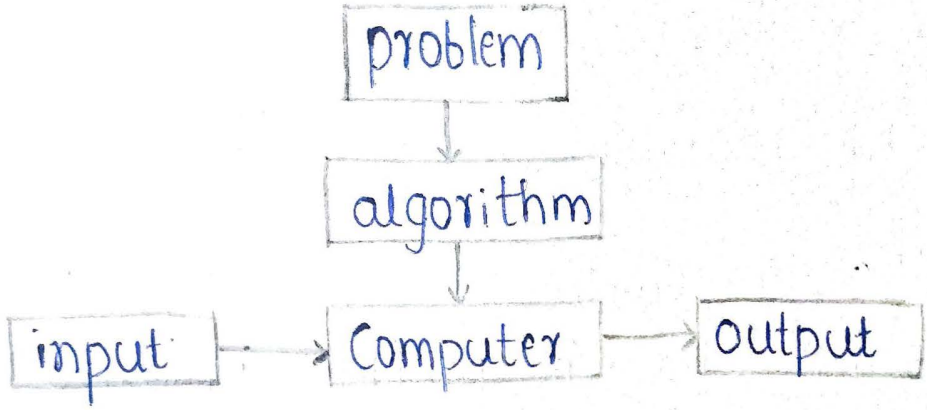
UNIT - 1

7/1/2020

Algorithm:

An algorithm is a sequence of unambiguous instructions for solving a problem i.e. for obtaining a required output for any legitimate input in a finite amount of time.



11/1/2020

Why to study algorithms:

- The algorithm forms the core part of any software or a program making it more important for a computer science aspirants.
- A good algorithm should possess following requirements.

1. finiteness: the algorithm terminates after finite no. of steps

using a program or a software.

1. understanding the problem : before solving any problem in any way(s) it is very important to know, imagine, visualize the given problem.
2. Take decisions about the following:
   a. The capabilities of the computational device for which the algorithm need to be developed
   b. deciding upon whether the algorithm should provide exact solution or an approximate solution
   c. identifying and finalizing the possible data structure(s) that can be used / applied in the algorithm
   d. deciding upon the methodology / algorithmic technique that need to be followed to solve the problem.
3. Designing an algorithm:

   In this step the solution for the problem is written / solved / deduced that can help for further analysis.

4 Prove the correctness

Here the correctness about required o/p of the problem is tested for different algorithmic techniques designed in previous step.

5. Analyse the algorithm :

The set of the algorithms that solve the given problem are evaluated/studied both theoretically and empirically wrt following parameters.

a. Time - the time taken by different algorithms is compared.

b. Space - the amt of extra space consumed by the alg. is compared.

c. Optimality - A alg. that provides optimal or more suitable or more acceptable soln for the given problem is identified.

6. Code the algorithm :

Convert the alg. to a program of interest.

The alg. process can be applied to some of the well known problems as listed.

- Sorting
- Searching
- Shortest paths in a graph

- minimum spanning tree
- primality testing.
- Travelling salesman problem
- Knapsack problem
- Chess
- Towers of Hanoi
- Program termination.

* Following are some of the design techniques/ strategies that can be applied to solve a problem

- Brute force
- Divide and Conquer
- Decrease and Conquer
- Transform and conquer
- Greedy approach
- Dynamic programming
- Roadtracking and branch-and-bound.
- Space and time trade offs

* Data Structures:

List - array, linked list

stack

queue

graphs

trees

23/1/2020:

Algorithmic design Techniques
or
Analysis of algorithmic design techniques.

1 The algorithms which have been designed to solve ~~th~~ a specific problem can be analysed by the help of ~~eg~~ evaluation parameters like time space, optimality and correctness

2 The time efficiency of algorithms can be analysed by identifying the number of times basic operation is executed.

3 The number of times basic operation ~~is~~ executed is denoted as $C(n)$ for some input size.

4. The $C(n)$ of any algorithm is expressed interms of $C_{worst}(n)$, $C_{best}(n)$ and $C_{avg}(n)$

For example,

the Sequential search of n numbers for some key element has:

worst efficiency = $C_{worst}(n) = n$,

best efficiency = $C_{best}(n) = 1$ and

average efficiency = $C_{avg}(n) = n/2$.

The $C(n)$ for any algorithm is also reffered as efficiency class. Some of the standard efficiency classes and their nature of growth (order of growth) is represented in the following table.

