# Chapter 1 _ Introduction to Data Structures and Algorithms

# I. Data Structure & ADT(Abstract Data Type)

## 1. Data Structure

### 1) Definition and Purpose

> - **A collection of**
>   **- data** values
>   - **the relationships** among them
>   - **the funcions or operations** that can be applied to the data.
>
> - A data organization, management, and storage format that is usually **chosen for efficient access to data.**
>
> - A specific way of **organizing and storing data** in a computer: for ***efficiency in accessing & modifying*** them
>   Source from Wikipedia

Shortly, data structure can be understood in this way :

**"A structural concept, or a structure itself which is designed to have its own specific way of organizing/storing/manipulating data and operations."**

### 2) Usage

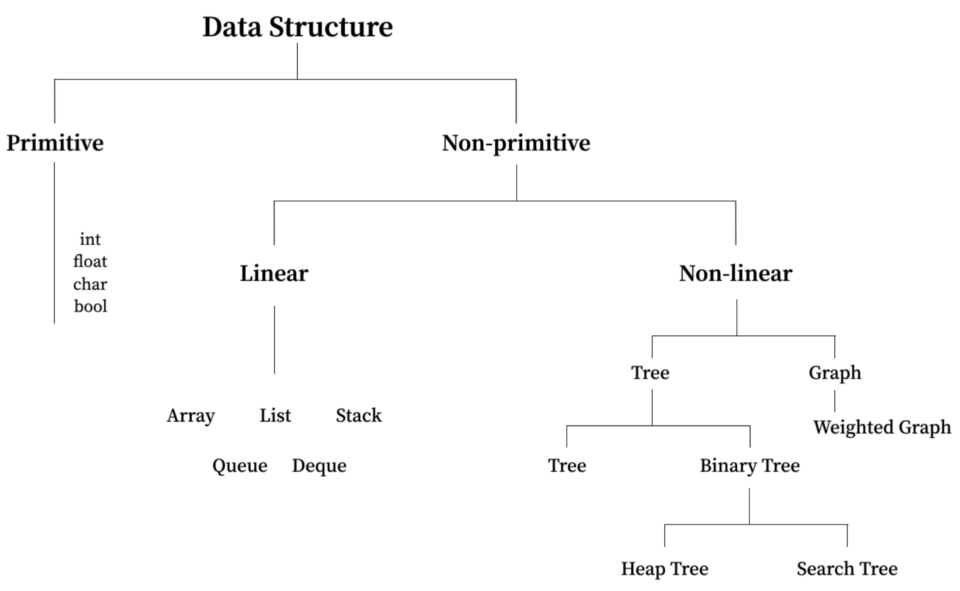
- Serves as conceptual basis for ***ADT(Abstract Data Type)***.
- And data structure ***implements*** physical fom of the data type. - In here, ***implements*** means 'use'.
- Difference in these structures makes each of them specialized for specific tasks and had them pros and cons in its usage circumstances compared to others.
  - RDB : B-Tree
  - Compiler implementations : hash tables
- Therefore,

**"understanding each structures properly and knowing what and when to use is the key to designing *efficient* softwares and algorithms."**

### 3) Classification

- Data structures can be classified by the shape of their data organization.
  - ***Primitive*** data structure
  - Non-primitive data structure
    - ***Linear*** data structure : They can be implemented in following ways - Array structure / Linked structure

- ***Non-linear*** data structure



## 2. ADT(Abstract Data Type)

### "the *logical form* definition of the data type"

### 1) Definition

> - ***A mathematical model*** for *data types*.
> - **A logical description** of,
>   - *Data* - what kind(type) of data the data type or data structure deals with.
>   - Operations - possible operations which can be applied or processed with specified data above.
> - An ADT does **not specify how the data type is implemented.**(hidden from the user)
>   ⇒ **Encapsulation**

- ADT is defined by its behavior - in semantics. not syntactics. And this is one of the key difference between "Data Type" - from the POV of a user, data
- Both ADT and data structure have 'data and its behavior' in common as its definition, data structure also can be described in form of ADT.

### 2) Example

Again,

#### ADT is the specification of a data type within *some language*, independent of an implementation.

For example, let's suppose we want to represent 'the bag' - means literally 'the bag' - in computer world.
To do so, we have to capture **what bag does(operations)**, and **what can be in the bag(data).**
Then if we describe or specify what we've captured from bag in language(semantic), that is ADT of bag.

> The Bag ADT (Source : https://www.cs.miami.edu/home/geoff/Courses/MTH517-00S/Content/ArrayBasedADTs/BagsStacksQueues.html)
>
> - An *bag* object is an unordered collection of zero or more elements of some type. If the all the elements are of the same type, it is a homogeneous bag, otherwise it is a *heterogeneous* bag.
> - Operations include adding an element to the bag, removing an element from the bag, testing if the bag is full or empty.

# 3. Data Type

## Data's Type - Usually constraints for data and its behavior.

## 1) Definition

> **A collection/grouping of data values**, usually ***specified by*** followings:
> - A set of possible values.
> - A set of allowed operations on these values.
> - And / Or representation of these values as machine types.

And there are ***five definitions of a "type"*** **identified by Parnas, Shore & Weiss**.

> **Syntactic**
> A type is **a purely syntactic label** associated with a variable when it is declared.
> Although useful for advanced type systems such as substructural type systems, such definitions provide no intuitive meaning of the types.
>
> **Representation**
> A type is defined in terms of **a composition of more primitive types**-often machine types.
>
> **Representation and behaviour**
> A type is defined as **its representation** and **a set of operators** manipulating these representations.
>
> **Value space**
> A type is **a set of possible values** which a variable can possess.
> Such definitions make it possible to speak about unions or Cartesian products and other mathematically acceptable topics.
> - This point of view defines data type strognly based on mathematics : set theory or number theory
> - For example, suppose that we want to define 'natural number type' in perspective of 'value space' definition.
> - We would **abstractly** define 'natural number type' based on ***Peano Axiom***. And then we would implement that in codes.
>
> **Value space and behaviour**
> A type is a set of values which a variable can possess and a set of functions that one can apply to these values.
>
> (Source : Wikipedia, Original Work "Abstract Types Defined As Classes Of Variables by D. L. Parnas, John E. Shore, David Weiss, March 1976" from https://sigmodrecord.org/publications/sigmodRecord/7603/pdfs/984344.807133.pdf)

- Definition on 'types' which **including behavior** align more closely with **object-oriented models.**
  - Considering OOP paradigm's conceptual basis, this fits well.
    - OOP's conceptual basis - ***objects' status and behavior***
- Implementation of data type specification works as a **constraint**
  - Because they specify the possible values - an expression, a variable or a function call might take.
  - Then why do we specify data type?
    - In cosidering of simplicity, computability, or regularity.
    - Specifying possible input/output and operations helps compiler to manage semantic errors.

