UNIT- I
Introduction to Algorithms & Data Structures

Contents

- **Algorithms-** Problem Solving, Introduction to Algorithms, Characteristics of algorithms, Algorithm design tools: Pseudo code and flowchart, Analysis of Algorithms, Complexity Of algorithms- Space complexity, Time complexity, asymptotic notation, standard measures of Efficiency.
- **Data Structures-** Data structure, Abstract Data Types (ADT), Concept of linear and Non-linear, Static and dynamic, persistent and ephemeral data structures, and relationship among data, data Structures, and algorithms, From Problem to Program, Algorithmic Strategies- Introduction to Algorithm design strategies- Divide and Conquer, and Greedy strategy.

Introduction

- People Make Decisions everyday to solve their problems that affect their lives
- If bad decision is made time and resources are wasted
- Six steps for problem Solving
  - Identify the problem
  - Understand the problem
  - Identify alternative ways to solve the problem
  - Select the best way to solve the problem
  - List instructions that enable you to solve the problem using the selected solution
  - Evaluate the solution

Example: Problem what to do this evening

1. **Indentify the Problem:** How do the Individuals wish to spend th
2. **Understand the problem:** The knowledge base of the participants must be considered**.** The only solutions that should be selected are ones that everyone would know how to do. You probably would not select as a possible solution playing a game of chess if the participants do not know how to play chess.
3. **Identify Alternatives**:
   - Watch Television
   - Invite friends over
   - Play games
   - Go to the movie
   - Play miniature golf
   - Go to a friends party
   - Go to the amusement park
   - List is complete only when you can think of no more alternatives
4. **Select best way to solve the problem**
   - Cut out alternatives that are not acceptable
   - Specify pros and Cons of each Remaining alternative
   - Compare pros and cons to make final decision
5. **Prepare a list of steps that will result in a fun evening.**
6. **Evaluate the solution. Are we having fun yet?**

General Problem-Solving Concepts

Types of Problems

- Problems can be solved with series of actions
  - **Algorithmic Solution**
- a commonsense rule (or set of rules) intended to increase the probability or speed of solving some problem
  - **Heuristic solutions**
- Problem solver can use six steps for both algorithmic and heuristic solutions
- Most problems require a combination of the two kinds of solutions

Problem solving with computers

- **Results** means the outcome or the completed computer assisted answer
- **Program** means the set of instructions that make up the solution using programming language
- Computers are built to deal with algorithmic solutions
- Difficulty lies in Programming
- Solutions must be transformed into an algorithmic format

Problem Solving Concepts for the Computer

Data

- ***Data*** is nothing but a piece of information.
- Data input, data manipulation (or data processing), and data output are the functions of computers.
- Hence all information taken as input, processed within a computer, or provided as output to the user is nothing but data. It can be a number, a string, or a set of many numbers and strings.

Atomic and Composite Data

- ***Atomic data*** is the data that we choose to consider as a single, non-decomposable entity. For example,
- The integer 1234 may be considered as a single integer value. Of course, we can decompose it into digits, but the decomposed digits will not have the same characteristics of the original integer; they will be four single digit integers ranging from 0 to 9.

Composite data

- The opposite of atomic data is *composite data*. Composite data can be broken down into subfields that have meaning.

For example,

- a student's record consists of Roll_Number, Name, Branch, Year, and so on. Composite data is also referred to as structured data and can be implemented using a structure or a class in C++.

Data Type

- *Data type* refers to the kind of data a variable may store. Whenever we try to implement any algorithm in some programming language, we need variables.
- A variable may have any value as per the facilities provided by that language. Data type is a term that specifies the type of data that a variable may hold in the programming language.

Built-in Data Types

- In general, languages have their built-in data types. However, they also allow the user to define his or her own data types, called *user-defined data types*, using the built-in data types.
- for example
- In the C/C++ languages, int, float, and char are built-in data types. Using these built-in data types, we can design (define) our own data types by means of structures, unions, and classes.

User-defined Data Types

- Suppose we want to maintain a record of 100 students with the following fields in each record: roll number, name of student, and percentage of marks of the students. Then we use the C++ class as follows:

```
class Student
{
 private:
        int roll;
 public:
      void GetRecord();
      void PrintRecord();
}
```

- Class, structure, and union are the user-defined data types in C++.

Data Types

- To process solutions computer must have data.
- Data are unorganized facts
- Computer Must be told the data types of each variable and constant

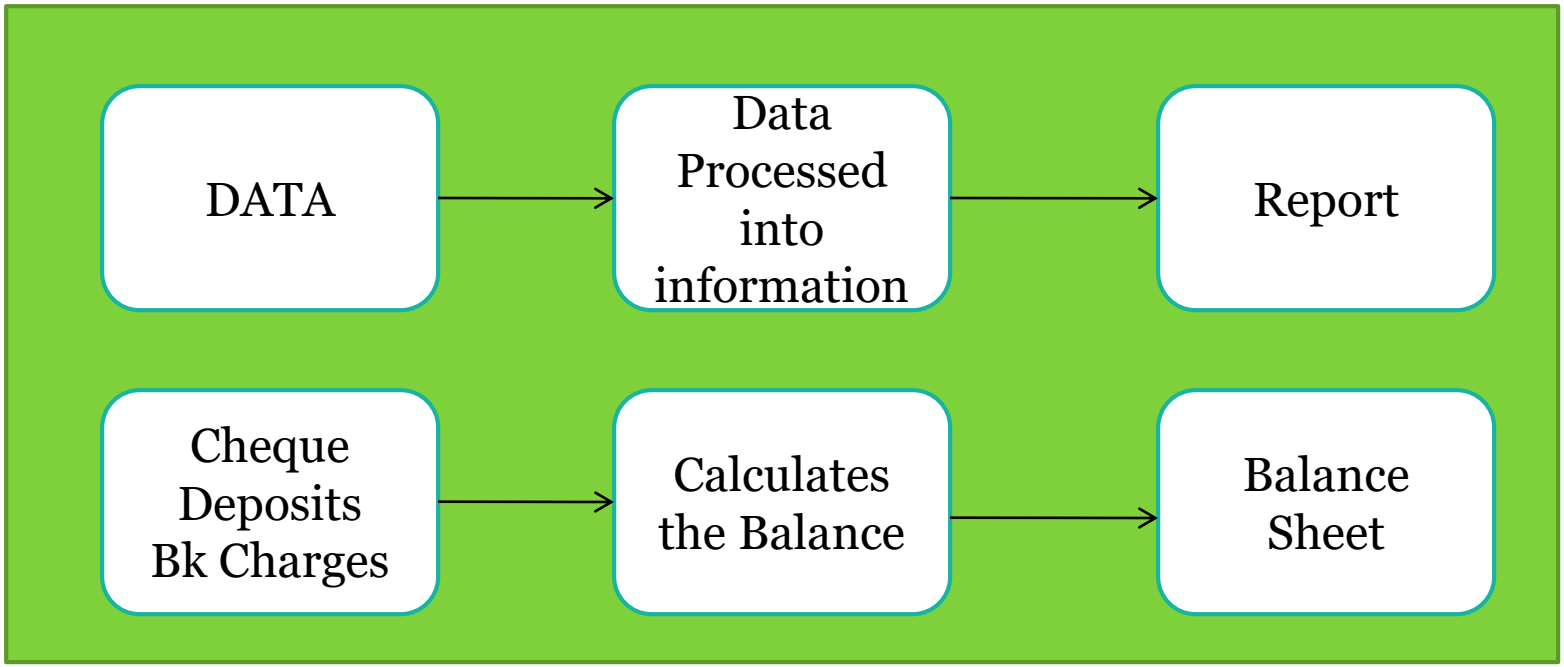


Fig. Processing Data: How a computer Balances a Checkbook

Data Type	Data Sets	
Numeric: Integer	All whole numbers	3400 -34
Numeric: Real	All real Numbers (Whole + Decimal)	3256.344 12333.0 0.32323
Character : Surrounded by quotation mark	All letters, Numbers and Special Symbols	‘a, ‘1’, ‘A’, ‘F’, ‘%’, ‘&’ ‘$’
String: Surrounded by quotation mark	Combination of more than one character	“Rahul” “343443” “7070-3232-232”
Logical	True, False	True, False

Rules for Data Types

1. The data that define the value of a variable or a constant will be of numeric, character, logical
2. Computer then associate variable with desired data type
3. Data types can not be mixed
4. Each of the data types uses what is called a data sets
5. Any numeric item that must be used in calculations resulting in a numeric result must be of numeric type

Algorithm

"Algorithm is any well defined computational procedure that takes some values or set of values as input and produces some value or a set of values as output"

Characteristics of algorithm

- **Input** : algorithm is supplied with zero or more external quantities
- **Output**: Algorithm must produce one or more results/output
- **Definiteness**: Each step in an algorithm must be clear and unambiguous
- **Finiteness:** Must have finite number of steps
- **Effectiveness**: Every instruction must be sufficiently basic
- **Feasibility** − Should be feasible with the available resources.
- **Independent** − An algorithm should have step-by-step directions, which should be independent of any programming code.
-

- Design an algorithm to add two numbers and display the result.
- **Step 1** − START
- **Step 2** − declare three integers **a**, **b** & **c**
- **Step 3** − define values of **a** & **b**
- **Step 4** − add values of **a** & **b**
- **Step 5** − store output of step 4 to **c**
- **Step 6** − print **c**
- **Step 7** − STOP

- Algorithms tell the programmers how to code the program.

