DESIGN AND ANALYSIS OF ALGORITHMS

Course Code: 19CS4DCDAA

Module-1 INTRODUCTION

Topics : What is an algorithm, Fundamentals of Algorithmic Problem Solving. Fundamentals of the Analysis of Algorithm Efficiency: The Analysis Framework, Asymptotic and Basic Efficiency Classes, Mathematical Analysis of Non recursive, Algorithms Mathematical Analysis of Recursive Algorithms**. Brute Force Method:** Introduction, Selection sort, linear search.

What Is an 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.

All algorithms must satisfy the following criteria:

1. Input. Zero or more quantities are externally supplied.
2. Output. At least one quantity is produced.
3. Definiteness. Each instruction is clear and produced.
4. Finiteness. If we trace out the instruction of an algorithm, then or all cases, the algorithm terminates after a finite number of steps.
5. Effectiveness. Every instruction must be very basic so that it can be arried out, in principal, by a person using only pencil and paper. It is not enough that each operation be definite as in criterion 3; it also must be feasible.

Example: **ALGORITHM** *Euclid(m, n)*

```
//Computes gcd(m, n) by Euclid's algorithm
//Input: Two nonnegative, 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
```

Eg; GCD(6,10)

m	*n*	*R=m%n*
6	*10*	*6=6%10*
10	*6*	*4=10%6*
6	*4*	*2=6%4*
4	*2*	*0=4%2*
2	*0*	*stop*

Consecutive integer checking algorithm for computing gcd*(m, n)*

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

Step 2: Divide *m* by *t*. If the remainder of this division is 0, go to Step 3;
otherwise, go to Step 4.

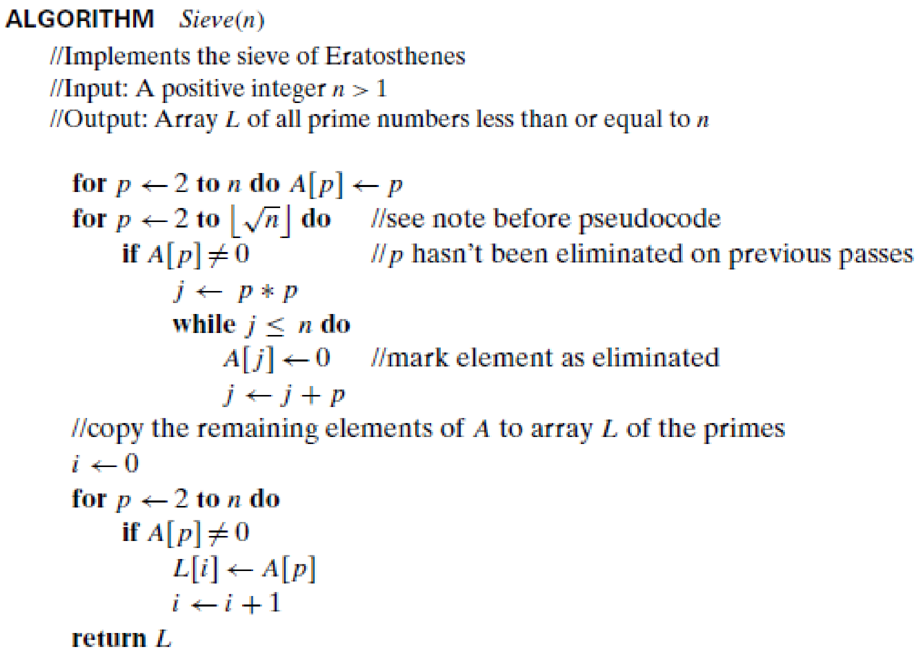
Step 3: Divide *n* by *t*. If the remainder of this division is 0, return the value of *t* as the answer and stop; otherwise, proceed to Step 4.

Step 4: Decrease the value of *t* by 1. Go to Step 2.

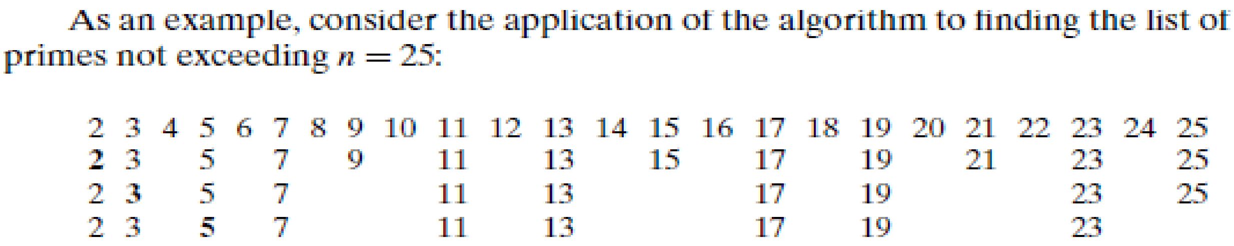
Eg: GCD(10,6)

min	*m%min*	*n%min*	*Is Small GCD?*
6	*10%6=4*		
5	*10%5=0*	*6%5=1*	
4	*10%4=2*	*6%4=2*	
3	*10%3=1*	*6%3=0*	
2	*10%2=0*	*6%2=0*	*2 is GCD*

Sieve of Eratosthenes



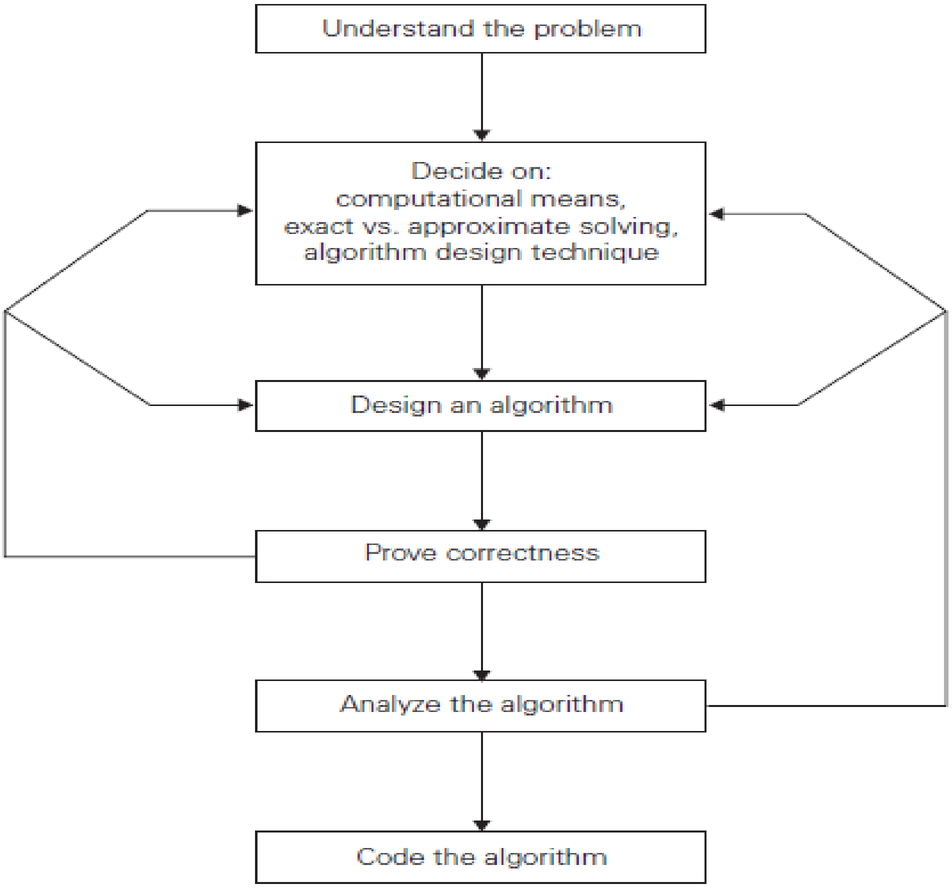
```
ALGORITHM  Sieve(n)
  //Implements the sieve of Eratosthenes
  //Input: A positive integer $n > 1$
  //Output: Array L of all prime numbers less than or equal to n

    for $p \leftarrow 2$ to n do $A[p] \leftarrow p$
    for $p \leftarrow 2$ to $\lfloor \sqrt{n} \rfloor$ do    //see note before pseudocode
        if $A[p] \neq 0$                //p hasn't been eliminated on previous passes
            $j \leftarrow p * p$
            while $j \leq n$ do
                $A[j] \leftarrow 0$     //mark element as eliminated
                $j \leftarrow j + p$
    //copy the remaining elements of A to array L of the primes
    $i \leftarrow 0$
    for $p \leftarrow 2$ to n do
        if $A[p] \neq 0$
            $L[i] \leftarrow A[p]$
            $i \leftarrow i + 1$
    return L
```



