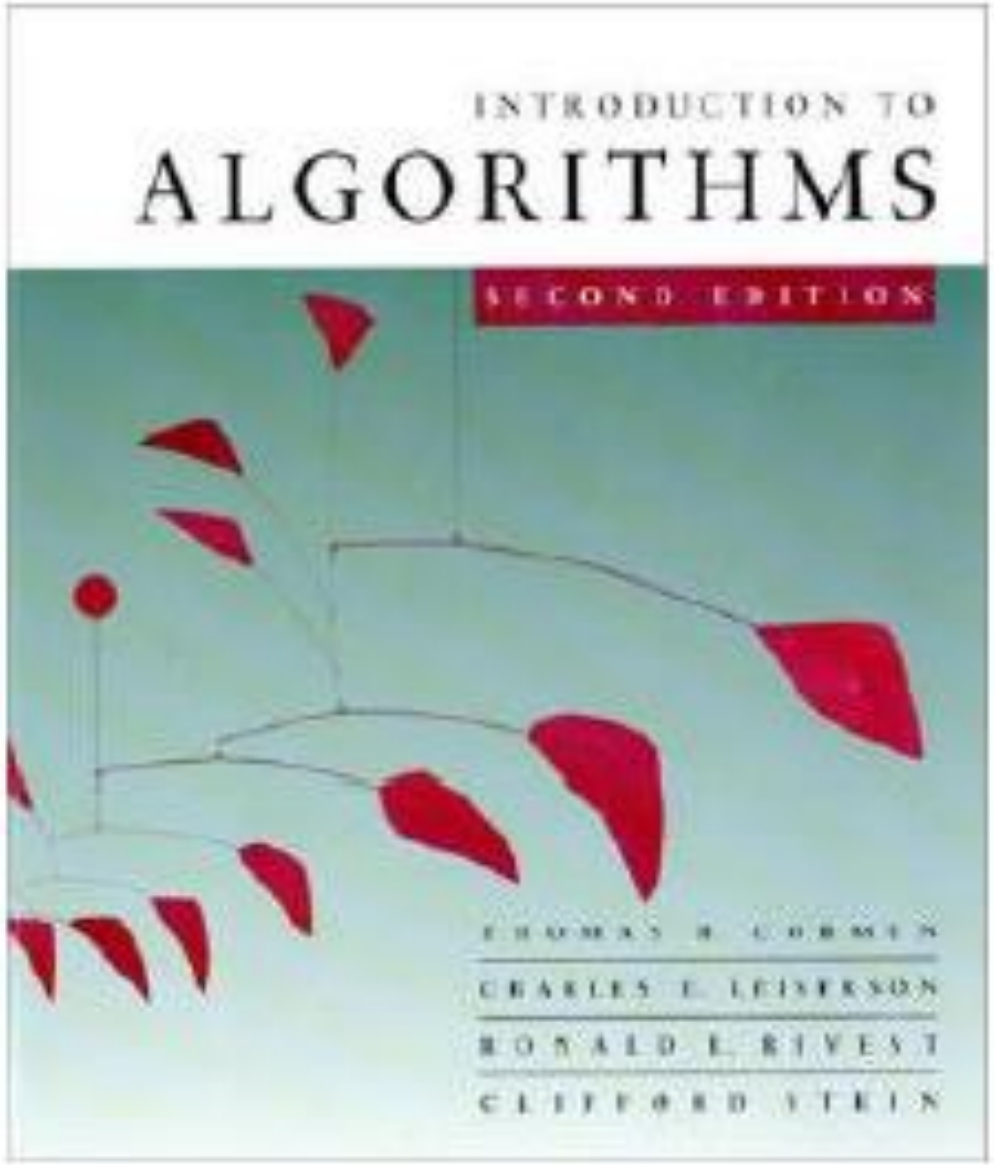
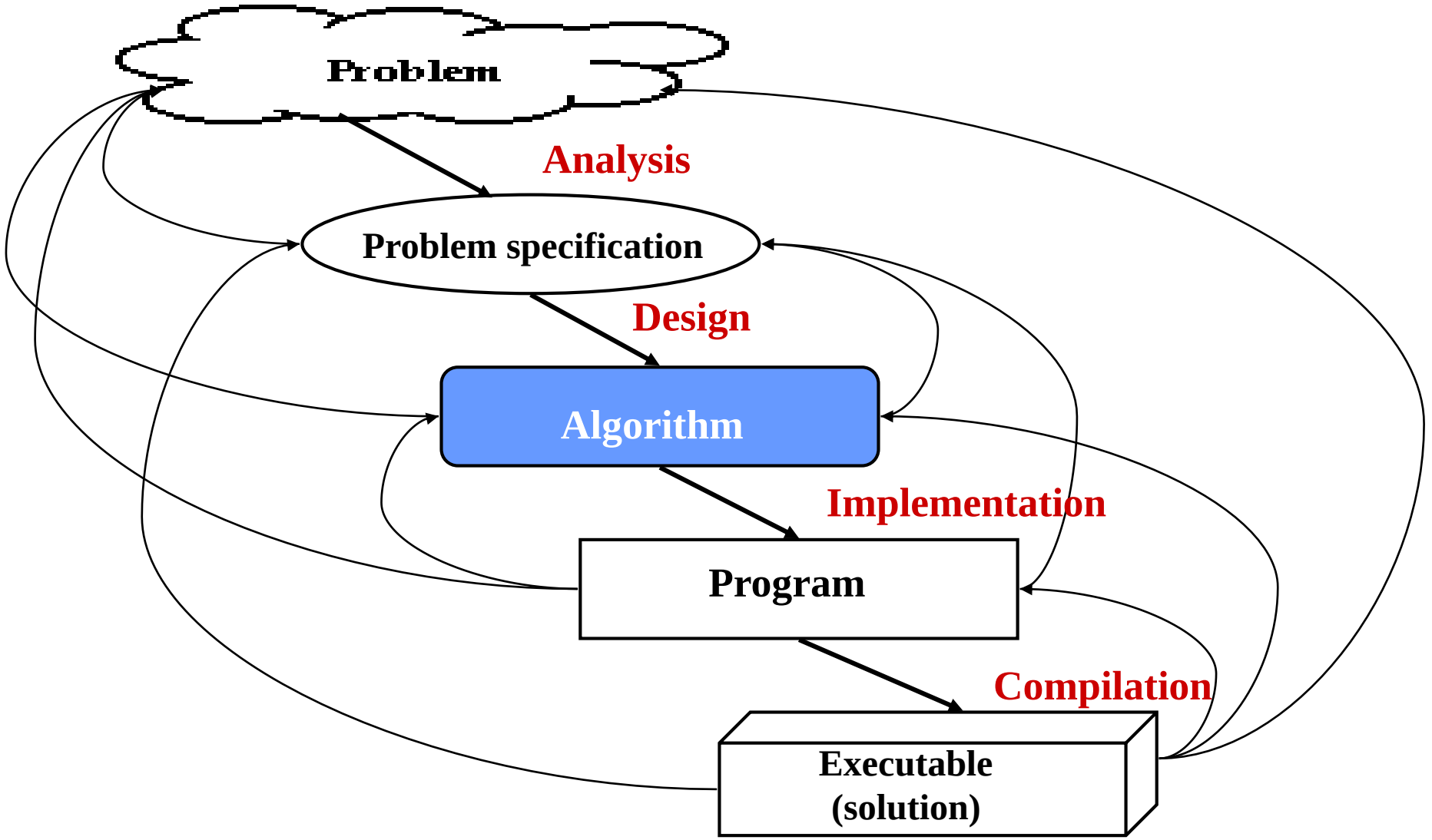
Introduction to Algorithms



Computational problems

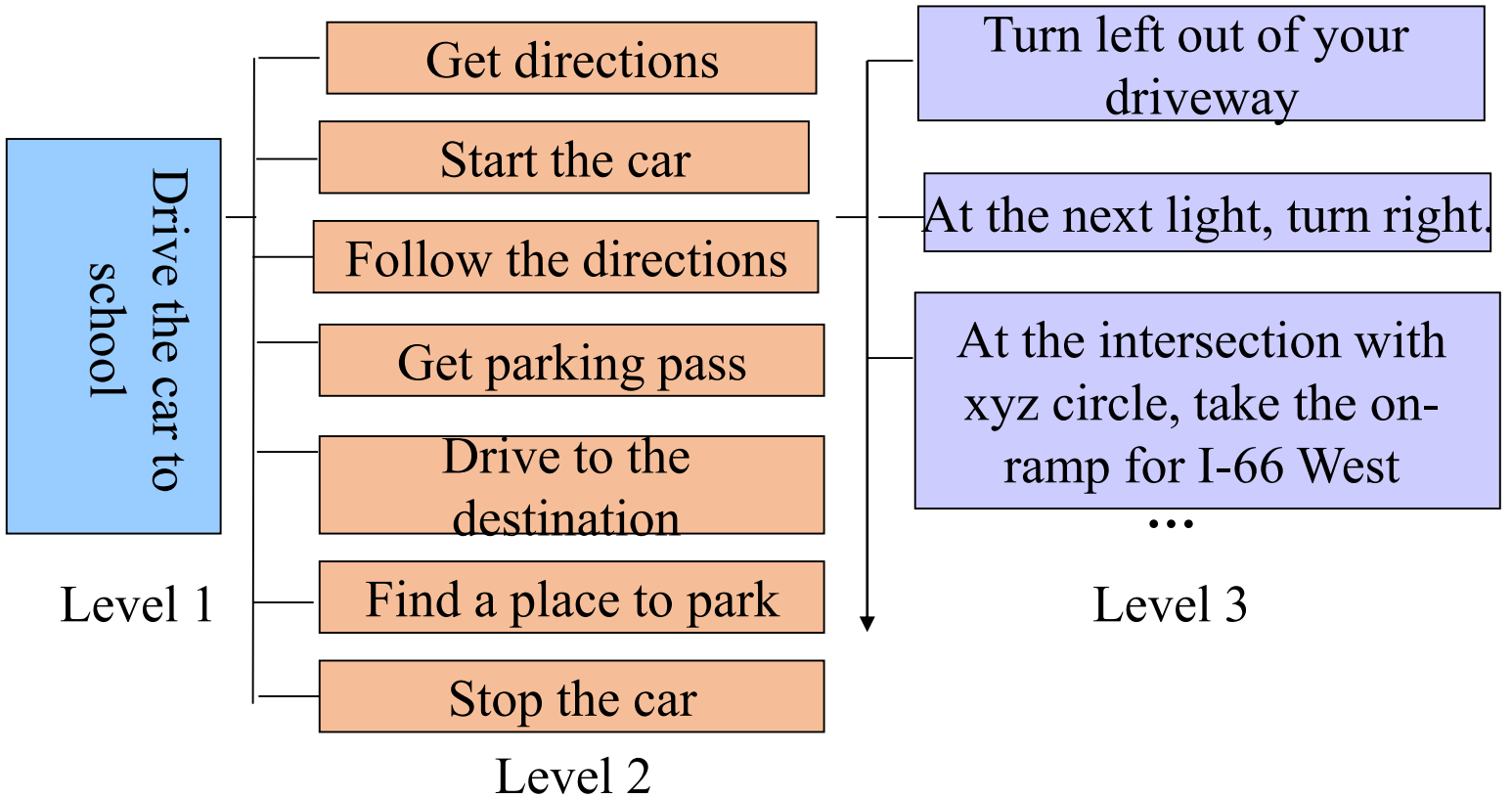
- A computational problem specifies an input-output relationship
  - What does the input look like?
  - What should the output be for each input?
- Example:
  - Input: an integer number n
  - Output: Is the number prime?
- Example:
  - Input: A list of names of people
  - Output: The same list sorted alphabetically

The Problem-solving Process



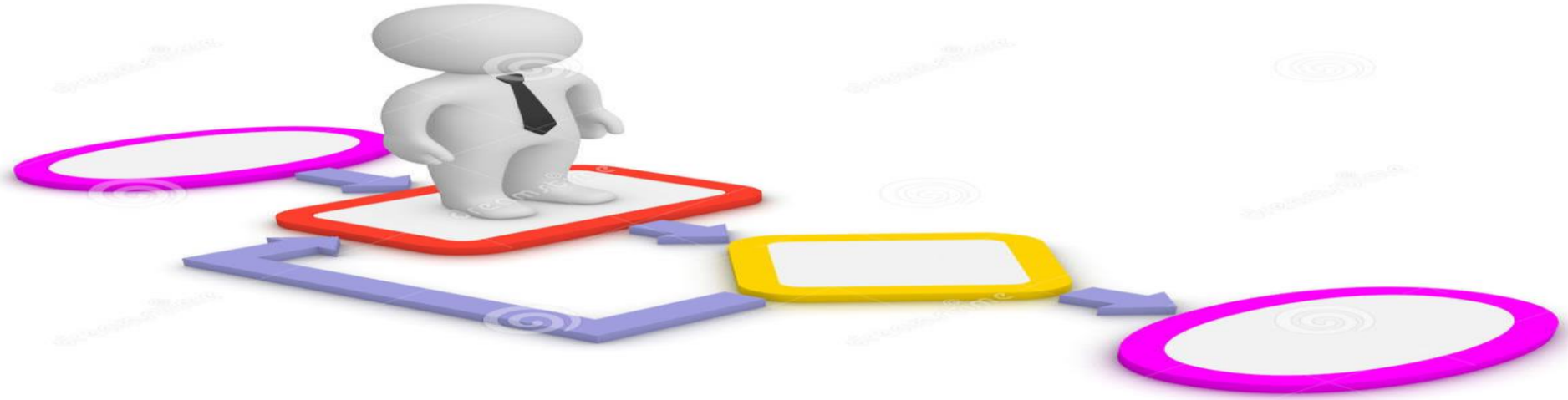
Problems and Solution as Algorithm

- For example, we need to solve a computational problem
  - "Convert a weight in pounds to Kg"
- An algorithm specifies how to solve it,
  1. *Read weight-in-pounds*
  2. *Calculate weight-in-Kg = weight-in-pounds * 0.455*
  3. *Print weight-in-Kg*
- Another example, Suppose you have to drive a car to school.



