UNIT III - DECOMPOSITION AND ALGORITHMIC THINKING

Decomposition,
Algorithmic thinking – creating oral algorithms for everyday tasks –
Visualizing algorithms through sequence of steps, pseudocode, flow charts, selection, iteration, functions, procedures and parameter

DECOMPOSITION

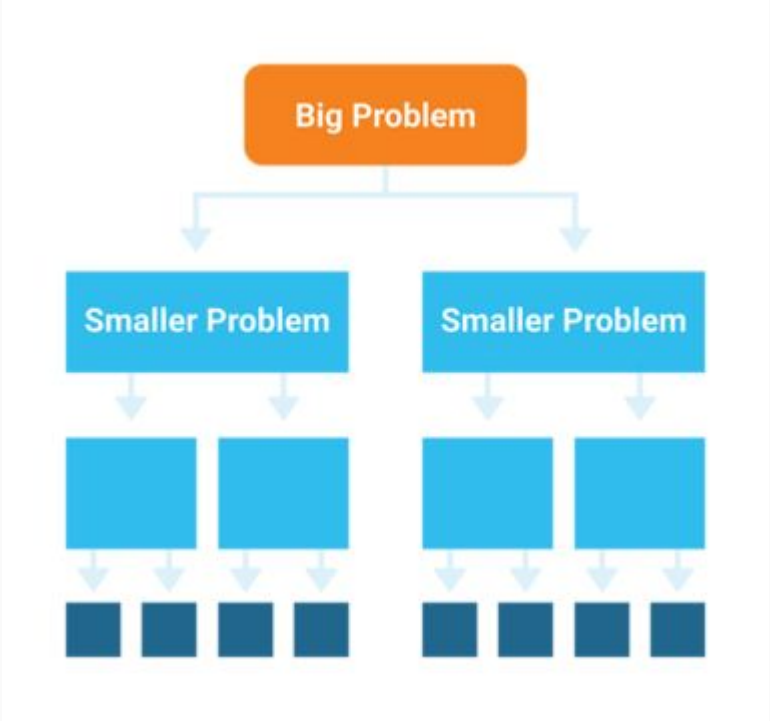
Decomposition in computational thinking is the **process of breaking down a problem into a number of smaller problems** that can more easily be addressed.

It is considered to be an effective method of solving complex problems as it helps to

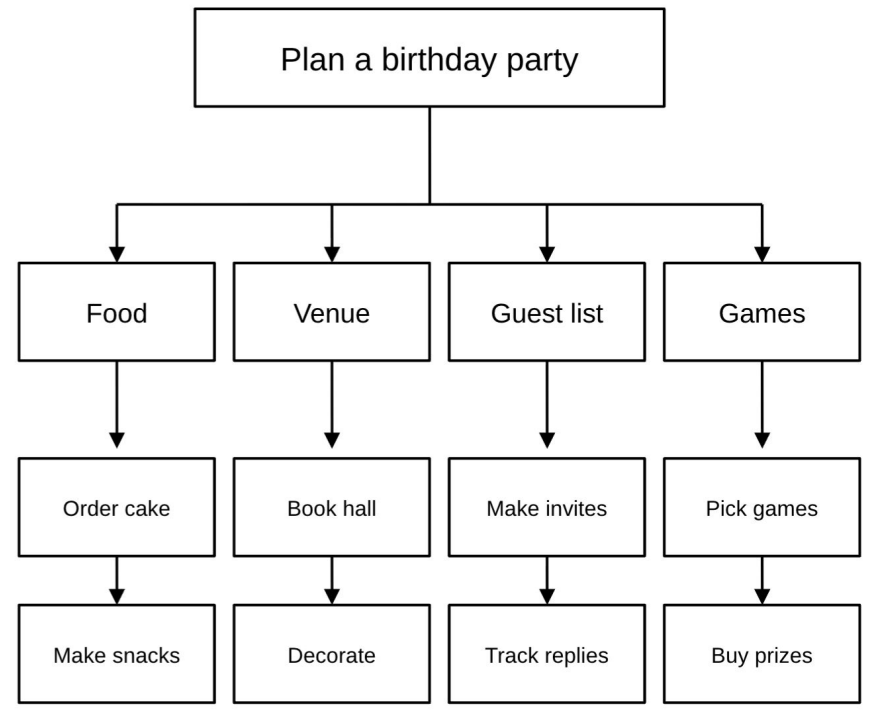
- **identify patterns**,
- **eliminate extraneous details** and
- **solve the problem step by step** instead of trying to do so all at once.

