An algorithm is a set of commands that must be followed for a computer to perform calculations or other problem-solving operations. an algorithm is a finite set of instructions carried out in a specific order to perform a particular task.

Properties:-

Input: An algorithm requires some input values. An algorithm should have 0 or more well-defined inputs.

OUTPUT: An algorithm should have 1 or more well-defined outputs, and should match the desired output. At the end of an algorithm, you will have one or more outcomes.

Unambiguity: A perfect algorithm is defined as unambiguous, which means that its instructions should be clear and straightforward.

Finiteness: An algorithm must be finite. Finiteness in this context means that the algorithm should have a limited number of instructions, i.e., the instructions should be countable.

EFFECTIVENESS: For an algorithm to be effective, it means that all those steps that are required to get to output must be feasible with the available resources. It should not contain any unnecessary and redundant steps which could make an algorithm ineffective.

Language independence: An algorithm must be language-independent, which means that its instructions can be implemented in any language and produce the same results.

Difference between algorithm and flow chart:-

Flow chart is pictorial representation of flow/process of task to perform whereas Algorithm is statement based description of task to be performed for achievement of desired output.

Parameters	Flowchart	Algorithm
Description	A flowchart is a graphical representation of the steps a program takes to process data. In this, we can use several geometric patterns to illustrate the numerous actions the program carries out.	An algorithm is a procedure or set of rules that defines how a program is to be executed. Or we can say that it is a set of instructions for solving a well-defined computational problem.
Complexity	It is easy to design and also very user friendly.	It is comparatively difficult to create and also a bit challenging to be understood by a layman.
Geometrical diagrams	It utilizes different types of geometrical shapes, symbols, and patterns.	An algorithm does not include any sort of geometrical pattern.
Scope of Usage	A flowchart can be used in different disciplines to describe a process.	Algorithms are used in the domain of mathematics and computer science.
Use	A flowchart is used in documenting, designing, and analyzing a program in different disciplines.	An algorithm is used to represent the concept of decidability.
Users	A Flowchart doesn't demand the knowledge of a computer programming language.	An algorithm demands the knowledge of a computer programming language.
Debugging	It is easy to debug the errors in flowcharts.	It is difficult to debug the errors in algorithms.
Implementation	In flowcharts, no rules are used.	In algorithms, predefined rules are used.
Branching and Looping	Simple to display branching and looping.	Hard to display branching and looping.
Solution	In a flowchart, the solution is represented in a graphical format.	In an algorithm, the solution is presented in non non-computer language.

Time complexity is the time taken by the algorithm to execute each set of instructions. It is always better to select the most efficient algorithm when a simple problem can solve with different methods.
Space complexity is usually referred to as the amount of memory consumed by the algorithm. It is composed of two different spaces; *Auxiliary space* and *Input space*.
Time complexity is profoundly related to the input size. As the size of the input increases, the run time (which is — time taken by the algorithm to run) also increases.
Primarily there are three types of Asymptotic notations:
1.Big-Oh (O) notation.
2.Big Omega (Ω) notation.
3.Big Theta (Θ) notation — widely used.
4.Best Case: It defines as the condition that allows an algorithm to complete the execution of statements in the minimum amount of time. In this case, the execution time acts as a lower bound on the algorithm's time complexity.
5.Average Case: In the average case, we get the sum of running times on every possible input combination and then take the average. In this case, the execution time acts as both lower bound and upper bound on the algorithm's time complexity.
6.Worst Case: It defines as the condition that allows an algorithm to complete the execution of statements in the maximum amount of time. In this case, the execution time acts as an upper bound on the algorithm's time complexity.

Computer scientists learn by experience. We learn by seeing others solve problems and by solving problems by ourselves.
Being exposed to different problem-solving techniques and seeing how different algorithms are designed helps us to take on the next challenging problem that we are given.
By considering a number of different algorithms, we can begin to develop pattern recognition so that the next time a similar problem arises, we are better able to solve it. As we study algorithms, we can learn analysis techniques that allow us to compare and contrast solutions based solely on their own characteristics, not the characteristics of the program or computer used to implement them.

A problem is said to satisfy the Principle of Optimality if the sub solutions of an optimal solution of the problem are themselves optimal solutions for their subproblems. Examples: The shortest path problem satisfies the Principle of Optimality.

