
# DESIGN AND ANALYSIS OF ALGORITHMS (DAA)

## Unit-1

**Dr. G. S. Naveen Kumar,**

**Dean, Quality**

**Associate Professor, CSE**

# DAA Syllabus

## UNIT- I

Introduction : Algorithms, Performance Analysis- Space complexity, Time complexity, Asymptotic notation – Big Oh notation, Theta notation, and Little oh notation.

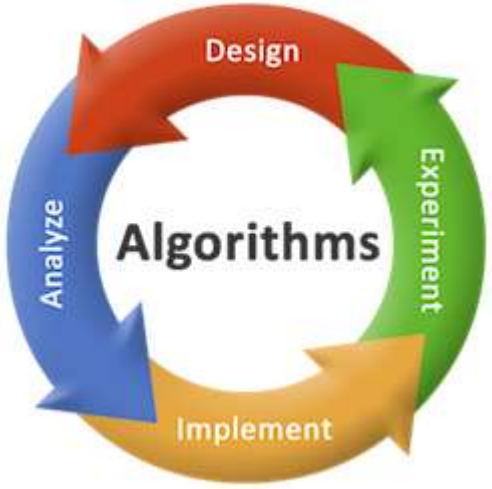
Divide and conquer- General method, applications - Binary search, Merge sort, Quick sort, Strassen’s Matrix Multiplication.

# UNIT- I

## CONTENTS:

- **Introduction** : Algorithms,
- Pseudo code for expressing algorithms,
- Performance Analysis- Space complexity, Time complexity,
- Asymptotic notations – Big Oh notation, Omega notation, Theta notation, and Little oh notation.
- **Divide and conquer-** General method,
- Applications - Binary search, Merge sort, Quick sort, Strassen’s Matrix Multiplication

# What is Algorithm?




A finite set of instruction that specifies a sequence of operation is to be carried out

in order to solve a specific problem or class of problems is called an Algorithm.

# Why study Algorithm?

1. To understand the basic idea of the problem.
2. To find an approach to solve the problem.
3. To improve the efficiency of existing techniques.
4. To understand the basic principles of designing the algorithms.
5. To compare the performance of the algorithm with respect to other techniques.
6. It is the best method of description without describing the implementation detail.
7. The Algorithm gives a clear description of requirements and goal of the problem to the designer.
8. A good design can produce a good solution.
9. To understand the flow of the problem.
10. To measure the  performance of the methods in all cases (best cases, worst cases, average cases)

# DAA Algorithm

- The word algorithm has been derived from the Persian
- Author's name, Abu Ja 'far Mohammed ibn Musa al Khowarizmi (c. 825 A.D.), who has written a textbook on Mathematics.
- The word is taken based on providing a special significance in computer science.
- The algorithm is understood as a method that can be utilized by the computer as when required to provide solutions to a particular problem.

# DAA Algorithm

- An algorithm can be defined as a finite set of steps which has to be followed while carrying out a particular problem.
- It is nothing but a process of executing actions step by step.
- An algorithm is a distinct computational procedure that takes input as a set of values and results in the output as a set of values by solving the problem. .
- An algorithm can be described by incorporating a natural language
  - such as English, Computer language, or a hardware language.

# Algorithm

Definition :

- An Algorithm is a finite sequence of instructions, each of which has a clear meaning
- It can be performed with a finite amount of effort in a finite length of time.
- what the input values may be, an algorithm terminates after executing a finite number of instructions.

# Characteristics of an Algorithm

Algorithm must satisfy the following criteria:

| | |
|---|---|
| Input: | There are zero or more quantities, which are externally supplied; |
| Output: | at least one quantity is produced |
| Definiteness: | each instruction must be clear and unambiguous; |
| Finiteness: | if we trace out the instructions of an algorithm, then for all cases the algorithm will terminate after a finite number of steps; |
| Effectiveness: | every instruction must be sufficiently basic that it can in principle be carried out by a person using only pencil and paper. It is not enough that each operation be definite, but it must also be feasible. |

# Psuedo code for expressing algorithms

Algorithm Specification: Algorithm can be described in three ways.

1. Natural language like English: When this way is chooses care should be taken, we should ensure that each & every statement is definite.

2. Graphic representation called flowchart: This method will work well when the algorithm is small& simple.

3. Pseudo-code Method: In this method, we should typically describe algorithms as program, which resembles language like Pascal & algol

# Pseudo-Code Conventions:

1. Comments begin with // and continue until the end of line.

2. Blocks are indicated with matching braces {and}.

3. An identifier begins with a letter. The data types of variables are not explicitly declared.

4. Compound data types can be formed with records. Here is an example, Node. Record
{
data type – 1 data-1;
.
.
.
data type – n data – n; node * link; }

Here link is a pointer to the record type node. Individual data items of a record can be accessed with ➪ and period.

# Pseudo-Code Conventions:

5. Assignment of values to variables is done using the assignment statement.

   <Variable>:= <expression>;

6. There are two Boolean values TRUE and FALSE.

   → Logical Operators AND, OR, NOT

   → Relational Operators <, <=,>,>=, =, !=

# Pseudo-Code Conventions:

7. The following looping statements are employed.

For, while and repeat-until

**While Loop:**

```
While < condition > do
{
        <statement-1>
                .
                .
                .

        <statement-n>
}
```

**For Loop:**

```
For variable: = value-1 to value-2 step step do

{
  <statement-1>
        .
        .
        .
<statement-n>
```

repeat-until:

repeat

<statement-1>
.
.
.

<statement-n> until<condition>

# Pseudo-Code Conventions:

8. A conditional statement has the following forms.

   → If <condition> then <statement>
   → If <condition> then <statement-1>
   Else <statement-1>

   **Case statement:**

   Case
   {
   : <condition-1>: <statement-1>
   .
   .
   .
   : <condition-n>: <statement-n>
   : else : <statement-n+1>
   }

9. Input and output are done using the instructions read & w

10. There is only one type of procedure:
    Algorithm, the heading takes the form,

*Algorithm <Name> (<Parameter lists>)*

→ As an example, the following algorithm fields & returns the maximum of 'n' given numbers:

```
1.  Algorithm Max(A,n)
2.  // A is an array of size n
3. {
4.  Result := A[1];
5.  for I:= 2 to n do
6.     if A[I] > Result then
7.          Result :=A[I];
8.    return Result;
9. }
```

In this algorithm (named Max), A & n are procedure parameters. Result & I are Local variables.

# Pseudo-Code Conventions:

Algorithm:

```
1.      Algorithm selection sort (a,n)
2.      // Sort the array a[1:n] into non-decreasing order. 3.{
4.      for I:=1 to n do
5.      {
6.      j:=I;

7.      for k:=i+1 to n do
8.      if (a[k]<a[j])
9.      t:=a[I];
10.     a[I]:=a[j];
11.     a[j]:=t;
12.     }
13. }
```

# Analysis of Algorithm Efficiency

## FUNDAMENTALS OF THE ANALYSIS OF ALGORITHM EFFICIENCY

- The efficiency of an algorithm can be in terms of time and space. The algorithm efficiency can be analyzed by the following ways.

1. Analysis Framework.
2. Asymptotic Notations and its properties.
3. Mathematical analysis for Recursive algorithms.
4. Mathematical analysis for Non-recursive algorithms.

# Analysis of Algorithm Efficiency

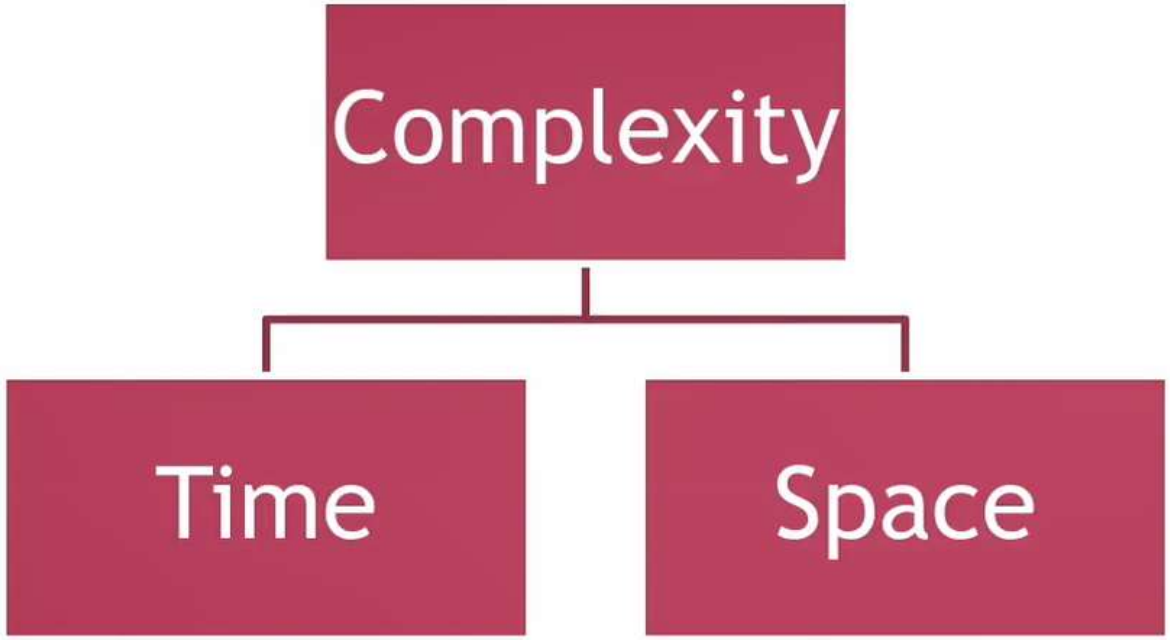
## COMPLEXITY

- The performance of a computer program is the amount of the memory and time needed to run a program.
- We study it in terms of complexity of the algorithm.
- "Complexity of a algorithm is a function which is defined in terms of input size and gives the running time and space required by the algorithm"

# Complexity :

## TYPES OF COMPLEXITY

- Thus, complexity can be divided into two parts:



# Complexity :

- There are two kinds of efficiencies to analyze the efficiency of any algorithm. They are:
  - ***Time efficiency***, indicating how fast the algorithm runs, and
  - ***Space efficiency***, indicating how much extra memory it uses.

