CMC UNIVERSITY

# CHAPTER 1. INTRODUCTION TO DATA STRUCTURES AND ALGORITHMS

Phạm Thị Anh Lê

CMC UNIVERSITY

# References


1. Kent D. Lee & Steve Hubbard, Data Structures and Algorithms with Python, Springer International Publishing, 2015

2. Dr. Basant Agarwal, Hands-On Data Structures and Algorithms with Python, 3rd Edition, Copyright © 2022 Packt Publishing

3. Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein, Introduction to Algorithms, 3rd edition, MIT Press 2009.

4. Benjamin Baka, Python Data Structures and Algorithms, Copyright © 2017 Packt Publishing

5. https://www.programiz.com/dsa

CMC UNIVERSITY

# Outline

- ✓ Basic concepts of Data structures, Algorithms
- ✓ Abstract Data Types
- ✓ Computational Complexity
- ✓ Recursion

CMC UNIVERSITY

# Basic concepts of Data structures, Algorithms

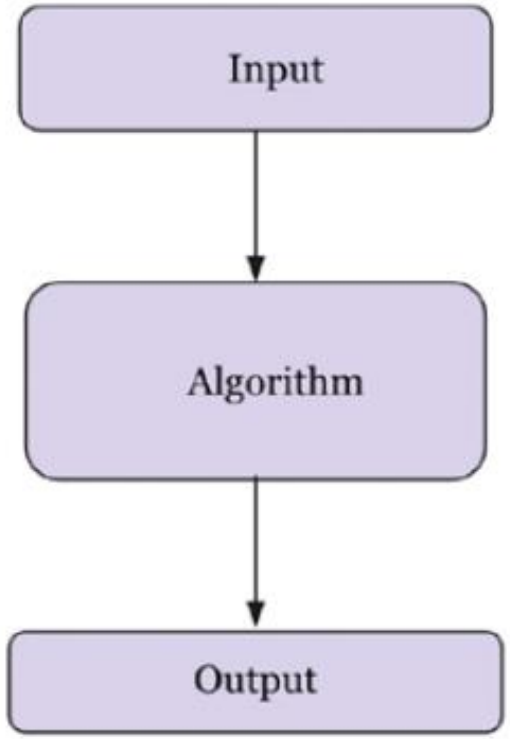
CMC UNIVERSITY

# What is an Algorithm?

In computer programming terms, an algorithm is a set of well-defined instructions to solve a particular problem. It takes a set of input(s) and produces the desired output.

Example: An algorithm to add two numbers:

- Take two number inputs
- Add numbers using the + operator
- Display the result



CMC UNIVERSITY

# What is an Algorithm?

## Qualities of a Good Algorithm

- Input and output should be defined precisely.
- Each step in the algorithm should be clear and unambiguous.
- Algorithms should be most effective among many different ways to solve a problem.
- An algorithm shouldn't include computer code. Instead, the algorithm should be written in such a way that it can be used in different programming languages.

CMC UNIVERSITY

# What is an Algorithm?

## Algorithm Examples

- Algorithm to add two numbers
- Algorithm to find the largest among three numbers
- Algorithm to find all the roots of the quadratic equation
- Algorithm to find the factorial
- Algorithm to check prime number
- Algorithm of Fibonacci series

CMC UNIVERSITY

# Why study algorithms?

Summarized below are some important reasons for studying algorithms:

- Essential for computer science and engineering
- Important in many other domains (such as computational biology, economics, ecology, communications, physics, and so on)
- They play a role in technology innovation
- They improve problem-solving and analytical thinking

There are two aspects that are of prime importance in solving a given problem:

- Firstly, we need an efficient mechanism to store, manage, and retrieve data, which is required to solve a problem (this comes under data structures);
- Secondly, we require an efficient algorithm that is a finite set of instructions to solve that problem.

CMC UNIVERSITY

# Why study algorithms?

Thus, the study of data structures and algorithms is key to solving any problem using computer programs. An efficient algorithm should have the following characteristics:

- It should be as specific as possible
- It should have each instruction properly defined
- There should not be any ambiguous instructions
- All the instructions of the algorithm should be executable in a finite amount of time and in a finite number of steps
- It should have clear input and output to solve the problem
- Each instruction of the algorithm should be integral in solving the given problem

CMC UNIVERSITY

# Why study algorithms?

Consider an example of an algorithm (an analogy) to complete a task in our daily lives; let us take the example of preparing a cup of tea. The algorithm to prepare a cup of tea can include the following steps:

1. Pour water into the pan
2. Put the pan on the stove and light the stove
3. Add crushed ginger to the warming water
4. Add tea leaves to the pan
5. Add milk
6. When it starts boiling, add sugar to it
7. After 2-3 minutes, the tea can be served

CMC UNIVERSITY

# Why study algorithms?

- The above procedure is one of the possible ways to prepare tea. In the same way, the solution to a real-world problem can be converted into an algorithm, which can be developed into computer software using a programming language.
- Since it is possible to have several solutions for a given problem, it should be as efficient as possible when it is to be implemented using software. Given a problem, there may be more than one correct algorithm, defined as the one that produces exactly the desired output for all valid input values.
- The costs of executing different algorithms may be different; it may be measured in terms of the time required to run the algorithm on a computer system and the memory space required for it.

