Subject Code : MC03094011
Subject Name : Design and Analysis of Algorithms
Unit – 1 Introduction

PREPARED BY:

YOGESHWARI M. LOHAR

Introduction

- ❑An **algorithm** is a set of steps of operations to solve a problem performing calculation, data processing, and automated reasoning tasks.
- ❑ An algorithm is an efficient method that can be expressed with in **finite amount of time and space**.
- ❑An algorithm is the best way to represent the solution of a particular problem in a very **simple and efficient way**.
- ❑ If we have an algorithm for a specific problem, then we can implement it in any programming language, meaning that the **algorithm is independent from any programming languages**.

OR

- An algorithm is a distinct computational procedure that takes input as a set of values and results in the output as a set of values by solving the problem.
- More precisely, an algorithm is correct, if, for each input instance, it gets the correct output and gets terminated.

Design & Analysis of Algorithms

"**algos**" = Greek word for pain.

"**algor**" = Latin word for to be cold.

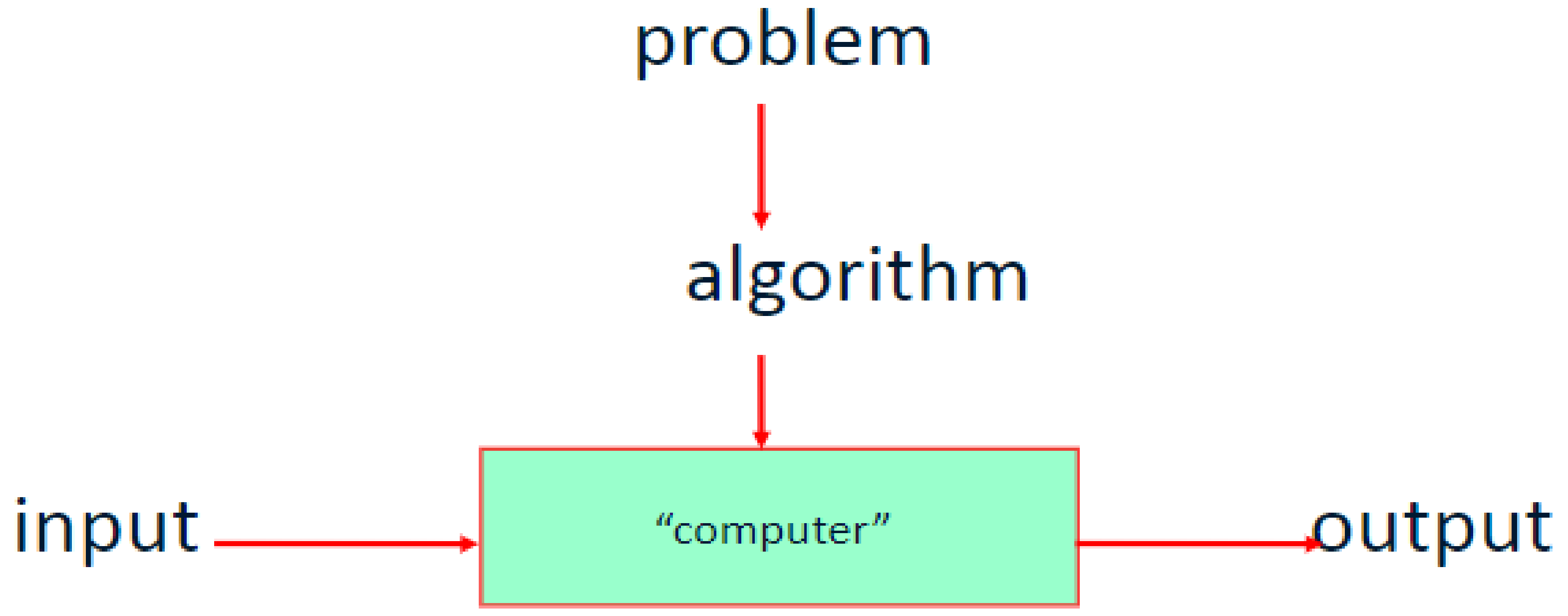
▪The "Concise Oxford Dictionary" defines **"Algorithm" (also 'Algorism")** as: a process or a set of rules used for calculation or problem solving, especially with a computer.

Why study this subject?