# Complexity :

## TIME AND SPACE

- To analyze an algorithm means:
  - Developing a formula for predicting how fast an algorithm is, based on the size of the input (Time Complexity)
  - Developing a formula for predicting how much memory an algorithm requires, based on the size of the input. (Space Complexity).

- Usually time is our biggest concern
- Most of the algorithms require a fixed amount of space.

# Complexity :

## WHAT DOES SIZE OF INPUT MEANS?

- If we are searching an array, the “size” of the input could be the size of the array.
- If we are merging two arrays, the “size” could be the sum of the two array sizes.
- If we are computing the nth fibonacci number, or the nth factorial the “size” is n.

- We choose the “size” as the parameter that influences the actual time/space required.

# Complexity :

## TIME COMPLEXITY

- The time required by an algorithm as the function of its input size of a program is called its time complexity.

- T=T(compilation) + T(Execution)
- T≈ T(Execution)

- E.g. Execution time depends on the number of loops, which depends on input size.

# Complexity :

## SPACE COMPLEXITY

- The space complexity of an algorithm is the amount of space (memory) it needs till completion.
- The space needed by a program has following components:
- Instruction space
- Data Space
- Environment stack

- S = S(fix) + S(variable)

**Instruction Space:** Space required in the computer in order to store instruction is known as Instruction space.

**Environmental Stack:** Amount of space required to store partially executed function.

**Data Space:** Data space is required in order to store variables and constants.

Space Complexity of an algorithm is calculated by using 2 parts.
i) fixed part: variables that have independent charterstics in constants
ii) Variable part: Variables that have dependent characteristics.

S(P)=c+S.P

# Analysis of Algorithm

- There are three types of analysis that we perform on a particular algorithm.

**1.Best Case** In which we analyze the performance of an algorithm for the input, for which the algorithm takes less time or space.

**2.Worst Case** In which we analyse the performance of an algorithm for the input, for which the algorithm takes long time or space.

**3.Average Case** In which we analyze the performance of an algorithm for the input, for which the algorithm takes time or space that lies between best and worst case.

# Analysis of Algorithm

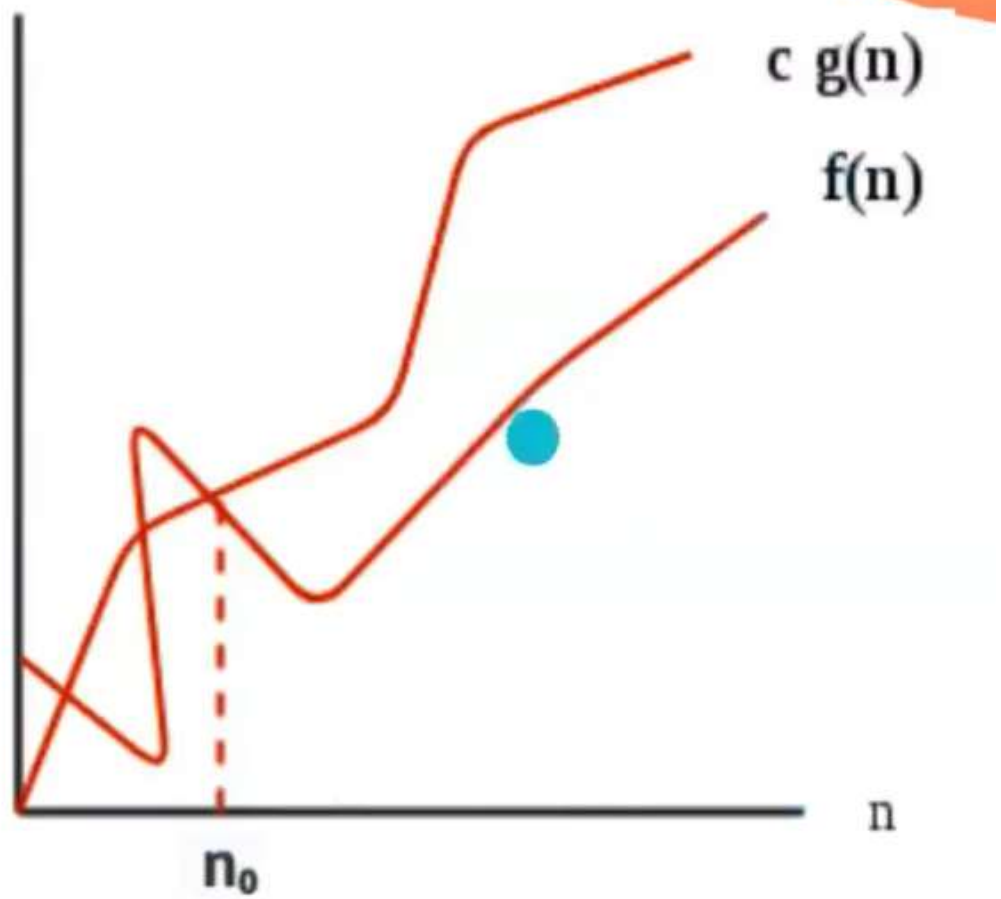
## There are mainly three asymptotic notations:

| | |
|---|---|
| Big-O Notation (O) | (describes the worst-case scenario.) |
| Omega Notation (Ω) | (describes the best-case scenario.) |
| Theta Notation (θ) | (average complexity of an algorithm.) |

# Analysis of Algorithm

## Big-O Notation (O)

- $f(n) = O(g(n))$ , if there exist a positive integer n0 and a positive number c such that $|f(n)| \leq c|g(n)|$ for all $n \geq n0$ . Here $g(n)$ is the upper bound of the function $f(n)$.



# Analysis of Algorithm

## Big-O Notation (O)

- Big O notation specifically describes the worst-case scenario. It represents the upper bound running time complexity of an algorithm.

# Example 1

O(1) Big O notation O(1) represents the complexity of an algorithm that always execute in same time or space regardless of the input data.

```
x = 2
y = 3
z = x + y
print(z)
```

# Example 2

O(n) Big O notation O(N) represents the complexity of an algorithm, whose performance will grow linearly (in direct proportion) to the size of the input data.

```
k = 5
for i in range(5):
    print(i)
```

# Example 3

O(n^2) Big O notation O(n^2) represents the complexity of an algorithm, whose performance is directly proportional to the square of the size of the input data.

```
k = 5
for i in range(5):
    for j in range(5):
        print(i*j)
```

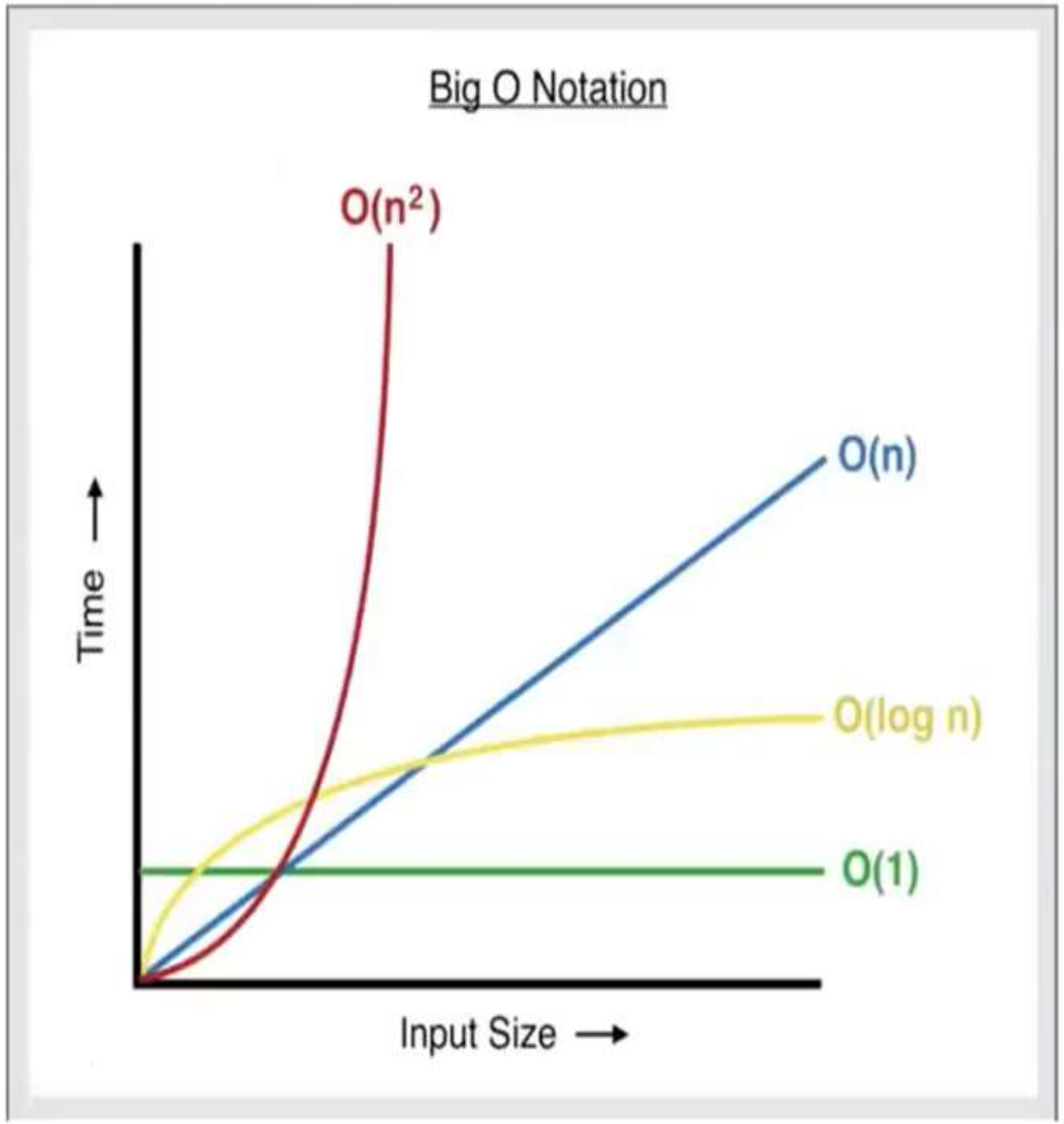



Image Source: How To Calculate Time Complexity With Big O Notation | by Maxwell Harvey Croy | DataSeries | Medium