CMC UNIVERSITY

# Data Structures

## What are Data Structures?

- Data structure is a storage that is used to store and organize data. It is a way of arranging data on a computer so that it can be accessed and updated efficiently.
- Depending on your requirement and project, it is important to choose the right data structure for your project. For example, if you want to store data sequentially in the memory, then you can go for the Array data structure.

CMC UNIVERSITY

# Data Structures

## Types of Data Structure

Basically, data structures are divided into two categories:

- Linear data structure
- Non-linear data structure

## Linear data structures

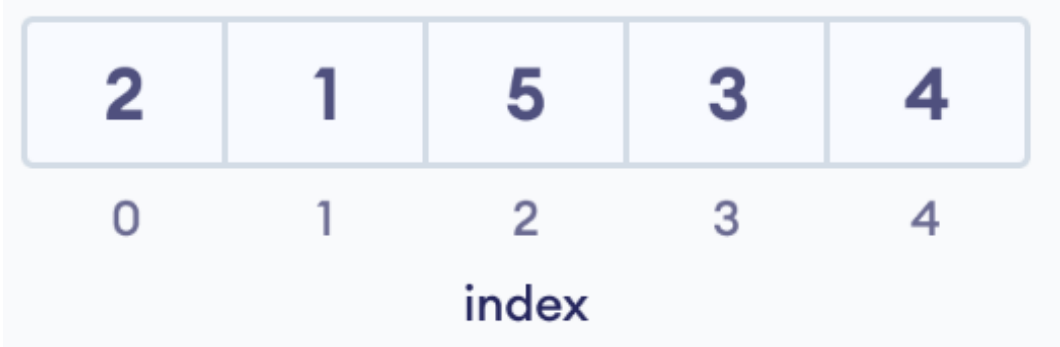
- In linear data structures, the elements are arranged in sequence one after the other. Since elements are arranged in particular order, they are easy to implement.
- However, when the complexity of the program increases, the linear data structures might not be the best choice because of operational complexities.
- Popular linear data structures are:

CMC UNIVERSITY

# Data Structures

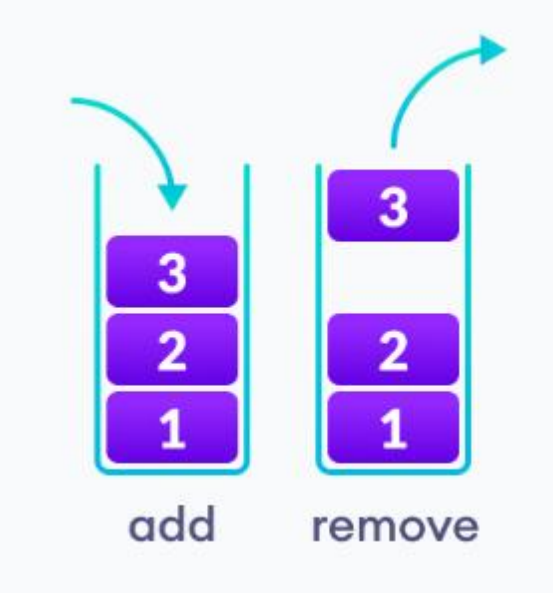
## 1. Array Data Structure

In an array, elements in memory are arranged in continuous memory. All the elements of an array are of the same type. And, the type of elements that can be stored in the form of arrays is determined by the programming language.




## 2. Stack Data Structure

In stack data structure, elements are stored in the LIFO principle. That is, the last element stored in a stack will be removed first. It works just like a pile of plates where the last plate kept on the pile will be removed first.



CMC UNIVERSITY

# Data Structures

## 3. Queue Data Structure

- Unlike stack, the queue data structure works in the FIFO principle where first element stored in the queue will be removed first.
- It works just like a queue of people in the ticket counter where first person on the queue will get the ticket first.


## 4. Linked List Data Structure

- In linked list data structure, data elements are connected through a series of nodes. And, each node contains the data items and address to the next node.

CMC UNIVERSITY

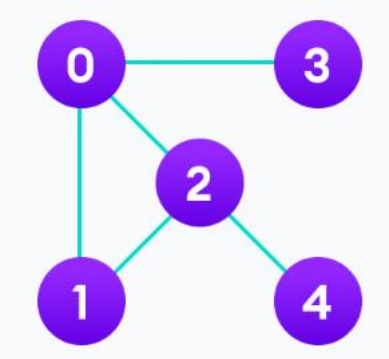
# Data Structures

## Non linear data structures

- Unlike linear data structures, elements in non-linear data structures are not in any sequence. Instead they are arranged in a hierarchical manner where one element will be connected to one or more elements.
- Non-linear data structures are further divided into graph and tree based data structures.

## 1. Graph Data Structure

- In graph data structure, each node is called vertex and each vertex is connected to other vertices through edges.
- **Popular Graph Based Data Structures:**
  - Spanning Tree and Minimum Spanning Tree
  - Strongly Connected Components
  - Adjacency Matrix
  - Adjacency List