  Alternatively, the algorithm can be written as −

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

Pseudocode

1. Pseudocode Notations
2. Algorithm Header
3. Purpose
4. Condition and Return Statements
5. Statement Numbers
6. Variables
7. Statement Constructs
   - 1. sequence
   - 2. decision
   - 3. repetition
8. Subalgorithms

Algorithm search (val list<array>,val X<integer>)

Pre list containing data array to be searched and argument containing data to be located

Post None

Return Location

1. Let list be the array and X be the element to be searched

2. For I = 1 to N do

```
    begin
    if(List(I) = X)
    then
        Return I
    end if
end
```

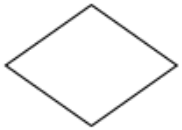
3. Return -I

4. Stop

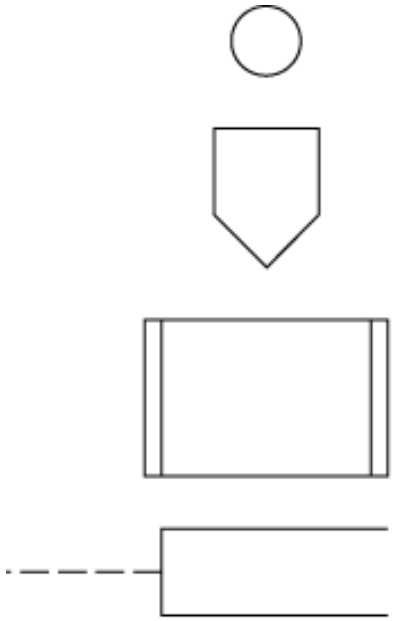
Drawing Flowchart

- A very effective tool to show the logic flow of a program is the flowchart.
- A flowchart is a pictorial representation of an algorithm.
- It hides all the details of an algorithm by giving a picture; it shows how the algorithm flows from beginning to end.
- The primary purpose of a flowchart is to show the design of the algorithm.

Flowchart symbols

Symbol	Name	Meaning
	Flowline	Used to connect symbols and indicate the flow of logic.
	*Terminal*	Used to represent the beginning (Start) or the end (End) of a task.
	*Input/Output*	Used for input and output operations, such as reading and displaying. The data to be read or displayed are described inside.
	*Processing*	Used for arithmetic and data-manipulation operations. The instructions are listed inside the symbol.
	*Decision*	Used for any logic or comparison operations. Unlike the input/ouput and processing symbols, which have one entry and one exit flowline, the decision symbol has one entry and two exit paths. The path chosen depends on whether the answer to a question is "yes" or "no."

Flowchart symbols continued

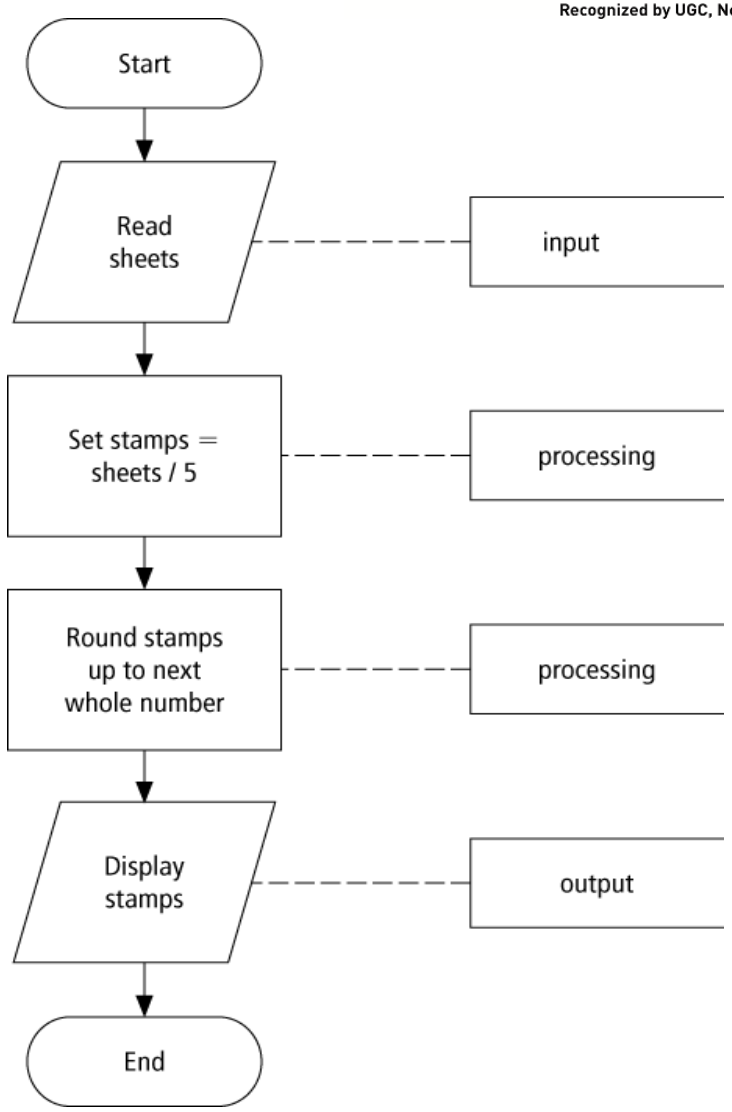


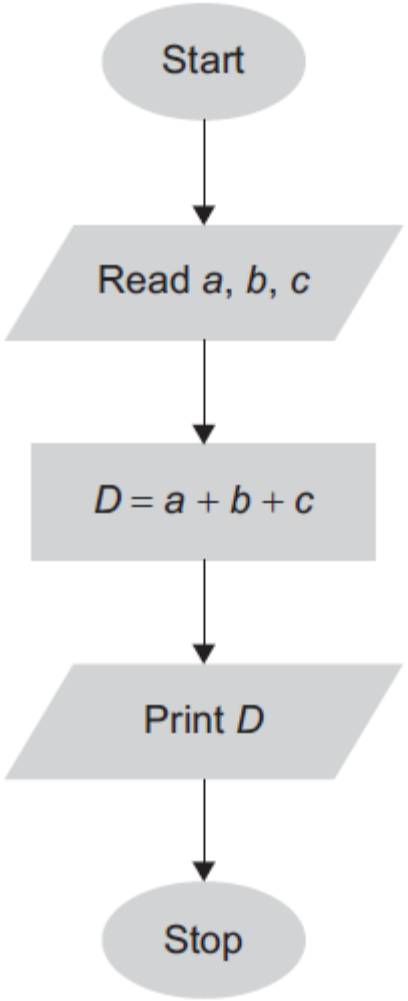
Connector	Used to join different flowlines.
Offpage Connector	Used to indicate that the flowchart continues to a second page.
Predefined Process	Used to represent a group of statements that perform one processing task.
Annotation	Used to provide additional information about another flowchart symbol.

Rules for drawing flowcharts

- You should write instructions inside the blocks
- If there is something you need to remember you can write a note besides a block
- A flowchart always start at the top of the page and flows to the bottom
- Use flowchart template to draw the symbols and flow lines
- Make the blocks big enough to write instructions so they can be easily read
- Final copy of the flowchart may not be the first draft

Flowchart example




Start
Read a, b, c
D = a + b + c
Print D
Stop

Analyzing an Algorithm

- How to find which is the **best Algorithm**?
- We can compare one algorithm with other and choose the best.
- Performance is measured in terms of
  - **Time Complexity** : Amount of time taken by algorithm to perform the task
  - **Space Complexity** : Amount of memory needed to perform the task

Space Complexity

- Space complexity specifies amount of computer memory required during the program execution as a function of input size
- Can be done at two different times
  - Compile Time
  - Run Time

Compile Time Space Complexity

- *It* is defined as the storage requirement of a program at compile time.
- It can be determined by summing up the storage size of each variable using declaration statements.
- For example, the space complexity of a non-recursive function of calculating the factorial of number *n* depends on the number *n* itself.

Space complexity = Space needed at compile time

- This includes memory requirement before execution starts.

Example1:

```
int z = a + b + c;
    return(z);
```

Memory requirement : 4*4+4 = 20

Int take 4 bytes, 4 bytes is for **return value**

space requirement is fixed for the above example, hence it is called **Constant Space Complexity**

Example2:

```
// n is the length of array a[]
int sum(int a[], int n)
{
   int x = 0;                    // 4 bytes for x
   for(int i = 0; i < n; i++) // 4 bytes for i
    {
      x  = x + a[i];
    }
    return(x);
}
```

- In the above code, 4*n bytes of space is required for the array a[] elements.
- 4 bytes each for x, n, i and the return value.
- Hence the total memory requirement will be (4n + 12), which is increasing linearly with the increase in the input value n, hence it is called as **Linear Space Complexity.**

```
int findSum(int n)
{
int sum = 0; // ----------> constant time
for(int i = 1; i <= n; ++i)
        for(int j = 1; j <= i; ++j)
                sum++; // ----------> it will run [n * (n + 1) / 2]
return sum; // ----------> constant time }
```

Run-time Space Complexity

- If the program is recursive or uses dynamic variables or dynamic data structures, then
- there is a need to determine space complexity at run-time.
- It is difficult to estimate memory requirement accurately, as it is also determined by the efficiency of compiler.
- Memory requirement is the summation of the program space, data space, and stack space.

