DESIGN AND ANALYSIS OF ALGORITHMS -22AIM43

AKSHATHA P S

WHY ALGORITHMS?

MODULE-1	**INTRODUCTION**	**22AIM43.1**	**8 Hours**
Introduction to Algorithms, Role of algorithms in computing, Time and Space Complexity of Algorithms, Asymptotic notations, worst-case, Average-case and Best-case analysis, Analysis Framework- Empirical analysis- Mathematical analysis for Recursive and Non-recursive algorithms.			
Case Study	Illustrate real-world applications of algorithms and growth functions.		
Text Book	Text Book 1:1.1,1.2,1.3		
MODULE-2	**DIVIDE AND CONQUER**	**22AIM43.2**	**8 Hours**
Divide and Conquer Methodology: Binary search, Merge sort, Quick sort, Finding the maximum and minimum, Strassen's matrix, advantages and disadvantages of divide and conquer.			
Case Study	Compare and contrast the time complexity and suitability of the bubble sort, merge sort, and quicksort algorithms. Provide scenarios where one might be preferred over the others.		
Text Book	Text Book 1: 2.1,2.2		
MODULE-3	**GREEDY METHOD AND DYNAMIC PROGRAMMING**	**22AIM43.3**	**8 Hours**
Greeedy method: Introduction, Job scheduling problem, Minimum Spanning tree algorithms – Kruskals & Prims. Optimal Tree Problem: Huffman Trees. **DYNAMIC PROGRAMMING:** Introduction, Knapsack problems, Travelling Salesman problem. Transitive closure - Warshall's and Floyds algorithm.			

MODULE-4	**DECREASE & CONQUER, TRANSFORM & CONQUER**	**22AIM43.4**	**8 Hours**
Decrease &conquer: Introduction – Decrease by constant, decrease by constant factor-Fake Coin Problem-Russian Peasant Multiplication, variable size decrease. **Transform & conquer**: Introduction, Balanced Search trees – AVL trees & 2-3 trees, Red Black Trees			
Text Book	Text Book 1: 5.1,5.2,5.3,5.4,5.5,5.6		
MODULE-5	**BACKTRACKING, BRANCH AND BOUND**	**22AIM43.5,22AIM43.6**	**8 Hours**
Backtracking: Introduction, N Queens problem, subset sum problem, **Branch and Bound:** Introduction, Travelling Salesman problem, Knapsack problem, Assignment problem, NP-Hard and NP-Complete problems: Basic concepts, non-deterministic algorithms.			

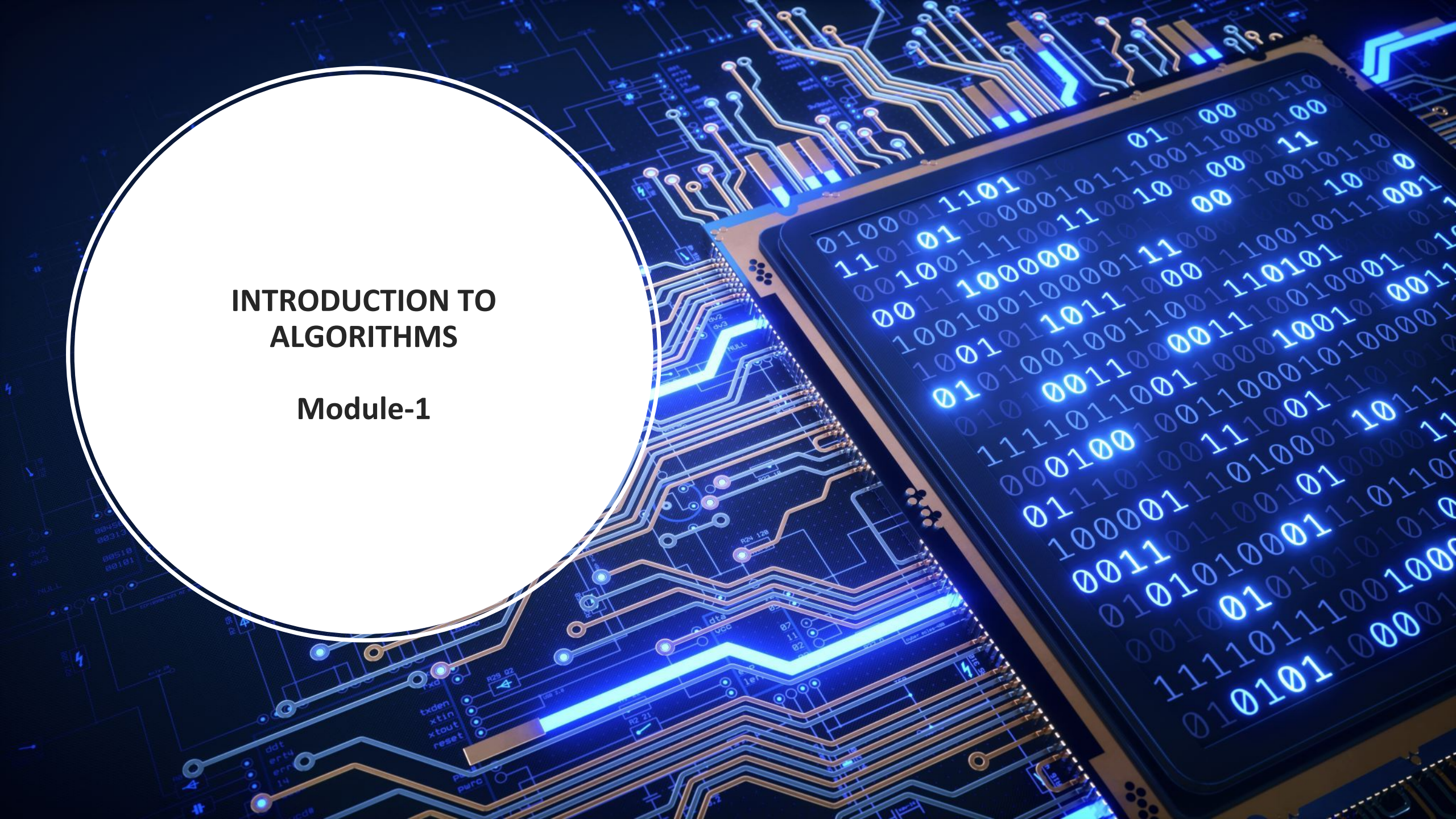
Suggested Learning Resources:

Text Books:

1. Anany Levitin,"Introduction to the Design &Analysis of Algorithms",3rd Edition, PEARSON Education, 2012.

ReferenceBook:

1.Thomas H Cormen, Charles E Leiserson, Ronald R Rivest & Clifford Stein, "Introduction to Algorithms ", THIRD Edition, Eastern Economy Edition

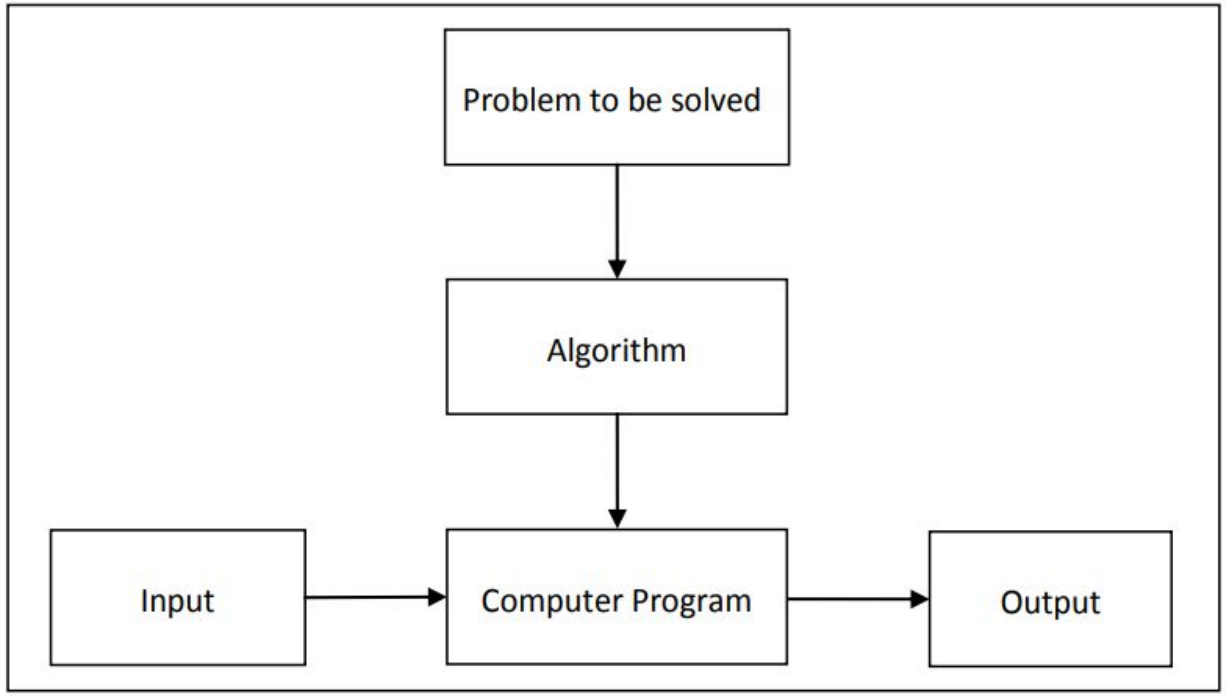


INTRODUCTION TO ALGORITHMS

Module-1

What is an Algorithm?

- An algorithm is a sequence of unambiguous instructions for solving a problem, i.e., for obtaining a required output for any legitimate input in a finite amount of time.



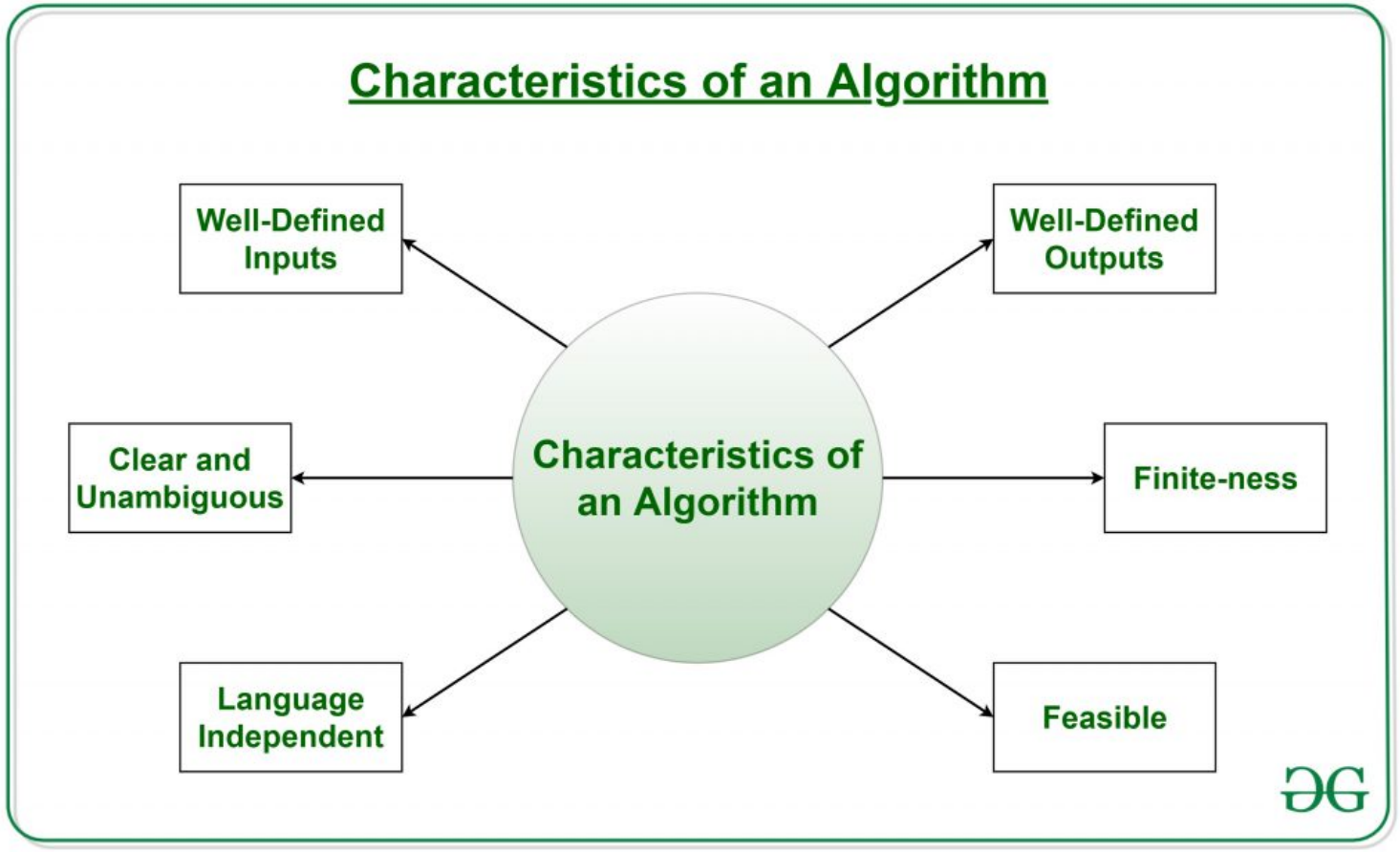
Notion of algorithm

Notion of algorithm

The notion of the algorithm illustrates some important points:

- The non-ambiguity requirement for each step of an algorithm cannot be compromised.
- The range of inputs for an algorithm work has to be specified carefully.
- The same algorithm can be represented in several different ways.
- There may exist several algorithms for solving the same problem.
- Algorithms for the same problem can be based on very different ideas and can solve the problem at dramatically different speeds.

