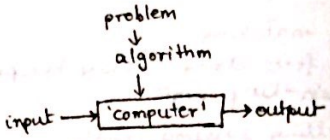
The step which takes more time is called a basic step.
The number of basic steps determines time efficiency.

Methods to find time
- Substitution method — Forward / Backward
  Similar to ∮ Mathematical Induction
- Recursive tree method
  Generate tree from equation
- Master theorem



Algorithm:
Step by step procedure of solving a problem

It is a method or technique which specifies how to solve a problem. The problem is given as an algorithm. It is programmed into a computer. It converts a given input to an output.

Defn 1.
Sequence of computational steps that transforms the input to the output.

Defn 2.
∴ Unambiguous (clear in meaning) set of instructions to solve a problem.
↳ Expected definition
i.e. for obtaining a reqd o/p for any legitimate i/p in a finite amt of time

Properties of a good algorithm.
1. Finiteness : Always terminates (with a proper output)
2. Definiteness: Each step in the algorithm should be defined clearly. ~~Each step should take a definite~~
3. Effectiveness: Each step takes a definite amount of time. (Limited/bound time)
4. Input: Should be apply to supply i/p
5. Output: Should be able to produce proper output
∴ 6. Flexibility: Each step in the algorithm can be changed.
(not mentioned in most books, first 5 must be mentioned)

eg:
Algo. 1 Euclid's Algorithm. (recursive)
Based on: gcd(m,n) = gcd(n, m mod n).

input → Two positive numbers a & b.
output → Largest integer which divides both a & b

Step 1: If b is 0, GCD is a, else go to step 2.
Step 2: ~~If b≠0~~, Find r, which is the remainder ~~when~~ after dividing a by b.
Step 3: Exchange a by b, & b value by r.

GCD(a,b) = GCD(b,a) if b>a

Consider a=6, b=10.
b ≠ 0.
∴ r = 6/10 = 6. or 10/6 = 4 (since exchanged, gives same answer).

a	b	r
6	10	4
10	4	2
4	2	0
GCD ← 2	0	

a	b	r	
6	10	6	5 iterations
10	6	4	
6	4	2	
4	2	0	
GCD ← 2	0		

Pseudocode → Comb^n of syntax of programming language + English sentences.

```
Algorithm Euclids(a,b)
// To find GCD of 2 no.
// Input: 2 non negative no.
// Output: GCD of 2 numbers.
while b≠0 do
    r ← a mod b
    a ← b
    b ← r
return a
```

Algo 2. Consecutive integer checking method.

$$gcd(a,b) \leq \min(a,b)$$

Same i/p & o/p as Euclid's

Step 1: Assign $t = \min(a,b)$

Step 2: Perform the operation a mod t; if remainder is zero, go to step 3. Else, go to step 4.

Step 3: Perform b mod t.

If remainder is zero, then GCD is t. Else, Else, step 4.

Step 4: Decrement the value of 't' by 1.

Consider $a = 6$, $b = 10$.

$t = \min(a,b) = 6$.

$t = \min(a,b)$	a mod t	b mod t
6	6%6 = 0	10%6 = 4
5	6%5 = 1	-
4	6%4 = 2	-
3	6%3 = 0	10%3 = 1
2	6%2 = 0	0

∴ $t = 2$ is GCD.

In this case, no. of iterations for both Euclid's & consecutive method is same (= 3)

In case of larger numbers, Euclid's method has fewer steps.

Eg Find GCD of 54 & 36 by Euclid's & consecutive integer checking method.

$a = 54$, $b = 36$.

Euclid's

a	b	r
54	36	18
36	18	0
GCD ← 18	0	

∴ GCD is 18.

i.e, 3 iterations reqd.

Consecutive integer checking method

$t = \min(a,b) = 36$.

$t = \min(a,b)$	a mod t	b mod t
36	18	-
35	19	-
34	20	-
33	21	-
32	22	-
31	23	-
30	24	-
29	25	-
28	26	-
27	0	19
26	18 → 2	-
25	29 → 3	-
24	30 → 4	-
23	31 → 5	-
22	32 → 6	-
21	33 → 7	-
20	34 → 8	-
19	35 → 9	-
GCD ← 18	0	0

19 iterations reqd. [Usually, worst case if $\frac{1}{2} \times \min(a,b)$]

If a or b is zero, this algorithm fails.

Algo.3 Middle school method. (Brute force)

Step 1: Find prime factors of a

Step 2: Find prime factors of b

Step 3: The product of the common prime factors is the GCD

Drawback:

Does not include time to find prime factors.

Each step should take definite time.

Here, time to find prime factors is not known.