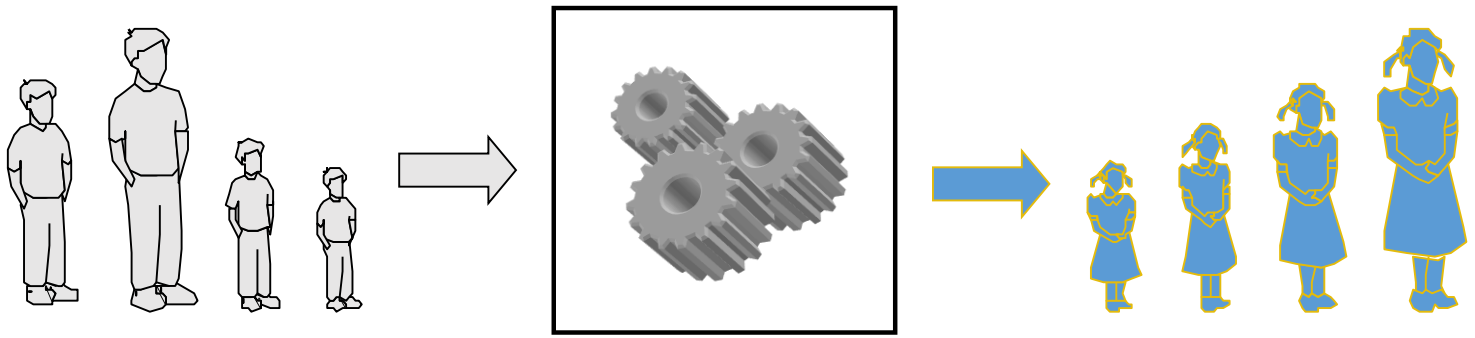
What is an algorithm?

- The word algorithm comes from the name of a Persian mathematician Abu Ja’far Mohammed ibn-i Musa al Khowarizmi.
- In computer science, this word refers to a special method useable by a computer for solution of a problem.
- An algorithm is a finite set of (step-by-step) instructions that if followed accomplishes a particular task or to solve a given Problem.



What is an algorithm?

- A tool for solving a well-specified computational problem
- A well-defined computational procedure that takes some value, or set of values, as input and produces some value, or set of values, as output.
- Written in a pseudo code which can be implemented in the language of programmer's choice.



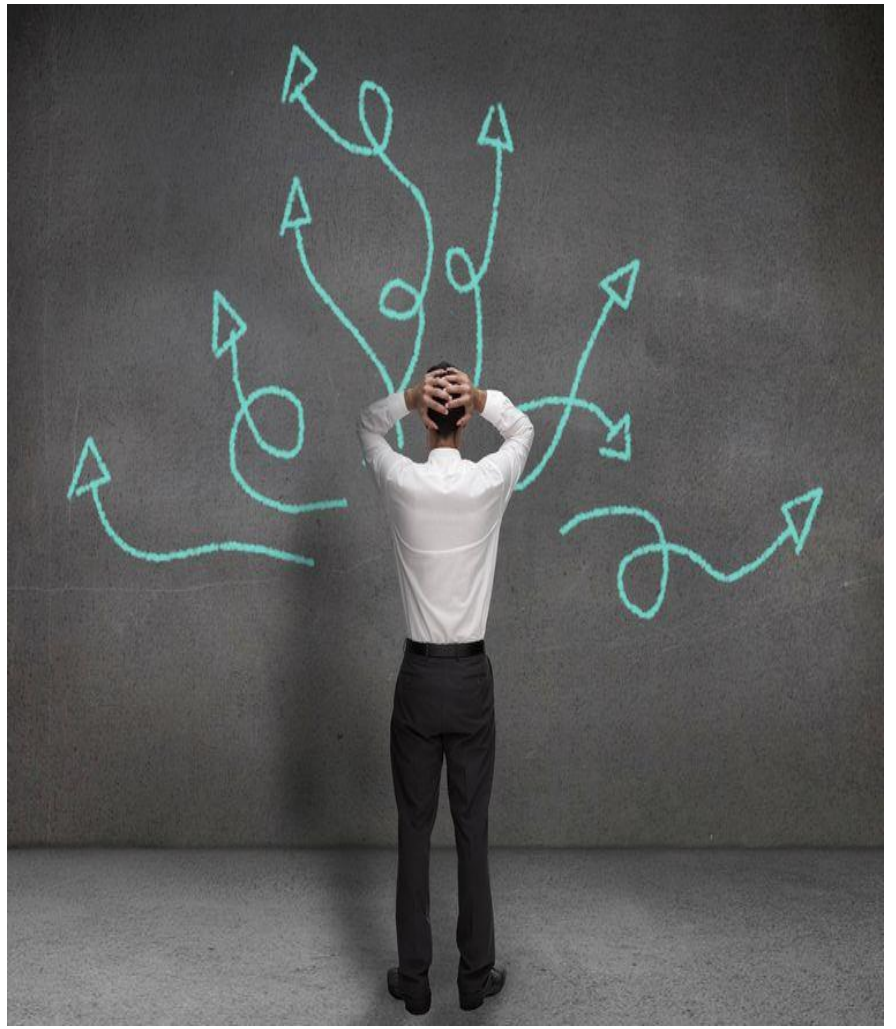
Properties of algorithms

- **Input**: what the algorithm takes in as input, Zero or more quantities supplied.
- **Output:** what the algorithm produces as output, At least one quantity is produced.
- **Definiteness**: the steps are defined precisely, Each instruction is clear and unambiguous.
- **Finiteness**: the steps required should be finite, If we trace out the instructions of an algorithm, then for all cases, the algorithm terminates after a finite number of steps.
- **Effectiveness**: each step must be able to be performed in a finite amount of time, Every instruction must be basic so that it can be carried out in in principal by a person using only pen and pencil.

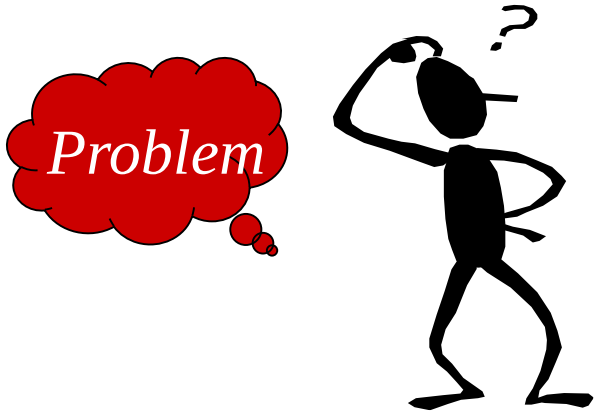
Other Properties of algorithms

- **An algorithm should**

- **Non –ambiguous**: more than one interpretation
- **Generality**: applicable to all problems of a similar form.
- **Correctness**: produce an appropriate and correct output For each input produce.
- **Efficient**: run as quickly as possible, and use as little memory as possible.

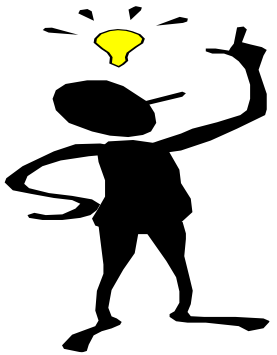


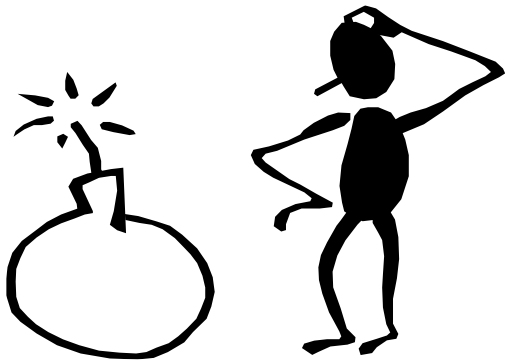
From Algorithms to Programs



Algorithm: A sequence of instructions describing how to do a task (or process)

Pseudocode: A simple way of writing programming code in English.





Java/Python/C/C++ Program

Specification of Algorithms

- Algorithms are generally expressed in human readable form
- Two Approaches are:natural language and pseudo code

Use of Natural Language

- English words and phrases can be used to express statements and processing steps
  - F ▶ Operators are identified by algebraic symbols (+, -,*, /, <, >,=, !=)
  - a
    le ▶ Input and Output statements are expressed using words read and write
  - A ▶ Control Structures are described using phrases if-then, if-then-else
  - C ▶ Repetitive operations are described by phrases Repeat, for, while, until
- For example, Euclid's algorithm for finding GCD of two positive integers can be expressed in plain language as

computation

tiply

Read positive integers n,m
Set x to n, y to m
Repeat steps 4-5 while x does not equal y
If x is greater than y then set x to x-y
If y is less than x then set y to y-x
Write x y

Specification of Algorithms

Use of Pseudocode

- Algorithms in natural language tend to be wordy and verbose
- Pseudocode provides an alternative way of expressing algorithms
- It is a mixture of natural language and programming notation
- In practice several convention are used to write pseudo code
- The following notation is commonly used:
  - Algorithm is identified by a name
  - Comments are enclosed in square brackets
  - Variable names are written in capital letters
  - Assignment statement is coded using left arrow ←
- Operators are identified by algebraic symbols (+, -,*, /, <, >,=, !=)
- Input and Output statements are expressed using words read and write
- Control Structures are described using phrases if-then, if-then-else
- Repetitive operations are described by phrases Repeat, for, while, until

Specification of Algorithms

- The Euclid's algorithm can be expressed in pseudocode as under:

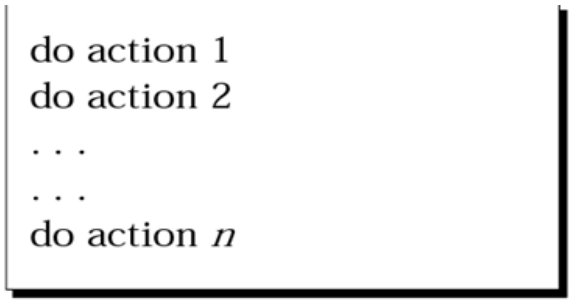
Algorithm gcd [This algorithm returns the greatest common divisor (gcd) of two positive integers]
1. Read : N,M[Read integers N,M]
2. X◄— N;Y◄—M M[Assign values to x and Y]
3. Repeat steps 4 to 5 while (X != Y)
4. if X > Y then X◄— X-Y
5. if Y > X then Y◄— Y-X
6. Write : "Greatest common divisor :", X
7. Exit

- C++ code for the gcd algorithm:

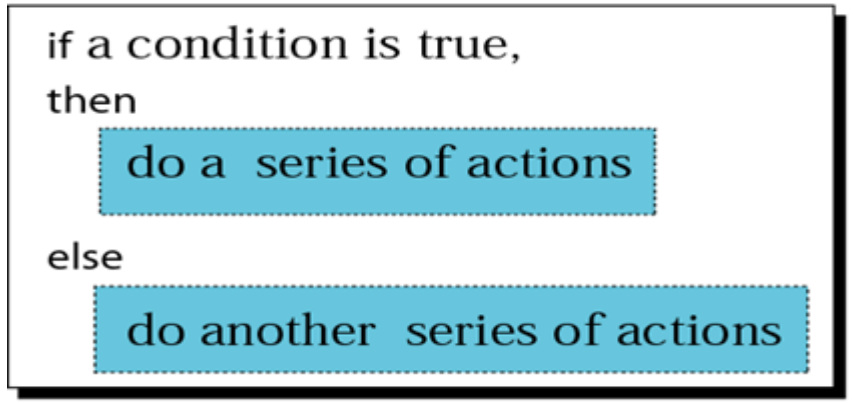
```
// GCD Algorithm
# include <iostream.h>
main() {
int m,n,x,y;
cin >> n;
cin >> m;
x=n;
y=m;
while (x != y)
{
if (x > y)
x=x-y;
else
y=y-x;
}
cout << "Greatest common divisor" <<x;
}
```

Components of an Algorithm

- Variables and values
- Instructions
- Sequences
  - A series of instructions

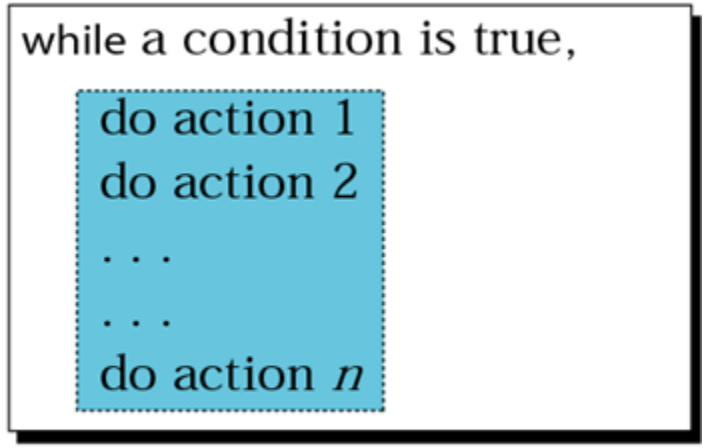


a. Sequence



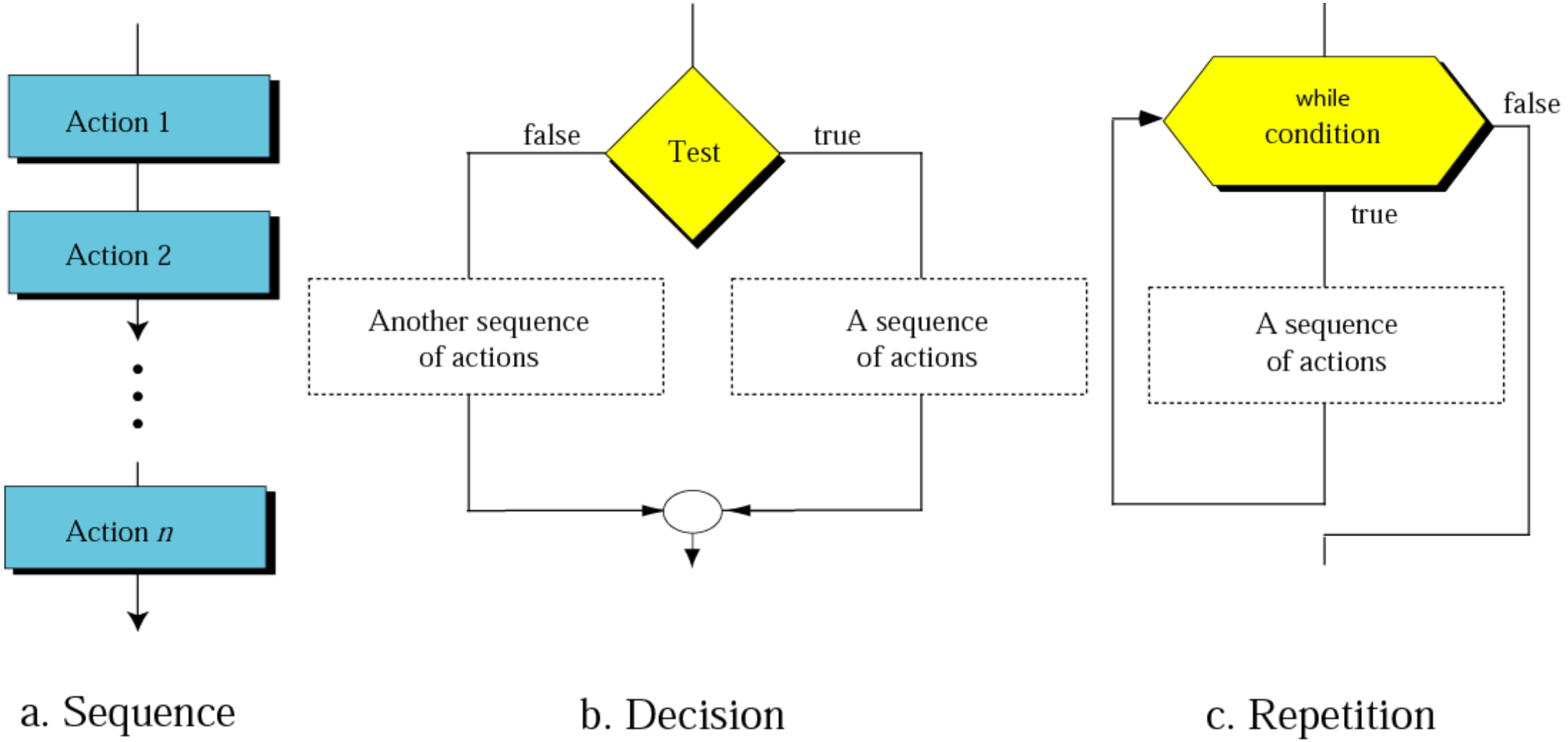
b. Decision

- Procedures (also referred as function/module/sub-routine)
  - A named sequence of instructions
- Selections (Decision)
  - An instruction that decides which of two possible sequences is executed
  - The decision is based on true/false condition
- Repetitions
  - Also known as iteration or loop
- Documentation
  - Records what the algorithm does



c. Repetition

Flowcharts of Algorithm Components



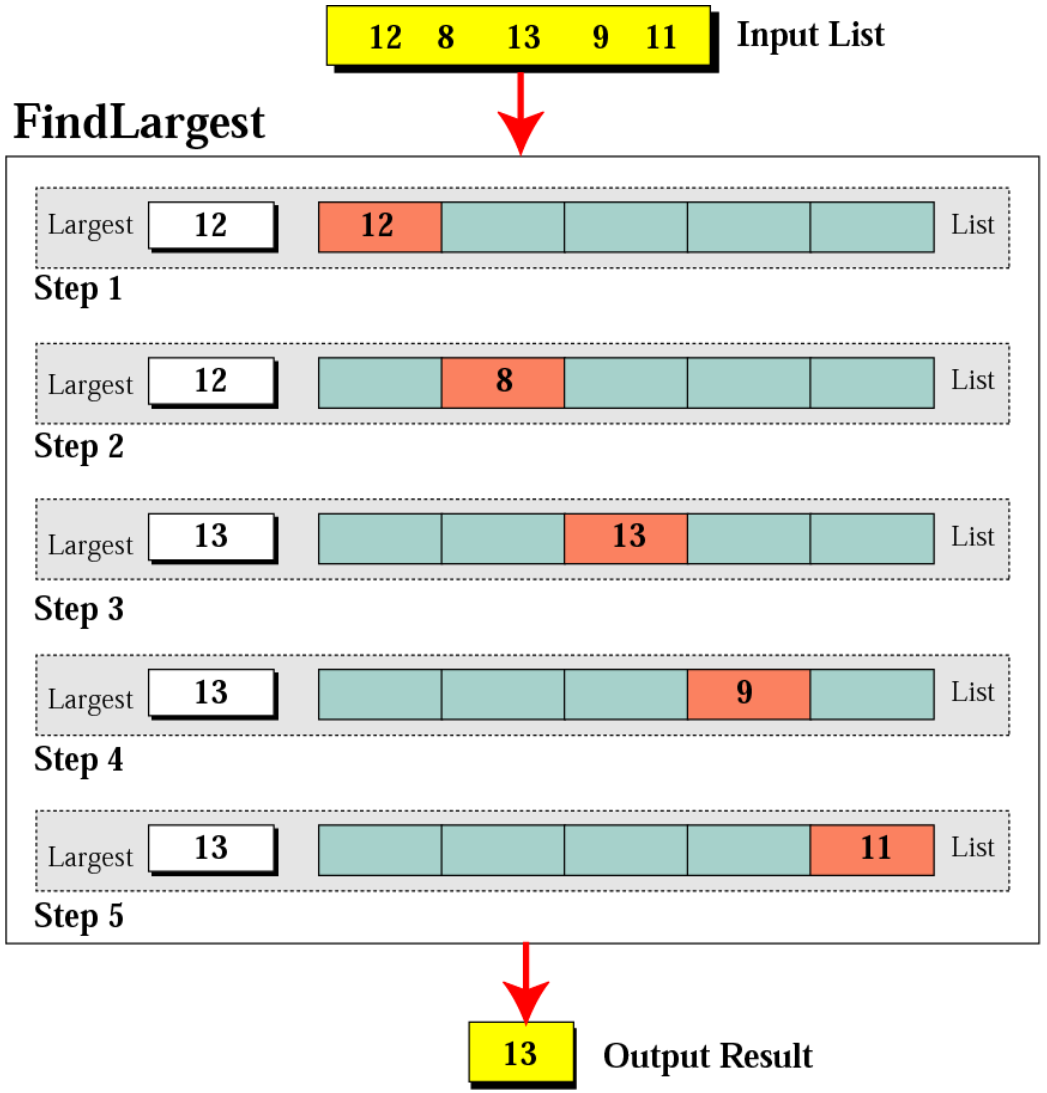
a. Sequence

b. Decision

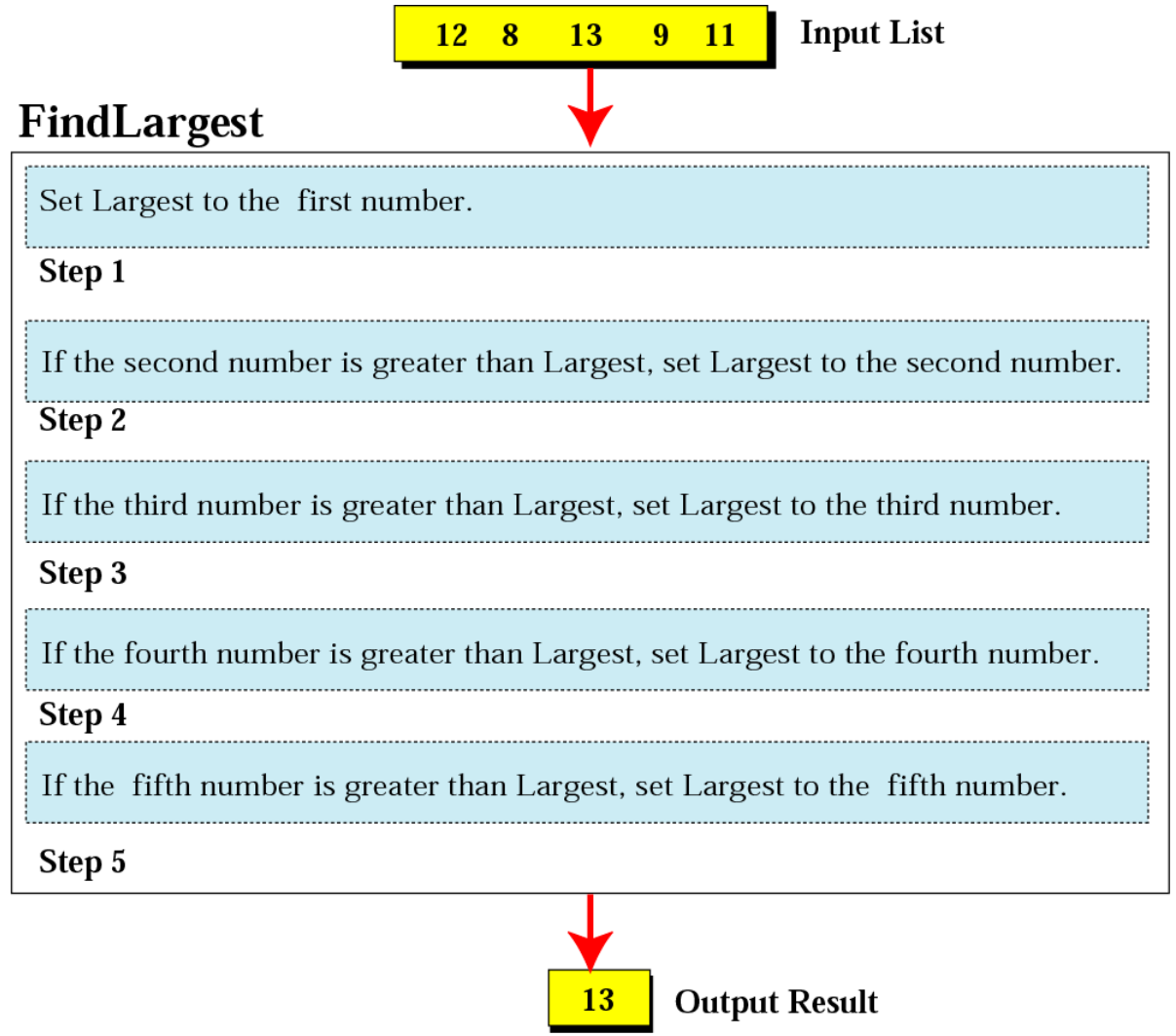
c. Repetition

How to write algorithm:

Find the largest integer among five integers !