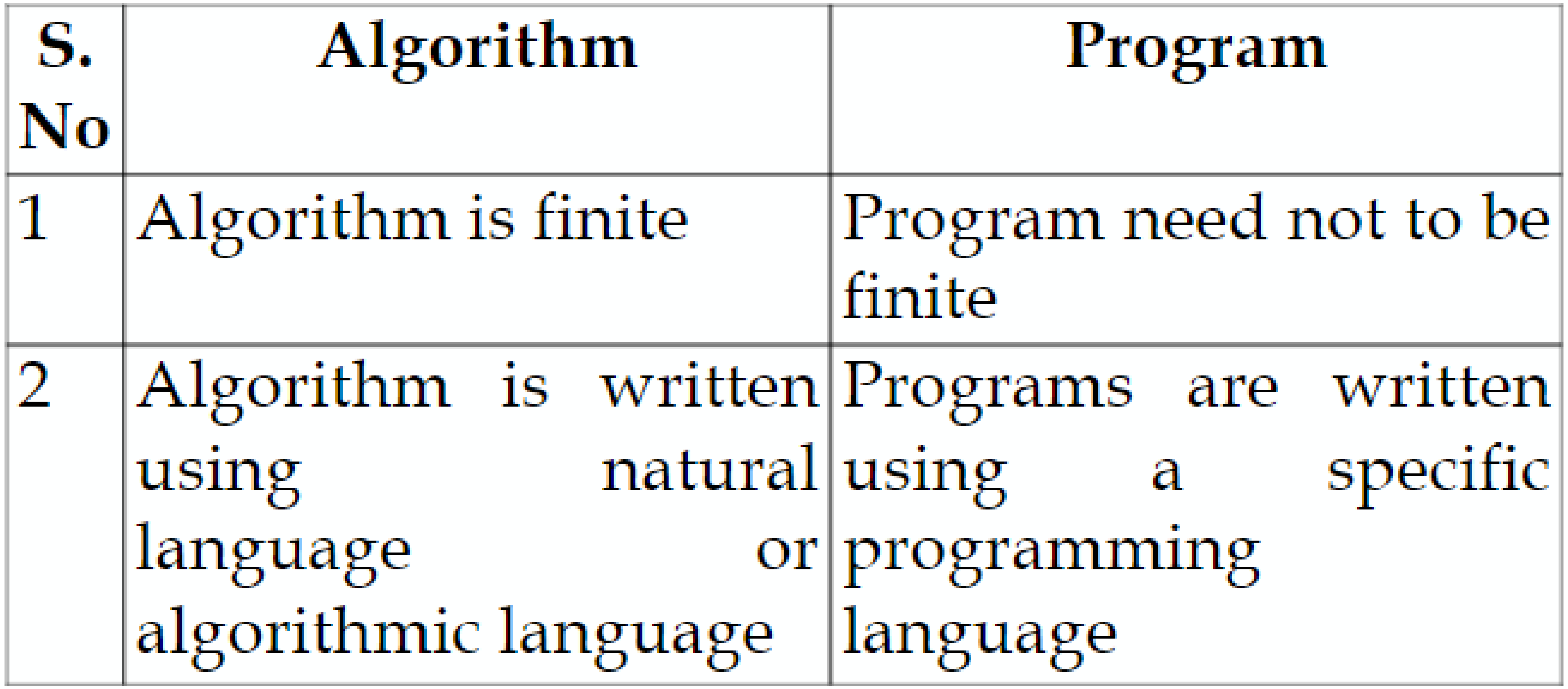
- Efficient algorithms lead to efficient programs.
- Efficient programs sell better.
- Efficient programs make better use of hardware.
- Programmers who write efficient programs are preferred.

What is an algorithm?

An ***algorithm*** is a list of steps (sequence of unambiguous instructions) for solving a problem that transforms the input into the output.



Difference between Algorithm and Program



S. No	Algorithm	Program
1	Algorithm is finite	Program need not to be finite
2	Algorithm is written using natural language or algorithmic language	Programs are written using a specific programming language

Characteristics of Algorithms

- **Input:** It should externally supply zero or more quantities.
- **Output:** It results in at least one quantity.
- **Definiteness:** Each instruction should be clear and ambiguous.
- **Finiteness:** An algorithm should terminate after executing a finite number of steps.
- **Effectiveness:** Every instruction should be fundamental to be carried out, in principle, by a person using only pen and paper.
- **Feasible:** It must be feasible enough to produce each instruction.
- **Flexibility:** It must be flexible enough to carry out desired changes with no efforts.

- **Efficient:** The term efficiency is measured in terms of time and space required by an algorithm to implement. Thus, an algorithm must ensure that it takes little time and less memory space meeting the acceptable limit of development time.
- **Independent:** An algorithm must be language independent, which means that it should mainly focus on the input and the procedure required to derive the output instead of depending upon the language.

Advantages of an Algorithm

- **Effective Communication:** Since it is written in a natural language like English, it becomes easy to understand the step-by-step delineation of a solution to any particular problem.
- **Easy Debugging:** A well-designed algorithm facilitates easy debugging to detect the logical errors that occurred inside the program.
- **Easy and Efficient Coding:** An algorithm is nothing but a **blueprint** of a program that helps develop a program.
- **Independent of Programming Language:** Since it is a language-independent, it can be easily coded by incorporating any high-level language.

Disadvantages of an Algorithm

- Developing algorithms for complex problems would be **time-consuming and difficult to understand**.
- It is a challenging task to understand **complex logic** through algorithms.

