# GE8151 PROBLEM SOLVING AND PYTHON PROGRAMMING
# UNIT I
# ALGORITHMIC PROBLEM SOLVING

Algorithms, building blocks of algorithms (statements, state, control flow, functions), notation (pseudo code, flow chart, programming language), algorithmic problem solving, simple strategies for developing algorithms (iteration, recursion). Illustrative problems: find minimum in a list, insert a card in a list of sorted cards, Guess an integer number in a range, Towers of Hanoi.

## 1.PROBLEM SOLVING

Problem solving is the systematic approach to define the problem and creating number of solutions.
The problem solving process starts with the problem specifications and ends with a Correct program.

## 1.1 PROBLEM SOLVING TECHNIQUES

Problem solving technique is a set of techniques that helps in providing logic for solving a problem.

**Problem Solving Techniques/Program Design Tools:**

Problem solving can be expressed in the form of

1. Algorithms.
2. Flowcharts.
3. Pseudo codes.

$\Big\} \longrightarrow$ Programs

## 1.2.ALGORITHM

Algorithm is an ordered sequence of finite, well defined, unambiguous instructions for completing a task. It is an English-like representation of the logic which is used to solve the problem. It is a step- by-step procedure for solving a task or a problem.

It is also defined as “any problem whose solution can be expressed in a list of executable instruction”.

It is defined as a sequence of instructions that describe a method for solving a problem. In other words it is a step by step procedure for solving a problem.

**Example- Algorithm to display your name ,dept**

```
1. Start
2. Get/Read the name and department
3. Print the name and department
4. Stop
```

**Algorithm to find the area of the circle**

```
1. Start
2. Read the value of radius r
3. Calculate - Area=3.14*r*r
4. Print the Area of the circle
5. Stop
```

**Characteristics of algorithm**

- Should be written in simple English
- Each and every instruction should be precise and unambiguous.
- Instructions in an algorithm should not be repeated infinitely.
- Algorithm should conclude after a finite number of steps.
- Should have an end point
- Derived results should be obtained only after the algorithm terminates.

**Qualities of a good algorithm**

The following are the primary factors that are often used to judge the quality of the algorithms.

**Time** – To execute a program, the computer system takes some amount of time. The lesser is the time required, the better is the algorithm.

**Memory** – To execute a program, computer system takes some amount of memory space. The lesser is the memory required, the better is the algorithm.

**Accuracy** – Multiple algorithms may provide suitable or correct solutions to a given problem, some of these may provide more accurate results than others, and such algorithms may be suitable.

Or

**Qualities of a good algorithm**

**Time** - Lesser time required.

**Memory** - Less memory required.

**Accuracy** - Suitable or correct solution obtained.

**Sequence** - Must be sequence and some instruction may be repeated in number of times or until particular condition is met.

**Generability** - Used to solve single problem and more often algorithms are designed to handle a range of input data.

## 2.BUILDING BLOCKS OF ALGORITHMS (statements, state, control flow, functions)

Algorithms can be constructed from basic building blocks namely, sequence, selection and iteration.

### 2.1.Statements:

Statement is a single action in a computer.

In a computer statements might include some of the following actions

- input data-information given to the program
- process data-perform operation on a given input
- output data-processed result

### 2.2.State:

Transition from one process to another process under specified condition with in a time is called state.

## 2.3.Control flow:

The process of executing the individual statements in a given order is called control flow.

The control can be executed in three ways

1. sequence
2. selection
3. iteration

### Sequence:

All the instructions are executed one after another is called sequence execution.

**Example:**

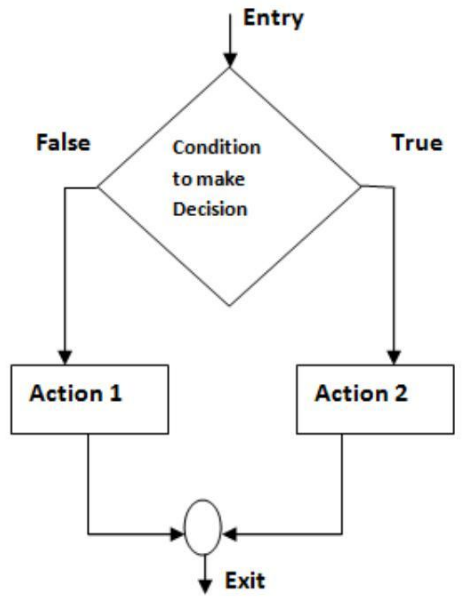
***Add two numbers:***

Step 1: Start
Step 2: get a,b
Step 3: calculate c=a+b
Step 4: Display c
Step 5: Stop

### Selection:

A selection statement causes the program control to be transferred to a specific part of the program based upon the condition.

If the conditional test is true, one part of the program will be executed, otherwise it will execute the other part of the program.



**Example**

**Write an algorithm to check whether he is eligible to vote?**

**Step 1:** Start
**Step 2:** Get age
**Step 3:** if age >= 18 print "Eligible to vote"
**Step 4:** else print "Not eligible to vote"
**Step 6:** Stop

## Iteration:

In some programs, certain set of statements are executed again and again based upon conditional test. i.e. executed more than one time. This type of execution is called **looping or repetition or iteration**.

**Example**

**Write an algorithm to print all natural numbers up to n**

**Step 1:** Start
**Step 2:** get n value.
**Step 3:** initialize i=1
**Step 4:** if (i<=n) go to step 5 else go to step 7
**Step 5:** Print i value and increment i value by 1
**Step 6:** go to step 4
**Step 7:** Stop

## 2.4.Functions:

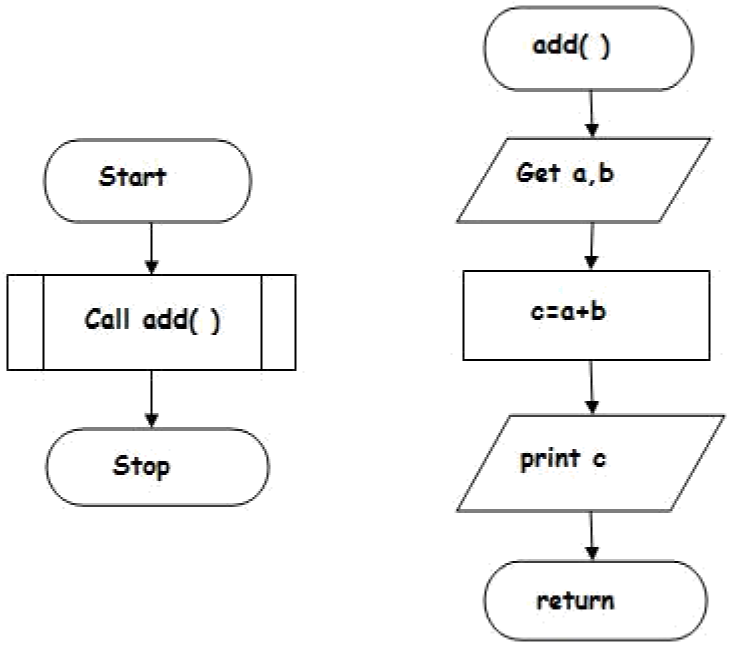
- Function is a sub program which consists of block of code(set of instructions) that performs a particular task.
- For complex problems, the problem is been divided into smaller and simpler tasks during algorithm design.

***Benefits of Using Functions***

- Reduction in line of code
- code reuse
- Better readability
- Information hiding
- Easy to debug and test
- Improved maintainability

**Example:**
**Algorithm for addition of two numbers using function**
**Main function()**
**Step 1:** Start
**Step 2:** Call the function add()
**Step 3:** Stop

**sub function add()**
**Step 1:** Function start
**Step 2:** Get a, b Values
**Step 3:** add c=a+b
**Step 4:** Print c
**Step 5:** Return



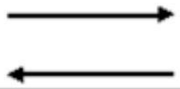
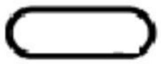
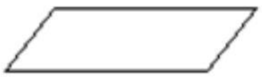
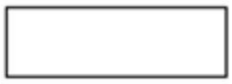
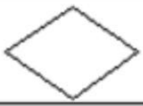
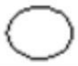
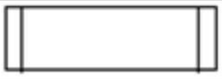
## 3.NOTATIONS

### *3.1.FLOW CHART*

Flow chart is defined as graphical or diagrammatic representation of the logic for problem solving.

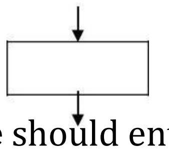
The purpose of flowchart is making the logic of the program clear in a visual representation.

A flowchart is a picture of the separate steps of a process in sequential order.

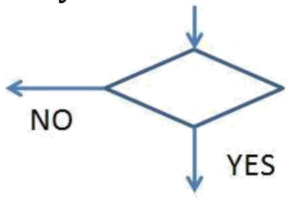
| Symbol | Symbol Name | Description |
|---|---|---|
|  | Flow Lines | Used to connect symbols |
|  | Terminal | Used to start, pause or halt in the program logic |
|  | Input/output | Represents the information entering or leaving the system |
|  | Processing | Represents arithmetic and logical instructions |
|  | Decision | Represents a decision to be made |
|  | Connector | Used to Join different flow lines |
|  | Sub function | used to call function |

**<u>Rules for drawing a flowchart</u>**

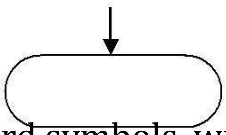
1. The flowchart should be clear, neat and easy to follow.
2. The flowchart must have a logical start and finish.
3. Only one flow line should come out from a process symbol.



4. Only one flow line should enter a decision symbol. However, two or three flow lines may leave the decision symbol.



5. Only one flow line is used with a terminal symbol.



6. Within standard symbols, write briefly and precisely.
7. Intersection of flow lines should be avoided.

*Advantages/Benefits of flowchart:*

1. **Communication: -** Flowcharts are better way of communicating the logic of a system to all concerned.
2. **Effective analysis: -** With the help of flowchart, problem can be analyzed in more effective way.

3. **Proper documentation: -** Program flowcharts serve as a good program documentation, which is needed for various purposes.
4. **Efficient Coding: -** The flowcharts act as a guide or blueprint during the systems analysis and program development phase.
5. **Proper Debugging: -** The flowchart helps in debugging process.
6. **Efficient Program Maintenance: -** The maintenance of operating program
   becomes easy with the help of flowchart. It helps the programmer to put efforts more efficiently on that part.

## *Disadvantages/Limitation of using flowchart*

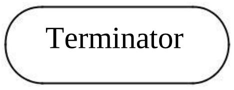
1. **Complex logic: -** Sometimes, the program logic is quite complicated. In that case, flowchart becomes complex and clumsy.
2. **Alterations and Modifications: -** If alterations are required the flowchart may require re-drawing completely.
3. **Reproduction: -** As the flowchart symbols cannot be typed, reproduction of flowchart becomes a problem.
4. **Cost: F**or large application the time and cost of flowchart drawing becomes costly.

## GUIDELINES FOR DRAWING A FLOWCHART

Flowcharts are usually drawn using some standard symbols; however, some special symbols can also be developed when required. Some standard symbols, which are frequently required for flowcharting many computer programs.

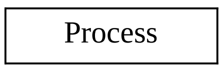
**Terminator:**

An oval flow chart shape indicates the start or end of the process, usually containing the word “Start” or “End”.



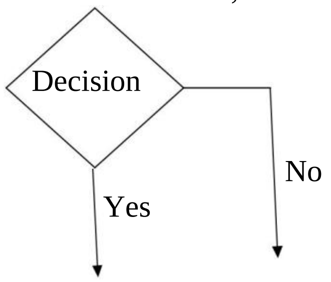
**Process:**

A rectangular flow chart shape indicates a normal/generic process flow step. For example, “Add 1 to X”, “M = M*F” or similar.



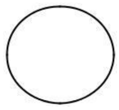
**Decision:**

A diamond flow chart shape indicates a branch in the process flow. This symbol is used when a decision needs to be made, commonly a Yes/No question or True/False test.



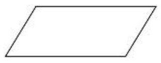
**Connector:**

A small, labelled, circular flow chart shape used to indicate a jump in the process flow. Connectors are generally used in complex or multi-sheet diagrams.



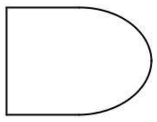
**Data:**

A parallelogram that indicates data input or output (I/O) for a process. Examples: Get X from the user, Display X.



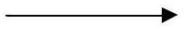
**Delay:**

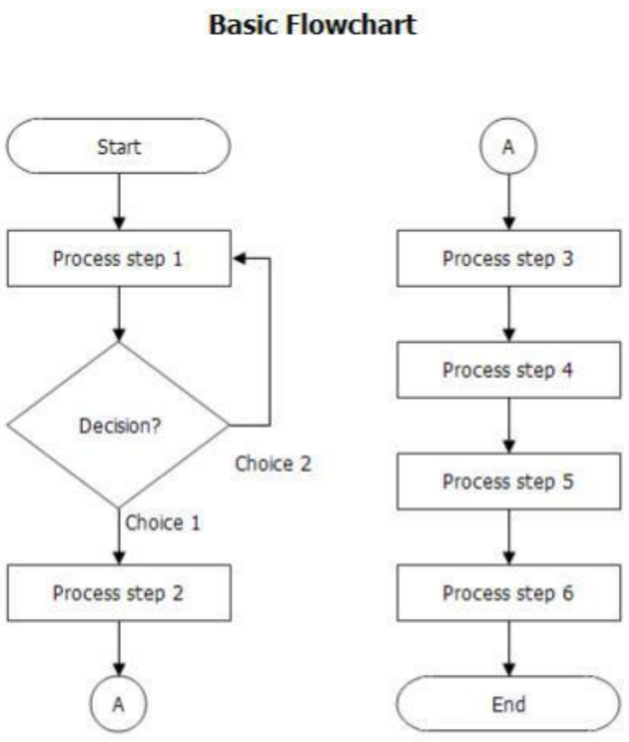
Used to indicate a delay or wait in the process for input from some other process.



