Data Structures

Sec 2 : DS?

By: Eng.Rahma Osama Eng.Sandra Sameh

DEFINITION

- Data structure is the **structural representation** of **logical** relationship between data elements.
- This means that a **data structure** organizes data items based on the relationship between the data elements.
- The study of data structure helps you to understand how data is organized and how data flow is managed **to increase efficiency of any process or program.**

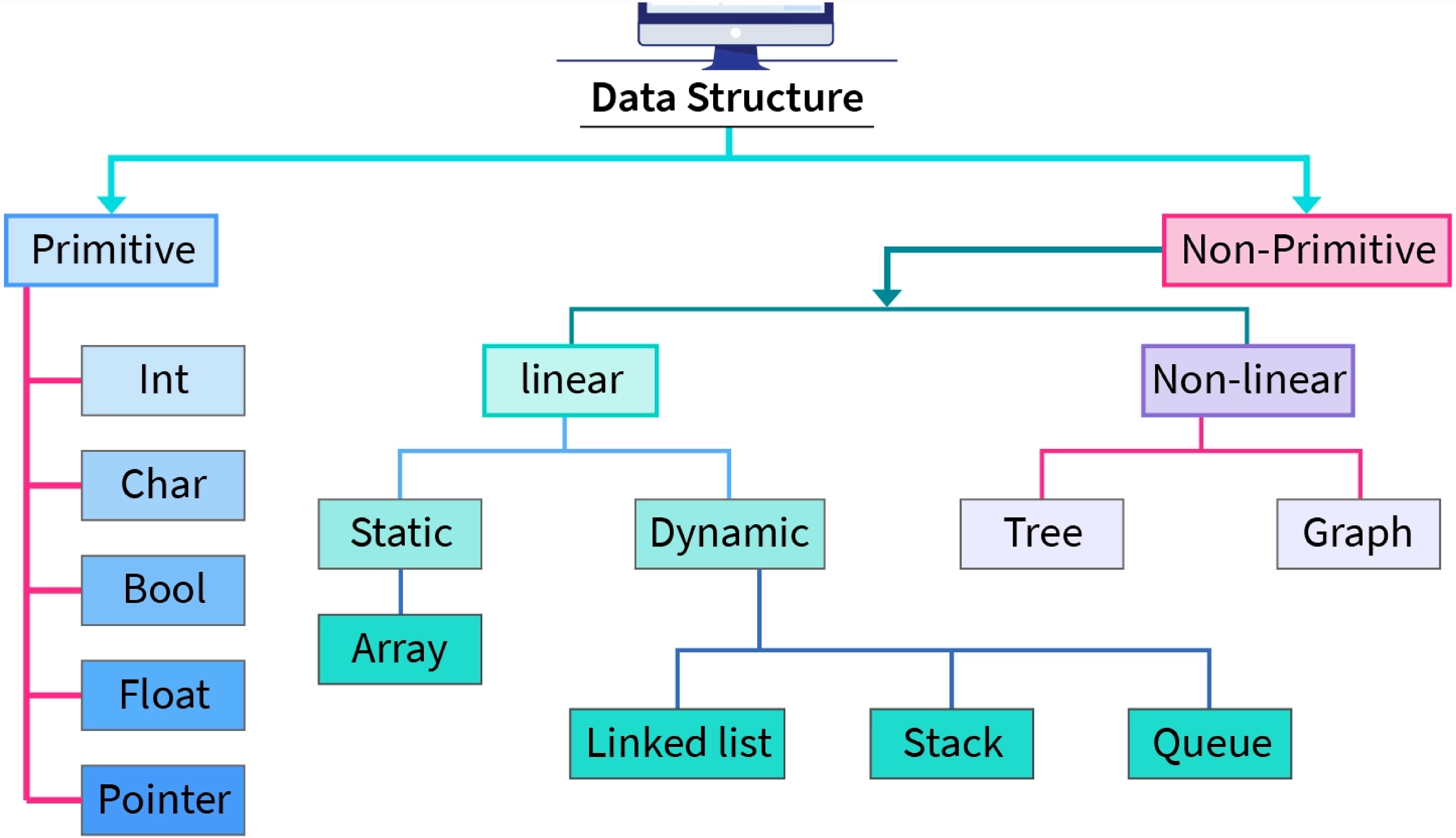
DSA (Data Structures and algorithms)

- **Data Structures and Algorithms (DSA)** is a fundamental part of Computer Science that teaches you how to **think** and **solve** complex problems systematically.
- Today, DSA is a key part of Computer Science education and professional programming, helping us to create **faster** and **more powerful** software.