Study of Algorithms

There are five distinct areas of study of algorithms:

- **how to devise algorithms?** – strategies
- **how to express an algorithm?** flow-chart, pseudo-code, a program, etc.
- **how to validate an algorithm?** test if it satisfies its precise specifications, and check for its correctness.
- **how to analyze an algorithm?** analysis of algorithms, the amount of computer time and storage that is required.
- **how to test a program?** debugging and profiling. **E.W.Dijkstra** : "debugging can only point to the presence of errors and never their absence."

Pseudo code:

•It’s simply an implementation of an algorithm in the form of annotations and informative text written in plain English.

•It has no syntax like any of the programming language and thus can’t be compiled or interpreted by the computer.

Pseudocode to find the area of a Rectangle is as follows:

```
AreaOfRectangle()
Begin
        Read: width, length;
        Set area = width * length;
        Print area;
End
```

Algorithm:

- Define the width of the rectangle.
- Define the Height of the rectangle.
- Define Area of the rectangle.
- Calculate the area of the rectangle by multiplying the width and height of the rectangle.
- Assign the area of the rectangle to the area variable.
- print the area of the rectangle.

Problem: Suppose there are 60 students in the class. How will you calculate the number of absentees in the class?

Algorithmic Approach:

- Initialize a variable called as **Count** to zero, **absent** to zero, **total** to 60
- FOR EACH Student PRESENT DO the following:

  Increase the **Count** by One
- Then Subtract **Count** from **total** and store the result in **absent**
- Display the number of absent students

Pseudo Approach:

```
•Count <- 0, absent <- 0, total <- 60
•REPEAT till all students counted
Count <- Count + 1
•absent <- total - Count
•Print "Number absent is:" , absent
```

Basic steps to solve a problem

The development of an algorithm (a plan) is a key step in solving a problem. Once we have an algorithm, we can translate it into a computer program in some programming language. Our algorithm development process consists of five major steps.

Step 1: Obtain a description of the problem.

Step 2: Analyze the problem.

Step 3: Develop a high-level algorithm.

Step 4: Refine the algorithm by adding more detail.

Step 5: Review the algorithm.

Step-1 Obtain a description of the problem.

Problem description may suffer from one or more of the following types of defects:

(1) the description relies on unstated assumptions,

(2) the description is ambiguous,

(3) the description is incomplete, or

(4) the description has internal contradictions.

Step-2 Analyze the problem.

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

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

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

How will we know when we're done? Asking the following questions often helps to determine the ending point.

What new facts will we have?

What items will have changed?

What changes will have been made to those items?

What things will no longer exist?

Step-3 Develop a high-level algorithm.

Problem: I need a send a birthday card to my brother

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

High-level algorithm:

1. Go to a store that sells greeting cards
2. Select a card
3. Purchase a card
4. Mail the card

Step-4 Refine the algorithm by adding more detail.

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

The above algorithm lacks details, These details include answers to questions such as the following.

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

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

Step-5 Review the algorithm.

The final step is to review the algorithm.

Does this algorithm solve a **very specific problem** or does it solve a **more general problem**? If it solves a very specific problem, should it be generalized?

formula $\pi * 5.2^2$

formula $\pi * R^2$

Can this algorithm be **simplified**?

length + width + length + width

2.0 * (*length + width*)

Is this solution **similar** to the solution to another problem? How are they alike? How are they different?

For example, consider the following two formulae:

Rectangle area = *length* * *width*
Triangle area = 0.5 * *base* * *height*

Similarities: Each computes an area. Each multiplies two measurements.

Differences: Different measurements are used. The triangle formula contains 0.5.

Write an algorithm to add two numbers entered by user.

```
Step 1: Start
Step 2: Declare variables num1, num2 and sum.
Step 3: Read values num1 and num2.
Step 4: Add num1 and num2 and assign the result to sum.
        sum←num1+num2
Step 5: Display sum
Step 6: Stop
```

Write an algorithm to find the largest among three different numbers entered by user.

```
Step 1: Start
Step 2: Declare variables a,b and c.
Step 3: Read variables a,b and c.
Step 4: If a>b && a>c
            Display a is the largest number.
        Else
            Display c is the largest number.
        Else
            If b>c
                Display b is the largest number.
            Else
                Display c is the greatest number.
Step 5: Stop
```

Write an algorithm to find the factorial of a number entered by user.

```
Step 1: Start
Step 2: Declare variables n,factorial and i.
Step 3: Initialize variables
        factorial←1
        i←1
Step 4: Read value of n
Step 5: Repeat the steps until i=n
    5.1: factorial←factorial*i
    5.2: i←i+1
Step 6: Display factorial
Step 7: Stop
```

Analysis of algorithm

The analysis is a process of estimating the efficiency of an algorithm. There are two fundamental parameters based on which we can analysis the algorithm:

Space Complexity: The space complexity can be understood as the amount of space required by an algorithm to run to completion.

Time Complexity: Time complexity is a function of input size **n** that refers to the amount of time needed by an algorithm to run to completion.

