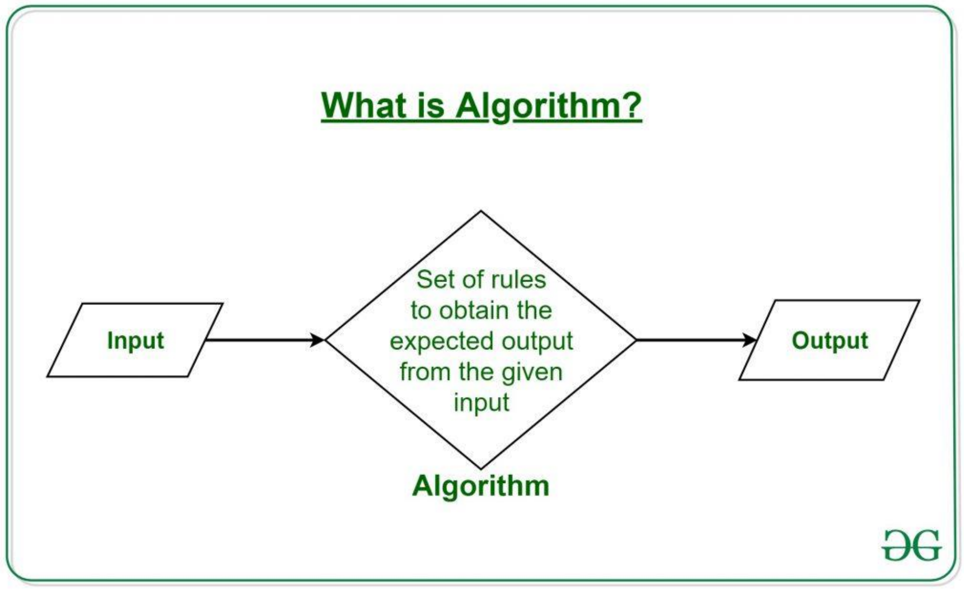
Chapter - 1

1.Algorithm:

Definition of Algorithm

The word Algorithm means " A set of finite rules or instructions to be followed in calculations or other problem-solving operations "
Or
" A procedure for solving a mathematical problem in a finite number of steps that frequently involves recursive operations".


What is Algorithm?
Input
Set of rules to obtain the expected output from the given input
Output
Algorithm

What is the need for algorithms?

1. Algorithms are necessary for solving complex problems efficiently and effectively.
2. They help to automate processes and make them more reliable, faster, and easier to perform.
3. Algorithms also enable computers to perform tasks that would be difficult or impossible for humans to do manually.
4. They are used in various fields such as mathematics, computer science, engineering, finance, and many others to optimize processes, analyze data, make predictions, and provide solutions to problems.

Properties of Algorithm:

- It should terminate after a finite time.
- It should produce at least one output.
- It should take zero or more input.
- It should be deterministic means giving the same output for the same input case.
- Every step in the algorithm must be effective i.e. every step should do some work.

How to Design an Algorithm?

To write an algorithm, the following things are needed as a pre-requisite:

1. The problem that is to be solved by this algorithm i.e. clear problem definition.
2. The constraints of the problem must be considered while solving the problem.
3. The input to be taken to solve the problem.
4. The output is to be expected when the problem is solved.
5. The solution to this problem is within the given constraints.

Algorithm to add 3 numbers and print their sum:

1. START
2. Declare 3 integer variables num1, num2, and num3.
3. Take the three numbers, to be added, as inputs in variables num1, num2, and num3 respectively.
4. Declare an integer variable sum to store the resultant sum of the 3 numbers.
5. Add the 3 numbers and store the result in the variable sum.
6. Print the value of the variable sum
7. END

2. Performance Analysis:

Performance analysis helps us to select the best algorithm from multiple algorithms to solve a problem.

Performance of an algorithm is a process of making evaluative judgement about algorithms.

Performance of an algorithm means predicting the resources which are required to an algorithm to perform its task.

Generally, the performance of an algorithm depends on the following elements...

1. Whether that algorithm is providing the exact solution for the problem?
2. Whether it is easy to understand?
3. Whether it is easy to implement?
4. How much space (memory) it requires to solve the problem?
5. How much time it takes to solve the problem? Etc.,

Performance analysis of an algorithm is performed by using the following measures...

1. Space required to complete the task of that algorithm (**Space Complexity**). It includes program space and data space
2. Time required to complete the task of that algorithm (**Time Complexity**)

3.Space Complexity:

Space complexity is the total amount of memory space used by an algorithm/program including the space of input values for execution. So to find space-complexity, it is enough to calculate the space occupied by the variables used in an algorithm/program.

But often, people confuse Space-complexity with Auxiliary space. Auxiliary space is just a temporary or extra space and it is not the same as space-complexity. In simpler terms,

Space Complexity = Auxiliary space + Space use by input values

Important Note: The best algorithm/program should have the lease space-complexity. The lesser the space used, the faster it executes.

Why do you need to calculate space complexity?

If an algorithm takes up a lot of time, you can still wait, run/execute it to get the desired output. But, if a program takes up a lot of memory space, the compiler will not let you run it**.**

How to calculate Space Complexity of an Algorithm?

Let us understand the Space-Complexity calculation through examples.

Example #1

```
#include<stdio.h>
int main()
{
  int a = 5, b = 5, c;
  c = a + b;
  printf("%d", c);
}
```

In the above program, 3 integer variables are used. The size of the integer data type is 2 or 4 bytes which depends on the compiler. Now, lets assume the size as 4 bytes. So, the total space occupied by the above-given program is 4 * 3 = 12 bytes. Since no additional variables are used, no extra space is required.

Hence, **space complexity for the above-given program is O(1), or constant.**

Example #2

```
#include <stdio.h>
int main()
{
  int n, i, sum = 0;
  scanf("%d", &n);
  int arr[n];
  for(i = 0; i < n; i++)
  {
    scanf("%d", &arr[i]);
    sum = sum + arr[i];
  }
  printf("%d", sum);
}
```

Explanation:

In the above-given code, the array consists of n integer elements. So, the space occupied by the array is 4 * n. Also we have integer variables such as n, i and sum. Assuming 4 bytes for each variable, the total space occupied by the program is 4n + 12 bytes. Since the highest order of n in the equation 4n + 12 is n, so **the space complexity is O(n) or linear.**

4. Time Complexity:

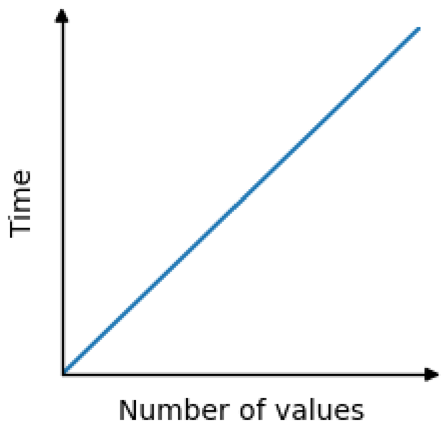
Time complexity is more abstract than actual runtime, and does not consider factors such as programming language or hardware.

Time complexity is the number of operations needed to run an algorithm on large amounts of data. And the number of operations can be considered as time because the computer uses some time for each operation.

For example, in the algorithm that finds the lowest value in an array, each value in the array must be compared one time. Every such comparison can be considered an operation, and each operation takes a certain amount of time. So the total time the algorithm needs to find the lowest value depends on the number of values in the array.

The time it takes to find the lowest value is therefore linear with the number of values. 100 values results in 100 comparisons, and 5000 values results in 5000 comparisons.

The relationship between time and the number of values in the array is linear, and can be displayed in a graph like this:



There are different types of time complexities used, let's see one by one:

1. Constant time – O (1)

2. Linear time – O (n)

3. Logarithmic time – O (log n)

4. Quadratic time – O (n^2)

5. Cubic time – O (n^3)

and many more complex notations like **Exponential time, Quasilinear time, factorial time, etc.** are used based on the type of functions defined.

Constant time – O (1)

An algorithm is said to have constant time with order O (1) when it is not dependent on the input size n. Irrespective of the input size n, the runtime will always be the same.

The above code shows that irrespective of the length of the array (n), the runtime to get the first element in an array of any length is the same. If the run time is considered as

1 unit of time, then it takes only 1 unit of time to run both the arrays, irrespective of length. Thus, the function comes under constant time with order O (1).

Linear time – O(n)

An algorithm is said to have a linear time complexity when the running time increases linearly with the length of the input. When the function involves checking all the values in input data, with this order O(n).

The above code shows that based on the length of the array (n), the run time will get linearly increased. If the run time is considered as 1 unit of time, then it takes only n times 1 unit of time to run the array. Thus, the function runs linearly with input size and this comes with order O(n).

Logarithmic time – O (log n)

An algorithm is said to have a logarithmic time complexity when it reduces the size of the input data in each step. This indicates that the number of operations is not the same as the input size. The number of operations gets reduced as the input size increases. Algorithms are found in binary trees or binary search functions. This involves the search of a given value in an array by splitting the array into two and starting searching in one split. This ensures the operation is not done on every element of the data.

Quadratic time – O (n^2)

An algorithm is said to have a non-linear time complexity where the running time increases non-linearly (n^2) with the length of the input. Generally, nested loops come under this order where one loop takes O(n) and if the function involves a loop within a loop, then it goes for O(n)*O(n) = O(n^2) order.

Similarly, if there are 'm' loops defined in the function, then the order is given by O (n ^ m), which are called **polynomial time complexity** functions.

Thus, the above illustration gives a fair idea of how each function gets the order notation based on the relation between run time against the number of input data sizes and the number of operations performed on them.

5. Asymptotic Notations

Asymptotic notations are the mathematical notations used to describe the running time of an algorithm when the input tends towards a particular value or a limiting value.

For example: In bubble sort, when the input array is already sorted, the time taken by the algorithm is linear i.e. the best case.

But, when the input array is in reverse condition, the algorithm takes the maximum time (quadratic) to sort the elements i.e. the worst case.

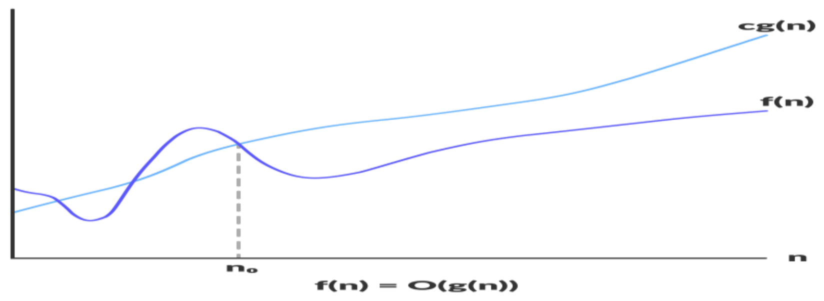
When the input array is neither sorted nor in reverse order, then it takes average time. These durations are denoted using asymptotic notations.

