Introduction (Youtube)

Algorithm: Its a peoccesse method of solving a problem by following a set of rules. It is informal way of writing a peogram.

An algorithm is said to be efficient if it consumes less time and less memory.

Six defnt Algorithm is a method to solve a problem which consist of sequence of computational steps that transform input to output.

Properties of good algorithm

1) i) Finiteness
   ii) Definiteness
   iii) Effectiveness
   4) Easy inputting (input)
   5) clear output
   6) Flexibility

2) Each instruction should be defined clearly

3) Each step should take defined time i,e time bound

4) Should be capable of modifying for better results

5) Should terminate.

Example To find GCD of 2 numbers by Eudid's algorithm.

Sol

Input: Take two +ve integers
Output: The largest number which divides a & b

Step1: If b is zero then GCD is a else goto step 2

Step2: Find r which is remainder of 'a' divided by 'b'

Step3: Exchange a by b and b by r and goto step 1

Algorithm: Euclids (a, b)
// input - 2 +ve no
// output - Greatest common divisor of a & b

while $b \neq 0$ do
    $r \leftarrow a \% b$
    $a \leftarrow b$
    $b \leftarrow r$
return a

(2) Consecutive integer checking method.

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

Step1: If $a = 0$ then gcd b or $b = 0$, gcd is a

Step2: Assign t = minimum (a, b)

Step3: Perform a mod t, if remainder is zero go to to Step 4 else step 5

Euclids ~~three~~ algorithm is efficient than consecutive integer checking method.

Middle school method:

Step 1: Find prime factor of a
Step 2: Find prime factors of b
Step 3: Product of common prime factors is GCD

Effective-ness is not present.

Algorithms to generate prime numbers
// input: a +ve integer 'n'
// output: list of prime no from 2 to n

Step 1: Store the numbers from 2 to n in array
Step 2:- Discard all the multiples of 2 but not 2
Step 3:- " " " " 3 but not 3
" " " 4 but not 4
5 but not 5
7 but not 7
⋮

No of iteration for n is $\sqrt{n}$

Algorithm
// input: n
// output: list of prime no from n

```
for p ← 2 to n do A[p] ← p
for p ← 2 to √n do
    if A[p] ≠ 0
        j ← p * p
    while (j ≤ n) do
        A[j] ← 0
        j ← j + p
i ← 0  // initialization
for p ← 2 to n do
    if (A[p] ≠ 0)
        L[i] ← A[p]  // L is array of prime no
        i ← i + 1
return L
```