As an example, consider the application of the algorithm to finding the list of primes not exceeding $n = 25$:

2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25
2	3		5		7		9		11		13		15		17		19		21		23		25
2	**3**		5		7				11		13				17		19				23		25
2	3		**5**		7				11		13				17		19				23		

2. Fundamentals of Algorithmic Problem Solving:



Understanding the Problem

The first thing you need to do before designing an algorithm is to understand completely the problem given. Read the problem's description carefully and ask questions if you have any doubts about the problem, do a few small examples by hand, think about special cases, and ask questions again if needed.

Ascertaining the Capabilities of the Computational Device

Once you completely understand a problem, you need to ascertain the capabilities of the computational device the algorithm is intended for. The vast majority of algorithms in use today are still destined to be programmed for a computer closely

resembling the von Neumann machine. Its central assumption is that instructions are executed one after another, one operation at a time. Accordingly, algorithms designed to be executed on such machines are called sequential algorithms. The central assumption of the RAM model does not hold for some newer computers that can execute operations concurrently, i.e., in parallel. Algorithms that take advantage of this capability are called parallel algorithms

Choosing between Exact and Approximate Problem Solving

To choose between solving the problem exactly or solving it approximately. In the former case, an algorithm is called an exact algorithm; in the latter case, an algorithm is called an approximation algorithm.

Algorithm Design Techniques

An algorithm design technique is a general approach to solving problems algorithmically that is applicable to a variety of problems from different areas of computing. Algorithm design techniques make it possible to classify algorithms according to an underlying design idea; therefore, they can serve as a natural way to both categorize and study algorithms.

Designing an Algorithm and Data Structures

Designing an algorithm for a particular problem may still be a challenging task.

Some design techniques can be simply inapplicable to the problem in question. Sometimes, several techniques need to be combined, and there are algorithms that are hard to pinpoint as applications of the known design techniques. One should pay close attention to choosing data structures appropriate for the operations performed by the algorithm.

Methods of Specifying an Algorithm

Once you have designed an algorithm, you need to specify it in some fashion. There are the two options that are most widely used nowadays for specifying algorithms.

a. algorithm is described in words
b. pseudocode

Pseudocode is a mixture of a natural language and programming language like constructs. Pseudocode is usually more precise than natural language, and its usage often yields more succinct algorithm descriptions

Proving an Algorithm’s Correctness

Once an algorithm has been specified, you have to prove its correctness. You have to prove that the algorithm yields a required result for every legitimate input in a finite amount of time. For some algorithms, a proof of correctness is quite easy; for others, it can be quite complex. A common technique for proving correctness is to use mathematical induction because an algorithm’s iterations provide a natural sequence of steps needed for such proofs.

Analyzing an Algorithm

After correctness, by far the most important is efficiency. In fact, there are two kinds of algorithm efficiency: time efficiency, indicating how fast the algorithm runs, and space efficiency, indicating how much extra memory it uses.

Coding an Algorithm

Most algorithms are destined to be ultimately implemented as computer programs. Programming an algorithm presents both a peril and an opportunity. The peril lies in the possibility of making the transition from an algorithm to a program either incorrectly or very inefficiently.

Fundamentals of the Analysis of Algorithm Efficiency: The Analysis Framework

There are two kinds of efficiency: **time efficiency** and **space efficiency**. Time efficiency, also called **time complexity**, indicates how fast an algorithm in question runs. Space efficiency, also called **space complexity**, refers to the amount of memory units required by the algorithm in addition to the space needed for its input and output**.**

Units for Measuring Running Time

We can simply use some standard unit of time measurement—a second, or millisecond, and so on—to measure the running time of a program implementing the algorithm. The thing to do is to identify the most important operation of the algorithm, called the **basic operation**, the operation contributing the most to the total running time, and compute the number of times the basic operation is executed.

The established framework for the analysis of an algorithm's time efficiency suggests measuring it by counting the number of times the algorithm's basic operation is executed on inputs of size n. Let c_{op} be the execution time of an algorithm's basic operation on a particular computer, and let C(n) be the number of times this operation needs to be executed for this algorithm. Then we can estimate the running time T (n) of a program implementing this algorithm on that computer by the formula

$$T\ (n) \approx c_{op}\, C(n).$$

Orders of Growth

We expect all algorithms to work faster for all values on n. some algorithms will execute faster for smaller values on n, but as the n value increases they tend to very slow. This change in behavior can be analyzed by considering the highest order n.

The order of growth is usually determined for larger values of n.

N	Log N	N	N log N	N^2	N^3	2^N	N!
1	0	1	0	1	1	2	1
2	1	2	2	4	8	4	2
4	2	4	8	16	64	16	24
8	3	8	24	64	512	256	40320
16	4	16	64	256	4096	65536	High
32	5	32	160	1024	32768	4294967296	Very High

As N increases order of growth increases

- 1or any constant indicates the running time of program is constant.
- Log N: indicates the running time of program is logarithmic.
  Eg. Binary search (solving the large problems by reducing the problem in constant factor)
- N: indicates the running time of program is linear. Eg. Linear Search
- N log N: indicates the running time of program is N log N. Eg: quick sort, merge sort etc.
- N^2 : indicates the running time of program is quadratic. Algorithms which have two loops. Eg. Sorting algorithms like bubble sort, selection sort, addition and subtraction of two matrices.
- N^3: indicates the running time of program is cubic. Algorithms which have three loops. Eg. Matrix Multiplication
- 2^N: indicates the running time of program is exponential. Eg. Tower of Hanoi

- N!: indicates the running time of program is factorial.

Note: Order of growth from lowest to highest

$1<\log N<N<N\log N<N^2<N^3<2^N<N!$

Worst-case, Best-case, Average case efficiencies

Algorithm efficiency depends on the input size n. And for some algorithms efficiency depends on type of input. We have best, worst & average case efficiencies.