            - And thinking of difficulty in managing semantic errors, this can guarantee those three above in certain level.

- Almost all programming languages explicitly include the notion of data type.
    - But, there can be different semantics for similar types. For example,
        - Int
            - Python : ***Arbitrary-precision integer***
                - Python **'arbitrarily'** defines **possible length of integers**. They assign more memory space when it is needed.
                - Java supports this with 'BigInteger'
            - Java : ***Fixed-precision integer***(32-bit integers)
                - Strictly defined length. (-2^31 ~ +2^31-1)

## 2) Classification

- Classification can be vary depends on perspectives.

### Is it built-in? or not?

- **Primitive data types**
    - Pre-defined and ***built-in to a language implementation.***
        - Example: Numeric types in Java
- **User-defined data types**
    - Literally **user-defined** data types.
        - Example: Classes which are defined by user.

### Is it broken into other components? or not?

- **Atomic type**
    - **A single data item** that *can't be broken into component parts*.
        - Example: Integer(Of course, it consists of a sequence of bits, commonly considered atomic.)
- **Composite type / Aggregate type**
    - **A collection of data items** that *can be accessed individually*.
        - Example: array of integer or string.

### Is it original? or derived from other data types?

- **Basic data type / Fundamental data type**
    - Defined ***axiomatically*** from *fundamental notions* or *by enumeration of their elements*.
        - Example: Integer ⇒ Basic type defined in mathematics by axioms.
- **Generated data type / Derived data type**
    - *Specified* and *partly defined* ***in terms of other data types***.
        - Example: Array of integers ⇒ The result of applying an array type generator to the integer type.

## 4. Data Structure v.s Data Type

To precisely compare those two concepts, review on their term has to be preceded.

- Data Structure
    - : A collection of data values, the relationships among them, the operations that can be applied for them. Structure of how they orginized.
- Data Type : A collection of data values. Usually specified by, a) a set of possible values, a set of possible operations.

- Abstract Data Type : A logical form of data type. Specified, but not implemented form of data type.

I was confused by their similarity in definition. Both are defined as 'a collection of data values, and possible operations'.
Even some people on StackOverflow, Quora refers to data structures as the term 'data type' - like somehow they can be used interchangeably.
In my humble opinion, that confusion seems to be derived from two following points
- Formal definition of both terms have 'value(data) and operations' in common.
- And in python and java, data structures are coded using ***class*** - Which is provided way for user to define their own data type(user defined data type) by programming language.
- As stated above, **data structure implements a logical form of data type** - This corresponds to using class to code data structures in programming language.
- Comparing the code implementation of data structure and data type, people would be able to see the similar looking(implemented form of code, both have data & behavior in its def) between them.
Thus, the term 'data structure' and 'data type' can be confusing.

Summing it up, my conclusion is this.

> ***Data structure*** refers to a structure - ***A specific way of storing and organizing data*** for efficient access.
> ***Data type*** refers to a type - ***A specification of data and its possible operations***.
>
> **Those terms are not binary opposition.**
> This means they if one has the other's characteristic or deterministic attribute, then there's no reason for it not to be the other.

# II. Algorithm

## 1. Understanding

### 1) Definition

> - ***A process or set of rules*** to be followed in calculations or other problem-solving operations, especially by a computer.

### 2) 5 Properties of Algotithm

#### - Input Specified

```
- Number of inputs must be larger than or equal to 0.
```

#### - Output Specified

```
- Number of outputs must be larger than or equal to 1.
```

#### - Definiteness

```
- Every process statment has to be absolutely clear.
```

#### - Finiteness

```
- Algorithm has to be stopped after intended or needed number of processes.
```

#### - Effectiveness

```
- Every process has to be properly executable.
```

### 3) Algorithm Description methods

#### - Pseudo Code

```
- Popularly used for its simplicity and clarity. Especially in research papers.
```

#### - Flow Chart

```
- Intuitive, but chart's complexity grows dramatically as algorithm's complexity does.
```

#### - Natural Language

```
- Convenient, but there's always certain level of ambiguity.
```

#### - Implementation Code

```
- In C or Java, implementation code more likely longer than pseudo code.
- But Python has almost overcome that.
```

## 2. Algorithms's efficiency measurement - Time & Space

In comparing efficiency of different algorithms, key factor of algorithm which determines efficiency would be either of the two followings.

- **Time** - Time taken in processing algorithm. Less time cost, better algorithms
- **Space** - Memory space taken in processing in algorithm. Less memory space requiring, better algorithms.

But these days, computers are fully armed with good enough size of memory. ***So people pay more attention to time cost side.***
And how do we estimate the time cost of algorithm?
Like this?

```
import time # importing built-in time module.

start = time.time() #Assigning started time to variable 'start'
'''
algorithms's processes
'''
end  = time.time() #Assigning ended time to variable 'end'
time_taken = end - start
```

But simply estimating processing time(runtime) can't be univesally accepted because of following reasons:

- **Must be implemented** : Algorithm must be fully implemented in physical form. - It can't provide reliable results from conceptual algorithms.

- **Hardware dependency** : Even though algorithms has been implemented, **processing time varies from on which machine(hardware) an algorithm is executed.**

- **Software dependency** : Processing time can be different by whether it is implemented in compiler language or interpreter language. - Compiler language generally provide faster processing than interpreter language.

- **Uncertainty** : Estimated runtime can't be claimed for untested input.

