1/01/19

Unit - I

Introduction

- Why do we need to study algorithms?

→ Theoretical stand point - Algorithms → corner ston of CS.
→ Practical stand point.

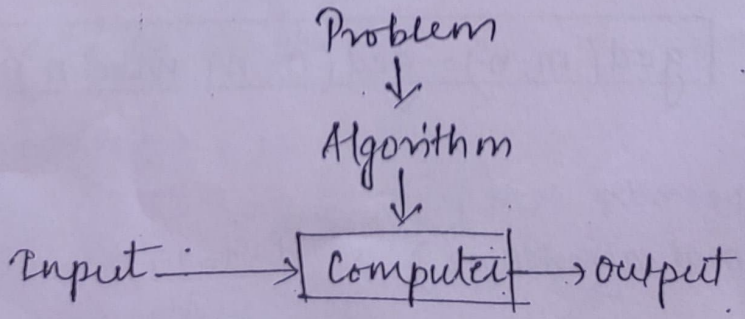
Study of algorithms is called algorithmics.

→ Standard set of algorithms
new algorithm and analyse
* time complexity.
* space complexity.

ALGORITHM: It is a sequence of unambiguous instruction for solving a problem i.e. for obtaining a required o/p for any legitimate i/p in finite amount of time.

Properties of algorithms:

1) Finiteness - Terminates after finite no. of steps.
2) Definiteness - Each step must be unambiguously specified.
3) Input - Valid i/p must be clearly specified.
4) Output - The data that results upon completion must be specified.
5) Effectiveness - Steps must be sufficiently simple & basic.



- Finding Gcd of two numbers using Euclid Algorithm:

Step 1: If $n=0$, return m as answer and stop, otherwise proceed to step 2.

Step 2: Divide m by n and assign the value of remainder to r.

Step 3: Assign the value of n to m and r to n, goto step 1.

Eg: gcd(48, 18)

Here, $m=48$ and $n=18$.

$$\begin{array}{r} 18)48(2 \\ \underline{36} \\ 12 \end{array}$$

i.e, $r=12$, $m=18$ and $n=12$.

Again check $n=0$, not zero

$$\begin{array}{r} \therefore \quad 12)18(1 \\ \underline{12} \\ 6 \end{array}$$

$\therefore r=6$, $m=12$ and $n=6$.

Again, $$\begin{array}{r} 6)12(2 \\ \underline{12} \\ r=0. \end{array}$$

Now $m=6$ and $n=0$.

$\therefore$ gcd(48, 18) = 6.

$\therefore$ gcd(48, 18) = gcd(18, 12)
= gcd(12, 6)
= gcd(6, 0).
↓ gcd.
return 6.

$$\boxed{gcd(m,n) = gcd(n, m \bmod n)}$$

- ALGORITHM {pseudo code}
(Formal algorithm.)

[illegible]

Algorithm Euclid(m, n)
// Compute gcd(m, n) by Euclid method.
// Input: two non-negative, not both zero integers m and n.
// Output: greatest common divisor of m and n.

```
while(n≠0) do
    r ← m mod n
    m ← n
    n ← r
return m.
```

r < m

Eg: gcd(78, 57)

Here, m = 78 and n = 57.

```
57)78(1
  -57
  ---
r= 21 ,  m = 57 and n = 21.
```

Now again,

```
21)57(2
  42
  --
r= 15 ,  m = 21 and n = 15
```

Again,

```
(15)21(1
  -15
  ---
r= 6.  ,  m = 15 and n = 6.
```

Again,

```
6)15(2
  -12
  ---
r= 3  ,  m = 6 and n = 3.
```

Now.

```
3)6(2
  -6
  --
r= 0  ,  m = 3 and n = 0.
```

∴ gcd(78, 57) = 3.

or,

gcd(78, 57) = gcd(57, 21)
= gcd(21, 15)
= gcd(15, 6)
= gcd(6, 3)
= gcd(3, 0).
↑ gcd.
return 3.

- Consecutive integer checking to compute gcd(m,n).

Step 1: Assign the value of minimum {m,n} to t.

Step 2: Divide m by t. If the remainder of this division is zero.
Goto Step 3, otherwise goto Step 4.

Step 3: Divide n by t. If the remainder of this division is zero, return the value of t as the answer and Stop; otherwise proceed to step 4.

Step 4: Decrease the value of t by 1. Goto Step 2.

- Middle school procedure for computing gcd(m,n).

Eg: gcd(56,21)

Find prime factors of both m, n.

$56 = 2 \times 2 \times 2 \times 7$.

$21 = 3 \times 7$.

$\therefore gcd(56,21) = 7$.

2	56
2	28
2	14
	7

Eg: gcd(60,24)

$60 = 2 \times 2 \times 3 \times 5$

$24 = 2 \times 2 \times 2 \times 3$.

$\therefore gcd(60,24) = 2 \times 2 \times 3 = 12$

2	60
2	30
3	15
	5

2	24
2	12
2	6
	3

Step 1: Find the prime factors of m.

Step 2: Find the prime factors of n.

Step 3: Identify all the common factors in the two prime expansions, found in Step 1 and Step 2.

[If p is a common factor occuring p_m and p_n times in m and n respectively it should be repeated $\min\{p_m, p_n\}$ times.]

Step 4: Compute the product of all the common factors and return it as the gcd of given numbers.

1> WAP to check whether the given number is prime or not.
2> WAP to generate prime number upto n.

①
```
#include <stdio.h>
#include <stdlib.h>
int main()
{
    int n, i, flag=0;
    printf("Enter a positive no: \n");
    scanf("%d", &n);
    for(i=2; i<=n/2; i++)
    {
        if(n%i==0)
        {
            flag=1;
            break;
        }
    }
    if(flag==0)
        printf("The number is prime\n");
    else
        printf("The number is not prime\n");
    return 0;
}
```

②
```
int main()
{
    int i, j, m, n, c;
    printf("Enter value of n:\n");
    scanf("%d", &n);
    for(i=1; i<=n; i++)
    {
        for(j=2; j<=sqrt(i); j++)
        {
            if(i%j==0)
                c++;
        }
        if(c<=1)
        {
            printf("Not a prime no:");
        }
```

```
        else
            printf("Prime numbers\n");
        }
    }
}
```

- Sieve of Eratosthenes:

n = 18.

2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
[2]	3		5		7		9		11		13		15		17	
2	[3]		5		7				11		13				17	

∴ prime numbers are 2 3 5 7 11 13 17, upto 18.