- **Worst-case efficiency:** Efficiency (number of times the basic operation will be executed) for the worst case input of size n. i.e. The algorithm runs the longest among all possible inputs of size n.
- **Best-case efficiency:** Efficiency (number of times the basic operation will be executed) for the best case input of size n. i.e. The algorithm runs the fastest among all possible inputs of size n.
- **Average-case efficiency**: Average time taken (number of times the basic operation will be executed) to solve all the possible instances (random) of the input. NOTE: NOT the average of worst and best case

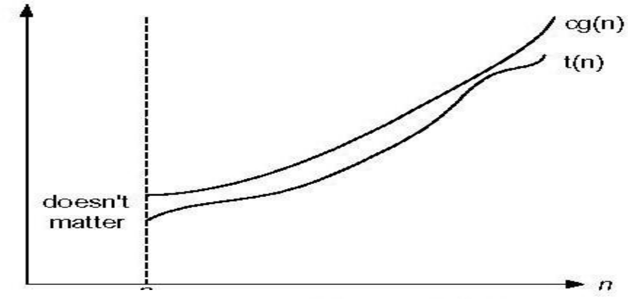
Asymptotic Notations

Asymptotic notation is a way of comparing functions that ignores constant factors and small input sizes. Three notations used to compare orders of growth of an algorithm‘s basic operation count are:

- **Big Oh- O notation Definition:**

A function t(n) is said to be in O(g(n)), denoted $t(n) \approx O(g(n))$, if t(n) is bounded above by some constant multiple of g(n) for all large n, i.e., if there exist some positive constant c and some nonnegative integer n0 such that

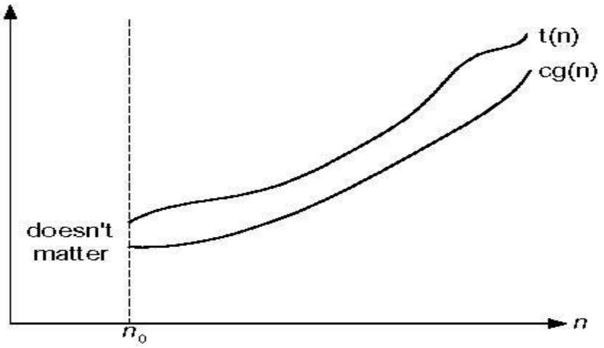
t(n) ≤ cg(n) for all n ≥ n0



Big-oh notation: $t(n) \in O(g(n))$

- **Big Omega- Ω notation**

A function t (n) is said to be in **Ω** (g(n)), denoted t(n) ≈ **Ω** (g (n)), if t (n) is bounded below by some constant multiple of g (n) for all large n, i.e., if there exist some positive constant c and some nonnegative integer n0 such that **t(n) ≥ cg(n) for all n ≥ n0**

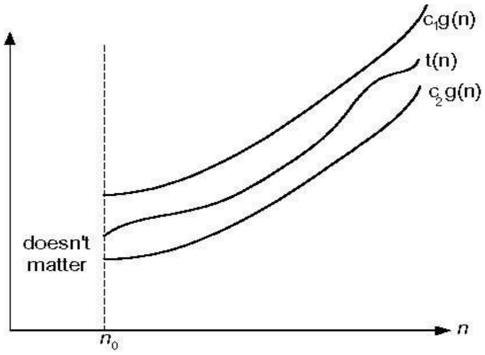


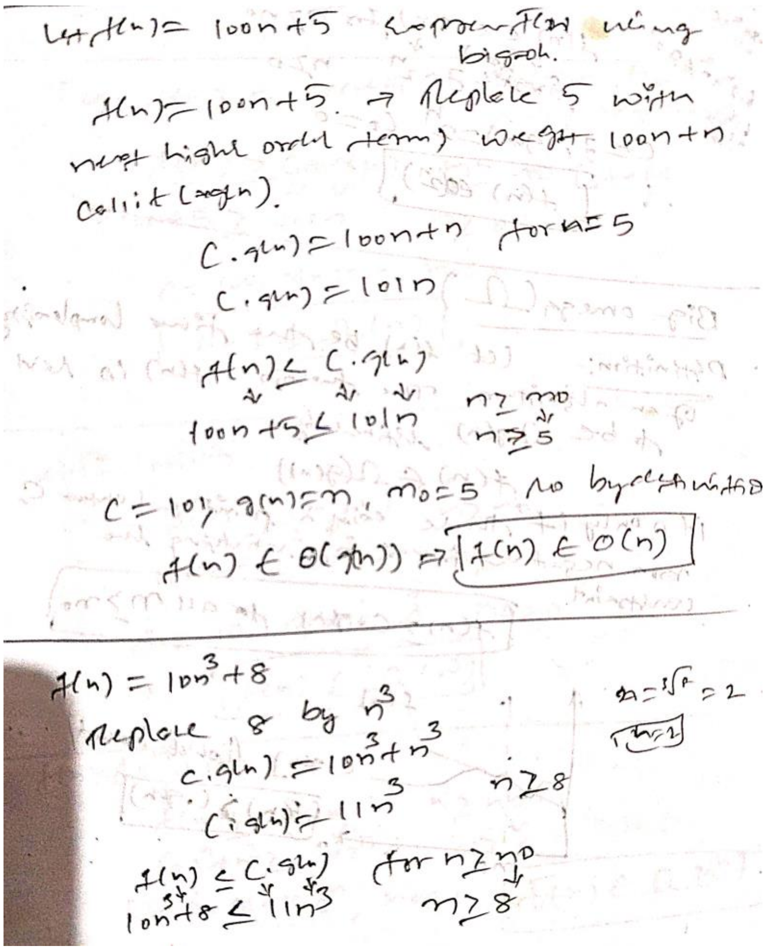
Big-omega notation: $t(n) \in \Omega(g(n))$

Big Theta- Θ notation

A function t (n) is said to be in Θ(g (n)), denoted t(n) ≈ Θ(g (n)), if t (n) is bounded both above and below by some constant multiple of g (n) for all large n, i.e., if there exist some positive constant c1 and c2 and some nonnegative integer n0 such that

c2 g (n) ≤ t (n) ≤ c1 g (n) for all n ≥ n0





Let $f(n) = 100n+5$ represent f(n) using big oh.

$f(n) = 100n+5$. → Replace 5 with next higher order term) we get $100n+n$ Call it (c.g(n)).

$c.g(n) = 100n+n$ for $n=5$

$c.g(n) = 101n$

$$f(n) \le c.g(n)$$

$$100n+5 \le 101n \qquad n \ge n_0,\ n \ge 5$$

$c = 101$, $g(n) = n$, $n_0 = 5$ no by substitution

$$f(n) \in O(g(n)) \Rightarrow \boxed{f(n) \in O(n)}$$

$f(n) = 10n^3+8$

Replace 8 by n^3

$c.g(n) = 10n^3+n^3$ $\qquad n \ge 8$

$c.g(n) = 11n^3$

$n = \sqrt[3]{8} = 2$

$\boxed{n=2}$

$f(n) \le c.g(n)$ for $n \ge n_0$

$10n^3+8 \le 11n^3$ $\qquad n \ge 8$

$c = 11, \ g(n) = n^3, \ n_0 = 8$

$$\boxed{f(n) \in O(n^3)}$$

$f(n) = 6 * 2^n + n^2$

$f(n) = 6 * 2^n + n^2$ replace n^2 with 2^n

$c \cdot g(n) = 6 * 2^n + 2^n$

$c \cdot g(n) = 7 * 2^n \quad n \geq 0$

2×16 [illegible] 112

$8 \times 7 = 56$

$6 \cdot 2^n + n^2 \leq 7 \cdot 2^n$

$6 \cdot 16 \quad 96 + 16 = 112$

$f(n) \leq c \cdot g(n)$ for $n \geq n_0$

$6 * 2^n + n^2 \leq 7 * 2^n \quad n \geq 0$

$c = 7, \ g(n) = 2^n \ c_0 = 0$

$$\boxed{f(n) \in O(2^n)}$$

Big omega (Ω)