•Before writing any program in C, we should implement it in English language for it to be more readable and understandable before implementing it, which is nothing but the **concept of Algorithm.**

e.g. Suppose problem is **P1,**

Solutions might be **A1, A2, A3, ……. An**

Here, P=Problem and A=Algorithm

•Before you implement any algorithm as a program, it is better to find out which among these algorithms are good **<u>in terms of time and memory.</u>**

•Here we will **<u>focus on time than that of the space.</u>**

•When we run the same algorithm on a **different computer** or use **different programming languages**, we will encounter that **the same algorithm takes a different time**.

Generally, there are 3 types of analysis, which is as follows:

1. **Worst-case time complexity**
2. **Average case time complexity**
3. **Best case time complexity**

1. **Worst-case time complexity:** For '**n**' input size, the worst-case time complexity can be defined as the maximum amount of time needed by an algorithm to complete its execution. Thus, it is nothing but a function defined by the maximum number of steps performed on an instance having an input size of n.

2. **<u>Average case time complexity</u>:** For '**n**' input size, the average-case time complexity can be defined as the average amount of time needed by an algorithm to complete its execution. Thus, it is nothing but a function defined by the average number of steps performed on an instance having an input size of n.

3. **<u>Best case time complexity:</u>** For '**n**' input size, the best-case time complexity can be defined as the minimum amount of time needed by an algorithm to complete its execution. Thus, it is nothing but a function defined by the minimum number of steps performed on an instance having an input size of n.

Role of Algorithms in Computing

- Provide step-by-step solutions to problems
- Enhance system efficiency and performance
- Automate tasks in software and hardware
- Essential in AI, cyber security, data processing, and science

Algorithms as a Technology

- Core of all software systems
- Enable scalable and platform-independent solutions
- Backbone of AI, Block chain, and IoT
- Proprietary algorithms offer commercial advantages

Designing of algorithms

1. Divide and Conquer Approach
2. Greedy Technique
3. Dynamic Programming
4. Branch and Bound
5. Randomized Algorithms
6. Backtracking Algorithm

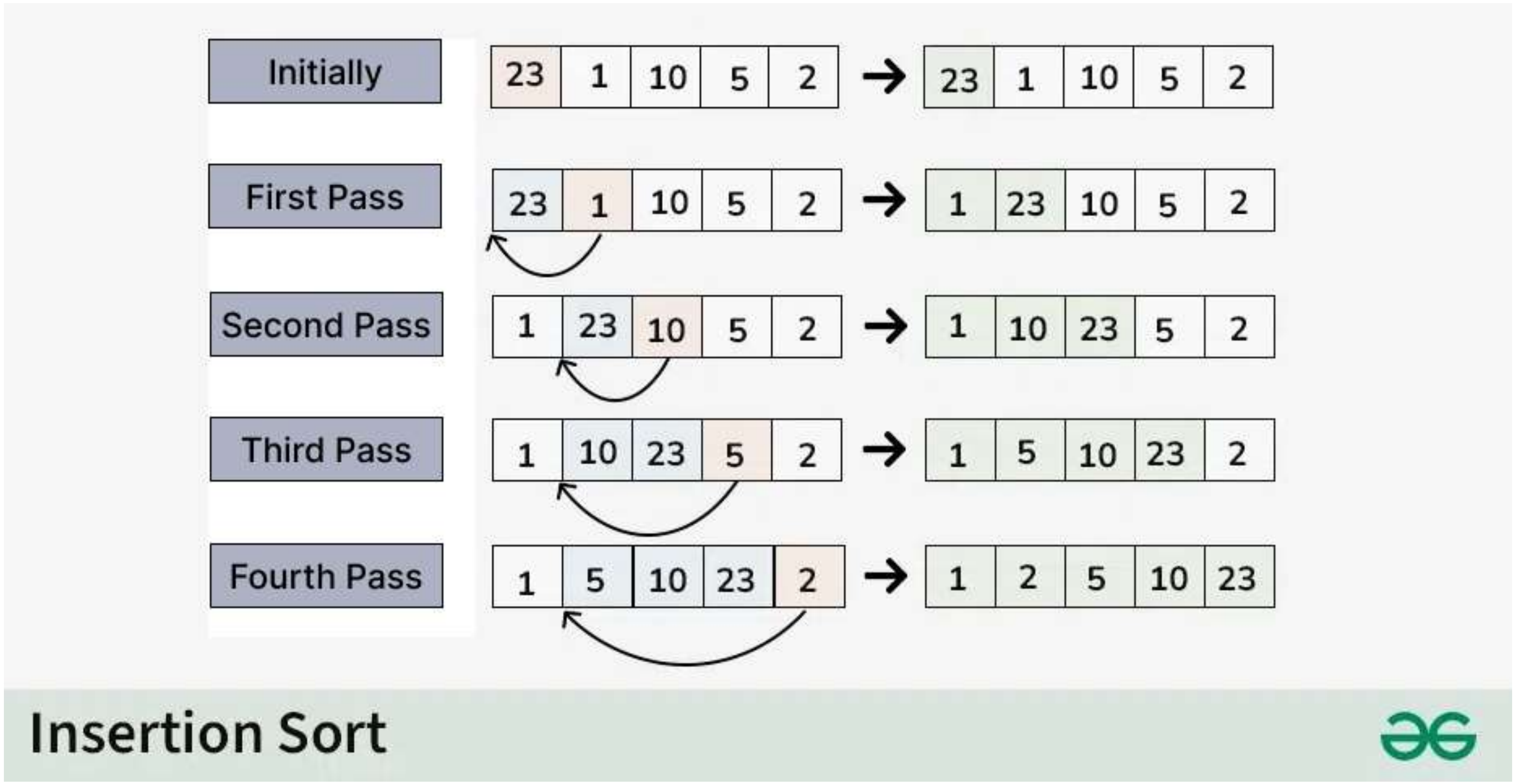
$\left.\begin{array}{l}1\\2\\3\end{array}\right\}$ Primary three main categories

