ALGORITHMS

An algorithm is any well-defined computational procedure that takes some value, or set of values, as input and produces some value, or set of values, as output. An algorithm is thus a sequence of computational steps that transform the input into the output. Algorithms are used to solve complex problems, automate tasks, and make informed decisions.

Some basic properties of an Algorithm

- Finiteness

An algorithm must end after a finite time. This means that the algorithm must be limited by a finite number of characters or memory.

- Definiteness

Each step in an algorithm must be clearly defined and unambiguous. This means that there should be no room for confusion or misinterpretation.

- Determinism

An algorithm must produce the same output for the same input.

- Effectiveness

Each step in an algorithm must be simple enough to express on paper and make sense for the quantities used.

- Input

An algorithm takes input data, which can be in the form of numbers, text, or images.

- Output

An algorithm produces at least one output, which is the result of the computation performed on the input.

- Other properties of an algorithm include: well-defined inputs, language independence, and feasibility.

Complexity of an Algorithm

- To measure performance of algorithms, we typically use time and space complexity analysis. The idea is to measure order terms of input size.
- Independent of the machine and its configuration, on which the algorithm is running on.
- Shows a direct correlation with the number of inputs.
- Can distinguish two algorithms clearly without ambiguity.

Time Complexity

The time complexity of an algorithm quantifies the amount of time taken by an algorithm to run as a function of the length of Note that the time to run is a function of the length of the input and not the actual execution time of the machine on which is running on. The valid algorithm takes a finite amount of time for execution. The time required by the algorithm to solve is called ***time complexity*** of the algorithm. Time complexity is very useful measure in algorithm analysis.

Space Complexity

- Problem-solving using computer requires memory to hold temporary data or final result while the program is in execution. The amount of memory required by the algorithm to solve given problem is called **space complexity** of the algorithm.
- The space complexity of an algorithm quantifies the amount of space taken by an algorithm to run as a function of the length of the input. Consider an example: Suppose a problem to find the frequency of array elements.
- It is the amount of memory needed for the completion of an algorithm.
- To estimate the memory, we need to focus on two parts:

(1) A fixed part: It is independent of the input size. It includes memory for instructions (code), constants, variables, etc.

(2) A variable part: It is dependent on the input size. It includes memory for recursion stack, referenced variables, etc.

Time and Space Trade-off

- A "time and space trade-off" in algorithms refers to the concept where an algorithm can solve a problem faster by using more memory, or conversely, use less memory by taking more time to execute, meaning there is an inherent balance between the computational time needed and the amount of storage space required to perform a calculation; essentially, optimizing one often comes at the cost of the other.

When designing an algorithm, developers often need to decide whether to prioritize faster execution (lower time complexity) even if it means using more memory (higher space complexity), or vice versa.

Example scenarios:

1. Pre-computing data: Storing pre-calculated values in a lookup table can significantly speed up lookups but requires additional memory to store the table.

2. Dynamic programming: While often faster than brute force methods, dynamic programming may require storing intermediate results, increasing space usage.

3. Data structures: Choosing a data structure like a hash table can provide fast access time (O(1)) but might need more space to maintain the hash function.

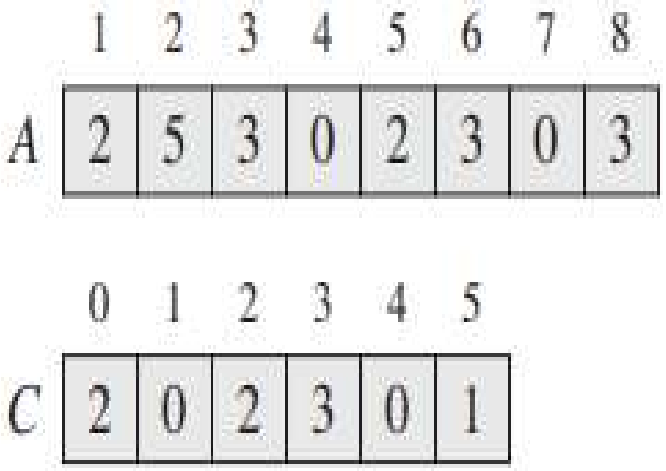
Introduction

Counting sort assumes that each of the n input elements is an integer in the range 0 to k, for some integer k. When k =O(n), the sort runs in Θ(n) time. Time Complexity is Θ(n+k), if k <<<<n, then complexity is Θ(n).

Example:

A={2,5,3,0,2,3,0,3}, here, n=8,k varies from 0 to 5, hence mod(k)<n.