Program = algorithm + Data structure.

- **Data Structures** is about how data can be **stored** in different structures.
- **Algorithms** is about how to **solve** different problems, often by searching through and manipulating data structures.
- **Theory about Data Structures and Algorithms (DSA) helps us to use large amounts of data to solve problems efficiently.**

CLASSIFICATION OF DATA STRUCTURES



Primitive Data Structures

- **Primitive data** structures consist of the **numbers** and the **characters** which are built in programs.
- These can be manipulated or operated directly by the **machine level instructions.**

such as integer, real, character, and Boolean come under primitive data structures. These data types are .also known as **simple/basic** data types because they consist of characters that cannot be divided

Non-primitive Data Structures

- **Non-primitive data structures** are those that are **derived from primitive** data structures.
- These data structures **cannot** be operated or manipulated **directly** by the machine level instructions.

They focus on formation of a set of data elements that is either homogeneous (same data type) or heterogeneous (different data type). These are further divided into **linear** and **non-linear** data

Linear VS Non-Linear Data Structures

- A data structure that maintains a **linear relationship** among its elements is called a **linear data structure**. Here, the data is arranged in a **linear fashion**. But in the memory, the arrangement may not be sequential.

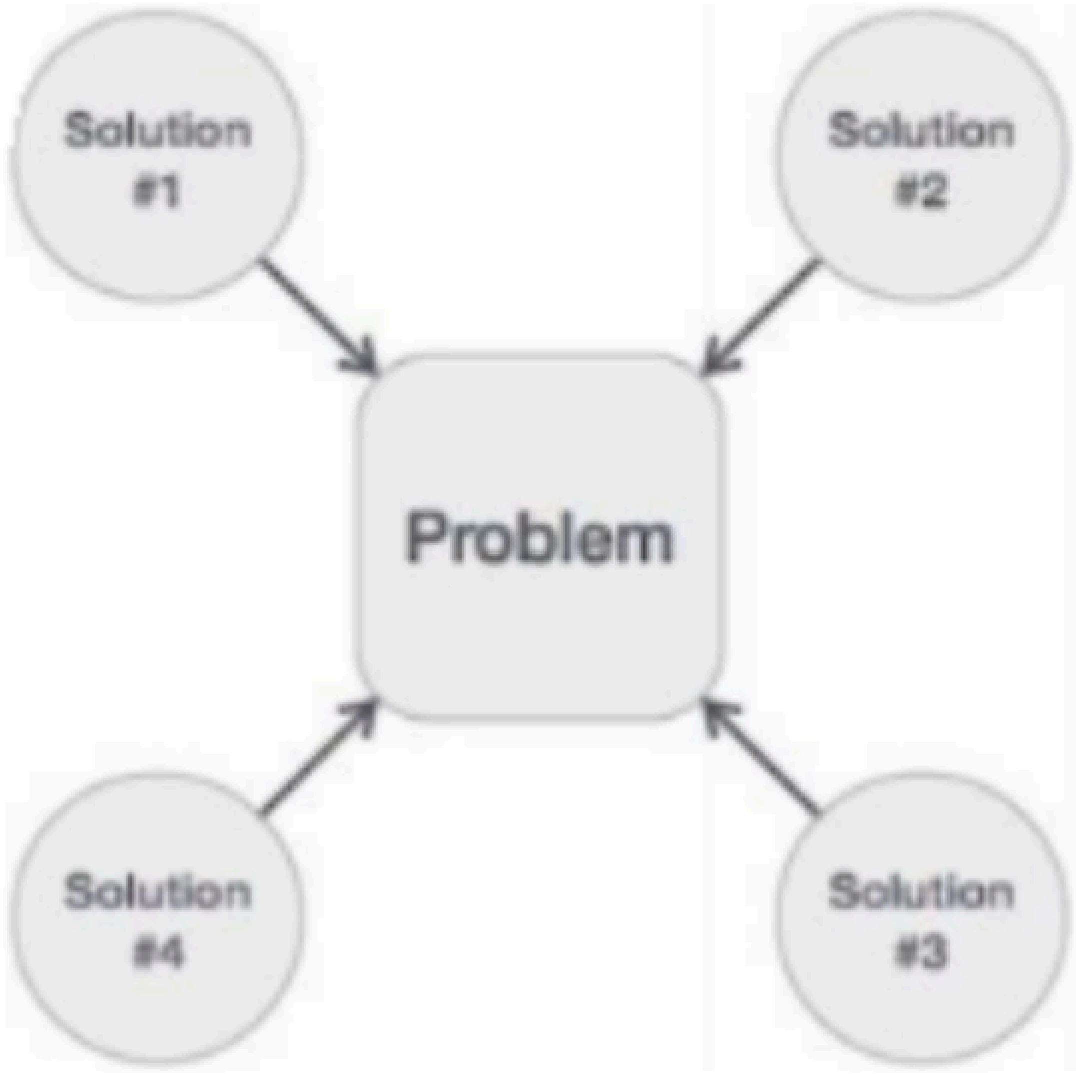
- **Non-linear data structure** is a kind of data structure in which data elements are not arranged in a sequential order. There is a **hierarchical** relationship between individual data items. Here, the insertion and deletion of data is not possible in a linear fashion.