∴ does not satisfy effectiveness.

i.e, prime factorization steps are not defined unambiguously.

Also, special case has to be considered if any one of the i/ps is 1

This is because, mathematically, 1 is not considered a prime no.

∴ this, this method fails for such inputs.

Algorithm to generate prime numbers.

"Sieve of Eratosthenes".

Input: Positive integer number 'n'.

Eg. Assume n = 50.

Step 1: Store the numbers 2 to n in an array 'a'.

Step 2: Discard all multiples of 2 (but not 2). 4 first eliminated

Step 3: Discard all multiples of 3 (but not 3). 9 first eliminated

Step 4: Discard all multiples of 5 (but not 5). 25

Step 5: Discard all multiples of 7 (but not 7). 49

a[0]	a[1]								
A[9] 2	3	~~4~~	5	~~6~~	7	~~8~~	~~9~~		
~~10~~	11	~~12~~	13	~~14~~	15	~~16~~	17	~~18~~	19
~~20~~	21	~~22~~	23	~~24~~	~~25~~	~~26~~	27	~~28~~	29
~~30~~	31	~~32~~	33	~~34~~	~~35~~	~~36~~	37	~~38~~	39
			~~49~~	~~50~~					

Maximum no. of iterations for value 'n' is $\sqrt{n}$. Here n = 50, $\sqrt{n} \sim 7$.

Iterations: 2 to 7

"Discard": set to 0 or -1.

Non-discarded numbers are prime.

```
Algorithm Sieve_Eratosthenes(n)
// Generate prime numbers
// Input: An integer n, n>2
// Output: An array L of all prime no. <=n.
    for p<-2 to n do A[p]<-p  // Copy p to A[p]
    for p<-2 to √n
        if A[p] ≠ 0 // for e.g. when it reaches 4, all multiples of 4 are also multiples of 2
                    // 2 have been eliminated on a previous pass,
                    // iteration is skipped
            j<-p*p
            while j<=n do
                A[j]<-0  // or -1
                j<-j+p

    i<-0
    for p<-2 to n do
        if A[p] ≠ 0   // or A[p] ≠ -1
            L[i]<-A[p]
        i<-i+1
    return L
```

SEE ✓ common Q^n

10 marks Define algorithm. Discuss 3 algo. to find GCD.

- If 'p' is the no. whose multiples are being eliminated on the current pass, then the first multiple considered is p*p because smaller multiples 2p ... (p-1)p have been eliminated in earlier passes through the list.
- p*p should not exceed n, ∴ p cannot exceed $\sqrt{n}$.

Analysis of algorithm.

- Space on system
- Time for exec^n — Best case, Worst case, Avg. case } w.r.t. to inputs being supplied to algo

Eg. linear search.

Best: searching for 1st el.

Worst: last element / element not present in given set

Space

Efficiency of an algorithm is expressed in terms of time & space.

Space on system is in terms of → Text segment, Data segment, Stack segment.

Time efficiency depends on:

i) which system

ii) which language the prg. is written in

iii) Compiler used.

This does not give actual time efficiency.

∴ to calculate the actual time efficiency, we find the "basic operation" & find how many times the basic operation is repeated / executed.

$$T(n) = C_{op} C(n)$$

Time efficiency ↓ basic operation. → f^n expressing how many times basic op^n is executed.

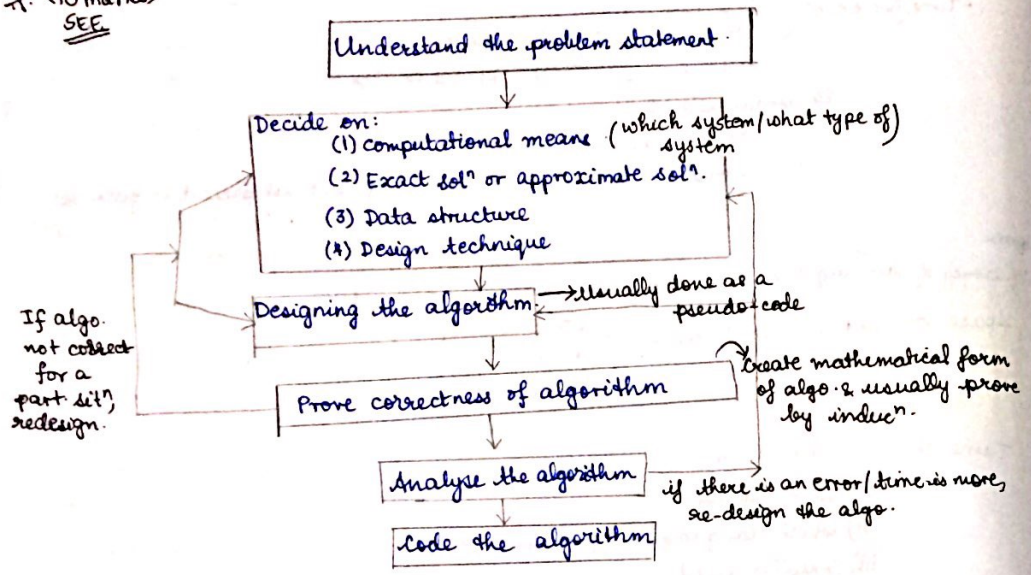
Time efficiency indicates how fast the algorithm runs