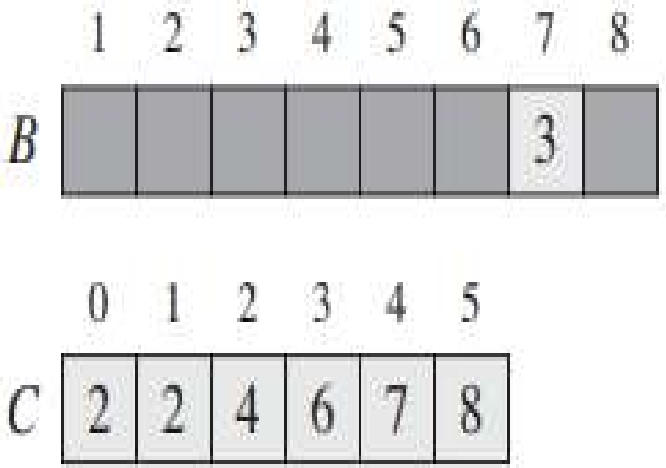
2	**5**	**3**	**0**	**2**	**3**	**0**	**3**


1 2 3 4 5 6 7 8
A 2 5 3 0 2 3 0 3
0 1 2 3 4 5
C 2 0 2 3 0 1

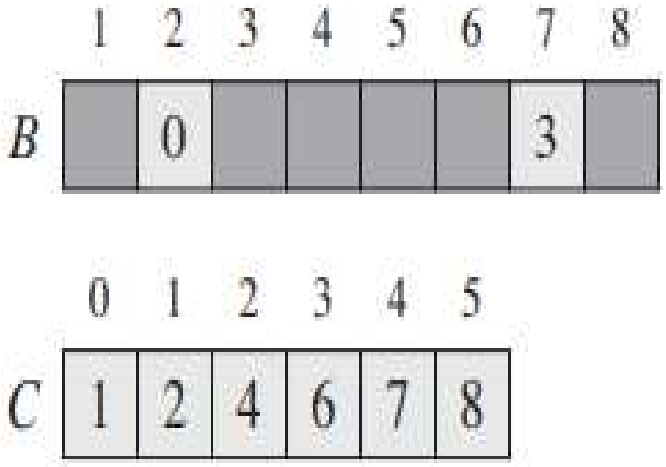
(a)

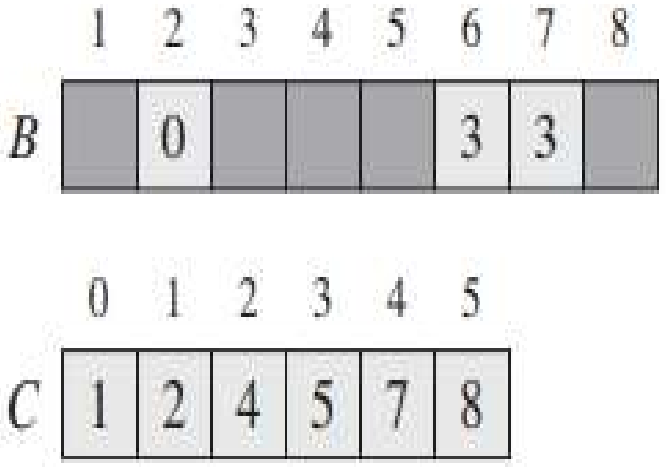

0 1 2 3 4 5
C 2 2 4 7 7 8

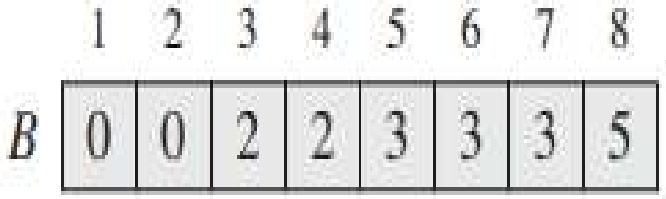
(b)


1 2 3 4 5 6 7 8
B 3
0 1 2 3 4 5
C 2 2 4 6 7 8

(c)