There are mainly three asymptotic notations:

- Big-O notation
- Omega notation
- Theta notation

Big-O Notation (O-notation)

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



Big-O gives the upper bound of a function

```
O(g(n)) = { f(n): there exist positive constants c and n₀
            such that 0 ≤ f(n) ≤ cg(n) for all n ≥ n₀ }
```

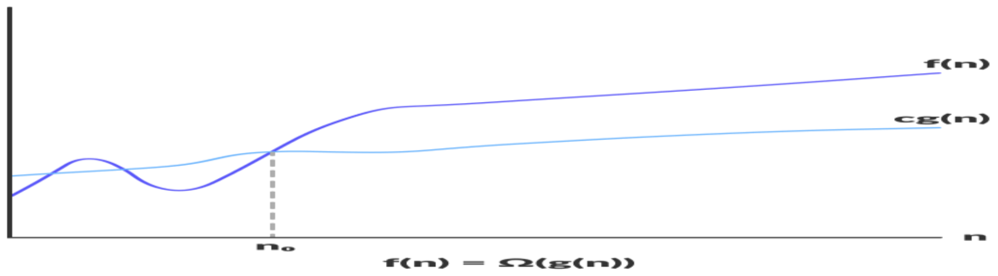
The above expression can be described as a function `f(n)` belongs to the set `O(g(n))` if there exists a positive constant `c` such that it lies between `0` and `cg(n)`, for sufficiently large `n`.

For any value of `n`, the running time of an algorithm does not cross the time provided by `O(g(n))`.

Since it gives the worst-case running time of an algorithm, it is widely used to analyze an algorithm as we are always interested in the worst-case scenario.

Omega Notation (Ω-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 gives the lower bound of a function

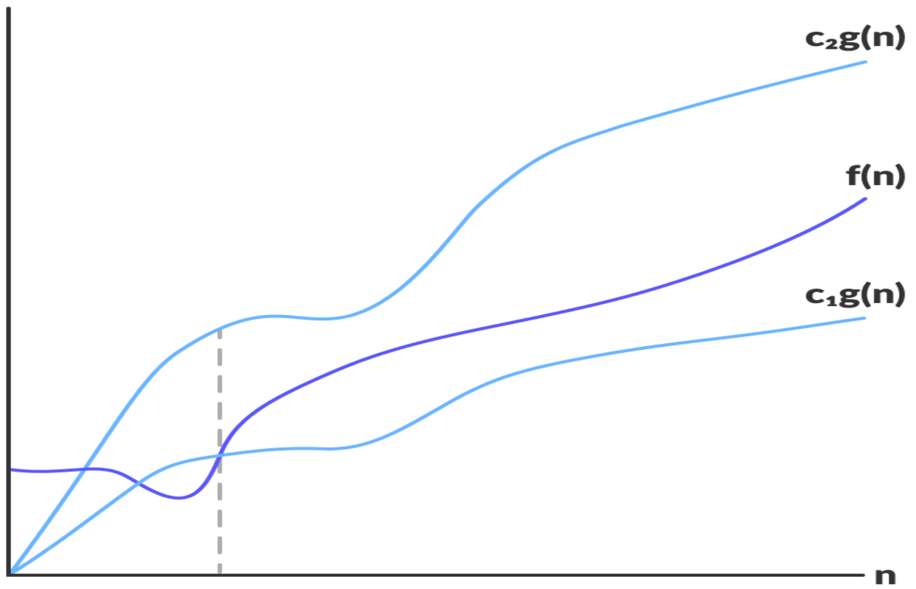
```
Ω(g(n)) = { f(n): there exist positive constants c and n₀
            such that 0 ≤ cg(n) ≤ f(n) for all n ≥ n₀ }
```

The above expression can be described as a function `f(n)` belongs to the set `Ω(g(n))` if there exists a positive constant `c` such that it lies above `cg(n)`, for sufficiently large `n`.

For any value of `n`, the minimum time required by the algorithm is given by Omega `Ω(g(n))`.

Theta Notation (Θ-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 analyzing the average-case complexity of an algorithm.



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

Theta bounds the function within constants factors

For a function $g(n)$, $\Theta(g(n))$ is given by the relation:

```
Θ(g(n)) = { f(n): there exist positive constants c₁, c₂ and n₀
            such that 0 ≤ c₁g(n) ≤ f(n) ≤ c₂g(n) for all n ≥ n₀ }
```

The above expression can be described as a function $f(n)$ belongs to the set $\Theta(g(n))$ if there exist positive constants c_1 and c_2 such that it can be sandwiched between $c_1g(n)$ and $c_2g(n)$, for sufficiently large n.

If a function $f(n)$ lies anywhere in between $c_1g(n)$ and $c_2g(n)$ for all $n \geq n0$, then $f(n)$ is said to be asymptotically tight bound.

Examples on Upper Bound Asymptotic Notation

Example: Find upper bound of running time of constant function f(n) = 6993.
To find the upper bound of f(n), we have to find c and n_0 such that $0 \le f(n) \le c.g(n)$ for all $n \ge n_0$
$0 \le f(n) \le c \times g(n)$

$0 \le 6993 \le c \times g(n)$

$0 \le 6993 \le 6993 \times 1$

So, c = 6993 and g(n) = 1

Any value of c which is greater than 6993, satisfies the above inequalities, so all such values of c are possible.

$0 \le 6993 \le 8000 \times 1 \rightarrow$ true

$0 \le 6993 \le 10500 \times 1 \rightarrow$ true

Function f(n) is constant, so it does not depend on problem size n. So $n_0 = 1$
$f(n) = O(g(n)) = O(1)$ for c = 6993, $n_0 = 1$
$f(n) = O(g(n)) = O(1)$ for c = 8000, $n_0 = 1$ and so on.
Example: Find upper bound of running time of a linear function f(n) = 6n + 3.
To find upper bound of f(n), we have to find c and n_0 such that $0 \le f(n) \le c \times g(n)$ for all $n \ge n_0$
$0 \le f(n) \le c \times g(n)$

$0 \le 6n + 3 \le c \times g(n)$

$0 \le 6n + 3 \le 6n + 3n$, for all $n \ge 1$ (There can be such infinite possibilities)

$0 \le 6n + 3 \le 9n$

So, c = 9 and g(n) = n, $n_0 = 1$
Tabular Approach
$0 \le 6n + 3 \le c \times g(n)$

$0 \le 6n + 3 \le 7n$

Now, manually find out the proper n_0, such that $f(n) \le c.g(n)$

n	**f(n) = 6n + 3**	**c.g(n) = 7n**
1	9	7
2	15	14
3	21	21

n	**$f(n) = 6n + 3$**	**$c.g(n) = 7n$**
4	27	28
5	33	35

From Table, for $n \geq 3$, $f(n) \leq c \times g(n)$ holds true. So, $c = 7$, $g(n) = n$ and $n_0 = 3$, There can be such multiple pair of (c, n_0)
$f(n) = O(g(n)) = O(n)$ for $c = 9$, $n_0 = 1$
$f(n) = O(g(n)) = O(n)$ for $c = 7$, $n_0 = 3$
and so on.

Example: Find upper bound of running time of quadratic function $f(n) = 3n^2 + 2n + 4$.
To find upper bound of $f(n)$, we have to find c and n_0 such that $0 \leq f(n) \leq c \times g(n)$ for all $n \geq n_0$
$0 \leq f(n) \leq c \times g(n)$

$0 \leq 3n^2 + 2n + 4 \leq c \times g(n)$
$0 \leq 3n^2 + 2n + 4 \leq 3n^2 + 2n^2 + 4n^2$,
for all $n \geq 1$:

$0 \leq 3n^2 + 2n + 4 \leq 9n^2$
$0 \leq 3n^2 + 2n + 4 \leq 9n^2$
So, $c = 9$, $g(n) = n^2$ and $n_0 = 1$
Tabular approach:
$0 \leq 3n^2 + 2n + 4 \leq c.g(n)$
$0 \leq 3n^2 + 2n + 4 \leq 4n^2$
Now, manually find out the proper n_0, such that $f(n) \leq c.g(n)$

n	**$f(n) = 3n^2 + 2n + 4$**	**$c.g(n) = 4n^2$**
1	9	4
2	20	16
3	37	36
4	60	64
5	89	100

From Table, for $n \geq 4$, $f(n) \leq c \times g(n)$ holds true. So, $c = 4$, $g(n) = n^2$ and $n_0 = 4$. There can be such multiple pair of (c, n_0)
$f(n) = O(g(n)) = O(n^2)$ for $c = 9$, $n_0 = 1$
$f(n) = O(g(n)) = O(n^2)$ for $c = 4$, $n_0 = 4$
and so on.

Example: Find upper bound of running time of a cubic function $f(n) = 2n^3 + 4n + 5$.

To find upper bound of f(n), we have to find c and n_0 such that $0 \le f(n) \le c \times g(n)$ for all $n \ge n_0$
$0 \le f(n) \le c.g(n)$

$0 \le 2n^3 + 4n + 5 \le c \times g(n)$
$0 \le 2n^3 + 4n + 5 \le 2n^3 + 4n^3 + 5n^3$, for all n ³ 1
$0 \le 2n^3 + 4n + 5 \le 11n^2$
So, c = 11, $g(n) = n^3$ and $n_0 = 1$
Tabular approach

$0 \le 2n^3 + 4n + 5 \le c \times g(n)$
$0 \le 2n^3 + 4n + 5 \le 3n^3$
Now, manually find out the proper n_0, such that $f(n) \le c \times g(n)$

n	**$f(n) = 2n^3 + 4n + 5$**	**$c.g(n) = 3n^3$**
1	11	3
2	29	24
3	71	81
4	149	192

From Table, for $n \ge 3$, $f(n) \le c \times g(n)$ holds true. So, c = 3, $g(n) = n^3$ and $n_0 = 3$. There can be such multiple pair of (c, n_0)
$f(n) = O(g(n)) = O(n^3)$ for c = 11, $n_0 = 1$
$f(n) = O(g(n)) = O(n^3)$ for c =3, $n_0 = 3$ and so on.