Characteristics of an algorithm



Characteristics of an algorithm

- **Clear and Unambiguous**: The algorithm should be unambiguous. Each of its steps should be clear in all aspects and must lead to only one meaning.
- **Well-Defined Inputs**: If an algorithm says to take inputs, it should be well-defined inputs. It may or may not take input.
- **Well-Defined Outputs:** The algorithm must clearly define what output will be yielded and it should be well-defined as well. It should produce at least 1 output.
- **Finite-ness:** The algorithm must be finite, i.e. it should terminate after a finite time.
- **Feasible:** The algorithm must be simple, generic, and practical, such that it can be executed with the available resources. It must not contain some future technology or anything.
- **Language Independent:** The Algorithm designed must be language-independent, i.e. it must be just plain instructions that can be implemented in any language, and yet the output will be the same, as expected.
- **Definiteness:** All instructions in an algorithm must be unambiguous, precise, and easy to interpret. By referring to any of the instructions in an algorithm one can clearly understand what is to be done. Every fundamental operator in instruction must be defined without any ambiguity.
- **Effectiveness:** An algorithm must be developed by using very basic, simple, and feasible operations so that one can trace it out by using just paper and pencil.

Steps for writing an algorithm:

1. An algorithm is a procedure. It has two parts; the first part is **head** and the second part is **body**.
2. The Head section consists of keyword **Algorithm** and Name of the algorithm with parameter list. E.g. Algorithm name1(p1, p2,…,p3)

   The head section also has the following:

   //Problem Description:

   //Input:

   //Output:
3. In the body of an algorithm various programming constructs like **if, for, while** and some statements like assignments are used.
4. The compound statements may be enclosed with { and } brackets. **if**, **for**, **while** can be closed by **endif**, **endfor**, **endwhile** respectively. Proper indention is must for block.
5. Comments are written using **//** at the beginning.
6. The **identifier** should begin by a letter and not by digit. It contains alpha numeric letters after first letter. No need to mention data types.
7. The left arrow "←" used as assignment operator. E.g. v←10
8. **Boolean** operators (TRUE, FALSE), **Logical** operators (AND, OR, NOT) and **Relational** operators (<,<=, >, >=,=, ≠, <>) are also used.
9. Input and Output can be done using **read** and **write**.
10. **Array[], if then else condition**, **branch** and **loop** can be also used in algorithm.

Example: GCD

The greatest common divisor(GCD) of two nonnegative integers m and n (not-both-zero), denoted GCD(m, n), is defined as the largest integer that divides both m and n evenly, i.e., with a remainder of zero.

To find GCD the various algorithms used are:

- ❖ Middle-School Procedure
- ❖ Integer Counter Checking
- ❖ Differential Algorithm
- ❖ Euclid's Algorithm

GCD Using Middle-School Procedure

Input: m = 12, n = 14
Output: 2
Prime factor of 12 = 1*2*2*3
Prime factor of 14 = 1*2*7
GCD(12, 14) = 2

Input: m = 5, n = 10
Output: 5
Prime factor of 10 = 1*2*5
Prime factor of 5 = 1*5
GCD(5, 10) = 5

Title: GCD Calculation Using Prime Factorization Method
Input: Two positive integers m and n
Output: The Greatest Common Divisor (GCD) of m and n
Start
Find the prime factorization of m.
Find the prime factorization of n.
Find all the common prime factors.
Compute the product of all the common prime factors and return it as GCD (m, n).
End

GCD Using Consecutive Integer Checking Algorithm

Example: gcd(10,6) = 2

t	m % t	n % t
6	10 % 6 = 4	
5	10 % 5 = 0	6 % 5 = 1
4	10 % 4 = 2	
3	10 % 3 = 1	
2	10 % 2 = 0	6 % 2 = 0

2 is the GCD, since m % t and n % t are zero.

Title: GCD Calculation Using Counter Integer Checking Method
Input: Two integers a and b.
Output: The Greatest Common Divisor (GCD) of a and b.
Start
Step 1: t= min{m,n}.
Step 2: if m% t returns zero then go to step 3 otherwise step 4.
Step 3: if n%t returns zero then return s otherwise step 4.
Step 4: Decrement t value by 1 and go to step 2.
End

GCD Using Euclid'sAlgorithm

Euclid's algorithm is based on repeatedly applying the equality GCD(m, n) = GCD(n, m mod n), where m mod n is the remainder of the m division by n until m mod n equals 0. Since GCD(m, 0) = m, the last value of m is also the greatest common divisor of the initial m and n.

GCD(60, 24) can be computed as follows:

GCD(60, 24) = GCD(24, 12) = GCD(12, 0) = 12.

Euclid's algorithm for computing gcd*(m, n)* expressed in pseudocode

ALGORITHM *Euclid_gcd(m, n)*

```
//Computes gcd(m, n) by Euclid's algorithm
//Input: Two nonnegative, not-both-zero integers m and n
//Output: Greatest common divisor of m and n
while n ≠ 0 do
    r ←m mod n
    m←n
    n←r
return m
```

- First, divide the larger number by the smaller number

$$135 \div 95 = 1 \text{ remainder } 40$$

- Divide 95 by 40

$$95 \div 40 = 2 \text{ remainder } 15$$

- Divide 40 by 15

$$40 \div 15 = 2 \text{ remainder } 10$$

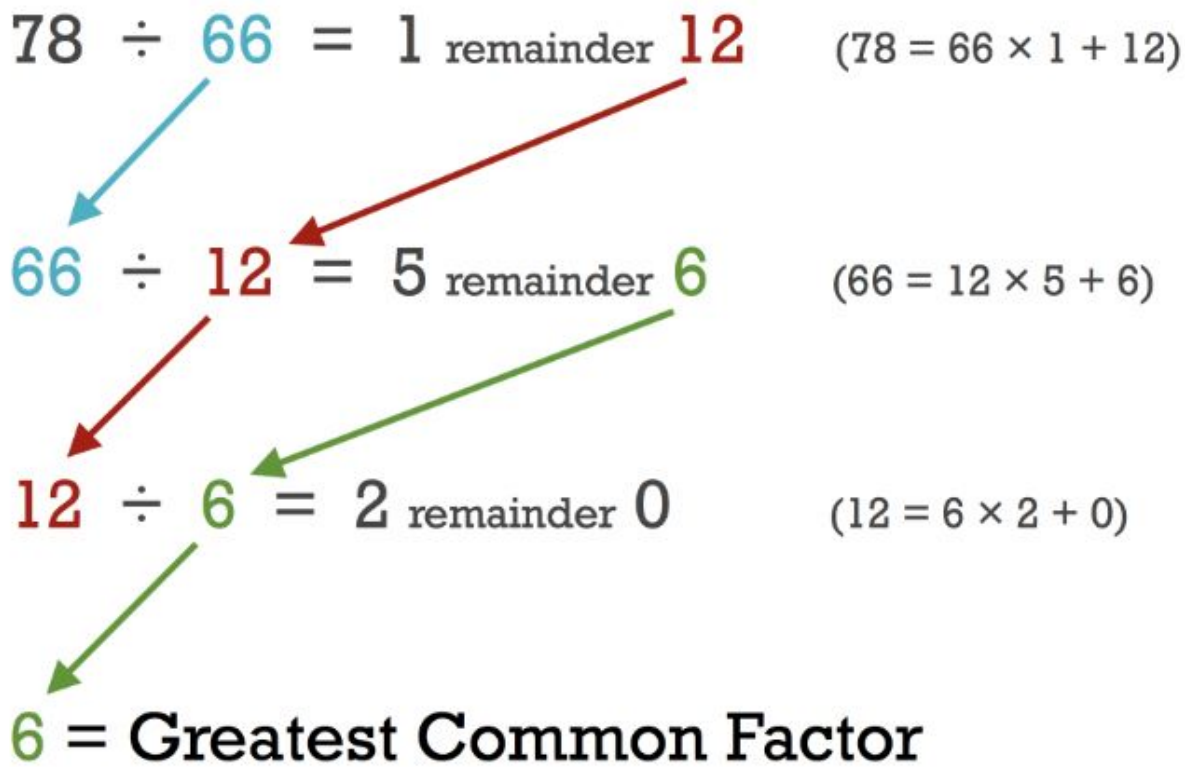
- Divide 15 by 10

$$15 \div 10 = 1 \text{ remainder } 5$$

- Divide 10 by 5

$$10 \div 5 = 2 \text{ remainder } 0$$

- Since the remainder is 0, the divisor **5** is the greatest common factor.



GCD Using Differential Algorithm

gcd(48, 18):

1. a = 48, b = 18. Since b ≠ 0, enter the loop.
2. a > b, so subtract b from a: a = 48 - 18 = 30, b = 18.
3. Repeat the loop.
4. a > b, so subtract b from a: a = 30 - 18 = 12, b = 18.
5. Repeat the loop.
6. b > a, so subtract a from b: a = 12, b = 18 - 12 = 6.
7. Repeat the loop.
8. a > b, so subtract b from a: a = 12 - 6 = 6, b = 6.
9. Repeat the loop.
10. a = b = 6. Exit the loop.
11. Return a, which is 6.

Title: GCD Calculation Using Differential Algorithm
Input: Two positive integers a and b.
Output: The Greatest Common Divisor (GCD) of a and b.
Start
Step 1: Define a function gcd(a, b) to calculate the GCD of two positive integers a and b.
Step 2: While $b \neq 0$ do:
- If $a > b$, then subtract b from a (i.e., $a = a - b$).
- Otherwise, subtract a from b (i.e., $b = b - a$).
Step 3: Return a, which is the GCD of the original two numbers.

Find the GCD for below Problems using all 4 methods:

i) (15, 10)
ii) (60, 36)
iii) (30, 12)

Generating Prime Numbers using Sieve of Eratosthenes

- *Let us take an example when n = 100. So, we need to print all prime numbers smaller than or equal to 100.*
- *We create a list of all numbers from 2 to 100.*

	2	3	4	5	6	7	8	9	10
11	12	13	14	15	16	17	18	19	20
21	22	23	24	25	26	27	28	29	30
31	32	33	34	35	36	37	38	39	40
41	42	43	44	45	46	47	48	49	50
51	52	53	54	55	56	57	58	59	60
61	62	63	64	65	66	67	68	69	70
71	72	73	74	75	76	77	78	79	80
81	82	83	84	85	86	87	88	89	90
91	92	93	94	95	96	97	98	99	100

According to the algorithm we will mark all the numbers which are divisible by 2 and are greater than or equal to the square of it.