# Example:

Suppose, $f(n) = 2n + 3$

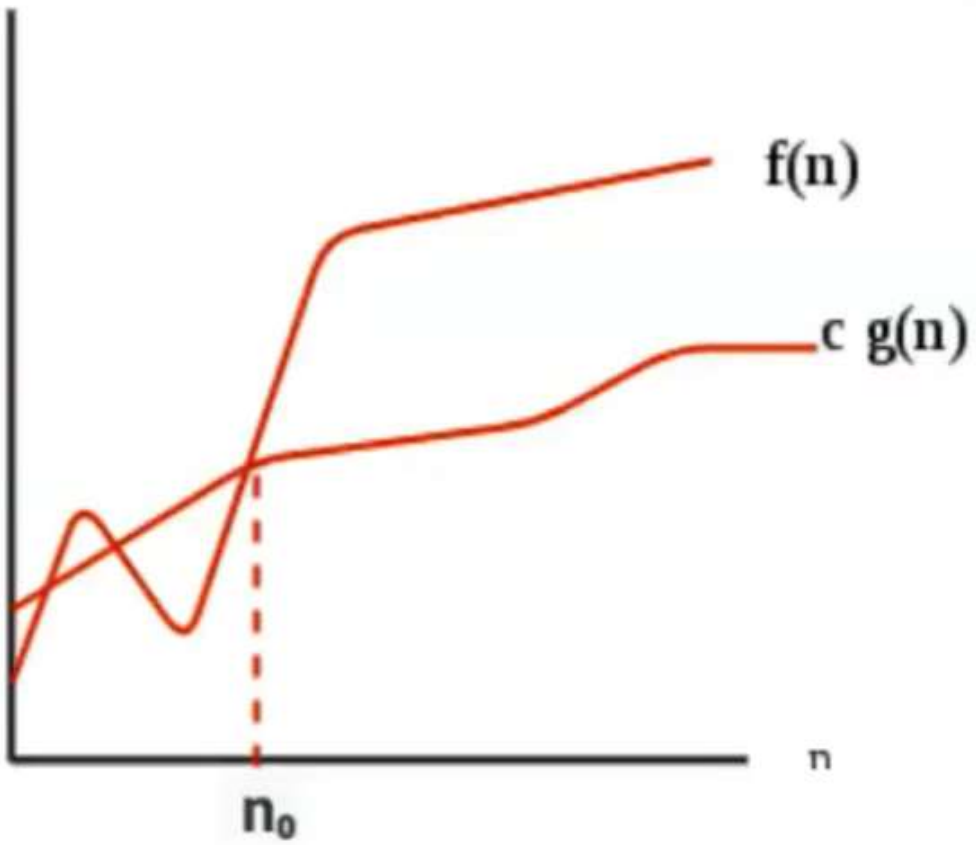
so, $2n+3 <= 2n + 3n$ ; where $n >= 1$

$2n+3 <= 5n$

So, here $f(n)$ is $O(n)$

# Omega Notation (Ω)

- $f(n) = \Omega(g(n))$ , if there exists a positive integer n0 and a positive number c such that $|f(n)| \geq c\,|g(n)|$ for all $n \geq n0$. Here g(n) is the lower bound of the function f(n).



# Omega Notation (Ω)

- Omega notation specifically describes best case scenario.
- It represents the lower bound running time complexity of an algorithm.

## Example:

Suppose, $f(n) = 2n + 3$

so, $2n+3 >= 2n$ ; where $n >= 0$

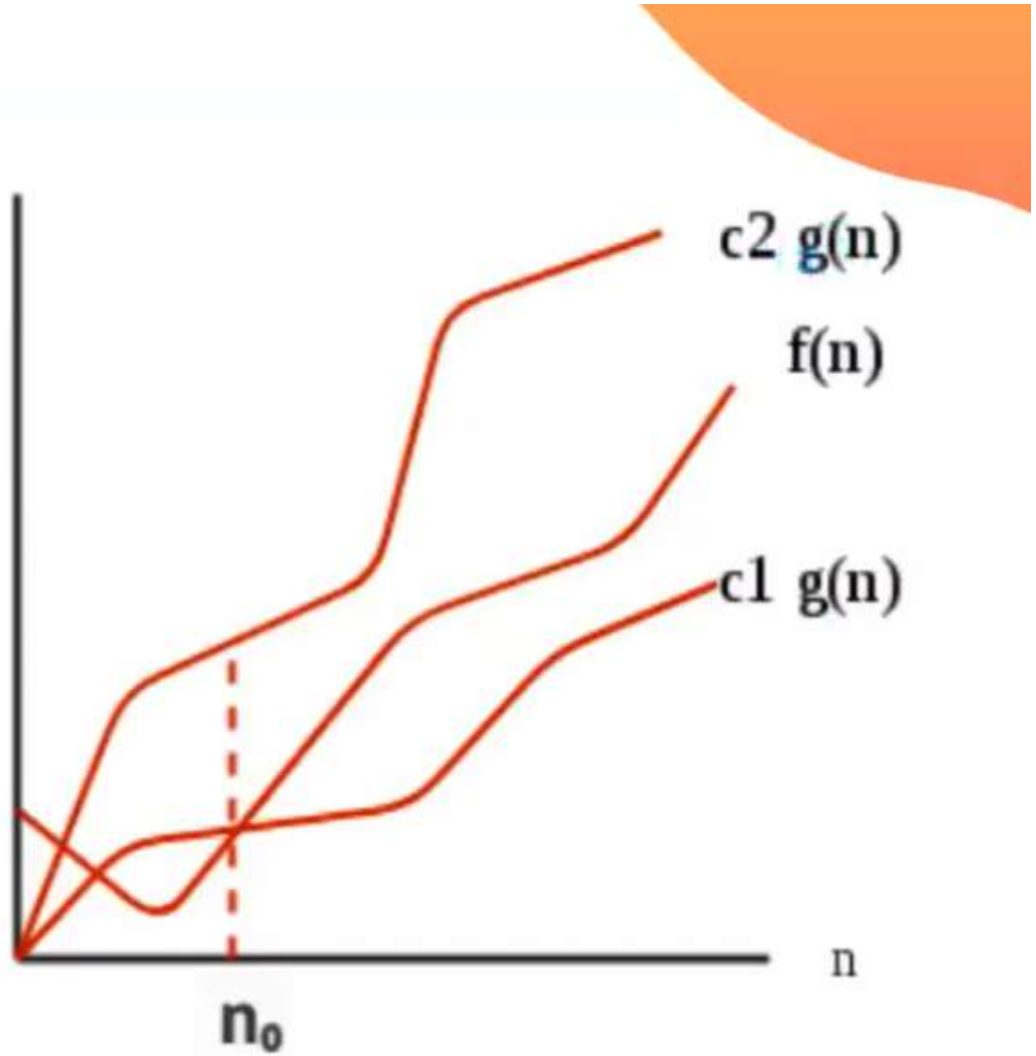
So, here $f(n)$ is $\Omega(n)$

# Theta Notation (θ)

- $f(n) = \Theta(g(n))$, if there exists a positive integer n0 and two positive constants c1 and c2 such that $c1\ |g(n)| \leq |f(n)| \leq c2\ |g(n)|$ for all $n \geq n0$. The function g(n) is both an upper bound and a lower bound for the function f(n) for all values of n, $n \geq n0$ .

## Theta Notation (θ)

This notation describes both the upper bound and lower bound of an algorithm so we can say that it defines exact asymptotic behavior.



# Example:

Suppose, $f(n) = 2n + 3$

So, $2n <= f(n) <= 5n$ ; for all $n >= 1$

So, here $f(n)$ is $\theta(n)$

# Little o Notation:

- we can say that f(n) = o(g(n)) means,

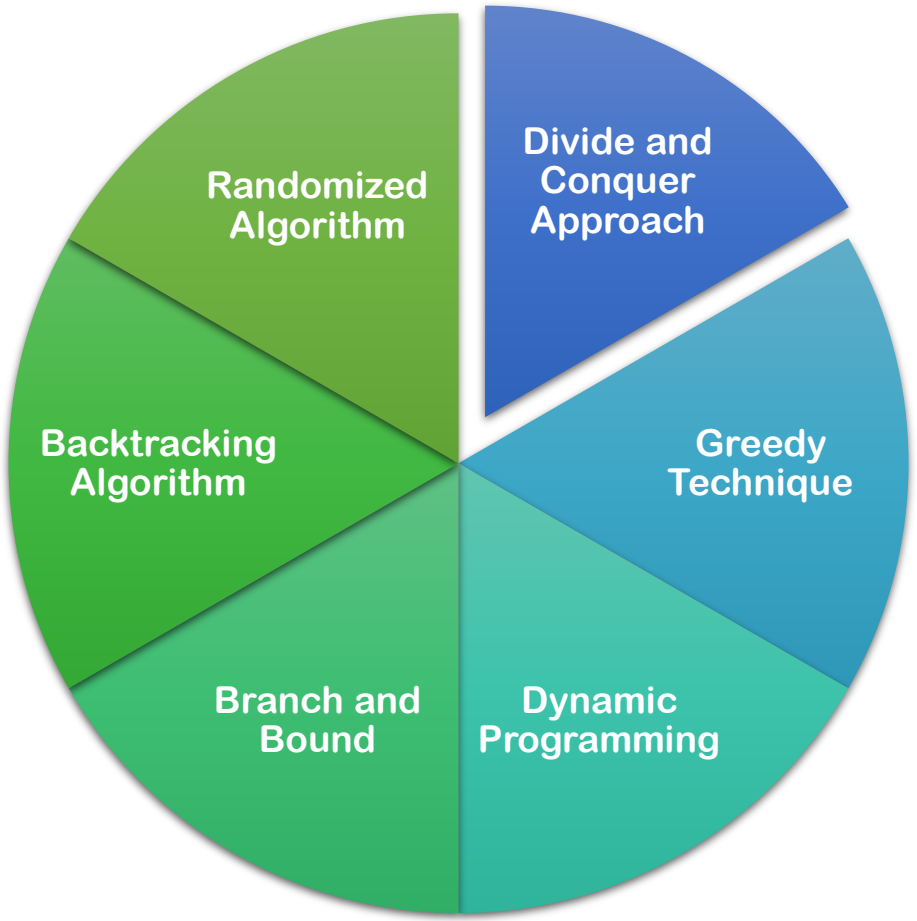
$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$$

- **Example on c notation**
- If $f(n) = n^2$ and $g(n) = n^3$ then check whether f(n) = o(g(n)) or not.

