i;
    }
    return -1;
}
```

```
int main()
{
    int arr[] = { 1, 10, 30, 15 };
    int x = 30;
    int n = sizeof(arr) / sizeof(arr[0]);
    cout << x << " is present at index "
        << search(arr, n, x);

    getch();
    return 0;
}
```

## Output:

30 is present at index 2



## Worst Case Analysis (Usually Done)

In the worst case analysis, we calculate upper bound on running time of an algorithm. We must know the case that causes maximum number of operations to be executed.

For Linear Search, the worst case happens when the element to be searched ( $x$  in the above code) is not present in the array. When  $x$  is not present, the `search()` function compares it with all the elements of `arr[]` one by one. Therefore, the worst case time complexity of linear search would be  $\Theta(n)$ .

- Most of the times, we do worst case analysis to analyze algorithms. In the worst analysis, we guarantee an upper bound on the running time of an algorithm which is good information.

# Average Case Analysis (Sometimes done)

- The average case analysis is not easy to do in most of the practical cases and it is rarely done. In the average case analysis, we must know (or predict) the mathematical distribution of all possible inputs.
- In average case analysis, we take all possible inputs and calculate computing time for all of the inputs. Sum all the calculated values and divide the sum by total number of inputs. We must know (or predict) distribution of cases. For the linear search problem, let us assume that all cases are uniformly distributed (including the case of x not being present in array). So we sum all the cases and divide the sum by (n+1). Following is the value of average case time complexity.
- Average Case Time =
- $$= \frac{\sum_{i=1}^{n+1} \theta(i)}{(n+1)}$$
- $$= \frac{\theta((n+1)*(n+2)/2)}{(n+1)}$$
- $$= \Theta(n)$$

## Best Case Analysis (Rare)

In the best case analysis, we calculate lower bound on running time of an algorithm. We must know the case that causes minimum number of operations to be executed. In the linear search problem, the best case occurs when  $x$  is present at the first location. The number of operations in the best case is constant (not dependent on  $n$ ). So time complexity in the best case would be  $\Theta(1)$  which is rarely achieved.

# Summary of Analysis

Most of the times, we do worst case analysis to analyze algorithms.

- In the worst analysis, we guarantee an upper bound on the running time of an algorithm which is good information.
- The average case analysis is not easy to do in most of the practical cases and it is rarely done. In the average case analysis, we must know (or predict) the mathematical distribution of all possible inputs.
- The Best Case analysis is rarely done. Guaranteeing a lower bound on an algorithm doesn't provide any information as in the worst case, an algorithm may take years to run. For some algorithms, all the cases are asymptotically same, i.e., there are no worst and best cases.
- **For example, Merge Sort does  $\Theta(n \log n)$  operations in all cases.** Most of the other sorting algorithms have worst and best cases.
- **For example, in the typical implementation of Quick Sort (where pivot is chosen as a corner element), the worst occurs when the input array is already sorted and the best occur when the pivot elements always divide array in two halves. For insertion sort, the worst case occurs when the array is reverse sorted and the best case occurs when the array is sorted in the same order as output.**

# Space Complexity

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

# Memory Footprint Analysis of Algorithms

- For performance analysis of an algorithm, runtime measurement is not only relevant metric but also we need to consider the memory usage amount of the program. This is referred to as the Memory Footprint of the algorithm, shortly known as Space Complexity. Here also, we need to measure and compare the worst case theoretical space complexities of algorithms for the performance analysis.

**It basically depends on two major aspects described below:**

- Firstly, the implementation of the program is responsible for memory usage. For example, we can assume that recursive implementation always reserves more memory than the corresponding iterative implementation of a particular problem.
- And the other one is  $n$ , the input size or the amount of storage required for each item. For example, a simple algorithm with a high amount of input size can consume more memory than a complex algorithm with less amount of input size.

## Algorithmic Examples of Memory Footprint Analysis

- The algorithms with examples are classified from the best-to-worst performance (Space Complexity) based on the worst-case scenarios are mentioned below:
- Ideal algorithm -  $O(1)$  - Linear Search, Binary Search,
- Bubble Sort, Selection Sort, Insertion Sort, Heap Sort, Shell Sort.
- Logarithmic algorithm -  $O(\log n)$  - Merge Sort.
- Linear algorithm -  $O(n)$  - Quick Sort.
- Sub-linear algorithm -  $O(n+k)$  - Radix Sort.



# Space-Time Tradeoff and Efficiency

- There is usually a trade-off between optimal memory use and runtime performance.
- In general for an algorithm, space efficiency and time efficiency reach at two opposite ends and each point in between them has a certain time and space efficiency. So, the more time efficiency you have, the less space efficiency you have and vice versa. **For example**, Merge sort algorithm is exceedingly fast but requires a lot of space to do the operations. On the other side, Bubble Sort is exceedingly slow but requires the minimum space.
- In this context, if we compare **bubble sort** and **merge sort**. Bubble sort does not require additional memory, but merge sort requires additional space. Though time complexity of bubble sort is higher compared to merge sort, we may need to apply bubble sort if the program needs to run in an environment, where memory is very limited.
- **At the end of this topic, we can conclude that finding an algorithm that works in less running time and also having less requirement of memory space, can make a huge difference in how well an algorithm performs.**



# Recurrence Relation

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.

Let us consider  $T(n)$  to be the running time on a problem of size  $n$ .