Decomposition **aids collaboration**.

If you decompose a problem well (so that the subproblems can be solved independently), then different people can work on different tasks, possibly in parallel



Example of decomposition in everyday life



Problems split into sub-problems, which split again until each piece is simple

ALGORITHMIC THINKING

Consider **Tower Of Hanoi** (Play Tower of Hanoi)

Goal :

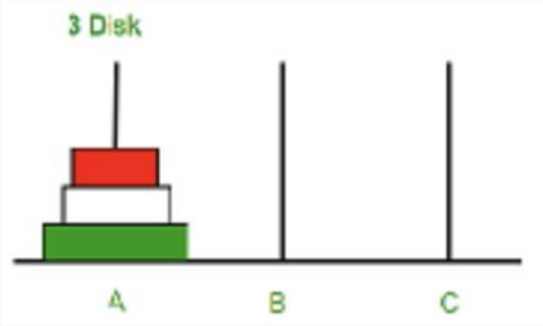
- Move the whole stack of disks from peg A to peg C, in the same order. Peg B is the spare.

Rules :

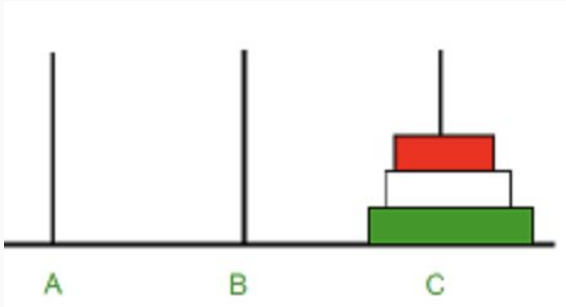
- Move only one disk at a time.
- Move only the top disk of a peg.
- Never place a larger disk on a smaller disk.

