ADA

UNIT-I

* Algorithm :- An algorithm is a finite sequence of instructions each of which has a clear meaning and can be performed with a finite amount of effort in a finite length of time.

→ No matter what the input values may be, an algorithm terminates after executing a finite number of instruction.

→ In addition every algorithm must satisfy the following criteria.

Input :- There are zero or more quantities, which are externally supplied.

Output :- atleast one quantity is produced

Definiteness :- each instruction must be clear and unambiguous

finiteness :- if we trace out the instruction of a algorithm then for all cases the algorithm will terminates after a finite number of steps.

effectiveness :-

Pseudo code for expressing algorithms :-

Algorithm specification :- Algorithm can be described in tree ways.

→ Natural language like english : When this way is choused care should be taken, we should ensure that each & every stmt is definite.

* Performace Analysis :- → an algorithm is said to be efficient & fast if it take less time to execute and consume less memory space
the Algorithm Performance can be Analysised by with the help of two measurement we can analysis algorithm.

* space complexity
* Time complexity.

* space complexity :- which means measuring the amount of space memory space required by an algorithm during forces of execution.

=> The algorithm general requires space for instruction space, dataspace and environment space.

=> All instruction consisting of executable program.

=> space :- Instruction space is the space needed to store the compiled version of the program instruction

Data space :- Data space is the space needed to store all constant and variable values.

* Data space has two components:
  - Space needed by constants and simple variable in Program.
  - Space needed by dynamically allocated objects such as arrays and class instances.

Environment stack space :- The environment stack is used to save information needed to resume execution of partially completed function.

Instruction space :- The amount of instruction space that is needed depend on factors such as.

- The compiler used to complete the program into machine.
- The compiler options in effect at the time of compilation.
- The traget computer.

The space requirement S(P) of any algorithm P may therefore be written as, S(P) = C + SP (Instance characteristics) where "C" is a constant.

Example :-

```
Algorithm sum(a,n)
{
s=0.0;
for I=1 to n do
do s=s+a[I];
return s;
}
```

* Time complexity :-

→ The time needed by an algorithm expression as a function of the size of a problem is called time complexity of the algorithm.

→ The time complexity of a program is the amount of computer time it needs to run to completion

=> The limiting behaviour of the complexity as size increases is called asymptotic time complexity.

=> It is the asymptotic complexity of an algorithm which ultimately determines the size of problems that can be saved by the algorithm

The Running time of a Program :-

When solving a problem we are faced with a choice among algorithm. The basis for this can be any one of the following.

i. We would like an algorithm that is easy to understand codeandduge.

ii. We would like an algorithm that makes efficient use of the computer's resources, especially, one that runs as fast aspossible.

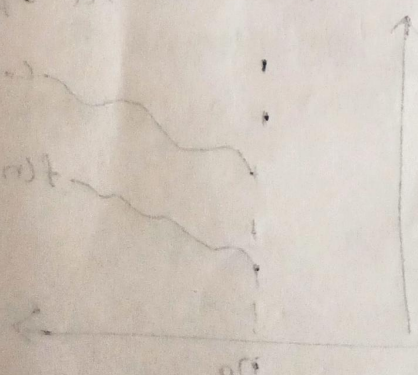
Measuring the runing time of a Program :-

The running time of a program depends on factors such as.

1. The i/p to the Program
2. The quality of code generated by the compiler used to create the object program
3. The nature and speed of the instructions on the machine used to execute the program.
4. The time complexity of the algorithm underlying the Program.

Statement	s/e	frequency	Total
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

* complexity of algorithm :-



* Asymptotic Notation :-

- $\mathcal{O}$, Θ, Ω, o, ω.
- used to describe the running time of algorithms
- instead of exact running time, say $O(n^2)$
- Defined for functions whose domain is the set of natural numbers, N
- Determine sets of functions, in practice used to compare two functions.

