BIG O NOTATION

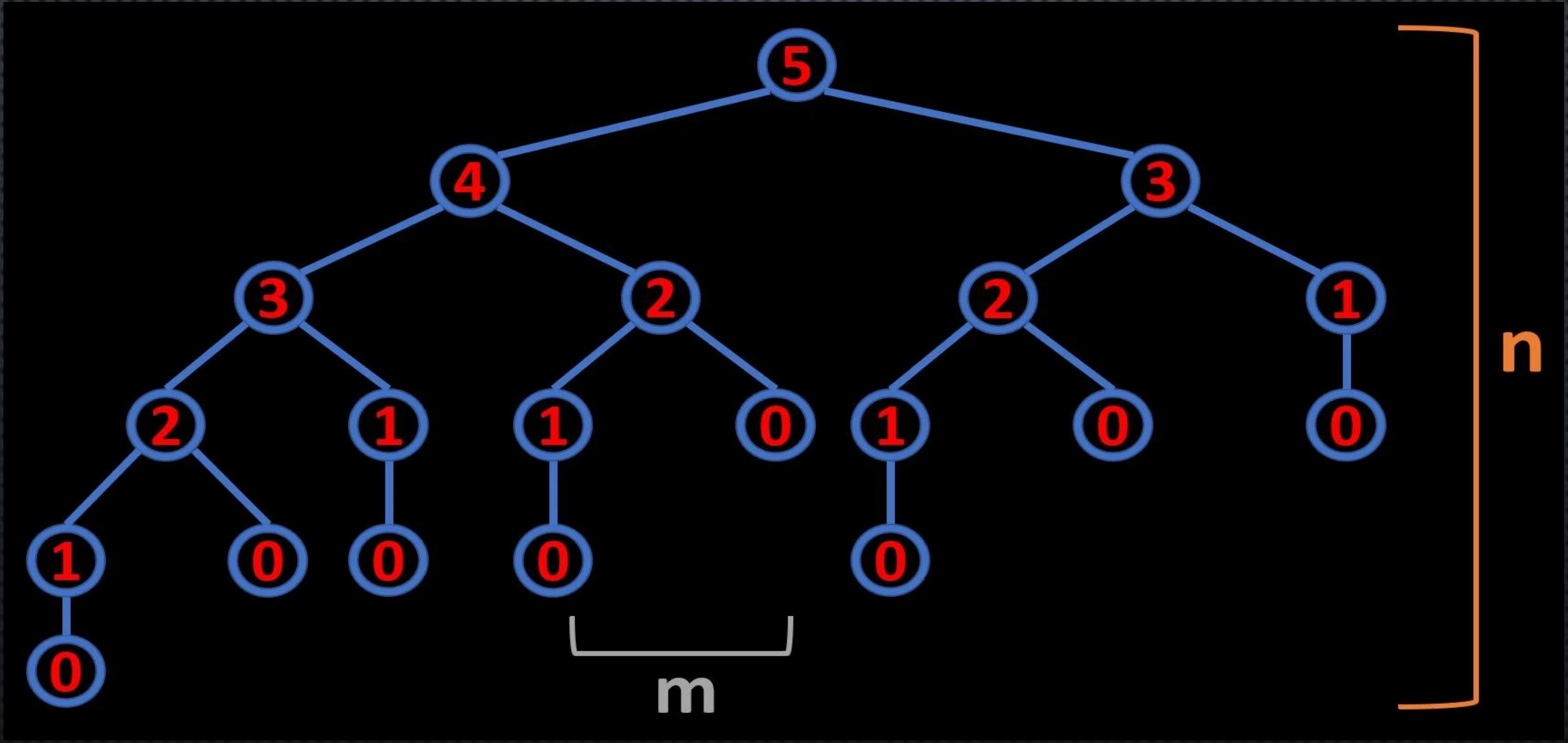
RECURSION

- A complicated function can be split down into smaller sub-problems utilizing recursion.
- Sequence creation is simpler through recursion than utilizing any nested iteration.
- A Recursive function will continue to call itself and repeat it's self-referential act until a condition is met in order to return a result
- A typical recursive function shares a common structure that consists of two parts:
  (i) **The recursive case**: The part which breaks down the problem into simpler ones.. and
  (ii) **The base case**: The terminating condition which stops the function once it is met.

