PROBLEM SOLVING **USING** PYTHON

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.

1.1.Algorithms

What is algorithm?

An algorithm is a collection of **well-defined, unambiguous and effectively computable instructions** ,if execute it will return the proper output.

well-defined- The instructions given in an algorithm should be simple and defined well.

**Unambiguous-** The instructions should be clear,there should not be ambiguity .

effectively computable- The instructions should be written step by step ,which helps computer to understand the control flow .

We often use algorithm in our day-to-day life,but when and how?

- **Our cooking receipe**
- **Our daily routine as a student**
- **When we buy something**
- **When we go outing**
- **Our class routine**

How it helps?

To understand the usage of algorithm,look over the following conversation.
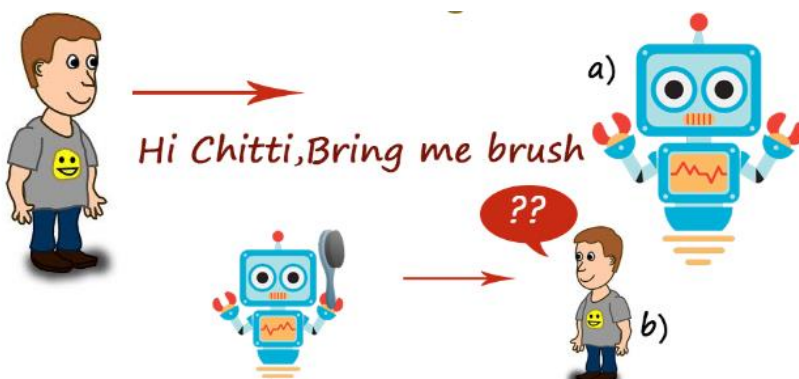
Lets discuss about the conversation,tom wants brush his teeth,so he asks chitti to bring brush,what happens chitti returns cleaning brush.

Why this was happened ?

Because the statement given by tom was **not** well defined and it is **ambiguous** statement so chitti get confused and bring some brush to Tom.This is what happen if the user gives ambiguity statement to the computer.Therefore an algorithm should be **simple and well defined**.

How an algorithm should be?

It should be in simple English ,what a programmer wants to say.It has a **start,a middle and an end**. Probably an algorithm should have,

---

Start
1.In the middle it should have set of tasks that computer wants to do and it should be in simple English and clear.
2.To avoid ambiguous should give no for each step.
**Stop**

Lets look over the simple example,
The following algorithm helps the computer to validate user's email address.

Start
Create a variable to get the user's email address
clear the variable,incase its not empty.
Ask the user for an email address.
Store the response in the variable.
Check the stored response to see if it is a valid email address
Not valid?Go back
Stop
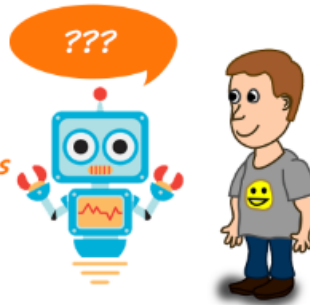

Lets see how it works?
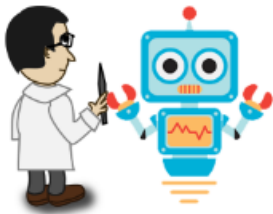
Dr·Paul feed the above algorithm to chitti and gonna test how it will work·

Chitti asked Tom to enter email address·Tom entered ,but chitti gets confused and donot know which process to be done next??

Why this Happened?

This was happened because the instructions given in an algorithm does not have numbering for each step.So Chitti gets confused which step have to do.To avoid this ambiguity ,we should number each step while writing an algorithm.

So lets rewrite the algorithm ....

Step1: Start
Step2: Create a variable to get the user's email address
Step3: Clear the variable,incase its not empty.
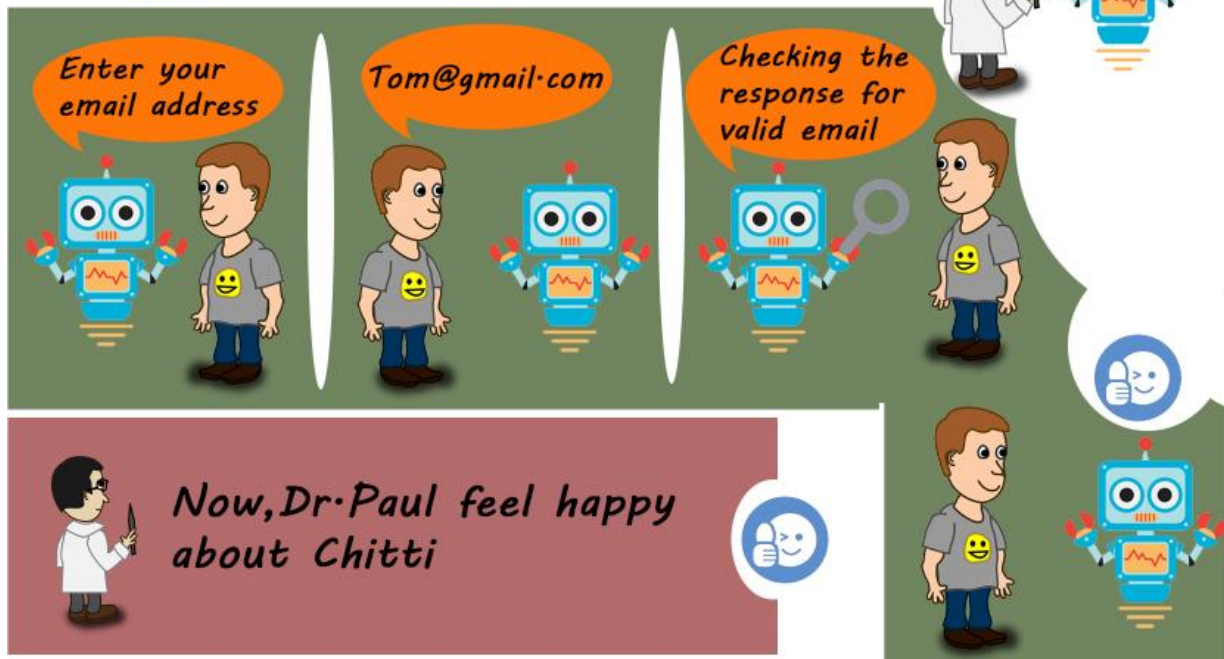**Step4:** Ask the user for an email address.
Step5: Store the response in the variable.
Step6: Check the stored response to see if it is a valid email address
Step7: Not valid?Go back
Step8: Stop

