

UNIT- I

Informal Definition:

An Algorithm is any well-defined computational procedure that takes some value or set of values as Input and produces a set of values or some value as output. Thus algorithm is a sequence of computational steps that transforms the I/P into the O/P.

Algorithm Definition : An Algorithm is a finite set of instructions that, if followed, accomplishes a particular task. In addition, all algorithms should satisfy the following criteria.

Properties (or) Characterstics of Algorithm

An algorithm should posses following properties (or) characterstics –

- 1 : Input :** The algorithm should produce zero or more number of quantities, as an input.
- 2 : Output :** The algorithm should produce at least one quantity, as an output.
- 3 : Definiteness :** Each instruction is clear and unambiguous.
- 4 : Finiteness :** If we trace out the instructions of an algorithm, then for all cases, the algorithm terminates after a finite number of steps.
- 5 : Effectiveness :** Every step of algorithm should be effective. Every operation can be roughly by pen and paper. So tracing of each step of algorithm should be done.

Issues (or) Design of Algorithm

There are various steps for algorithm design.

- 1 : Understanding the problem.
- 2 : Decision making.
- 3 : Specification of algorithm.
- 4 : Verification of algorithm.
- 5 : Analysis of algorithm.
- 6 : Implementation or coding of algorithm.
- 7 : Testing a program.

1 : Understanding the problem : The first thing you need to do before designing an algorithm is to understand problem completely. This is a crucial phase, so we should be very careful. If we did any mistake in this step the entire algorithm becomes wrong. Ask questions if you have doubts about the problem and think about special cases.

2 : Decision Making : After finding the required input set for the given problem we have

to analyze the input and need to decide certain issues such as

A) Capability of computational devices : It is necessary to know the computational capabilities of devices on which the algorithm will be running. Globally we can classify an algorithm from execution point of view as sequential algorithm and parallel algorithm.

Sequential algorithm : specifically runs on the machine in which the instruction are executed one after another. Such machine is called as Random Access Machine (RAM).

Parallel algorithm : are run on the machine in which the instruction are executed in parallel.

B) Choice for either exact or approximate problem solving method : The next important decision is to decide whether the problem is to be solved exactly or approximately. If the problem needs to be solved correctly then we need exact algorithm. Otherwise if the problem is so complex that we won't get the exact solution then we need to choose approximation algorithm.

C) Algorithm Design Techniques : In this we will use different design techniques.

Different types of algorithm design techniques :

- 1 : Brute – force.
- 2 : Divide and Conquer.
- 3 : Dynamic programming.
- 4 : Greedy technique.
- 5 : Back tracking.
- 6 : Branch and Bound

3 : Specification of algorithm : There are various ways we can specify an algorithm.

- 1 : Natural language.
- 2 : Pseudo code.
- 3 : Flowchart.

1 : Natural language : It is very simple to specify an algorithm using natural language. But many times specification of algorithm by using natural language is not clear, and thereby we get brief specification.

Ex : Write an algorithm to perform addition of two numbers.

Step 1 : Read the first number say a.

Step 2 : Read the second number say b.

Step 3 : Add the two numbers and store the result in a variable c.

Step 4 : Display the result.

Such a specification creates difficulty while actually implementing it. Hence many programmers prefer to have specification of algorithm by means of pseudo code.

2:pseudo code: pseudo code is nothing but a combination of natural language and programming language constructs. A pseudo code is usually more precise than a natural language.

Ex : Write an algorithm to perform addition of two numbers.

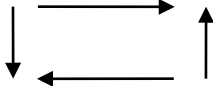
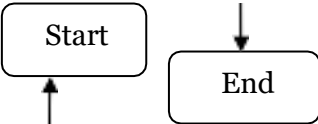
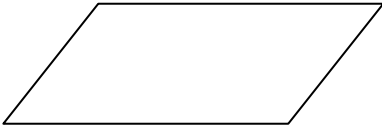

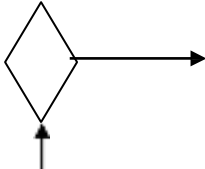
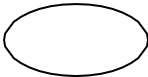
Algorithm sum(a,b)

```
{  
    // Problem description : This algorithm performs addition of two integers.  
    // Input : Two integers a and b.  
    // Output : addition of two integers.  
  
    c <- a + b;  
    write ( c );  
}
```

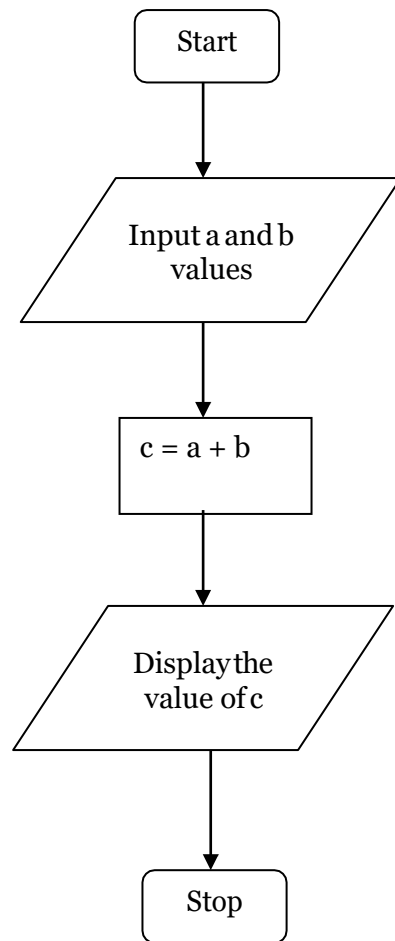
This specification is more useful from implementation point of view.

3 : Flowchart: flowchart is a graphical representation of an algorithm.

Flowchart Symbols

S.NO	Description	Symbols
1	Flowlines : These are the left to right or top to bottom lines connection symbols. These lines shows the flow of control through the program.	
2	Terminal Symbol : The oval shaped symbol always begins and ends the flowchart. Every flow chart starting and ending symbol is terminal symbol.	
3	Input / Output symbol : The parallelogram is used for both input (Read) and Output (Write) is called I/O symbol. This symbol is used to denote any function of an I/O device in the program.	
4	Process Symbol : The rectangle symbol is called process symbol. It is used for calculations and initialization of memory locations	
5	Decision symbol : The diamond shaped symbol is called decision symbol. This box is used for decision making. There will be always two exists from a decision symbol one is labeled YES and other labeled NO.	
6	Connectors : The connector symbol is represented by a circle. Whenever a complex flowchart is morethan one page, in such a situation, the connector symbols are used to connect the flowchart.	

Example :



4 : Verification of algorithm: Verification of algorithm means checking correctness of an algorithm. After specifying an algorithm we go for checking its correctness. We normally check whether the algorithm gives correct output in finite amount of time for a valid set of input. The proof of correctness of an algorithm can be complex sometimes. A common method of proving the correctness of an algorithm is by using mathematical induction. But to show that an algorithm works incorrectly we have to show that at least for one instance of valid input the algorithm gives wrong result.

5 : Analysis of algorithm: Analyzing an algorithm we should consider following factors.

- 1 : Time Complexity.
- 2 : Space Complexity.
- 3 : Simplicity.
- 4 : Generality.
- 5 : Range of input.

1 : Time Complexity: The Time Complexity can be defined as amount of computer time required by an algorithm to run to completion. By computing time complexity we can analyze whether an algorithm requires slow or fast.

2 : Space Complexity: The Space Complexity can be defined as amount of space required by an algorithm to run to completion. By computing space complexity we can analyze whether an algorithm requires more or less space.

3 : Simplicity: Simplicity of an algorithm means generating sequence of instructions which are easy to understand. This is an important characteristic of an algorithm. Because simple algorithms can be understood quickly and one can then write simpler programs for such algorithms.

4 : Generality: Generality shows that sometimes it becomes easier to design an algorithm in more general way rather than designing it for particular set of input. Hence we should write general algorithms always.

5 : Range of input: Range of inputs comes in picture when we execute an algorithm. The design of algorithm should be handle the range of input which is the most natural to corresponding problem.

Analysis of algorithm means checking the characteristics such as : Time complexity, Space complexity, simplicity, generality and range of input. If these factors are not satisfactory then we must redesign the algorithm.

6 : Implementation or coding of algorithm: The implementation of an algorithm is done by suitable programming language. For example: if an algorithm consists of object and related methods then it will be better to implement such algorithm using some object-oriented programming language like C++ and JAVA. While writing a program for given algorithm it is essential to write an optimized code. This will reduce the burden on computer.

7 : Testing a program: Testing a program is an activity carried out to expose as many errors as possible and to correct them. There are two phases for testing a program.

1 : Debugging.

2: Profiling.

1 : Debugging: Debugging is a technique in which a sample set of data is tested to see whether faulty results occur or not. If any faulty result occurs then those results are corrected.

But in debugging technique only presence of error is pointed out. Any hidden error cannot be identified. Thus in debugging **we cannot verify correctness** of output on sample data. Hence profiling concept is introduced.

2 : Profiling: Profiling is the process of executing a correct program on a sample set of data. Then the time and space required by the program to execute is measured.

PSEUDO CODE FOR EXPRESSING ALGORITHM: Hypothetical language is used to define the algorithm is called pseudo code. It is not a programming language because it is not having any compiler. It is used to represent the algorithm.