Ans in next page

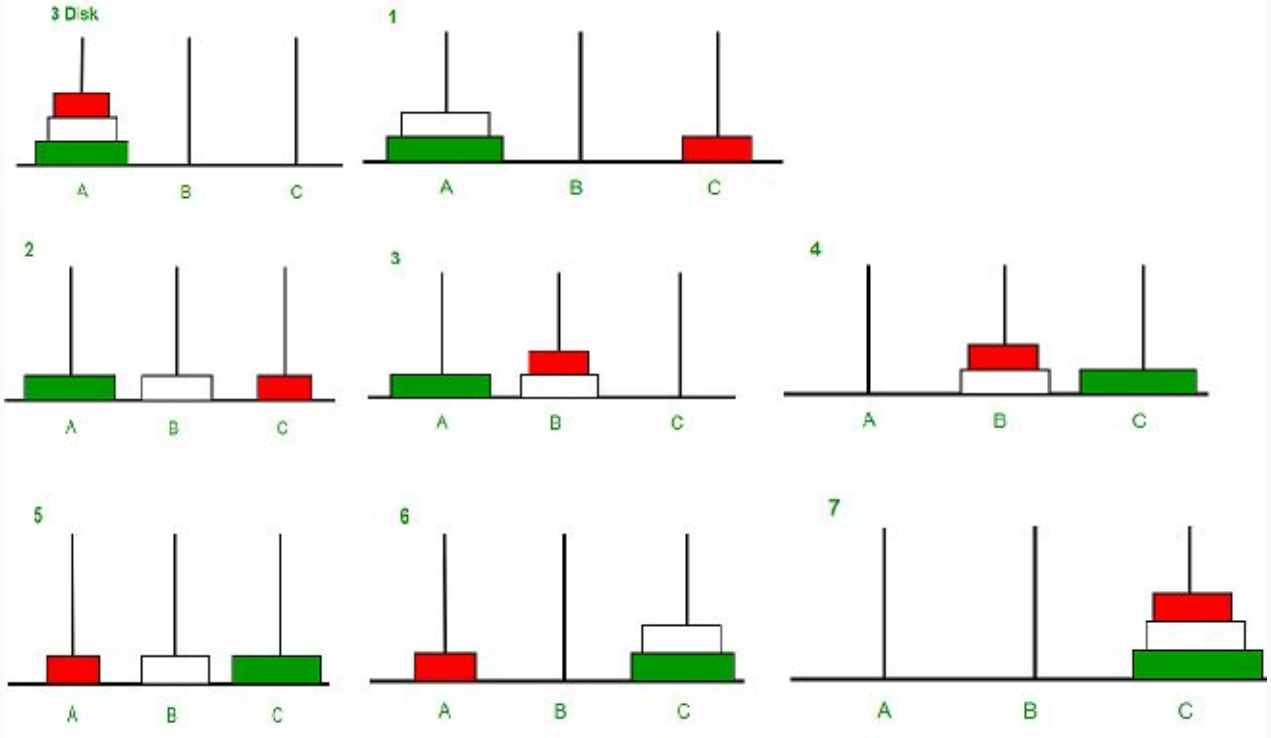
Start (3 disks) :



Goal :



The answer and the logic behind it:



A move is legal only if both are true

- The disk is the top disk of its peg
- The target peg is empty OR its top disk is larger.

Note: Minimum number of moves to move ‘n’ disks from A to C is $T(n)=2^n-1$

What logic can and cannot do ?

Logic CAN	Logic CANNOT
Say whether a move is legal	Choose between the legal moves
Say whether the goal is reached	Decide what to do when the goal is not reached
Describe the rules exactly	Plan the order of all moves

Example:

- At the start, **2 moves are legal** (disk 1 to C or disk 1 to B). Logic cannot rank them.
- Starting with **disk 1 to C finishes in 7 moves** in total. Starting with **disk 1 to B needs 8**.

- **Logic checks the rules but cannot decide the moves.**
- So we write exact steps(algorithm) that make sure **anyone can follow them and get the same answer**.
- That is **algorithmic thinking.**

Algorithm is a **sequence of clearly defined steps** that describe a process to follow a finite set of unambiguous instructions with clear start and end points

An algorithm must:

- give the specified output,
- stop after a finite time, and
- have clearly defined (unambiguous) steps.

Properties of an Algorithm

- **Finiteness:** must stop after a finite number of steps.
  - Example: a loop '**repeat while I >= 1**' with I only increasing never stops, so it is not an algorithm.
- **Definiteness:** every step is precisely and unambiguously defined.
  - Example: '**add sugar**' is vague, '**add two spoons**' is definite.
- **Input:** values given before it begins.
  - Example: the **radius r** for the area of a circle.
- **Output:** results with a specified relation to the input. Intermediate results appear midway, the end result at the finish.
- **Effectiveness:** steps are basic enough to be done exactly, in finite time, with paper and pencil.
  - Example: **add, compare, multiply.**

Oral Algorithms for Everyday Tasks

- **Oral Algorithm :** the steps of a **task spoken aloud in plain language**, in order, so the listener can do it without guessing.
- It is the first draft of an algorithm. Pseudocode and flowcharts are written later from it.
- Good oral steps are
  - short,
  - ordered, and
  - exact (not 'add some sugar' but 'add two spoons of sugar').