  Thus, **most generally accepted way** of estimating efficiency of algorithm is ***Complexity Analysis*** - **analyzing how runtime(the number of operations) or memory usage grows as input size grows.**
  And becuase this analysis focuses on functional relationship between runtime and input size, It has premise for its purpose that -

  **Premise**
  Every operation(+, -, x, /, assigning) takes same unit time(1) for processing.

## 1) Time Complexity Analysis - T(n)

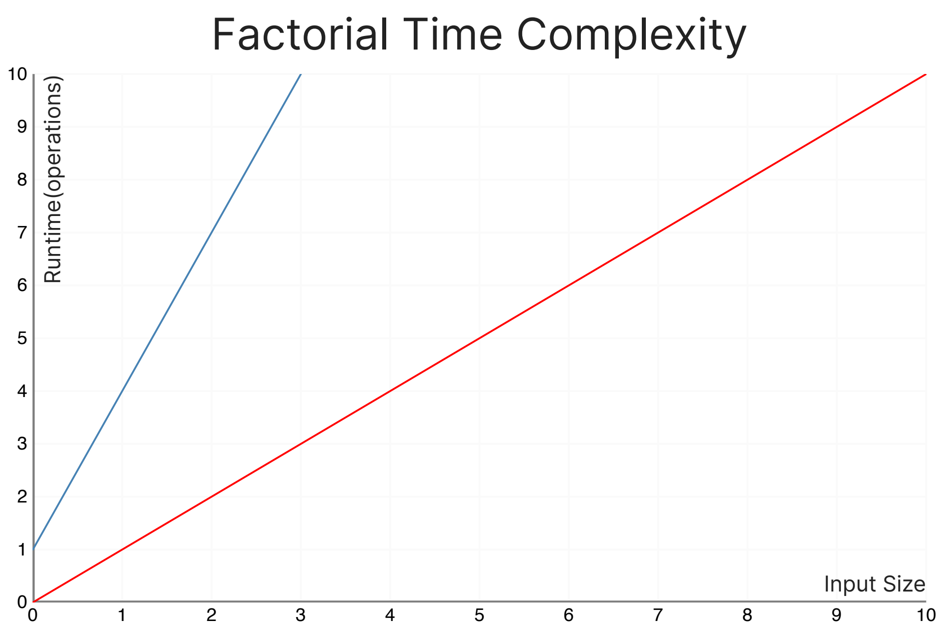
### Expression which describes functional relationship between runtime(the number of operations) and input size in specific algorithm.

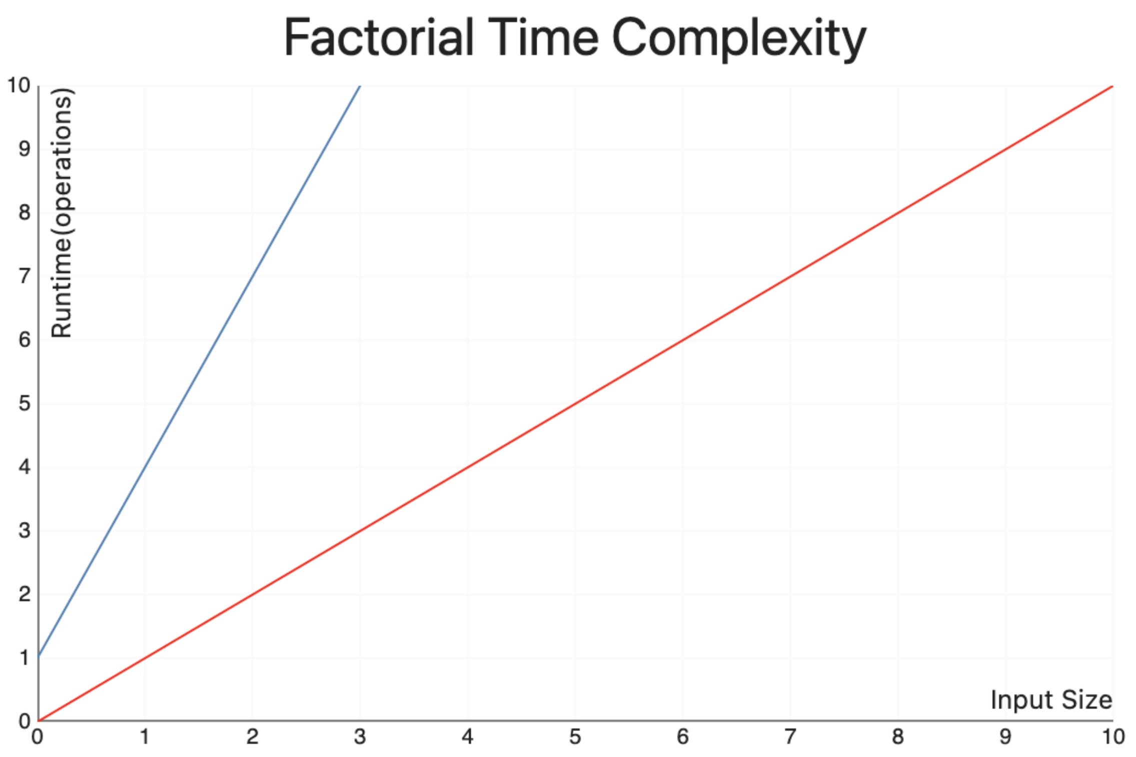
Time complexity analysis is mostly done *by counting frequency* - also called **Frequency Counting** or Operation Counting Method.

- Frequncy counting is just simply counting operations required for complete process of algorithm.
- So precise result of time complexity analysis is only one - Independent of the person analyzing it.
  - However, some operations are ignored mostly because of following reason:
    - Based on level of detail and rigor applied for some intention.

#### Example 1 - Time complexity analysis for factorial

```
'''
Mathematical definition on Factorial for positive integer is as following:

factorial(n) ----- 1 (n = 1)
                         ----- n(n-1) (n > 1)

'''
#1) Recursive : T(n) = 3n + 1 or c1n + c2 (In case of not considering premise.)
def factorial_recursive(n):
        if n == 1:
                return 1
        else :
                return n * factorial_recursive(n-1)

#2) Iteration : T(n) = n(considering only multiplications.)
def factorial_iteration(n):
        accum = 1
        if n == 1:
                return n
        else:
                for i in range(n, 0, -1):
                        acuum *= i
                return result
```





## 2) Space Complexity

**Space Complexity = Auxiliary Space(constants, variables, data structures used in algorithm) + Space used for input values.**

Space complexity analysis focuses on efficiency of memory usage by algorithm.
And there are 3 main reasons/purposes for algorithm to takes memory space.