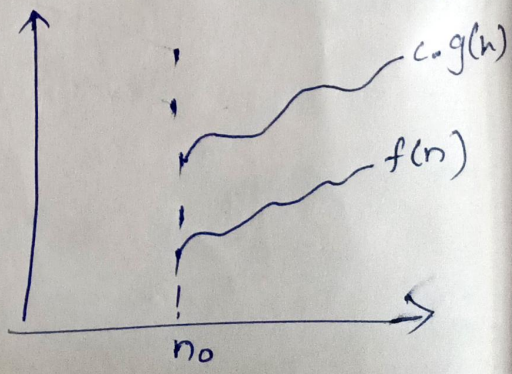
1) Big-O (upper Bound) (O)
2) omega notation (Ω)
3) Theta notation (Θ)
4) little-o notation (o)

1) Big-O :-

The function $f(n) = O(g(n))$ if and only if there exists positive constants c and n_0 such that

$f(n) \leq c * g(n)$ for all, $n > n_0$

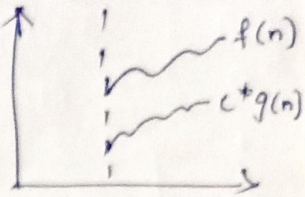
→ Bigh-oh notation denotes the upper bound of given function.



2) Omega notation (Ω)

→ The function $f(n) = \Omega g(n)$ iff there exists positive constants c and n_0 such that $f(n) \geq c * g(n)$ for all n, $n > n_0$.

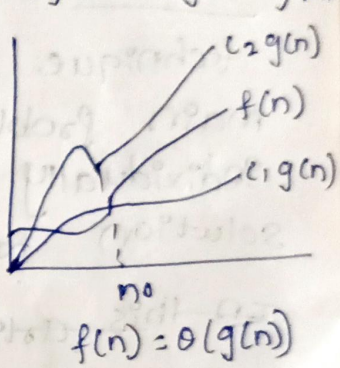
→ Ω notation denotes lower bound and best case



3) Theta notation (Θ) :-

The function $f(n) = \Theta(g(n))$ if there exists positive constants c_1, c_2 and n_0 such that $c_1(g(n)) \leq c_2(g(n))$ where $n > n_0$

→ Θ notation denotes upper and the lower bound. the average case



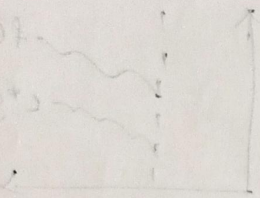
$f(n) = \Theta(g(n))$

4) little - o notation (o) :-

The function $f(n) = o(g(n))$ if and only if

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

It denotes an upper bound that is not asymptotically tight.



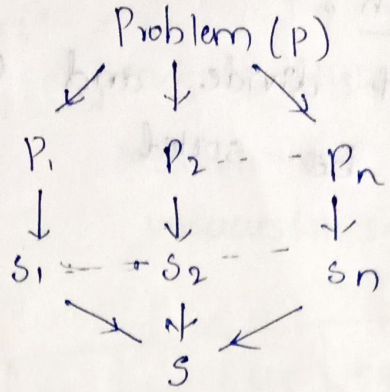
Part - II

Divide and conquer : Algorithm :- is a Problem solving technique used to solve Problem by divide the main Problem into sub programs solving them individually and then merging them to find solution to the original Problem.

~~In this article we are going~~

Divide and conquer algorithm consists of a dispute using the following tree steps.

1. Divide :- the original Problem into a set of subproblems.
2. conquer :- solve every subproblem individually recursively
3. combine :- Put together the solution of the subproblems to get the solution to the whole Problem.



Algorithm:-

```
DAC(P)
{
if (small(P))
 {
S(P);
 }
else
{
divide P into P1, P2 --- Pn
apply DAC(P1), DAC(P2), ---- DAC(Pn)
combine ( DAC(P1) - - - DAC(Pn));
```

Application:-

1. Binary Search
2. Quick Sort
3. merge sort
4. stressens matrix multiplication.

* Binary search :-

→ application of divide and conquer.

