Chapter 1

① What is an Algorithm? ***

An algorithm is a finite set of step by step instructions that is followed to solve any problem. It is followed to accomplished a particular task.

Properties of a good Algorithm: ****[[illegible]]

1. Input: Zero or more quantities are externally supplied.
2. Output: At least one quantity is produced.
3. Definiteness: Each instruction is clear & unambiguous.
4. Finiteness: If we trace out the instructions of an algorithm, then for all cases, the algorithm terminates after a finite number of steps.
5. Effectiveness: Every instruction must be very basic, so that it can be carried out, in principle, by a person using only pencil and paper.

How to devise an algorithm ?

Define the problem clearly, including the input and the desired output. Identify the main steps needed to reach the solution, thinking logically and in sequence. Use simple pseudocode to outline the steps.

How to validate an algorithm ?

Once an algorithm in devised, it is necessary to show that it computes the correct answer for all possible legal inputs. We refer to these process as validation. The purpose of the validation is to assure that this algorithm will work correctly, independently. After this step, program varification starts.

Ⓐ How to analyze an Algorithm?

As an algorithm is executed, it uses the computer's central processing unit (cpu) to perform operations and it's memory. Analysis of algorithm refer's to the task of determining how much computing time and storage an algorithm requires.

Ⓐ How to test an algorithm?

Testing a program consists of two phases. Debugging and profiling (or performance measurement). Debugging is the process of executing programs on sample data set. Run the program as a whole to confirm it meets all requirements and performs correctly in a realistic environment.

Complexity: Complexity referr's to the amount of ressource a program uses relative to the size of input, typically measured in time and space Complexity.

1. **Time Complexity:** Indicates how the run-time of an algorithm changes with the size of the input. It's commonly expressed in Big O notation. $O(n), O(n^2)$ which describes the upper bound of time taken as input size grows.

2. **Space Complexity:** Measures the amount of memory (space) an algorithm needs to execute, also expressed in Big O notation (e.g. $O(n)$, $O(1)$).

Asymptotic Notation: Asymptotic notation in algorithm is a way to describe the behaviour of an algorithm's complexity as the input size grows towards infinity. It allows us to analyze the efficiency of an algorithm

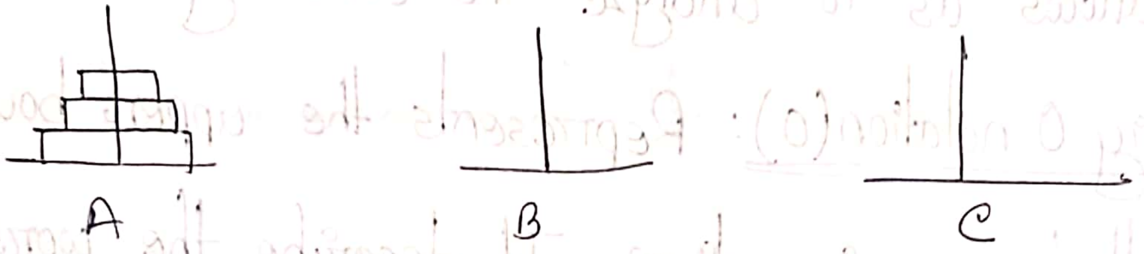
1) Big O notation(O): Represents the upper bound of an algorithm's running time. It describe the worst case scenario, ~~giving~~. For example, $O(n^2)$ means that the input size n increases with run-time grows.

2. Omega notation(Ω): Describes the lower bound of an algorithm's performances, representing the best scenario. for example, $\Omega(n)$ means the run time grows at least lineally with input size.

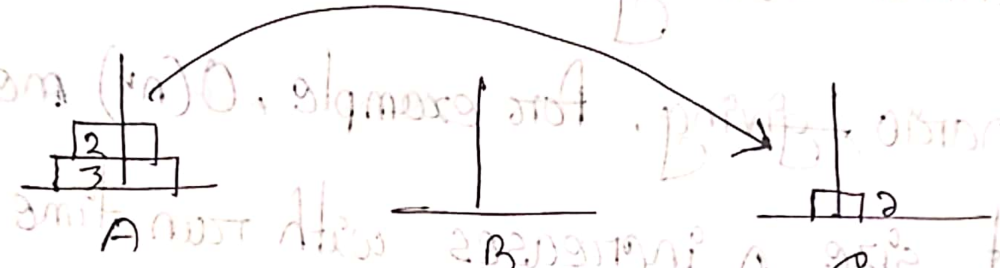
3. Theta notation(Θ): Provide a tight bound by combining both upper and lower bounds. When an algorithm is $\Theta(n)$, it's run time grows exactly

linearly with input size in both best and worst cases.