Analysing of algorithm

~~i) Efficiency~~ Efficiency — storage
— time

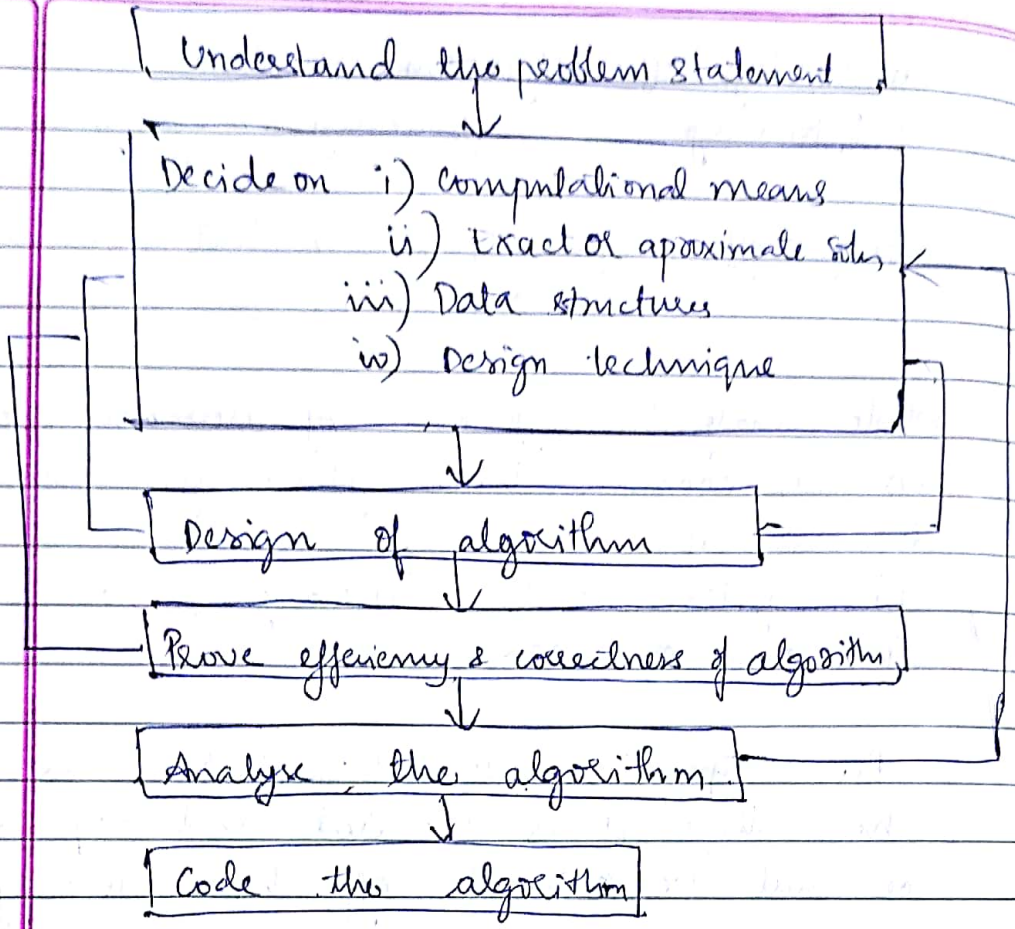
i) Storage efficiency: depends on text segments, data segments etc

While analysing efficiency of algorithm we mainly consider time efficiency which depends on
(i) System used ii) Language / OS
iii) Compiler used.

Basic operation is determined as we see how time it is executed and depending on that we calculate ~~eff~~ time efficiency.

✱ Process of design and analysis of algorithms

Step 1: To understand problem statement
Step 2: Decide on i) Computational means
ii) Extenent or property of result (approx or exact)
iii) Data structures to be used.
iv) Designing the ~~algorithms~~ technique
Step 3 Designing algorithms.



Algorithm design technique is a general approach to solve problem algorithmically i/e it is applicable to vareity of problems from different area of computat -ions.

A pseudo code is a mixture of natural language and progeraming language.

★ Order of growth of algorithms

Efficiency mainly depends on the size which is usually denoted by 'n'

$\log(n) < n < n\log(n) < n^2 < n^3 < 2^n < n!$

Q1) Arrange the following efficiency classes in descending order.

$\sqrt{n}$, 3^n, 2^{2n}, $n-2!$, $0.0001n^3$, $2n^4+1$, $\log n$ & $5\log_{10}(n+100)^{10}$

Sol

$n-2! > 2^{2n} > 3^n > 2n^4+1 > 0.0001n^3 > \sqrt{n} > 5\log_{10}(n+100)^{10} > \log n$

Q2) Algorithm xyz(n)
// input: +ve integer
s ← 0
for i ← 1 to n do
s ← s + i
return s

i) What does the algorithm do?
Sum of n natural numbers

ii) What is the basic operation?
s ← s + i

iii) How many times the basic operation is executed?
n times

iv) What is the efficiency class of algorithm
linear i.e. n

If '+' and '*' are present in same operation then we take multiplication.

Asymptotic notation

Asymptotic notation are languages that allow us to analyze an algorithm's efficiency by varying the input size of algorithm. This is also known as growth rate.