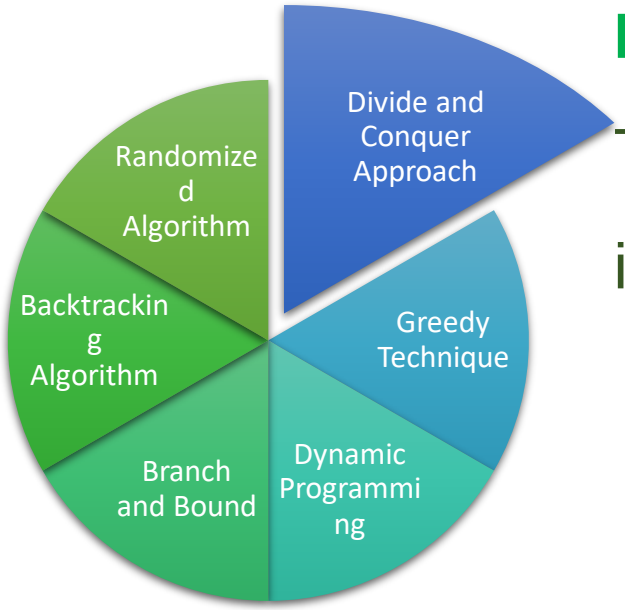
- The result is 0, and it satisfies t ntioned above. So we can say that f(n) = o(g(n)).

$$\lim_{n\to\infty} \frac{n^2}{n^3}$$
$$= \lim_{n\to\infty} \frac{1}{n}$$
$$= \frac{1}{\infty}$$
$$= 0$$

# Algorithm Design Techniques



# Algorithm Design Techniques

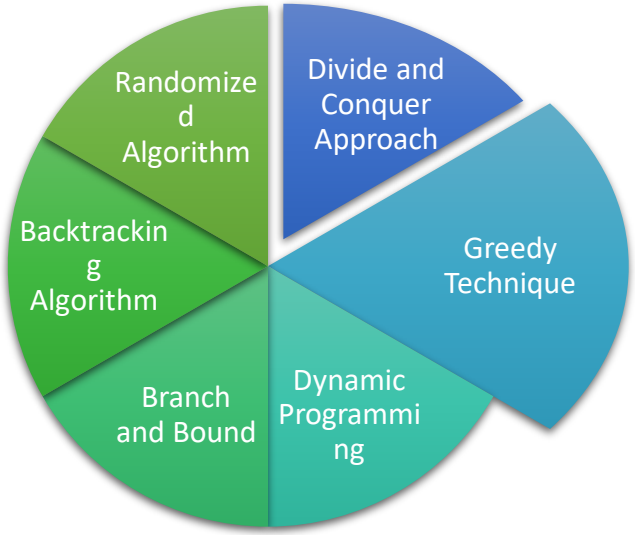



**Divide and Conquer Approach:** It is a top-down approach.

The algorithms which follow the divide & conquer techniques involve three steps:

- Divide the original problem into a set of subproblems.
- Solve every subproblem individually, recursively.
- Combine the solution of the subproblems (top level) into a solution of the whole original problem.

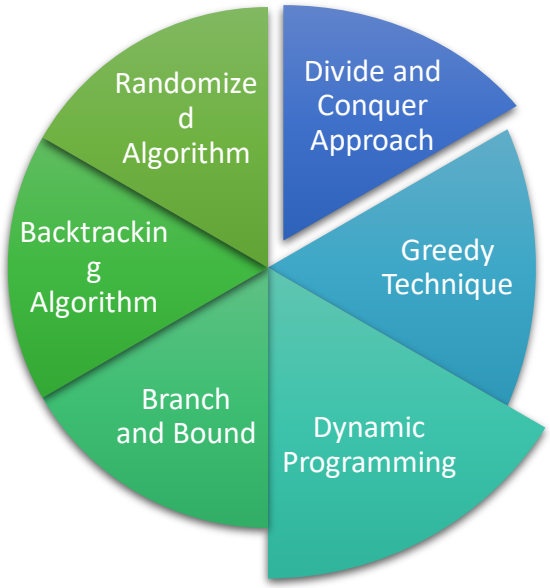
# Algorithm Design Techniques




## Greedy Technique:

- Greedy method is used to solve the optimization problem. (Ex: Find the best route to reach the destination city from the given starting point using a greedy method.)
- An optimization problem is one in which we are given a set of input values, which are required either to be maximized or minimized (known as objective), i.e. some constraints or conditions.
- Greedy Algorithm always makes the choice (greedy criteria) looks best at the moment, to optimize a given objective.
- The greedy algorithm doesn't always guarantee the optimal solution however it generally produces a solution that is very close in value to the optimal.

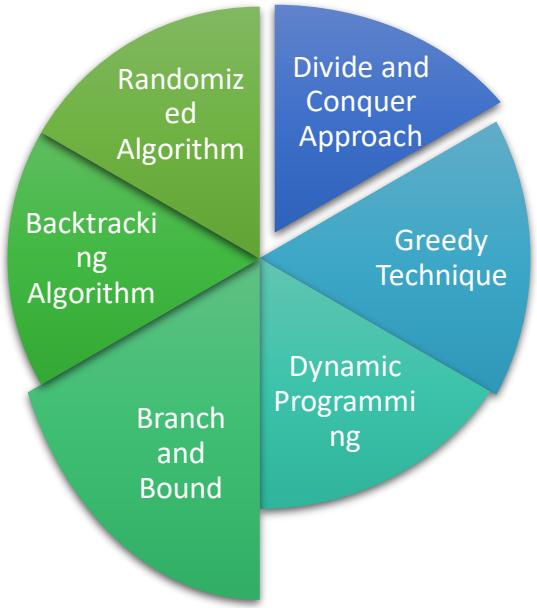
# Algorithm Design Techniques




## Dynamic Programming:

- ✓ Dynamic Programming is a bottom-up approach.
- ✓ Dynamic Programming is also used in optimization problems.
- ✓ Like divide-and-conquer method, Dynamic Programming solves problems by combining the solutions of subproblems. Moreover, Dynamic Programming algorithm solves each sub-problem just once and then saves its answer in a table, thereby avoiding the work of re-computing the answer every time.
- ✓ Two main properties of a problem suggest that the given problem can be solved using Dynamic Programming. These properties are overlapping sub-problems(For example, recursive program of Fibonacci numbers have many overlapping sub-problems.) and optimal substructure.

# Algorithm Design Techniques




## Branch and Bound:

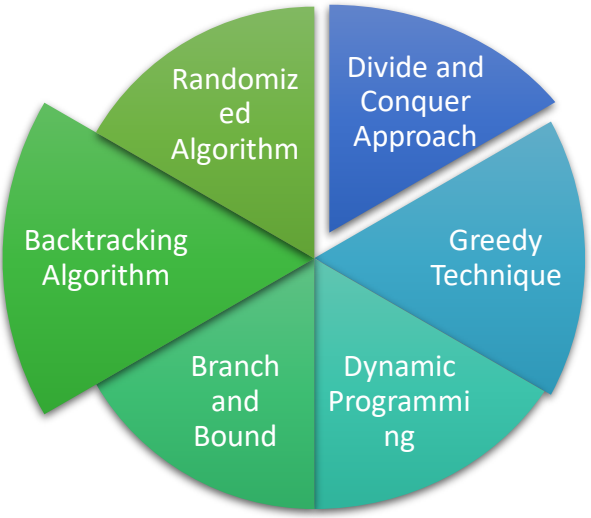
✓A branch and bound algorithm is an optimization technique to get an optimal solution to the problem.

✓It looks for the best solution for a given problem in the entire space of the solution. The bounds in the function to be optimized are merged with the value of the latest best solution. It allows the algorithm to find parts of the solution space completely.

✓The purpose of a branch and bound search is to maintain the lowest-cost path to a target. Once a solution is found, it can keep improving the solution. Branch and bound search is implemented in depth-bounded search and depth–first search

.

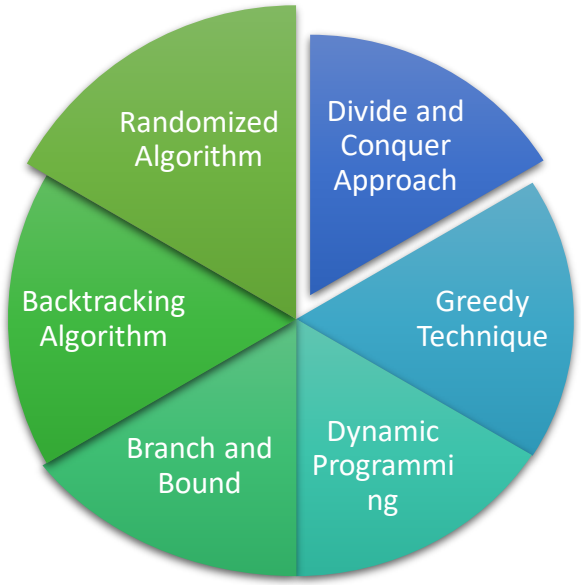
# Algorithm Design Techniques




## Backtracking Algorithm :

- Backtracking Algorithm tries each possibility until they find the right one.(Eight queen problem, Sudoku puzzle and going through a maze are popular examples where backtracking algorithm is used.)
- It is a depth-first search of the set of possible solution.
- During the search, if an alternative doesn't work, then backtrack to the choice point
- The place which presented different alternatives, and tries the next alternative.