**Arrow:**

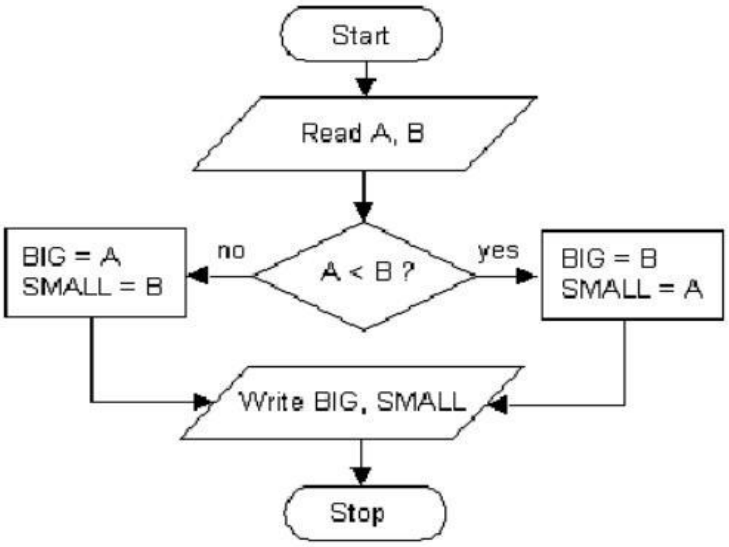
Used to show the flow of control in a process. An arrow coming from one symbol and ending at another symbol represents that control passes to the symbol the arrow points to.



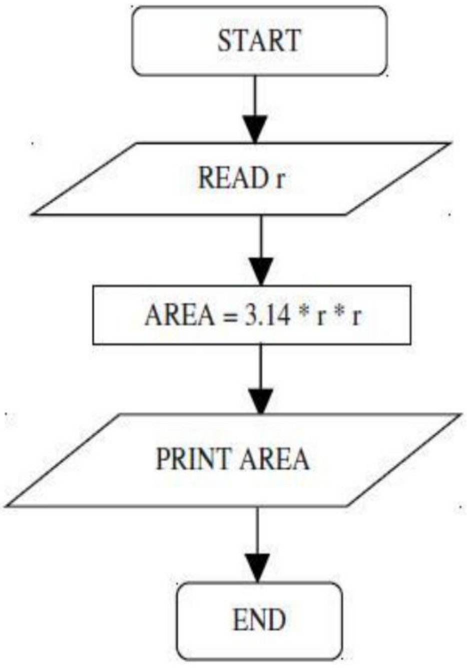


**Example Flowchart**

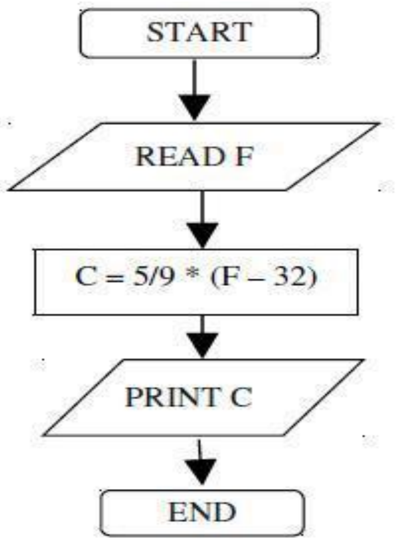
***Problem 1:*** Draw the flowchart to find the largest number between A and B



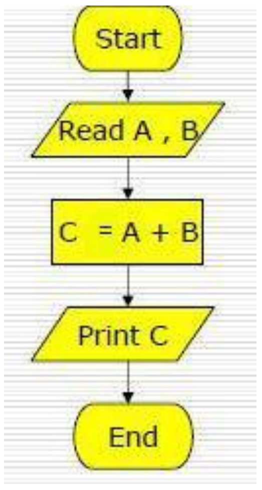
***Problem*** 2: Find the area of a circle of radius r.



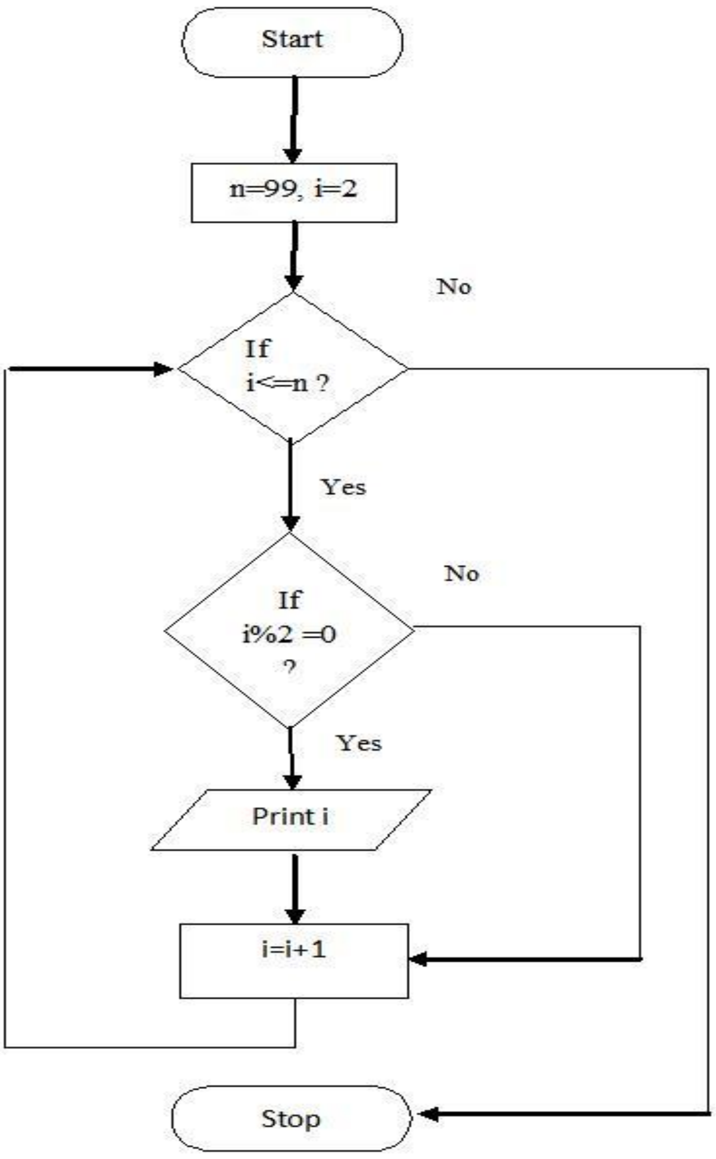
***Problem 3***: Convert temperature Fahrenheit to Celsius.



***Problem 4***: Flowchart for an algorithm which gets two numbers and prints sum of their value



***Problem5***: Flowchart for the problem of printing even numbers between 0 and 99.



## ***3.2.PSEUDO CODE:***

"Pseudo" means initiation or false.
"Code" means the set of statements or instructions written in a programming language. Pseudocode is also called as "Program Design Language [PDL]".

- Pseudo code consists of short, readable and formally styled English languages used for explaining an algorithm.
- It does not include details like variable declaration, subroutines.
- It is easier to understand for the programmer or non programmer to understand the general working of the program, because it is not based on any programming language.
- It gives us the sketch of the program before actual coding.
- It is not a machine readable
- Pseudo code can't be compiled and executed.
- There is no standard syntax for pseudo code.

### **Rules for writing Pseudocode**

- Write one statement per line
- Capitalize initial keyword(READ, WRITE, IF, WHILE, UNTIL).
- Indent to hierarchy
- End multiline structure
- Keep statements language independent

### **Common keywords used in pseudocode**

The following gives common keywords used in pseudocodes. 1.
**//:** This keyword used to represent a comment.
2. **BEGIN,END:** Begin is the first statement and end is the last statement.
3. **INPUT, GET, READ**: The keyword is used to inputting data.
4. **COMPUTE, CALCULATE**: used for calculation of the result of the given expression.
**5. ADD, SUBTRACT, INITIALIZE** used for addition, subtraction and initialization**.**
6. **OUTPUT, PRINT, DISPLAY**: It is used to display the output of the program.
7. **IF, ELSE, ENDIF**: used to make decision.
8. **WHILE, ENDWHILE**: used for iterative statements.
9. **FOR, ENDFOR**: Another iterative incremented/decremented tested automatically.

***Example:***

**Addition of two numbers:**

```
BEGIN
GET a,b
ADD c=a+b
PRINT c
END
```

| **Syntax for if else:** | **Example: Greates of two numbers** |
|---|---|
| IF (condition)THEN | BEGIN |
| statement | READ a,b |
| ... | IF (a>b) THEN |
| ELSE | DISPLAY a is greater |
| statement | ELSE |
| ... | DISPLAY b is greater |
| ENDIF | END IF |
|  | END |
| **Syntax for For:** | **Example: Print n natural numbers** |
| FOR( *start-value* to *end-value)* DO | BEGIN |
| *statement* | GET n |
| ... | INITIALIZE i=1 |
| ENDFOR | FOR (i<=n) DO |
|  | PRINT i |
|  | i=i+1 |
|  | ENDFOR |
|  | END |
| **Syntax for While:** | **Example: Print n natural numbers** |
| WHILE (condition) DO | BEGIN |
| statement | GET n |
| ... | INITIALIZE i=1 |
| ENDWHILE | WHILE(i<=n) DO |
|  | PRINT i |
|  | i=i+1 |
|  | ENDWHILE |
|  | END |

**Advantages:**

- Pseudo is independent of any language; it can be used by most programmers.
- It is easy to translate pseudo code into a programming language.
- It can be easily modified as compared to flowchart.
- Converting a pseudo code to programming language is very easy as compared with converting a flowchart to programming language.

- It does not provide visual representation of the program's logic.
- There are no accepted standards for writing pseudo codes.
- It cannot be compiled nor executed.
- For a beginner, It is more difficult to follow the logic or write pseudo code as compared to flowchart.

**Disadvantage**

It is not visual.

We do not get a picture of the design.

There is no standardized style or format.

For a beginner, it is more difficult to follow the logic or write pseudocode as compared to flowchart.

| **Algorithm** | **Flowchart** | **Pseudo code** |
|---|---|---|
| An algorithm is a sequence of instructions used to solve a problem | It is a graphical representation of algorithm | It is a language representation of algorithm. |
| User needs knowledge to write algorithm. | not need knowledge of program to draw or understand flowchart | Not need knowledge of program language to understand or write a pseudo code. |

## <u>4.ALGORITHMIC PROBLEM SOLVING:</u>

Algorithmic problem solving is solving problem that require the formulation of an algorithm for the solution.

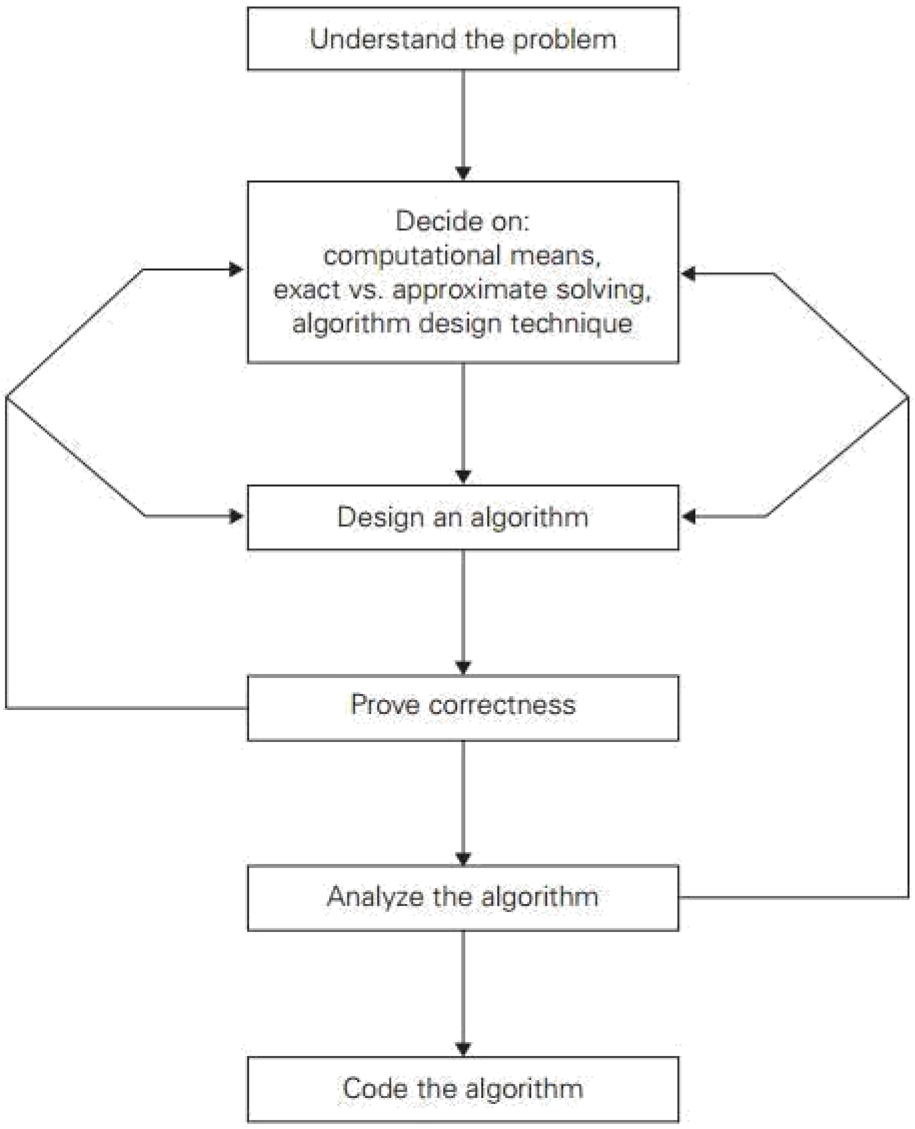


**FIGURE 1.2** Algorithm design and analysis process.

### <u>Understanding the Problem</u>

- ❖ It is the process of finding the input of the problem that the algorithm solves.
- ❖ It is very important to specify exactly the set of inputs the algorithm needs to handle.
- ❖ A correct algorithm is not one that works most of the time, but one that works correctly for *all* legitimate inputs.