Algorithm Template Format

The real power of using a template to describe each algorithm is that you can quickly compare and contrast different algorithms and identify commonalities in seemingly different algorithms. Each algorithmis presented using a fixed set of sections that conform to this template.

We may omit a section if it adds no value to the algorithm description or add sections as needed to illuminate a particular point.

Name
A descriptive name for the algorithm. We use this name to communicate concisely the algorithm to others. For example, if we talk about using a Sequential Search , it conveys exactly what type of search algorithm we are talking about. The name of each algorithm is always shown in **Bold Font**
.

Input/Output
Describes the expected format of input data to the algorithm and the resulting values computed.

Context
A description of a problem that illustrates when an algorithm is useful and when it will perform at its best. A description of the properties of the problem/solution that must be addressed and maintained for a successful implementation. They are the things that would cause you to choose this algorithm specifically.

Solution
The algorithm description using real working code with documentation. All code solutions can be found in the associated code repository.

Analysis
A synopsis of the analysis of the algorithm, including performance data and information to help you understand the behavior of the algorithm. Although the analysis section is not meant to "prove" the described performance of an algorithm, you should be able to under-stand why the algorithm behaves as it does. We will provide references to actual texts that present the appropriate lemmas and proofs to explain why the algorithms behave as described

1.2.Building Blocks of Algorithms

It has been proven that any algorithm can be constructed from just three basic building blocks. These three building blocks are **Sequence, Selection, and Iteration(Repetition)**.

Sequence

This describes a sequence of actions that a program carries out one after another, unconditionally.
Execute a list of statements **in order**.

Consider an example,

Algorithm for Baking Bread

Step1: Add flour.
Step 2: Add salt.
Step 3: Add yeast.
Step 4: Mix.
Step 5: Add water.
Step 6: Knead.
Step 7: Let rise.
Step 8: Bake.

Bread has been baked successfully.

Algorithm for Addition of two numbers:

Step1: Start
Step 2: Get two numbers as input and store it in to a and b
Step 3: Add the number a & b and store it into c
Step 4: Print c
Step 5: Stop.

Selection

Selection is the program construct that allows a program to choose between different actions.Choose at most one action from several alternative conditions.

Algorithm for path choser

Step 1: Check for the destination located from current position.
Step 2:If it is located in right then choose right way
Step 3:If it is located in left then choose left way.
Step 4:Else comeback and search for new way.

Path has been chosen successfully.

Algorithm to find biggest among 2 nos:

Step1: Start
Step 2: Get two numbers as input and store it in to a and b
Step 3: If a is greater than b then
Step 4: Print a is big
Step 5: else
Step 6: Print b is big
Step 7: Stop

Repetition

Repetition(loop) may be defined as a smaller program the can be executed several times in a main program.Repeat a block of statements while a condition is true.

Algorithm for Washing Dishes

Step1: Stack dishes by sink.
Step 2: Fill sink with hot soapy water.
Step 3: While moreDishes
Step 4: Get dish from counter,Wash dish
Step 5: Put dish in drain rack.
**Step 6: End While**
Step 7: Wipe off counter.
**Step 8: Rinse out sink.**

Algorithm to calculate factorial no:

Step1: Start
Step 2: Read the number num.
Step 3: Initialize i is equal to 1 and fact is equal to 1
Step 4: Repeat step4 through 6 until I is equal to num
Step 5: fact←fact * i
Step 6: i←i+1
Step 7: Print fact
Step 8: Stop

1.3.Algorithm Notations(Expressing Algorithms)

As we know that ,an algorithm is a sequence of finite instructions,often used for **calculation and data processing**.
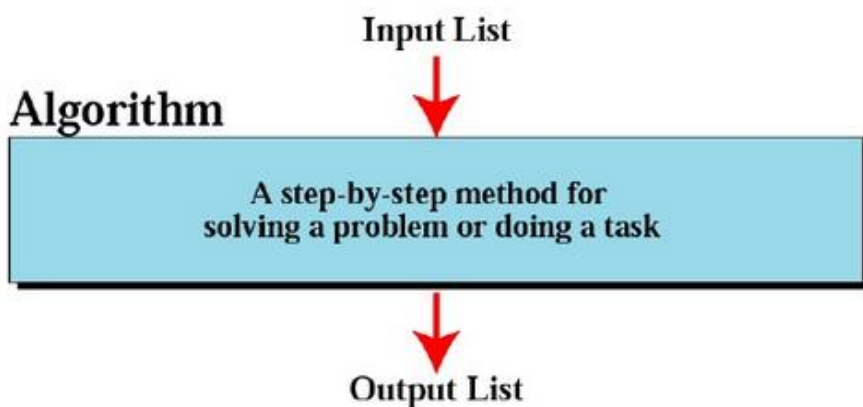
Algorithms can be expressed in many kinds of notation,including
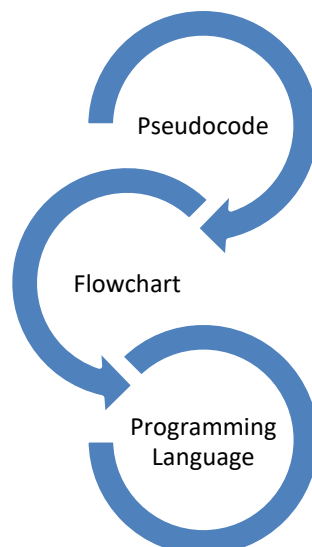
1.3.1.Pseudocode