The following rules are involved to represent the algorithm by using pseudo code

1 : Algorithm is a procedure which consisting of heading and body. The heading consisting of keyword **Algorithm** and name of the algorithm and parameter list.

Syntax: Algorithm name (p1, p2,, pn)

This keyword should be written first **name of an Algorithm** **write parameters (if any)**

2 : The heading section we should write following things :

```
// Problem Description:  
// Input :  
// Output :
```

3 : The body of an algorithm is written, in which various programming constructs like if, for, while or some assignment statements may be written.

4 : The beginning and end of block should be indicated by { and } respectively. The compound statements should be enclosed within { and } brackets.

5 : The delimiters; are used at the end of each statement.

6 : Single line comments are written using // as beginning of comment.

7 : The identifier should begin by letter and not by digit. An identifier can be a combination of alphanumeric string.

It is not necessary to write data types explicitly for identifiers. It will be represented by the context itself. Basic data types used are integer, float, char, Boolean and so on.

The pointer type is also used to point memory location.

8 : The input and output can be done by **read** and **write**

```
Ex: write ("This message will be displayed on console");  
      read(val);
```

9 : Boolean operators such as **True or False.**

Logical operators such as **and, or, not.**

Relational operators such as **<, <=, >, >=, ==, #.**

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

Syntax : variable := expression;

Ex : a := 10;

11 : The array indices are stored with in square brackets '[' '']'. The index of array usually starts at zero. The multidimensional arrays can also be used in algorithm.

12 : The conditional statements such as **if-then or if-then-else** are written in following form :

Syntax: if(condition) **then**
 statement 1;

 if(condition) **then**
 statement 1;
 else
 statement 2;

if the if-then statement is of compound type then { and } should be used for enclosing block.

13 : while statement can be written as :

```
while(condition) do  
{  
    statement 1  
    statement 2  
    :  
    :  
    statement n  
}
```

While the condition is true the block enclosed with { } gets executed otherwise statement after } will be executed.

14 : for loop can be written as :

```
for variable ← value1 to valuen do  
{  
    statement 1;  
    statement 2;  
    :  
    :  
    statement n;  
}
```


Here **value1** is initialization condition and **value n** is a terminating condition.

Sometime a keyword **step** is used to denote increment or decrement the value of variable.

Ex: for $i \leftarrow 1$ **to** n step 1
 {
 Write(i);
 }

Here variable i is incremented by 1 at each iteration.

15 : The **repeat – until** statement can be written as :

```
repeat
{
    statement 1;
    statement 2;
    :
    :
    statement n;
}
until(condition)
```

16 : The break statement is used to exit from inner loop. The return statement is used to return control from one point to another. Generally used while exiting from function.

Note that statements in an algorithm executes in sequential order i.e, in the same order as they appear-one after the other.

PERFORMANCE ANALYSIS: Performance Analysis refers to the task of determining the computing time and storage space requirements of an algorithm. It is also known as performance analysis.

The main purpose of algorithm analysis is to design most efficient algorithms. The efficiency of the algorithm mainly depends on two factors

1 : Space Complexity

2 : Time Complexity

1 : Space Complexity : The Space Complexity can be defined as amount of space required by an algorithm to run to completion. By computing space complexity we can analysis the algorithm requires less or more space.

The space needed by an algorithm consists of the following components.

1 : The fixed static part that is independent of the characteristics (example number size) of the input and outputs. This part includes the instruction space, space for simple variable and fixed size component variables, space for constants etc..

2 : A variable dynamic part that consists of the space needed by component variable whose size is dependent on the particular problem instance at run time being solved, the space needed by referenced variables and the recursion stack space.

The space complexity requirement $S(p)$ can be given as :

$$S(p) = C + Sp$$

Where **C** is a constant i.e, fixed part. It denotes amount of space required for inputs, outputs, variables and instructions. And **Sp** is a instance characteristics. This is a variable part amount of space required for problem is depends on particular problem instance.

Consider the examples of algorithms to compute the space complexity

Ex 1 : Algorithm add(a,b,c)

```
{
    // Problem Description : This algorithm computes the addition of 3 numbers
    // Input : a, b and c are of floting type.
    // Output : The addition is returned.

    return a+b+c;
}
```

The space requirement for algorithm given in

$$S(p) = C$$

If we assume that a, b and c occupy one word size then total size comes to be 3.

2 : Time Complexity : The Time Complexity can be defined as amount of computer time required by an algorithm to run to completion.

They are two types of computing time.

1 : compile time.

2 : run time.

For calculating the time complexity, only run time is considered.

It is difficult to compute the time complexity in terms of physically clocked time. Because runtime depends on many factors such as system load, number of other programs running and processor speed.

For avoiding this, we are using frequency count is basically a count denoting number of times of execution of statement.

The method to determine the step count of an algorithm is to build a table in which we list the total number of steps contributes by each statement.

First determine the number of steps per execution (s/e) of the statement and the total number of times (ie., frequency) each statement is executed.

By combining these two quantities, the total contribution of all statements, the step count for the entire algorithm is obtained.

Example 1 :

Statement	S / E	Frequency	Total Steps
1 Algorithm Sum(a,n)	0	-	0
2. {	0	-	0
3. S=0.0;	1	1	1
4. for I=1 to n do	1	N + 1	N + 1
5. s=s+a[I];	1	N	N
6. return s;	1	1	1
7. }	0	-	0
Total			2N + 3

ASYMPTOTIC NOTATIONS

Expressing the complexity in terms of its relationship to know function. This type of Notation is called Asymptotic Notations.

Various types of notations :

1 : Big Oh Notation ('O')

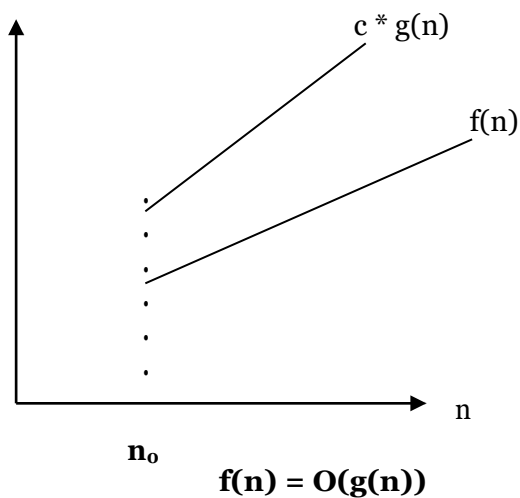
2 : Omega Notation ('Ω')

3 : Theta Notation ('Θ')

1 : Big Oh Notation ('O') :

Definition : Let $f(n)$ and $g(n)$ be two non-negative functions. $f(n)$ is said to be $O(g(n))$ if and only if there exists positive constants 'c' and ' n_0 ' such that $f(n) \leq c * g(n)$ for all non-negative values of n. where $n \geq n_0$.

The following graph the curve for Big Oh Notation.



This notation provides an upper bound for the function i.e, the function $g(n)$ is an upper bound on the value of $f(n)$ for all n, where $n \geq n_0$.

1 : Show that $f(n) = 8n + 128 = O(n^2)$.

To Prove.

$$f(n) = 8n + 128 = O(n^2)$$

Proof: If $f(n) = O(g(n))$. We have there exists positive constants ' c ' and ' n_0 ' such that $f(n) \leq c * g(n)$ for all non-negative values of n . where $n \geq n_0$.

Given problem in the form of $f(n) = O(g(n))$.

Here

$$f(n) = 8n + 128$$

$$g(n) = n^2$$

$$\text{Now } 8n + 128 \leq 3 * n^2$$

Now $N = 1$.

$$8.1 + 128 \leq 3 * 1 \text{ (false)}$$

Now take $N=2$

$$8.2 + 128 \leq 3 * 4 \text{ (false)}$$

Now take $N=3$

$$8.3 + 128 \leq 3 * 9 \text{ (false)}$$

Now take $N=4$

$$8.4 + 128 \leq 3 * 16 \text{ (false)}$$

Now take $N=5$

$$8.5 + 128 \leq 3 * 25 \text{ (false)}$$

Now take $N=6$

$$8.6 + 128 \leq 3 * 36 \text{ (false)}$$

Now take $N=7$

$$8.7 + 128 \leq 3 * 49 \text{ (false)}$$

Now take $N=8$

$$8.8 + 128 \leq 3 * 64 \text{ (true)}$$

There exists two positive constants $c = 3$, $n_0 = 8$ such that $8n + 128 \leq 3 * n^2$ for all $n \geq 8$.

$$\text{Therefore } 8n + 128 = O(n^2)$$

2: Show that $f(n) = 12n^2 + 6n = O(n^3)$

To Prove.

$$f(n) = 12n^2 + 6n = O(n^3)$$

Proof: If $f(n) = O(g(n))$. We have there exists positive constants ' c ' and ' n_0 ' such that $f(n) \leq c * g(n)$ for all non-negative values of n . where $n \geq n_0$.

Given problem in the form of $f(n) = O(g(n))$.