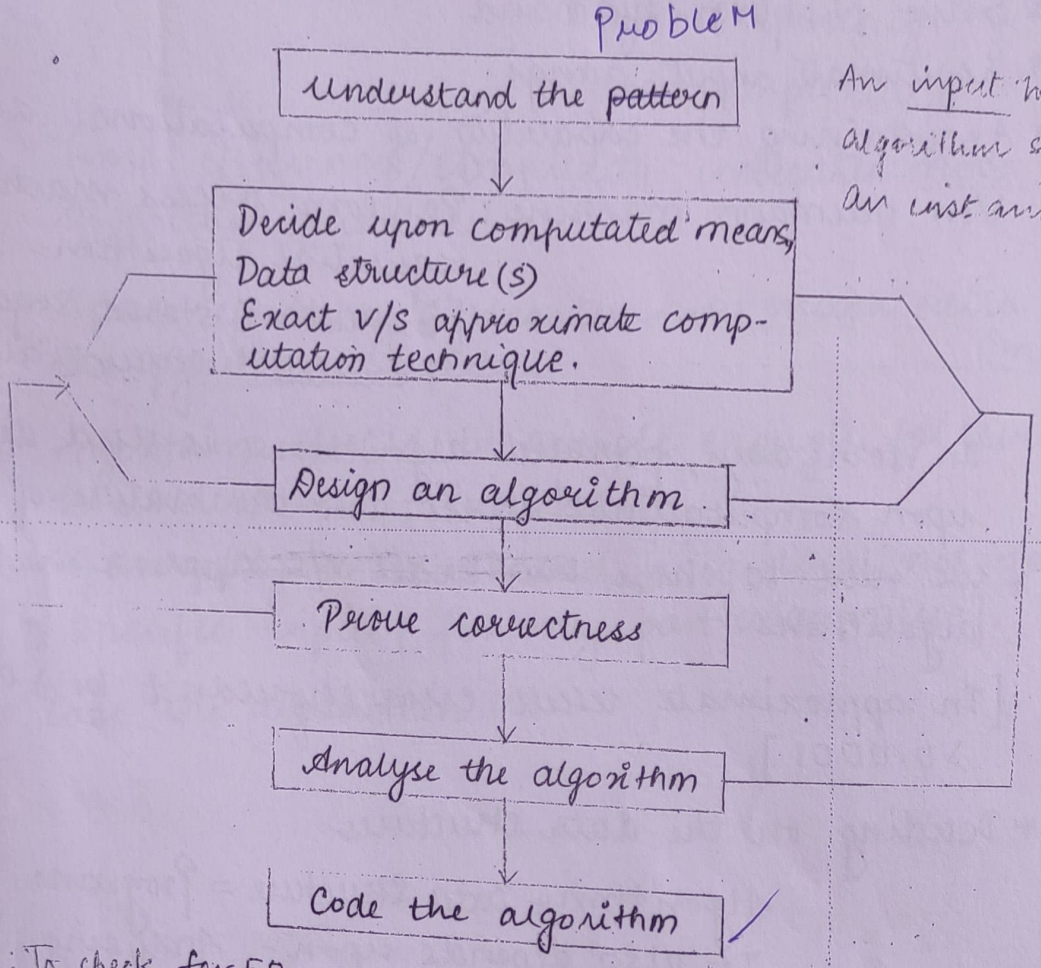
Eg: To check for 50.

- ALGORITHM-

```
Algorithm Seive(n)
// Input: an integer n ≥ 2
// Output: array L of all prime numbers less than or
            equal to 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
// Copy non-zero elements of A to array L.
i ← 0
for p ← 2 to n do
    if A[p] ≠ 0
        L[i] ← A[p]
        i ← i+1
return L
```

3/01/19

- Fundamental of algorithmic problem solving



Eg: To check for 50.

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 40 41 42 43 44 45 46 47
48 49 50.

[2] 3 5 7 9 11 13 15 17 19 21 23 25 27
29 31 33 35 37 39 41 43 45 47 49

2 [3] 5 7 11 13 17 19 23 25 29 31 35
37 41 43 47 49

2 3 [5] 7 11 13 17 19 23 29 31 37 41
43 47 49.

2 3 5 [7] 11 13 17 19 23 29 31 37 41 43 47.

∴ Prime numbers are 2 3 5 7 11 13 17 19 23 29 31
37 41 43 47.

04/01/19

Understand problems.

* Solve problem by hand.
* Legitimate input range.
* Ascertaining the capability of computational device.

Von neumann machine: Random access machine.
Sequential algorithm.
[instruction stored sequentially & executed sequentially.]

In recent days, parallel algorithms is used depending upon computational device, also the nature of m/c.

* we need to chose btw exact v/s approximate algorithms.

[In approximate result, error shouldn't be ≥ 0.001 or > 0.0001]

* Deciding on the data structures.

Algorithm + Data Structure = Program.
It also depends upon - Analysing its performance.

* Algorithm Design Technique: We should chose proper algorithm for the program.

* Method for specifying algorithm

natural language pseudo code.

PSEUDO CODE - It's a mixture of natural language and programming language constraints / constructs.

* Prove algorithmic correctness: It should yeild correctness in a finite amount of time with legitimate input.

[we use mathematical induction to prove the correctness.]

* Analyse the algorithm: Two types of algorithm efficiency
  - Time complexity.
  - Space complexity.

- Time efficiency/complexity indicates how fast the algorithm runs.
- Space efficiency indicates how much extra space (memory) the algorithm needs.
- Algorithm should be simple enough for everyone to understand - Simplicity.
- Generality i.e it should be general not for some specific input [for only particular value].

* Code the algorithm.

05/01/19

Analysing efficiency of an algorithm -

* Measuring input size:

To investigate an algorithm's efficiency as a function of some parameter is quite of the value 'n' indicating the algorithm's input size.

$P(x) = a_n x^n + \ldots + a_0$ of degree n, it will be the polynomial's degree or the number of its coefficients, which is larger by one than its degree.

Time of execution depends upon the number / size of the input.

polynomial - degree.
matrix - order.

* How to measure time:

Running time -

let Cop be execution time of algorithms, basic operation on a particular computer.

Let C(n) be the number of times this operation needs to be executed for this algorithm, then

T(n) be the running time can be estimated by,

$$\boxed{T(n) \approx C_{op} C(n)}$$

'Cop' is an approximate & C(n) doesn't contain any information about operations, that are not basic.

∴ This formula can be give a reasonable estimate of the algorithms running time.

1) Assuming $C(n) = \frac{1}{2} n(n-1)$. How much longer will be the algorithm run if it doubles it input size.

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

If 'n' is too large as compared to n^2, n can be neglected

$\therefore C(n) \approx \frac{n^2}{2}$.

$\therefore T(n) = cop\, C(n) = cop\, \frac{n^2}{2}$.

& $T(2n) = cop\, C(2n) = cop\, \frac{(2n)^2}{2}$

$= cop\, \frac{4n^2}{2}$.

Now, $\frac{T(2n)}{T(n)} = \frac{cop\, \frac{4n^2}{2}}{cop \cdot \frac{n^2}{2}}$

$\therefore \frac{T(2n)}{T(n)} = 4 \Rightarrow \boxed{T(2n) = 4T(n)}$

Hence, it requires 4 times when input is doubled.

* Order of growth:

n	logen	n	nlogen	n^2	n^3	2^n	n!
10	3.3	10^1	3.3×10^1	10^2	10^3	10^3	3.6×10^6
10^2	6.6	10^2	6.6×10^2	10^4	10^6	1.3×10^{30}	9.3×10^{157}
10^3	10	10^3	1.0×10^4	10^6	10^9		
10^4	13	10^4	1.3×10^5	10^8	10^{12}		
10^5	17	10^5	1.7×10^6	10^{10}	10^{15}		

Values (some approximate) of several functions important for analysis of algorithm.