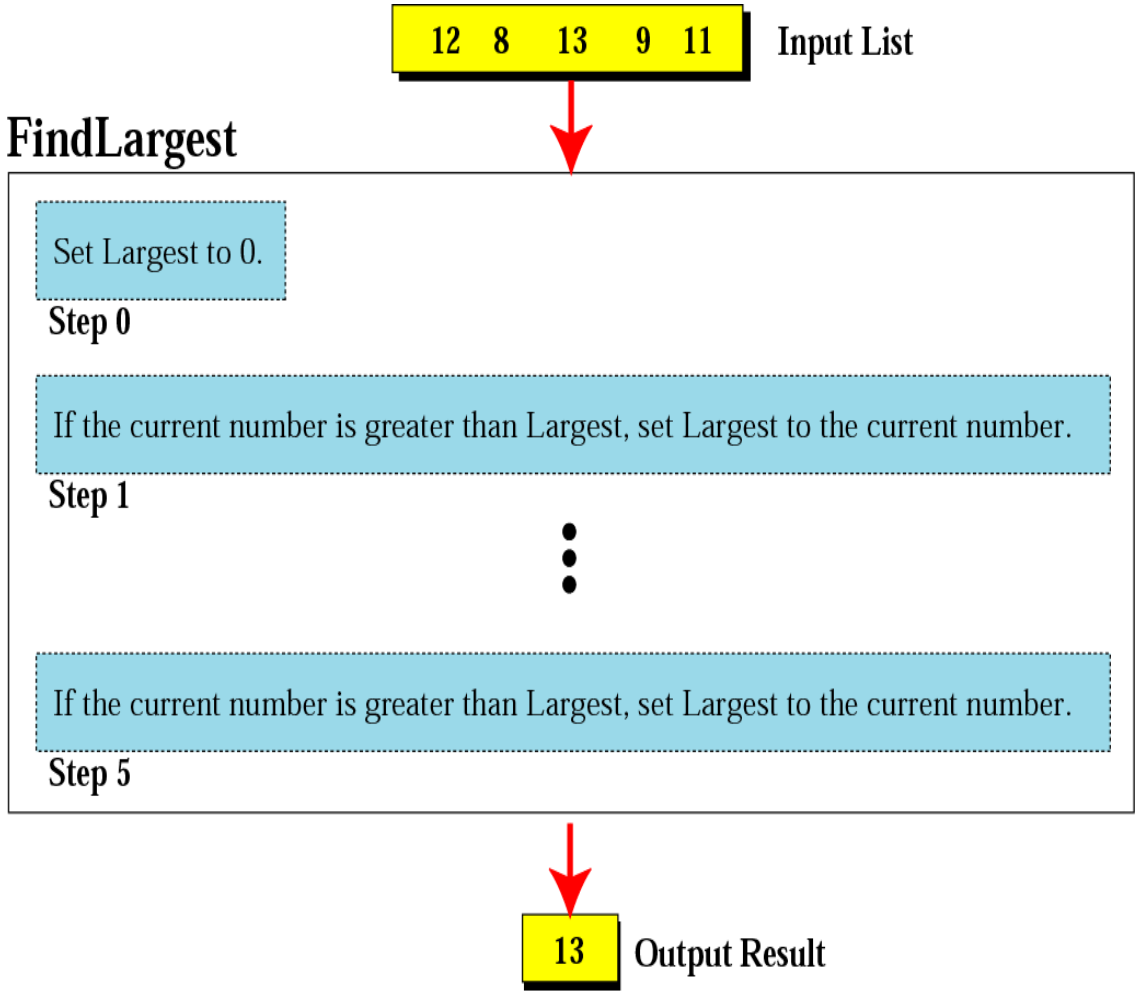


Define actions for FindLargest algorithm

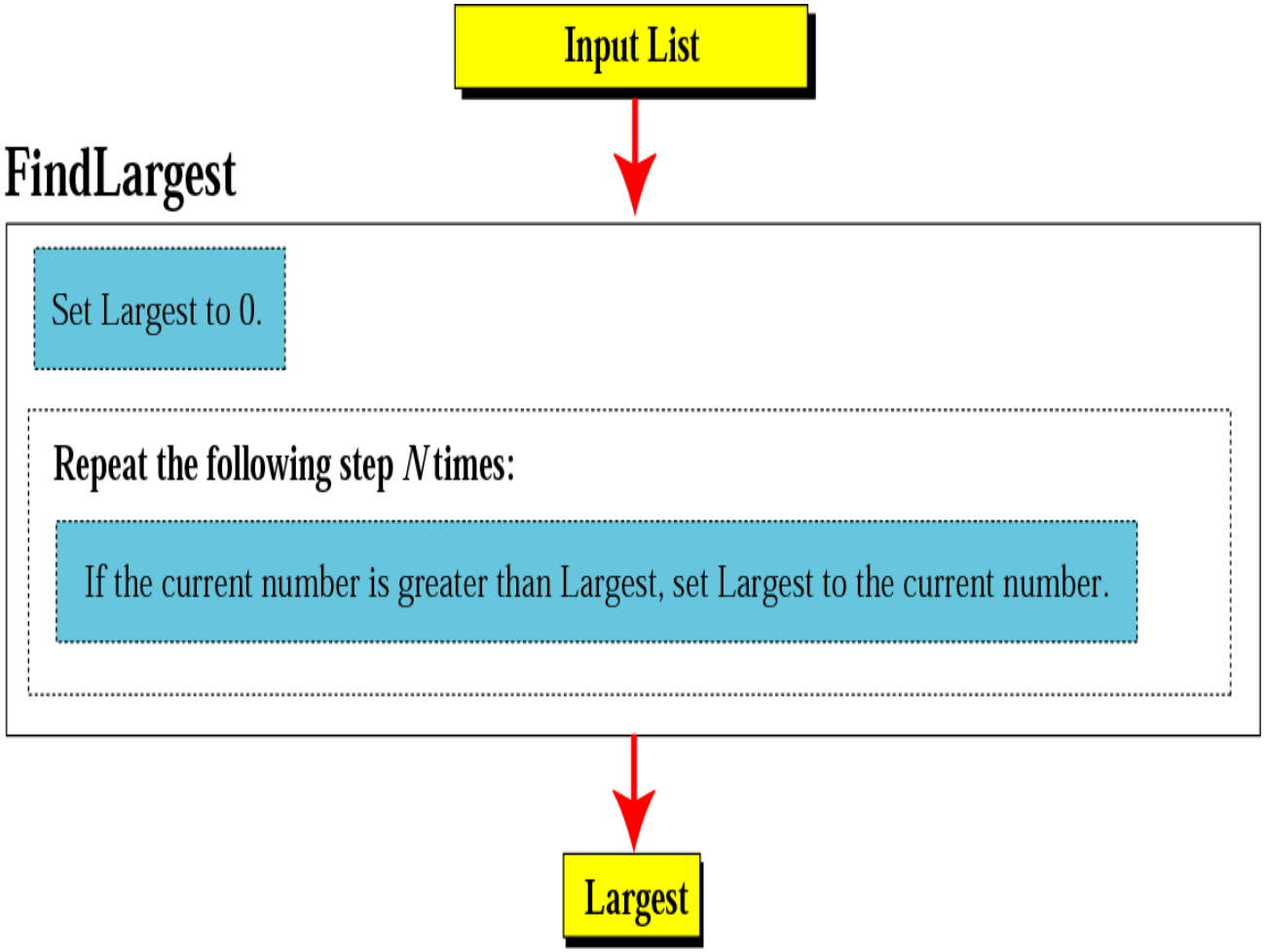


How to write algorithm:

Find the largest integer among five integers !



FindLargest refined



Generalization of FindLargest

How to write algorithm:

Find the largest integer among five integers !

Algorithm : FindLargest

Input: A list of positive integers

Output: largest positive integer

1. *Set Largest to 0*
2. *while (more integers)*
   - *2.1 if (the integer is greater than Largest)*
     - *then*
       - *2.1.1 Set largest to the value of the integer*
     - *End if*
   - *End while*
3. *Return Largest*

End

A Simple Algorithm to find smallest element in an array

Algorithm : FindSmallest

Input: a sequence of n numbers
T is an array of n elements T[1], T[2], ..., T[n]
Output: the smallest number among them
min = T[1]
for i = 2 to n do
{
if T[i] < min
min = T[i]
}
Output min
End

- Performance of this algorithm is a function of **n**

Example 1 :Write an algorithm that finds the average of two numbers.

Example 2 : Write an algorithm to change a numeric grade to a pass/fail grade.

Example 3 : Write an algorithm to change a numeric grade to a letter grade.

Solution 1 : **AverageOfTwo**

Input: Two numbers

1. Add the two numbers
2. Divide the result by 2
3. Return the result by step 2

End

Solution 2 : **Pass_failGrade**

Input: One number

1. if (the number is greater than or equal to 70)
   then
      1.1 Set the grade to “pass”
   else
      1.2 Set the grade to “nopass”
   End if
2. Return the grade

   End

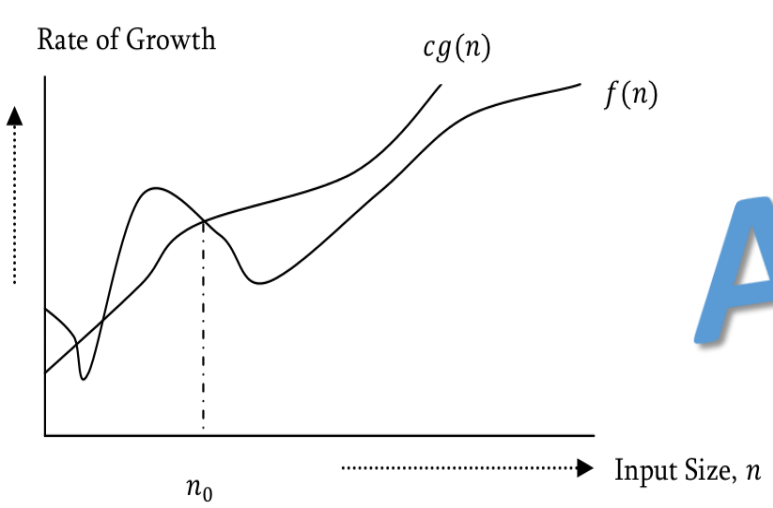
Solution 3 : **LetterGrade**

Input: One number

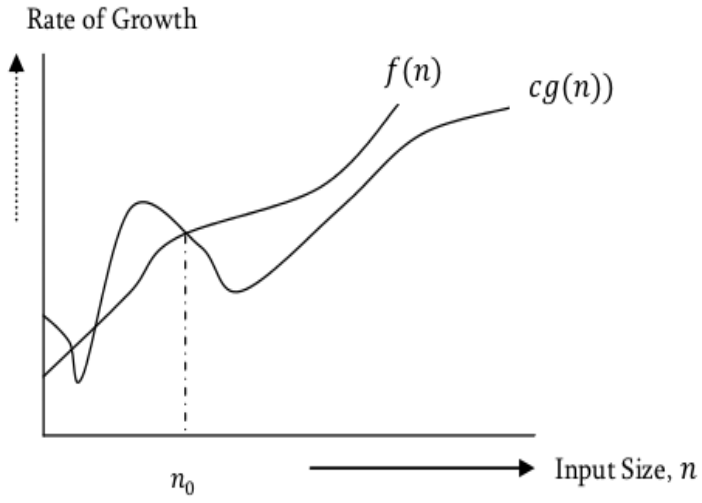
1. if (the number is between 90 and 100, inclusive)
   then
      1.1 Set the grade to “A”
   End if
2. if (the number is between 80 and 89, inclusive)
   then
      2.1 Set the grade to “B”
   End if
3. if (the number is between 70 and 79, inclusive)
   then
      3.1 Set the grade to “C”
   End if
4. if (the number is between 60 and 69, inclusive)
   then
      4.1 Set the grade to “D”
   End if