	2	3	4	5	6	7	8	9	10
11	12	13	14	15	16	17	18	19	20
21	22	23	24	25	26	27	28	29	30
31	32	33	34	35	36	37	38	39	40
41	42	43	44	45	46	47	48	49	50
51	52	53	54	55	56	57	58	59	60
61	62	63	64	65	66	67	68	69	70
71	72	73	74	75	76	77	78	79	80
81	82	83	84	85	86	87	88	89	90
91	92	93	94	95	96	97	98	99	100

Prime Numbers

2

- *Now we move to our next unmarked number 3 and mark all the numbers which are multiples of 3 and are greater than or equal to the square of it.*

	2	3	4	5	6	7	8	9	10
11	12	13	14	15	16	17	18	19	20
21	22	23	24	25	26	27	28	29	30
31	32	33	34	35	36	37	38	39	40
41	42	43	44	45	46	47	48	49	50
51	52	53	54	55	56	57	58	59	60
61	62	63	64	65	66	67	68	69	70
71	72	73	74	75	76	77	78	79	80
81	82	83	84	85	86	87	88	89	90
91	92	93	94	95	96	97	98	99	100

Prime Numbers

2 3

- *We move to our next unmarked number 5 and mark all multiples of 5 and are greater than or equal to the square of it.*

	2	3	4	5	6	7	8	9	10
11	12	13	14	15	16	17	18	19	20
21	22	23	24	25	26	27	28	29	30
31	32	33	34	35	36	37	38	39	40
41	42	43	44	45	46	47	48	49	50
51	52	53	54	55	56	57	58	59	60
61	62	63	64	65	66	67	68	69	70
71	72	73	74	75	76	77	78	79	80
81	82	83	84	85	86	87	88	89	90
91	92	93	94	95	96	97	98	99	100

Prime Numbers

2 3 5

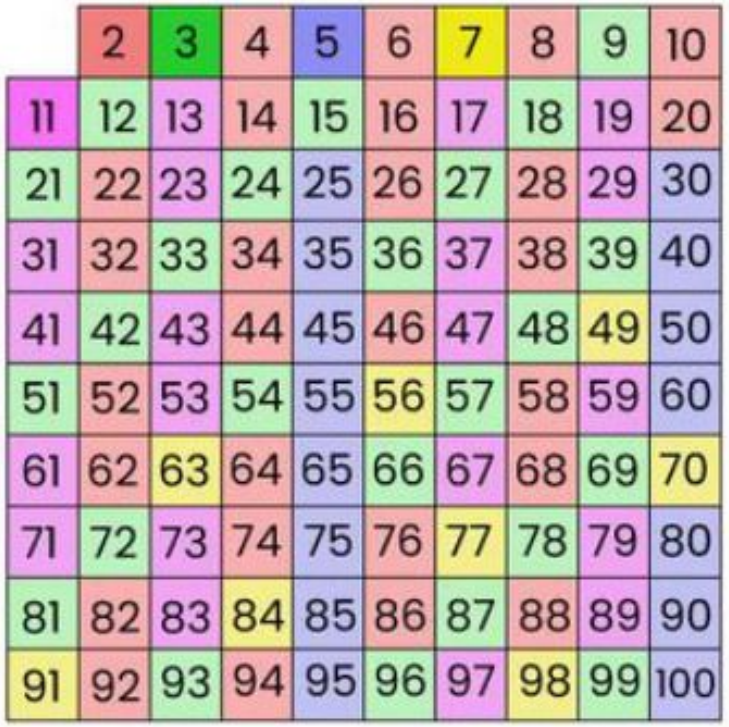
- *We move to our next unmarked number 7 and mark all multiples of 7 and are greater than or equal to the square of it.*

	2	3	4	5	6	7	8	9	10
11	12	13	14	15	16	17	18	19	20
21	22	23	24	25	26	27	28	29	30
31	32	33	34	35	36	37	38	39	40
41	42	43	44	45	46	47	48	49	50
51	52	53	54	55	56	57	58	59	60
61	62	63	64	65	66	67	68	69	70
71	72	73	74	75	76	77	78	79	80
81	82	83	84	85	86	87	88	89	90
91	92	93	94	95	96	97	98	99	100

Prime Numbers

2 3 5 7

We continue this process, and our final table will look like below:



	2	3	4	5	6	7	8	9	10
11	12	13	14	15	16	17	18	19	20
21	22	23	24	25	26	27	28	29	30
31	32	33	34	35	36	37	38	39	40
41	42	43	44	45	46	47	48	49	50
51	52	53	54	55	56	57	58	59	60
61	62	63	64	65	66	67	68	69	70
71	72	73	74	75	76	77	78	79	80
81	82	83	84	85	86	87	88	89	90
91	92	93	94	95	96	97	98	99	100

Prime Numbers

2	3	5	7
11	13	17	19
23	29	31	37
41	43	47	53
59	61	67	71
73	79	83	89
97			

So, the prime numbers are the unmarked ones: 2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59, 61, 67, 71, 73, 79, 83, 89 and 97.

Generating Prime Numbers using Sieve of Eratosthenes

Example: n = 25

2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25
2	3	0	5	0	7	0	9	0	11	0	13	0	15	0	17	0	19	0	21	0	23	0	25
2	3	0	5	0	7	0	0	0	11	0	13	0	0	0	17	0	19	0	0	0	23	0	25
2	3	0	5	0	7	0	0	0	11	0	13	0	0	0	17	0	19	0	0	0	23	0	0

2, 3, 5, 7, 11, 13, 17, 19, 23 are the generated prime numbers.

Note : If P is a number whose multiples are being eliminated, then P.P should not be greater than n, and therefore P cannot exceed $\lfloor \sqrt{n} \rfloor$.

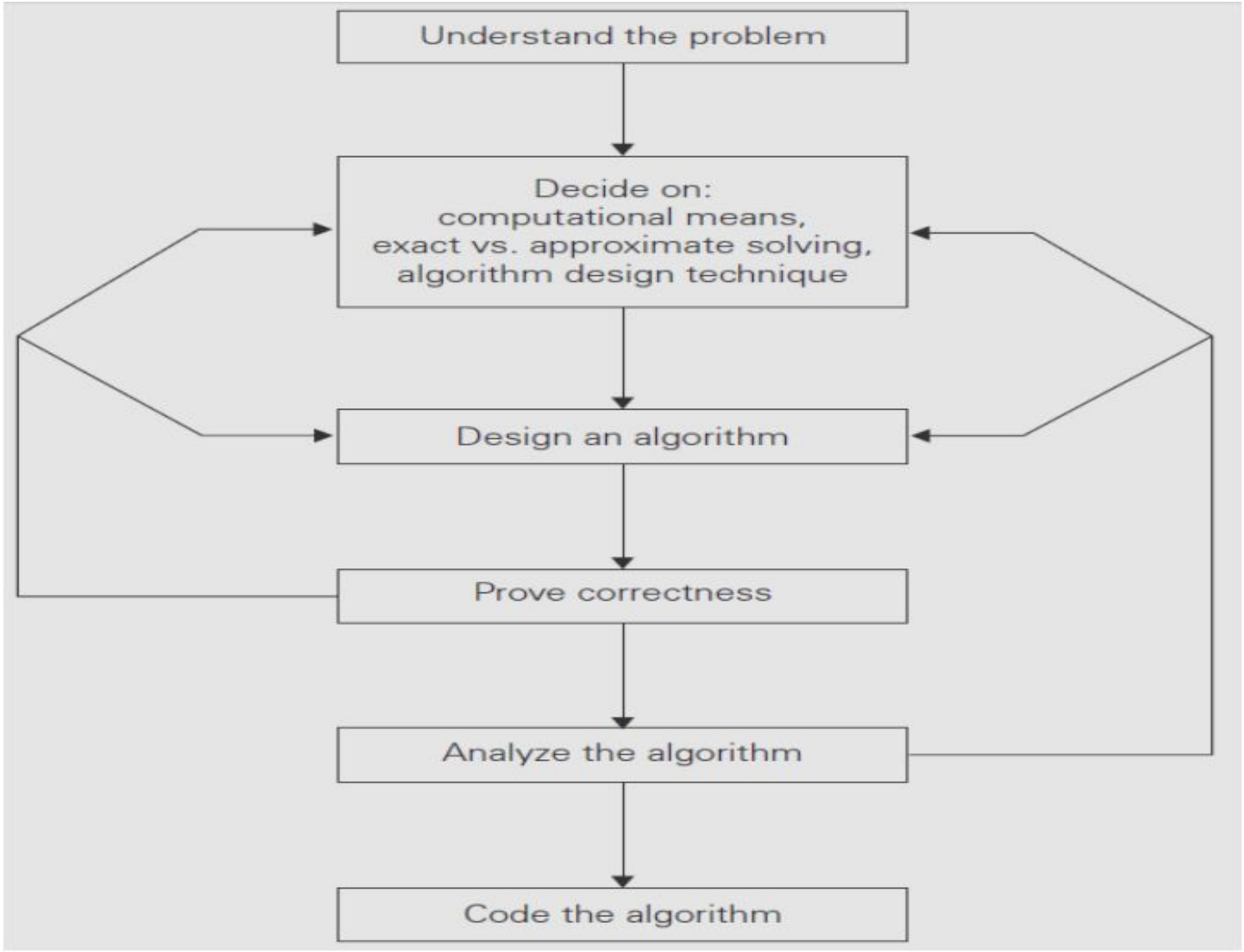
Role of algorithm in computing

- **Data processing:** Algorithms are used to process and analyze large amounts of data, such as sorting and searching algorithms.
- **Problem solving:** Algorithms are used to solve computational problems, such as mathematical problems, optimization problems, and decision-making problems.
- **Computer graphics:** Algorithms are used to create and process images and graphics, such as image compression algorithms and computer-generated graphics algorithms.

Role of algorithm in computing

- **Artificial Intelligence:** Algorithms are used to develop intelligent systems, such as machine learning algorithms, natural language processing algorithms, and computer vision algorithms.
- **Database management:** Algorithms are used to manage and organize large amounts of data in databases, such as indexing algorithms and query optimization algorithms.
- **Network communication:** Algorithms are used for efficient communication and data transfer in networks, such as routing algorithms and error correction algorithms.
- **Operating systems:** Algorithms are used in operating systems for tasks such as process scheduling, memory management, and disk management.

FUNDAMENTALS OF ALGORITHMIC PROBLEM SOLVING



Algorithm design and analysis process.

❖ Understanding the Problem

This is the first step in designing of an algorithm.

- Read the problem's description carefully to understand the problem statement completely.
- Ask questions to clarify the doubts about the problem.
- Identify the problem types and use existing algorithms to find solutions.
- Input (***instance***) to the problem and range of the input gets fixed.

❖ Decision making

The Decision making is done on the following:

- a) Ascertaining the Capabilities of the Computational Device
- In a *random-access machine* (*RAM*), instructions are executed one after another (The central assumption is that one operation at a time). Accordingly, algorithms designed to be executed on such machines are called ***sequential algorithms***.
- In some newer computers, operations are executed **concurrently**, i.e., in parallel. Algorithms that take advantage of this capability are called ***parallel algorithms***.
- choice of computational devices like Processor and memory is mainly based on space and time efficiency

Choosing between Exact and Approximate Problem Solving:

- ✔ The next principal decision is to choose between solving the problem exactly or solving it approximately.
- ✔ An algorithm used to solve the problem exactly and produce the correct result is called an exact algorithm.
- ✔ If the problem is so complex and not able to get exact solution, then we have to choose an algorithm called an approximation algorithm. i.e., produces an Approximate answer. E.g., extracting square roots, solving nonlinear equations, and evaluating definite integrals.