1 2 3 4 5 6 7 8
B 0 3
0 1 2 3 4 5
C 1 2 4 6 7 8

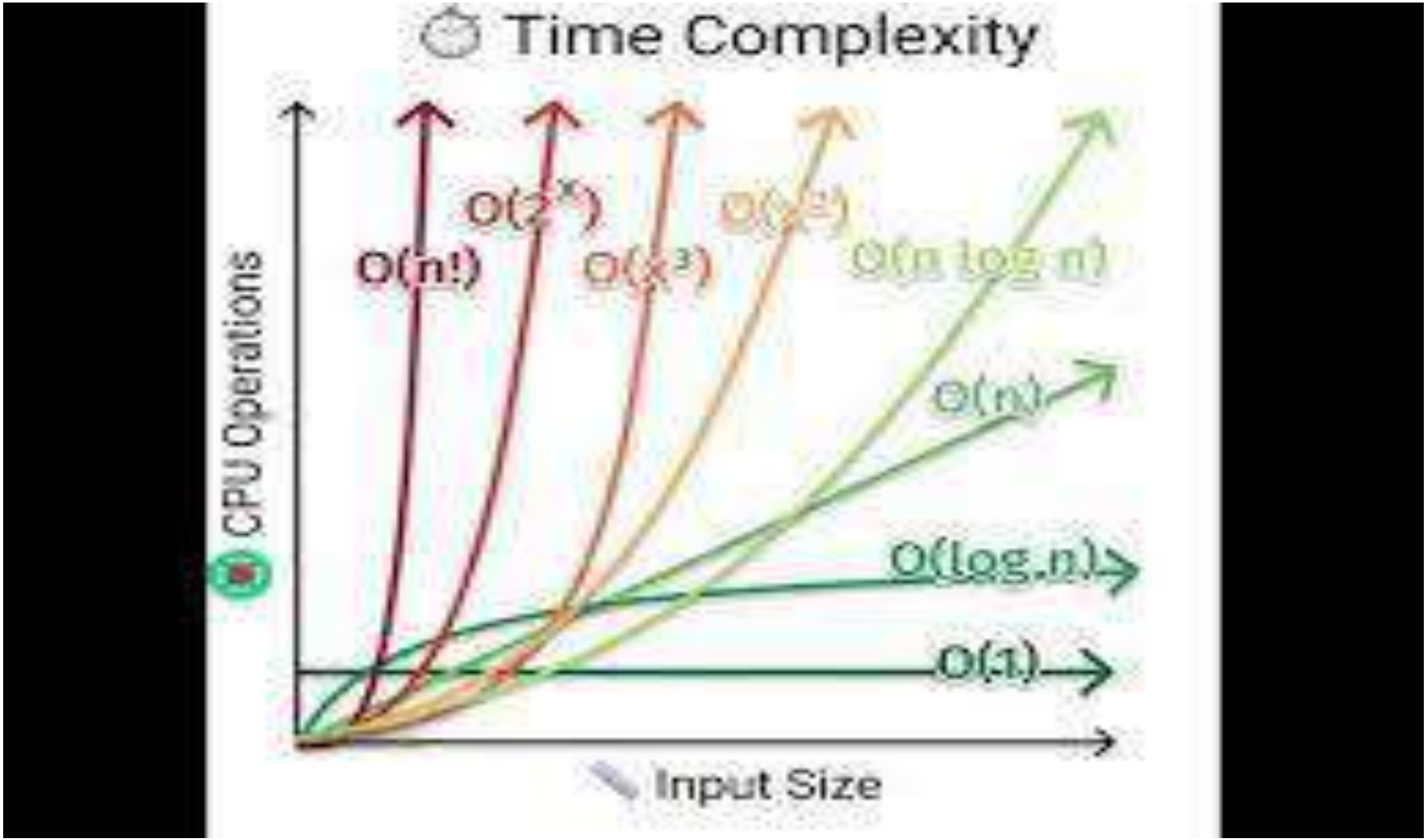

1 2 3 4 5 6 7 8
B 0 3 3
0 1 2 3 4 5
C 1 2 4 5 7 8


1 2 3 4 5 6 7 8
B 0 0 2 2 3 3 3 5

COUNTING-SORT(A, B, k)

```
//let C [0-k] be a new array (Count array)
for i = 0 to k
  C[i]= 0
for j =1 to A.length
  C[A[j]]= C[A[j]]+1
// C[i] now contains the number of elements equal to i.
for i = 1 to k
  C[i]= C[i] + C[i-1]
for j =A.length down to 1
  B[C[A[j ]]]=A[j]
  C[A[j]] = C[A[j]]-1
```

- **O(1): Constant-time complexity**. The running time is independent of the input size, it always takes the same time to compute.
- **O(log n): Logarithmic time complexity.** The running time grows logarithmically in proportion to the input size.
- **O(n): Linear time complexity.** The running time grows linearly with the size of the input.
- **O(n log n): Log-linear or Linearithmic time complexity**. Running time grows linearly and in the order of the logarithm of the size of the input.
- **O(n^2): Quadratic time complexity.** The running time is proportional to the square of the size of the input data.
- **O(n^3): Cubic time complexity.** The running time is proportional to the cube of the size of the input data.
- **O(2^n): Exponential time complexity.** The resources needed for algorithm execution double with each addition to the input data set.
- **O(n!): Factorial time complexity.** The running time grows in proportion to the factorial of the input size. This is typical of algorithms that solve problems by generating all possible combinations.


Time Complexity
CPU Operations
Input Size
O(n!)
O(2^x)
O(x^3)
O(x^2)
O(n log n)
O(n)
O(log n)
O(1)

- big O of log(n)1 < big O √n. When looking at graphs of $\log(n)^2$ and $\log(n)^3$ compared to √n, we can see that for very big values, √n is bigger, meaning for big O that $\log(n)^3 < \sqrt{n}$
- $n \log n$, n^3, 3^n, $n/(\log n)$, $\text{sqrt}(n)$, $\log(\log n)$, $\log n$, $(\log n)^2$, $(\log n)\text{sqrt}(n)$, $n^2 \log n$.
- *Arranged in order:*
  $\log(\log n)$, $\log n$, $(\log n)^2$, $\text{sqrt}(n)$, $(\log n)\text{sqrt}(n)$, $n/(\log n)$, $n \log n$, $n^2 \log n$, n^3, 3^n.
- When comparing "n" and "n/lg n", "n" is always larger than "n/lg n" for any positive value of n greater than 1, as "lg n" (logarithm of n) will always be a smaller number than n, meaning dividing n by a smaller number results in a smaller value overall.