CMC UNIVERSITY

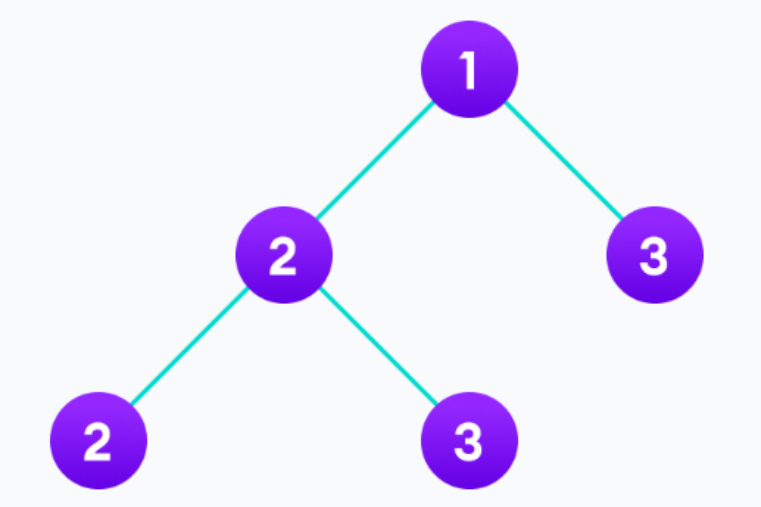
# Data Structures

## 2. Trees Data Structure

- Similar to a graph, a tree is also a collection of vertices and edges. However, in tree data structure, there can only be one edge between two vertices.

## Popular Tree based Data Structure

- Binary Tree
- Binary Search Tree
- AVL Tree
- B-Tree
- B+ Tree
- Red-Black Tree



CMC UNIVERSITY

# Data Structures

| Linear Data Structures | Non Linear Data Structures |
| --- | --- |
| The data items are arranged in sequential order, one after the other. | The data items are arranged in non-sequential order (hierarchical manner). |
| All the items are present on the single layer. | The data items are present at different layers. |
| It can be traversed on a single run. That is, if we start from the first element, we can traverse all the elements sequentially in a single pass. | It requires multiple runs. That is, if we start from the first element it might not be possible to traverse all the elements in a single pass. |

CMC UNIVERSITY

# Data Structures

| Linear Data Structures | Non Linear Data Structures |
|---|---|
| The memory utilization is not efficient. | Different structures utilize memory in different efficient ways depending on the need. |
| The time complexity increase with the data size. | Time complexity remains the same. |
| Example: Arrays, Stack, Queue | Example: Tree, Graph, Map |

**Why Data Structure?**

Knowledge about data structures help you understand the working of each data structure. And, based on that you can select the right data structures for your project.

This helps you write memory and time efficient code.

CMC UNIVERSITY

# Abstract Data Types

CMC UNIVERSITY

# Abstract Data Types

- The Data Type is basically a type of data that can be used in different computer program. It signifies the type like integer, float etc, the space like integer will take 4-bytes, character will take 1-byte of space etc
- The Abstract Data Type (ADT) is special kind of datatype, whose behavior is defined by a set of values and set of operations. The keyword “Abstract” is used as we can use these datatypes, we can perform different operations. But how those operations are working that is totally hidden from the user. The ADT is made of with primitive datatypes, but operation logics are hidden.
- For examples of ADT: Stack, Queue, List, …

CMC UNIVERSITY

# Abstract Data Types

***Some operations of Stack:***

- isFull(), This is used to check whether stack is full or not
- isEmpry(), This is used to check whether stack is empty or not
- push(x), This is used to push x into the stack
- pop(), This is used to delete one element from top of the stack
- peek(), This is used to get the top most element of the stack
- size(), this function is used to get number of elements present into the stack

CMC UNIVERSITY

# Abstract Data Types

Some operations of Queue −

- isFull(), This is used to check whether queue is full or not
- isEmpry(), This is used to check whether queue is empty or not
- insert(x), This is used to add x into the queue at the rear end
- delete(), This is used to delete one element from the front end of the queue
- size(), this function is used to get number of elements present into the queue

CMC UNIVERSITY

# Abstract Data Types

Some operations of List

- size(), this function is used to get number of elements present into the list
- insert(x), this function is used to insert one element into the list
- remove(x), this function is used to remove given element from the list
- get(i), this function is used to get element at position i
- replace(x, y), this function is used to replace x with y value

CMC UNIVERSITY

# Computational Complexity

CMC UNIVERSITY

# Performance analysis of an algorithm

There are primarily two things that one should keep in mind while designing an efficient algorithm:

1. The algorithm should be correct and should produce the results as expected for all valid input values

2. The algorithm should be optimal in the sense that it should be executed on the computer within the desired time limit, in line with an optimal memory space requirement

Performance analysis of the algorithm is very important for deciding the best solution for a given problem. If the performance of an algorithm is within the desired time and space requirements, it is optimal.

One of the most popular methods of estimating the performance of an algorithm is through analyzing its complexity. Analysis of the algorithm helps us to determine which one is most efficient in terms of the time and space consumed.

# Performance analysis of an algorithm

The performance of an algorithm is generally measured by the size of its input data, *n*, and the time and the memory space used by the algorithm.

**Time complexity**

The time complexity of the algorithm is the amount of time that an algorithm will take to execute on a computer system to produce the output.

- The running time required by an algorithm depends on the input size; as the input size, n, increases, the runtime also increases. Input size is measured as the number of items in the input

Example: A sorting algorithm will have an increased runtime to sort a list of input size 5,000 than that of a list of input size 50.

- The time required is measured by the key operations to be performed by the algorithm (such as comparison operations), where key operations are instructions that take a significant amount of time during execution.

Example: the key operation for a sorting algorithm is a comparison operation that will take up most of the runtime, compared to assignment or any other operation

# Performance analysis of an algorithm

The performance of an algorithm is generally measured by the size of its input data, *n*, and the time and the memory space used by the algorithm.