Space efficiency indicates how much extra memory the algorithm needs

Basic operation: Most imp. operation of the algorithm, the op^n contributing the most to the total running time

Process of design & analysis of an algorithm.

* <10 marks> SEE



- Algorithm design technique is a general approach to solving problems algorithmically that is applicable to a variety of problems from different areas of computing. → provide guidance for creating new algos → way to categorize & study algos
- A pseudo-code is a mixture of natural language & programming language constructs.

Since time efficiency depends on input size,
Input size is denoted by 'n'.

If a for loop runs for 1 to n, & basic op^n is within the for,
executes 'n' times,
∴ efficiency is linear. ∝ n
i.e. time efficiency ∝ input size.

In case of bubble sort:

Time efficiency: $\frac{n(n-1)}{2}$ = no. of comparisons

$= \frac{n^2-n}{2}$

Since n^2 is max, it is said to have efficiency $O(n^2)$ } uppa bound value.

Const < log n < n < n log n < n^2 < n^3 < 2^n < n!

↓ Constant value.

Order of growth.

Order of growth of n! > order of growth of n^3 > n^2.

→ Order can be verified by subs values

} efficiency class.

Eg. SEE Q^n Arrange the foll. efficiency classes in descending order

$\sqrt{n}$, 3^n, 2^{2n}, (n-2)!, 0.000001 n^3, $2n^4+1$, log n, $5\log_{10}(n+100)^{10}$

↓ n^3 ↓ n^4

Ignore constant & take only efficiency class

$(n-2)! > 2^{2n} > 3^n > n^4 > n^3 > \sqrt{n} > \underbrace{5\log_{10}(n+100)^{10}}_{5\log n} > \log n$.

If base 'e' for log n.

$> \sqrt{n} > \log_e^2 n > 5\log_{10}(n+100)^{10}$.

algorithm xyz(n)
// Input: A non-negative integer n
s ← 0
for i ← 1 to n do
s ← s + i
return s

1. What does the algorithm compute?
Ans Sum of 'n' natural numbers
2. What is the basic operation?
Ans Addition.
3. How many times the basic operation executed?
Ans 'n' times.
4. What is the efficiency class of the algorithm?
Ans: Linear efficiency class, i.e, n.

} Sub. Q^n are common CIE Q^n

```
Algorithm xyz(n)
// Input: A non-negative integer n.
S ← 0
for i ← 1 to n do
    S ← S + i * i
return S
```

Q. What does algo compute?
Sum of squares

Q. Basic operation:
Multiplication (Time taken by * > time taken by +)

Q. Basic operation is executed: 'n' times
Since * is repeated add^n, + is also accepted as long as it is in the exp^n.

Q. Efficiency class:
linear

Efficiency class

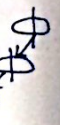
If time efficiency of algo is const · it defines the best case

n, scans whole set once

n^2, basic op^n is within 2 for loops eg. selection sort, bubble sort.

n^3, Eg: 2-D array, matrix multiplication, dynamic prg, Floyd's alg
3 for loops used.
Using divide & conquer can be reduced to $n^{2.\ldots}$

$\log n$, When prg. is progressively ~~does~~ divided, one half only solved to get sol^n.
Eg binary search.



$n \log n$, usually seen in divide & conquer
Eg. merge sort or quick sort.
Both parts of problem solved.

2^n, generating subsets to solve the given problem.
Backtracking concept uses.

$n!$, generating all permutations to solve the problem.

Decrease & conquer may be $\log n$ (usually) or $n \log n$.
Divide & conquer usually $\log n$.

$\Rightarrow$ for (i=1; i<=n; i++)
Basic operation
Efficiency class = n.

$\Rightarrow$ for (j=1; j<=n; j++)
for (i=1; i<=n ; i++)
Basic operation
Efficiency class = n^2.

$\Rightarrow$ for (i=1; i<=n; i=i*2)
Basic operation

i :	1	2	4	8	16	32	n
corresponds to :	2^0	2^1	2^2	2^3	2^4	2^5	2^k.