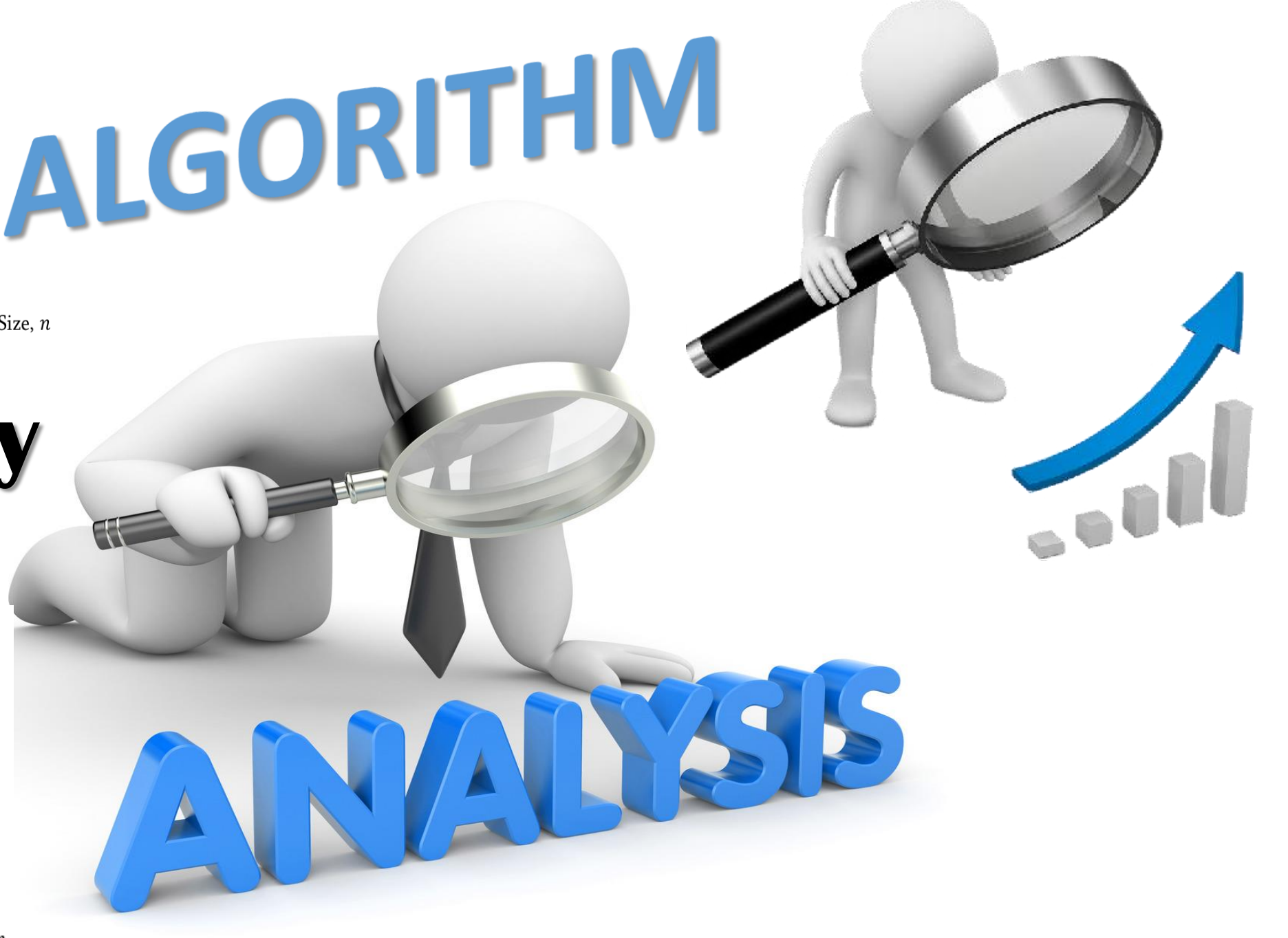
5. If (the number is less than 60)
   then
      5.1 Set the grade to “F”
   End if
6. Return the grade

End


Rate of Growth
$cg(n)$
$f(n)$
n_0
Input Size, n

Complexity


Rate of Growth
$f(n)$
$cg(n))$
n_0
Input Size, n


ALGORITHM
ANALYSIS

Why Analyse an Algorithm?

- To compare it with other algorithms for the same application.
- The efficiency of algorithm is decided by the care-full analysis of algorithm.
- The analysis of an algorithm can help us in understanding of solution in better way.
- To discover its characteristics in order to evaluate its suitability for various applications.
- Used for predicting the resources that the algorithm requires.
- Occasionally, resources such as memory, communication bandwidth or computer hardware are of primary concern.
- But most often it is computational time that we want to measure.
- Time & Space analysis are of crucial importance for performance measurement of a software module.

Different Approach of algorithm Analysis

- **To Compare Algorithms can we use**
  - Execution Time?
  - Number of Statement in the algorithm?
- Two broad category of Analysis:
  - Apriori
  - Posteriori
- Three types of approaches:
  - Empirical approach: gaining knowledge by means of direct and indirect observation or experience. *Is it feasible?*
  - Mathematical approach: determining the quantity of resources needed by each algorithm as a function of the size of instances mathematically. *Are the efforts put-in worth?*
  - Asymptotic Analysis: we evaluate the performance of an algorithm in terms of input size, where we calculate, how does the time (or space) taken by an algorithm increases with the input size(i.e., *f(n)*). *Way to Go*

Order of Growth

- Any algorithm is expected to work fast for any input size.
- For smaller input size any algorithm will work fine but for higher input size the execution time is much higher.
- By increasing the size of input size (n) we can analyze how well our algorithm works.
- Let input size, n=5 and we have to sort the list of elements for e.g. 5,29,10,15,2
- So for n=5 our algorithm will work fine but what if n=5000?
- So how the behaviour of algorithm changes with the no. of inputs will give the analysis of the algorithm and is called the **Order of Growth**.

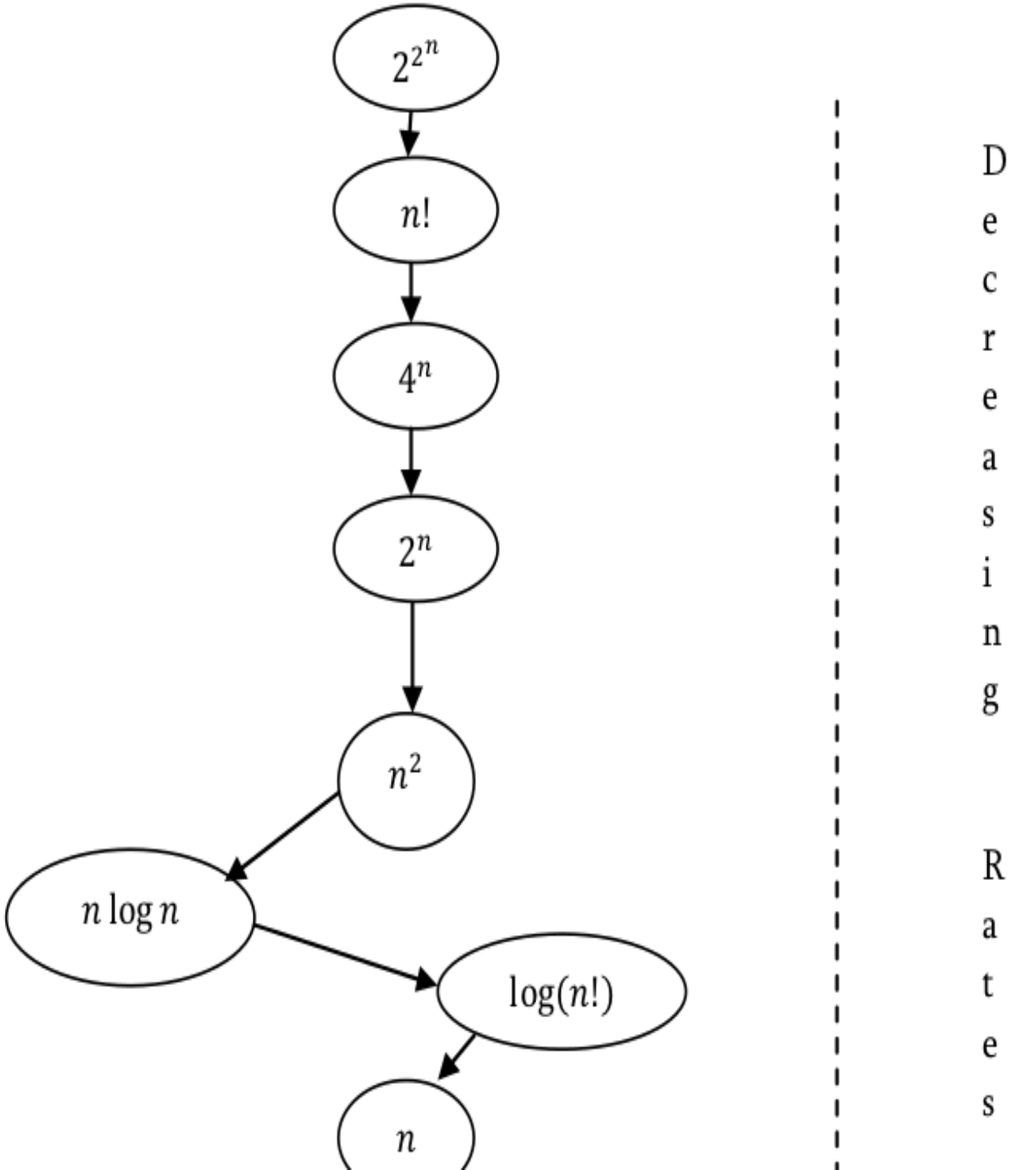
Algorithm falls under three types −

- **Best Case** − Minimum time required for program execution.
- **Average Case** − Average time required for program execution.
- **Worst Case** − Maximum time required for program execution.

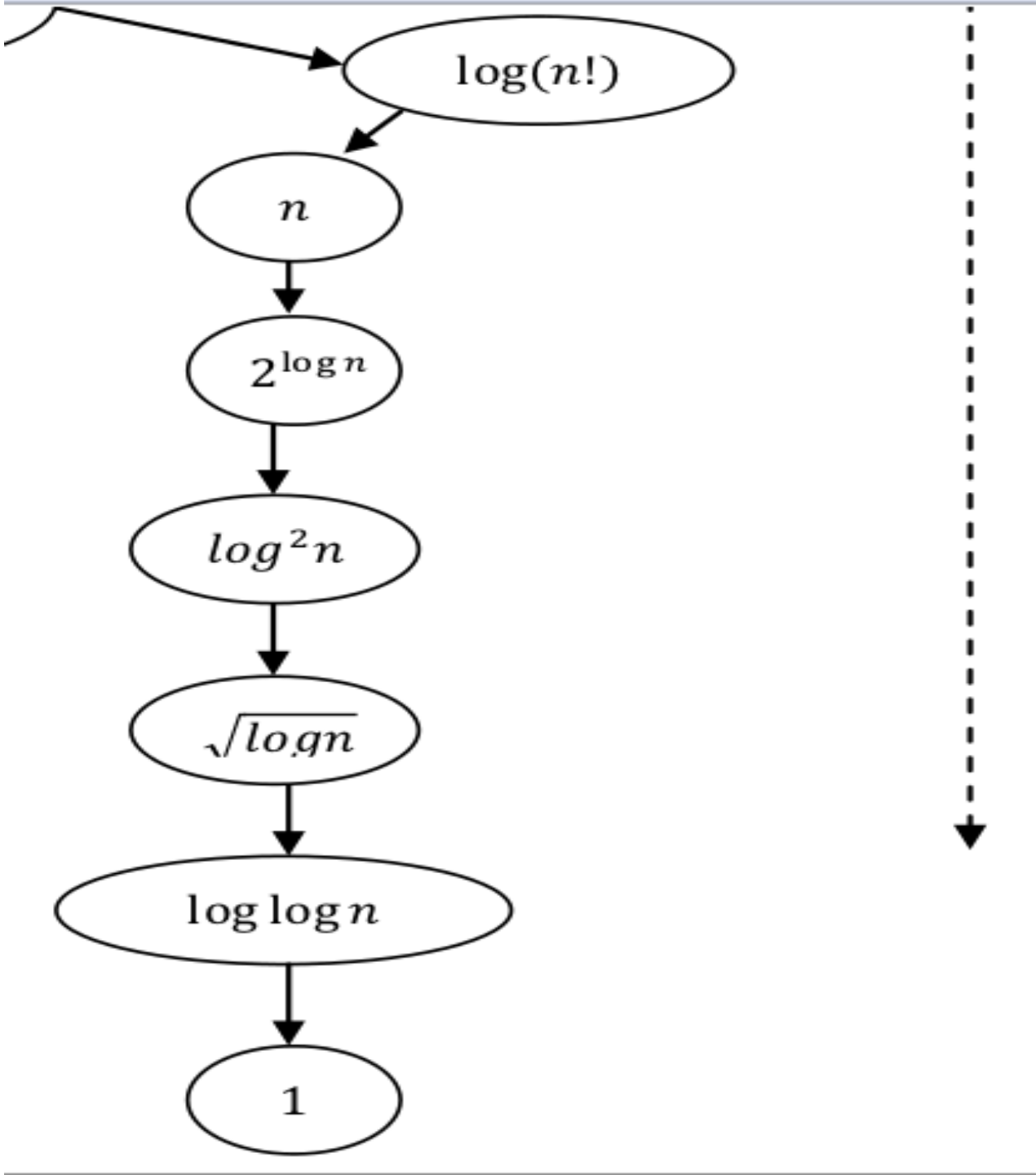
What is Rate of Growth?