### <u>Ascertaining the Capabilities of the Computational Device</u>

 If the instructions are executed one after another, it is called sequential algorithm.

- ❖ If the instructions are executed concurrently, it is called parallel algorithm.

**Choosing between Exact and Approximate Problem Solving**

- The next principal decision is to choose between solving the problem exactly or solving it approximately.
- Based on this, the algorithms are classified as exact *algorithm* and *approximation algorithm.*

- Data structure plays a vital role in designing and analysis the algorithms.
- Some of the algorithm design techniques also depend on the structuring data specifying a problem's instance
- Algorithm+ Data structure=programs.

**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.
- Learning these techniques is of utmost importance for the following reasons.
- First, they provide guidance for designing algorithms for new problems,
- Second, algorithms are the cornerstone of computer science

**Methods of Specifying an Algorithm**

- ***Pseudocode*** is a mixture of a natural language and programming language-like constructs. Pseudocode is usually more precise than natural language, and its usage often yields more succinct algorithm descriptions.

- In the earlier days of computing, the dominant vehicle for specifying algorithms was a ***flowchart***, a method of expressing an algorithm by a collection of connected geometric shapes containing descriptions of the algorithm's steps.

- **Programming language** can be fed into an electronic computer directly. Instead, it needs to be converted into a computer program written in a particular computer language. We can look at such a program as yet another way of specifying the algorithm, although it is preferable to consider it as the algorithm's implementation.

- Once an algorithm has been specified, you have to prove its ***correctness***. That is, you have to prove that the algorithm yields a required result for every legitimate input in a finite amount of time.
- 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.
- It might be worth mentioning that although tracing the algorithm's performance for a few specific inputs can be a very worthwhile activity, it cannot prove the algorithm's correctness conclusively. But in order to show that an algorithm is incorrect, you need just one instance of its input for which the algorithm fails.

## Analysing an Algorithm

1. ***Efficiency***.

*Time efficiency,* indicating how fast the algorithm runs,
*Space efficiency*, indicating how much extra memory it uses.

2. ***simplicity***.

- ❖ An algorithm should be precisely defined and investigated with mathematical expressions.
- ❖ Simpler algorithms are easier to understand and easier to program.
- ❖ Simple algorithms usually contain fewer bugs.

## Coding an Algorithm

- ❖ Most algorithms are destined to be ultimately implemented as computer programs. Programming an algorithm presents both a peril and an opportunity.
- ❖ A working program provides an additional opportunity in allowing an empirical analysis of the underlying algorithm. Such an analysis is based on timing the program on several inputs and then analysing the results obtained.

## 5.SIMPLE STRATEGIES FOR DEVELOPING ALGORITHMS:

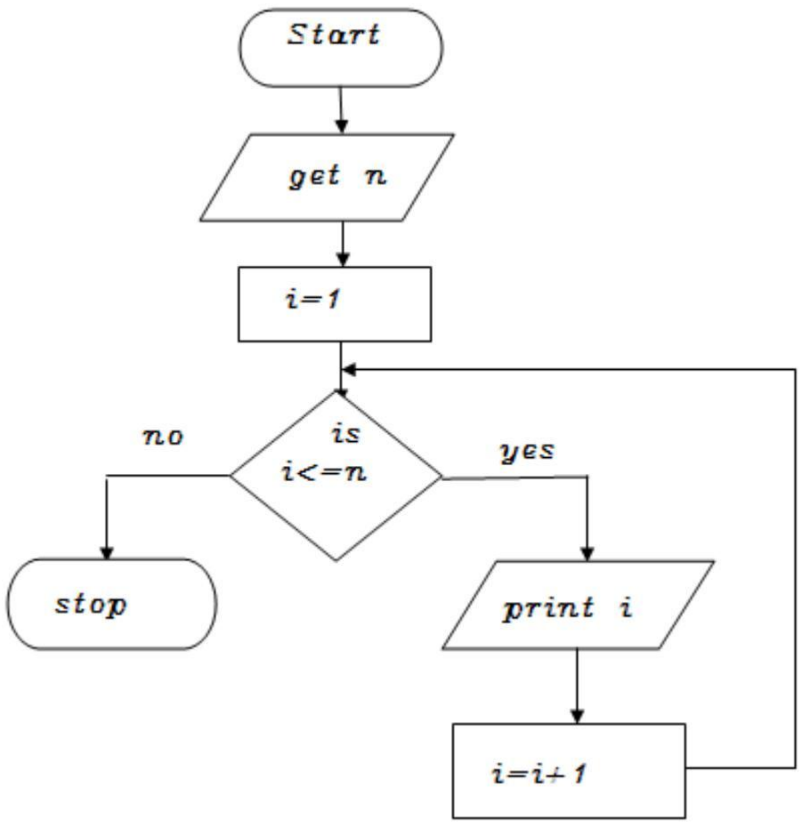
1. iterations
2. Recursions

### 5.1.Iterations:

A sequence of statements is executed until a specified condition is true is called iterations.

1. for loop
2. While loop

| Syntax for For: | Example: Print n natural numbers |
|---|---|
| FOR( *start-value* to *end-value)* DO<br>   *statement*<br>   ...<br>ENDFOR | BEGIN<br>GET n<br>INITIALIZE i=1<br>FOR (i<=n) DO<br>    PRINT i<br>    i=i+1<br>ENDFOR<br>END |
| **Syntax for While:** | **Example: Print n natural numbers** |
| WHILE (condition) DO<br>   statement<br>   ...<br>ENDWHILE | BEGIN<br>GET n<br>INITIALIZE i=1<br>WHILE(i<=n) DO<br>    PRINT i<br>    i=i+1<br>ENDWHILE<br>END |



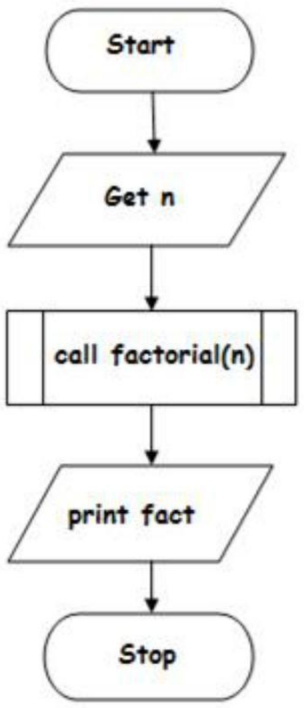
## 5.2.Recursions:

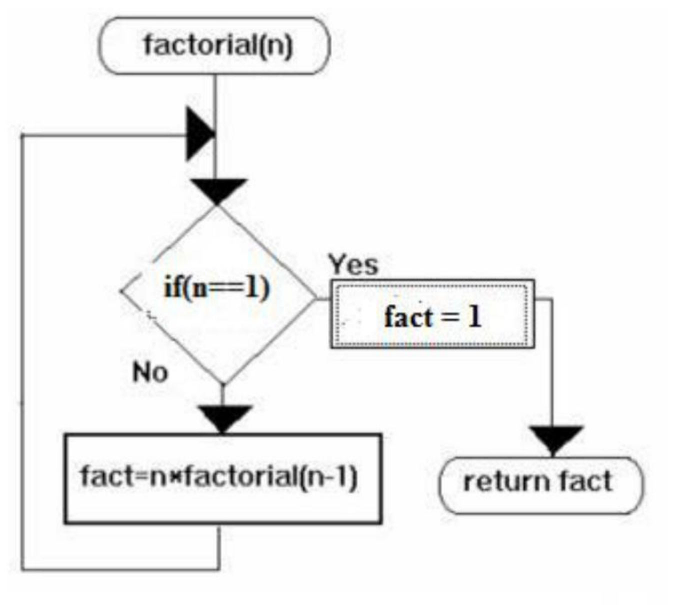
- A function that calls itself is known as recursion.
- Recursion is a process by which a function calls itself repeatedly until some specified condition has been satisfied.

## Algorithm for factorial of n numbers using recursion:

**Main function:**
Step1: Start
Step2: Get n
Step3: call factorial(n)
Step4: print fact
Step5: Stop

**Sub function factorial(n):**
Step1: if(n==1) then fact=1 return fact
Step2: else fact=n*factorial(n-1) and return fact





**Pseudo code for factorial using recursion:**

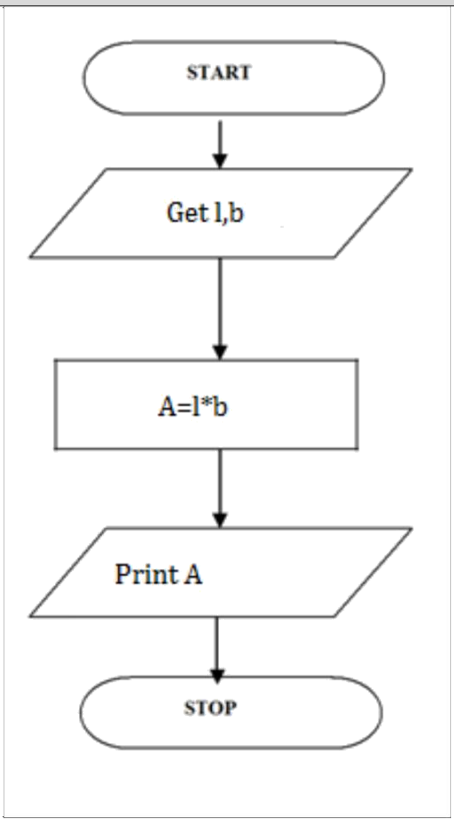
**Main function**:

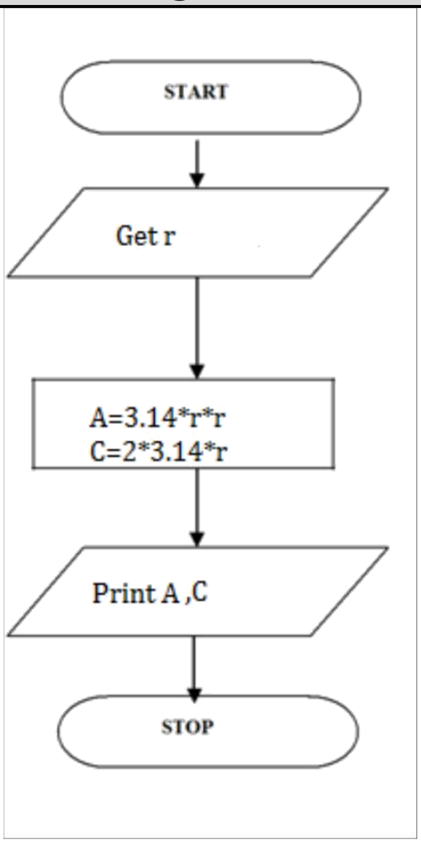
```
BEGIN
GET n
CALL factorial(n)
PRINT fact
BIN
```

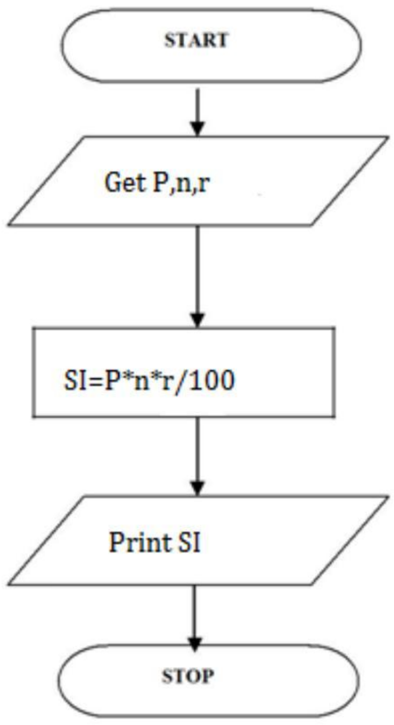
**Sub function factorial(n):**

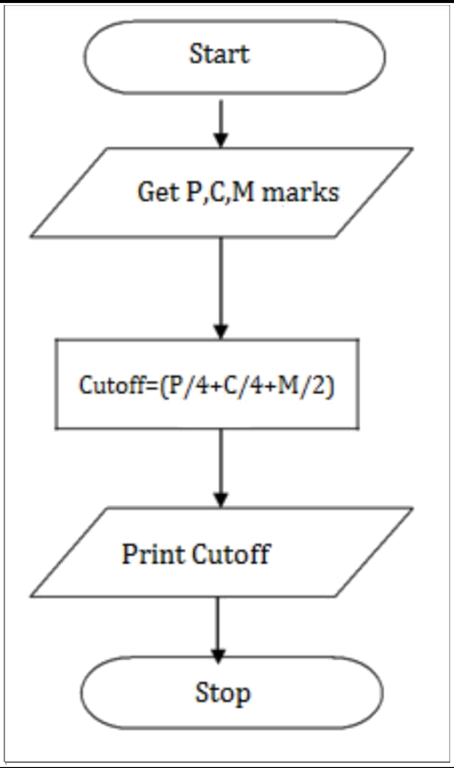
```
IF(n==1) THEN
      fact=1
      RETURN fact
ELSE
     RETURN fact=n*factorial(n-1)
```