- The table indicates from top to bottom the time efficiency ^or decreasing the no. of times basic operation executed increasing

Basic Asymptotic efficiency classes

1	constant
$\log n$	~~algo~~ logarithmic
n	linear
$n \log n$	n-log-n
n^2	quadratic
n^3	cubic
2^n	exponential
$n!$	factorial.

25-1-2020:

Asymptotic notations:

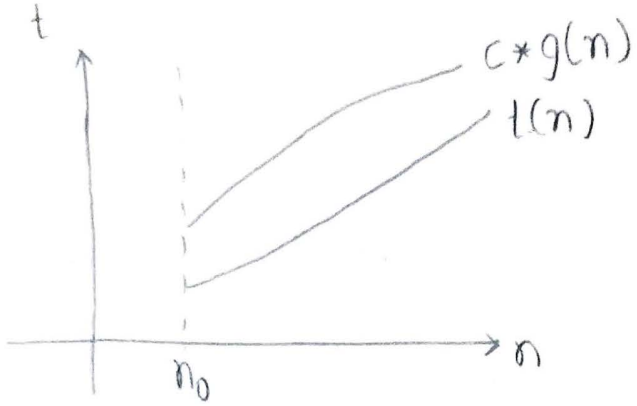
In the analysis of algorithm wrt time the efficiency or C(n) of the developed algorithm should be expressed interms of one of the above standard efficiency classes

- This comparison of obtained c(n) with standard c(n) can be accomplished by 3 notations
- The notations will help in expressing obtained c(n) to be either more, less or both interms of standard efficiency classes

1. Big-oh (O)

A function t(n) is said to be in big-oh O(g(n)) and represented as $t(n) \in O(g(n))$. if t(n) is bounded above by some constant multiple of g(n) for all large values of n, i.e. if there exist some common positive constant c and some non-negativ integer no then $t(n) \leq c \times g(n)$ for all $n \geq n_0$.

Graphically this relation can be represented as shown in the following fig.



for eg: $2n+3 \in O(n)$ when

$$2n+3 \leq c * n$$

for $c=5$ and $n=n_0 \geq 1$

The given function $2n+3$ (obtained efficiency of the algorithm) can also be represented as

$2n+3 \in O(n^2)$, $2n+3 \in O(n^3)$ and so on..

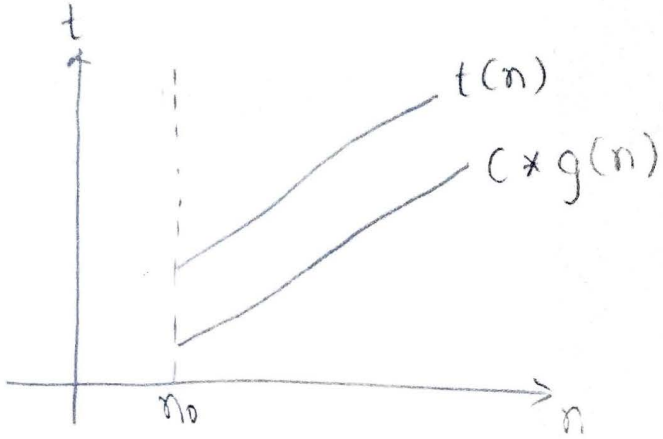
however $2n+3 \in O(n)$ is found to be the nearest minimal efficiency class. hence the best way to express or the optimal efficiency class for which $2n+3$ can belong to is n

2. Big-Omega (Ω)

The function $t(n)$ is said to be in $\Omega(g(n))$ denoted as $t(n) \in \Omega(g(n))$ if $t(n)$ is bounded below by some positive constant multiple of $g(n)$ for all large n i.e. if there exist some positive const

c and some non-negative integer n_0 such that $t(n) \geq c * g(n)$ for $n \geq n_0$

Graphically $\Omega(n)$ is represented as shown in the following fig.



for eg: The function $2n+3 \in \Omega(n)$

for $c = 1$ and $n = n_0 \geq 0$

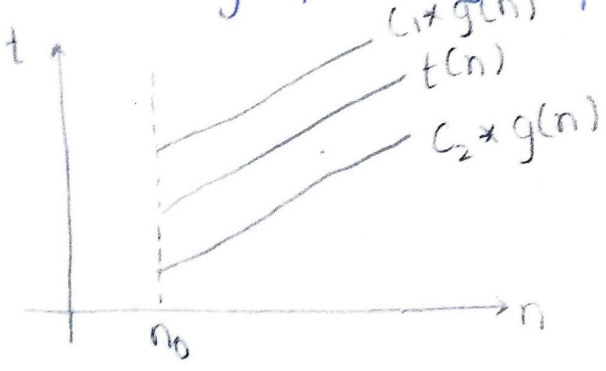
$2n+3$ can also be expressed interms of different efficiency classes but the efficiency class n is the nearest minimal efficiency class, hence $2n+3$ is best expressed or optimally expressed in terms of $\Omega(n)$.