An optimal solution is **a feasible solution where the objective function reaches its maximum (or minimum) value** – for example, the most profit or the least cost. A globally optimal solution is one where there are no other feasible solutions with better objective function values

A feasible solution is **one that satisfies all defined constraints and requirements**. A solution is infeasible when no combination of decision variable values can satisfy the entire set of requirements and constraints

$$1 < \log n < n < n\log n < n^2 < 2^n < 3^n$$

$1 < \log n < n < n\log n < n^2 < n^3 < 2^n < n!$

The main idea of asymptotic analysis is to have a measure of the efficiency of algorithms that don't depend on machine-specific constants and don't require algorithms to be implemented and time taken by programs to be compared.
Asymptotic notations are mathematical tools to represent the time complexity of algorithms for asymptotic analysis.

There are mainly three asymptotic notations:

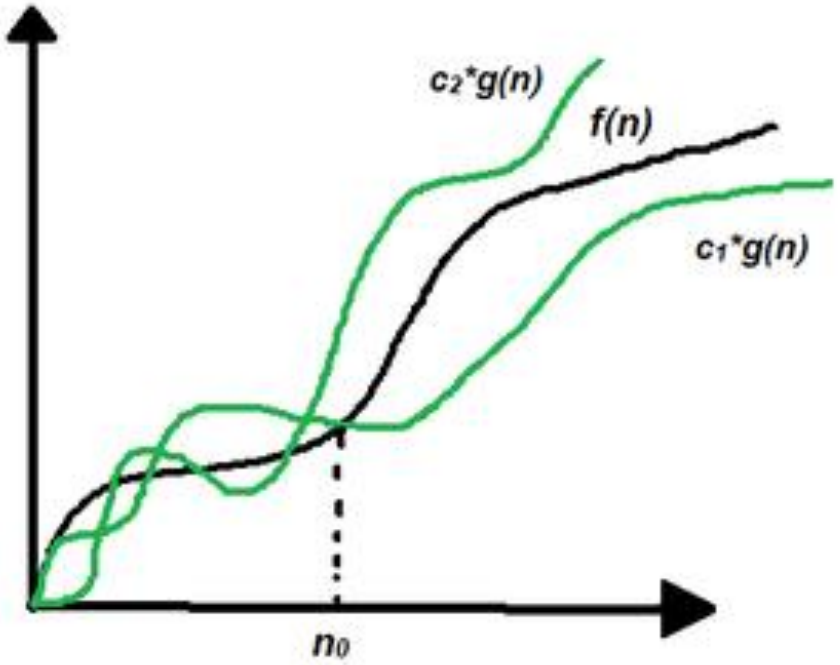
1. Big-O Notation (O-notation)
2. Omega Notation (Ω-notation)
3. Theta Notation (Θ-notation)

Theta Notation

- Theta notation encloses the function from above and below. Since it represents the upper and the lower bound of the running time of an algorithm, it is used for analysing the average-case complexity of an algorithm.

- Θ (g(n)) = {f(n): there exist positive constants c1, c2 and n0 such that $0 \leq c1 * g(n) \leq f(n) \leq c2 * g(n)$ for all $n \geq n0$}

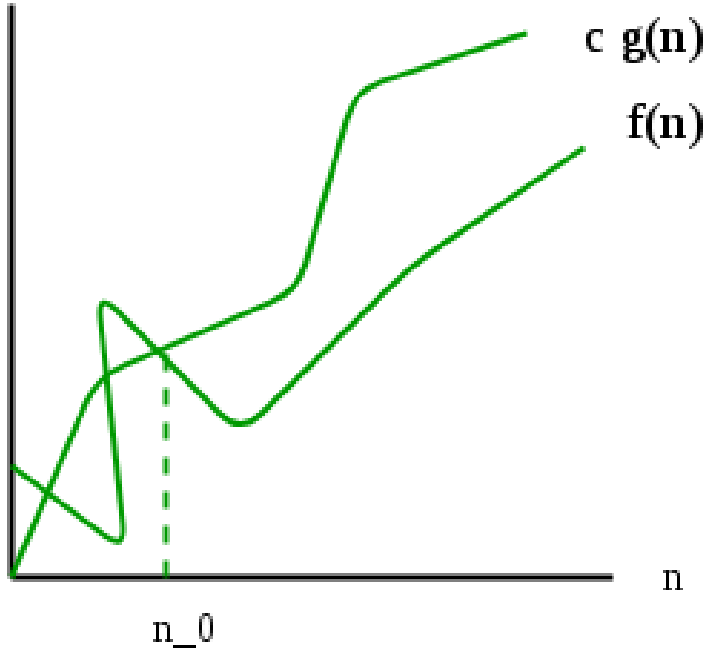
Note: Θ(g) is a set



Big-O Notation

- Big-O notation represents the upper bound of the running time of an algorithm. Therefore, it gives the worst-case complexity of an algorithm.