07/02/19

Worst case, best case and average case efficiency:

* Algorithm sequential search (A[0...n-1])

Algorithm sequential search(A[0...n-1], k)
// Searches for a given value in a given array by sequential search.
// Input : An array A[0...n-1] and a search key k.
// Output : An index of the first element of A that matches k or -1 if there are no matching elements.

```
i ← 0
while i<n and A[i]≠k do
    i ← i+1
if i<n return i
else return -1
```

Best case: $C_{best}(n) = 1$

Worst case: $C_{worst}(n) = n$.

* Assumption for analysis of algorithm -

- The probability of a successful search is equal to ($0 \le p < 1$).
- The probability of 1st match occuring at ith position is same for every i. ($0 \le i < n$).
- Now we find average number of key comparison as follows:

  In case of successful search, the probability of first match occuring at i^{th} position of the list is p/n, for every i the number of comparison is i.

  In case of unsuccessful search, the number of comparison is 'n' with the probability of such a search being $(1-p)$.

$$C_{avg}(n) = \left[1 \cdot \frac{p}{n} + 2 \cdot \frac{p}{n} + 3 \cdot \frac{p}{n} + \ldots + n \cdot \frac{p}{n}\right] + n \cdot (1-p$$

$$= \frac{p}{n}\left[1+2+3+\ldots+n\right] + n(1-p)$$

$$= \frac{p}{n}\left[\frac{n(n+1)}{2}\right] + n(1-p).$$

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

$$\therefore C_{avg}(n) = \frac{p(n+1)}{2} + n(1-p)$$ For sequential search.

→ For successful search, $p = 1$

$\therefore C_{avg}(n) = \frac{n+1}{2} \approx \frac{n}{2}$.

→ For unsuccessful search, $p = 0$

$\therefore C_{avg}(n) = n$.

Asymptotic notations and basic efficiency classes -

There are 3 asymptotic notations:

- big-oh (O)
- big-theta (Θ)
- big-omega (Ω)

Let $t(n)$ and $g(n)$ be the non-negative functions, defined on the set of natural numbers.

O-notation -

$O(g(n))$ is a set of all function with a smaller or same order of growth as $g(n)$.

Eg: $n \in O(n^2)$, $t(n) = n$ and $g(n) = n^2$

n	n^2
0	0
1	1
2	4
3	9
4	16

Eg:- $100n+5 \in O(n^2)$

[Order of linear function will be smaller than order of quadratic function]

Substitute $n=101$.

$100 \times 101 + 5 = 10105$.

$n^2 = 101 \times 101$.

$= 10201$

n	$100n+5$	$g(n)$ n^2
0	5	0
1	105	1
2	205	4
3	305	9
⋮	⋮	⋮
101	10105	10201 (larger)
102	10205	10404

$g(n) < 5$

$\therefore 100n+5 \in O(n^2)$ for $n_0 \geq 101$ ✓

11/01/19

- O notation: $t(n) \in O(g(n))$, Small order or same.
- Ω notation: $t(n) \in \Omega(g(n))$, larger or same order.
- Θ notation: $t(n) \in \Theta(g(n))$, same order.

Eg: $t(n) = n^2 + n \in O(n^3)$.

$n^3 \in \Omega(n^2)$.

$an^2 + bn + c \in \Theta(n^2 + n)$.

$n^3 + n^2 \in \Theta(n^3 + n)$.

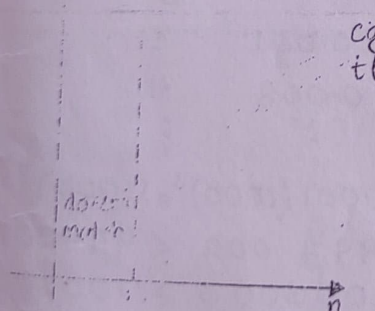
$20n + 5 \notin \Theta(n^2)$.

linear quadratic.

1) O notation: A function $t(n)$ is said to be in big-oh(n) denoted as,

$t(n) \in O(g(n))$, if $t(n)$ is bounded above by some constant multiple of $g(n)$ for all large n i.e. if there exists some positive constant 'c' and some non-negative integer n_0 such that:

$$\boxed{t(n) \leq cg(n) \ \forall \ n \geq n_0}$$



Eg: $100n + 5 \in O(n^2)$.

* $100n + 5 \in 101n$

n	LHS		RHS
n=0	5		0
n=1	105	>	101
n=2	205	>	202
n=3	305	>	303
n=4	405	>	404
n=5	505	=	505
n=6	605	<	606

$\therefore 100n + 5 \in 101n \ \forall \ n \geq 5$ and $c = 101$.

* $101n \in 101n^2 \ \forall \ n \geq 0$ and $c = 101$.

* $\frac{1}{2}n(n-1) \in O(n^2)$.

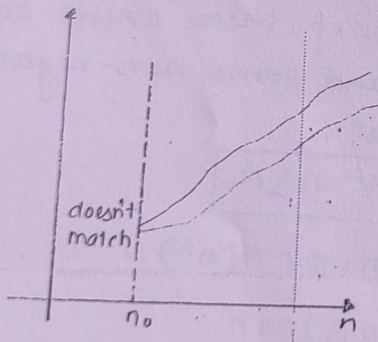
n	LHS		RHS
$n=0$	0	=	0
$n=1$	0	<	1
$n=2$	1	<	4.

$\therefore \frac{1}{2}n(n-1) \in O(n^2)$ $\forall$ $n \geq 0$ and $C=1$.

2) Ω notation: A function $t(n)$ is said to be in $\Omega(g(n))$ denoted,

$t(n) \in \Omega g(n)$, if $t(n)$ is bounded below by some positive constant multiple of $g(n)$ for all large n, i.e. if there exists some positive constant C and some non-negative integer n_0 such that:

$$t(n) \geq C g(n) \quad \forall \quad n \geq n_0$$



Eg: $n^3 \in \Omega(n^2)$ $\forall$ $n \geq 0$ and $C=1$

* $0.001n^3 \in \Omega(n^2)$. $\forall$ $n \geq 1000,00$ and $C=1$.

n	LHS	RHS
$n=0$	0	0
$n=1$	0.001	1
$n=2$	0.008	4.
⋮	⋮	⋮
$n=1000$	$0.001(1000)^3 = (1000)^2$	
$n=999$	997 002 < 999001	
$n=1001$	1003003 > 1002001	

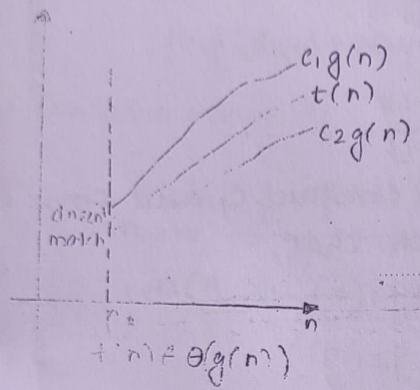
[We need to consider C as the coefficient of higher degree]

3) Θ notation: A function $t(n)$ is said to be in $\Theta(g(n))$ denoted,

$t(n) \in \Theta(g(n))$, if $t(n)$ is bounded both upper/above and below by some

positive constants multiple of $g(n)$. For all larger n, i.e. if there exists some positive constant c_1 and c_2 and some non-negative integer n_0 such that:

$$\boxed{c_2 g(n) \leq t(n) \leq c_1 g(n) \quad \forall \; n \geq n_0}$$



Eg: $\frac{1}{2} n(n-1) \in \Theta(n^2)$.