**More examples:**

| Write an algorithm to find area of a rectangle | | |
|---|---|---|
| **Step 1:** Start<br>**Step 2:** get l,b values<br>**Step 3:** Calculate A=l*b<br>**Step 4:** Display A<br>**Step 5:** Stop |  | BEGIN<br>READ l,b<br>CALCULATE A=l*b<br>DISPLAY A<br>END |

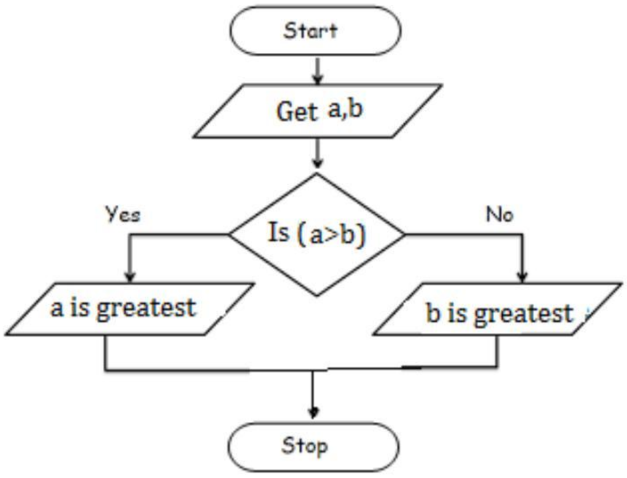
| Write an algorithm for Calculating area and circumference of circle | | |
|---|---|---|
| **Step 1:** Start<br>**Step 2:** get r value<br>**Step 3:** Calculate A=3.14*r*r<br>**Step 4:** Calculate C=2.3.14*r<br>**Step 5:** Display A,C<br>**Step 6**: Stop |  START<br>Get r<br>A=3.14*r*r<br>C=2*3.14*r<br>Print A ,C<br>STOP | BEGIN<br>READ r<br>CALCULATE A and C<br>A=3.14*r*r<br>C=2*3.14*r<br>DISPLAY A<br>END |

| **Write an algorithm for Calculating simple interest** | | |
|---|---|---|
| **Step 1**: Start<br>**Step 2**: get P, n, r value<br>**Step3:**Calculate<br>SI=(p*n*r)/100<br>**Step 4**: Display S<br>**Step 5**: Stop |  | BEGIN<br>READ P, n, r<br>CALCULATE S<br>SI=(p*n*r)/100<br>DISPLAY SI<br>END |

| **Write an algorithm for Calculating engineering cutoff** | | |
|---|---|---|
| Step 1: Start<br>Step2: get P,C,M value<br>Step3:calculate<br>Cutoff= (P/4+C/4+M/2)<br>Step 4: Display Cutoff<br>Step 5: Stop |  Start<br>Get P,C,M marks<br>Cutoff=(P/4+C/4+M/2)<br>Print Cutoff<br>Stop | BEGIN<br>READ P,C,M<br>CALCULATE<br>Cutoff= (P/4+C/4+M/2)<br>DISPLAY Cutoff<br>END |

| **To check greatest of two numbers** |
|---|
| Step 1: Start<br>Step 2: get a,b value<br>Step 3: check if(a>b) print a is greater<br>Step 4: else b is greater<br>Step 5: Stop |

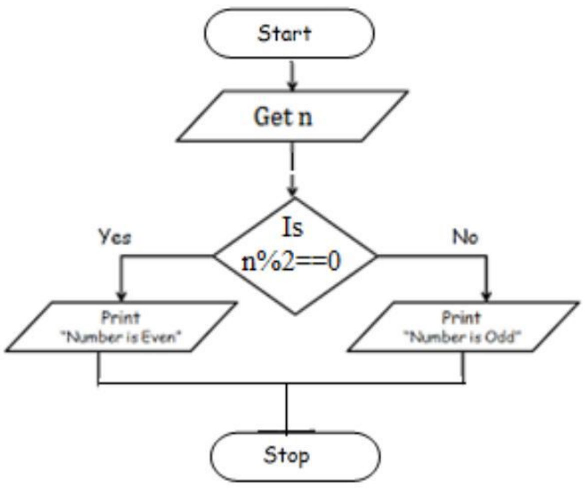
```
BEGIN
READ a,b
IF (a>b) THEN
DISPLAY a is greater
ELSE
DISPLAY b is greater
END IF
END
```



**To check leap year or not**

```
Step 1: Start
Step 2: get y
Step 3: if(y%4==0) print leap year
Step 4: else print not leap year
Step 5: Stop
```

```
BEGIN
READ y
IF (y%4==0) THEN
DISPLAY leap year
ELSE
DISPLAY not leap year
END IF
END
```



## To check positive or negative number

**Step 1:** Start

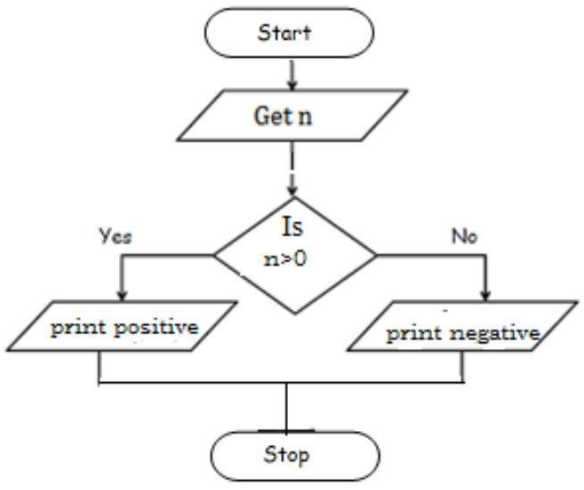
**Step 2:** get num

**Step 3:** check if(num>0) print a is positive

**Step 4:** else num is negative

**Step 5:** Stop

```
BEGIN
READ num
IF (num>0) THEN
DISPLAY num is positive
ELSE
DISPLAY num is negative
END IF
END
```



## To check odd or even number

Step 1: Start

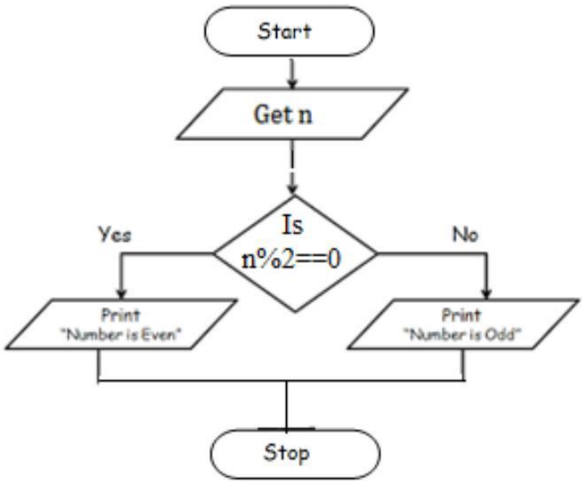
Step 2: get num

Step 3: check if(num%2==0) print num is even

Step 4: else num is odd

Step 5: Stop

```
BEGIN
READ num
IF (num%2==0) THEN
DISPLAY num is even
ELSE
DISPLAY num is odd
END IF
END
```



## To check greatest of three numbers

Step1: Start

Step2: Get A, B, C

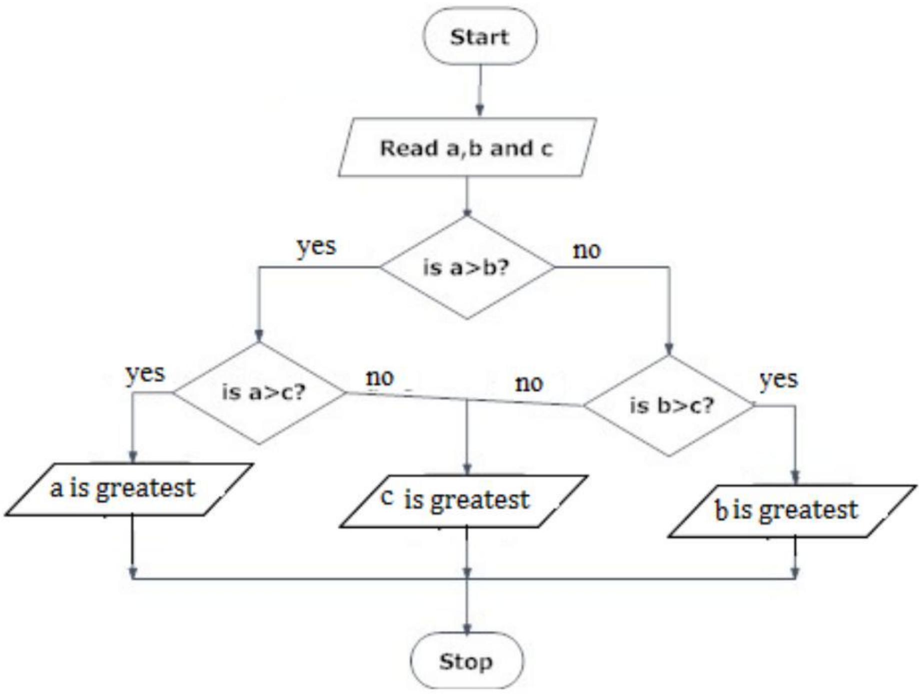
Step3: if(A>B) goto Step4 else goto step5

Step4: If(A>C) print A else print C

Step5: If(B>C) print B else print C

Step6: Stop

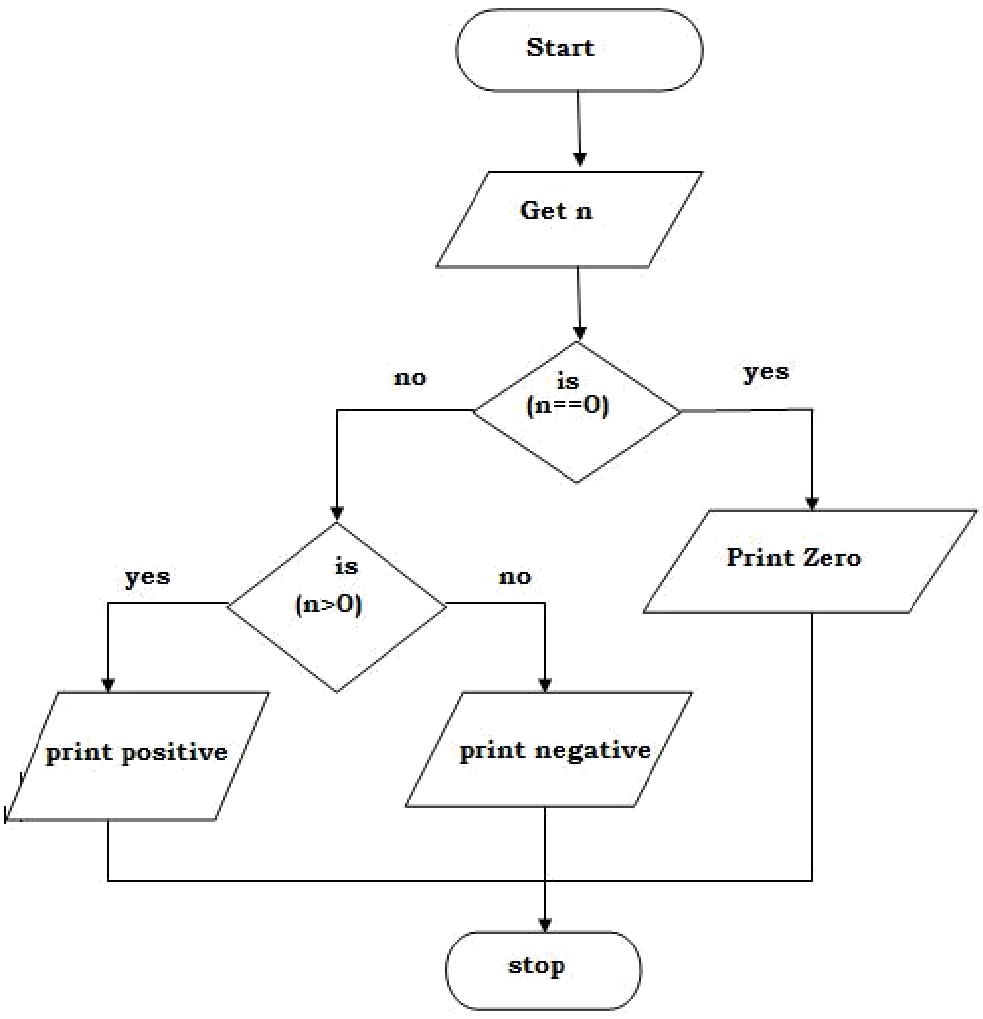
```
BEGIN
READ a, b, c
IF (a>b) THEN
    IF(a>c) THEN
        DISPLAY a is greater
    ELSE
        DISPLAY c is greater
    END IF
ELSE
    IF(b>c) THEN
        DISPLAY b is greater
    ELSE
        DISPLAY c is greater
    END IF
END IF
END
```