$\therefore n = 2^k$

$\therefore k = \log_2 n$

$\therefore$ Efficiency class is $\log_2 n$

$\Rightarrow$ for(i=1; i*i<=n; i++)
Basic operation
Efficiency class = $\sqrt{n}$.

$\Rightarrow$ for(i=1; i<=n; i++)
for (j=1; j<=i; j++)
for(k=1; k<=50; k++)
Basic operation.

i=1	i=2	i=3	.....	i=n
j=1	j=2	j=3		j=n.
k=50	k=50x2	k=50x3		k=50xn.

Efficiency class
$= 50 + 50\times2 + 50\times3 + \ldots + 50\times n$.
$= 50 (1+2+3+\ldots+n)$
$= 50 \times \frac{n(n-1)}{2}$
$= 25n^2 - 25n$.
$= n^2$

Constant values & lower efficiency class values are discarded.

⇒ for (i=1; i<=n; i++)
  for (j=1; j<=i²; j++)
    for (k=1; k<=n/2; k++)

Basic operation.

i=1	i=2	i=3	.....	i=n
j=1	j=4	j=9		j=n²
k=n/2	k=4×n/2	k=9×n/2		k=n²×n/2.

No. of times = $\frac{n}{2} + 4\times\frac{n}{2} + 9\times\frac{n}{2} + \dots n^2\times\frac{n}{2}$

$= \frac{n}{2}(1+4+9+\dots n^2)$

$= \frac{n}{2} * \frac{n(n+1)(2n+1)}{6}$

$= n^4$ as efficiency class.

⇒ for (i=1; i<=n; i++)
  for (j=1; j<=n; j=j+i)

Basic operation.

i=1	i=2	i=3	.....	i=n
j=n (1,2,3...n)	j=1,3,5.... = n/2	j=1,4,7.... = n/3		j=1

No. of iterations $= n\left(1+\frac{1}{2}+\frac{1}{3}+\frac{1}{4}+\dots\frac{1}{n}\right)$

$= n \log n$ as efficiency class.

Asymptotic notation

To compare & rank the orders of growth of a function (the function which expresses the time efficiency) the following asymptotic notations are used.

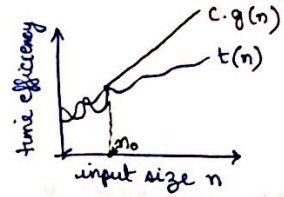
1. Big oh (O) - Specifies the upper bound (worst case of algo)
2. Omega (Ω) - Specifies the lower bound (best case of algo) ↕ only generally, do not relate
3. Theta (Θ) - Specifies the average

Big oh notation

Let $t(n)$ & $g(n)$ be the non-negative functions where $t(n)$ represents the actual time taken by the algorithm & $g(n)$ is a sample function considered for comparison.

Defn A function $t(n)$ is said to be $O(g(n))$ denoted by $t(n) \in O(g(n))$, if $t(n)$ is bounded above some positive constant multiple of $g(n)$ for all larger values of 'n', if there exists a positive integer constant 'c' and a positive integer n_0 satisfying the statement: $t(n) \le c\,g(n)\ \forall\ n \ge n_0$

$O(g(n))$ is the set of all f^n with a smaller or same order of growth as $g(n)$

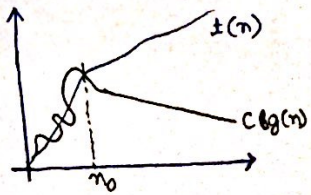


$t(n) = n^2$, $g(n) = n^3$. $n^2 = O(n^3)$. after some const. value.

Omega notation

Defn The function $t(n)$ is said to be $\Omega(g(n))$ denoted by $t(n) \in \Omega(g(n))$, if $t(n)$ is bounded below some positive constant multiples of $g(n)$ for all larger values of 'n', if there exists some positive constant 'c' & some non-negative integer 'n_0' such that $t(n) \ge c\,g(n)\ \forall\ n \ge n_0$

$\Omega(g(n))$ is the set of all f^n with a larger or same order of growth as $g(n)$

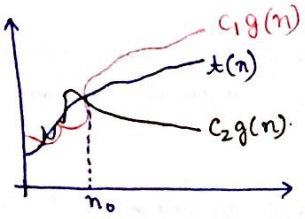


Theta notation

Defⁿ The function $t(n)$ is said to be $\Theta(g(n))$ denoted by $t(n) \in \Theta(g(n))$, if $t(n)$ is bounded both above & below by some positive constant multiples of $g(n)$ for all larger values of n if there exist some positive constants C_1 & C_2 & some non-negative integer n_0 such that

$$C_2 g(n) \leq t(n) \leq C_1 g(n) \quad \forall \; n \geq n_0$$

$\Theta(g(n))$ is the set of all fⁿ having the same order of growth as $g(n)$



$n^3 = O(n^2)$ ✗
$n^3 = O(n^3)$ ✓
$n^4 = O(2^n)$ ✓
$n^2 = \Omega(2^n)$ ✗
$n^2 = \Omega(n^2)$ ✓
$2^n = \Omega(n^2)$ ✓