- The rate at which the running time increases as a function of input is called **rate of growth**.

Time complexity	Name	Example
1	Constant	Adding an element to the front of a linked list
$logn$	Logarithmic	Finding an element in a sorted array
n	Linear	Finding an element in an unsorted array
$nlogn$	Linear Logarithmic	Sorting n items by 'divide-and-conquer'-Mergesort
n^2	Quadratic	Shortest path between two nodes in a graph
n^3	Cubic	Matrix Multiplication
2^n	Exponential	The Towers of Hanoi problem

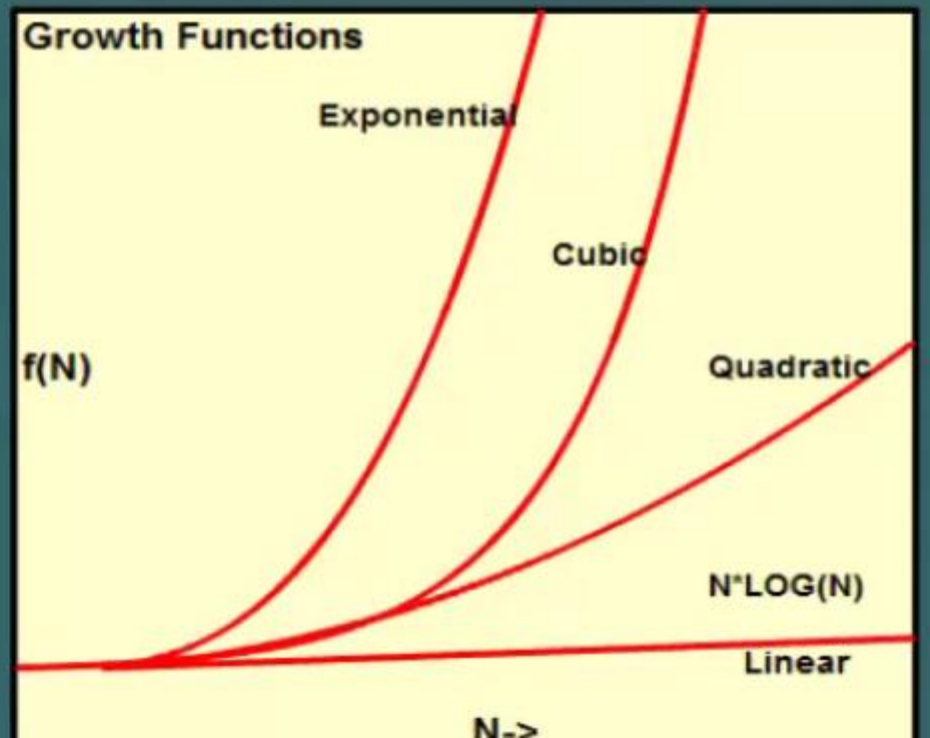


Decreasing Rates



Algorithm Growth Functions

Running time	Function	Performance
O(Log N)	Logarithm	Very good
O(N)	Linear	Good
O(N log N)	N-logarithm	Fair
$O(N^2)$	Quadratic	Acceptable
$O(N^3)$	Cubic	Poor
$O(2^N)$	Exponential	Bad



Types of Analysis

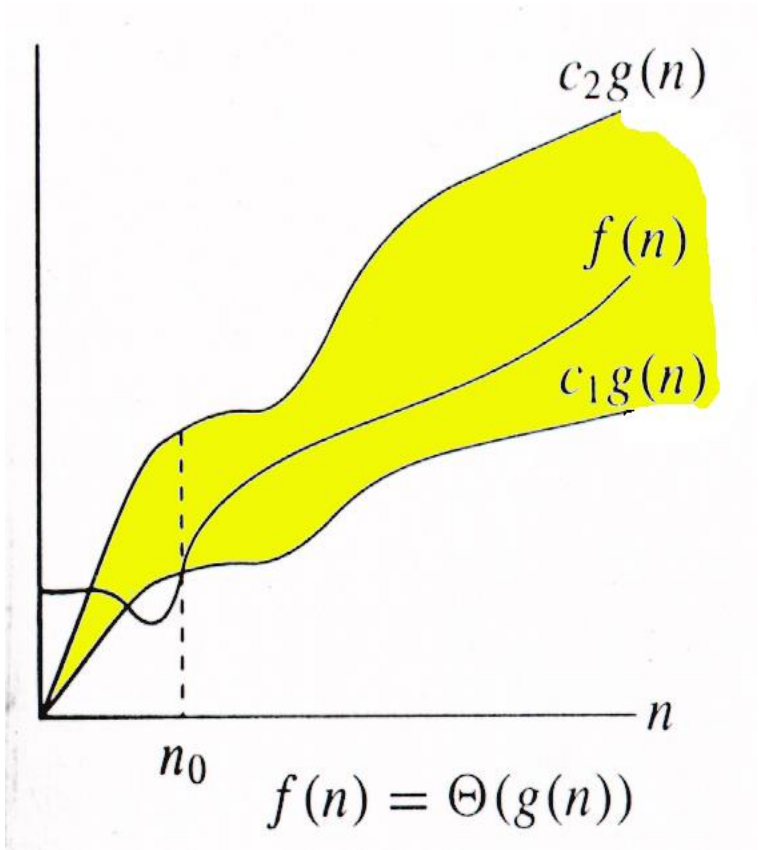
$$Lower\ Bound \leq Running\ Time \leq Upper\ Bound$$

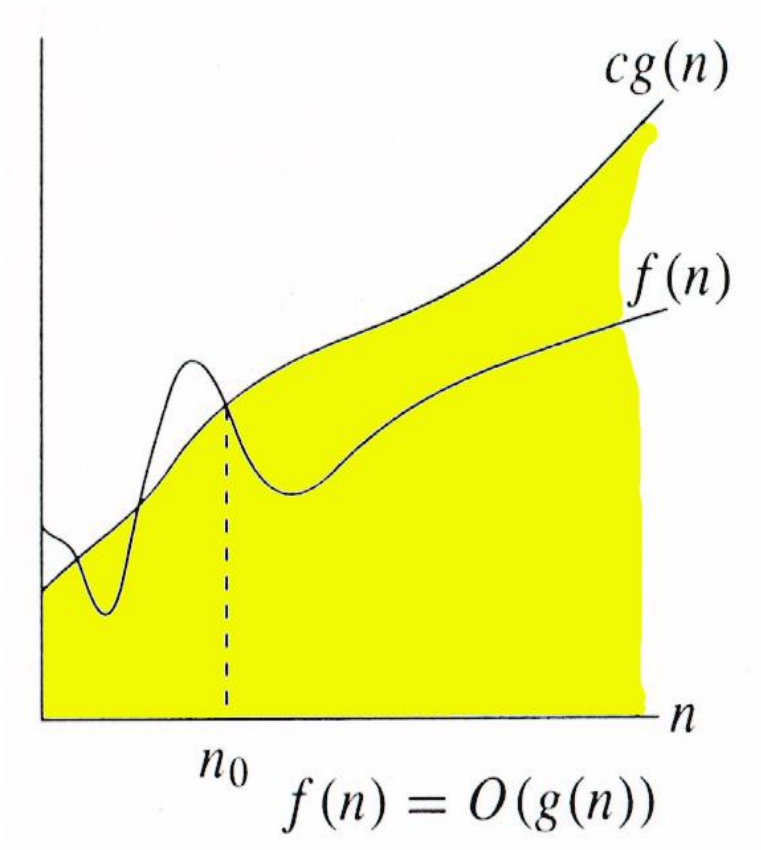
- Worst case
  - Provides an upper bound on running time
  - An absolute guarantee that the algorithm would not run longer, no matter what the inputs are
- Best case
  - Provides a lower bound on running time
  - Input is the one for which the algorithm runs the fastest
- Average case
  - Provides a prediction about the running time
  - Assumes that the input is random

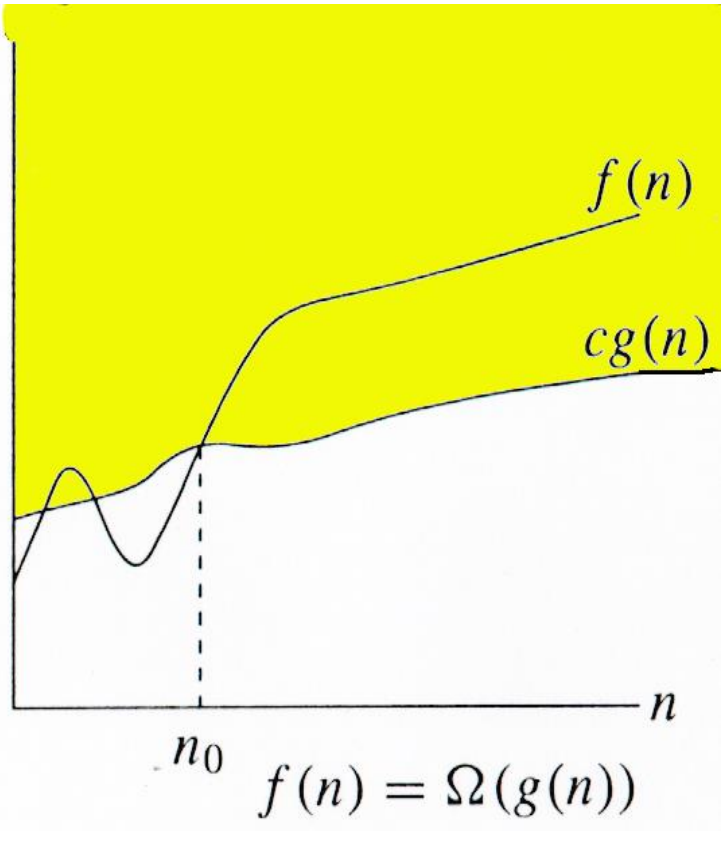
Asymptotic Analysis

- Asymptotic analysis of an algorithm refers to defining the mathematical bound of its run-time performance.
- The term **asymptotic** means approaching a value or curve arbitrarily.
- Asymptotic Notations
  - O Notation : "express the tight upper bound of an algorithm"
    - *f(n)=O(g(n)) implies: O(g(n)) = { f(n) : there exists positive constants c>0 and n_0 such that $f(n) \leq c.g(n)$ for all $n > n_0$. }*
  - Ω Notation : "express the lower bound of an algorithm"
    - *f(n)= Ω(g(n)) implies: Ω(g(n)) = { f(n) : there exists positive constants c>0 and n_0 such that $f(n) \geq g(n)$ for all $n > n_0$. }*
  - θ Notation : "express both the lower bound and the upper bound "
    - *f(n)= Θ(g(n)) implies: Θ(g(n)) = { f(n) : there exists positive constants c_1>0, c_2>0 and n_0 such that $c_1.g(n) \leq f(n) \leq c_2.g(n)$ for all $n > n_0$. }*
    - ***OR**, if & only if f(n)=O(g(n)) and f(n)= Ω(g(n)) for all $n > n_0$*

Relations Between Θ, *O*, Ω







Complexity

- Complexity of an algorithm is a measure of the amount of time and/or space required by an algorithm for an input of a given size (n).
- What effects run time of an algorithm?