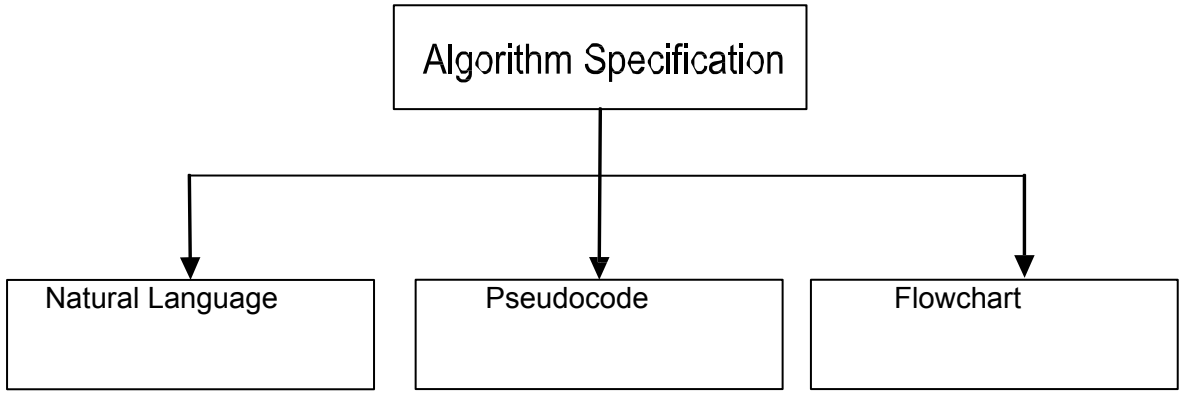
Algorithm Design Techniques

• An algorithm design technique (or "strategy" or "paradigm") is a general approach to solving problems algorithmically that is applicable to a variety of problems from different areas of computing.
• Though Algorithms and Data Structures are independent, but they are combined together to develop a program. Hence the choice of proper data structure is required before designing the algorithm.
• Implementation of algorithm is possible only with the help of Algorithms and Data Structures
• Algorithmic strategy/technique/paradigm are a general approach by which many problems can be solved algorithmically. E.g., Brute Force, Divide and Conquer, Dynamic Programming, Greedy Technique, and soon.

Methods of Specifying an Algorithm

There are three ways to specify an algorithm. They are:

- **Natural language**
- **Pseudocode**
- **Flowchart**



- Pseudocode and flowchart are the two options that are most widely used nowadays for specifying algorithms.

a. Natural Language

- It is very simple and easy to specify an algorithm using natural language. But many times specification of an algorithm by using natural language is not clear and thereby we get brief specifications.

Example: An algorithm to perform the addition of two numbers.

Step 1: Read the first number, say a.

Step 2: Read the first number, say b.

Step 3: Add the above two numbers and store the result in c.

Step 4: Display the result from c.

Such a specification creates difficulty while actually implementing it. Hence many programmers prefer to have specification of algorithm by means of Pseudocode.

b) Pseudocode:

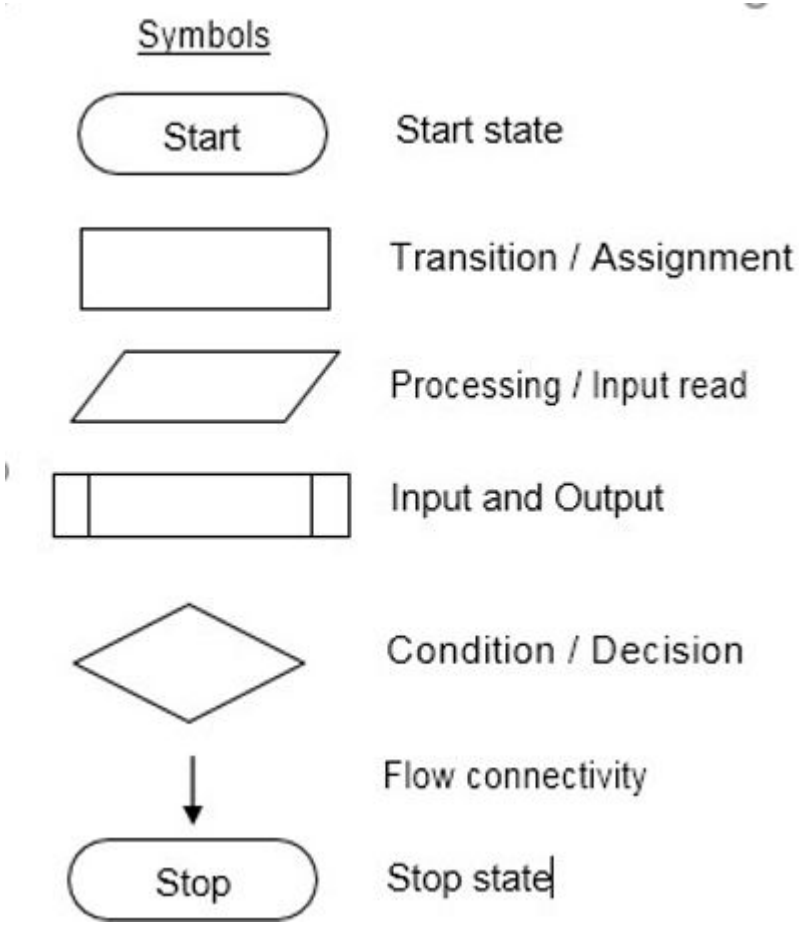
- Pseudocode is a mixture of a natural language and programming language constructs. Pseudocode is usually more precise than natural language.
- For Assignment operation left arrow "←", for comments two slashes "//",if condition, for, while loops are used.

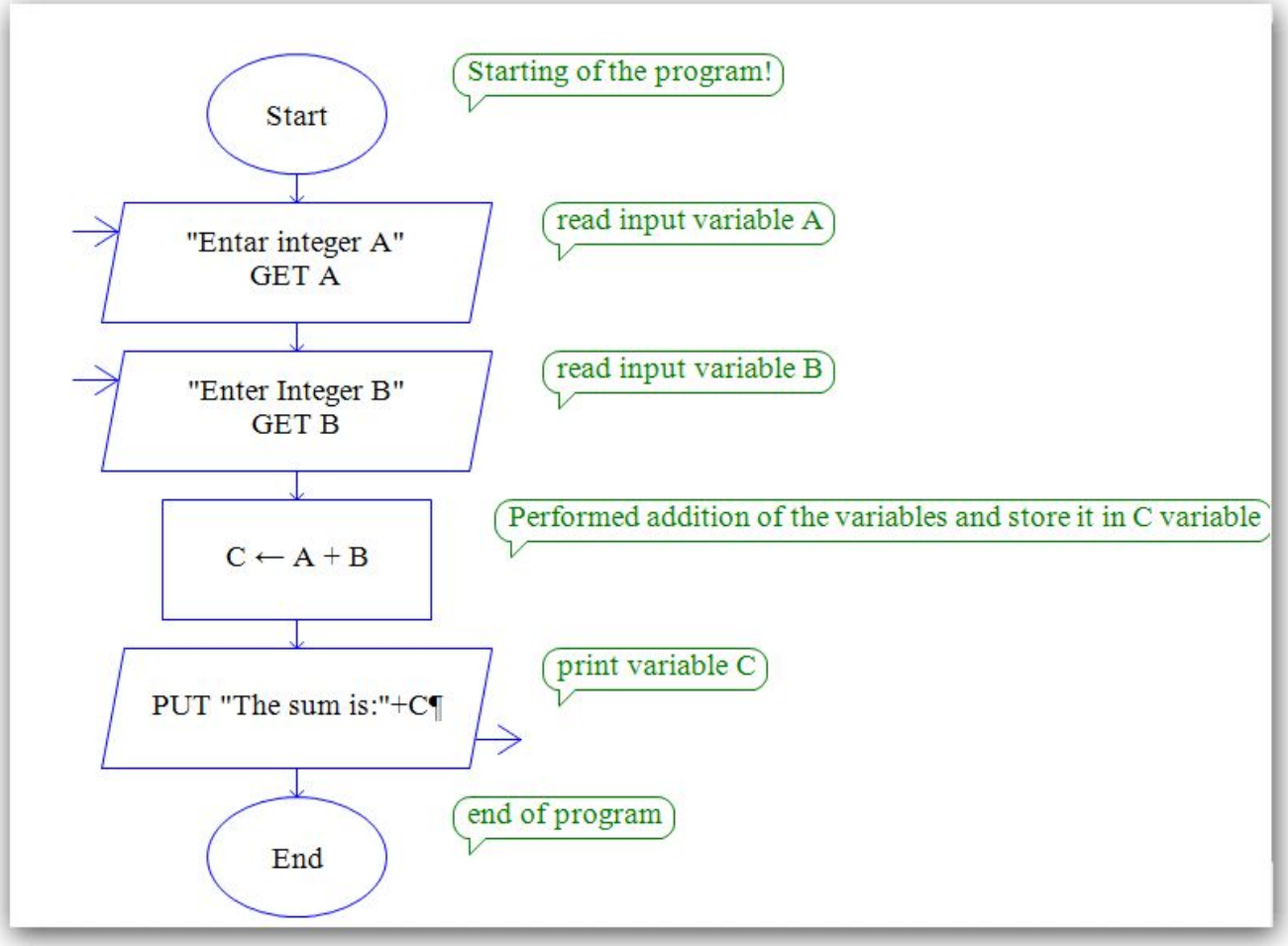
```
ALGORITHM Sum(a,b)
    //Problem Description: This algorithm performs addition of two numbers
    //Input: Two integers a and b
    //Output: Addition of two integers
    c←a+b
    Return c
```

c) Flowchart

• In the earlier days of computing, the dominant method for specifying algorithms was a flowchart, this representation technique has proved to be inconvenient.

• A Flowchart is a graphical representation of an algorithm. It is a method of expressing an algorithm by a collection of connected geometric shapes containing descriptions of the algorithm's steps.





(iv) Proving an Algorithm's Correctness

• Once an algorithm has been specified then its correctness must be proved.
• An algorithm must yield a required result for every legitimate input in a finite amount of time.
• For Example, the correctness of Euclid's algorithm for computing the greatest common divisor stems from the correctness of the equality gcd(m, n) = gcd(n, m mod n).
• A common technique for proving correctness is to use mathematical induction because an algorithm's iterations provide a natural sequence of steps needed for such proofs.
• The notion of correctness for approximation algorithms is less straightforward than it is for exact algorithms. The error produced by the algorithm should not exceed a predefined limit.

(v) Analyzing an Algorithm

For an algorithm the most important is efficiency. There are two kinds of algorithm efficiency.
They are:
• Time efficiency, indicating how fast the algorithm runs, and
• Space efficiency, indicating how much extra memory it uses.
• The efficiency of an algorithm is determined by measuring both time efficiency and space efficiency.
• So factors to analyze an algorithm are:
- Time efficiency of an algorithm
- Space efficiency of an algorithm
- Simplicity of an algorithm
- Generality of an algorithm

(vi) Coding an Algorithm

- The coding/implementation of an algorithm is done by a suitable programming language like C, C++, or JAVA.
- The transition from an algorithm to a program can be done incorrectly or inefficiently. Implementing an algorithm correctly is necessary. The Algorithm's power should not be reduced by efficient implementation.
- Standard tricks like computing a loop's invariant (an expression that does not change its value) outside the loop, collecting common subexpressions, replacing expensive operations with cheap ones, selecting programming language, and so on should be known to the programmer.
- Typically, such improvements can speed up a program only by a constant factor, whereas a better algorithm can make a difference in running time by orders of magnitude. But a 10–50% speed up may be worthwhile once an algorithm is selected.
- It is very essential to write an optimized code (efficient code) to reduce the burden of a compiler.