Eg. Given the foll. statements & a fⁿ $f(n) = \frac{n(n+1)}{2}$, state whether they are true or false

1. $f(n) \in O(n^3)$ True.
2. $f(n) \in O(n^2)$ True
3. $f(n) \in O(n)$ False.
4. $f(n) \in \Omega(n^3)$ False
5. $f(n) \in \Omega(n)$ True

Eg. Compare fⁿ: $100n^2$ & $n(n+1)$
same efficiency class. <2 marks>

$100n^2$ & $0.01n^3$
Diff. efficiency class.

Comparison of the efficiency classes

In order to compare the two efficiency classes, we need to find the value based on the following equation.

$$\lim_{n\to\infty} \frac{t(n)}{g(n)} = \begin{cases} 0 & \text{; implies that } t(n) \text{ has a smaller order growth compared to } g(n) \text{ & Big-O notation} \\ c > 0 & \text{; implies that } t(n) \text{ has the same order growth compared to } g(n) \text{ & } \Theta \text{ notation} \\ & (c \text{ is a constant}) \\ \infty & \text{; implies that } t(n) \text{ has a larger order growth compared to } g(n) \text{ & } \Omega \text{- notation} \end{cases}$$

Q. Compare orders of growth of $\frac{1}{2}n(n-1)$ & n^2.

Ans $\lim_{n\to\infty} \frac{\frac{1}{2}n(n-1)}{n^2} = \lim_{n\to\infty} \frac{1}{2}\left(\frac{n^2-n}{n^2}\right) = \lim_{n\to\infty} \frac{1}{2}\left(1-\frac{1}{n}\right)$

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

$= \frac{1}{2}$

∴ Θ notation.

∴ $\frac{1}{2}n(n-1) \in \Theta(n^2)$ [same for $\frac{1}{2}n(n+1)$]

Q. Compare $n!$ & 2^n.

Ans $\lim_{n\to\infty} \frac{n!}{2^n}$

As per Sterling formula, $n! = \sqrt{2\pi n}\left(\frac{n}{e}\right)^n$.

gives value ∞, ∴ Ω-notation.

Eg (Recursive & non-recursive : 8-10 marks).

Mathematical Analysis of Non-Recursive Algorithm

Eg. Bubble sort

```
for i ← 1 to n-1 do
    for j ← 1 to n-i do
        if (a[j] > a[j+1])
            swap(a[j], a[j+1])
```

B/n swapping & comparison, comparison always done.
∴ comparison is basic op^n.

Find basic op^n & express as a sum, no. of times basic op^n is executed

$$\sum_{i=1}^{n-1}\sum_{j=1}^{n-i} 1 \Rightarrow \sum_{i=1}^{n-1} n-i-1+1$$

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

$$= (n-1)+(n-2)+(n-3)+\ldots\ldots n-n+1$$

$$= 1+2+3+\ldots\ldots+n-1$$

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

∴ Efficiency class $= n^2$.

Next, check if basic op^n executes diff. no. of times for diff i/p size. If yes, best, worst & avg case.

General plan for finding efficiency of non-recursive algorithm

Step 1 : Decide on the parameter or parameters indicating the input size

Step 2 : Identify the algorithm's basic operation (Located in the innermost loop)
Eg. `while (n>1) do` } comparison is basic op^n.

Step 3 : Check whether the number of times the basic operation executed depends only on the size of the input. If it also depends on some additional property, then, the worst & the average & the best case is necessary, i.e, these efficiencies must be investigated.

Step 4 : Setup a sum expressing the number of times the basic operation gets executed.

Step 5 : Using standard formula & rules of manipulation, either find a closed form for the count or atleast establish the order of growth.

Algo 1 : Algorithm to find the maximum element in an array

```
Algorithm MaxElement(A[0...n-1])
// Finds the max Element in the given array
// Input : An array A
// Output : Maximum value stored in array
    maxval ← A[0]
    for i ← 1 to n-1 do
        if A[i] > maxval
            maxval ← A[i]
    return maxval
```

~~comparison statem~~

There are two statements within the for loop, i.e, the comparison & the assignment statements. As the comparison statement will be executed more no. of times compared to the assignment, the basic operation of the above algorithm is comparison.