**Time complexity**

- The running time required by an algorithm depends on the input size; as the input size, n, increases, the runtime also increases. Input size is measured as the number of items in the input

Example: A sorting algorithm will have an increased runtime to sort a list of input size 5,000 than that of a list of input size 50.

- The time required is measured by the key operations to be performed by the algorithm (such as comparison operations), where key operations are instructions that take a significant amount of time during execution.

Example: the key operation for a sorting algorithm is a comparison operation that will take up most of the runtime, compared to assignment or any other operation

- The space requirement of an algorithm is measured by the memory needed to store the variables, constants, and instructions during the execution of the program.

CMC UNIVERSITY

# Performance analysis of an algorithm

Ideally, these key operations should not depend upon the hardware, the operating system, or the programming language being used to implement the algorithm.

A constant amount of time is required to execute each line of code; however, each line may take a different amount of time to execute.

***Example 1***:

| Code | Time required (Cost) |
|---|---|
| `if n==0 \|\| n == 3 #constant time`<br>`  print("data")`<br>`else:`<br>`  for i in range( #loop run for n times`<br>`  print("structure")` | c1<br>c2<br>c3<br>c4<br>c5 |

CMC UNIVERSITY

# Performance analysis of an algorithm

The running time of the algorithm is the sum of time required by all the statements.

For the above code, assume statement 1 takes $c_1$ amount of time, statement 2 takes $c_2$ amount of time, and so on. So, if the ith statement takes a constant amount of time $c_i$ and if the ith statement is executed $n$ times, then it will take $c_i n$ time.

The total running time $T(n)$ of the algorithm for a given value of $n$ (assuming the value of $n$ is not zero or three) will be as follows.

$$T(n) = c_1 + c_3 + c_4 \times n + c_5 \times n$$

If the value of n is equal to zero or three, then the time required by the algorithm will be as follows $T(n) = c_1 + c_2$

CMC UNIVERSITY

# Performance analysis of an algorithm

Therefore, the running time required for an algorithm also depends upon what input is given in addition to the size of the input given.

For the given example:

- The best case will be when the input is either zero or three, and in that case, the running time of the algorithm will be constant.
- In the worst case, the value of n is not equal to zero or three, then, the running time of the algorithm can be represented as $a \times n + b$. Here, the values of $a$ and $b$ are constants that depend on the statement costs, and the constant times are not considered in the final time complexity. In the worst case, the runtime required by the algorithm is a linear function of $n$.

CMC UNIVERSITY

# Performance analysis of an algorithm

## *Example 2*: linear search

```
def linear_search(input_list, element):
    for index, value in enumerate(input_list):
        if value == element:
            return index

    return -1

input_list = [3, 4, 1, 6, 14]
element = 4
print("Index position for the element x is:", linear_search(input_
list,element))
```

The output in this instance will be as follows:

```
Index position for the element x is: 1
```

CMC UNIVERSITY

# Performance analysis of an algorithm

- The worst-case running time of the algorithm is the upper-bound complexity; it is the maximum runtime required for an algorithm to execute for any given input.

Example: in the linear search problem, the worst case occurs when the element to be searched is found in the last comparison or not found in the list. In this case, the running time required will linearly depend upon the length of the list.

- The average-case running time is the average running time required for an algorithm to execute. In this analysis, we compute the average over the running time for all possible input values.

Example: in the linear search, the number of comparisons at all positions would be *1* if the element to be searched was found at the 0th index; and similarly, the number of comparisons would be *2*, *3*, and so forth, up to $n$, respectively, for elements found at the *1, 2, 3, ... (n-1)* index positions.

Thus, the average-case running time will be: $T(n) = \frac{1+2+\cdots+n}{n} = \frac{n(n+1)}{2n}$

CMC UNIVERSITY

# Performance analysis of an algorithm

- The best-case running time is the minimum time needed for an algorithm to run; it is the lower bound on the running time required for an algorithm;

Example: in the linear search problem, the search element will be found in the first comparison

**Space complexity**

The space complexity of the algorithm estimates the memory requirement to execute it on a computer to produce the output as a function of input data.

While executing the algorithm on the computer system, storage of the input is required, along with intermediate and temporary data in data structures, which are stored in the memory of the computer.

CMC UNIVERSITY

# Performance analysis of an algorithm

## *Example 3*

```
def squares(n):
    square_numbers = []
    for number in n:
        square_numbers.append(number * number)
    return square_numbers

nums = [2, 3, 5, 8 ]
print(squares(nums))
```

The output of the code is:

```
[4, 9, 25, 64]
```

In this code, the algorithm will require allocating memory for the number of items in the input list.
The number of elements in the input is *n*, then the space requirement increases with the input size, therefore, the space complexity of the algorithm becomes *O(n)*
Suppose there are two search algorithms, one has O(n) and another algorithm has O(nlogn) space complexity.
The first algorithm is the better algorithm as compared to the second with respect to the space requirements

CMC UNIVERSITY

# Asymptotic notation

- To analyze the time complexity of an algorithm, the rate of growth (order of growth) is very important when the input size is large.
- When the input size becomes large, we only consider the higher-order terms and ignore the insignificant terms.
- In asymptotic analysis, we analyze the efficiency of algorithms for large input sizes considering the higher order of growth and ignoring the multiplicative constants and lower-order terms

CMC UNIVERSITY

# Asymptotic notation

The following asymptotic notations are commonly used to calculate the running time complexity of an algorithm:

- θ: denotes the worst-case running time complexity with a tight bound.
- O: denotes the worst-case running time complexity with an upper bound, which ensures that the function never grows faster than the upper bound.
- Ω: denotes the lower bound of an algorithm's running time. It measures the best amount of time to execute the algorithm

CMC UNIVERSITY

# Asymptotic notation

## Theta notation

- The following function characterizes the worst-case running time for the example 1: $T(n) = c_1 + c_3 + c_4 \times n + c_5 \times n$

Here, for a large input size, the worst-case running time will be $\theta(n)$, i.e. the time complexity of an algorithm whose growth order (referred to as order) is $n$

- We usually consider one algorithm to be more efficient than another if its worst-case running time has a lower order of growth

$\theta$ denotes the worst-case running time for an algorithm with a tight bound. For a given function $F(n)$, the asymptotic worst-case running time complexity can be defined as follows:

$$T(n) = \theta(F(n))$$

iff there exists constants $n_0$, $c_1$, and $c_2$ such that:

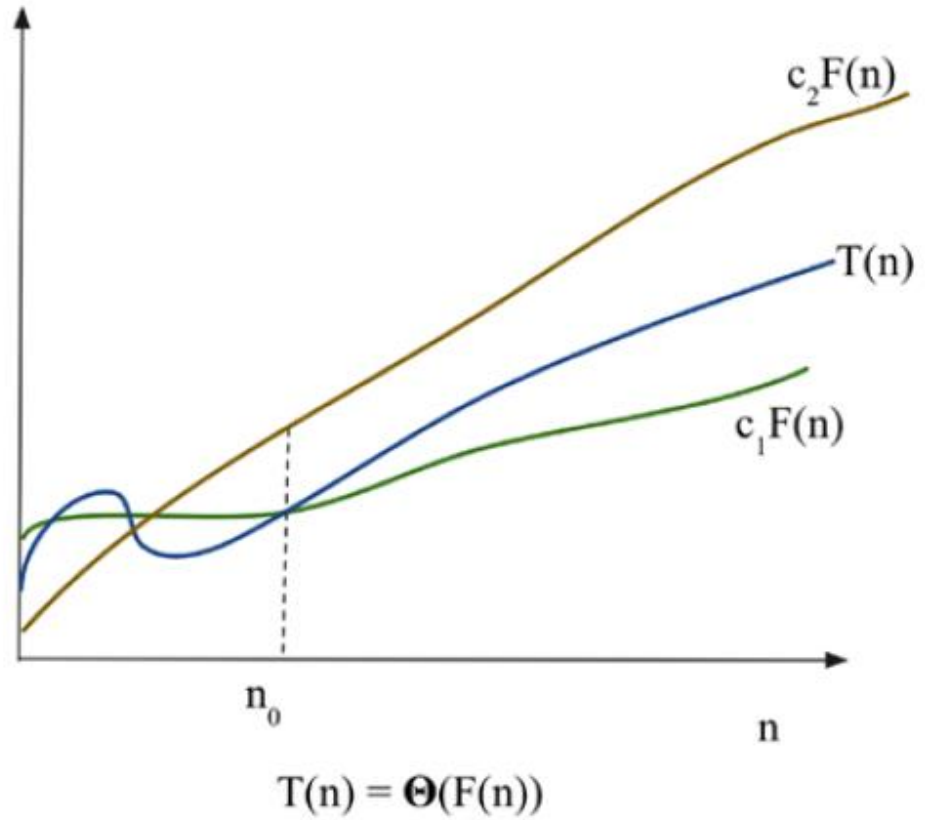
$$0 \leq c_1(F(n)) \leq T(n) \leq c_2(F(n)) \text{ for all } n \geq n_0$$

CMC UNIVERSITY

# Asymptotic notation

- The function $T(n)$ belongs to a set of functions $\Theta(F(n))$ if there exists positive constants $c_1$ and $c_2$ such that the value of $T(n)$ always lies in between $c_1F(n)$ and $c_2F(n)$ for all large values of $n$. If this condition is true, then we say $F(n)$ is asymptotically tight bound for $T(n)$

The figure shows that the value of $T(n)$ always lies in between $c_1F(n)$ and $c_2F(n)$ for values of $n$ greater than $n_0$



# Asymptotic notation

***Example 4:***

$$f(n) = n^2 + n$$

In order to determine the time complexity with the Θ notation definition, we have to first identify the constants $c_1$ , $c_2$, $n_0$ such that

$$0 \le c_1 n^2 \le n^2 + n \le c_2 n^2 \quad \text{for all } n \ge n_0$$

Dividing by $n^2$ will produce

$$0 \le c_1 \le 1 + 1/n \le c_2 \quad \text{for all } n \ge n_0$$

By choosing $c_1 = 1$, $c_2 = 2$, $n_0 = 1$, the following condition can satisfy the definition of theta notation

$$0 \le n^2 \le n^2 + n \le 2n^2 \quad \text{for all } n \ge 1$$

That gives: $f(n) = \Theta(g(n))$, means $f(n) = \Theta(n^2)$

# Asymptotic notation

## Big O notation

We have seen that the $\theta$ is asymptotically bound from the upper and lower sides of the function whereas the Big $O$ notation characterizes the worst-case running time complexity, which is only the asymptotic upper bound of the function. Big $O$ notation is defined as follows.