Let $f(n) = 100n+5$ Represent $f(n)$ using big omega

$f(n) \geq c * g(n)$ $n \geq n_0$ c take

$100n+5 \geq 100n$ $n \geq 0$

$c = 100$, $g(n) = n$, $n_0 = 0$

$\boxed{f(n) \in \Omega(n)}$

$f(n) = 10n^3+5$

$f(n) \geq c * g(n)$ $n \geq n_0$

$10n^3+5 \geq 10n^3$ $n \geq 0$

$c = 10$, $g(n) = n^3$, $n_0 = 0$

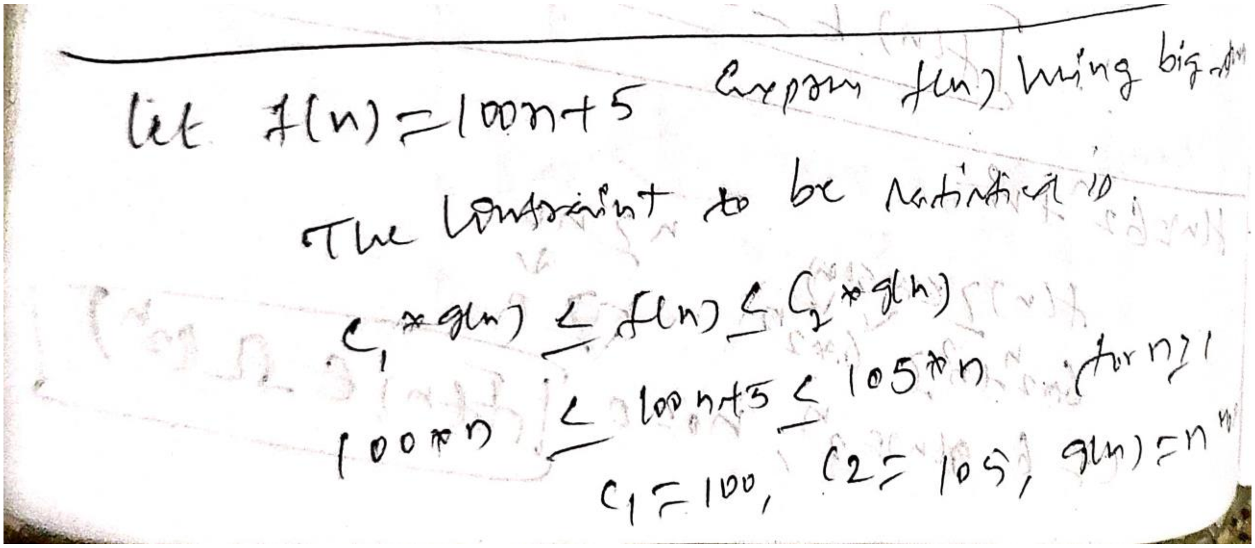
$\boxed{f(n) \in \Omega(n^3)}$

$f(n) = 6 2^n + n^2$

$f(n) \geq c * g(n)$ $n \geq n_0$

$6*2^n + n^2 \geq 6*2^n$ $n \geq 0$

$c = 6$, $g(n) = 2^n$, $n_0 = 0$ $\boxed{f(n) \in \Omega(2^n)}$



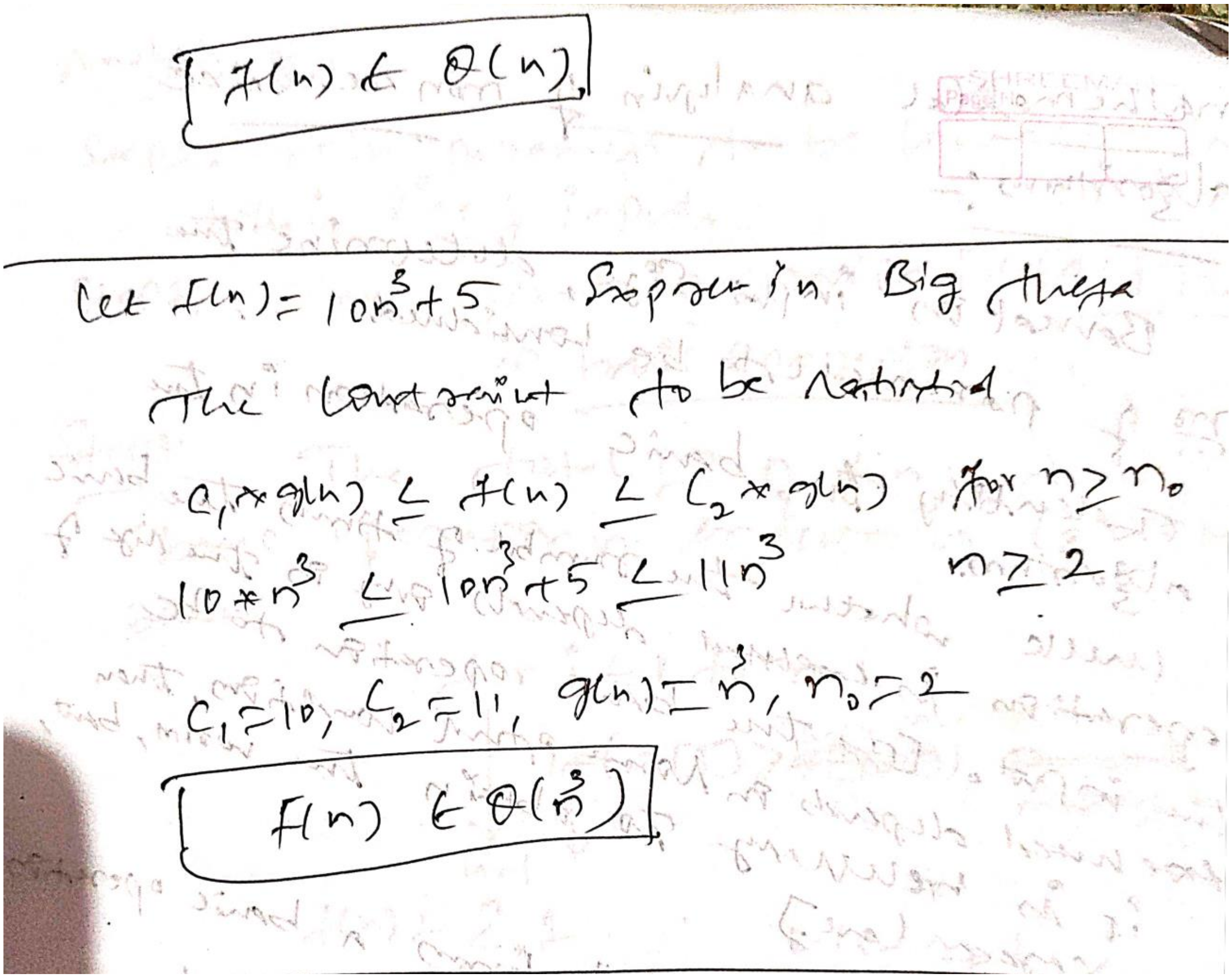
let $f(n) = 100n+5$ Express $f(n)$ using big-