Let C(n) denotes the no. of times the basic operation gets executed & it can be expressed as $C(n) = \sum_{i=1}^{n-1} 1$

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

$$= n-1$$

$$C(n) = \Theta(n).$$

↖ irrespective of input size 'n', (n-1) comparisons done since efficiency class is same for diff. i/p size, expressed in terms of Θ

Algo 2 : To find the uniqueness of an element in an array

```
Algorithm ElementUniqueness(A[0....n-1])
// Returns true if all elements are unique/distinct. If repetition false, i.e checks if elements are unique
// Input : An array A
// Output : Returns true if all elements are unique, otherwise false.
    for i ← 0 to (n-2) do
        for j ← i+1 to (n-1) do
            if A[i] = A[j]
            return false
    return true
```

The basic op^n of the algorithm is the comparison
The no. of times the basic op^n gets executed depends on the type of input that we are supplying.

In the worst case, the comparison statement will be executed max. no. of times if all elements are unique OR last & last-but-one elements are same.

If C(n) denotes the no. of times the ... it can be expressed as $C_{worst}(n) = \sum_{i=0}^{n-2} \sum_{j=i+1}^{n-1} 1$ — inclusive prop.

$$= \sum_{i=0}^{n-2} n-1-(i+1)+1 = \sum_{i=0}^{n-2} n-1-i-1+1 = (n-1)+(n-2)+(n-3)+\cdots$$

$$= (n+1)+(n+2)+(n+3)+\cdots+(n+n-2+1)$$

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

$$= O(n^2)$$

Consider the following algorithm.

```
Algorithm XYZ(n)
// Input: A positive integer 'n'
// Output = ?
    count ← 1
while n>1 do
        count ← count + 1
        n ← ⌊n/2⌋
    return count
```

1. What does the algorithm compute?
2. What is the basic op^n?
3. What is the efficiency class of the algorithm?

Ans.

1. Finds the no. of digits in binary representation
   i.e, $\log_2 n$ value.

   Eg. n = 8 (1000)

   c = 1 (n = 4) c = 2 (n = 2) c = 3 (n = 1) c = 4

   The algorithm returns the no. of bits in the binary representation

2. The basic operation is the comparison. (always exec^d, irrespective of 'n' value)

3. $\log n$
   (Problem halved in each sequence)

Step 1: Decide on the parameter/parameters indicating the input size 'n'.
Step 2: Identify the basic operation of the algorithm.
Step 3: Check whether the number of times the basic operation is executed can vary on different inputs of the same size.
If so, we need to find the best, worst, & average case efficiencies

Step 4: Set up a recurrance relation with an appropriate base case, i.e. initial condition, for the number of times the basic operation gets executed.

Step 5: Solve the recurrance relation with appropriate methods to find the time complexity.

Methods
1. Substitution — Forward / Backward
2. Recursive tree
3. Master theorem.

Algo 1:

```
Algorithm Fact(n)
// To find the factorial of a given number
// Input: A positive integer 'n'.
// Output: Factorial of the given number.
    if n=0 return 1.
    return n * Fact(n-1)
```

The basic operation is multiplication

Let M(n) denotes the number of times the multiplication operation is executed by the algorithm.

for n=0, M(0)=0 ; base condition

M(0)=0 is the base condition, and multiplication is not performed.

M(n)

$$M(n) = M(n-1)+1 \quad \forall\ n \geq 1.$$

↳ multiplication is performed once more to find n! from (n-1)!

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

$$= M(n-3)+3$$

$$\vdots$$

$$M(n) = M(n-k)+k$$

if n-k = 0, i.e, n=k

$$M(n) = M(0)+n = 0+n$$

$$M(n) = n$$

∴ this algorithm has efficiency of order 'n'.

Algo 2: To solve the tower of Hanoi problem

```
Algorithm TowerofHanoi(n, S, D, T)
//
//Input
//Output
if n=1
    move a disc from S to D
else
    TowerofHanoi(n-1, S, T, D)
    move nth disc from S to D
    TowerofHanoi(n-1, T, D, S)
```

The basic operation in the Tower of Hanoi problem is the movement of a disc.

Let M(n) denotes the no. of moves

The base case is if n = 1

$M(1) = 1$

otherwise, (moving n^{th} disc from S to D.)