If the problem size is small enough, say  $n < c$  where  $c$  is a constant, the straightforward solution takes constant time, which is written as  $\theta(1)$ . If the division of the problem yields a number of sub-problems with size  $nb$ .

To solve the problem, the required time is  $a.T(n/b)$ . If we consider the time required for division is  $D(n)$  and the time required for combining the results of sub-problems is  $C(n)$ , the recurrence relation can be represented as –

$T(n) = \begin{cases} \theta(1) & \text{if } n \leq c \\ a.T(n/b) + D(n) + C(n) & \text{otherwise} \end{cases}$

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.

## Example of Factorial of a number

- For the recursive solution, there will be  $n$  recursive calls, so there will be  $n$  stacks used; one for each call. Hence the  $O(n)$  space. This is not the case for the iterative solution - there's just one stack and you're not even using an array, there is only one variable. So the space complexity is  $O(1)$ .
- For the time complexity of the iterative solution, you have  $n$  multiplications in the for loop, so a loose bound will be  $O(n)$ . Every other operation can be assumed to be unit time or constant time, with no bearing on the overall efficiency of the algorithm. The time complexity for the recursive solution will also be  $O(n)$  as the recurrence is  $T(N) = T(N-1) + O(1)$ , assuming that multiplication takes constant time. Clearly this means the time Complexity is  $O(n)$ .

# Fibonacci Series Example

## Algorithm1:F(n) //Iterative

Input: Some non-negative integer n

Output: The nth no. in the Fibonacci sequence

if  $n \leq 1$  then

return n

else

return  $F(n-1) + F(n-2)$

## Algorithm2:F(n) //Recursive

Input: Some non-negative integer n

Output: The nth no. in the Fibonacci sequence

$A[0] \leftarrow 0$

$A[1] \leftarrow 1$

For  $i \leftarrow 2$  to  $n-1$  do

$A[i] \leftarrow A[i-1] + A[i-2]$

return  $A[n-1]$

# Time Complexity for Recursive version

Let's use  $T(n)$  to denote the time complexity of  $F(n)$ .

The number of additions required to compute  $F(n-1)$  and  $F(n-2)$  will then be  $T(n-1)$  and  $T(n-2)$ , respectively. We have one more addition to sum our results. Therefore, for  $n > 1$ :

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

When  $n = 0$  and  $n = 1$ , no additions occur. This implies that:

$$T(0) = T(1) = 0$$

Now that we have our equation, we need to solve for  $T(n)$ .

Let's start by assuming that  $T(n-2) \approx T(n-1)$ . Don't worry about *why* just yet – this will become apparent shortly.

Substituting the value of  $T(n-1) = T(n-2)$  into our relation  $T(n)$ , we get:

$$T(n) = T(n-1) + T(n-1) + 1 = 2 * T(n-1) + 1$$

**By doing this, we have reduced  $T(n)$  into a much simpler recurrence. As a result, we can now solve for  $T(n)$  using backward substitution.**

# Time Complexity for Recursive version

## Solving $T(n)$ Using Backward Substitution

To do this, we first substitute  $T(n-1)$  into the right-hand side of our recurrence.

Since  $T(n-1) = 2 * T(n-2) + 1$ , we get:

$$T(n) = 2 * [2 * T(n-2) + 1] + 1 = 4 * T(n-2) + 3$$

Next, we can substitute in  $T(n-2) = 2 * T(n-3) + 1$ :

$$T(n) = 2 * [2 * [2 * T(n-3) + 1] + 1] + 1 = 8 * T(n-3) + 7$$

And once more for  $T(n-3) = 2 * T(n-4) + 1$ :

$$T(n) = 2 * [2 * [2 * [2 * T(n-4) + 1] + 1] + 1] + 1 = 16 * T(n-4) + 15$$

We can see a pattern starting to emerge here, so let's attempt to form a general solution for  $T(n)$ . It appears to stand that:

$$T(n) = 2^k * T(n-k) + (2^k - 1)$$

For any positive integer  $k$ . We can prove this equation holds through simple induction – for brevity's sake, we'll skip this process.

Finally, we can find  $k$  and, thereby, solve  $T(n)$ , by substituting in  $T(0) = 1$ .

For  $T(0)$ , we can see that  $n - k = 0$ . Rearranging, we get  $k = n$ . Now, substituting in our values for  $T(0)$  and  $k$ , we get:

$$T(n) = 2^n * T(0) + (2^n - 1) = 2^n + 2^n - 1 = O(2^n)$$

# Time Complexity for Iterative version

- Analyzing the time complexity for our iterative algorithm is a lot more straightforward than its recursive counterpart.  
In this case, our most costly operation is assignment. Firstly, our assignments of  $F[0]$  and  $F[1]$  cost  $O(1)$  each. Secondly, our loop performs one assignment per iteration and executes  $(n-1)-2$  times, costing a total of  $O(n-3) = O(n)$ .
- **Therefore, our iterative algorithm has a time complexity of  $O(n-2) + O(1) + O(1) = O(n)$ .**
- This is a marked improvement from our recursive algorithm.

## Space Complexity

- For the **iterative** approach, the amount of space required is the same for `fib(6)` and `fib(100)`, i.e. as  $N$  changes the space/memory used remains the same. Hence its space **complexity** is  **$O(1)$  or constant**. As you can see the maximum depth is proportional to the  $N$ , hence the space **complexity** of **Fibonacci recursive** is  **$O(N)$** .



# *Q & A*