Lower Bound

Lower bound of any function is defined as follow:

Let f(n) and g(n) are two nonnegative functions indicating the running time of two algorithms. We say the function g(n) is lower bound of function f(n) if there exist some positive constants c and n_0 such that $0 \le\le c.g(n) \le f(n)$ for all $n \ge n_0$. It is denoted as $f(n) = \Omega(g(n))$.

Lower bound – Big Omega

Examples on Lower Bound Asymptotic Notation

Example: Find lower bound of running time of constant function f(n) = 23.
To find lower bound of f(n), we have to find c and n_0 such that $\{ 0 \le c \times g(n) \le f(n)$ for all $n \ge n_0 \}$
$0 \le c \times g(n) \le f(n)$
$0 \le c \times g(n) \le 23$
$0 \le 23.1 \le 23 \rightarrow$ true

$0 \le 12.1 \le 23 \rightarrow$ true

$0 \le 5.1 \le 23 \rightarrow$ true

Above all three inequalities are true and there exists such infinite inequalities

So c = 23, c = 12, c = 5 and g(n) = 1. Any value of c which is less than or equals to 23, satisfies the above inequality, so all such value of c are possible. Function f(n) is constant, so it does not depend on problem size n. Hence $n_0 = 1$
$f(n) = \Omega(g(n)) = \Omega(1)$ for $c = 23$, $n_0 = 1$
$f(n) = \Omega(g(n)) = \Omega(1)$ for $c = 12$, $n_0 = 1$ and so on.
Example: Find lower bound of running time of a linear function f(n) = 6n + 3.
To find lower bound of f(n), we have to find c and n_0 such that $0 \le c.g(n) \le f(n)$ for all $n \ge n_0$
$0 \le c \times g(n) \le f(n)$
$0 \le c \times g(n) \le 6n + 3$
$0 \le 6n \le 6n + 3 \rightarrow$ true, for all $n \ge n_0$
$0 \le 5n \le 6n + 3 \rightarrow$ true, for all $n \ge n_0$
Above both inequalities are true and there exists such infinite inequalities. So,

$f(n) = \Omega(g(n)) = \Omega(n)$ for $c = 6$, $n_0 = 1$
$f(n) = \Omega(g(n)) = \Omega(n)$ for $c = 5$, $n_0 = 1$
and so on.

Example: Find lower bound of running time of quadratic function $f(n) = 3n^2 + 2n + 4$.
To find lower bound of f(n), we have to find c and n_0 such that $0 \le c.g(n) \le f(n)$ for all $n^3 n_0$
$0 \le c \times g(n) \le f(n)$
$0 \le c \times g(n) \le 3n^2 + 2n + 4$
$0 \le 3n^2 \le 3n^2 + 2n + 4, \rightarrow$ true, for all $n \ge 1$
$0 \le n^2 \le 3n^2 + 2n + 4, \rightarrow$ true, for all $n \ge 1$
Above both inequalities are true and there exists such infinite inequalities.

So, $f(n) = \Omega(g(n)) = \Omega(n^2)$ for $c = 3$, $n_0 = 1$
$f(n) = \Omega(g(n)) = \Omega(n^2)$ for $c = 1$, $n_0 = 1$
and so on.

Example: Find lower bound of running time of quadratic function $f(n) = 2n^3 + 4n + 5$.
To find lower bound of f(n), we have to find c and n_0 such that $0 \le c.g(n) \le f(n)$ for all $n \ge n_0$
$0 \le c \times g(n) \le f(n)$
$0 \le c \times g(n) \le 2n^3 + 4n + 5$
$0 \le 2n^3 \le 2n^3 + 4n + 5 \rightarrow$ true, for all $n \ge 1$
$0 \le n^3 \le 2n^3 + 4n + 5 \rightarrow$ true, for all $n \ge 1$
Above both inequalities are true and there exists such infinite inequalities.

So, $f(n) = \Omega(g(n)) = \Omega(n^3)$ for $c = 2$, $n_0 = 1$
$f(n) = \Omega(g(n)) = \Omega(n^3)$ for $c = 1$, $n_0 = 1$
and so on.

Tight Bound

Tight bound of any function is defined as follow:

Let f(n) and g(n) are two nonnegative functions indicating running time of two algorithms. We say the function g(n) is tight bound of function f(n) if there exist some positive constants c_1, c_2, and n_0 such that $0 \le c_1 \times g(n) \le f(n) \le c_2 \times g(n)$ for all $n \ge n_0$. It is denoted as $f(n) = \Theta(g(n))$.

Tight Bound – Big Theta

Examples on Tight Bound Asymptotic Notation:

Example: Find tight bound of running time of constant function f(n) = 23.
To find tight bound of f(n), we have to find c_1, c_2 and n_0 such that, $0 \le c_1 \times g(n) \le f(n) \le c_2 \times g(n)$ for all $n \ge n_0$
$0 \le c_1 \times g(n) \le 23 \le c_2 \times g(n)$
$0 \le 22 \times 1 \le 23 \le 24 \times 1$, → true for all $n \ge 1$
$0 \le 10 \times 1 \le 23 \le 50 \times 1$, → true for all $n \ge 1$
Above both inequalities are true and there exists such infinite inequalities.

So, $(c_1, c_2) = (22, 24)$ and $g(n) = 1$, for all $n \ge 1$
$(c_1, c_2) = (10, 50)$ and $g(n) = 1$, for all $n \ge 1$
$f(n) = \Theta(g(n)) = \Theta(1)$ for $c_1 = 22$, $c_2 = 24$, $n_0 = 1$
$f(n) = \Theta(g(n)) = \Theta(1)$ for $c_1 = 10$, $c_2 = 50$, $n_0 = 1$
and so on.

Example: Find tight bound of running time of a linear function f(n) = 6n + 3.
To find tight bound of f(n), we have to find c_1, c_2 and n_0 such that, $0 \le c_1 \times g(n) \le f(n) \le c_2 \times g(n)$ for all $n \ge n_0$
$0 \le c_1 \times g(n) \le 6n + 3 \le c_2 \times g(n)$
$0 \le 5n \le 6n + 3 \le 9n$, for all $n \ge 1$

Above inequality is true and there exists such infinite inequalities.

So, $f(n) = \Theta(g(n)) = \Theta(n)$ for $c_1 = 5$, $c_2 = 9$, $n_0 = 1$
Example: Find tight bound of running time of quadratic function $f(n) = 3n^2 + 2n + 4$.
To find tight bound of f(n), we have to find c_1, c_2 and n_0 such that, $0 \le c_1 \times g(n) \le f(n) \le c_2 \times g(n)$ for all $n \ge n_0$
$0 \le c_1 \times g(n) \le 3n^2 + 2n + 4 \le c_2 \times g(n)$
$0 \le 3n^2 \le 3n^2 + 2n + 4 \le 9n^2$, for all $n \ge 1$
Above inequality is true and there exists such infinite inequalities. So,

$f(n) = \Theta(g(n)) = \Theta(n^2)$ for $c_1 = 3$, $c_2 = 9$, $n_0 = 1$
Example: Find tight bound of running time of a cubic function $f(n) = 2n^3 + 4n + 5$.
To find tight bound of f(n), we have to find c_1, c_2 and n_0 such that, $0 \le c_1 \times g(n) \le f(n) \le c_2 \times g(n)$ for all $n \ge n_0$
$0 \le c_1 \times g(n) \le 2n^3 + 4n + 5 \le c_2 \times g(n)$
$0 \le 2n^3 \le 2n^3 + 4n + 5 \le 11n^3$, for all $n \ge 1$

Above inequality is true and there exists such infinite inequalities. So,

f(n) = Θ(g(n)) = Θ(n^3) for $c_1 = 2$, $c_2 = 11$, $n_0 = 1$.

Little o Notations

There are some other notations present except the Big-Oh, Big-Omega and Big-Theta notations. The little o notation is one of them.

Little o notation is used to describe an upper bound that cannot be tight. In other words, loose upper bound of f(n).

Let f(n) and g(n) are the functions that map positive real numbers. We can say that the function f(n) is o(g(n)) if for any real positive constant c, there exists an integer constant n0 ≤ 1 such that f(n) > 0.

Mathematical Relation of Little o notation

Using mathematical relation, we can say that f(n) = o(g(n)) means,

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

Example on little o asymptotic notation

If f(n) = n^2 and g(n) = n^3 then check whether f(n) = o(g(n)) or not.