- **"Pseudo"** means imitation or false and **"code"** refers to the instructions written in a programming language.
- **Pseudocode** is another programming analysis tool that is used for planning a program
- **Pseudocode** is also called Program Design Language(PDL)

Guidelines to write Pseudocode

- Pseudocode is written using structured English.
- Pseudocode should be concise
- Keyword should be in capital letter

Pseudocode is made up of the following logic structure ,

- **Sequential logic**
- **Selection logic**
- **Iteration logic**

Sequence Logic

- It is used to perform instructions in a sequence,that is **one after another**
- Thus,for sequence logic ,pseudocode instructions are written in an order in which they are to be performed.
- The logic flow of pseudocode is from **top to bottom.**

Pseudocode to add two numbers:

```
START
READ a,b
COMPUTE c by adding a &b
PRINT c
STOP
```

Selection Logic

- It is used for making decisions and for selecting the proper path out of two or more alternative paths in program logic.
- It is also known as decision logic.
- Selection logic is depicted as either an **IF..THEN or an IF…THEN..ELSE Structure**.

Pseudocode to add two numbers:

```
START
READ a and b
IF a>b THEN
PRINT "A is big"
ELSE
PRINT "B is big"
ENDIF
STOP
```

Repetition Logic

- It is used to produce loops when one or more instructions may be executed several times depending on some conditions .
- It uses structures called **DO_WHILE,FOR and REPEAT__UNTIL**

Pseudocode to print first 10 natural numbers

```
START
INITIALIZE a←0
WHILE a<10
PRINT a
ENDWHILE
STOP
```

1.3.2.Flowchart

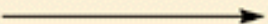A flowchart is a visual representation of the sequence of steps and decision needed to perform a process.
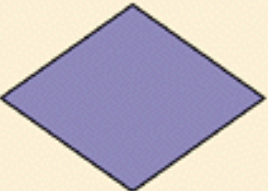
Flowchart for computer programming/algorithms

- As a visual representation of data flow, flowcharts are useful in writing a program or algorithm and explaining it to others or collaborating with them on it.
- You can use a flowchart to spell out the logic behind a program before ever starting to code the automated process.
- It can help to organize big-picture thinking and provide a guide when it comes time to code. More specifically, flowcharts can:

  - Demonstrate the way code is organized.
  - Visualize the execution of code within a program.
  - Show the structure of a website or application.
  - Understand how users navigate a website or program.

Flowchart symbols

Here are some of the common flowchart symbols.

Name	Symbol	Use in flowchart
Oval		Denotes the beginning or end of a program.
Flow line	→	Denotes the direction of logic flow in a program.
Parallelogram		Denotes either an input operation (e.g., INPUT) or an output operation (e.g, PRINT).
Rectangle		Denotes a process to be carried out (e.g., an addition).
Diamond		Denotes a decision (or branch) to be made. The program should continue along one of two routes (e.g., IF/THEN/ELSE).

Flowchart is made up of the following logic structure ,

- **Sequential logic**
- **Selection logic**
- **Iteration logic**

Sequence Logic

In a computer program or an algorithm, sequence involves simple steps which are to be executed one after the other. The steps are executed in the same order in which they are written.

Below is an example set of instructions to add two numbers and display the answer.
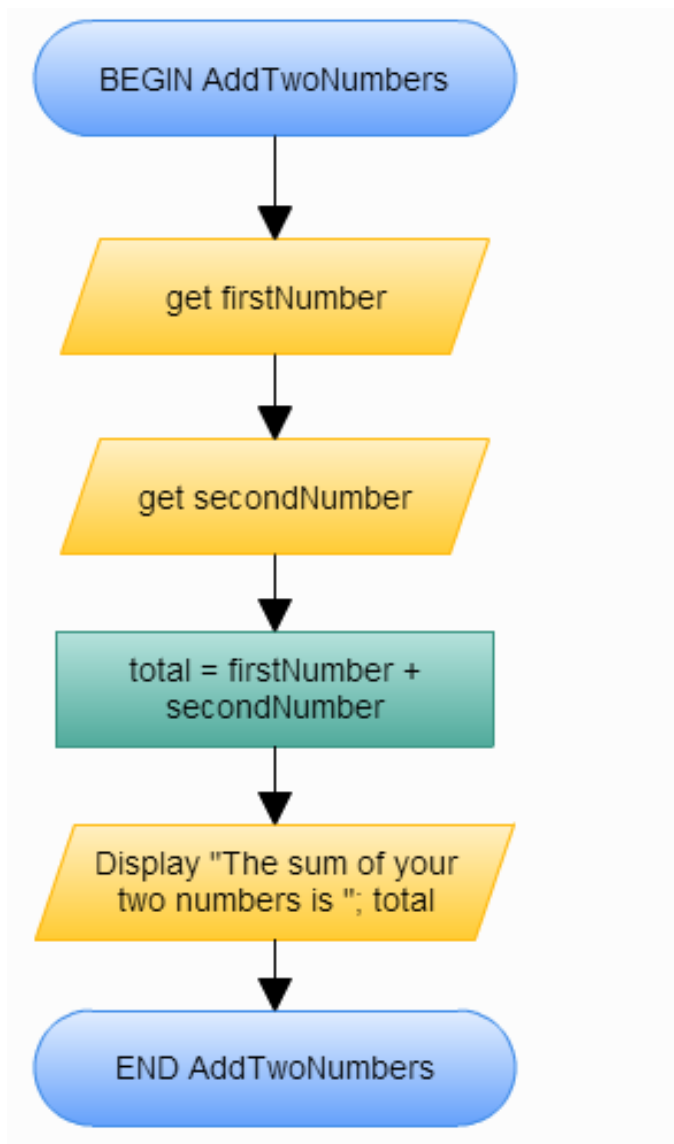
Selection Logic

Selection is used in a computer program or algorithm to determine which particular step or set of steps is to be executed. This is also referred to as a 'decision'.

