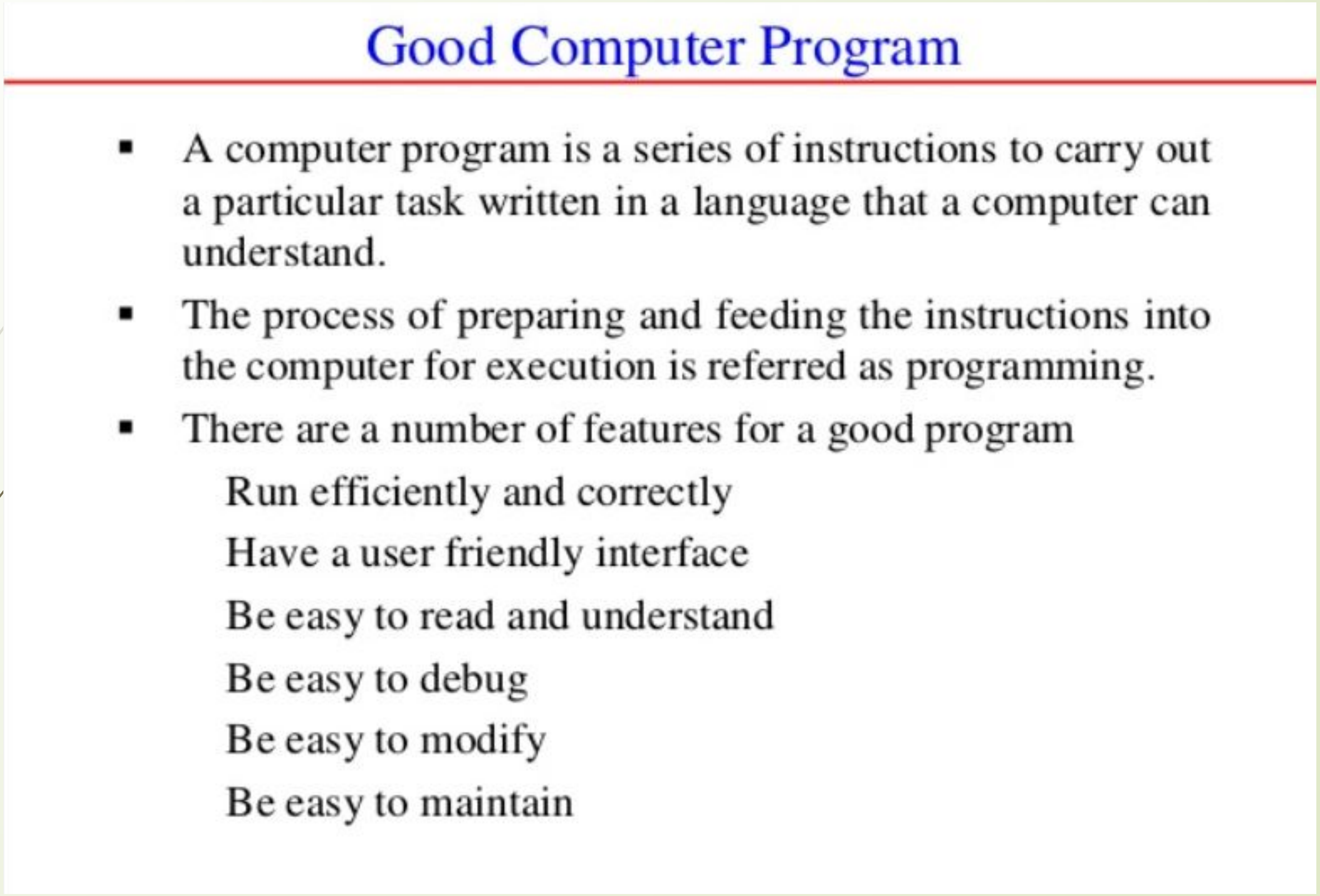
Algorithms



Good Computer Program

- A computer program is a series of instructions to carry out a particular task written in a language that a computer can understand.
- The process of preparing and feeding the instructions into the computer for execution is referred as programming.
- There are a number of features for a good program