FUNDAMENTALS OF THE ANALYSIS OF ALGORITHM EFFICIENCY

- The efficiency of an algorithm can be in terms of **time** and **space.**
- The algorithm efficiency can be analyzed by the following ways.

a. Analysis Framework.

b. Asymptotic Notations and its properties.

c. Mathematical analysis for Recursive algorithms.

d. Mathematical analysis for Non-recursive algorithms.

a. Analysis Framework

- There are two kinds of efficiencies to analyze the efficiency of any algorithm. They are:
    - ***Time efficiency*** indicates how fast the algorithm runs, and
    - ***Space efficiency*** indicates how much extra memory it uses.
- The algorithm analysis framework consists of the following:
    - Measuring an Input's Size
    - Units for Measuring Running Time
    - Orders of Growth
    - Worst-Case, Best-Case, and Average-Case Efficiencies

- **Measuring an Input's Size**
  - An algorithm's efficiency is defined as a function of some parameter n indicating the algorithm's input size. In most cases, selecting such a parameter is quite straightforward. For example, it will be the size of the list for problems of sorting, searching.
  - For the problem of evaluating a polynomial $p(x) = a_n x^n + \ldots + a_0$ of degree n, the size of the parameter will be the polynomial's degree or the number of its coefficients, which is larger by 1 than its degree.
  - In computing the product of two $n \times n$ matrices, the choice of a parameter indicating an input size does matter.
  - Consider a spell-checking algorithm. If the algorithm examines individual characters of its input, then the size is measured by the number of characters.
  - In measuring input size for algorithms solving problems such as checking primality of a positive integer n. the input is just one number.
  - The input size by the number b of bits in the n's binary representation is $b=(\log_2 n)+1$.

- **Units for Measuring Running Time**
- Some standard unit of time measurement such as a second, or millisecond, and so on can be used to measure the running time of a program after implementing the algorithm Drawbacks,
  - Dependence on the speed of a particular computer.
  - Dependence on the quality of a program implementing the algorithm.
  - The compiler used in generating the machine code.
- The difficulty of clocking the actual running time of the program. So, we need metric to measure an algorithm's efficiency that does not depend on these extraneous factors. One possible approach is to count the number of times each of the algorithm's operations is executed. This approach is excessively difficult.
- The most important operation (+, -, *, /) of the algorithm, called the basic operation. Computing the number of times, the basic operation is executed is easy. The total running time is basic operations count.

- **ORDERS OF GROWTH**
  - A difference in running times on small inputs is not what distinguishes efficient algorithms from in efficient ones.
  - For example, for the greatest common divisor of two small numbers, it is not immediately clear how much more efficient Euclid's algorithm is compared to the other algorithms, the difference in algorithm efficiencies becomes clear for larger numbers only.
  - For large values of *n*,
  - it is the function's order of growth that counts just like Table 1.1, which contains values of a few functions particularly important for the analysis of algorithms.

n	$\sqrt{n}$	$\log_2 n$	n	$n \log_2 n$	n^2	n^3	2^n	$n!$
1	1	0	1	0	1	1	2	1
2	1.4	1	2	2	4	4	4	2
4	2	2	4	8	16	64	16	24
8	2.8	3	8	$2.4 \cdot 10^1$	64	$5.1 \cdot 10^2$	$2.6 \cdot 10^2$	$4.0 \cdot 10^4$
10	3.2	3.3	10	$3.3 \cdot 10^1$	10^2	10^3	10^3	$3.6 \cdot 10^6$
16	4	4	16	$6.4 \cdot 10^1$	$2.6 \cdot 10^2$	$4.1 \cdot 10^3$	$6.5 \cdot 10^4$	$2.1 \cdot 10^{13}$
10^2	10	6.6	10^2	$6.6 \cdot 10^2$	10^4	10^6	$1.3 \cdot 10^{30}$	$9.3 \cdot 10^{157}$
10^3	31	10	10^3	$1.0 \cdot 10^4$	10^6	10^9	Very big computation	
10^4	10^2	13	10^4	$1.3 \cdot 10^5$	10^8	10^{12}		
10^5	$3.2 \cdot 10^2$	17	10^5	$1.7 \cdot 10^6$	10^{10}	10^{15}		
10^6	10^3	20	10^6	$2.0 \cdot 10^7$	10^{12}	10^{18}		

Example on Orders of growth

To understand the concept of Order of Growth of an algorithm, we will consider the following time complexities of algorithms.

The time complexity of Algorithm A: 100n + 1

The time complexity of Algorithm B: n^2 + n + 1

As shown below, let's try to analyze the number of steps these algorithms execute as the input size increases.

Input Size	Run Time of Algorithm A	Run time of Algorithm B
10	1,001	111
100	10,001	10,101
1,000	100,001	10,01,001
10,000	10,00,001	

- **Worst-Case, Best-Case, and Average-Case Efficiencies**

Best case efficiency : The best-case efficiency of an algorithm is its efficiency for the best case input of size n.

- The algorithm runs the fastest among all possible inputs of that size n.
- In sequential search, If we search a first element in list of size n. (i.e. first element equal to a search key), then the running time is $C_{best}(n) = 1$

Average case efficiency :

- The Average case efficiency lies between best case and worst case.
- To analyze the algorithm's average case efficiency, we must make some assumptions about possible inputs of size n.

Worst-case efficiency : The worst-case efficiency of an algorithm is its efficiency for the worst case input of size n.

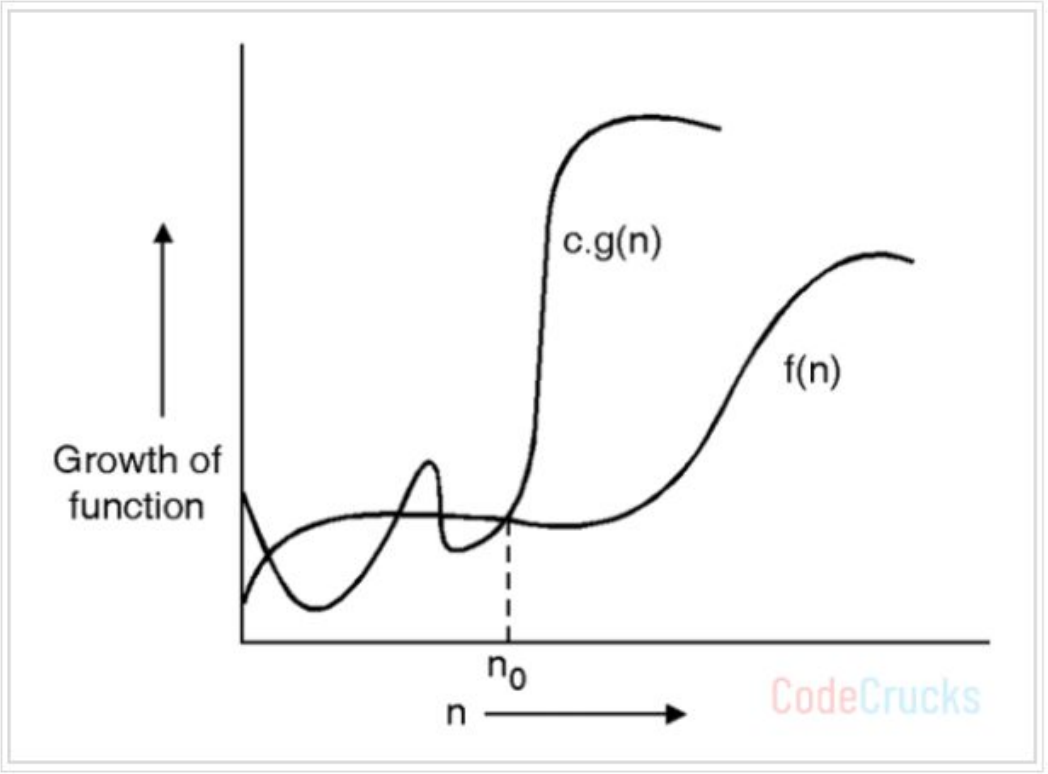
- The algorithm runs the longest among all possible inputs of that size.
- For the input of size n, the running time is $C_{worst}(n) = n$.

ASYMPTOTIC NOTATIONS AND ITS PROPERTIES

- Asymptotic notation is a notation, which is used to take meaningful statements about the **efficiency of a program**.
- It is the mathematical way of representing the **time complexity**.
- computer scientists use three notations, they are:
- **O - Big oh notation - upper bound**
- **Θ - Big theta notation - Average bound**
- **Ω - Big omega notation - lower bound**

Big-Oh/ Upper Bound

Let f(n) and g(n) are two nonnegative functions indicating the running time of two algorithms. We say g(n) is the upper bound of f(n) if there exist some positive constants c and n_0 such that $f(n) \leq c.g(n)$ for all $n \geq n_0$. It is denoted as $f(n) = O(g(n))$.



Upper bound – Big oh notation

Examples:

Find upper bound of running time of constant function f(n) = 6993.

To find the upper bound of f(n), we have to find c and n_0 such that $0 \leq f(n) \leq c.g(n)$ for all $n \geq n_0$

$f(n) \leq c \times g(n)$

$6993 \leq c \times g(n)$

$6993 \leq 6993 \times 1$

So, $c = 6993$ and $g(n) = 1$

Any value of c which is greater than 6993, satisfies the above inequalities, so all such values of c are possible.

$0 \leq 6993 \leq 8000 \times 1 \rightarrow$ true

$0 \leq 6993 \leq 10500 \times 1 \rightarrow$ true

Function f(n) is constant, so it does not depend on problem size n. So $n_0 = 1$

$f(n) = O(g(n)) = O(1)$ for $c = 6993$, $n_0 = 1$

$f(n) = O(g(n)) = O(1)$ for $c = 8000$, $n_0 = 1$ and so on.

Example: Find upper bound of running time of a linear function f(n) = 6n + 3.

Tabular Approach

- Now, manually find out the proper n_0, such that f (n) ≤ c.g (n)

n	f(n) = 6n + 3	c.g(n) = 7n
1	9	7
2	15	14
3	21	21
4	27	28
5	33	35

f(n) = O(g(n)) = O(n) for c = 7, n_0 = 3

Find upper bound of running time of quadratic function $f(n) = 3n^2 + 2n + 4$.

Tabular approach:

$3n^2 + 2n + 4 \leq c.g\ (n)$

$3n^2 + 2n + 4 \leq 4n^2$

Now, manually find out the proper n_0, such that $f(n) \leq c.g(n)$

n	$f(n) = 3n^2 + 2n + 4$	$c.g\ (n) = 4n^2$
1	9	4
2	20	16
3	37	36
4	60	64
5	89	100

$f(n) = O\ (g(n)) = O\ (n^2)$ for $c = 4$, $n_0 = 4$

Example: Find upper bound of running time of a cubic function $f(n) = 2n^3 + 4n + 5$.

Tabular approach

$2n^3 + 4n + 5 \leq c \times g(n)$

$2n^3 + 4n + 5 \leq 3n^3$

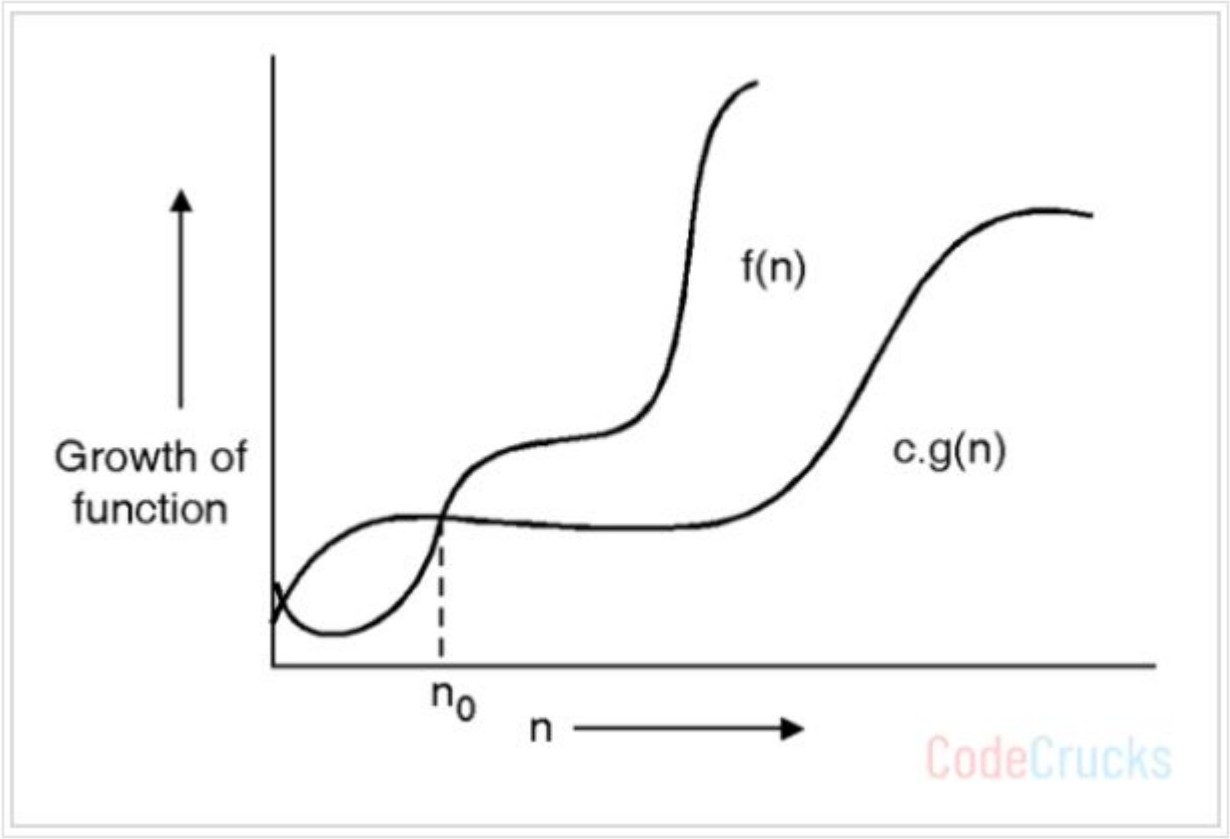
Now, manually find out the proper n_0, such that $f(n) \leq c \times g(n)$