 A selection statement can be used to choose a specific path dependent on a condition.

There are two types of selection:

- **binary selection (two possible pathways)**
- **multi-way selection (many possible pathways)**

Following is the example flowchart to find biggest among two numbers
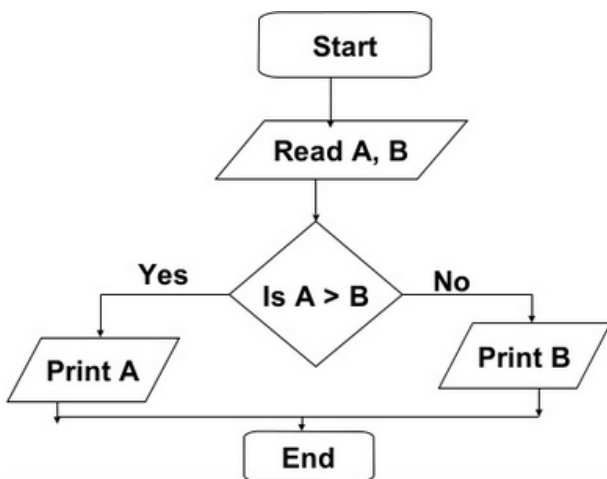
Repetition Logic

Repetition allows for a portion of an algorithm or computer program to be executed any number of times dependent on some condition being met.

An occurrence of repetition is usually known as a **loop**.

The termination condition can be checked or tested at the **beginning** or **end** of the loop, and is known as a **pre-test** or **post-test**, respectively.

Flowchart to find factorial of given no
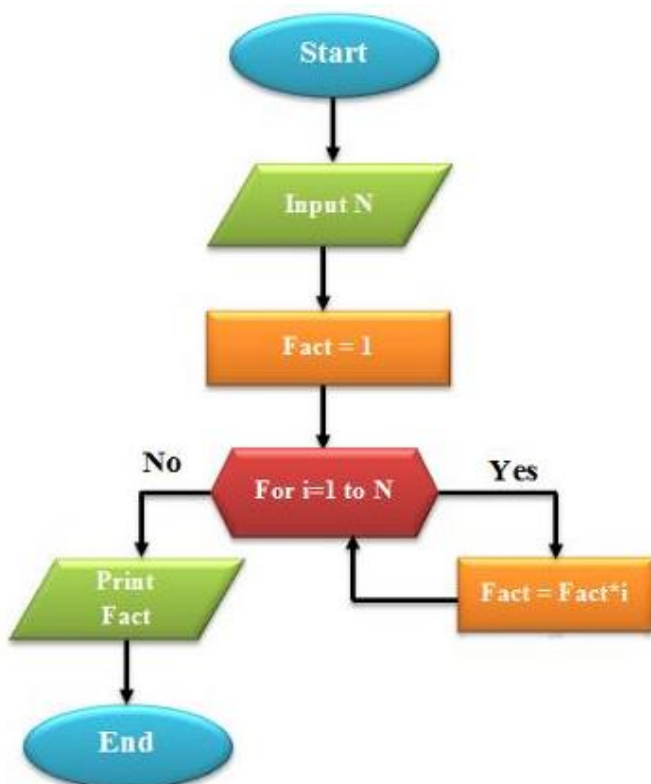
pre-test loop
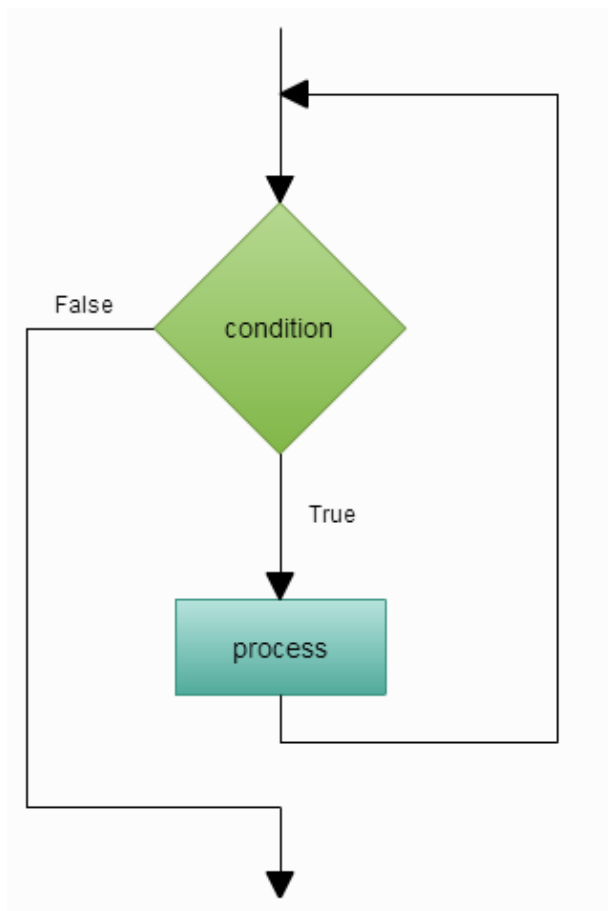
post-test
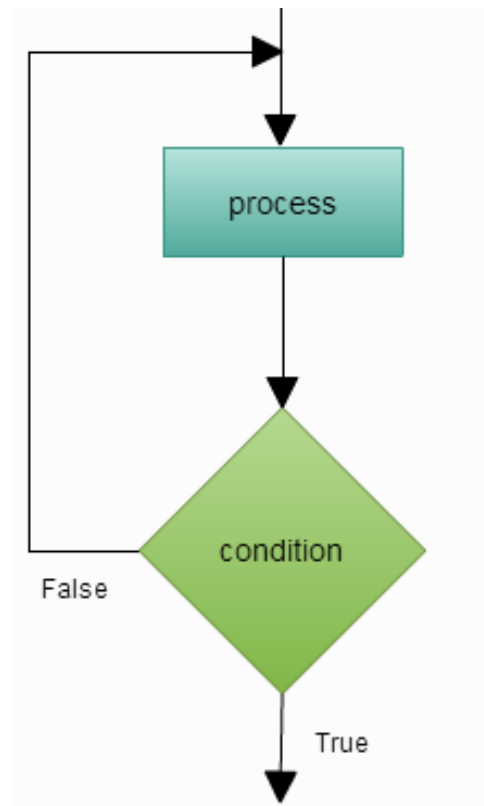
1.3.3.Representation of Algorithm using Programming Language