(a) computer used, hardware platform

(b) representation of abstract data types (ADT's)

(c) efficiency of compiler

(d) competence of implementer (programming skills)

(e) complexity of underlying algorithm

(f) size of the input

Out of these above (e) and (f) are generally most significant

Complexity

- Time complexity
  - The amount of time that an algorithm needs to run to completion.
  - Time requirements can be defined as a numerical function T(n),
  - where T(n) can be measured as the number of steps, provided each step consumes constant time.
  - For example, addition of two n-bit integers takes n steps. Consequently, the total computational time is T(n) = c*n, where c is the time taken for the addition of two bits.
- Space complexity
  - The amount of memory an algorithm needs to run.
  - Space complexity S(P) of any algorithm P is S(P) = C + SP(I),
  - where C is the fixed part, which is space required to store certain data and variables, that are independent of the size of the problem. For example, simple variables and constants used, program size, etc.
  - S(I) is the variable part of the algorithm, which depends on instance characteristic I. The space required by variables, whose size depends on the size of the problem. For example, dynamic memory allocation, recursion stack space, etc.

Example : Complexity

Algorithm 1

	Cost
arr[0] = 0;	c_1
arr[1] = 0;	c_1
arr[2] = 0;	c_1
...	
arr[N-1] = 0;	c_1
----------	----------

$c_1+c_1+...+c_1 = c_1 \times N$

Algorithm 2

	Cost
for(i=0; i<N; i++)	c_2
arr[i] = 0;	c_1
	--

$(N+1) \times c_2 + N \times c_1 = (c_2 + c_1) \times N + c_2$

- Both algorithms are of the same order: *O(N)*

Algorithm 3

	Cost
sum = 0;	c_1
for(i=0; i<N; i++)	c_2
for(j=0; j<N; j++)	c_2
sum += arr[i][j];	c_3

$c_1 + c_2 \times (N+1) + c_2 \times N \times (N+1) + c_3 \times N^2 = O(N^2)$

Algorithm 4 (Calculate time complexity?)

```
1.  j = n
2.  while (j>=1){
3.  for i = 1 to j
4.  x = x + 1
5.  j = n / 2
}
```

Big–O Examples

(Drop Lower order term & Constant Factor to Calculate Big –O)

Example–1 Find upper bound for $f(n) = 3n + 8$

Solution: $3n + 8 \leq 4n$, for all $n \geq 1$

$\therefore 3n + 8 = O(n)$ with c = 4 and $n_0 = 8$

Example–2 Find upper bound for $f(n) = n^2 + 1$

Solution: $n^2 + 1 \leq 2n^2$, for all $n \geq 1$

$\therefore n^2 + 1 = O(n^2)$ with $c = 2$ and $n_0 = 1$

Example–3 Find upper bound for $f(n) = n^4 + 100n^2 + 50$

Solution: $n^4 + 100n^2 + 50 \leq 2n^4$, for all $n \geq 1$

$\therefore n^4 + 100n^2 + 50 = O(n^4)$ with $c = 2$ and $n_0 = 100$

Example–4 Find upper bound for $f(n) = 2n^3 - 2n^2$

Solution: $2n^3 - 2n^2 \leq 2n^3$, for all $n \geq 1$

$\therefore 2n^3 - 2n^2 = O(2n^3)$ with $c = 2$ and $n_0 = 1$

Example–5 Find upper bound for $f(n) = n$

Solution: $n \leq n^2$, for all $n \geq 1$

$\therefore n = O(n^2)$ with $c = 1$ and $n_0 = 1$

Example–6 Find upper bound for $f(n) = 410$

Solution: $410 \leq 410$, for all $n \geq 1$

$\therefore 100 = O(1)$ with $c = 1$ and $n_0 = 1$

1) O(1): Time complexity of a function (or set of statements) is considered as O(1) if it doesn't contain loop, recursion and call to any other non-constant time function.

```
// set of non-recursive and non-loop statements
```

For example swap() function has O(1) time complexity.

A loop or recursion that runs a constant number of times is also considered as O(1). For example the following loop is O(1).

```
// Here c is a constant
for (int i = 1; i <= c; i++) {
     // some O(1) expressions
}
```

2) O(n): Time Complexity of a loop is considered as O(n) if the loop variables is incremented / decremented by a constant amount. For example following functions have O(n) time complexity.

```
// Here c is a positive integer constant
for (int i = 1; i <= n; i += c) {
     // some O(1) expressions
}

for (int i = n; i > 0; i -= c) {
     // some O(1) expressions
}
```

3) $O(n^c)$: Time complexity of nested loops is equal to the number of times the innermost statement is executed. For example the following sample loops have $O(n^2)$ time complexity

```
for (int i = 1; i <=n; i += c) {
    for (int j = 1; j <=n; j += c) {
       // some O(1) expressions
    }
}

for (int i = n; i > 0; i += c) {
    for (int j = i+1; j <=n; j += c) {
       // some O(1) expressions
}
```

For example Selection sort and Insertion Sort have $O(n^2)$ time complexity.

4) O(Logn) Time Complexity of a loop is considered as O(Logn) if the loop variables is divided / multiplied by a constant amount.

```
for (int i = 1; i <=n; i *= c) {
    // some O(1) expressions
}
for (int i = n; i > 0; i /= c) {
    // some O(1) expressions
}
```

For example Binary Search(refer iterative implementation) has O(Logn) time complexity.

5) O(LogLogn) Time Complexity of a loop is considered as O(LogLogn) if the loop variables is reduced / increased exponentially by a constant amount.

```
// Here c is a constant greater than 1
for (int i = 2; i <=n; i = pow(i, c)) {
    // some O(1) expressions
}
//Here fun is sqrt or cuberoot or any other constant root
for (int i = n; i > 0; i = fun(i)) {
    // some O(1) expressions
}
```

How to combine time complexities of consecutive loops?

When there are consecutive loops, we calculate time complexity as sum of time complexities of individual loops.

```
for (int i = 1; i <=m; i += c) {
     // some O(1) expressions
}
for (int i = 1; i <=n; i += c) {
     // some O(1) expressions
}
Time complexity of above code is O(m) + O(n) which is O(m+n)
If m == n, the time complexity becomes O(2n) which is O(n).
```

How to calculate time complexity when there are many if, else statements inside loops?

As discussed here, worst case time complexity is the most useful among best, average and worst. Therefore we need to consider worst case. We evaluate the situation when values in if-else conditions cause maximum number of statements to be executed.

For example consider the linear search function where we consider the case when element is present at the end or not present at all.

When the code is too complex to consider all if-else cases, we can get an upper bound by ignoring if else and other complex control statements.

Commonly used Logarithms and Summations

Logarithms

$$log\ x^y = y\ log\ x$$
$$log\ n = log_e^n$$
$$log\ xy = logx + logy$$
$$log^k n = (logn)^k$$
$$log\ logn = log(logn)$$
$$log\ \frac{x}{y} = logx - logy$$
$$a^{log_b^x} = x^{log_b^a}\ log_b^x = \frac{log_a^x}{log_a^b}$$

Arithmetic series

$$\sum_{K=1}^{n} k = 1 + 2 + \cdots + n = \frac{n(n+1)}{2}$$

Geometric series

$$\sum_{k=1}^{n} x^k = 1 + x + x^2 \ldots + x^n = \frac{x^{n+1} - 1}{x - 1} (x \neq 1)$$

Harmonic series

$$\sum_{k=1}^{n} \frac{1}{k} = 1 + \frac{1}{2} + \ \ldots + \ \frac{1}{n} \approx log\ n$$

Other important formulae

$$\sum_{k=1}^{n} log\ k \approx nlogn$$

Properties of Notations

- Transitivity: $f(n) = \Theta(g(n))$ and $g(n) = \Theta(h(n)) \Rightarrow f(n) = \Theta(h(n))$. Valid for O and Ω also.
- Reflexivity: $f(n) = \Theta(f(n))$. Valid for O and Ω also.
- Symmetry: $f(n) = \Theta(g(n))$ if and only if g(n) $= \Theta$(f(n)).
- Transpose symmetry: $f(n) = O(g(n))$ if and only if $g(n) = \Omega(f(n))$.

Guidelines for Asymptotic Analysis?

There are some general rules to help us in determining the running time of an algorithm. Below are few of them.

1) **Loops**: The running time of a loop is, at most, the running time of the statements inside the loop (including tests) multiplied by the number of iterations.

```
// executes n times
for (i=1; i<=n; i++)
{
    m = m + 2; // constant time, c
}
```

2) **Nested loops:** Analyze from inside out. Total running time is the product of the sizes of all the loops.

```
//outer loop executed n times
for (i=1; i<=n; i++)
{
    // inner loop executed n times
    for (j=1; j<=n; j++)
    {
            k = k+1; //constant time
    }
}
```

Total time = $c \times n \times n = cn^2 = O(n^2)$.

3) **Consecutive statements:** Add the time complexities of each statement.

```
x = x +1; //constant time

// executed n times
for (i=1; i<=n; i++)
{
     m = m + 2; //constant time
}

//outer loop executed n times
for (i=1; i<=n; i++)
{
     //inner loop executed n times
     for (j=1; j<=n; j++)
     {
               k = k+1; //constant time
     }
}
```

Total time = $c_0 + c_1 n + c_2 n^2 = O(n^2)$.

4) **If-then-else statements:** Worst-case running time: the test, plus *either* the *then* part *or* the *else* part (whichever is the larger).

```
    //test: constant
    if (length ( ) != otherStack. length ( ) )
    {
        return false; //then part: constant
}
else
{
    // else part: (constant + constant) * n
    for (int n = 0; n < length( ); n++)
    {
            // another if : constant + constant (no else part)
            if (!list[n].equals(otherStack.list[n]))
            //constant
            return false;
    }
}
```

Total time = $c_0 + c_1 + (c_2 + c_3) * n = O(n)$.

5) **Logarithmic complexity:** An algorithm is $O(logn)$ if it takes a constant time to cut the problem size by a fraction (usually by ½).

As an example let us consider the following program:

```
for (i=1; i<=n;)
{
    i = i*2;
}
```

If we observe carefully, the value of i is doubling every time. Initially $i = 1$, in next step $i = 2$, and in subsequent steps $i = 4, 8$ and so on.

Let us assume that the loop is executing some k times. That means at k^{th} step $2^i = n$ and we come out of loop. So, if we take logarithm on both sides,

$$log(2^i) = logn$$
$$ilog2 = logn$$
$$i = logn \text{ //if we assume base-2}$$

So the total time $= O(logn)$.

Complexities of various Data Structures

Ref: Eric- http://bigoche

Data Structure	Time Complexity								Space Complexity
	Average				Worst				Worst
	Access	Search	Insertion	Deletion	Access	Search	Insertion	Deletion	
Array	O(1)	O(n)	O(n)	O(n)	O(1)	O(n)	O(n)	O(n)	O(n)
Stack	O(n)	O(n)	O(1)	O(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Singly-Linked List	O(n)	O(n)	O(1)	O(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Doubly-Linked List	O(n)	O(n)	O(1)	O(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Skip List	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)	O(n)	O(n)	O(n)	O(n log(n))
Hash Table	-	O(1)	O(1)	O(1)	-	O(n)	O(n)	O(n)	O(n)
Binary Search Tree	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)	O(n)	O(n)	O(n)	O(n)
Cartesian Tree	-	O(log(n))	O(log(n))	O(log(n))	-	O(n)	O(n)	O(n)	O(n)
B-Tree	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)
Red-Black Tree	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)
Splay Tree	-	O(log(n))	O(log(n))	O(log(n))	-	O(log(n))	O(log(n))	O(log(n))	O(n)
AVL Tree	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)

Algorithm	Time Complexity			Space Complexity
	Best	Average	Worst	Worst
Quicksort	O(n log(n))	O(n log(n))	O(n^2)	O(log(n))
Mergesort	O(n log(n))	O(n log(n))	O(n log(n))	O(n)
Timsort	O(n)	O(n log(n))	O(n log(n))	O(n)
Heapsort	O(n log(n))	O(n log(n))	O(n log(n))	O(1)
Bubble Sort	O(n)	O(n^2)	O(n^2)	O(1)
Insertion Sort	O(n)	O(n^2)	O(n^2)	O(1)
Selection Sort	O(n^2)	O(n^2)	O(n^2)	O(1)
Shell Sort	O(n)	O((nlog(n))^2)	O((nlog(n))^2)	O(1)
Bucket Sort	O(n+k)	O(n+k)	O(n^2)	O(n)
Radix Sort	O(nk)	O(nk)	O(nk)	O(n+k)

Problem-23 What is the running time of the following function (specified as a function of the input value *n*)

```
void Function(int n)
{
        int i=1 ;
        int s=1 ;
        while( s <= n)
        {
                i++ ;
                s= s+i ;
                print(\*");
        }
}
```

Solution: Consider the comments in below function:

```
void Function (int n)
{
        int i=1 ;
        int s=1 ;
        // s is increasing not at rate 1 but i
        while( s <= n)
        {
                i++ ;
                s= s+i ;
                print("*");
        }
}
```

We can define the terms '*s*' according to the relation $s_i = s_{i-1} + i$. The value of '*i*' increases by one for each iteration. So the value contained in '*s*' at the i^{th} iteration is the sum of the first '*i*' positive integers. If k is the total number of iterations taken by the program, then the while loop terminates once.

$$1 + 2 + \ldots + k = \frac{k(k+1)}{2} > n \Longrightarrow k = O(\sqrt{n}).$$

Problem-24 Find the complexity of the function given below.

```
void Function(int n)
{
        int i, count =0;;
        for(i=1; i*i<=n; i++)
                count++;
}
```

In the above function the loop will end, if $i^2 \leq n \Rightarrow T(n) = O(\sqrt{n})$. The reasoning is same as that of Problem-23.