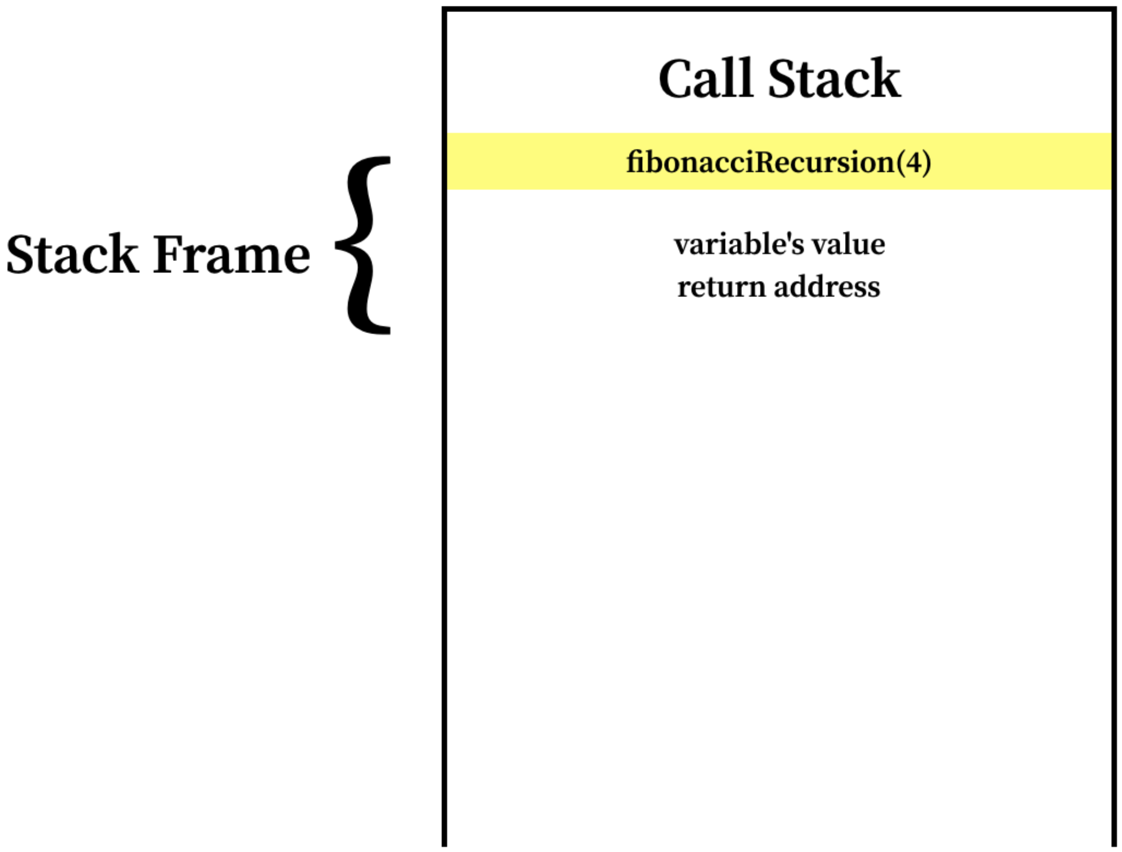
- **Instruction Space** : Space where complied code lines are stored.
- **Environmental Space**: Space where partially executed functions' values and addresses are stored. - ex. each stack frame in call stack per subroutines.
- **Data Space**: Space where constants, variables's values are stored.

**Example 2 - Space complexity analysis for fibonacci**

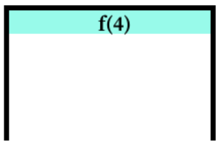
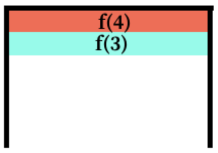
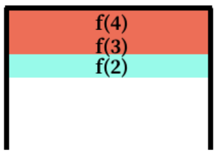
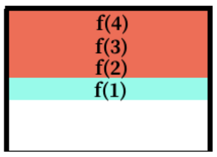
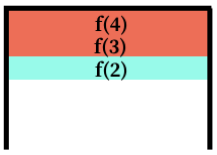
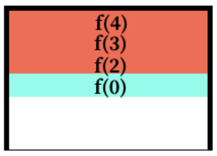
```
class Main {
        // 1) Recursive Implementation : Space(n) = Space taken by input + auxiliary space =
kn + 1(k = the size of single stack frame)
        static void fibonacciRecursion(int n) {
                // Base Case
                if(n<=1) {
                        return n;
                } else {
                        return fibonacciRecursion(n-1) + fibonacciRecursion(n-2);
                }
        }

        // 2) Iterative Implementation : Space(n) = j*5(fib0, fib1, fib2, cnt, n)(j = the size
of each variable's data type)
        static void fibonacciIter(int n) {
                int fib0 = 0;
                int fib1 = 1;
                int fib2;

                for(cnt==0;cnt<n;cnt++) {
                        fib2 = fib1 + fib0;
                        fib1 = fib2;
                        fib0 = fib1;
                        return fib0;
                }
        }
}
```

In iterative case, there's not much to take a look because its complexity is constant. - irrelevant to input.
But in recursion, input size really matters.
But to understand space complexity anaysis in case of recursion, what happening in memory during whole procedure has to be unsealed.
In memory, there's an area called '**Call Stack**' and inside it, there are '**Stack Frames**'.

- **Call Stack**(Interchangeably used with the term 'run-time stack')
    - A certain area in memory which structured in stack form.
    - Stores information(variables' value, return address, status) of active subroutines of computer program.
- **Stack Frame**
  - A unit of information stored in call stack
  - Grows rom highest address in allocated area to lower address(Downward)
  - Is allocated for both non-leaf functions & leaf functions
  - non-leaf functions: It calls other functions inside its procedure.
  - leaf functions : It doesn't call other functions.
  - Contains followings
  - Local variables / Arguements
  - Return address. - mostly fixed length(size)