Run-time Space Complexity

- ***Program space*** This is the memory occupied by the program itself.
- ***Data space*** This is the memory occupied by data members such as constants and variables.
- ***Stack space*** This is the stack memory needed to save the function's run-time environment while another function is called.

Time Complexity

- Time complexity : - Amount of computer time that it needs to run program to completion
- Depends upon
  - The machine we are executing on
  - Its machine language instruction set
  - The Time required by each machine instruction
  - The translation, compiler will make from source to machine language

Frequency count

- Denotes how many times particular statement is executed.
- Eg. Consider the code
- Void fun()
- {
  - int a;
  - a=10; … …….. 1
  - Printf(“ %d”, a) …… 1
  - }
  - **Frequency count is 2**
-

- Eg. Consider the code
- Void fun()
- {
  - int a;
  - a=10; … ……. 1
    for(i=0;i<n; i++) ……. n+1
    { a=a+i ; ……. N
    }
  - Printf(“ %d”, a) ……. 1
  - }
  - **Frequency count is 2n+3**

```
Void fun(int a[] [], int b[] [])
{
  int c[3][3];
for (i=0;i<m; i++)     ..... m+1
{
  for (j=0;j<n; j++)   ..... m(n+1)
  {
     c[i][j]= a[i][j] + b[i][j];  ..... m.n
}}
```

Frequency count is 2m(1+n)+1

Find frequency count for

1. for(i=1;i<=n; i++) …….. n+1

    for(j=1;j<=m; j++)….n(m+1)

    for(k=1;k<=p; k++)…n.m.(p+1)

    Sum=sum +i;….nmp

2. i=n; ……1

While(i>=1) ……..n+1

{

i-- ; …… n

}

f.c=2n+2

```
Double pow(double x, int n)
{
Double result=1;…….  1
While(n>0)…….. (n+1)
{
   result=result * x;…….. n
N--;                                  …….n
}
Return result; } ……1     f=3n+3
```

Calculating Time Complexity

```
c = a+b
```

Time complexity: **constant**

```
for(i=0; i < N; i++)
{
    statement;
}
```

Time complexity: **Linear**

```
for(i=0; i < N; i++)
{
    for(j=0; j < N;j++)
    {
    statement;
    }
}
```

Time complexity: **Quadratic**

```
while(low <= high)
{
    mid = (low + high) / 2;
    if (target < list[mid])
        high = mid - 1;
    else if (target > list[mid])
        low = mid + 1;
    else break;
}
```

Time complexity: **Logarithmic**

Calculating Time Complexity

```
void quicksort(int list[], int left, int right)
{
    int pivot = partition(list, left, right);
    quicksort(list, left, pivot - 1);
    quicksort(list, pivot + 1, right);
}
```

Time complexity: **Linear and logarithmic**

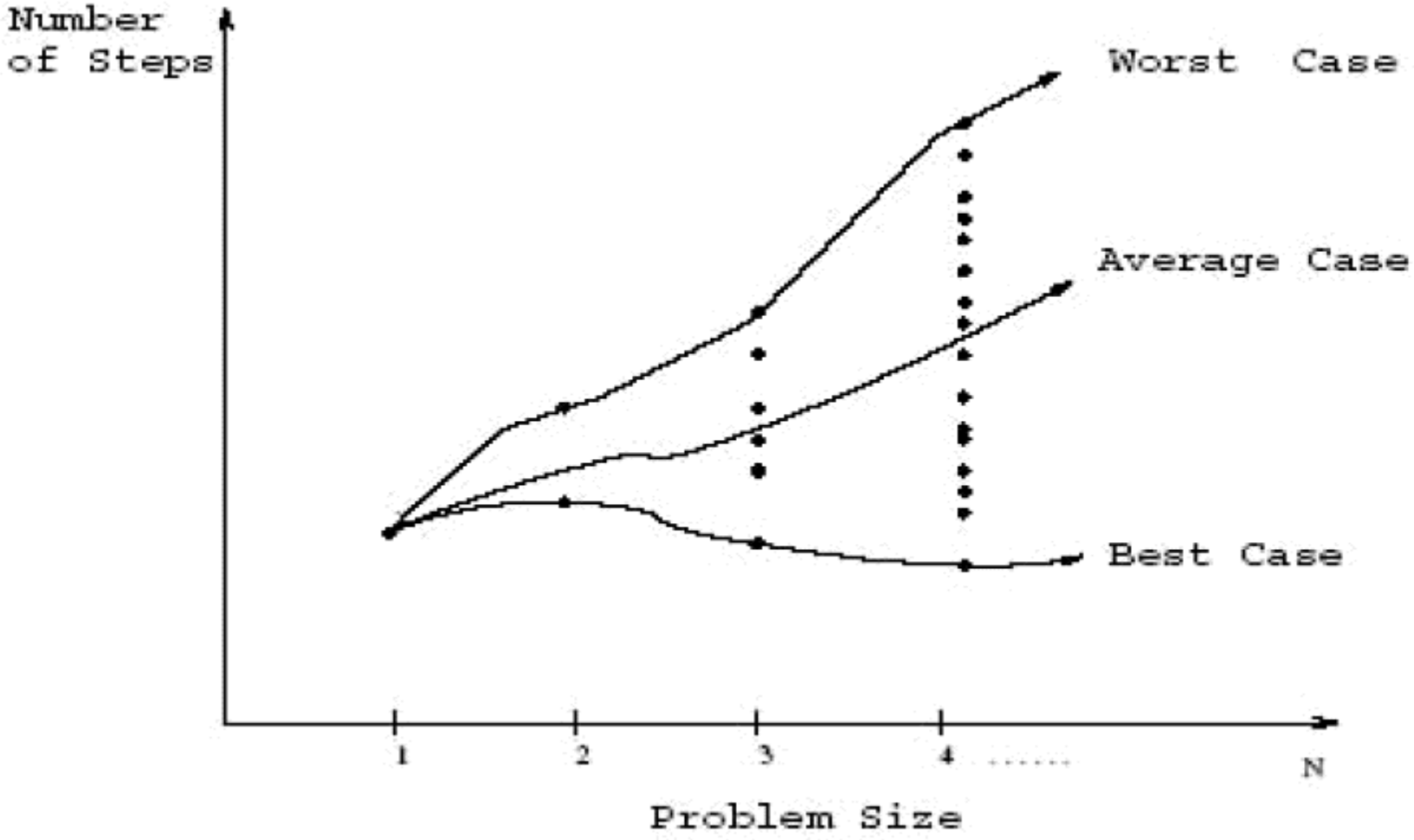
Standard Time Complexity Functions

Functions	Name
C	Constant
log N	Logarithmic
$Log_2 N$	Log-Squared
N	Linear
N^2	Quadratic
N^3	Cubic
C^N	Exponential

An Algorithm can be characterized by a timing function T (N)

Asymptotic Notations

- Ignoring machine and language dependent factors, consider only order of magnitude of ***frequency count***
- The notations are
  - O(n)- Upper Bound on the time complexity value
  - Ω(n)- Lower Bound of time an algorithm needs
  - Θ(n)- when best and worst case requires same time, Θ(n) is required