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

$$= \lim_{n\to\infty} \frac{1}{n}$$

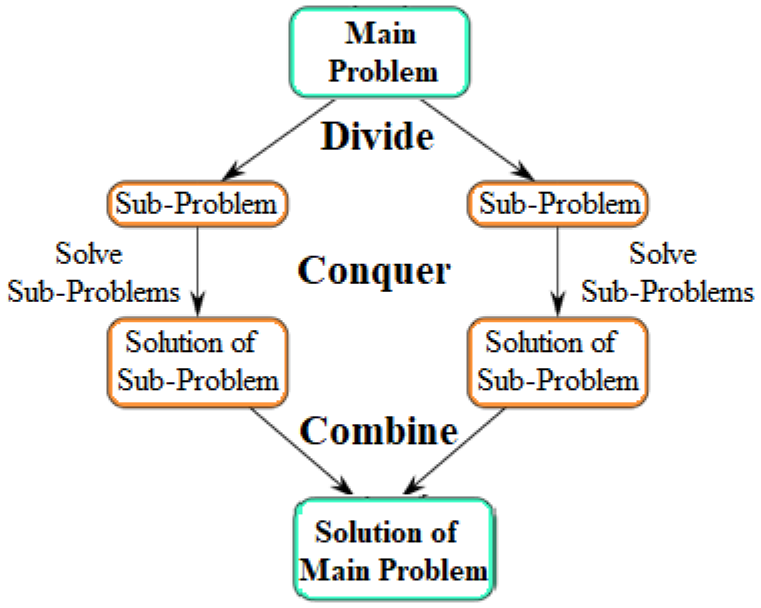
$$= \frac{1}{\infty}$$

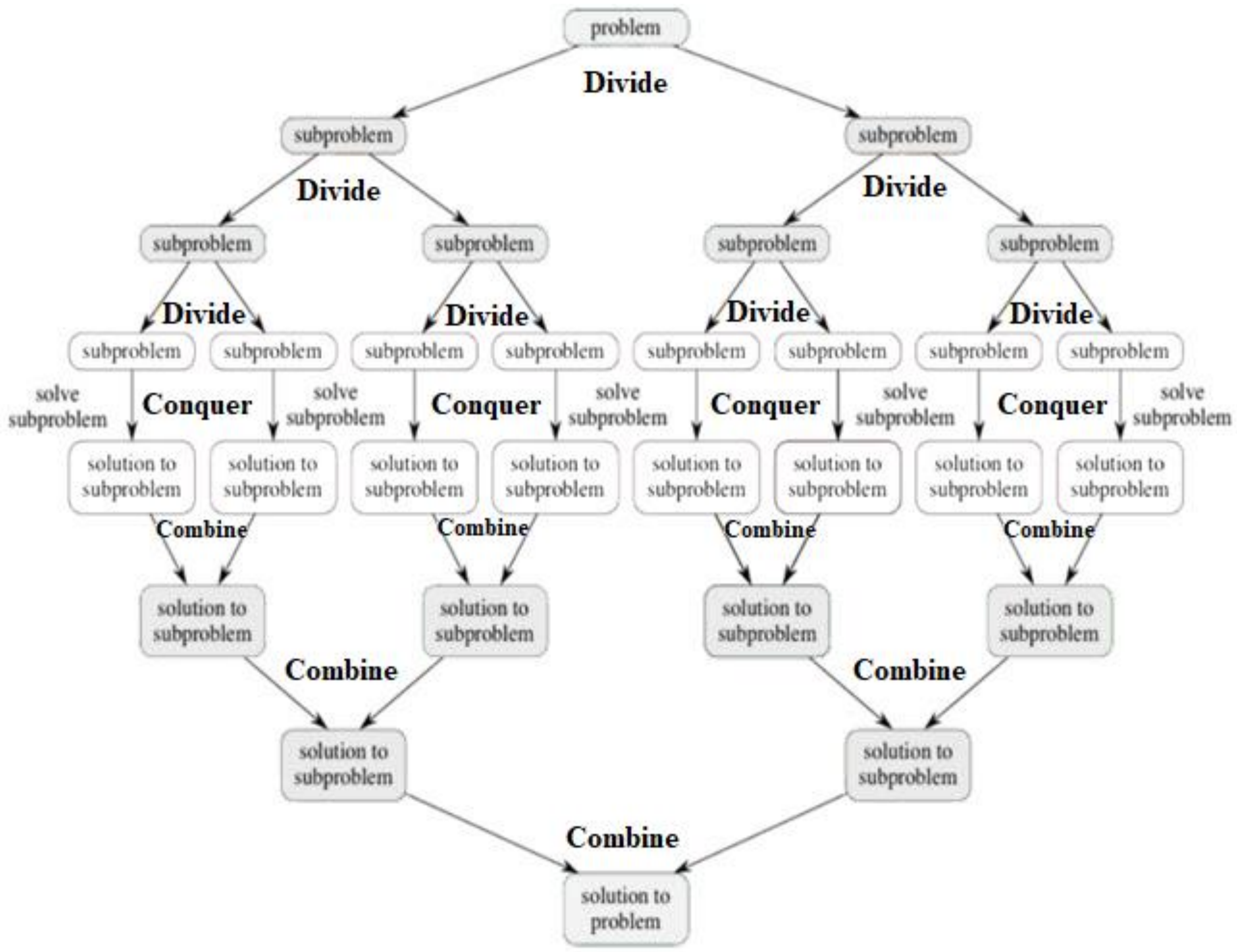
$$= 0$$

The result is 0, and it satisfies the equation mentioned above. So we can say that f(n) = o(g(n)).

6.General Procedure of Divide-and-Conquer Method

- In simple words, Divide-and-Conquer break down the main problem into small sub-problems. Then solve that sub-problem independently, and at last combine the solutions of small sub-problems as a solution for the main problem.
- Divide-and-Conquer creates at least two sub-problems, a divide-and-conquer algorithm makes multiple recursive calls.
- Some divide-and-conquer algorithms create more than two sub-problems also.
- Divide-and-Conquer solve sub-problems recursively, each sub-problem must be smaller than the original problem, and there must be a base case for sub-problems.
- Divide-and-Conquer algorithms have ***three parts*** as follows:
    - **Divide** the problem into a number of sub-problems that are small instances of the main problem.
    - **Conquer** the sub-problems by solving them recursively. If they are small enough, solve the sub-problems as base cases.
    - **Combine** the solutions of the sub-problems as one complete solution for the main problem.





Benefits of Divide-and-Conquer:

- Easily solve large problems faster than other algorithms.
- It solves simple sub-problems within the cache memory instead of accessing the slower main memory.
- It supports parallelism because sub-problems are solved independently.
- Hence, the algorithm made using this approach runs on the multiprocessor system.

Drawback of Divide and Conquer:

- It uses recursion therefore sometimes memory management is a very difficult task.
- An explicit stack may overuse the space.

- It may crash the system if the recursion is carried out rigorously greater than the stack present in the CPU.

Applications of Divide and Conquer:

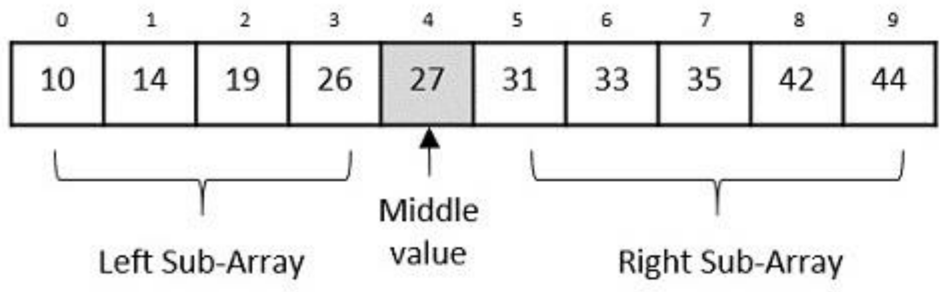
There are lots of applications of this divide-and-conquer approach as follows:

- Merge Sort
- Quick Sort
- Binary Search
- Tower of Hanoi
- Closest Pair of Points
- Karatsuba Algorithm
- Strassen's Matrix multiplication
- Cooley-Tukey Fast Fourier Transform (FFT) algorithm
- Finding the maximum and minimum of a sequence of numbers

7.Binary Search:

Binary search is a fast search algorithm with run-time complexity of O(log n). This search algorithm works on the principle of divide and conquer, since it divides the array into half before searching. For this algorithm to work properly, the data collection should be in the sorted form.

Binary search looks for a particular key value by comparing the middle most item of the collection. If a match occurs, then the index of item is returned. But if the middle item has a value greater than the key value, the right sub-array of the middle item is searched. Otherwise, the left sub-array is searched. This process continues recursively until the size of a subarray reduces to zero.



Binary Search Algorithm

Binary Search algorithm is an interval searching method that performs the searching in intervals only. The input taken by the binary search algorithm must always be in a sorted array since it divides the array into subarrays based on the greater or lower values. The algorithm follows the procedure below −

Step 1 − Select the middle item in the array and compare it with the key value to be searched. If it is matched, return the position of the median.

Step 2 − If it does not match the key value, check if the key value is either greater than or less than the median value.

Step 3 − If the key is greater, perform the search in the right sub-array; but if the key is lower than the median value, perform the search in the left sub-array.

Step 4 − Repeat Steps 1, 2 and 3 iteratively, until the size of sub-array becomes 1.

Step 5 − If the key value does not exist in the array, then the algorithm returns an unsuccessful search.

Pseudocode

The pseudocode of binary search algorithms should look like this −

```
Procedure binary_search
   A ← sorted array
   n ← size of array
   x ← value to be searched

   Set lowerBound = 1
   Set upperBound = n

   while x not found
      if upperBound < lowerBound
         EXIT: x does not exists.

      set midPoint = lowerBound + ( upperBound - lowerBound ) / 2

      if A[midPoint] < x
         set lowerBound = midPoint + 1

      if A[midPoint] > x
      set upperBound = midPoint - 1

    if A[midPoint] = x
      EXIT: x found at location midPoint
  end while
end procedure
```

Analysis

Since the binary search algorithm performs searching iteratively, calculating the time complexity is not as easy as the linear search algorithm.

The input array is searched iteratively by dividing into multiple sub-arrays after every unsuccessful iteration. Therefore, the recurrence relation formed would be of a dividing function.

To explain it in simpler terms,

- During the first iteration, the element is searched in the entire array. Therefore, length of the array = n.
- In the second iteration, only half of the original array is searched. Hence, length of the array = n/2.
- In the third iteration, half of the previous sub-array is searched. Here, length of the array will be = n/4.
- Similarly, in the ith iteration, the length of the array will become $n/2^i$

To achieve a successful search, after the last iteration the length of array must be 1. Hence,

```
n/2i = 1
```

That gives us −

```
n = 2i
```

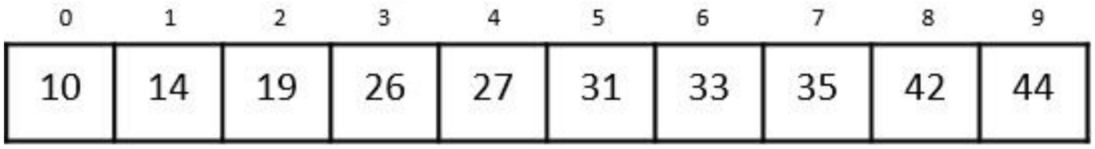
Applying log on both sides,

```
log n = log 2i
log n = i. log 2
i = log n
```

The time complexity of the binary search algorithm is **O(log n)**

Example

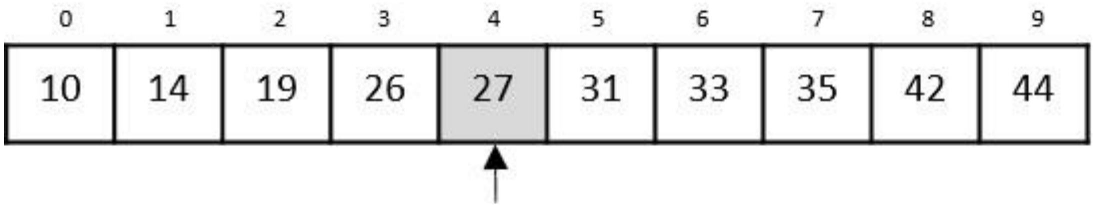
For a binary search to work, it is mandatory for the target array to be sorted. We shall learn the process of binary search with a pictorial example. The following is our sorted array and let us assume that we need to search the location of value 31 using binary search.