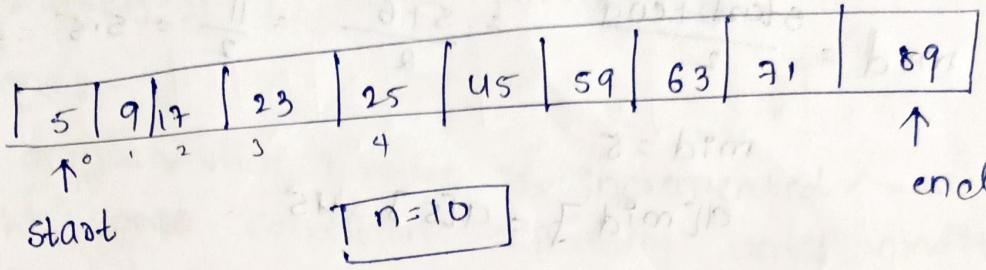
→ arry should be sorted

Binary search algorithm :-

```
binary (a, n, key)
{
start = 0, end = n-1;
while (start ≤ end)
 {
mid = start + end / 2 :
if (key == a[mid])
  return mid;
else if (key < a[mid])
    end = mid - 1:
else:                    // key > a[mid]
```

```
start = mid+1;
}                    #end of while loop
return -1;           # start > end
}                    unsuccesful search
```

Ex:-



key = 59

$$mid = \frac{start + end}{2} = \frac{0+9}{2} = 4.5 \checkmark$$

a[mid] = a[4] = 25

three case

key = mid

key > mid

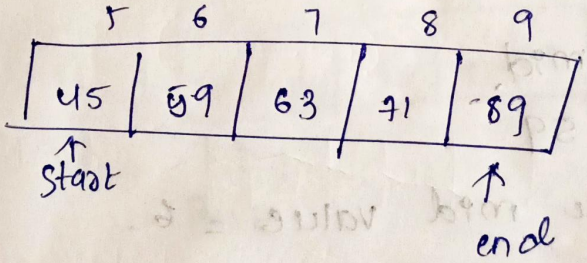
59 > 25

end = mid-1 ⇐ key < mid

start = mid+1 ⇐ key > mid

So

start = 4+1 = 5



$$mid = \frac{start + end}{2} = \frac{5+9}{2} = \frac{14}{2} = 7$$

a[mid] = a[7] = 63

key < mid

59 < 63

end = mid - 1 => 7 - 1 ; end = 6

5	6
45	59

$$mid = \frac{start + end}{2} = \frac{5+6}{2} = \frac{11}{2} = 5.5 = 5$$

mid = 5

a[mid] = a[5] = 45

key > mid

59 > 45

start = mid + 1

= 5 + 1

= 6

6
59

start end.

$$mid = \frac{start + end}{2} = \frac{6+6}{2} = \frac{12}{6}$$

a[mid] = a[6] = 59

key = mid.

59 = 59

return the mid value = 6.

element present the six position.

-: Quick Sort :-

Quick sort is the application of divide and conquer in the quick sort the division into two subArray is made so that sorted sub array need not to be merged This rearranging is called partitioning

conditions :-

1) If a[i] < v then i value is incremented we will check the same condition repeatedly until condition becomes false.

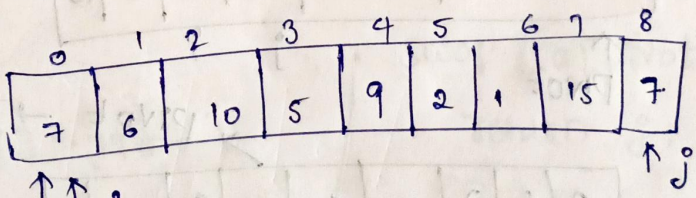
2) If a[j] > v then j value is decremented we will check the same condition repeatedly until condition becomes false.