  Run efficiently and correctly

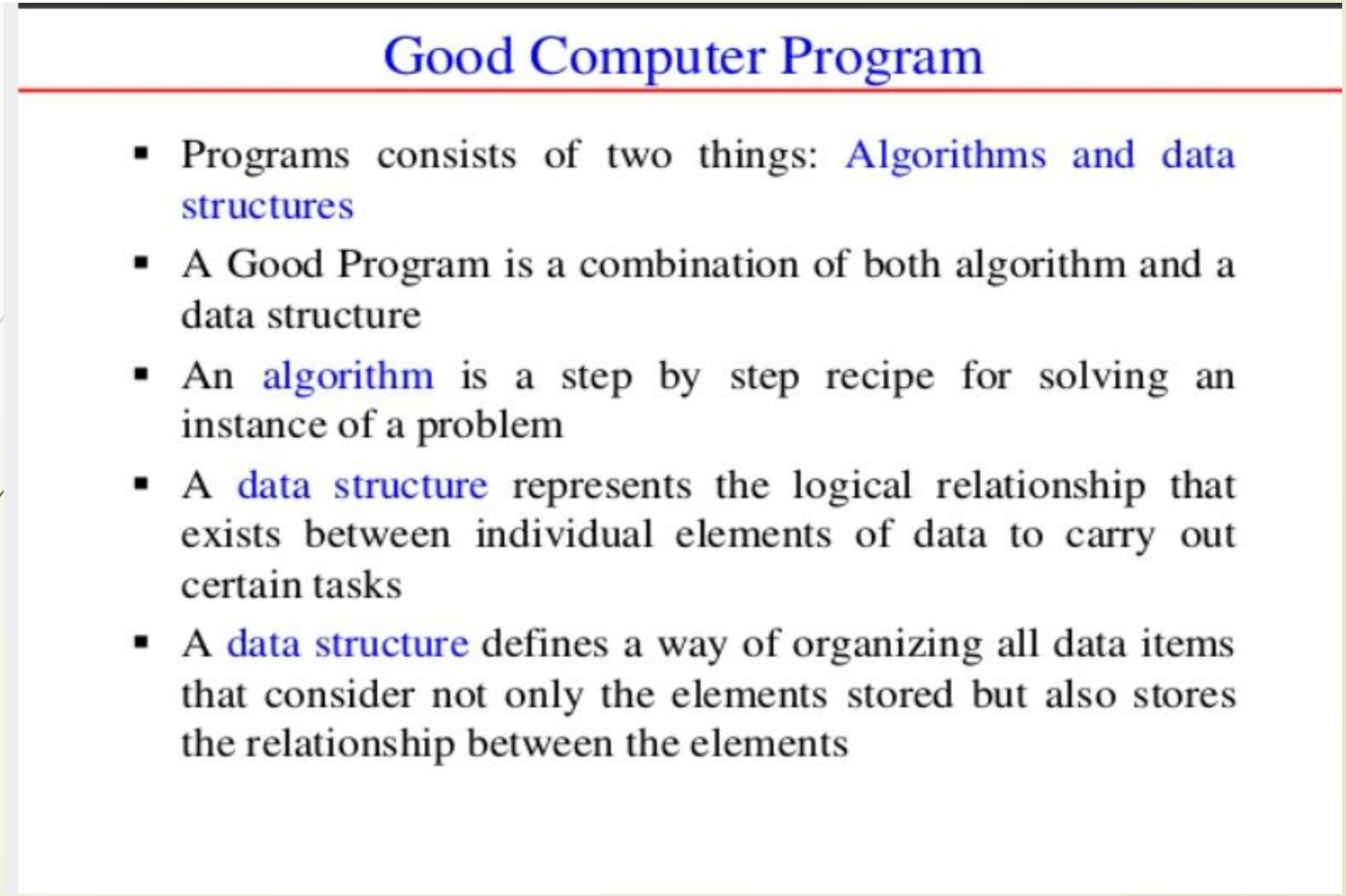
  Have a user friendly interface

  Be easy to read and understand

  Be easy to debug

  Be easy to modify

  Be easy to maintain

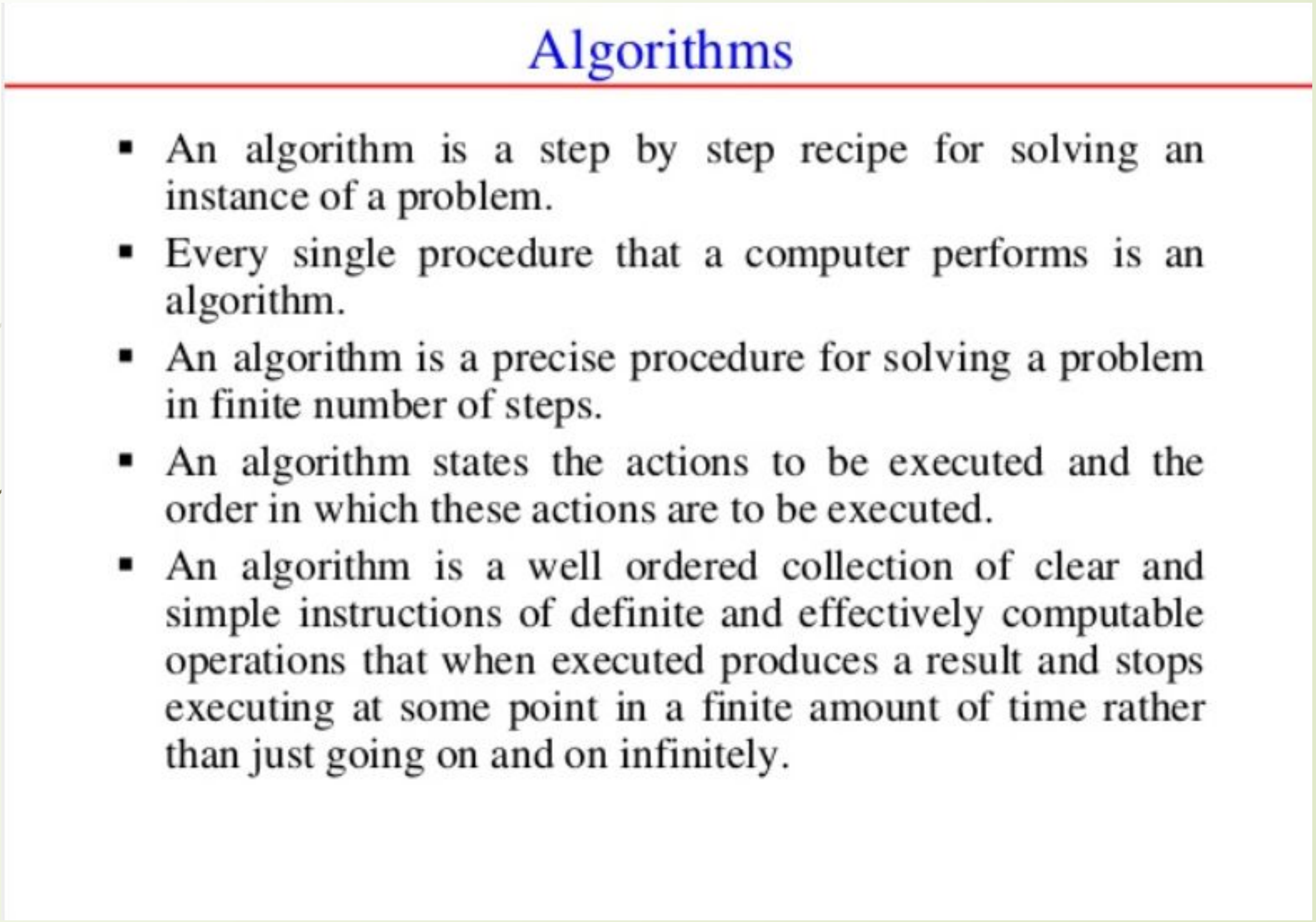


Good Computer Program

- Programs consists of two things: Algorithms and data structures
- A Good Program is a combination of both algorithm and a data structure
- An algorithm is a step by step recipe for solving an instance of a problem
- A data structure represents the logical relationship that exists between individual elements of data to carry out certain tasks
- A data structure defines a way of organizing all data items that consider not only the elements stored but also stores the relationship between the elements

Algorithms

- ? Basically, an algorithm is a description of the various steps involved in finding the solution of a particular problem.
- ? In general, a problem can have a direct solution or an iterative solution.
- ? In the direct solution, there are fixed steps, which are not repeating, that we apply to arrive at the solution of the problem.
- ? However, in iterative solution, there are certain steps within the algorithm that may have to be repeated (iterated).
- ? Algorithm can be defined as a finite set of steps defining the correct solution of a particular problem.



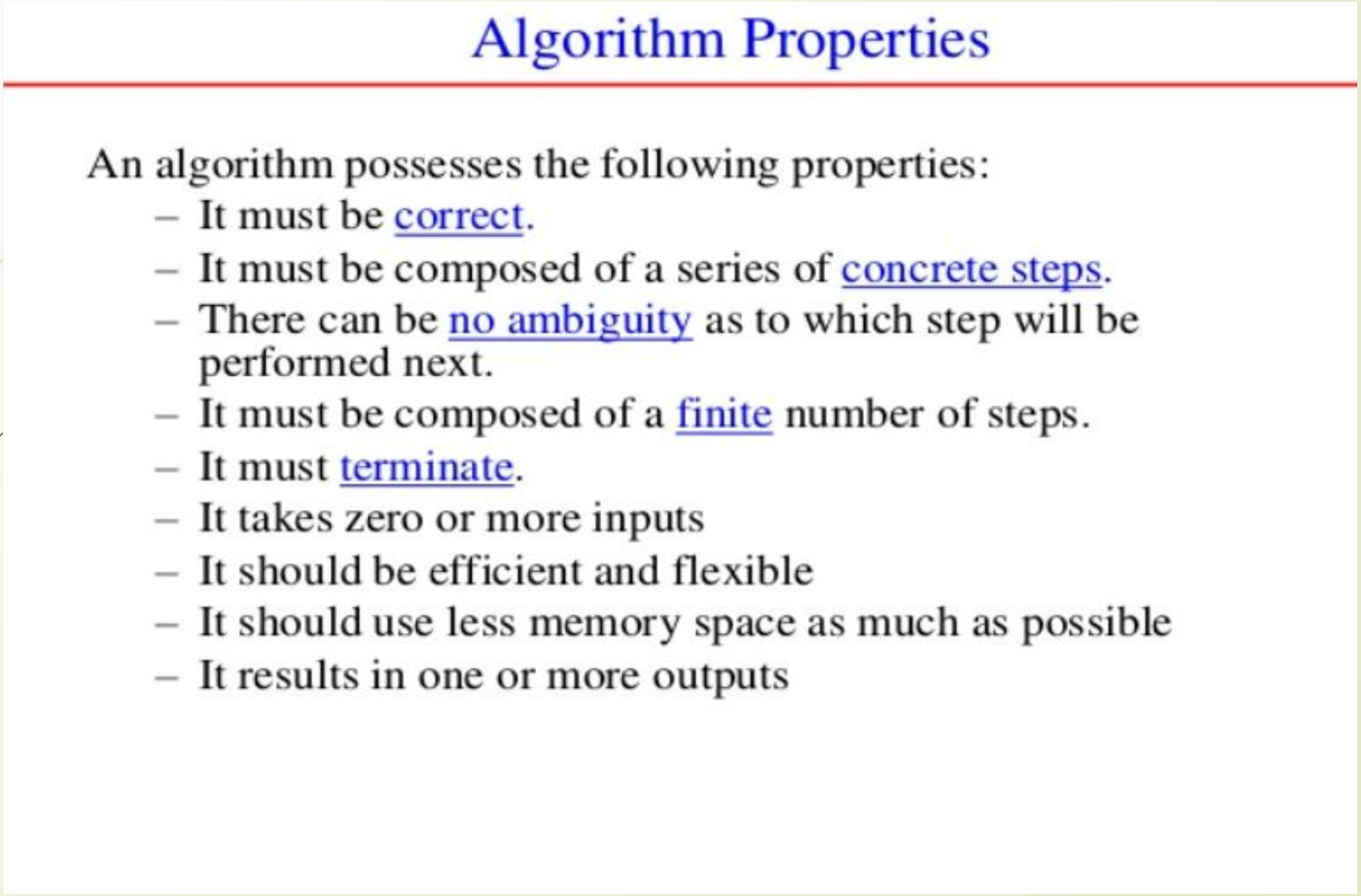
Algorithms

- An algorithm is a step by step recipe for solving an instance of a problem.
- Every single procedure that a computer performs is an algorithm.
- An algorithm is a precise procedure for solving a problem in finite number of steps.
- An algorithm states the actions to be executed and the order in which these actions are to be executed.
- An algorithm is a well ordered collection of clear and simple instructions of definite and effectively computable operations that when executed produces a result and stops executing at some point in a finite amount of time rather than just going on and on infinitely.