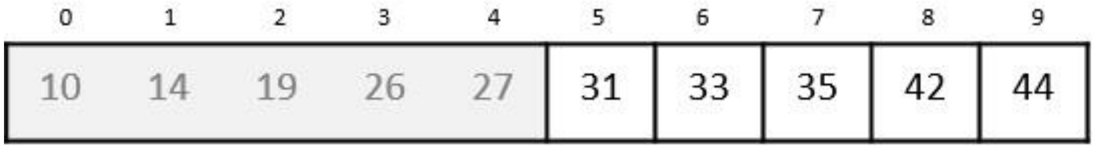
First, we shall determine half of the array by using this formula −

```
mid = low + (high - low) / 2
```

Here it is, 0 + (9 - 0) / 2 = 4 (integer value of 4.5). So, 4 is the mid of the array.



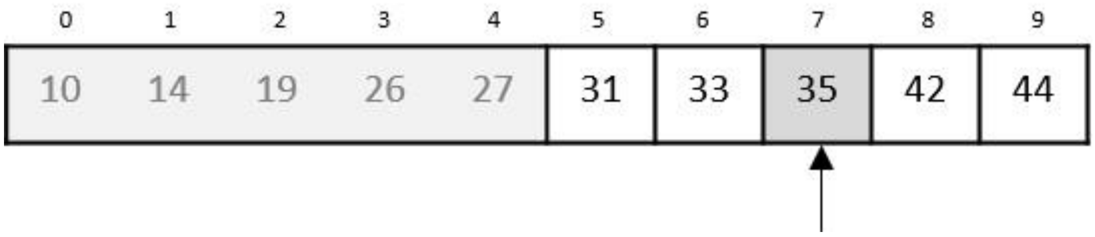
Now we compare the value stored at location 4, with the value being searched, i.e. 31. We find that the value at location 4 is 27, which is not a match. As the value is greater than 27 and we have a sorted array, so we also know that the target value must be in the upper portion of the array.



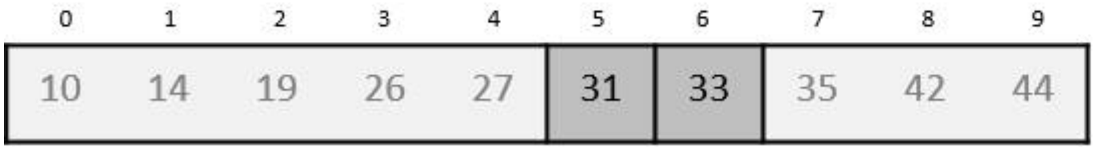
We change our low to mid + 1 and find the new mid value again.

```
low = mid + 1
mid = low + (high - low) / 2
```

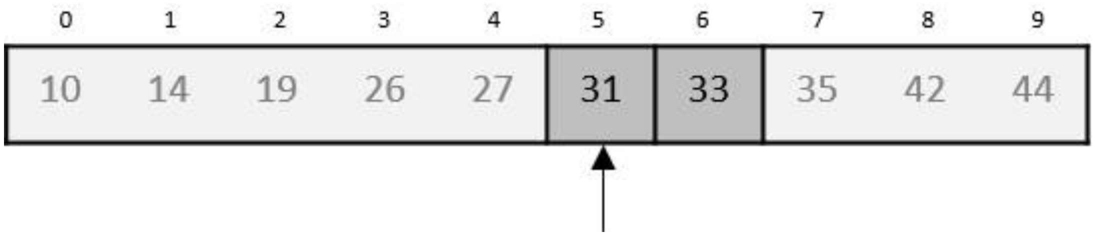
Our new mid is 7 now. We compare the value stored at location 7 with our target value 31.



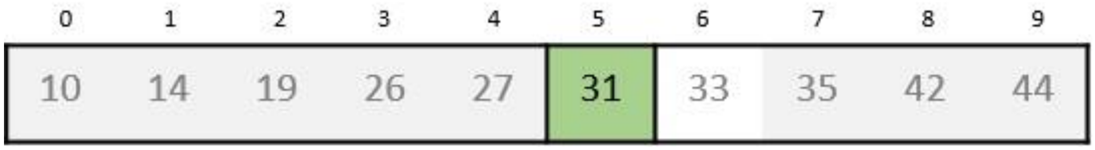
The value stored at location 7 is not a match, rather it is less than what we are looking for. So, the value must be in the lower part from this location.



Hence, we calculate the mid again. This time it is 5.



We compare the value stored at location 5 with our target value. We find that it is a match.



We conclude that the target value 31 is stored at location 5.

Binary search halves the searchable items and thus reduces the count of comparisons to be made to very less numbers.

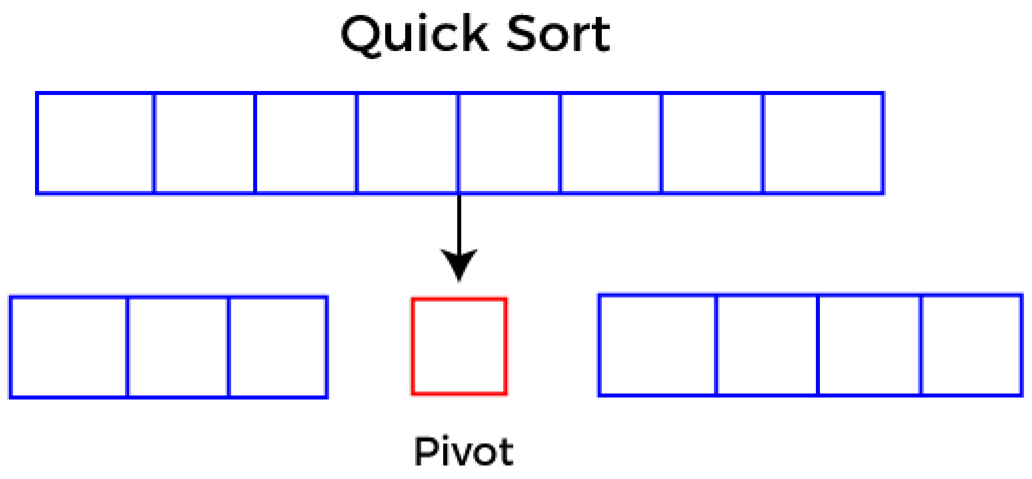
8.Quick Sort:

Quick sort is a highly efficient sorting algorithm and is based on partitioning of array of data into smaller arrays. A large array is partitioned into two arrays one of which holds values smaller than the specified value, say pivot, based on which the partition is made and another array holds values greater than the pivot value.

Quicksort partitions an array and then calls itself recursively twice to sort the two resulting subarrays. This algorithm is quite efficient for large-sized data sets as its average and worst-case complexity are O(n2), respectively.

Quicksort picks an element as pivot, and then it partitions the given array around the picked pivot element. In quick sort, a large array is divided into two arrays in which one holds values that are smaller than the specified value (Pivot), and another array holds the values that are greater than the pivot.

After that, left and right sub-arrays are also partitioned using the same approach. It will continue until the single element remains in the sub-array.



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.

Algorithm

Algorithm:

```
1.  QUICKSORT (array A, start, end)
2.  {
3.   1 if (start < end)
```

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

Partition Algorithm:

The partition algorithm rearranges the sub-arrays in a place.

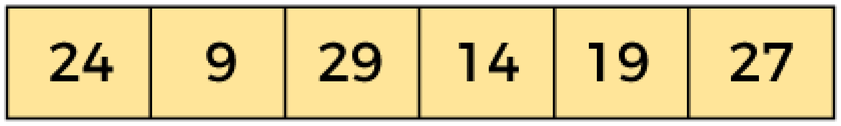
```
PARTITION (array A, start, end)
{
 1 pivot ? A[end]
 2 i ? start-1
 3 for j ? start to end -1 {
 4 do if (A[j] < pivot) {
 5 then i ? i + 1
 6 swap A[i] with A[j]
 7  }}
 8 swap A[i+1] with A[end]
 9 return i+1
}
```

Working of Quick Sort Algorithm

Now, let's see the working of the Quicksort Algorithm.

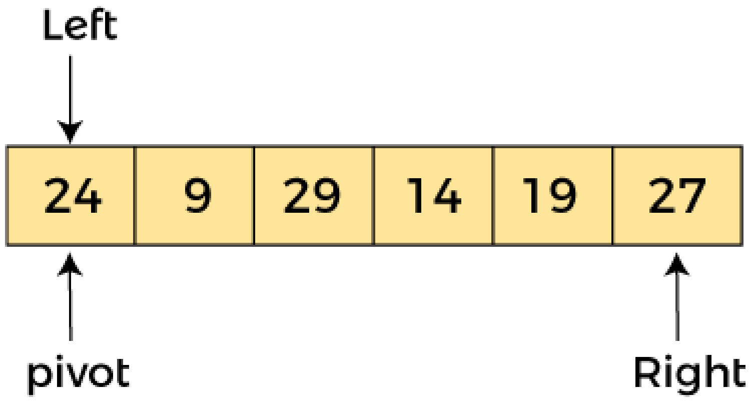
To understand the working of quick sort, let's take an unsorted array. It will make the concept more clear and understandable.

Let the elements of array are -

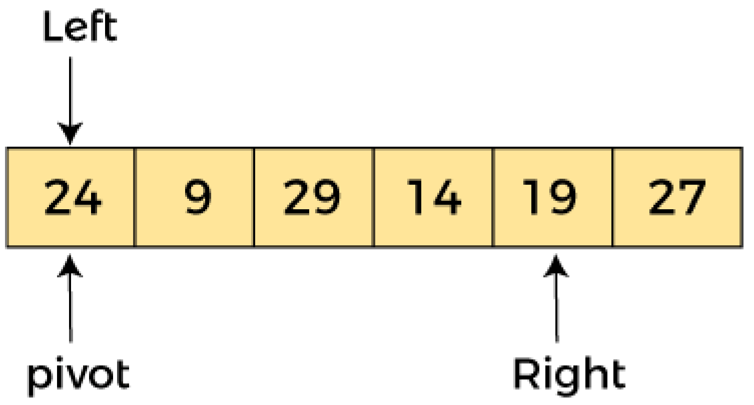


In the given array, we consider the leftmost element as pivot. So, in this case, a[left] = 24, a[right] = 27 and a[pivot] = 24.

Since, pivot is at left, so algorithm starts from right and move towards left.