Insertion sort

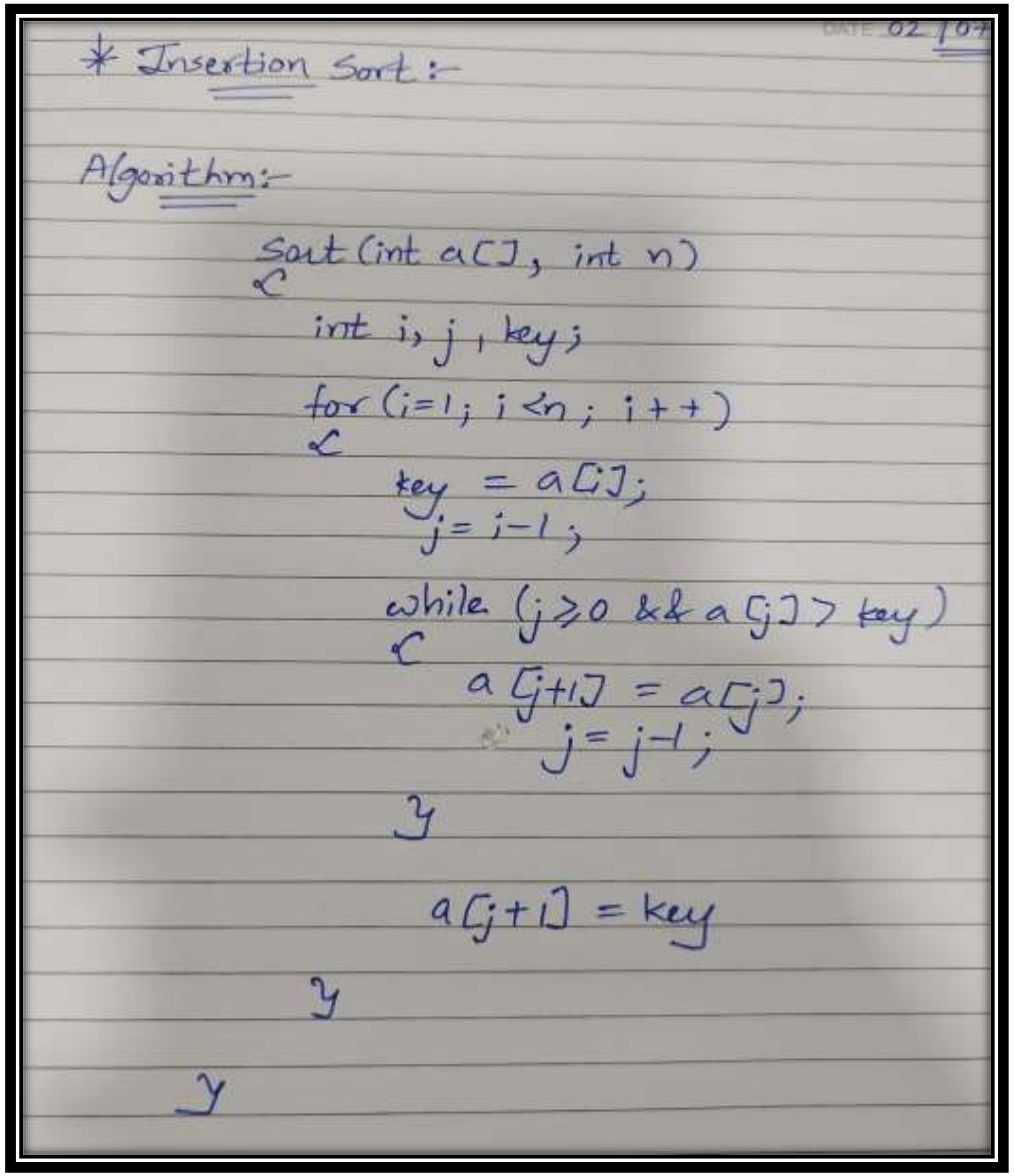
Insertion sort is a simple sorting algorithm that works by iteratively inserting each element of an unsorted list into its correct position in a sorted portion of the list. It is like sorting playing cards in your hands. You split the cards into two groups: the sorted cards and the unsorted cards. Then, you pick a card from the unsorted group and put it in the right place in the sorted group.