n	$f(n) = 2n^3 + 4n + 5$	$c.g(n) = 3n^3$
1	11	3
2	29	24
3	71	81
4	149	192

$f(n) = O(g(n)) = O(n^3)$ for $c = 3$, $n_0 = 3$ and so on.

Big- Omega (Lower Bound)

Let f(n) and g(n) are two nonnegative functions indicating the running time of two algorithms. We say the function g(n) is a lower bound of function f(n) if there exist some positive constants c and n_0 such that c.g(n) ≤ f(n) for all n ≥ n_0. It is denoted as f(n) = Ω (g(n)).



Examples

Find the lower bound of running time of constant function f(n) = 23.

To find lower bound of f(n), we have to find c and n_0 such that $\{c \times g(n) \leq f(n)$ for all $n \geq n_0\}$
$c \times g(n) \leq f(n)$
$c \times g(n) \leq 23$
$23.1 \leq 23 \rightarrow$ true
$12.1 \leq 23 \rightarrow$ true
$5.1 \leq 23 \rightarrow$ true
Above all three inequalities are true and there exists such infinite inequalities
So c = 23, c = 12, c = 5 and g(n) = 1. Any value of c which is less than or equals to 23, satisfies the above inequality, so all such value of c are possible. Function f(n) is constant, so it does not depend on problem size n. Hence $n_0 = 1$
$f(n) = \Omega(g(n)) = \Omega(1)$ for $c = 23$, $n_0 = 1$
$f(n) = \Omega(g(n)) = \Omega(1)$ for $c = 12$, $n_0 = 1$ and so on.

Find the lower bound of the running time of a linear function $f(n) = 6n + 3$.

To find lower bound of f(n), we have to find c and n_0 such that $c.g(n) \leq f(n)$ for all $n \geq n_0$

$c \times g(n) \leq f(n)$

$c \times g(n) \leq 6n + 3$

$6n \leq 6n + 3 \rightarrow$ true, for all $n \geq n_0$

$5n \leq 6n + 3 \rightarrow$ true, for all $n \geq n_0$

Above both inequalities are true and there exists such infinite inequalities. So,

$f(n) = \Omega(g(n)) = \Omega(n)$ for $c = 6$, $n_0 = 1$

$f(n) = \Omega(g(n)) = \Omega(n)$ for $c = 5$, $n_0 = 1$

and so on.

Find lower bound of running time of quadratic function $f(n) = 3n^2 + 2n + 4$.

To find lower bound of f(n), we have to find c and n_0 such that $0 \leq c.g(n) \leq f(n)$ for all $n^3 \geq n_0$

$c \times g(n) \leq f(n)$

$c \times g(n) \leq 3n^2 + 2n + 4$

$3n^2 \leq 3n^2 + 2n + 4, \rightarrow$ true, for all $n \geq 1$

$n^2 \leq 3n^2 + 2n + 4, \rightarrow$ true, for all $n \geq 1$

Above both inequalities are true and there exists such infinite inequalities.

So, $f(n) = \Omega(g(n)) = \Omega(n^2)$ for $c = 3$, $n_0 = 1$

$f(n) = \Omega(g(n)) = \Omega(n^2)$ for $c = 1$, $n_0 = 1$

and so on.

Find lower bound of running time of quadratic function $f(n) = 2n^3 + 4n + 5$.

To find lower bound of f(n), we have to find c and n_0 such that $0 \leq c.g(n) \leq f(n)$ for all $n \geq n_0$

$c \times g(n) \leq f(n)$

$c \times g(n) \leq 2n^3 + 4n + 5$

$2n^3 \leq 2n^3 + 4n + 5 \rightarrow$ true, for all $n \geq 1$

$n^3 \leq 2n^3 + 4n + 5 \rightarrow$ true, for all $n \geq 1$

Above both inequalities are true and there exists such infinite inequalities.

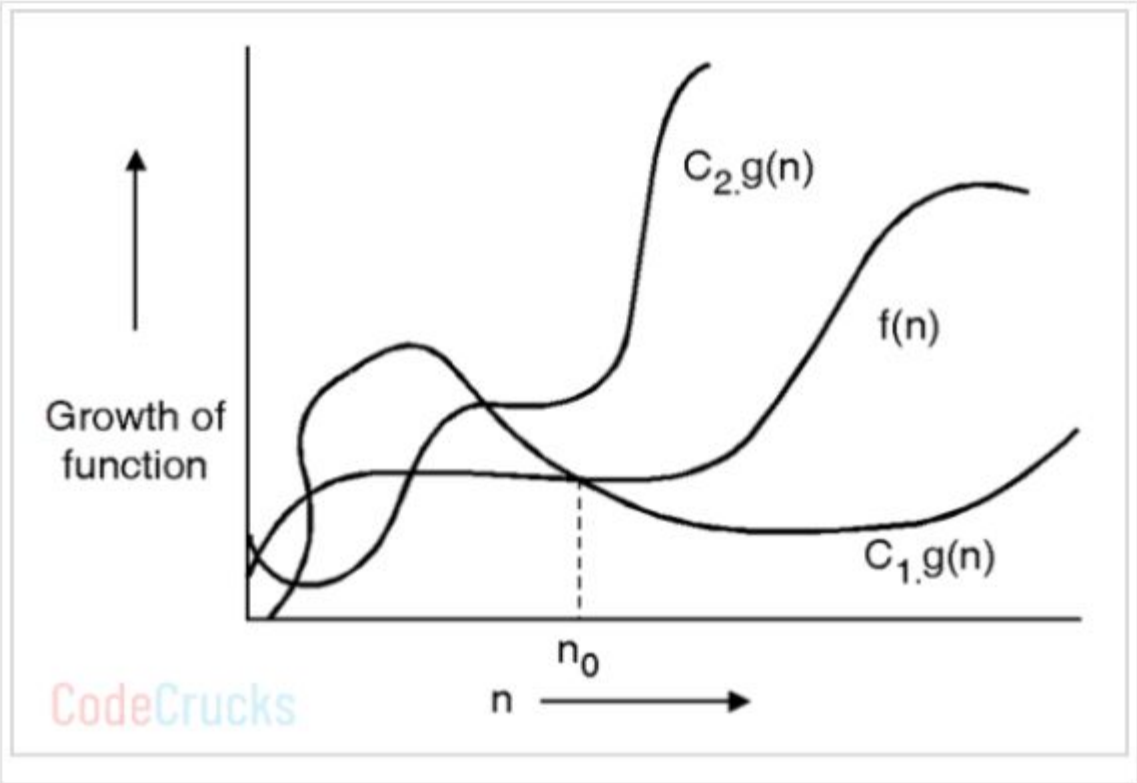
So, $f(n) = \Omega(g(n)) = \Omega(n^3)$ for $c = 2$, $n_0 = 1$

$f(n) = \Omega(g(n)) = \Omega(n^3)$ for $c = 1$, $n_0 = 1$

and so on.

Tight Bound/ Big Theta/ Average Bound

Let f(n) and g(n) are two nonnegative functions indicating running time of two algorithms. We say the function g(n) is tight bound of function f(n) if there exist some positive constants c_1, c_2, and n_0 such that $c_1 \times g(n) \leq f(n) \leq c_2 \times g(n)$ for all $n \geq n_0$. It is denoted as $f(n) = \Theta(g(n))$.



Tight Bound – Big Theta

Find tight bound of running time of constant function f(n) = 23.

To find tight bound of f(n), we have to find c_1, c_2 and n_0 such that, $c_1 \times g(n) \leq f(n) \leq c_2 \times g(n)$ for all $n \geq n_0$
$c_1 \times g(n) \leq 23 \leq c_2 \times g(n)$

$22 \times 1 \leq 23 \leq 24 \times 1, \rightarrow$ true for all $n \geq 1$

Above both inequalities are true and there exists such infinite inequalities.
So, $(c_1, c_2) = (22, 24)$ and $g(n) = 1$, for all $n \geq 1$

$f(n) = \Theta(g(n)) = \Theta(1)$ for $c_1 = 22$, $c_2 = 24$, $n_0 = 1$

Find tight bound of running time of a linear function $f(n) = 6n + 3$.

To find tight bound of f(n), we have to find c_1, c_2 and n_0 such that, $c_1 \times g(n) \leq f(n) \leq c_2 \times g(n)$ for all $n \geq n_0$

$c_1 \times g(n) \leq 6n + 3 \leq c_2 \times g(n)$

$5n \leq 6n + 3 \leq 9n$, for all $n \geq 1$

Above inequality is true and there exists such infinite inequalities.

So, $f(n) = \Theta(g(n)) = \Theta(n)$ for $c_1 = 5$, $c_2 = 9$, $n_0 = 1$

Find tight bound of running time of quadratic function $f(n) = 3n^2 + 2n + 4$.

To find tight bound of f(n), we have to find c_1, c_2 and n_0 such that,
$c_1 \times g(n) \leq f(n) \leq c_2 \times g(n)$ for all $n \geq n_0$

$c_1 \times g(n) \leq 3n^2 + 2n + 4 \leq c_2 \times g(n)$

$3n^2 \leq 3n^2 + 2n + 4 \leq 4n^2$, for all $n \geq 1$

The above inequality is true and there exist such infinite inequalities. So,
$f(n) = \Theta(g(n)) = \Theta(n^2)$ for $c_1 = 3$, $c_2 = 4$, $n_0 = 1$

Example: Find tight bound of running time of a cubic function $f(n) = 2n^3 + 4n + 5$.