Static VS Dynamic Data Structures

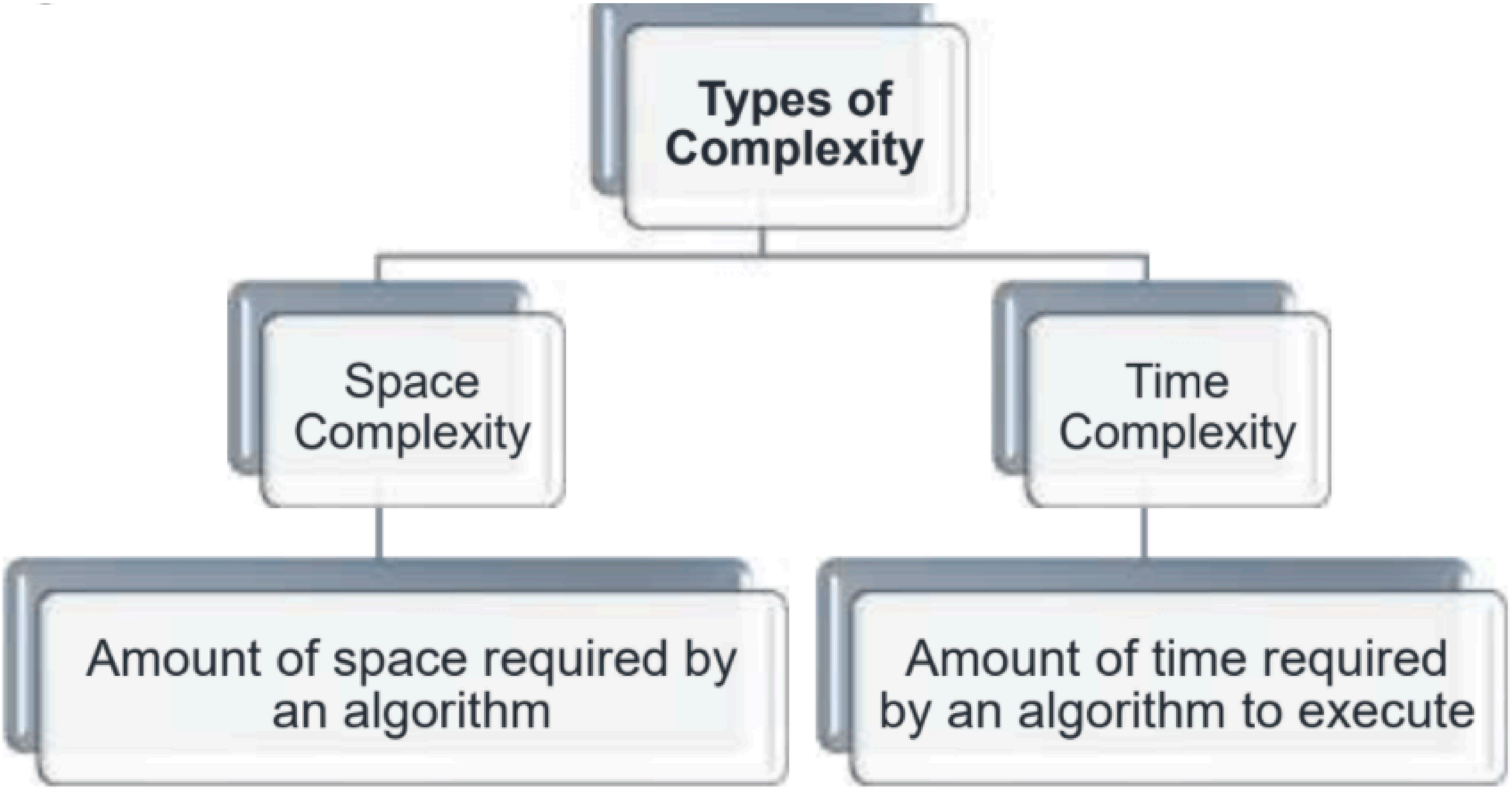
- In **Static data structure** the size of the structure is **fixed**. The content of the data structure can be modified but **without changing** the **memory space** allocated to it.
- Example of Static Data Structures: **Array.**