NOTE:

1) n - linear
2) n^2 - 2 loops
3) n^3 - 3 loops
4) log n and n log n - divide & conquer.
5) for (i = val; i<=n; i = i*2) ⇒ efficiency class $\log_2 n$
6) for (i = 1; i^2 <= n; i++) $\sqrt{n}$

To compare and rank the orders of growth of a function

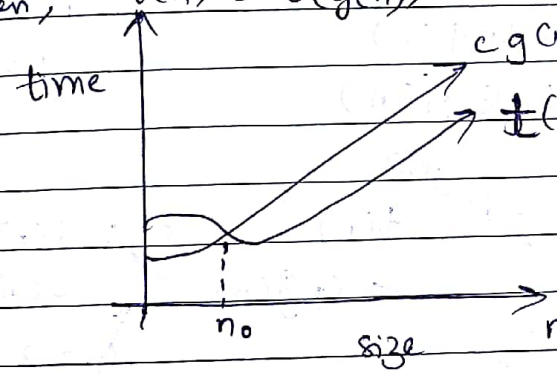
The function which expresses the time efficiency the following asymptotic notation are used :-

i) Big oh (O) - upper bound specifies (worst)
ii) Omega (Ω) - lower bound specifies (best)
iii) theta (Θ) - Average.

Let t(n) and g(n) be the non-negative functions where t(n) represents the actual time taken by the algorithm and g(n) is sample function considered for comparisons

1) 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 constant +ve multiplication of $g(n)$ for large values of n if there exist a +ve integer constant c & +ve integer n_0 satisfying the statment $t(n) \leq c \cdot g(n)$ $\forall$ $n \geq n_0$ & $c > 0$ & $n_0 \geq 1$

Then, $t(n) = O(g(n))$



eg: $t(n) = 3n+2$

$g(n) = n$

$3n+2 \leq c \cdot n$ ($c=4$)

$3n+2 \leq 4n$

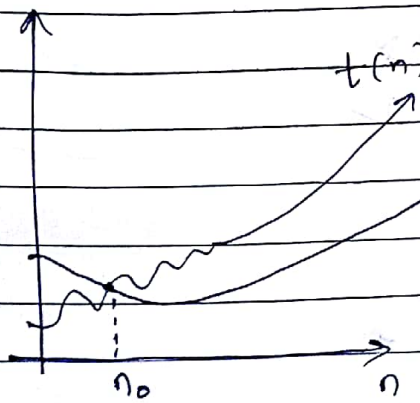
$n \geq 2$

$\therefore$ $\forall$ $n \geq 2$ & $c \geq 3$

then, $t(n) = O(g(n))$

2) A function $t(n)$ is said to be $\Omega(g(n))$ if $t(n)$ is bound below by some +ve integer constant multiples of $g(n)$ (ie $(c \cdot g(n))$) for all larger values of n. If there exist a +ve value c & +ve n_0 such that

$$t(n) \geq c \cdot g(n) \quad \forall \; n_0 \geq 1 \;\&\; n \geq n_0, \; c > 0.$$



eg: $t(n) = 3n+2$, $g(n) = n$

$t(n) \geq c \cdot g(n)$; $\forall$ $c < 3$

$3n+2 \geq 2 \cdot n$

$n \geq -2$

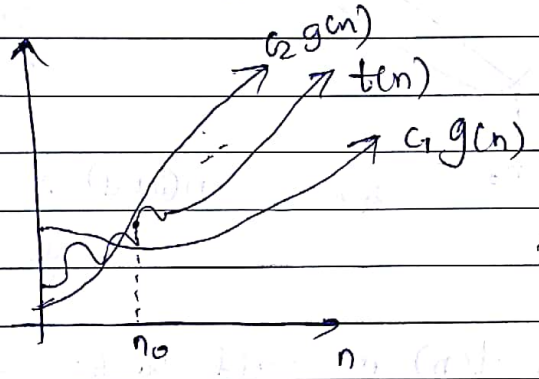
$\therefore$ $t(n) = \Omega(g(n))$

3) A function $t(n) = \Theta(g(n))$ if $t(n)$ is bounded by $g(n)$ above and below by some +ve constant multiples of $g(n)$ for all large values of n if there exists some +ve constant c_1 & c_2 & +ve n_0 then,

$$c_1 g(n) \leq t(n) \leq c_2 g(n) \quad \forall \; c_1 \& c_2 > 0$$
$$n \geq n_0$$
$$n_0 \geq 1$$