To find tight bound of f(n), we have to find c_1, c_2 and n_0 such that,
$c_1 \times g(n) \leq f(n) \leq c_2 \times g(n)$ for all $n \geq n_0$

$c_1 \times g(n) \leq 2n^3 + 4n + 5 \leq c_2 \times g(n)$
$2n^3 \leq 2n^3 + 4n + 5 \leq 11n^3$, for all $n \geq 1$

Above inequality is true and there exists such infinite inequalities. So,
$f(n) = \Theta(g(n)) = \Theta(n^3)$ for $c_1 = 2$, $c_2 = 11$, $n_0 = 1$

Exercise:

Show that : (i) $3n + 2 = \Theta(n)$ (ii) $6*2^n + n^2 = \Theta(2^n)$

Mathematical analysis of nonrecursive algorithms

Steps in mathematical analysis of nonrecursive algorithms:

1. Decide on parameter n indicating input size
2. Identify the algorithm's basic operation
3. Determine worst, average, and best case for input of size n
4. Set up summation for C(n) reflecting the algorithm's loop structure
5. Simplify summation using standard formulas

Input size and Basic Operation examples

Problem	**Input Size**	**Basic Operation**
Search for key in list of n items	Number of items in list n	Comparison of Key element with Array element
Sort an array of numbers	The number of elements in the array	Comparison of two array entries
Multiply two matrices of floating point numbers	Dimensions of matrices	Floating point multiplication
Compute a^n	n	Floating point multiplication
Graph problem	Number of vertices and edges	Visiting a vertex or traversing an edge

Useful Formulas for the Analysis of Algorithms

Important Summation Formulas

1. $\sum_{i=l}^{u} 1 = \underbrace{1+1+\cdots+1}_{u-l+1 \text{ times}} = u-l+1$ (l, u are integer limits, $l \le u$); $\sum_{i=1}^{n} 1 = n$
2. $\sum_{i=1}^{n} i = 1+2+\cdots+n = \frac{n(n+1)}{2} \approx \frac{1}{2}n^2$
3. $\sum_{i=1}^{n} i^2 = 1^2+2^2+\cdots+n^2 = \frac{n(n+1)(2n+1)}{6} \approx \frac{1}{3}n^3$
4. $\sum_{i=1}^{n} i^k = 1^k+2^k+\cdots+n^k \approx \frac{1}{k+1}n^{k+1}$
5. $\sum_{i=0}^{n} a^i = 1+a+\cdots+a^n = \frac{a^{n+1}-1}{a-1}$ ($a \ne 1$); $\sum_{i=0}^{n} 2^i = 2^{n+1}-1$
6. $\sum_{i=1}^{n} i2^i = 1\cdot 2+2\cdot 2^2+\cdots+n2^n = (n-1)2^{n+1}+2$
7. $\sum_{i=1}^{n} \frac{1}{i} = 1+\frac{1}{2}+\cdots+\frac{1}{n} \approx \ln n+\gamma$, where $\gamma \approx 0.5772\ldots$ (Euler's constant)
8. $\sum_{i=1}^{n} \lg i \approx n \lg n$

Sum Manipulation Rules

1. $\sum_{i=l}^{u} ca_i = c\sum_{i=l}^{u} a_i$
2. $\sum_{i=l}^{u} (a_i \pm b_i) = \sum_{i=l}^{u} a_i \pm \sum_{i=l}^{u} b_i$
3. $\sum_{i=l}^{u} a_i = \sum_{i=l}^{m} a_i + \sum_{i=m+1}^{u} a_i$, where $l \le m < u$
4. $\sum_{i=l}^{u} (a_i - a_{i-1}) = a_u - a_{l-1}$

Commonly used sum manipulation rules and Summation formulas

- Two frequently used basic rules of sum manipulation

$$\sum_{i=l}^{u} ca_i = c\sum_{i=l}^{u} a_i, \qquad \textbf{(R1)}$$

$$\sum_{i=l}^{u} (a_i \pm b_i) = \sum_{i=l}^{u} a_i \pm \sum_{i=l}^{u} b_i, \qquad \textbf{(R2)}$$

- Summation Formulas

$$\sum_{i=l}^{u} 1 = u - l + 1 \quad \text{where } l \le u \text{ are some lower and upper integer limits,} \qquad \textbf{(S1)}$$

$$\sum_{i=0}^{n} i = \sum_{i=1}^{n} i = 1 + 2 + \cdots + n = \frac{n(n+1)}{2} \approx \frac{1}{2}n^2 \in \Theta(n^2). \qquad \textbf{(S2)}$$

Example: Maximum element in an Array

ALGORITHM *MaxElement*$(A[0..n-1])$

```
//Determines the value of the largest element in a given array
//Input: An array A[0..n − 1] of real numbers
//Output: The value of the largest element in A
maxval ← A[0]
for i ← 1 to n − 1 do
    if A[i] > maxval
        maxval ← A[i]
return maxval
```

Basic Operation: Comparison operation A[i] > Maxval

Let us denote *T(n)* the number of times this comparison is executed and try to find a formula expressing it as a function of size *n*. The algorithm makes one comparison on each execution of the loop, which is repeated for each value of the loop's variable *i* within the bounds 1 and *n* − 1, inclusive. Therefore, we get the following sum for

$$T(n) = \sum_{i=1}^{n-1} 1.$$

This is an easy sum to compute because it is nothing other than 1 repeated *n* − 1 times. Thus,

$$T(n) = \sum_{i=1}^{n-1} 1 = n - 1 \in \Theta(n).$$

Questions:

Consider the Algorithm ALGORITHM Sum(n)

```
// Input: A nonnegative integer(n)
S = 0
for i =1 to n do
        S = S + i
return S
```

a. What does this algorithm compute?
b. Considering the, basic operation as S =S + i, Find out how many times the basic operation will be executed?
c. What is the efficiency class of this algorithm?

Answer:

a. Computes the sum of the first n numbers
b. Number of executions of basic operation S= S + i is n
d. The basic operation is always (worst, average, best case) executed n times, so it's Θ(n).

Example: Finding the Maximum and Minimum Element in an array

Algorithm *Secret*$(A[0..n-1])$
//Input: An array $A[0..n-1]$ of n real numbers
$minval \leftarrow A[0]$; $maxval \leftarrow A[0]$
for $i \leftarrow 1$ **to** $n-1$ **do**
    if $A[i] < minval$
        $minval \leftarrow A[i]$
    if $A[i] > maxval$
        $maxval \leftarrow A[i]$
return $maxval - minval$

a. Finding maximum and minimum
b. Basic Operation: Comparison (A[i]<Minvalue and A[i]>maxvalue)
c. $T(n)=\sum_{i=1}^{i=n-1} 2 = 2\sum_{i=1}^{i=n-1} 1 = 2(n-1) = 2n-2 = \Theta(n)$

a. What does this algorithm compute?
b. Considering the basic operation as A[i] < minval and A[i]> maxval, find out how many times the basic operations will be executed.
c. What is the efficiency class of this algorithm?

Questions:

In the following code find out how many times the **sum++** will be executed

```
sum = 0;
for( i = 1; i <= n; i++)
{
   for( j = 1; j <= n; j++)
      { sum++; }
}
```

$$T(n)=\sum_{i=1}^{i=n}\sum_{j=1}^{j=n}1 = \ominus(n^2)$$

Example: Checking all elements in an array are distinct

Element uniqueness problem: check whether all the elements in a given array of n elements are distinct.

ALGORITHM *UniqueElements*$(A[0..n-1])$

```
//Determines whether all the elements in a given array are distinct
//Input: An array A[0..n − 1]
//Output: Returns "true" if all the elements in A are distinct
//          and "false" otherwise
for i ← 0 to n − 2 do
    for j ← i + 1 to n − 1 do
        if A[i] = A[j] return false
return true
```

$$T(n) = \sum_{i=0}^{n-2} \sum_{j=i+1}^{n-1} 1 = \sum_{i=0}^{n-2} [(n-1) - (i+1) + 1] = \sum_{i=0}^{n-2} (n-1-i)$$

We also could have computed the sum $\sum_{i=0}^{n-2}(n-1-i)$ faster as follows:

$$\sum_{i=0}^{n-2}(n-1-i) = (n-1) + (n-2) + \cdots + 1 = \frac{(n-1)n}{2},$$

$$\frac{(n-1)n}{2} \approx \frac{1}{2}n^2 \in \Theta(n^2).$$

Questions:

In a competition, four different functions are observed. All the functions use a single for loop and within the for loop, same set of statements are executed. Consider the following for loops:

A) for(i = 0; i < n; i++)

B) for(i = 0; i < n; i += 2)

C) for(i = 1; i < n; i *= 2)

D) for(i = n; i > -1; i /= 2)

If **n** is the size of input(positive), which function is most efficient(if the task to be performed is not an issue)?

Answer: (C)

Explanation: The time complexity of first for loop is Θ(n). The time complexity of second for loop is Θ(n/2), equivalent to Θ(n) in asymptotic analysis.
The time complexity of third for loop is Θ(logn). The fourth for loop doesn't terminate.

Questions:

Consider the following function Void unknown(int n)

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

How many times the statement k=k+n/2 will be executed

Answer

The outer loop runs n/2 or Theta(n) times.

The inner loop runs (Logn) times (Note that j is multiplied by 2 in every iteration).

So the statement "k = k + n/2;" runs Theta(n*logn) times.

Questions:

Given two $n \times n$ matrices A and B, find the time efficiency of the definition-based algorithm for computing their product $C = AB$

ALGORITHM *MatrixMultiplication*$(A[0..n-1, 0..n-1], B[0..n-1, 0..n-1])$

```
//Multiplies two square matrices of order n by the definition-based algorithm
//Input: Two n × n matrices A and B
//Output: Matrix C = AB
for i ← 0 to n − 1 do
    for j ← 0 to n − 1 do
        C[i, j] ← 0.0
        for k ← 0 to n − 1 do
            C[i, j] ← C[i, j] + A[i, k] ∗ B[k, j]
return C
```

$$T(n) = \sum_{i=0}^{n-1} \sum_{j=0}^{n-1} \sum_{k=0}^{n-1} 1 = \sum_{i=0}^{n-1} \sum_{j=0}^{n-1} n = \sum_{i=0}^{n-1} n^2 = \Theta(n^3)$$

The total running time is the maximum of the running time of the individual fragments

```
sum = 0;
for( i = 0; i < n; i++)
        sum = sum + i;  sum = 0;
for( i = 0; i < n; i++)
        for( j = 0; j < 2n; j++)
                sum++;
```

The first loop runs in $\Theta(n)$ time,

the second $\Theta(n^2)$ time, the maximum is $\Theta(n^2)$

Mathematical analysis of Recursive algorithms

Steps in mathematical analysis of recursive algorithms:

- Decide on parameter n indicating input size
- Identify algorithm‘s basic operation
- Determine worst, average, and best case for input of size n
- Set up a recurrence relation and initial condition(s) for T(n)-the number of times the basic operation will be executed for an input of size n (alternatively count recursive calls).
- Solve the recurrence to obtain a closed form or estimate the order of magnitude of the solution

Example: Analysis of recursive algorithm to find factorial of a given number

ALGORITHM $F(n)$

//Computes $n!$ recursively
//Input: A nonnegative integer n
//Output: The value of $n!$
if $n = 0$ **return** 1
else return $F(n-1) * n$

The basic operation of the algorithm is multiplication, whose number of executions we denote *M(n).*

$$M(n) = \underbrace{M(n-1)}_{\text{to compute } F(n-1)} + \underbrace{1}_{\text{to multiply } F(n-1) \text{ by } n} \quad \text{for } n > 0.$$

$M(0) = 0.$

the calls stop when $n = 0$ ← → no multiplications when $n = 0$

Recurrence relation and initial condition for the algorithm's number of multiplications *M(n)*:

$$M(n) = M(n-1) + 1 \quad \text{for } n > 0,$$
$$M(0) = 0.$$

Example: Factorial

$$M(n) = M(n-1) + 1 \quad \text{for } n > 0,$$
$$M(0) = 0.$$

Solving the above recurrence relation using the *method of* ***backward substitutions***.

$$M(n) = M(n-1) + 1 \qquad \text{substitute } M(n-1) = M(n-2) + 1$$
$$= [M(n-2) + 1] + 1 = M(n-2) + 2 \qquad \text{substitute } M(n-2) = M(n-3) + 1$$
$$= [M(n-3) + 1] + 2 = M(n-3) + 3.$$

$$= M(n-i) + i$$

$$= M(n-n) + n$$

M(n)=n Hence the time complexity of recursive factorial algorithm is T(n) = $\ominus$(n)

Solve the following recurrence relations using the Substitution method

a. $x(n) = x(n-1) + 5 \quad \text{for } n > 1, \quad x(1) = 0$

b. $x(n) = 3x(n-1) \quad \text{for } n > 1, \quad x(1) = 4$

d. $x(n) = x(n/2) + n \quad \text{for } n > 1, \quad x(1) = 1 \quad (\text{solve for } n = 2^k)$