**Write an algorithm to check whether given number is +ve, -ve or zero.**

**Step 1:** Start
**Step 2:** Get n value.
**Step 3:** if (n ==0) print “Given number is Zero” Else goto step4
**Step 4:** if (n > 0) then Print “Given number is +ve”
**Step 5:** else Print “Given number is -ve”
**Step 6:** Stop

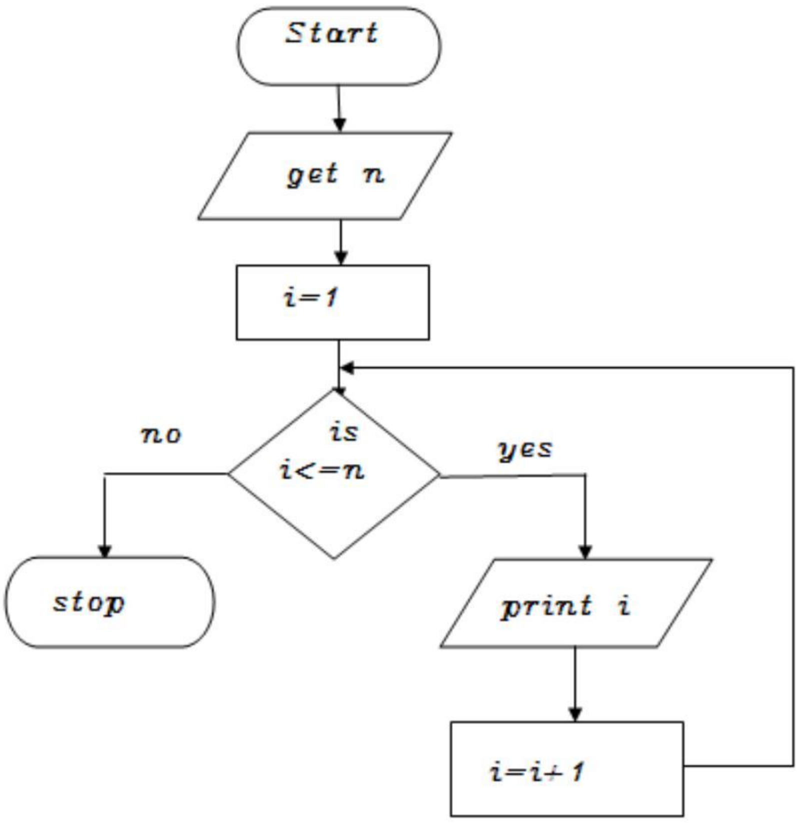
```
BEGIN
GET n
IF(n==0) THEN
    DISPLAY “ n is zero”
ELSE
    IF(n>0) THEN
          DISPLAY “n is positive”
    ELSE
          DISPLAY “n is positive”
     END IF
END IF
END
```



## Write an algorithm to print all natural numbers up to n

**Step 1:** Start
**Step 2:** get n value.
**Step 3:** initialize i=1
**Step 4:** if (i<=n) go to step 5 else go to step 8
**Step 5:** Print i value
**step 6 :** increment i value by 1
**Step 7:** go to step 4
**Step 8:** Stop

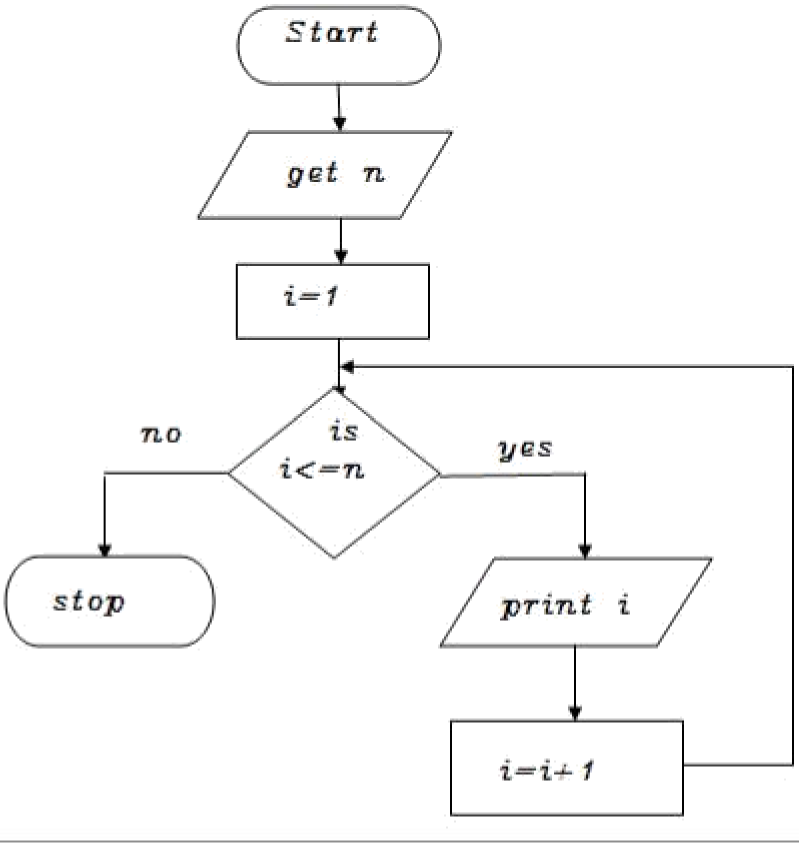
```
BEGIN
GET n
INITIALIZE i=1
WHILE(i<=n) DO
    PRINT i
    i=i+1
ENDWHILE
END
```



**Write an algorithm to print n odd numbers**

```
Step 1: start
step 2: get n value
step 3: set initial value i=1
step 4: check if(i<=n) goto step 5 else goto step 8
step 5: print i value
step 6: increment i value by 2
step 7: goto step 4
step 8: stop
```

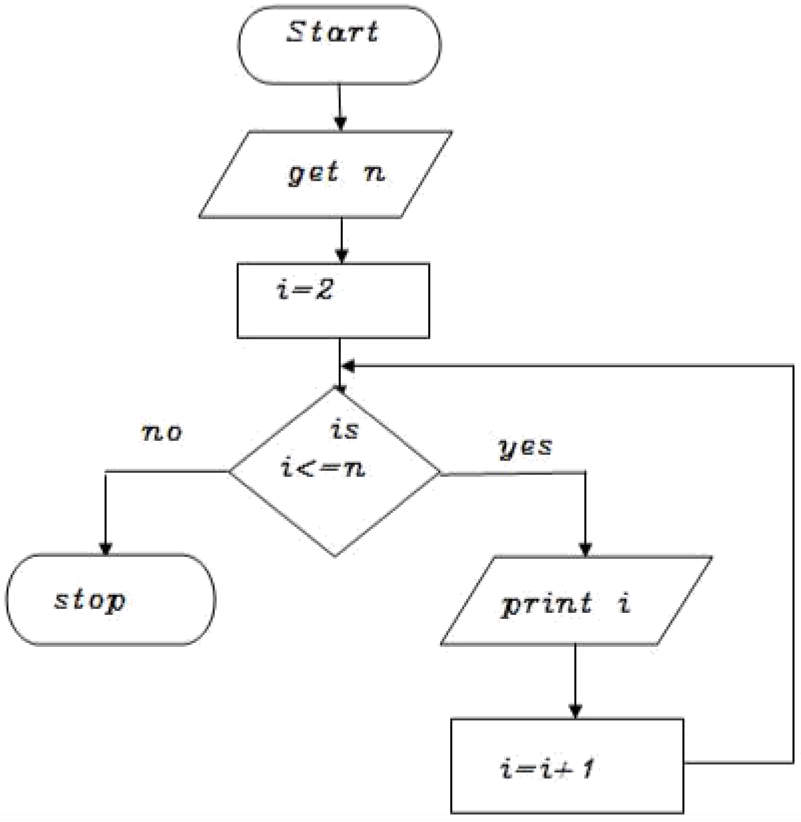
```
BEGIN
GET n
INITIALIZE i=1
WHILE(i<=n) DO
    PRINT i
    i=i+2
ENDWHILE
END
```



**Write an algorithm to print n even numbers**

```
Step 1: start
step 2: get n value
step 3: set initial value i=2
step 4: check if(i<=n) goto step 5 else goto step8
step 5: print i value
step 6: increment i value by 2
step 7: goto step 4
step 8: stop
```

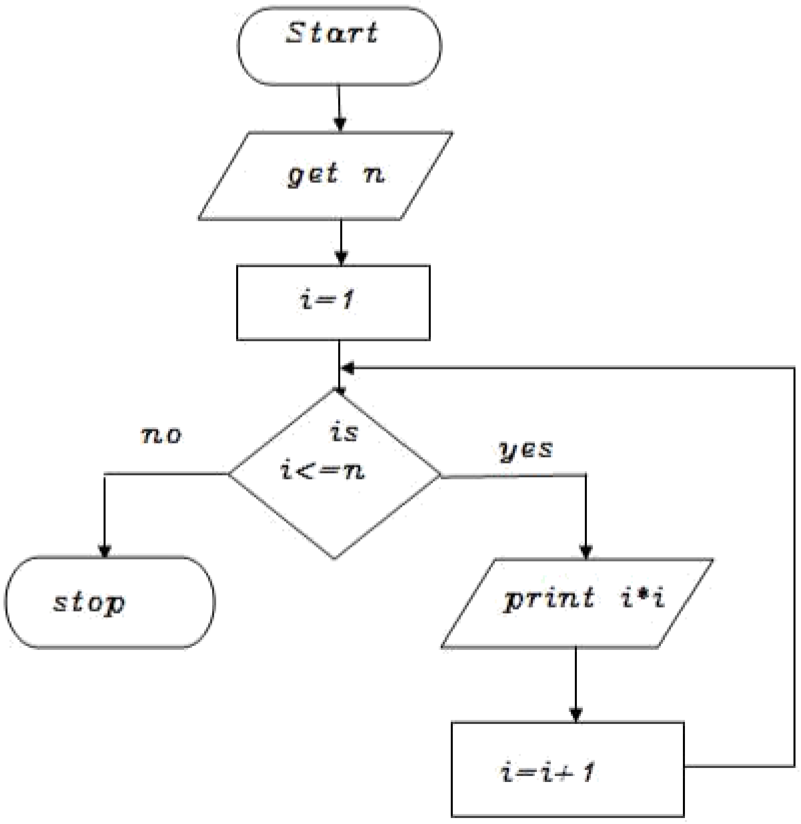
```
BEGIN
GET n
INITIALIZE i=2
WHILE(i<=n) DO
    PRINT i
    i=i+2
ENDWHILE
END
```



**Write an algorithm to print squares of a number**

```
Step 1: start
step 2: get n value
step 3: set initial value i=1
step 4: check i value if(i<=n) goto step 5 else goto step8
step 5: print i*i value
step 6: increment i value by 1
step 7: goto step 4
step 8: stop
```

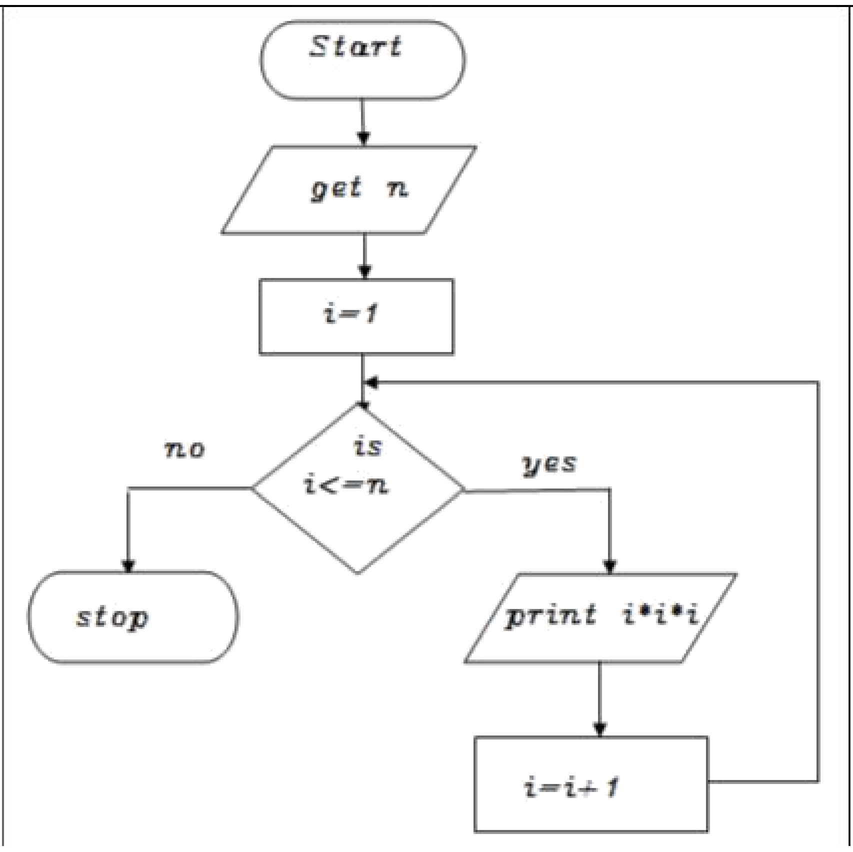
```
BEGIN
GET n
INITIALIZE i=1
WHILE(i<=n) DO
    PRINT i*i
    i=i+2
ENDWHILE
END
```



**Write an algorithm to print to print cubes of a number**

Step 1: start
step 2: get n value
step 3: set initial value i=1
step 4: check i value if(i<=n) goto step 5 else goto step8
step 5: print i*i *i value
step 6: increment i value by 1
step 7: goto step 4
step 8: stop