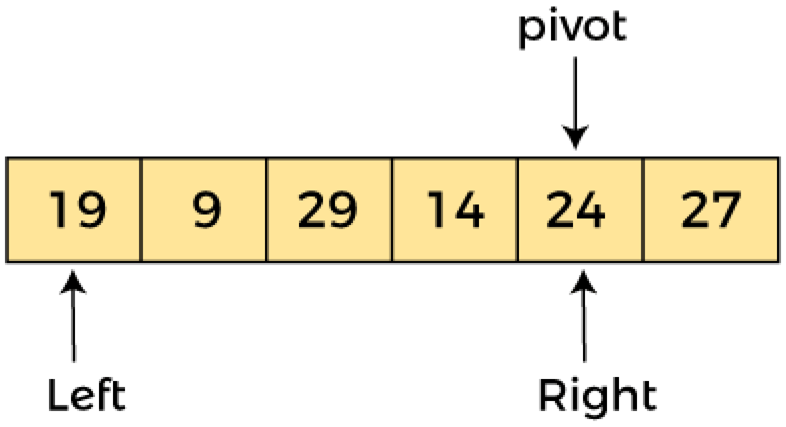


Now, a[pivot] < a[right], so algorithm moves forward one position towards left, i.e. -



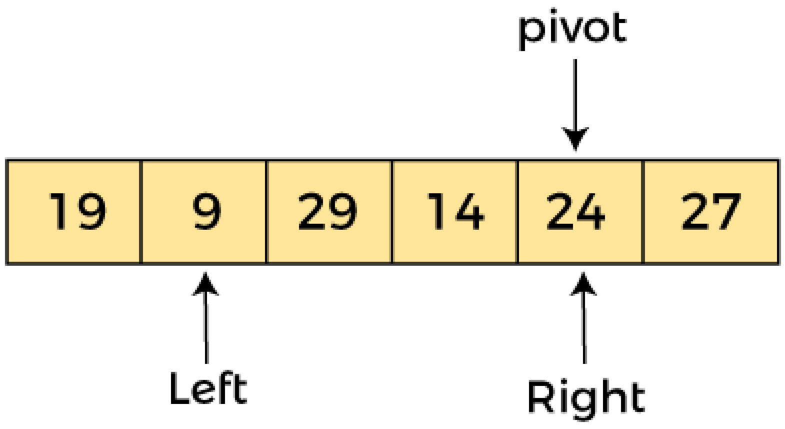
Now, a[left] = 24, a[right] = 19, and a[pivot] = 24.

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

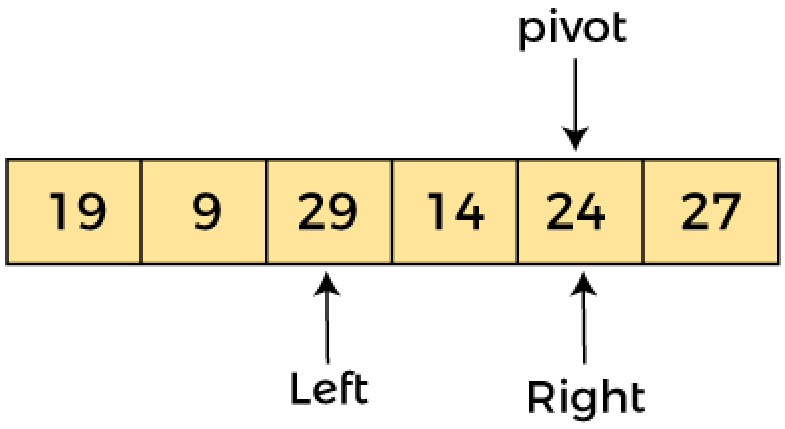


Now, a[left] = 19, a[right] = 24, and a[pivot] = 24. Since, pivot is at right, so algorithm starts from left and moves to right.

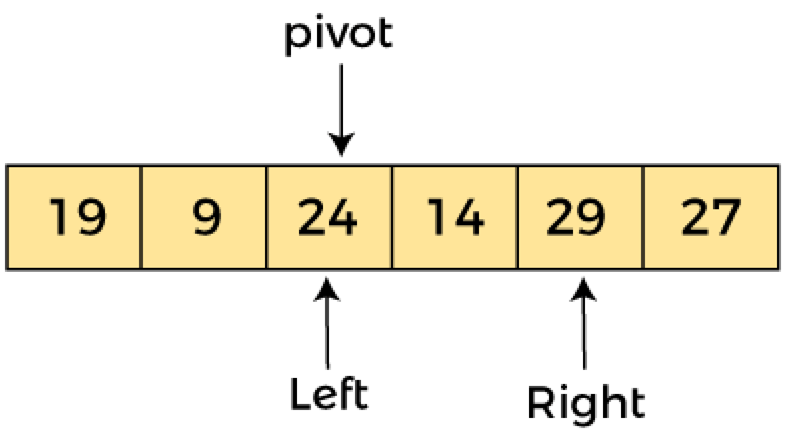
As a[pivot] > a[left], so algorithm moves one position to right as -



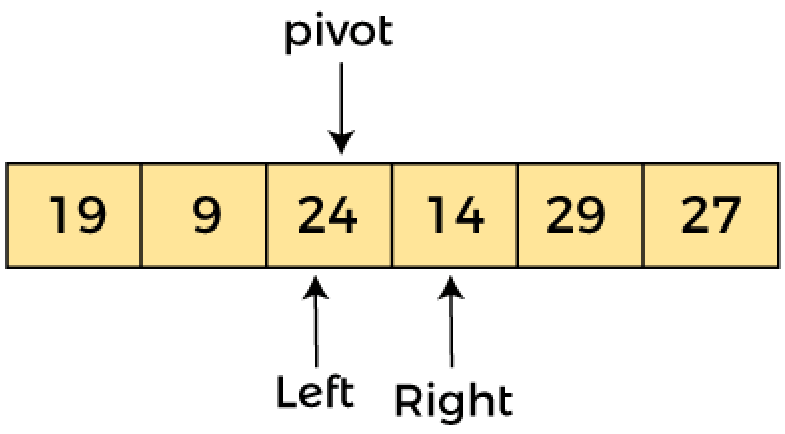
Now, a[left] = 9, a[right] = 24, and a[pivot] = 24. As a[pivot] > a[left], so algorithm moves one position to right as -



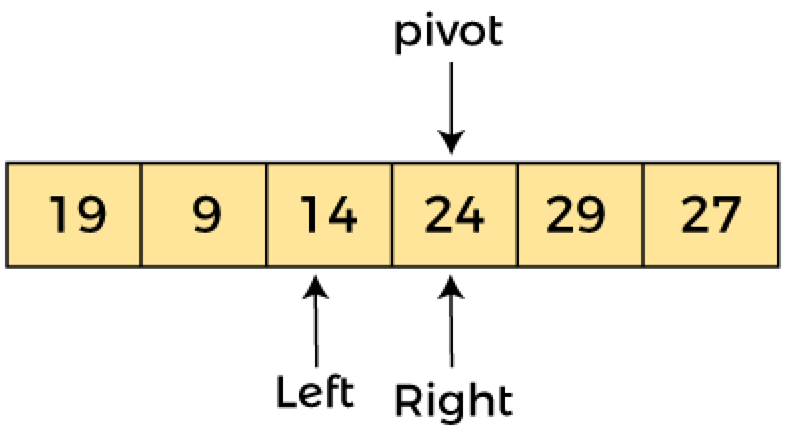
Now, a[left] = 29, a[right] = 24, and a[pivot] = 24. As a[pivot] < a[left], so, swap a[pivot] and a[left], now pivot is at left, i.e. -



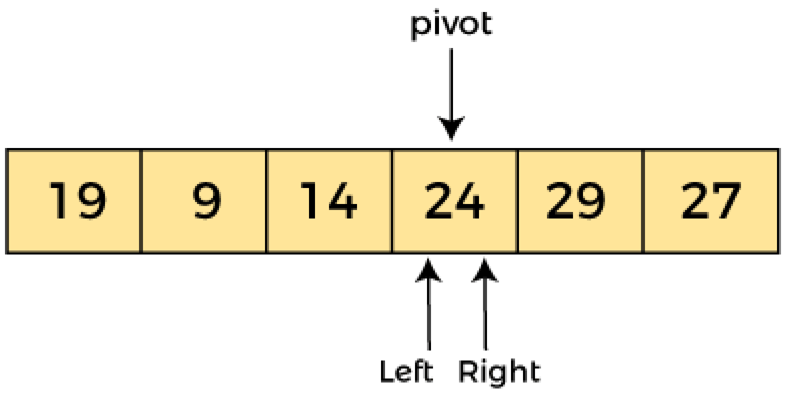
Since, pivot is at left, so algorithm starts from right, and move to left. Now, a[left] = 24, a[right] = 29, and a[pivot] = 24. As a[pivot] < a[right], so algorithm moves one position to left, as -



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. -



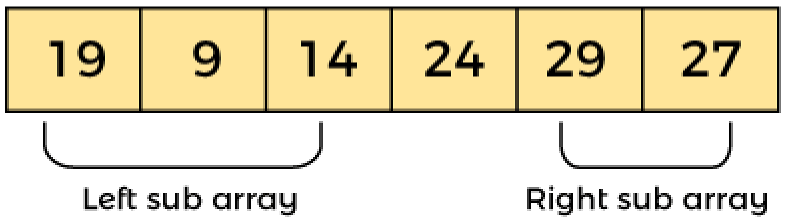
Now, a[pivot] = 24, a[left] = 14, and a[right] = 24. Pivot is at right, so the algorithm starts from left and move to right.



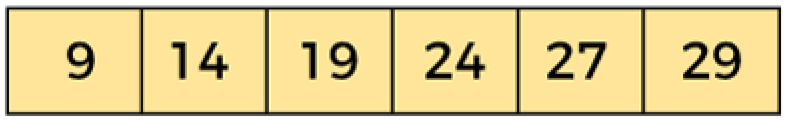
Now, a[pivot] = 24, a[left] = 24, and a[right] = 24. So, pivot, left and right are pointing the same element. It represents the termination of procedure.

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

Elements that are right side of element 24 are greater than it, and the elements that are left side of element 24 are smaller than it.



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 -



9.Merge Sort:

Merge sort is a sorting technique based on divide and conquer technique. With worst-case time complexity being O(n log n), it is one of the most used and approached algorithms.

Merge sort first divides the array into equal halves and then combines them in a sorted manner.

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**.

And then, 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.