Solution

a. $x(n) = x(n-1) + 5 \quad \text{for } n > 1, \quad x(1) = 0$

$$\begin{aligned} x(n) &= x(n-1) + 5 \\ &= [x(n-2) + 5] + 5 = x(n-2) + 5 \cdot 2 \\ &= [x(n-3) + 5] + 5 \cdot 2 = x(n-3) + 5 \cdot 3 \\ &= \ldots \\ &= x(n-i) + 5 \cdot i \\ &= \ldots \\ &= x(1) + 5 \cdot (n-1) = 5(n-1). \end{aligned}$$

Note: The solution can also be obtained by using the formula for the n term of the arithmetical progression:

$$x(n) = x(1) + d(n-1) = 0 + 5(n-1) = 5(n-1).$$

b. $x(n) = 3x(n-1)$ for $n > 1$, $x(1) = 4$

$$\begin{aligned} x(n) &= 3x(n-1) \\ &= 3[3x(n-2)] = 3^2 x(n-2) \\ &= 3^2[3x(n-3)] = 3^3 x(n-3) \\ &= \ldots \\ &= 3^i x(n-i) \\ &= \ldots \\ &= 3^{n-1} x(1) = 4 \cdot 3^{n-1}. \end{aligned}$$

Note: The solution can also be obtained by using the formula for the n term of the geometric progression:

$$x(n) = x(1)q^{n-1} = 4 \cdot 3^{n-1}.$$

d. $x(n) = x(n/2) + n \quad \text{for } n > 1, \quad x(1) = 1 \quad (\text{solve for } n = 2^k)$

$$
\begin{aligned}
x(2^k) &= x(2^{k-1}) + 2^k \\
&= [x(2^{k-2}) + 2^{k-1}] + 2^k = x(2^{k-2}) + 2^{k-1} + 2^k \\
&= [x(2^{k-3}) + 2^{k-2}] + 2^{k-1} + 2^k = x(2^{k-3}) + 2^{k-2} + 2^{k-1} + 2^k \\
&= \ldots \\
&= x(2^{k-i}) + 2^{k-i+1} + 2^{k-i+2} + \ldots + 2^k \\
&= \ldots \\
&= x(2^{k-k}) + 2^1 + 2^2 + \ldots + 2^k = 1 + 2^1 + 2^2 + \ldots + 2^k \\
&= 2^{k+1} - 1 = 2 \cdot 2^k - 1 = 2n - 1.
\end{aligned}
$$

Excercise

Solve the following recurrence relations.

c. $x(n) = x(n-1) + n \quad \text{for } n > 0, \quad x(0) = 0$

e. $x(n) = x(n/3) + 1 \quad \text{for } n > 1, \quad x(1) = 1 \quad (\text{solve for } n = 3^k)$

Solution to Exercises:

c. $x(n) = x(n-1) + n \quad \text{for } n > 0, \quad x(0) = 0$

$$\begin{aligned}
x(n) &= x(n-1) + n \\
&= [x(n-2) + (n-1)] + n = x(n-2) + (n-1) + n \\
&= [x(n-3) + (n-2)] + (n-1) + n = x(n-3) + (n-2) + (n-1) + n \\
&= \ldots \\
&= x(n-i) + (n-i+1) + (n-i+2) + \ldots + n \\
&= \ldots \\
&= x(0) + 1 + 2 + \ldots + n = \frac{n(n+1)}{2}.
\end{aligned}$$

e. $x(n) = x(n/3) + 1 \quad \text{for } n > 1, \quad x(1) = 1 \quad (\text{solve for } n = 3^k)$

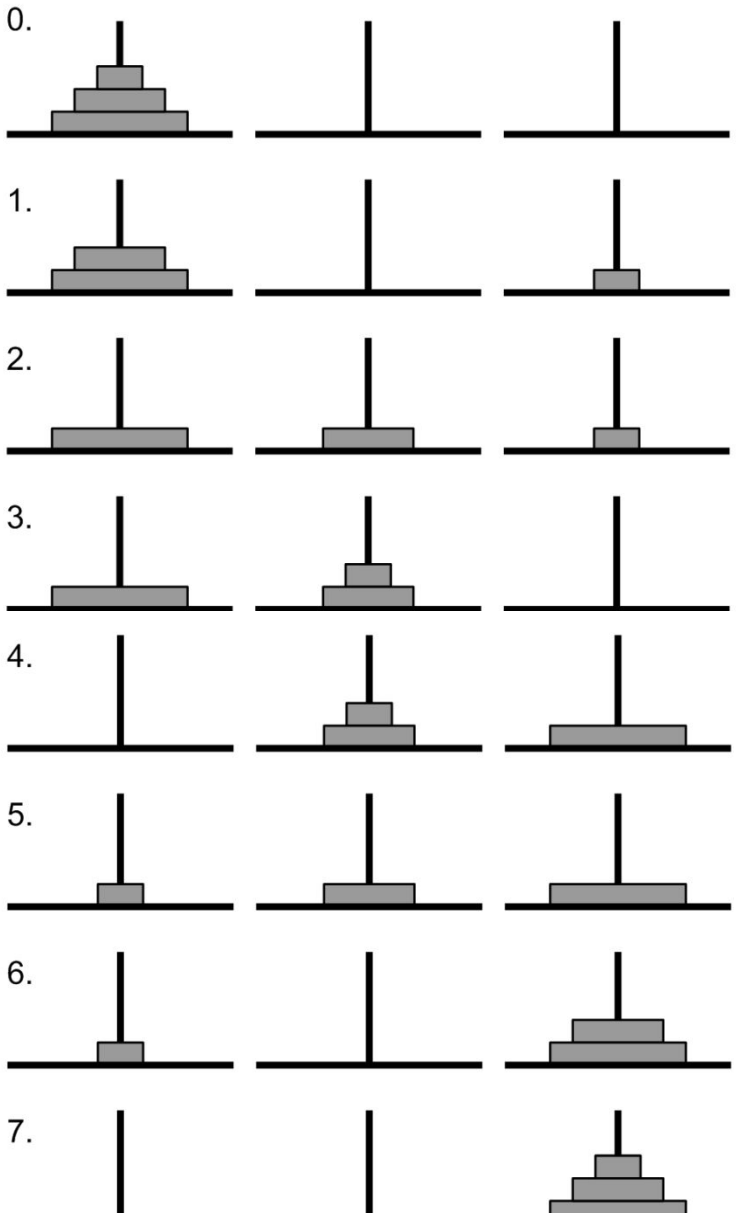
$$\begin{aligned}
x(3^k) &= x(3^{k-1}) + 1 \\
&= [x(3^{k-2}) + 1] + 1 = x(3^{k-2}) + 2 \\
&= [x(3^{k-3}) + 1] + 2 = x(3^{k-3}) + 3 \\
&= \ldots \\
&= x(3^{k-i}) + i \\
&= \ldots \\
&= x(3^{k-k}) + k = x(1) + k = 1 + \log_3 n.
\end{aligned}$$

Example: Tower of Hanoi

- There are three pegs, Source(A), Auxiliary(B) and Destination(C). Peg A contains a set o disks stacked to resemble a tower, with the largest disk at the bottom and the smallest disk at the top. The objective is to transfer the entire tower of disks in peg A to peg C maintaining the same order of the disks.

Obeying the following rules:

- Only one disk can be transfer at a time.
- Each move consists of taking the upper disk from one of the peg and placing it on the top of another peg i.e. a disk can only be moved if it is the uppermost disk of the peg.
- Never a larger disk is placed on a smaller disk during the transfer.



- Move **(n-1)** discs from the source post to the auxiliary post.
- Move the **last** disc to the destination post.
- Move **(n-1)** discs back from the auxiliary post to the destination post.

The number of disks *n* is the obvious choice for the input's size indicator, and so is moving one disk as the algorithm's basic operation. The number of moves *M(n)* depends on *n* only, and we get the following recurrence equation for it:

$$M(n) = M(n-1) + 1 + M(n-1) \text{ for } n > 1.$$

With the obvious initial condition $M(1) = 1$, we have the following recurrence relation for the number of moves *M(n)*:

$$M(n) = 2M(n-1) + 1 \quad \text{for } n > 1$$

$$M(1) = 1 \qquad \text{for n=1}$$

$M(n) = 2M(n-1) + 1$ for $n > 1$

$M(1) = 1$ for n=1

Solving the above recurrence relation using the *method of* ***backward substitutions***.

$$M(n) = 2M(n-1) + 1 \qquad \text{sub. } M(n-1) = 2M(n-2) + 1$$
$$= 2[2M(n-2) + 1] + 1 = 2^2 M(n-2) + 2 + 1 \qquad \text{sub. } M(n-2) = 2M(n-3) + 1$$
$$= 2^2[2M(n-3) + 1] + 2 + 1 = 2^3 M(n-3) + 2^2 + 2 + 1.$$

The pattern of the first three sums on the left suggests that the next one will be $2^4 M(n-4) + 2^3 + 2^2 + 2 + 1$, and generally, after i substitutions, we get

$$M(n) = 2^i M(n-i) + 2^{i-1} + 2^{i-2} + \cdots + 2 + 1 = 2^i M(n-i) + 2^i - 1.$$

Since the initial condition is specified for $n = 1$, which is achieved for $i = n - 1$, we get the following formula for the solution to recurrence

$$M(n) = 2^{n-1} M(n-(n-1)) + 2^{n-1} - 1$$
$$= 2^{n-1} M(1) + 2^{n-1} - 1 = 2^{n-1} + 2^{n-1} - 1 = 2^n - 1.$$

Hence, $T(n) = \ominus(2^n)$

Time and Space Complexity

- Designing an efficient algorithm for a program plays a crucial role in a large scale computer system.
- **Time complexity** and **space complexity** are the two most important considerations for deciding the efficiency of an algorithm.
- The **time complexity** of an algorithm is the number of instructions that it needs to run to completion.
- The **space complexity** of an algorithm is the amount of memory that it needs to run to completion.
- The analysis of **running time** generally has received more attention than **memory** because any program that uses huge amounts of memory automatically requires a lot of time.

Time Complexity

- In analyzing algorithm we will not consider the following information although they are very important.
  1. The machine we are executing on.
  2. The machine language instruction set.
  3. The time required by each machine instruction
  4. The translation, a compiler will make from the source to the machine language.
- The **exact time** we determine would no apply to many machines.
- There would be the problem of the **compiler** which could vary from machine to machine.
- It is often difficult to get reliable timing figures because of **clock limitations** and a **multi-programming** or **time sharing** environment.
- We will concentrate on developing only the **frequency count** for all statements.

Time Complexity

```
1: x ← x + 1                                   ▷ Frequency count is 1
```

```
1: for I ← 1 to n do                           ▷ Frequency count is n+1
2:     x ← x + 1;                              ▷ Frequency count is n
3: end for
```

```
1: for I ← 1 to n do                           ▷ Frequency count is n+1
2:     for J ← 1 to n do                       ▷ Frequency count is n(n+1)
3:         x ← x + 1;                          ▷ Frequency count is n^2
4:     end for
5: end for
```

Space Complexity

The space needed by a program is the sum of the following components.

- **Fixed space requirement:** The component refers to space requirement that do not depend on the number and size of the program's **inputs** and **outputs**. The fixed requirements include the **instruction space** (space needed to store the code), space for **simple variables**, **fixed size structured variable and constants**.
- **Variable space requirement:** This component consists of the space needed by structured variables whose **size** depends on the particular **instance i**, of the problem being solved. It also includes the **additional space** required when a function uses **recursion**.

The space requirement **S(P)** of an algorithm P may therefore be written as $\mathbf{S(P) = c + S_P}$, where **c** and $\mathbf{S_P}$ are the **constant** and **instance characteristics**, respectively. First, we need to determine which instance characteristics to use for a give problem to reduce the space requirements.

Space Complexity

Algorithm 3 Sum of array elements

```
1: procedure CALCULATE_SUM(A, n)
2:     sum ← 0
3:     for i ← 0 to n − 1 do
4:         sum ← sum + A[i]
5:     end for
6: end procedure
```

n, **sum** and i take constant sum of **3 units**, but the variable **A** is an array, it's **space** consumption **increases** with the increase of input size **n**.