$\frac{1}{2} n(n-1) = \frac{1}{2} n^2 - \frac{n}{2} \leq \frac{1}{2} n^2 \; \forall \; n \geq 0$

Consider, $t(n) = \frac{1}{2} n^2 - \frac{1}{2} n$ and

$g(n) = \frac{1}{2} n^2$. $c_1 = \frac{1}{2}$.

n	$t(n) = \frac{n^2}{2} - \frac{n}{2}$		$g(n) = \frac{1}{2} n^2 \quad \forall \; n \geq 0$.
$n=0$	0	=	0
$n=1$	0	<	$\frac{1}{2}$
$n=2$	1	<	2
$n=3$	3	<	$\frac{9}{2}$
$n=4$	6	<	8

$\frac{1}{2} n^2 - \frac{1}{2} n \geq c_2 n^2$

n	$t(n)$		$g(n)$.
$n=0$	0	=	0
$n=1$	0	$< \frac{1}{4} = c_2 \times$.	
$n=2$	1	$> 4c_2$ ✓	
$n=3$	3	$> 9c_2$	
$n=4$	6	$> 16 c_2$.	

For what value of c_2 $t(n) \geq g(n)$.

If $c_2 = 1/4$.

i.e. $\frac{1}{2} n^2 - \frac{1}{2} n \geq \frac{1}{4} n^2 \; \forall \; n \geq 2$.

$\therefore \frac{1}{2} n^2 - \frac{1}{2} n \in \Theta(n^2) \; \forall \; n \geq 2$ and $c_1 = \frac{1}{2}$ and $c_2 = \frac{1}{4}$.

14/01/19

* If $t_1(n) \in O(g_1(n))$ and $t_2(n) \in O(g_2(n))$ then,

$$t_1(n) + t_2(n) \in O \max\{g_1(n), g_2(n)\}.$$

Proof: Let a_1, b_1, a_2, b_2 be four arbitary real numbers.
If $a_1 \leq b_1$ and $a_2 \leq b_2$ then,

$$a_1 + a_2 \leq 2 \max\{b_1, b_2\}.$$

Since $t_1(n) \in O(g_1(n))$
there exists some (+ve) constant c_1 and some non-negative integer n_1 such that,

$$t_1(n) \leq c_1 g_1(n) \quad \forall \; n \geq n_1.$$

Similarly; $t_2(n) \in O(g_2(n))$
such that,

$$t_2(n) \leq c_2 g_2(n) \quad \forall \; n \geq n_2.$$

Let us consider $c_3 = \max\{c_1, c_2\}$ and $n \geq \max\{n_1, n_2\}$, so that we can use both inequalities.

$$\begin{aligned} t_1(n) + t_2(n) &\leq c_1 g_1(n) + c_2 g_2(n) \\ &\leq c_3 g_1(n) + c_3 g_2(n) \\ &\leq c_3 [g_1(n) + g_2(n)] \\ &\leq c_3 \, 2 \max\{g_1(n), g_2(n)\}. \end{aligned}$$

Hence, $t_1(n) + t_2(n) \in O \max\{g_1(n), g_2(n)\}$ with c and n_0, where $c = 2c_3 = 2 \max\{c_1, c_2\}$ and: $n_0 = \max\{n_1, n_2\}$.

* Using limits for comparing order of growth:

$$\lim_{n \to \infty} \frac{t(n)}{g(n)} = \begin{cases} 0, & \text{implies that } t(n) \text{ has a smaller order of growth than } g(n). \\ c > 0, & \Rightarrow t(n) \text{ has the same order of growth as } g(n). \\ \infty, & \Rightarrow t(n) \text{ has a larger order of growth than } g(n). \end{cases}$$

$$\lim_{n\to\infty} \frac{t(n)}{g(n)} = \lim_{n\to\infty} \frac{t'(n)}{g'(n)}.$$

↑ L'Hospital rule.

$n! \approx \sqrt{2\pi n}\left(\frac{n}{e}\right)^n$ [Sterling's Formula].
for large value of n.

- Compare order of growth of $\frac{1}{2}n(n-1)$ and n^2.

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

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

$$= \frac{1}{2}\times\left(1-\frac{1}{\infty}\right) = \frac{1}{2}(1-0).$$

$$= \frac{1}{2}. \quad \left(\because \frac{1}{\infty} > 0\right)$$

$\therefore \frac{1}{2}n(n-1) \in \Theta(n^2)$.

- Compare the order of growths of $n!$ and 2^n.

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

$\sqrt{2\pi n}\,(n/e)^n$

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

$$= \infty.$$

$\therefore n! \in \Omega(2^n)$
though 2^n grows fast but $n!$ grows still faster.

t	constant
$\log n$	logarithms
n	linear
n^2	quadratic
n^3	cube
2^n	exponential
$n!$	Factorial.

- Mathematical Analysis of non-recursive algorithms.

General Plan for analysing time efficiency in non-recursive algorithm -

Step 1 : Decide on a parameter indicating input size.

Step 2 : Identify the algorithm basic operations. (located in its innermost loop).

Step 3 : Check whether the number of times the basic operation is executed depends only on the size of an input.
If it also depends on some additional property the worst case, avg. case, and if necessary best case efficiency have to be investigated seperately.

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

Step 5 : Using the standard formula & rules of some manipulation either find a closed formula for the count or at the very least establish its order of growth.

Table 2.2 Basic Asymptotic Efficiency Classes.

Class	Name	Comments
1	constant	short of best-case efficiencies, very few reasonable examples can be given since an algorithm's running time typically goes to infinity when its input size grows infinitely large.
$\log n$	logarithmic	Typically, a result of cutting a problem size by a constant factor on each iteration of the algorithm. Note that a logarithm algorithm cannot take into account all its input (or even a fixed fraction of it) any algorithm that does so will have atleast linear running time.
n	linear	Algorithms that scan a list of size n belong to this class.
$n \log n$	n log-n	Many divide-and-conquer algorithms including mergesort & quicksort in the average case, fall into this category.
n^2	quadratic	Typically, characterizes efficiency of algorithms with two embedded loops: Elementary sorting algorithms and certain operations on n-by-n matrices are standard examples.
n^3	cubic	Typically, characterizes efficiency of algorithms with three embedded loops. Several non-trivial algorithms from linear algebra fall into this class.
2^n	exponential	Typical for algorithms that generate all subsets of an n-element set. Often, the term "exponential" is used in a broader sense to include this and [illegible]

Class	Name	Comments
		of growth as well.
$n!$	factorial	Typical for algorithms that generate all permutations of an n-element set.

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

- Find maximum element in an array and analyse its complexity.

//Determines the value of the largest element in a given array.
//Input : Array of 0 to n-1 of real numbers.
//Output: The value of largest element in A.