Here ,

$$f(n) = 12n^2 + 6n$$

$$g(n) = n^3$$

$$\text{Now } f(n) = 12n^2 + 6n \leq 4 * n^3$$

Now $N = 1$.

$$12.1 + 6.1 \leq 4 * 1 \text{ (false)}$$

Now $N = 2$.

$$12.2 + 6.2 \leq 4 * 8 \text{ (false)}$$

Now $N = 3$.

$$12.3 + 6.3 \leq 4 * 27 \text{ (True)}$$

There exists two positive constants $c = 4$, $n_0 = 3$ such that $12n^2 + 6n \leq 4 * n^3$ for all $n \geq 3$.

$$\text{Therefore } 12n^2 + 6n = O(n^3)$$

3: consider function $f(n) = 2n + 2$ and $g(n) = n^2$.

Proof: If $f(n) = O(g(n))$. We have there exists positive constants ' c ' and ' n_0 ' such that $f(n) \leq c * g(n)$ for all non-negative values of n . where $n \geq n_0$.

Given,

$$f(n) = 2n + 2.$$

$$g(n) = n^2$$

$$\text{Now } f(n) = 2n + 2 \leq 2 * n^2$$

Now Take $N = 1$.

$$2 \cdot 1 + 2 \leq 2 \cdot 1 \text{ (false)}$$

Now Take $N = 2$.

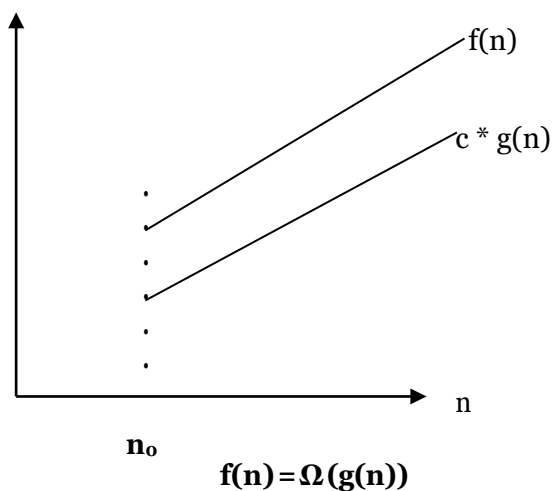
$$2 \cdot 2 + 2 \leq 2 \cdot 4 \text{ (True)}.$$

There exists two positive constants $c = 2$, $n_0 = 2$ such that $2n + 2 \leq 2 \cdot n^2$ for all $n \geq 2$.

2 : Omega Notation (Ω) :

Definition: Let $f(n)$ and $g(n)$ be two non-negative functions. $f(n)$ is said to be $\Omega(g(n))$ if and only if there exists positive constants ' c ' and ' n_0 ' such that $f(n) \geq c \cdot g(n)$ for all non-negative values of n , where $n \geq n_0$.

The following graph the curve for Omega Notation.



This notation provides an upper bound for the function i.e, the function $g(n)$ is an lower bound on the value of $f(n)$ for all n , where $n \geq n_0$.

1 : Show that $f(n) = 8n + 128 = \Omega(n^2)$.

To Prove.

$$f(n) = 8n + 128 = \Omega(n^2)$$

Proof: If $f(n) = O(g(n))$. We have there exists positive constants ' c ' and ' n_0 ' such that $f(n) \geq c * g(n)$ for all non-negative values of n . where $n \geq n_0$.

Given problem in the form of $f(n) = O(g(n))$.

Here

$$f(n) = 8n + 128$$

$$g(n) = n^2$$

$$\text{Now } 8n + 128 \geq 1 * n^2$$

Now $N = 1$.

$$8.1 + 128 \geq 1 * 1 \text{ (TRUE)}$$

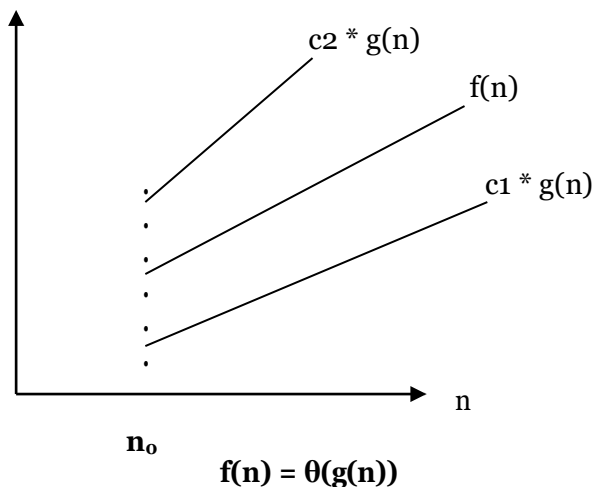
There exists two positive constants $c = 1$, $n_0 = 1$ such that $8n + 128 \leq 1 * n^2$ for all $n > 1$.

$$\text{Therefore } 8n + 128 = O(n^2)$$

3 : Theta Notation (θ):

Definition : Let $f(n)$ and $g(n)$ be two non-negative functions. $f(n)$ is said to be $\theta(g(n))$ if and only if there exists positive constants ' c_1 ', ' c_2 ' and ' n_0 ' such that $c_1 * g(n) \leq f(n) \leq c_2 * g(n)$ for all non-negative values of n . where $n \geq n_0$.

The following graph the curve for Theta Notation.



This notation provides both lower bound and upper bound for the function $f(n)$ i.e.,

$g(n)$ is both lower and upper bound on the value of $f(n)$, for large n .

1 : Show that $f(n) = 8n + 128 = \theta(n^2)$.

To Prove.

$$f(n) = 8n + 128 = O(n^2)$$

Proof :

If $f(n) = O(g(n))$. We have there exists positive constants ' c_1 ', ' c_2 ' and ' n_0 '

such that $c_1g(n) \leq f(n) \leq c_2g(n)$ for all non-negative values of n . where $n > n_0$.

Given problem in the form of $f(n) = \theta(g(n))$.

Here

$$f(n) = 8n + 128$$

$$g(n) = n^2$$

Now,

$$1 * n^2 \leq 8n + 128 \leq 3 * n^2$$

Now Take $N = 1$

$$1 * 1 \leq 8.1 + 128 \leq 3 * 1 \text{ (False)}$$

Now Take $N = 2$

$$1 * 4 \leq 8.2 + 128 \leq 3 * 4 \text{ (False)}$$

Now Take $N = 3$

$$1 * 9 \leq 8.3 + 128 \leq 3 * 9 \text{ (False)}$$

Now Take $N = 4$

$$1 * 16 \leq 8.4 + 128 \leq 3 * 16 \text{ (False)}$$

Now Take $N = 5$

$$1 * 25 \leq 8.5 + 128 \leq 3 * 25 \text{ (False)}$$

Now Take $N = 6$

$$1 * 36 \leq 8.6 + 128 \leq 3 * 36 \text{ (False)}$$

Now Take N = 7

$$1 * 49 \leq 8.7 + 128 \leq 3 * 49 \text{ (False)}$$

Now Take N = 8

$$1 * 64 \leq 64 + 128 \leq 3 * 64 \text{ (true).}$$

There exists positive constants $c_1 = 1$, $c_2 = 3$, $n_0 = 8$ such that $1 * n^2 \leq 8n + 128 \leq 3 * n^2$

for all $n \geq 8$

4 : Little Oh Notation : The Little Oh Notation is denoted by 'o'. The function $f(n) = o(g(n))$ iff

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

Average of Best Case, Worst Case, Average Case

1 : Best Case : If the algorithm takes the minimum amount of time to run to completion for a specific set of inputs is called Best Case Time Complexity.

Example : While searching a particular element by using sequential search. We get the desired element at starting of the list.

2 : Worst Case : If the algorithm takes maximum amount of time to run to completion for a specific set of inputs is called Worst Case Time Complexity.

Example : While searching an element by using linear search. If we get the desired element at end of the list.

3 : Average Case : If the algorithm takes average amount of time to run to completion for a specific set of inputs is called Average Case.

Example : BinarySearch.

DIVIDE AND CONQUER METHOD

If the given problem is larger than divide the problem into sub problems and again sub problem is large enough then divide the sub problem into smaller sub problems. Find the solution of the sub problems finally we get the solution of the given problems. By combining the solution of sub problem.



Control Abstraction for Divide Conquer Strategy: Control abstraction is flow of control of the procedure. It gives the basic function of the method. So it represents the logic code for divide and conquer strategy.

Algorithm DAndC(p)

```
{
  if small(p) then
    return s(p);
  else
  {
    divide p into smaller instance p1, p2, p3, p4, ....., pk;
    apply DAndC to each of these subproblems;
    return combine(DAndC(p1), DAndC(p2), ....., DAndC(pk));
  }
}
```

P->Problem

Here small(P) is a Boolean value function. If it is true, then the function S is invoked. Otherwise the problem P is divided into smaller subproblems. These problems p1, p2, p3, p4,, pk are solved by recursive applications of DAndC. Combine is a function that determines the solution to p using the solutions to the k sub problems.

Calculate time for DAndC:

If the problem P of size is n and K sub problems size is n1, n2, n3, ---- nk then

$$T(n) = g(n)$$

$$T(n_1) + T(n_2) + \dots + T(n_k) + f(n)$$

T(n) = Time for DAndC of any Input size n;

g(n) = Time to compute the answer directly for small inputs

T(n1) = Time for DAndC of small input size n1;

T(n2) = Time for DAndC of small input size n2;

*

*

T(nk) = Time for DAndC of small input size nk;

f(n) = Time for dividing P and combining the solution of subproblems.