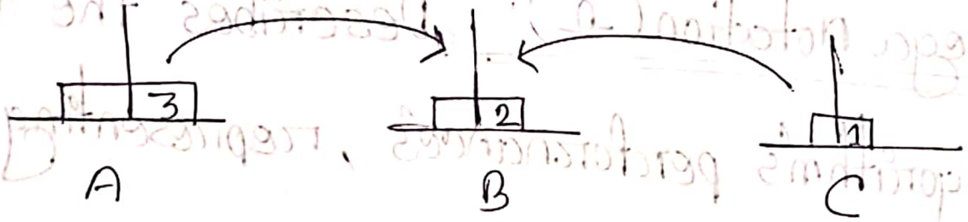
Tower of Hanoi using recursive fn.



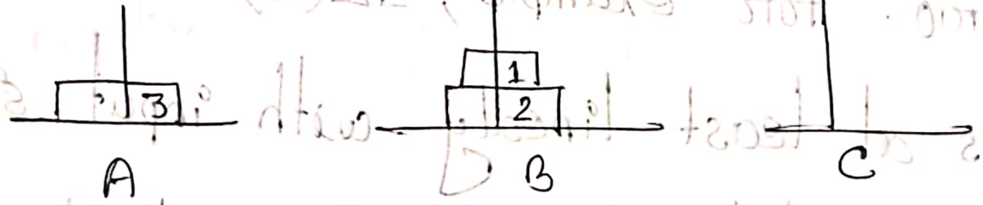
Step 1:



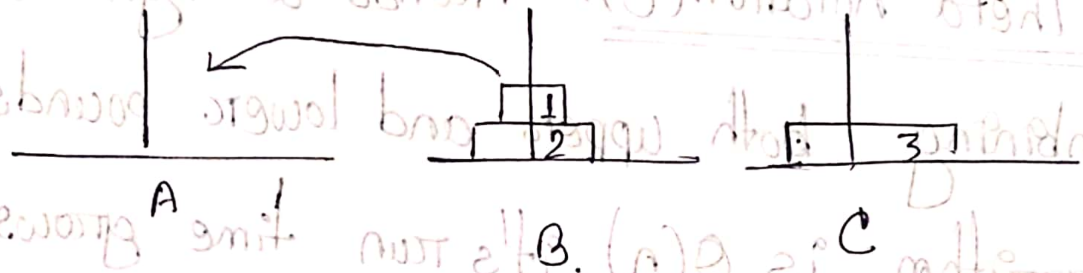
Step 2:



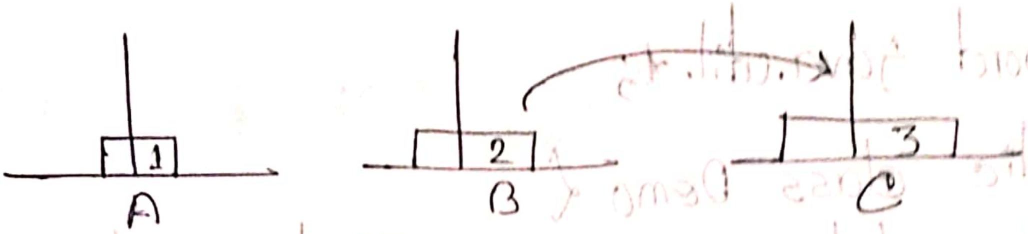
Step 3:



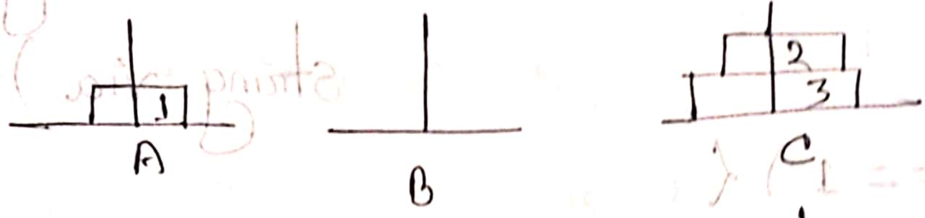
Step 4:



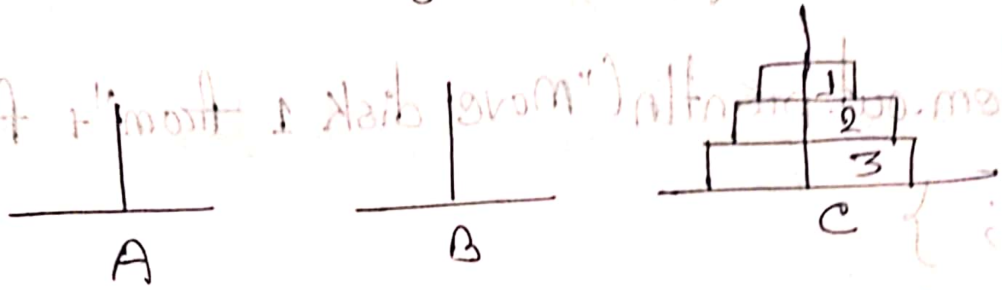
Step 5:



Step 6:



Step 7:



Stepwise Movement

move disk	from Pole	To target Pole
1	A	C
2	A	B
1	C	A
1	A	B
3	A	C
1	B	A
2	B	C
1	A	C

2^n
$= 2^3$
$= 8$

```
import java.util.*;
public class Demo {
public static void hanoi (int n, String from, String to,
                               string via ) {
if (n==1) {
 System.out.println("Move disk 1 from"+ from +" to"+
 to); }
 else {
       hanoi(n-1, from , via, to);
System.out.println ("move disk "+ n + " from " + from " to"
+ to);
 hanoi (n-1, via , to, from ); } }
public static void main (String[] args ) {
int    n= 3;
string  from = "A";
string   to  = "B";
string   via = "B";
hanoi (n, from, via, to);
 }
}
```

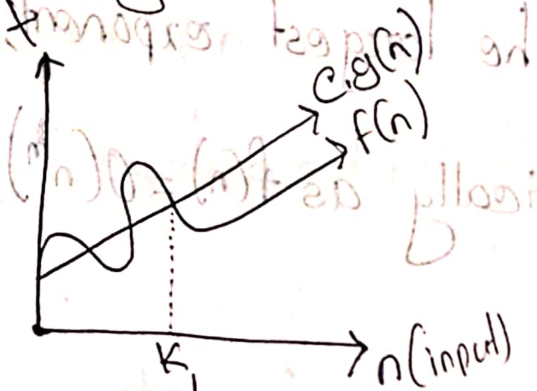
Theorem 1.4 :- If $f(n) = a_m n^m + \cdots + a_1 n + a_0$ and $a_m > 0$, then $f(n) = \Theta(n^m)$.

Soln: Since the highest degree ~~fun~~ term in the polynomial is $a_m n^m$ with m as the largest exponent, we can approximate $f(n)$ aysmptotically as $f(n) = O(n^m)$.

Asymptotic Notation

Mathematic way to represent the time Complexity

① Big O notation (to do any work maximum time)



$f(n) = O(g(n))$; c = constant

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

$c > 0$

$n \geq K$

$K \geq 0$

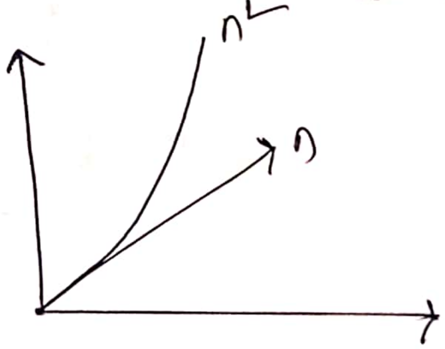
→ worst case

→ least upper bound (At most)