- $O(g(n)) = \{ f(n)$: there exist positive constants c and n0 such that $0 \leq f(n) \leq cg(n)$ for all $n \geq n0 \}$

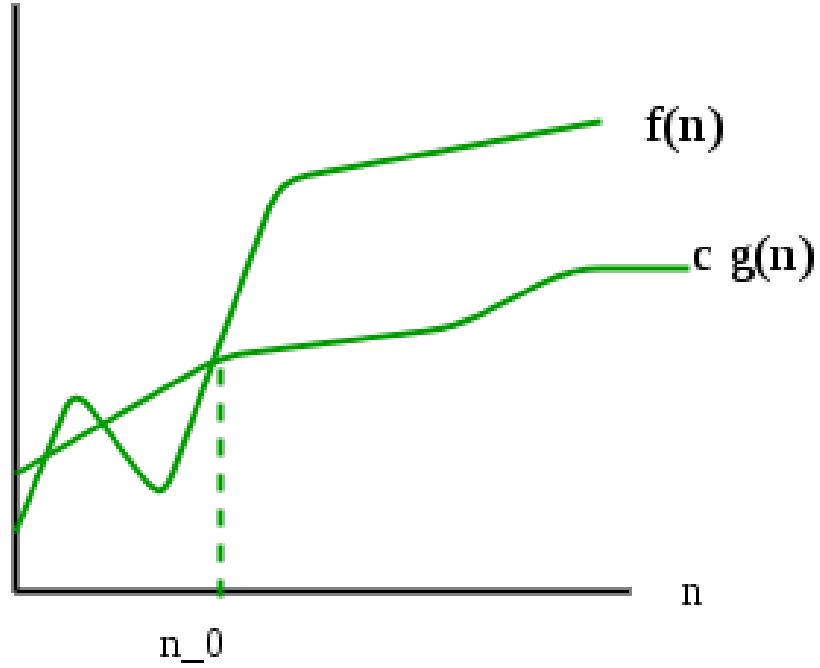


f(n) = O(g(n))

Omega Notation

- Omega notation represents the lower bound of the running time of an algorithm. Thus, it provides the best case complexity of an algorithm.

- $\Omega(g(n)) = \{ f(n)$: there exist positive constants c and n0 such that $0 \leq cg(n) \leq f(n)$ for all $n \geq n0 \}$



f(n) = Omega(g(n))

$$T(n) = 2T(n/2) + n^2$$

$$= 2^2 T\left(\frac{n}{2^2}\right) + 2\left(\frac{n}{2}\right)^2 + n^2$$

$$= 2^n T\left(\frac{n}{2^n}\right) + 2^{n-1}\left(\frac{n}{2}\right)^{n-1} + \cdots n$$

$\therefore O(2^n)$ if $n = \infty$

$$T(n) = 7T(n-1) + n^3$$

$$= 7^2 T(n-2) + 7(n-1)^3 + n^3$$

$$= 7^n T(n-n) + 7^{n-1}(n-(n-1))^3 + \cdots + n^3$$

$\therefore O(7^n)$ when $n \to \infty$

Counting Sort

- Counting sort is a sorting technique based on keys between a **specific range**.

- It works by counting the number of objects having distinct key values (a kind of hashing).

- Then do some arithmetic operations to calculate the position of each object in the output sequence.

Contd.