Characteristics of an Algorithm

? Not all procedures can be called an algorithm. An algorithm should have the following characteristics −

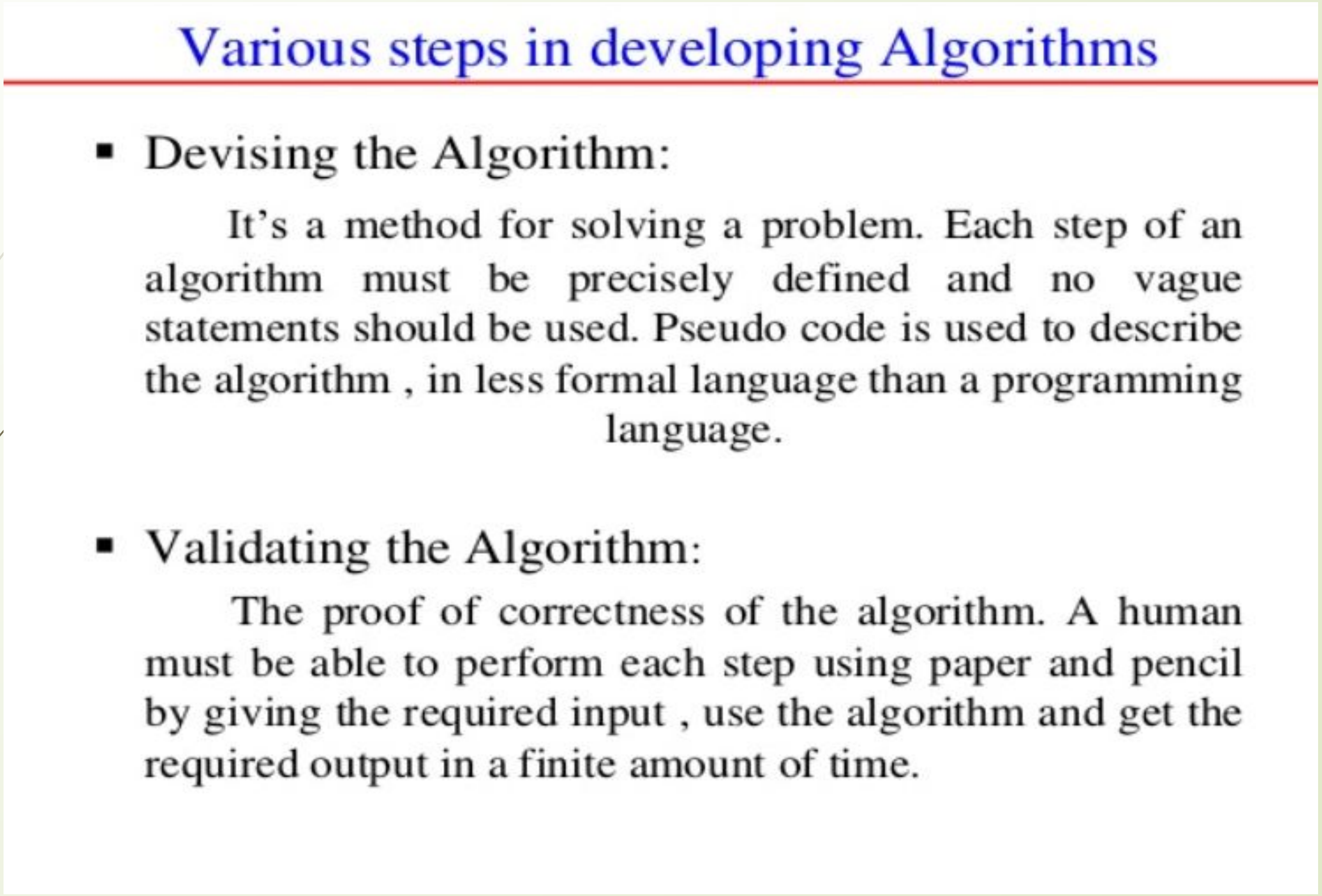
- **Unambiguous** − Algorithm should be clear and unambiguous. Each of its steps (or phases), and their inputs/outputs should be clear and must lead to only one meaning.
- **Input** − An algorithm should have 0 or more well-defined inputs.
- **Output** − An algorithm should have 1 or more well-defined outputs, and should match the desired output.
- **Finiteness** − Algorithms must terminate after a finite number of steps.
- **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.



Algorithm Properties

An algorithm possesses the following properties:

- It must be correct.
- It must be composed of a series of concrete steps.
- There can be no ambiguity as to which step will be performed next.
- It must be composed of a finite number of steps.
- It must terminate.
- It takes zero or more inputs
- It should be efficient and flexible
- It should use less memory space as much as possible
- It results in one or more outputs



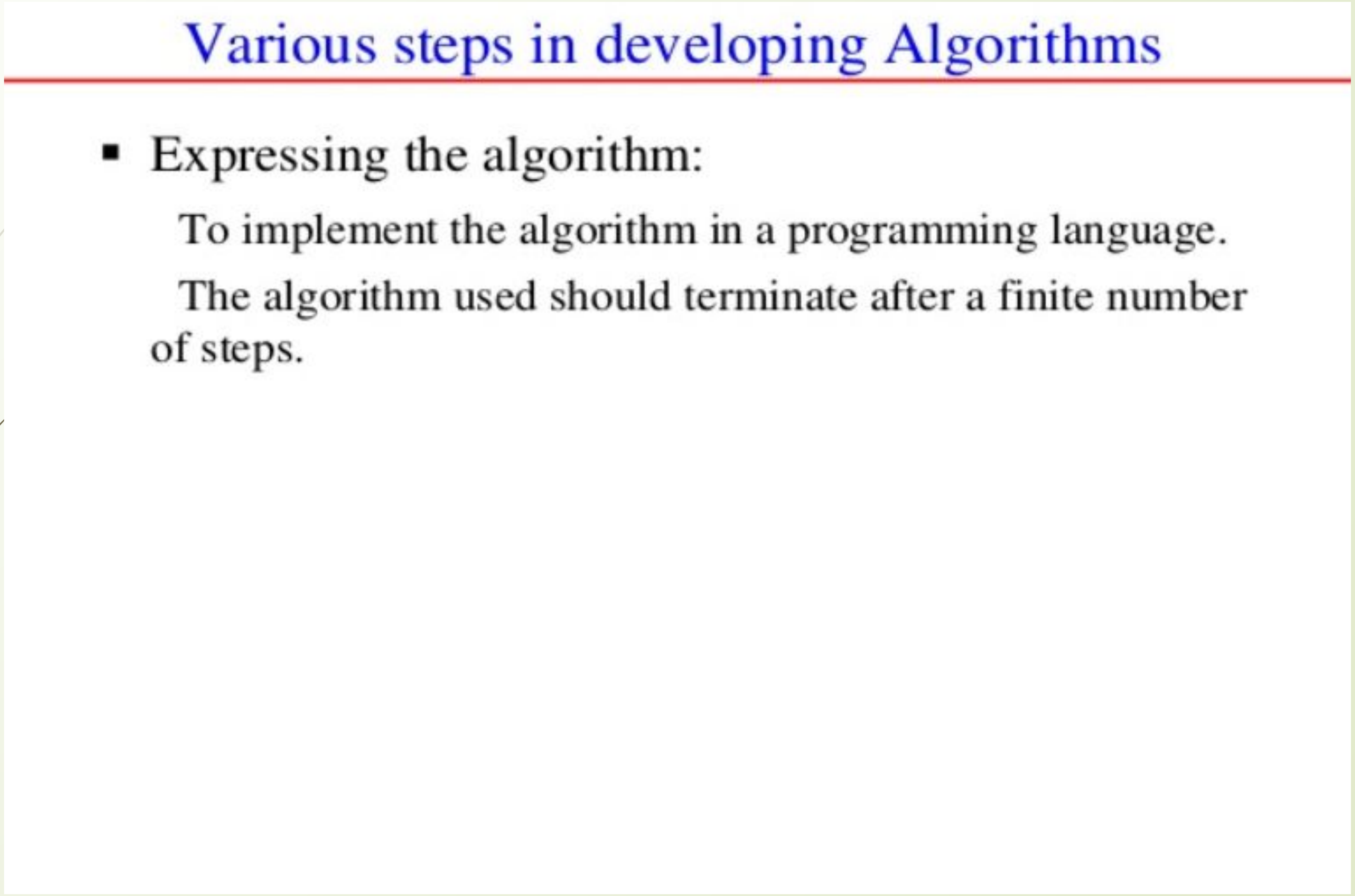
Various steps in developing Algorithms

- Devising the Algorithm:

  It's a method for solving a problem. Each step of an algorithm must be precisely defined and no vague statements should be used. Pseudo code is used to describe the algorithm , in less formal language than a programming language.

- Validating the Algorithm:

  The proof of correctness of the algorithm. A human must be able to perform each step using paper and pencil by giving the required input , use the algorithm and get the required output in a finite amount of time.