Example: making a cup of tea

1. Pour one cup of water into the pot and heat it.
2. While the water is not boiling, wait. <- repetition
3. Put one tea bag in the cup and pour the water.
4. If you want sugar, add two spoons. <- selection
5. Wait three minutes, remove the tea bag.
6. Stop.

Example: Crossing a road

1. Go to the zebra crossing.
2. Look at the signal.
3. While the signal is red, keep waiting.
4. Look left and right.
5. If no vehicle is coming, walk across.
   Otherwise, go back to step 4.
6. Stop on the other side.

VISUALIZING ALGORITHMS

SEQUENCE OF STEPS

- An algorithm written as **numbered, plain-language steps.**
- Each step is one clear action, and the steps run in order, from the first to the last.

How to write sequence of steps ?

- Begin with Start, give each action its own number, show the input and the output, and finish with End.

Example: print area of a circle

```
Step 1: Start
Step 2: Read radius r
Step 3: Area <- 3.14 * r * r
Step 4: Print Area
Step 5: End
```

PSEUDOCODE

- It is a step-by-step description of an **algorithm written in simple English** using a code-like structure.
- It is designed for human understanding, **not for machine execution**, and does not follow the syntax of any programming language.

Why Do We Need Pseudocode ?

- Helps programmers plan the solution before writing actual code
- Makes the logic easy to understand for others reading the solution
- Bridges the gap between algorithm design and coding
- Reduces logical errors before implementation



How to write pseudocode ?

- **Organize tasks** logically and write steps in sequence.
- Use standard programming constructs:
    - IF–ELSE
    - FOR
    - WHILE
- **Indent blocks** properly to improve readability and clarity.
- Use **clear and meaningful names** for variables and steps.
- Write **keywords in CAPITAL** letters (IF, ELSE, WHILE).
- Ensure the pseudocode is:
    - Complete
    - Finite
    - Easy to understand
- Avoid writing **actual programming syntax**—keep it simple and language-independent.

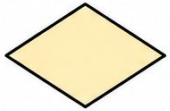
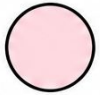
Example: find the largest of three numbers

```
BEGIN
    INPUT num1, num2, num3

    IF (num1 >= num2) AND (num1 >= num3) THEN
        largest = num1
    ELSE IF (num2 >= num1) AND (num2 >= num3) THEN
        largest = num2
    ELSE
        largest = num3
    ENDIF

    PRINT "The largest number is: ", largest
END
```

FLOWCHART

- Flowcharts are **graphical representations** of data, algorithms, or processes, providing a visual approach to understanding code.
- Flowcharts **use symbols/shapes** like arrows, rectangles, and diamonds to properly explain the sequence of steps involved in the algorithm or process.
- It increases the **readability** of the program
- A flowchart
    - shows logic of an algorithm
    - Emphasizes **individual steps and their interconnections**