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

→ Analysis

input size ← n

basic operation ← comparison

$$C(n) = \sum_{i=1}^{n-1} 1 = n-1 \quad \text{or} \quad n-1+1-1 = n-1 \text{ [By formula].}$$

$$\therefore C(n) = \sum_{i=1}^{n-1} 1 = n-1 \in \Theta(n) \quad \{U-L+1\}.$$

18/01/19

- Find whether all the elements in a given array is distinct or not.

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

// Determines all the element in a given array is distinct.

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

// Output: Returns true, if all the elements are distinct
false otherwise.

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

Tracing n = 6

A[0] 44
[1] 50
[2] 36
[3] 12
[4] 16
[5] 2

→ Analysis

input size ← n

basic operation ← comparison

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

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

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

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

$\therefore \frac{n^2}{2} \in \Theta(n^2)$.

~~DOE~~ 20/01/19 ~~WRONG~~

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

```
dmin ← 9999
for i ← 0 to n-1 do
    for j ← 0 to n-1 do
        if i ≠ j & |A[i] - A[j]| < dmin
            dmin ← |A[i] - A[j]|
return dmin
```

a) what does this algorithm compute?
b) what is the basic operation?
c) what is the complexity of this algorithm?
d) How do you improve this algorithm?
e) How many times the basic operation executed?
f) what is the efficiency class of the new algorithm?

a) Finds the distance btw the 2 closest elements in an array of numbers (min distance btw 2 elements of array).

b) Comparison.

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

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

$= 2(\overbrace{n+n+\dots+n}^{n \text{ times}}) \; n \cdot n = n^2$

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

d) Algorithm MinDistance(A[0.....n-1])

```
dmin ← 9999
for i ← 0 to n-2 do
    for j ← i+1 to n-1 do
        if |A[i] - A[j]| < dmin
            dmin ← |A[i] - A[j]|
return dmin
```

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

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

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

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

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

b) $\theta(n^2)$

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

for $i \leftarrow 0$ to $n-2$ do // Palindrome.

  for $j \leftarrow i+1$ to $n-1$ do

    if $A[i,j] \neq A[j,i]$

      return false.

Return true.

Answer the following questions:

a) What does this algorithm do?

b) What is the basic operation?

c) How many times the basic operation executed?

d) what is the efficiency class of this algorithm?

a) It finds whether the matrix is symmetric or not over principal diagonal.

b) Comparison

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

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

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

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

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

$= \frac{(n-1)n}{2} \approx \frac{n^2}{2} \in \theta(n^2)$.

d) $\theta(n^2)$.

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

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

a) To find maximum and minimum elements in an array.

b) Comparison

c) $C(n) = \sum_{i=1}^{n-1} 2$

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

$= 2(n-1) \approx 2n \in \Theta(n)$

d) Improved algorithm

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

⋮

else if A[i] > maxval

1) Estimate the running time of multiplication of 2 matrices of order n×n.

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

```
// Multiplies 2 n×n matrices
// Input : 2 n×n matrices A and B
// Output : matrix C = A·B
for i ← 0 to n-1 do
    for j ← 0 to n-1 do
        C[i,j] ← 0
        for k ← 0 to n-1 do
            C[i,j] ← C[i,j] + A[i,k] * B[k,j]
return C
```

~~A[i,j] * B[i,j]~~

Sol: Basic operations are addition and multiplication it is sufficient if we analyse for multiplication operation as it is more time consuming than addition.

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

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

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

$$= \underbrace{(n^2+n^2+\dots+n^2)}_{n\text{-times}} = n^2 \times n = n^3 \in \theta(n^3).$$

If C_m is the time required to perform one multiplication operation then running time $= T(n) \simeq C_m M(n)$

$$T(n) = C_m \cdot n^3$$

To get more accurate estimate we need to take into account, time spent on addition also,

$\therefore T(n) \approx C_m M(n) + C_a A(n)$ where C_a is the time for one addition.

$\approx C_m n^3 + C_a n^3$

$$\boxed{\therefore T(n) \approx n^3 (C_m + C_a)}$$

21/01/19

- Algorithm to find the binary digits in a binary representation of positive decimal integer.

Algorithm binaryCount(n)

// Finds number of binary digits
// Input: Positive decimal integer.
// Output: Number of binary digits in n's binary representation.

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

or count ← 0
while n > 0 do.

count ← 1

→ Analysis

input size ← n

basic ← Comparison in while loop condition [because one extra time it will get executed]. Entry control loop.

Complexity ← $\sum_{i=1}^{n/2} 1$

∵ $n/2, n/4, n/8, \ldots\ldots n/2^i$

WKT, $n/2^i = 1$

$\therefore i = \log_2 n$

$\therefore (\log_2 n + 1) \approx \log_2 n \in \Theta(\log_2 n)$ ← extra comparison

*: Mathematical Analysis of recursive algorithm

Recursive Algorithm:

$F(n) = n!$

$n! = n \times (n-1) \times (n-2) \times \ldots \ldots \ldots 2 \times 1.$

$\boxed{n! = n * (n-1)!}$ Recursively

$n! = n * (n-1)! \quad \forall\ n \geq 1$

$0! = 1$

Algorithm F(n)

// Compute n! recursively

// Input: A non negative integer n

// Output: The value of n!

if $n = 0$ return 1

else return $F(n-1) * n$

→ Analysis

input size ← n

basic operation ← Multiplication.

Recursive relation.

$M(n) = M(n-1) + 1$ for $\forall\ n > 0$ → to multiply $F(n-1)$ by n.

$M(n) = 0$

$M(n-1)$: no. of multiplication required to compute $F(n-1)$

- To solve reccurence relation, Backward subsitution is required.

$M(n) = M(n-1) + 1 \quad \forall\ n > 0$

$M(n) = 0$

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

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

$= M(n-2) + 2$

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

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

$=$ ⋮

$= M(n-1)+i$

$= M(n-n)+n$

$= M(0)+n$

$= 0+n$

$\therefore M(n) = n$

- General plan for analysing time efficiency of recursive algorithm -

Step 1: Decide on a parameter indication / indicating input size.

Step 2: Identify the algorithm basic operation.

Step 3: Check whether the number of times the basic operation is executed can vary and on different input of same size. If it can in the worst case, best case and avg case efficiency must be investigated seperately.

Step 4: Setup a recurrence relation with an appropriate initial condition for the number of times the basic operation is executed.

Step 5: Solve the recurrence relation.

* Write recursive algorithm to compute sum of first n cubes and analyse its complexity. Compare the performance of this algorithm with straight forward non-recursive algorithm.

$S(n) = 1^3+2^3+3^3+\cdots\cdots+n^3$.