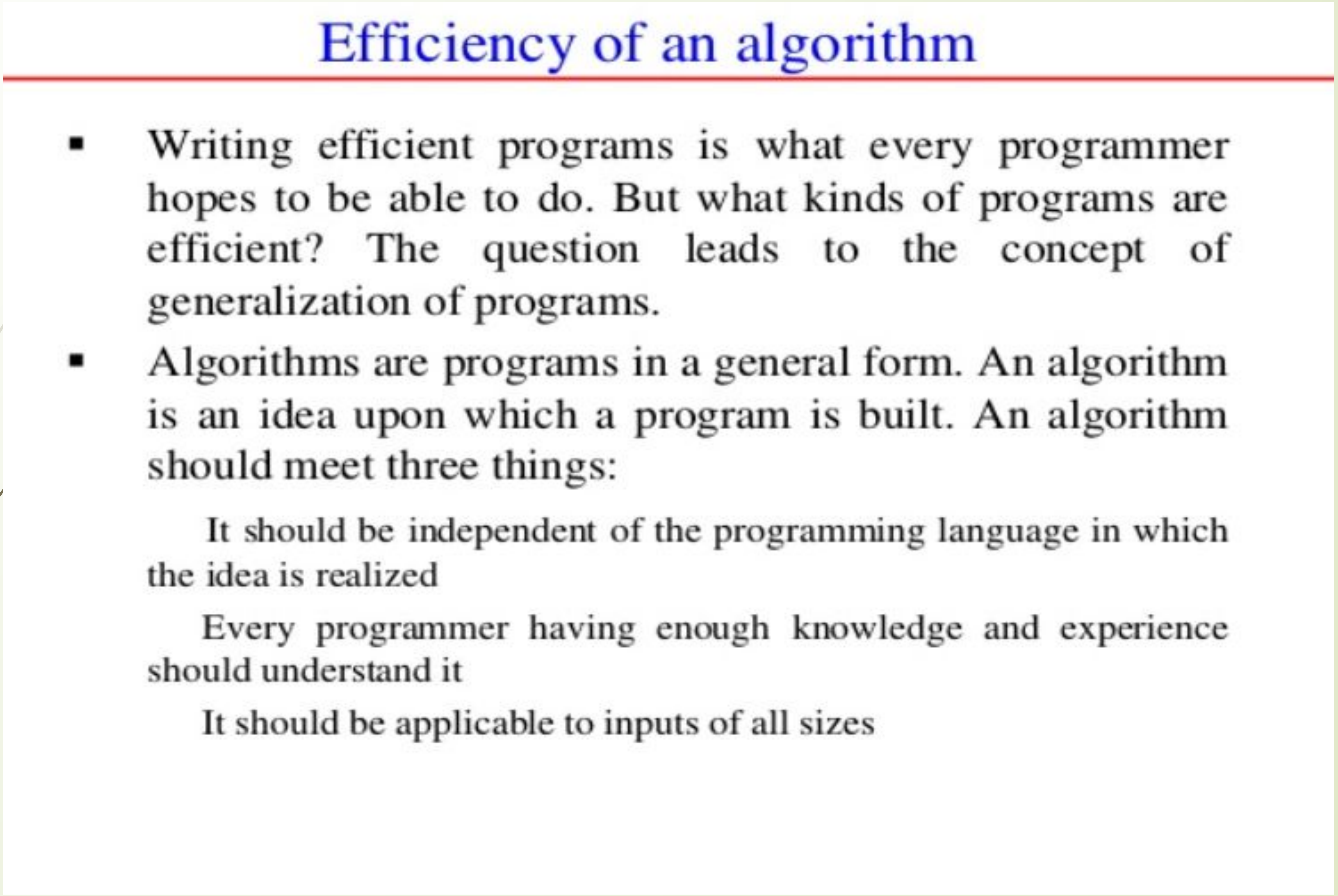


Various steps in developing Algorithms

- Expressing the algorithm:

  To implement the algorithm in a programming language.

  The algorithm used should terminate after a finite number of steps.



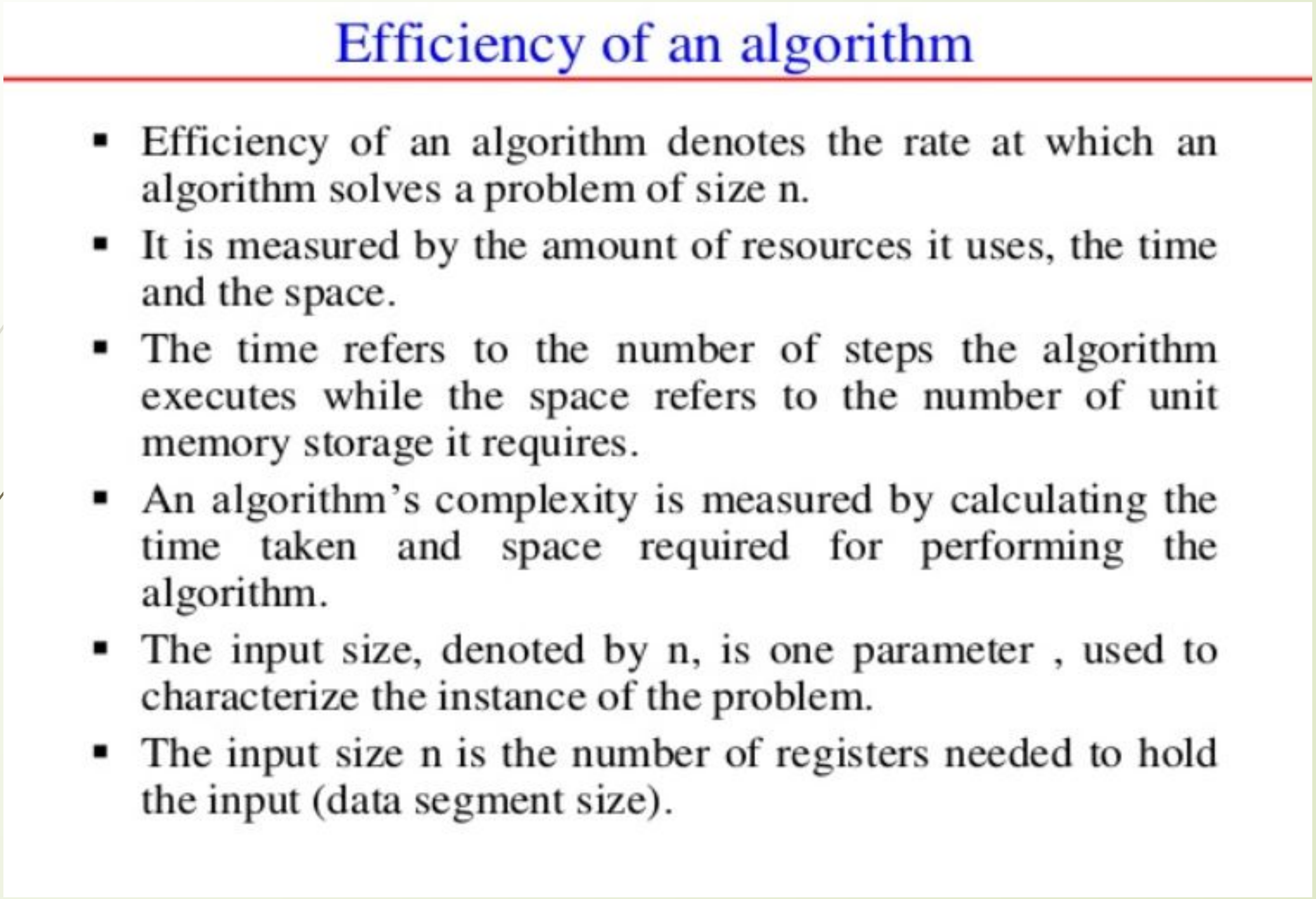
Efficiency of an algorithm

- Writing efficient programs is what every programmer hopes to be able to do. But what kinds of programs are efficient? The question leads to the concept of generalization of programs.
- Algorithms are programs in a general form. An algorithm is an idea upon which a program is built. An algorithm should meet three things:

  It should be independent of the programming language in which the idea is realized

  Every programmer having enough knowledge and experience should understand it

  It should be applicable to inputs of all sizes



Efficiency of an algorithm

- Efficiency of an algorithm denotes the rate at which an algorithm solves a problem of size n.
- It is measured by the amount of resources it uses, the time and the space.
- The time refers to the number of steps the algorithm executes while the space refers to the number of unit memory storage it requires.
- An algorithm's complexity is measured by calculating the time taken and space required for performing the algorithm.
- The input size, denoted by n, is one parameter , used to characterize the instance of the problem.
- The input size n is the number of registers needed to hold the input (data segment size).

How to Design an Algorithm?

? Inorder to write an algorithm, following things are needed as a pre-requisite:

1. The **problem** that is to be solved by this algorithm.
2. The **constraints** of the problem that must be considered while solving the problem.
3. The **input** to be taken to solve the problem.
4. The **output** to be expected when the problem the is solved.
5. The **solution** to this problem, in the given constraints.

? Then the algorithm is written with the help of above parameters such that it solves the problem.

Example: Consider the example to add three numbers and print the sum.

- **Step 1: Fulfilling the pre-requisites**
- As discussed above, in order to write an algorithm, its pre-requisites must be fulfilled.
  - **The problem that is to be solved by this algorithm**:
    - Add 3 numbers and print their sum.
  - **The constraints of the problem that must be considered while solving the problem**:
    - The numbers must contain only digits and no other characters.
  - **The input to be taken to solve the problem:**
    - The three numbers to be added.
  - **The output to be expected when the problem the is solved:**
    - The sum of the three numbers taken as the input.
  - **The solution to this problem, in the given constraints:**
    - The solution consists of adding the 3 numbers. It can be done with the help of '+' operator, or bit-wise, or any other method.

Step 2: <u>Designing the algorithm</u>

Now let's design the algorithm with the help of above pre-requisites:

- **Algorithm to add 3 numbers and print their sum:**

    - START
    - Declare 3 integer variables num1, num2 and num3.
    - Take the three numbers, to be added, as inputs in variables num1, num2, and num3 respectively.
    - Declare an integer variable sum to store the resultant sum of the 3 numbers.
    - Add the 3 numbers and store the result in the variable sum.
    - Print the value of variable sum
    - END

Step 3: <u>Testing the algorithm by implementing it.</u>

Inorder to test the algorithm, implement it in any language.

Algorithmic Notation

? Following conventions are used

? **Comments**

  ? Each instruction must be followed by a comment. The comments begin with a double slash, and explains the purpose of the instruction.

// this is a sample comment

? Appropriate use of comments enhances the readability of the algorithm, which in turn help in maintaining the algorithm.