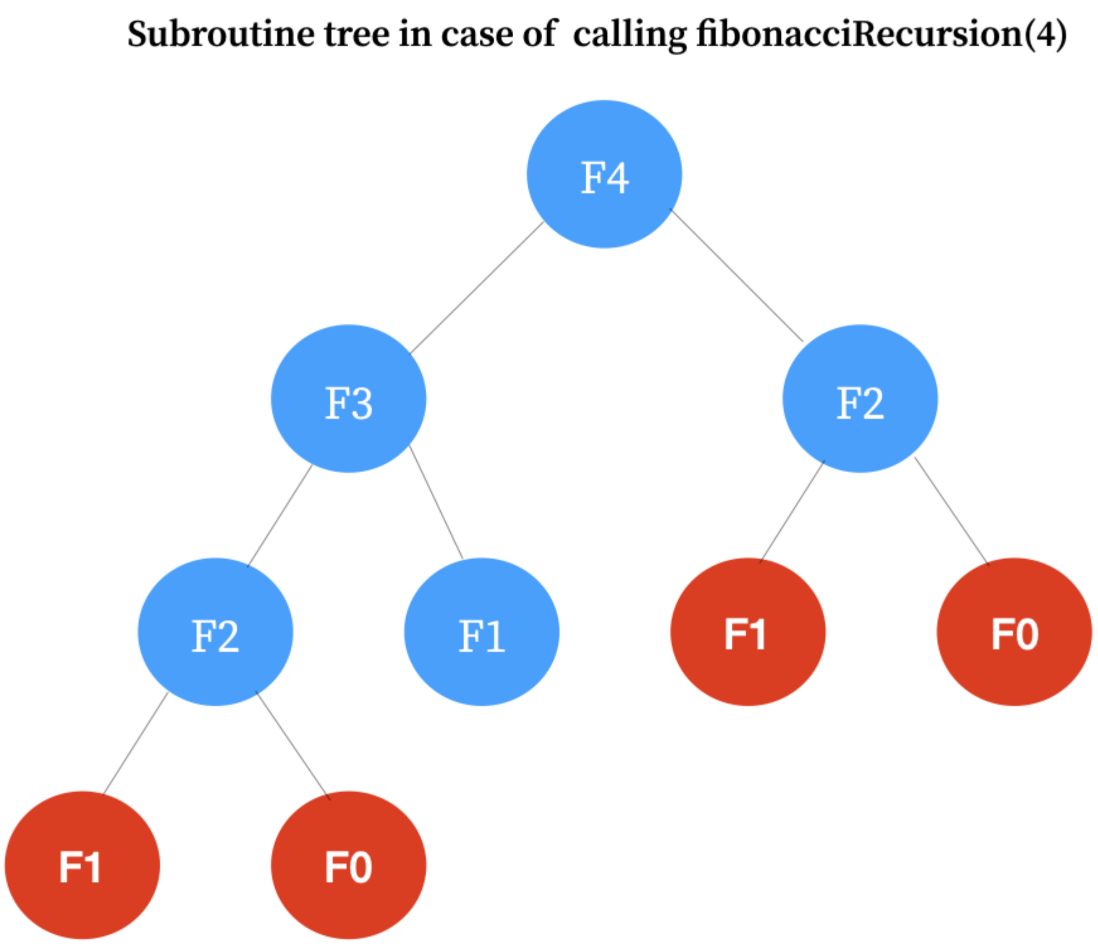


In case of fibonacciREcursion(4), overall function call procedures and call stack status at each line looks like this.

| **Command** | **Operation** | **Call Stack** |
|---|---|---|
| **1. Call f(4)** | f(3) + f(2) |  |
| **2. Call f(3)** | f(2) + f(1) |  |
| **3. Call f(2)** | f(1) + f(0) |  |
| **4. Call f(1)** | f(1) = 1 |  |
| **5. Return to f(2) stack frame** | f(2) = 1 + f(0) |  |
| **6. Call f(0)** | f(0) = 0 |  |

.
.
.

This function call procedures look just like a tree.



As seen, call stack usage of recursive function call reaches its max size when it arrives in ***base case***.
Simply speaking, memory usage of this algorithm takes as much **as the depth of this subroutine tree** at maximum.
And the depth of this tree is $n$.
Thus, space complexity of recursive fibonacci is $n + 1$.

## 2.1 Representing and comparing efficiency of algorithm: Asymptotic Analysis, Big-O/Big-Omega/Big-Theta notation.

But representing time/space complexity in functional term is not intuitive. And it has bunch of redundants.

```
'''
Time / Space Complexity Analysis on factorial and fibonacci sequence.

1. Factorial
'''

#1) Recursive : Time(n) = 3n + 1 / Space(n) = n + 1
def factorial_recursive(n):
        if n == 1:
                return 1

        else:
                return n * factorial_recursive(n-1)
#2) Iterative : Time(n) = n / Space(n) = 3(for variable accum, i, and constant n)
def factorial_iterative(n):
        accum = n
        for i in range(n, 0, -1):
                accum *= i
        return accum
```

```
'''
2. Fibonacci
'''

#1) Recursive : Time(n) = 2^n(roughly considered) / Space(n) = n
def fibonacci_recursive(n):
        if n <= 1:
                return n
        else:
                return fibonacci_recursive(n-1) + fibonacci_recursive(n-2)
#2) Iterative
def fibonacci_iterative(n):
        fib0 = 0
        fib1 = 1
        for i in range(2, n+1):
                fib2 = fib0 + fib1
                fib0 = fib1
                fib1 = fib2
        return fib2
```

But, using this analysis to represent algorithms' efficiency is not enough for mainly two reasons.

- In functional expressions of complexity analysis, **not every term in functional expressions can have meaningful effect** *if input size tends to infinity*.
- And in massively complicated algorithm, specifying exact time / space complexity in terms of function won't be easy. Eventually that might leads to imprecision, which means this methods won't be insightful anymore.
  Thus, we need **something that can overcome pre-stated cons to represent the functional relationship between input and runtime / space requirement.**
  Something reliable even in the situation where input size tends to infinity - this is the part making sure ***this representation's generalizability*** - and still bearing ***provability***.
  And they are **Big-O / Big-Omega / Big-Theta notation** - And they are all theorically based on **Asymptotic Analysis**(also known as **Asymptotics**)

## 2.1.1 Asymptotic Analysis

### 1) Definition

> $f(x) \sim g(x) \quad (\text{as } x \to \infty)$ (This is read as "f(x) is asymptotic to g(x)")

```
 *if and only if*  (by De brujin, "Asymptotic Methods in Analysis", 1981)
```

```
 ![{\displaystyle \lim _{x\to \infty }{\frac {f(x)}{g(x)}}=1.}]
(https://wikimedia.org/api/rest_v1/media/math/render/svg/5cb9bd7842f604400245e3e1f8ae62d1fbd5a
d20) (This limiting behavior description means that 'f(x) converges as argument tends to
inifinity'.)
```

Asymptotic analysis describes limiting behavior of a function.
Though the definition fomula tells almost everything(at least the essential part) about what it is, It's important for me to understand formula in a form decribed in language.
In language, asymptotic analysis can be understood or described in this sentence. - *As argument tends to infinity, **the term which has highest order dominate the whole result of the function***.