Disadvantages

- A lot of memory and time is taken through recursive calls which makes it expensive for use.
- Recursive functions are challenging to debug.
- The reasoning behind recursion can sometimes be tough to think through.

VISUAL REPRESENTATION



SIMPLE RECURSION EXAMPLE

```
DEF HELLO():
    PRINT('HELLO')
    HELLO()
HELLO()
```

- THIS WILL GIVE RECURSION ERROR.

RECURSION ERROR EXPLAINED

- A RECURSION ERROR HAPPENS WHEN THE PROGRAM HAS PERFORMED A RECURSIVE FUNCTION A NUMBER OF TIMES GREATER THAN THAN THE RECURSION LIMIT. IN ORDER WORDS, A RECURSION ERROR IS THROWN WHEN YOUR (RECURSIVE) FUNCTION HAS CALLED ITSELF UP TO A PRE-DEFINED LIMIT CALLED THE RECURSION LIMIT.
- BY DEFAULT, THE RECURSION LIMIT IN A PYTHON PROGRAM IS 1000 TIMES.
- TO CHECK YOUR APPLICATIONS RECURSION LIMIT, YOU CAN RUN THE FOLLOWING CODE SNIPPET:

```
IMPORT SYS

PRINT(SYS.GETRECURSIONLIMIT())
```

- YOU WANT TO INCREASE OR DECREASE THE PREDEFINED RECURSION LIMIT IN YOUR PROGRAM, YOU CAN DO THAT BY CALLING THE `SYS.SETRECURSIONLIMIT()`

EXAMPLE THE FIBONACCI SEQUENCE

```
DEF RECURSIVE_FIBONACCI(N):
   IF N <= 1:
         RETURN N
   ELSE:
         RETURN(RECURSIVE_FIBONACCI(N-1) + RECURSIVE_FIBONACCI(N-2))
N_TERMS = 10
# CHECK IF THE NUMBER OF TERMS IS VALID
IF N_TERMS <= 0:
   PRINT("INVALID INPUT")
ELSE:
   PRINT("FIBONACCI SERIES:")
FOR I IN RANGE(N_TERMS):
      PRINT(RECURSIVE_FIBONACCI(I))
```

IMPORTANCE OF ANALYZING YOUR CODE

- Let's take an example for a code that calculates a factorial.

```
DEF FACT(N):
PRODUCT = 1
FOR I IN RANGE(N):
    PRODUCT = PRODUCT * (I+1)
    RETURN PRODUCT
PRINT(FACT(5))
```

A SECOND WAY OF DOING IT

```
DEF FACT2(N):
IF N == 0:
    RETURN 1
ELSE:
    RETURN N * FACT2(N-1)
PRINT(FACT2(5))
```

Continued

- The execution time shows that the first algorithm is faster compared to the second algorithm involving recursion. This example shows the importance of algorithm analysis. In the case of large inputs, the performance difference can become more significant.
- However, execution time is not a good metric to measure the complexity of an algorithm since it depends upon the hardware. A more objective complexity analysis metrics for the algorithms is needed. This is where Big O notation comes to play.

Algorithm Analysis with Big-O Notation

- Big-O notation is a metrics used to find algorithm complexity.
- Basically, Big-O notation signifies the relationship between the input to the algorithm and the steps required to execute the algorithm.
- It is denoted by a big "O" followed by opening and closing parenthesis.
- Inside the parenthesis, the relationship between the input and the steps taken by the algorithm is presented using "n".

COMMON BIG-O FUNCTIONS:

Name	**Big O**
Constant	O(c)
Linear	O(n)
Quadratic	O(n^2)
Cubic	O(n^3)
Exponential	O(2^n)
Logarithmic	O(log(n))
Log Linear	O(nlog(n))

CONSTANT COMPLEXITY (O(C))

- The complexity of an algorithm is said to be constant if the steps required to complete the execution of an algorithm remain constant, irrespective of the number of inputs. The constant complexity is denoted by O(c) where c can be any constant number.
- Example: a simple algorithm in Python that finds the square of the first item in the list and then prints it on the screen.