•We start with the second element of the array as the first element is assumed to be sorted.
•Compare the second element with the first element if the second element is smaller then swap them.
•Move to the third element, compare it with the first two elements, and put it in its correct position
•Repeat until the entire array is sorted.

Insertion sort



Insertion sort



DATE 02/07

* Insertion Sort :-

Algorithm:-

```
Sort (int a[], int n)
{
   int i, j, key;
   for (i=1; i<n; i++)
   {
      key = a[i];
      j = i-1;

      while (j>=0 && a[j] > key)
      {
         a[j+1] = a[j];
           j = j-1;

      }

        a[j+1] = key

   }

}
```

Asymptotic notation

- ✓Mathematical way of representing the time complexity
- ✓We need some notations:

1. **Big – Oh (O)** $\longrightarrow$ **Upper Bound of Function**
2. **Big – Omega (Ω)** $\longrightarrow$ **Lower Bound of Function**
3. **Theta – (θ)** $\longrightarrow$ **Average Bound of Function**

1. Big - Oh (O) Notation:

$$f(n) \le c\, g(n)$$

2. Big Omega (Ω) Notation:

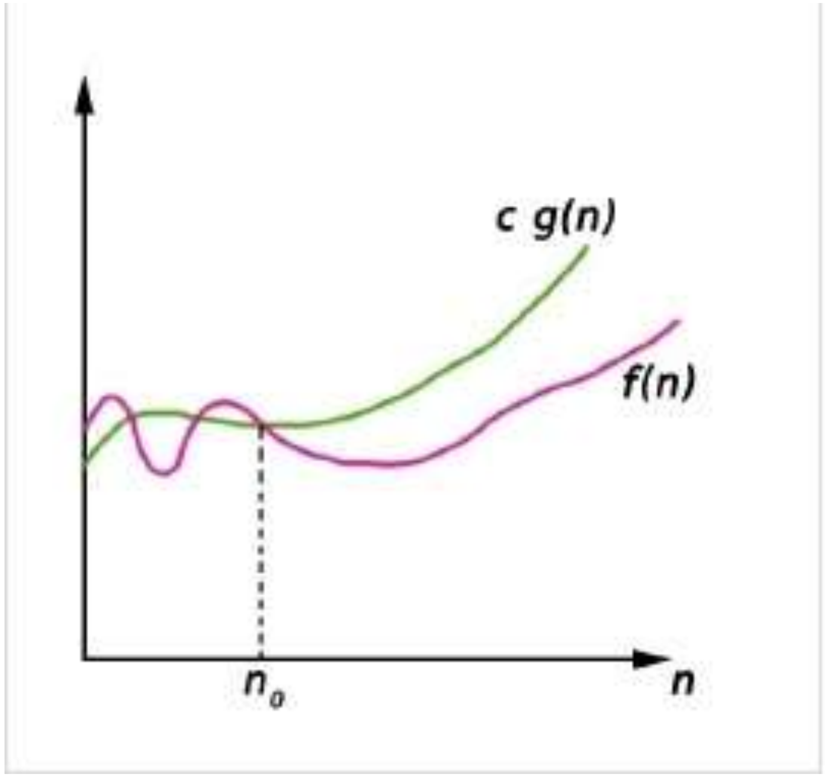
$$f(n) \ge c\, g(n)$$

3. Theta – (θ) Notation:

$$c_1\, g(n) \le f(n) \le c_2\, g(n)$$

1. Big - Oh (O) Notation:

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



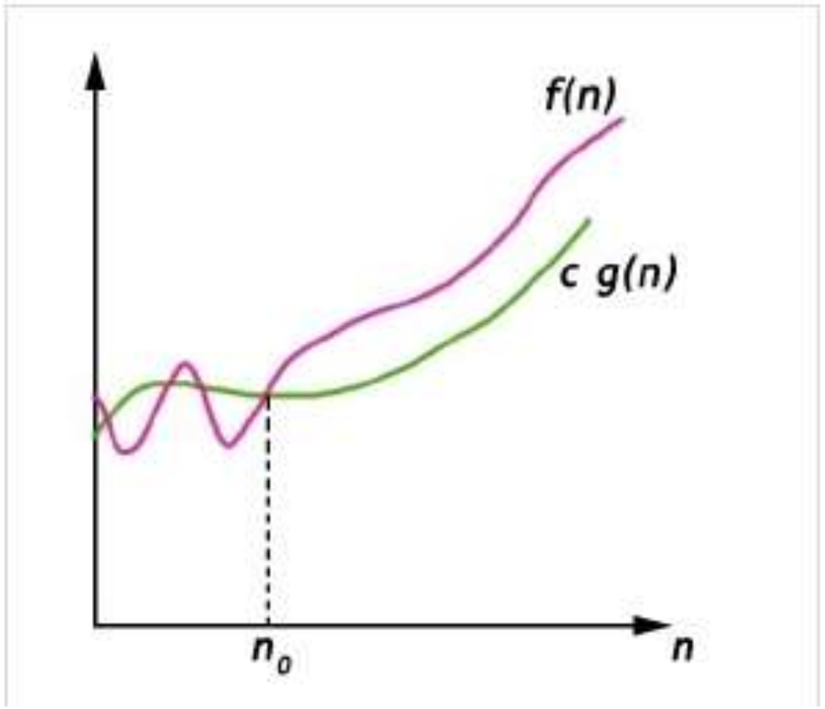
•Big-Oh (O) notation gives an **upper bound** for a function f(n) to within a constant factor.