Time complexity of Divide and Conquer algorithms is given by recurrences of the form

$$\begin{aligned} T(n) &= T(1) && \text{if } n=1 \\ &= aT(n/b) + f(n) && \text{if } n>1 \end{aligned}$$

Here a and b are known constants. This is called the **general divide and-conquer recurrence**.

One of the methods for solving any such recurrence relation is called the substitution method. This method repeatedly makes substitution for each occurrence of the function T in the right hand side until all such occurrences disappear.

Example : Consider the case in which a =2 and b=2. Let T(1) = 2 and f(n) = n.

$$\begin{aligned} T(n) &= aT(n/b) + f(n) \\ T(n) &= 2T(n/2) + n \\ &= 2[2T(n/2/2) + n/2] + n \\ &= 2[2T(n/4) + n/2] + n \\ &= 4T(n/4) + \cancel{2n/2} + n \\ &= 4T(n/4) + n + n \\ &= 4T(n/4) + 2n \\ &= 4[2T(n/4/2) + n/4] + 2n \\ &= 4[2T(n/8) + n/4] + 2n \\ &= 8T(n/8) + \cancel{4n/4} + 2n \\ &= 8T(n/8) + n + 2n \\ &= 8T(n/8) + 3n \\ &= 2^3 T(n/2^3) + 3n \end{aligned}$$

⋮

$$\begin{aligned} &= 2^i T(n/2^i) + in \\ &= 2^{\log_2 n} T(1) + \log_2 n \cdot n \\ &= n \cdot 2 + n \cdot \log_2 n \\ &= 2n + n \log_2 n \end{aligned}$$

In particular

$$n = 2^i$$

$$i = \log_2 n$$

$$a^{\log_2 n} = n$$

The applications of Divide and Conquer Strategy are

- 1 : Binary Search.
- 2: QuickSort.
- 3: Merge Sort.
- 4: Strassen's Matrix Multiplication.

1 : BINARY SEARCH : In binary search method we are using the divide and conquer strategy. In this method first sort the elements in ascending order. An element which is to be searched from the list of elements stored in array $A[0 \dots n-1]$ is called KEY element.

Let $A[mid]$ be the mid element of array A. Then there are three conditions that need to be tested while searching the array using this method.

- 1 : if $KEY == A[mid]$ then desired element is present in the list.
- 2 : Otherwise if $KEY < A[mid]$ then search the left sub list.
- 3 : Otherwise if $KEY > A[mid]$ then search the right sub list.

Algorithm : Binary Search Non – Recursive Method

Algorithm Bin-search($A[0 \dots n-1]$, KEY)

```
{
    low = 0;
    high = n-1;

    while ( low < high ) do
    {
        mid = (low + high) / 2;

        if( KEY == A[mid]) then
            return mid          // desired element is present in the list.

        else if ( KEY < A[mid] )then
            high = mid- 1;      // Search the left sub list

        else if ( KEY > A[mid])then
            low = mid + 1;      // Search the right sub list
    }

    return -1; //if element is not present in the list
}
```

Example : Consider a list of elements stored in array A as

10 , 20 , 30, 40, 50, 60, 70

Solution:

Diagram illustrating a 7-element array (indices 0 to 6) with values 10, 20, 30, 40, 50, 60, 70. The value 40 at index 3 is highlighted in red. Arrows indicate the 'Low' and 'High' pointers, both pointing to the first element (index 0).

The **Key element** (i.e the element to be searched) is 60.

Now to obtain middle element we will apply formula :

$$\begin{aligned}\text{mid} &= (\text{low} + \text{high}) / 2 \\ &= (0 + 6) / 2 \\ &= (6 / 2) = 3\end{aligned}$$

Then check $key == A[mid]$
i.e, $60 == A[3]$ **No**

Then check $\text{key} < A[\text{mid}]$
i.e, $60 < 40$ **No**

```
Then check  $key > A[mid]$   

       60 > 40 TRUE
```

-
- .. Search the right sublist.

Diagram illustrating the partitioning process:

0	1	2	3	4	5	6
10	20	30	40	50	60	70

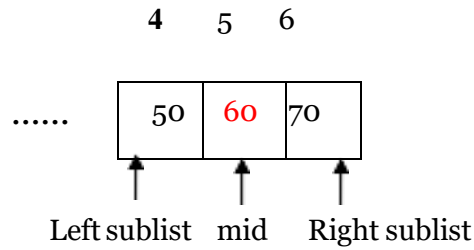
Labels below the array:

- Left sublist (points to index 1)
- mid (points to index 3)
- Right sublist (points to index 5)

The right sublist is

.....	50	60	70
-------	----	----	----

Now we will again divide this list and check the mid element.



Now to obtain middle element we will apply formula :

$$\begin{aligned} \text{mid} &= (\text{low} + \text{high}) / 2 \\ &= (4 + 6) / 2 \\ &= (10 / 2) = 5 \end{aligned}$$

Then check $\text{key} == \text{A}[\text{mid}]$
i.e, $60 == \text{A}[5]$ **Yes**

∴ i.e, The number is present in the list.

Algorithm : Binary Search Recursive Method

Algorithm RBin-search (a, n, key)

```
{
    low = 1;
    high = n;
    while ( low < high )
    {
        mid = low + high / 2;
        if ( key == a[mid] )
            return mid;
        else if ( key < a[mid] )
            RBin-search ( a, key, low, mid - 1 );
        else if ( key > a[mid] )
            RBin-search( a, key, mid+1, high );
    }
}
```


Time Complexity of Binary Search Method : Time complexity of Binary Search method is described by the recurrence relation is

$$\begin{array}{ll} T(1) & n=1 \\ T(n) = T(n/2)+1 & n>1 \end{array}$$

$$\begin{aligned} T(n) &= T(n/2)+1 \\ &= [T(n/2/2)+1]+1 \\ &= T(n/4)+1+1 \\ &= T(n/4)+2 \\ &= [T(n/4/2)+1]+2 \\ &= T(n/8)+1+2 \\ &= T(n/8)+3 \\ &= T(n/2^3) + 3 \end{aligned}$$

•
•
•
•

$$\begin{aligned} &= T(n/2^i) + i \\ &= T(1) + \log_2 n \\ &= 1 + \log_2 n \end{aligned}$$

$$T(n) = O(\log_2 n)$$

In particular

$$n = 2^i$$

$$i = \log_2 n$$

Time complexity of Binary Search is : $O(\log n)$

Quick Sort: Quick Sort is a sorting algorithm that uses the divide and conquers strategy. They are three steps of quick sort are as follows.

1 : Split the array into two sub arrays that each element in the left sub array is less than or equal to the middle element and each element in the right sub array is greater than or equal to the middle element. The splitting of the array into two sub arrays is based on pivot element. All the elements that are less than pivot should be in left sub array and all the elements that are greater than pivot should be in right sub array.

2: recursively sort the two sub arrays.

3: combine all the sorted elements in a group to form a list of sorted element.

Algorithm for QuickSort :

Algorithm : Quick(A[0...n-1], low, high)

```
{
    if ( low < high ) then
        m := partition(A[ low..... high]); //Split the array into two sub arrays.
        Quick(a[low...m-1]); leftsublist
        Quick(m+1....high]); righ sublist
}
```

Algorithm: partition (A [low...high]) partition

```
{
    pivot=A[low];
    i = low;
    j = high;
    while ( i <= j) do
    {
        while (A[i]<= pivot ) && ( i < j ) do
            i = i + 1;

        while (A[j]>= pivot) do
            j = j - 1;

        if ( i <= j) then
            swap ( A[i],A[j] )    //swap A[i] and A[j]
    }

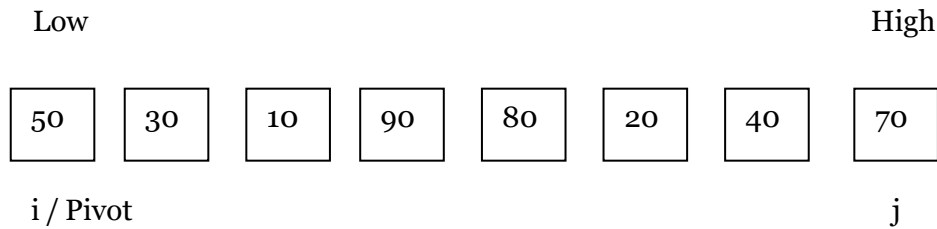
    swap ( A[low],A[j])    //swap A[low] and A[j]

    return j;
}
```

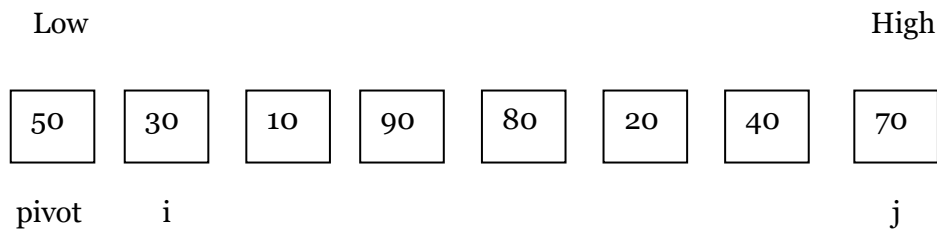
Example : Consider a list of elements stored in array A as

50, 30, 10, 90, 80, 20, 40, 70

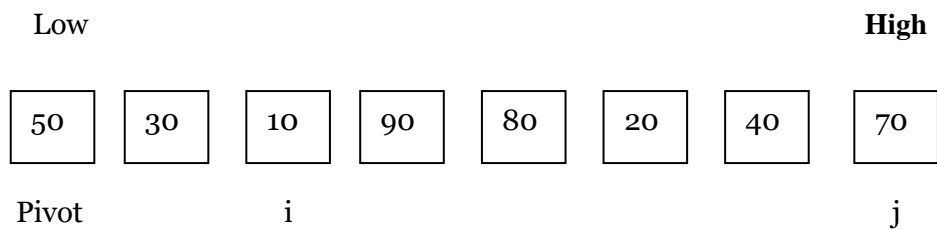
Step 1 : split the array in two parts. The left sublist will contain the elements less than pivot and right sublist contains elements greater than pivot.