3. Theta (Θ):

A function $t(n)$ is said to be in $\Theta(g(n))$ denoted as $t(n) \in \Theta(g(n))$ if $t(n)$ is bounded both above and below by some positive constant multiple of $g(n)$

for all large values of n, i.e if there exist some positive constants c_1 and c_2 and some non-negative integer n_0 such that $c_2 g(n) \leq t(n) \leq c_1 g(n)$ for $n \geq n_0$.

The same is graphically represented as



for eg: The function $2n+3$ is under $\theta(n)$ as it is bounded above and below by the efficiency class n i.e.

$$1 * n \leq 2n+3 \leq 5n \quad \forall\ n_0 \geq 1$$

Some important observations

1. The values of $t(n)$ and $g(n)$ do not matter much for the values lesser than n_0
2. The pratical requirement of big-oh notation is to express maximum upper bound of obtained efficiency class.

3. The practical requirement of big-omega notation is to express maximum lower bound of obtained efficiency class

4. If a function covers both lower bound and upper bound of some efficiency class then the obtained efficiency is almost similar to the standard efficaency class. This can be represented by Θ

Write an algorithm for bubble sort.

I/P :

O/p :

Step 1 :
```
for (i = 0 ; i < n-1 ; i++)
    for (j = 0 ; j < n-i-1 ; j++)
        if (a[j] > a[j+1])
            temp = a[j]
            a[j] = a[j+1]
            a[j+1] = temp.
```

The efficiency of above algorithm may come under n^2

27/1/2020

Some properties of asymptotic order of growth

- $f(n) \in O(f(n))$
- $f(n) \in O(g(n))$ iff $g(n) \in \Omega(f(n))$
- if $f(n) \in O(g(n))$ and $g(n) \in O(h(n))$, then $f(n) \in O(h(n))$

Note similary with $a \leq b$

- if $f_1(n) \in O(g_1(n))$ and $f_2(n) \in O(g_2(n))$, then

$$f_1(n) + f_2(n) \in O(\max\{g_1(n), g_2(n)\})$$

Establishing the order of growth using limits

$$\lim_{n \to \infty} T(n)/g(n) : \begin{cases} 0 & \text{order of growth of } T(n) < \text{order of growth of } g(n) \\ c > 0 & \text{order of growth of } T(n) = \text{order of growth of } g(n) \\ \infty & \text{order of growth of } T(n) > \text{order of growth of } g(n) \end{cases}$$

L'Hôpital's rule

If $\lim_{n \to \infty} f(n) = \lim_{n \to \infty} g(n) = \infty$ and the derivatives of f and g exists then, $\lim_{n \to \infty} \frac{f(n)}{g(n)} = \lim_{n \to \infty} \frac{f'(n)}{g'(n)}$

Stirling's formula :

$$n! \approx (2\pi n)^{1/2} \cdot (n/e)^n$$

1. PT : $10n \in O(n^2)$

$$\lim_{n\to\infty} \frac{10n}{n^2} = 10 \lim_{n\to\infty} \frac{1}{n} = 0$$

2. PT : $n(n+1)/2 \in \Theta(n^2)$

$$\frac{1}{2} \lim_{n\to\infty} \frac{n^2+n}{n^2} = \frac{1}{2} \lim_{n\to\infty} (1+0) = 1/2.$$

3. PT $\log n$ is under $O(n)$

$$\lim_{n\to\infty} \frac{\log n}{n} : \frac{\infty}{\infty}$$

∴ By L'Hopital's Rule : $\lim_{n\to\infty} \frac{1/n}{1} = 0$

4. Apply Stirling's formula $n!$ v/s 2^n

$$\frac{1}{2}(2n\pi) \cdot 2\pi$$

$$\lim_{n\to\infty} \frac{n!}{2^n} : \lim_{n\to\infty} \frac{\sqrt{2n\pi} \cdot (n/e)^n}{2^n} \cdot \sqrt{2\pi} \lim_{n\to\infty} (n)^{1/2} \left(\frac{n}{2e}\right)^n$$

$$= \infty$$

Arrange the following terms in terms of order of decay from highest to lowest.