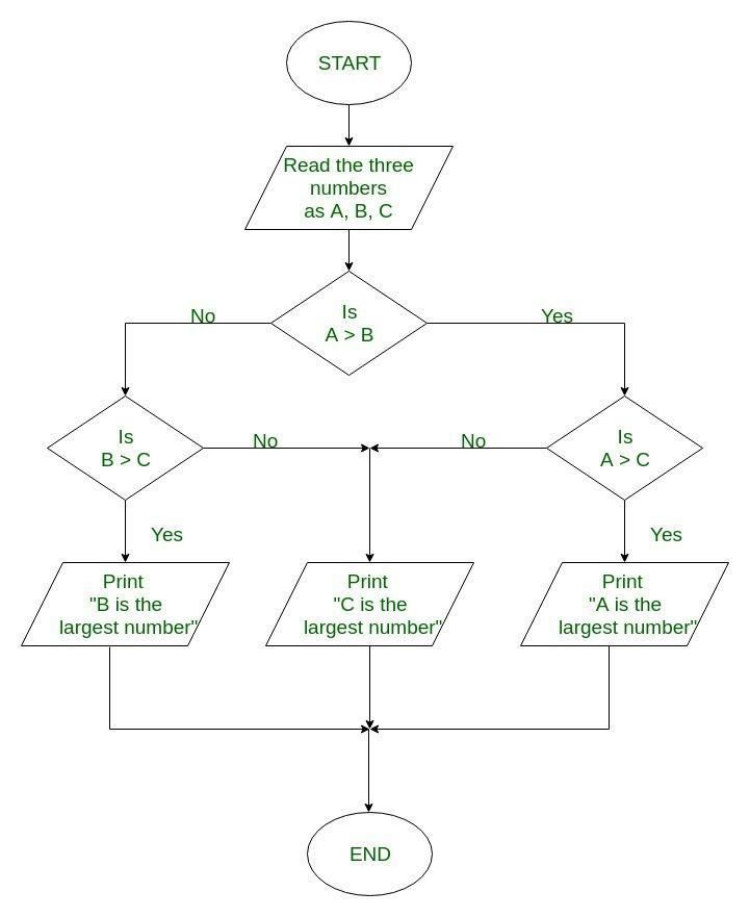
Flowchart symbols

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, process, or interrupt program.
	**Flow Lines**	Shows direction of flow.

General rules of flowcharting

- All boxes are connected with arrows, not plain lines.
- Every symbol has one entry on top and one exit at the bottom, except the Decision symbol and start and end terminals.
- Flow goes top to bottom. Upward flow is allowed only up to 3 symbols.
- Use connectors for breaks: page to page, bottom of page to top, or upward flow of more than 3 symbols.
- Subroutines and interrupt programs have their own separate flow charts.
- Every flowchart starts with a Terminal (or a Predefined Process for a subroutine).
- Every flowchart ends with a Terminal or a continuous loop.

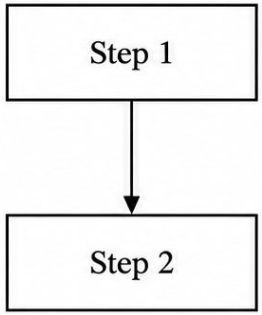
Example: find the largest of three numbers



THREE CONTROL STRUCTURES

Every algorithm can be built from these three. Combining them is common.

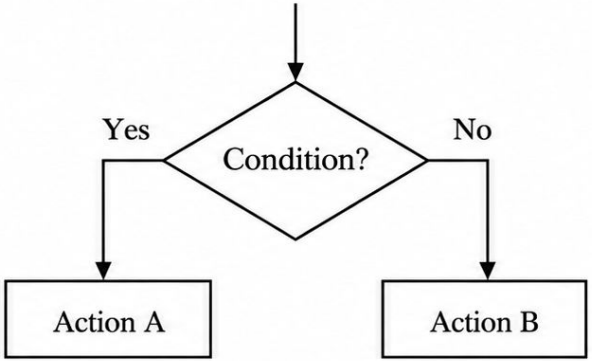
1. Sequence



Statements run one after another, top to bottom. In a flow chart: rectangular process boxes.

Example: Wear socks, then shoes.

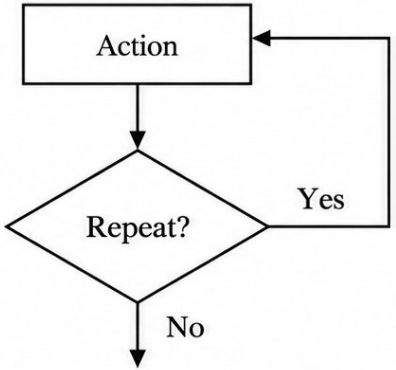
2. Selection



A binary decision (if-then-else). True takes one branch, false the other. Diamond box.