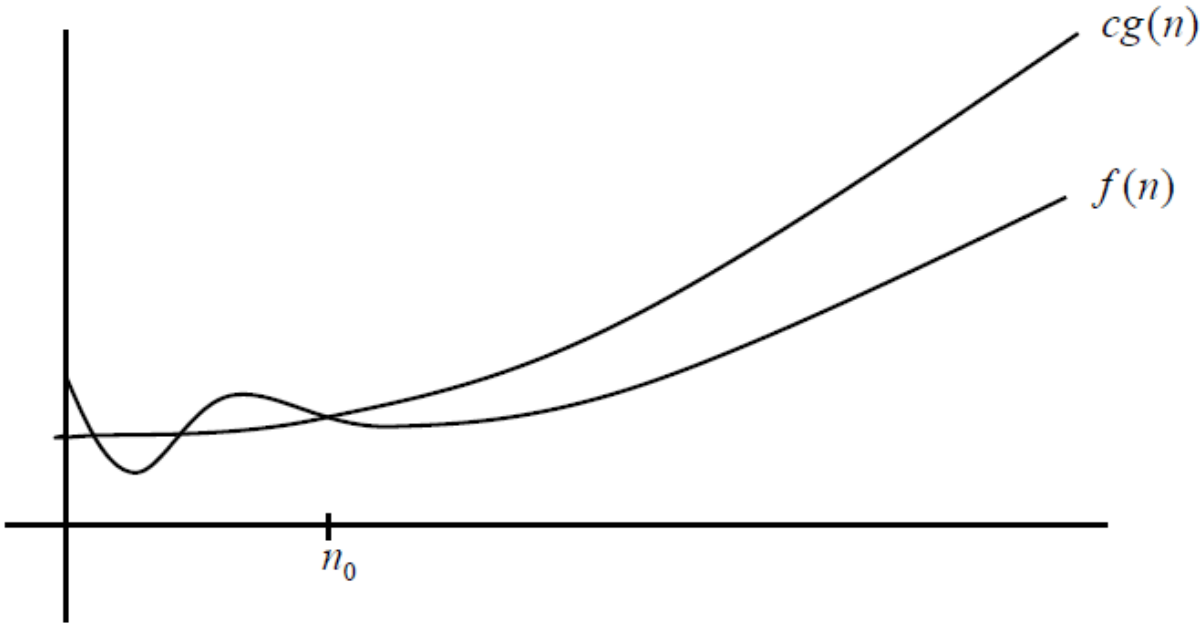

Number of Steps
Worst Case
Average Case
Best Case
1
2
3
4
.....
N
Problem Size

Asymptotic Analysis

- Goal: To simplify analysis of running time by getting rid of "details", which may be affected by specific implementation and hardware
  - Like "Rounding" 1000001=1000000
  - $3n^2=n^2$
- Capturing the essence : how the running time of an algorithm increases with the size of input in the limit

Asymptotic Notation

- The "big oh" Notation
  - Asymptotic upper bound
  - f(n)= O(g(n)) if there exist constant c and n_0, s.t f(n)<=c *g(n) for n>=n_0
  - f(n) and g(n) are non negative functions
- Used for worst case analysis
- 3n+2 is O(n) as 3n+2 <=4n for n>=2



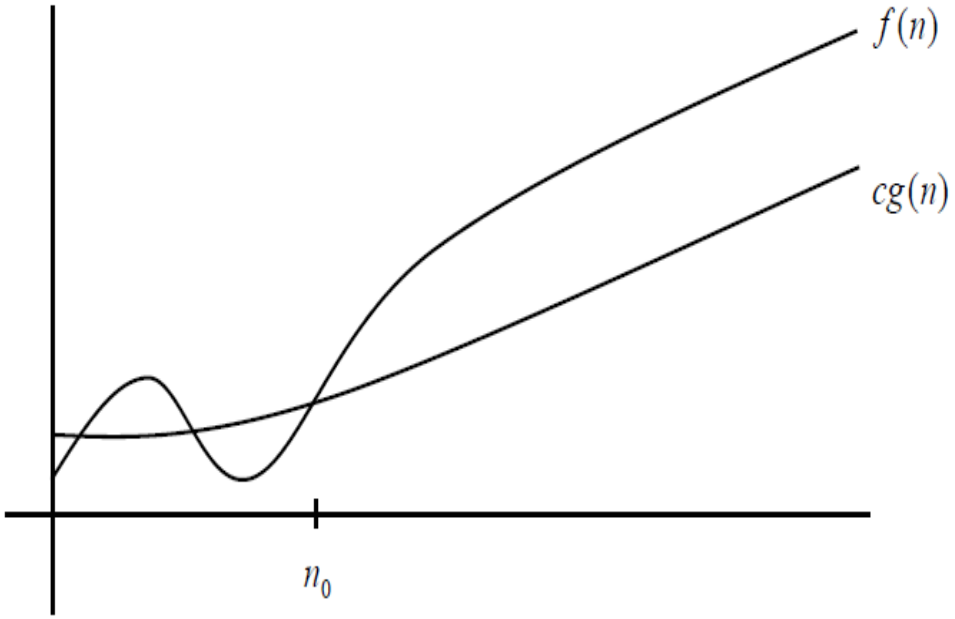
- 3n + 3 $O(n)$ as 3n+3 <= 4n for all n>=3
- 100n+6 $O(n)$ as 100n+6 <= 101n
- $10n^2 + 4n + 2$ is $O(n^2)$ as $10n^2 + 4n + 2 <= 11n^2$ for all n>= 5
- $1000n^2 + 100n + 6$ $O(n^2)$ as $1000n^2 + 100n - 6 <= 1001n^2$ for all n>=100

Asymptotic Notation

- The Omega Notation
  - Asymptotic lower bound
  - $f(n)=\Omega(g(n))$ if there exists constant c & n_0 s.t. $f(n)>=c*g(n)$ for $n>=n_0$
- Used to describe best case