•We write **f(n) = O(g(n)),** If there are positive constants n0 and c such that, to the right of n0 the f(n) always lies on or below c*g(n).

•O(g(n)) = { f(n) : There exist positive constant c and n0 such that $0 \leq f(n) \leq c\, g(n)$, for all $n \leq n0$}

2. Big Omega (Ω) Notation:

$f(n) \geq c\ g(n)$



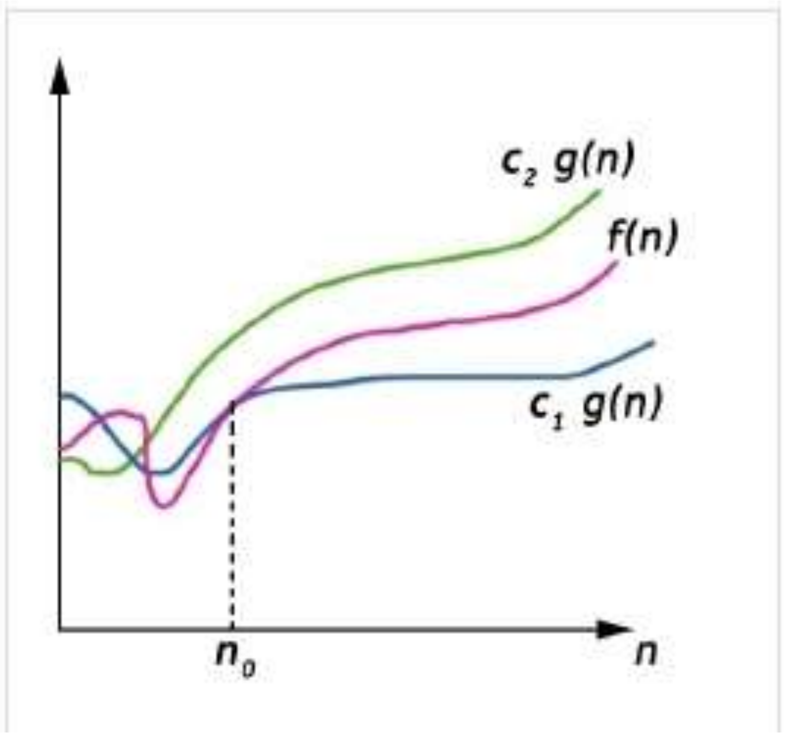
- Big-Omega (Ω) notation gives a **lower bound** for a function f(n) to within a constant factor.

- We write **f(n) = Ω(g(n)),** If there are positive constants n0 and c such that, to the right of n_0 the f(n) always lies on or above c*g(n).

- Ω(g(n)) = { f(n) : There exist positive constant c and n0 such that $0 \leq c\ g(n) \leq f(n)$, for all $n \leq n_0$}
-

3. Theta – (θ) Notation:

$c_1\, g(n) \le f(n) \le c_2\, g(n)$



- Big-Theta(Θ) notation gives bound**(average bound)** for a function f(n) to within a constant factor.

- We write **f(n) = Θ(g(n)),** If there are positive constants n0 and c_1 and c_2 such that, to the right of n_0 the f(n) always lies between c_1*g(n) and c_2*g(n) inclusive.

- Θ(g(n)) = {f(n) : There exist positive constant c_1, c_2 and n_0 such that $0 \le c_1\, g(n) \le f(n) \le c_2\, g(n)$, for all $n \ge n_0$}
-

4. Little - oh (o) Notations (slower growth rate):

•Little o notation is used to describe an upper bound that cannot be tight. In other words, loose upper bound of f(n).

•Let f(n) and g(n) are the functions that map positive real numbers. We can say that the function f(n) is o(g(n)) if for any real positive constant c, there exists an integer constant n0 ≤ 1 such that f(n) > 0.

•Using mathematical relation, we can say that **f(n) = o(g(n))** means,

$$\lim_{n \to \infty} \frac{f(n)}{g(n)} = 0$$

5. Little omega (ω) Notations (faster growth rate):

- Another asymptotic notation is little omega notation. it is denoted by (ω).
- Little omega (ω) notation is used to describe a loose lower bound of f(n).
- The relation **f(n)=ω(g(n))** implies that the following limit exists

$$\lim_{n\to\infty}\left(\frac{f(n)}{g(n)}\right)=\infty$$

That is, ***f(n)*** becomes arbitrarily large relative to ***g(n)*** as **n** approaches infinity.