Algorithmic Notation

- ? **Variable Names**
  - ? For variable names use italicized lowercase letters such as *max, loc,* etc.
  - ? For defined constants, if any, use uppercase letters

- ? **Assignment statement**
  - ? The assignment statement will use the notation as

    Set *max* = a

    - ? To assign the value of a to max
  - ? The right hand side of the assignment statement can have a value, a variable or an expression.
  - ? If several assignment statements appear in the same line, such as

    Set *k* = 1, *loc* = 1, *max* = a

  - ? then they are executed from left to right

Algorithmic Notation

- **Input/Output**
  - Data may be input and assigned to variables by means of a read statement with the following format

    Read: *variable* list

    where the variable list consists one or more variables separated by comma.
  - Similarly, the data held by the variables and the messages, if any, enclosed in double quoted can be output by means of a write statement with the following format:

    Write/Print: *message* and/or *variable* list

- **Execution of Instructions**
  - The instructions in the algorithm are usually executed one after the other as they appear in the algorithm. However, there may be instances when some instructions are skipped or some instructions may be repeated as a result of certain conditions.

Algorithmic Notation

- **Calling a Sub Algorithm**
  - To call a sub algorithm from the algorithm or sub algorithm, a special statement, called call instruction is used. This instruction is used along with the name of the sub algorithm followed by the list of the parameters.
  - Example: **Call *FindLocation* (*a, n, loc*)**

- **Completion of the Sub Algorithm**
  - The sub algorithm is completed with the execution of the return statement.

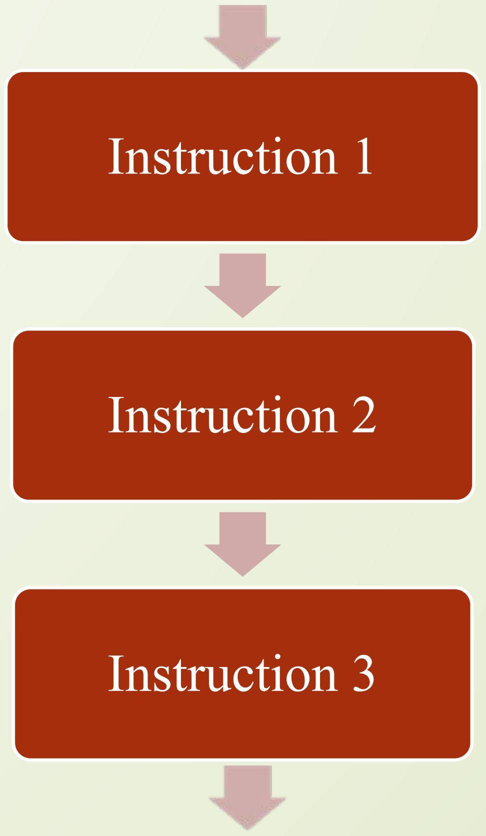
- **Completion of the Algorithm**
  - The algorithm is completed with the execution of the last statement. However, the algorithm can be terminated at any intermediate state using the exit instruction.

Control Structures

- An Algorithm is made up of the following basic control (Logic) structures that have been proved to be sufficient for expressing a solution and hence for writing any computer program
  - Sequence
  - Selection
  - Iteration

Control Structures

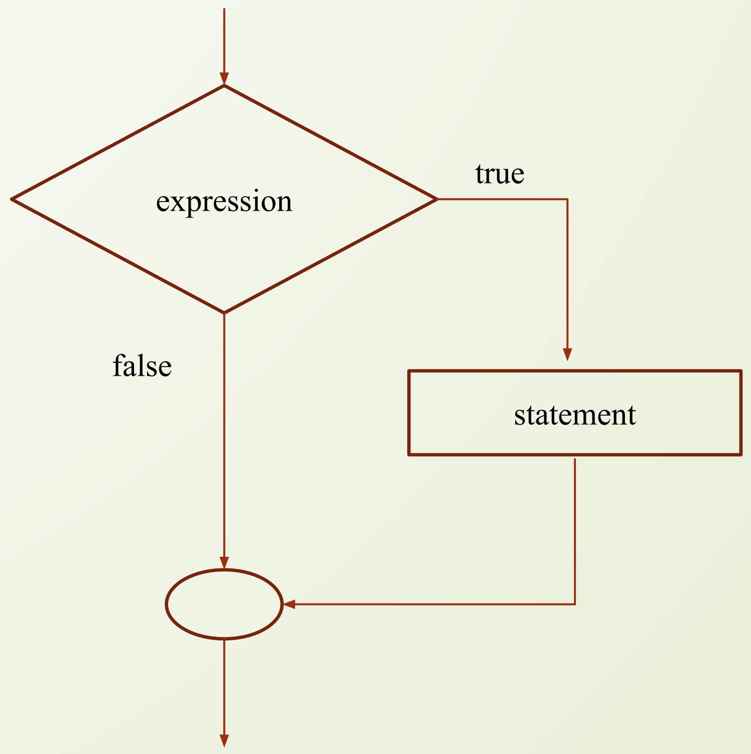
- ? **Sequence Logic**
  - ? Used for performing instructions one after another in sequence. Thus, for sequence logic, instructions are written in the sequence in which they are to be executed. The flow of logic is from top to bottom.



Control Structures

? **Selection Logic**

- ? Also known as Decision Logic, is used for making decision. It is used for selecting a proper path out of the alternative paths in the program logic.
- ? Decision Logic is depicted as either *If, If-Else* or *Else-If* ladder.

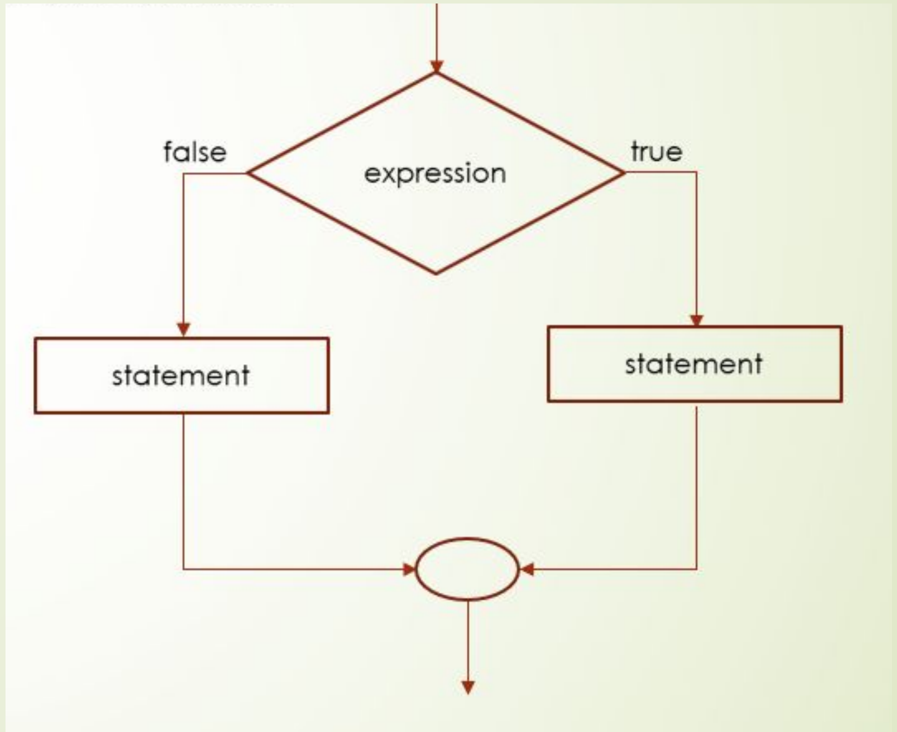


If Selection Structure

```
          :
          :
If ( expression ) then
        Statement
EndIf
          :
          :
```

Control Structures

```
            :
            :
If ( expression ) then
        Statement 1
Else
        Statement 2
EndIf
            :
            :
```



If-Else Selection Structure

Control Structures

? **Iterative Logic**

- ? Used to produce loops when one or more instructions are to be executed several times depending on some expressions. It uses two loop structures called For and While.