- Example:

Input size 10

Range is 6

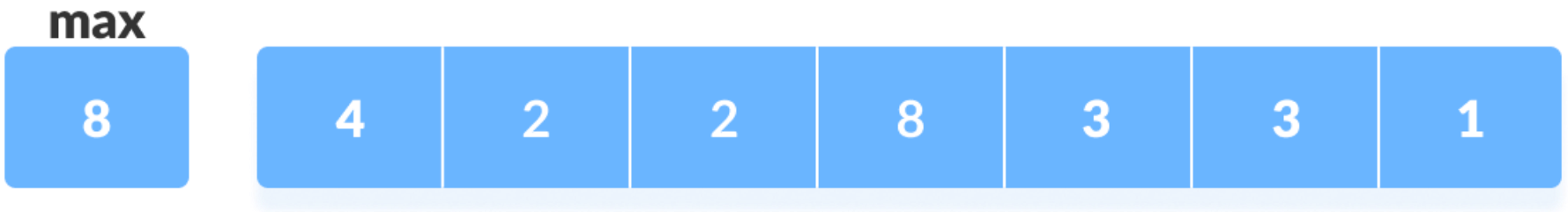
Sort: 1, 5, 6, 2, 3, 2, 5, 1, 6, 2

Characteristics

- Counting sort makes assumptions about the data, for example, **it assumes that values are going to be in the range of 0 to 10 or 10 – 99**, etc, Some other assumption counting sort makes is **input data will be all integer numbers (workaround for floating point numbers is possible)**.
- Like other algorithms this sorting algorithm is **not a comparison-based** algorithm, it hashes the value in a temporary count array and uses them for sorting.
- It uses a **temporary array** making it a **non-In Place** algorithm.

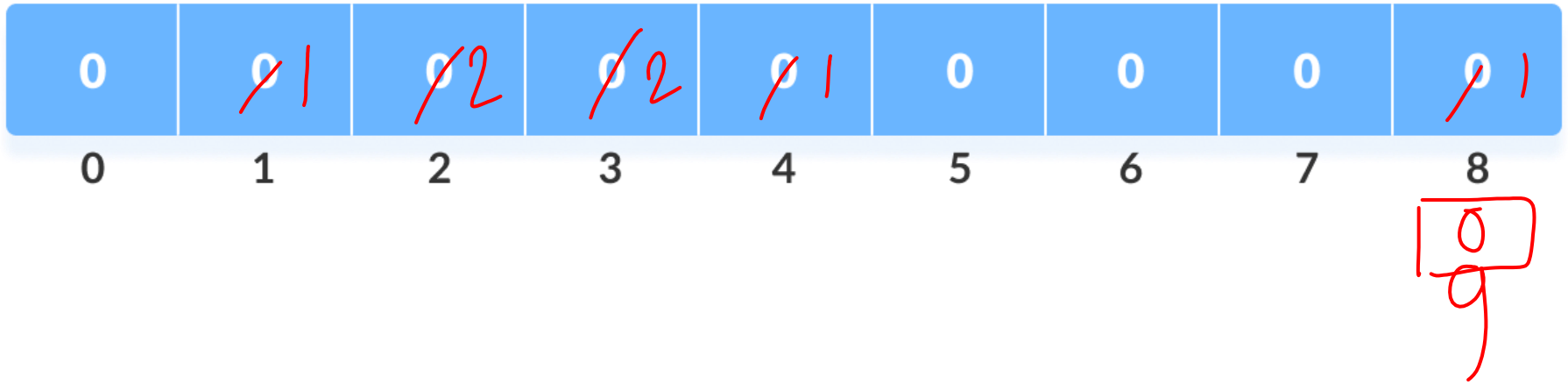
Example

- For simplicity, consider the data in the range of 0 to 9.
- Input data: {4, 2, 2, 8, 3, 3, 1}
- Find out the maximum element (let it be max) from the given array.



Contd.

- Initialize an array of length max+1 with all elements 0. This array is used for storing the count of the elements in the array.



Contd.