Problem-25 What is the complexity of the below program:

```
void function(int n)
{
        int i, j, k , count =0;
        for(i=n/2; i<=n; i++)
                for(j=1; j + n/2<=n; j= j++)
                        for(k=1; k<=n; k= k * 2)
                                count++;
}
```

Solution: Consider the comments in the following function.

```
void function(int n)
{
        int i, j, k , count =0;

          //outer loop execute n/2 times
          for(i=n/2; i<=n; i++)
                  //Middle loop executes n/2 times
                  for(j=1; j + n/2<=n; j= j++)
                          //outer loop execute logn times
                          for(k=1; k<=n; k= k * 2)
                                  count++;
 }
```

The complexity of the above function is $O(n^2 \log n)$.

Problem-26 What is the complexity of the below program:

```
void function(int n)
{
        int i, j, k , count =0;
        for(i=n/2; i<=n; i++)
                for(j=1; j<=n; j= 2 * j)
                        for(k=1; k<=n; k= k * 2)
                                count++;
}
```

Solution: Consider the comments in the following function.

```
void function(int n)
{
        int i, j, k , count =0;
        //outer loop execute n/2 times
        for(i=n/2; i<=n; i++)
                //Middle loop executes logn times
                for(j=1; j<=n; j= 2 * j)
                        //outer loop execute logn times
                        for(k=1; k<=n; k= k*2)
                                count++;
}
```

The complexity of the above function is $O(n\log^2 n)$.

Problem-27 Find the complexity of the below program.

```
function( int n )
{
    if (n == 1) return;
    for( i = 1 ; i <= n ; i + + )
    {
            for( j= 1 ; j <= n ; j + + )
            {
                    print("*" ) ;

                        break;
                }
        }
}
```

Solution: Consider the comments in the following function.

```
function( int n )
{
        //constant time
        if ( n == 1 ) return;
        //outer loop execute n times
        for( i = 1 ; i <= n ; i + + )
        {
                // inner loop executes only time due to break statement.
                for( j= 1 ; j <= n ; j + + )
                {
                        print("*" ) ;
                        break;
                }
        }
}
```

The complexity of the above function is $O(n)$. Even though the inner loop is bounded by n, but due to the break statement it is executing only once.

Problem-38 What is the running time of the following recursive function (specified as a function of the input value n)? First write a recurrence formula, and show its solution using induction.

```
function(int n)
{
        if (n <= 1)
                return;
        for (i=1 ; i <= 3 ; i++ )
                function (n – 1).
}
```

Solution: Consider the comments in below function:

```
function (int n)
{
        //constant time
        if (n <= 1)
                return;
        //this loop executes 3 times with recursive call of n-1 value
        for (i=1 ; i <= 3 ; i++ )
                function (n – 1).
}
```

The *if statement* requires constant time ($O(1)$). With the *for loop*, we neglect the loop overhead and only count the three times that the function is called recursively. This implies a time complexity recurrence:

$$T(n) = c, if\ n \leq 1;$$
$$= c + 3T(n-1), if\ n > 1.$$

Now we use repeated substitution to guess at the solution when we substitute k times:

$$T(n) = c + 3T(n-1)$$

Using the *Subtraction and Conquer* master theorem, we get $T(n) = \Theta(3^n)$.

Problem-46 Running time of following program?

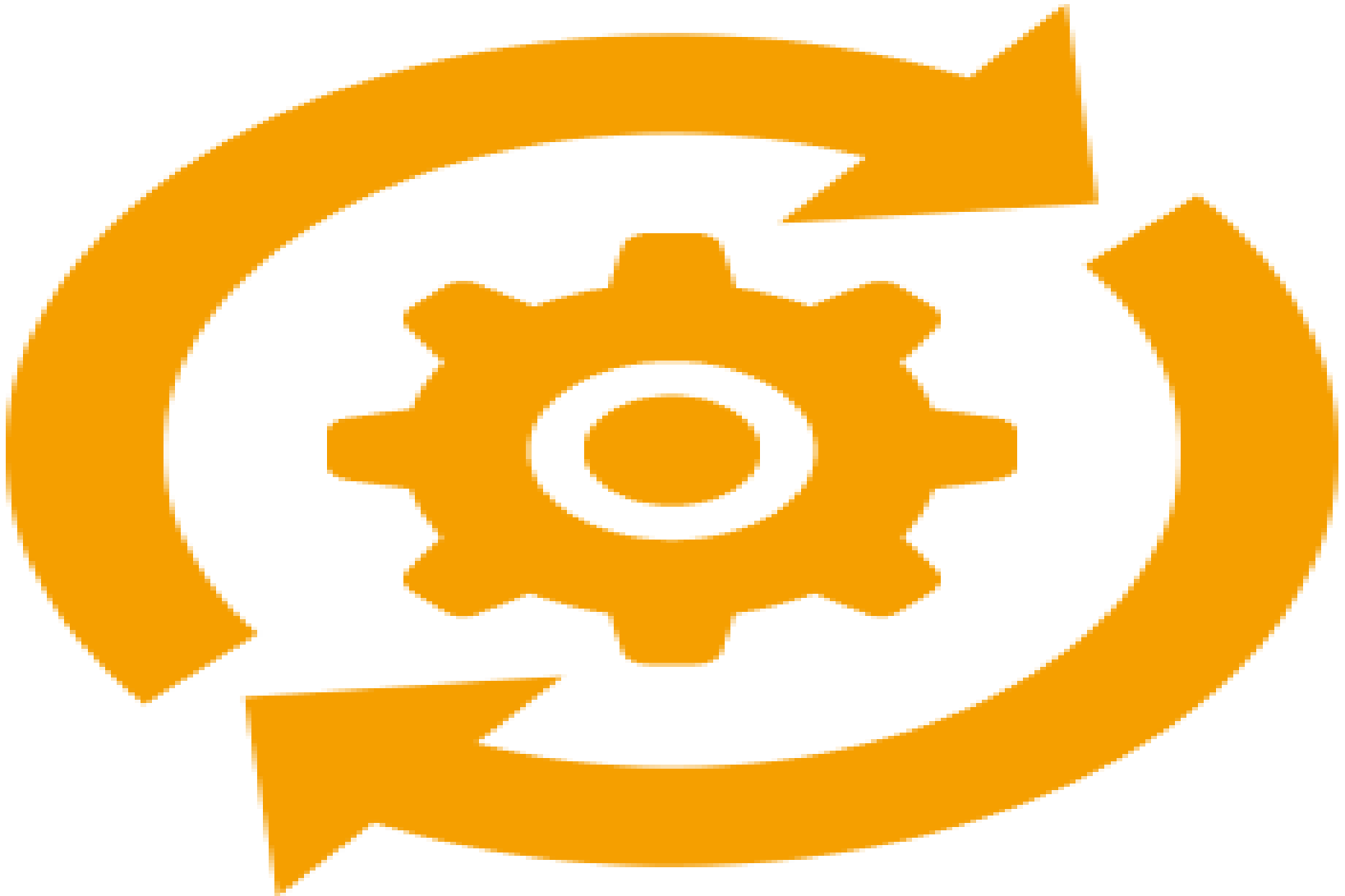
```
function(int n)
{
        for( i = 1 ; i <= n ; i + + )
                for( j = 1 ; j <= n ; j * = 2 )
                        print( "*" ) ;
}
```

Solution: Consider the comments in below function:

```
function(int n)
{
        // this loops executes n times
        for( i = 1 ; i <= n ; i + + )
                // this loops executes logn times from our logarithms
                //guideline
                for( j = 1 ; j <= n ; j * = 2 )
                        print( "*" ) ;
}
```

Complexity of above program is : $O(nlogn)$.

Recurrence Relation

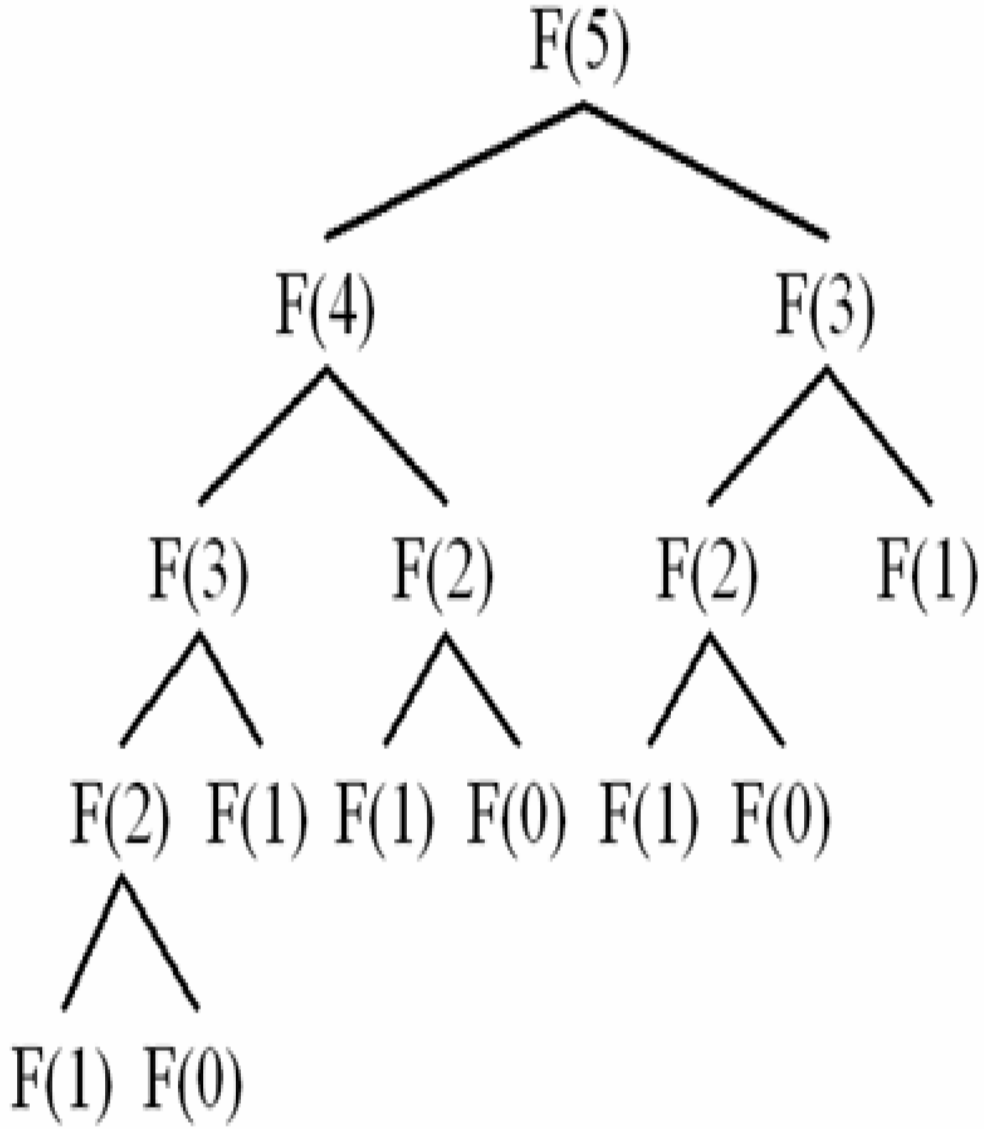


Recurrence Relation

- A **recurrence relation** is an equation that defines a sequence based on a rule that gives the next term as a function of the previous terms.

- In mathematics, a **recurrence relation** is an equation that recursively defines a sequence or multidimensional array of values, once one or more initial terms are given: each further term of the sequence or array is defined as a function of the preceding terms.

- A recurrence relation is an equation that recursively defines a sequence where the next term is a function of the previous terms .



Will Continue in next Chapter : Divide and conquer Strategy