- In **Dynamic data structure** the size of the structure in **not fixed** and **can be modified** during the operations performed on it.
- Dynamic data structures are designed to facilitate change of data structures in the run time.
- Example of Dynamic Data Structures: **Linked List**.

We design an algorithm to get a solution of a given problem. A problem can be solved in more than one ways.



Hence, many solution algorithms can be derived for a given problem. The next step is to analyze those proposed solution algorithms and implement the best suitable solution.


Types of Complexity
Space Complexity
Time Complexity
Amount of space required by an algorithm
Amount of time required by an algorithm to execute

Time Complexity

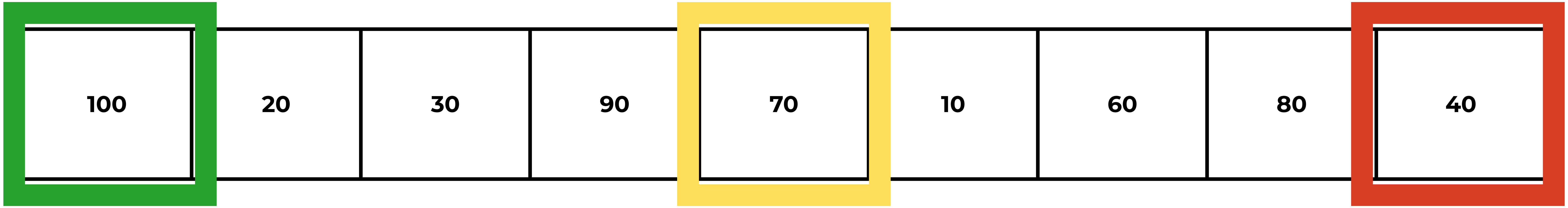
- When considering the **runtime** for different algorithms, we **will not look at the actual tim**e an implemented algorithm uses to run, and here is **why?.**

If we implement an algorithm in a programming language, and run that program, the actual time it will use depends on **many factors**:

  - the programming language used to implement the algorithm
  - how the programmer writes the program for the algorithm
  - the compiler or interpreter used so that the implemented algorithm can run
  - the hardware on the computer the algorithm is running on
  - the operating system and other tasks going on on the computer
  - the amount of data the algorithm is working on

- To **evaluate** and compare different algorithms, instead of looking at the actual runtime for an algorithm, it makes more sense to **use something called time complexity.**
- Time complexity is **more abstract** than actual runtime, and does not consider factors such as programming language or hardware.
- Time complexity is **the number of operations needed to run an algorithm on large amounts of data.** And the number of operations can be considered as time because the computer uses some time for each operation.