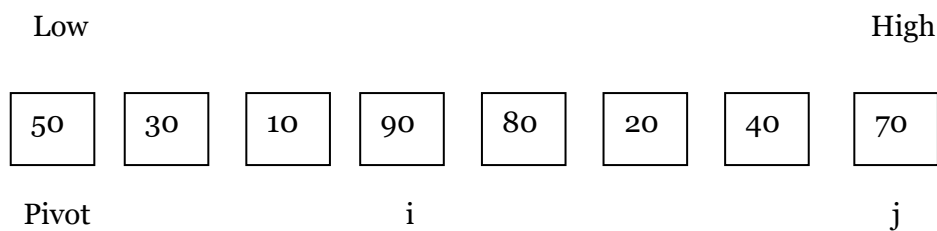
Step 2 : if $A[i] \leq \text{pivot} \ \&\& \ (i < j)$ ($50 \leq 50$) $\&\& (0 < 7)$ – condition true, we will increment i.



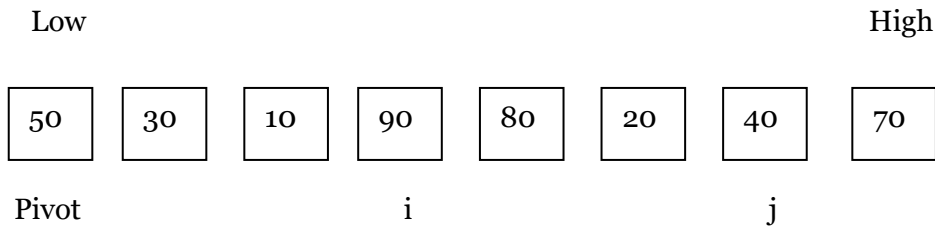
Step 3 : if $A[i] \leq \text{pivot} \ \&\& \ (i < j)$ ($30 \leq 50$) $\&\& (1 < 7)$ – condition true, we will increment i.



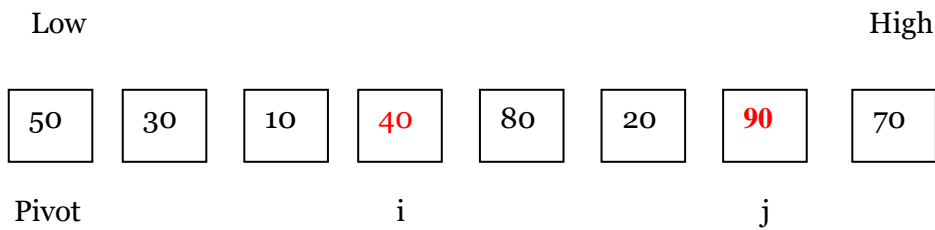
Step 4 : if $A[i] \leq \text{pivot} \ \&\& \ (i < j)$ ($10 \leq 50$) $\&\& (2 < 7)$ – condition true, we will increment i.



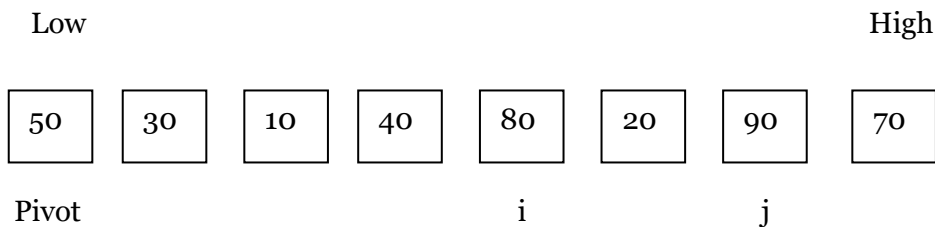
Step 5 : if $A[i] \leq \text{pivot} \ \&\& \ (i < j) \ (90 \leq 50) \ \&\& (3 < 7)$ – condition false, if $A[j] \geq \text{pivot} \ (70 \geq 50)$ – condition true, we will decrement j .



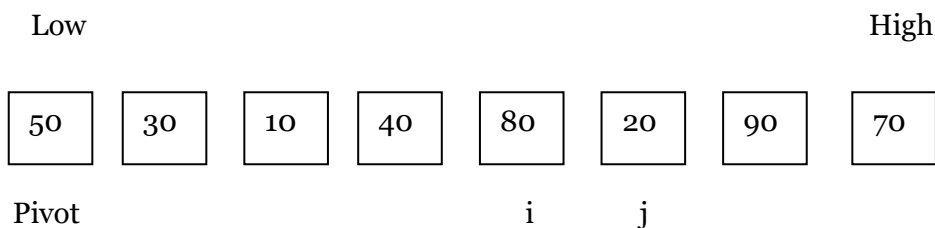
Step 6 : if $A[j] \geq \text{pivot} \ (40 \geq 50)$ – condition false, if $i \leq j \ (3 \leq 6)$ – true, we will swap $A[i]$ and $A[j]$ i.e , 90 and 40.



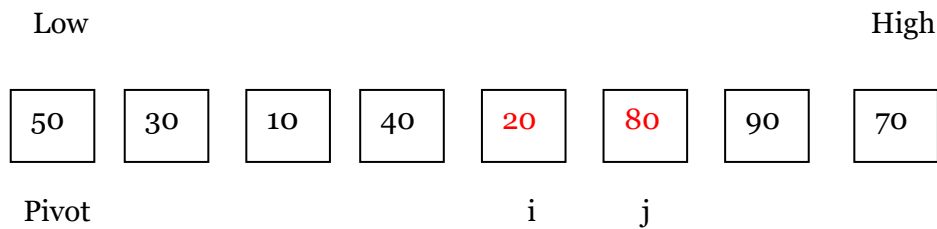
Step 7 : if $A[i] \leq \text{pivot} \ \&\& \ (i < j) \ (40 \leq 50) - \&\& (3 < 6)$ – condition true, we will increment i .



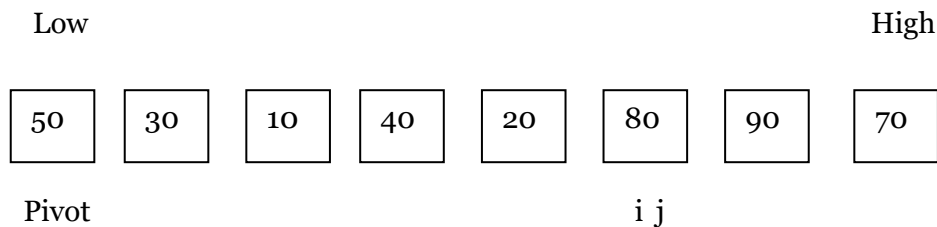
Step 8 : if $A[i] \leq \text{pivot} \ \&\& \ (i < j) \ (80 \leq 50) \ \&\& (4 < 6)$ – condition false, if $A[j] \geq \text{pivot} \ (90 \geq 50)$ – condition true, we will decrement j .



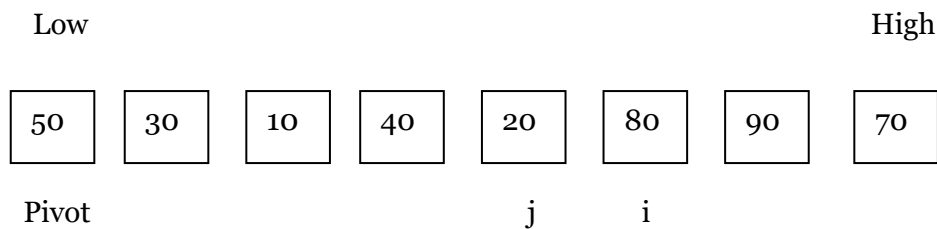
Step 9 : if $A[j] \geq \text{pivot}$ ($20 \geq 50$) – condition false, if $(i \leq j)$ ($4 \leq 5$)- condition true, we will swap $A[i]$ and $A[j]$. i.e, 80 and 20.



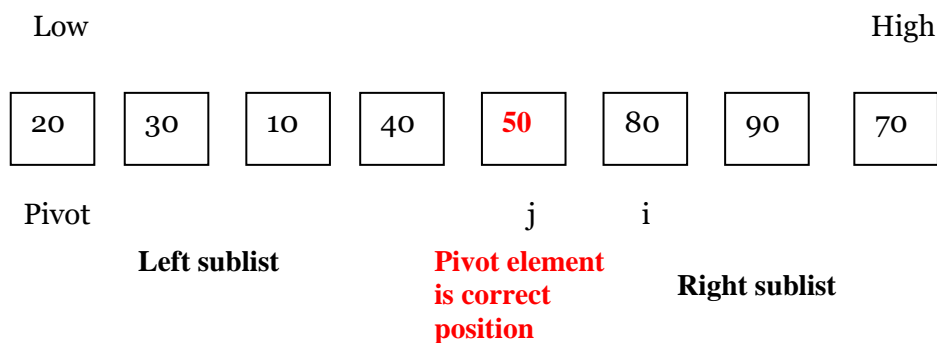
Step 10 : if $A[i] \leq \text{pivot} \ \&\& \ (i < j)$ ($20 \leq 50$) $\&\& (4 < 5)$ – condition true, we will increment i.