EXAMPLE

```
DEF CONSTANT_ALGO(ITEMS):
    RESULT = ITEMS[0] * ITEMS[0]
    PRINT(RESULT)
CONSTANT_ALGO([4, 5, 6, 8])
```

Visual Representation

```
import matplotlib.pyplot as plt
x = [2, 4, 6, 8, 10, 12]
y = [2, 2, 2, 2, 2, 2]
plt.plot(x, y, 'b')
plt.xlabel('Inputs')
plt.ylabel('Steps')
plt.title('Constant Complexity')
plt.show()
```

LINEAR COMPLEXITY (O(N))

- The complexity of an algorithm is said to be linear if the steps required to complete the execution of an algorithm increase or decrease linearly with the number of inputs. Linear complexity is denoted by O(n).

EXAMPLE

```
DEF LINEAR_ALGO(ITEMS):
FOR ITEM IN ITEMS:
    PRINT(ITEM)
LINEAR_ALGO([4, 5, 6, 8])
```

VISUAL REPRESENTATION

```
IMPORT MATPLOTLIB.PYPLOT AS PLT
X = [2, 4, 6, 8, 10, 12]
Y = [2, 4, 6, 8, 10, 12]
PLT.PLOT(X, Y, 'B')
PLT.XLABEL('INPUTS')
PLT.YLABEL('STEPS')
PLT.TITLE('LINEAR COMPLEXITY')
PLT.SHOW()
```

EXAMPLE 2

- ANOTHER POINT TO NOTE HERE IS THAT IN CASE OF A HUGE NUMBER OF INPUTS THE CONSTANTS BECOME INSIGNIFICANT. FOR INSTANCE, TAKE A LOOK AT THE FOLLOWING SCRIPT: THIS IS 0(2N)

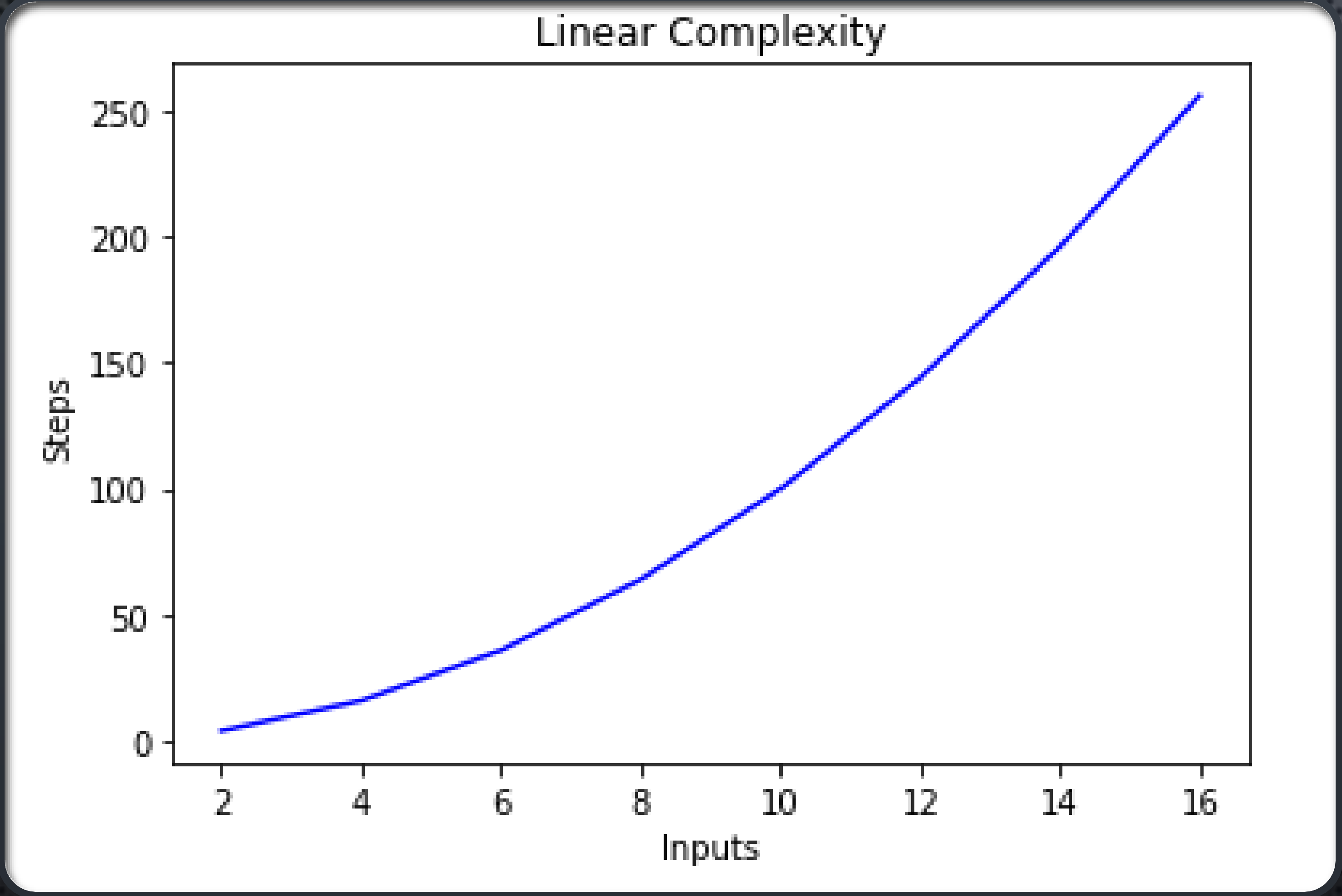
```
DEF LINEAR_ALGO(ITEMS):
FOR ITEM IN ITEMS:
    PRINT(ITEM)
FOR ITEM IN ITEMS:
    PRINT(ITEM)
LINEAR_ALGO([4, 5, 6, 8])
```