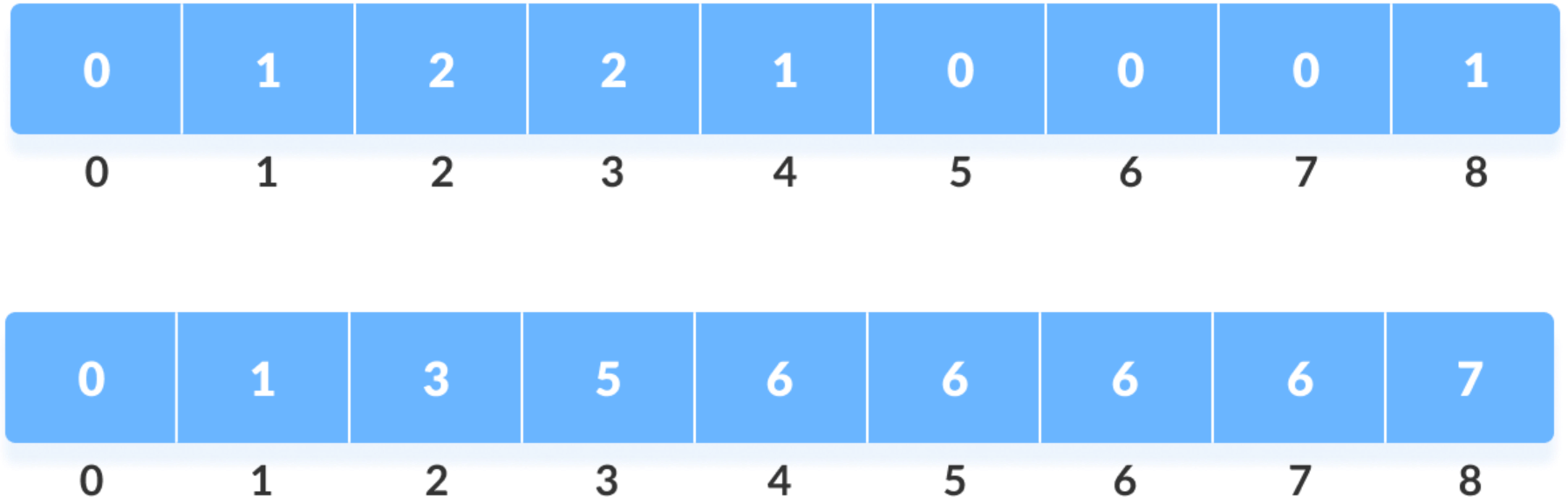
- Store the count of each element at their respective index in count array

- For example: if the count of element 3 is 2 then, 2 is stored in the 3rd position of count array. If element "5" is not present in the array, then 0 is stored in 5th position.

Contd.

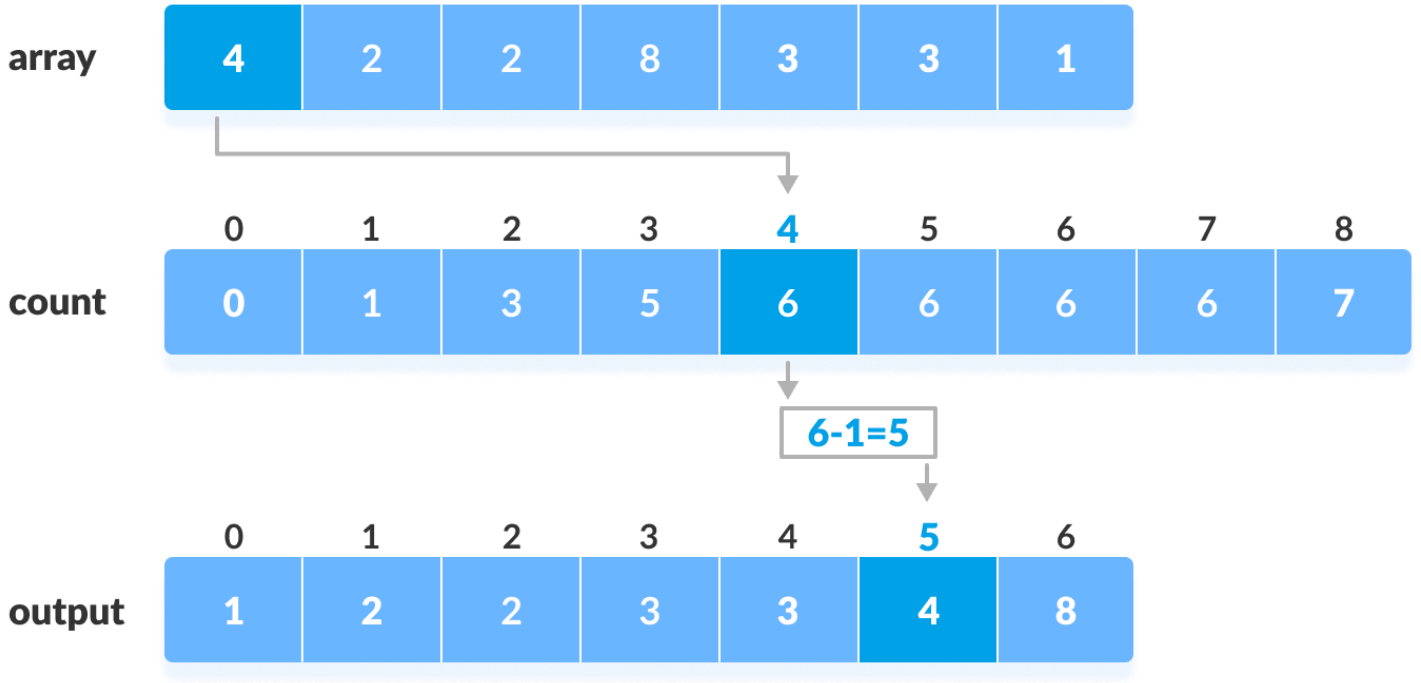
Contd.