Algorithm S(n)
// Compute the sum of first n^3
// Input: A positive integer n
// Output: returns sum of first n^3.
if n=1 return 1
else return S(n-1) + n*n*n

→ Analysis

input size ← n
basic operation ← Multiplication

Recurrence relation

$M(n) = M(n-1) + 2 \quad \forall\ n > 1$

$M(1) = 0$

$M(n) = M(n-1)+2 = (M(n-2)+2)+2 = M(n-2)+4$
$= M(n-3)+2+4 = M(n-3)+6$
$= \dots\dots\dots = M(n-i)+2i$
$= M(n-(n-1))+2(n-1)$ ∴ $\boxed{M(n) = 2n-2 \approx 2n \in \Theta(n)}$

Tower of Hanoi

```
void towerofHanoi(int disk, char from, char to, char aux)
{
    if(disk==1)
    {
        printf("Move disk 1 from %c peg to %c\n", from, to);
        return;
    }
towerofHanoi(disk-1, from, aux, to);
printf("Move disk %d from %c peg to %c peg", disk, from,
                                              to);
towerofHanoi(disk-1, aux, to, from);
}
```

23/01/19

- **Non-recursive Algorithm**

// Compute the sum of first n^3.
// Input: A positive integer n.
// Output: A positive integer that is sum of first n^3.

Algorithm S(n)

```
Sum ← 0
for i ← 1 to n do
    sum ← sum + i * i * i
return sum
```

→ Analysis

i) Input size ← n
ii) Basic operation ← multiplication

Complexity:

$$M(n) = \sum_{i=1}^{n} 2$$

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

$$= 2 \times (n - 1 + 1)$$

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

$$\therefore \boxed{M(n) \in \Theta(n)}$$

[Complexity of non-recursive algo is greater as compared to recursive algo].

This is same as same as recursive version but non recursive version does not carry the time and space overhead associated with recursive stack.

- **Tower of Hanoi.**

input size ← number of disks n
basic operation ← no. of moves of disk.

For every invocation of function it recursively calls itself twice.

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

$= 2M(n-1) + 1$ for all $n > 1$

$M(1) = 1$.

$\therefore M(n) = 2M(n-1) + 1$ Substitute $M(n-1) = 2M(n-2) +$

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

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

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

i.e. $2^2(2M(n-3)+1) + (2+1)$.

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

i.e.

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

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

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

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

Substitute $n - i = 1$

$n - 1 = i$.

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

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

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

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

$\boxed{\therefore M(n) = 2^n - 1}$

binary(25)

↓ 5

rem(binary(12)) + 1

↓ 4

rem(binary(6)) + 1

↓ 1 3

rem(binary(3)) + 1

↓ 1 2

rem(binary(1)) + 1

• Write a recursive algorithm to find number of digits in binary representation of positive decimal integer.

// Compute the binary digits in binary representation.
// Input: Positive decimal integer.
// Output: Number of binary digits in n's binary representation.

if (n=1) return 1
else return binrec([n/2])+1

→ $A(n) = A(n/2)+1 \quad \forall\, n>1$

$A(1) = 0.$

WKT, $M(n) = M(n-1)+1$ ✗

$A(n) = A(n/2)+1$ // Put $n = 2^k$

$\therefore A(2^k) = A\left(\frac{2^k}{2}\right)+1$

$= A\left(\frac{n-2}{2}\right)+1$ ✗ $= A(2^{k-1})+1$

Smoothness Rule:

Let, $n = 2^k$ then recurrence relation represented as,

$\frac{2^k}{2} = 2^{k-1}$

$A(2^k) = A(2^{k-1})+1, \forall\, n>1$ // if we sub then.

$\therefore A(2^0) = 0.$

$A(2^{k-1}) = A(2^{k-1-1})+1$

$= A(2^{k-2})+1$

$A(2K) = A(2^{k-1})+1 \quad \forall\, k>0$

$= A(2^{k-2})+1+1$ Substitution,

$= A(2^{k-3})+1+1+1$ $A(2K-1) = A(2^{k-2})+1$

$= A(2K-3)+3$ $A(2^{k-2}) = A(2^{k-3})+1$

$\vdots$

$A(2^k) = A(2^{k-i})+i$ $K-i=0$

$= A(2^{k-k})+k$ $\therefore i=k$

$\therefore A(2^k) = k.$

$2^k = n$

$k = \log_2 n.$

$\therefore \boxed{A(n) = \log_2 n} \in \Theta(\log_2 n)$

Algorithm Tower of Hanoi:

// Algorithm for tower of hanoi.
// Input: No. of disk n, name of three pegs
// Output: Move of disk from one peg to another.

```
Algorithm TowerofHanoi(n, src, aux, dest)
if disk == 1
        move disk from src to dest
        return
TowerofHanoi(n-1, src, dest, aux)
move disk from src to dest
TowerofHanoi(n-1, aux, src, dest)
```

25/01/19

→ Solve the following recurrence relation:

i) $x(n) = x(n-1) + 5$ for all $n \geq 1$ and $x(1) = 0$.

Sol°: Given: $x(n) = x(n-1) + 5 \quad \forall\ n \geq 1$

and,

$x(1) = 0$.

$x(n) = x(n-1) + 5$

$= (x(n-2) + 5) + 5$

$= x(n-2) + 5 + 5$

$= x(n-3) + 5 + 5 + 5$

$= x(n-3) + 3 \cdot 5$

Substitute:

$x(n-2) = x(n-2) + 5$

$\vdots$

$= x(n-i) + 5i$

Now, $n - i = 1 \Rightarrow i = n-1$ bcoz we have if $n-i$

$= x(n-(n-1)) + 5(n-1)$

$= x(n-n+1) + 5(n-1)$

$= x(1) + 5(n-1)$.

$\therefore x(n) = x(1) + 5(n-1)$

$= 0 + 5(n-1)$.

$\therefore \boxed{x(n) = 5(n-1)} \in \Theta(n)$

ii) $x(n) = 3x(n-1)$ for all $n \geq 1$ and $x(1) = 4$.

Sol°: Given, $x(n) = 3 \cdot x(n-1) \quad \forall\ n \geq 1$

$\Rightarrow x(n) = 3 \cdot 3\, x(n-2)$

$= 3 \cdot 3 \cdot 3\, x(n-3)$

$= 3^3 x(n-3)$.

$\vdots$

$= 3^i x(n-i)$.