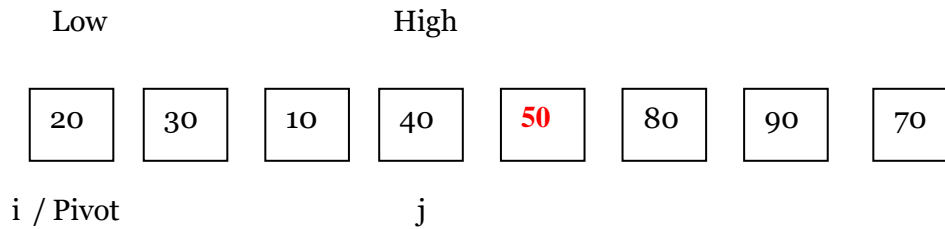
Step 11 : if $A[i] \leq \text{pivot}$ ($80 \leq 50$) $\&\& (5 < 5)$ – condition false, if $A[j] \geq \text{pivot}$ ($80 \geq 50$) – condition true, we will decrement j.



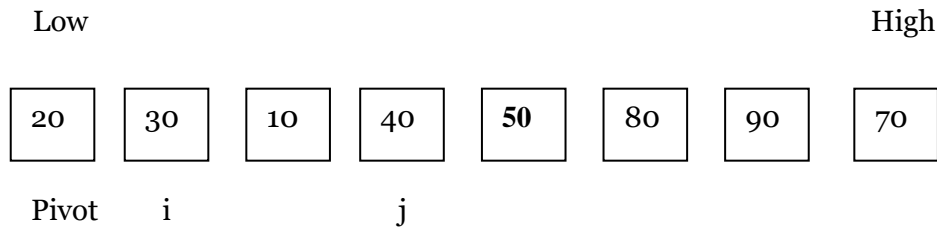
Step 12 : if $A[j] \geq \text{pivot}$ ($20 \geq 50$) – condition false, if $(i \leq j)$ ($5 \leq 4$)- condition false, we will swap $A[\text{low}]$ and $A[j]$. i.e, 50 and 20.



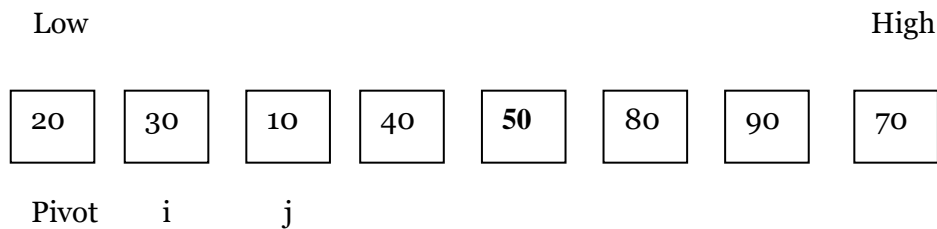
Step 13 : Now we will start Left sub list.



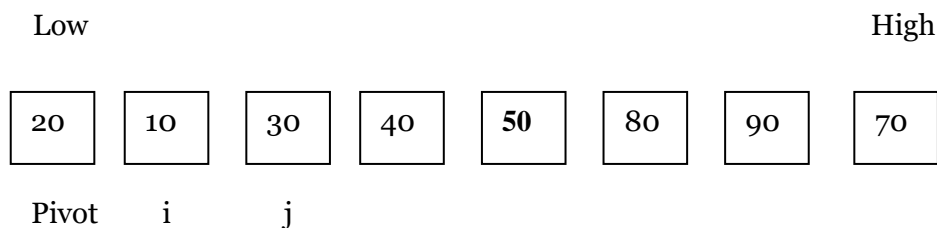
Step 14 : If $A[i] \leq \text{pivot} \ \&\& \ (i < j)$ ($20 \leq 20$) $\&\& \ (0 < 3)$ -- condition true, we will increment i.



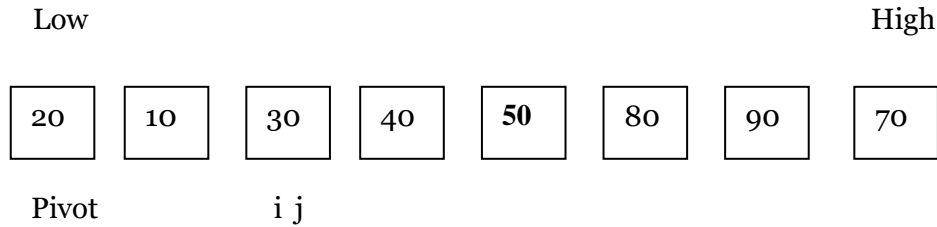
Step 15 : If $A[j] \leq \text{pivot} \ \&\& \ (i < j)$, ($30 \leq 20$) $\&\& \ (1 < 3)$ -- condition false, $A[j] \geq \text{pivot}$, ($40 \geq 20$) - condition true, we will decrement j.



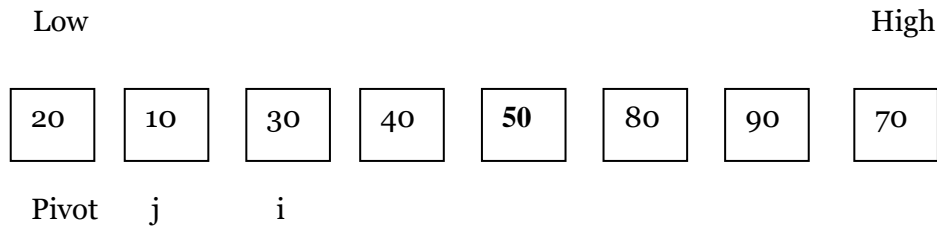
Step 16 : $A[j] \geq \text{pivot}$, ($10 \geq 20$) - condition false, if $(i < j)$, ($1 < 2$) - condition true, we will swap $A[i]$ and $A[j]$, i.e, 30 and 10.



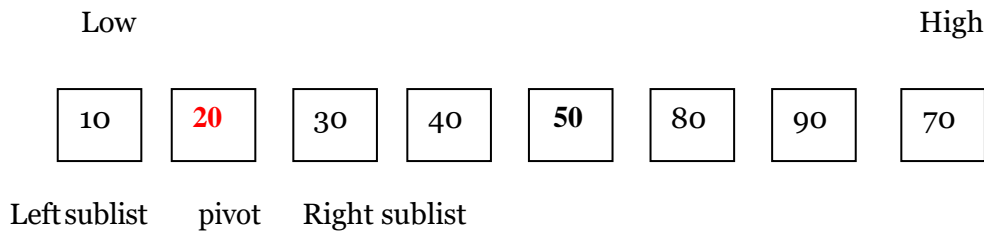
Step 17 : $A[i] \leq \text{pivot} \ \&\& (i < j)$ ($10 \leq 20$) $\&\& (1 < 2)$ -- condition true, we will increment i.



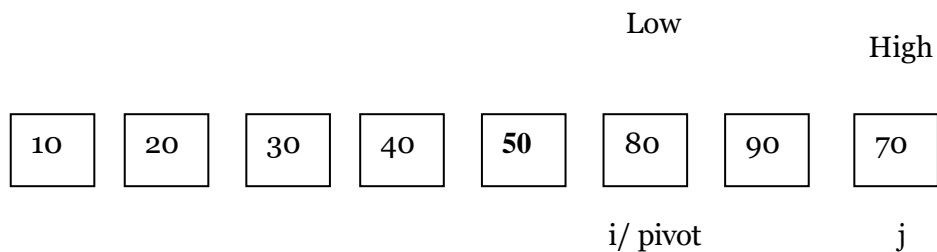
Step 18 : $A[i] \leq \text{pivot} \ \&\& (i < j)$ ($30 \leq 20$) $\&\& (2 < 2)$ -- condition false, $A[j] \geq \text{pivot}$, ($30 \geq 20$) - condition true, we will decrement j.



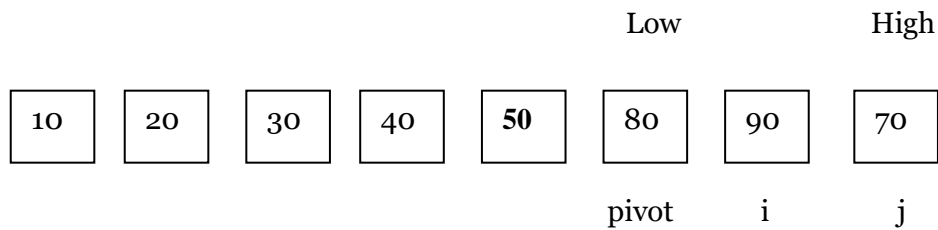
Step 19 : $A[j] \geq \text{pivot}$ ($10 \geq 20$) - condition false, if ($i < j$), ($2 < 1$) - condition false, we will swap $A[\text{low}]$ and $A[j]$, i.e, 20 and 10.