  running time or lower bound
- $3n+2 = \Omega(n)$ as $3n+2 >=3n$ for $n>=1$

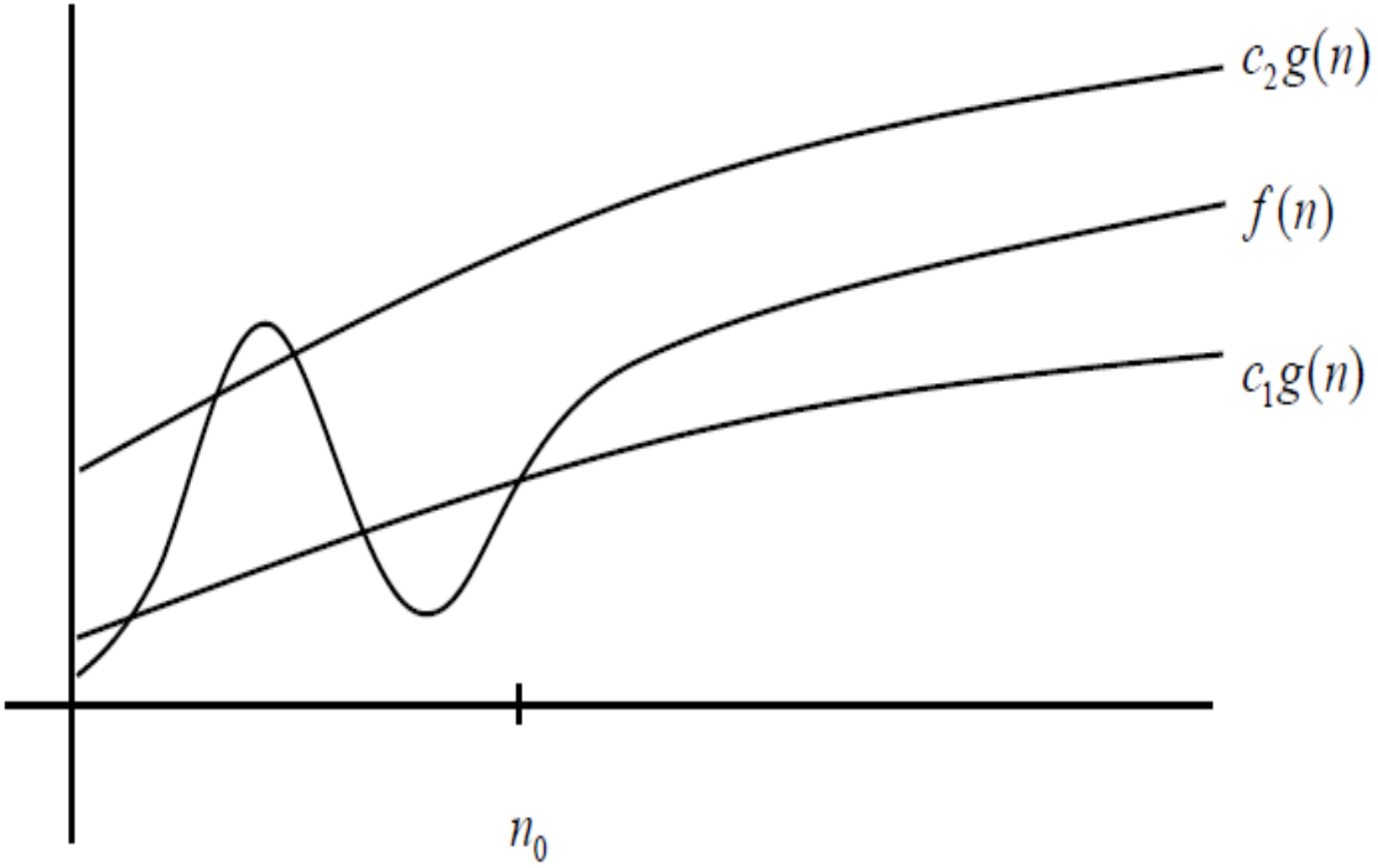
Examples

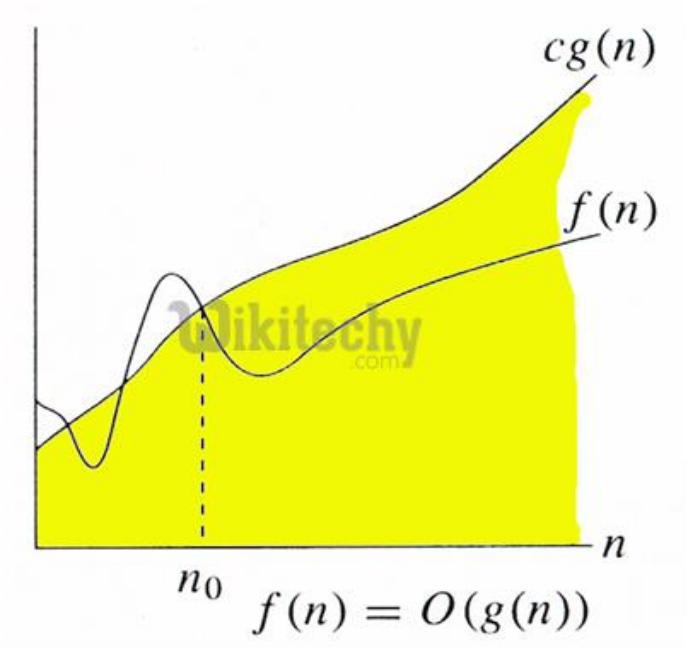


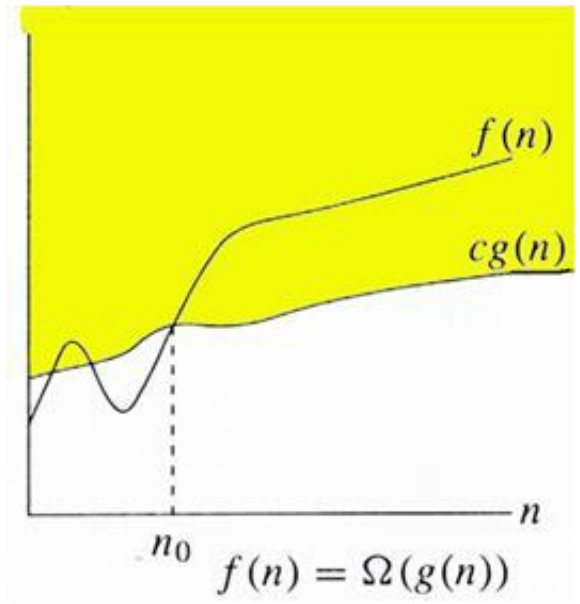
- $3n + 3 = \Omega(n)$ as $3n+3 >= 3n$ for all $n>= 1$
- $100n + 6 = \Omega(n)$ as $100n + 6 >= 100n$ for all $n>=1$
- $10n^2 + 4n + 2 = \Omega(n^2)$ as $100n^2 + 4n + 2 >=1$

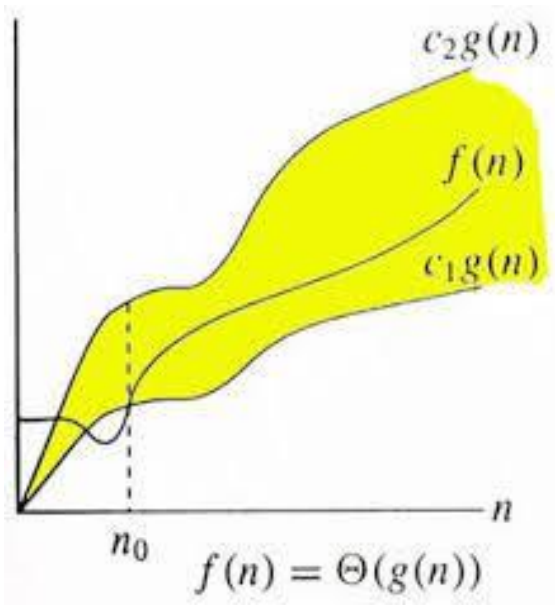
Asymptotic Notation

- The theta Notation [Θ]
  - Function f(n) = Θ (g(n))
  - If there exists positive constants c1,c2 and n0 c1*g(n)<= f(n) >=c2*g(n) for all n>= n_0
- 3n + 2 = Θ (n) as 3n + 2 >=3n for all n>=2 and 3n +2<=4n for all n>=2;
- Theta notation is more precise than both the big oh and omega notations


$c_2 g(n)$
$f(n)$
$c_1 g(n)$
n_0


cg(n)
f(n)
n
n0
f(n) = O(g(n))


f(n)
cg(n)
n
n0
f(n) = Ω(g(n))


c2g(n)
f(n)
c1g(n)
n
n0
f(n) = Θ(g(n))

Introduction to Data Structures

- *Data* is nothing but a piece of information. Data input, data manipulation (or data processing), and data output are the functions of computers.
- Hence all information taken as input, processed within a computer, or provided as output to the user is nothing but data.
- It can be a number, a string, or a set of many numbers and strings.

- *Atomic and Composite Data*
- *Atomic data* is the data that we choose to consider as a single, non-decomposable entity.
- Ex. 1234
- Composite data can be broken down into subfields that have meaning.
- For example, a student's record consists of Roll_Number, Name, Branch, Year, and so on.

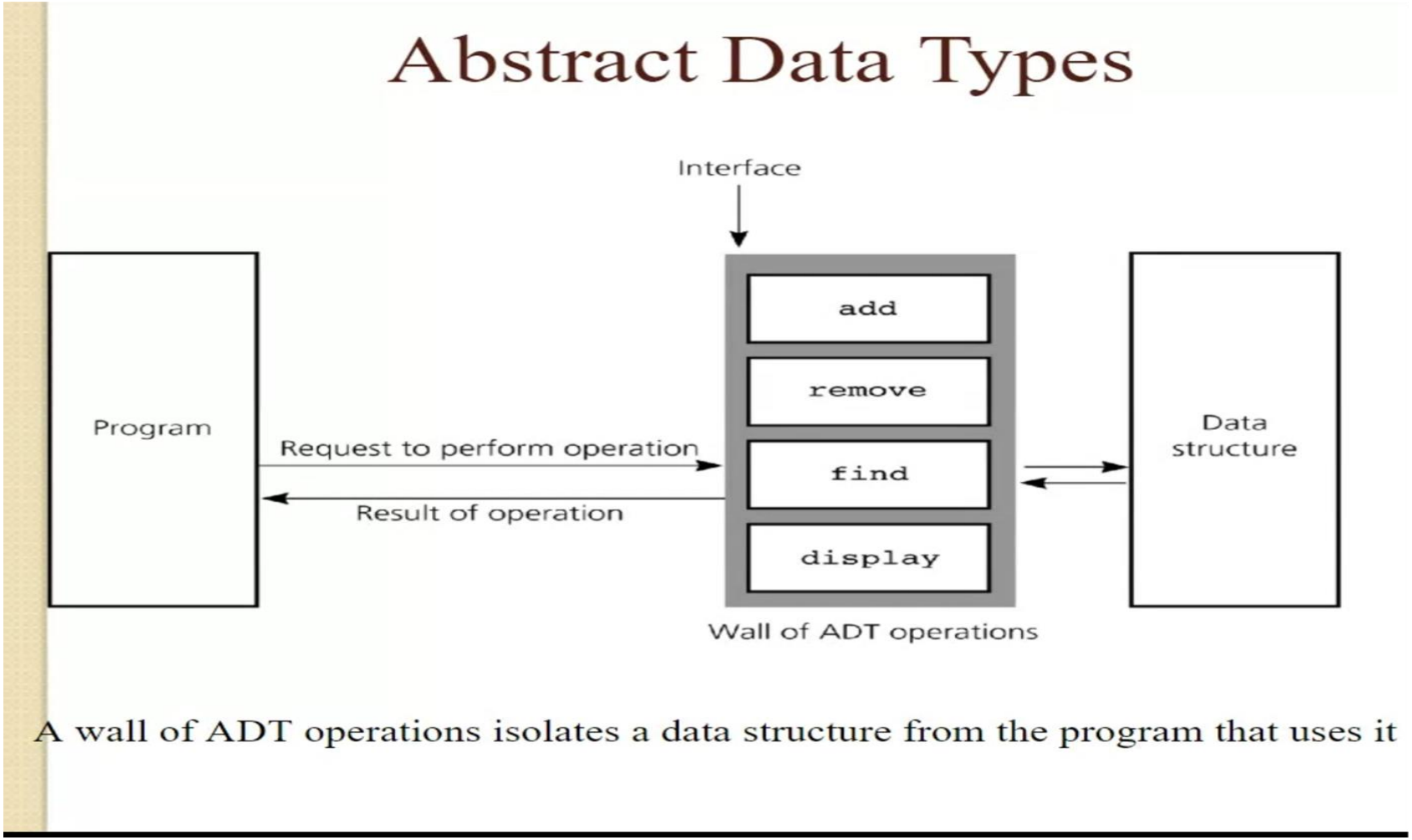
- Data Type
- *Data type* refers to the kind of data a variable may store.
- *Built-in Data Types:* int, float, and char are built-in data types
- *User-defined Data Types:* Class, structure, and union are the user-defined data types in C++.

- In brief, a data structure is
- 1. a combination of elements, each of which is either as a data type or another data structure and
- 2. a set of associations or relationships (structures) involving the combined elements.