The constraint to be satisfied is

$c_1 * g(n) \le f(n) \le c_2 * g(n)$

$100*n \le 100n+5 \le 105*n$ for $n \ge 1$

$c_1 = 100$, $c_2 = 105$, $g(n) = n$



$\boxed{f(n) \in \theta(n)}$

let $f(n) = 10n^3+5$ Express in Big theta

The constraint to be satisfied

$c_1 * g(n) \le f(n) \le c_2 * g(n)$ for $n \ge n_0$

$10*n^3 \le 10n^3+5 \le 11n^3$ $n \ge 2$

$c_1 = 10$, $c_2 = 11$, $g(n) = n^3$, $n_0 = 2$

$\boxed{f(n) \in \theta(n^3)}$

Important Mathematical formulas:

1. $\sum_{\mathrm{i=lb}}^{\mathrm{ub}} 1 = \mathrm{ub} - \mathrm{lb} + 1$

2. $\sum_{\mathrm{i=lb}}^{\mathrm{ub}} \mathrm{k} = \mathrm{k}(\mathrm{ub} - \mathrm{lb} + 1)$

3. $\sum_{\mathrm{i=1}}^{\mathrm{n}} \mathrm{i} = 1 + 2 + 3 + 4 + \cdots + n = n(n + 1)/2$

Mathematical Analysis of Non-Recursive and Recursive Algorithms

Mathematical analysis (Time Efficiency) of Non-recursive Algorithms

General plan for analyzing efficiency of non-recursive algorithms:

1. Decide on parameter n indicating input size

2. Identify algorithm‘s basic operation

3. Check whether the number of times the basic operation is executed depends only on the input size n. If it also depends on the type of input, investigate worst, average, and best case efficiency separately.

4. Set up summation for C(n) reflecting the number of times the algorithm‘s basic operation is executed.

5. Simplify summation using standard formulas

Example: Finding the largest element in a given array

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

```
//Determines the value of largest element in a given array

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

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

currentMax ← A[0]

for i ← 1 to n - 1 do

if A[i] > currentMax

currentMax ← A[i]
```

return currentMax

Analysis:

1. Input size: number of elements = n (size of the array)

2. Basic operation:

a) Comparison

b) Assignment

3. NO best, worst, average cases.

4. Let C (n) denotes number of comparisons: Algorithm makes one comparison on each execution of the loop, which is repeated for each value of the loop‘s variable i within the bound between 1 and n – 1.

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

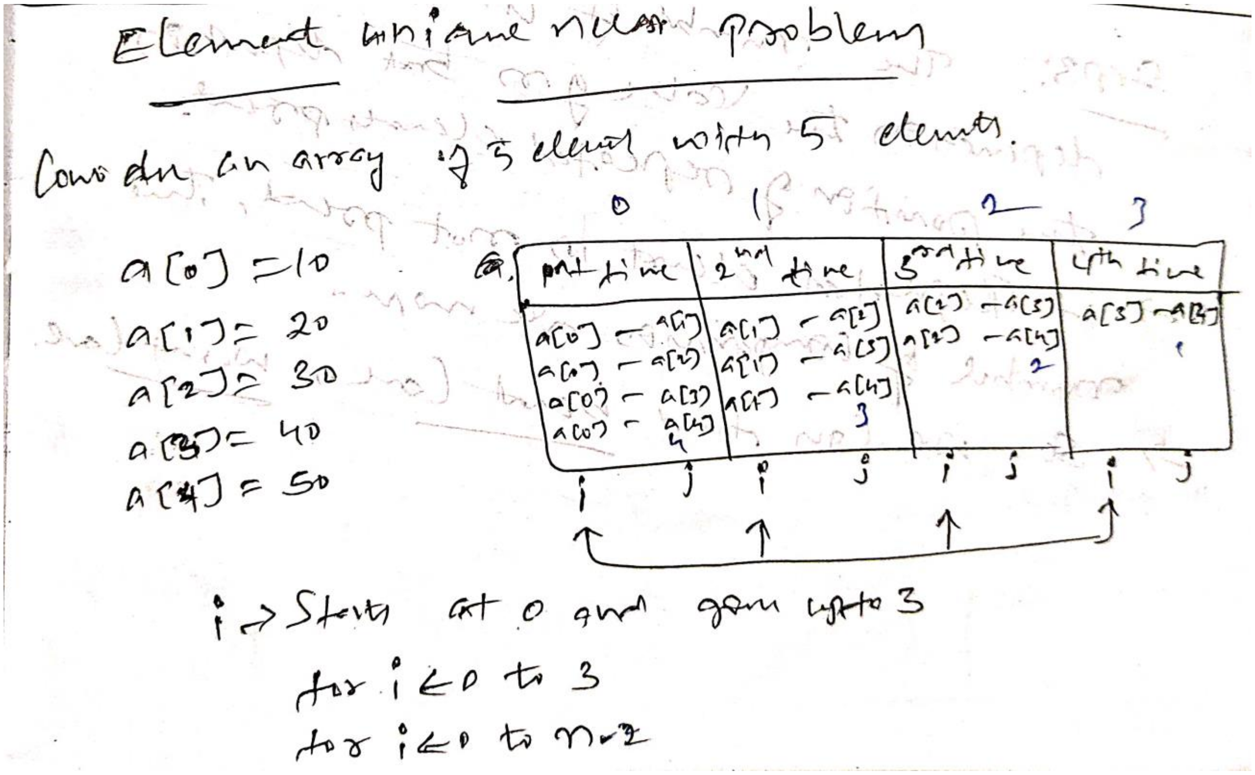
5. **Simplify summation** using standard formulas

$$C(n) = \sum_{i=1}^{n-1} 1 \qquad \begin{array}{l} 1+1+1+\ldots+1 \\ \text{[(n-1) number of times]} \end{array}$$

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

$$C(n)\ \square\ \Theta(n)$$

Example: Element uniqueness problem



Element uniqueness problem

Consider an array of 5 elemts with 5 elements.

a[0] = 10

a[1] = 20

a[2] = 30

a[3] = 40

a[4] = 50

0 1st time	1 2nd time	2 3rd time	3 4th time
a[0] – a[1] a[0] – a[2] a[0] – a[3] a[0] – a[4] 4	a[1] – a[2] a[1] – a[3] a[1] – a[4] 3	a[2] – a[3] a[2] – a[4] 2	a[3] – a[4] 1
i j	i j	i j	i j

i → Starts at 0 and goes upto 3

for i ← 0 to 3

for i ← 0 to n-2

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

```
//Checks 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
```