Quadratic Complexity (O(n^2))

- The complexity of an algorithm is said to be quadratic when the steps required to execute an algorithm are a quadratic function of the number of items in the input. Quadratic complexity is denoted as O(n^2).

EXAMPLE

```
DEF QUADRATIC_ALGO(ITEMS):
FOR ITEM IN ITEMS:
    FOR ITEM2 IN ITEMS:
        PRINT(ITEM, ' ' ,ITEM)
QUADRATIC_ALGO([4, 5, 6, 8])
```



VISUAL REPRESENTATION

WHAT IS THE COMPLEXITY AND WHY?

```
DEF COMPLEX_ALGO(ITEMS):
    FOR I IN RANGE(5):
        PRINT("TEST")
    FOR ITEM IN ITEMS:
        PRINT(ITEM)
    FOR ITEM IN ITEMS:
        PRINT(ITEM)
PRINT("BIG O")
PRINT("BIG O")
PRINT("BIG O")
COMPLEX_ALGO([4, 5, 6, 8])
```

WORST VS BEST CASE COMPLEXITY

- USUALLY, WHEN SOMEONE ASKS YOU ABOUT THE COMPLEXITY OF THE ALGORITHM HE IS ASKING YOU ABOUT THE WORST CASE COMPLEXITY.

```
DEF SEARCH_ALGO(NUM, ITEMS):
    FOR ITEM IN ITEMS:
        IF ITEM == NUM:
            RETURN TRUE
        ELSE:
            PASS
NUMS = [2, 4, 6, 8, 10]
PRINT(SEARCH_ALGO(2, NUMS))
```

EXPLANATION

In the script above, we have a function that takes a number and a list of numbers as input. It returns true if the passed number is found in the list of numbers, otherwise it returns **None**. If you search 2 in the list, it will be found in the first comparison. This is the best case complexity of the algorithm that the searched item is found in the first searched index. The best case complexity, in this case, is O(1). On the other hand, if you search 10, it will be found at the last searched index. The algorithm will have to search through all the items in the list, hence the worst case complexity becomes O(n).