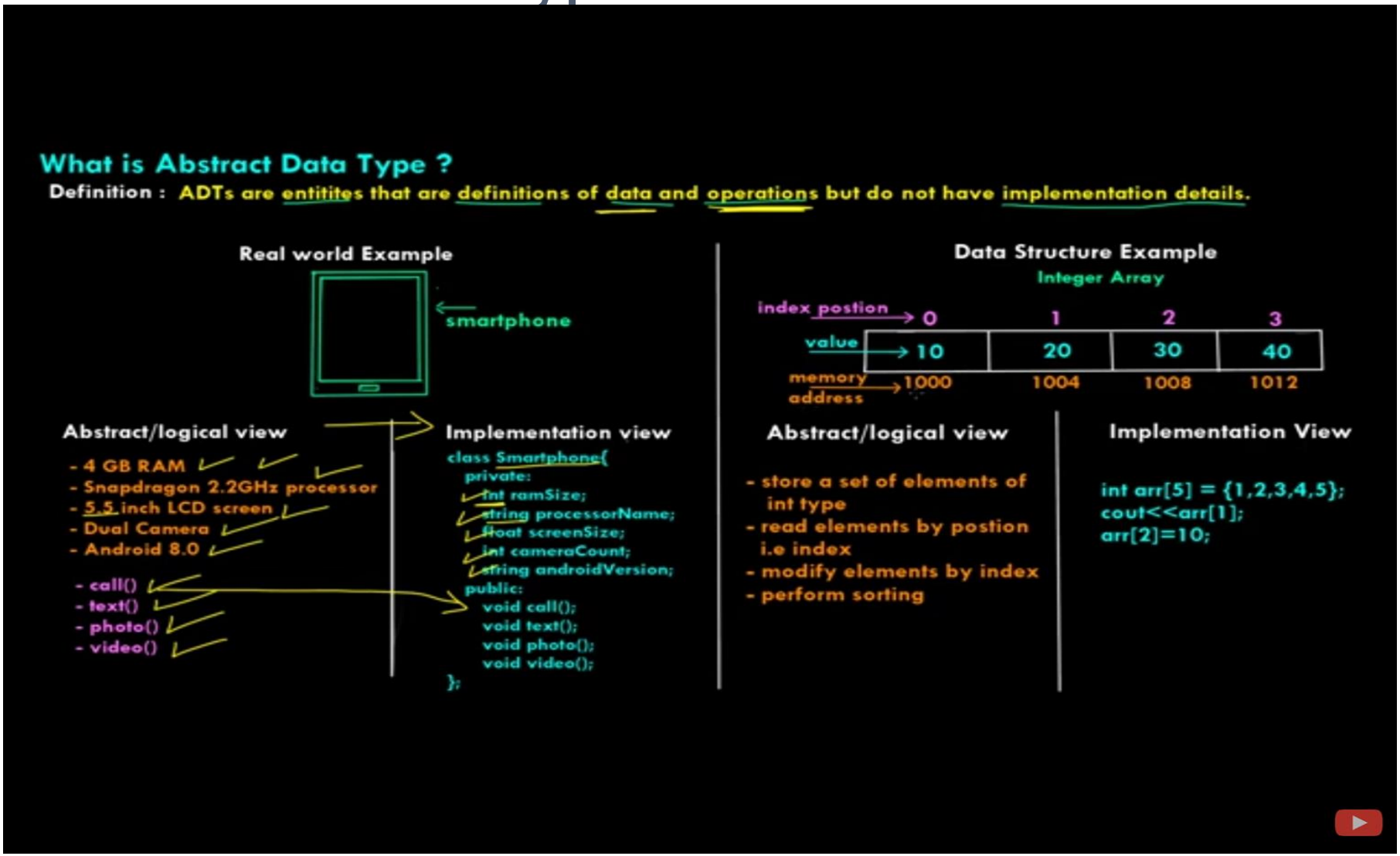
Abstract Data Type

- ADT Species what operations are to be performed but not how these operations will be implemented.
- It does not specify how data will be organized in memory and what algorithms will be used for implementing the operations.
- It is called “abstract” because it gives an implementation independent view.
- The process of providing only the essentials and hiding the details is known as abstraction.