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

($M(n-1)$: moves to shift n-1 discs from S to T; $M(n-1)$: moves to shift n-1 discs from T to D)

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

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

$= 4M(n-2)+2+1$

$= 4(2M(n-3)+1)+2+1$

$= 8M(n-3)+4+2+1$

$= 2^3 M(n-3) + 2^2 + 2^1 + 2^0$

3 discs → 7 moves
4 discs → 15 moves

$M(n) = 2^k M(n-k) + 2^{k-1} + 2^{k-2} + \dots$ ①

$M(1) = 1$

$n-k = 1$

$k = n-1$

$M(n) = 2^{n-1} M(1) + 2^{n-2} + 2^{n-3} + \dots + 2^{n-n}$

$= 2^{n-1} + 2^{n-2} + 2^{n-3} + \dots + 2^0$

$= 2^n - 1$

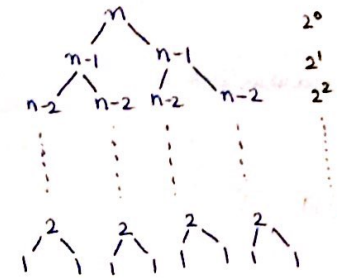
$= O(2^n)$

Order of efficiency $= 2^n$

Recursive Tree method

Let no of discs = n

RECURSIVE TREE ⇒



Level of root node = 0

No. of moves = sum of nodes in tree which depends on level

$$M(n) = \sum_{l=0}^{n-1} 2^l$$

The efficiency of Towers of Hanoi can be expressed with a recursive tree as shown above.

{ homogeneous, order 1 of recurrence rel^n → recursive tree ✓.

The efficiency can be calculated by counting the no. of nodes in the tree

i.e., $M(n) = \sum_{l=0}^{n-1} 2^l$

(where, 'l' denotes the level of a tree)

$$M(n) = \sum_{l=0}^{n-1} 2^l = 2^n - 1.$$

Algo 3: To count the no. of binary digits for a given number.

```
Algorithm BitCount(n)
//
//Input: A positive integer n, greater than zero
//Output:
if n=1 return 1
   else
      return BitCount(⌊n/2⌋)+1
```

The basic operation is addition.

Let A(n) denotes the no. of times addition is performed.

Base case:

if n=1 then, no. of additions is zero.

i.e, $A(1)=0$

Otherwise,

$$A(n) = A(n/2)+1$$

$$= A\left(\frac{n}{4}\right)+1+1$$

$$= A\left(\frac{n}{8}\right)+1+1+1$$

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

$A(1)=0$.

$\frac{n}{2^k}=1$.

$n=2^k$.

$k=\log_2 n$

$A(n) = A(1)+\log_2 n$.

$A(n) = 0+\log_2 n$

$\therefore A(n) = \log_2 n$.

Algo 4: To find n^{th} Fibonacci number

```
Algorithm Fibonacci(n)
//
//Input:
//Output:
if n≤1 return n
return Fibonacci(n-1)+Fibonacci(n-2)
```

The basic operation of the algorithm is addition

Let A(n) be no. of times addⁿ is done

Base case:

if n=0 or n=1, no addⁿ performed.

A(0)=0

A(1)=0.

Otherwise,

$A(n) = A(n-1)+A(n-2)+1$ → current addⁿ.

to find Fib(n-1) to find Fib(n-1)

$\left(\frac{1\pm\sqrt{5}}{2}\right)^n$ expanding

H.W Solve recurrance relation & find efficiency class

$A(n) = A(n-1)+A(n-2)+1$.

$r^n - r^{n-1} - r^{n-2} = 1$.

$r^2 - r - 1 = 0$

$r = \frac{1\pm\sqrt{5}}{2}$

$a_n - a_{n-1} - a_{n-2} = 1$.

$r^2 - r - 1 = 1$

$r = AC^n = A1^n$.

~~$A^n - A^{n-1} - A^{n-2} = 1$~~.

$A^2 - A - 1 = 1$

$A^2 - A = 2$.

$A^2 - A - 2 = 0$.

$A=2,\ A=-1$.

Any recurrence $T(n) = aT\left(\frac{n}{b}\right) + f(n)$ of this form can be done by Master Theorem

Master Theorem

The master theorem method applies to the recurrances of the type
$T(n) = aT\left(\frac{n}{b}\right) + f(n)$, where, $a \geq 1$, $b > 1$ & $f(n)$ is asymptotically positive.

Three common cases are:

i) $f(n) = O(n^{\log_b a - \epsilon})$ for some constant epsilon $\epsilon > 0$.
$f(n)$ grows polynomically slower than $n^{\log_b a}$ by an n^{ϵ} factor, then solution is
$T(n) = \Theta(n^{\log_b a})$

ii) $f(n) = \Theta(n^{\log_b a} (\log n)^k)$ for some $k \geq 0$, $f(n)$ and $n^{\log_b a}$ grows at similar rate, then solution is $T(n) = \Theta(n^{\log_b a} (\log n)^{k+1})$

iii) $f(n) = \Omega(\log_b a + \epsilon)$ for some constant $\epsilon > 0$, $f(n)$ grows polynomically faster than $n^{\log_b a}$ by a factor n^{ϵ}. then, $T(n) = \Theta(f(n))$.