As you can see in the image given below, the merge sort algorithm recursively divides the array into halves until the base condition is met, where we are left with only 1 element in the array. And then, the merge function picks up the sorted sub-arrays and merge them back to sort the entire array.

The following figure illustrates the dividing (splitting) procedure.

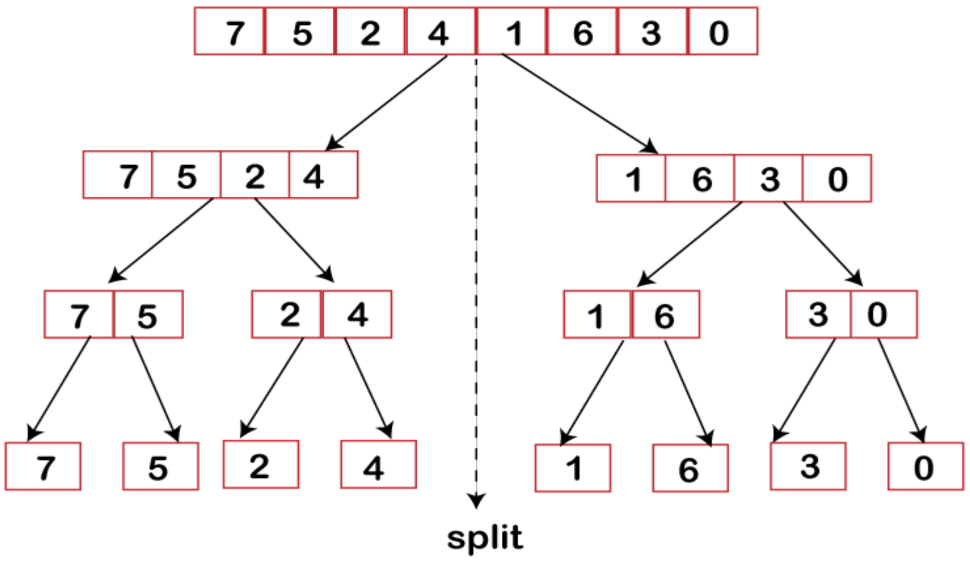


Figure 1: Merge Sort Divide Phase

```
FUNCTIONS: MERGE (A, p, q, r)

1. n 1 = q-p+1
2. n 2= r-q
3. create arrays [1.....n 1 + 1] and R [ 1.....n 2 +1 ]
4. for i ← 1 to n 1
5. do [i] ← A [ p+ i-1]
6. for j ← 1 to n2
7. do R[j] ← A[ q + j]
8. L [n 1+ 1] ← ∞
9. R[n 2+ 1] ← ∞
10. I ← 1
11. J ← 1
12. For k ← p to r
13. Do if L [i] ≤ R[j]
14. then A[k] ← L[ i]
15. i ← i +1
16. else A[k] ← R[j]
17. j ← j+1
```

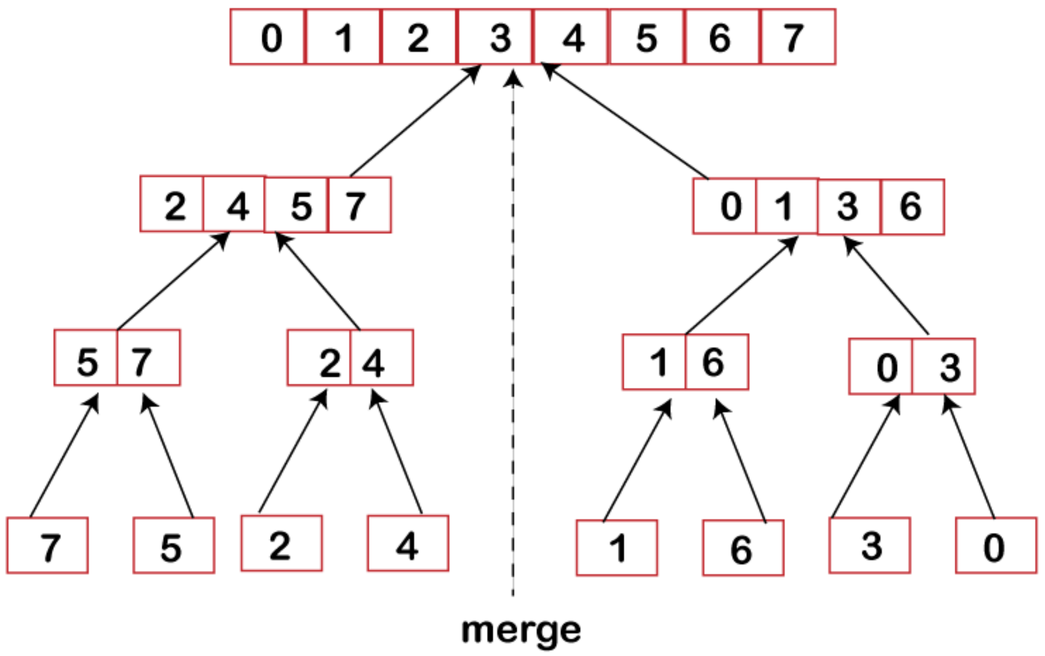


Figure 2: Merge Sort Combine Phase

The merge step of Merge Sort

Mainly the recursive algorithm depends on a base case as well as its ability to merge back the results derived from the base cases. Merge sort is no different algorithm, just the fact here the **merge** step possesses more importance.

To any given problem, the merge step is one such solution that combines the two individually sorted lists(arrays) to build one large sorted list(array).

The merge sort algorithm upholds three pointers, i.e., one for both of the two arrays and the other one to preserve the final sorted array's current index.

```
1.  Did you reach the end of the array?
2.      No:
3.          Firstly, start with comparing the current elements of both the arrays.
4.          Next, copy the smaller element into the sorted array.
5.          Lastly, move the pointer of the element containing a smaller element.
6.      Yes:
7.          Simply copy the rest of the elements of the non-empty array
```

Merge( ) Function Explained Step-By-Step

Consider the following example of an unsorted array, which we are going to sort with the help of the Merge Sort algorithm.

A= (36,25,40,2,7,80,15)

Step1: The merge sort algorithm iteratively divides an array into equal halves until we achieve an atomic value. In case if there are an odd number of elements in an array, then one of the halves will have more elements than the other half.

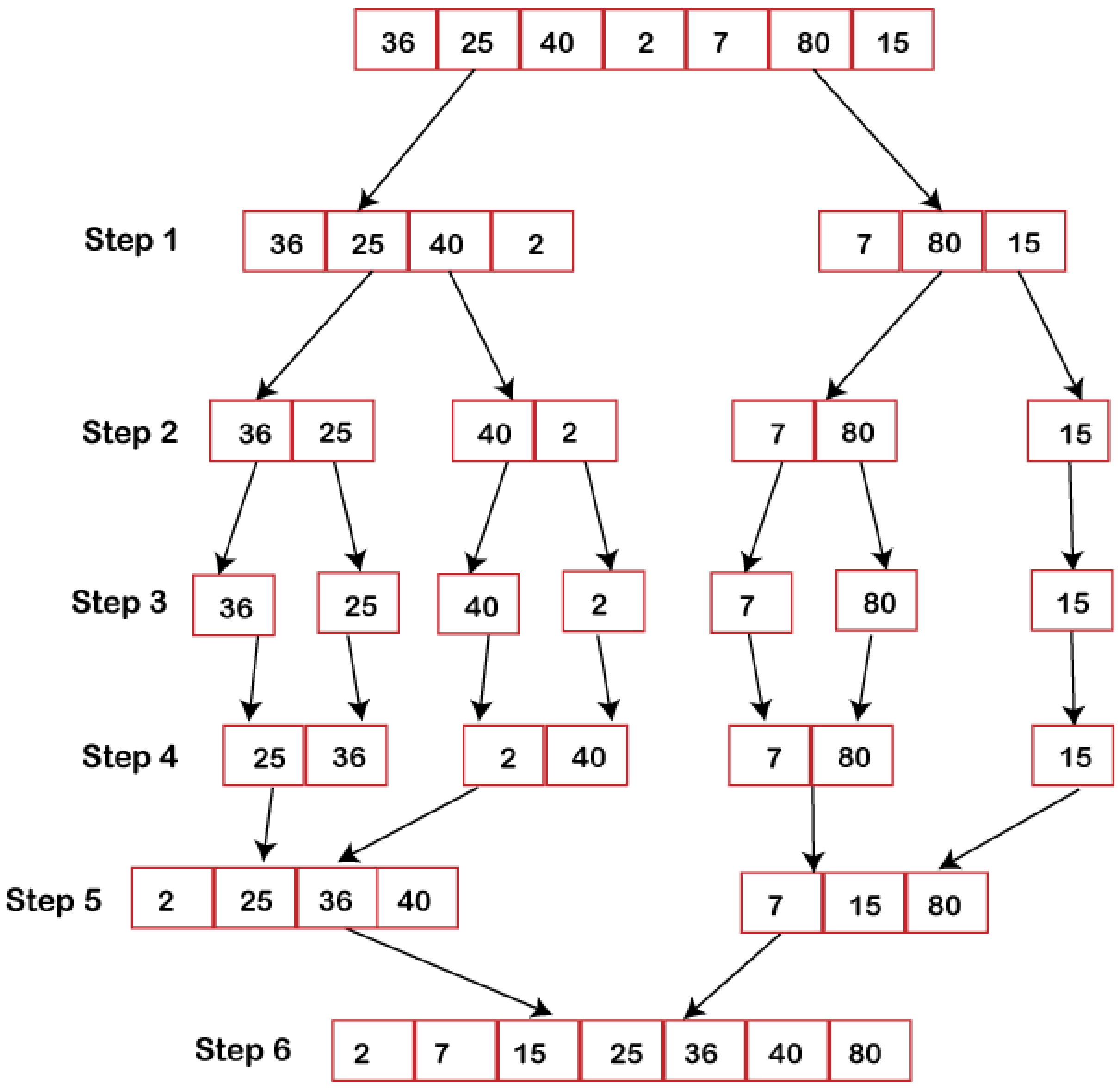
Step2: After dividing an array into two subarrays, we will notice that it did not hamper the order of elements as they were in the original array. After now, we will further divide these two arrays into other halves.

Step3: Again, we will divide these arrays until we achieve an atomic value, i.e., a value that cannot be further divided.

Step4: Next, we will merge them back in the same way as they were broken down.

Step5: For each list, we will first compare the element and then combine them to form a new sorted list.