OR

$$t(n) \geq c_1 g(n)$$
$$t(n) \leq c_2 g(n)$$



eg: $t(n) = 3n+2$

$g(n) = n$

$c_1 = 2$ & $c_2 = 4$

then,

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

1) Given, the following statements and a function $f(n) = \frac{n(n+1)}{2}$. State true or false

i) $f(n) \in O(n^3)$ ✓
ii) $f(n) \in O(n^2)$ ✓
iii) $f(n) \in O(n)$ ✗
iv) $f(n) \in \Omega(n^3)$ ✗
v) $f(n) \in \Omega(n)$ ✓

Comparision of efficiency classes

In order to compare 2 efficiency class we need to find the value based on following equation.

$$\lim_{n \to \infty} \frac{t(n)}{g(n)} = \begin{cases} 0 ; & \Rightarrow O(t(n)) \\ c>0, & \Theta(t(n)) \\ \infty ; & \Omega(t(n)) \end{cases}$$

i) If result is 0, then $t(n)$ has smaller order growth than $g(n)$

ii) If result is c, then $t(n)$ has same growth order compared to $g(n)$

iii) If result is ∞, then $t(n)$ has larger grown than $g(n)$

1) Compare $\frac{1}{2} n(n-1)$ & n^3

Sol)

$$\lim_{n \to \infty} \frac{\frac{1}{2} n(n-1)}{n^3} = \lim_{n \to \infty} \frac{\frac{1}{2} n^2 - n}{2n^3} = \lim_{n \to \infty} \frac{1}{2n} - \frac{1}{2n^2}$$

$$= 0$$

$$\therefore \frac{1}{2}(n-1) = O(n^3)$$

2) Compare $n!$ & 2^n

$$\lim_{n \to \infty} \frac{n!}{2^n} = \frac{\sqrt{2\pi n}\left(\frac{n}{e}\right)^n}{2^n}$$

$\because n! = \sqrt{2\pi n}\left(\frac{n}{e}\right)^n$

$$\lim_{n \to \infty} \sqrt{2\pi n}\left(\frac{n}{2e}\right)^n$$

$$n! = \Omega(2^n)$$ //

Done

Mathematical Analysis of non-recursive algorithm.

Eg: Bubble sort

Algo: for $i \leftarrow 1$ to $n-1$ do
    for $j \leftarrow 1$ to $n-i$ do
        if $(a[j] > a[j+1])$
            swap $(a[j], a[j+1])$

Basic step if statement

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

$$= n-1 + n-2 + \cdots + n-n+1$$

$$= 1+2+3+\cdots+n-1 \quad //$$

$$= n(n-1)/2$$

The general plan for finding efficiency of non recursive algo:-

Step1: Decided on the parameters indicating the input size.

Step2: Identify the algorithms basic operation (located in the inner most loop)

Step3: The check wether the number of times the basic operation executed depends only on size of the input. If it also depends on some additional properties. then the worst and the average

and the best case is ~~necessary. of~~ calculated.

Step 4: Set up a ~~sum~~ sum expressing the number of times the basic operation gets excuted.

Step 5: Using standard formula & rule of manuplation either find a closed form for the count or atleast establish oder of the growth.

① Algorithm to find the maximum element in an array.

— Algorithm MaxElement (A[0...n-1])
// Find the max element in the given array.
// Input: elements of array
// output: max element of array.

max value ← A[0]
for i ← 1 to n-1
    if A[i] > max val (1)
        maxvalue ← A[i] (2)
return max val

There are two statments with in the for loop i/e the comparison (1) and assignment statent (2) as the compasion statement will be excuted more no of times compared to assignment. The basic operation of algo is comparision.