$\vdots$

$= 3^{n-1} x(n-n+1)$

$= 3^{n-1} x(1)$

$n - i = 1$

$i = n-1$

$\Rightarrow x(n) = 3^{n-1} \times 4.$

$= 4.3^{n-1}.$

$\therefore \boxed{x(n) = 4.3^{n-1}} \in \Theta(3^n).$

iii) $x(n) = x(n-1) + n \quad \forall\ n > 0$ and $x(0) = 0.$

Sol°: Given: $x(n) = x(n-1) + n$

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

$= x(n-3) + (n-1) + (n-2)$

$= x(n-4) + (n-1) + (n-2) + \cdots$

Given: $x(n) = x(n-1) + n$

$= x(n-2) + (n-1) + n$ Subs. $x(n-1) = x(n-2) + (n-1)$

$= x(n-3) + (n-2) + (n-1) + n.$

$\vdots$

$= x(n-i) + [(n-(i-1)) + (n-(i-2)) + \cdots + (n-1) + n].$

put, $n - i = 0$, $n = i$. i.e $i = n.$

$x(n) = x(0) + [1 + 2 + 3 + \cdots + n]$

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

$\therefore \boxed{x(n) = \frac{n(n+1)}{2}} \in \Theta(n^2).$

iv) $x(n) = x(n/3) + 1 \quad \forall\ n \geq 1,\ x(1) = 1.$

Sol°, Given: $x(n) = x(n/3) + 1.$

By Smoothness Rule,

$n = 3^k.$

$\Rightarrow x(n) = x(n/3) + 1$

$\Rightarrow x(3^k) = x(3^{k-1}) + 1 \quad \forall\ k \geq 0,\ x(3^0) = 1.$

$= x(3^{k-2}) + 1 + 1$

$= x(3^{k-3}) + 1 + 1 + 1$

$= x(3^{k-3}) + 3$

$\therefore x(3^k) = x(3^{k-3}) + 3$

$\vdots$

$x(3^k) = x(3^{k-i}) + i$; $k-i=0$

$= x(3^{k-k+1}) + i$ $k=i$

$= x(3^0) + k$

$= k+1.$

$\therefore x(3^k) = k+1.$

$\therefore \boxed{x(n) = 1+\log_3 n \in \theta(\log_3 n).}$

v) $x(n) = x(n/2) + n$ $n>1$, $x(1)=1.$

Sol°: Given, $x(n) = x(n/2) + n$ $k>1$

Put $n=2^k.$

$x(2^k) = x(2^{k-1}) + 2^k.$, Substitute $x(2^{k-1}) = x(2^{k-2}) + 2^{k-1}$

$= x(2^{k-2}) + 2^{k-1}$

$= x(2^{k-3}) + 2^{k-2} + 2^{k-1}$

$\vdots$

$= x(2^{k-i}) + 2^{k-(i-1)} + 2^{k-(i-2)} + \ldots + 2^k$

Put $k-i=0$ $\boxed{i=k.}$ $= x(2^{k-k}) + 2^{k-(k-1)} + 2^{k-(k-2)} + \ldots + 2^k$

$\therefore x(2^k) = x(2^0) + 2^1 + 2^2 + 2^3 + \ldots + 2^k$

$= x(1) + [2 + 2^2 + 2^3 + \ldots + 2^k]$

$= 1 + 2 + 2^2 + 2^3 + \ldots + 2^k$

$= 2^0 + 2^1 + 2^2 + \ldots + 2^k$

$= 2^{n+1} - 1$ here, $n=k.$

$\therefore \boxed{x(2^k) = 2^{k+1} - 1} \in \theta(n)$

$\therefore x(2^k) = 2 \cdot 2^k - 1$

$\boxed{x(n) = 2n - 1} \in \theta(n)$

28/01/19

* Bruteforce Method:

→ Selection sort - (No other method can be used to solve sort i.e selection and bubble sort except ...)

Algorithm selection sort (A[0.... n-1])

// 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]
```

→ Analysis

Input size ← n
Basic operation ← comparison

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

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

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

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

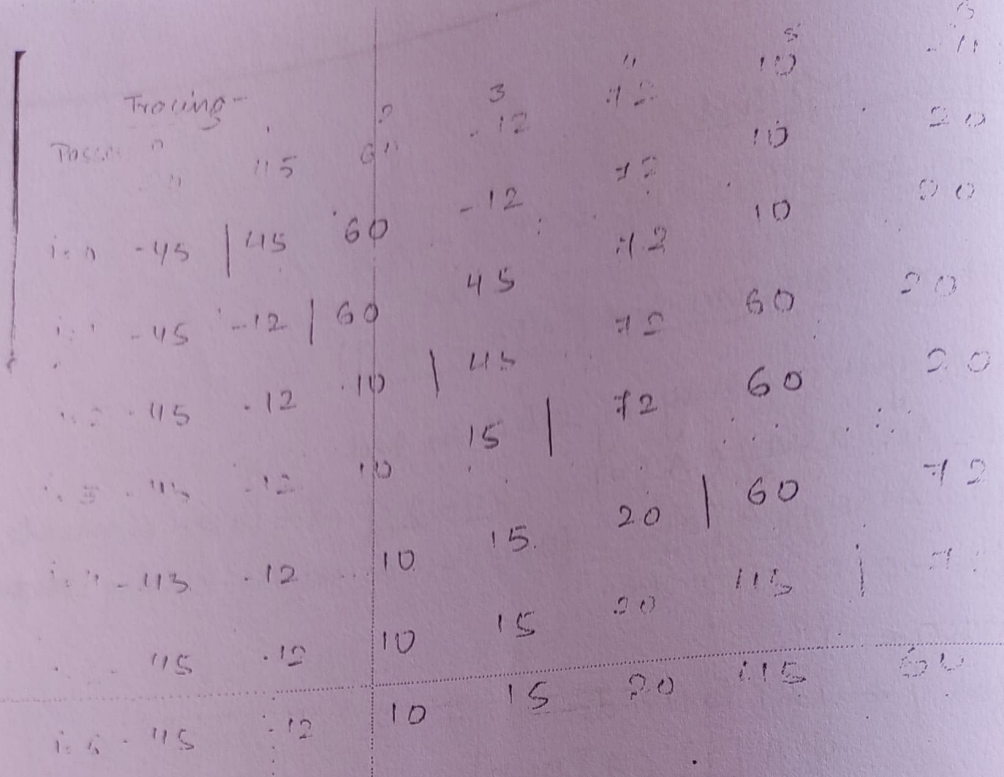
$$= \frac{n(n-1)}{2} \approx \frac{n^2}{2} \in \Theta(n^2).$$

$$E(n) = \sum_{i=0}^{n-2} 1 \quad \text{// (swapping)}$$

$$= n-2-0+1$$

$$= n-1.$$

$$\therefore \boxed{E(n) = n-1 \in \Theta(n)}$$



→ Bubble Sorting:

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

// Sorts a given array by bubblesort

// 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

  for j ← 0 to n-2-i do

    if A[j] > A[j+1]

      swap A[j] and A[j+1]

→ Analysis

Input size ← n

Basic operation ← comparison

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

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

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

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

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

$$= \frac{n(n-1)}{2} \approx \frac{n^2}{2} \in \Theta(n^2).$$

$$E(n) = \sum_{i=0}^{n-1} \sum_{j=0}^{n-1-i} 1$$

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

$$\therefore \boxed{E(n) = \frac{n^2}{2} \in \Theta(n^2)}$$

[Selection sort is better than bubble sort because its iteration is linear in nature i.e $E(n) \in \Theta(n)$ but in bubble sort $E(n) \in \Theta(n^2)$.]

Tracing :

	0	1	2	3	4	5	6	7
	20	45	60	-12	72	10	-45	15
i=0	20	45	-12	60	10	-45	15	72
i=1	20	-12	45	10	-45	15	60	72
i=2	-12	20	10	-45	15	45	60	72
i=3	-12	10	-45	15	20	45	60	72
i=4	-12	-45	10	15	20	45	60	72
i=5	-45	-12	10	15	20	45	60	72

	0	1	2	3	4	5	6	7
	20	45	60	-12	72	10	-45	15
i=0	20	45	-12	60	10	-45	15	72
i=1	20	-12	45	10	-45	15	60	72
i=2	-12	20	10	-45	15	45	60	72
i=3	-12	10	-45	15	20	45	60	72
i=4	-12	-45	10	15	20	45	60	72
i=5	-45	-12	10	15	20	45	60	72

Work-

1) Consider 10 elements and sort by selection and bubble sort

SELECTION SORT TRACE

	0	1	2	3	4	5	6	7	8	9
	20	9	45	60	52	68	-12	72	10	-45
i=0	-45	9	45	60	52	68	-12	72	10	20
i=1	-45	-12	45	60	52	68	9	72	10	20
i=2	-45	-12	9	60	52	68	45	72	10	20
i=3	-45	-12	9	10	52	68	45	72	60	20
i=4	-45	-12	9	10	20	68	45	72	60	52
i=5	-45	-12	9	10	20	45	68	72	60	52
i=6	-45	-12	9	10	20	45	52	72	60	68
i=7	-45	-12	9	10	20	45	52	60	72	68
i=8	-45	-12	9	10	20	45	52	60	68	72

∴ The traced sorting is done.

Bubble Sort:

Let the elements be- 20, 9, 45, 60, 52, 68, -12, 72, 10, -45.

BUBBLE SORT TRACE

	0	1	2	3	4	5	6	7	8	9
	20	9	45	60	52	68	-12	72	10	-45
i=0	9	20	45	52	60	-12	68	10	-45	72
i=1	9	20	45	52	-12	60	10	-45	68	72
i=2	9	20	45	-12	52	10	-45	60	68	72
i=3	9	20	-12	45	10	-45	52	60	68	72
i=4	9	-12	20	10	-45	45	52	60	68	72
i=5	-12	9	10	-45	20	45	52	60	68	72
i=6	-12	9	-45	10	20	45	52	60	68	72
i=7	-12	-45	9	10	20	45	52	60	68	72
i=8	-45	-12	9	10	20	45	52	60	68	72

29/1/19

- Sequential Search:

Algorithm SequentialSearch(A[0... n-1], k)
// • Implements sequential search with a given search key element.
// Input: An array A[0... n-1] and a search key element
// Output: returns the positions of the key element if it is found otherwise returns -1.

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

- Bruteforce string matching:

Algorithm BruteforceStringMatch(T[0.... n-1], P[0.... m-1])
// Implements bruteforce string matching.
// Input: An array T[0.... n-1] of n characters representing a text and an array P[0.... m-1] of m characters representing a pattern.
// Output: The index of first character in the text that starts a matching substring or -1 if the search is unsuccessful.