Example search problem



best case — Lower bound

average case — Tight bound

worst case — Upper bound

best case	Omega Notation, Ω
average case	theta notation, Θ
worst case	Big O notation, O

Example on time complexity

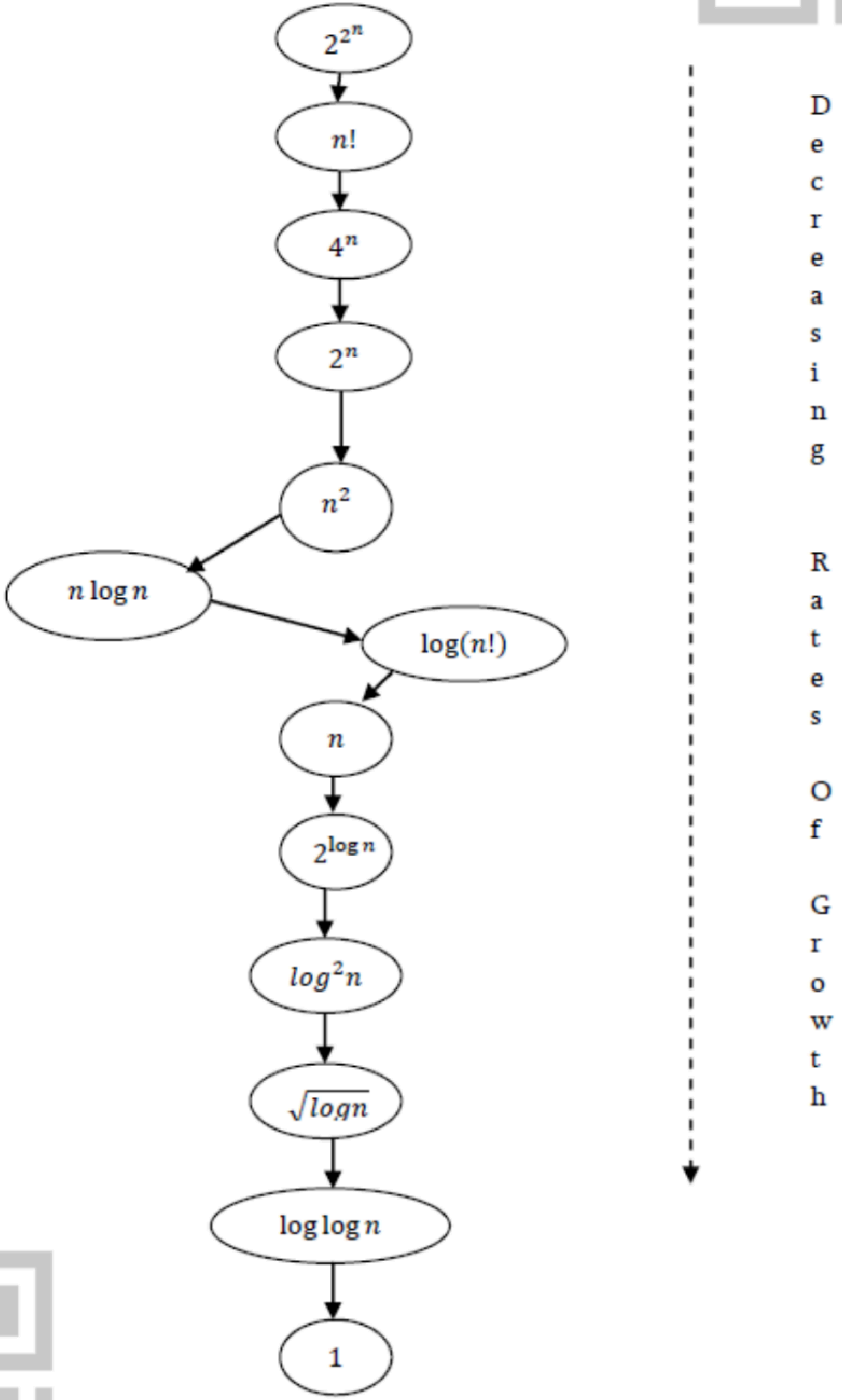
```
int total = 0;
for (int i = 1; i <= n; i++) {
    total += i;
}
```

O(n)

```
int sum = n * (n + 1) / 2;
```

O(1)

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


2^{2^n}
$n!$
4^n
2^n
n^2
$n \log n$
$\log(n!)$
n
$2^{\log n}$
$log^2 n$
$\sqrt{log n}$
$\log \log n$
1
Decreasing Rates Of Growth

Some General Rules
To Determine Time Complexity

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
}
```

Any Arithmetic Operation is constant time (+,-,*,/,++,--,+=,-=,==,...)

Total Time = c * n

O(n)

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 * n * n

$O(n^2)$

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;
}
```

when increment or decrement by (*,/) time is log n

Total time= $\log_2 n$

O(log n)

THANK YOU!