Example: If it rains, take an umbrella, else take sunglasses.

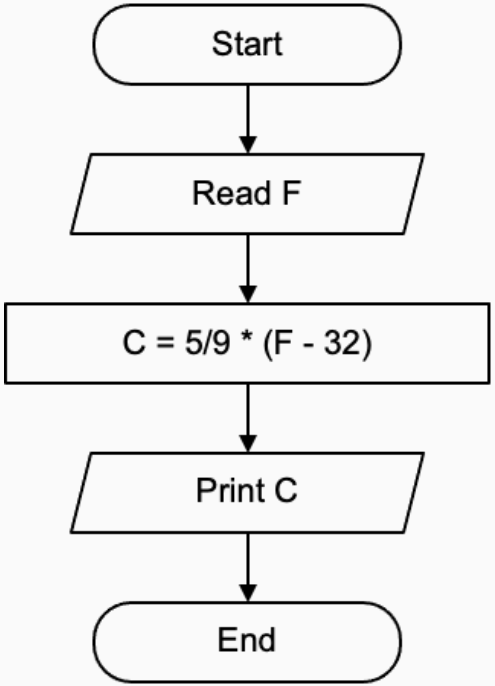
3. Iteration



Repeat steps while a condition holds (while / for). A back arrow marks the loop. Each trip is an iteration.

Example: While the glass is not full, keep pouring.

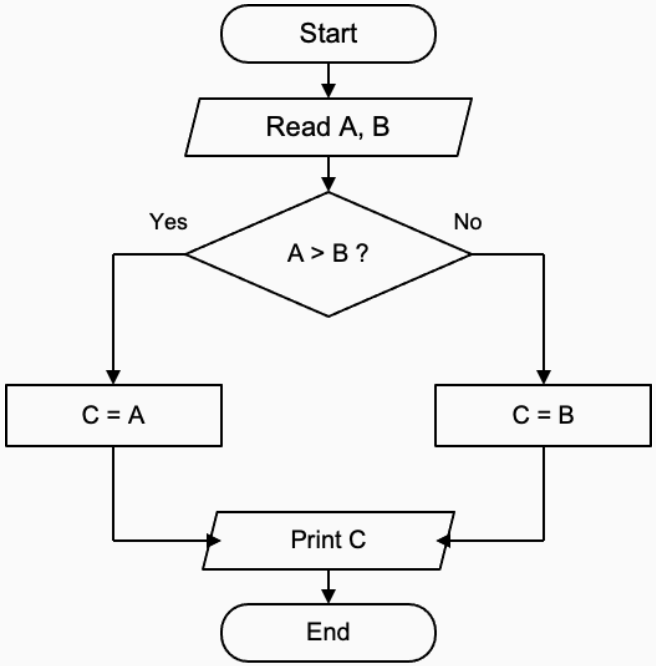
SEQUENCE: Convert fahrenheit to celsius



Input: temperature F.
Output: temperature C.

Each step runs once, in order.
No decisions, no repeats.

SELECTION: Greater of two numbers




```
READ A, B
IF A > B THEN
  C <- A
ELSE
  C <- B
ENDIF
PRINT C
```

- **Relational operators:** <, >, <=, >=, =, != compare two values.
- **Logical operators:** AND (both true), OR (at least one true), NOT (flips true/false).

SELECTION: more examples

Absolute value of X

```
READ x
IF x < 0 THEN
  f <- -x
ELSE
  f <- x
ENDIF
PRINT f
```

Largest of three numbers

```
READ A, B, C
IF A >= B AND A >= C THEN
  Max <- A
ELSE IF B >= C THEN
  Max <- B
ELSE
  Max <- C
ENDIF
PRINT Max
```

ITERATION

- Repeating a step or group of steps while a condition holds. One trip around the loop is an iteration.