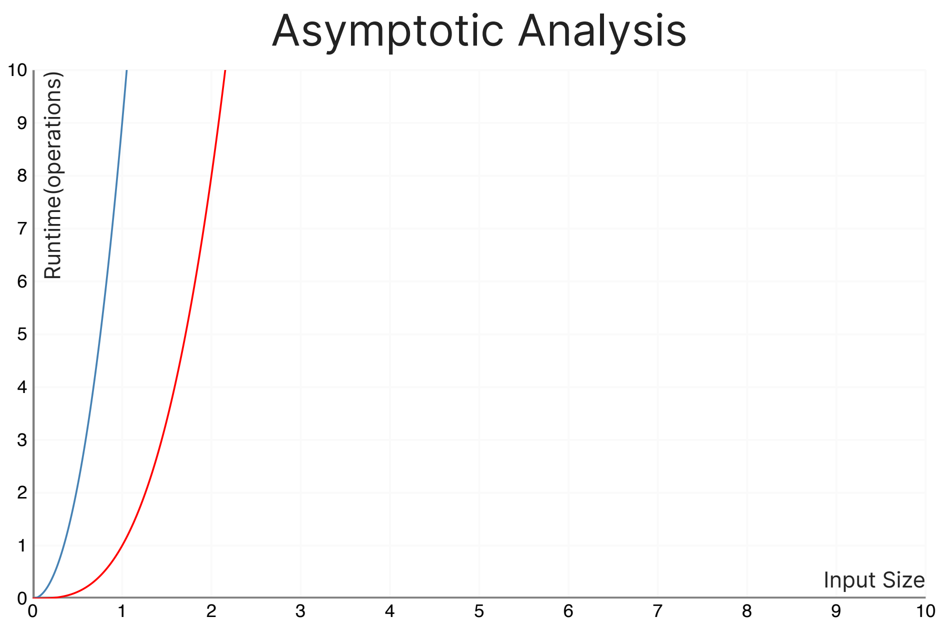
For example, let's suppose that a algorithm name A has time complexity for input size x as following-

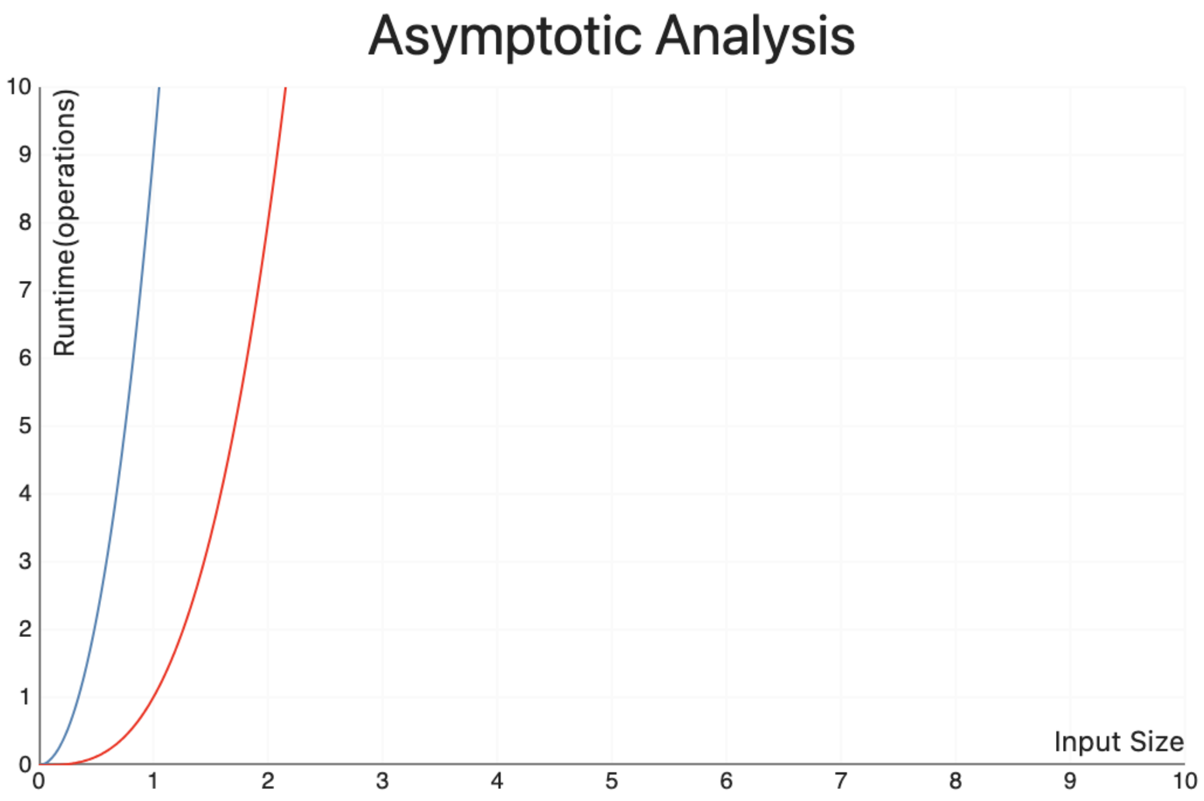
$$T(x) = x^3 + 8x^2$$

And let's set g(x) - Let it has simpler than, but still has same highest order with T(x).