- Store cumulative sum of the elements of the count array. It helps in placing the elements into the correct index of the sorted array.



Contd.

- Find the index of each element of the original array in the count array. This gives the cumulative count.



Contd.

- After placing each element at its correct position, decrease its count by one.

Code: Python

```
# Counting sort in Python programming



def countingSort(array):
    size = len(array)
    output = [0] * size

    # Initialize count array
    count = [0] * 10

    # Store the count of each elements in count array
    for i in range(0, size):
        count[array[i]] += 1

    # Store the cummulative count
    for i in range(1, 10):
        count[i] += count[i - 1]

    # Find the index of each element of the original array
in count array
    # place the elements in output array
    i = size - 1
    while i >= 0:
        output[count[array[i]] - 1] = array[i]
        count[array[i]] -= 1
        i -= 1

    # Copy the sorted elements into original array
    for i in range(0, size):
        array[i] = output[i]


data = [4, 2, 2, 8, 3, 3, 1]
countingSort(data)
print("Sorted Array in Ascending Order: ")
print(data)
```

Complexity

Best	Average	Worst
O(n+k)	O(n+k)	O(n+k)

Space
O(max)

Radix Sort

- Radix sort is a sorting algorithm that sorts the elements by first grouping the individual digits of the same place value. Then, sort the elements according to their increasing/decreasing order.

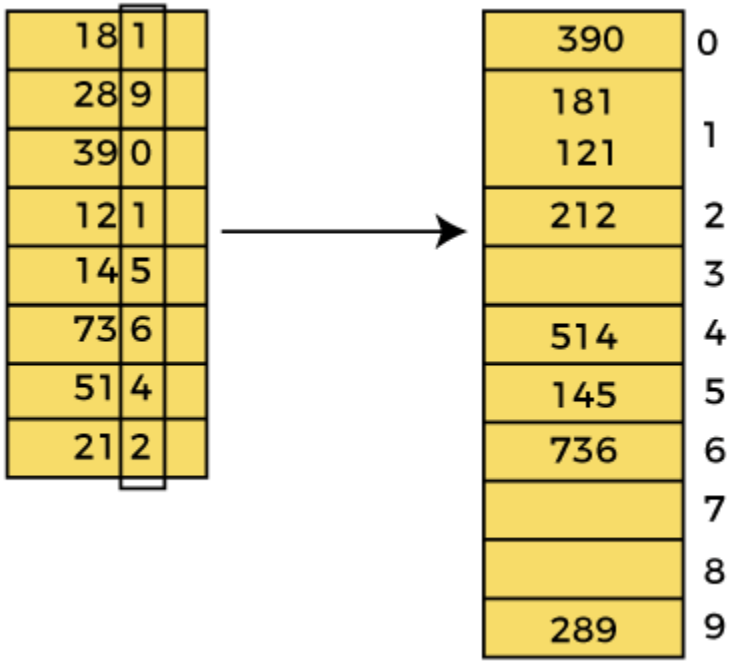
- Suppose, we have an array of 8 elements. First, we will sort elements based on the value of the unit place. Then, we will sort elements based on the value of the tenth place. This process goes on until the last significant place.

Steps

- First, we have to find the largest element (suppose **max**) from the given array. Suppose '**x**' be the number of digits in **max**. The '**x**' is calculated because we need to go through the significant places of all elements.