Let C(n) denotes the no of times the basic operation gets excuted and it can be expressed has

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

$$= n-1 \not{+1} \not{+1}$$

$$= n-1 //$$

$\therefore C(n) = \Theta(n)$ (∵ worst & best case are same)

eg: 30 20 10
10 30 20
10 20 30

(∵ Average case, efficiency is independent of input)

(2) To find the uniqueness of an element in an array (no element should be repeated)

Soln

```
Algorithm Element Uniqueness (A[0...n-1])
// check wether all element are distinct or not
// Input : An array
// Output : Return true if all elements
            are unique else 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 operation of algorithm is the comparission statement.

The number of times the basic operation gets excuted depends on type of input that we are supplying. In the worst case the comparision statement will be excuted maximum no of time if all the elements unique or last 2 elements are same

If C(n) denotes the no of times the basic operth excuted. It can be expressed has.

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

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

$$= n-1 + n-2 + \cdots + 1$$

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

$$\boxed{C_{worst}(n) = \in O(n^2)}$$

(3) Consider the following algorithm.

Algorithm XYZ(n) → name.

Input : A +ve integer n

Output : ?

```
count ← 1
while n>1 do
    count ← count+1
    n ← ⌊n/2⌋
return count.
```

* Dividing ⇒ efficiency is log n

a) What does the algorithm computes?
— The algo returns the no of bits in the ~~numb~~ binary representation.

b) What is basic operation.
— · The basic operation comparision ~~op~~-statement (while

3) What is efficiency class of algorithm.
— log (n) (∵ Divide and conquer)

The general plan to & analize the effeiency of recursive algorithm.

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

Step 2: Identify the basic operation of the algorithm number of times

Step 3: Check whether the ~~basic~~ operation is executed can vary on different input of same size if so we need to find the best, worst and average case effeiency.

Step 4: Setup a recurrence relation with an appriate base case or initial condition for the number of times the basic operation gets executed.

Step 5: Solve the recurrence relation with appriate methods to find time complexity

Methods

(1) Substitution → forward substitution
→ Backward substitution. (Mainly used)

(2) Recursive tree

(3) Master theorem.

Substitution method.

① Factorial of number

Sol

Algorithm Fact(n)
// Factorial of given number
// Input : A +ve integer n
// output : factorial of given number.

if n = 0 return 1
else return n * Fact(n-1)

The Basic operation is multiplication.

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

i/e $n = 0 \Rightarrow M(0) = 0$, base condition
$n \neq 0 \Rightarrow M(n) = M(n-1) + 1 \quad \forall\ n \geq 1$