Given a function $F(n)$, the $T(n)$ is a Big $O$ of function $F(n)$, and we define this as follows:

$$T(n) = O(F(n))$$

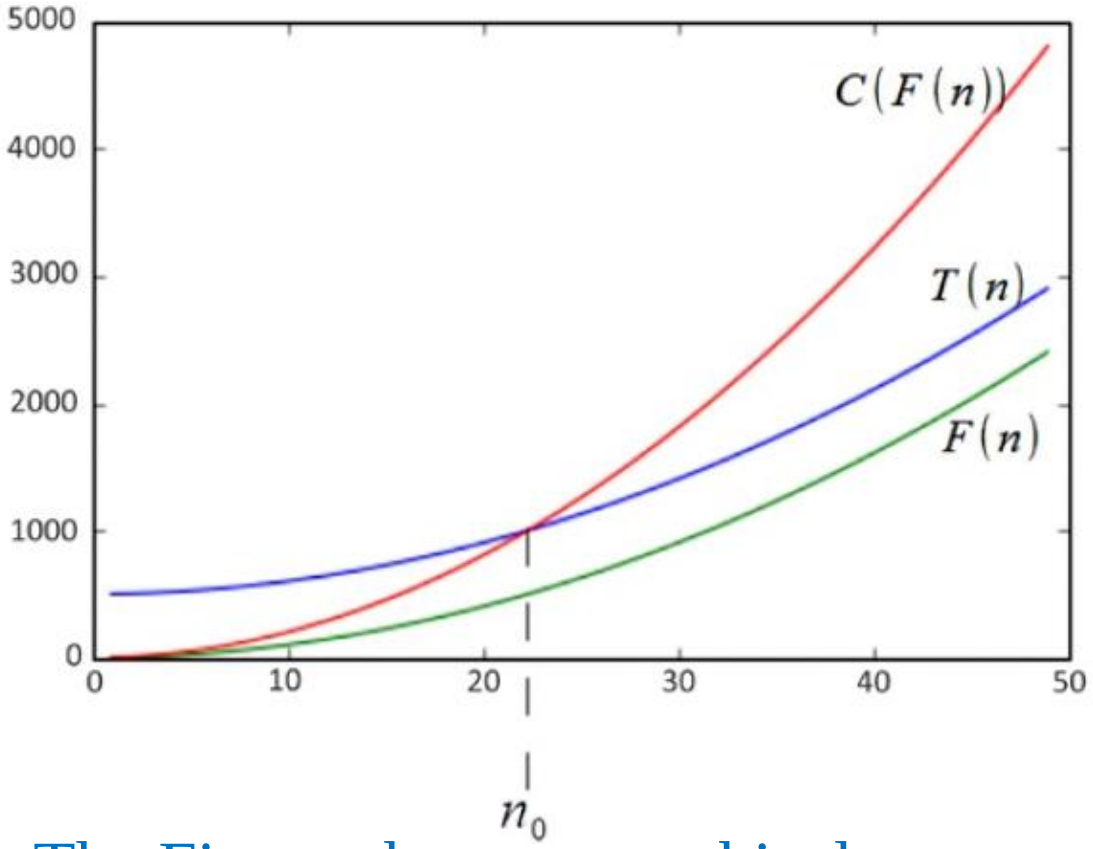
iff there exists constants $n_0$ and $c$ such that:

$$T(n) \leq c(F(n)) \text{ for all } n \geq n_0$$

In Big O notation, a constant multiple of $F(n)$ is an asymptotic upper bound on $T(n)$, and the positive constants $n_0$ and $c$ should be in such a way that all values of $n$ greater than $n_0$ always lie on or below function $c.F(n)$.

CMC UNIVERSITY

# Asymptotic notation




The Figure shows a graphical representation of function $T(n)$ with a varying value of $n$. We can see that $T(n) = n^2 + 500 = O(n^2)$, with $c = 2$ and $n_0$ being approximately 23

In $O$ notation, $O(F(n))$ is really a set of functions that includes all functions with the same or smaller rates of growth than $F(n)$. For example, $O(n^2)$ also includes $O(n)$, $O(\log n)$,...

However, Big $O$ notation should characterize a function as closely as possible, for example, $F(n) = 2n^3 + 2n^2 + 5$ is $O(n^4)$, however, it is more accurate that $F(n)$ is $O(n^3)$.

CMC UNIVERSITY

# Asymptotic notation

| Time Complexity | Name |
|---|---|
| O(1) | Constant |
| O(logn) | Logarithmic |
| O(n) | Linear |
| O(nlogn) | Linear-logarithmic |
| $O(n^2)$ | Quadratic |
| $O(n^3)$ | Cubic |
| $O(2^n)$ | Exponential |

| $Log_2n$ | n | $nlog_2n$ | $n^2$ | $n^3$ | $2^n$ |
|---|---|---|---|---|---|
| 0 | 1 | 0 | 1 | 1 | 2 |
| 1 | 2 | 2 | 4 | 8 | 4 |
| 2 | 4 | 8 | 16 | 64 | 16 |
| 3 | 8 | 24 | 64 | 512 | 256 |
| 4 | 16 | 64 | 256 | 4096 | 65536 |
| 5 | 32 | 160 | 1024 | 32768 | 2 147 483 648 |

Functions $2^n$, $n!$, $n^n$ are called exponential functions. An algorithm whose execution time is exponential is very slow. Functions $n^3$, $n^2$, $nlog_2n$, $n$, $log_2n$ are called polynomial functions. Algorithms with polynomial-level execution times are generally acceptable.