Analysis

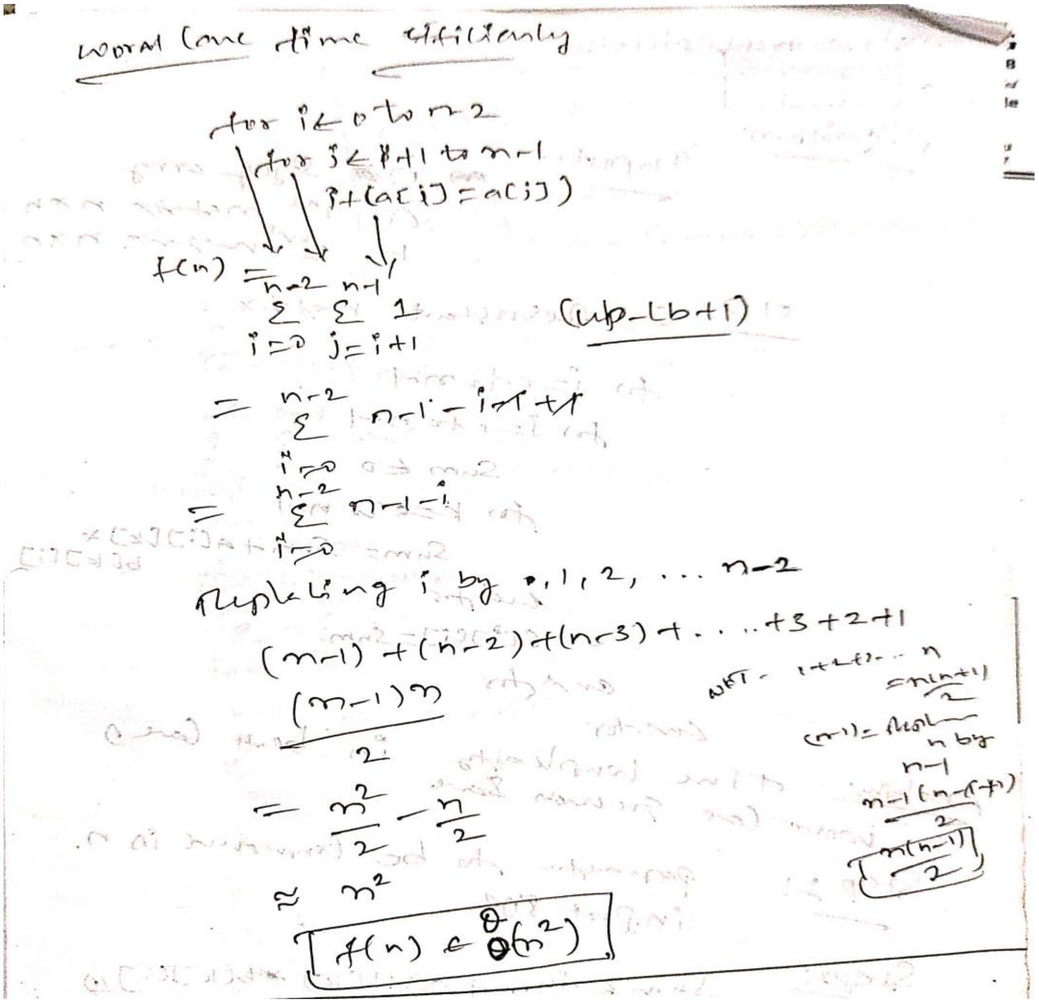
1. Input size: number of elements = n (size of the array)
2. Basic operation: Comparison

3. Best, worst, average cases EXISTS.

Worst case input is an array giving largest comparisons.

- Array with no equal elements
- Array with last two elements are the only pair of equal elements

4. Let C (n) denotes number of comparisons in worst case: Algorithm makes one comparison for each repetition of the innermost loop i.e., for each value of the loop's variable j between its limits i + 1 and n – 1; and this is repeated for each value of the outer loop i.e, for each value of the loop's variable i between its limits 0 and n – 2



worst case time efficiently

```
for i ← 0 to n-2
    for j ← i+1 to n-1
        if (a[i] = a[j])
```

$$f(n) = \sum_{i=0}^{n-2} \sum_{j=i+1}^{n-1} 1 \qquad \underline{(ub-lb+1)}$$

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

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

Replacing i by 0, 1, 2, ... n-2

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

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

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

$$\approx n^2$$

$$\boxed{f(n) \in \Theta(n^2)}$$

WKT - $1+2+3\dots n = \frac{n(n+1)}{2}$

$(n-1)$ = replace n by $n-1$

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

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

matrix multiplication

Algorithms:

Inputs: m, n size of arry
a. 1st matrix n×n
b. 2nd matrix, n×n

O/P C: Resultant matrix

```
for i=0 to n-1 | 1 to n
    for j=0 to n-1 | 1 to n
    c[i][j]←0 sum ←0
        for k=0 to n-1 | 1 to n
            sum= sum+a[i][k]* b[k][j]
            c[i][j] ← c[i][j]+
        endfor
    ~~c~~// c[i][j]=sum
    endfor
endfor
```

Analysis: Time complexity in best case worst case average same

Step 1: parameter to be considered is n. input size.

Step 2: sum←sum+a[i][k]*b[k][j] is the basic operation.

Step3:

```
for i ← 0 to n-1
    for j ← 0 to n-1
        Sum ← 0
        for k ← 0 to n-1
            Sum ← Sum + a[i][k] * b[k][j]
```

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

ub-lb+1

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

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

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

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

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

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

$$n^2(n-\cancel{1}-0+\cancel{1})$$

$$t(n) = \underline{\underline{n^3}}$$

∴ time complexity is

$$\boxed{t(n) \in \Theta(n^3)}$$

Mathematical analysis (Time Efficiency) of recursive Algorithms

General plan for analyzing efficiency of recursive algorithms:

1. Decide on parameter n indicating input size

2. Identify algorithm‛s basic operation

3. Check whether the number of times the basic operation is executed depends only on the input size n. If it also depends on the type of input, investigate worst, average, and best case efficiency separately.

4. Set up recurrence relation, with an appropriate initial condition, for the number of times the algorithm‛s basic operation is executed.

5. Solve the recurrence.

Example: Factorial function

ALGORITHM Factorial (n)

```
//Computes n! recursively
//Input: A nonnegative integer n
//Output: The value of n!
if n = = 0 return 1 else
return Factorial (n – 1) * n
```

Analysis:

1. Input size: given number = n

2. Basic operation: multiplication

3. NO best, worst, average cases.

4. Let M (n) denotes number of multiplications.

   $M(0) = 0$ initial condition

   $M(n) = M(n-1) + 1$ for $n > 0$

Where: $M(n-1)$: to compute Factorial $(n-1)$

1 :to multiply Factorial $(n-1)$ by n

5. Solve the recurrence: Solving using “Backward substitution method”:

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

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

$= M(n-2) + 2$

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

$= M(n-3) + 3$

…

In the ith recursion, we have

$= M(n-i) + i$

When $i = n$, we have

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

Since $M(0) = 0$

$= n$

$M(n) \in \Theta(n)$

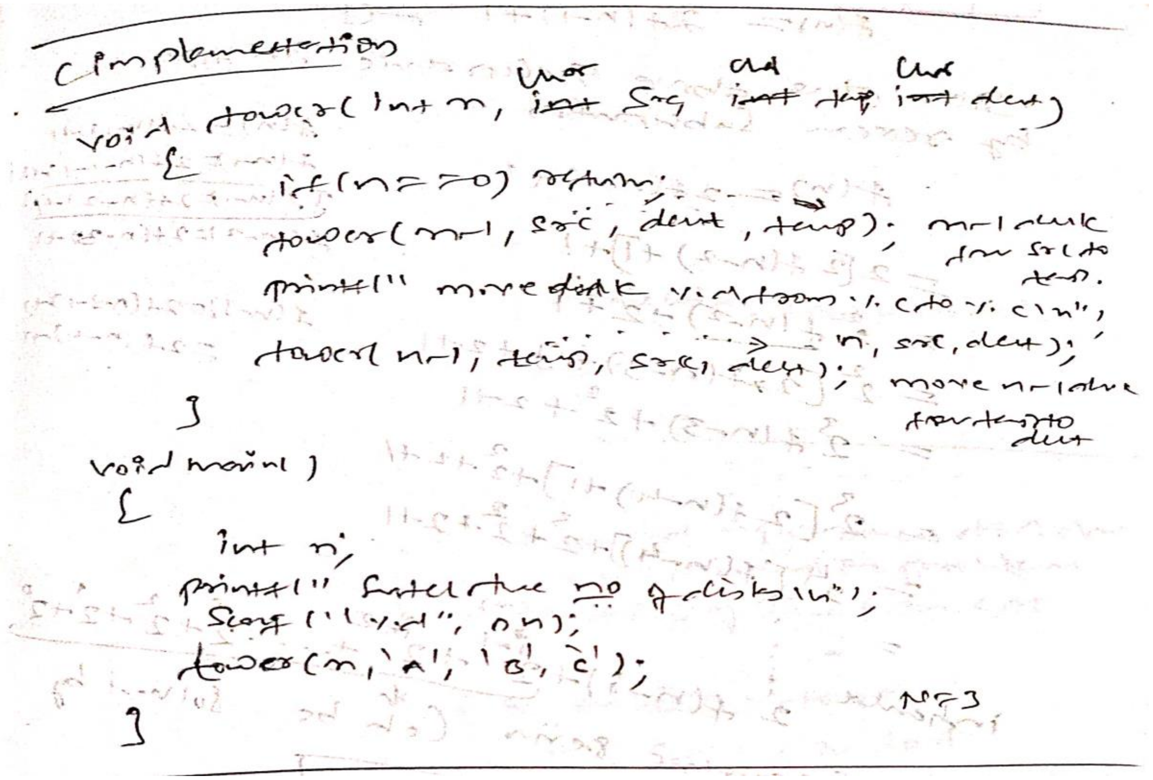
Eg: Tower of Hanoi

Tower of Hanoi is a mathematical puzzle where we have three rods and n disks. The objective of the puzzle is to move the entire stack to another rod, obeying the following simple rules:

1) Only one disk can be moved at a time.

2) Each move consists of taking the upper disk from one of the stacks and placing it on top of another stack i.e. a disk can only be moved if it is the uppermost disk on a stack.

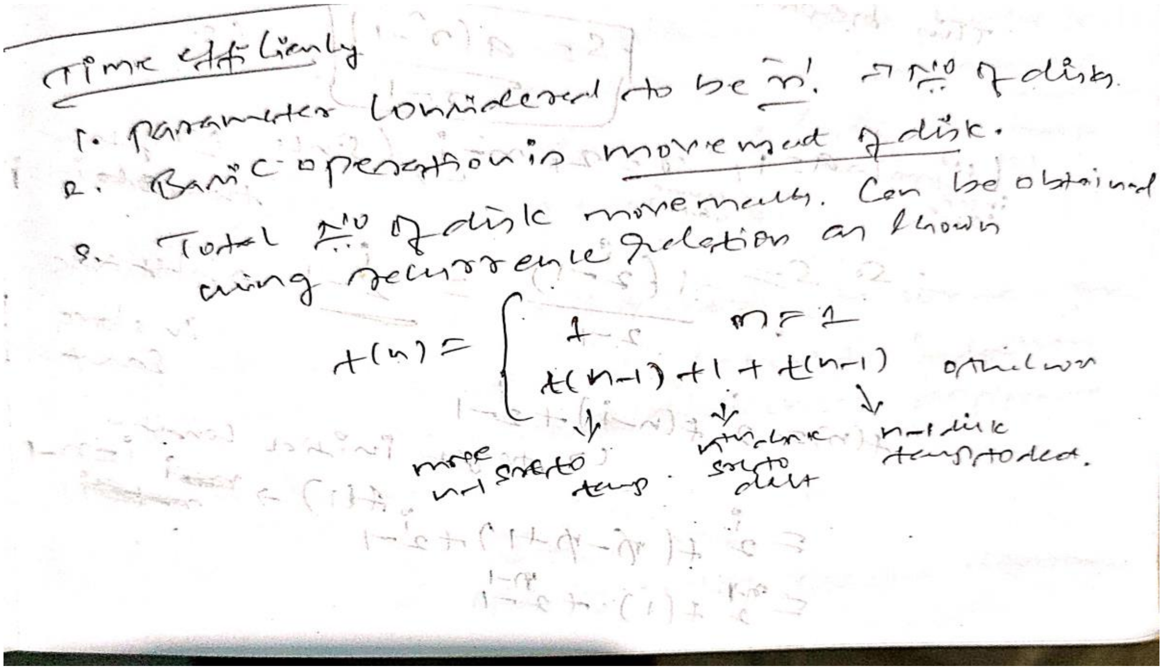
3) No disk may be placed on top of a smaller disk.



C implementation

```
void tower(int n, ~~int~~ char src, ~~int~~ char temp, ~~int~~ char dest)
{
        if(n==0) return;
        tower(n-1, src, dest, temp);     n-1 disk from src to temp.
        printf(" move disk %d from %c to %c\n", n, src, dest);
        tower(n-1, temp, src, dest);     move n-1 disk from temp to dest
}
void main()
{
      int n;
     printf(" Enter the no of disks\n");
     scanf("%d", &n);
     tower(n, 'A', 'B', 'C');
}
                                          n=3
```

https://www.transum.org/Maths/Investigation/Tower_Of_Hanoi/Default.asp?Level=3



Time efficiency

1. parameter considered to be 'n'. → no of disks.
2. Basic operation is movement of disk.
3. Total no of disk movements. Can be obtained using recurrence relation as shown

$$t(n) = \begin{cases} 1 & n=1 \\ \underbrace{t(n-1)}_{\text{move n-1 src to temp}} + \underbrace{1}_{\text{nth disk src to dest}} + \underbrace{t(n-1)}_{\text{n-1 disk temp to dest.}} & \text{otherwise} \end{cases}$$

$\therefore \quad f(n) = f(n-1) + 1 + f(n-1)$

$f(n) = 2f(n-1) + 1 \quad — ①$

Solve the above recurrence relation by reverse substitution

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

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

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

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

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

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

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

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

$= 2^2 \underline{f(n-2)} + 2 + 1$

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

$= 2^3 f(n-3) + 2^2 + 2 + 1$

$= 2^3[2f(n-4)+1] + 2^2 + 2 + 1$

$= 2^4.[f(n-4)] + 2^3 + 2^2 + 2 + 1$

$= \quad \vdots$

in general $2^i f(n-i) + \underbrace{2^{i-1} + 2^{i-2} + \dots 2^3 + 2^2 + 2^1 + 2^0}_{}$ (n)

This geometric series can be solved by

$$\boxed{S = \frac{a(r^n-1)}{r-1}}$$ (Si

(1st term) $\rightarrow a = 1$, $r = 2$, $\underset{\rightarrow}{n = i}$ (since No of term from 0 to $i-1 = i$)

in series i.e, $2^0 = 1$ (common ratio)