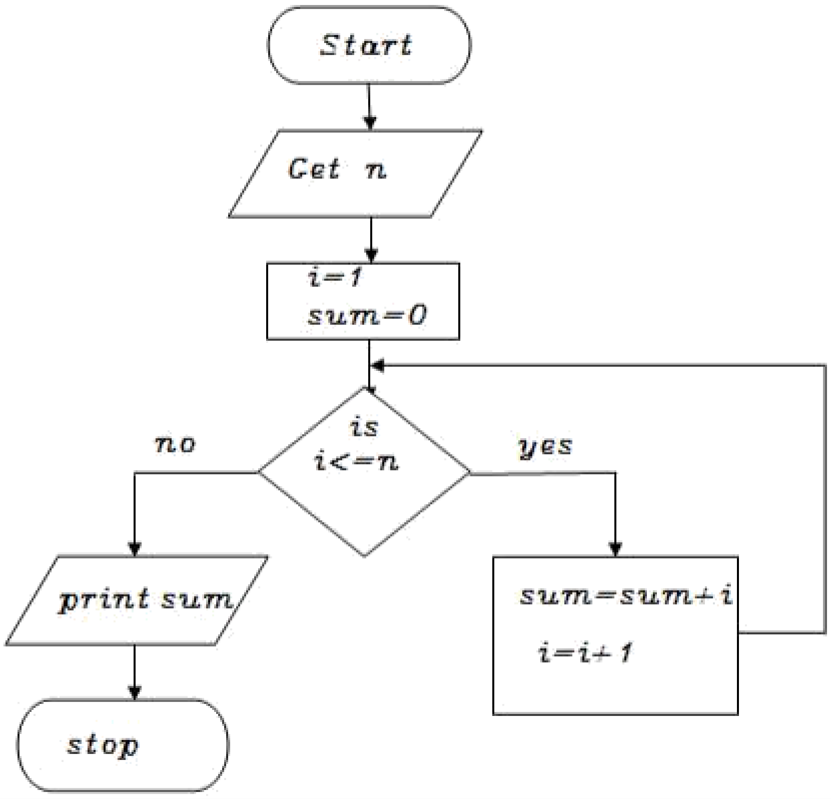
```
BEGIN
GET n
INITIALIZE i=1
WHILE(i<=n) DO
    PRINT i*i*i
    i=i+2
ENDWHILE
END
```



**Write an algorithm to find sum of a given number**

Step 1: start
step 2: get n value
step 3: set initial value i=1, sum=0
Step 4: check i value if(i<=n) goto step 5 else goto step8
step 5: calculate sum=sum+i
step 6: increment i value by 1
step 7: goto step 4
step 8: print sum value
step 9: stop

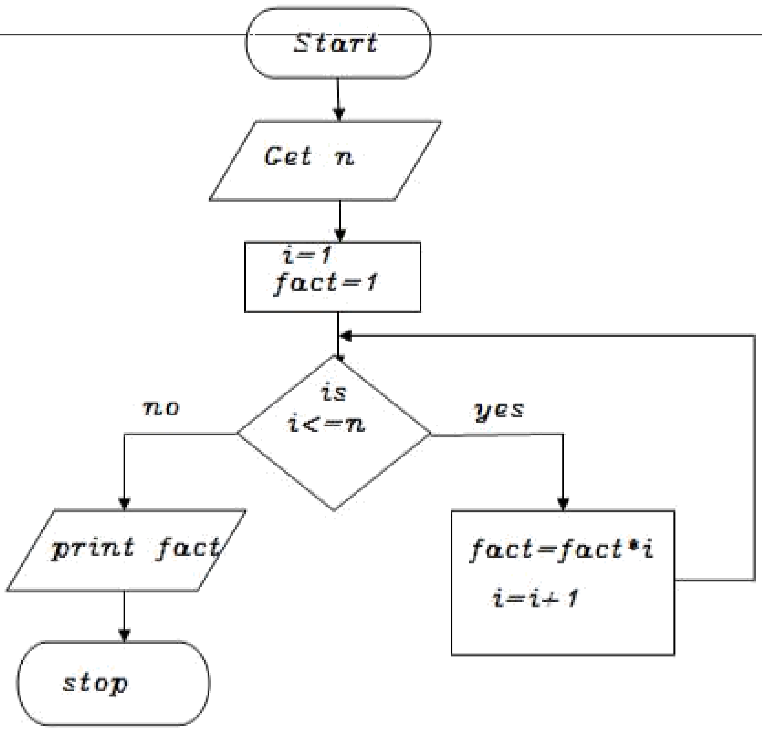
```
BEGIN
GET n
INITIALIZE i=1,sum=0
WHILE(i<=n) DO
    sum=sum+i
    i=i+1
ENDWHILE
PRINT sum
END
```



## Write an algorithm to find factorial of a given number

```
Step 1: start
step 2: get n value
step 3: set initial value i=1, fact=1
Step 4: check i value if(i<=n) goto step 5 else goto step8
step 5: calculate fact=fact*i
step 6: increment i value by 1
step 7: goto step 4
step 8: print fact value
step 9: stop
```

```
BEGIN
GET n
INITIALIZE i=1,fact=1
WHILE(i<=n) DO
    fact=fact*i
    i=i+1
ENDWHILE
PRINT fact
END
```



## ILLUSTRATIVE PROBLEM

1.Guess an integer in a range

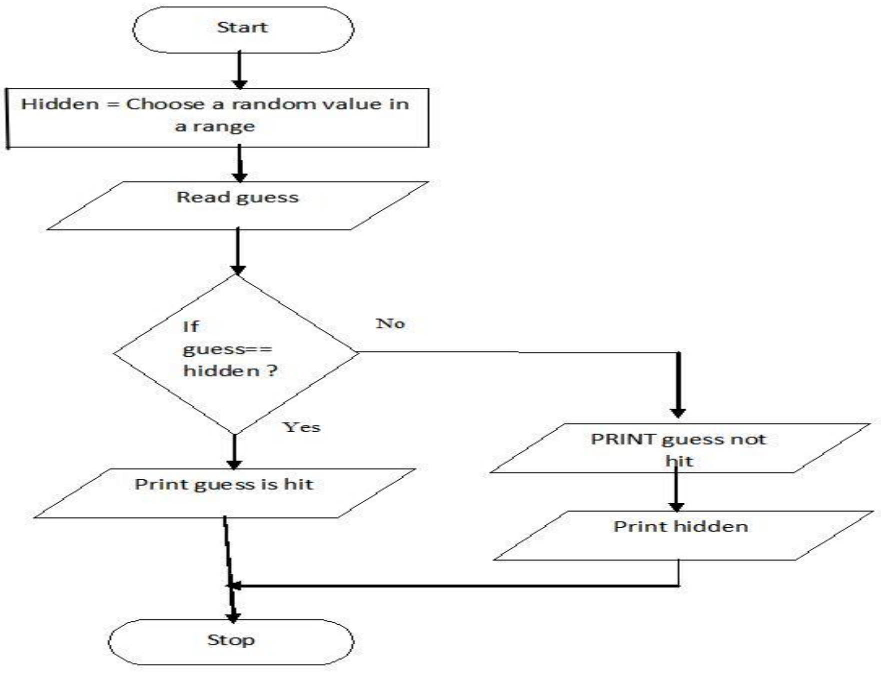
**Algorithm:**

```
Step1: Start
Step 2: Declare hidden, guess,range=1 to 100
Step 3: Compute hidden= Choose a random value in a range
Step 4: Read guess
Step 5: If guess=hidden, then Print
                Guess is hit
        Else
                Print Guess not hit
                Print hidden
Step 6: Stop
```

**Pseudocode:**

```
BEGIN
COMPUTE hidden=random value in a range
READ guess
IF guess=hidden, then PRINT
        Guess is hit
ELSE
        PRINT Guess not hit
        PRINT hidden
END IF-ELSE
END
```

**Flowchart:**



2.Find minimum in a list

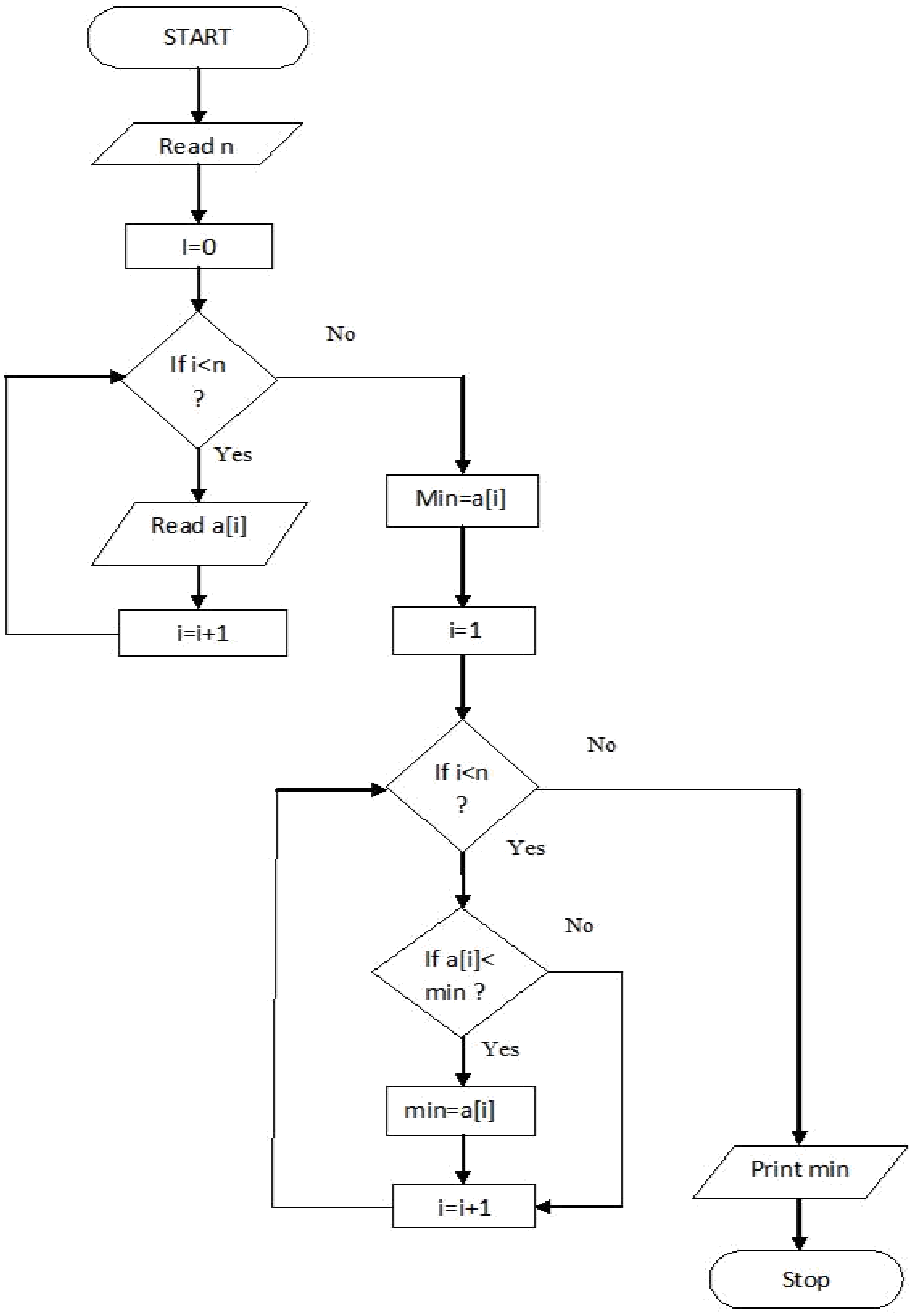
**Algorithm:** Step 1:
Start Step 2: Read n
Step 3:Initialize i=0

```
Step 4: If i<n, then goto step 4.1, 4.2 else goto step 5
        Step4.1: Read a[i]
        Step 4.2: i=i+1 goto step 4
Step 5: Compute min=a[0]
Step 6: Initialize i=1
Step 7: If i<n, then go to step 8 else goto step 10
Step 8: If a[i]<min, then goto step 8.1,8.2 else goto 8.2
        Step 8.1: min=a[i]
        Step 8.2: i=i+1 goto 7
Step 9: Print min
Step 10: Stop
```

**Pseudocode:**

```
        BEGIN
        READ n
        FOR i=0 to n, then READ
                a[i] INCREMENT
                i
        END FOR COMPUTE
        min=a[0] FOR i=1 to n,
        then
                IF a[i]<min, then CALCULATE
                        min=a[i] INCREMENT i
                ELSE
                        INCREMENT i
                END IF-ELSE
        END FOR
        PRINT min
END
```

**Flowchart:**


START
Read n
I=0
If i<n ?
No
Yes
Read a[i]
i=i+1
Min=a[i]
i=1
If i<n ?
No
Yes
If a[i]< min ?
No
Yes
min=a[i]
i=i+1
Print min
Stop