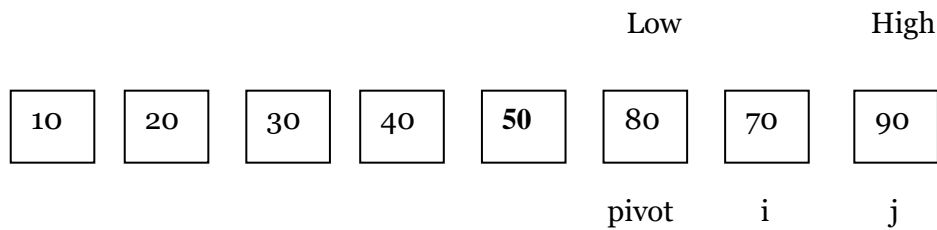
Step 20 : There is one element in left sublist hence **we will start Right sub list.**



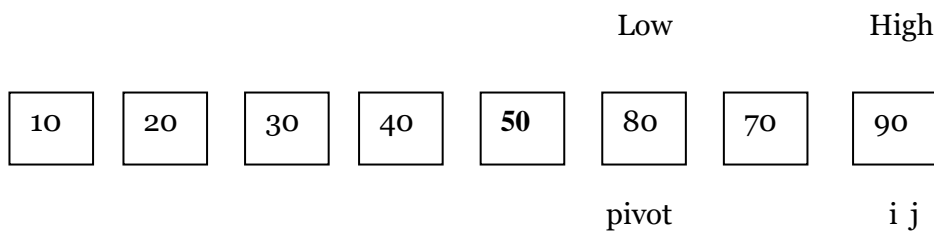
Step 21 : if $A[i] \leq \text{pivot} \ \&\& (i < j)$ ($80 \leq 80$) $\&\& (5 < 7)$ - condition true, we will increment i.



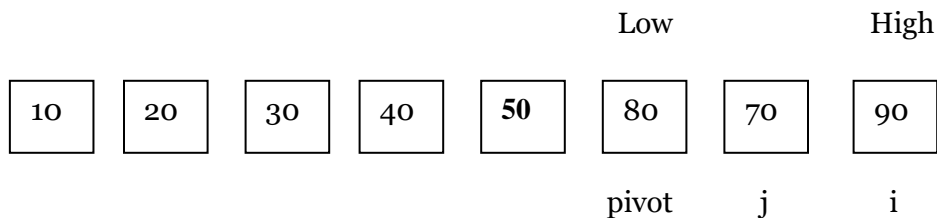
Step 22 : if $A[i] \leq \text{pivot} \ \&\& (i < j)$ ($90 \leq 80$) $\&\& (6 < 7)$ - condition false, if $A[j] \geq \text{pivot}$, ($70 \geq 80$) - condition false, if $(i \leq j)$, ($6 \leq 7$) - condition true, we will swap $A[i]$ and $A[j]$, i.e, 90 and 70.



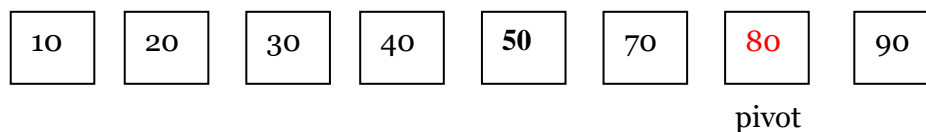
Step 23 : if $A[i] \leq \text{pivot} \ \&\& (i < j)$ ($70 \leq 80$) $\&\& (6 < 7)$ - condition true, we will increment i.



Step 24 : if $A[i] \leq \text{pivot} \ \&\& (i < j)$ ($90 \leq 80$) $\&\& (7 < 7)$ - condition false, if $A[j] \geq \text{pivot}$, ($90 \geq 80$) - condition true, we will decrement j.



Step 25 : if $A[j] \geq \text{pivot}$, ($70 \geq 80$) - condition false, if $(i \leq j)$, ($7 \leq 6$) - condition false, we will swap $A[\text{low}]$ and $A[j]$, 80 and 70.



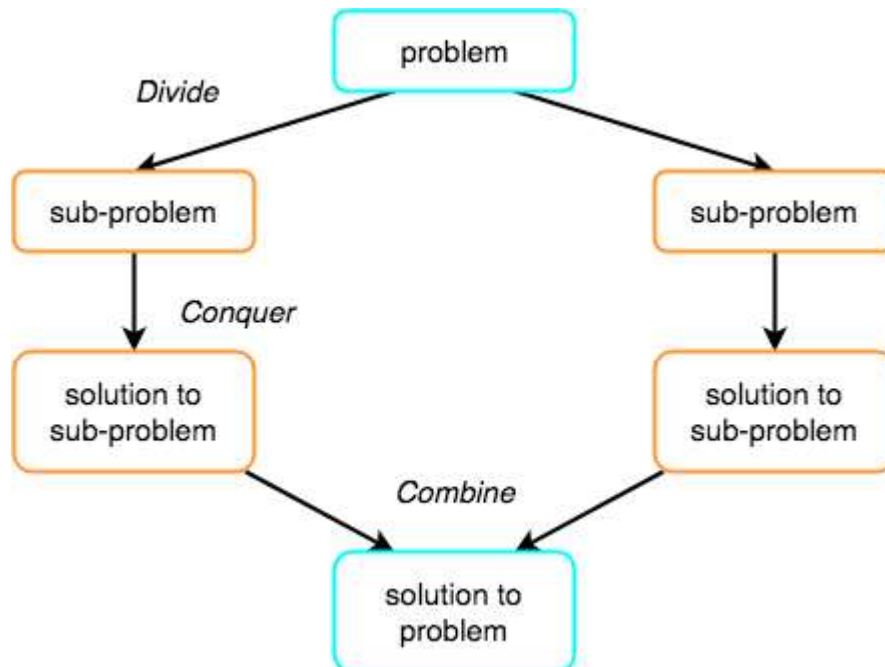
Therefore the sorted list is :

10 20 30 40 50 60 70 80 90

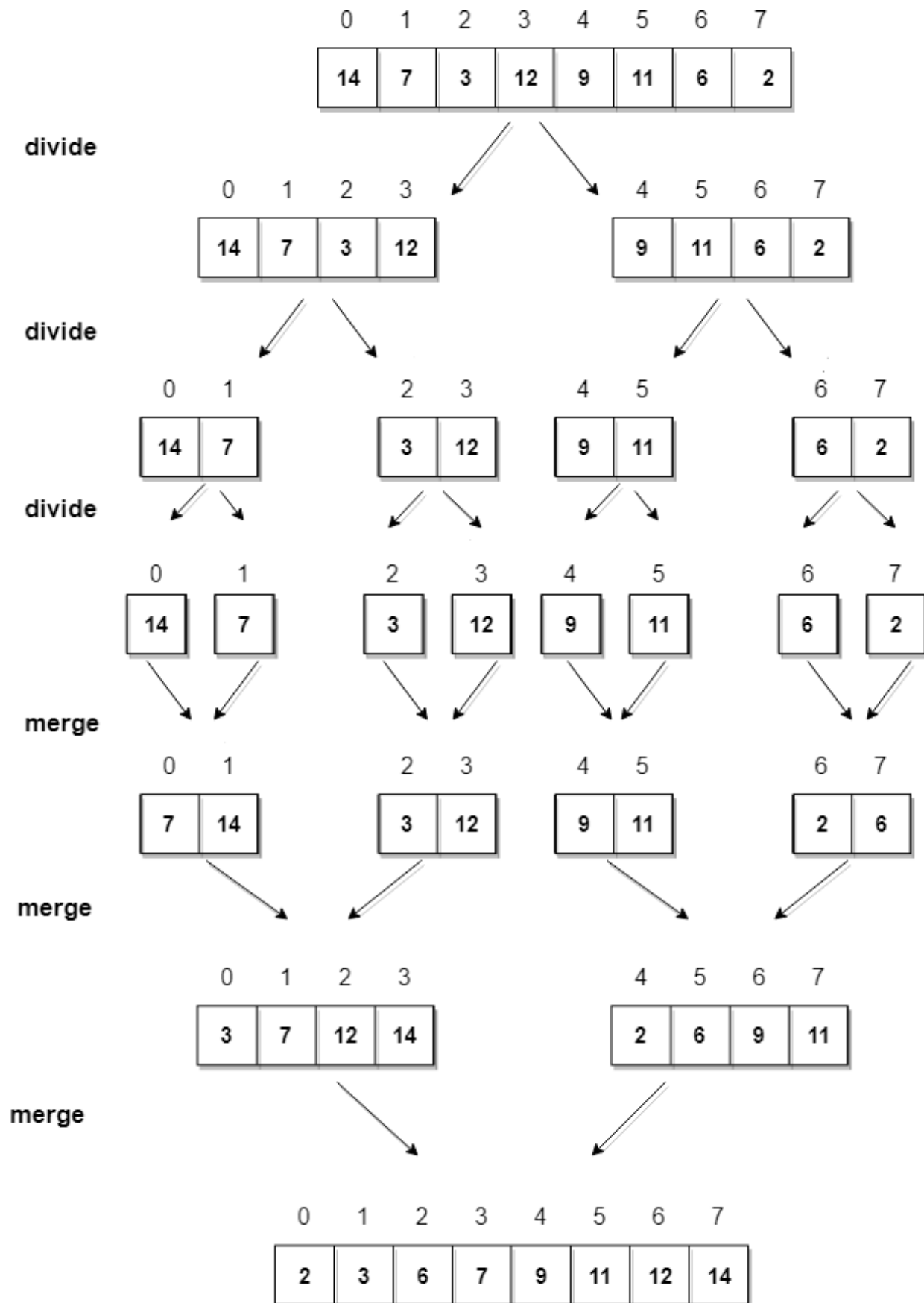
Merge Sort : Merge Sort is one of the most popular sorting algorithms that is based on the principle of Divide and Conquer Algorithm.