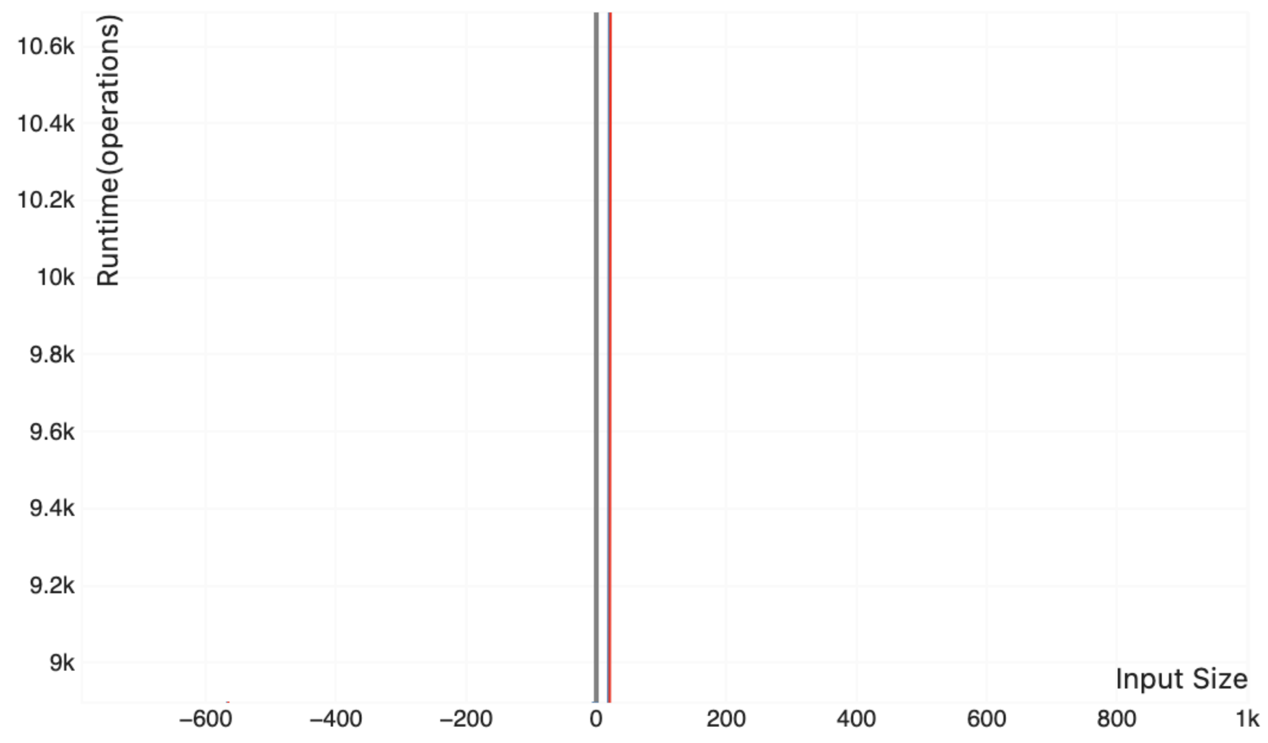
$$g(x) = x^3$$

And when we have subjectively small size of input(x), there are distinguishable difference between two functions.





But as input size gets large enough, difference in results between those two functions converges to zero.



And limiting behavior of fraction of T(x)/g(x) also satisfies the premise. - The fraction converges to '1'
Thus, ***T(x) is asymptotic to g(x)***.

Now, I get the basic concepts of this approach. But how this can be applied for evaluating algorithms?

Asymptotic analysis can give solid theorical basis and even quite handy representation of complicated polynomial function, even for algorithm's time/space complexity.
**But by itself alone is not sufficient for representing algorithm's complexity.**
**Because algorithm's complexity varies under certain circumstances.** - ex. Searching algorithm's time complexity can be different depends on whether the array is sorted or not.
Which means - ***Algorithm can show different efficiency under certain circumstances.***
It means thst **algorithm can have its worst, and its best, and its average case of efficiency.**
Then very important question float onto surface here
***"Then which efficiency of different case do we have to use to represent algorithm's efficiency / complexity?"***
Natrually, We need other representation or notation for algorithm's complexity that can imply contexts above to properly answer the question above.
Let me introduce, Big-O, Big-Omega, Big-Theta.

## 2.1.2 Big-O Notation - Upper bound

### 1) Definition

> Let,
> *f(x) : real / complex valued function. Expression for describing runtime/space usage functional growth as input size grows.*
> ***g(x)*** *: real valued function.* / Usually set as ***monomial function expression which f(x) is aymptotic to***.
> Two steps for characterizing f(x) in view of its functional growth rates and set g(x) of them.
> 1) If f(x) is polynomial, **keep the term which has the highest order** and omit all the others. - ***Asymptotics***
> 2) And if that term with the highest order is product of multiple factors, They can be omitted too if they're irrelevant to argument. - ex. constant multiplied to that term.
>
> ```
>   And let both funtions be unbounded subset of the *positive real numbers*, specifically
>  *g(x) be strictily positive for all large enough values of x*.
> ```
>
> **Formal Definition**
> *If positive real number M and a real number x0 that satisfy conditions below exists,*

> $|f(x)| \leq Mg(x) \quad \text{for all } x \geq x_0.$ (When the absolute value of f(x) is ***at most*** constant multiple of g(x) for x equal to or greater than x0)
>
> One can write as this.
> $f(x) = O(g(x)) \quad \text{as } x \to \infty$ *** Precisely, it is proper to represent in this way:** f(x)∊O(g(x)). **
>
> f(x) is O(g(x)) also true when following condition is true:
> $\limsup_{n\to\infty} \frac{f(n)}{g(n)} < \infty$ this can understood in this context: f(n) can be asymptotically decribed by product of finite constant and g(x).
>
> Since Big-O notation is originally invented and used in mathematical area, slightly more restricted definition of it used in computer science.
> Fundamental theory and logic is identical, but different in defining functions domain and codomain
> **-** *f(x) and g(x) set to be unbounded subset of positive integer to the non-negative real numbers.***
> \- Because both functions' argument is input size(which cannot be less than 0), and has complexity(the number of operations / space required which also can't be less than zero) as y-axis.

- Big-O notation is ***mathematical notation*** describes function's ***limiting behavior*** when x tends to infinity or specific value. - ***Asymptotic notation rooted in asymptotical analysis***
  - Mostly x tends to infinity when it comes to complexity analysis in computer science. - So that this notation of complexity/efficiency of algorithm **can be claimed onto untested inputs**
- A family of notation including Big-O notation are also called **Bachmann-Landau notation**.
  - This notations are invented by Paul Bachmann, Edmund Landau and others.
- As seen from formal definition above, Big-O notation(O(g(x))) represents ***upper bound*** of f(x) as x tends to infinity.
  - Big-O notation allows for only g(x) which satisfy $|f(x)| \leq Mg(x) \quad \text{for all } x \geq x_0.$ to stands for f(x) asymptotically.
  - This can be elaborated a bit easier in this phrase : **There's no way for f(x) to be slower(in runtime) / greater(in space complexity) than Mg(x) for all x equal to or greater than x0. *Thus, O(g(x)) stands for upper bound of f(x)*** - which is time/space complexity growth rate of an algorithm.
- Therefore, ***Big-O notation describes growth rate and its upper bound of f(x) in an asympotic way using eligible g(x) - which satisfies specific conditions, and to which f(x) is asymptotic.***
- Any other functions can be set as g(x) as far as they satisfy the conditions. - **But which offers minimum(most tight) upper bound considered as g(x) mostly.**
- Big-O v.s Little-o
  - Little-o
    - **"g(x) grows much faster than f(x)"**
    - Premises for f(x), g(x) are same with Big-O - f to be a real or complex valued function and g a real valued function. Both defined on some unbounded subset of the positive real numbers.
    - $f(x) = o(g(x)) \quad \text{as } x \to \infty$
      if,
      $|f(x)| \leq \varepsilon g(x) \quad \text{for all } x \geq x_0.$ But this was quite hard for me to identify the difference with Big-O notation.
      Landau originally defined little-o in this way.
      $$\lim_{x\to\infty} \frac{f(x)}{g(x)} = 0$$
      This literally meaning that g(x) grows much faster than f(x) as argument tends to inifinity.\