```
for i ← 0 to n-m+1 do   // for i ← 1 to n-m+1 do
    j ← 0
    while j<m and P[j]=T[i+j] do
        j ← j+1
    if j=m return i
return -1
```

COMPUTER n
7-3 m
=5

Tracing:

n = 25, m = 7.

Text: INFORMATION SCIENCE AND ENGG (0 1 2 3 4 5 6 7 8 ... 25)

i = 0 S
i = 1 S
i = 2 S
i = 3 S
i = 4 S
i = 5 S
i = 11 match. S

match

Then j = m returns 11

j = 1 P[1] = T[12] ✓
j = 2 P[2] = T[13] ✓
j = 5 P[5] = T[16] ✓ match.

→ j < m i.e 7 < 11 not equal goto if i.e j = m ✓ i.e 6 = 6
j = m True returns i i.e 11.

Total comparison made = 18.

Pattern: SCIENCE (0 1 2 3 4 5 6)

i = 0.
P[0] = T[0].
S = I not equal
comeout.
j = 0 i.e j ≠ m ×

i = 1
j = 0.
P[0] = T[1].
S = N ×.
j ≠ m × again for loop

i = 2.

Pattern: FORMULA (0 1 2 3 4 5 6)

A with U. ✗

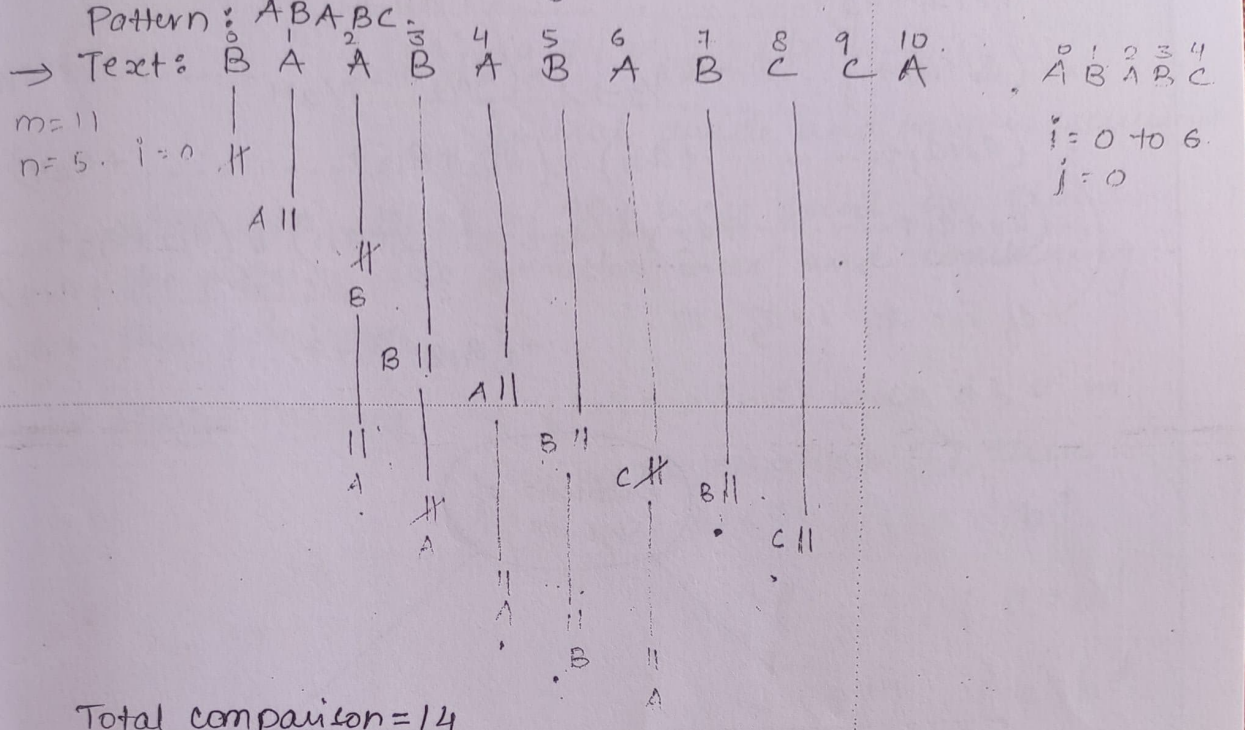
Text: INFORMATIONSCIENCEAND (0 1 2 3 4 5 6 7 8 9 10)

F ✗
F ✓
O ✓
R ✓
M ✓
A ✗
A ✗

Total 23 comparison.

1) Find the number of character comparison that can be made by straight forward string matching for the pattern ABABC in the following text, BAABABABCCA.

Pattern: ABABC.



Total comparison = 14

$C_{best}(n) = m \in \theta(m)$

$C_{worst}(n) = (n-m+1)m \in \theta(nm)$

$C_{avg}(n) = (n+m) \in \theta(m+n) \in \theta(n)$, linear.

∴ neglect m.