# Algorithm Design Techniques




## Randomized Algorithms:

- An algorithm that uses random numbers to decide what to do next anywhere in its logic is called Randomized Algorithm.

For example, in Randomized Quick Sort, we use random number to pick the next pivot (or we randomly shuffle the array). Typically, this randomness is used to reduce time complexity or space complexity in other standard algorithms.

# Divide and Conquer

Divide and Conquer is an algorithmic pattern.

In algorithmic methods,

- the design is to take a dispute on a huge input,
- break the input into minor pieces,
- decide the problem on each of the small pieces,
- merge the piecewise solutions into a global solution.

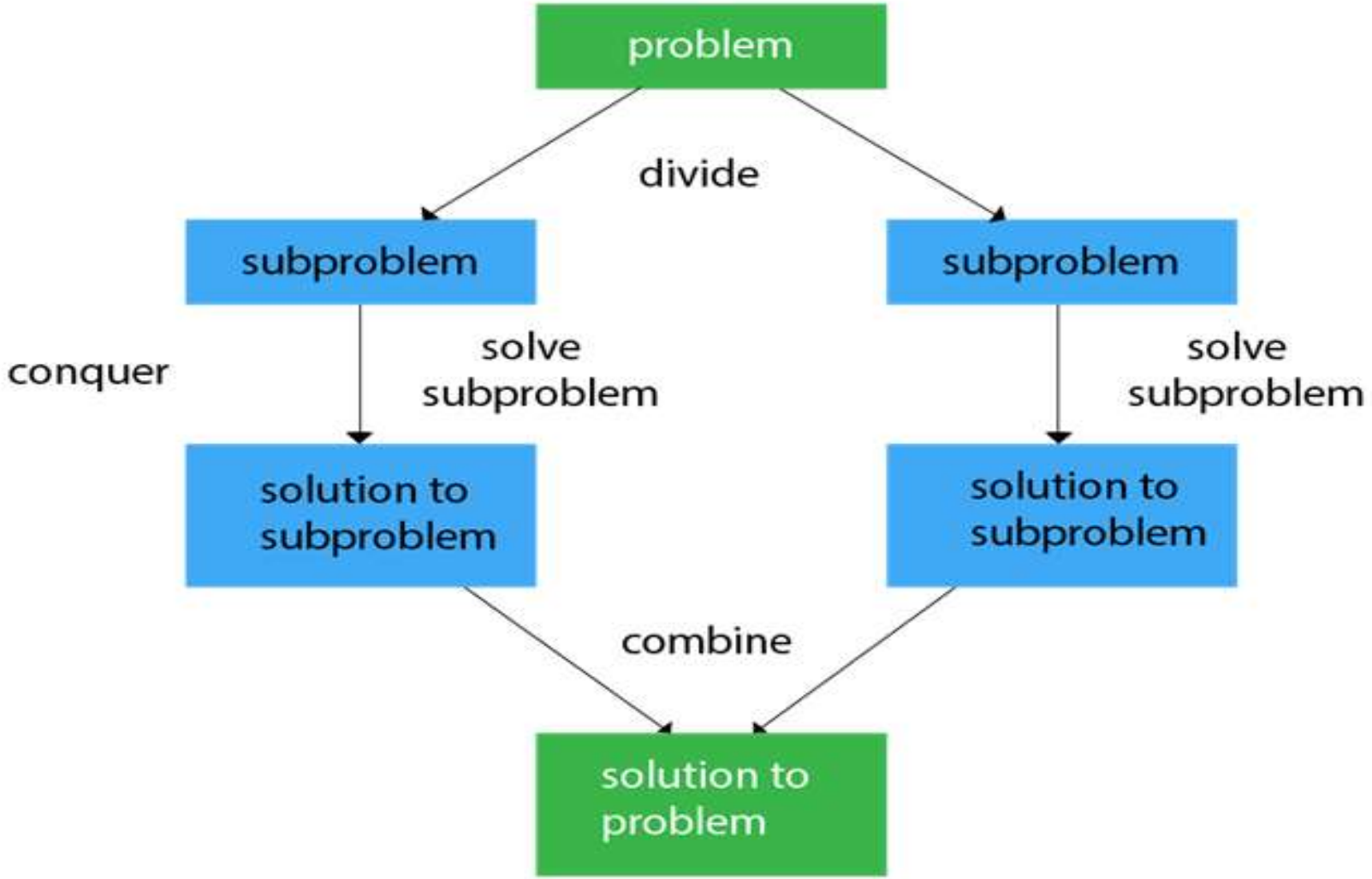
This mechanism of solving the problem is called the Divide & Conquer Strategy.

# Divide and Conquer

Divide and Conquer algorithm consists of a dispute using the following three steps.

- Divide the original problem into a set of subproblems.
- Conquer: Solve every subproblem individually, recursively.
- Combine: Put together the solutions of the subproblems to get the solution to the whole problem.

# Divide and Conquer



# Divide and Conquer

**Examples:** The specific computer algorithms are based on the Divide & Conquer approach:

1. Maximum and Minimum Problem
2. Binary Search
3. Sorting (merge sort, quick sort)
4. Tower of Hanoi.

# Fundamental of Divide and Conquer Strategy:

There are two fundamental of Divide & Conquer Strategy:

1. **Relational Formula**
2. **Stopping Condition**

**1. Relational Formula:** It is the formula that we generate from the given technique. After generation of Formula we apply D&C Strategy, i.e. we break the problem recursively & solve the broken subproblems.

**2. Stopping Condition:** When we break the problem using Divide & Conquer Strategy, then we need to know that for how much time, we need to apply divide & Conquer. So the condition where the need to stop our recursion steps of D&C is called as Stopping Condition.

# Applications of Divide and Conquer Approach:

1. **Binary Search: .**
2. **Quick sort:.**
3. **Merge Sort:**
4. **Closest Pair of Points:**
5. **Strassen's Matrix multiplication Algorithm:**

# Advantages of Divide and Conquer:

1. Divide and Conquer tend to successfully solve one of the biggest problems, such as the Tower of Hanoi, a mathematical puzzle.
2. This algorithm is much faster than other algorithms.
3. It efficiently uses cache memory without occupying much space because it solves simple subproblems within the cache memory instead of accessing the slower main memory.
4. Divide and conquer approach supports parallelism as sub-problems are independent. Hence, an algorithm, which is designed using this technique, can run on the multiprocessor system or in different machines simultaneously.

# Disadvantages of Divide and Conquer:

1. Since most of its algorithms are designed by incorporating recursion, so it necessitates high memory management.
2. An explicit stack may overuse the space.
3. It may even crash the system if the recursion is performed rigorously greater than the stack present in the CPU.

# Binary Search

- In Binary Search technique, we search an element in a sorted array by recursively dividing the interval in half.
- Firstly, we take the whole array as an interval.
- If the key Element (the item to be searched) is less than the item in the middle of the interval, We discard the second half of the list and recursively repeat the process for the first half of the list by calculating the new middle and last element.
- If the key Element (the item to be searched) is greater than the item in the middle of the interval,
- We discard the first half of the list and work recursively on the second half by calculating the new beginning and middle element.
- Repeatedly, check until the value is found or interval is empty.

# Analysis:

**1.Input:** an array A of size n, already sorted in the ascending or descending order.

**2.Output:** analyze to search an element item in the sorted array of size n.

**3.Logic:** Let T (n) = number of comparisons of an item with n elements in a sorted array.

- Set BEG = 1 and END = n
- Find mid = $\text{int}\left(\frac{beg + end}{2}\right)$
- Compare the search item with the mid item.

**Case 1:** item = A[mid], then LOC = mid, but it the best case and T (n) = 1

**Case 2:** item ≠A [mid], then we will split the array into two equal parts of size . $\frac{n}{2}$

And again find the midpoint of the half-sorted array and compare with search element.

Repeat the same process until a search element is found.

# Analysis:

$T(n) = T\left(\frac{n}{2}\right) + 1$ ..... (Equation 1)

{Time to compare the search element with mid element, then with half of the selected half part of array}

$T\left(\frac{n}{2}\right) = T\left(\frac{n}{2^2}\right) + 1$, putting $\frac{n}{2}$ in place of n.

Then we get: $T(n) = \left(T\left(\frac{n}{2^2}\right) + 1\right) + 1$.....By putting $T\frac{n}{2}$ in (1) equation

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

$T\left(\frac{n}{2^2}\right) = T\left(\frac{n}{2^3}\right) + 1$..... Putting $\frac{n}{2}$ in place of n in eq 1.