For

```
    :
    :
For k = r to s in steps of t
    Statement
EndFor
    :
    :
```

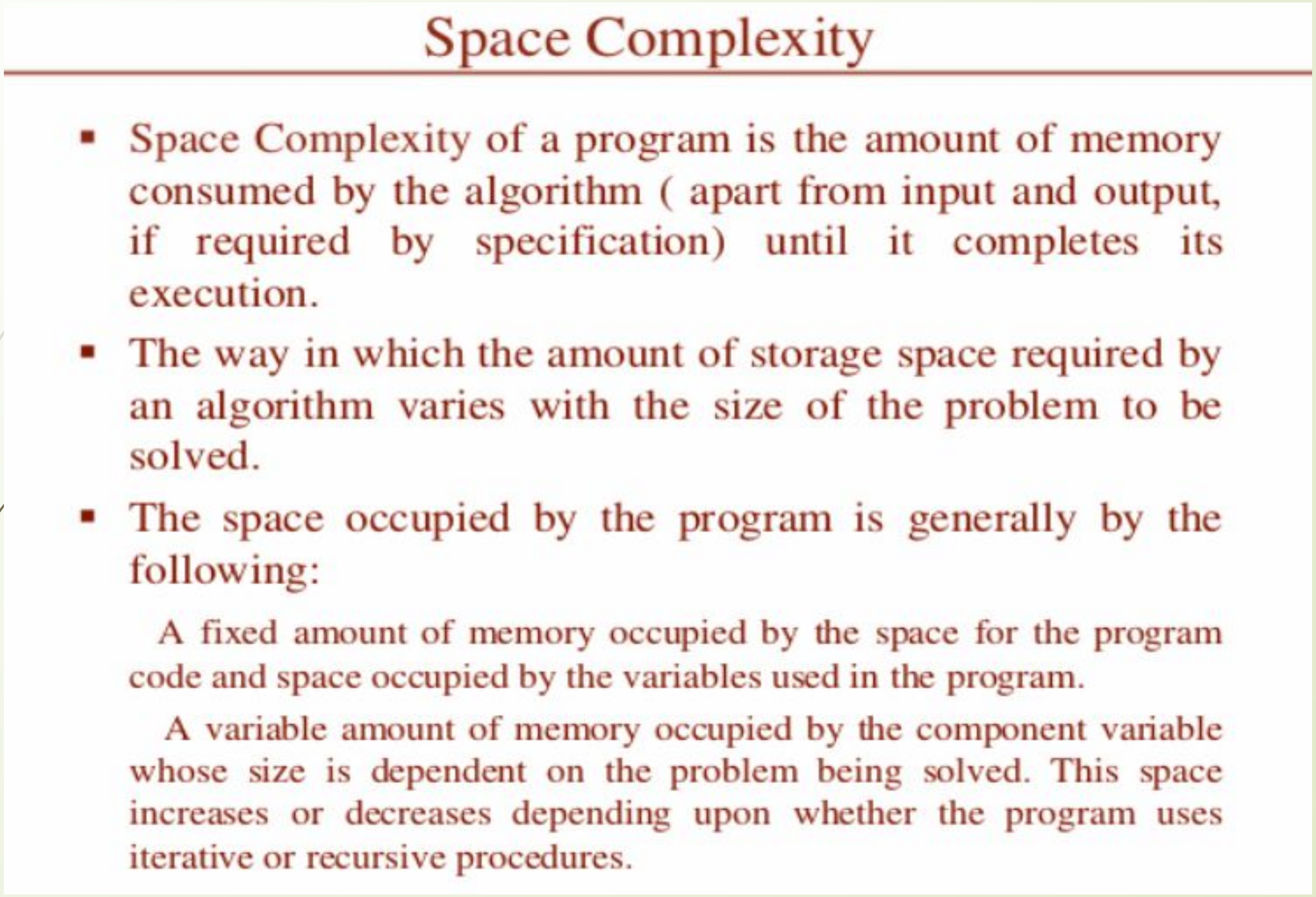
While

```
    :
    :
While ( expression)
    Statement
EndWhile
    :
    :
```

Algorithm Complexity

- There are basically two aspects of computer programming
  - One is the Data Organization, i.e., the data structures to represent the data of the problem.
  - The other one involves choosing the appropriate algorithm to solve the problem. Data structures and algorithms are inseparably linked.
- As an Algorithm is a sequence of steps to solve a problem, there may be more than one algorithm to solve a problem. The choice of particular algorithm depends on following considerations:
  - Performance requirements, *i.e.,* ***Time Complexity***
  - Memory Requirements, *i.e.,* ***Space Complexity***
- Performance requirements are usually more critical than memory requirements; hence, in general, it is not necessary to worry about memory unless they grow faster than performance requirements. Therefore, in general,

Algorithms are analyzed only on the basis of performance requirements, i.e., running-time efficiency

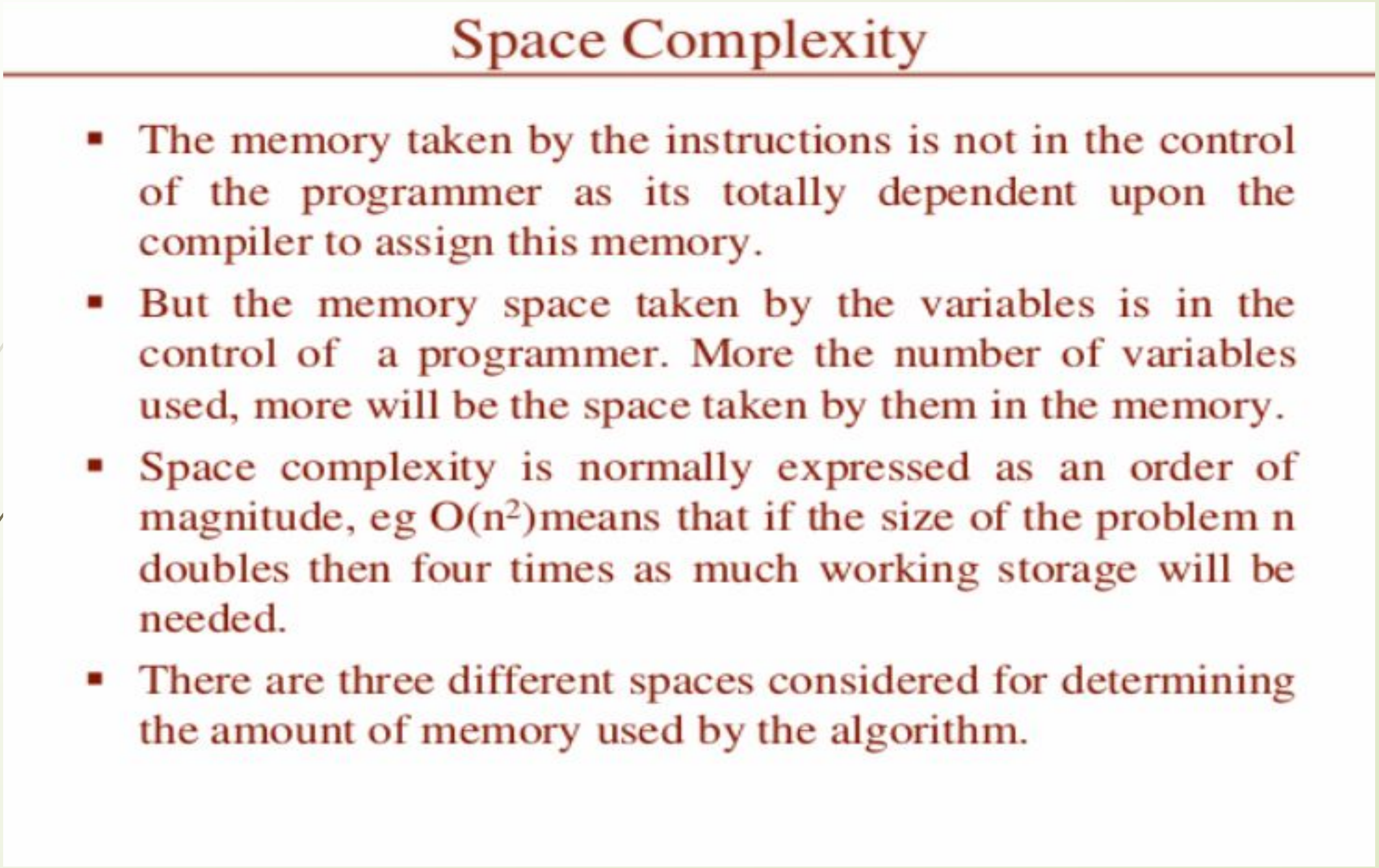


Space Complexity

- Space Complexity of a program is the amount of memory consumed by the algorithm (apart from input and output, if required by specification) until it completes its execution.
- The way in which the amount of storage space required by an algorithm varies with the size of the problem to be solved.
- The space occupied by the program is generally by the following:

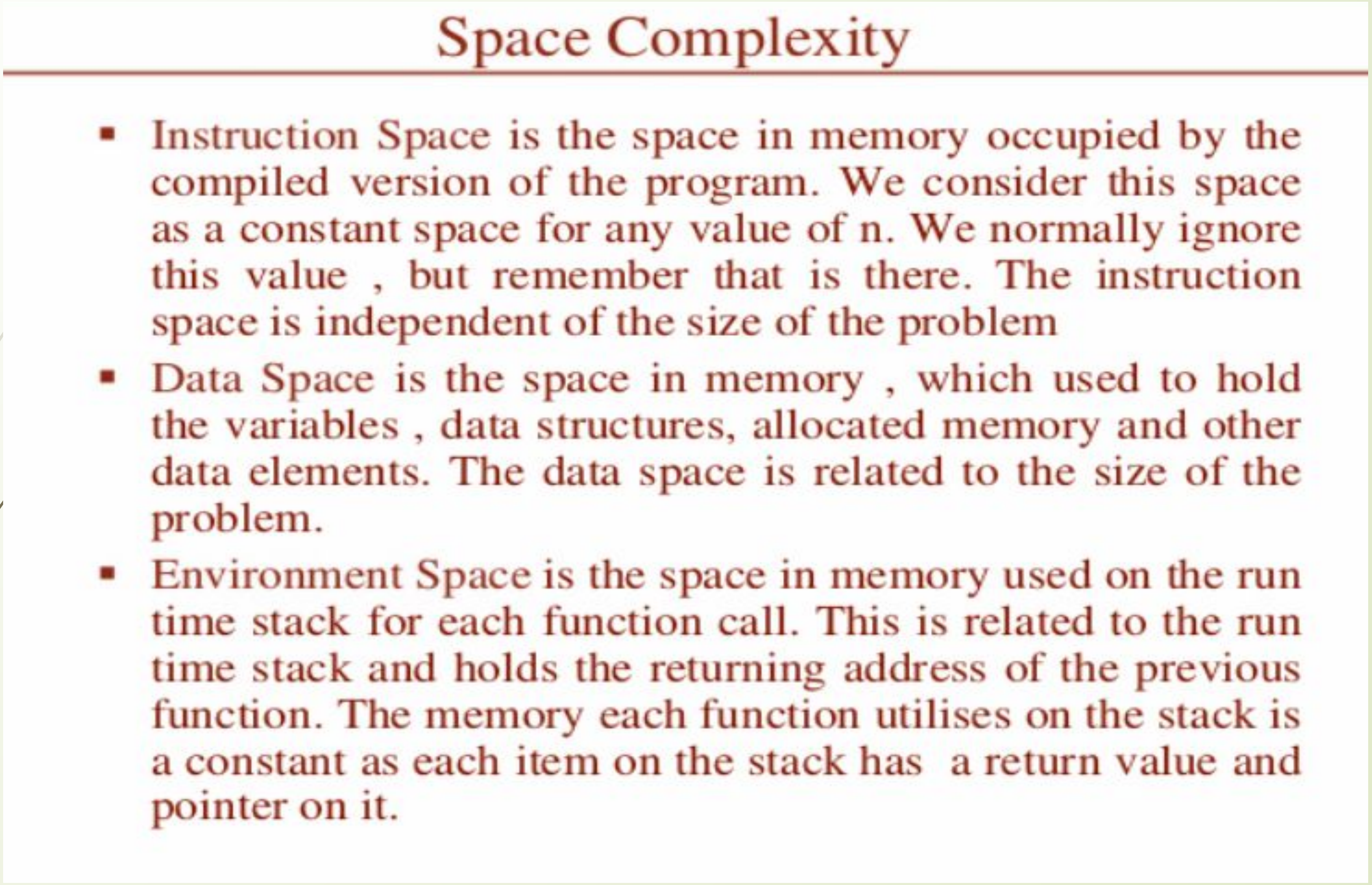
A fixed amount of memory occupied by the space for the program code and space occupied by the variables used in the program.

A variable amount of memory occupied by the component variable whose size is dependent on the problem being solved. This space increases or decreases depending upon whether the program uses iterative or recursive procedures.



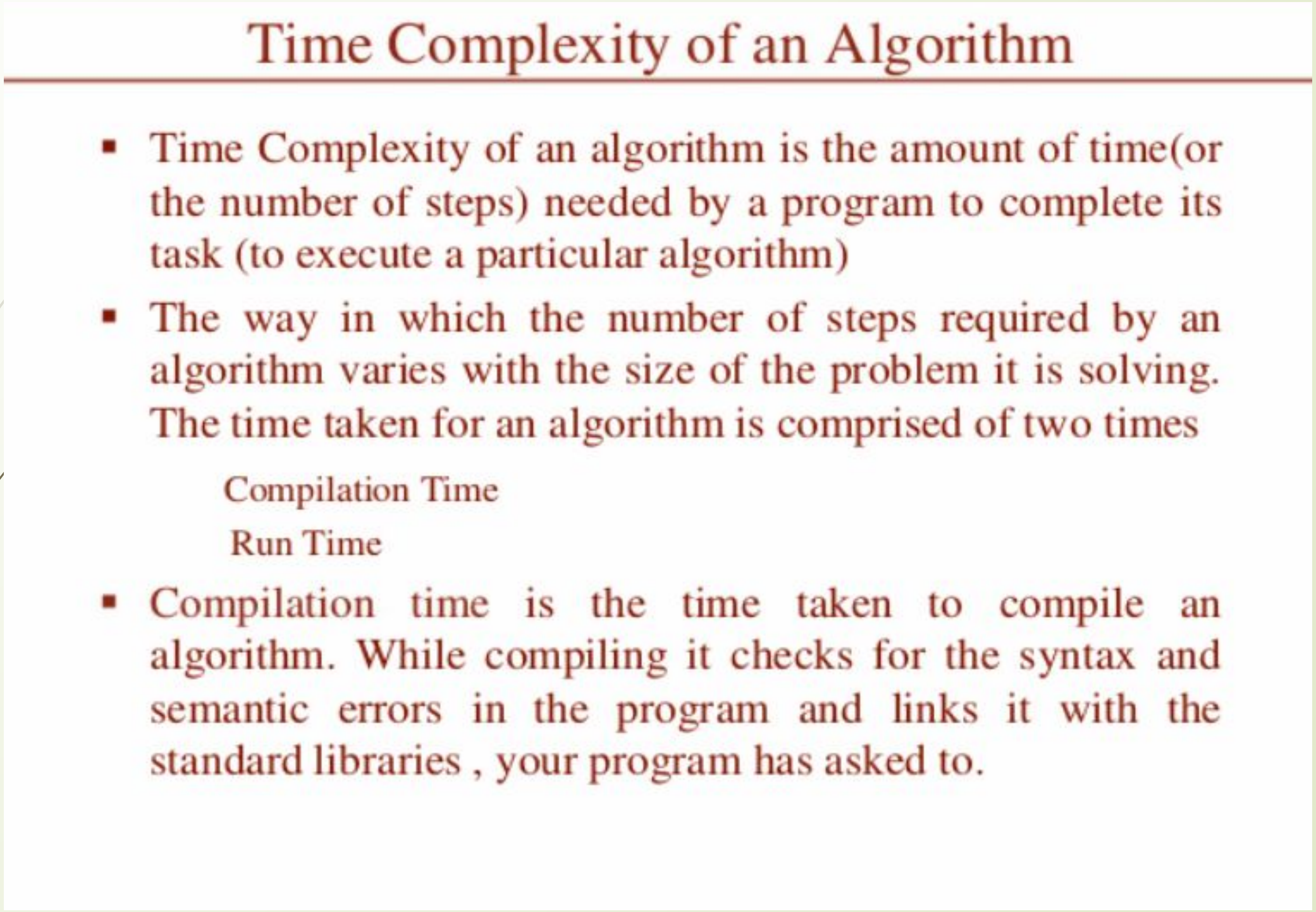
Space Complexity

- The memory taken by the instructions is not in the control of the programmer as its totally dependent upon the compiler to assign this memory.
- But the memory space taken by the variables is in the control of a programmer. More the number of variables used, more will be the space taken by them in the memory.
- Space complexity is normally expressed as an order of magnitude, eg $O(n^2)$means that if the size of the problem n doubles then four times as much working storage will be needed.
- There are three different spaces considered for determining the amount of memory used by the algorithm.



Space Complexity

- Instruction Space is the space in memory occupied by the compiled version of the program. We consider this space as a constant space for any value of n. We normally ignore this value , but remember that is there. The instruction space is independent of the size of the problem
- Data Space is the space in memory , which used to hold the variables , data structures, allocated memory and other data elements. The data space is related to the size of the problem.
- Environment Space is the space in memory used on the run time stack for each function call. This is related to the run time stack and holds the returning address of the previous function. The memory each function utilises on the stack is a constant as each item on the stack has a return value and pointer on it.