$f(n) = 2n^2+n$ we have to represent

$f(n) = O(\downarrow)$ what

∴ $\underset{\substack{\downarrow \\ \text{Quadratic fn}}}{2n^2} + \underset{\searrow \text{ linear eqn}}{n} \le c.g(n^2)$



here n^2 is dominating

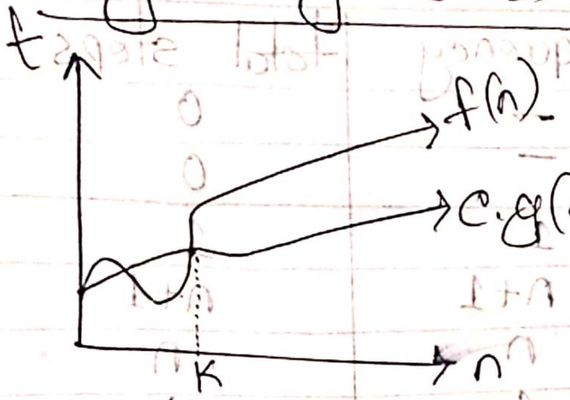
$\Rightarrow 2n^2+n \le 3n^2$

$\Rightarrow n \le n^2$

$\therefore 1 \le n$

$: n \geq 1$

2) Big Omega (Ω)



$f(n) = \Omega\, g(n)$

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

$2n^2 + n \geq c.n^2$

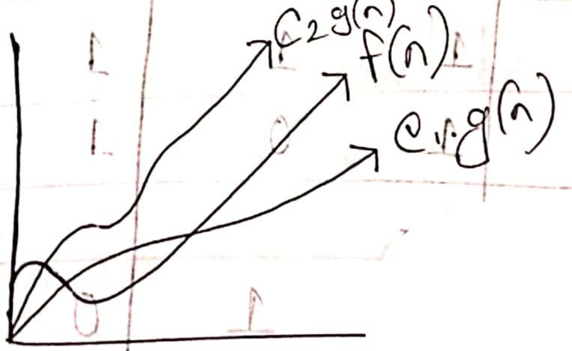
$2n^2 + n \geq 2.n^2$

→ Best Case

greatest

→ Lower Bound (At least) ⇒ $n \geq 0$

3) Theta notation (Θ)



$c_1.g(n) \leq f(n) \leq c_2.g(n)$

$2n^2 \leq 2n^2 + n \leq 3n^2$

→ Averagage Case

→ Exact time

$\boxed{f(n)} \leq c.g(n)$

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

statement	s/e.	frequency	total steps
1. Algorithm Sum(a,n)	0	—	0
{	0	—	0
S:=0.0;	1	1	1
for i:=1 to n do	1	n+1	n+1
S:=S+a[i];	1	n	n
return S;	1	1	1
}	0	—	0
			total = 2n+3

Magic Square

Algorithm RSum(a,n)	s/e	frequen n=0	n>0	total ste. n=0	n>0
{					
if (n≤0) then		1	1	1	1
return 0.0;		1	0	1	0
else return		0 -			
Rsum(a,n-1)+a[n];		0	1	0	1+x
		-		0	0
}				2	2+x

statement	s/e	frequency	total steps
1. Algorithm Add(a,b,c,m,n)	0	—	0
{	0	—	0
for i:=1 to m do	1	m+1	m+1
for j:=1 to n do	1	m(n+1)	mn+m
c[i][j]:=a[i][j]+b[i][j]	1	mn	mn
}	0	—	0
			2mn+2m+1

Magic Square

```
import java.util.*;
public class main {
public static void main(string[]args) {
Scanner sc= new Scanner(System.in);
Sout("Enter the order ");
int  n = sc.nextInt();
If (n%4!=0){
Sout ("Order must be a multiple of 4");
return ;}
int [][] magic = new int[n][n];
for (int = 0; i<n ;i++){
for (int j=0 ; j<n ; j++){
magic [i][j] =(n*i+j+1+n/2*(i+j))%(n*n)+1;
}
}  for (int i=0 ;i<n ;i++){
   for (int j=0; j<n; j++) {
   Sout ( magic[i][j] + " "); }
   Sout ( );
   }
  }
}
```