3.Insert a card in a list of sorted cards

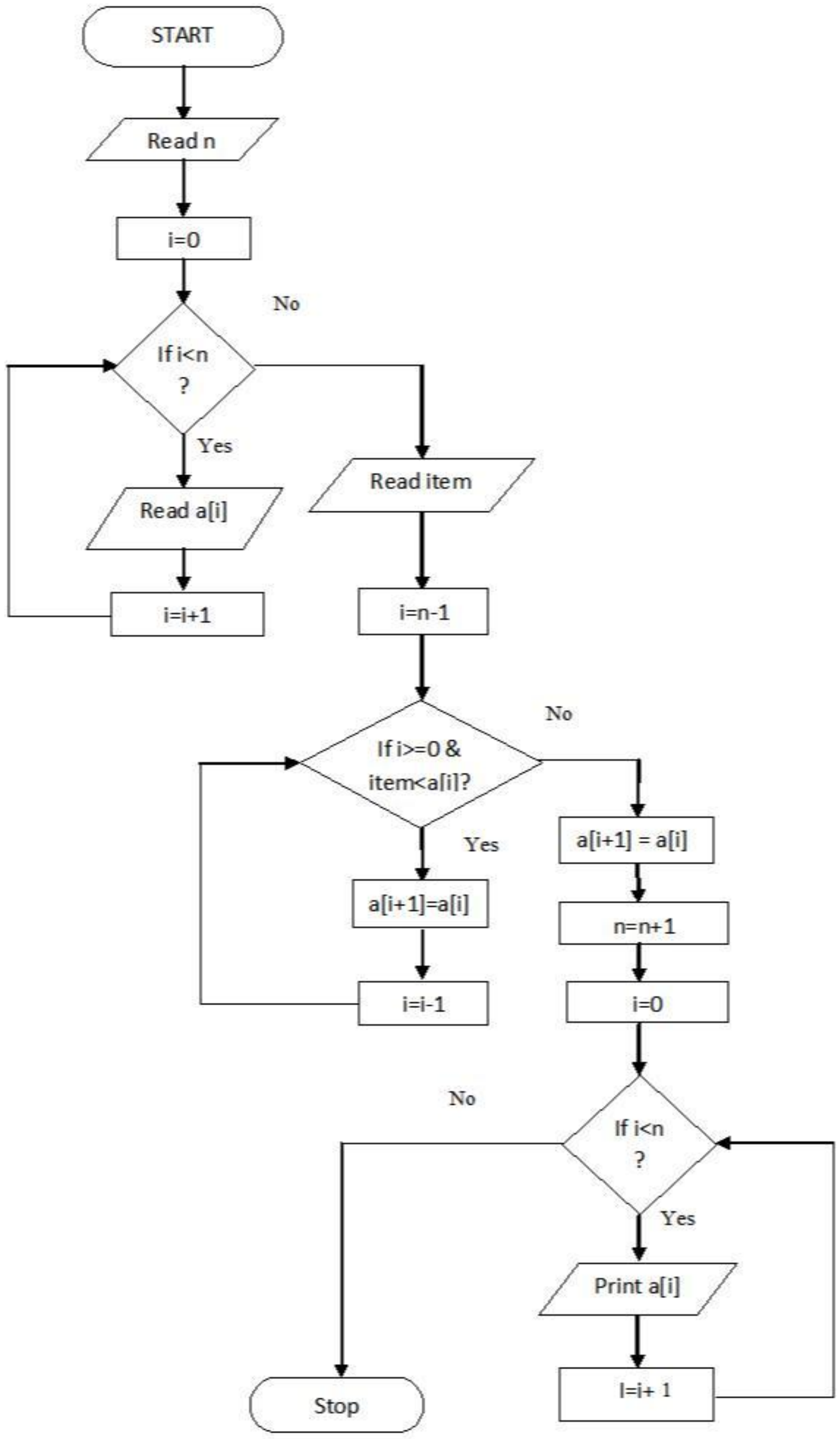
```
Algorithm: Step 1:
Start Step 2: Read n
Step 3:Initialize i=0

Step 4: If i<n, then goto step 4.1, 4.2 else goto step 5
        Step4.1: Read a[i]
        Step 4.2: i=i+1 goto step 4
Step 5: Read item
Step 6: Calculate i=n-1
Step 7: If i>=0 and item<a[i], then go to step 7.1, 7.2 else goto step 8
        Step 7.1: a[i+1]=a[i]
        Step 7.2: i=i-1 goto step 7
Step 8: Compute a[i+1]=item
Step 9: Compute n=n+1
Step 10: If i<n, then goto step 10.1, 10.2 else goto step 11
        Step10.1: Print a[i]
        Step10.2: i=i+1 goto step 10
Step 11: Stop
```

**Pseudocode:**

```
BEGIN
READ n
FOR i=0 to n, then READ
        a[i] INCREMENT
        i
END FOR
READ item
FOR i=n-1 to 0 and item<a[i], then
        CALCULATE a[i+1]=a[i]
        DECREMENT i
END FOR COMPUTE
a[i+1]=a[i] COMPUTE
n=n+1 FOR i=0 to n, then
        PRINT a[i]
        INCREMENT i
END FOR
END
```

**Flowchart:**


START
Read n
i=0
No
If i<n ?
Yes
Read a[i]
i=i+1
Read item
i=n-1
No
If i>=0 & item<a[i]?
Yes
a[i+1]=a[i]
i=i-1
a[i+1] = a[i]
n=n+1
i=0
No
If i<n ?
Yes
Print a[i]
I=i+ 1
Stop

4. Tower of Hanoi

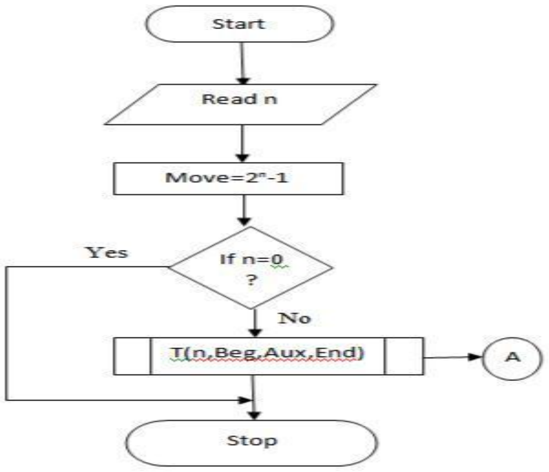
**Algorithm:**

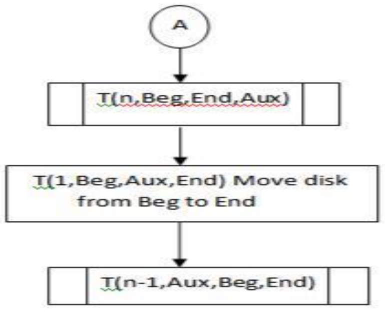
```
Step 1: Start
Step 2: Read n
Step 3: Calculate move=pow(2,n)-1
Step 4: Function call T(n,Beg,Aux,End) recursively until n=0
    Step 4.1: If n=0, then goto step 5 else goto step 4.2 Step
    4.2: T(n-1,Beg,End,Aux)
        T(1,Beg,Aux,End) , Move disk from source to destination
        T(n-1,Aux,Beg,End)
Step 5: Stop
```

**Pseudcode:**

```
BEGIN
READ n
CALCULATE move=pow(2,n)-1
FUNCTION T(n,Beg,Aux,End) Recursively until n=0
PROCEDURE IF
n=0 then,
    No disk to move
Else
    T(n-1,Beg,End,Aux)
    T(1,Beg,Aux,End), move disk from source to destination
    T(n-1,Aux,Beg,End)
END PROCEDURE
END
```

**Flowchart:**



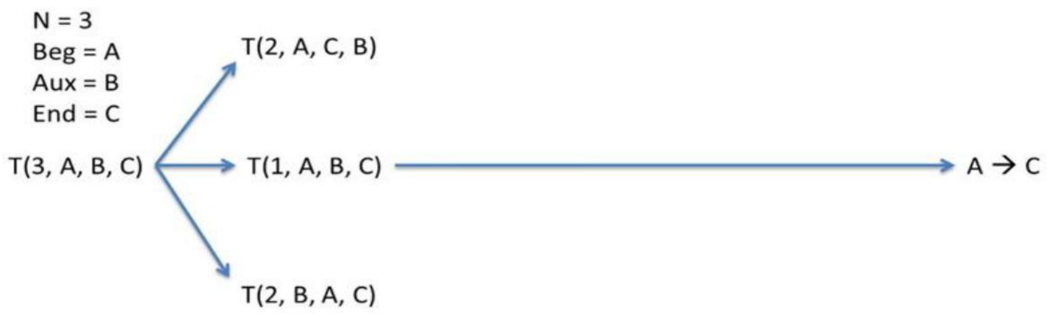


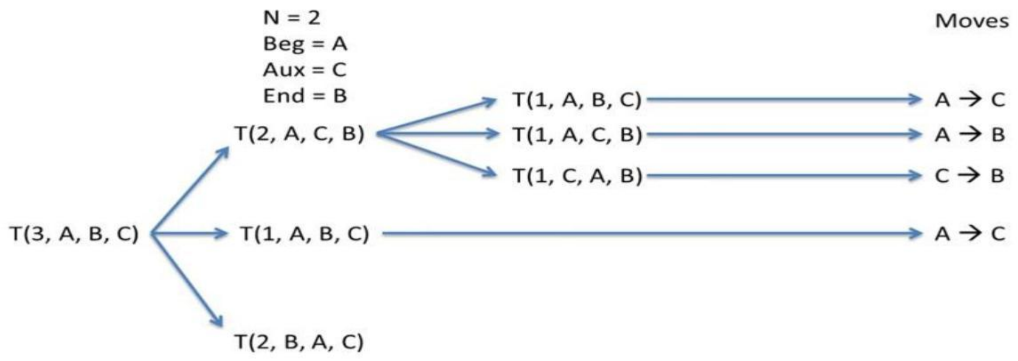
**Procedure to solve Tower of Hanoi**

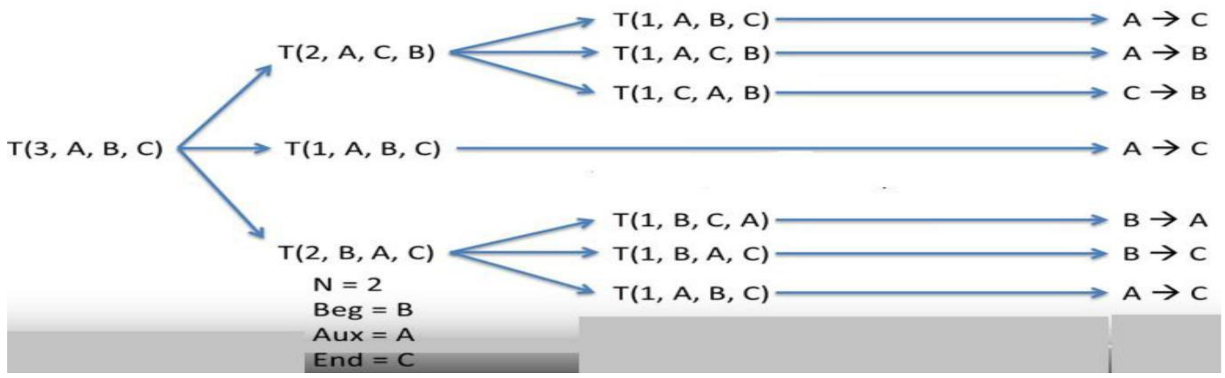
The goal of the puzzle is to move all the disks from leftmost peg to rightmost peg.

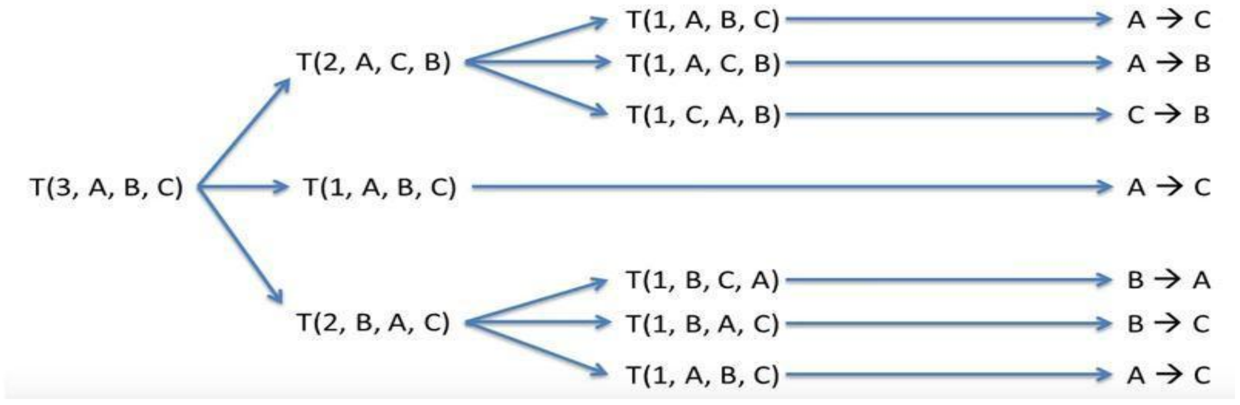
1. Move only one disk at a time.
2. A larger disk may not be p1aced on top of a smaller disk.