- Algorithms describe the solution to a problem in terms of the data needed to represent the problem instance and the set of steps necessary to produce the intended result.
- Programming languages must provide a notational way to represent both the process and the data.
- To this end, languages provide control constructs and data types.

Programming is the process of taking an algorithm and encoding it into a notation, a programming language, so that it can be executed by a computer.

Although many programming languages and many different types of computers exist, the important first step is the need to have the solution.

Without an algorithm there can be no program.

- Control constructs allow algorithmic steps to be represented in a convenient yet unambiguous way.
- At a minimum, algorithms require constructs that perform sequential processing, selection for decision-making, and iteration for repetitive control.
- As long as the language provides these basic statements, it can be used for algorithm representation.

Simply we can say programming as like below

Programming is implementing the already solved problem (algorithm) in a specific computer language where syntax and other relevant parameters are different, based on different programming languages.

Computer Languages

Low Level Language
(Machine Language)

Use 1's & 0's to create instructions

Ex: Binary Language

Middle Level Language
(Assembly Language)

Use mnemonics to create instructions

Assembly Language

High Level Language

Similar to human langugae

COBOL, FORTRAN, BASIC C, C++, JAVA
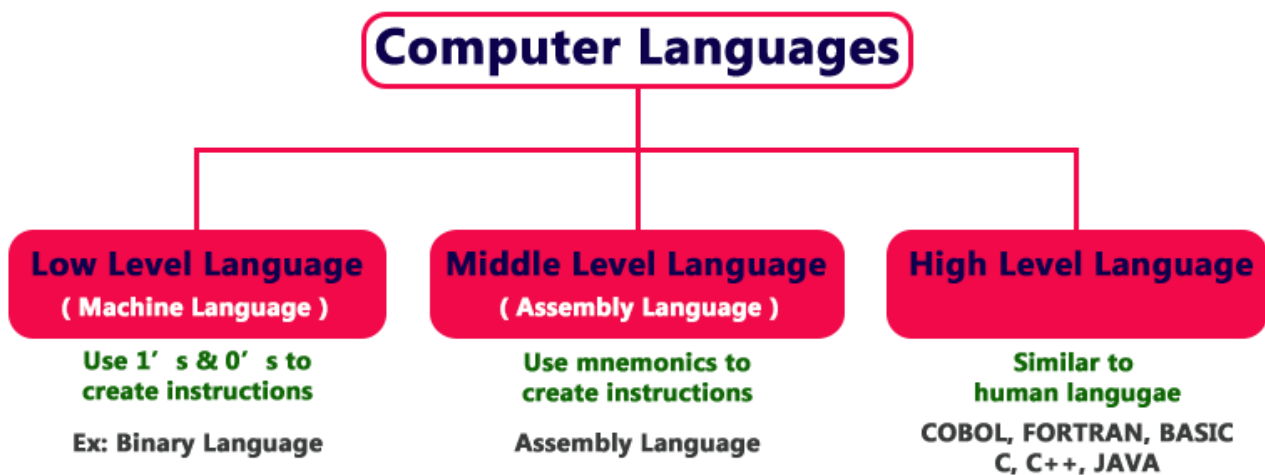
Low level Language(Machine level Language)

A *low-level language* is a programming *language* that deals with a computer's hardware components and constraints.

In simple we can say that ,low level language can only be understand by computer processor and components.

Binary and assembly languages are examples for low level language.

Middle level Language(Intermediate Language)

Medium-level language serves as the bridge between the raw hardware and programming layer of a computer system.

Medium-level language is also known as intermediate programming language and pseudo language.

C intermediate language and Java byte code are some examples of medium-level language.

High level Language (Human understandable Language)

A high-level language is any programming language that enables development of a program in a much more user-friendly programming context.

High-level languages are designed to be used by the human operator or the programmer.