$\Rightarrow M(n) = M(n-1) + 1$
$= M(n-2) + 1 + 1$
$= M(n-3) + 1 + 1 + 1$
$= M(n-k) + k$ in general.

w.k.t, $M(0) = 0$

$\therefore M(n) = M(0) + n \quad \because k = n$

$M(n) = n \Rightarrow O(n)$ //

(2) Algorithm Towers of Hanoi (n, source, destin, temp)
// input :
// output :

if n = 1
    move a disc from S to D
else
    Towers of Hanoi (n-1, S, T, D)
    Move n^{th} disc from S to D
    Towers of Hanoi (n-1, T, D, S)

The basic operation is Towers of Hanoi Problem is movement of a disc

Let M(n) denotes the number of movements

If n=1, M(1) = 1

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

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

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

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

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

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

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

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

$M(1) = 1 \Rightarrow n-K = 1$

$\Rightarrow K = n-1$

$= 2^{n-1} \times M(1) + 2^{n-2} + 2^{n-3} + \dots + 2^0$

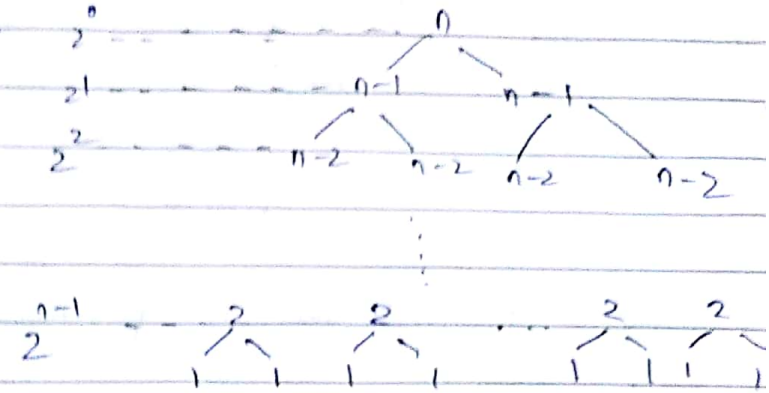
$= 2^{n-1} + 2^{n-2} + 2^{n-3} + \dots + 2^0$

$= 2^n - 1$

$= O(2^n)$

The efficiency of towers of Hanoi problem can be expressed with

Recursive tree



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

The efficiency can be calculated, counting number of nodes in the tree

i/e $M(n) = \sum_{l=0}^{n-1} 2^l$ (where l = level of tree)

$= 2^n - 1$ //

(3) To count no of binary bits.

Algorithm bitcount(n)
//
// Input : +ve integer
//
if n=1 return 1
else
return bitcount($\lfloor n/2 \rfloor + 1$)

The basic operation is addition

Let A(n) denotes the total no of addition

Base case : n=1, A(1) = 0

otherwise

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

$= A(n/4) + 1 + 1$

$= A(n/8) + 1 + 1 + 1 \ldots$

$A(n) = A(n/2^k) + k$

$n = 2^k$

$k = \log_2 n$

$A(n) = \log_2 n$ //

(4) To find the n^{th} fibinocci number.

Algorithm Fibinacci (n)

//

//

//

if $n \leq 1$ return n

return fib(n-1) + fib(n-2)

The basic operation is addition.

Let A(n) be the total no of addition.

base case, $n = 0 \Rightarrow A(0) = 0$

$n = 1 \Rightarrow A(1) = 0$

otherwise,

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

Solve recurrence relation part, $A(n) = r^n$

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

$r = \frac{1}{r} + \frac{1}{r^2} + x$

$r^2 + r = 0$

$r(r+1) = 0$

$r = 0, -1$

Master theorem

Applies to the recurrence of type

$$T(n) = aT(n/b) + f(n)$$

where, $a \geq 1$, $b > 1$ & $f(n)$ is asymptotically +ve

cases

(1) $f(n) = O(n^{\log_b a - \epsilon})$ for some constant $\epsilon > 0$ $f(n)$ grows polynomically slower than $n^{(\log_b a)}$ by a n^{ϵ} factor. Then,

$$T(n) = \theta(n^{\log_b a})$$

(2) $f(n) = \theta(n^{\log_b a} \cdot \log^k n)$ for $k \geq 0$ $f(n)$ & $n^{\log_b a}$ grows at similar rate then,