Abstract Data Type

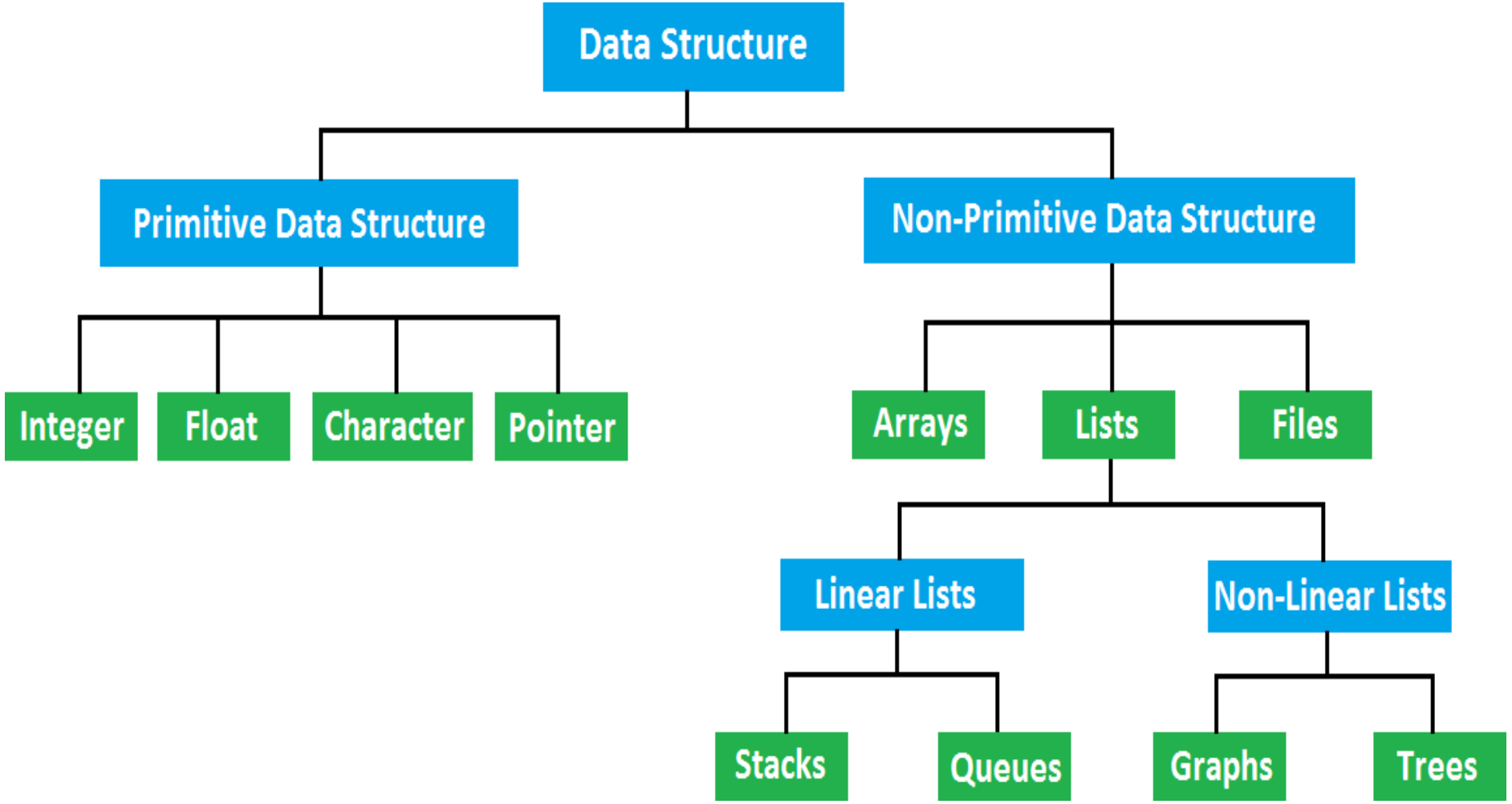


Abstract Data Type



Types of Data Structures

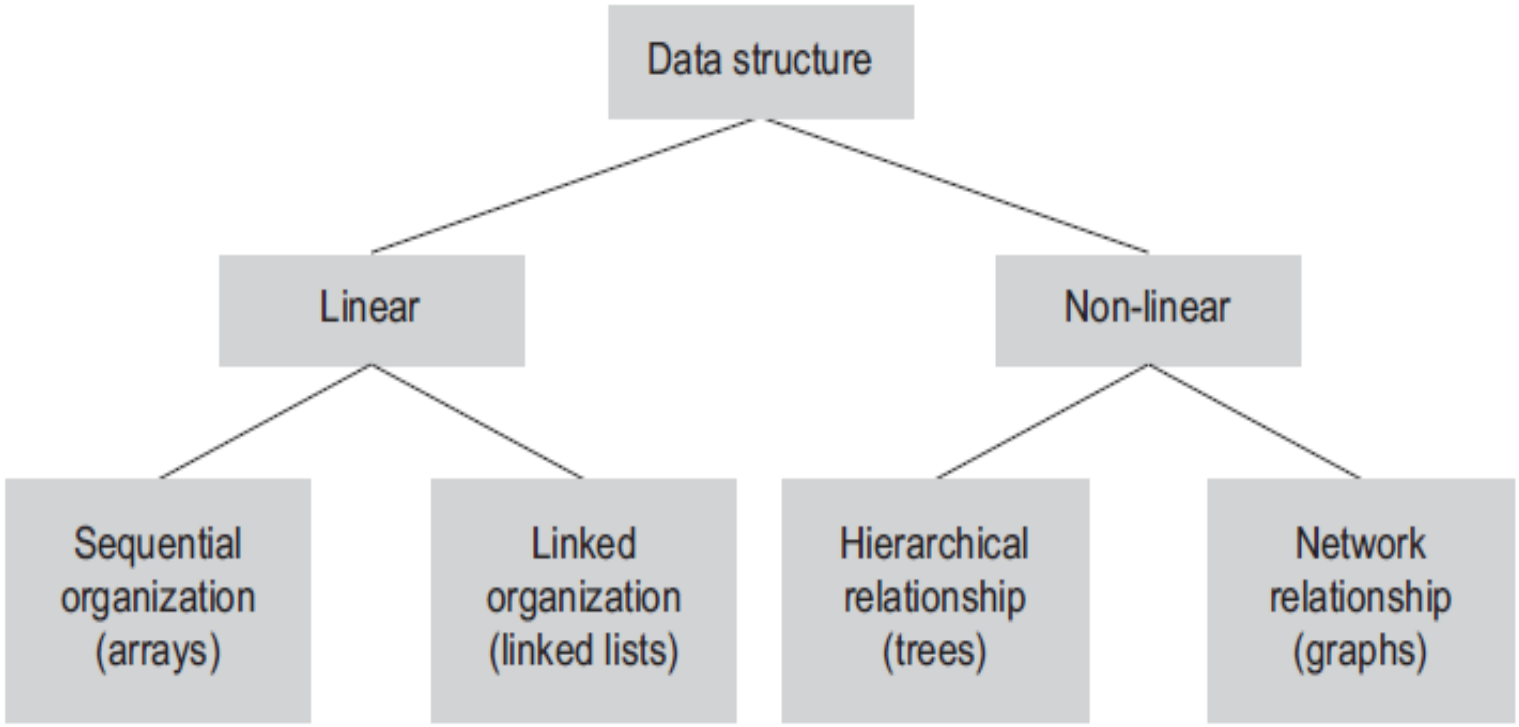
- Primitive and non-primitive
- Linear and non-linear
- Static and dynamic
- Persistent and ephemeral
- Sequential and direct access


Data Structure
Primitive Data Structure
Non-Primitive Data Structure
Integer
Float
Character
Pointer
Arrays
Lists
Files
Linear Lists
Non-Linear Lists
Stacks
Queues
Graphs
Trees

- *Primitive* data structures define a set of primitive elements that do not involve any other elements as its subparts.
- Example: int, char, float
- *Non-primitive* data structures are those that define a set of derived elements such as arrays.
- Example: Arrays in C++ consist of a set of similar type of elements.

Linear and Non-linear Data Structures

- A data structure is said to be *linear* if its elements form a sequence or a linear list. In a linear data structure, every data element has a unique successor and predecessor.
- Example: Arrays.
- *Non-linear* data structures are used to represent the data containing hierarchical or network relationship among the elements.
- Example: Trees and graphs


Data structure
Linear
Non-linear
Sequential organization (arrays)
Linked organization (linked lists)
Hierarchical relationship (trees)
Network relationship (graphs)

Static and Dynamic Data Structures

- A data structure is referred to as a *static data structure* if it is created before program execution begins (also called during compilation time).
- The variables of static data structure have user-specified names.
- An array is a static data structure.

Static and Dynamic Data Structures

- A data structure that is created at run-time is called *dynamic data structure*.
- The variables of this type are not always referenced by a user-defined name.
- These are accessed indirectly using their addresses through pointers.
- A linked list is a dynamic data structure

Persistent and Ephemeral Data Structures

- A data structure that supports operations on the most recent version as well as the previous version is termed as a *persistent data structure*.
- An ephemeral data structure is one that supports operations only on the most recent version.

Sequential Access and Direct Access

- This classification is with respect to the access operations associated with data structures.
- *Sequential access* means that to access the *n*th element, we must access the preceding (*n* - 1) data elements.
- A linked list is a sequential access data structure.
- *Direct access* means that any element can be accessed without accessing its predecessor or successor; we can directly access the *n*th element.
- An array is an example of a direct access data structure.

Relationship among data , data structure, and algorithm

- There is an close relationship between the structuring of data and analysis of algorithms.
- In fact, a data structure and an algorithm should be thought of as one single unit; neither one making sense without the other.
- Let us consider the example of searching for a person's phone number in a directory.
- The procedure we follow to search a person and get his/her phone number critically depends on how the phone number and names are arranged in the directory.

Relationship among data , data structure, and algorithm

- Let us consider two ways of organizing the data (phone numbers and names) in the directory.

1. **The data is organized randomly**. Then to search a person by name, one has to linearly start from the first name till the last name in the directory. There is no other option.

2. If the **data is organized by sorting the names** (alphabetically sorted in ascending order), then the search is much easier. Instead of linearly searching through all records, one may search in a particular area for particular alphabets, similar to using a dictionary.

Relationship among data , data structure, and algorithm

- We can say that there is a strong relationship between the structuring of data (along with inter-relationship among data structures) and the operations to process the data (algorithms).
- In fact, the way we process our data depends on the way we organize it.

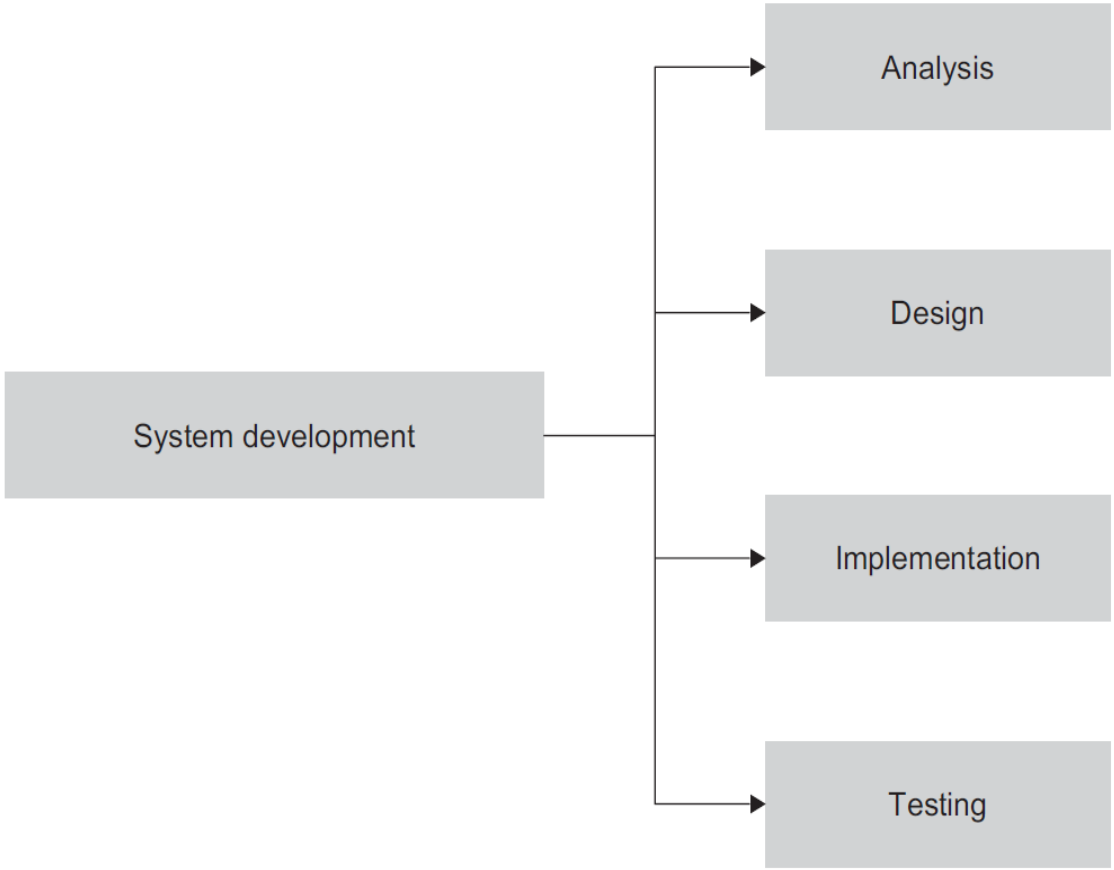
From Problem to Program

- If certain aspects of a problem can be expressed in terms of a **formal model**, it is usually beneficial to do so, for once a **problem is formalized**, we can look for solutions in terms of a **precise model** and determine whether a **program already exists** to solve that problem.
- **Software engineering** is the field that emphasizes on such a systematic approach for software development.