Step6: In the next iteration, we will compare the lists of two data values and merge them back into a list of found data values, all placed in a sorted manner.



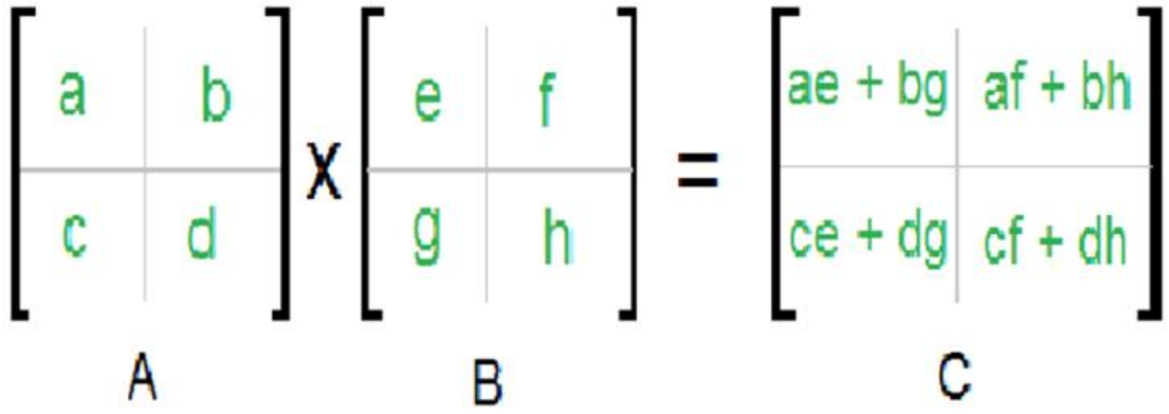
Hence the array is sorted.

11.Strassen Matrix Multiplication:

Strassen's Matrix Multiplication is the divide and conquer approach to solve the matrix multiplication problems. The usual matrix multiplication method multiplies each row with each column to achieve the product matrix. The time complexity taken by this approach is **$O(n^3)$**, since it takes two loops to multiply. Strassen's method was introduced to reduce the time complexity from **$O(n^3)$** to **$O(n^{\log 7})$**

Matrix multiplication is based on a divide and conquer-based approach. Here we divide our matrix into a smaller square matrix, solve that smaller square matrix and merge into larger results. For larger matrices this approach will continue until we recurse all the smaller sub matrices.
Suppose we have two matrices, A and B, and we want to multiply them to form a new matrix, C.

C=AB, where all A,B,C are square matrices. We will divide these larger matrices into smaller sub matrices n/2; this will go on.



Now from above we see:
r=ae+bg
s=af+bh
t=ce+dg
u=cf+dh

Each of the above four equations satisfies two multiplications of n/2Xn/2 matrices and addition of their n/2xn/2 products. Using these equations to define a divide and conquer strategy we can get the relation among them as:

T(N) = 8T(N/2) + O(N2)
From the above we see that simple matrix multiplication takes eight recursion calls.
T(n)=O(n^3)
Thus, this method is faster than the ordinary one.

It takes only seven recursive calls, multiplication of n/2xn/2 matrices and O(n^2) scalar additions and subtractions, giving the below recurrence relations.

T(N) = 7T(N/2) + O(N2)

Steps of Strassen's matrix multiplication:

1. Divide the matrices A and B into smaller submatrices of the size n/2xn/2.
2. Using the formula of scalar additions and subtractions compute smaller matrices of size n/2.
3. Recursively compute the seven matrix products Pi=AiBi for i=1,2,…7.
4. Now compute the r,s,t,u submatrices by just adding the scalars obtained from above points.

Submatrix Products:

We have read many times how two matrices are multiplied. We do not exactly know why we take the row of one matrix A and column of the other matrix and multiply each by the below formula.

Pi=AiBi

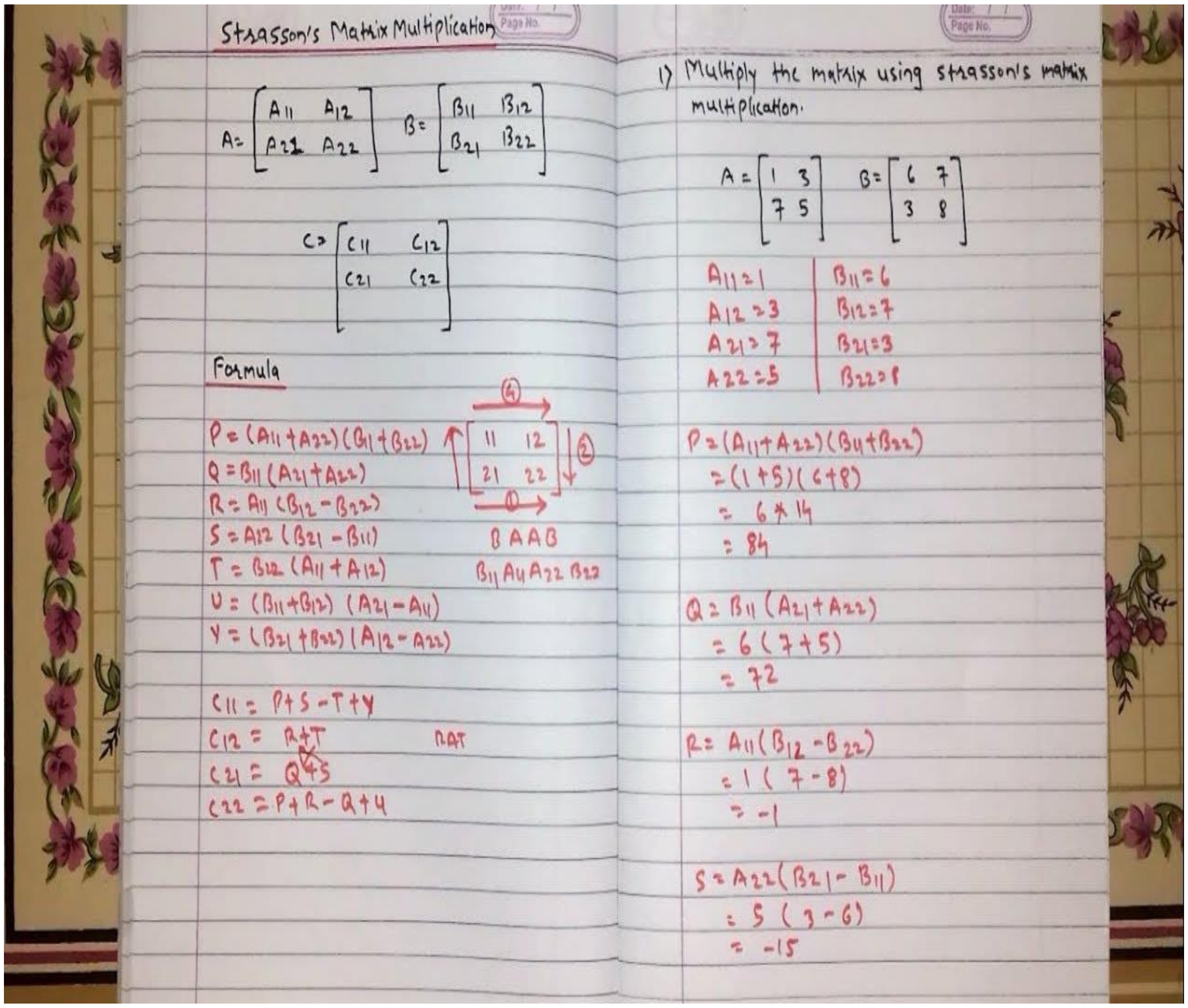
=(α1ia+α2ib+α3ic)(β1ie+β2if+β2ig)

Where a,b ,β,α are the coefficients of the matrix that we see here, the product is obtained by just adding and subtracting the scalar.

$$p1 = a(f - h) \qquad p2 = (a + b)h$$
$$p3 = (c + d)e \qquad p4 = d(g - e)$$
$$p5 = (a + d)(e + h) \qquad p6 = (b - d)(g + h)$$
$$p7 = (a - c)(e + f)$$

The A x B can be calculated using above seven multiplications.
Following are values of four sub-matrices of result C

$$\underbrace{\begin{bmatrix} a & b \\ c & d \end{bmatrix}}_{A} \times \underbrace{\begin{bmatrix} e & f \\ g & h \end{bmatrix}}_{B} = \underbrace{\begin{bmatrix} p5 + p4 - p2 + p6 & p1 + p2 \\ p3 + p4 & p1 + p5 - p3 - p7 \end{bmatrix}}_{C}$$



Strasson's Matrix Multiplication

$A = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix}$ $B = \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix}$

$C = \begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix}$

Formula

$P = (A_{11} + A_{22})(B_{11} + B_{22})$
$Q = B_{11}(A_{21} + A_{22})$
$R = A_{11}(B_{12} - B_{22})$
$S = A_{22}(B_{21} - B_{11})$
$T = B_{22}(A_{11} + A_{12})$
$U = (B_{11} + B_{12})(A_{21} - A_{11})$
$Y = (B_{21} + B_{22})(A_{12} - A_{22})$

$$\begin{tikzcd} 11 \arrow[r, "4"] & 12 \arrow[d, "2"] \\ 21 \arrow[u] \arrow[r, "1"'] & 22 \end{tikzcd}$$

B A A B
$B_{11}\ A_{11}\ A_{22}\ B_{22}$

$C_{11} = P + S - T + Y$
$C_{12} = R + T$ RAT
$C_{21} = Q + S$
$C_{22} = P + R - Q + U$

1) Multiply the matrix using strasson's matrix multiplication.

$A = \begin{bmatrix} 1 & 3 \\ 7 & 5 \end{bmatrix}$ $B = \begin{bmatrix} 6 & 7 \\ 3 & 8 \end{bmatrix}$

$A_{11} = 1$	$B_{11} = 6$
$A_{12} = 3$	$B_{12} = 7$
$A_{21} = 7$	$B_{21} = 3$
$A_{22} = 5$	$B_{22} = 8$

$P = (A_{11} + A_{22})(B_{11} + B_{22})$
$= (1 + 5)(6 + 8)$
$= 6 * 14$
$= 84$

$Q = B_{11}(A_{21} + A_{22})$
$= 6(7 + 5)$
$= 72$

$R = A_{11}(B_{12} - B_{22})$
$= 1(7 - 8)$
$= -1$

$S = A_{22}(B_{21} - B_{11})$
$= 5(3 - 6)$
$= -15$