I. $T(n) = 4T(n/2) + n$.
$a = 4$
$b = 2$.
$n^{\log_b a} = n^{\log_2 4} = n^2$.
$\therefore T(n) = \Theta(n^2)$

II $T(n) = 4T(n/2) + n^2$.
$n^{\log_b a} = n^2 . = n^2 (\log n)^0$
$f(n)$ & $n^{\log_b a}$ have same growth rate.
$\therefore T(n) = \Theta(n^2 (\log n)^{k+1})$
$(\log n)^{k+1} = (\log n)$
$k+1 = 1$
$\therefore k = 0$.

III $T(n) = 4T\left(\frac{n}{2}\right) + n^3$.
$n^{\log_2 4} = n^2$
$f(n) > n^2$
$\therefore \Theta(f(n)) = \Theta(n^3)$

IV $T(n) = T\left(\frac{2n}{3}\right) + 1$
$1 = n^{\log_{3/2} 1}$ $a = 1$, $b = 3/2$
$f(n) \alpha n^{\log_{\frac{3}{2}} b}$

$f(n) = 1$
$= 1 (\log n)^0$
$\therefore k = 0$.

$\therefore T(n) = \Theta(\log n)$

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

$\frac{n^2}{\log n}$.

$n^{\log_2 4} = n^2$.

$\frac{n^2}{\log n} = n^2 (\log n)^{-1}$.
$k = -1$.
$\therefore T(n) = \Theta(n^2 (\log n)^{-1+1})$
$= \Theta(n^2)$
$\therefore$ Cannot be sorted since k must be ≥ 0

Solve using substitution.

1. $T(n) = T(n-1) + n$.
$T(n) = T(n-2) + 2n$.
$= T(n-k) + kn$.
at n,
$T(n) = T(n-n) + n^2 . = T(0) + n^2 = n^2$.

$T(0) = 0$ as base case.
$T(n-1) = T(n-2) + n$.

2. $T(n) = 2T(n/2) + 2$
$= 4T(n/4) + 4 + 2$
$= 8T(n/8) + 8 + 4 + 2$
$= 2^3 T\left(\frac{n}{2^3}\right) + 2^4 - 2$
$= 2^k T\left(\frac{n}{2^k}\right) + 2^{k+1} - 2$

$T(1) = 0$, $n = 2^k$, $T\left(\frac{n}{2^k}\right) = 0$.

$2^n . T(1) + 2(n-1)$
Efficiency class = n.
$O(n)$.
$(n-1)$, $\therefore \Theta(n)$.

Empirical analysis

General plan for empirical analysis of an algorithm are as follows:

i) Understand the experiment purpose

ii) Decide on the efficiency matrix 'm' to be measured and measurement unit (Operation count)

iii) Decide on the characteristics of input sample (size)

iv) Prepare a program implementing the algorithm for experimentation

v) Generate a sample of inputs

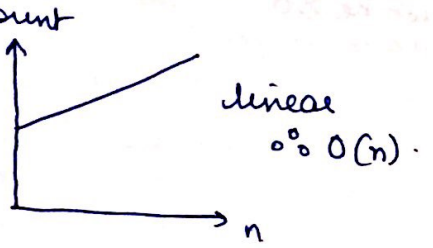
vi) Run the program on the sample inputs and record the data observed.

vii) Analyse the data obtained.

viii) To analyse the output, a graph needs to be plotted wherein 'x' axis denotes input size 'n' & y denotes the output.

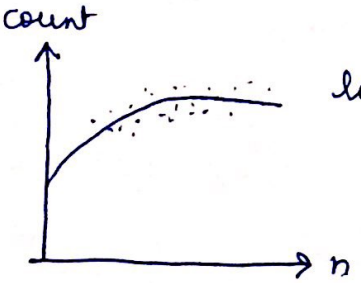
Some of the observations made on plotting the graphs are:

i)



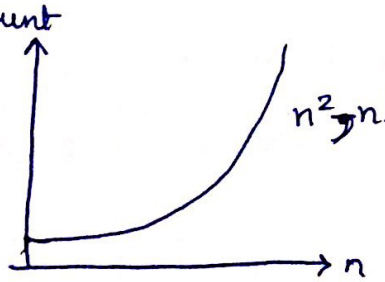
linear
∴ O(n).

ii)



logarithmic in nature
Concave shape.

iii)



n^2, nlogn

Convex shape.