From Problem to Program

- A fundamental concept in software engineering is the software development life cycle (SDLC).
- The development process in the software life cycle broadly involves four phases:
  - Analysis
  - Design
  - Implementation
  - Testing

From Problem to Program



From Problem to Program

- **Analysis Phase:** The development process starts with the *analysis phase*; the systems analyst defines requirements that specify what the proposed system is to accomplish.
- **Design Phase:** In the design phase, the systems are determined, and the design of the files and/or the databases is completed.
- **Implementation Phase:** In the *implementation phase*, we create the actual programs.

From Problem to Program

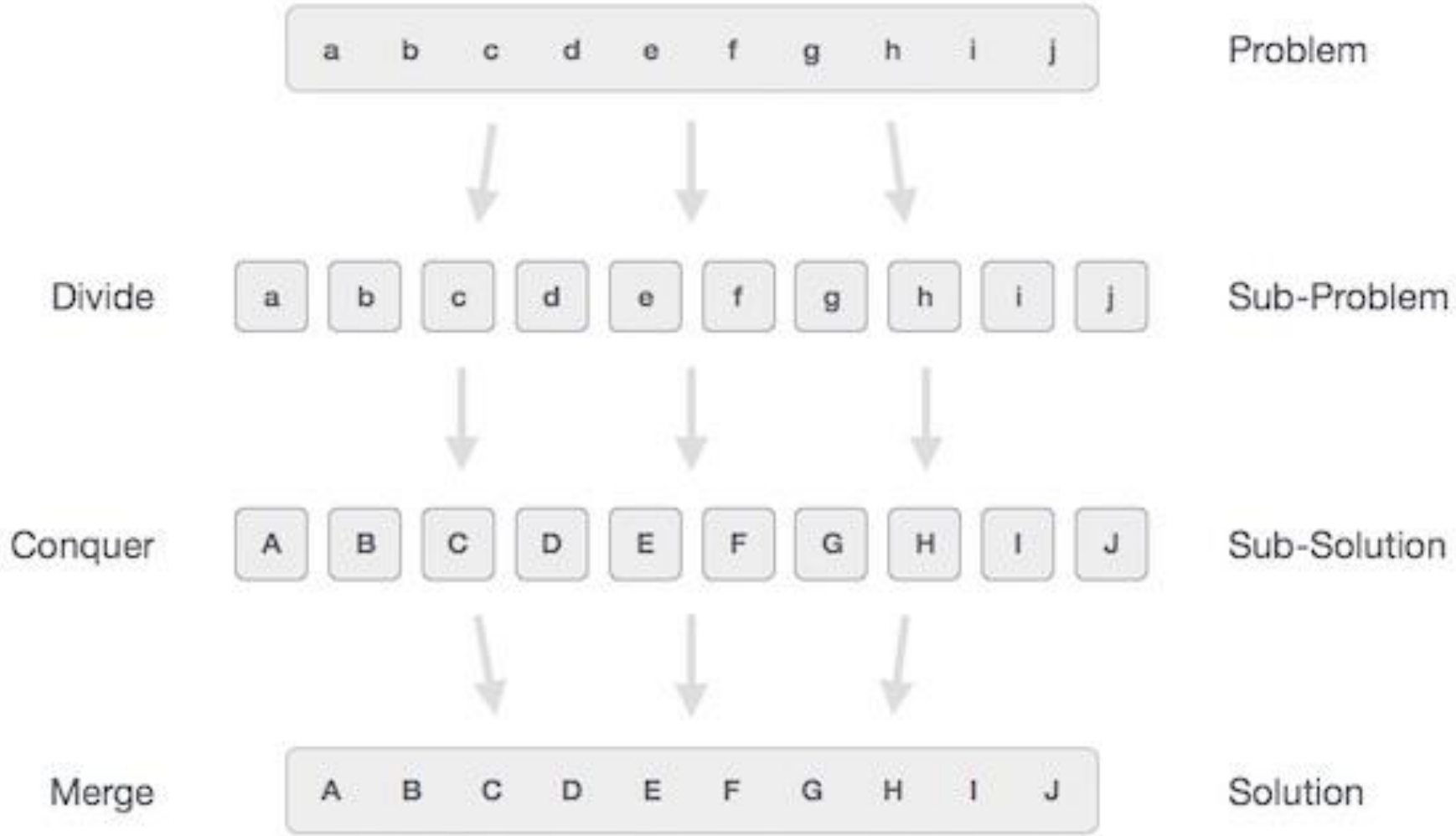
- **Testing Phase:** The programmers are completely responsible for testing the system as a whole, that is, testing to make sure all the programs work properly together. There are two types of testing—black box and white box. The system test that engineers and users do is black box testing. White box testing is the responsibility of the programmer.

Algorithmic Strategies

- Introduction to Algorithm design strategies-
  - **Divide and Conquer**
  - **Greedy strategy**
  - Dynamic Programming
  - Linear Programming
  - Reduction
  - Backtracking
  - Branch and Bound

Introduction to Divide & Conquer Strategy

- In divide and conquer approach, the problem is divided into smaller sub-problems and then each problem is solved independently. When we keep on dividing the sub problems into even smaller sub-problems, we may eventually reach a stage where no more division is possible.
- Those "atomic" smallest possible sub-problem (fractions) are solved.
- The solution of all sub-problems is finally merged in order to obtain the solution of an original problem.


a b c d e f g h i j
Problem
Divide
a b c d e f g h i j
Sub-Problem
Conquer
A B C D E F G H I J
Sub-Solution
Merge
A B C D E F G H I J
Solution

- **divide-and-conquer** approach in a three-step process.

Divide/Break

- This step involves breaking the problem into smaller sub-problems.
- Sub-problems should represent a part of the original problem.
- This step generally takes a recursive approach to divide the problem until no sub-problem is further divisible. At this stage, sub-problems become atomic in nature but still represent some part of the actual problem.

Conquer/Solve

- This step receives a lot of smaller sub-problems to be solved. Generally, at this level, the problems are considered 'solved' on their own.

Merge/Combine

- When the smaller sub-problems are solved, this stage recursively combines them until they formulate a solution of the original problem. This algorithmic approach works recursively and conquer & merge steps works so close that they appear as one.

Algorithm of D&C

- Step1: Accept set of inputs from user
- Step 2: Divide the input till more division is not possible (Apply Recursive division strategy.)
- Step 3:Solve the problem Independently.
- Step 4:Use Recursive approach to combine the solution.

Introduction to Greedy Algorithm

- An algorithm is designed to achieve optimum solution for a given problem.
- In greedy algorithm approach, decisions are made from the given solution domain.
- The closest solution that seems to provide an optimum solution is chosen.
- Greedy algorithms try to find a localized optimum solution, which may eventually lead to globally optimized solutions.
- However, generally greedy algorithms do not provide globally optimized solutions.

The greedy method

- Suppose that a problem can be solved by a sequence of decisions. The greedy method has that each decision is locally optimal. These locally optimal solutions will finally add up to a globally optimal solution.
- Only a few optimization problems can be solved by the greedy method.

A simple example

- Problem: Pick k numbers out of n numbers such that the sum of these k numbers is the largest.
- Algorithm:

```
FOR i = 1 to k
        pick out the largest number and
        delete this number from the input.
ENDFOR
```

Counting Coins

- This problem is to count to a desired value by choosing the least possible coins and the greedy approach forces the algorithm to pick the largest possible coin. If we are provided coins of ₹ 1, 2, 5 and 10 and we are asked to count ₹ 18 then the greedy procedure will be −
- **1** − Select one ₹ 10 coin, the remaining count is 8
- **2** − Then select one ₹ 5 coin, the remaining count is 3
- **3** − Then select one ₹ 2 coin, the remaining count is 1
- **4** − And finally, the selection of one ₹ 1 coins solves the problem

- Though, it seems to be working fine, for this count we need to pick only 4 coins. But if we slightly change the problem then the same approach may not be able to produce the same optimum result.
- For the currency system, where we have coins of 1, 7, 10 value, counting coins for value 18 will be absolutely optimum but for count like 15, it may use more coins than necessary. For example, the greedy approach will use 10 + 1 + 1 + 1 + 1 + 1, total 6 coins. Whereas the same problem could be solved by using only 3 coins (7 + 7 + 1)
- Hence, we may conclude that the greedy approach picks an immediate optimized solution and may fail where global optimization is a major concern.

Advantages of Greedy Algorithm

- It is quite easy to **come up with a greedy algorithm** (or even multiple greedy algorithms) for a problem.
- **Analyzing the run time for greedy algorithms will generally be much easier** than for other techniques (like Divide and conquer). For the Divide and conquer technique, it is not clear whether the technique is fast or slow. This is because at each level of recursion the size of gets smaller and the number of sub-problems increases.
- The difficult part is that for greedy algorithms **you have to work much harder to understand correctness issues**. Even with the correct algorithm, it is hard to prove why it is correct. Proving that a greedy algorithm is correct is more of an art than a science. It involves a lot of creativity.

Algorithm of Greedy Algorithm

- Step1: Accept Set of inputs from user.
- Step2: Apply Selection Strategy and perform combination.
- Stpe3: Try for all possible combinations till it satisfies condition.
- Step 4: Among all solutions select one optimal solution which satisfies or nearly satisfies criteria.

Thank You