Types of loops:

- **Unbounded loop (WHILE):** the count is not known in advance. It ends when the condition fails.
- **Bounded loop (FOR):** the count is known beforehand.
- **Infinite loop:** a loop whose condition never becomes false.

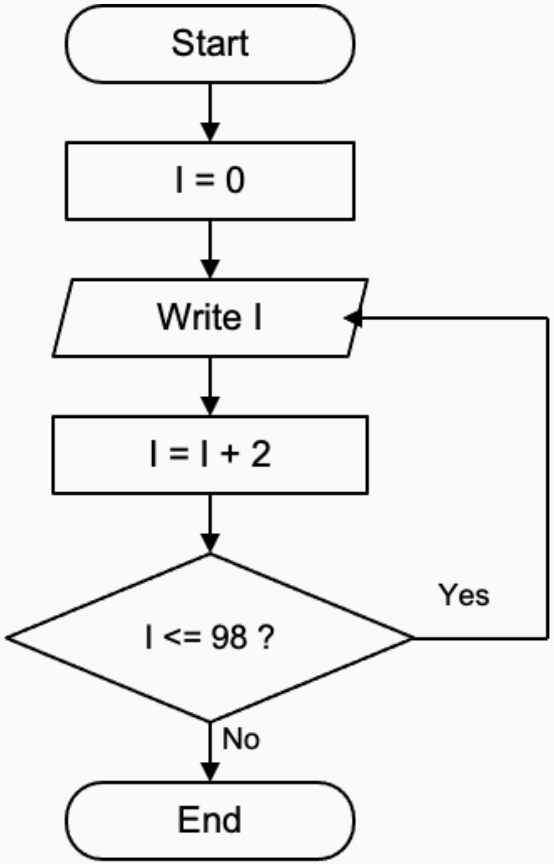
WHILE: wait for correct password

```
READ pwd
WHILE pwd != 'abc' DO
  PRINT 'Wrong'
  READ pwd
ENDWHILE
```

FOR: print 1 to 5

```
FOR i <- 1 TO 5
  PRINT i
ENDFOR
Infinite loop (bug):
i <- 1
WHILE i <= 5: PRINT i  (no i update)
```

ITERATION: Even numbers from 0 to 98




```
1.  Start
2.  I <- 0
3.  Write I
4.  I <- I + 2
5.  If (I <= 98) then go to 3
6.  End
```

- I is the loop variable.
- It starts at 0, is printed, then increases by 2.
- The check at step 5 sends control back (the back arrow) until I passes 98, so the loop is finite.

ITERATION: more examples

Odd numbers up to n

```
READ n
I <- 1
REPEAT
  WRITE I
  I <- I + 2
UNTIL I > n
```

Series S = 1/2 + 1/4 + ... + 1/n

```
READ n
I <- 2
S <- 0
WHILE I <= n
  S <- S + 1/I
  I <- I + 2
ENDWHILE
WRITE S
```

FUNCTIONS, PROCEDURES AND PARAMETERS

Procedure: a named block of steps that does a task. It does not return a value.

Function: a named block that computes something and returns a value to the caller.

Parameter: a placeholder name in the definition, e.g. r in area(r).

Why functions and procedures ?

The problem: the same steps are often needed many times. Finding the area of a circle for 3 different radii means writing the same formula 3 times.

Repeating makes programs long, and a mistake has to be fixed in every copy.

The solution: give the steps a name once (define it), then use the name whenever needed (call it).

PROCEDURE

- It is a named block of steps that performs a task.
- It does not give a value back to the caller but may print an output.
- **Defined once** between PROCEDURE and ENDPROCEDURE, and run with CALL.
- **Typical jobs:** print something, draw something, store something.
- **Used as a statement on its own.** It cannot be placed inside a calculation.