They are referred to as "closer to humans." In other words, their programming style and context is easier to learn and implement than low-level languages

BASIC, C/C++ and Java are popular examples of high-level languages.
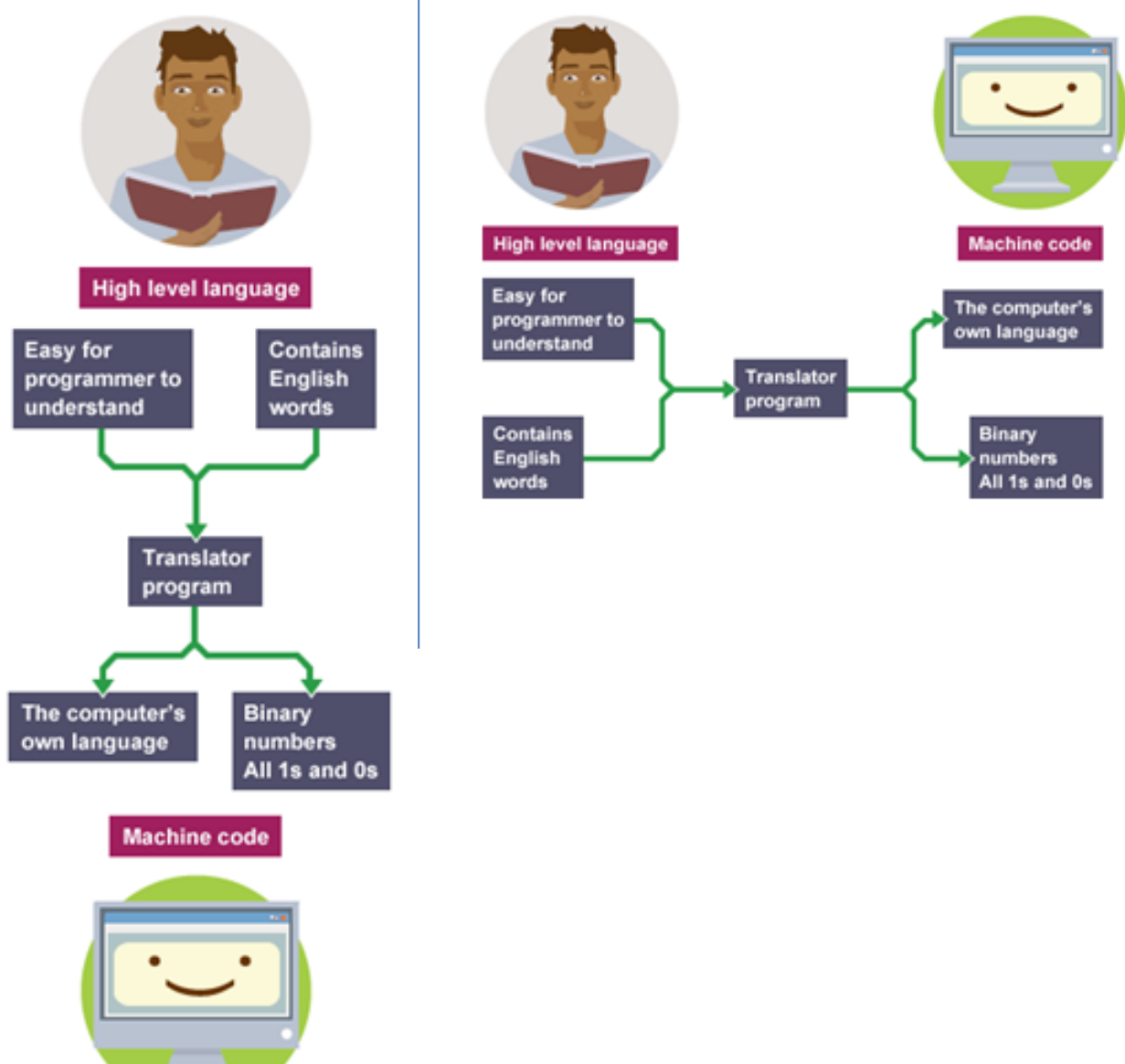
1.4. Algorithmic Problem Solving

"Algorithmic-problem solving"; this means solving problems that require the formulation of an algorithm for their solution.

The formulation of algorithms has always been an important element of problem-solving .

Why we need to go for algorithm to solve problem?

- A computer is a tool that can be used to implement a plan for solving a problem.
- A computer program is a set of instructions for a computer. These instructions describe the steps that the computer must follow to implement a plan.
- An algorithm is a plan for solving a problem.
- A person must design an algorithm.
- A person must translate an algorithm into a computer program.

An algorithmic Development Process

Every problem solution starts with a plan. That plan is called an algorithm.

An algorithm is a plan for solving a problem.

There are many ways to write an algorithm.

- **Some are very informal.**
- **some are quite formal .**
- **mathematical in nature.**
- **some are quite graphical.**

Once we have an algorithm, we can translate it into a computer program in some programming language. Our algorithm development process consists of five major steps.

Step 1: Obtain a description of the problem.

Step 2: Analyze the problem.

Step 3: Develop a high-level algorithm.

Step 4: Refine the algorithm by adding more detail.

Step 5: Review the algorithm.

Step 1: Obtain a description of the problem.

This step is much more difficult than it appears. In the following discussion,

> ▸ **The word *client* refers to someone who wants to find a solution to a problem,**
> ▸ **The word *developer* refers to someone who finds a way to solve the problem.**
> ▸ **The developer must create an algorithm that will solve the client's problem.**

Step 2: Analyze the problem.

The purpose of this step is to determine both the starting and ending points for solving the problem. This process is analogous to a mathematician determining what is given and what must be proven. A good problem description makes it easier to perform this step.

When determining the starting point, we should start by seeking answers to the following questions:

- What data are available?
- Where is that data?
- What formulas pertain to the problem?
- What rules exist for working with the data?
- What relationships exist among the data values?

When determining the ending point, we need to describe the characteristics of a solution. In other words, how will we know when we're done? Asking the following questions often helps to determine the ending point.

- What new facts will we have?
- What items will have changed?
- What changes will have been made to those items?
- What things will no longer exist?

Step 3: Develop a high-level algorithm.