$T(n) = T\left(\frac{n}{2^3}\right) + 1 + 2$

$T(n) = T\left(\frac{n}{2^3}\right) + 3$..... (Equation 3)

$T\left(\frac{n}{2^3}\right) = T\left(\frac{n}{2^4}\right) + 1$..... Putting $\frac{n}{3}$ in place of n in eq1

Put $T\left(\frac{n}{2^3}\right)$ in eq (3)

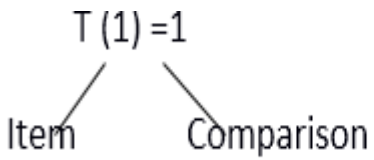
$T(n) = T\left(\frac{n}{2^4}\right) + 4$

Repeat the same process ith times

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

**Stopping Condition:** $T(1) = 1$

# Analysis:




**Is the last term of the equation and it will be equal to 1**

$$\frac{n}{2^i} = 1$$

$\frac{n}{2^i}$ **Is the last term of the equation and it will be equal to 1**

$$n = 2^i$$

Applying log both sides

$$\log n = \log_2 i$$

$$\text{Log } n = i \log 2$$

$$\frac{\log n}{\log 2} = 1$$

$$\log_2 n = i$$

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

$\frac{n}{2^i}$ **= 1 as in eq 5**

$$= T(1) + i$$

$$= 1 + i \quad \text{..... } T(1) = 1 \text{ by stopping condition}$$

$$= 1 + \log_2 n$$

$$= \log_2 n \quad \text{..... (1 is a constant that's why ignore it)}$$

**Therefore, binary search is of order o ($\log_2 n$)**

# Merge Sort algorithm

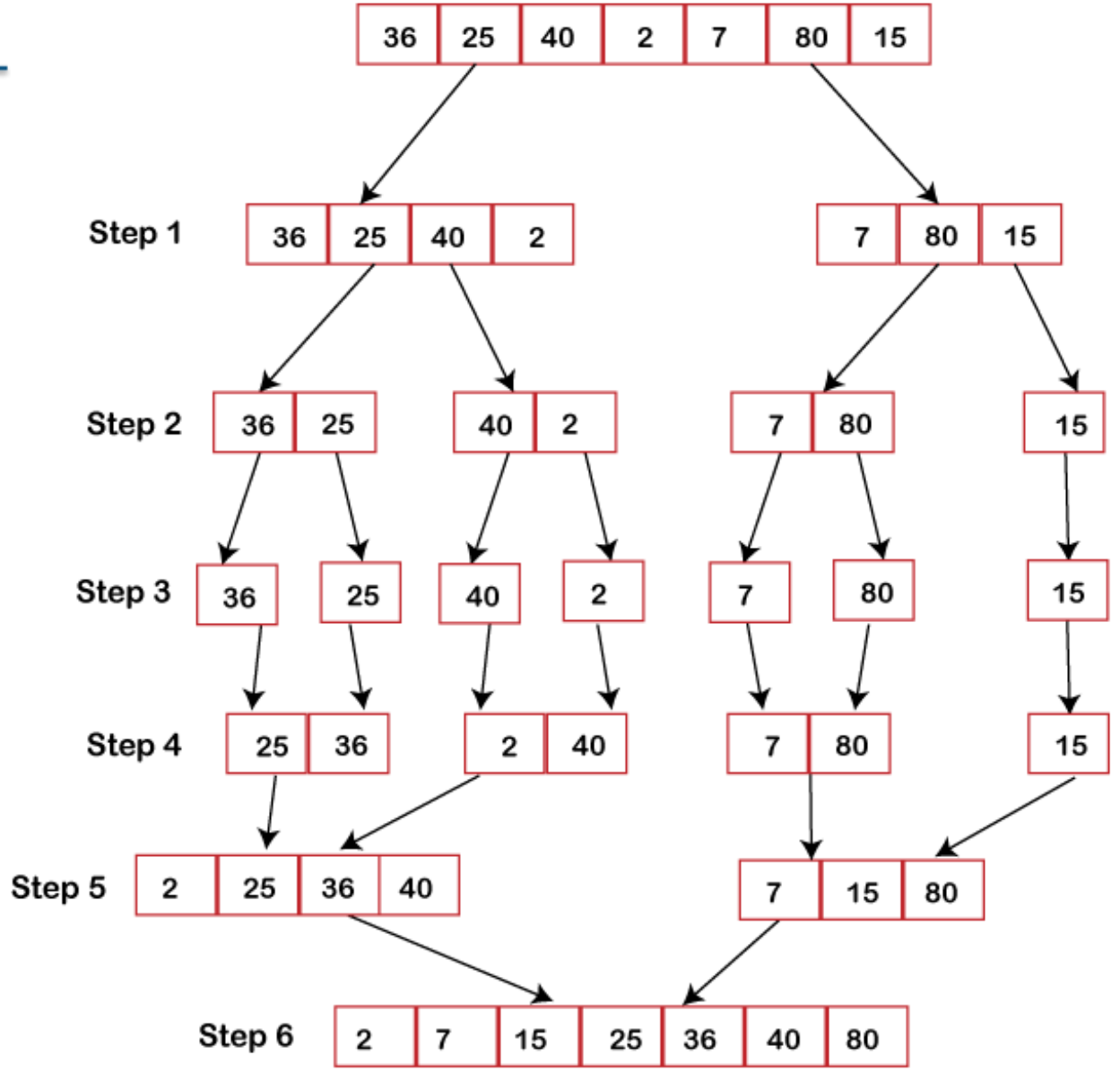
- The MergeSort function keeps on splitting an array into two halves until a condition is met where we try to perform MergeSort on a subarray of size 1, i.e., **p == r**.
- It combines the individually sorted subarrays into larger arrays until the whole array is merged.

ALGORITHM-MERGE SORT

1. If p**<r**
2. Then q → ( p+ r)/2
3. MERGE-SORT (A, p, q)
4. MERGE-SORT ( A, q+1,r)
5. MERGE ( A, p, q, r)

Here we called **MergeSort(A, 0, length(A)-1)** to sort the complete array.

# Merge Sort


36 25 40 2 7 80 15
Step 1
36 25 40 2
7 80 15
Step 2
36 25
40 2
7 80
15
Step 3
36
25
40
2
7
80
15
Step 4
25 36
2 40
7 80
15
Step 5
2 25 36 40
7 15 80
Step 6
2 7 15 25 36 40 80

# Analysis of Merge Sort:

Let T (n) be the total time taken by the Merge Sort algorithm.

- Sorting two halves will take at the most $2T\frac{n}{2}$ time.
- When we merge the sorted lists, we come up with a total n-1 comparison because the last element which is left will need to be copied down in the combined list, and there will be no comparison.

Thus, the relational formula will be

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

But we ignore '-1' because the element will take some time to be copied in merge lists.

So $T(n) = 2T\left(\frac{n}{2}\right) + n$...equation 1

Putting $n=\frac{n}{2}$ in place of n in .....equation 1

$T\left(\frac{n}{2}\right) = 2T\left(\frac{n}{2^2}\right) + \frac{n}{2}$.....equation2

Put 2 equation in 1 equation

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

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

$T(n) = 2^2\,T\left(\frac{n}{2^2}\right)+2n$.....equation 3

Putting $n=\frac{n}{2^2}$ in equation 1

$T\left(\frac{n}{2^3}\right) = 2T\left(\frac{n}{2^3}\right) + \frac{n}{2^2}$.....equation4

Putting 4 equation in 3 equation

$T(n) = 2^2\left[2T\left(\frac{n}{2^3}\right) + \frac{n}{2^2}\right] + 2n$

$T(n) = 2^3\,T\left(\frac{n}{2^3}\right) + n + 2n$

$T(n) = 2^3\,T\left(\frac{n}{2^3}\right) + 3n$.....equation5

From eq 1, eq3, eq 5.....we get

$T(n) = 2^i\,T\left(\frac{n}{2^i}\right) + in$.....equation6

From Stopping Condition:

$\frac{n}{2^i} = 1$ And $T\left(\frac{n}{2^i}\right)=0$

$n = 2^i$

Apply log both sides:

$\log n=\log_2 i$

$\log n= i \log 2$

$\frac{\log n}{\log 2} = i$

$\log_2 n=i$

From 6 equation

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

$= 2^i \times 0 + \log_2 n \cdot n$

**$= T(n) = n.\log n$**

# Analysis of Merge Sort:

**Best Case Complexity:** The merge sort algorithm has a best-case time complexity of **O(n*log n)** for the already sorted array.

**Average Case Complexity:** The average-case time complexity for the merge sort algorithm is **O(n*log n)**, which happens when 2 or more elements are jumbled, i.e., neither in the ascending order nor in the descending order.

**Worst Case Complexity:** The worst-case time complexity is also **O(n*log n)**, which occurs when we sort the descending order of an array into the ascending order.

**Space Complexity:** The space complexity of merge sort is **O(n)**.

## Merge Sort Applications

The concept of merge sort is applicable in the following areas:

- Inversion count problem
- External sorting
- E-commerce applications

# Quick Sort Algorithm

- Quicksort is the widely used sorting algorithm
- It is makes **n log n** comparisons in average case for sorting an array of n elements.
- It is a faster and highly efficient sorting algorithm.
- This algorithm follows the divide and conquer approach.
- Divide and conquer is a technique of breaking down the algorithms into subproblems, then solving the subproblems,
- It it combining the results back together to solve the original problem.

# Quick Sort Algorithm

**Divide:** In Divide, first pick a pivot element.

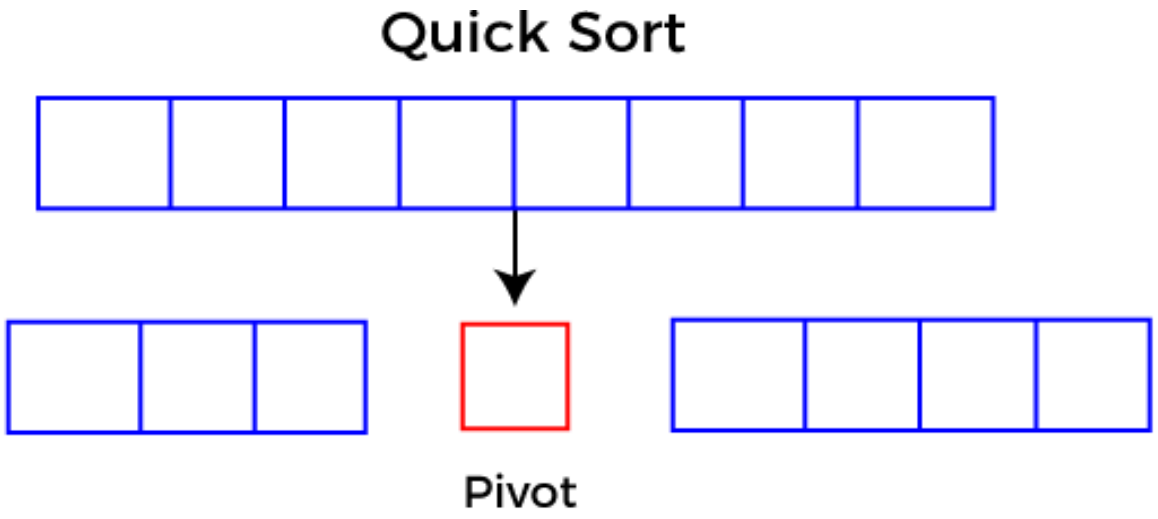
After that, partition or rearrange the array into two sub-arrays

- such that each element in the left sub-array is less than or equal to the pivot element
- In each element in the right sub-array is larger than the pivot element.