Definitions

```
PROCEDURE showLine()
  PRINT '--------'
ENDPROCEDURE

PROCEDURE greet(name)
  PRINT 'Hello', name
ENDPROCEDURE

PROCEDURE printTable(n)
  FOR i <- 1 TO 5
    PRINT n, 'x', i, '=', n * i
  ENDFOR
ENDPROCEDURE
```

Main program

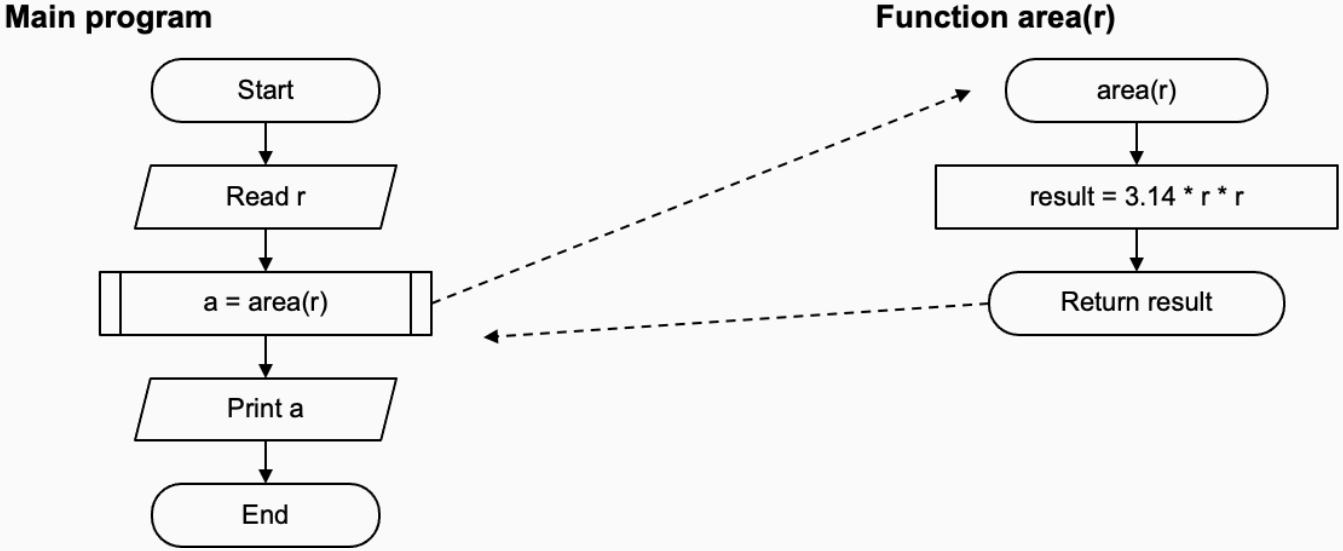
```
MAIN
  CALL showLine()
  CALL greet('Asha')
  CALL printTable(3)
END
```

Output

```
--------
Hello Asha
3 x 1 = 3
3 x 2 = 6
3 x 3 = 9
3 x 4 = 12
3 x 5 = 15
```

FUNCTION

- It is a named block that works something out and sends the answer back with RETURN.
- **Used inside expressions:** a <- area(5), or even PRINT maxOf(3, 8) + 1.
- **Stops at RETURN.** Anything written after RETURN in that path does not run.
- **A function can call another function,** e.g. cost <- area(r) * rate.



- The double-bar box is the **Predefined Process symbol:** it means 'go run another flow chart'.
- Control jumps to the function with the argument, runs it, and returns the result to the main program.

PROCEDURE VS FUNCTION

	Procedure	**Function**
Purpose	Does a task (print, draw, store)	Works out a value and gives it back
Returns a value?	No	Yes, with RETURN
How it is used	As a statement: CALL greet('Asha')	Inside an expression: a <- area(5)
Result can be stored?	No	Yes
Examples	greet(name), printTable(n)	area(r), maxOf(a, b), isEven(n)

NOTE : A procedure DOES something, a function GIVES something back. **Both can have parameters,** and both can contain sequence, selection and iteration.