3) If i > j then swap a[i] with a[j]

4) If i ≥ j then swap a[j] with pivot element (v)

5) Apply quick sort for two subsets recursively.

Ex :-



a[j] ≤ pivot 7 ≤ 7

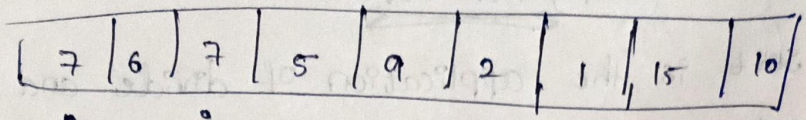
i++

7 > 10 NO

So stop it

a[j] > pivot 7 > 6

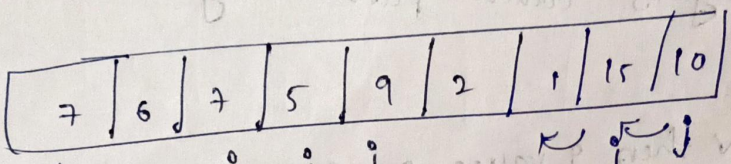
j--



a[i] ≤ pivot

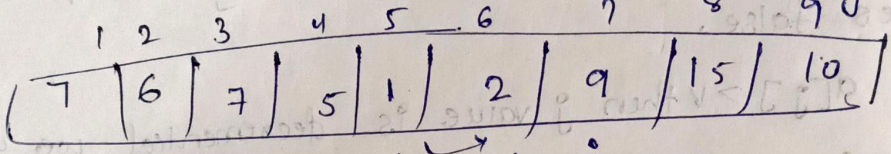
7 ≤ 5

i++.



a[j] ≤ pivot

j--

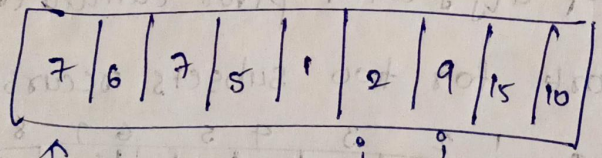


a[i] ≤ Pivot

i=6, j=5

True

i++

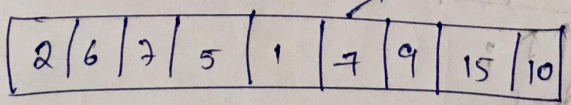


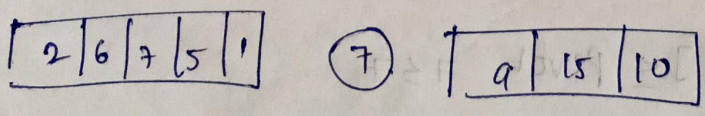
a[j] > pivot

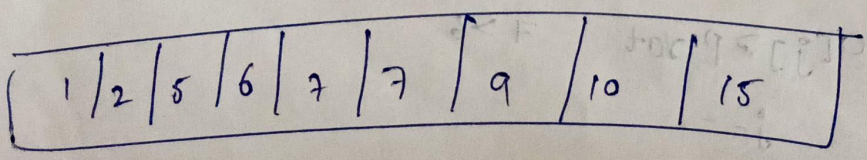
~~4~~ > 9

~~j--~~

pivot → Iteration for Partition







* Quick sort Algorithm :-

```
Quick.Sort (A,i,j)
 {
 if (i<j)
 {
 loc = partition (A,i,j);
   Quicksort (A,i,loc-1);
   Quicksort (A,loc+1,j);
   }
```

* Partition Algorithm :-

```
Partition (A,i,j)
 Pivot = a[i];
  i=0;
  j=n-1;
which (a[i] <= pivot )
  {
  i++;
while (a[j] >pivot )
  {
  j--;
  }
if (i<j)
{
swap (a[i], a[j];
 }
 }
```

```
swap (a[pivot], a[j]);
  return j;
  }
```

complexity

worst case → O(n²)

Best case → O(nlogn)

merge sort $O(n \log n)$

Merge sort - Divide

Merge sort & merge.

merge sort is the application of divide and conquer. The time complexity in worst case is $O(n \log n)$

→ Given a sequence of 'n' element $a[1] \dots a[n]$ to split the element into two sets.

$$a[1] \dots a[n/2] \quad \& \quad a[n/2+1] \dots a[n]$$

→ each set is individually sorted and the resulting sorted sequence are merged to produce a single sorted sequence of n element.