The concept of Divide and Conquer involves three steps:

- 1. Divide :** A problem is divided into multiple sub-problems.
- 2. Conquer :** Each sub-problem is solved individually.
- 3 : Combine :** sub-problems are combined to form the final solution.



Example : Consider an array with values 14, 7, 3, 12, 9, 11, 6, 2



Algorithm : MergeSort(int, A[0...n-1], low, high)

```
{  
  
    If(low < high ) then  
    {  
        mid=( low + high ) / 2  
        MergeSort ( A , low, mid )  
        MergeSort ( A , mid+1, high )  
        Combine(A, low, mid, high)  
    }  
}
```

Algorithm : Combine (A[0..n-1], low, mid, high)

```
{  
    k <- low;  
    i <- low;  
    j <- mid + 1;  
    while ( i <= mid and j <= high ) do  
    {  
        if ( A[i] <= A[j]) then // if smaller element is present in left sublist.  
        {  
            temp [ k ] = A[i]  
            i <- i + 1;  
            k <- k + 1;  
        }  
        else // if smaller element is present in right sublist.  
        {  
            temp [ k ] = A[j]  
            j <- j + 1;  
            k <- k + 1;  
        }  
    }  
    while( i <= mid ) do // copy remaining elements of left sublist to temp.  
    {  
        temp [ k ] = A[i]  
        i <- i + 1;  
        k <- k + 1;  
    }  
    while ( j <= high ) do // copy remaining elements of right sublist to temp.  
    {  
        temp [ k ] = A[j]  
        j <- j + 1;  
        k <- k + 1;  
    }  
    for(k=low;k<=high;k++) //Copy the elements from temp array to A  
        A[k] = temp[k];  
}
```

Time Complexity of Merge Sort : The time complexity of merge sort is given by recurrence relation is

$$T(1) = a$$

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

$$= 2[2T(n/2/2) + cn/2] + cn$$

$$= 2[2T(n/4) + cn/2] + cn$$

$$= 4T(n/4) + \cancel{2cn}/2 + cn$$

$$= 4T(n/4) + cn + cn$$

$$= 4T(n/4) + 2cn$$

$$= 4[2T(n/4/2) + cn/4] + 2cn$$

$$= 4[2T(n/8) + cn/4] + 2cn$$

$$= 8T(n/8) + \cancel{4cn}/4 + 2cn$$

$$= 8T(n/8) + cn + 2cn$$

$$= 8T(n/8) + 3cn$$

$$= 2^3 T(n/2^3) + 3cn$$

.

.

.

.

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

$$= 2^{\log_2 n} T(1) + \log_2 n \cdot cn$$

$$= n \cdot a + cn \log_2 n$$

$$= an + cn \log_2 n$$

In particular

$$n = 2^i$$

$$i = \log_2 n$$

$$a^{\log_2 n} = n$$

By representing it by in the form of Asymptotic notation O is $T(n) = O(n \log n)$

STRASSEN'S MATRIX MULTIPLICATION :

- Let A and B be two $n \times n$ matrices. The product matrix $C = AB$ is also an $n \times n$ matrix whose i, j^{th} element is formed by taking the elements in the i^{th} row of A and j^{th} column of B and multiplying them to get

$$C(i, j) = \sum_{1 \leq k \leq n} A(i, k) B(k, j)$$

- The time complexity for the matrix Multiplication is $O(n^3)$.
- Divide and conquer method suggest another way to compute the product of $n \times n$ matrix.
- We assume that N is a power of 2 .In the case N is not a power of 2 ,then enough rows and columns of zero can be added to both A and B .SO that the resulting dimension are the powers of two.
- If $n=2$ then the following formula as a computed using a matrix multiplication operation for the elements of A & B.
- If $n>2$,Then the elements are partitioned into sub matrix $n/2 \times n/2$..since 'n' is a power of 2 these product can be recursively computed using the same formula .This Algorithm will continue applying itself to smaller sub matrix until 'N' become suitable small($n=2$) so that the product is computed directly .
- The formula are

$$\begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix} = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} * \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix}$$

The Multiplication gives,

$$C_{11} = A_{11} * B_{11} + A_{12} * B_{21}$$

$$C_{12} = A_{11} * B_{12} + A_{12} * B_{22}$$

$$C_{21} = A_{21} * B_{11} + A_{22} * B_{21}$$

$$C_{22} = A_{21} * B_{12} + A_{22} * B_{22}$$

To compute AB using the above decomposition. We need to perform 8 multiplications of $n/2 \times n/2$ matrices and four additions of $n/2 \times n/2$ matrices. Since two $n/2 \times n/2$ matrices can be added in time cn^2 for some constant c., the overall computing time $T(n)$ of the resulting divide and conquer algorithm is given by the recurrence relation is

$$T(n) = \begin{cases} b & n \leq 2 \\ 8T(n/2) + cn^2 & n > 2 \end{cases}$$

Where b and c are constants.

The time complexity is $T(n) = O(n^3)$

Hence no improvement over the conventional method.

- Strassen has discovered a way to compute the C_{ij} using only 7 multiplication and 18 addition or subtraction.

Strassen's Formulas :

$$P = (A_{11} + A_{22})(B_{11} + B_{22})$$

$$Q = (A_{21} + A_{22}) * B_{11}$$

$$R = A_{11} * (B_{12} - B_{22})$$

$$S = A_{22} * (B_{21} - B_{11})$$

$$T = (A_{11} + A_{12}) * B_{22}$$

$$U = (A_{21} - A_{11}) * (B_{11} + B_{12})$$

$$V = (A_{12} - A_{22}) * (B_{21} + B_{22})$$

$$C_{11} = P + S - T + V$$

$$C_{12} = R + T$$

$$C_{21} = Q + S$$

$$C_{22} = P + R - Q + U$$

Example : 2 * 2 matix

$$\text{Ex : if } A = \begin{pmatrix} 1 & 1 \\ 1 & 1 \end{pmatrix}, B = \begin{pmatrix} 0 & 0 \\ 1 & 1 \end{pmatrix}$$

Solution : The given matrix is order of 2 * 2. hence we will compute P, Q, R, S, T, U, V.

Let

$$\begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} = \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix}$$

$$A_{11} = 1$$

$$B_{11} = 0$$

$$A_{12} = 1$$

$$B_{12} = 0$$

$$A_{21} = 1$$

$$B_{21} = 1$$

$$A_{22} = 1$$

$$B_{22} = 1$$

Now,

$$P = (A_{11} + A_{22})(B_{11} + B_{22})$$

$$= (1 + 1)(0 + 1)$$

$$= 2 * 1 = 2$$

$$Q = (A_{21} + A_{22})B_{11}$$

$$= (1 + 1) * 0$$

$$= 2 * 0 = 0$$

$$R = A_{11} * (B_{12} - B_{22})$$

$$= 1 * (0 - 1)$$

$$= 1 * (-1) = -1$$

$$S = A_{22} * (B_{21} - B_{11})$$

$$= 1 * (1 - 0)$$

$$= 1 * 1 = 1$$

$$T = (A_{11} + A_{12})B_{22}$$

$$= (1 + 1) * 1$$

$$= 2 * 1 = 2$$

$$U = (A_{21} - A_{11}) * (B_{11} + B_{12})$$

$$= (1 - 1) * (0 + 0)$$

$$= 0 * 0 = 0$$

$$\begin{aligned}
 V &= (A_{12} - A_{22}) * (B_{21} + B_{22}) \\
 &= (1 - 1) * (1 + 1) \\
 &= 0 * 2 = 0
 \end{aligned}$$

$$\begin{aligned}
 C_{11} &= P + S - T + V \\
 &= 2 + 1 - 2 + 0 \\
 &= 3 - 2 = 1
 \end{aligned}$$

$$\begin{aligned}
 C_{12} &= R + T \\
 &= -1 + 2 = 1
 \end{aligned}$$

$$\begin{aligned}
 C_{21} &= Q + S \\
 &= 0 + 1 = 1
 \end{aligned}$$

$$\begin{aligned}
 C_{22} &= P + R - Q + U \\
 &= 2 + -1 - 0 + 0 \\
 &= 2 - 1 - 0 + 0 = 1
 \end{aligned}$$

$$\begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix} = \begin{pmatrix} 1 & 1 \\ 1 & 1 \end{pmatrix}$$

Example 2 : 4 * 4 matrices

$$4 * 4 = \begin{pmatrix} 2 & 2 & 2 & 2 \\ 2 & 2 & 2 & 2 \\ 2 & 2 & 2 & 2 \\ 2 & 2 & 2 & 2 \end{pmatrix} * \begin{pmatrix} 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \end{pmatrix}$$