An algorithm is a plan for solving a problem, but plans come in several levels of detail. It's usually better to start with a high-level algorithm that includes the major part of a solution, but leaves the details until later. We can use an everyday example to demonstrate a high-level algorithm.

Problem: I need a send a birthday card to my brother, Mark.

Analysis: I don't have a card. I prefer to buy a card rather than make one myself.

High-level algorithm:

Go to a store that sells greeting cards
Select a card
Purchase a card
Mail the card

This algorithm is satisfactory for daily use, but it lacks details that would have to be added were a computer to carry out the solution. These details include answers to questions such as the following.

- "Which store will I visit?"
- "How will I get there: walk, drive, ride my bicycle, take the bus?"
- "What kind of card does Mark like: humorous, sentimental, risqué?"

These kinds of details are considered in the next step of our process.

Step 4: Refine the algorithm by adding more detail.

A high-level algorithm shows the major steps that need to be followed to solve a problem.

Now we need to add details to these steps, but how much detail should we add? Unfortunately, the answer to this question depends on the situation.

We have to consider who (or what) is going to implement the algorithm and how much that person (or thing) already knows how to do.

If someone is going to purchase Mark's birthday card on my behalf, my instructions have to be adapted to whether or not that person is familiar with the stores in the community and how well the purchaser known my brother's taste in greeting cards.

Most of our examples will move from a high-level to a detailed algorithm in a single step, but this is not always reasonable.

For larger, more complex problems, it is common to go through this process several times, developing intermediate level algorithms as we go.

Each time, we add more detail to the previous algorithm, stopping when we see no benefit to further refinement.

This technique of gradually working from a high-level to a detailed algorithm is often called stepwise refinement.

Stepwise refinement is a process for developing a detailed algorithm by gradually adding detail to a high-level algorithm.

Step 5: Review the algorithm.

The final step is to review the algorithm. What are we looking for? First, we need to work through the algorithm step by step to determine whether or not it will solve the original problem. Once we are satisfied that the algorithm does provide a solution to the problem, we start to look for other things. The following questions are typical of ones that should be asked whenever we review an algorithm. Asking these questions and seeking their answers is a good way to develop skills that can be applied to the next problem.

1. Does this algorithm solve a **very specific problem** or does it solve a **more general problem**?
2. If it solves a very specific problem, should it be generalized?
3. Can this algorithm be **simplified**?
4. Is this solution **similar** to the solution to another problem? How are they alike? How are they different?

1.5. Simple Strategies for Developing algorithms

Remember the strategies what we have discussed in previous chapter,Lets look over once again,

Step 1: Obtain a description of the problem.

Step 2: Analyze the problem.

Step 3: Develop a high-level algorithm.

Step 4: Refine the algorithm by adding more detail.

Step 5: Review the algorithm.

An Example Algorithm

Let's look at a very simple algorithm called **find_max().**

Problem: Given a list of positive numbers, return the largest number on the list.

Inputs: A list L of positive numbers. This list must contain at least one number. (Asking for the largest number in a list of no numbers is not a meaningful question.)

Outputs: A number n, which will be the largest number of the list.

Algorithm:

1. Start
2. Set max to 0.
3. For each number x in the list L, compare it to max. If x is larger, set max to x.
4. max is now set to the largest number in the list.
5. Stop

Lets see a simple implementation of above algorithm in python programming language. Just look over the implementation ,anyway we will discuss a lot about python language in forecoming chapters.

An implementation in Python:

```
def find_max (L):
    max = 0
    for x in L:
        if x > max:
            max = x
    return max
```

Does this meet the criteria for being an algorithm?

- ***Is it unambiguous?***

  Yes. Each step of the algorithm consists of primitive operations, and translating each step into Python code is very easy.

- ***Does it have defined inputs and outputs?***

  Yes.

- ***Is it guaranteed to terminate?***

  Yes. The list L is of finite length, so after looking at every element of the list the algorithm will stop.

- ***Does it produce the correct result?***

  Yes. In a formal setting you would provide a careful proof of correctness. In the next section I'll sketch a proof for an alternative solution to this problem.

A Recursive Version of find_max()

There can be many different algorithms for solving the same problem. Here's an alternative algorithm for find_max():

Algorithm:

```
1.  Start
2.  If L is of length 1, return the first item of L.
3.  Set v1 to the first item of L.
4.  Set v2 to the output of performing find_max() on the rest of L.
5.  If v1 is larger than v2, return v1. Otherwise, return v2.
6.  Stop
```

Implementation:

```
def find_max (L):
    if len(L) == 1:
        return L[0]
    v1 = L[0]
    v2 = find_max(L[1:])
    if v1 > v2:
        return v1
    else:
        return v2
```

Let's ask our questions again.

- *Is it unambiguous?*

   Yes. Each step is simple and easily translated into Python.

- *Does it have defined inputs and outputs?*

   Yes.

- *Is it guaranteed to terminate?* Yes. The algorithm obviously terminates if L is of length 1. If L has more than one element, find_max() is called with a list that's one element shorter and the result is used in a computation.

   Does the nested call to find_max() always terminate? Yes. Each time, find_max() is called with a list that's shorter by one element, so eventually the list will be of length 1 and the nested calls will end.

1.6.Example algorithms(Illustrative Problems)

1.6.1 Find minimum in a list

Problem: Given a list of positive numbers, return the smallest number on the list.

Inputs: A list L of positive numbers. This list must contain at least one number. (Asking for the smallest number in a list of no numbers is not a meaningful question.)

Outputs: A number n, which will be the smallest number of the list.