$(n+1)!$. 2^{3n}, $(2n^4+2n^3+4)$, $n\log n$, $\log n$, $6n$, $8n^2$

Time efficiency of non-recursive algorithms

General plan for analysis.

- Decide on parameter n indicating input size
- Identify algorithm's basic operation
- determine worst, average and best cases for input of size n
- set up a sum for the number of times the basic operation is executed
- Similarly the sum using standard formulas and rules (See Appendix A)

Useful Summation formulas and rules

1. $\sum_{l \le i \le u} 1 = 1 + 1 + \dots + 1 = u - l + 1$

In particular, $\sum_{l \le i \le u} 1 = n - 1 + 1 = n \in \Theta(n)$

2. $\sum_{1 \le i \le n} i = 1 + 2 + \ldots + n = \frac{n(n+1)}{2} \approx n^2/2 \in \Theta(n^2)$

3. $\sum_{1 \le i \le n} i^2 = 1^2 + 2^2 + \ldots n^2 = \frac{n(n+1)(2n+1)}{6} \approx n^3/3 \in \Theta(n^3)$

Example 1 : Maximum Element

Consider maximum element algorithm for an array of n numbers. Evaluate the algorithm or analyse the algorithm

Algorithm MaxElement (A[0.. n-1])

// Determines the values of the largest element in a given array

// Input : An array A[0 .... n-1] of real no.s

// Output : The value of the largest element in A

```
maxval ← A[0]
for i ← 1 to n-1 do
    if A[i] > maxval
return maxval
```

- Input size : n
- Basic operation : Comparison in the for loop
  if A[i] > maxval.
- $C(n) = 1 + 1 + \ldots n-1$

$= \sum_{i=1}^{n-1} 1 = (n-1) - 1 + 1$

$= n-1 \in \Theta(n)$

2 Analyse the above algorithm for minimum element in an array A[0 ... n-1]

Algorithm Min Element (A[0 ... n-1])
// Determines the values of the smallest element in a given array.
// Input : An array A[0 ... n-1] of real no.s
// Output : The value of the smallest element in A

```
minval ← A[0]
for i ← 1 to n-1 do
    if A[i] < minval
        minval ← A[i]
return minval
```

- input size : n
- Basic operation : Comparison in the for loop if A[i] < minval

$C(n) = 1 + 1 + \dots n-1$

$= \sum_{i=1}^{n-1} 1 = (n-1) - 1 + 1$

$= (n-1) \in \Theta(n)$

3. Analyse the unique element algorithm for the time efficiency.

Algorithm Unique Element (A[0....n-1])

// Determines whether all the elements in a given array are distinct

// Input : An array A[0....n-1]

// Output : Returns "true" if all the elements in A are distinct and "false" otherwise.

```
for i ← 0 to n-2 do
    for j ← i+1 to n-1 do
        if A[i] = A[j]  return false
return true
```

- Input Size : n
- Basic operation : Comparison (A[i] == A[j])
- $C(n) = \sum_{0}^{n-2} \sum_{i+1}^{n-1} 1$

  $= \sum_{0}^{n-2} (n-1)-(i+1)+1$

  $= \sum_{0}^{n-2} n-i-1$

$$= \sum_{0}^{n-2} (n-1) + \sum_{0}^{n-2} i$$

$$= (n-1) \sum_{0}^{n-2} 1 + \frac{n(n+1)}{2}$$

$$(n-2-0+1)$$

$$\times \rightarrow = (n-1)(\cancel{n-2}) + \frac{n(n+1)}{2}$$

$$(n-1)^2 + \frac{n(n+1)}{2}$$

4 Analyse the time efficiency of matrix multiplication algorithm.

Matrix Multiplication (A[0....n-1, 0...n-1], B[0.... n-1, 0...n-1])

// Multiplies 2 n by n matrices by the definition based algorithm.