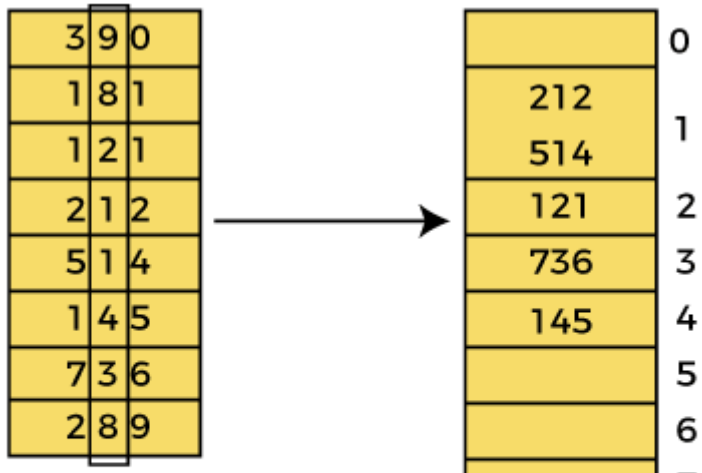
- After that, go through one by one each significant place. Here, we have to use any stable sorting algorithm to sort the digits of each significant place.

Pass 1



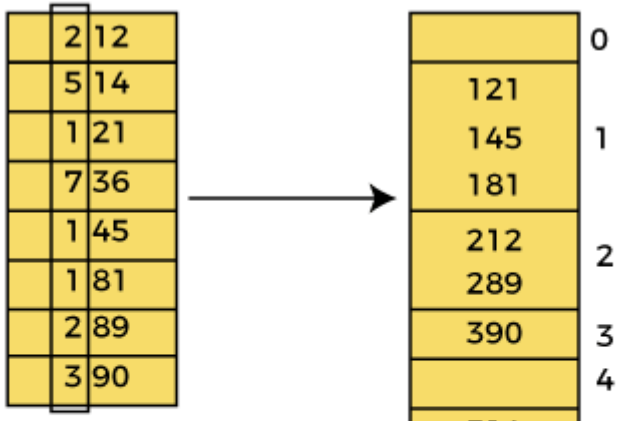
390	181	121	212	514	145	736	289

Pass 2



212	514	121	736	145	181	289	390

Pass 3



121	145	181	212	289	390	514	736

Code

```
def countingSort(arr, exp1):
    n = len(arr)
    output = [0] * (n)
    count = [0] * (10)
    for i in range(0, n):
        index = arr[i] // exp1
        count[index % 10] += 1
    for i in range(1, 10):
        count[i] += count[i - 1]
    i = n - 1
    while i >= 0:
        index = arr[i] // exp1
        output[count[index % 10] - 1] = arr[i]
        count[index % 10] -= 1
        i -= 1
    i = 0
    for i in range(0, len(arr)):
        arr[i] = output[i]

def radixSort(arr):
    max1 = max(arr)
    exp = 1
    while max1 / exp >= 1:
        countingSort(arr, exp)
        exp *= 10

arr = [170, 45, 75, 90, 802, 24, 2, 66]
radixSort(arr)
for i in range(len(arr)):
    print(arr[i],end=" ")
```

Complexity Analysis

Best	Average	Worst
Ω(d*k)	θ(d*n)	O(d*n)

d is the number of digits in the largest number
n is the length of input

The time complexity is independent of whether the input is sorted or not.

Fibonacci Series

```
def fib(n):
    # Stop condition
    if (n == 0):
        return 0
    # Stop condition
    if (n == 1 or n == 2):
        return 1
    # Recursion function
    else:
        return (fib(n - 1) + fib(n - 2))
```