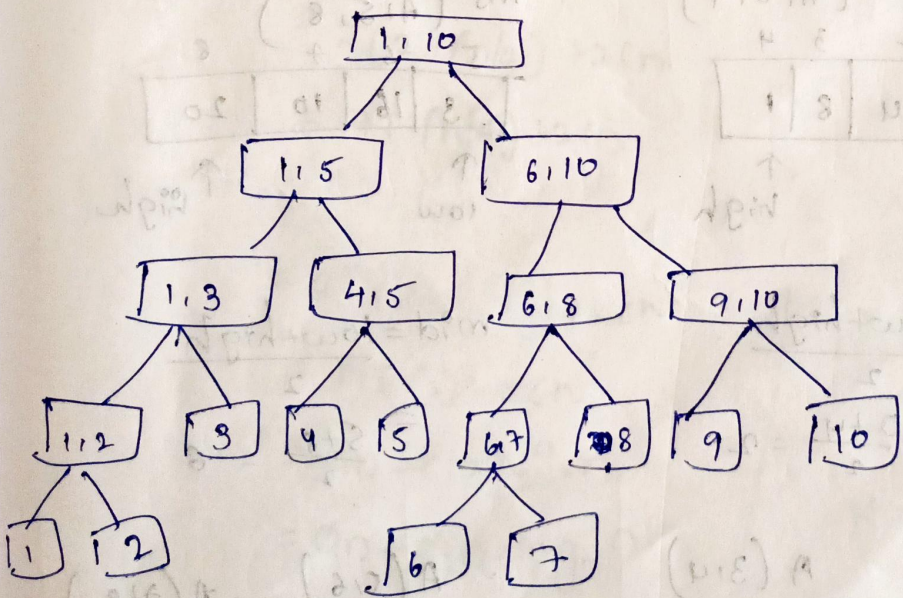
Algorithm :-

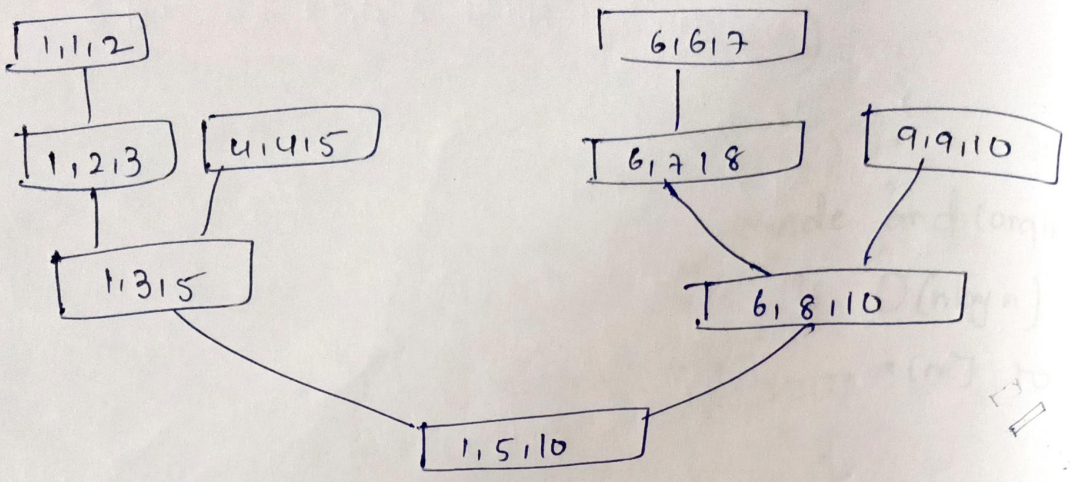
```
1. Algorithm mergesort (low, high)
2. {
3.   if (low < high) then
4.   {
5.     mid := [ (low + high) / 2 ]
6.     mergesort (low, mide);
7.     mergesort (mid+1, high);
8.     merge sort (low, mid, high);
9.   }
10. }
```