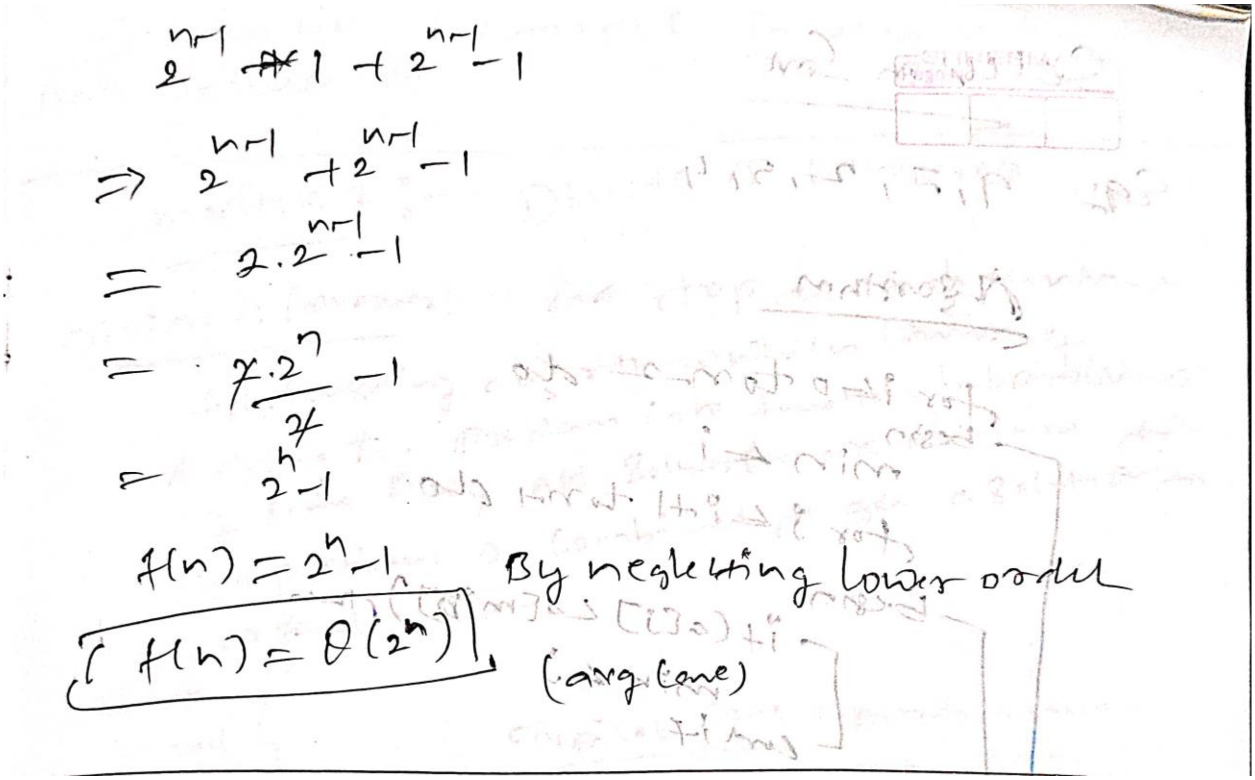
So $S = \dfrac{1(2^i-1)}{2-1} = \underline{2^i - 1}$ → substitute in above eqn

$f(n) = 2^i f(n-i) + 2^i - 1$

(→ to get initial condition $i = n-1$, $f(1) \rightarrow$

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

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



$2^{n-1} \cancel{*} 1 + 2^{n-1} - 1$

$\Rightarrow 2^{n-1} + 2^{n-1} - 1$

$= 2 \cdot 2^{n-1} - 1$

$= \frac{\cancel{2} \cdot 2^{n}}{\cancel{2}} - 1$

$= 2^{n} - 1$

$f(n) = 2^{n} - 1$, By neglecting lower order

$\boxed{f(n) = \theta(2^{n})}$ (avg. Case)

Brute Force:

Brute force is a straightforward approach to problem solving, usually directly based on the problem‘s statement and definitions of the concepts involved. Though rarely a source of clever or efficient algorithms, the brute-force approach should not be overlooked as an important algorithm design strategy. Unlike some of the other strategies, brute force is applicable to a very wide variety of problems. For some important problems (e.g., sorting, searching, string matching),the brute-force approach yields reasonable algorithms of at least some practical value with no limitation on instance size Even if too inefficient in general, a brute-force algorithm can still be useful for solving small-size instances of a problem. A brute-force algorithm can serve an important theoretical or educational purpose.

Selection Sort:

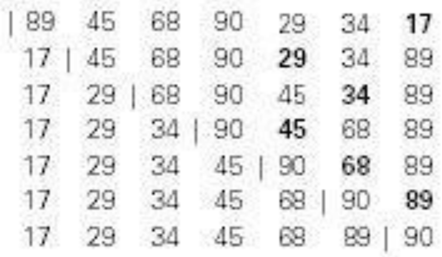
Problem: Given a list of n orderable items (e.g., numbers, characters from some alphabet, character strings), rearrange them in non- decreasing order.

ALGORITHM Selection Sort(A[0..n - 1])

```
//The 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
        end if
    end for
    swap A[i] and A[min]
end for
```

Example: i=1, min=4,j=7

i=0	1	2	3	4	5	6
89	45	68	90	29	34	17
17	45	68	90	29	34	89
17	29	68	90	45	34	89



| 89	45	68	90	29	34	**17**
17	| 45	68	90	**29**	34	89
17	29	| 68	90	45	**34**	89
17	29	34	| 90	**45**	68	89
17	29	34	45	| 90	**68**	89
17	29	34	45	68	| 90	**89**
17	29	34	45	68	89	| 90

Selection sort's operation on the list 89, 45, 68, 90, 29, 34, 17. Each line corresponds to one iteration of the algorithm, i.e., a pass through the list's tail to the right of the vertical bar; an element in bold indicates the smallest element found. Elements to the left of the vertical bar are in their final positions and are not considered in this and subsequent iterations.

Performance Analysis of the selection sort algorithm: The input‘s size is given by the number of elements n. The algorithm‘s basic operation is the key comparison A[j]<A[min]. The number of times it is executed depends only on the array‘s size and is given by

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

Thus, selection sort is a O(n^2) algorithm on all inputs. The number of key swaps is only O(n) or, more precisely, n-1 (one for each repetition of the i loop).This property distinguishes selection sort positively from many other sorting algorithms

EG: Apply Selection sort for the word EXAMPLE

Sequential Search:

Linear Search : The idea behind linear search is to compare the search item with the elements in the list one by one (using a loop) and stop as soon as we get the first copy of the search element in the list. Now considering the worst case in which the search element does not exist in the list of size **N** then the **Simple Linear Search** will take a total of **2N+1** comparisons (**N** comparisons against every element in the search list and **N+1** comparisons to test against the end of the loop condition).

```
int linearsearch(int  arr[], int n, int key)
{
   int i;
   for (i = 0; i < =n-1;  i++)
         {
      if (arr[i] == key)
         return i;
         }
   return -1;
}
```

Efficiency: **Best Case:** Occurs when key element is present in 1st location.

Worst Case: if key element is not present or if key element is at last location.

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

=(n-1-0+1)

= O(n)

Sentinel Linear Search : Here the idea is to reduce the number of comparisons required to find an element in a list. Here we replace the last element of the list with the search element itself and run a **while loop** to see if there exists any copy of the search element in the list and quit the loop as soon as we find the search element.

Here we see that the **while loop** makes only one comparison in each iteration and it is sure that it will terminate since the last element of the list is the search element itself. So in the worst case (if the search element does not exists in the list) then there will be at most **N+1** comparisons (**N** comparisons in the while loop and **1** comparisons in the if condition). Which is better than (**2N+1**) comparisons as found in **Simple Linear Search**.

Take note that both the algorithms have time complexity of **O(n)**.

Sequential Search

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

```
//The algorithm implements sequential search with a search key as a
sentinel
//Input: An array A of n elements and a search key K
//Output: The position of the first element in A[0..n - 1] whose value is
// equal to K or -1 if no such element is found
A[n]=K
```

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

Example:

k=6, i=5

i=0	1	2	3	4	5
10	20	30	5	65	6