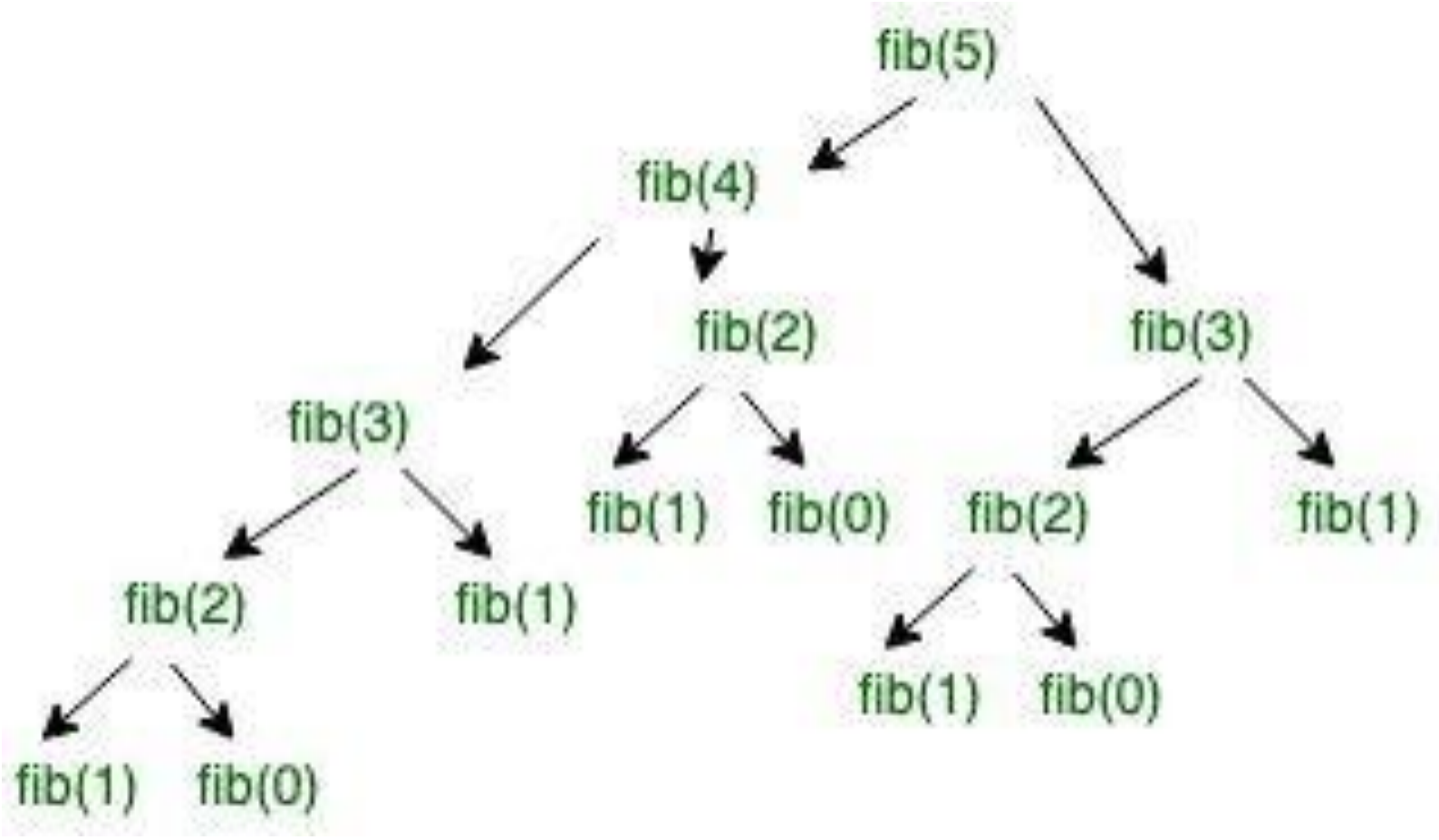
```
n = 5;

print("Fibonacci series of 5
numbers is :",end=" ")


for i in range(0,n):

    print(fib(i),end=" ")
```

Working


fib(5)
fib(4)
fib(3)
fib(2)
fib(3)
fib(2)
fib(1)
fib(1)
fib(0)
fib(2)
fib(1)
fib(1)
fib(0)
fib(1)
fib(0)

Recurrence Relation

- A recurrence relation is an equation which represents a sequence based on some rule.

- It helps in finding the subsequent term (next term) dependent upon the preceding term (previous term).

- If we know the previous term in a given series, then we can easily determine the next term.

Example

Base Case:

n if n == 0, n == 1;

Meaning $T(0) = 0$, $T(1) = 1$

Recursive Case:

fib(n) = fib(n-1) + fib(n-2) otherwise;

Meaning $T(n) = T(n-1) + T(n-2) + 1$

Solving Recurrence Relation

$$T(n) = T(n-1) + T(n-2) + 1$$

Assuming $T(n-1) \approx T(n-2)$

$$T(n) = 2 * T(n-1) + 1$$

Solving it we get

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

Space Complexity

- The space complexity for Fibonacci series is O(n). Let us look at process stack.