sort the given numbers by using merge sort

310 , 285 , 179 , 652 , 351 , 423 , 861 , 254 , 450 , 520

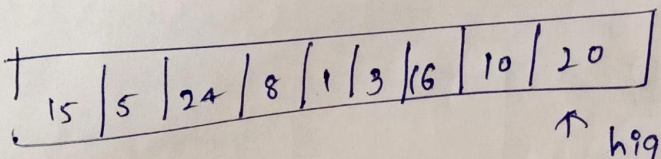
Tree of calls of mergesort (1,10)





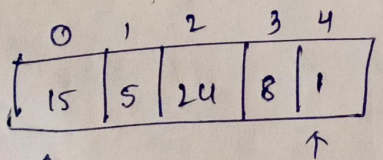
Tree of call of merge

Ex:-

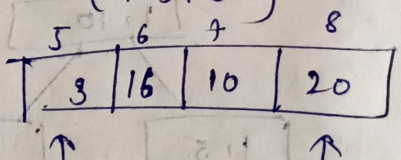


$mid = \frac{low + high}{2} = \frac{0+8}{2} = 4$

ms (A, 0, 4)



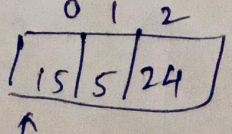
ms (A, 5, 8)



$mid = \frac{low + high}{2} = \frac{0+4}{2} = 2$

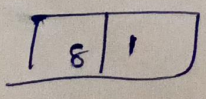
$mid = \frac{low + high}{2} = \frac{5+8}{2} = 6$

A (0, 2)

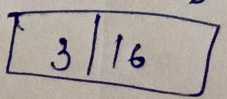


$\frac{0+2}{2} = 1$

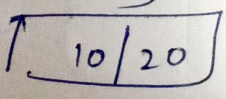
A (3, 4)

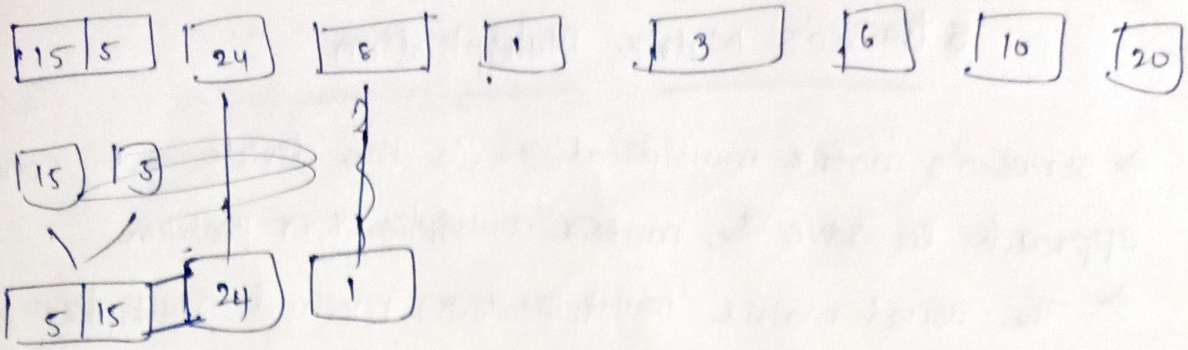


A (5, 6)



A (7, 8)