Algorithm:

1. Start
2. Get positive numbers from user and add it in to the List L
3. Set min to L[0].
4. For each number x in the list L, compare it to min. If x is smaller, set min to x.
5. min is now set to the minimum number in the list.
6. Stop

Lets represent the above algorithm in python programming language.

```
b = [1,2,3,4,5]

def minimum(x):
    mini = x[0]
    for i in x[0:]:
        if i < mini:
            mini = i
        else:
            mini = x[0]
    return (mini)

print minimum(b)
```

1.6.2. Insert a card in a list of sorted cards

1. Start
2. Ask for value to insert
3. Find the correct position to insert, If position cannot be found ,then insert at the end.
4. Move each element from the backup to one position, until you get position to insert.
5. Insert a value into the required position
6. Increase array counter
7. Stop

1.6.3.Guess an integer number in a range

Let's play a little game to give you an idea of how different algorithms for the same problem can have wildly different efficiencies.

The computer is going to randomly select an integer from 1 to 16. We have to guess the number by making guesses until you find the number that the computer chose.

Algorithm:

1. Start
2. Generate a random number from 1 to 20 and store it into the variable number.
3. Ask the user to guess number between 1 and 20 and store it into guess for six chances.
4. Check if guess is equal to number
5. If guess is greater than number print ,the number you guessed is greater
6. If guess is lesser then print,the number you guessed is lesser.
7. If guess is equal to number then,print you guessed is right.
8. If guess is not equal and chance is greater than six print you fail and stop the execution
9. Stop

Python Implementation of above algorithm

```
guessesTaken = 0
number = random.randint(1, 20)
while guessesTaken < 6:
        print('Take a guess.')
        guess = input()
        guess = int(guess)
        guessesTaken = guessesTaken + 1
if guess < number:
        print('Your guess is too low.')
if guess > number:
    print('Your guess is too high.')
if guess == number:
    break
if guess == number:
    guessesTaken = str(guessesTaken)
    print('Good job, ! You guessed my number in ' + guessesTaken + ' guesses!')
if guess != number:
    number = str(number)
    print('Nope. The number I was thinking of was ' + number)
```
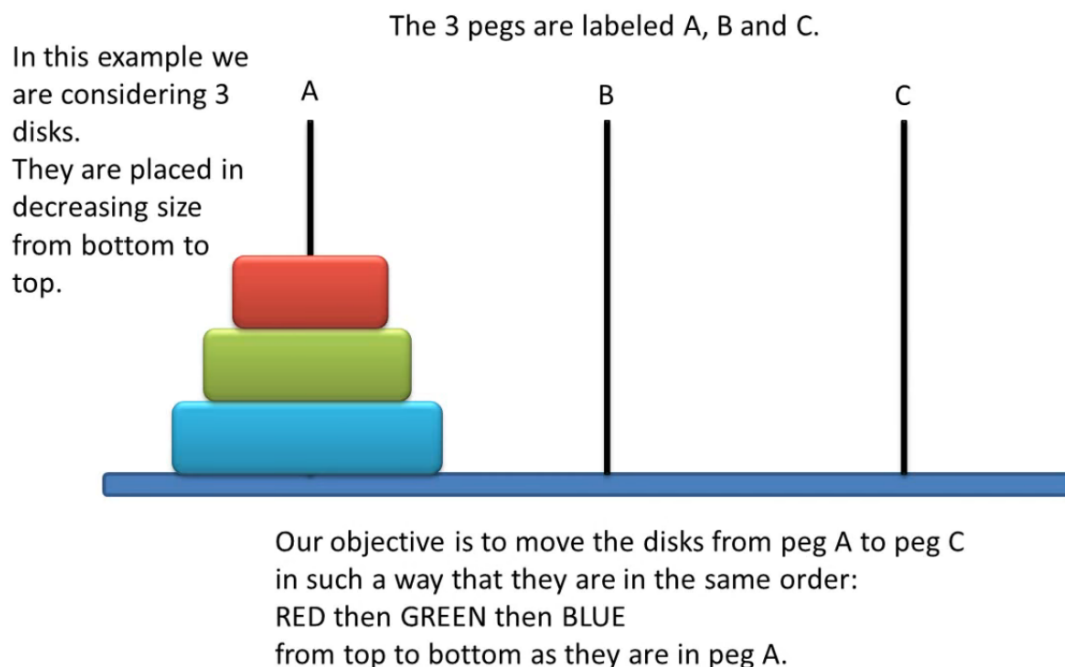
1.6.4.Tower of Hanoi

Tower of Hanoi is a very famous game.

> **In this game there are 3 rods(pegs) and N number of disks placed one over another in decreasing size.**
>
> **The objective of the game is to move the disks one by one from the first rod to last rod.**
>
> **And there is only one condition, we cannot place a bigger disk on top of a smaller disk.**

The 3 pegs are labeled A, B and C.

In this example we are considering 3 disks.
They are placed in decreasing size from bottom to top.

Our objective is to move the disks from peg A to peg C
in such a way that they are in the same order:
RED then GREEN then BLUE
from top to bottom as they are in peg A.

1. Start
2. Move disk1 from pegA to pegC
3. Move disk2 from pegA to pegB
4. Move disk3 from pegC to pegB
5. Move disk1 from pegA to pegC
6. Move disk1 from pegB to pegA
7. Move disk2 from pegB to pegC
8. Move disk1 from pegA to peg C
9. Stop

Python Implementation of above algorithm

```
def moveTower(height,fromPole, toPole, withPole):

    if height >= 1:

        moveTower(height-1,fromPole,withPole,toPole)

        moveDisk(fromPole,toPole)

        moveTower(height-1,withPole,toPole,fromPole)

def moveDisk(fp,tp):

    print("moving disk from",fp,"to",tp)

moveTower(3,"A","B","C")
```

Note:

Don't worry about python implementation, we are going to learn python in fore coming chapters.All the sources are taken from internet.