CMC UNIVERSITY

# Asymptotic notation

## Omega notation

- Omega notation Ω describes an asymptotic lower bound on algorithms.
- Omega notation computes the best-case runtime complexity of the algorithm.
- The Ω notation is a set of functions in such a way that there are positive constants $n_0$ and $c$ such that for all values of $n$ greater than $n_0$, $T(n)$ always lies on or above a function to $c.F(n)$.

$$T(n) = \Omega\,(F(n))$$

Iff constants $n_0$ and c are present, then:

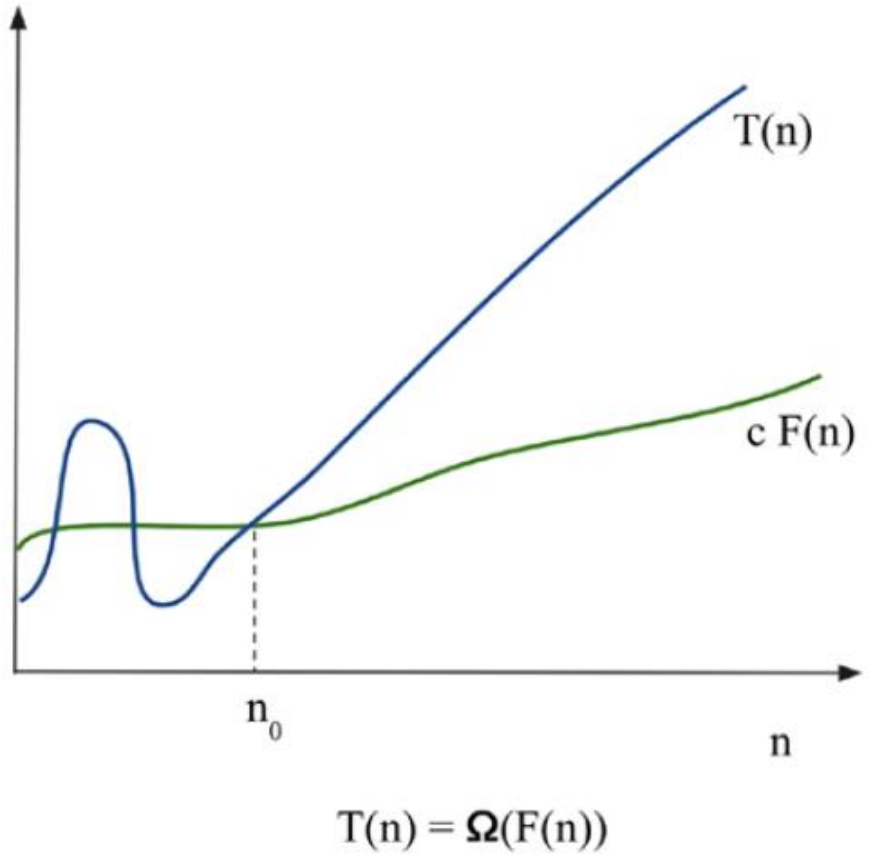
$$0 \leq c(F(n)) \leq T(n) \quad \text{for all} \quad n \geq n_0$$

CMC UNIVERSITY

# Asymptotic notation

If the running time of an algorithm is $\Omega(F(n))$, it means that the running time of the algorithm is at least a constant multiplier of $F(n)$ for sufficiently large values of input size ($n$).

The $\Omega$ notation gives a lower bound on the best-case running time complexity of a given algorithm. It means that the running time for a given algorithm will be at least $F(n)$ without depending upon the input.



# Asymptotic notation

***Example 5***: Find $F(n)$ for the function $T(n) = 2n^2 + 3$ such that $T(n) = \Omega(F(n))$

***Solution***: Using the $\Omega$ notation, the condition for the lower bound is:

$$c.F(n) \leq T(n)$$

This condition holds true for all values of n greater than 0, and $c = 1$

$$0 \leq cn^2 \leq 2n^2 + 3, \text{ for all } n \geq 0$$

$$2n^2 + 3 = \Omega(n^2)$$

$$F(n) = n^2$$

CMC UNIVERSITY

# Computing the time complexity of an algorithm

- To analyze an algorithm with respect to the best-, worst-, and average-case runtime of the algorithm, it is not always possible to compute these for every given function or algorithm. However, it is always important to know the upper-bound worst-case runtime complexity of an algorithm in practical situations; therefore, we focus on computing the upper-bound Big O notation to compute the worst-case runtime complexity of an algorithm.
- Determining the time complexity of any algorithm can lead to complex problems. However, in practice, for some algorithms, it can be analyzed by some simple rules.

# Computing the time complexity of an algorithm

## Sum rule

- Assuming $T_1(n)$ and $T_2(n)$ are the execution time of two program segments $P_1$, $P_2$, where $T_1(n) = O(f(n))$, $T_2(n) = O(g(n))$, then the execution time of $P_1$ and $P_2$ successively will be:

$$T_1(n) + T_2(n) = O(\max(f(n), g(n)))$$

***Example 6***: In a program with 3 execution steps where the execution time of each step is $O(n^2)$, $O(n^3)$ and $O(n\log_2 n)$ respectively, the execution time of the first 2 steps is $O(\max(n^2, n^3)) = O(n^3)$. The program execution time will be $O(\max(n^3, n\log_2 n)) = O(n^3)$.