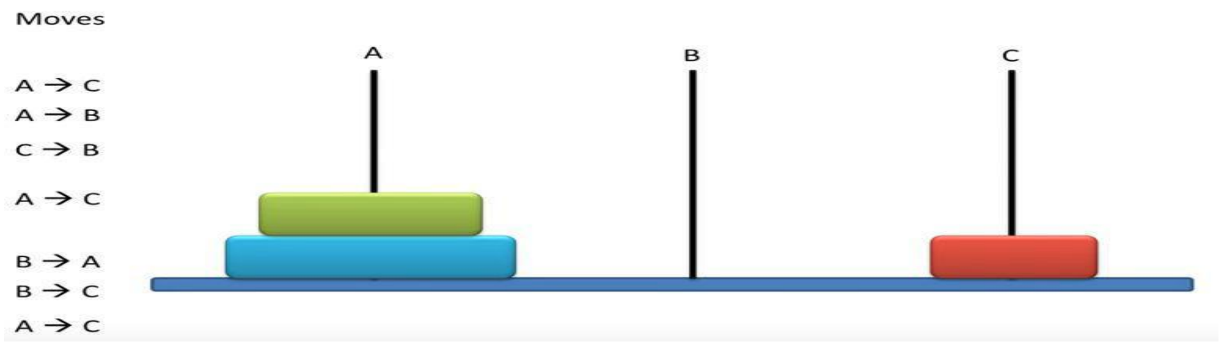
For example, consider n=3 disks

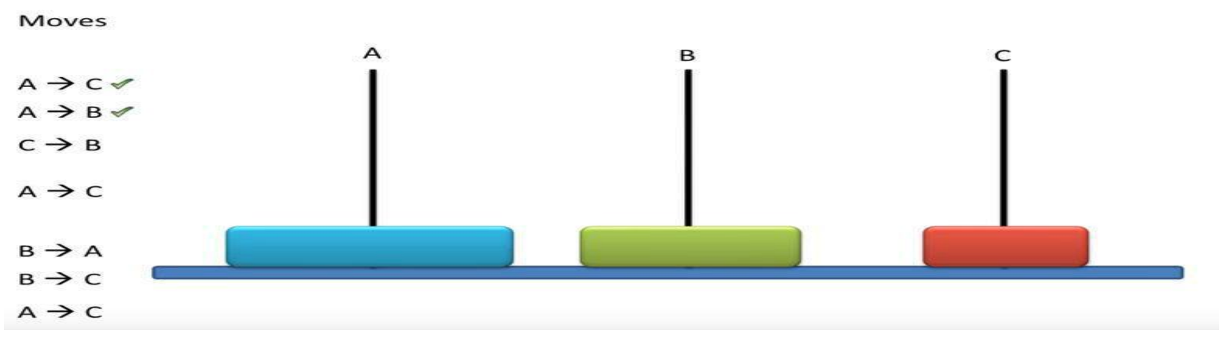


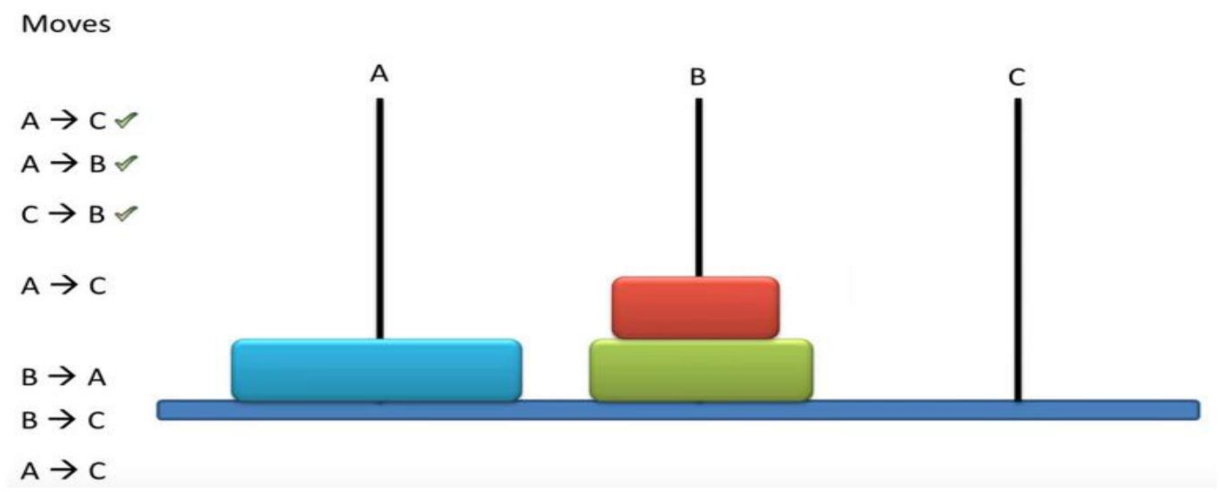


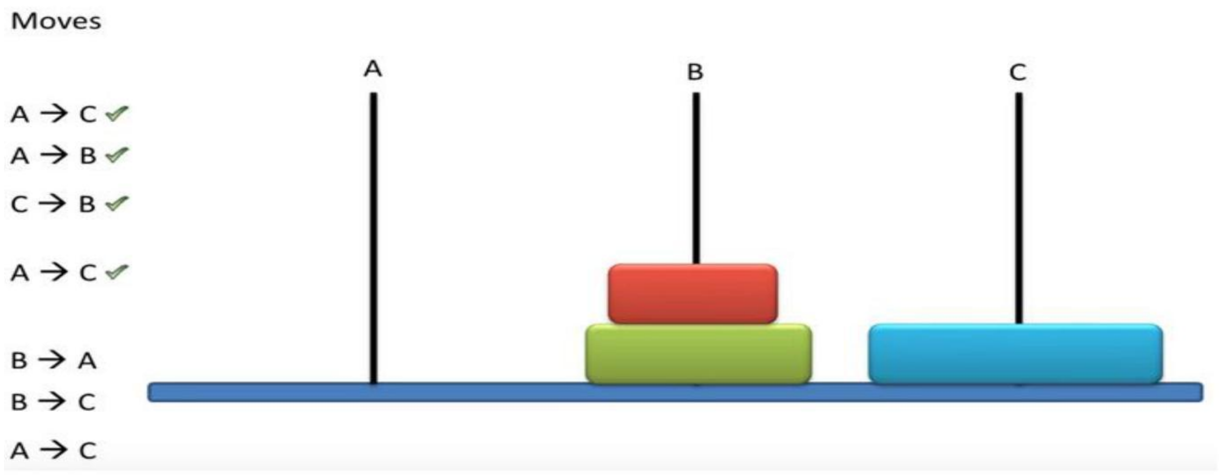


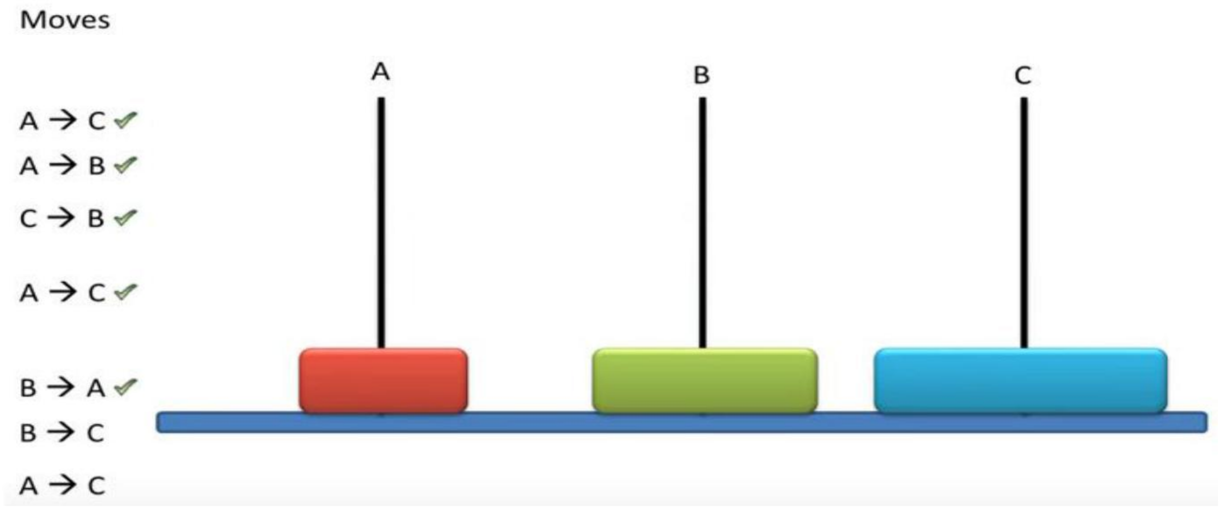

T(3, A, B, C)
T(2, A, C, B)
T(1, A, B, C)
T(1, A, C, B)
T(1, C, A, B)
T(1, A, B, C)
T(2, B, A, C)
T(1, B, C, A)
T(1, B, A, C)
T(1, A, B, C)
A → C
A → B
C → B
A → C
B → A
B → C
A → C

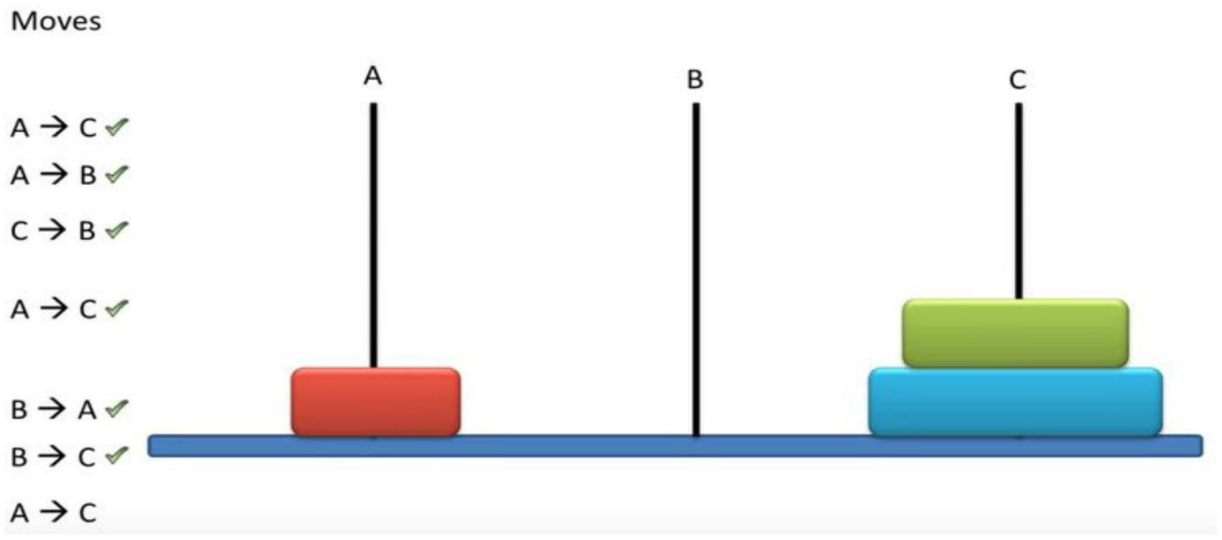

Moves
A → C
A → B
C → B
A → C
B → A
B → C
A → C
A
B
C

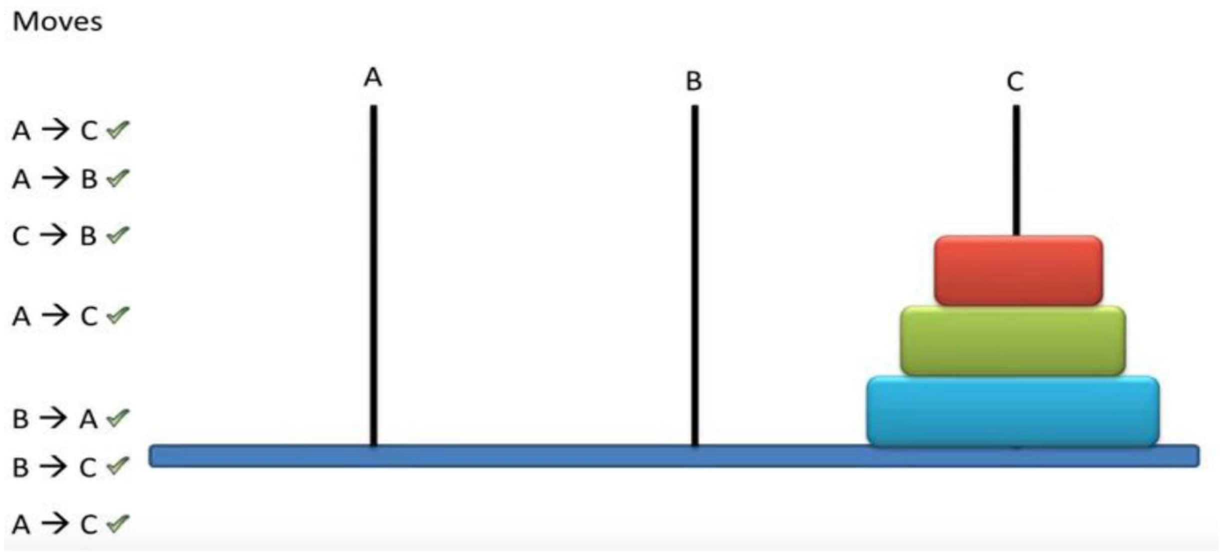

Moves
A → C ✓
A → B ✓
C → B
A → C
B → A
B → C
A → C
A
B
C


Moves
A
B
C
A → C ✓
A → B ✓
C → B ✓
A → C
B → A
B → C
A → C


Moves
A
B
C
A → C ✓
A → B ✓
C → B ✓
A → C ✓
B → A
B → C
A → C


Moves
A
B
C
A → C ✓
A → B ✓
C → B ✓
A → C ✓
B → A ✓
B → C
A → C


Moves
A
B
C
A → C ✓
A → B ✓
C → B ✓
A → C ✓
B → A ✓
B → C ✓
A → C


Moves
A
B
C
A → C ✓
A → B ✓
C → B ✓
A → C ✓
B → A ✓
B → C ✓
A → C ✓