**Conquer:** Recursively, sort two subarrays with Quicksort.

**Combine:** Combine the already sorted array.

# Quick Sort Algorithm




## Choosing the pivot

Picking a good pivot is necessary for the fast implementation of quicksort. However, it is typical to determine a good pivot. Some of the ways of choosing a pivot are as follows -

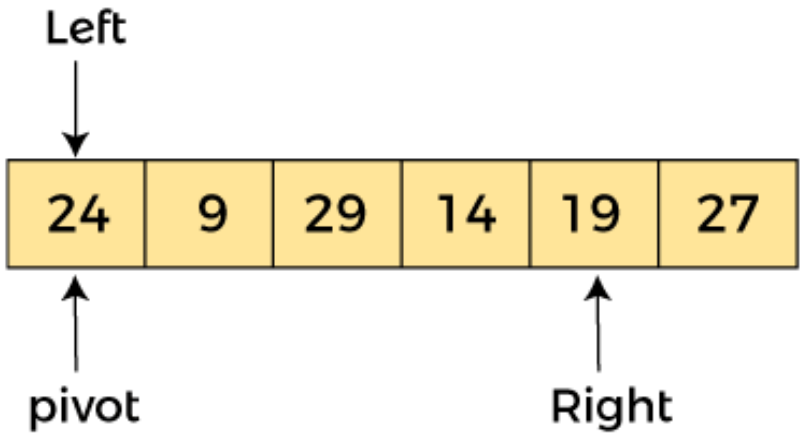
- Pivot can be random, i.e. select the random pivot from the given array.
- Pivot can either be the rightmost element of the leftmost element of the given array.
- Select median as the pivot element.

# Quick Sort Algorithm

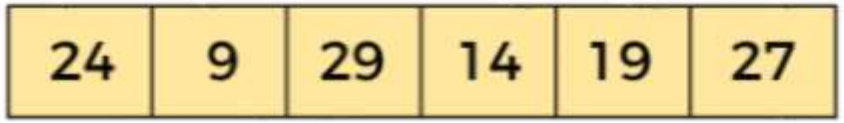
## Algorithm

**Algorithm:**

```
QUICKSORT (array A, start, end)
{
 1 if (start < end)
 2 {
3 p = partition(A, start, end)
4 QUICKSORT (A, start, p - 1)
5 QUICKSORT (A, p + 1, end)
6 }
}
```

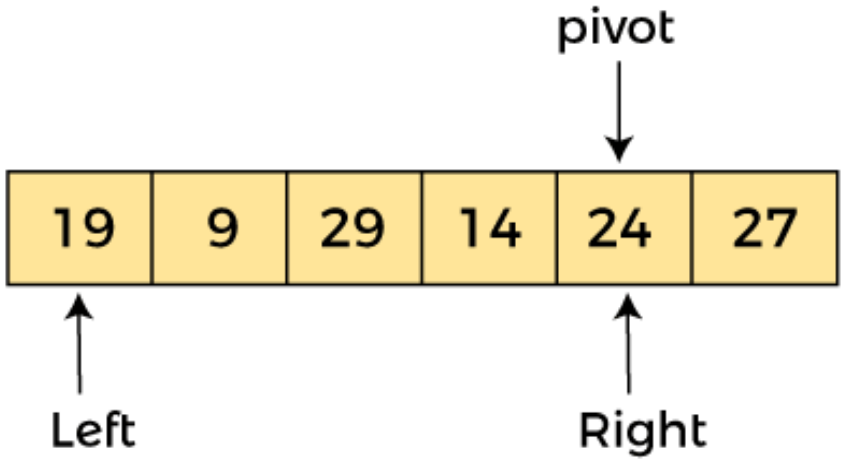



Let the elements of array are -



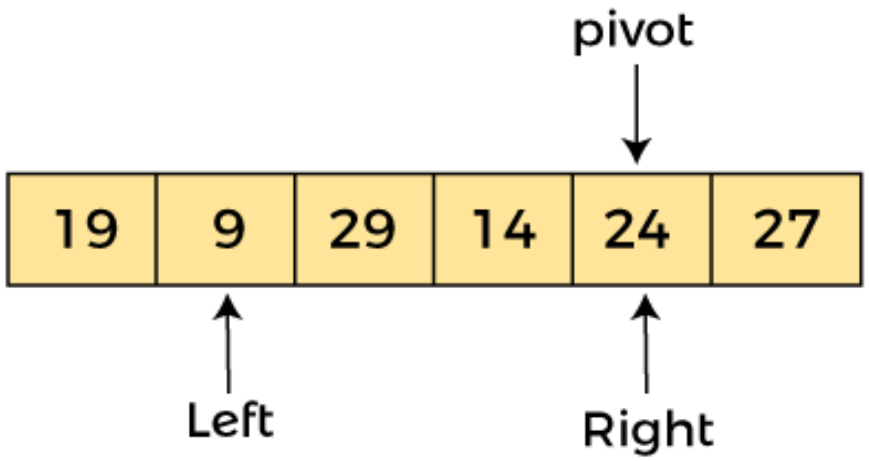

Now, a[left] = 24, a[right] = 19, and a[pivot] = 24.

Because, a[pivot] > a[right], so, algorithm will swap a[pivot] with a[right], and pivot moves to right, as -

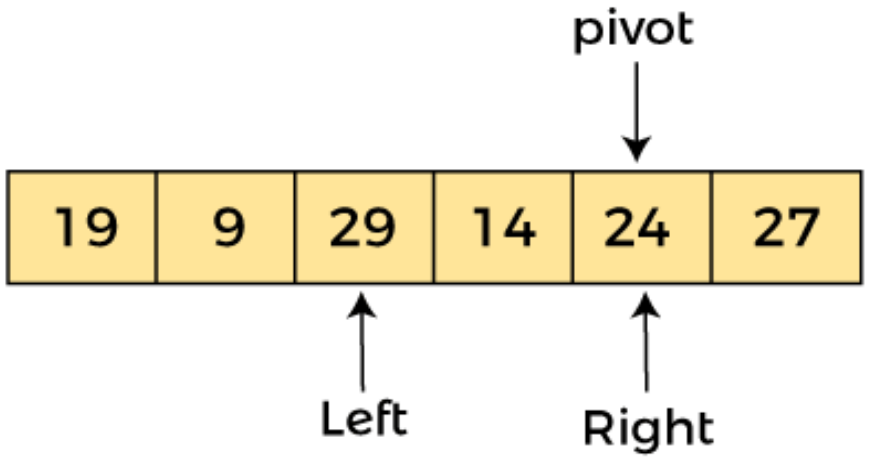



Now, a[left] = 19, a[right] = 24, and a[pivot] = 24. Since, pivot is at right, so algorithm starts from left and moves to right.

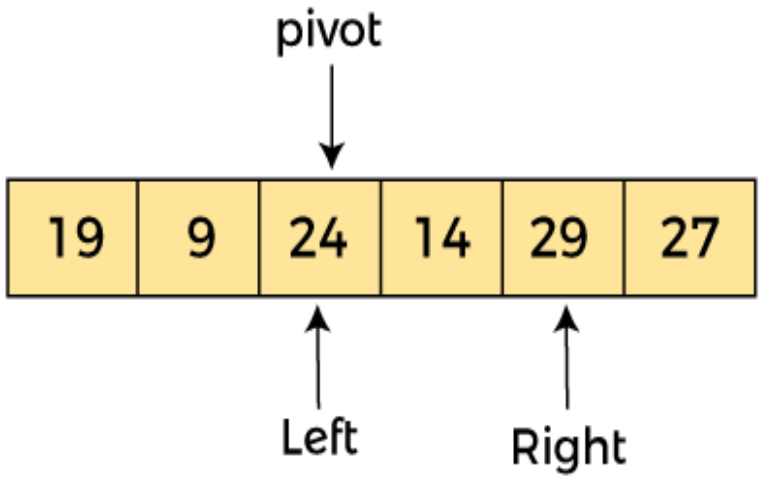
As a[pivot] > a[left], so algorithm moves one position to right as -



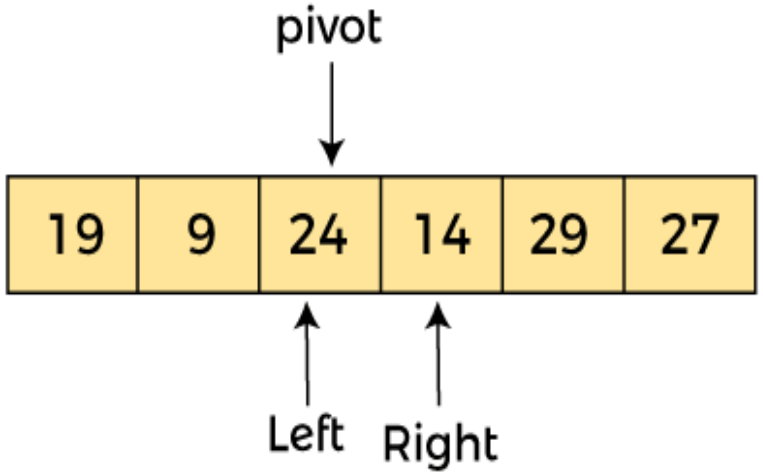

Now, a[left] = 9, a[right] = 24, and a[pivot] = 24. As a[pivot] > a[left], so algorithm moves one position to right as -