$$T(n) = \theta(n^{\log_b a} \cdot \log^{k+1} n)$$

(3) $f(n) = \Omega(\log_b^a + \epsilon)$ for $\epsilon > 0$ $f(n)$ grows polynomically faster than $n^{\log_b a}$ by a n^{ϵ} factor, then the solution

$$T(n) = \theta(f(n))$$

//

$\log^k n = \log^0 n : \log n$ $\Rightarrow k = 0$

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

$\Rightarrow n^{\log_a^b} \Rightarrow n^{\log_2^4} = n^2$

$\Rightarrow T(n) = \theta(n^2)$ Case (1)

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

$n^{\log_b^a} = n^2$ & $f(n) = n^2$

$T(n) = \theta(n^2 \cdot \log^{k+1} n)$ $k > 0$

$= \theta(n^2 \cdot \log n)$ Case (2)

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

$n^{\log_b^a} = n^2$ & $f(n) = n^3$

$T(n) = \theta(n^3)$ Case (3)

(4) $T(n) = T(3n/2) + 1$

$a = 1 ; b = 3/2$

$n^{\log_{3/2}^1} = 1$

Case (2)

$T(n) = \theta(1 \times \log n)$

(3) $T(n) = 4T(n/2) + \frac{n^2}{\log n}$

Sol $T(n) = 4T(n/2) + (\log n)^{-1} \cdot n^2$

$\Rightarrow k = 1$ ~~$k \geq 0$~~ ∵ K should be greater than or $k \geq 0$

No Solution

if $T(n) = 4T(n/2) + n^2 \log n$
then it will come under case 2

Solve the following recurrence by substitution.

1) $T(n) = T(n-1) + 1 \quad ; T(1) = 1$

$= T(n-2) + 1 + 1 \qquad n-k = 1$
$= T(n-k) + k \qquad k = n-1$
$= 1 + n - 1$
$= n$

$T(n) = T(n-1) + n \qquad T(1) = 1$
$= T(n-2) + n-1 + n$
$= T(n-3) + n-2 + n-1 + n$
$= T(n-k) + kn - (k-1)$
$= 1 + (n-1)n - (n-1)$

2) $T(n) = T(n/2) + 2 \quad ; \quad T(1) = 0$

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

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

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

$= 0 + 2n$

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

$= 2(2 \times T(n/4) + 2) + 2$

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

$= 2^k T(n/2^k) + \sum_{i=1}^{k} 2^i \qquad T(1) = 0$

$= 2^{\log_2 n} \times 0 + 2.(2^k - 1) \qquad n = 2^k, \; k = \log_2 n$

$= O(n)$

Emperical Analysis

General plan for empeical analysis has pl

Step 1 Understand the expeiment purpose.

Step 2 Decide on the efficiency matrix 'M' to be measured and the measurement unit (operation count)

Step 3: Decide on the characteristic of the input samp

Step 4: Prepare a program implementing the algorithm for the experimentation.

Step 5 Generate a sample of inputs

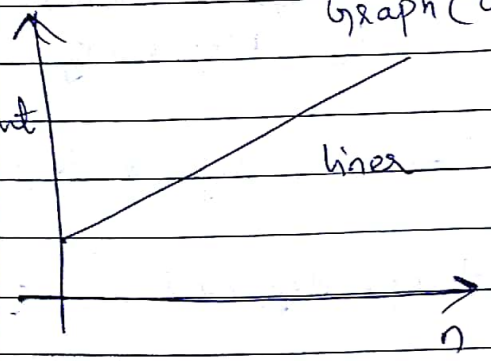
Step 6 Run the program on the sample inputs and record the data observed

Step 7 Analyse the data obtained.

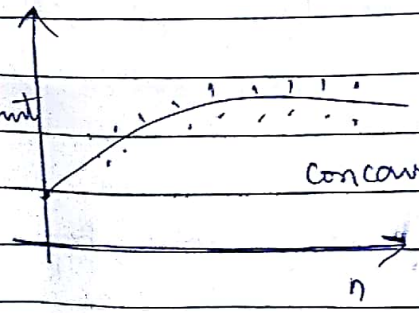
To analyse the output a graph needs to be plotted where in x axis denote the input size and y axis denotes the output (count of basic operation)

Some of the observations made on plotting the graphs are:

(i) Graph (catesplot) Effecieny is order of n

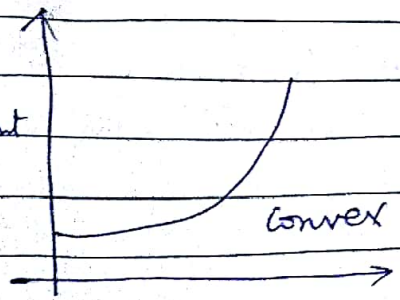


(ii) log(n)



i/e logarithamic in nature

(iii)



n log n
n^2
n^3