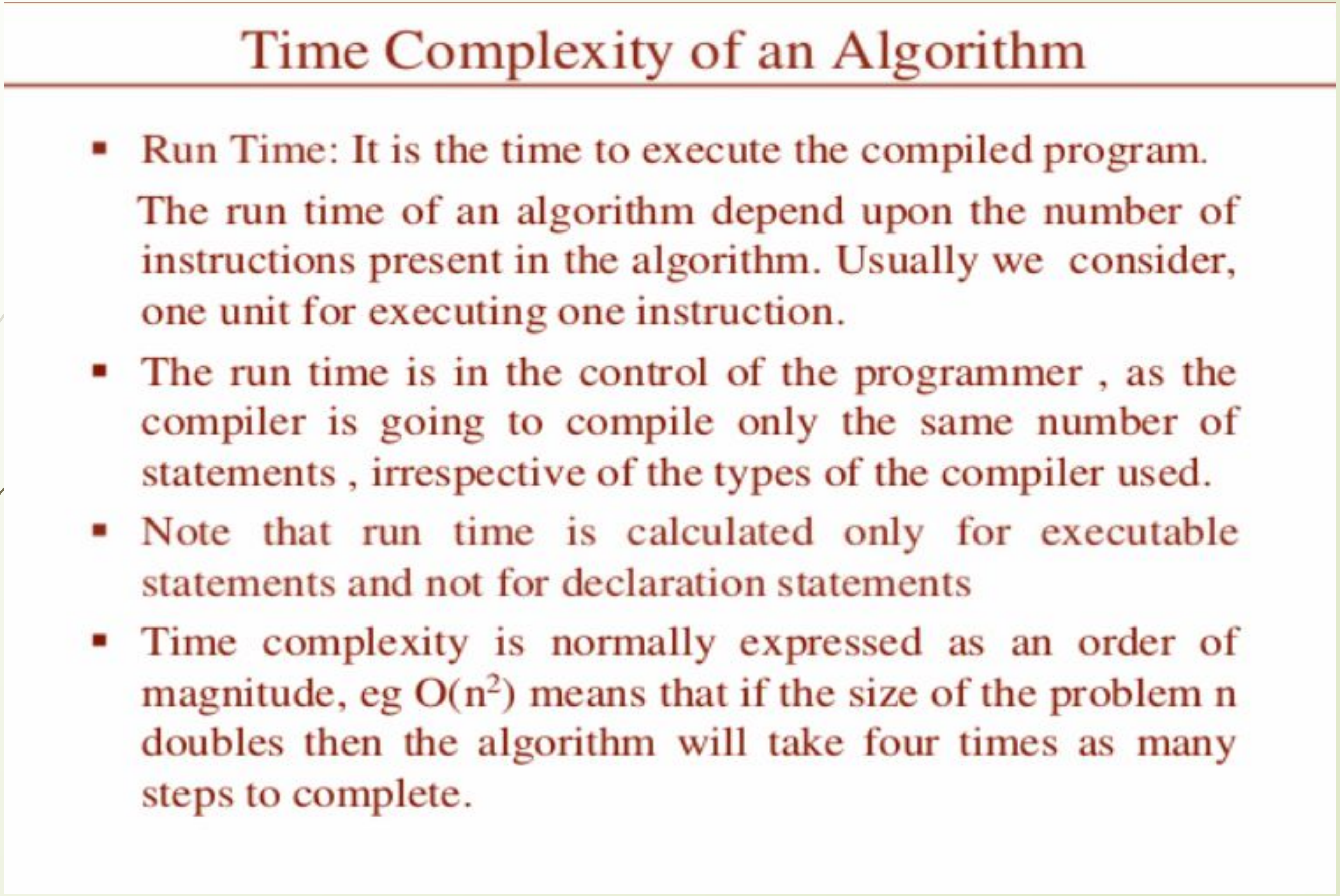


Time Complexity of an Algorithm

- Time Complexity of an algorithm is the amount of time(or the number of steps) needed by a program to complete its task (to execute a particular algorithm)
- The way in which the number of steps required by an algorithm varies with the size of the problem it is solving. The time taken for an algorithm is comprised of two times

  Compilation Time

  Run Time
- Compilation time is the time taken to compile an algorithm. While compiling it checks for the syntax and semantic errors in the program and links it with the standard libraries , your program has asked to.



Time Complexity of an Algorithm

- Run Time: It is the time to execute the compiled program.

  The run time of an algorithm depend upon the number of instructions present in the algorithm. Usually we consider, one unit for executing one instruction.
- The run time is in the control of the programmer , as the compiler is going to compile only the same number of statements , irrespective of the types of the compiler used.
- Note that run time is calculated only for executable statements and not for declaration statements
- Time complexity is normally expressed as an order of magnitude, eg $O(n^2)$ means that if the size of the problem n doubles then the algorithm will take four times as many steps to complete.

Time Complexity of an Algorithm

? There are three categories for determining the time complexity of algorithms.

? **Worst Case** − This is the scenario where a particular data structure operation takes maximum time it can take. If an operation's worst case time is $f(n)$ then this operation will not take more than $f(n)$ time where $f(n)$ represents function of n.

? **Average Case** − This is the scenario depicting the average execution time of an operation of a data structure. If an operation takes $f(n)$ time in execution, then m operations will take $mf(n)$ time.

? **Best Case** − This is the scenario depicting the least possible execution time of an operation of a data structure. If an operation takes $f(n)$ time in execution, then the actual operation may take time as the random number which would be maximum as $f(n)$.

Time-Space Trade-off

- The best algorithm to solve given problem is one that requires less space in memory and takes less time to complete its execution.
- But in practice, it is not always possible to achieve both of these objectives.
- That is why we can say that there exists a ***time-space*** trade among algorithms.

- Therefore, if space is our constraint, then we have to select an algorithm that requires less space at the cost of more execution time.
- On the other hand, if time is our constraint such as in real time systems, we have to choose a program that takes less time to complete its execution at the cost of more space.

Algorithm Complexity

?**Algorithm complexity** is a **measure** which evaluates the order of the **count of operations**, performed by a given or algorithm as a function of the size of the input data.

?In other words, complexity is a rough **approximation of the number of steps** necessary to execute an algorithm.

?Algorithmic complexity is concerned about how fast or slow particular algorithm performs.

?We define complexity as a numerical function *T(n)* - time versus the input size *n*.

?We want to define time taken by an algorithm without depending on the implementation details.

?But you agree that *T(n)* does depend on the implementation! A given algorithm will take different amounts of time on the same inputs depending on such factors as: processor speed; instruction set, disk speed, brand of compiler and etc.

?The way around is to estimate efficiency of each algorithm *asymptotically*. We will measure time *T(n)* as the number of elementary "steps" (defined in any way), provided each such step takes constant time.

Example

Let us consider an example: addition of two integers.

We will add two integers digit by digit (or bit by bit), and this will define a "step" in our computational model.
Therefore, we say that addition of two n-bit integers takes n steps.
Consequently, the total computational time is

$$T(n) = c * n,$$

where c is time taken by addition of two bits.

On different computers, addition of two bits might take different time, say c_1 and c_2, thus the addition of two n-bit integers takes $T(n) = c_1 * n$ and $T(n) = c_2 * n$ respectively.
This shows that different machines result in different slopes, but time $T(n)$ grows linearly as input size increases.

The process of abstracting away details and determining the rate of resource usage in terms of the input size is one of the fundamental ideas in computer science.

Asymptotic Notations

The goal of computational complexity is to classify algorithms according to their performances. We will represent the time function T(n) using the "big-O" notation to express an algorithm runtime complexity. For example, the following statement

$$T(n) = O(n^2)$$

says that an algorithm has a quadratic time complexity.

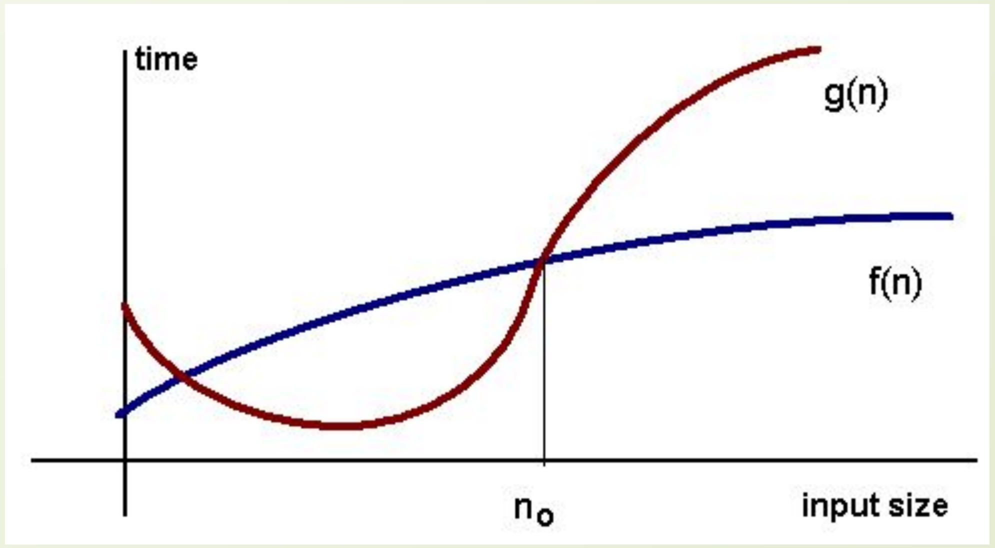
Big-O Analysis of Algorithms

For any monotonic functions f(n) and g(n) from the positive integers to the positive integers, we say that f(n) = O(g(n)) when there exist constants $c > 0$ and $n_0 > 0$ such that

$$f(n) \leq c * g(n), \text{ for all } n \geq n_0$$

Intuitively, this means that function f(n) does not grow faster than g(n), or that function g(n) is an **upper bound** for f(n), for all sufficiently large $n \to \infty$

Here is a graphic representation of f(n) = O(g(n)) relation:



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

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

Categories of Big Oh Notation

Constant Time: O(1)

An algorithm is said to run in constant time if it requires the same amount of time regardless of the input size. Examples:

- array: accessing any element
- fixed-size stack: push and pop methods
- fixed-size queue: enqueue and dequeue methods

Linear Time: O(n)

An algorithm is said to run in linear time if its time execution is directly proportional to the input size, i.e. time grows linearly as input size increases. Examples:

- array: linear search, traversing, find minimum
- ArrayList: contains method
- queue: contains method

Categories of Big Oh Notation

Logarithmic Time: O(log n)

An algorithm is said to run in logarithmic time if its time execution is proportional to the logarithm of the input size. Example:

- binary search

Quadratic Time: $O(n^2)$

An algorithm is said to run in logarithmic time if its time execution is proportional to the square of the input size. Examples:

- bubble sort, selection sort, insertion sort