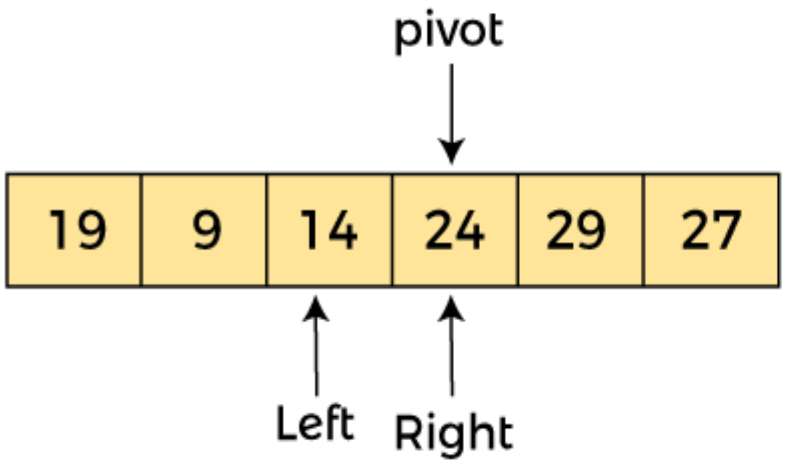

Now, a[left] = 29, a[right] = 24, and a[pivot] = 24. As a[pivot] < a[left], so, swap a[pivot] and a[left], now pivot is at left, i.e. -



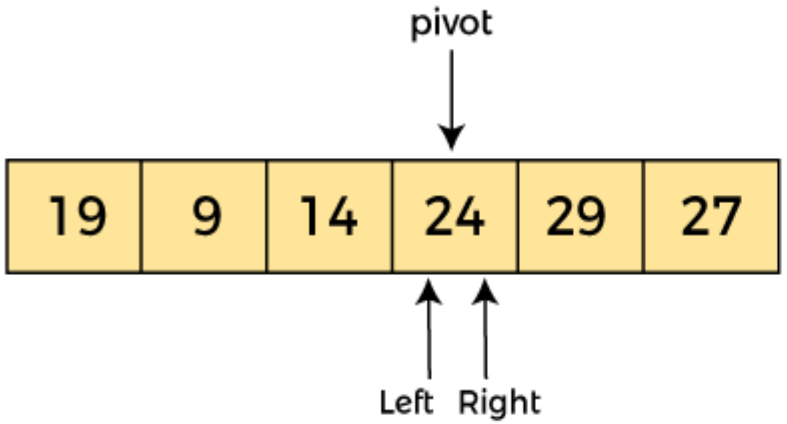

Since, pivot is at left, so algorithm starts from right, and move to left. Now, a[left] = 24, a[right] = 29, and a[pivot] = 24. As a[pivot] < a[right], so algorithm moves one position to left, as -




Now, a[pivot] = 24, a[left] = 24, and a[right] = 14. As a[pivot] > a[right], so, swap a[pivot] and a[right], now pivot is at right, i.e. -



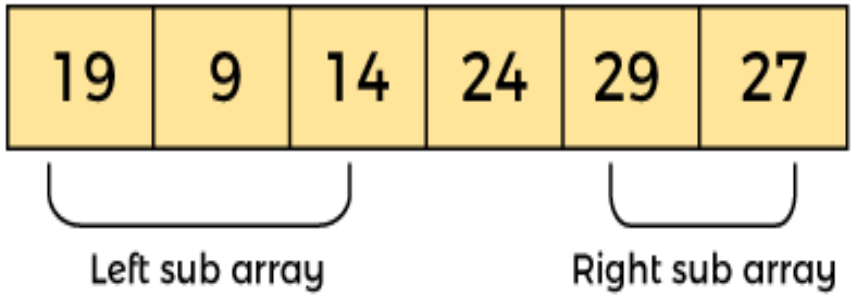

Now, a[pivot] = 24, a[left] = 14, and a[right] = 24. Pivot is at right, so the algorithm starts from left and move to right.



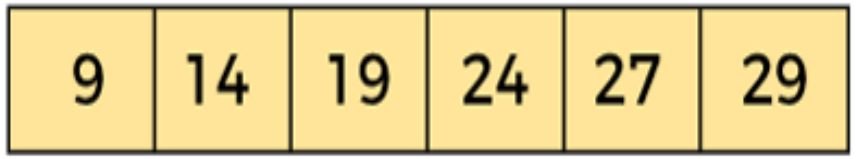

Now, a[pivot] = 24, a[left] = 24, and a[right] = 24. So, pivot, left and right are pointing the same element. It represents the termination of procedure.

Element 24, which is the pivot element is placed at its exact position.

# Quick Sort Algorithm




Now, in a similar manner, quick sort algorithm is separately applied to the left and right sub-arrays. After sorting gets done, the array will be -



# Quicksort complexity

Now, let's see the time complexity of quicksort in best case, average case, and in worst case. We will also see the space complexity of quicksort.

## 1. Time Complexity

| Case | Time Complexity |
|---|---|
| **Best Case** | O(n*logn) |
| **Average Case** | O(n*logn) |
| **Worst Case** | $O(n^2)$ |

- **Best Case Complexity -** In Quicksort, the best-case occurs when the pivot element is the middle element or near to the middle element. The best-case time complexity of quicksort is **O(n*logn)**.
- **Average Case Complexity -** It occurs when the array elements are in jumbled order that is not properly ascending and not properly descending. The average case time complexity of quicksort is **O(n*logn)**.
- **Worst Case Complexity -** In quick sort, worst case occurs when the pivot element is either greatest or smallest element. Suppose, if the pivot element is always the last element of the array, the worst case would occur when the given array is sorted already in ascending or descending order. The worst-case time complexity of quicksort is **$O(n^2)$**.

# Quick Sort Algorithm

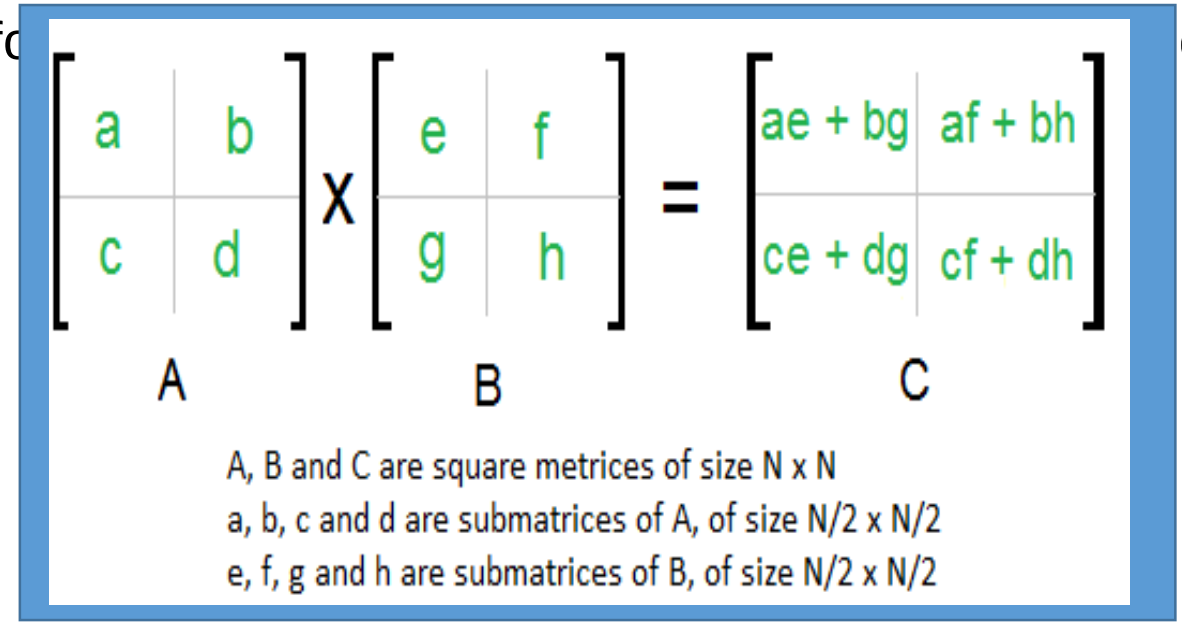
## 2. Space Complexity

| | |
|---|---|
| **Space Complexity** | O(n*logn) |
| **Stable** | NO |

- The space complexity of quicksort is O(n*logn).

# Strassen's Matrix Multiplication Algorithm

- ***Divide and Conquer***
  Following is simple Divide and Conquer method to multiply two square matrices.
  1) Divide matrices A and B in 4 sub-matrices of size N/2 x N/2 as shown in the below diagram.
  2) Calculate fo dg and cf + dh.



# Strassen's Matrix Multiplication Algorithm

## Strassen's Matrix Multiplication Algorithm

In this context, using Strassen's Matrix multiplication algorithm, the time consumption can be improved a little bit.

Strassen's Matrix multiplication can be performed only on **square matrices** where **n** is a **power of 2**. Order of both of the matrices are **n × n**.

Divide **X**, **Y** and **Z** into four (n/2)×(n/2) matrices as represented below –

$$Z = \begin{bmatrix} I & J \\ K & L \end{bmatrix} \qquad X = \begin{bmatrix} A & B \\ C & D \end{bmatrix} \text{ and } \qquad Y = \begin{bmatrix} E & F \\ G & H \end{bmatrix}$$

# Strassen's Matrix Multiplication Algorithm

Using Strassen's Algorithm compute the following –

$$M_1 := (A + C) \times (E + F)$$

$$M_2 := (B + D) \times (G + H)$$

$$M_3 := (A - D) \times (E + H)$$

$$M_4 := A \times (F - H)$$

$$M_5 := (C + D) \times (E)$$

$$M_6 := (A + B) \times (H)$$

$$M_7 := D \times (G - E)$$

Then,

$$I := M_2 + M_3 - M_6 - M_7$$

$$J := M_4 + M_6$$

$$K := M_5 + M_7$$

$$L := M_1 - M_3 - M_4 - M_5$$

Analysis

$$T(n) = \begin{cases} c & if\ n = 1 \\ 7\ x\ T(\frac{n}{2}) + d\ x\ n^2 & otherwise \end{cases}$$ where *c* and *d* are constants

Using this recurrence relation, we get $T(n) = O(n^{log7})$

Hence, the complexity of Strassen's matrix multiplication algorithm is $O(n^{log7})$ .


Thank
you!