3/2/2020

// Input : Two nxn matrices A and B

// Output : Matrix C = AB

```
for i ← 0 to n-1 do
    for j ← 0 to n-1 do
        C[i,j] ← 0.0
        for k ← 0 to n-1 do
            C[i,j] = C[i,j] + A[i,k] * B[k,j]
```

input size : n

basic operation : C[i,j] ← c[i,j] +

No. of basic operation is given by C(n)

$$C(n) = \sum_{i=0}^{n-1} \sum_{j=0}^{n-1} \sum_{k=0}^{n-1} 1.$$

General Plan for Analysis of Recursive Algorithms

A given problem can be solved appropriately by recursion when following criterias are attained

1. The solution of the problem is complex and difficult to be solved by non-recursive method
2. The solution to the problem is the repitation of the same operation applied on previous results to get final solution.

Following are the steps to be applied to analyse the recursive algorithms.

1. Decide on a parameter indicating an inputs size
2. Identify the algorithm's basic operation
3. Check whether of the number of times the basic op. is executed may vary on different inputs of the same size (If it may, the worst, average and best cases must be investigated separately)

4. Set up a recurrance relation with an appropriate initial condition expressing the number of times the basic operation is executed

5. Solve the recurrance by the backward substitutions

For ex: Analyse the recursive version of finding factorial of n

Def^n : $n! = 1 * 2 * \ldots\ldots * (n-1) * n$ for $n \geq 1$

and $0! = 1$

Recursive def^n of $n!$: $F(n) = F(n-1) * n$ for $n \geq 1$.

and $F(0) = 1$

Algorithm : F(n)

// Computes n! recursively

// Input : A non-negative integer n

// Output : The value of n!

if n = 0 return 1

else return F(n-1) * n

Size : n

Basic operation : $F(n-1) * n$ (Recursive call)

The deduction of no. of times basic operation is executed ($C(n)$) for an recursive algorithm is also referred as a recurrence relation or simply recurrence

$C(0) = 0! = 1$

$C(n) = C(n-1) * n$

$= n * C(n-1)$

$= n * C(n-1) * C(n-2)$

$\vdots$

$= n * C(n-1) * C(n-2) * \ldots\ldots C(n-n)$

$= n * \ldots\ldots\ldots 1$

$\approx n-1 \rightarrow$ no of multiplications

$= \Theta(n)$

4/2/2020

Perform the time efficiency analysis for the towers of hanoi problem

- For the Towers of Hanoi problem following points are to be taken care.

1. The input size is no. of discs and it is represented as n.
2. We are given 3 towers which are to be arranged with the discs such that at any time the larger disc is below the smaller disc.
3. The solution for such problem can be attained by reducing the problem of size n to the smaller size and applying the repeated method of solving.

Hence for the Towers of Hanoi problem recursion will be the best soln to solve the problem or write an algorithm.

Input size : n discs and 3 towers

Basic operation : Moving the discs from 1 tow to another by the help of intermediate tower.

From this logic it can be identified that to move n ~~towers~~ discs we need to move n-1 disc 2 times and 1 more movement of n^{th} disc ~~from~~

$$C(n) = C(n-1) + 1 + C(n-1)$$

$$= 2C(n-1) + 1$$

$$= 2[2C(n-2) + 1] + 1$$

$$= 2^2 C(n-2) + 3$$

$$= 2^2[2C(n-3) + 1] + 3$$

$$= 2^3 C(n-3) + 2^2 + 3$$

$$= 2^n C(n-n) + 2^{n-1} + n$$

$$= 2^{n-1} + n \qquad \because C(0) = 0$$

$$\in \theta(2^n)$$

Brute Force

- Brute Force is an algorithmic design technique usually described as a straight forward approach to solve a problem algorithmically.

The Brute Force approach or technique provides the solution based on problem statement and definations of the concepts (algorithmic problem solving skills) involved in the problem statement

Eg. Computing a^n

Computing $n!$

Multiplying two matrices

Searching for a key of a given value in a list.

Write Selection Sort.

Scan the array to find its smallest element and swap it with the first element. Then, starting with the second element, scan the elements to the right of it to find the smallest among them and swap it with the second element. Generally on pass i ($0 \leq i \leq n-2$), find the smallest element in

```
for(i 0 , i<n-1, i++)
    for (j i+1 , j<n , j++)
        if(a[j]<a(i])
            temp a[j]
            a[j] a[i]
            a[i] temp
```

A[i ... n-1] and swap it with A[i]

A[0] ≤ ..... ≤ A[i-1] | A[i], ... A[min] ... A[n-1]

in their final positions.