Another application of this rule: if $g(n) \leq f(n)$, $\forall n \geq n_0$ then $O(f(n) + g(n))$ is also $O(f(n))$. Example, $O(n^4 + n^2) = O(n^4)$, $O(n + \log_2 n) = O(n)$.

# Computing the time complexity of an algorithm

## Product rule

- Assuming $T_1(n)$ and $T_2(n)$ are the execution time of two program segments $P_1$, $P_2$, where $T_1(n) = O(f(n))$, $T_2(n) = O(g(n))$, then the execution time of the nested segments $P_1$ and $P_2$ will be:

$$T_1(n)\, T_2(n) = O(f(n)g(n))$$

***Example 7***: for i in range(n): x = x+1

The assignment statement has an execution time of c (constant), so it evaluates to O(1).

The loop statement has an execution time of O(n.1) = O(n).

The statement for i in range(n)

for j in range(n): x = x+1 is evaluated as $O(n.n) = O(n^2)$

Another application of this rule: O(cf(n)) = O(f(n)). Example, $O(n^2/2) = O(n^2)$

# Computing the time complexity of an algorithm

***Example 8***: Find the worst-case runtime complexity of the following Python snippet:

```python
# loop will run n times
for i in range(n):
    print("data")  #constant time
```

$T(n) = cn = O(n)$

```python
for i in range(n):
    for j in range(n):  # This loop will also run for n times
        print("run")
```

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

```python
for i in range(n):
    for j in range(n):
        print("run fun")
        break
```

$T(n) = O(n)$

```python
def fun(n):
    for i in range(n):
        print("data")  #constant time
 #outer loop execute for n times
    for i in range(n):
        for j in range(n):  #inner loop execute n times
            print("run fun")  #constant time
```

$T(n) = c_1n + c_2n^2 = O(n^2)$

# Computing the time complexity of an algorithm

***Example 8***: Find the worst-case runtime complexity of the following Python snippet:

```
if  n == 0:      #constant time
    print("data")
else:
    for i in range(n):     #loop run for n times
      print("structure")
```

$T(n) = c_1 + c_2 n = O(n)$

```
i = 1
j = 0
while i*i < n:
    j = j +1
    i = i+1
    print("data")
```

$T(n) = O(\sqrt{n})$

```
i = 0
for i in range(int(n/2), n):
    j = 1
    while j+n/2 <= n:
    k = 1
          while k < n:
          k *= 2
          print("data")
          j += 1
```

$T(n) = O(n*n*\log_2 n) = O(n^2 \log_2 n)$

CMC UNIVERSITY

# Computing the time complexity of an algorithm

Find the worst-case runtime complexity of the following Python snippet:

```python
import math

def approximate_exp(x, n):
    approx = 0
    for i in range(n):
        approx += (x ** i) / math.factorial(i)
    return approx

x = float(input("Nhập giá trị x: "))
n = int(input("Nhập số lượng đơn vị trong xấp xỉ: "))

result = approximate_exp(x, n)
print("e^{} ≈ {}".format(x, result))
```

```python
import math

def approximate_exp(x, n):
    approx = 0
    for i in range(n):
        term = 1
        for j in range(1, i+1):
            term *= x / j
        approx += term
    return approx

x = float(input("Nhập giá trị x: "))
n = int(input("Nhập số lượng đơn vị trong xấp xỉ: "))

result = approximate_exp(x, n)
print("e^{} ≈ {}".format(x, result))
```

CMC UNIVERSITY

# Recursion

CMC UNIVERSITY

# Recursion

A recursive algorithm calls itself repeatedly in order to solve the problem until a certain condition is fulfilled. Each recursive call itself spins off other recursive calls.

A recursive function can be in an infinite loop; therefore, it is required that each recursive function adheres to certain properties. At the core of a recursive function are two types of cases:

1. Base cases: These tell the recursion when to terminate, meaning the recursion will be stopped once the base condition is met

2. Recursive cases: The function calls itself recursively, and we progress toward achieving the base criteria

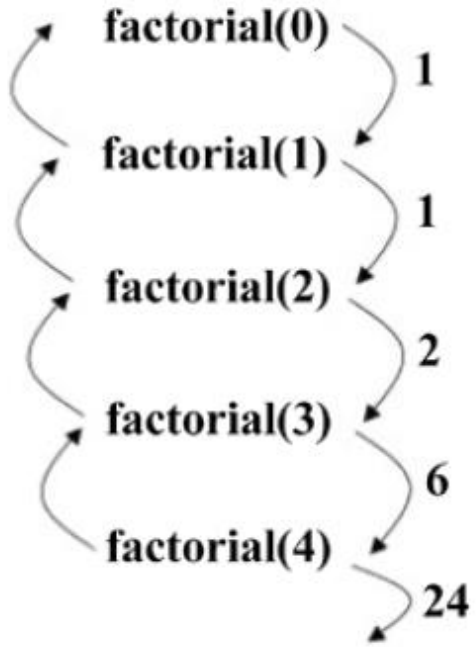
CMC UNIVERSITY

# Recursion

- A simple problem that naturally lends itself to a recursive solution is calculating factorials. The recursive factorial algorithm defines two cases: the base case when n is zero (the terminating condition) and the recursive case when n is greater than zero (the call of the function itself).
- A typical implementation is as follows:

```
def factorial(n):
    # test for a base case
    if n == 0:
        return 1
    else:
    # make a calculation and a recursive call
        return n*factorial(n-1)

print(factorial(4))
```




CMC UNIVERSITY
THANK YOU