Standard Notations of common functions:

1. Monotonicity:

- A function $f(n)$ is ***monotonically increasing*** if $m \le n$ implies $f(m) \le f(n)$.
- Similarly, it is ***monotonically decreasing*** if $m \le n$ implies $f(m) \ge f(n)$.
- A function $f(n)$ is ***strictly increasing*** if $m < n$ implies $f(m) < f(n)$ and ***strictly decreasing*** if $m < n$ implies $f(m) > f(n)$.

2. Floors and ceilings

For any real number x, we denote the greatest integer less than or equal to x by **$\lfloor x \rfloor$ (read "the floor of x")** and the least integer greater than or equal to x by **$\lceil x \rceil$ (read "the ceiling of x")**. For all real x,

$$x - 1 < \lfloor x \rfloor \le x \le \lceil x \rceil < x + 1$$

For any integer n,

$\lceil n/2 \rceil + \lfloor n/2 \rfloor = n$, and for any real number $n \ge 0$ and integers $a, b > 0$,

The floor function $f(x) = \lfloor x \rfloor$ is monotonically increasing, as is the ceiling function $f(x) = \lceil x \rceil$.

3. Modular arithmetic:

For any integer *a* and any positive integer *n*, the value *a* mod *n* is the ***remainder*** (or ***residue***) of the quotient *a*/*n*:

$$a \bmod n = a - \lfloor a/n \rfloor n$$

•Given a well-defined notion of the remainder of one integer when divided by another, it is convenient to provide special notation to indicate equality of remainders. If (a mod n) = (b mod n), we write $a \equiv b$ (mod n) and say that *a* is ***equivalent*** to *b*, modulo *n*.

•In other words, **$a \equiv b$ (mod *n*)** if *a* and *b* have the same remainder when divided by *n*. Equivalently, $a \equiv b$ (mod n) if and only if *n* is a divisor of $b - a$. We write $a \not\equiv b$ (mod n) if *a* is not equivalent to *b*, modulo *n*.

4. Polynomials:

Given a nonnegative integer *d*, a ***polynomial in n of degree d*** is a function $p(n)$ of the form

$$p(n) = \sum_{i=0}^{d} a_i n^i \ ,$$

•where the constants $a_0, a_1, \ldots, a_d$ are the ***coefficients*** of the polynomial and $a_d \neq 0$. A polynomial is asymptotically positive if and only if $a_d > 0$.

• For an asymptotically positive polynomial $p(n)$ of degree *d*, we have $p(n) = \Theta(n^d)$. For any real constant $a \geq 0$, the function n^a is monotonically increasing, and for any real constant $a \leq 0$, the function n^a is monotonically decreasing.

•We say that a function $f(n)$ is ***polynomially bounded*** if $f(n) = O(n^k)$ for some constant *k*.

5. **Exponentials:**

- For all real $a > 0$, m, and n, we have the following identities:
- For all n and $a \geq 1$, the function a^n is monotonically increasing in n. When convenient, we shall assume $0^0 = 1$.

$$\begin{aligned} a^0 &= 1, \\ a^1 &= a, \\ a^{-1} &= 1/a, \\ (a^m)^n &= a^{mn}, \\ (a^m)^n &= (a^n)^m, \\ a^m a^n &= a^{m+n}. \end{aligned}$$

6. Logarithms:

We shall use the following notations:

$\lg n$	=	$\log_2 n$	(binary logarithm) ,
$\ln n$	=	$\log_e n$	(natural logarithm) ,
$\lg^k n$	=	$(\lg n)^k$	(exponentiation) ,
$\lg \lg n$	=	$\lg(\lg n)$	(composition) .

An important notational convention we shall adopt is that *logarithm functions will apply only to the next term in the formula*, so that $\lg n + k$ will mean $(\lg n) + k$ and not $\lg(n + k)$.

If we hold $b > 1$ constant, then for $n > 0$, the function $\log_b n$ is strictly increasing.

For all real $a > 0$, $b > 0$, $c > 0$, and n,

$$a = b^{\log_b a} \qquad \log_b a = \frac{\log_c a}{\log_c b}$$

7. Factorials:

The notation $n!$ (read "n factorial") is defined for integers $n \geq 0$ as

$$n! = \begin{cases} 1 & \text{if } n = 0 , \\ n \cdot (n-1)! & \text{if } n > 0 . \end{cases}$$

Thus, $n! = 1 \cdot 2 \cdot 3 \, n$.

8. Fibonacci numbers:

The ***Fibonacci numbers*** are defined by the following recurrence:

$$\begin{aligned} F_0 &= 0 , \\ F_1 &= 1 , \\ F_i &= F_{i-1} + F_{i-2} \qquad \text{for } i \geq 2 . \end{aligned}$$

Thus, each Fibonacci number is the sum of the two previous ones, yielding the sequence
0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, ...

Reference: http://www.euroinformatica.ro/documentation/programming/!!!Algorithms_CORMEN!!!/DDU0020.html

THANK YOU