Time complexity :-

$$T(n) = \begin{cases} a & n=1,\ a \text{ constant} \\ 2T(n/2) + cn & n>1,\ c \text{ constant} \end{cases}$$

$$T(n) = 2T(n/2) + cn$$
$$= 2(2T(n/4) + 2cn$$
$$= 4T(n/4) + 2cn$$
$$\vdots$$
$$= 2^k T(n/2^k) + kcn$$
$$= nT(1) + kcn$$
$$= an + c\log_2 n \cdot n$$
$$= an + c \cdot n \cdot \log_2 n$$
$$= O(n \log n)$$

$n/2^k = 1$

$n = n^k$

$k = \log_2 n$

Strassen's Matrix Multiplication

⇒ Strassen's matrix multiplication is the divide and conquer approach to solve the matrix multiplication problem

⇒ 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 $O(n^{\log 7})$

$$a = \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix} \qquad b = \begin{bmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{bmatrix}$$

$$c = \begin{bmatrix} c_{11} & c_{12} \\ c_{21} & c_{22} \end{bmatrix}$$

$$c_{11} = a_{11} \times b_{11} + a_{12} \times b_{21}$$
$$c_{12} = a_{11} \times b_{12} + a_{12} \times b_{22}$$
$$c_{21} = a_{21} \times b_{11} + a_{22} \times b_{21}$$
$$a_{22} = a_{21} \times b_{12} + a_{22} \times b_{22}$$

$$C(i) = \sum_{k=1}^{n} a_{ik} * b_{kj}$$

* Time complexity :- $O(n^3)$

2^3 - multiplications

$n = 4 - n/2 = 2$

2^2 - additions

$n^3 + n^2$

$$a = \begin{bmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ a_{31} & a_{32} & a_{33} & a_{34} \\ a_{41} & a_{42} & a_{43} & a_{44} \end{bmatrix} \qquad b = \begin{bmatrix} b_{11} & b_{12} & b_{13} & b_{14} \\ b_{21} & b_{22} & b_{23} & b_{24} \\ b_{31} & b_{32} & b_{33} & b_{34} \\ b_{41} & b_{42} & b_{43} & b_{44} \end{bmatrix}$$

A_{11} A_{12} A_{21} A_{22} B_{11} B_{12} B_{21} B_{22}

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

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

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

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

Algorithm: -

```
1. Algorithm MM (A1K1n)
2. {
3.  if (n<=2)
4.  {
5.  c11 = a11b11 + a12 b21;
6.  c12 = a11b12 + a12 b22;
7.  c21 = a21b11 + a22b21;
8.  c22 = a21 b12 + a22b22;
9.  }
10. else
11. {
12. mid := n/2;
13. MM ( A11, B11, n/2 ) + MM (A12, B21, n/2 )
14. mm (A11, B21, n/2) + MM ( A12, B21, n/2 )
15. mm(A21, B11, n/2 ) + MM ( A22, B21, n/2 )
16. MM (A21, B12, n/2 ) + MM ( A22, B22, n/2 )
17. }
18. }
```

Time complexity

$$T(n) = \begin{cases} 1 & n \le 2 \\ 8T(n/2) + 4n^2 \end{cases}$$

$\Rightarrow aT(n/b) + n^k$ $\quad \Rightarrow a = 8,\ b = 2,\ k = 2$

$\rightarrow \log_b^a + n^k$

$\rightarrow \log_2^8 + n^2$

$= n^3 + n^2 \Rightarrow O(n^3)$

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

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

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

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

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

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

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

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

$C_{12} = R + T$

$C_{21} = Q + S$

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

$$T(n) = \begin{cases} 1 & n \le 2 \\ 7T(n/2) + 18n^2 & n \ge 2 \end{cases}$$

$aT(n/2) + n^k$

$\Rightarrow \log_2^7 + n^2$ $\quad a = 7,\ b = 2,\ k = 2$

$= n^{2.81} + n^2$

$\Rightarrow O(n^{2.81})$ or $O(\log_2^7)$