## 2) Available Properties

- Product
    - f1 = O(g1) and f2 = O(g2) ⇒ f1f2 = O(g1g2)
    - f(O(g)) = O(fg)
- Sum
- Conjunction

### 4) Order of common Big-O functions

| Notation | Name | Example |
|---|---|---|
| $O(1)$ | constant | Determining if a binary number is even or odd; Calculating $(-1)^n$; Using a constant-size lookup table |
| $O(\log \log n)$ | double logarithmic | Average number of comparisons spent finding an item using interpolation search in a sorted array of uniformly distributed values |
| $O(\log n)$ | logarithmic | Finding an item in a sorted array with a binary search or a balanced search tree as well as all operations in a binomial heap |
| $O((\log n)^c)$ $c > 1$ | polylogarithmic | Matrix chain ordering can be solved in polylogarithmic time on a parallel random-access machine. |
| $O(n^c)$ $0 < c < 1$ | fractional power | Searching in a k-d tree |
| $O(n)$ | linear | Finding an item in an unsorted list or in an unsorted array; adding two *n*-bit integers by ripple carry |
| $O(n \log^* n)$ | *n* log-star *n* | Performing triangulation of a simple polygon using Seidel's algorithm, or the union–find algorithm. Note that $\log^*(n) = \begin{cases} 0, & \text{if } n \leq 1 \\ 1 + \log^*(\log n), & \text{if } n > 1 \end{cases}$ |
| $O(n \log n) = O(\log n!)$ | linearithmic, loglinear, quasilinear, or "*n* log *n*" | Performing a fast Fourier transform; fastest possible comparison sort; heapsort and merge sort |
| $O(n^2)$ | quadratic | Multiplying two *n*-digit numbers by schoolbook multiplication; simple sorting algorithms, such as bubble sort, selection sort and insertion sort; (worst-case) bound on some usually faster sorting algorithms such as quicksort, Shellsort, and tree sort |
| $O(n^c)$ | polynomial or algebraic | Tree-adjoining grammar parsing; maximum matching for bipartite graphs; finding the determinant with LU decomposition |
| $L_n[\alpha, c] = e^{(c+o(1))(\ln n)^\alpha (\ln \ln n)^{1-\alpha}}$ $0 < \alpha < 1$ | L-notation or sub-exponential | Factoring a number using the quadratic sieve or number field sieve |
| $O(c^n)$ $c > 1$ | exponential | Finding the (exact) solution to the travelling salesman problem using dynamic programming; determining if two logical statements are equivalent using brute-force search |
| $O(n!)$ | factorial | Solving the travelling salesman problem via brute-force search; generating all unrestricted permutations of a poset; finding the determinant with Laplace expansion; enumerating all partitions of a set |

Source: Wikipedia

## 2.1.3 Big-Omega Notation - Lower bound

- There are two different definition on Big-Omega - And they are incompatiable.
    - Hardy-Littlewood Definition: Not going to be dealt in this notes deeply. Only definition would be noted for comparison.
        - Mostly used in analytic number theory.
        - They defined Big Omega as negation of little o.

- Knuth Definition
    - Mostly used in computational complexity theory

### 1) Hardy-Littlewood Definition

$f(x) = \Omega(g(x))$ as $x \to \infty$
if,

$$\limsup_{x\to\infty} \left|\frac{f(x)}{g(x)}\right| > 0.$$

### 2) Knuth Definition

Premises for both function f, g is identical to those of Big-O.
f: Set to be real / complex valued. Defined on some subset of positive integer to non-negative real number.
g: Set to be real valued. Defined on some subset of positive integer to non-negative real number.

As argument(let's say it is '$n$') tends to infinity and if there are positive real number $k$ which satisfies following condition for all $n >= n0$,

$$\exists k > 0 \, \exists n_0 \, \forall n > n_0 \colon f(n) \geq k \, g(n)$$

Then,

$$f(n) = \Omega(g(n))$$

Or, it can be defined with limiting behavior.

$$\liminf_{n\to\infty} \frac{f(n)}{g(n)} > 0$$

As can be led by definition above, this version of Big-Omega has something with Big-O.

$$f(x) = \Omega(g(x)) \Leftrightarrow g(x) = O(f(x))$$

This can be summarized by following sentence.
**"g(x) is asymptotical lower bound of f(x) as x tends to infinity."** - **As x tends to infinity, f(x) is as efficient as g(x) at its best case.**

## 2.1.4 Big-Theta Notation - On its average(By means, upper bound = lower bound).

- Big-Theta notation can be understood for ***representing algorithm A's time/space complexity's average efficiency asymptotically using another function***(which is asymptotic of it's complexity function expression) which satisfies certain conditions.
    - In here, the word 'average' seems a bit confusing but still valid ***because Big-Theta is used when Mg(x)(M is constant) can be both upper bound & lower bound of f(x)'s growth rate.***

### 1) Definition

f: Set to be real/complex valued function. Defined on some subset of positive integer to non-negative number.
g: Set to be real valued function. Defined same way with function f.

If there exist positive numbers $k0$ and $k1$ and they satisfy following condition for all $n$ larger than some positive number $n0$ ,

$\exists k_1 > 0 \, \exists k_2 > 0 \, \exists n_0 \, \forall n > n_0 \colon \; k_1 \, g(n) \leq f(n) \leq k_2 \, g(n)$,

Then,

$$f(n) = \Theta(g(n))$$

In limiting definition, Big-Theta comes to us slightly easier concepts.
If
$f(n) = O(g(n))$ and $f(n) = \Omega(g(n))$(Knuth version),

Then ,

$$f(n) = \Theta(g(n))$$