Algorithm :

// sorts a given array by selection sort

// Input : An array A[0 .... n-1] of orderable elements

// Output : Array A[0 .... n-1] sorted in ascending order

```
for i ← 0 to n-2 do
    min ← i
    for j ← i+1 to n-1 do
        if A[j] < A[min]  min ← j
    Swap A[i] and A[min]
```

Apply the selection sort for the following list of nos

89, 45, 68, 90, 29, 34, 17

Iteration	min	i	j	Result
1	0	0	1	29, 45, 68
			2	45, 89, 68, 90, 29, 34, 17
			3	29 89 68 90 45 34 17
				17 89 68 90 45 34 29
2	1	1	0	17 68 89 90 45 34 29
				17 45 89 90 68 34 29
				17 34 89 90 68 45 29
				17 29 89 90 68 45 34
3	2	2	0	17 29 68 90 89 45 34
				17 29 45 90 89 68 34
				17 29 34 90 89 68 45
4	3	3	0	17 29 34 89 90 68 45
				17 29 34 68 90 89 45
				17 29 34 45 90 89 68
5	4	4	0	17 29 34 45 89 90 68
				17 29 34 45 68 90 89
6	5	5	0	17 29 34 45 68 89 90

6/2/2020

Input size : n

Basic operation : comparison in the inner for loop

$$C(n) = \sum_{i=0}^{n-2} \sum_{j=i+1}^{n-1} 1 = \sum_{i=0}^{n-2} (n-1-i-1+1)$$

$$= \sum_{i=0}^{n-2} n-1 - \sum_{i=0}^{n-2} i$$

$$= (n-1)(n-2-0+1) - \frac{(n-2)(n-2+1)}{2}$$

$$(n-1)^2 - \frac{(n-2)(n-1)}{2}$$

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

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

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

$$\frac{n(n-1)}{2} = \frac{n^2}{2} - \frac{n}{2} = \frac{1}{2}(n^2-n)$$

Sequential Search

Algorithm : Sequential Search (A[0... n], K)

// Implements sequential search with a search key as a sentinel

// Input : An array A of n elements and a search key k.

// Output : The index of the first element in A[0... n-1] whose value is equal to K or -1 if no such element is found.

```
i ← 0
while A[i] = K do
    i = i + 1
if i < n return i
else return -1
```

~~Input size : n~~

~~Basic operation~~ : ~~while A[i] = K~~

The above sequential search is an non-recursive algorithm, which can be analysed as follows

i. Input size : n

ii. Basic Op : while $A[i] = k$

iii. $C(n) = \sum_{i=0}^{n-1} 1$

$= n - 1 - 0 + 1$

$= n \in \theta(n)$

UNIT - 2

Divide and Conquer Techniques

- The most well known algorithm design strategy:
- Divide instances of problem into 2 or more smaller instances
- Solve smaller instances recursively.
- Obtain solution to original (larger) instance by combining these solutions.

7/2/2020

General divide and Conquer Recurrence solving Mechanism:

Consider a recurrence relation expressed as

$T(n) = aT(n/b) + f(n)$ where $f(n) \in \theta(n^d)$, $d \geq 0$

Then the master theorem for $T(n)$ can be expressed as follows:

If $a < b^d$, $T(n) \in \theta(n^d)$

$a = b^d$, $T(n) \in \theta(n^d \log n)$

$a > b^d$, $T(n) \in \theta(n^{\log_b a})$

Note:

The same results hold with O instead of Θ

Eg:

1. $T(n) = 4T(n/2) + n$ has $a = 4, b = 2, d = 1$

As $b^d = 2^1 = 2$ is smaller than a

$$a > b^d$$

$$\therefore T(n) \in \Theta(n^{\log_2 4})$$

$$\in \Theta(n^2)$$

2. $T(n) = 4T(n/2) + n^2$

$a = 4 \quad b = 2 \quad d = 2$

As $b^d = 2^2 = 4$

$$a = b^4$$

by applying master's theorem.

$$T(n) \in \Theta(n^2 \log n)$$

3. $T(n) = 4T(n/2) + n^3$

$a = 4 \quad b = 2 \quad d = 3$

$b^d = 2^3 = 8$

$a < b^d$

by applying master's theorem

$T(n) \in \Theta(n^3)$

ALGORITHM : MergeSort

// Sorts array $A[0 \ldots n-1]$ by recursive mergesort

// Input : An array $A[0 \ldots n-1]$ of orderable elements

// Output : Array $A[0 \ldots n-1]$ sorted in non-decreasing order.